Silhouette, Hierarchical, DBScan Clustering Notes

Silhouette Evaluation Metric

The **Silhouette Score** is a metric used to evaluate the quality of clustering algorithms. It measures how similar an object is to its own cluster compared to other clusters, providing insight into the separation between clusters.

The score is defined for each sample and ranges between `−1` and `1`:

- A **value close to 1** indicates that the sample is well-clustered (i.e., it is close to the center of its cluster and far from other clusters).
- A **value close to 0** indicates that the sample lies on or near the decision boundary between two clusters.
- A **value close to -1** indicates that the sample may have been assigned to the wrong cluster.

Silhouette Score Formula

For each point (i):

1. **Compute the average intra-cluster distance** (a(i)): the average distance between the point and all other points in the same cluster.
2. **Compute the average nearest-cluster distance** (b(i)): the average distance between the point and points in the nearest cluster (i.e., the next best cluster to which the point might belong).

The **silhouette score** for point (i) is given by:

$$S(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

Interpretation:

- (S(i)) close to 1 means the point is well separated from neighboring clusters.
- (S(i)) close to 0 means the point is on or near the boundary between two clusters.
- (S(i)) close to -1 means the point might be misclassified to the wrong cluster.

The overall **silhouette score** for a dataset is the average of the silhouette scores of all points.

Example:

Imagine we have clustered data into two groups using the **K-Means algorithm**:

- **Cluster 1** contains points `[1, 2, 3, 4, 5]`.
- **Cluster 2** contains points `[10, 11, 12, 13, 14]`.

1. **For a point in Cluster 1** (e.g., point `3`):
    - Compute `a(i)` = average distance between `3` and other points in Cluster 1.
    - Compute `b(i)` = average distance between `3` and points in Cluster 2.
    - Substitute these values into the silhouette formula to get the silhouette score for that point.
2. **For a point in Cluster 2** (e.g., point `12`):
    - Similarly, compute `a(i)` (distance to Cluster 2) and `b(i)` (distance to Cluster 1), and calculate the silhouette score.

Application in Python (Code Example):

Using `scikit-learn` to compute the Silhouette Score:

```
from sklearn.metrics import silhouette_score
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Generate synthetic data
X, _ = make_blobs(n_samples=300, centers=4, cluster_std=0.60,
random_state=0)

# Apply KMeans clustering
kmeans = KMeans(n_clusters=4)
labels = kmeans.fit_predict(X)

# Compute the silhouette score
score = silhouette_score(X, labels)
print(f"Silhouette Score: {score}")
```

Pros of Silhouette Score:

- Easy to interpret.
- Provides insights into both cohesion and separation of clusters.
- Works well for any type of clustering (e.g., K-Means, hierarchical, etc.).

Cons of Silhouette Score:

- Not ideal for **clusters with varying densities** or **non-spherical clusters**.
- Can be computationally expensive for large datasets.

Summary:

The Silhouette Score is a powerful metric to evaluate clustering results by examining how well each point is classified relative to its own cluster and other clusters. It is

widely used in machine learning, particularly in unsupervised learning, to assess the effectiveness of a clustering algorithm.

Hierarchical Clustering

Hierarchical Clustering is a method of cluster analysis that seeks to build a hierarchy of clusters. Unlike flat clustering methods such as **K-Means**, hierarchical clustering does not require the user to pre-define the number of clusters. Instead, it creates a tree-like structure of clusters called a **dendrogram**.

Hierarchical clustering comes in two types:

1. **Agglomerative Clustering** (Bottom-Up approach):
    - Start with each point as its own cluster.
    - Merge the closest pair of clusters step by step until all points belong to a single cluster.
2. **Divisive Clustering** (Top-Down approach):
    - Start with all points in one cluster.
    - Recursively split the most heterogeneous clusters until each point is in its own cluster.

The most common approach is **agglomerative hierarchical clustering**.

Steps of Agglomerative Hierarchical Clustering:

1. **Start with individual clusters**: Treat each data point as a separate cluster (singleton clusters).
2. **Calculate proximity**: Compute the distance (or similarity) between each pair of clusters using a distance metric like **Euclidean distance** or **Manhattan distance**.
3. **Merge clusters**: Find the two closest clusters and merge them into a single cluster.
4. **Repeat**: Continue merging the two nearest clusters at each step until only one cluster remains, which includes all data points.
5. **Build a Dendrogram**: The result of hierarchical clustering can be visualized using a **dendrogram**, which is a tree-like diagram that shows how clusters are merged at each step.

Example:

Consider the following points in 2D space:

- Point A: (1, 2)

- Point B: (2, 3)
- Point C: (5, 6)
- Point D: (8, 8)

Step-by-Step Explanation:

1. **Initialize Clusters**:
    - A, B, C, and D are each their own cluster.
2. **Calculate Distances**: Compute pairwise distances between the clusters (using Euclidean distance):
    - Distance between A and B: $\sqrt{((2 - 1)^2 + (3 - 2)^2)} = \sqrt{2} \approx 1.41$
    - Distance between A and C: $\sqrt{((5 - 1)^2 + (6 - 2)^2)} = \sqrt{32} \approx 5.66$
    - Distance between B and C: $\sqrt{((5 - 2)^2 + (6 - 3)^2)} = \sqrt{18} \approx 4.24$
    - ... and so on.
3. **Merge Closest Clusters**: A and B are the closest clusters (distance ≈ 1.41), so merge them into a single cluster: **Cluster {A, B}**.
4. **Recompute Distances**: Compute the distances between the new cluster {A, B} and the remaining clusters:
    - Distance between {A, B} and C: Compute the average or single linkage distance, depending on the linkage criterion used.
    - Continue this process until all points are merged into a single cluster.
5. **Dendrogram**: As the merging progresses, a dendrogram is built, which shows at what distance each pair of clusters was merged. The height of the branches represents the distance at which clusters were merged.

Linkage Methods for Distance Calculation:

Several linkage criteria are used to decide how to calculate the distance between clusters during the merging process:

1. **Single Linkage**: Distance between two clusters is the minimum distance between points in the two clusters.
2. **Complete Linkage**: Distance between two clusters is the maximum distance between points in the two clusters.
3. **Average Linkage**: Distance between two clusters is the average distance between all pairs of points in the two clusters.
4. **Centroid Linkage**: Distance between two clusters is the distance between their centroids.

Code Example:

You can use **Scikit-learn**'s `AgglomerativeClustering` to perform hierarchical clustering.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.datasets import make_blobs

# Generate sample data
X, _ = make_blobs(n_samples=10, random_state=0)

# Perform hierarchical clustering
clustering = AgglomerativeClustering(n_clusters=2).fit(X)

# Plot the results
plt.scatter(X[:, 0], X[:, 1], c=clustering.labels_,
cmap='rainbow')
plt.show()

# Create a dendrogram
Z = linkage(X, 'ward')
dendrogram(Z)
plt.show()
```

Advantages of Hierarchical Clustering:

- **No need to specify the number of clusters** in advance.
- **Dendrogram provides a detailed view** of how clusters are formed and at what level they merge.
- Can be used for **any type of distance metric**, not just Euclidean.

Disadvantages of Hierarchical Clustering:

- **Computational complexity** is high, especially for large datasets.
- Once a decision is made to merge or split clusters, **it cannot be undone** (no global optimization).
- Sensitive to **noise and outliers**.

Stopping Criteria:

The algorithm can be stopped when a predefined number of clusters are obtained, or you can cut the dendrogram at a desired level to get the clusters.

Example Use Cases:

- **Hierarchical clustering** is widely used in **taxonomy**, **biology**, and **genomics** (e.g., organizing species or genes into hierarchical categories).
- It is useful for **market segmentation**, **document classification**, and other domains where the hierarchy of clusters is meaningful.

Summary:

Hierarchical clustering is a powerful tool for grouping data into a hierarchy of clusters. It differs from flat clustering algorithms in that it provides a tree-based view of how clusters are formed, which can be especially useful when the number of clusters is not known beforehand.

DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

DBSCAN is a clustering algorithm that groups together points that are closely packed (i.e., points with many nearby neighbors), and it marks points that lie alone in low-density regions (or outliers) as noise. Unlike K-Means, DBSCAN does not require the number of clusters to be specified a priori, making it advantageous for discovering clusters of arbitrary shapes and identifying noise.

How DBSCAN Works

DBSCAN is based on two parameters:

1. **Epsilon (ε)**: The maximum radius of the neighborhood around a point (radius of the circle around a point).
2. **MinPts**: The minimum number of points required to form a dense region (minimum number of points to be a core point).

Key Terms in DBSCAN:

- **Core Point**: A point is a core point if it has at least **MinPts** points (including itself) within a radius of ε (epsilon).
- **Border Point**: A border point is not a core point but lies within the neighborhood of a core point.
- **Noise (Outlier)**: A point that is neither a core point nor a border point.

DBSCAN Algorithm Steps:

1. **Start with an unvisited point**. Check if it is a core point by calculating the number of points within its ε-radius.
    - If it is a core point, a new cluster is created.
    - If it is not a core point, it is labeled as noise temporarily (this may change if it's found to be a border point later).
2. **Expand the cluster**. For a core point, recursively visit all points within the ε-radius, adding any core points and their neighbors to the cluster.
3. **Repeat the process** for each unvisited point until all points are either assigned to a cluster or labeled as noise.

DBSCAN Pseudocode:

```
For each point P in the dataset:
    If P is not visited:
        Mark P as visited
        Find all the points within ε distance from P (called
neighborhood of P)
        If the neighborhood has more than MinPts points:
            Start a new cluster and add all points from the
neighborhood to this cluster
            Iterate over the points in the neighborhood and for
each core point, add its neighbors to the cluster if they are not
already visited.
        Else:
            Mark P as noise.
```

DBSCAN Parameters:

1. **ε (Epsilon)**: This defines the maximum radius of the neighborhood around a point. A smaller value leads to more clusters being formed, while a larger value results in fewer clusters.

2. **MinPts (Minimum Points)**: This defines the minimum number of points required to form a dense region or a cluster. A smaller value may lead to more clusters and noise, while a larger value results in fewer but larger clusters.

Example:

Consider the following set of 2D points:

X	Y
1	1
2	2
3	3
8	8
8	9
25	30

With **ε = 2** and **MinPts = 2**, DBSCAN would:

- Identify points like (1, 1), (2, 2), and (3, 3) as a cluster, since they are close together within ε-distance and satisfy the MinPts condition.
- Identify (8, 8) and (8, 9) as another cluster.
- Mark (25, 30) as noise, since it has no points within the ε-distance.

DBSCAN Strengths:

1. **Does not require the number of clusters (k) to be defined**: Unlike K-Means, DBSCAN automatically detects the number of clusters based on the density of points.

2. **Can identify clusters of arbitrary shape**: It can find clusters that are not necessarily spherical.
3. **Resistant to noise**: DBSCAN effectively handles outliers by labeling them as noise.
4. **Efficient with large datasets**: If an efficient spatial data structure (like a KD-Tree) is used, DBSCAN can handle large datasets efficiently.

DBSCAN Weaknesses:

1. **Sensitive to parameter selection (ε and MinPts)**: The performance of DBSCAN depends heavily on the correct choice of ε and MinPts. Poor parameter choices can lead to erroneous clustering results.
2. **Difficulty with varying densities**: DBSCAN struggles with datasets where clusters have varying densities because a single global ε and MinPts might not apply well to all clusters.

DBSCAN Example Using Python (Scikit-Learn):

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_blobs

# Generate sample data
X, _ = make_blobs(n_samples=100, centers=3, cluster_std=0.5,
random_state=0)

# Apply DBSCAN
dbscan = DBSCAN(eps=0.3, min_samples=5)
labels = dbscan.fit_predict(X)

# Plot the clusters
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='rainbow',
marker='o')
plt.title("DBSCAN Clustering")
plt.show()
```

Distance Calculation in DBSCAN:

DBSCAN uses a distance metric (typically **Euclidean distance**) to determine the proximity of points to one another. This distance determines which points are considered to be within the ε-radius of a core point.

- **Euclidean Distance**:
  - Formula: `"√((x2 − x1)² + (y2 − y1)²)"`

Stopping Criteria:

The algorithm stops when all points have either been assigned to a cluster or classified as noise.

Use Cases:

- **Geographical data analysis**: Identifying clusters of nearby geographical points.
- **Anomaly detection**: Points that are not part of any cluster can be treated as outliers or anomalies.
- **Image processing**: Detecting regions of interest in images based on pixel intensity.

Summary:

DBSCAN is a powerful clustering algorithm when the number of clusters is unknown, and the clusters are of arbitrary shapes. It is well-suited for tasks involving noise and outliers, but it is sensitive to the parameters ε and MinPts, which must be chosen carefully for the algorithm to work effectively.

Import of required libraries

In [2]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
```

Records of the given dataset

unzipped data from 'Day_18/data.csv.zip' and use

In [3]:
```
data = pd.read_csv("../Day_18_1/data.csv", delimiter=',', encoding = "ISO
data.head(10)
```

Out[3]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerI
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	12/1/2010 8:26	2.55	17850.
1	536365	71053	WHITE METAL LANTERN	6	12/1/2010 8:26	3.39	17850.
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	12/1/2010 8:26	2.75	17850.
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	12/1/2010 8:26	3.39	17850.
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	12/1/2010 8:26	3.39	17850.
5	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	12/1/2010 8:26	7.65	17850.
6	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	12/1/2010 8:26	4.25	17850.
7	536366	22633	HAND WARMER UNION JACK	6	12/1/2010 8:28	1.85	17850.
8	536366	22632	HAND WARMER RED POLKA DOT	6	12/1/2010 8:28	1.85	17850.
9	536367	84879	ASSORTED COLOUR BIRD ORNAMENT	32	12/1/2010 8:34	1.69	13047.

Information of the columns in a data

In [4]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 541909 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   InvoiceNo    541909 non-null  object 
 1   StockCode    541909 non-null  object 
 2   Description  540455 non-null  object 
 3   Quantity     541909 non-null  int64  
 4   InvoiceDate  541909 non-null  object 
 5   UnitPrice    541909 non-null  float64
 6   CustomerID   406829 non-null  float64
 7   Country      541909 non-null  object 
dtypes: float64(2), int64(1), object(5)
memory usage: 33.1+ MB
```

Description of numeric features

In [5]:
```
data.describe().T
```

Out[5]:

	count	mean	std	min	25%	50%
Quantity	541909.0	9.552250	218.081158	-80995.00	1.00	3.00
UnitPrice	541909.0	4.611114	96.759853	-11062.06	1.25	2.08
CustomerID	406829.0	15287.690570	1713.600303	12346.00	13953.00	15152.00

Description of non numeric features

In [6]:
```
data.describe(exclude=['int64', 'float64']).T
```

Out[6]:

	count	unique	top	freq
InvoiceNo	541909	25900	573585	1114
StockCode	541909	4070	85123A	2313
Description	540455	4223	WHITE HANGING HEART T-LIGHT HOLDER	2369
InvoiceDate	541909	23260	10/31/2011 14:41	1114
Country	541909	38	United Kingdom	495478

Check for null values

In [7]:
```
data.isnull().sum()
```

Out[7]:
```
InvoiceNo          0
StockCode          0
Description     1454
Quantity           0
InvoiceDate        0
UnitPrice          0
CustomerID     135080
Country            0
dtype: int64
```

Shape of the available data

In [8]:
```
data.shape
```

Out[8]:
```
(541909, 8)
```

Delete all the missing records

In [9]:
```
data.dropna(inplace=True)
```

In [10]:
```
data.shape
```

Out[10]:
```
(406829, 8)
```

In [11]:
```
data.isnull().sum()
```

Out[11]:
```
InvoiceNo      0
StockCode      0
Description    0
Quantity       0
InvoiceDate    0
UnitPrice      0
CustomerID     0
Country        0
dtype: int64
```

Checking of whether negative values present in Quantity and Unit Price

In [12]:
```
data[data['Quantity']<0]
```

Out[12]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	Cus
141	C536379	D	Discount	-1	12/1/2010 9:41	27.50	
154	C536383	35004C	SET OF 3 COLOURED FLYING DUCKS	-1	12/1/2010 9:49	4.65	
235	C536391	22556	PLASTERS IN TIN CIRCUS PARADE	-12	12/1/2010 10:24	1.65	
236	C536391	21984	PACK OF 12 PINK PAISLEY TISSUES	-24	12/1/2010 10:24	0.29	
237	C536391	21983	PACK OF 12 BLUE PAISLEY TISSUES	-24	12/1/2010 10:24	0.29	
...	...	...	...	...	...	...	
540449	C581490	23144	ZINC T-LIGHT HOLDER STARS SMALL	-11	12/9/2011 9:57	0.83	
541541	C581499	M	Manual	-1	12/9/2011 10:28	224.69	
541715	C581568	21258	VICTORIAN SEWING BOX LARGE	-5	12/9/2011 11:57	10.95	
541716	C581569	84978	HANGING HEART JAR T-LIGHT HOLDER	-1	12/9/2011 11:58	1.25	
541717	C581569	20979	36 PENCILS TUBE RED RETROSPOT	-5	12/9/2011 11:58	1.25	

8905 rows × 8 columns

In [13]:
```
data[data['UnitPrice']<0].shape[0]
```

Out[13]:
```
0
```

Drop the negative values or quantity

In [14]:
```
data.drop(data[data['Quantity'] < 0].index, inplace=True)
```

In [15]:
```
data.shape
```

Out[15]: (397924, 8)

Calculation of total amount or monetary value

In [16]:
```
data['Sales'] = data['Quantity'] * data['UnitPrice']
new_data = data.groupby('CustomerID')['Sales'].sum().reset_index()
new_data
```

Out[16]:

	CustomerID	Sales
0	12346.0	77183.60
1	12347.0	4310.00
2	12348.0	1797.24
3	12349.0	1757.55
4	12350.0	334.40
...	...	...
4334	18280.0	180.60
4335	18281.0	80.82
4336	18282.0	178.05
4337	18283.0	2094.88
4338	18287.0	1837.28

4339 rows × 2 columns

In [17]:
```
type(new_data)
```

Out[17]: pandas.core.frame.DataFrame

Calculating the number of transactions of each customer

In [18]:
```
num_trans = data.groupby('CustomerID')['InvoiceNo'].count().reset_index()
num_trans
```

Out[18]:

	CustomerID	InvoiceNo
0	12346.0	1
1	12347.0	182
2	12348.0	31
3	12349.0	73
4	12350.0	17
...	...	...
4334	18280.0	10
4335	18281.0	7
4336	18282.0	12
4337	18283.0	756
4338	18287.0	70

4339 rows × 2 columns

In [19]:
```
type(num_trans)
```

Out[19]:
```
pandas.core.frame.DataFrame
```

Calculate Last Transaction

In [20]:
```
data['InvoiceDate'] = pd.to_datetime(data['InvoiceDate'])
```

In [21]:
```
data['InvoiceDate']
```

Out[21]:
```
0        2010-12-01 08:26:00
1        2010-12-01 08:26:00
2        2010-12-01 08:26:00
3        2010-12-01 08:26:00
4        2010-12-01 08:26:00
                 ...
541904   2011-12-09 12:50:00
541905   2011-12-09 12:50:00
541906   2011-12-09 12:50:00
541907   2011-12-09 12:50:00
541908   2011-12-09 12:50:00
Name: InvoiceDate, Length: 397924, dtype: datetime64[ns]
```

In [22]:
```
data['Last Transaction'] = (data['InvoiceDate'].max() - data['InvoiceDate
```

In [23]:
```
data['Last Transaction']
```

Out[23]:
```
0         373
1         373
2         373
3         373
4         373
          ...
541904      0
541905      0
541906      0
541907      0
541908      0
Name: Last Transaction, Length: 397924, dtype: int64
```

In [24]:
```
data.head(10)
```

Out[24]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerI
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 08:26:00	2.55	17850.
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	3.39	17850.
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 08:26:00	2.75	17850.
3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	2010-12-01 08:26:00	3.39	17850.
4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	2010-12-01 08:26:00	3.39	17850.
5	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	2010-12-01 08:26:00	7.65	17850.
6	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	2010-12-01 08:26:00	4.25	17850.
7	536366	22633	HAND WARMER UNION JACK	6	2010-12-01 08:28:00	1.85	17850.
8	536366	22632	HAND WARMER RED POLKA DOT	6	2010-12-01 08:28:00	1.85	17850.
9	536367	84879	ASSORTED COLOUR BIRD ORNAMENT	32	2010-12-01 08:34:00	1.69	13047.

In [25]:
```
lt = data.groupby(['CustomerID','Country'])['Last Transaction'].max().res
```

In [26]:
```
lt
```

Out[26]:

	CustomerID	Country	Last Transaction
0	12346.0	United Kingdom	325
1	12347.0	Iceland	366
2	12348.0	Finland	357
3	12349.0	Italy	18
4	12350.0	Norway	309
...	...	...	...
4342	18280.0	United Kingdom	277
4343	18281.0	United Kingdom	180
4344	18282.0	United Kingdom	125
4345	18283.0	United Kingdom	336
4346	18287.0	United Kingdom	201

4347 rows × 3 columns

In [27]:
```
merge_table = pd.merge(lt, num_trans, how='inner', on='CustomerID')
new_df = pd.merge(merge_table, new_data, how='inner', on='CustomerID')
new_df
```

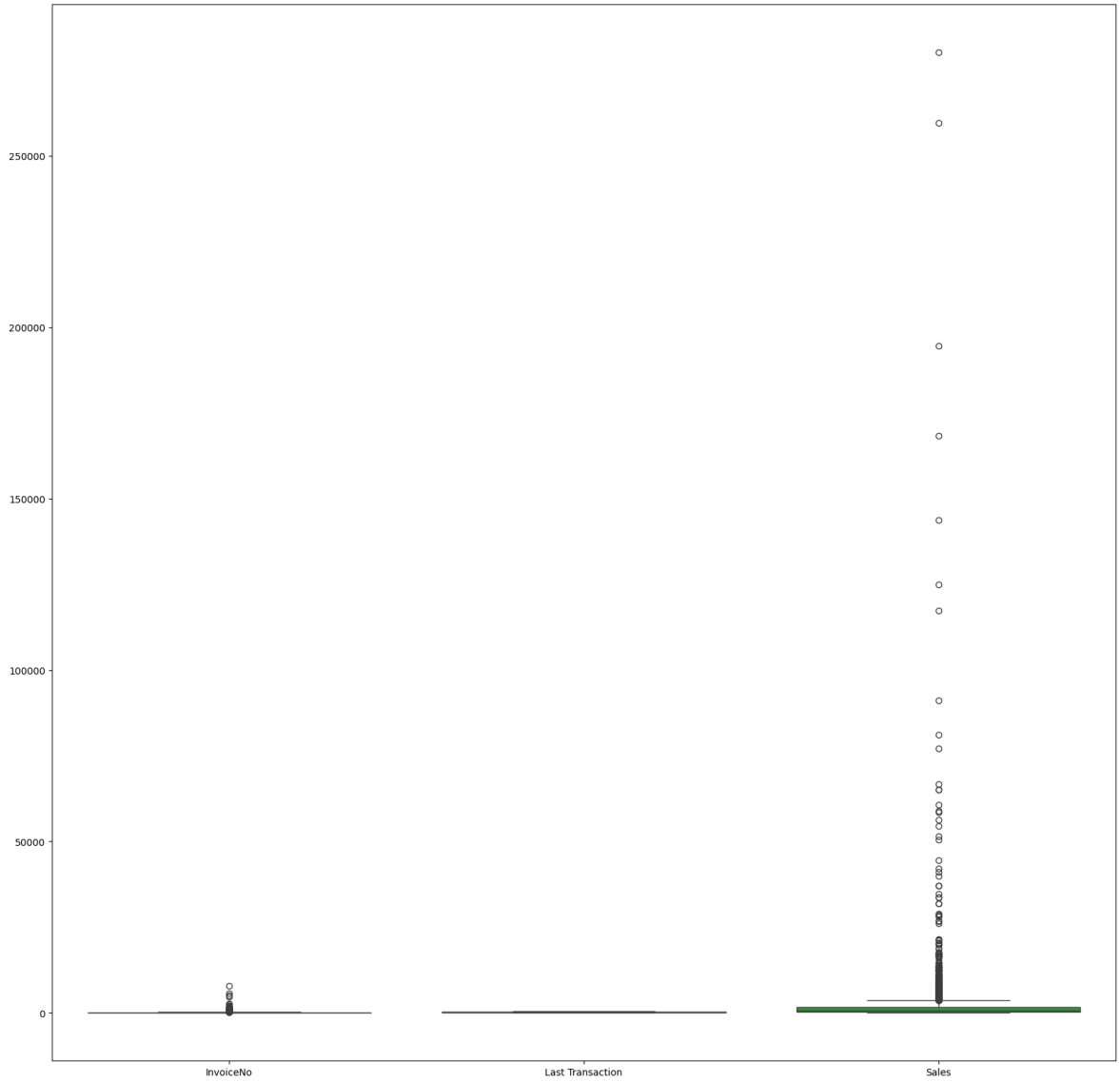
Out[27]:

	CustomerID	Country	Last Transaction	InvoiceNo	Sales
0	12346.0	United Kingdom	325	1	77183.60
1	12347.0	Iceland	366	182	4310.00
2	12348.0	Finland	357	31	1797.24
3	12349.0	Italy	18	73	1757.55
4	12350.0	Norway	309	17	334.40
...	...	...	...	...	...
4342	18280.0	United Kingdom	277	10	180.60
4343	18281.0	United Kingdom	180	7	80.82
4344	18282.0	United Kingdom	125	12	178.05
4345	18283.0	United Kingdom	336	756	2094.88
4346	18287.0	United Kingdom	201	70	1837.28

4347 rows × 5 columns

Removal of Outliers using boxplot

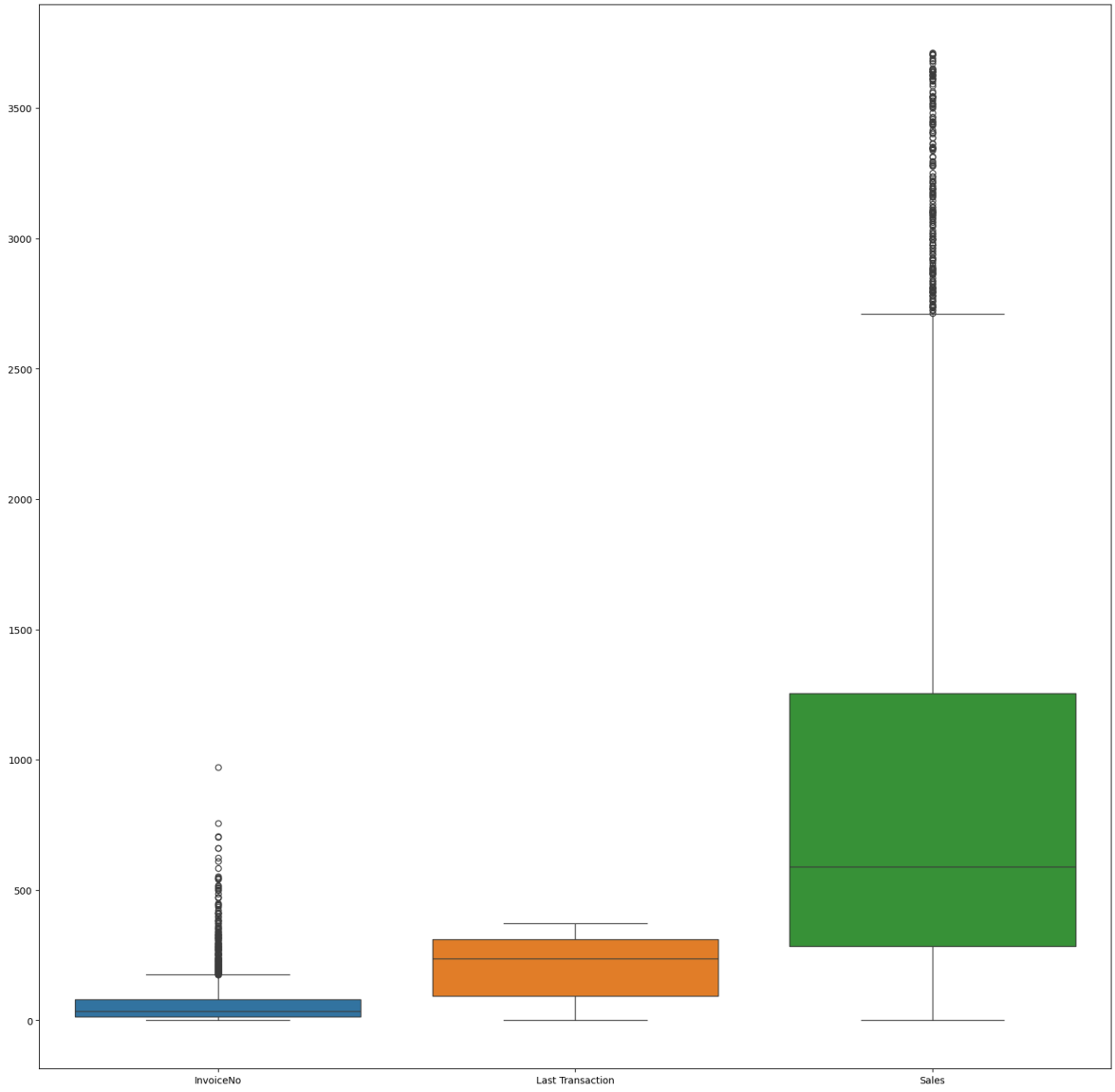
In [28]:
```
plt.figure(figsize=(20,20))
sns.boxplot(data = new_df[['InvoiceNo', 'Last Transaction', 'Sales']])
plt.show()
```



In [29]:
```
IQR = new_df['Sales'].quantile(0.75) - new_df['Sales'].quantile(0.25)
lower_limit = new_df['Sales'].quantile(0.25) - 1.5*IQR
upper_limit = new_df['Sales'].quantile(0.75) + 1.5*IQR
new_df_iqr = new_df[(new_df['Sales'] < upper_limit) & (new_df['Sales']>lo
new_df_iqr.shape
```

Out[29]: (3923, 5)

In [30]:
```
plt.figure(figsize=(20,20))
sns.boxplot(data = new_df_iqr[['InvoiceNo', 'Last Transaction', 'Sales']]
plt.show()
```



In [31]:
```
new_df_iqr.reset_index(drop=True, inplace=True)
new_df_iqr
```

Out[31]:

	CustomerID	Country	Last Transaction	InvoiceNo	Sales
0	12348.0	Finland	357	31	1797.24
1	12349.0	Italy	18	73	1757.55
2	12350.0	Norway	309	17	334.40
3	12352.0	Norway	296	85	2506.04
4	12353.0	Bahrain	203	4	89.00
...	...	...	...	...	...
3918	18280.0	United Kingdom	277	10	180.60
3919	18281.0	United Kingdom	180	7	80.82
3920	18282.0	United Kingdom	125	12	178.05
3921	18283.0	United Kingdom	336	756	2094.88
3922	18287.0	United Kingdom	201	70	1837.28

3923 rows × 5 columns

Visualization of correlation matrix using heatmap

In [ ]:
```
plt.figure(figsize=(20,20))
sns.heatmap(new_df_iqr.corr(),cmap="Greens", annot=True)
plt.show()
```

Normalization of given dataset using MinMaxScaler

In [33]:
```
from sklearn.preprocessing import MinMaxScaler
```

In [34]:
```
new2_df= new_df_iqr[['Last Transaction','InvoiceNo','Sales']]
scaler = MinMaxScaler()
scaled_df = scaler.fit_transform(new2_df)
scaled_df = pd.DataFrame(scaled_df)
scaled_df.columns = ['Last Transaction','InvoiceNo','Sales']
scaled_df['Country'] = new_df_iqr['Country']
scaled_df
```

Out[34]:

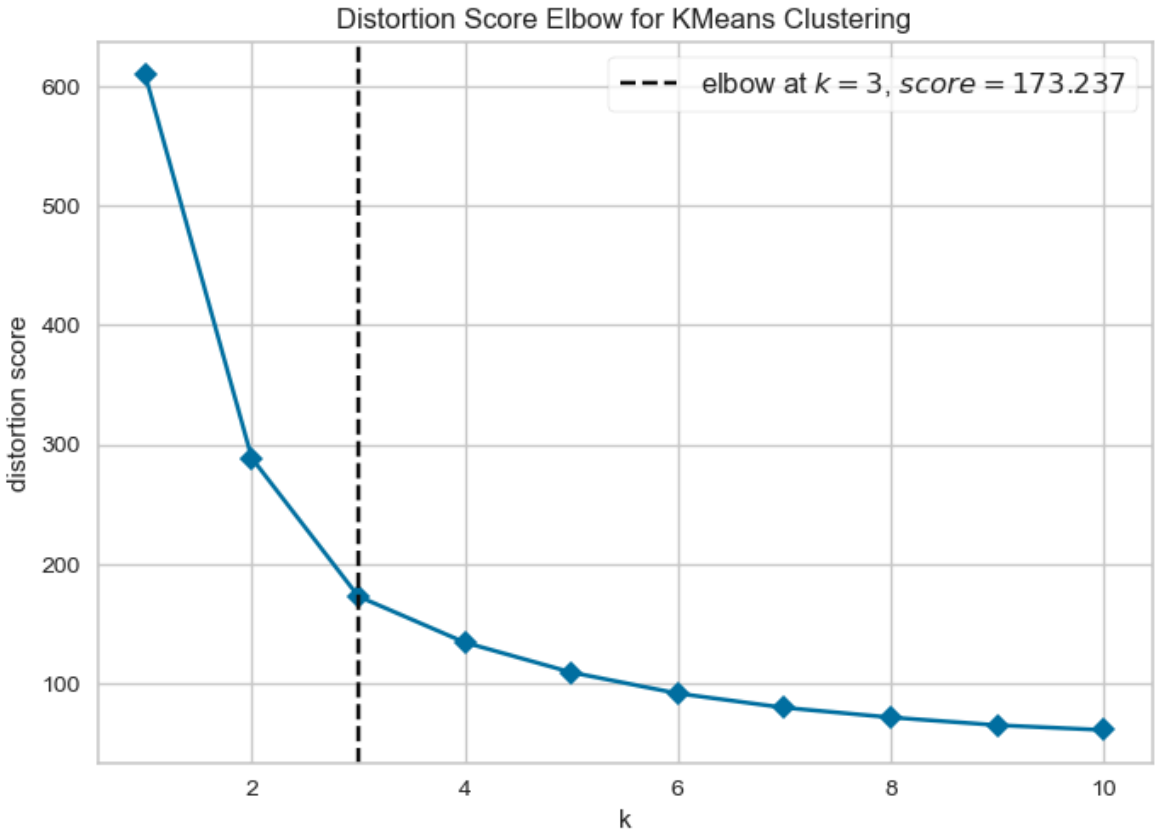
	Last Transaction	InvoiceNo	Sales	Country
0	0.957105	0.030960	0.484200	Finland
1	0.048257	0.074303	0.473507	Italy
2	0.828418	0.016512	0.090092	Norway
3	0.793566	0.086687	0.675160	Norway
4	0.544236	0.003096	0.023978	Bahrain
...	...	...	...	...
3918	0.742627	0.009288	0.048656	United Kingdom
3919	0.482574	0.006192	0.021774	United Kingdom
3920	0.335121	0.011352	0.047969	United Kingdom
3921	0.900804	0.779154	0.564388	United Kingdom
3922	0.538874	0.071207	0.494988	United Kingdom

3923 rows × 4 columns

K Means Clustering - Plot the graph using elbow method

In [35]:
```
# !conda install -c districtdatalabs yellowbrick
```

In [36]:
```
df_k=scaled_df.drop(columns=['Country'],axis=1)
# Elbow Method for K means
# Import ElbowVisualizer
from yellowbrick.cluster import KElbowVisualizer
model = KMeans()
# k is range of number of clusters.
visualizer = KElbowVisualizer(model, k=(1,11), timings= False)
visualizer.fit(df_k)        # Fit data to visualizer
visualizer.show()
```



Out[36]:
```
<Axes: title={'center': 'Distortion Score Elbow for KMeans Clustering'},
xlabel='k', ylabel='distortion score'>
```

In [37]:
```
km = KMeans(n_clusters=3)
y_predicted = km.fit_predict(df_k)
df_k['clusters']= y_predicted
df_k
```

Out[37]:

	Last Transaction	InvoiceNo	Sales	clusters
0	0.957105	0.030960	0.484200	2
1	0.048257	0.074303	0.473507	1
2	0.828418	0.016512	0.090092	0
3	0.793566	0.086687	0.675160	2
4	0.544236	0.003096	0.023978	0
...	...	...	...	...
3918	0.742627	0.009288	0.048656	0
3919	0.482574	0.006192	0.021774	1
3920	0.335121	0.011352	0.047969	1
3921	0.900804	0.779154	0.564388	2
3922	0.538874	0.071207	0.494988	2

3923 rows × 4 columns

In [38]:
```
km.cluster_centers_
```

Out[38]:
```
array([[0.77317137, 0.04263809, 0.15710448],
       [0.21717256, 0.04061798, 0.14981966],
       [0.80074265, 0.15760362, 0.63323941]])
```

Model Evaluation silhouette

In [39]:
```
from sklearn.metrics import silhouette_samples, silhouette_score
score = silhouette_score(df_k, km.labels_, metric='euclidean')
print(score)
```

```
0.7825447555040321
```

Hierarchical clustering

In [40]:
```
from sklearn.cluster import AgglomerativeClustering
hierarchial = AgglomerativeClustering(n_clusters=3)
y_predicted_hierarchial = km.fit_predict(df_k)
df_k['clusters_hierarchial']= y_predicted_hierarchial
df_k
```

Out[40]:

	Last Transaction	InvoiceNo	Sales	clusters	clusters_hierarchial
0	0.957105	0.030960	0.484200	2	2
1	0.048257	0.074303	0.473507	1	1
2	0.828418	0.016512	0.090092	0	0
3	0.793566	0.086687	0.675160	2	2
4	0.544236	0.003096	0.023978	0	0
...	...	...	...	...	...
3918	0.742627	0.009288	0.048656	0	0
3919	0.482574	0.006192	0.021774	1	1
3920	0.335121	0.011352	0.047969	1	1
3921	0.900804	0.779154	0.564388	2	2
3922	0.538874	0.071207	0.494988	2	2

3923 rows × 5 columns

In [41]:
```
from sklearn.cluster import AgglomerativeClustering
hierarchial = AgglomerativeClustering(n_clusters=2)
y_predicted_hierarchial = km.fit_predict(df_k)
df_k['clusters_hierarchial']= y_predicted_hierarchial
df_k.head(30)
```

Out[41]:

	Last Transaction	InvoiceNo	Sales	clusters	clusters_hierarchial
0	0.957105	0.030960	0.484200	2	2
1	0.048257	0.074303	0.473507	1	0
2	0.828418	0.016512	0.090092	0	1
3	0.793566	0.086687	0.675160	2	2
4	0.544236	0.003096	0.023978	0	1
5	0.619303	0.058824	0.290805	0	1
6	0.571046	0.012384	0.123768	0	1
7	0.871314	0.059856	0.757436	2	2
8	0.402145	0.018576	0.314691	1	0
9	0.536193	0.132095	0.717194	2	2
10	0.766756	0.009288	0.051162	0	1
11	0.646113	0.022704	0.148716	0	1
12	0.297587	0.086687	0.353767	1	0
13	0.777480	0.021672	0.172796	0	1
14	0.008043	0.010320	0.045504	1	0
15	0.957105	0.171311	0.955256	2	2
16	0.962466	0.171311	0.955256	2	2
17	0.158177	0.063983	0.508641	1	0
18	0.793566	0.052632	0.349709	0	1
19	0.831099	0.013416	0.098228	0	1
20	0.064343	0.033024	0.200155	1	0
21	0.260054	0.016512	0.123257	1	0
22	0.949062	0.078431	0.438637	2	2
23	0.453083	0.040248	0.229605	1	0
24	0.495979	0.106295	0.734100	2	2
25	0.319035	0.088751	0.497151	1	0
26	0.941019	0.101135	0.498565	2	2
27	0.324397	0.026832	0.157679	1	0
28	0.981233	0.009288	0.108277	0	1
29	0.873995	0.102167	0.749147	2	2

In [42]:

```
from sklearn.cluster import AgglomerativeClustering
hierarchial = AgglomerativeClustering(n_clusters=3)
y_predicted_hierarchial = km.fit_predict(df_k)
df_k['clusters_hierarchial']= y_predicted_hierarchial
df_k.head(30)
```

Out[42]:

	Last Transaction	InvoiceNo	Sales	clusters	clusters_hierarchial
0	0.957105	0.030960	0.484200	2	2
1	0.048257	0.074303	0.473507	1	1
2	0.828418	0.016512	0.090092	0	0
3	0.793566	0.086687	0.675160	2	2
4	0.544236	0.003096	0.023978	0	0
5	0.619303	0.058824	0.290805	0	0
6	0.571046	0.012384	0.123768	0	0
7	0.871314	0.059856	0.757436	2	2
8	0.402145	0.018576	0.314691	1	1
9	0.536193	0.132095	0.717194	2	2
10	0.766756	0.009288	0.051162	0	0
11	0.646113	0.022704	0.148716	0	0
12	0.297587	0.086687	0.353767	1	1
13	0.777480	0.021672	0.172796	0	0
14	0.008043	0.010320	0.045504	1	1
15	0.957105	0.171311	0.955256	2	2
16	0.962466	0.171311	0.955256	2	2
17	0.158177	0.063983	0.508641	1	1
18	0.793566	0.052632	0.349709	0	0
19	0.831099	0.013416	0.098228	0	0
20	0.064343	0.033024	0.200155	1	1
21	0.260054	0.016512	0.123257	1	1
22	0.949062	0.078431	0.438637	2	2
23	0.453083	0.040248	0.229605	1	1
24	0.495979	0.106295	0.734100	2	2
25	0.319035	0.088751	0.497151	1	1
26	0.941019	0.101135	0.498565	2	2
27	0.324397	0.026832	0.157679	1	1
28	0.981233	0.009288	0.108277	0	0
29	0.873995	0.102167	0.749147	2	2

DBSCAN

In [43]:
```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps=0.2, min_samples=4)
```

```
y_predicted_dbscan = dbscan.fit_predict(df_k)
df_k['clusters_dbscan'] = y_predicted_dbscan
df_k
```

Out[43]:

	Last Transaction	InvoiceNo	Sales	clusters	clusters_hierarchial	clusters_db
0	0.957105	0.030960	0.484200	2	2	
1	0.048257	0.074303	0.473507	1	1	
2	0.828418	0.016512	0.090092	0	0	
3	0.793566	0.086687	0.675160	2	2	
4	0.544236	0.003096	0.023978	0	0	
...	...	...	...	...	...	
3918	0.742627	0.009288	0.048656	0	0	
3919	0.482574	0.006192	0.021774	1	1	
3920	0.335121	0.011352	0.047969	1	1	
3921	0.900804	0.779154	0.564388	2	2	
3922	0.538874	0.071207	0.494988	2	2	

3923 rows × 6 columns

In [44]:
```
from sklearn.metrics import silhouette_samples, silhouette_score
score_dbscan = silhouette_score(df_k, dbscan.labels_, metric='euclidean')
print(score_dbscan)
```

```
0.8480678877758114
```

checkout page: https://scikit-learn.org/stable/modules/clustering.html