Importation des bibliothèques

Cette section est essentielle car elle introduit les bibliothèques nécessaires pour effectuer diverses opérations tout au long du script

- **import numpy as np**: Importe la bibliothèque NumPy, qui est utilisée pour des opérations mathématiques efficaces sur des tableaux.
- **import pandas as pd**: Importe la bibliothèque Pandas, qui est utilisée pour la manipulation des données et l'analyse.
- **import datetime**: Importe le module datetime pour travailler avec des objets de date et d'heure.
- **import matplotlib**, **import matplotlib.pyplot as plt**: Importe la bibliothèque Matplotlib pour la visualisation des données.
- **from matplotlib import colors**: Importe le module colors de Matplotlib pour manipuler les couleurs.
- **import seaborn as sns**: Importe la bibliothèque Seaborn pour la visualisation statistique améliorée.
- **from sklearn.preprocessing import LabelEncoder**: Importe LabelEncoder de scikit-learn pour encoder les données catégorielles en données numériques.
- **from sklearn.preprocessing import StandardScaler**: Importe StandardScaler de scikit-learn pour normaliser les données.
- **from sklearn.decomposition import PCA**: Importe PCA (Principal Component Analysis) de scikit-learn pour la réduction de dimension.
- **from yellowbrick.cluster import KElbowVisualizer**: Importe KElbowVisualizer de Yellowbrick pour visualiser le coude dans la méthode du coude pour le choix du nombre optimal de clusters.
- **from sklearn.cluster import KMeans**: Importe KMeans de scikit-learn pour effectuer le clustering K-means.
- **import matplotlib.pyplot as plt, numpy as np**: Importe à nouveau Matplotlib et NumPy.
- **from mpl_toolkits.mplot3d import Axes3D**: Importe Axes3D de mpl_toolkits.mplot3d pour les graphiques tridimensionnels.
- **from sklearn.cluster import AgglomerativeClustering**: Importe AgglomerativeClustering de scikit-learn pour le clustering hiérarchique.
- **from matplotlib.colors import ListedColormap**: Importe ListedColormap de Matplotlib pour la définition de la colormap.
- **from sklearn import metrics**: Importe metrics de scikit-learn pour l'évaluation des modèles.
- **import warnings**: Importe le module warnings pour gérer les avertissements.
- **import sys**: Importe le module sys pour des opérations système.

```
if not sys.warnoptions:
    warnings.simplefilter("ignore")
```

- Le bloc de code vérifie si les options d'avertissement (**sys.warnoptions**) ne sont pas déjà définies.
- S'il n'y a pas d'options d'avertissement définies, cela signifie que les avertissements ne sont pas gérés, et dans ce cas, le filtre d'avertissement est défini sur "ignore".
- Cela est fait pour supprimer les avertissements pendant l'exécution du code, améliorant ainsi la lisibilité de la sortie.

```
#Loading the dataset

data = pd.read_csv("marketing_campaign.csv", sep="\t")

print("Number of datapoints:", len(data))

data.head()
```

DATA CLEANING

```
# Information on features

data.info()
```

- La méthode **info()** de Pandas est utilisée pour afficher des informations sur le jeu de données, notamment :
  - Le nombre total d'entrées non nulles pour chaque colonne.
  - Les types de données de chaque colonne (entiers, flottants, objets, etc.).
- Ces informations sont utiles pour comprendre la structure du jeu de données et identifier s'il y a des valeurs manquantes.

```
#To remove the NA values

data = data.dropna()
```

- La méthode **dropna()** de Pandas est utilisée pour supprimer toutes les lignes du jeu de données qui contiennent des valeurs manquantes (NaN).
- Cela est souvent fait lorsqu'il y a des valeurs manquantes dans certaines colonnes et que nous voulons travailler uniquement avec les données complètes.

```
data["Dt_Customer"] = pd.to_datetime(data["Dt_Customer"])

dates = []

for i in data["Dt_Customer"]:

    i = i.date()

    dates.append(i)

#Dates of the newest and oldest recorded customer

print("The newest customer's enrolment date in therecords:",max(dates))

print("The oldest customer's enrolment date in the records:",min(dates))
```

- Utilisation de la fonction **pd.to_datetime()** de Pandas pour convertir la colonne "Dt_Customer" en format de date et d'heure.

- Cela permet de traiter la colonne comme un objet de date, facilitant les opérations basées sur la date.
- Utilisation d'une boucle for pour itérer sur chaque élément de la colonne "Dt_Customer".
- La méthode i.date() est utilisée pour extraire uniquement la partie date de chaque timestamp et ajouter cette date à la liste dates.
- Utilisation des fonctions max() et min() pour trouver la date d'enrôlement la plus récente et la plus ancienne dans la liste dates.
- Affichage de ces dates.

```
# Created a feature "Customer_For"

days = []

d1 = max(dates)  # prenant la date d'enrôlement la plus récente

for i in dates:

   delta = d1 - i

   days.append(delta)
```

- Création d'une liste vide appelée **days** pour stocker la différence de jours entre la date d'enrôlement du client le plus récent et la date d'enrôlement de chaque client.
- **d1** est défini comme la date d'enrôlement la plus récente dans le jeu de données.
- Une boucle **for** itère sur chaque date dans la liste **dates** (qui contient les dates d'enrôlement de tous les clients).
- Pour chaque date, la différence en jours entre la date d'enrôlement du client le plus récent (**d1**) et la date actuelle est calculée et ajoutée à la liste **days**.

```
data["Customer_For"] = days

data["Customer_For"] = pd.to_numeric(data["Customer_For"], errors="coerce")
```

- Ajout de la liste **days** en tant que nouvelle colonne "Customer_For" dans le DataFrame **data**.
- Conversion de la colonne "Customer_For" en numérique à l'aide de **pd.to_numeric()**. L'argument **errors="coerce"** permet de traiter toute valeur non numérique comme NaN.

En résumé, cette partie du code crée une nouvelle caractéristique "Customer_For" qui représente la durée depuis la date d'enrôlement du client le plus récent jusqu'à la date d'enrôlement de chaque client, exprimée en jours.

```
print("Total categories in the feature Marital_Status:\n",
data["Marital_Status"].value_counts(), "\n")

print("Total categories in the feature Education:\n", data["Education"].value_counts())
```

- **data["Marital_Status"].value_counts()** renvoie une série qui compte le nombre d'occurrences de chaque catégorie unique dans la colonne "Marital_Status".
- La fonction **print()** affiche le résultat avec une étiquette descriptive.

- De manière similaire, **data["Education"].value_counts()** renvoie une série qui compte le nombre d'occurrences de chaque catégorie unique dans la colonne "Education".
- La fonction **print()** affiche le résultat avec une étiquette descriptive.

```
#Feature Engineering

#Age of customer today

data["Age"] = 2023-data["Year_Birth"]


#Total spendings on various items

data["Spent"] = data["MntWines"]+ data["MntFruits"]+ data["MntMeatProducts"]+
data["MntFishProducts"]+ data["MntSweetProducts"]+ data["MntGoldProds"]


#Deriving living situation by marital status"Alone"

data["Living_With"]=data["Marital_Status"].replace({"Married":"Partner",
"Together":"Partner", "Absurd":"Alone", "Widow":"Alone", "YOLO":"Alone",
"Divorced":"Alone", "Single":"Alone",})


#Feature indicating total children living in the household

data["Children"]=data["Kidhome"]+data["Teenhome"]


#Feature for total members in the householde

data["Family_Size"] = data["Living_With"].replace({"Alone": 1, "Partner":2})+
data["Children"]
```

```
#Feature pertaining parenthood

data["Is_Parent"] = np.where(data.Children> 0, 1, 0)


#Segmenting education levels in three groups

data["Education"]=data["Education"].replace({"Basic":"Undergraduate","2n
Cycle":"Undergraduate", "Graduation":"Graduate", "Master":"Postgraduate",
"PhD":"Postgraduate"})


#For clarity

data=data.rename(columns={"MntWines":
"Wines","MntFruits":"Fruits","MntMeatProducts":"Meat","MntFishProducts":"Fish","MntSwe
etProducts":"Sweets","MntGoldProds":"Gold"})


#Dropping some of the redundant features
```

```
to_drop = ["Marital_Status", "Dt_Customer", "Z_CostContact", "Z_Revenue", "Year_Birth", "ID"]

data = data.drop(to_drop, axis=1)
```

1. **Calcul de l'âge du client :**

```
data["Age"] = 2023 - data["Year_Birth"]
```

- Crée une nouvelle colonne "Age" représentant l'âge du client en soustrayant l'année de naissance de l'année actuelle (2021).

2. **Calcul des dépenses totales sur divers articles :**

```
data["Spent"] = data["MntWines"] + data["MntFruits"] + data["MntMeatProducts"] + data["MntFishProducts"] + data["MntSweetProducts"] + data["MntGoldProds"]
```

- Crée une nouvelle colonne "Spent" représentant la somme des dépenses sur différents types de produits.

3. **Déduction de la situation de vie en fonction de l'état civil :**

```
data["Living_With"] = data["Marital_Status"].replace({"Married": "Partner", "Together": "Partner", "Absurd": "Alone", "Widow": "Alone", "YOLO": "Alone", "Divorced": "Alone", "Single": "Alone",})
```

- Crée une nouvelle colonne "Living_With" pour indiquer si le client vit seul ("Alone") ou avec un partenaire ("Partner").

4. **Calcul du nombre total d'enfants dans le ménage :**

```
data["Children"] = data["Kidhome"] + data["Teenhome"]
```

- Crée une nouvelle colonne "Children" représentant le nombre total d'enfants dans le ménage.

5. **Calcul de la taille de la famille :**

```
data["Family_Size"] = data["Living_With"].replace({"Alone": 1, "Partner": 2}) + data["Children"]
```

- Crée une nouvelle colonne "Family_Size" représentant la taille totale de la famille en ajoutant le nombre de personnes vivant avec le client et le nombre d'enfants.

6. **Caractéristique indiquant si le client est parent :**

```
data["Is_Parent"] = np.where(data.Children > 0, 1, 0)
```

- Crée une nouvelle colonne "Is_Parent" avec la valeur 1 si le client a des enfants et 0 sinon.

7. **Segmentation des niveaux d'éducation en trois groupes :**

```
data["Education"] = data["Education"].replace({"Basic": "Undergraduate", "2n Cycle": "Undergraduate", "Graduation": "Graduate", "Master": "Postgraduate", "PhD": "Postgraduate"})
```

- Modifie les niveaux d'éducation en les regroupant en trois catégories : "Undergraduate", "Graduate", et "Postgraduate".

8. **Renommage de certaines colonnes pour plus de clarté :**

```
data = data.rename(columns={"MntWines": "Wines", "MntFruits": "Fruits",
"MntMeatProducts": "Meat", "MntFishProducts": "Fish", "MntSweetProducts": "Sweets",
"MntGoldProds": "Gold"})
```

- Renomme certaines colonnes pour une meilleure clarté et lisibilité.

9. **Suppression de certaines caractéristiques redondantes :**

```
to_drop = ["Marital_Status", "Dt_Customer", "Z_CostContact", "Z_Revenue", "Year_Birth",
"ID"]

data = data.drop(to_drop, axis=1)
```

- Supprime certaines colonnes qui sont considérées comme redondantes ou inutiles pour l'analyse ultérieure.

En résumé, cette section du code effectue diverses transformations sur les caractéristiques existantes pour créer de nouvelles caractéristiques qui peuvent être plus informatives ou mieux adaptées à l'analyse ultérieure.

```
data.describe()
```

- La méthode **describe()** génère un résumé statistique du DataFrame **data**.
- Le résumé inclut des statistiques telles que la moyenne, l'écart type, les quartiles, la valeur maximale et la valeur minimale pour chaque colonne numérique du DataFrame.

Explications spécifiques :

- "count" : Nombre d'observations non manquantes pour chaque colonne.
- "mean" : Moyenne des valeurs.
- "std" : Écart type, mesurant la dispersion des valeurs.
- "min" : Valeur minimale.
- "25%" : Premier quartile (Q1), médiane de la première moitié des données.
- "50%" : Deuxième quartile (Q2), médiane ou valeur médiane.
- "75%" : Troisième quartile (Q3), médiane de la deuxième moitié des données.
- "max" : Valeur maximale.

Ces statistiques aident à comprendre la distribution et la variabilité des données dans chaque colonne du DataFrame.

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

```
# Setting up colors preferences

sns.set(rc={"axes.facecolor":"#FFF9ED","figure.facecolor":"#FFF9ED"})

pallet = ["#682F2F", "#9E726F", "#D6B2B1", "#B9C0C9", "#9F8A78", "#F3AB60"]
```

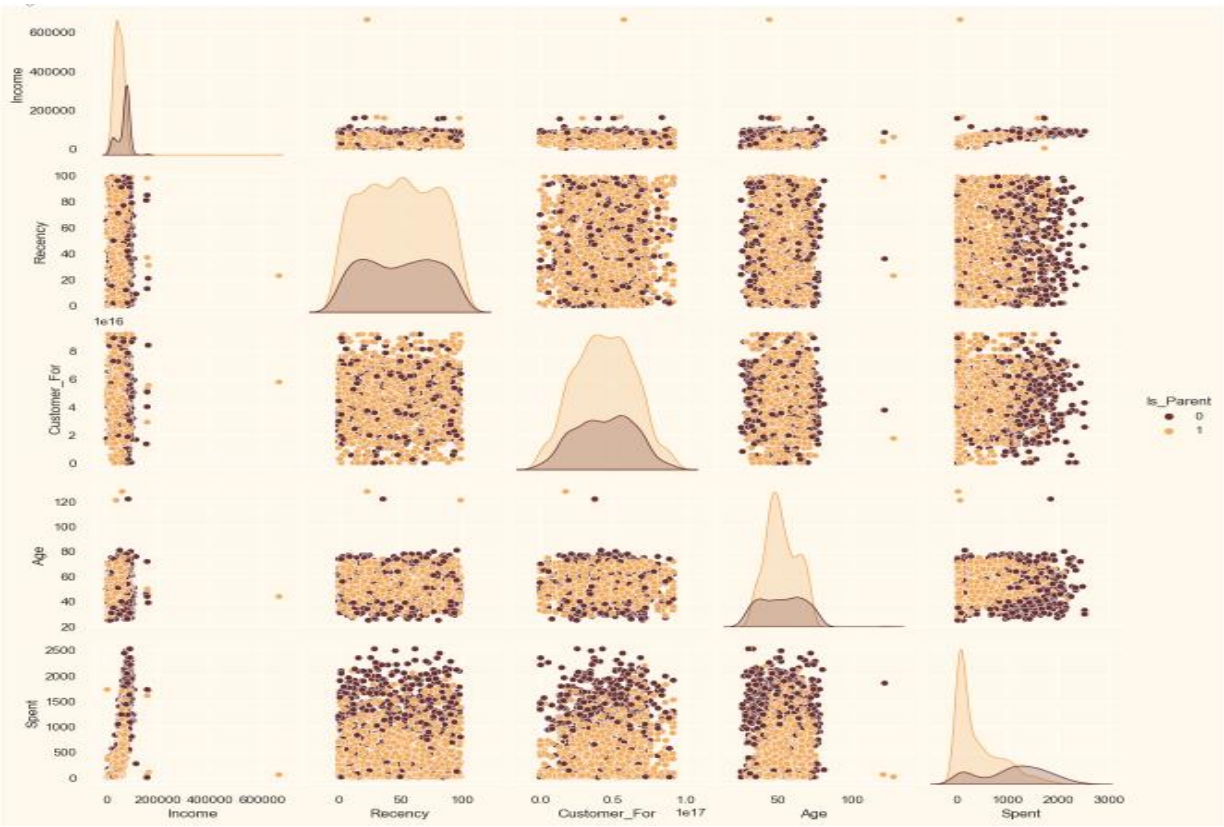
```
cmap = colors.ListedColormap(["#682F2F", "#9E726F", "#D6B2B1", "#B9C0C9", "#9F8A78", "#F3AB60"])


# Plotting following features
To_Plot = [ "Income", "Recency", "Customer_For", "Age", "Spent", "Is_Parent"]
print("Relative Plot Of Some Selected Features: A Data Subset")
plt.figure()
sns.pairplot(data[To_Plot], hue= "Is_Parent", palette= (["#682F2F","#F3AB60"]))
# Taking hue
plt.show()
```

Explication :

- **sns.set(rc={"axes.facecolor":"#FFF9ED","figure.facecolor":"#FFF9ED"})** : Définit les préférences de couleurs pour les graphiques Seaborn, en spécifiant la couleur de fond de l'axe et de la figure.
- **pallet** et **cmap** : Définissent les palettes de couleurs utilisées dans les graphiques.
- **To_Plot** : Liste des caractéristiques à inclure dans les graphiques (Income, Recency, Customer_For, Age, Spent, Is_Parent).
- **sns.pairplot(data[To_Plot], hue= "Is_Parent", palette= (["#682F2F","#F3AB60"]))** : Génère une matrice de graphiques de dispersion pairplot pour les caractéristiques spécifiées dans **To_Plot**. La couleur des points est différenciée en fonction de la caractéristique "Is_Parent".
- **plt.show()** : Affiche les graphiques générés.

En résumé, cette section du code crée un pairplot pour explorer les relations entre les caractéristiques sélectionnées, en utilisant la couleur pour différencier les points en fonction de la caractéristique "Is_Parent". Cela offre une visualisation rapide des tendances et des relations dans les données.




```
# Dropping the outliers by setting a cap on Age and income.

data = data[(data["Age"] < 90)]

data = data[(data["Income"] < 600000)]

print("The total number of data-points after removing the outliers are:", len(data))
```

Explication :

- **data = data[(data["Age"] < 90)]** : Supprime les lignes du DataFrame où l'âge est supérieur ou égal à 90.
- **data = data[(data["Income"] < 600000)]** : Supprime les lignes du DataFrame où le revenu est supérieur ou égal à 600000.
- **print("The total number of data-points after removing the outliers are:", len(data))** : Affiche le nombre total de points de données restants après la suppression des valeurs aberrantes.

En résumé, cette partie du code vise à éliminer les valeurs aberrantes dans les colonnes "Age" et "Income" en fixant des limites spécifiques. Cela peut contribuer à une analyse plus robuste en excluant des observations qui pourraient être des exceptions ou des erreurs dans les données.

```
# Correlation matrix

corrmat = data.corr()


# Set up the size of the figure

plt.figure(figsize=(20, 20))
```

```
# Generate a heatmap of the correlation matrix
sns.heatmap(corrmat, annot=True, cmap=cmap, center=0)
```

Explication :

- **corrmat = data.corr()** : Calcule la matrice de corrélation entre toutes les colonnes numériques du DataFrame.
- **plt.figure(figsize=(20, 20))** : Définit la taille de la figure à 20x20 pouces.
- **sns.heatmap(corrmat, annot=True, cmap=cmap, center=0)** : Génère une heatmap de la matrice de corrélation avec des annotations montrant les valeurs sur chaque cellule. La couleur est déterminée par la colormap (**cmap**) et le point central (**center**) est fixé à 0.

La matrice de corrélation et la heatmap sont utiles pour identifier les relations linéaires entre les paires de caractéristiques. Les valeurs proches de 1 indiquent une corrélation positive, proche de -1 indique une corrélation négative, et une valeur proche de 0 indique une faible corrélation.

En résumé, cette section permet de visualiser graphiquement les corrélations entre les différentes caractéristiques du DataFrame.

DATA PREPROCESSING

```
#Get list of categorical variables
s = (data.dtypes == 'object')
object_cols = list(s[s].index)


print("Categorical variables in the dataset:", object_cols)
```

Explication :

- **data.dtypes** : Retourne le type de données de chaque colonne du DataFrame.
- **s = (data.dtypes == 'object')** : Crée une série booléenne indiquant True pour les colonnes ayant le type 'object' (c'est-à-dire les variables catégorielles) et False pour les autres types.
- **object_cols = list(s[s].index)** : Convertit les indices (noms de colonnes) des colonnes catégorielles en une liste **object_cols**.
- **print("Categorical variables in the dataset:", object_cols)** : Affiche la liste des variables catégorielles dans le dataset.

```
# Label Encoding the object dtypes.
LE = LabelEncoder()


# Iterate through each categorical column and apply label encoding
for i in object_cols:
    data[i] = data[[i]].apply(LE.fit_transform)
```

```
# Print a message indicating that all features are now numerical
print("All features are now numerical")
```

Explication :

- **LE = LabelEncoder()** : Crée une instance de la classe LabelEncoder de scikit-learn, qui est utilisée pour effectuer l'encodage des étiquettes.
- La boucle **for i in object_cols:** itère à travers chaque colonne catégorielle identifiée précédemment.
- **data[i] = data[[i]].apply(LE.fit_transform)** : Applique l'encodage des étiquettes à la colonne catégorielle **i** à l'aide de **fit_transform** de l'objet LabelEncoder. Cela remplace les valeurs catégorielles par des valeurs numériques.
- **print("All features are now numerical")** : Affiche un message indiquant que toutes les caractéristiques sont maintenant numériques après l'encodage des étiquettes.

En résumé, cette section du code transforme les variables catégorielles en variables numériques en utilisant l'encodage des étiquettes, ce qui est nécessaire pour l'entrée dans de nombreux modèles d'apprentissage automatique qui exigent des données numériques.

```
# Creating a copy of data
ds = data.copy()


# Creating a subset of the dataframe by dropping the features on deals accepted and promotions
cols_del = ['AcceptedCmp3', 'AcceptedCmp4', 'AcceptedCmp5', 'AcceptedCmp1', 'AcceptedCmp2', 'Complain', 'Response']
ds = ds.drop(cols_del, axis=1)


# Scaling
scaler = StandardScaler()
scaler.fit(ds)
scaled_ds = pd.DataFrame(scaler.transform(ds), columns=ds.columns)
print("All features are now scaled")
```

Explication :

- **ds = data.copy()** : Crée une copie du DataFrame **data** appelée **ds**. Cela permet de travailler sur une copie tout en préservant le DataFrame original.
- **cols_del = ['AcceptedCmp3', 'AcceptedCmp4', 'AcceptedCmp5', 'AcceptedCmp1', 'AcceptedCmp2', 'Complain', 'Response']** : Liste des colonnes à exclure du sous-ensemble.
- **ds = ds.drop(cols_del, axis=1)** : Supprime les colonnes spécifiées dans **cols_del** du DataFrame **ds** pour créer un sous-ensemble.

- **scaler = StandardScaler()** : Crée une instance de l'objet **StandardScaler** de scikit-learn, qui sera utilisé pour mettre à l'échelle les caractéristiques.
- **scaler.fit(ds)** : Calcule la moyenne et l'écart-type des caractéristiques dans **ds** pour normaliser les données.
- **scaled_ds = pd.DataFrame(scaler.transform(ds), columns=ds.columns)** : Applique la mise à l'échelle aux caractéristiques du DataFrame **ds** et crée un nouveau DataFrame appelé **scaled_ds**.
- **print("All features are now scaled")** : Affiche un message indiquant que toutes les caractéristiques sont maintenant mises à l'échelle.

En résumé, cette section du code crée une copie du DataFrame, exclut certaines colonnes, puis met à l'échelle les caractéristiques du DataFrame résultant en utilisant la normalisation standard. La normalisation est souvent nécessaire pour garantir que les caractéristiques sont à des échelles comparables lors de l'utilisation de certains algorithmes d'apprentissage automatique.

```
# Scaled data to be used for reducing the dimensionality

print("Dataframe to be used for further modelling:")

scaled_ds.head()
```

Explication :

- **print("Dataframe to be used for further modelling:")** : Affiche le message indiquant que le DataFrame suivant sera utilisé pour la modélisation ultérieure.
- **scaled_ds.head()** : Affiche les premières lignes du DataFrame mis à l'échelle (**scaled_ds**). Cela donne un aperçu des données après la mise à l'échelle.

En résumé, cette section du code vise à montrer les premières lignes du DataFrame mis à l'échelle, permettant ainsi de vérifier visuellement comment la mise à l'échelle a affecté les valeurs des caractéristiques.

DIMENSIONALITY REDUCTION

```
# Initiating PCA to reduce dimensions aka features to 3

pca = PCA(n_components=3)

pca.fit(scaled_ds)

PCA_ds = pd.DataFrame(pca.transform(scaled_ds), columns=["col1", "col2", "col3"])

PCA_ds.describe().T
```

Explication :

- **pca = PCA(n_components=3)** : Crée une instance de l'objet PCA avec **n_components=3**, spécifiant que nous voulons réduire la dimensionnalité à trois dimensions.
- **pca.fit(scaled_ds)** : Ajuste le modèle PCA aux données mises à l'échelle **scaled_ds**.
- **PCA_ds = pd.DataFrame(pca.transform(scaled_ds), columns=["col1", "col2", "col3"])** : Transforme les données mises à l'échelle en trois nouvelles dimensions

(composantes principales) et les stocke dans un nouveau DataFrame appelé **PCA_ds** avec les colonnes "col1", "col2", et "col3".

- **PCA_ds.describe().T** : Affiche les statistiques descriptives (moyenne, écart type, etc.) des composantes principales. La transposition (**T**) est utilisée pour afficher les statistiques dans un format plus lisible.

En résumé, cette section du code utilise l'analyse en composantes principales (PCA) pour réduire la dimensionnalité des données à trois dimensions, puis affiche les statistiques descriptives de ces nouvelles dimensions. La PCA est couramment utilisée pour réduire la dimensionnalité tout en préservant autant d'information que possible.

```
# A 3D Projection Of Data In The Reduced Dimension

x = PCA_ds["col1"]

y = PCA_ds["col2"]

z = PCA_ds["col3"]


# To plot

fig = plt.figure(figsize=(10, 8))

ax = fig.add_subplot(111, projection="3d")

ax.scatter(x, y, z, c="maroon", marker="o")

ax.set_title("A 3D Projection Of Data In The Reduced Dimension")

plt.show()
```

Explication :

- **x = PCA_ds["col1"]**, **y = PCA_ds["col2"]**, **z = PCA_ds["col3"]** : Extraient les valeurs des composantes principales du DataFrame **PCA_ds** pour les axes x, y et z respectivement.
- **fig = plt.figure(figsize=(10, 8))** : Crée une nouvelle figure avec une taille de 10x8 pouces.
- **ax = fig.add_subplot(111, projection="3d")** : Ajoute un sous-graphique 3D à la figure.
- **ax.scatter(x, y, z, c="maroon", marker="o")** : Crée un nuage de points 3D avec les composantes principales comme coordonnées, la couleur "maroon" et la forme du marqueur "o".
- **ax.set_title("A 3D Projection Of Data In The Reduced Dimension")** : Ajoute un titre à l'axe des z.
- **plt.show()** : Affiche le graphique.

En résumé, cette section du code produit une visualisation 3D des données dans les dimensions réduites après l'application de l'analyse en composantes principales (PCA). Chaque point dans le nuage de points représente une observation dans les nouvelles dimensions.

CLUSTERING

```
# Quick examination of elbow method to find the number of clusters to be formed.

print('Elbow Method to determine the number of clusters to be formed:')

Elbow_M = KElbowVisualizer(KMeans(), k=10)

Elbow_M.fit(PCA_ds)

Elbow_M.show()
```

Explication :

- **Elbow_M = KElbowVisualizer(KMeans(), k=10)** : Crée une instance de l'objet **KElbowVisualizer** de la bibliothèque Yellowbrick, qui utilise la méthode du coude pour trouver le nombre optimal de clusters. **KMeans()** est utilisé comme estimateur, et **k=10** spécifie le nombre maximal de clusters à considérer.
- **Elbow_M.fit(PCA_ds)** : Ajuste le modèle à l'aide des données en dimensions réduites (**PCA_ds**).
- **Elbow_M.show()** : Affiche la visualisation du coude, qui est un graphique montrant l'inertie (somme des carrés des distances entre les points et leur centre de cluster) pour différents nombres de clusters.

En résumé, cette section du code utilise la méthode du coude pour visualiser l'inertie en fonction du nombre de clusters et aide à déterminer un nombre approprié de clusters à utiliser dans l'algorithme de KMeans. La visualisation permet de repérer le point où l'ajout de clusters supplémentaires n'améliore pas significativement l'inertie, formant ainsi un coude dans le graphique. Le nombre de clusters correspondant au coude est souvent choisi comme le nombre optimal de clusters.

```
# Initiating the Agglomerative Clustering model

AC = AgglomerativeClustering(n_clusters=4)


# Fit model and predict clusters

yhat_AC = AC.fit_predict(PCA_ds)


# Creating a DataFrame with the clustered data

PCA_ds["Clusters"] = yhat_AC


# Adding the Clusters feature to the original dataframe

data["Clusters"] = yhat_AC
```

Cette partie du code utilise le modèle de regroupement hiérarchique agglomératif (Agglomerative Clustering) pour attribuer des clusters aux observations basées sur les composantes principales obtenues après l'analyse en composantes principales (PCA).

Explication :

- **AC = AgglomerativeClustering(n_clusters=4)** : Crée une instance du modèle de regroupement hiérarchique agglomératif avec **n_clusters=4**, spécifiant le nombre de clusters souhaité.
- **yhat_AC = AC.fit_predict(PCA_ds)** : Ajuste le modèle aux données en dimensions réduites (**PCA_ds**) et prédit les clusters pour chaque observation.
- **PCA_ds["Clusters"] = yhat_AC** : Ajoute une colonne "Clusters" au DataFrame **PCA_ds** contenant les étiquettes de cluster prédites.
- **data["Clusters"] = yhat_AC** : Ajoute également une colonne "Clusters" au DataFrame original **data** avec les mêmes étiquettes de cluster. Cela permet d'associer les clusters au jeu de données original.

En résumé, cette section du code applique le modèle de regroupement hiérarchique agglomératif avec 4 clusters aux données en dimensions réduites (composantes principales) obtenues après l'analyse en composantes principales (PCA). Les étiquettes de cluster sont ensuite ajoutées aux DataFrames **PCA_ds** et **data**.

```
# Plotting the clusters

fig = plt.figure(figsize=(10, 8))

ax = plt.subplot(111, projection='3d', label="bla")

ax.scatter(x, y, z, s=40, c=PCA_ds["Clusters"], marker='o', cmap=cmap)

ax.set_title("The Plot Of The Clusters")

plt.show()
```

Explication :

- **fig = plt.figure(figsize=(10, 8))** : Crée une nouvelle figure avec une taille de 10x8 pouces.
- **ax = plt.subplot(111, projection='3d', label="bla")** : Ajoute un sous-graphique 3D à la figure.
- **ax.scatter(x, y, z, s=40, c=PCA_ds["Clusters"], marker='o', cmap=cmap)** : Crée un nuage de points 3D où chaque point représente une observation, les coordonnées x, y et z étant basées sur les composantes principales, la taille des points est définie par **s=40**, la couleur est déterminée par les étiquettes de cluster (**PCA_ds["Clusters"]**), le marqueur est "o" et la colormap (**cmap**) est définie par la palette de couleurs spécifiée précédemment.
- **ax.set_title("The Plot Of The Clusters")** : Ajoute un titre à l'axe des z.
- **plt.show()** : Affiche le graphique.

En résumé, cette section du code génère une visualisation 3D montrant la répartition des clusters dans les dimensions réduites obtenues après l'analyse en composantes principales (PCA). Chaque cluster est représenté par une couleur différente.

ÉVALUATION DES MODÈLES

```
# Plotting countplot of clusters

pal = ["#682F2F", "#B9C0C9", "#9F8A78", "#F3AB60"]
```

```
pl = sns.countplot(x=data["Clusters"], palette=pal)

pl.set_title("Distribution Of The Clusters")

plt.show()
```

Explication :

- **pal = ["#682F2F", "#B9C0C9", "#9F8A78", "#F3AB60"]** : Définit une palette de couleurs à utiliser dans le countplot, associant une couleur à chaque cluster.
- **pl = sns.countplot(x=data["Clusters"], palette=pal)** : Crée un countplot où l'axe x représente les clusters (**data["Clusters"]**). Chaque barre du countplot représente le nombre d'observations dans chaque cluster, et la couleur de la barre est déterminée par la palette spécifiée.
- **pl.set_title("Distribution Of The Clusters")** : Ajoute un titre au countplot.
- **plt.show()** : Affiche le graphique.

En résumé, cette section du code génère un countplot pour montrer visuellement la distribution des observations dans chaque cluster. Chaque barre représente le nombre d'observations dans un cluster spécifique, avec des couleurs différentes pour chaque cluster.

Cette partie du code génère un scatterplot pour visualiser la relation entre les dépenses (**Spent**) et le revenu (**Income**) en fonction des clusters. Chaque point dans le scatterplot représente une observation, et la couleur du point est déterminée par l'appartenance au cluster. Voici une explication détaillée :

```
pl = sns.scatterplot(data=data, x=data["Spent"], y=data["Income"],
hue=data["Clusters"], palette=pal)

pl.set_title("Cluster's Profile Based On Income And Spending")

plt.legend()

plt.show()
```

Explication :

- **pl = sns.scatterplot(data=data, x=data["Spent"], y=data["Income"], hue=data["Clusters"], palette=pal)** : Crée un scatterplot avec les dépenses (**x=data["Spent"]**) sur l'axe x, le revenu (**y=data["Income"]**) sur l'axe y, la couleur des points déterminée par l'appartenance au cluster (**hue=data["Clusters"]**), et une palette de couleurs spécifiée (**palette=pal**).
- **pl.set_title("Cluster's Profile Based On Income And Spending")** : Ajoute un titre au scatterplot.
- **plt.legend()** : Ajoute une légende pour indiquer quelle couleur correspond à chaque cluster.
- **plt.show()** : Affiche le graphique.

En résumé, cette section du code génère un scatterplot pour illustrer la relation entre les dépenses et le revenu, en différenciant les observations par cluster à l'aide de couleurs différentes. La légende indique quelle couleur correspond à chaque cluster.

```
# Income vs spending plot shows the clusters pattern
```

```
# group 0: high spending & average income
# group 1: high spending & high income
# group 2: low spending & low income
# group 3: high spending & low income
```

Cette partie du code crée un swarmplot et un boxenplot pour visualiser la distribution des dépenses (**Spent**) dans chaque cluster. Voici une explication détaillée :

```
plt.figure()
pl = sns.swarmplot(x=data["Clusters"], y=data["Spent"], color="#CBEDDD", alpha=0.5)
pl = sns.boxenplot(x=data["Clusters"], y=data["Spent"], palette=pal)
plt.show()
```

Explication :

- **plt.figure()** : Crée une nouvelle figure.
- **pl = sns.swarmplot(x=data["Clusters"], y=data["Spent"], color="#CBEDDD", alpha=0.5)** : Crée un swarmplot où les points représentent les observations, positionnés le long de l'axe x en fonction de leur cluster (**x=data["Clusters"]**), et l'axe y représente les dépenses (**y=data["Spent"]**). La couleur des points est définie par "#CBEDDD", et **alpha=0.5** contrôle la transparence des points.
- **pl = sns.boxenplot(x=data["Clusters"], y=data["Spent"], palette=pal)** : Ajoute également un boxenplot pour chaque cluster, montrant la distribution des dépenses dans chaque cluster. La palette de couleurs est spécifiée par **pal**.
- **plt.show()** : Affiche le graphique.

En résumé, cette section du code génère un swarmplot et un boxenplot pour visualiser la distribution des dépenses dans chaque cluster. Le swarmplot montre la répartition des points individuels, tandis que le boxenplot offre une vue plus détaillée de la distribution des données dans chaque cluster.

Cette partie du code crée une nouvelle fonctionnalité appelée "Total_Promos" qui représente la somme des promotions acceptées pour chaque client, en additionnant les valeurs des colonnes "AcceptedCmp1" à "AcceptedCmp5". Ensuite, un countplot est généré pour visualiser le nombre total de promotions acceptées, différencié par cluster. Voici une explication détaillée :

```
# Creating a feature to get a sum of accepted promotions

data["Total_Promos"] = data["AcceptedCmp1"] + data["AcceptedCmp2"] +
data["AcceptedCmp3"] + data["AcceptedCmp4"] + data["AcceptedCmp5"]



# Plotting count of total campaign accepted.

plt.figure()

pl = sns.countplot(x=data["Total_Promos"], hue=data["Clusters"], palette=pal)

pl.set_title("Count Of Promotion Accepted")

pl.set_xlabel("Number Of Total Accepted Promotions")

plt.show()
```

Explication :

- **data["Total_Promos"] = data["AcceptedCmp1"] + data["AcceptedCmp2"] + data["AcceptedCmp3"] + data["AcceptedCmp4"] + data["AcceptedCmp5"]** : Crée une nouvelle colonne "Total_Promos" dans le DataFrame, qui représente la somme des promotions acceptées pour chaque client.
- **plt.figure()** : Crée une nouvelle figure.
- **pl = sns.countplot(x=data["Total_Promos"], hue=data["Clusters"], palette=pal)** : Crée un countplot où l'axe x représente le nombre total de promotions acceptées (**x=data["Total_Promos"]**), différencié par cluster (**hue=data["Clusters"]**). La palette de couleurs est spécifiée par **pal**.
- **pl.set_title("Count Of Promotion Accepted")** : Ajoute un titre au countplot.
- **pl.set_xlabel("Number Of Total Accepted Promotions")** : Ajoute une étiquette à l'axe des x.
- **plt.show()** : Affiche le graphique.

En résumé, cette section du code génère un countplot pour visualiser le nombre total de promotions acceptées, avec une distinction par cluster. Cela permet d'observer la répartition des clients en fonction de leur participation aux promotions.

```
# Plotting the number of deals purchased

plt.figure()

pl = sns.boxenplot(y=data["NumDealsPurchases"], x=data["Clusters"], palette=pal)

pl.set_title("Number of Deals Purchased")

plt.show()
```

Explication :

- **plt.figure()** : Crée une nouvelle figure.
- **pl = sns.boxenplot(y=data["NumDealsPurchases"], x=data["Clusters"], palette=pal)** : Crée un boxenplot où l'axe y représente le nombre d'achats de "deals" (**y=data["NumDealsPurchases"]**), et l'axe x représente les clusters (**x=data["Clusters"]**). La palette de couleurs est spécifiée par **pal**.
- **pl.set_title("Number of Deals Purchased")** : Ajoute un titre au boxenplot.
- **plt.show()** : Affiche le graphique.

En résumé, cette section du code génère un boxenplot pour visualiser la distribution du nombre d'achats de "deals" dans chaque cluster. Le boxenplot offre une vue détaillée de la distribution des données, montrant la médiane, les quartiles et la dispersion des valeurs pour chaque cluster.

Cette partie du code génère des graphiques de type jointplot (distribution bivariée) pour explorer la relation entre différentes variables personnelles (telles que "Kidhome", "Teenhome", "Customer_For", "Age", "Children", "Family_Size", "Is_Parent", "Education", "Living_With") et les dépenses ("Spent"). Ces graphiques sont différenciés par cluster grâce à la couleur. Voici une explication détaillée :

```
Personal = ["Kidhome", "Teenhome", "Customer_For", "Age", "Children", "Family_Size", "Is_Parent", "Education", "Living_With"]


for i in Personal:
    plt.figure()
    sns.jointplot(x=data[i], y=data["Spent"], hue=data["Clusters"], kind="kde", palette=pal)
    plt.show()
```

Explication :

- **Personal = ["Kidhome", "Teenhome", "Customer_For", "Age", "Children", "Family_Size", "Is_Parent", "Education", "Living_With"]** : Liste des variables personnelles à explorer.
- La boucle **for i in Personal:** parcourt chaque variable personnelle.
    - **plt.figure()** : Crée une nouvelle figure pour chaque graphique jointplot.
    - **sns.jointplot(x=data[i], y=data["Spent"], hue=data["Clusters"], kind="kde", palette=pal)** : Crée un graphique jointplot où l'axe x représente la variable personnelle (**x=data[i]**), l'axe y représente les dépenses (**y=data["Spent"]**), la couleur est différenciée par cluster (**hue=data["Clusters"]**), et le type de tracé est une estimation de la densité kernel (kind="kde"). La palette de couleurs est spécifiée par **pal**.
    - **plt.show()** : Affiche chaque graphique jointplot.

En résumé, cette section du code génère une série de graphiques jointplot pour explorer la relation entre différentes variables personnelles et les dépenses, en différenciant les observations par cluster à l'aide de couleurs différentes. Les graphiques utilisent une estimation de la densité kernel pour visualiser la distribution bivariée.