K-means and DBSCAN without framework for customer segmentation

Carlos Ayala Medina A01703682

Abstract

This document presents the implementation of the K-means and density-based clustering algorithms without using a framework or external library. The problem to solve is customer segmentation on a dataset containing customer's records from a groceries store database.

Introduction

Customer segmentation is the process of dividing a customer base into groups of individuals that share similar characteristics like demographics, behavior, and preferences. It's a powerful tool used by businesses to tailor marketing strategies and improve customer satisfaction which leads to revenue growth.

In this implementation, we will use a dataset of customer behavior and demographics to perform customer segmentation using K-means and DBSCAN clustering algorithms without the use of any frameworks. We will compare the results of the two algorithms and evaluate which is more effective for customer segmentation.

Dataset

The dataset contains information about customers, including demographic information, purchasing behavior, and response to marketing campaigns. It was provided by Dr. Omar Romero-Hernandez and obtained from Kaggle.

It contains 2241 instances and 29 columns. Next, we have a list with all the attributes in the dataset.

Attributes

People

- ID: Customer's unique identifier
- Year_Birth: Customer's birth year
- Education: Customer's education level
- Marital_Status: Customer's marital status
- Income: Customer's yearly household income
- Kidhome: Number of children in customer's household
- Teenhome: Number of teenagers in customer's household
- Dt_Customer: Date of customer's enrollment with the company
- Recency: Number of days since customer's last purchase
- Complain: 1 if the customer complained in the last 2 years, 0 otherwise

Products

- MntWines: Amount spent on wine in last 2 years
- MntFruits: Amount spent on fruits in last 2 years
- MntMeatProducts: Amount spent on meat in last 2 years
- MntFishProducts: Amount spent on fish in last 2 years
- MntSweetProducts: Amount spent on sweets in last 2 years
- MntGoldProds: Amount spent on gold in last 2 years

Promotion

- NumDealsPurchases: Number of purchases made with a discount
- AcceptedCmp1: 1 if customer accepted the offer in the 1st campaign, 0 otherwise
- AcceptedCmp2: 1 if customer accepted the offer in the 2nd campaign, 0 otherwise
- AcceptedCmp3: 1 if customer accepted the offer in the 3rd campaign, 0 otherwise
- AcceptedCmp4: 1 if customer accepted the offer in the 4th campaign, 0 otherwise
- AcceptedCmp5: 1 if customer accepted the offer in the 5th campaign, 0 otherwise
- Response: 1 if customer accepted the offer in the last campaign, 0 otherwise

Place

- NumWebPurchases: Number of purchases made through the company’s website
- NumCatalogPurchases: Number of purchases made using a catalogue
- NumStorePurchases: Number of purchases made directly in stores
- NumWebVisitsMonth: Number of visits to company’s website in the last month

The **target** is to perform clustering to summarize customer segments.

Data Cleaning and Preprocessing

First, the missing values were removed using the "dropna()" function. Then, the "Year_Birth" column was used to calculate the "Age" of each customer and the "Spent" column was created by adding the amount spent on various items.

```
#Remove NA values
data = data.dropna()

#Changing features to more appropriate data features
#Customer age from year of birth
data["Age"] = 2023-data["Year_Birth"]

#Total spendings on various items by adding the columns of each item
data["Spent"] = data["MntWines"]+ data["MntFruits"]
+ data["MntMeatProducts"]+ data["MntFishProducts"]+ data["MntSweetProducts"]+ data["MntGoldProds"]
```

Next, new columns were created to indicate whether the customer lives alone or with a partner ("Living_With"), the total number of children in the household ("Children"), and the total family size ("Family_Size"). The "Education" column was modified to have more appropriate values.

```
#Customer is living with a partner or alone
#Total number of children in each household
#From those previous two columns, we can calculate the total family size
data["Living_With"]=data["Marital_Status"].replace({"Married":"Partner", "Together":"Partner", "Absurd":"Alone",
                                                    "Widow":"Alone", "YOLO":"Alone", "Divorced":"Alone", "Single":"Alone",})
data["Children"]=data["Kidhome"]+data["Teenhome"]
data["Family_Size"] = data["Living_With"].replace({"Alone": 1, "Partner":2})+ data["Children"]

#Replacing the education column with more appropriate values (undergraduate, graduate, postgraduate)
data["Education"]=data["Education"].replace({"Basic":"Undergraduate","2n Cycle":"Undergraduate",
                                             "Graduation":"Graduate", "Master":"Postgraduate", "PhD":"Postgraduate"})
```

Extra features that were not needed were removed using the "drop()" function. Outliers were removed by setting caps on Age and Income values. Categorical values were encoded as integers using the "LabelEncoder()" function and the data was standardized using the "StandardScaler()" function.

```
#Dropping extra features we don't need
to_drop = ["Marital_Status", "Dt_Customer", "Z_CostContact", "Z_Revenue", "Year_Birth", "ID"]
data = data.drop(to_drop, axis=1)

#Delete outliers by setting a cap on Age and income.
data = data[(data["Age"]<90)]
data = data[(data["Income"]<600000)]

# Encode categorical values as integers for dbscan and kmeans
cat_columns = [col for col in data.columns if data[col].dtype == 'object']
label_encoders = {}
for col in cat_columns:
    label_encoders[col] = LabelEncoder()
# Encode categorical values in each column
for col, enc in label_encoders.items():
    data[col] = enc.fit_transform(data[col])

# Standardize the data with StandardScaler
scaler = StandardScaler()
scaler.fit(data)
scaled_ds = pd.DataFrame(scaler.transform(data),columns= data.columns )
aux = scaled_ds.values
```

Principal Component Analysis (PCA) is used to reduce the dimensionality of the data. The PCA() function from the sklearn library is called with the parameter n_components=2, which indicates that the number of dimensions in the output should be reduced to 2. The fit_transform() method is then used to fit the PCA model to the data and transform the data.

```
# Use PCA (Principal Component Analysis) to reduce the dimensionality of the data
pca = PCA(n_components=2)
data_pca = pca.fit_transform(aux)
```

K-Means Algorithm without framework

I defined a function named k_means that takes in three arguments: data_points, num_clusters, and max_iterations. It implements the K-means clustering algorithm to cluster the data points into num_clusters clusters.

The function first initializes num_clusters centroids randomly within the range of minimum and maximum values of the data points' features. It then iterates over the data points and assigns each point to the closest centroid. Next, it updates the centroids as the mean of the assigned data points. This process of assigning points and updating centroids is repeated for max_iterations times or until convergence is achieved.

Finally, the function returns the final centroids and the assignments of the data points to the closest cluster.

```
#Function to run K-means clustering
def k_means(data_points, num_clusters, max_iterations):
    # Set the dimensions of the feature space
    num_features = data_points.shape[1]

    # Generate random initial centroids
    centroids = np.zeros((num_clusters, num_features))
    min_vals = np.min(data_points, axis=0)
    max_vals = np.max(data_points, axis=0)
    for i in range(num_clusters):
        centroids[i, :] = np.random.uniform(min_vals, max_vals)

    # Iterate until convergence or maximum number of iterations is reached
    for iteration in range(max_iterations):
        # Assign each data point to the closest centroid
        distances = np.zeros((data_points.shape[0], num_clusters))
        for i in range(num_clusters):
            # Calculate the Euclidean distance from each data point to the centroid
            distances[:, i] = np.linalg.norm(data_points - centroids[i, :], axis=1)
        # Assign each data point to the closest centroid
        closest = np.argmin(distances, axis=1)

        # Update the centroids as the mean of the assigned data points
        for i in range(num_clusters):
            centroids[i, :] = np.mean(data_points[closest == i, :], axis=0)

    # Return the final centroids and the assignments of the data points
    return centroids, closest
```

DBSCAN Algorithm without framework

The function named dbscanFunction takes in three arguments: data_points, eps, and min_samples. It implements the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm to cluster the data points.

First, we start computing the pairwise distance matrix between all data points using the Euclidean distance.

```
def dbscanFunction(data_points, eps, min_samples):
    # Compute pairwise distance matrix
    dist_mat = np.zeros((len(data_points), len(data_points)))
    for i in range(len(data_points)):
        for j in range(len(data_points)):
            dist_mat[i, j] = np.linalg.norm(data_points[i] - data_points[j])
```

Then, we identify the core points, which are the data points that have at least **min_samples** other points within distance of **eps**.

```
# Find core points
core_points = []
for i in range(len(data_points)):
    if np.sum(dist_mat[i] <= eps) >= min_samples:
        core_points.append(i)
```

After that we look for the border points which are the ones that have at least one core point within distance of eps.

```
# Find border points
border_points = []
for i in range(len(data_points)):
    if i not in core_points and np.sum(dist_mat[i] <= eps) > 0:
        border_points.append(i)
```

Then, we save the rest of the points as noise or outliers. These are the ones that are neither core nor border points.

```
# Find noise points / outliers
noise_points = []
for i in range(len(data_points)):
    if i not in core_points and i not in border_points:
        noise_points.append(i)
```

With all the points classified, we then start assigning labels. We assign clusters starting from 0 to a core point and its neighbors. The process continues until all core points ant their closest neighbors have a label. Every different label means adding a new cluster.

```
# Assign labels to core points and their neighbors
labels = np.full(len(data_points), -1)
cluster_id = 0
for i in core_points:
    if labels[i] == -1:
        labels[i] = cluster_id
        neighbors = [j for j in range(len(data_points)) if dist_mat[i, j] <= eps]
        for j in neighbors:
            if labels[j] == -1 or labels[j] == 0:
                labels[j] = cluster_id
        cluster_id += 1
```

Then, we label the border points by checking their neighbors' labels, if they belong to the same cluster, the border point is assigned that cluster label.

```
# Assign labels to border points
for i in border_points:
    neighbors = [j for j in range(len(data_points)) if dist_mat[i, j] <= eps]
    neighbor_labels = [labels[j] for j in neighbors]
    unique_labels = set(neighbor_labels)
    if len(unique_labels) == 1:
        labels[i] = neighbor_labels[0]

return labels
```

It returns an array of labels for each data point. In the case of the noise or outliers' points, they stay with the -1 label that we assigned to everyone at the start.

Plotting

This is the function for plotting our results. The function works for 2d and 3d in case we decide to chance the values of the pca function.

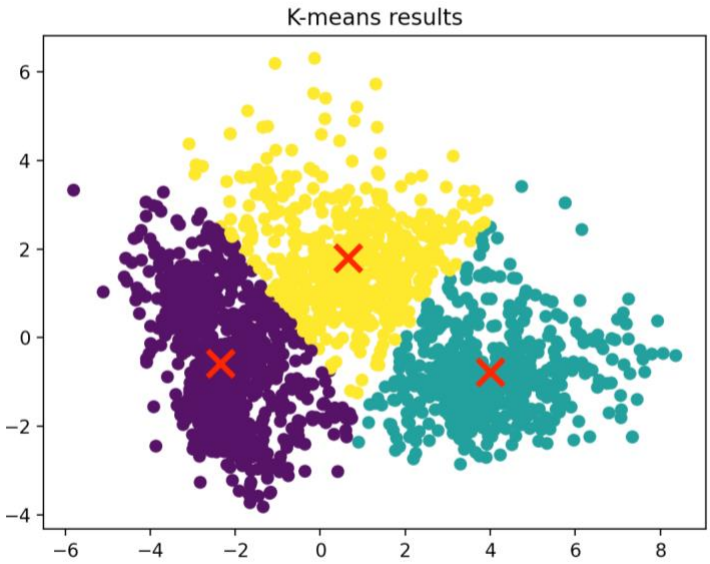
```python
def plot_clusters(data_points, labels, centroids=None, title=""):
    fig = plt.figure()

    if data_points.shape[1] == 2:
        plt.scatter(data_points[:, 0], data_points[:, 1], c=labels)
        if centroids is not None:
            plt.scatter(centroids[:, 0], centroids[:, 1], marker='x', s=200, linewidths=3, color='r')
    elif data_points.shape[1] == 3:
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(data_points[:, 0], data_points[:, 1], data_points[:, 2], c=labels)
        if centroids is not None:
            ax.scatter(centroids[:, 0], centroids[:, 1], centroids[:, 2], marker='x', s=200, linewidths=3, color='r')
    else:
        print("Cannot plot data with more than 3 features")

    plt.title(title)

    if centroids is None:
        # Create a color bar
        cb = plt.colorbar()
        cb.set_ticks(list(set(labels)))
        cb.set_ticklabels(list(range(len(set(labels)))))

    plt.show()
```

Results

After cleaning our data and utilizing the PCA function we end up with a number of 2212 data points. We'll start by having the PCA set to two components to have a 2D plot.

K-means parameters:

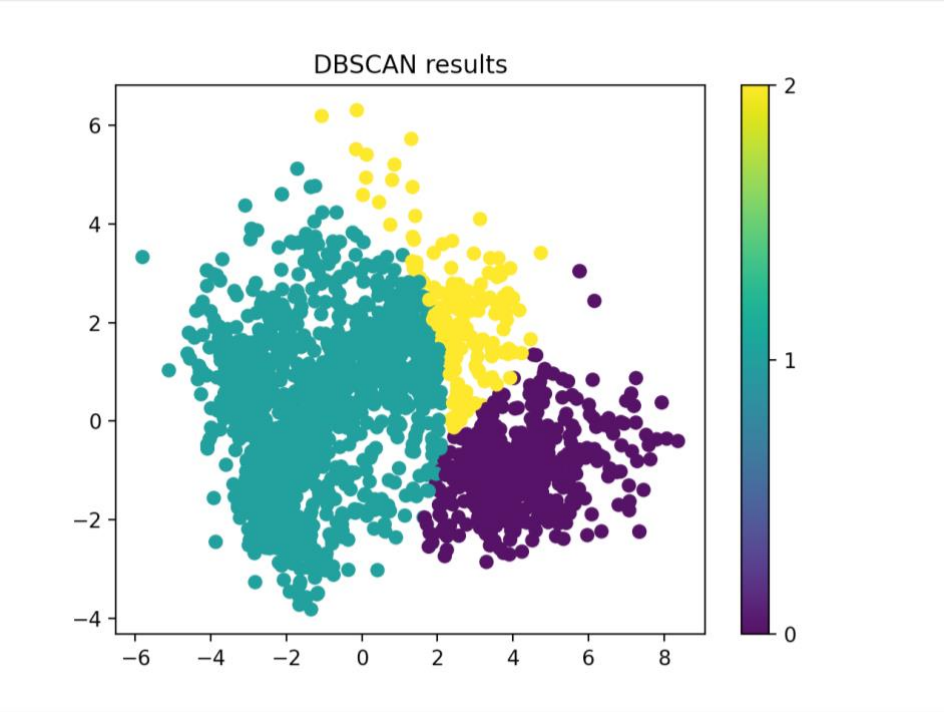
Number of clusters: 3
Maximum number of iterations: 100



DBSCAN parameters:

Radius of the neighborhood, eps: 5

Minimum number of points to form a dense region: 100.

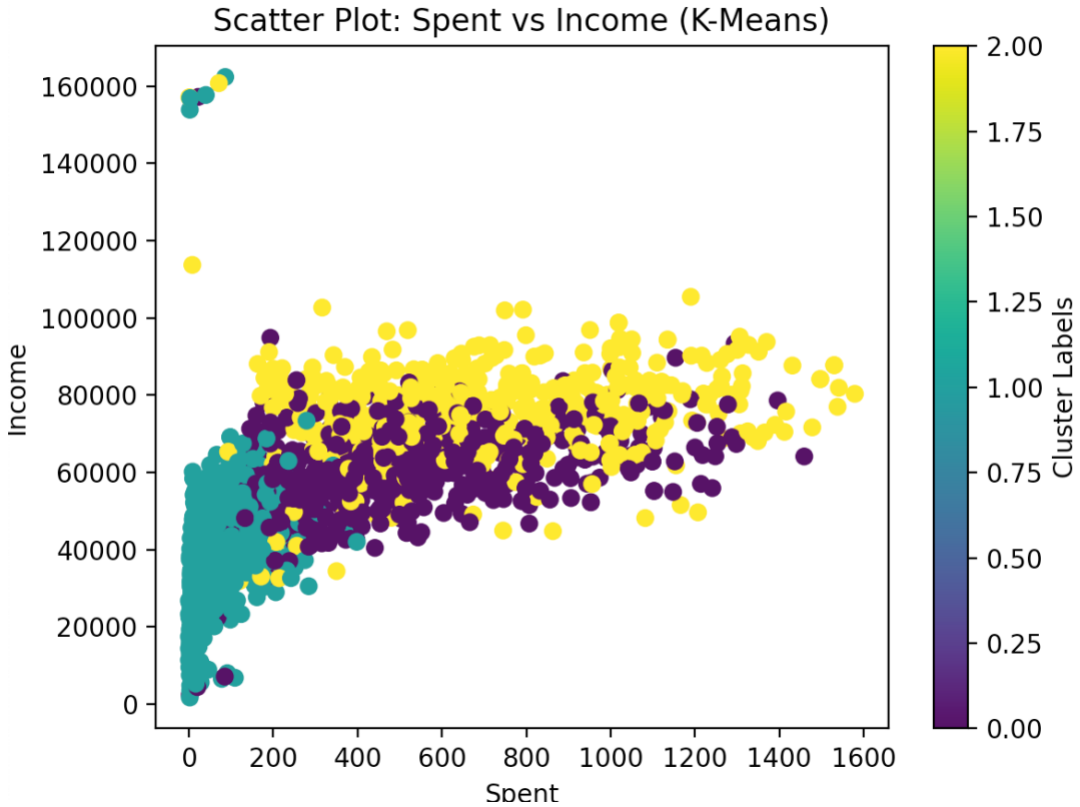


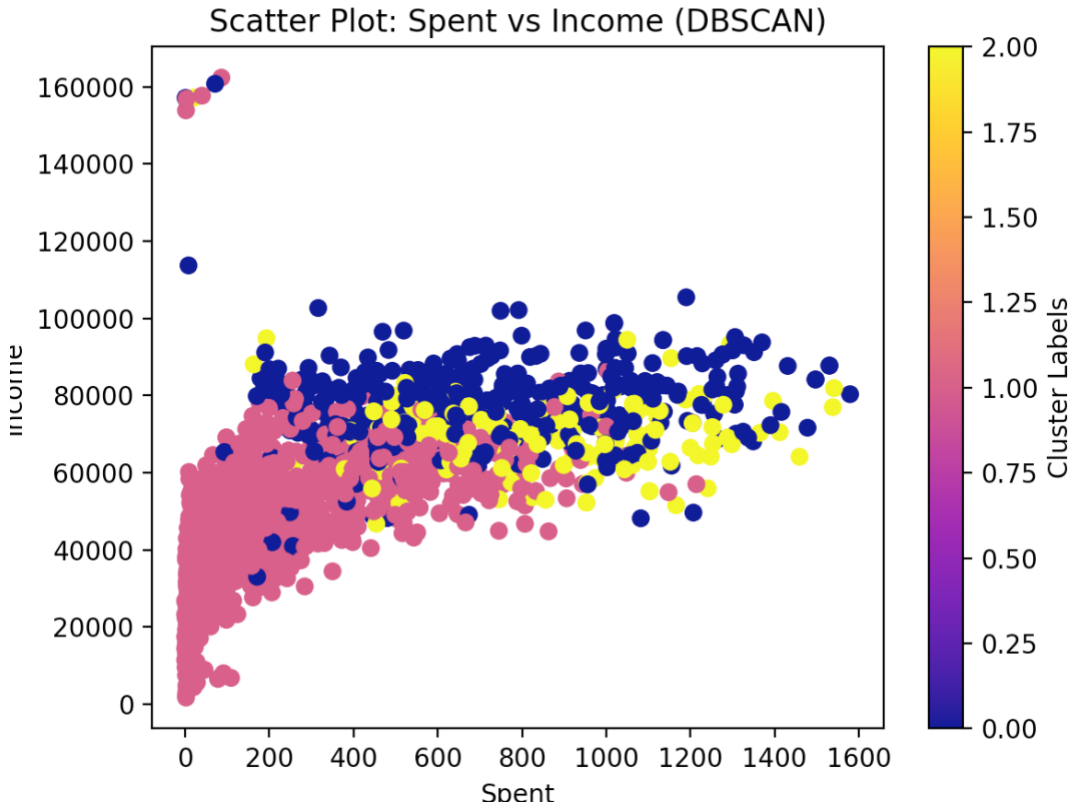
The parameters were chosen after some experimentation to try having the DBSCAN number of clusters to 3 to equal the number chosen for k means. You can try changing the parameters how you see fit by experimentation or using algorithms like the **elbow method** to select the best number of clusters. But since this is an implementation without using the **sklearn** framework we are not going to use it.

Customer Segmentation

Now that we have our clusters labels, we can start the customer segmentation by looking at some patterns inside the dataset.

After adding the labels to the original dataset, I made a scatter plot to see the relation between income and money spent of the customers.





We can notice some differences between the K means clusters and the DBSCAN but in general they both have some very easy to see patterns for each cluster.

Cluster 1: Lower side of income and money spent.
Cluster 2: In the middle of income and middle to high money spent.
Cluster 3: High income and ranges through middle to high money spent.

Number of instances by cluster

K Means Cluster 1: 1079
K Means Cluster 2: 592
K Means Cluster 3: 541

DBSCAN Cluster 1: 1577
DBSCAN Cluster 2: 173
DBSCAN Cluster 3: 462

Conclusion

Both cluster algorithms seem to have delivered similar results that can be used to segment the customers. Some things that can be improved is the number of clusters. Cluster 1 is too big in comparison to the other two and while that doesn`t mean is incorrect we can hope to get a new cluster from that. The elbow method can help to get a better number of clusters for k means.

Extra (3D Results)

Here are the graphs when using the PCA algorithm to 3 components.