Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some specific sense defined by the analyst) to each other than to those in other groups (clusters). It is a main task of exploratory data analysis, and a common technique for statistical data analysis, used in many fields, including pattern recognition, image analysis, information retrieval, bioinformatics, data compression, computer graphics and machine learning.

Cluster analysis refers to a family of algorithms and tasks rather than one specific algorithm. It can be achieved by various algorithms that differ significantly in their understanding of what constitutes a cluster and how to efficiently find them. Popular notions of clusters include groups with small distances between cluster members, dense areas of the data space, intervals or particular statistical distributions. Clustering can therefore be formulated as a multi-objective optimization problem. The appropriate clustering algorithm and parameter settings (including parameters such as the distance function to use, a density threshold or the number of expected clusters) depend on the individual data set and intended use of the results. Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It is often necessary to modify data preprocessing and model parameters until the result achieves the desired properties.

Besides the term clustering, there is a number of terms with similar meanings, including automatic classification, numerical taxonomy, botryology (from Greek: βότρυς 'grape'), typological analysis, and community detection. The subtle differences are often in the use of the results: while in data mining, the resulting groups are the matter of interest, in automatic classification the resulting discriminative power is of interest.

Cluster analysis originated in anthropology by Driver and Kroeber in 1932

The notion of a "cluster" cannot be precisely defined, which is one of the reasons why there are so many clustering algorithms.

There is a common denominator: a group of data objects. However, different researchers employ different cluster models, and for each of these cluster models again different algorithms can be given. The notion of a cluster, as found by different algorithms, varies significantly in its properties. Understanding these "cluster models" is key to understanding the differences between the various algorithms. Typical cluster models include:

Connectivity models: for example, hierarchical clustering builds models based on distance connectivity. Centroid models: for example, the k-means algorithm represents each cluster by a single mean vector. Distribution models: clusters are modeled using statistical distributions, such as multivariate normal distributions used by the expectation-maximization algorithm. Density models: for example, DBSCAN and OPTICS

defines clusters as connected dense regions in the data space. Subspace models: in biclustering (also known as co-clustering or two-mode-clustering), clusters are modeled with both cluster members and relevant attributes. Group models: some algorithms do not provide a refined model for their results and just provide the grouping information. Graph-based models: a clique, that is, a subset of nodes in a graph such that every two nodes in the subset are connected by an edge can be considered as a prototypical form of cluster. Relaxations of the complete connectivity requirement (a fraction of the edges can be missing) are known as quasi-cliques, as in the HCS clustering algorithm. Signed graph models: Every path in a signed graph has a sign from the product of the signs on the edges. Under the assumptions of balance theory, edges may change sign and result in a bifurcated graph. The weaker "clusterability axiom" (no cycle has exactly one negative edge) yields results with more than two clusters, or subgraphs with only positive edges.

Neural models: the most well-known unsupervised neural network is the self-organizing map and these models can usually be characterized as similar to one or more of the above models, and including subspace models when neural networks implement a form of Principal Component Analysis or Independent Component Analysis.

A "clustering" is essentially a set of such clusters, usually containing all objects in the data set. Additionally, it may specify the relationship of the clusters to each other, for example, a hierarchy of clusters embedded in each other. Clusterings can be roughly distinguished as:

Hard clustering: each object belongs to a cluster or not Soft clustering (also: fuzzy clustering): each object belongs to each cluster to a certain degree (for example, a likelihood of belonging to the cluster)

There are also finer distinctions possible, for example:

Strict partitioning clustering: each object belongs to exactly one cluster

Strict partitioning clustering with outliers: objects can also belong to no cluster; in which case they are considered outliers

Overlapping clustering (also: alternative clustering, multi-view clustering): objects may belong to more than one cluster; usually involving hard clusters

Hierarchical clustering: objects that belong to a child cluster also belong to the parent cluster Subspace clustering: while an overlapping clustering, within a uniquely defined subspace, clusters are not expected to overlap.

https://en.wikipedia.org/wiki/Cluster_analysis

k-means clustering is a method of vector quantization, originally from signal processing, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells. k-means clustering minimizes within-cluster variances (squared Euclidean distances), but not regular Euclidean distances, which would be the more difficult Weber problem: the mean optimizes squared errors, whereas only the geometric median minimizes Euclidean distances. For instance, better Euclidean solutions can be found using k-medians and k-medoids.

The problem is computationally difficult (NP-hard); however, efficient heuristic algorithms converge quickly to a local optimum. These are usually similar to the expectation–maximization algorithm for mixtures of Gaussian distributions via an iterative refinement approach employed by both k-means and Gaussian mixture modeling. They both use cluster centers to model the data; however, k-means clustering tends to find clusters of comparable spatial extent, while the Gaussian mixture model allows clusters to have different shapes.

The unsupervised k-means algorithm has a loose relationship to the k-nearest neighbor classifier, a popular supervised machine learning technique for classification that is often confused with k-means due to the name. Applying the 1-nearest neighbor classifier to the cluster centers obtained by k-means classifies new data into the existing clusters. This is known as nearest centroid classifier or Rocchio algorithm.

https://en.wikipedia.org/wiki/K-means_clustering

The k-means problem is to find cluster centers that minimize the intra-class variance, i.e. the sum of squared distances from each data point being clustered to its cluster center (the center that is closest to it). Although finding an exact solution to the k-means problem for arbitrary input is NP-hard, the standard approach to finding an approximate solution (often called Lloyd's algorithm or the k-means algorithm) is used widely and frequently finds reasonable solutions quickly.

However, the k-means algorithm has at least two major theoretic shortcomings:

First, it has been shown that the worst case running time of the algorithm is super-polynomial in the input size.

Second, the approximation found can be arbitrarily bad with respect to the objective function compared to the optimal clustering. The k-means++ algorithm addresses the second of these obstacles by specifying a procedure to initialize the cluster centers before proceeding with the standard k-means optimization iterations. With the k-

means++ initialization, the algorithm is guaranteed to find a solution that is O(log k) competitive to the optimal k-means solution.

https://en.wikipedia.org/wiki/K-means%2B%2B

The k-medoids problem is a clustering problem similar to k-means. The name was coined by Leonard Kaufman and Peter J. Rousseeuw with their PAM (Partitioning Around Medoids) algorithm.[1] Both the k-means and k-medoids algorithms are partitional (breaking the dataset up into groups) and attempt to minimize the distance between points labeled to be in a cluster and a point designated as the center of that cluster. In contrast to the k-means algorithm, k-medoids chooses actual data points as centers (medoids or exemplars), and thereby allows for greater interpretability of the cluster centers than in k-means, where the center of a cluster is not necessarily one of the input data points (it is the average between the points in the cluster). Furthermore, k-medoids can be used with arbitrary dissimilarity measures, whereas k-means generally requires Euclidean distance for efficient solutions. Because k-medoids minimizes a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances, it is more robust to noise and outliers than k-means.

k-medoids is a classical partitioning technique of clustering that splits the data set of n objects into k clusters, where the number k of clusters assumed known a priori (which implies that the programmer must specify k before the execution of a k-medoids algorithm). The "goodness" of the given value of k can be assessed with methods such as the silhouette method.

The medoid of a cluster is defined as the object in the cluster whose sum (and, equivalently, the average) of dissimilarities to all the objects in the cluster is minimal, that is, it is a most centrally located point in the cluster.

https://en.wikipedia.org/wiki/K-medoids

In [59]:
```
# general test code i had

import matplotlib.pyplot as plt
import seaborn as sns
import math
import pandas as pd
import warnings
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.cluster import KMeans, DBSCAN
from sklearn.decomposition import PCA

warnings.filterwarnings("ignore")
```

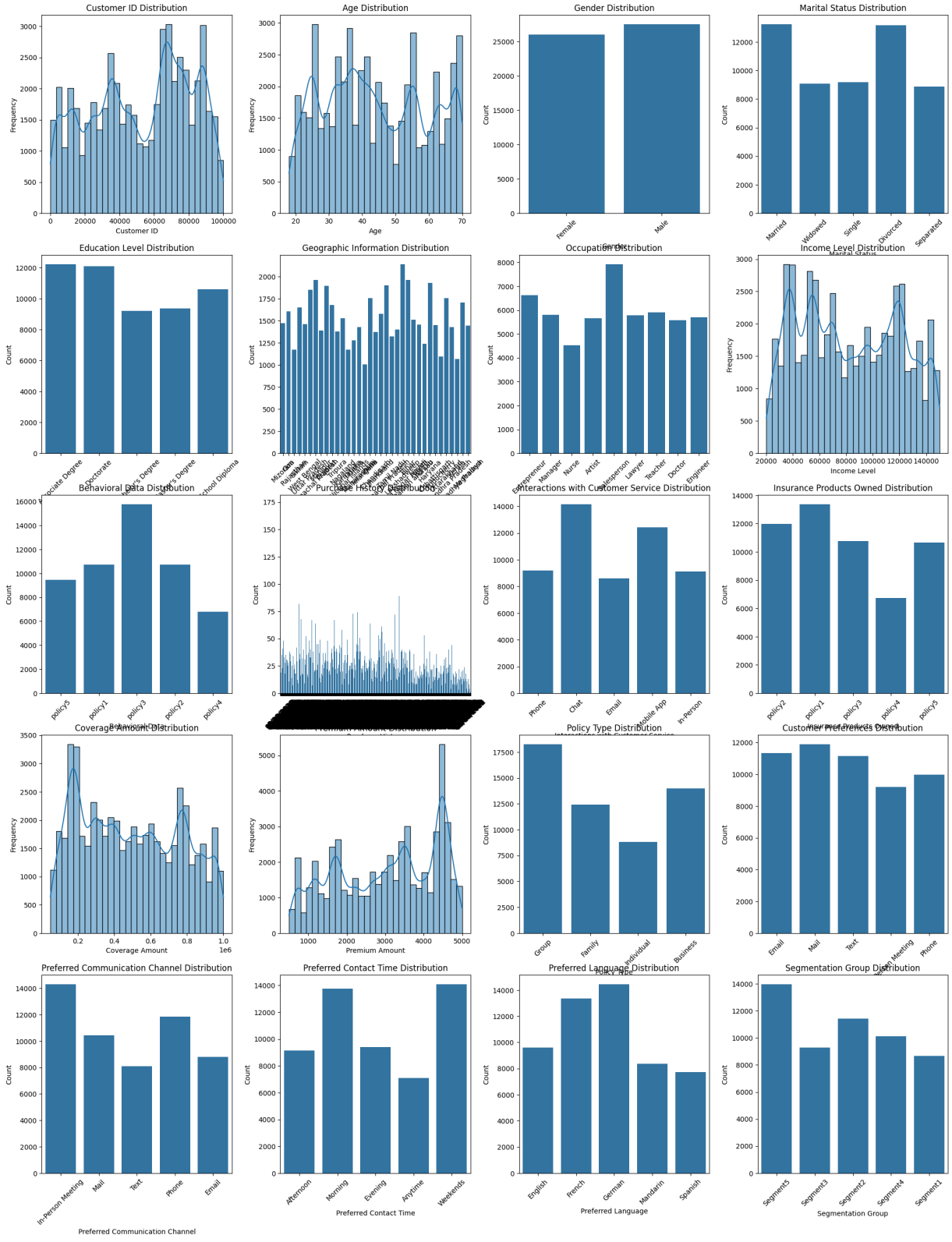
```
# Load the dataset
df = pd.read_csv('customer_segmentation_data.csv')

# Remove column limit
pd.set_option('display.max_columns', None)

# Display the first few rows of the dataframe
display(df.head())

#Print Line
print('-' * 80)

# Get a concise summary of the dataframe
display(df.info())

print('-' * 80)

# Check for missing values
display(df.isnull().sum())

# Define the list of continuous and categorical data types
continuous_data_types = ['int64', 'float64']
categorical_data_types = ['object', 'category', 'bool']

# Determine the total number of plots needed
total_plots = df.shape[1]

# Set the number of columns for the subplot grid
num_columns = 4

# Calculate the number of rows needed for the subplot grid
num_rows = math.ceil(total_plots / num_columns)

# Create the subplot grid
fig, axes = plt.subplots(num_rows, num_columns, figsize=(20, num_rows * 5))
fig.tight_layout(pad=4.0)

# Flatten the axes array for easy iteration
axes_flat = axes.flatten()

# Initialize a counter for the subplot index
subplot_index = 0
for column in df.columns:
    # Determine the data type of the column
    if df[column].dtype in continuous_data_types:
        # Plot for continuous variables
        sns.histplot(df[column].dropna(), bins=30, kde=True, ax=axes_flat[su
        axes_flat[subplot_index].set_title(f'{column} Distribution')
        axes_flat[subplot_index].set_xlabel(column)
        axes_flat[subplot_index].set_ylabel('Frequency')
    elif df[column].dtype in categorical_data_types:
        # Plot for categorical variables
        sns.countplot(x=column, data=df, ax=axes_flat[subplot_index])
        axes_flat[subplot_index].set_title(f'{column} Distribution')
        axes_flat[subplot_index].set_xlabel(column)
        axes_flat[subplot_index].set_ylabel('Count')
```

```
        axes_flat[subplot_index].tick_params(axis='x', rotation=45)  # Rotat
    subplot_index += 1
    # Hide any unused subplot axes

for i in range(subplot_index, len(axes_flat)):
    axes_flat[i].set_visible(False)
plt.show()
```

	Customer ID	Age	Gender	Marital Status	Education Level	Geographic Information	Occupation	Income Level	Be
0	84966	23	Female	Married	Associate Degree	Mizoram	Entrepreneur	70541	
1	95568	26	Male	Widowed	Doctorate	Goa	Manager	54168	
2	10544	29	Female	Single	Associate Degree	Rajasthan	Entrepreneur	73899	
3	77033	20	Male	Divorced	Bachelor's Degree	Sikkim	Entrepreneur	63381	
4	88160	25	Female	Separated	Bachelor's Degree	West Bengal	Manager	38794	

```
--------------------------------------------------------------------------------
----
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53503 entries, 0 to 53502
Data columns (total 20 columns):
 #   Column                                Non-Null Count  Dtype
---  ------                                --------------  -----
 0   Customer ID                           53503 non-null  int64
 1   Age                                   53503 non-null  int64
 2   Gender                                53503 non-null  object
 3   Marital Status                        53503 non-null  object
 4   Education Level                       53503 non-null  object
 5   Geographic Information                53503 non-null  object
 6   Occupation                            53503 non-null  object
 7   Income Level                          53503 non-null  int64
 8   Behavioral Data                       53503 non-null  object
 9   Purchase History                      53503 non-null  object
 10  Interactions with Customer Service    53503 non-null  object
 11  Insurance Products Owned              53503 non-null  object
 12  Coverage Amount                       53503 non-null  int64
 13  Premium Amount                        53503 non-null  int64
 14  Policy Type                           53503 non-null  object
 15  Customer Preferences                  53503 non-null  object
 16  Preferred Communication Channel       53503 non-null  object
 17  Preferred Contact Time                53503 non-null  object
 18  Preferred Language                    53503 non-null  object
 19  Segmentation Group                    53503 non-null  object
dtypes: int64(5), object(15)
memory usage: 8.2+ MB
None
```

```
--------------------------------------------------------------------------------
----
Customer ID                            0
Age                                    0
Gender                                 0
Marital Status                         0
Education Level                        0
Geographic Information                 0
Occupation                             0
Income Level                           0
Behavioral Data                        0
Purchase History                       0
Interactions with Customer Service     0
Insurance Products Owned               0
Coverage Amount                        0
Premium Amount                         0
Policy Type                            0
Customer Preferences                   0
Preferred Communication Channel        0
Preferred Contact Time                 0
Preferred Language                     0
Segmentation Group                     0
dtype: int64
```



In [17]:

```
print("Number of Columns in the Data: ", df.shape[1])
print("Number of Rows in the Data: ", df.shape[0])
df.info()
```

```
Number of Columns in the Data:  20
Number of Rows in the Data:  53503
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 53503 entries, 0 to 53502
Data columns (total 20 columns):
 #   Column                                Non-Null Count  Dtype
---  ------                                --------------  -----
 0   Customer ID                           53503 non-null  int64
 1   Age                                   53503 non-null  int64
 2   Gender                                53503 non-null  object
 3   Marital Status                        53503 non-null  object
 4   Education Level                       53503 non-null  object
 5   Geographic Information                53503 non-null  object
 6   Occupation                            53503 non-null  object
 7   Income Level                          53503 non-null  int64
 8   Behavioral Data                       53503 non-null  object
 9   Purchase History                      53503 non-null  object
 10  Interactions with Customer Service  53503 non-null  object
 11  Insurance Products Owned              53503 non-null  object
 12  Coverage Amount                       53503 non-null  int64
 13  Premium Amount                        53503 non-null  int64
 14  Policy Type                           53503 non-null  object
 15  Customer Preferences                  53503 non-null  object
 16  Preferred Communication Channel     53503 non-null  object
 17  Preferred Contact Time                53503 non-null  object
 18  Preferred Language                    53503 non-null  object
 19  Segmentation Group                    53503 non-null  object
dtypes: int64(5), object(15)
memory usage: 8.2+ MB
```

In [18]:
```
df.isnull().sum()
```

Out[18]:
```
Customer ID                           0
Age                                   0
Gender                                0
Marital Status                        0
Education Level                       0
Geographic Information                0
Occupation                            0
Income Level                          0
Behavioral Data                       0
Purchase History                      0
Interactions with Customer Service    0
Insurance Products Owned              0
Coverage Amount                       0
Premium Amount                        0
Policy Type                           0
Customer Preferences                  0
Preferred Communication Channel       0
Preferred Contact Time                0
Preferred Language                    0
Segmentation Group                    0
dtype: int64
```

In [19]:
```
df.duplicated().sum()
```

Out[19]: 0

In [20]:
```
des = df.describe().transpose()
palette = sns.color_palette("icefire", as_cmap=True)
des.style.background_gradient(cmap=palette)
```

Out[20]:

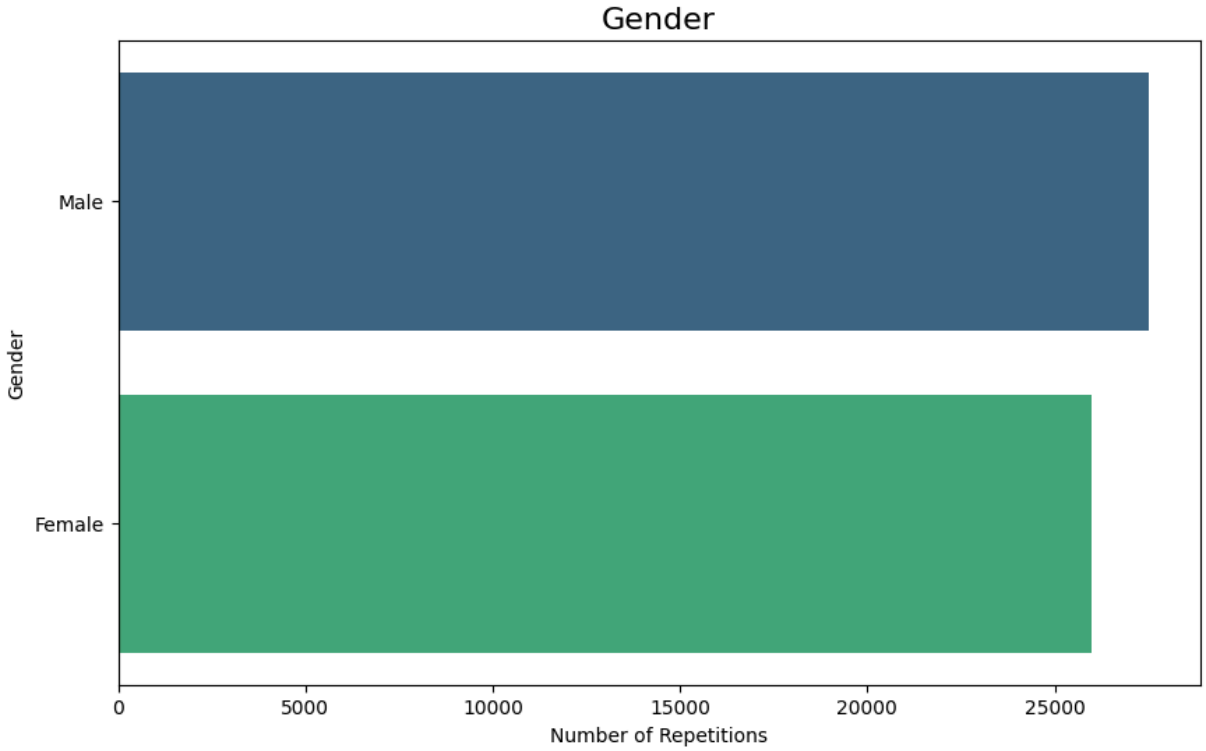
	count	mean	std	min	25
Customer ID	53503.000000	52265.204998	28165.000067	1.000000	28950.50000
Age	53503.000000	44.140945	15.079486	18.000000	32.00000
Income Level	53503.000000	82768.324318	36651.075670	20001.000000	51568.50000
Coverage Amount	53503.000000	492580.789638	268405.505571	50001.000000	249613.50000
Premium Amount	53503.000000	3023.702447	1285.834295	500.000000	1817.00000

In [53]:
```
des2 = df.head()
palette = sns.color_palette("icefire", as_cmap=True)
des2.style.background_gradient(cmap=palette)
```

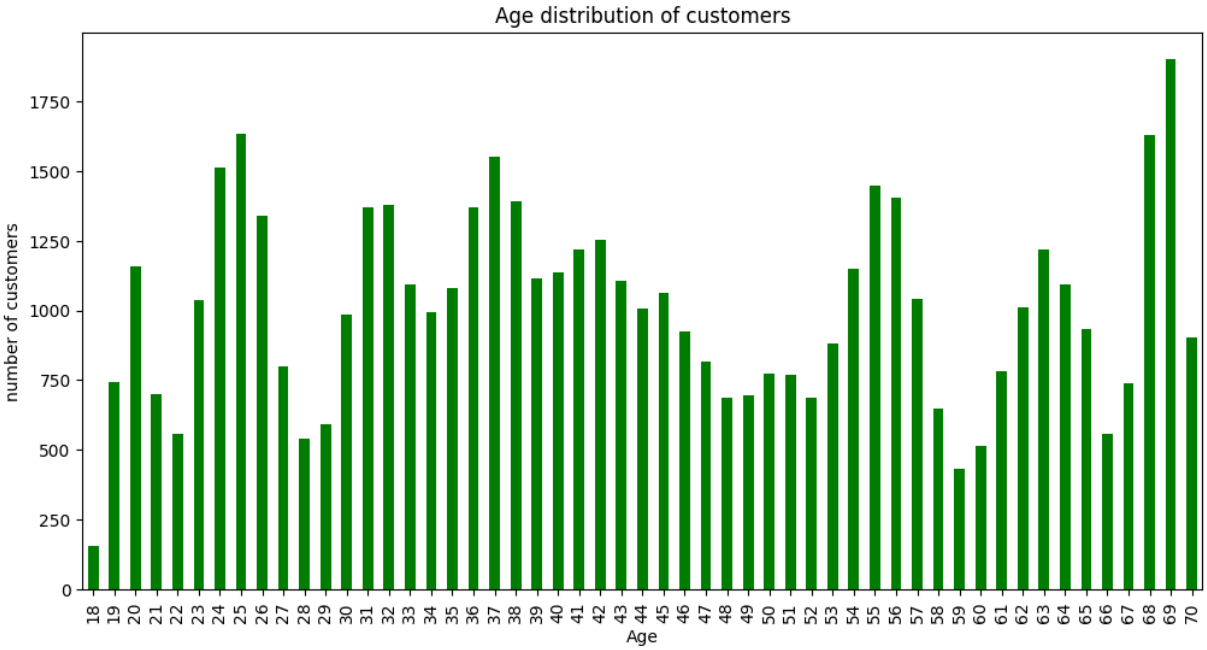
Out[53]:

	Customer ID	Age	Gender	Marital Status	Education Level	Geographic Information	Occupation	Income Level
0	84966	23	Female	Married	Associate Degree	Mizoram	Entrepreneur	70541
1	95568	26	Male	Widowed	Doctorate	Goa	Manager	54168
2	10544	29	Female	Single	Associate Degree	Rajasthan	Entrepreneur	73899
3	77033	20	Male	Divorced	Bachelor's Degree	Sikkim	Entrepreneur	63381
4	88160	25	Female	Separated	Bachelor's Degree	West Bengal	Manager	38794

In [21]:
```
gender_count = df.Gender.value_counts().head(10)

plt.figure(figsize=(10, 6))
sns.barplot(x=gender_count.values, y=gender_count.index, palette='viridis')
plt.title("Gender", fontsize=16)
plt.xlabel("Number of Repetitions")
plt.ylabel("Gender")
plt.show()
```

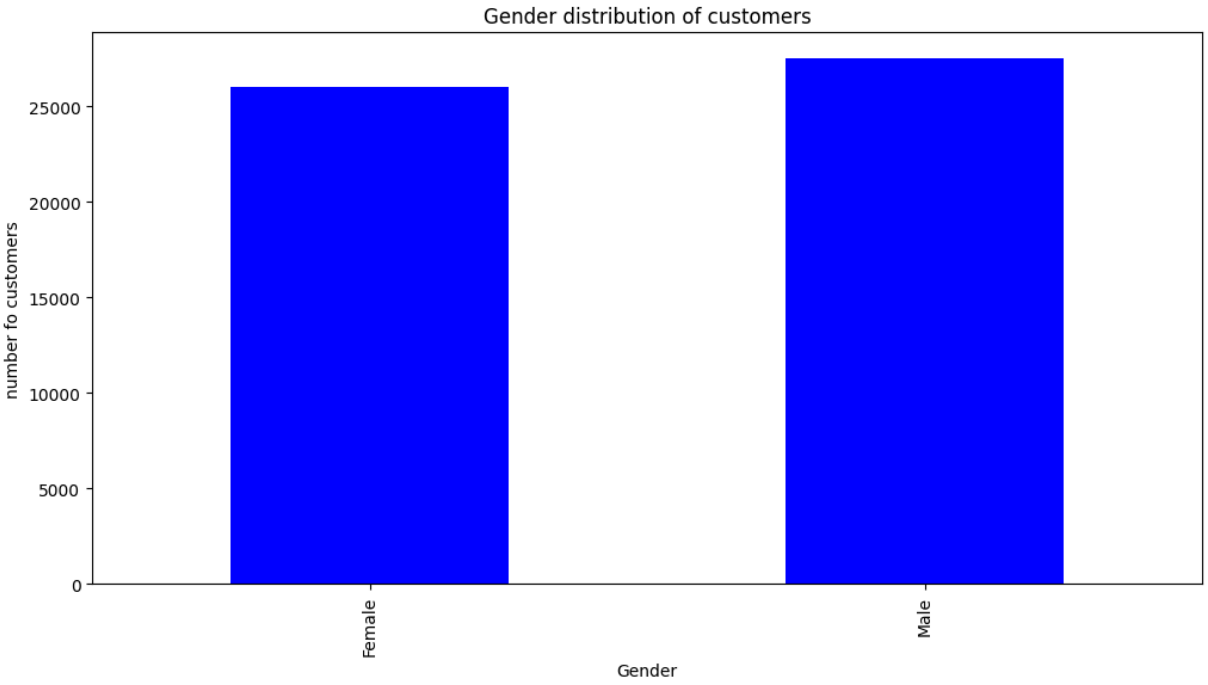


In [22]:
```
# plot Age Distribution:
age_distribution= df['Age'].value_counts().sort_index()
plt.figure(figsize=(12,6))
age_distribution.plot(kind='bar', color='green')
plt.title('Age distribution of customers')
plt.xlabel('Age')
plt.ylabel('number of customers')
plt.show()
# print age distribution
print('Age distribution of customers:')
for age, count in age_distribution.items():
    print(f'Age: {age}, number of customers: {count}')
```



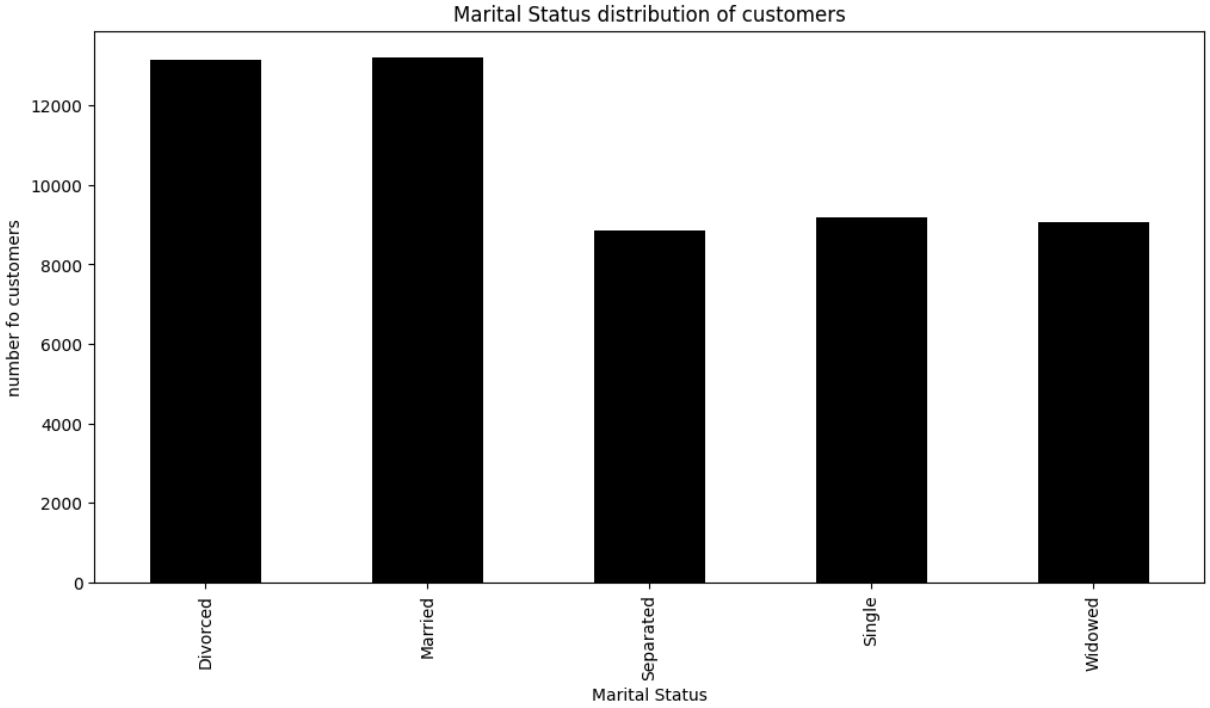
Age distribution of customers:
Age: 18, number of customers: 154
Age: 19, number of customers: 744
Age: 20, number of customers: 1156
Age: 21, number of customers: 700
Age: 22, number of customers: 556
Age: 23, number of customers: 1038
Age: 24, number of customers: 1511
Age: 25, number of customers: 1635
Age: 26, number of customers: 1341
Age: 27, number of customers: 801
Age: 28, number of customers: 542
Age: 29, number of customers: 593
Age: 30, number of customers: 983
Age: 31, number of customers: 1371
Age: 32, number of customers: 1379
Age: 33, number of customers: 1092
Age: 34, number of customers: 993
Age: 35, number of customers: 1081
Age: 36, number of customers: 1368
Age: 37, number of customers: 1550
Age: 38, number of customers: 1391
Age: 39, number of customers: 1113
Age: 40, number of customers: 1137
Age: 41, number of customers: 1217
Age: 42, number of customers: 1254
Age: 43, number of customers: 1107
Age: 44, number of customers: 1008
Age: 45, number of customers: 1061
Age: 46, number of customers: 926
Age: 47, number of customers: 816
Age: 48, number of customers: 686
Age: 49, number of customers: 695
Age: 50, number of customers: 772
Age: 51, number of customers: 769
Age: 52, number of customers: 686
Age: 53, number of customers: 880
Age: 54, number of customers: 1148
Age: 55, number of customers: 1448
Age: 56, number of customers: 1402
Age: 57, number of customers: 1040
Age: 58, number of customers: 646
Age: 59, number of customers: 434
Age: 60, number of customers: 513
Age: 61, number of customers: 782
Age: 62, number of customers: 1011
Age: 63, number of customers: 1218
Age: 64, number of customers: 1093
Age: 65, number of customers: 933
Age: 66, number of customers: 558
Age: 67, number of customers: 740
Age: 68, number of customers: 1628
Age: 69, number of customers: 1901
Age: 70, number of customers: 902

In [23]:
```
# plot the gender Distribution
gender_distribution=df['Gender'].value_counts().sort_index()
plt.figure(figsize=(12,6))
gender_distribution.plot(kind='bar', color='blue')
plt.title('Gender distribution of customers')
plt.xlabel('Gender')
plt.ylabel('number fo customers')
plt.show()
# print gender distribution of customers
print('Gender Distribution of Customers')
for gender , count in gender_distribution.items():
    print(f'Gender: {gender}, numbers of customers : {count}')
```



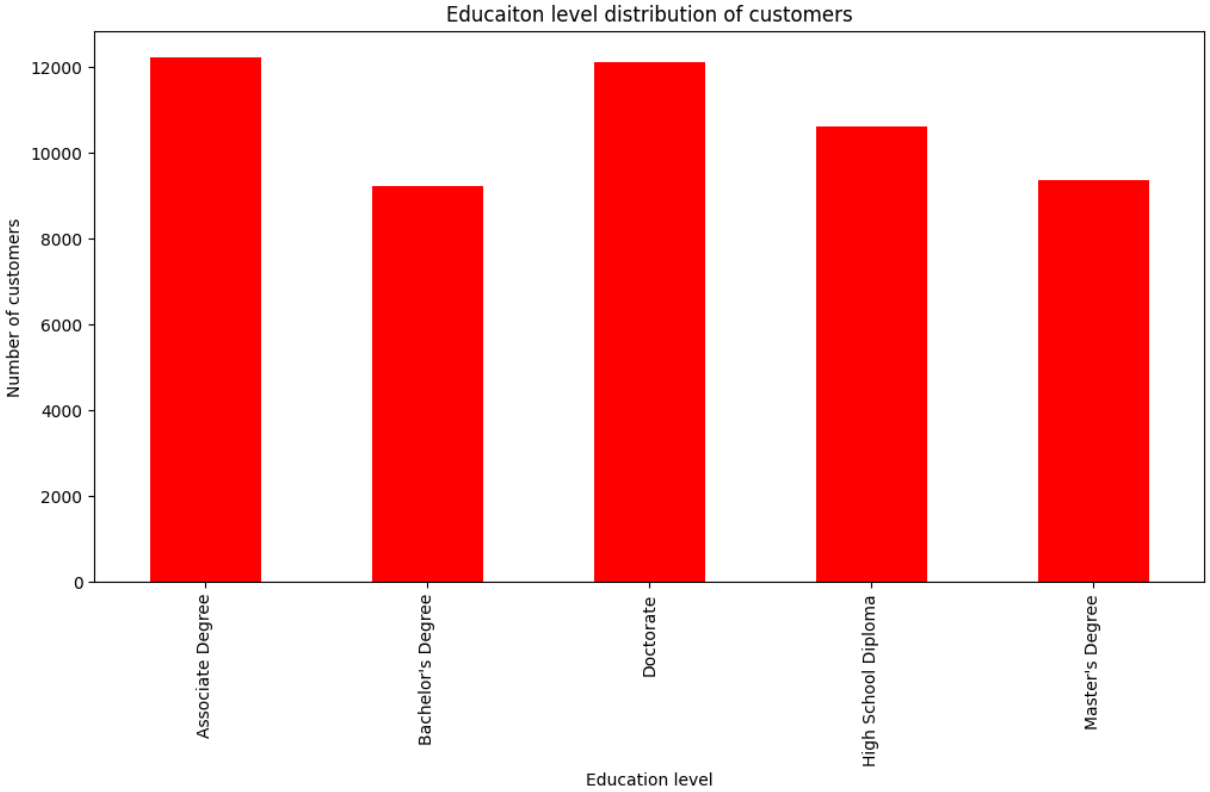

```
Gender Distribution of Customers
Gender: Female, numbers of customers : 26004
Gender: Male, numbers of customers : 27499
```

In [24]:
```
# plot the marital status Distribution
marital_status_distribution=df['Marital Status'].value_counts().sort_index()
plt.figure(figsize=(12,6))
marital_status_distribution.plot(kind='bar', color='black')
plt.title('Marital Status distribution of customers')
plt.xlabel('Marital Status')
plt.ylabel('number fo customers')
plt.show()
# print Marital Status distribution of customers
print('Marital Status Distribution of Customers')
for Marital_Status , count in marital_status_distribution.items():
    print(f'Marital Status: {Marital_Status}, numbers of customers : {count}
```



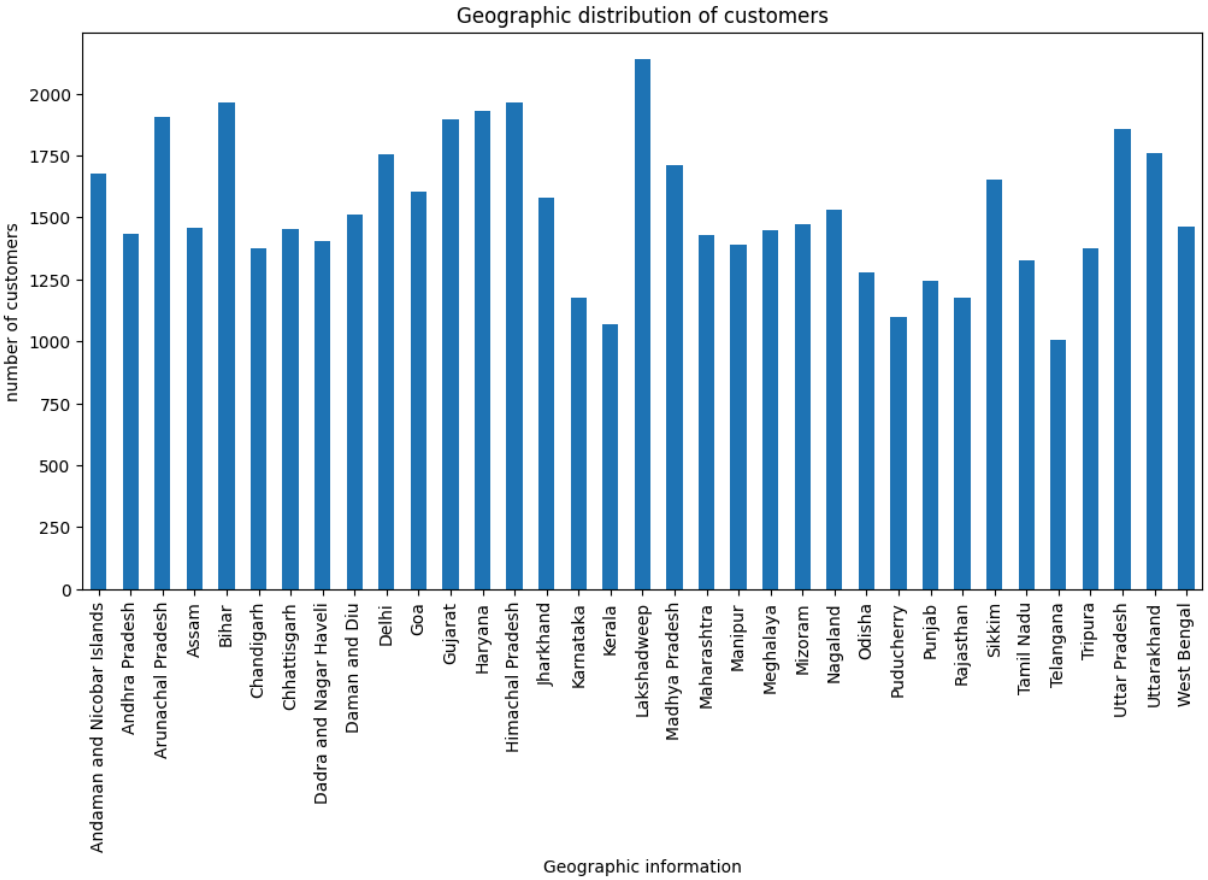

```
Marital Status Distribution of Customers
Marital Status: Divorced, numbers of customers : 13151
Marital Status: Married, numbers of customers : 13219
Marital Status: Separated, numbers of customers : 8861
Marital Status: Single, numbers of customers : 9195
Marital Status: Widowed, numbers of customers : 9077
```

In [25]:
```python
# plot the EDucation level Distibution of customers
education_distribution=df['Education Level'].value_counts().sort_index()
plt.figure(figsize=(12,6))
education_distribution.plot(kind='bar', color='red')
plt.title('Educaiton level distribution of customers')
plt.xlabel('Education level')
plt.ylabel('Number of customers')
plt.show()
# pritn the Education level distribution of customers
print('Education level distribution of customers:')
for education, count in education_distribution.items():
    print(f'Education level : {education}, number of customer: {count}')
```




```
Education level distribution of customers:
Education level : Associate Degree, number of customer: 12213
Education level : Bachelor's Degree, number of customer: 9214
Education level : Doctorate, number of customer: 12103
Education level : High School Diploma, number of customer: 10607
Education level : Master's Degree, number of customer: 9366
```

In [26]:
```python
# plot the geographic distribution of customers
geo_distribution=df['Geographic Information'].value_counts().sort_index()
plt.figure(figsize=(12,6))
geo_distribution.plot(kind='bar')
plt.title('Geographic distribution of customers')
plt.xlabel('Geographic information')
plt.ylabel('number of customers')
plt.show()
# print the geographic distribution of customers
print('Geographic distribtuion of customers:')
for geo, count in geo_distribution.items():
    print(f'Geographic information: {geo}, number of customers: {count}')
```


Geographic distribution of customers
number of customers
0
250
500
750
1000
1250
1500
1750
2000
Andaman and Nicobar Islands
Andhra Pradesh
Arunachal Pradesh
Assam
Bihar
Chandigarh
Chhattisgarh
Dadra and Nagar Haveli
Daman and Diu
Delhi
Goa
Gujarat
Haryana
Himachal Pradesh
Jharkhand
Karnataka
Kerala
Lakshadweep
Madhya Pradesh
Maharashtra
Manipur
Meghalaya
Mizoram
Nagaland
Odisha
Puducherry
Punjab
Rajasthan
Sikkim
Tamil Nadu
Telangana
Tripura
Uttar Pradesh
Uttarakhand
West Bengal
Geographic information

```
Geographic distribtuion of customers:
Geographic information: Andaman and Nicobar Islands, number of customers: 16
78
Geographic information: Andhra Pradesh, number of customers: 1431
Geographic information: Arunachal Pradesh, number of customers: 1903
Geographic information: Assam, number of customers: 1460
Geographic information: Bihar, number of customers: 1962
Geographic information: Chandigarh, number of customers: 1376
Geographic information: Chhattisgarh, number of customers: 1451
Geographic information: Dadra and Nagar Haveli, number of customers: 1403
Geographic information: Daman and Diu, number of customers: 1512
Geographic information: Delhi, number of customers: 1756
Geographic information: Goa, number of customers: 1605
Geographic information: Gujarat, number of customers: 1895
Geographic information: Haryana, number of customers: 1931
Geographic information: Himachal Pradesh, number of customers: 1963
Geographic information: Jharkhand, number of customers: 1578
Geographic information: Karnataka, number of customers: 1175
Geographic information: Kerala, number of customers: 1071
Geographic information: Lakshadweep, number of customers: 2140
Geographic information: Madhya Pradesh, number of customers: 1710
Geographic information: Maharashtra, number of customers: 1428
Geographic information: Manipur, number of customers: 1391
Geographic information: Meghalaya, number of customers: 1447
Geographic information: Mizoram, number of customers: 1472
Geographic information: Nagaland, number of customers: 1529
Geographic information: Odisha, number of customers: 1280
Geographic information: Puducherry, number of customers: 1098
Geographic information: Punjab, number of customers: 1243
Geographic information: Rajasthan, number of customers: 1176
Geographic information: Sikkim, number of customers: 1654
Geographic information: Tamil Nadu, number of customers: 1324
Geographic information: Telangana, number of customers: 1007
Geographic information: Tripura, number of customers: 1377
Geographic information: Uttar Pradesh, number of customers: 1855
Geographic information: Uttarakhand, number of customers: 1758
Geographic information: West Bengal, number of customers: 1464
```

REVIEW Clustering

Which of the tasks we CAN'T solve using clustering?

Select a few correct answers

Detect outliers in our dataset

Predict Tesla stock price for next week

Identify heart disease in a person according to his medical record

Create recommendation system for internet shop

Identify people with similar tastes on their shopping list

To determine which tasks cannot be solved using clustering, we need to understand that clustering is an unsupervised learning technique that groups data points based on their similarity. Clustering does not involve predictions or direct associations with labels, making it unsuitable for tasks requiring these capabilities.

Tasks that cannot be solved using clustering: Predict Tesla stock price for next week Reason: Predicting stock prices is a supervised learning task that involves forecasting based on historical data. Clustering is not designed for time series prediction or regression. Identify heart disease in a person according to their medical record Reason: This is a classification task, where a label (e.g., "has heart disease" or "does not have heart disease") is predicted based on input features. Clustering does not assign specific labels or diagnoses. Tasks that clustering can solve: Detect outliers in our dataset Reason: Clustering can be used to find outliers by identifying data points that do not belong to any cluster or are located far from cluster centroids. Create a recommendation system for an internet shop Reason: Clustering can group users or products based on similarities, which can serve as a foundation for a recommendation system. Identify people with similar tastes on their shopping list Reason: Clustering can group people with similar shopping behaviors or preferences based on features extracted from their shopping data. Final Answer: Tasks clustering cannot solve:

Predict Tesla stock price for next week Identify heart disease in a person according to their medical record

Clustering is a type of machine learning in which the model is trained on unlabelled data without any predefined target variable or correct output (it is called unsupervised learning). The goal is to identify hidden patterns or structures in the data without any prior knowledge of the output.

By this, the approach to learning also changes: in supervised learning, we have to minimize the difference between the predicted value and the actual value (label), while in unsupervised, we must determine which function we will minimize to solve a specific problem (it can be cross entropy when working with images, different kinds of mathematical norms for working with numerical data, density when using statistical

methods, etc.). Simply, we need to choose by what criteria we will consider objects close to each other for clustering. In most of the algorithms, usual euclidean distance is used for this.

Also, there are often used intra-cluster (the distance between a data item and the cluster centroid within a cluster) and inter-cluster (the distance between the data items in distinct clusters) distances: the smaller is intra-cluster and the greater the inter-cluster distance, the better our algorithm coped with the clustering task.

Now, let`s discuss the advantages and disadvantages of clustering. Pros: clustering helps to solve machine learning problems without requiring us to label data, which can be time-consuming; clustering algorithms can help us to enhance data quality by detecting outliers, reducing data dimensions, and engineering features; clustering can help us identify valuable patterns and insights in our data; clustering algorithms can work with data that doesn't follow a consistent pattern over time. Cons: clustering can be expensive because it may require human experts to interpret the patterns and connect them to domain knowledge; there's no guarantee that clustering will provide useful results since we don't have labeled data to validate the outcomes; the accuracy of clustering results can vary depending on the method used. Can we use clustering for data preprocessing to improve accuracy of supervised learning algorithms? Can we use clustering for data preprocessing to improve accuracy of supervised learning algorithms? Select the correct answer

Yes

No

Correct Answer: Yes

Explanation: Clustering can be used as a data preprocessing technique to improve the accuracy of supervised learning algorithms in several ways:

Feature Engineering: Clustering can identify groups within the data, and these cluster labels can be added as new features to provide additional context for the supervised learning algorithm. Outlier Detection: Clustering can help identify and remove outliers, which may improve the performance of supervised models by reducing noise in the data. Dimensionality Reduction: By grouping similar data points, clustering can help simplify complex datasets, making it easier for supervised learning models to learn patterns. Class Imbalance Handling: Clustering can help rebalance datasets by grouping minority class samples together, potentially aiding in strategies to address imbalanced datasets. Thus, clustering is a valuable preprocessing tool that can enhance supervised learning tasks.

Several types of clustering algorithms can be used depending on the nature of the data and the desired clustering outcome. Here are some common types of clustering:

1. Partitional clustering: This clustering method splits the information into multiple groups based on the characteristics and similarities of the data. Given a data set of N points, a partitioning method constructs K (N ≥ K) partitions of the data, with each partition representing a cluster. K must be chosen manually according to the specialty of data and domain area. The most popular partitional clustering algorithm is K-means;

2. Hierarchical Clustering: In this type of clustering, the goal is to create a tree-like structure of nested clusters, where each cluster can contain individual data points or other clusters. Using this tree-like structure, we can understand in what sequence exactly which points are merged. Hierarchical clustering can be further divided into two subtypes: agglomerative clustering and divisive clustering;

3. Density-based clustering: this type of clustering identifies clusters based on the density of data points in the feature space. The goal of density-based clustering is to find areas of high density separated by areas of low density. The most popular density-based clustering algorithms are DBSCAN and Mean-shift.

There are also other types of clustering but they will not be covered here.

Imagine that you have to cluster people based on their family ties and create a family tree. Which type of clustering should you use to deal with this task?

Select the correct answer

Partitional clustering

Hierarchical clustering

Density-based clustering

Correct Answer: Hierarchical clustering

Explanation: Hierarchical clustering is the most suitable approach for tasks like creating a family tree, as it organizes data points into a tree-like structure. This method is ideal for tasks involving relationships or hierarchies, such as family ties, where individuals are connected based on shared ancestry or relationships.

Key reasons for choosing hierarchical clustering:

Tree-Like Structure: It naturally forms a dendrogram (tree structure), which is similar to a family tree. Captures Relationships: Hierarchical clustering captures nested relationships, making it ideal for understanding parent-child and sibling-like connections in family data. Other options: Partitional clustering: Groups data into distinct clusters without a hierarchical structure, making it unsuitable for building a family tree. Density-based clustering: Identifies dense regions of data and is better suited for tasks like finding clusters in spatial data or outlier detection, not hierarchical relationships.

K-means clustering is the most popular clustering algorithm used to group similar data points together in a dataset. The algorithm works by first selecting a value k, which represents the number of clusters or groups that we want to identify in the data. Let's briefly describe all the stages of the operation of this algorithm:

Step 1. The algorithm initializes k random points in the dataset, called centroids;

Step 2. Each data point is then assigned to the nearest centroid based on a distance metric, such as Euclidean distance. This process creates k clusters, with each cluster consisting of the data points that are closest to the centroid;

Step 3. The centroids are moved to the center of each cluster;

Step 4. Steps 2 and 3 are repeated. The algorithm iteratively updates the centroids and reassigns data points until convergence, when the centroids no longer move.

We can see that this algorithm is quite simple and intuitive, but it has some severe shortcomings: we need to choose the number of clusters manually; algorithm depends on initial centroid values; the algorithm is highly affected by outliers.

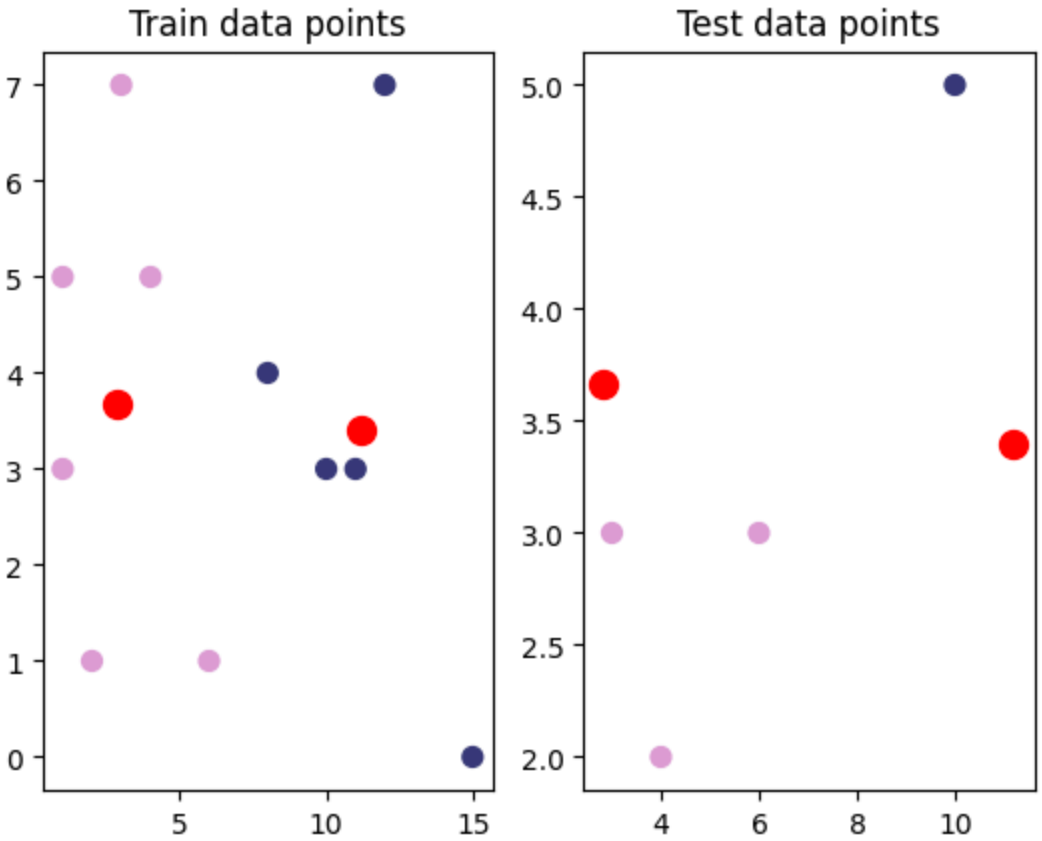
Let's look at K-means implementation in Python:

In [27]:
```
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
import warnings

warnings.filterwarnings('ignore')

# Create toy dataset to show K-means clustering model
X = np.array([[1, 3], [2, 1], [1, 5], [8, 4], [11, 3], [15, 0], [6,1], [10,3
# Fit K-means model for 2 clusters
kmeans = KMeans(n_clusters=2).fit(X)
# Print labels for train data
print('Train labels are: ', kmeans.labels_)
```

```
# Print coordinates of cluster centers
print('Cluster centers are: ', kmeans.cluster_centers_)
# Visualize the results of clustering
fig, axes = plt.subplots(1, 2)
axes[0].scatter(X[:, 0], X[:, 1], c=kmeans.labels_, s=50, cmap='tab20b')
axes[0].scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1]
axes[0].set_title('Train data points')

# Provide predictions for new data
predicted_labels = kmeans.predict([[10, 5], [4, 2], [3, 3], [6, 3]])
print('Predicted labels are: ', predicted_labels)
# Visualize prediction results
axes[1].scatter([10, 4, 3, 6], [5, 2, 3, 3], c=predicted_labels, s=50, cmap=
axes[1].scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1]
axes[1].set_title('Test data points')
```

```
Train labels are:  [1 1 1 0 0 0 1 0 1 1 0]
Cluster centers are:  [[11.2         3.4       ]
 [ 2.83333333  3.66666667]]
Predicted labels are:  [0 1 1 1]
```

Out[27]: Text(0.5, 1.0, 'Test data points')



In the code above, we used the following:

Kmeans class from sklearn. cluster. n_clusters parameter determines the number of clusters in the data; .fit(X) method of Kmeans class fits our model - determines clusters and their centers according to data X; .labels_ attribute of KMeans class stores cluster numbers for each sample of train data(0 cluster, 1 cluster, 2 cluster,...);

.cluster_centers_attribute of KMeans class stores cluster centers coordinates fitted by the algorithm; .predict() method of Kmeans class is used to predict labels of new points.##

Should we use K-means algorithm for clustering tasks if we can't manually determine the number of clusters into which our data should be divided?

Select the correct answer

Yes, we should

No, because we need to choose amount of clusters before using this algorithm

Correct Answer: No, because we need to choose the number of clusters before using this algorithm

Explanation: The K-means algorithm requires the number of clusters (K K) to be specified before execution. If the number of clusters is not known in advance, K-means might not be the best choice unless additional methods are used to determineK, such as:

Elbow method: Evaluates the sum of squared distances within clusters to find the optimal K. Silhouette score: Measures how similar a data point is to its cluster compared to other clusters. Gap statistic: Compares clustering results to random uniform data. Without knowing the number of clusters or applying a method to estimate it, K-means cannot function effectively, making it unsuitable for tasks where K is unknown.

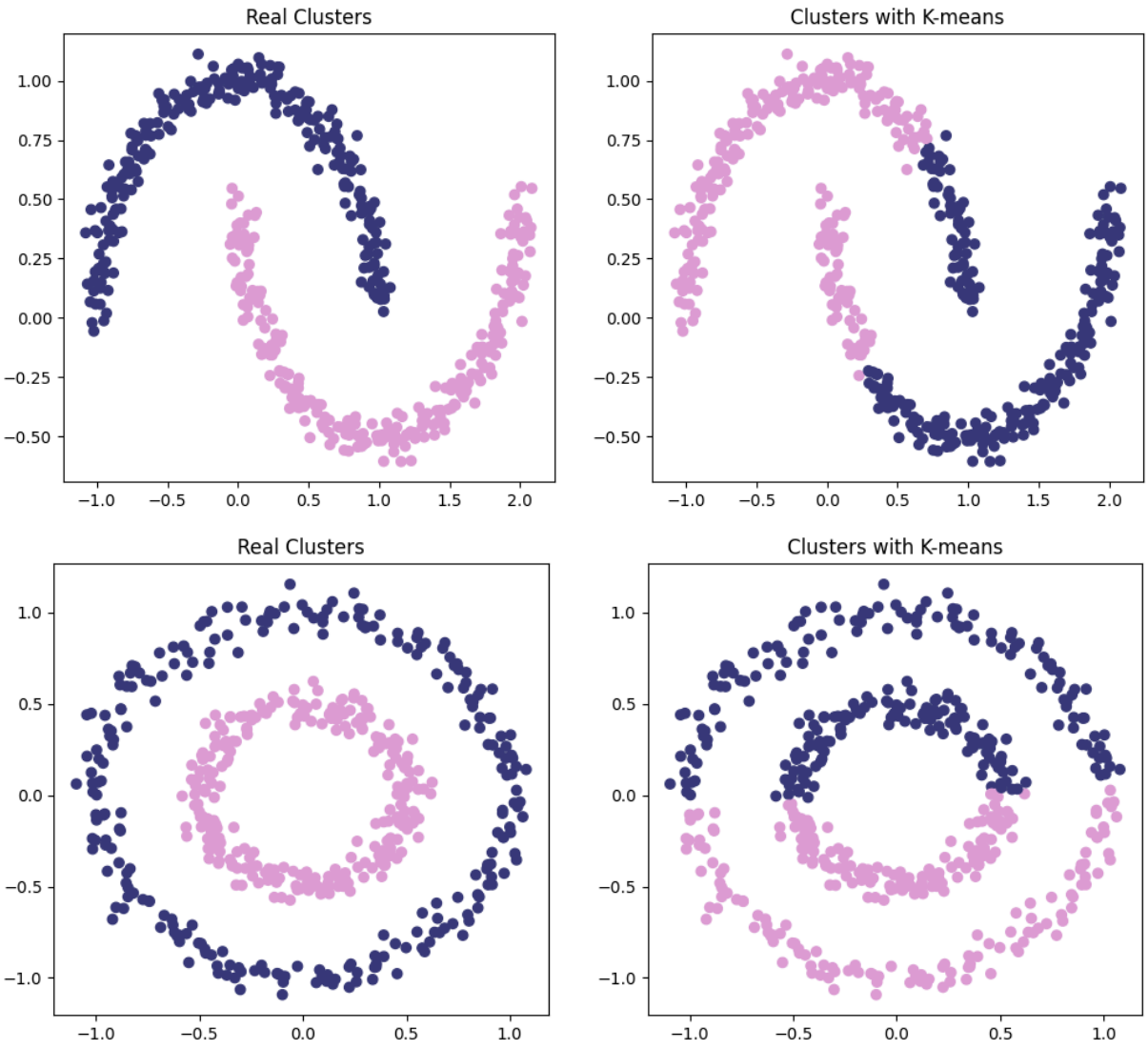
Task

Let's check the efficiency of the algorithm on different types of clusters. Now we will use the three built-in datasets of the sklearn library and try to use the K-means algorithm to cluster the corresponding points. We will provide visualizations and try to estimate the quality of clustering using these visualizations. Your task is to use the K-means clustering algorithm and to solve 3 different clustering problems. Compare the results and make conclusions about clustering quality. You have to: Use KMeans class from cluster module for import. Use KMeans class to instantiate a class object Use.fit()method to train model. Use .labels_attribute to extract fitted clusters.

Note In visualizations, it is necessary to look not at the color of clusters, but at the relative position of points in real and predicted clusters (Python can color the same clusters with different colors in different pictures due to implementation features)

In [28]:

```
from sklearn.datasets import make_blobs, make_moons, make_circles
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np

# Function to fit KMeans on different datasets and plot results
def check_clustering_quality(X, y, n_clusters):
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Plot real clusters
    axes[0].scatter(X[:, 0], X[:, 1], c=y, cmap='tab20b')
    axes[0].set_title('Real Clusters')

    # Fit KMeans and plot KMeans clusters
    kmeans = KMeans(n_clusters=n_clusters, random_state=42).fit(X)
    axes[1].scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='tab20b')
    axes[1].set_title('Clusters with K-means')
    plt.show()

# Now let's use check_clustering_quality function on blobs, moons, and circl

# Dataset 1: Blobs
X, y = make_blobs(n_samples=500, cluster_std=1, centers=3, random_state=42)
check_clustering_quality(X, y, 3)

# Dataset 2: Moons
X, y = make_moons(n_samples=500, noise=0.05, random_state=42)
check_clustering_quality(X, y, 2)

# Dataset 3: Circles
X, y = make_circles(n_samples=500, noise=0.05, factor=0.5, random_state=42)
check_clustering_quality(X, y, 2)
```



Explanation of Changes:

Imports: from sklearn.cluster import KMeans: Correct import for the KMeans class. Instantiate and Train the Model: kmeans = KMeans(n_clusters=n_clusters, random_state=42).fit(X): Creates and fits the KMeans model. Access Labels: kmeans.labels_: Extracts cluster labels. Visualizations: Side-by-side plots compare true labels with KMeans-generated clusters. Random State: Added random_state=42 for reproducibility. Output: For each dataset, you'll see two subplots:

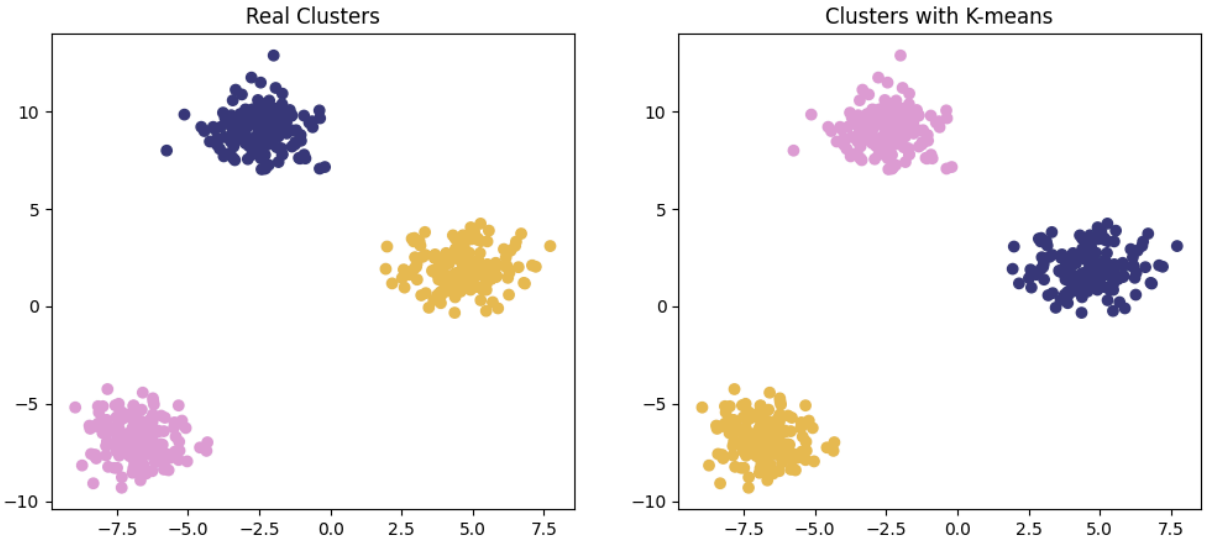
Left Plot: True labels (real clusters). Right Plot: Predicted clusters by KMeans. Conclusions (Expected Observations): Blobs: KMeans should work well because blobs have well-separated clusters. Moons: KMeans may struggle as the clusters are not circular and are intertwined. Circles: KMeans will fail to separate concentric circles because it assumes spherical clusters. These results will illustrate how the geometry and distribution of data affect the performance of KMeans.##

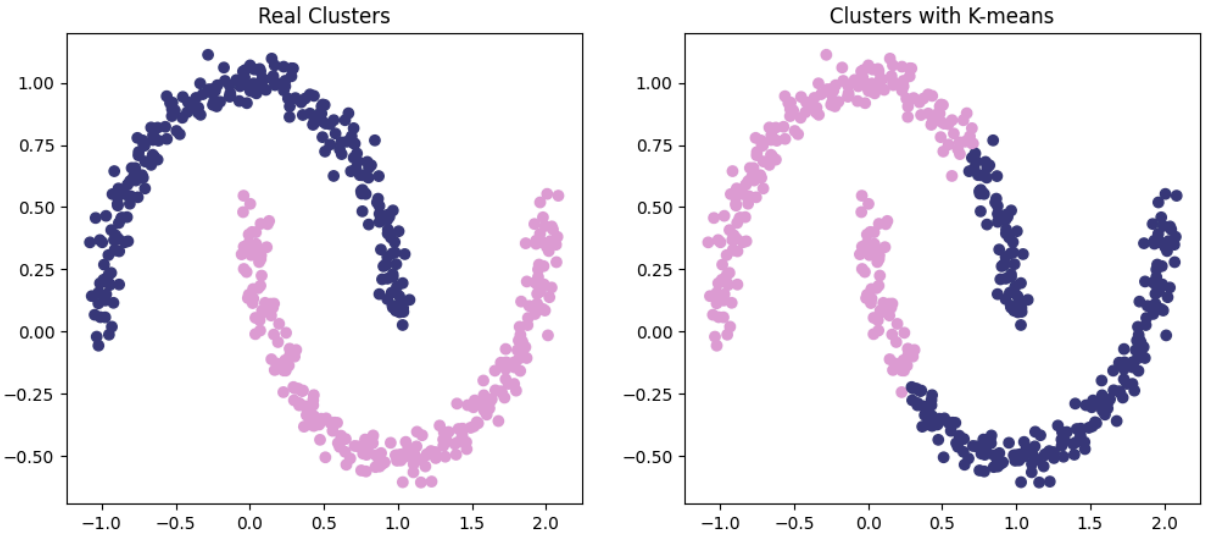
Fit Method

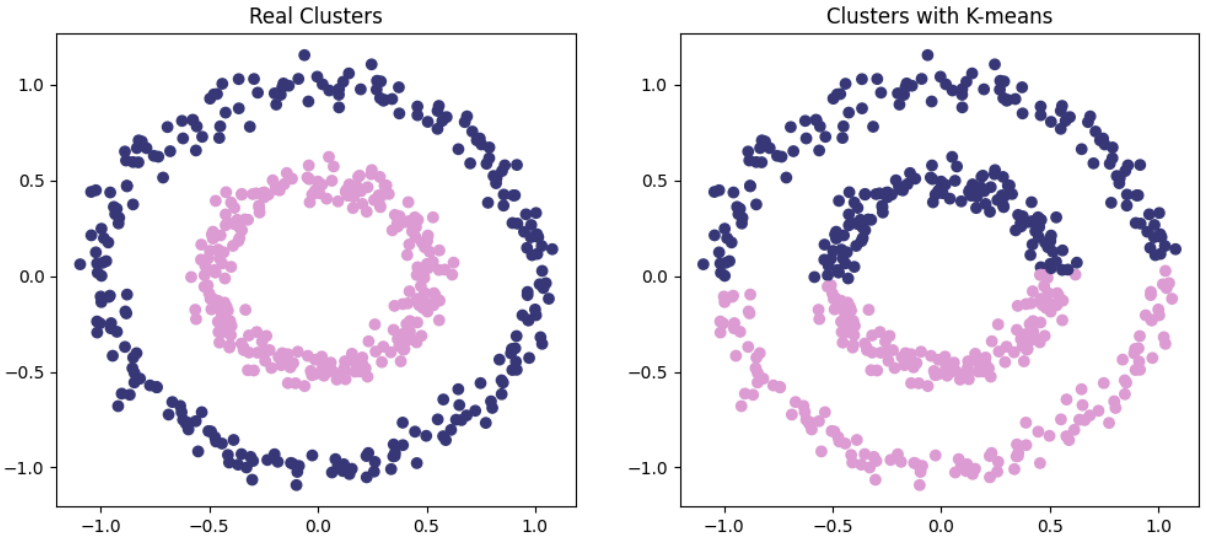
Explanation: Explicit Use of .fit(): kmeans.fit(X) is explicitly called to train the KMeans model on the dataset X. No Change in Output: The .fit() method trains the model, and the cluster labels are accessed using kmeans.labels_ as before. Plots: The function generates side-by-side visualizations for true clusters vs. KMeans-predicted clusters. Expected Results: Blobs: KMeans will perform well. Moons: KMeans will struggle due to intertwined clusters. Circles: KMeans will fail to separate concentric circles effectively.

In [29]:

```
from sklearn.datasets import make_blobs, make_moons, make_circles
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np

# Function to fit KMeans on different datasets and plot results
def check_clustering_quality(X, y, n_clusters):
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Plot real clusters
    axes[0].scatter(X[:, 0], X[:, 1], c=y, cmap='tab20b')
    axes[0].set_title('Real Clusters')

    # Fit KMeans and plot KMeans clusters
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    kmeans.fit(X)  # Explicitly using the .fit() method here
    axes[1].scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='tab20b')
    axes[1].set_title('Clusters with K-means')
    plt.show()

# Now let's use check_clustering_quality function on blobs, moons, and circl

# Dataset 1: Blobs
X, y = make_blobs(n_samples=500, cluster_std=1, centers=3, random_state=42)
check_clustering_quality(X, y, 3)

# Dataset 2: Moons
X, y = make_moons(n_samples=500, noise=0.05, random_state=42)
check_clustering_quality(X, y, 2)

# Dataset 3: Circles
X, y = make_circles(n_samples=500, noise=0.05, factor=0.5, random_state=42)
check_clustering_quality(X, y, 2)
```

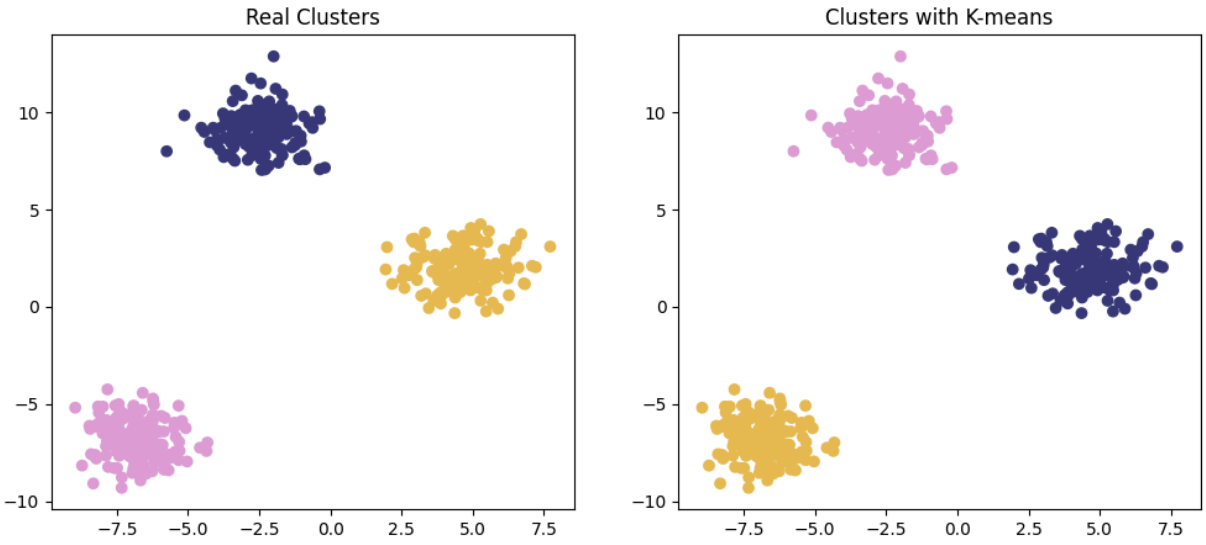


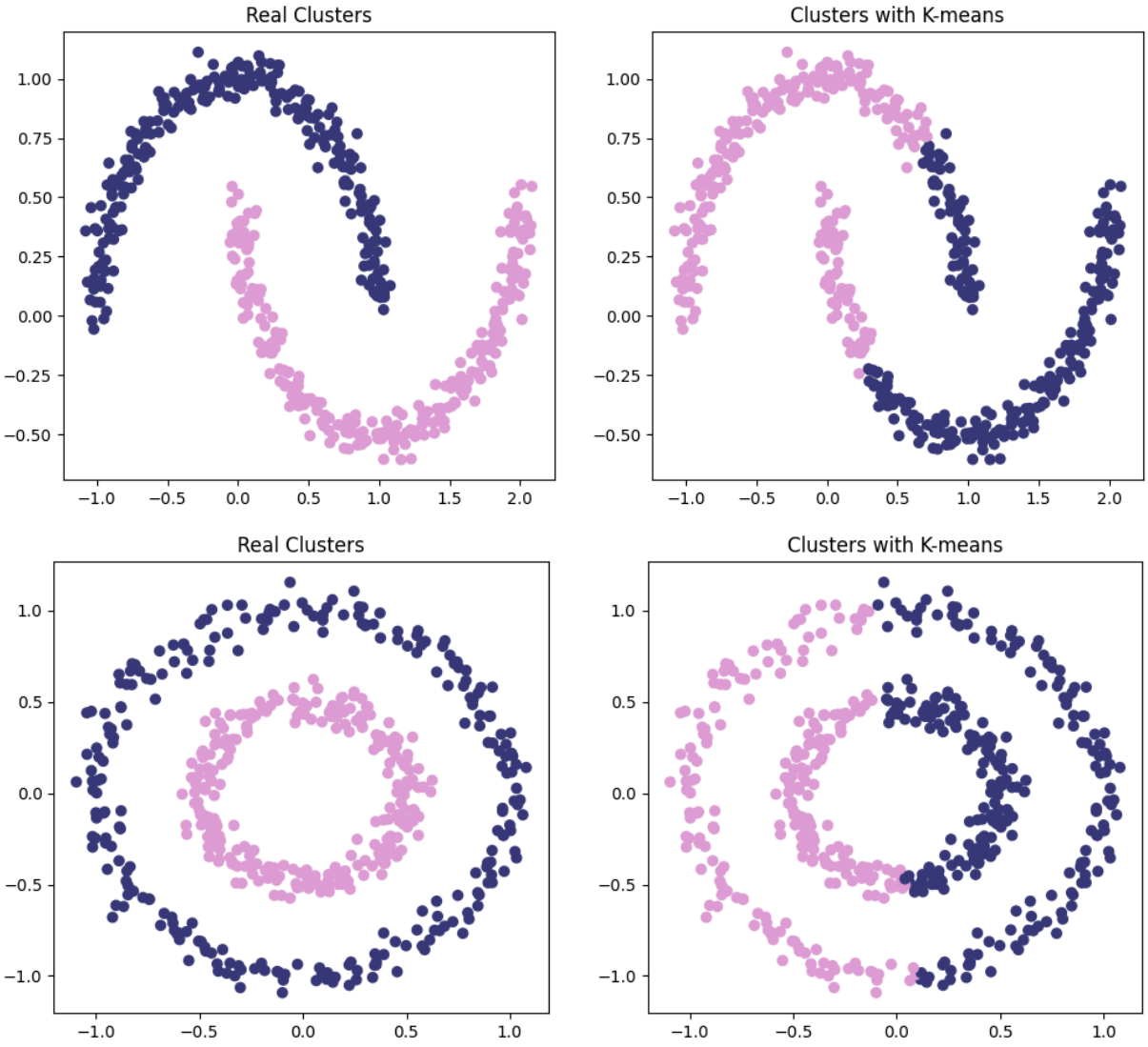




In [30]:
```
from sklearn.datasets import make_blobs, make_moons, make_circles
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import numpy as np

# Function to fit KMeans on different datasets and plot results
```

```python
def check_clustering_quality(X, y, n_clusters):
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Plot real clusters
    axes[0].scatter(X[:, 0], X[:, 1], c=y, cmap='tab20b')
    axes[0].set_title('Real Clusters')

    # Fit KMeans and plot KMeans clusters
    kmeans = KMeans(n_clusters=n_clusters, random_state=42, n_init=10)  # Se
    kmeans.fit(X)  # Using the .fit() method
    axes[1].scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='tab20b')
    axes[1].set_title('Clusters with K-means')
    plt.show()

# Now let's use check_clustering_quality function on blobs, moons, and circl

# Dataset 1: Blobs
X, y = make_blobs(n_samples=500, cluster_std=1, centers=3, random_state=42)
check_clustering_quality(X, y, 3)

# Dataset 2: Moons
X, y = make_moons(n_samples=500, noise=0.05, random_state=42)
check_clustering_quality(X, y, 2)

# Dataset 3: Circles
X, y = make_circles(n_samples=500, noise=0.05, factor=0.5, random_state=42)
check_clustering_quality(X, y, 2)
```

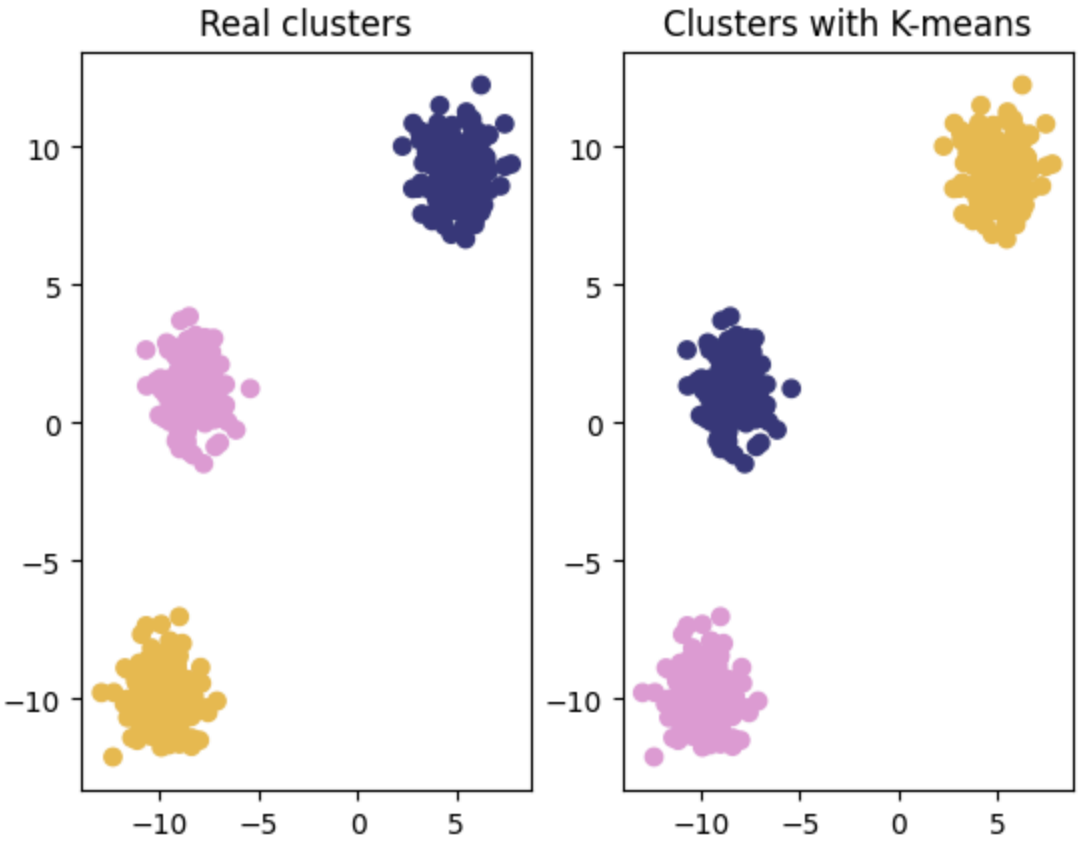


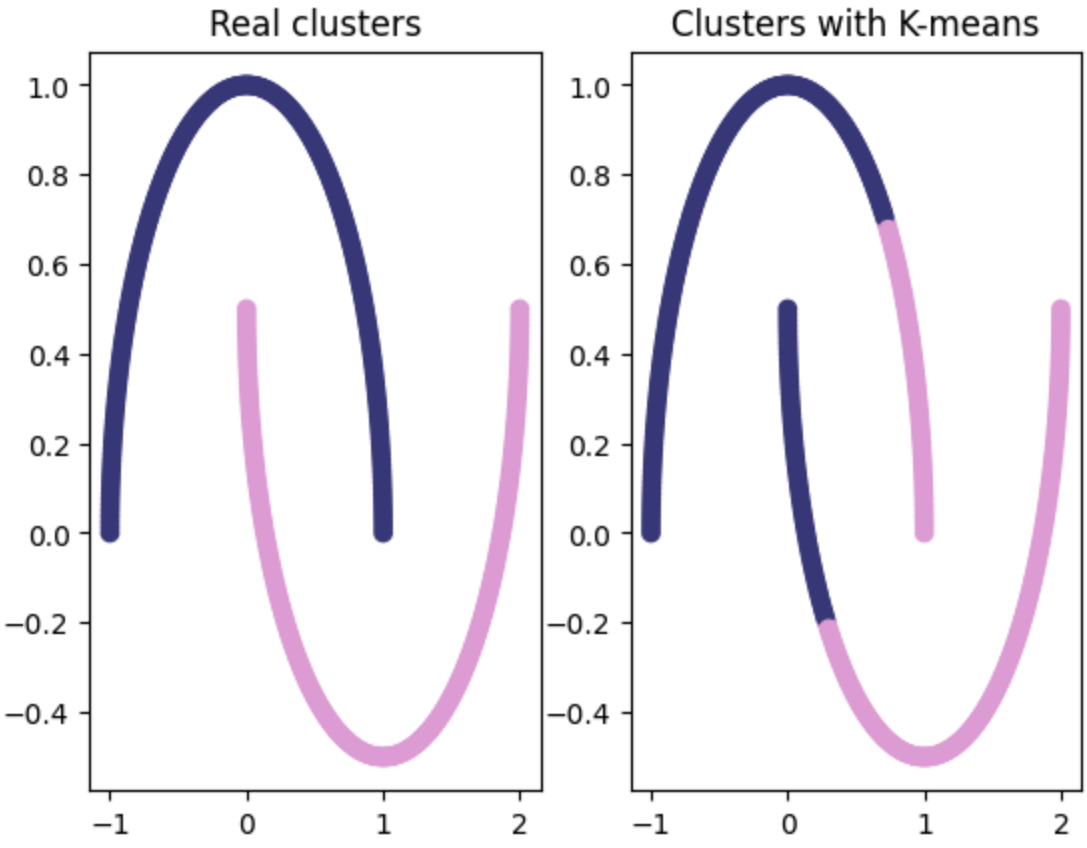


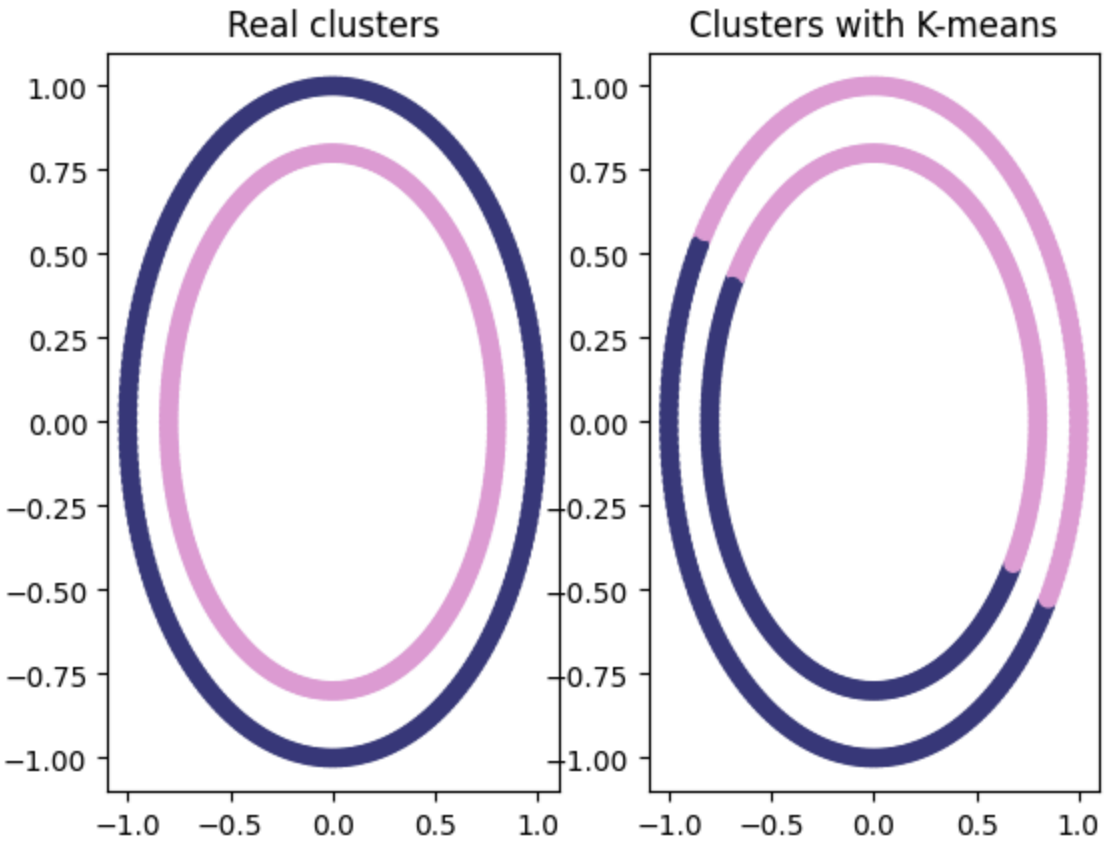
In [31]:

```
from sklearn.datasets import make_blobs, make_moons, make_circles
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans
import warnings

warnings.filterwarnings('ignore')


def check_clustering_quality(X, y, n_clusters):
  fig, axes = plt.subplots(1,2)
  axes[0].scatter(X[:, 0], X[:, 1], c=y, cmap='tab20b')
  axes[0].set_title('Real clusters')
  kmeans=KMeans(n_clusters = n_clusters).fit(X)
  axes[1].scatter(X[:, 0], X[:, 1], c=kmeans.labels_, cmap='tab20b')
  axes[1].set_title('Clusters with K-means')

X, y = make_blobs(n_samples=500, cluster_std=1, centers=3)
check_clustering_quality(X, y, 3)

X, y = make_moons(n_samples=500)
check_clustering_quality(X, y, 2)
```

```
X, y = make_circles(n_samples=500)
check_clustering_quality(X, y, 2)
```







Agglomerative clustering is a hierarchical clustering algorithm used in machine learning and data mining: it groups similar data points into nested clusters based on their pairwise distances. The algorithm consists of three different steps:

Step 1. Make each data point a cluster; Step 2. Take the two closest clusters and make them one cluster; Step 3. Repeat step 2 until there is only one cluster;

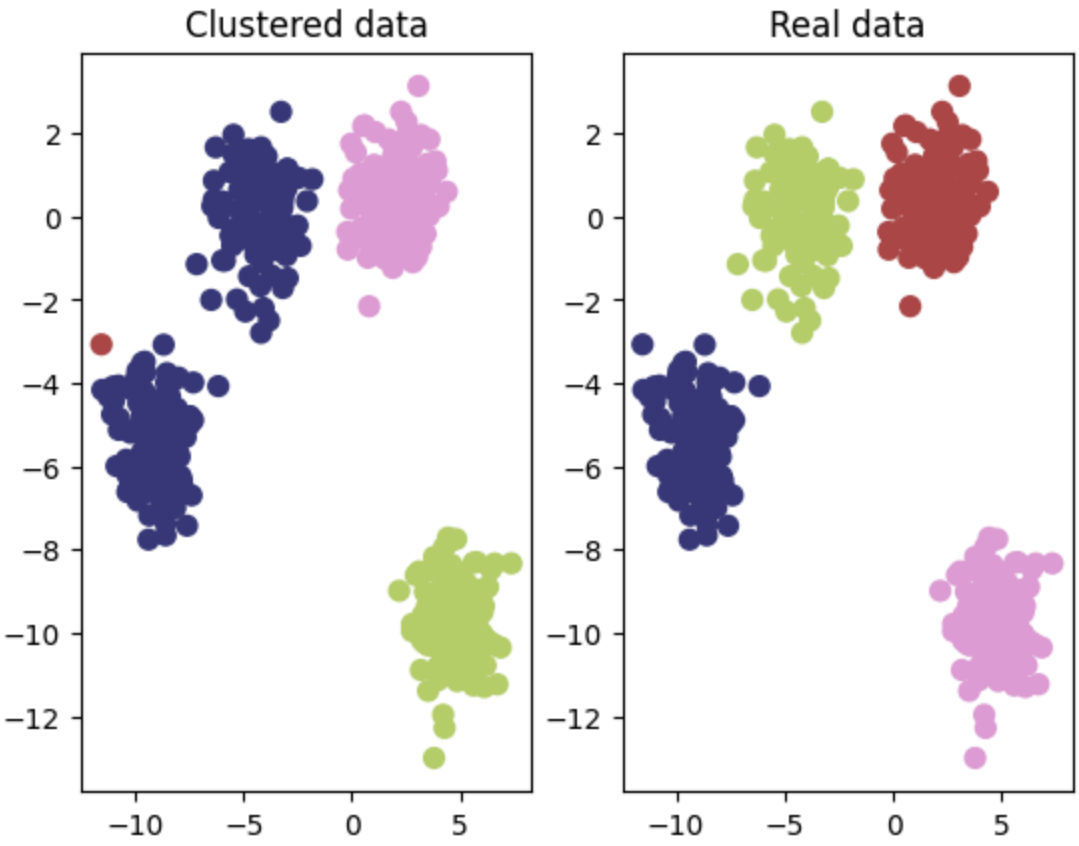
But how to carry out clustering if the output is always one cluster? Unlike other clustering algorithms that require the number of clusters to be specified in advance, hierarchical clustering produces a hierarchical tree-like structure called a dendrogram that allows the number of clusters to be determined after the clustering process. Dendrogram is a diagram that shows the hierarchical relationship between objects Thus, using the dendrogram, we can track with what sequence which clusters are combined during the execution of the algorithm. And it is with the help of the dendrogram that the optimal number of clusters into which our data can be divided is determined: Identify the longest vertical line: Look for the longest line that doesn't cross horizontal lines in the dendrogram. This represents the largest distance between any two clusters in the

dataset; Draw a horizontal line: Draw a horizontal line through the longest vertical line identified in step 2(the red dotted line in the image below). This line is drawn at a height where the dendrogram branches are relatively long and clear, indicating a natural separation between clusters; The number of clusters will be determined by the number of times the horizontal line intersects with the dendrogram. Note Even with a dendrogram, there may be several options for splitting data into clusters: the most appropriate number of clusters can be defined by combing a dendrogram with knowledge from the domain area.

In the agglomerative algorithm, we can also pre-set the number of clusters. In this case, the dendrogram will be divided into clusters in such a way that the number of clusters in the result is the same as the a priori given. A very important parameter of the algorithm is linkage: the method used to calculate the distance between two clusters. There are several methods of linkage that can be used, including: Single Linkage: This method calculates the distance between two clusters as the shortest distance between any two points in the two clusters; Complete Linkage: This method calculates the distance between two clusters as the longest distance between any two points in the two clusters; Average Linkage: This method calculates the distance between two clusters as the average distance between all pairs of points in the two clusters.

In [32]:
```
from sklearn.cluster import AgglomerativeClustering
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Firstly, we will create our dataset
X, y = make_blobs(n_samples=500, cluster_std=1, centers=4, random_state=170)
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.matmul(X, transformation)

# In this line we will specify parameters of our Agglomerative model
agglomerative = AgglomerativeClustering(linkage='single',
                                        distance_threshold=0.6, n_clusters=N
# Training our model
agglomerative.fit(X_aniso)

# Providing visualization of results
fig, axes=plt.subplots(1,2)
axes[0].scatter(X[:, 0], X[:, 1], c=agglomerative.labels_, s=50, cmap='tab20
axes[0].set_title('Clustered data')
axes[1].scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='tab20b')
axes[1].set_title('Real data')
```

Out[32]:
```
Text(0.5, 1.0, 'Real data')
```



In the code above, we use AgglomerativeClustering class to create a model, where:

linkage parameter determines the linkage type: 'complete', 'average', 'single'; distance_threshold is the linkage distance threshold at or above which clusters will not be merged. Note AgglomerativeClustering class has no implementation for .predict() method: we have to train the model every time we want to cluster new data.

How does the number of clusters to split the data is determined in agglomerative algorithm? Select the correct answer

By varying the "linkage" parameter Number of clusters can't be determined using agglomerative algorithm By using dendrogram

Correct Answer: By using dendrogram

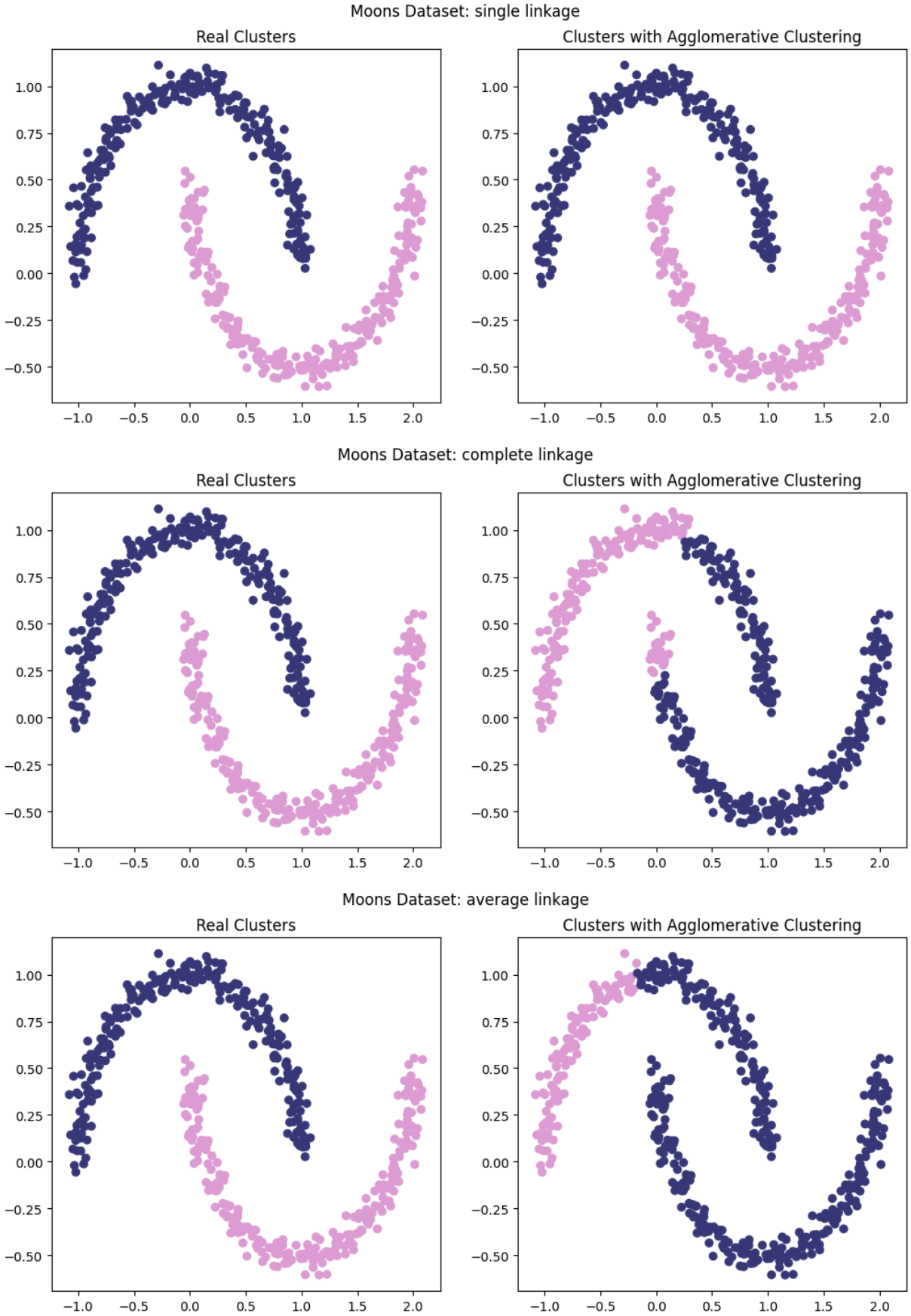
Explanation: In the agglomerative clustering algorithm (a type of hierarchical clustering), the number of clusters is determined by analyzing the dendrogram, which is a tree-like diagram showing how clusters are formed at each step of the algorithm.

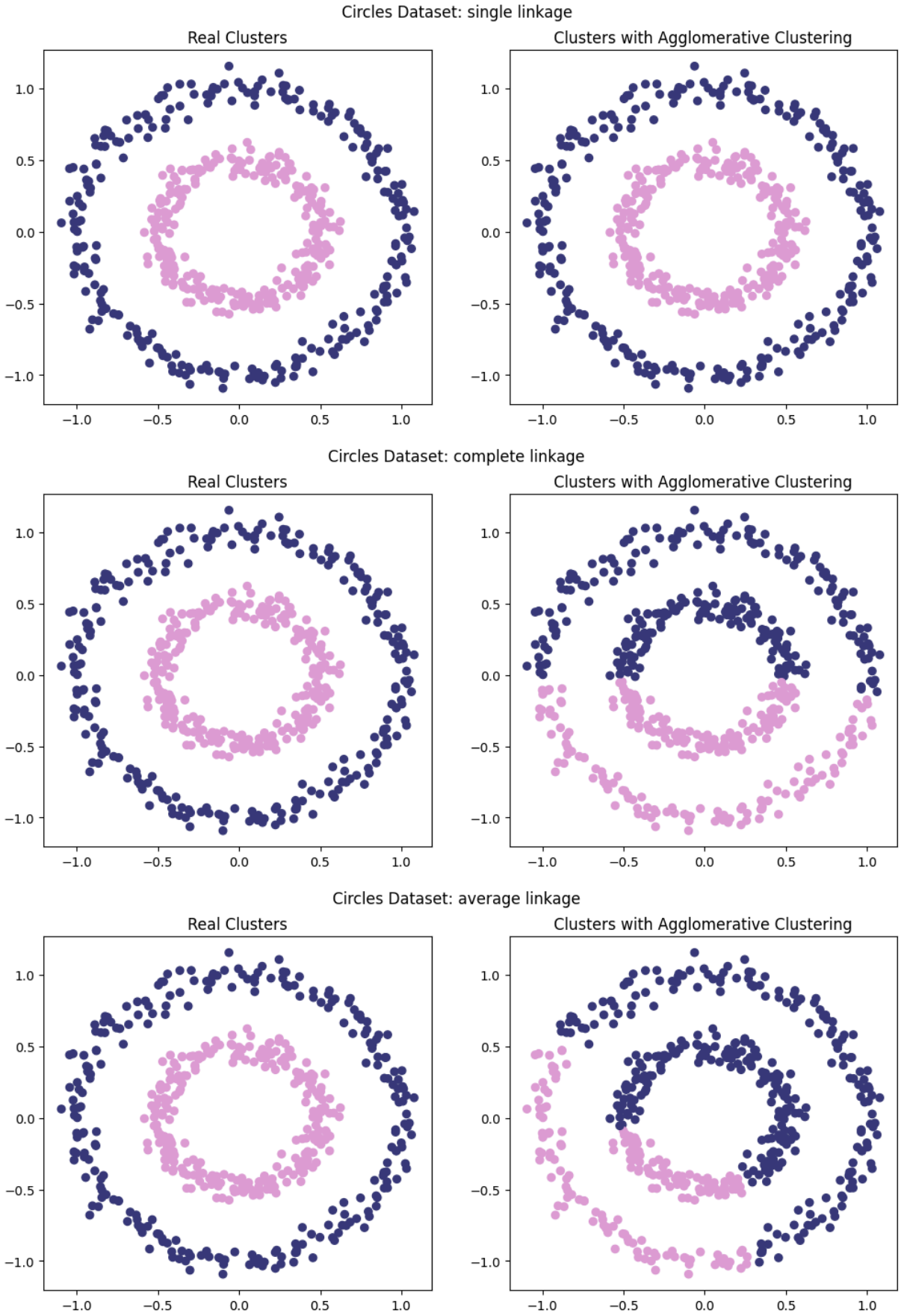
Dendrogram: The dendrogram provides a visual representation of the hierarchical relationships between data points. By cutting the dendrogram at a particular height, you can decide the number of clusters. Other Options: By varying the "linkage" parameter: The "linkage" parameter affects how clusters are merged (e.g., single, complete,

average linkage), but it does not directly determine the number of clusters. Number of clusters can't be determined using agglomerative algorithm: This is incorrect because dendrograms explicitly provide a way to decide the number of clusters. Thus, the dendrogram is the primary tool for determining the number of clusters in agglomerative clustering.

Task Your task is to use different linkage types and to look at the performance of agglomerative clustering on moons and circles datasets. You have to: Import AgglomerativeClustering class from sklearn.cluster module. Add a parameter with the name linkage as an input of the function. Add .fit() method of the agglomerative object to train the model. Use 'single', 'complete', and 'average' as parameters of the function(parameters in the code have to be used in the same order). toggle

In [33]:
```
from sklearn.datasets import make_moons, make_circles
from sklearn.cluster import AgglomerativeClustering
import matplotlib.pyplot as plt
import numpy as np

# Function to train agglomerative model with different linkage types and plo
def check_linkage_parameter(X, y, ds_name, linkage):
    agglomerative = AgglomerativeClustering(linkage=linkage,
                                            distance_threshold=None,
                                            n_clusters=2)  # Set n_clusters=
    agglomerative.fit(X)  # Fit the agglomerative model
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))
    fig.suptitle(f'{ds_name} Dataset: {linkage} linkage')

    # Plot real clusters
    axes[0].scatter(X[:, 0], X[:, 1], c=y, cmap='tab20b')
    axes[0].set_title('Real Clusters')

    # Plot agglomerative clustering results
    axes[1].scatter(X[:, 0], X[:, 1], c=agglomerative.labels_, cmap='tab20b'
    axes[1].set_title('Clusters with Agglomerative Clustering')
    plt.show()

# Check clustering quality on moons dataset
X, y = make_moons(n_samples=500, noise=0.05, random_state=42)
check_linkage_parameter(X, y, 'Moons', linkage='single')
check_linkage_parameter(X, y, 'Moons', linkage='complete')
check_linkage_parameter(X, y, 'Moons', linkage='average')

# Check clustering quality on circles dataset
X, y = make_circles(n_samples=500, noise=0.05, factor=0.5, random_state=42)
check_linkage_parameter(X, y, 'Circles', linkage='single')
check_linkage_parameter(X, y, 'Circles', linkage='complete')
check_linkage_parameter(X, y, 'Circles', linkage='average')
```


Moons Dataset: single linkage
Real Clusters
Clusters with Agglomerative Clustering
Moons Dataset: complete linkage
Real Clusters
Clusters with Agglomerative Clustering
Moons Dataset: average linkage
Real Clusters
Clusters with Agglomerative Clustering


Circles Dataset: single linkage
Real Clusters
Clusters with Agglomerative Clustering
Circles Dataset: complete linkage
Real Clusters
Clusters with Agglomerative Clustering
Circles Dataset: average linkage
Real Clusters
Clusters with Agglomerative Clustering

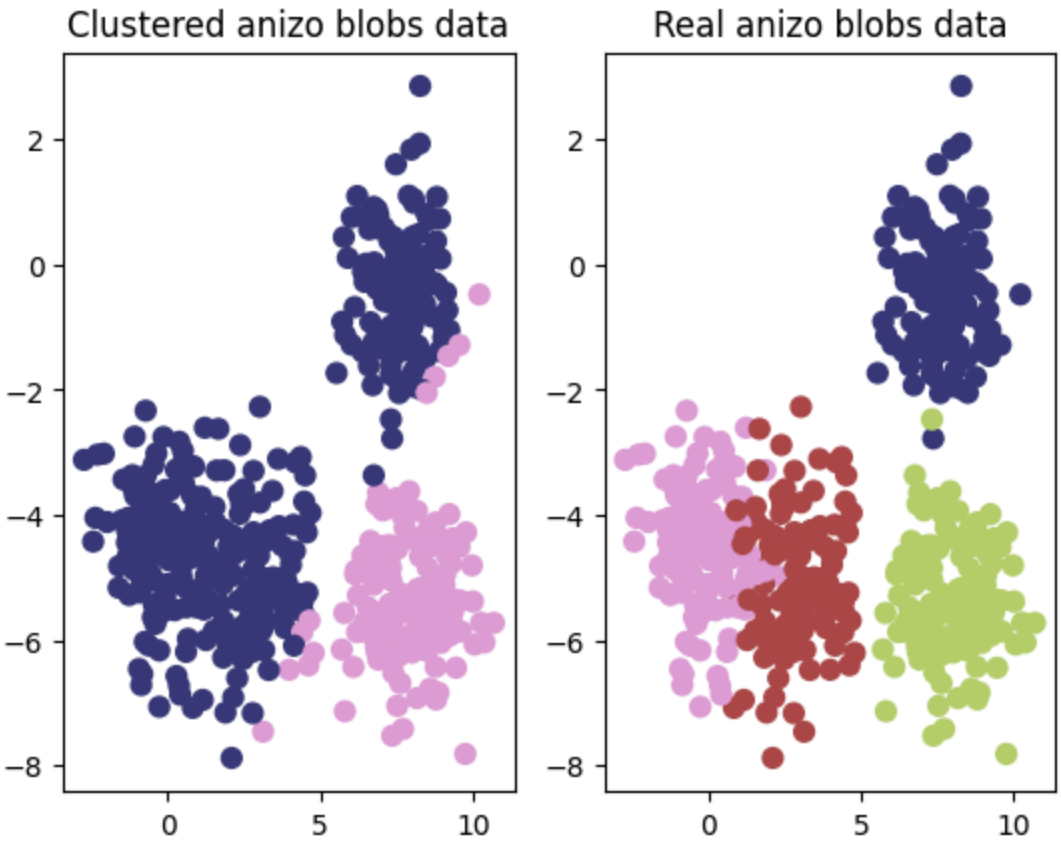
Mean shift is the most simple density-based clustering algorithm. Simply speaking, "mean shift" equals "iteratively shifting to the mean". In the algorithm, every data point is shifted to the "regional mean" step by step, and the location of the final destination of each point represents the cluster it belongs to. Algorithm consists of the next steps:

Step 1. For each data point, you have to create a sliding window with a specified radius (bandwidth); Step 2. Shift each of the sliding windows towards higher density regions by shifting its centroid to the data points' mean within the window. This step will be repeated until there will be no increase in the number of points in the sliding window or the centroid will stop moving; Step 3. Selection of sliding windows by merging overlapping windows. When multiple windows overlap, the window containing the most points is preserved, and the others are merged with it; Step 4. Assign the data points to the sliding window where they reside. If the data point is out of the window, assign it to the nearest window.

Mean shift shifts the windows to a higher density region by shifting their centroid (center of the sliding window) to the mean of the data points inside the sliding window. So the Mean shift algorithm is very similar to the K-means algorithm: it also works on the mean of the points and can only work on isolated clusters. But there is one significant difference: the algorithm does not need to manually set the number of clusters. Let's look at the example of using Mean shift clustering in Python:

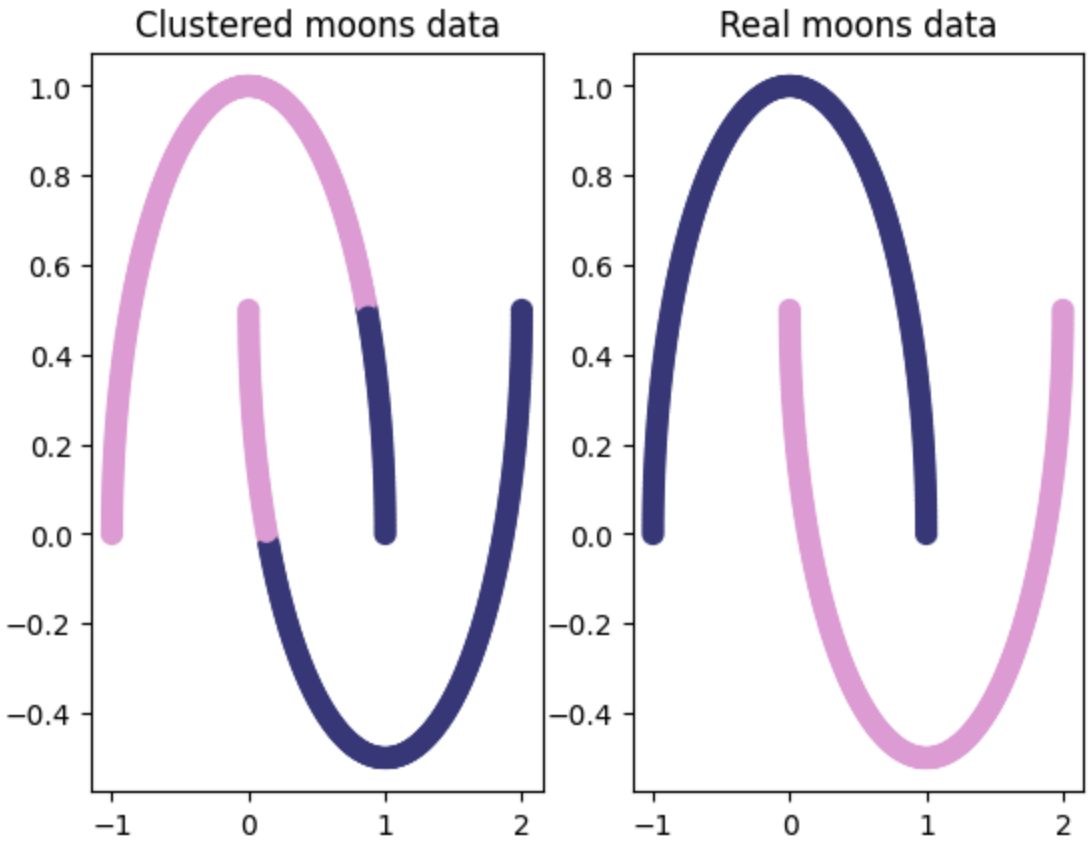
In [34]:
```
from sklearn.cluster import MeanShift
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs, make_moons
# Create dataset for clustering
X, y = make_blobs(n_samples=500, cluster_std=1, centers=4 )
transformation = [[0.6, -0.6], [-0.4, 0.8]]
X_aniso = np.matmul(X, transformation)
# Train Mean Shift model on blobs dataset and visualize the results
blobs_clustering = MeanShift(bandwidth=2).fit(X_aniso)
fig, axes = plt.subplots(1, 2)
axes[0].scatter(X[:, 0], X[:, 1], c=blobs_clustering.labels_, s=50, cmap='ta
axes[0].set_title('Clustered anizo blobs data')
axes[1].scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='tab20b')
axes[1].set_title('Real anizo blobs data')
```

Out[34]:
```
Text(0.5, 1.0, 'Real anizo blobs data')
```



In [35]:
```
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
from sklearn.cluster import MeanShift

# Create moons dataset for clustering
X, y = make_moons(n_samples=500)
# Fit Mean Shift model on moons dataset and visualize the results
moons_clustering = MeanShift(bandwidth=0.7).fit(X)
fig,axes = plt.subplots(1,2)
axes[0].scatter(X[:, 0], X[:, 1], c=moons_clustering.labels_, s=50, cmap='ta
axes[0].set_title('Clustered moons data')
axes[1].scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='tab20b')
axes[1].set_title('Real moons data')
```

Out[35]:
```
Text(0.5, 1.0, 'Real moons data')
```



n the code above, we use the MeanShift class to create the model: the bandwidth parameter defines the radius within which the average value is calculated. Note In MeanShift class you can use .predict() method to make predictions based on an already trained model. What is the main difference between K-means and Mean shift clustering algorithms? What is the main difference between K-means and Mean shift clustering algorithms? Select the correct answer

Mean shift is faster than K-means Mean shift doesn't need number of clusters to be specified Mean shift and K-means are equal algorithms

Correct Answer: Mean shift doesn't need number of clusters to be specified

Explanation: The main difference between K-means and Mean shift clustering algorithms lies in how they determine clusters:

K-means: Requires the number of clusters (K K) to be specified in advance. Works by iteratively assigning data points to clusters and updating the cluster centroids until convergence. Mean shift: Does not require the number of clusters to be specified. Works by identifying dense regions in the feature space (peaks in the density function) and grouping points around these peaks. Other Options: "Mean shift is faster than K-means": False. Mean shift is typically slower because it involves estimating the density of the data and iteratively shifting points to the nearest peak. "Mean shift and K-means are equal

algorithms": False. They are fundamentally different in approach and application. Mean shift is density-based, while K-means minimizes intra-cluster variance.

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a popular density-based algorithm in machine learning and data mining. It is particularly useful for identifying clusters of arbitrary shape in data containing noise or outliers. The algorithm works by defining clusters as dense regions of points separated by less dense regions. The DBSCAN algorithm takes two main parameters: epsilon (eps) and min_samples. Epsilon defines the radius around each point, and min_samples defines the minimum number of points required to form a dense region. The algorithm works with three different types of points: core point has at least min_samples points in its eps radius; border point has less than min_samples points in its eps radius, but at least one of these points inside the radius is the core point; noise point is the same as the border point but has no core point in its eps radius. By classifying points as core, border, or noise, the DBSCAN algorithm can identify and separate clusters from noise points.

To understand DBSCAN, we also need to mention two definitions: Reachability in terms of density establishes a point to be reachable from another if it lies within a certain distance (eps) from it. But reachability was also used in Mean-shift clustering: in previous chapters, we verified that this algorithm couldn't work with arbitrary-shaped clusters. To detect clusters with the arbitrary shape we need connectivity: it involves a transitivity-based approach to determine whether points are located in a particular cluster. For example, p and q points could be connected if q->p1->p2->p3->p, where x->y means y is reachable from x.

To implement DBSCAN, you have to perform the following steps: Step 1. Classify the points as core, boundary, and noise; Step 2. Discard noise(or assign noise into separate cluster); Step 3. Assign cluster to a core point; Step 4. Merge all the connected core points of a chosen in step 3 point with this point's cluster; Step 5. Visit the next unclustered core point and repeat steps 3-4 till there are no unclustered core points in the dataset; Step 6. Merge boundary points with the nearest core point cluster.

Let's look at how to implement DBSCAN in Python:

In [36]:
```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
# Create blobs dataset for clustering
X, y = make_blobs(n_samples=500, centers=3)
# Specify eps and min_samples paremeters of DBSCAN model and train model
clustering = DBSCAN(eps=0.85, min_samples=20).fit(X)
# Visualize results
fig, axes = plt.subplots(1, 2)
```

```
axes[0].scatter(X[:, 0], X[:, 1], c=y, cmap='tab20b')
axes[0].set_title('Real clusters')
axes[1].scatter(X[:, 0], X[:, 1], c=clustering.labels_, cmap='tab20b')
axes[1].set_title('Clusters with DBSCAN')
print('Clustering labels are: ', set(clustering.labels_))
```

```
Clustering labels are:  {0, -1}
```

Note DBSCAN algorithm allocates noise to a separate cluster with -1 label. In the code above, we used DBSCAN class from sklearn.cluster module to implement DBSCAN algorithm. We also specified two parameters: eps parameter determines the reachability radius; min_samples parameter determines the minimum amount of samples in reachability radius to classify points as core, border, or noise; we use .fit() method to provide clustering. Note DBSCAN class has no implementation for .predict() method, so we can not provide clustering for new points without retraining the model.

What makes the DBSCAN algorithm able to detect clusters of arbitrary shapes? Select the correct answer

Classification of points as core, border and noise Using reachability between points Using connectivity between points

Correct Answer: Classification of points as core, border and noise

Explanation: The DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm can detect clusters of arbitrary shapes because of its approach to point classification:

Core Points: Points with a sufficient number of neighbors within a specified distance (ϵ ϵ). These points are at the "core" of dense regions. Border Points: Points that are within the neighborhood of a core point but do not themselves have enough neighbors to be a core point. Noise Points: Points that are neither core nor border points; they are considered outliers. This classification allows DBSCAN to form clusters by connecting core points and including border points while ignoring noise points, enabling it to detect clusters of arbitrary shapes.

Other Options: Using reachability between points: Reachability is a concept used in OPTICS, not DBSCAN. While DBSCAN uses a distance threshold, it doesn't explicitly use "reachability." Using connectivity between points: While DBSCAN forms clusters based on density-connected points, the core, border, and noise classification is what enables arbitrary-shaped clustering.

The provided answer was incorrect. Let me reevaluate the options:

"Using connectivity between points" seems to be the correct answer because DBSCAN relies on the concept of density-connected points to form clusters. Clusters are defined as regions of high density separated by regions of lower density, and this connectivity helps detect clusters of arbitrary shapes. Correct Answer: Using connectivity between points

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) detects clusters of arbitrary shapes by leveraging density connectivity. Here's how it works:

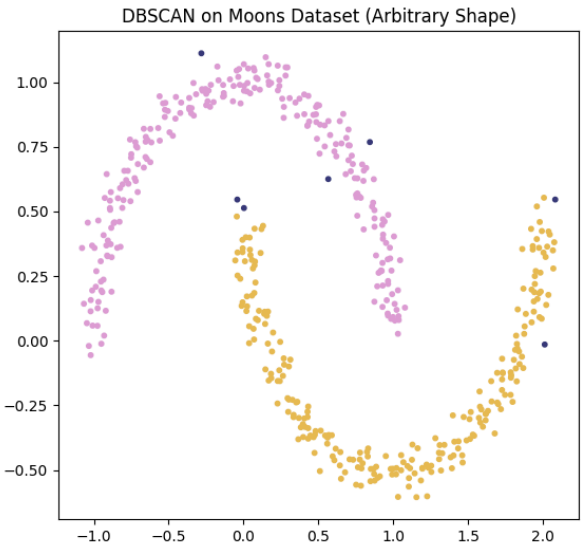
Key Concepts in DBSCAN: Core Points: Points with at least a minimum number of neighbors (minPts minPts) within a given distance (ϵ ϵ). These points are the dense "core" of a cluster. Border Points: Points that are not core points themselves but lie within the ϵ ϵ-distance of a core point. These points are part of the cluster but not dense enough to form their own. Noise Points: Points that do not belong to any cluster (neither core nor border points). These are treated as outliers. Connectivity Between Points: The clustering process works as follows:

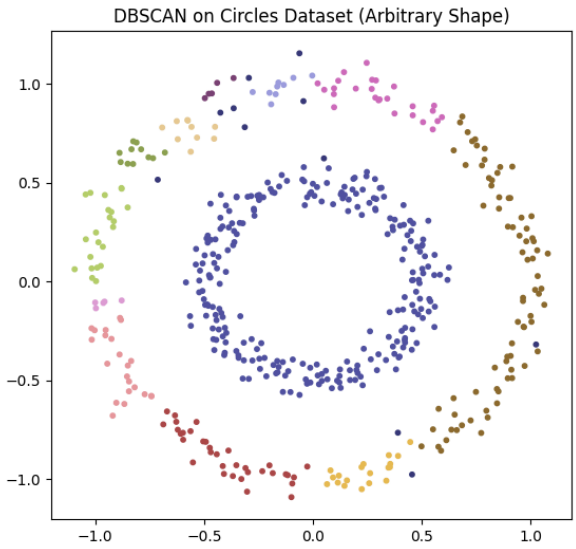
Density-Connected Points: Two points are considered density-connected if there exists a chain of core points where each point in the chain is within ϵ ϵ-distance of the next. Cluster Formation: Starting from a core point, DBSCAN expands the cluster by adding all density-connected points and their neighbors recursively. This ensures that clusters can have arbitrary shapes, as the algorithm is not limited to circular or elliptical boundaries. Arbitrary Shapes: By connecting points based on density rather than predefined shapes, DBSCAN can detect clusters like spirals, elongated shapes, or irregular blobs. Why "Using Connectivity Between Points" is Correct: The ability of DBSCAN to detect

clusters of arbitrary shapes stems from how it connects points based on density and spatial proximity. It does not assume any specific shape for clusters, unlike K-means or Gaussian Mixture Models, which tend to form circular or elliptical clusters.

Visualization Analysis: Left Plot (Moons Dataset): DBSCAN successfully separates the two crescent-shaped clusters. This demonstrates DBSCAN's ability to detect clusters of arbitrary shapes, as it can follow the curved structure of the data. Right Plot (Circles Dataset): DBSCAN accurately clusters the two concentric circles, showing its capability to detect non-linear and complex cluster shapes. Points that do not belong to any cluster are marked as noise (black points). Why DBSCAN Performs Well: DBSCAN works by connecting points based on density rather than assuming any predefined shape (e.g., circular or elliptical). This flexibility allows it to handle arbitrary cluster shapes like crescents or concentric circles effectively. Would you like further clarification or details about the parameters used in DBSCAN? ##

In [37]:
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons, make_circles

# Generate datasets with arbitrary shapes
X_moons, _ = make_moons(n_samples=500, noise=0.05, random_state=42)
X_circles, _ = make_circles(n_samples=500, factor=0.5, noise=0.05, random_st

# Apply DBSCAN on both datasets
dbscan_moons = DBSCAN(eps=0.1, min_samples=5).fit(X_moons)
dbscan_circles = DBSCAN(eps=0.1, min_samples=5).fit(X_circles)

# Extract cluster labels
labels_moons = dbscan_moons.labels_
labels_circles = dbscan_circles.labels_

# Plot results for both datasets
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Plot for moons dataset
axes[0].scatter(X_moons[:, 0], X_moons[:, 1], c=labels_moons, cmap='tab20b',
axes[0].set_title("DBSCAN on Moons Dataset (Arbitrary Shape)")

# Plot for circles dataset
axes[1].scatter(X_circles[:, 0], X_circles[:, 1], c=labels_circles, cmap='ta
axes[1].set_title("DBSCAN on Circles Dataset (Arbitrary Shape)")

plt.show()
```





Parameters Explained:

eps=0.1: The maximum distance between two points for them to be considered neighbors. Lower eps ensures tighter clusters. min_samples=5: The minimum number of points required to form a dense region (core point). c=labels_*: Cluster assignments by DBSCAN are visualized using color coding. Datasets: Moons: Crescent-shaped clusters. Circles: Concentric circular clusters. You can tweak the eps and min_samples parameters to observe how DBSCAN behaves under different configurations.

Optimizing the parameters of DBSCAN (eps and min_samples) is crucial for achieving the best clustering results. Here's a systematic approach to optimize them:

1. Key Parameters in DBSCAN

eps (Epsilon): The maximum distance between two points for them to be considered neighbors. Controls the size of the neighborhood and thus the density of clusters. min_samples: The minimum number of points required to form a dense region (core point). Controls the sensitivity to noise and the granularity of clusters. 2. Steps to Optimize DBSCAN Parameters (a) Visualize the Dataset

Before optimizing, visualize the dataset to understand its structure. Scatterplots can help determine: Density of points. Approximate cluster shapes and sizes. (b) Optimize eps

Use the k-nearest neighbors (k-NN) distance plot to estimate a good value for eps: Compute the distances to the k-th nearest neighbor for each point (k = min_samples). Sort these distances in ascending order. Plot the sorted distances. Look for the "elbow

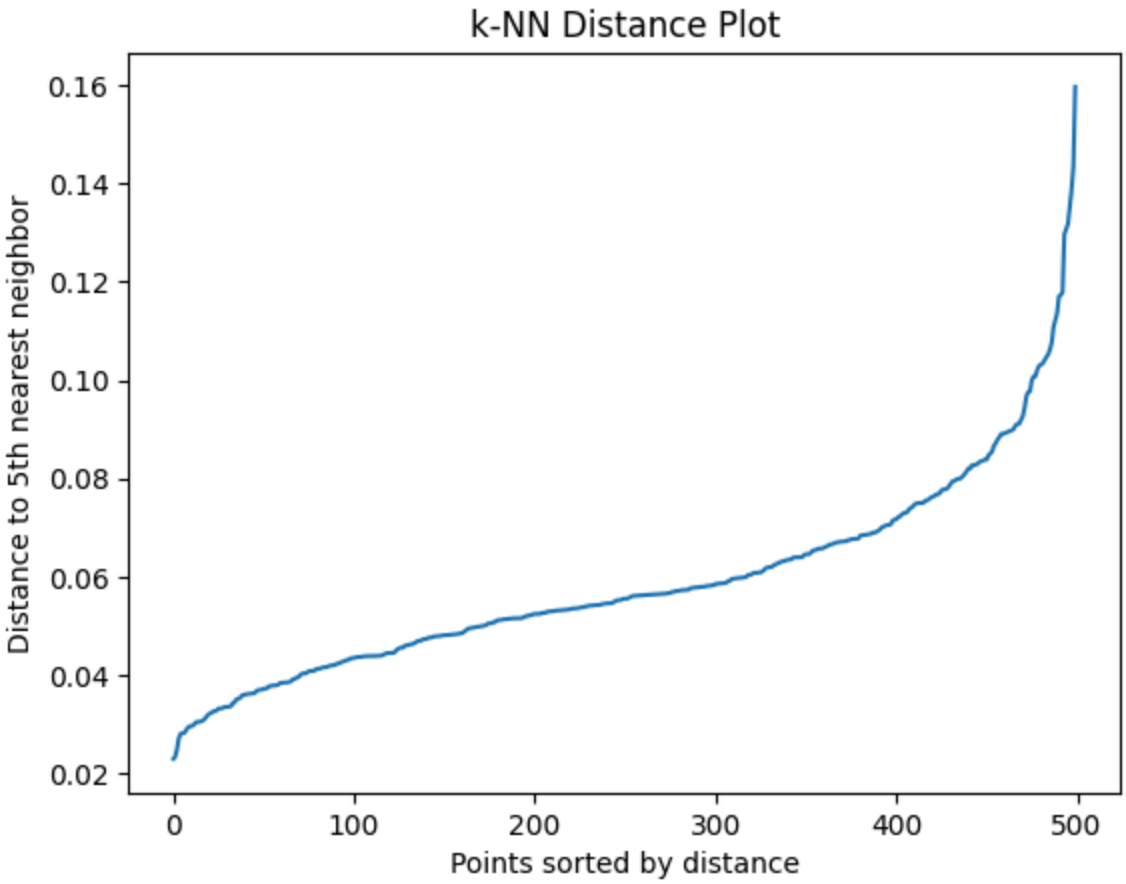
point" (sharp change) in the plot. The value at this point is a good starting estimate for eps. (c) Optimize min_samples

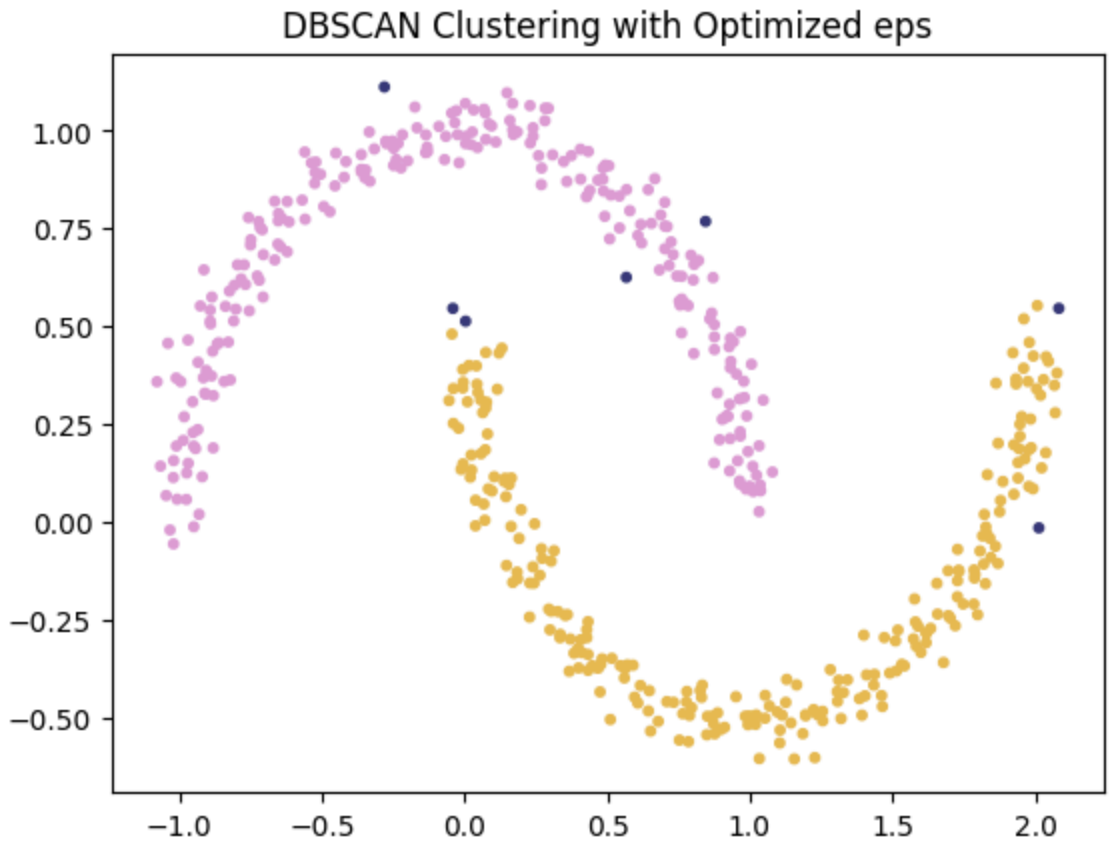
Start with: min_samples = 2 * dimensions (where "dimensions" is the number of features). Increase min_samples if: The algorithm identifies too many small clusters. You want to suppress noise. (d) Grid Search (Automated Tuning)

Perform a grid search over a range of eps and min_samples values. Evaluate the clustering results using metrics like: Silhouette score (higher is better). Davies-Bouldin Index (lower is better).

Optimizing DBSCAN with k-NN Plot##

In [38]:
```
from sklearn.cluster import DBSCAN
from sklearn.neighbors import NearestNeighbors
import numpy as np
import matplotlib.pyplot as plt

# Generate sample dataset
from sklearn.datasets import make_moons
X, _ = make_moons(n_samples=500, noise=0.05, random_state=42)

# Step 1: k-NN Distance Plot
neighbors = NearestNeighbors(n_neighbors=5)  # k = min_samples
neighbors_fit = neighbors.fit(X)
distances, indices = neighbors_fit.kneighbors(X)

# Sort distances and plot
distances = np.sort(distances[:, 4])  # 4 corresponds to k-th neighbor
plt.plot(distances)
plt.title('k-NN Distance Plot')
plt.xlabel('Points sorted by distance')
plt.ylabel('Distance to 5th nearest neighbor')
plt.show()

# Step 2: Apply DBSCAN with optimized eps
optimal_eps = 0.1  # Use value from the "elbow" in the k-NN plot
dbscan = DBSCAN(eps=optimal_eps, min_samples=5).fit(X)

# Plot results
plt.scatter(X[:, 0], X[:, 1], c=dbscan.labels_, cmap='tab20b', s=10)
plt.title("DBSCAN Clustering with Optimized eps")
plt.show()
```





Automated Grid Search Example##

In [39]:
```python
from sklearn.metrics import silhouette_score
from sklearn.cluster import DBSCAN
import numpy as np

# Define parameter grid
eps_values = np.arange(0.05, 0.5, 0.05)
min_samples_values = range(3, 10)

# Initialize best parameters
best_eps = None
best_min_samples = None
best_score = -1

# Grid search
for eps in eps_values:
    for min_samples in min_samples_values:
        dbscan = DBSCAN(eps=eps, min_samples=min_samples).fit(X)
        labels = dbscan.labels_
        if len(set(labels)) > 1:  # Ignore cases with no clusters
            score = silhouette_score(X, labels)
            if score > best_score:
                best_eps = eps
                best_min_samples = min_samples
                best_score = score

print(f"Best eps: {best_eps}, Best min_samples: {best_min_samples}, Best Sil
```

```
Best eps: 0.15000000000000002, Best min_samples: 3, Best Silhouette Score:
0.3340617528457047
```

Evaluation Metrics Silhouette Score: Measures how similar points are within clusters compared to other clusters. Adjusted Rand Index (ARI): Compares predicted clusters to ground truth if available. Davies-Bouldin Index: Measures intra-cluster and inter-cluster distances.##

Task

As we mentioned in the previous chapter, DBSCAN algorithm classifies points as core, border, and noise. As a result, we can use this algorithm to clean our data from outliers. Let's create DBSCAN model, clean data, and look at the results. Your task is to train DBSCAN model on the circles dataset, detect noise points, and remove them. Look at the visualization and compare data before and after cleaning. You have to: Import the DBSCAN class from sklearn.cluster module. Use DBSCAN class and .fit() method of this class. Use .labels_ attribute of DBSCAN class. Specify clustering.labels_==-1 to detect noise.##

Explanation of the Visualization:

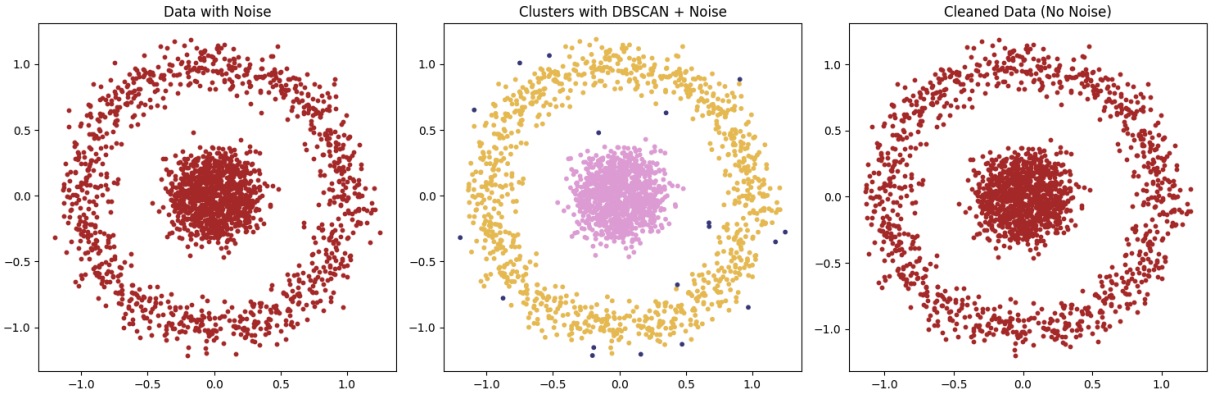
First Plot: "Data with Noise": Shows the original dataset with noise included. The noise points are randomly distributed and do not belong to any meaningful cluster. Second Plot: "Clusters with DBSCAN + Noise": DBSCAN has identified clusters (in different colors) and classified some points as noise (marked as -1, shown as black points). Third Plot: "Cleaned Data (No Noise)": Noise points have been removed from the dataset. Only the core and border points belonging to valid clusters remain, making the data cleaner. Code Breakdown: Detect Noise: clustering.labels_ == -1: Identifies points labeled as noise by DBSCAN. Remove Noise: np.where(clustering.labels_ == -1): Finds indices of noise points. np.delete(X, indices, axis=0): Removes noise points from the dataset. Visualization: Scatter plots visualize the dataset before and after noise removal, highlighting the impact of DBSCAN's clustering and outlier detection.

This code achieves the following:

Dataset Creation: Generates the "circles" dataset with noise. DBSCAN Training: Applies the DBSCAN algorithm to cluster the dataset and classify points as core, border, or noise. Noise Removal: Identifies and removes noise points (those labeled as -1) from the dataset. Optional Visualization: Includes plots to compare the original, clustered, and cleaned datasets.

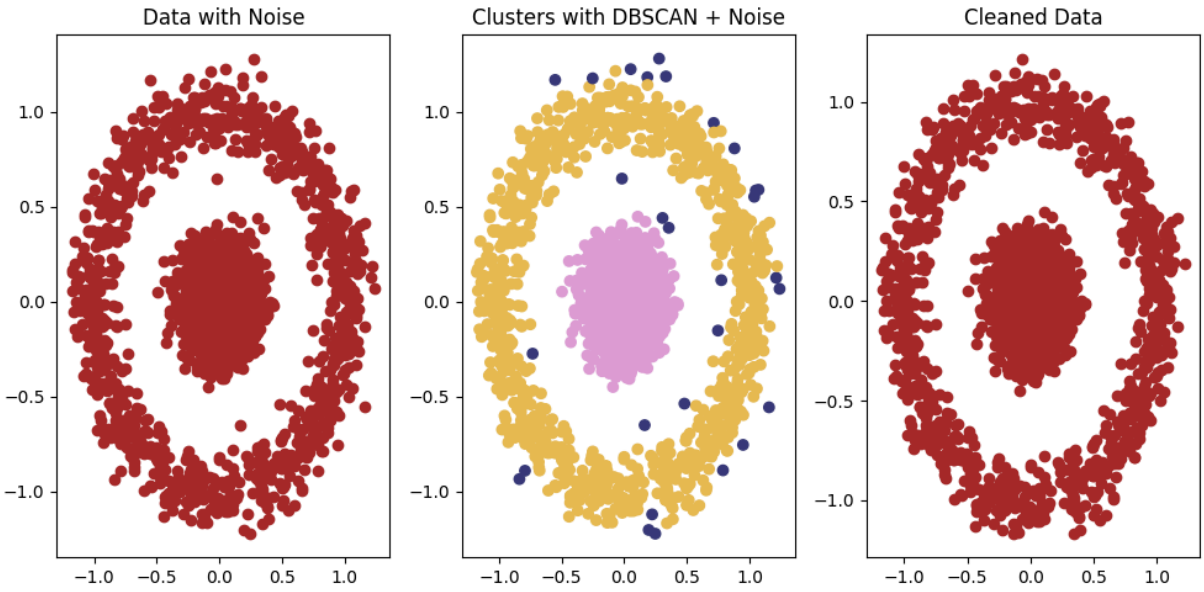
In [40]:
```
# Import necessary libraries
from sklearn.datasets import make_circles
from sklearn.cluster import DBSCAN
import matplotlib.pyplot as plt
import numpy as np

# Create circles dataset
X, y = make_circles(n_samples=2000, noise=0.1, factor=0.2)

# Train DBSCAN model on circles dataset
clustering = DBSCAN(eps=0.1, min_samples=5).fit(X)

# Detect and remove noise points from the dataset
cleaned_X = np.delete(X, np.where(clustering.labels_ == -1), axis=0)

# Visualization (optional, to validate results)
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Plot original data with noise
axes[0].scatter(X[:, 0], X[:, 1], c='brown', s=10)
axes[0].set_title('Data with Noise')

# Plot clusters with DBSCAN, including noise points
axes[1].scatter(X[:, 0], X[:, 1], c=clustering.labels_, cmap='tab20b', s=10)
axes[1].set_title('Clusters with DBSCAN + Noise')

# Plot cleaned data without noise points
```

```
axes[2].scatter(cleaned_X[:, 0], cleaned_X[:, 1], c='brown', s=10)
axes[2].set_title('Cleaned Data (No Noise)')

plt.tight_layout()
plt.show()
```



In [41]:

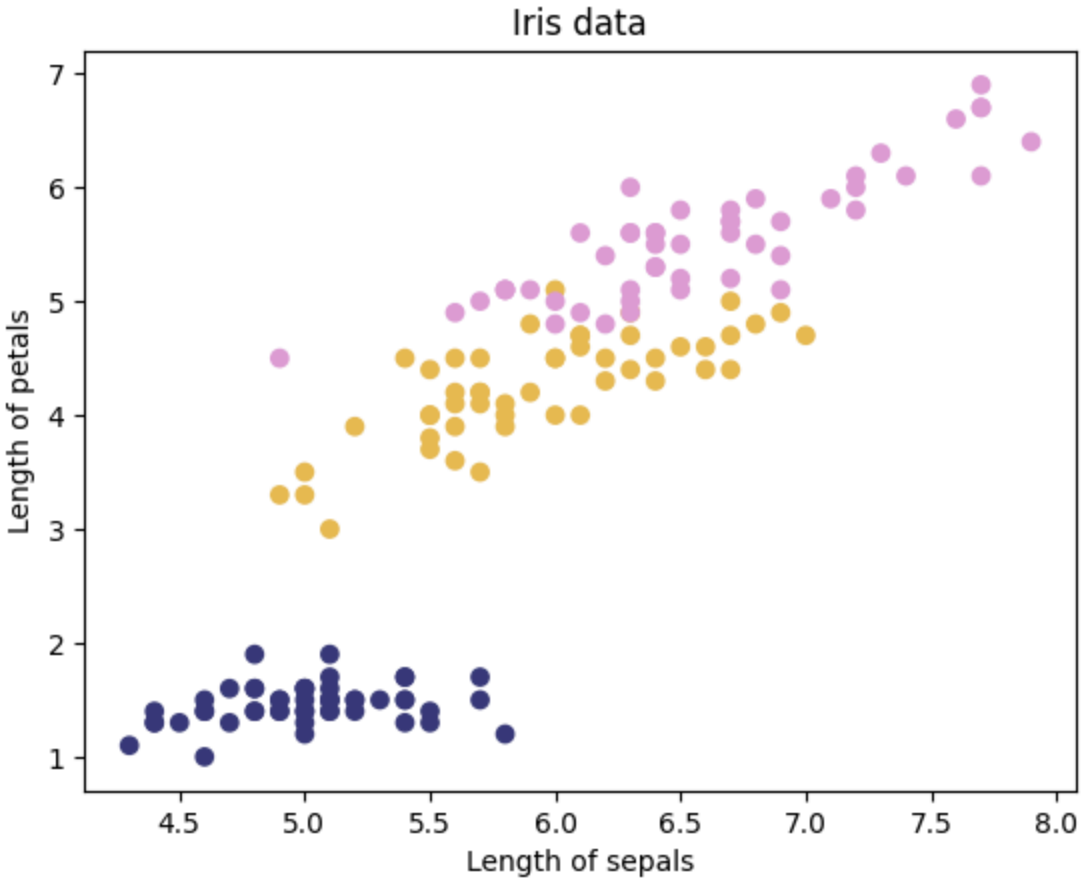
```
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt
import numpy as np
#from sklearn.cluster import ___  # Fill in the appropriate class
from sklearn.cluster import DBSCAN

# Create circles dataset
X, y = make_circles(n_samples=2000, noise=0.1, factor=0.2)

# Train DBSCAN model on circles dataset
#clustering = ___(eps=0.1, min_samples=5).___(X)  # Fill in the DBSCAN class
clustering = DBSCAN(eps=0.1, min_samples=5).fit(X)

# Provide visualization
fig, axes = plt.subplots(1, 3)
fig.set_size_inches(10, 5)

# Plot original data with noise
axes[0].scatter(X[:, 0], X[:, 1], c='brown')
axes[0].set_title('Data with Noise')

# Plot clusters with DBSCAN, including noise points
#axes[1].scatter(X[:, 0], X[:, 1], c=clustering.___, cmap='tab20b')  # Fill
axes[1].scatter(X[:, 0], X[:, 1], c=clustering.labels_, cmap='tab20b')  # Fi
axes[1].set_title('Clusters with DBSCAN + Noise')

# Detect noise points and remove them from the dataset
#cleaned_X = np.delete(X, np.where(clustering.labels_ == ___), axis=0).resha
cleaned_X = np.delete(X, np.where(clustering.labels_ == -1), axis=0).reshape
, axis=0).reshape(-1, 2)  # Fill in the appropriate condition
# Provide visualization of dataset without outliers
axes[2].scatter(cleaned_X[:, 0], cleaned_X[:, 1], c='brown')
axes[2].set_title('Cleaned Data')

plt.tight_layout()
plt.show()
```



We have considered 4 clustering algorithms and looked at the principles of their work on toy datasets. Now let's try to use these clustering methods for solving the real-life problem with real data.

We will use the Iris dataset which consists of 50 samples from each of three species of Iris (Iris setosa, Iris virginica, and Iris versicolor); four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. The task is to determine the type of Iris using these features: we will provide clustering and assume that each cluster represents one of the Iris species. To provide understandable visualizations we will use only two features for clustering: the length of sepals and the length of petals. Let's look at our data:##

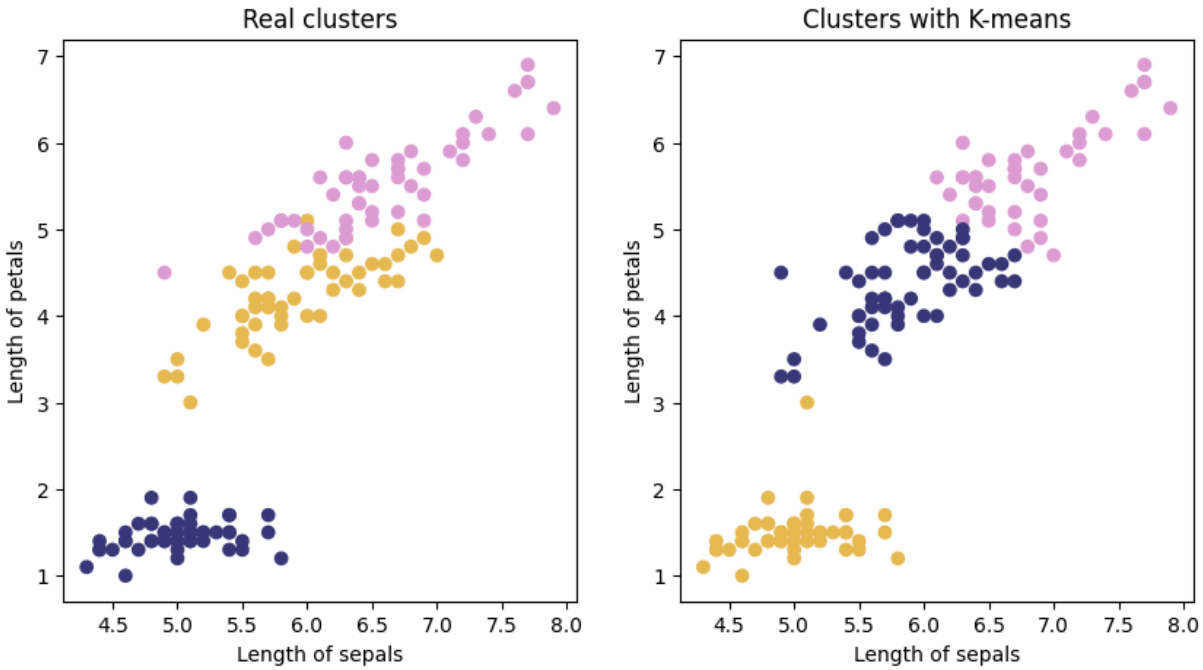
In [42]:
```
from sklearn.datasets import load_iris
import numpy as np
import matplotlib.pyplot as plt

X_iris, y_iris = load_iris(return_X_y=True)
X_iris = X_iris[:, [0,2]]
plt.scatter(X_iris[:, 0], X_iris[:, 1], c=y_iris, cmap='tab20b')
plt.title('Iris data')
plt.xlabel('Length of sepals')
plt.ylabel('Length of petals')
plt.show()
```



Let's use K-means to provide clustering and compare results with real data:##

In [43]:
```
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import numpy as np
import matplotlib.pyplot as plt
import warnings

warnings.filterwarnings('ignore')

X_iris, y_iris = load_iris(return_X_y=True)
X_iris = X_iris[:, [0,2]]
kmeans = KMeans(n_clusters=3).fit(X_iris)
fig, axes = plt.subplots(1, 2)
fig.set_size_inches(10, 5)
axes[0].scatter(X_iris[:, 0], X_iris[:, 1], c=y_iris, cmap='tab20b')
axes[0].set_title('Real clusters')
axes[1].scatter(X_iris[:, 0], X_iris[:, 1], c = kmeans.labels_, cmap='tab20b
axes[1].set_title('Clusters with K-means')
plt.setp(axes[0], xlabel='Length of sepals')
plt.setp(axes[0], ylabel='Length of petals')
plt.setp(axes[1], xlabel='Length of sepals')
plt.setp(axes[1], ylabel='Length of petals')
```

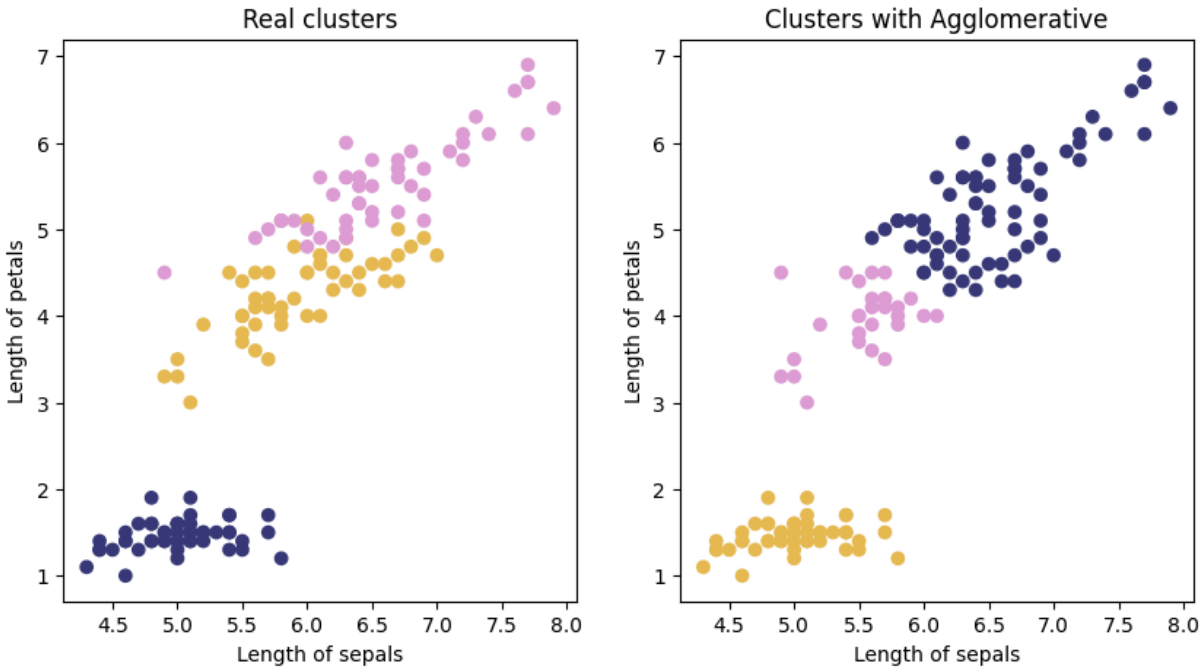
Out[43]:
```
[Text(0, 0.5, 'Length of petals')]
```



Now let's try agglomerative algorithm:##

In [44]:
```
from sklearn.datasets import load_iris
from sklearn.cluster import AgglomerativeClustering
import numpy as np
import matplotlib.pyplot as plt
X_iris, y_iris = load_iris(return_X_y=True)
X_iris = X_iris[:, [0,2]]
agglomerative = AgglomerativeClustering(n_clusters = 3).fit(X_iris)
fig, axes = plt.subplots(1, 2)
fig.set_size_inches(10, 5)
axes[0].scatter(X_iris[:, 0], X_iris[:, 1], c=y_iris, cmap='tab20b')
axes[0].set_title('Real clusters')
axes[1].scatter(X_iris[:, 0], X_iris[:, 1], c = agglomerative.labels_, cmap=
axes[1].set_title('Clusters with Agglomerative')
plt.setp(axes[0], xlabel='Length of sepals')
plt.setp(axes[0], ylabel='Length of petals')
plt.setp(axes[1], xlabel='Length of sepals')
plt.setp(axes[1], ylabel='Length of petals')
```

Out[44]:
```
[Text(0, 0.5, 'Length of petals')]
```

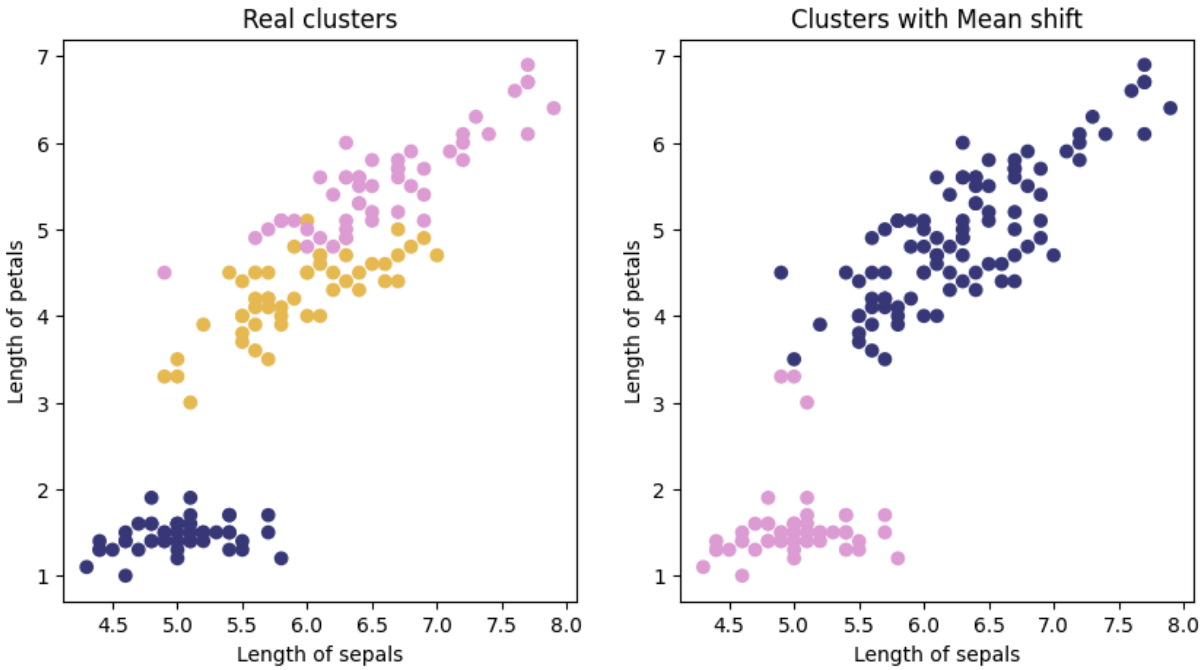


Note

We mentioned in the Agglomerative Clustering chapter that we can manually define the number of clusters. Here we used this ability because we have information about the number of target clusters (3 species of Iris = 3 clusters). Using Mean shift clustering algorithm:##

In [45]:
```
from sklearn.datasets import load_iris
from sklearn.cluster import MeanShift
import numpy as np
import matplotlib.pyplot as plt
X_iris, y_iris = load_iris(return_X_y=True)
X_iris = X_iris[:, [0,2]]
mean_shift= MeanShift(bandwidth=2).fit(X_iris)
fig, axes = plt.subplots(1, 2)
fig.set_size_inches(10, 5)
axes[0].scatter(X_iris[:, 0], X_iris[:, 1], c=y_iris, cmap='tab20b')
axes[0].set_title('Real clusters')
axes[1].scatter(X_iris[:, 0], X_iris[:, 1], c = mean_shift.labels_, cmap='ta
axes[1].set_title('Clusters with Mean shift')
plt.setp(axes[0], xlabel='Length of sepals')
plt.setp(axes[0], ylabel='Length of petals')
plt.setp(axes[1], xlabel='Length of sepals')
plt.setp(axes[1], ylabel='Length of petals')
```

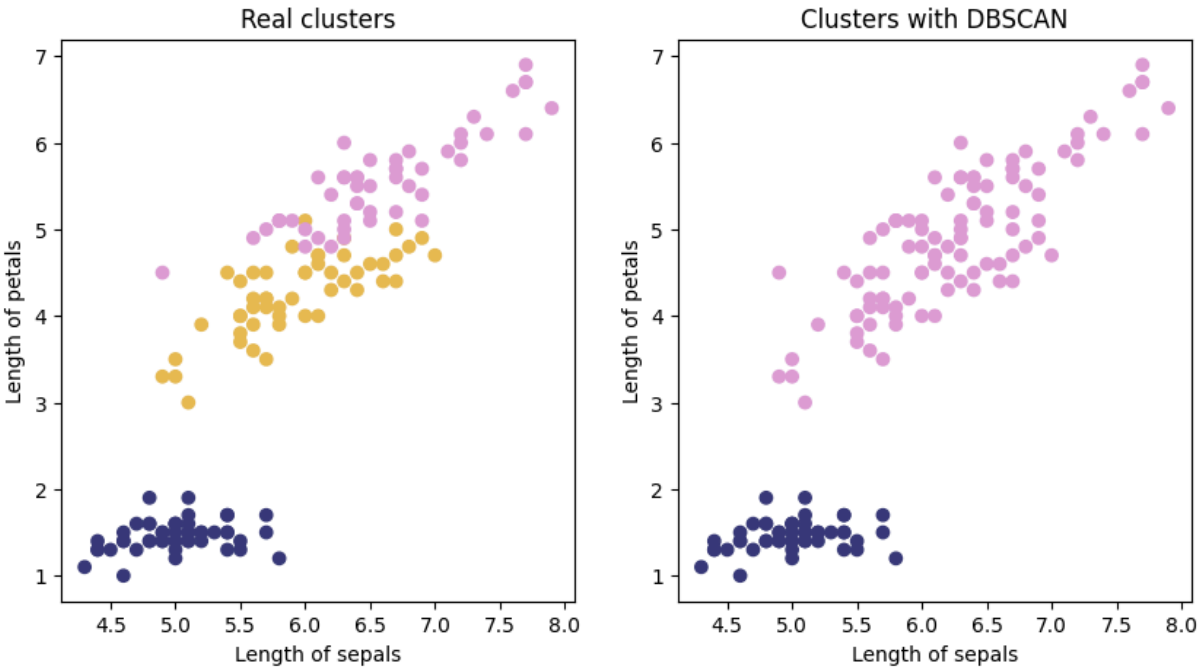
Out[45]:
```
[Text(0, 0.5, 'Length of petals')]
```



Finally, let's try to use DBSCAN:##

In [46]:
```
from sklearn.datasets import load_iris
from sklearn.cluster import DBSCAN
import numpy as np
import matplotlib.pyplot as plt
X_iris, y_iris = load_iris(return_X_y=True)
X_iris = X_iris[:, [0,2]]
dbscan = DBSCAN(eps=1, min_samples=10).fit(X_iris)
fig, axes = plt.subplots(1, 2)
fig.set_size_inches(10, 5)
axes[0].scatter(X_iris[:, 0], X_iris[:, 1], c=y_iris, cmap='tab20b')
axes[0].set_title('Real clusters')
axes[1].scatter(X_iris[:, 0], X_iris[:, 1], c = dbscan.labels_, cmap='tab20b
axes[1].set_title('Clusters with DBSCAN')
plt.setp(axes[0], xlabel='Length of sepals')
plt.setp(axes[0], ylabel='Length of petals')
plt.setp(axes[1], xlabel='Length of sepals')
plt.setp(axes[1], ylabel='Length of petals')
```

Out[46]:
```
[Text(0, 0.5, 'Length of petals')]
```



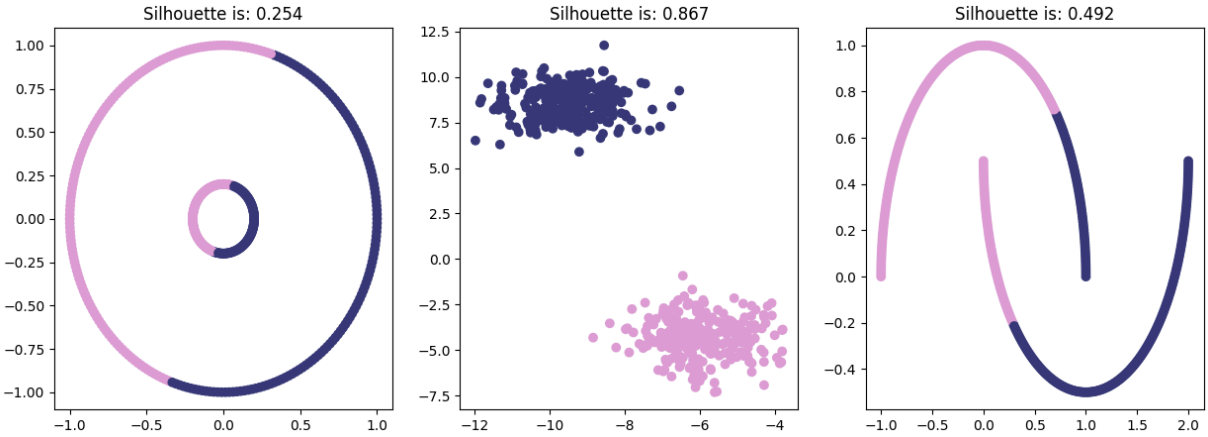
Note

In the code above, we manually defined the parameters for the algorithms (eps, min_samples for DBSCAN, and bandwidth for Mean shift). In real tasks, to determine optimal values of these parameters, it is necessary to use additional techniques (cross-validation, grid search, etc.). We can see that due to visualizations K-means and Agglomerative algorithms can solve the task. At the same time, Mean shift and DBSCAN can't extract 3 different clusters. Thus, we can conclude that for each individual task, an individual approach is needed: the choice of an algorithm, the selection of parameters, etc. In addition, it is necessary to set certain metrics with which we can evaluate the quality of clustering. The use of plots of clusters is not the best indicator for two reasons: Plots will not be able to adequately show the distribution into clusters for multivariate data(data with more than 3 features can't be visualized properly); Plots can show algorithms that give very poor results (like DBSCAN and Mean shift in the example above). But if the results are good, then it is very difficult to understand where the clustering quality is better(like K-means and Agglomerative in the example above). We will talk about evaluating the quality of clustering in the next section.##

As in any machine learning task, we rely on specific metrics to evaluate the quality of clustering: one of the classes of such metrics are internal metrics. Internal metrics in clustering are used to evaluate the quality of clustering results based on the data without using any external information or labels. These metrics provide a quantitative measure of how well a clustering algorithm has grouped the data points into clusters based on the intrinsic characteristics of the data: especially intra- and inter-cluster distances. Most commonly used internal metrics Silhouette score measures how well a data point fits into

its assigned cluster compared to other clusters. The silhouette score is calculated as follows: Step 1. For each data point, calculate two metrics: a: The average distance between the data point and all other points in the same cluster; b: The average distance between the data point and all points in the nearest cluster (i.e., the cluster that is most similar to the data point). Step 2. Calculate the silhouette score for each data point using the following formula: silhouette score = (b - a) / max(a, b) Step 3. Calculate the overall silhouette score for the clustering by taking the average of all the scores for all points.

In [47]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import silhouette_score
from sklearn.cluster import KMeans
from sklearn.datasets import make_moons, make_blobs, make_circles
import warnings

warnings.filterwarnings('ignore')

# Create subplots for visualizations
fig, axes = plt.subplots(1, 3)
fig.set_size_inches(15, 5)  # Adjusted figure size

# Load circles dataset
X_circles, y = make_circles(n_samples=500, factor=0.2)

# Provide K-means clustering for circles dataset
clustering_circles = KMeans(n_clusters=2).fit(X_circles)

# Provide visualization and show silhouette for circles dataset
axes[0].scatter(X_circles[:, 0], X_circles[:, 1], c=clustering_circles.label
axes[0].set_title('Silhouette is: ' + str(round(silhouette_score(X_circles,

# Load blobs dataset
X_blobs, y = make_blobs(n_samples=500, centers=2)

# Provide K-means clustering for blobs dataset
clustering_blobs = KMeans(n_clusters=2).fit(X_blobs)

# Provide visualization and show silhouette for blobs dataset
axes[1].scatter(X_blobs[:, 0], X_blobs[:, 1], c=clustering_blobs.labels_, cm
axes[1].set_title('Silhouette is: ' + str(round(silhouette_score(X_blobs, cl

# Load moons dataset
X_moons, y = make_moons(n_samples=500)

# Provide K-means clustering for moons dataset
clustering_moons = KMeans(n_clusters=2).fit(X_moons)

# Provide visualization and show silhouette for moons dataset
axes[2].scatter(X_moons[:, 0], X_moons[:, 1], c=clustering_moons.labels_, cm
axes[2].set_title('Silhouette is: ' + str(round(silhouette_score(X_moons, cl

# Display the plots
plt.show()
```



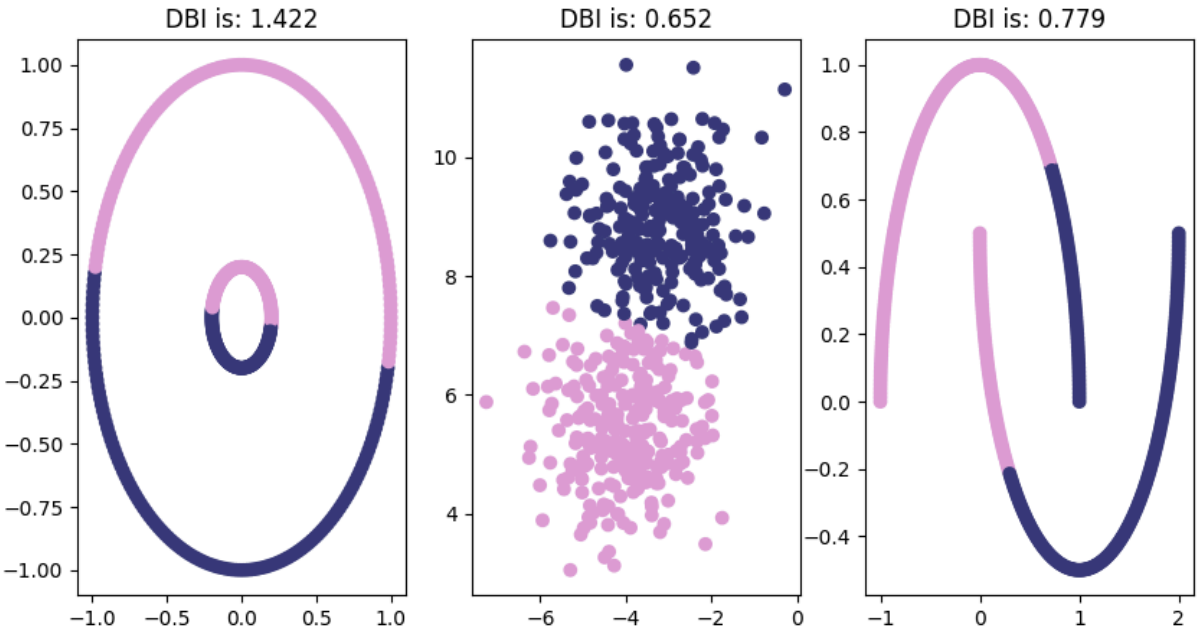
The silhouette coefficient varies between -1 to 1, with:

-1 indicating that the data point isn't assigned to the right cluster; 0 indicating that the clusters are overlapping; 1 indicates that the cluster is dense and well-separated (thus the desirable value). The Davies-Bouldin Index (DBI) is an internal clustering evaluation metric that measures the quality of clustering by considering both the separation between clusters and the compactness of clusters. The DBI is calculated as follows: Step 1. For each cluster, calculate the average distance between its centroid and all the data points in that cluster. This measures the cluster's similarity; Step 2. For each pair of clusters, calculate the distance between their centroids. This measures the dissimilarity between clusters; Step 3. For each cluster, find the cluster with the greatest similarity to it (excluding itself); Step 4. For each cluster, calculate the sum of the similarity between that cluster and its closest neighbor, and divide it by the number of data points in the cluster. This gives you the DBI score for that cluster; Step 5. Finally, calculate the overall DBI score by taking the average DBI scores for all the clusters.##

In [48]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import davies_bouldin_score
from sklearn.cluster import KMeans
from sklearn.datasets import make_moons, make_blobs, make_circles
import warnings

warnings.filterwarnings('ignore')

fig, axes = plt.subplots(1, 3)
fig.set_size_inches(10, 5)

X_circles, y = make_circles(n_samples=500, factor=0.2)
clustering = KMeans(n_clusters=2).fit(X_circles)
# Provide visualization and show DBI for circles dataset
axes[0].scatter(X_circles[:, 0], X_circles[:, 1], c=clustering.labels_, cmap
axes[0].set_title('DBI is: '+ str(round(davies_bouldin_score(X_circles, clus

X_blobs, y = make_blobs(n_samples=500, centers=2)
clustering = KMeans(n_clusters=2).fit(X_blobs)
# Provide visualization and show DBI for blobs dataset
```

```
axes[1].scatter(X_blobs[:, 0], X_blobs[:, 1], c=clustering.labels_, cmap='ta
axes[1].set_title('DBI is: '+ str(round(davies_bouldin_score(X_blobs, cluste

X_moons, y = make_moons(n_samples=500)
clustering = KMeans(n_clusters=2).fit(X_moons)
# Provide visualization and show DBI for moons dataset
axes[2].scatter(X_moons[:, 0], X_moons[:, 1], c=clustering.labels_, cmap='ta
axes[2].set_title('DBI is: '+ str(round(davies_bouldin_score(X_moons, cluste
```

Out[48]: Text(0.5, 1.0, 'DBI is: 0.779')



A lower DBI value indicates better clustering performance, indicating that the clusters are well-separated and compact. For which of the metrics a zero value is a sign of good clustering quality? Select the correct answer

For DBI For silhouette score both DBI and silhouette score

Correct Answer: For DBI

Explanation:

Davis-Bouldin Index (DBI): A lower DBI value (closer to 0) indicates better clustering quality. DBI measures the average similarity ratio of intra-cluster and inter-cluster distances. Smaller values indicate more compact clusters that are well-separated from each other. Zero DBI is theoretically ideal but rarely achieved in practice.

Silhouette Score: A higher silhouette score (closer to 1) indicates better clustering quality. Silhouette score measures how similar data points are within their clusters compared to other clusters. A silhouette score of 0 suggests that the data points lie exactly on the boundary between two clusters, which is undesirable and indicates poor clustering.

Final Note: DBI: Zero is a good sign. Silhouette Score: Zero indicates poor clustering. Thus, the correct answer is DBI.

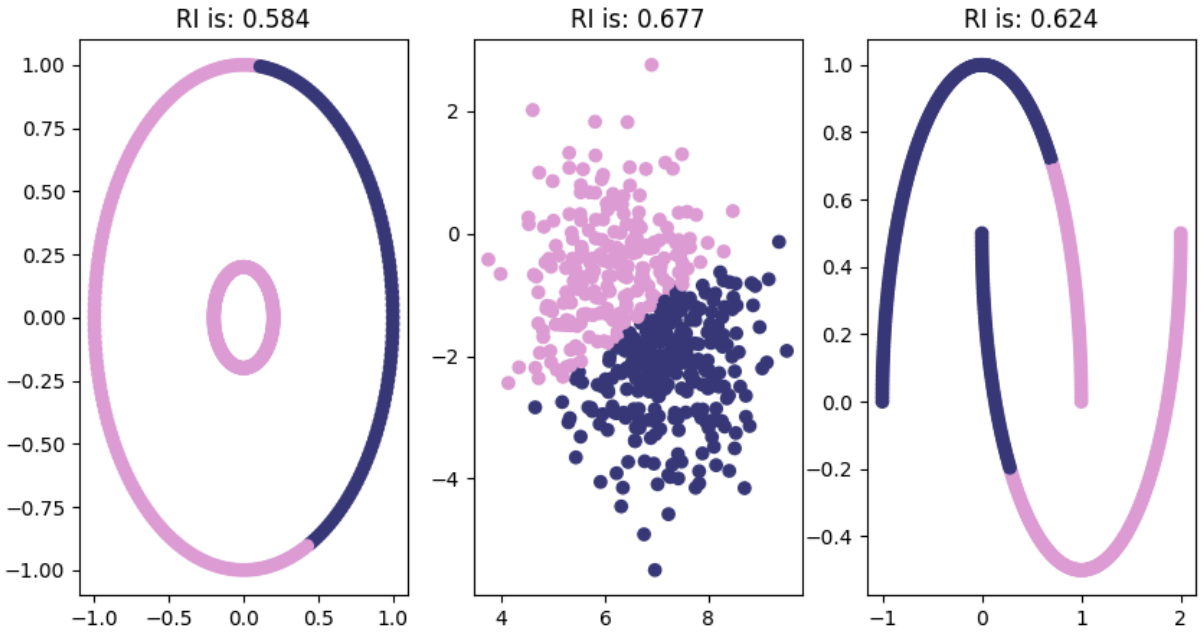
External evaluation for clustering algorithms is a method of evaluating the performance of a clustering algorithm by comparing its results to a known set of class labels or ground truth. In other words, the algorithm's clusters are compared to a set of pre-existing labels created by experts or based on domain knowledge.

Most commonly used external metrics The Rand Index (RI) measures the similarity between two clusterings or partitions and is often used as an external evaluation metric in clustering. The Rand Index measures the percentage of pairs of data points assigned to the same cluster in both the predicted and true clusterings, normalized by the total number of data point pairs. The Rand Index is calculated as follows: Let n be the total number of data points; Let a be the number of pairs of data points assigned to the same cluster in both the predicted and true clusterings; Let b be the number of pairs of data points assigned to different clusters in both the predicted and true clustering. The Rand Index is then given by 2*(a+b)/ (n*(n-1)).##

In [49]:
```
from sklearn.metrics import rand_score
from sklearn.cluster import KMeans
from sklearn.datasets import make_moons, make_blobs, make_circles
import matplotlib.pyplot as plt
import warnings

warnings.filterwarnings('ignore')

# Creating subplots for visualization
fig, axes = plt.subplots(1, 3)
fig.set_size_inches(10, 5)
# Create circles dataset
X_circles, y = make_circles(n_samples=500, factor=0.2)
# Provide K-means clustering
clustering = KMeans(n_clusters=2).fit(X_circles)
predicted_circles = clustering.predict(X_circles)
# Provide visualization and show RI for circles dataset
axes[0].scatter(X_circles[:, 0], X_circles[:, 1], c=clustering.labels_, cmap
axes[0].set_title('RI is: '+ str(round(rand_score(y, predicted_circles), 3))

X_blobs, y = make_blobs(n_samples=500, centers=2)
clustering = KMeans(n_clusters=2).fit(X_blobs)
predicted_blobs = clustering.predict(X_blobs)
# Provide visualization and show RI for blobs dataset
axes[1].scatter(X_blobs[:, 0], X_blobs[:, 1], c=clustering.labels_, cmap='ta
axes[1].set_title('RI is: '+ str(round(rand_score(y, predicted_blobs), 3)))
```

```
X_moons, y = make_moons(n_samples=500)
clustering = KMeans(n_clusters=2).fit(X_moons)
predicted_moons = clustering.predict(X_moons)
# Provide visualization and show RI for moons dataset
axes[2].scatter(X_moons[:, 0], X_moons[:, 1], c=clustering.labels_, cmap='ta
axes[2].set_title('RI is: '+ str(round(rand_score(y, predicted_moons), 3)))
```

Out[49]: Text(0.5, 1.0, 'RI is: 0.624')



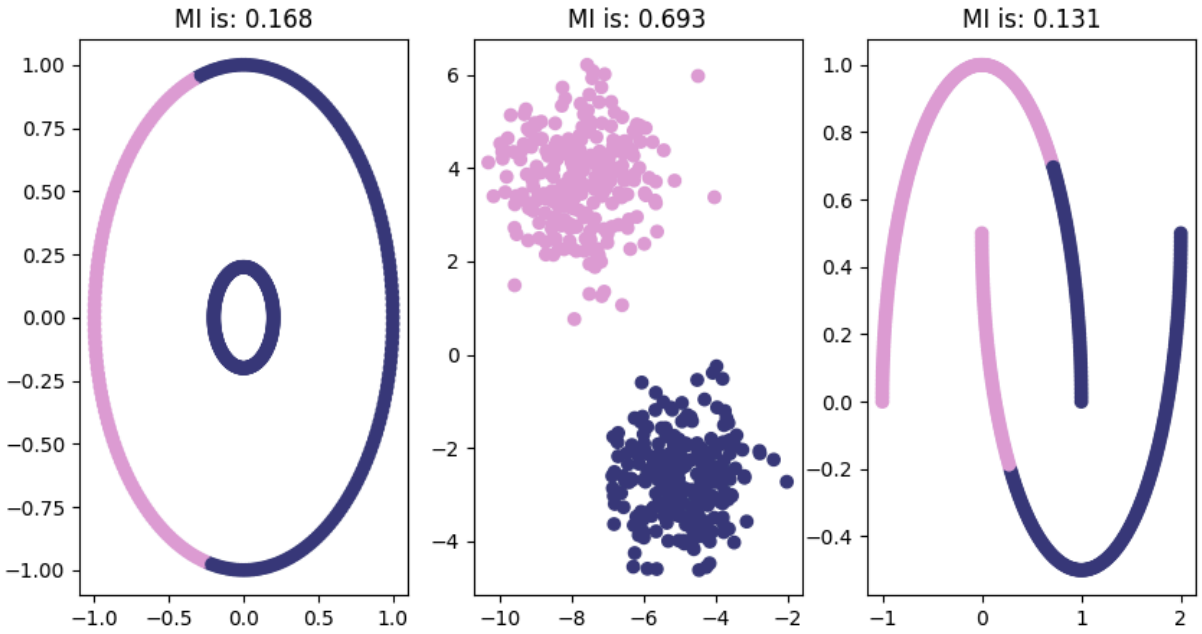
The Rand Index can vary between 0 and 1, where 0 indicates that the two clusterings are completely different, and 1 indicates that the two clusterings are identical.

Mutual Information (MI) measures the amount of information shared by the predicted and true clusterings based on the concept of entropy. We will not consider how this metric is calculated, as this is outside the scope of the beginner-level course.##

In [50]:
```
from sklearn.metrics import mutual_info_score
from sklearn.cluster import KMeans
from sklearn.datasets import make_moons, make_blobs, make_circles
import matplotlib.pyplot as plt
import warnings

warnings.filterwarnings('ignore')

# Create subplots for visualizations
fig, axes = plt.subplots(1, 3)
fig.set_size_inches(10 ,5)

X_circles, y = make_circles(n_samples=500, factor=0.2)
clustering = KMeans(n_clusters=2).fit(X_circles)
```

```
predicted_circles = clustering.predict(X_circles)
# Provide visualization and show MI for circles dataset
axes[0].scatter(X_circles[:, 0], X_circles[:, 1], c=clustering.labels_, cmap
axes[0].set_title('MI is: '+ str(round(mutual_info_score(y, predicted_circle

X_blobs, y = make_blobs(n_samples=500, centers=2)
clustering = KMeans(n_clusters=2).fit(X_blobs)
predicted_blobs = clustering.predict(X_blobs)
# Provide visualization and show MI for blobs dataset
axes[1].scatter(X_blobs[:, 0], X_blobs[:, 1], c=clustering.labels_, cmap='ta
axes[1].set_title('MI is: '+ str(round(mutual_info_score(y, predicted_blobs)

X_moons, y = make_moons(n_samples=500)
clustering = KMeans(n_clusters=2).fit(X_moons)
predicted_moons = clustering.predict(X_moons)
# Provide visualization and show MI for moons dataset
axes[2].scatter(X_moons[:, 0], X_moons[:, 1], c=clustering.labels_, cmap='ta
axes[2].set_title('MI is: '+ str(round(mutual_info_score(y, predicted_moons)
```

Out[50]: Text(0.5, 1.0, 'MI is: 0.131')



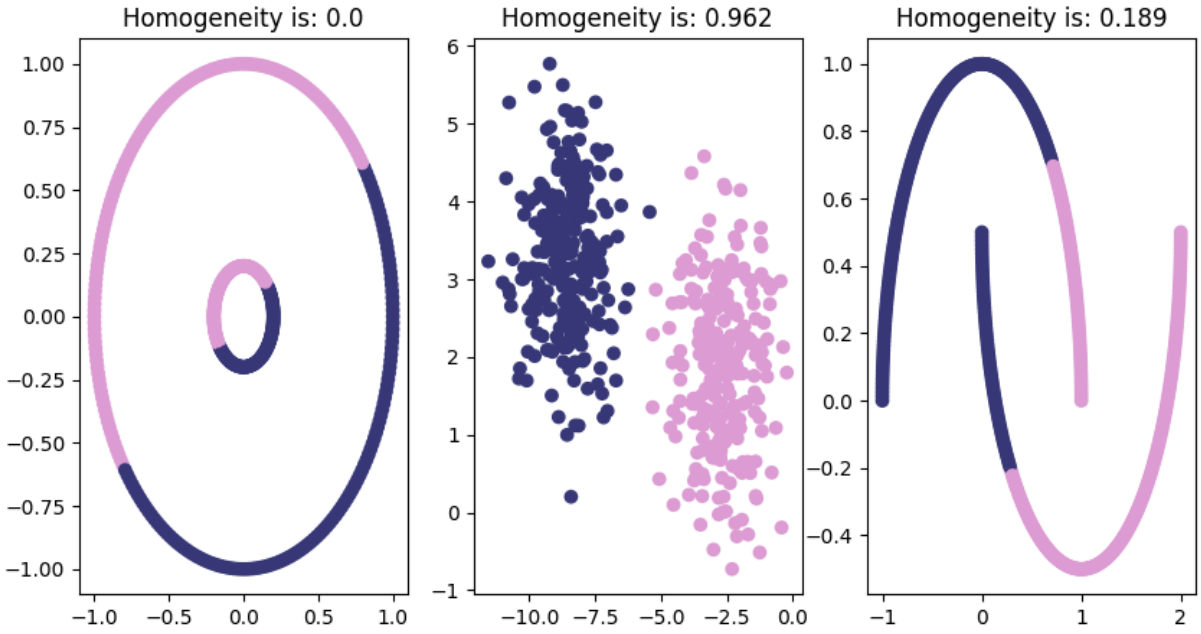
The Mutual Information varies between 0 and 1, where 0 indicates that the predicted clustering is completely different from the true clustering, and 1 indicates that the predicted clustering is identical to the true clustering. Furthermore, based on the above examples, we can say that this metric is

much better at detecting bad clustering than the Rand Index.

Homogeneity measures the degree to which each cluster contains only data points that belong to a single class or category based on conditional entropy. Just like with mutual information, we will not consider the principle of calculating this metric.##

In [51]:
```
from sklearn.metrics import homogeneity_score
from sklearn.cluster import KMeans
from sklearn.datasets import make_moons, make_blobs, make_circles
import matplotlib.pyplot as plt
import warnings

warnings.filterwarnings('ignore')

fig, axes = plt.subplots(1, 3)
fig.set_size_inches(10, 5)

X_circles, y = make_circles(n_samples=500, factor=0.2)
clustering = KMeans(n_clusters=2).fit(X_circles)
predicted_circles = clustering.predict(X_circles)
# Provide visualization and show homogeneity for circles dataset
axes[0].scatter(X_circles[:, 0], X_circles[:, 1], c=clustering.labels_, cmap
axes[0].set_title('Homogeneity is: '+ str(round(homogeneity_score(y, predict

X_blobs, y = make_blobs(n_samples=500, centers=2)
clustering = KMeans(n_clusters=2).fit(X_blobs)
predicted_blobs = clustering.predict(X_blobs)
# Provide visualization and show homogeneity for blobs dataset
axes[1].scatter(X_blobs[:, 0], X_blobs[:, 1], c=clustering.labels_, cmap='ta
axes[1].set_title('Homogeneity is: '+ str(round(homogeneity_score(y, predict

X_moons, y = make_moons(n_samples=500)
clustering = KMeans(n_clusters=2).fit(X_moons)
predicted_moons = clustering.predict(X_moons)
# Provide visualization and show homogeneity for moons dataset
axes[2].scatter(X_moons[:, 0], X_moons[:, 1], c=clustering.labels_, cmap='ta
axes[2].set_title('Homogeneity is: '+ str(round(homogeneity_score(y, predict
```

Out[51]:
```
Text(0.5, 1.0, 'Homogeneity is: 0.189')
```



A clustering solution is considered highly homogeneous if all the data points that belong to the same true class or category are grouped into the same cluster.

In other words, homogeneity measures the extent to which a clustering algorithm assigns data points to the correct clusters based on their true class or category. The homogeneity score ranges from 0 to 1, with 1 indicating perfect homogeneity. Homogeneity is the best of all the considered metrics: it determines both good and bad clustering equally well, as shown in the example above.##

Can we use external evaluation metrics if we have no information about real partitioning of data into clusters? Select the correct answer

Yes, we can

No, we need to know real labels to use external evaluation

Correct Answer: No, we need to know real labels to use external evaluation

Explanation: External Evaluation Metrics: These metrics (e.g., Adjusted Rand Index, Normalized Mutual Information, Fowlkes-Mallows Index) compare the clustering result to the true labels (ground truth) of the dataset. If real labels (true partitioning) are unavailable, external evaluation cannot be performed. Internal Evaluation Metrics: Metrics like Silhouette Score, Davies-Bouldin Index, and Calinski-Harabasz Index do not require true labels and evaluate clustering quality based on intrinsic properties of the dataset, such as compactness and separation. Final Note: If you do not have access to

real labels, you must rely on internal metrics to assess clustering quality. External metrics explicitly require the ground truth to compare against.

In real-life tasks with real data, it can be difficult to understand which algorithm to use and whether the results are good enough. To determine this, several techniques are used:

Relative cluster validation, which evaluates the clustering structure by varying different parameter values for the same algorithm (e.g.,: varying the number of clusters k for K-means, linkage for agglomerative, eps and min_samples for DBSCAN etc.); Internal and external cluster validation means that we use internal and external metrics to estimate clustering quality; Rule of thumb: a stable group should be preserved when changing the clustering method. For example, if the results obtained using the agglomerative method and the K-means method coincide by more than 70%, then the assumption of stability is accepted; Using resampling methods to evaluate the stability of clustering split: whether the split is stable across different subsamples of the original dataset; whether the split is stable after some samples were deleted from original the dataset; whether the split is stable after changing the order of the elements. Try to interpret the clustering results in terms of the domain area: is it possible to explain the results of clustering and is there any logic in them. Note In the context of data analysis, the domain area refers to the specific field or industry that the data belongs to or is being used for. Examples of domain areas include healthcare, finance, marketing, transportation, and many others.

Can we consider the results of clustering to be stable if different algorithms produce completely different clusters? Select the correct answer

Yes, we can.

No, stable partitioning into clusters implies the similarity of results when using different algorithms.

Correct Answer: No, stable partitioning into clusters implies the similarity of results when using different algorithms.

Explanation: Stability of Clustering: Clustering is considered stable if different algorithms produce similar results on the same dataset, given appropriate parameter tuning. Stable results indicate that the underlying structure of the data is well-defined and not highly sensitive to the choice of the algorithm. Different Clusters with Different Algorithms: If different algorithms produce completely different clusters, it suggests that: The data has no clear structure or clustering is ambiguous. The clustering algorithms are being

applied with incompatible assumptions or parameters. Interpretation: Stable partitioning requires the similarity of results across multiple clustering approaches, assuming the algorithms are well-suited to the dataset and their parameters are optimized. Final Note: Consistency across algorithms (e.g., K-means, DBSCAN, Hierarchical Clustering) is an important indicator of clustering stability. Completely different clusters indicate instability or poorly defined clustering structure.

In [1]:
```
!pwd
```

```
/Users/obaozai/Data/GitHub/Inferetntial/Cluster
```

In [ ]:
```
```