Customer Clustering Master Class:

Discover Hidden Patterns in Your Data

Minor In AI, IIT Ropar
14th May, 2025

1 Unveiling Customer Secrets – Our First Clustering Adventure

Welcome, future AI explorer! Imagine you're a friendly neighborhood shopkeeper. You have many customers, and you'd love to understand them better. Are there groups of customers who buy similar things? Or customers who visit very frequently? Knowing this could help you offer better deals, stock the right items, and make your customers happier! This is where the magic of **customer clustering** comes in. It's like sorting a big, mixed box of colorful beads into smaller boxes, where each box contains beads of similar colors or sizes. We're going to learn how to do this with data!

Today, we'll embark on a journey to group, or "cluster," customers based on their shopping habits. Think of it as a fun, light day where we put together different pieces of a puzzle to see the bigger picture.

1.1 Our Treasure Map: The Online Retail Dataset

Our adventure starts with a real-world dataset called "Online Retail." It's like a giant logbook of transactions from an online store, containing over half a million entries! This logbook has details like what was bought, how much of it, who bought it, and when.

figureVisual representation of the Online Retail dataset

Now, real-world data is often a bit messy, like a pirate's map that's been through a storm. Our first task is to clean it up!

1.2 The Clean-up Crew: Data Preprocessing

Before we can find our customer groups, we need to tidy up our data. Here's what we encountered in our dataset:

1. **Missing Customer IDs:** Some transactions didn't have a `CustomerID`. If we don't know who the customer is, we can't group them. So, we'll remove these entries.
2. **Cancelled Orders:** Some orders were cancelled (their `InvoiceNo` often starts with a 'C'). These don't represent actual purchases, so they shouldn't be part of our analysis.

3. **Negative Quantities:** Sometimes, a quantity might be negative, perhaps due to returns or errors. For our current goal of understanding purchase behavior, we'll focus on positive quantities.

The lecture explained, "If your customer ID is missing, you can probably use it for something else, but you can't use them to cluster your data... for the current experimentation... customer ID is required."

Here's how we can instruct our computer (using Python and a library called Pandas) to do this cleaning:

```python
import pandas as pd

# Load the dataset
df = pd.read_excel('Online Retail.xlsx')

# Remove rows with missing customer ids
df = df[pd.notnull(df['CustomerID'])]

# Remove canceled orders
df = df[~df['InvoiceNo'].astype(str).str.startswith('C')]

# Remove negative quantities
df = df[df['Quantity'] > 0]
```

Think of `df` as our digital spreadsheet. Each line of code tells the computer to filter out the unwanted data, making our dataset cleaner and more reliable for clustering.

1.3 Finding the Clues: Feature Engineering

With our data cleaned, we need to decide *what* information (or "features") about our customers we'll use to group them. We're interested in:

1. **Frequency:** How often does a customer make a purchase? (We can count their unique invoices).
2. **Total Quantity:** How many items, in total, does a customer buy?
3. **Monetary Value:** How much money, in total, does a customer spend?

First, let's calculate the total price for each item bought (`TotalPrice = Quantity * UnitPrice`). Then, we'll group all transactions by each `CustomerID` and sum up their quantities and total prices, and count their unique invoices.

```python
# Calculate TotalPrice
df['TotalPrice'] = df['Quantity'] * df['UnitPrice']

# Aggregate data by CustomerID
customer_df = df.groupby('CustomerID').agg({
    'InvoiceNo': 'nunique',  # Frequency
    'Quantity': 'sum',       # TotalQuantity
    'TotalPrice': 'sum'     # Monetary
}).rename(columns={
    'InvoiceNo': 'Frequency',
    'Quantity': 'TotalQuantity',
    'TotalPrice': 'Monetary'
}).reset_index()
```

Now, `customer_df` holds a neat summary for each customer: their purchasing frequency, total quantity of items bought, and total monetary spending.

1.4 Leveling the Playing Field: Feature Scaling

Imagine comparing a customer who buys 100 small items (total quantity) with another who buys 2 very expensive items (high monetary value). The numbers are on different scales! `Frequency` might be small numbers (e.g., 1-50 visits), `TotalQuantity` could be hundreds, and `Monetary` could be thousands. If we don't "level the playing field," features with larger numbers might unfairly dominate our clustering.

This is where **Standard Scaler** comes in. It transforms our data so that all features have a similar scale, without changing the underlying patterns.

```python
from sklearn.preprocessing import StandardScaler

features = ['Frequency', 'TotalQuantity', 'Monetary']
scaler = StandardScaler()
scaled_features = scaler.fit_transform(customer_df[features])
```

Our `scaled_features` are now ready for the main event: clustering!

2 Meet K-Means: Finding Natural Groupings

K-Means is a popular algorithm that helps us find k distinct groups (or clusters) in our data. It works by:

1. Picking k initial "centers" (called centroids) for our clusters.
2. Assigning each customer to the nearest centroid.
3. Recalculating the centroids based on the customers assigned to them.
4. Repeating steps 2 and 3 until the centroids don't change much.

The big question is: **What's the right number of clusters, k?**

2.1 The Elbow Method: Pointing to the Optimal k

One way to find a good k is the **Elbow Method**. We try different values of k (say, from 1 to 10) and for each k, we calculate the **Sum of Squared Errors (SSE)**, also called **inertia**. Inertia measures how spread out the customers are within their clusters – lower is better.

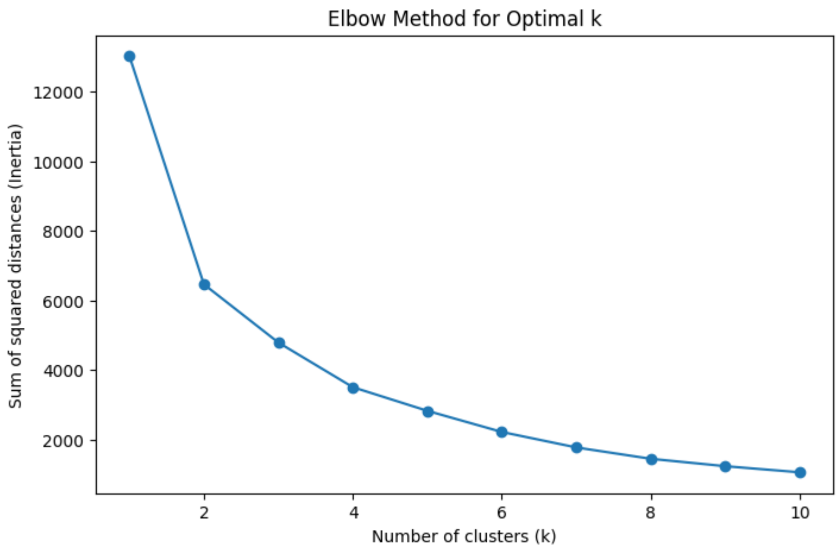
As we increase k, inertia will generally decrease. But at some point, adding more clusters doesn't give us much improvement. We look for an "elbow" point in the plot of k vs. inertia – the point where the rate of decrease sharply changes.

As the lecturer put it, we use the Elbow Method "to get an optimal number of clusters that will effectively define the customer pattern."

```python
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

sse = []
k_range = range(1, 11) # Try k from 1 to 10
for k_val in k_range:
    kmeans = KMeans(n_clusters=k_val, random_state=42, n_init=10) # n_init=10
    helps stabilize results
    kmeans.fit(scaled_features)
    sse.append(kmeans.inertia_)

# Plot the Elbow Curve
plt.figure(figsize=(8, 5))
plt.plot(k_range, sse, marker='o')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Sum of squared distances (Inertia)')
plt.show()
```



figureThe Elbow Curve showing inertia vs. number of clusters

Looking at the plot, you might see an elbow around k=3, 4, or 5. Sometimes the elbow isn't super sharp, and that's okay! The lecture highlighted this: "Sometimes it's very, very difficult to get in even though you use the elbow method... it's kind of flexible for you to take a decision."

2.2 The Silhouette Score: How Good Are Our Clusters?

While the Elbow Method helps pick k, the **Silhouette Score** helps us evaluate how well-defined our clusters are for a chosen k. For each customer, it measures:

1. **a**: The average distance to other customers in the *same* cluster (cohesion – we want this to be small).
2. **b**: The average distance to customers in the *nearest other* cluster (separation – we want this to be large).

The Silhouette Score for a customer is `(b - a) / max(a, b)`.

- A score near **+1** means the customer is well-clustered.
- A score near **0** means the customer is close to the boundary between two clusters.
- A score near **-1** means the customer might be in the wrong cluster.

We calculate the average Silhouette Score for all customers.

```python
from sklearn.metrics import silhouette_score

silhouette_scores = []
k_range_silhouette = range(2, 11) # Silhouette score needs at least 2 clusters
for k_val in k_range_silhouette:
    kmeans = KMeans(n_clusters=k_val, random_state=42, n_init=10)
    labels = kmeans.fit_predict(scaled_features)
    score = silhouette_score(scaled_features, labels)
    silhouette_scores.append(score)
    print(f'For n_clusters = {k_val}, the Silhouette Score is {score:.4f}')

# Plot Silhouette Scores
plt.figure(figsize=(8, 5))
plt.plot(k_range_silhouette, silhouette_scores, marker='o')
plt.title('Silhouette Scores for Various Clusters')
plt.xlabel('Number of clusters')
plt.ylabel('Silhouette Score')
plt.show()
```

The lecture noted, "for the number cluster from five... silhouette score has maybe... the number is not varying right it's around point six... whereas initially for two and three it's a little high nine point six nine point two." This suggests that 2 or 3 clusters might be quite distinct.

2.3 K-Means in Action: Visualizing Customer Segments

Let's say, after looking at both the Elbow Method and Silhouette Scores, we decide that `optimal_k = 4` is a reasonable choice for our shopkeeper.

```python
optimal_k = 4
kmeans = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)
customer_df['Cluster'] = kmeans.fit_predict(scaled_features)
```

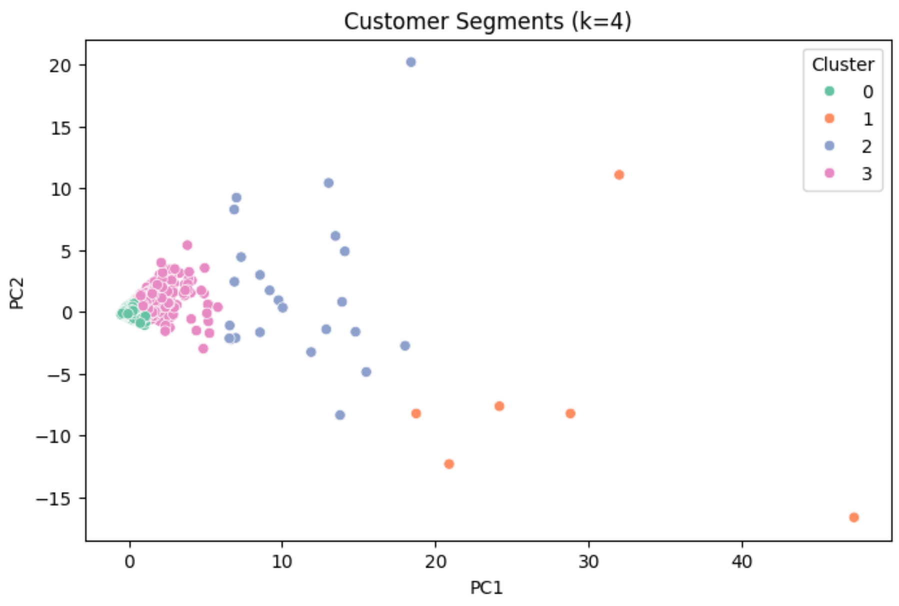
Now, each customer in `customer_df` has a `Cluster` label (0, 1, 2, or 3).

But how do we *see* these clusters? Our data has three features (Frequency, TotalQuantity, Monetary). It's hard to plot in 3D, let alone more. We can use a technique called **Principal Component Analysis (PCA)** to reduce our three features into two "principal components" that capture most of the information. This lets us plot our clusters on a 2D graph.

```python
from sklearn.decomposition import PCA
import seaborn as sns # For nicer plots

pca = PCA(n_components=2) # Reduce to 2 components
principal_components = pca.fit_transform(scaled_features)
customer_df['PC1'] = principal_components[:, 0]
customer_df['PC2'] = principal_components[:, 1]

plt.figure(figsize=(8, 5))
sns.scatterplot(data=customer_df, x='PC1', y='PC2', hue='Cluster', palette='Set2
    ')
plt.title('Customer Segments (k=4)')
plt.show()
```



figureVisualization of customer clusters in 2D space using PCA

Looking at this plot, our shopkeeper might see, for example, a dense green cluster (perhaps regular, loyal customers), a pink cluster (maybe big spenders but less frequent), and more scattered blue and orange points (perhaps occasional or diverse buyers). The lecture pointed out: "Maybe you can tell your shopkeeper now the one which you see in green, pink, and to some extent in the blue color can be considered for whatever you want to do... but for others they are very scattered, it's very, very difficult to come up with a common strategy."

3 A New Game: The Kid's Approach to Clustering

Imagine a child looking at a scatter of points (our customers). How might they group them? The lecture painted a wonderful picture of this, leading us to **Hierarchical Clustering**.

3.1 Approach 1: Building Up (Agglomerative Clustering)

The child might start by saying, "Each point is its own little group!" Then, they might look for the two closest "groups" (initially, two closest points) and merge them. They'd repeat this, merging the next closest pair of groups, and so on, until all points are in one giant group.

This "bottom-up" approach is called **Agglomerative Clustering**.

Key Concept

- **Start:** Each data point is its own cluster.
- **Process:** Iteratively merge the closest pair of clusters.
- **End:** All data points are in a single cluster.
- **Commonality:** This is the most commonly used hierarchical method.

Here's a tiny example of how it looks with code, using just a few sample points:

```python
from sklearn.cluster import AgglomerativeClustering
import numpy as np

# Sample 2D data points
```

```python
X_sample = np.array([[1, 2], [2, 3], [5, 6], [6, 6], [3, 6], [2, 5], [3, 4]])

# Agglomerative clustering (let's say we want 3 clusters at the end)
agg_clustering = AgglomerativeClustering(n_clusters=3)
labels_agg = agg_clustering.fit_predict(X_sample)

plt.scatter(X_sample[:, 0], X_sample[:, 1], c=labels_agg, cmap='rainbow')
plt.title("Agglomerative Clustering Example")
plt.show()
```

3.2 Approach 2: Breaking Down (Divisive Clustering)

Now, the child might try a different game. "Let's put ALL the points in ONE big circle first!" Then, they might look for a way to split this big group into two smaller, more distinct groups. They'd continue splitting the largest or most spread-out group until each point is in its own tiny group (or until they reach a desired number of groups).

This "top-down" approach is called **Divisive Clustering**.

Key Concept

- **Start:** All data points are in one single cluster.
- **Process:** Iteratively split a cluster into smaller sub-clusters.
- **End:** Each data point is its own cluster.
- **Commonality:** This method is less commonly used than agglomerative.

While `sklearn` doesn't have a direct DivisiveClustering class that works like this out-of-the-box for general cases, we can simulate the idea by recursively using K-Means with k=2 to split clusters:

```python
# (Conceptual code for Divisive Clustering using KMeans for splitting)
# The lecture's code demonstrates a custom function for this.
# For simplicity here, imagine a process:
# 1. Start with all X_sample in one cluster.
# 2. Split it into 2 using KMeans(n_clusters=2).
# 3. If more splits are needed, pick one of the resulting clusters and split it
    further.

# Here's a simplified version from the lecture code using a helper function:
def divisive_clustering_simulation(data, max_clusters=2): # Simplified from
    lecture
    clusters_data = [data] # List of arrays, each is a cluster
    final_labels = np.zeros(len(data), dtype=int)
    current_label_id = 0

    while len(clusters_data) < max_clusters:
        # Find the largest cluster to split (by number of points)
        sizes = [len(c) for c in clusters_data]
        idx_to_split = np.argmax(sizes)
        data_to_split = clusters_data.pop(idx_to_split)

        if len(data_to_split) < 2: # Cannot split if less than 2 points
            clusters_data.append(data_to_split) # Add it back and skip
            continue

        # Apply KMeans with 2 clusters to split
        kmeans_splitter = KMeans(n_clusters=2, n_init='auto', random_state=42)
        split_labels = kmeans_splitter.fit_predict(data_to_split)

        # Add the two new clusters
        clusters_data.append(data_to_split[split_labels == 0])
        clusters_data.append(data_to_split[split_labels == 1])

    # Assign final labels based on the resulting clusters
    for cluster_idx, cluster_points in enumerate(clusters_data):
        for point in cluster_points:
            # Find original index of the point in X_sample to assign label
            original_indices = np.where((X_sample == point).all(axis=1))[0]
            if len(original_indices) > 0:
                final_labels[original_indices[0]] = cluster_idx
    return final_labels

labels_divisive = divisive_clustering_simulation(X_sample, max_clusters=2) # Let's aim for 2 final clusters

plt.scatter(X_sample[:, 0], X_sample[:, 1], c=labels_divisive, cmap='viridis') # Using a different colormap
plt.title("Divisive Clustering Example (Simulated)")
plt.show()
```

Here's a quick comparison:

Feature	**Agglomerative (Bottom-Up)**	**Divisive (Top-Down)**
Start Point	Each data point is its own cluster.	All data points in one single cluster.
Process	Merge closest clusters one by one.	Recursively split clusters.
End Point	All data points merged into one cluster.	Every data point is its own cluster.
Approach	Bottom-Up	Top-Down
Commonality	Most commonly used.	Less commonly used.

4 What's Next?

We've had quite an adventure today! We started with a messy pile of online retail data, cleaned it up, figured out what features to use, and then used K-Means to find potential customer groups. We learned how to choose the number of clusters using the Elbow Method and evaluate them with the Silhouette Score. Finally, we peeked into a different way of thinking about clustering – the Agglomerative and Divisive approaches, inspired by how a child might intuitively group things.

In our next session, we'll dive deeper into these hierarchical clustering methods, explore their strengths, and see more cool applications! For now, remember that clustering is all about finding meaningful patterns and groups in your data. Just like our shopkeeper, you can use these techniques to gain valuable insights and make smarter decisions. Keep exploring!