Customer Churn Analysis for Telecommunications Company

Sishir Pandey - Project Manager

Fahim Arman - Data Engineer

Jitesh Akaveeti - Data Analyst (Predictive Modelling)

Chen - Data Analyst (Clustering)

Preeti Khatri - Data Analyst (Predictive Modelling)

Bishesh Aryal - Business Analyst

Clustering Analysis Documentation

Import Section:

Import Section

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
from scipy.spatial import ConvexHull
from matplotlib.patches import Polygon
import matplotlib.cm as cm
```

This code imports the essential Python libraries used for data preprocessing, clustering, and visualization. Pandas is used to load and manage the dataset efficiently, while NumPy handles numerical computations and array operations. The KMeans algorithm from Scikit-learn is employed to perform customer segmentation by grouping similar data points, and PCA (Principal Component Analysis) reduces the dataset's dimensions for easier visualization of clusters. Matplotlib.pyplot is used to plot and visualize the clustering results, while ConvexHull and Polygon from SciPy and Matplotlib help draw and fill boundaries around clusters to highlight them visually. Finally, matplotlib.cm provides color maps to assign distinct colors to different clusters, making the results more interpretable and visually appealing.

Data Loading:

Loading Data

```
In [2]: # Read data and display the first few rows

        data=pd.read_csv("data/X_train.csv")
```

This code reads the training dataset from a CSV file named "X_train.csv" using the pd.read_csv() function from the Pandas library and stores it in a DataFrame called data. A DataFrame is a structured data table that allows easy data manipulation and analysis. By loading the dataset, the analyst can access all the variables needed for preprocessing and modeling.

Data View:

Data View

```
In [3]: data.head()
```

Out[3]:

	SeniorCitizen	tenure	MonthlyCharges	gender_Female	gender_Male	Dependents_No	Dependents_Yes	PhoneService_No	PhoneService_Yes	MultipleLines_N
0	0	1	60	0	1	0	1	0	1	
1	0	30	55	1	0	1	0	0	1	
2	1	46	100	1	0	1	0	0	1	
3	0	43	66	0	1	1	0	0	1	
4	0	1	20	0	1	1	0	0	1	

The data.head() command displays the first five rows of the dataset. It helps the analyst quickly inspect the structure of the data, including column names, data types, and sample values, to ensure it has been loaded correctly before performing further analysis.

Data Normalization:

Data Normalization

```
In [4]: #Convert dataframe format to NumPy format
        X=data.values

        #data normalization
        x_mean=X.mean(axis=0)
        x_std=X.std(axis=0)
        X=(X-x_mean)/x_std
```

This code converts the dataset from a Pandas DataFrame to a NumPy array using data.values, which allows for faster numerical computations during modeling. Next, it performs data normalization to ensure all features are on a similar scale. It calculates the mean (x_mean) and standard deviation (x_std) for each feature, then standardizes the data by subtracting the mean and dividing by the standard deviation. This process transforms the dataset so that each feature has a mean of zero and a standard deviation of one, improving the performance and stability of machine learning algorithms such as K-Means and PCA.

PCA Dimensionality Reduction

```
In [5]: #PCA Dimensionality Reduction

        pca=PCA(n_components=2,random_state=0)
        pca.fit(X)
        X_pca=pca.transform(X)
```

This code performs Principal Component Analysis (PCA) to reduce the dataset's dimensionality while retaining as much important information as possible. The PCA function is initialized with n_components=2, meaning the data will be reduced to two principal components for easier visualization and analysis. By fitting the model using pca.fit(X), it identifies the directions (components) that capture the most variance in the dataset. The pca.transform(X) step then projects the original normalized data onto these two new axes, creating X_pca, a simplified version of the dataset that highlights the main patterns and relationships between variables.

Elbow Method:

Elbow method

In [6]:
```
#Elbow method for determining the optimal number of clusters

cs = []
k_list=np.array(list(range(1,16)))
for k in k_list:
    kmeans = KMeans(n_clusters = k,
                    init = 'k-means++',
                    max_iter = 300,
                    n_init = 10,
                    random_state = 0)
    kmeans.fit(X_pca)
    cs.append(kmeans.inertia_)
plt.figure(figsize=(10,8))
plt.plot(k_list+1, cs,'ko-')
plt.xlabel('Number of Clusters')
plt.ylabel('Within-Cluster Sum of Square')
plt.savefig("results/Elbow.png",dpi=500,bbox_inches='tight')
plt.show()
```

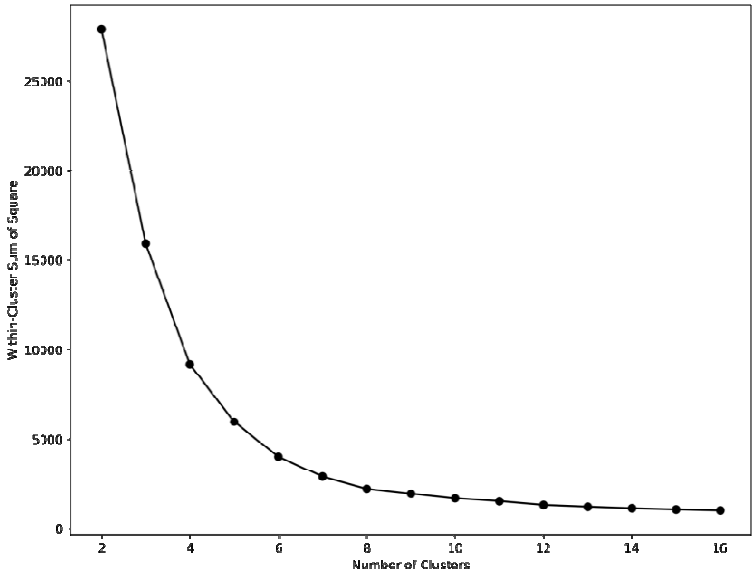
This code uses the Elbow Method to determine the optimal number of clusters for the K-Means algorithm. It tests different values of k (from 1 to 15) by repeatedly fitting K-Means models on the PCA-transformed dataset (X_pca). For each k, it records the inertia (within-cluster sum of squares) — a measure of how close data points are to their respective cluster centers. The goal is to find the point where adding more clusters does not significantly reduce inertia, forming an “elbow” shape on the graph.

The code then plots the number of clusters against their corresponding inertia values to visually identify this elbow point. Based on the resulting plot, the optimal number of clusters was determined to be k = 4, indicating that dividing the dataset into four clusters provides the best balance between model simplicity and accuracy. The graph is also saved as “Elbow.png” for documentation and reporting purposes.

Optimal Number of Clusters:

The elbow method, an unsupervised machine learning technique, is used to determine the optimal number of clusters for K-Means clustering. It helps identify the point at which adding more clusters no longer provides meaningful improvement in modeling the data.

This approach involves plotting inertia—the sum of squared distances between each data point and its assigned cluster centroid—against different values of K. As K increases, data points are divided into smaller, more precise clusters, making each cluster more compact. Beyond a certain point, however, the reduction in inertia becomes minimal. This point, where the curve starts to bend, is referred to as the “elbow.”



Picture : clusters was determined to be k = 4

In the graph, the y-axis represents inertia, while the x-axis shows the number of clusters. Reducing inertia from K=1 to K=4 improves model performance, but beyond K=4, the curve flattens, indicating that additional clusters provide little benefit. The elbow point, in this case K=4, represents the optimal balance between simplicity and accuracy. Using the elbow method, the K-Means model identifies the core structure of the data without creating unnecessary clusters or overfitting.

Clustering Analysis:

Clusters

```
In [7]: K=4
        kmeans=KMeans(n_clusters=K,random_state=0)
        kmeans.fit(X_pca)
        labels=kmeans.labels_
        cluster_centers=kmeans.cluster_centers_

        #Create graphics
        plt.figure(figsize=(8, 7))
        colors = cm.tab10(np.linspace(0, 1, K))
```

The code begins by defining the number of clusters, K = 4, meaning the dataset will be grouped into four distinct customer segments. The KMeans function from Scikit-learn is then initialized with four clusters and a fixed random state for reproducibility. The model is trained on the PCA-transformed dataset X_pca using the fit() method, which identifies the centroids of each cluster. After training, the assigned cluster labels for each data point are stored in labels, while the coordinates of the cluster centers are extracted into cluster_centers. This process allows the model to segment the data based on similarities in their features.

```
#Create graphics
plt.figure(figsize=(8, 7))
colors = cm.tab10(np.linspace(0, 1, K))
```

Next, a figure is created using Matplotlib with a size of 8x7 inches to visualize the clustering results. The code defines a set of distinct colors for the clusters using cm.tab10, which generates a palette of ten visually distinct colors evenly spaced across the color map. These colors help differentiate clusters visually in the scatter plot. This step prepares the plotting environment for clear and interpretable visualization of the clustered data.

```
for i in range(K):
    # Draw scatter plots and convex hulls for each cluster.
    cluster_points = X_pca[labels == i]

    # Draw a scatter plot
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1],
                c=[colors[i]], s=50, alpha=0.7,
                label=f'Cluster {i+1}', edgecolors='black', linewidth=0.1)

    if len(cluster_points) >= 3:  # The convex hull needs at least 3 points.
        hull = ConvexHull(cluster_points)

        # Get the coordinates of the convex hull vertex
        hull_points = cluster_points[hull.vertices]

        # Create a polygon and fill it with a light color
        polygon = Polygon(hull_points, closed=True,
                          facecolor=colors[i], alpha=0.2,
                          edgecolor=colors[i], linewidth=2)
        plt.gca().add_patch(polygon)
```

The code then loops through each cluster (from 0 to K–1) and plots its data points. For each cluster, the subset of points belonging to that cluster is extracted using X_pca[labels == i]. These points are plotted on a scatter plot with a distinct color, slight transparency (alpha=0.7), and a thin black edge to enhance visibility. To better highlight the boundary of each cluster, the code calculates the Convex Hull if the cluster contains at least three points. The convex hull defines the smallest polygon that encloses all points in the cluster, which is then drawn as a semi-transparent polygon on the plot. This helps visually separate clusters and understand their spatial boundaries.

```
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1],
            marker='x',s=100,c='k',
            label="Cluster centers",)
```

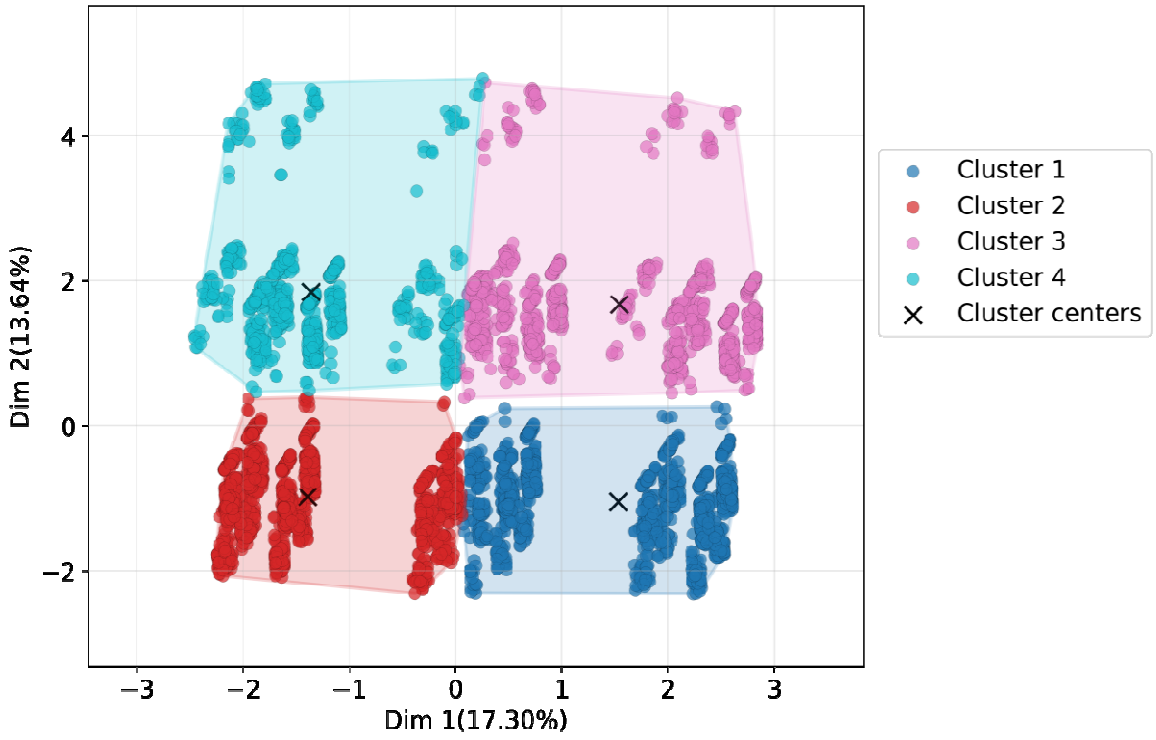
After plotting all individual data points and cluster boundaries, the code adds the cluster centers (centroids) to the plot. These centers are marked using an ‘x’ symbol in black (c='k') and sized larger (s=100) to make them easily visible. The label “Cluster centers” is added to identify them in the legend. These centroids represent the mean position of all the points belonging to each cluster and serve as reference points for interpreting cluster positions and relationships.

```
# Add legend and title
plt.legend(loc=(1.02,0.5),fontsize=14)

plt.xlabel('Dim 1({:.2f}%)'.format(100*pca.explained_variance_ratio_[0]), fontsize=14)
plt.ylabel('Dim 2({:.2f}%)'.format(100*pca.explained_variance_ratio_[1]), fontsize=14)
plt.grid(True, alpha=0.3)

# Set axis range
plt.xlim(X_pca[:, 0].min() - 1, X_pca[:, 0].max() + 1)
plt.ylim(X_pca[:, 1].min() - 1, X_pca[:, 1].max() + 1)
plt.tick_params(axis='both', which='major', labelsize=14)
plt.savefig("results/Cluster_scatter_plot.png", dpi=500, bbox_inches='tight')
plt.show()
```

Finally, the code enhances the visualization by adding legends, axis labels, and a title. The legend is positioned slightly outside the main plot to avoid overlap, and the axes are labeled with the percentage of variance explained by the first two PCA dimensions. A light grid is added for readability, and axis limits are adjusted to ensure all points are clearly visible. The tick labels are enlarged for clarity, and the finished scatter plot is saved as a high-resolution PNG file named "Cluster_scatter_plot.png" in the results folder. The plot is then displayed using plt.show(), providing a complete and visually informative representation of the customer segmentation results.



Picture: This Picture shows 4 cluster.

Save Data:

Save Data

In [8]:
```
#Preserve the Noise Center
cluster_centers=pca.inverse_transform(cluster_centers)
cluster_centers=cluster_centers*x_std+x_mean
centers_df=pd.DataFrame(cluster_centers,columns=data.columns,
                        index=["Clsuter {}".format(i+1) for i in range(K)])
centers_df.to_excel("results/cluster_center.xlsx")

#Save cluster labels
labels_df=data.copy()
labels_df["label"]=labels+1
labels_df.to_excel("results/cluster_label.xlsx", index=False)
#Save the clustering summary results
results_df=pd.DataFrame({"Cluster category":["cluster_{}".format(k+1) for k in range(K)],
                         "Frequency":[sum(labels==k) for k in range(K)],
                         "percentage(%)":["{:.2f}%".format(100*x/len(labels)) for x in [sum(labels==k) for k in range(K)]]})
results_df.to_excel("results/Clustering_results.xlsx", index=False)
results_df
```

This code section focuses on saving and documenting the results of the clustering analysis for further interpretation and reporting. First, it reconstructs the cluster centers from the PCA-reduced space back to the original feature space using the inverse_transform() function. These centers are then rescaled using the original data's mean (x_mean) and standard deviation (x_std) to preserve their original scale. The reconstructed centers are stored in a DataFrame (centers_df) and saved as an Excel file named "cluster_center.xlsx", which provides a summary of the key characteristics for each cluster.

Next, the code saves the cluster labels for each customer by adding a "label" column to the original dataset (data) and exporting it as "cluster_label.xlsx", allowing for easy identification of which cluster each record belongs to. Finally, it creates a summary report (results_df) showing each cluster's name, frequency (number of customers), and percentage representation in the dataset. This summary is saved as "Clustering_results.xlsx", providing a concise overview of the clustering distribution, which is valuable for business interpretation and presentation.