Appendix

IMPORT NECESSARY LIBRARIES

In [ ]:
```
# Import Necessary Libraries

from datetime import datetime, timedelta, date
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from __future__ import division
import seaborn as sns
import plotly.offline as pyoff
import plotly.graph_objs as go
import plotly.express as px
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.metrics import RocCurveDisplay
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report
from sklearn.preprocessing import label_binarize
import matplotlib.pyplot as plt
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle
```

EXPLORATORY ANALYSIS

In [ ]:
```
# Import data
import pandas as pd

# Read excel file
df_retail = pd.read_csv('C:/Users/Darius/Desktop/PROJECT//online_retail_II.csv')
print(df_retail.head())
df_retail = df_retail.rename(columns={'Customer ID': 'CustomerID'})
```

In [ ]:
```
# Check data info
df_retail.info()
```

In [ ]:
```
# Display the first 5 rows of the 'df_retail' DataFrame
df_retail.head(5)
```

In [ ]:
```
# Convert 'InvoiceDate' column to datetime format
# and display information about the 'df_retail' DataFrame
df_retail['InvoiceDate'] = pd.to_datetime(df_retail['InvoiceDate'])
df_retail.info()
```

In [ ]:
```
# Extract the year and month from the first entry in the 'InvoiceDate' column of the 'df_retail' DataFrame
df_retail['InvoiceDate'][0].year
df_retail['InvoiceDate'][0].month
```

How many countries and its distribution in the dataset

In [ ]:
```
# Group the data by country and count the occurrences
df_country_counts = df_retail.groupby('Country')['Invoice'].count().reset_index(name='Count')

# Create the pie chart using plotly
fig = px.pie(df_country_counts, values='Count', names='Country',
             title='Distribution of Countries', hole=.0)

# Customize the layout
fig.update_traces(textposition='inside', textinfo='percent+label')
fig.update_layout(legend=dict(orientation="h", yanchor="bottom", y=1.05, xanchor="right", x=1),
                  margin=dict(l=0, r=0, t=10, b=0),
                  title=dict(text='Distribution of Sales by Countries', font=dict(size=20), x=0, y=0.5))
# Display the chart
fig.show()
```

Filter data for 01/12/2010 to 01/12/2011

In [ ]:
```
# filter the data
start_date = pd.to_datetime('2010-12-01', format='%Y-%m-%d')
end_date = pd.to_datetime('2011-12-01', format='%Y-%m-%d')
df_retail = df_retail[(df_retail['InvoiceDate'] >= start_date) & (df_retail['InvoiceDate'] < end_date)]
```

In [ ]:
```
df_retail.info()
```


In [ ]:
```
# Create a new column 'InvoiceYearMonth' by applying a lambda function to 'InvoiceDate'
df_retail['InvoiceYearMonth'] = df_retail['InvoiceDate'].map(lambda x: x.year*100 + x.month)
```

In [ ]:
```
# Display the first 5 rows of the 'df_retail' DataFrame
df_retail.head(5)
```

Calculate Revenue from UK

In [ ]:
```
# Calculate monthly revenue by multiplying Quantity and Price
df_retail['Revenue'] = df_retail['Quantity'] * df_retail['Price']

# then grouping by InvoiceYearMonth
df_revenue = df_retail.groupby('InvoiceYearMonth')['Revenue'].sum().reset_index()

# Calculate the monthly active customers for the UK
# First, filter the data to include only UK customers
df_uk = df_retail[df_retail['Country'] == 'United Kingdom'].reset_index(drop = True)
```

Total Revenue by month in the UK

In [ ]:
```
fig.update_layout(
    xaxis_title='Month',
    yaxis_title='Revenue',
    xaxis=dict(
        tickmode='array',
        tickformat='%b',
        tickfont=dict(size=18),
        title=dict(text='Month', font=dict(size=20))
    ),
    yaxis=dict(
        tickfont=dict(size=18),
        title=dict(text='Revenue', font=dict(size=20))
    ),
    title=dict(
        text='Monthly Revenue and Total Revenue',
        font=dict(size=24)
    )
)
```

Monthly Growth Rate = Tang truong doanh thu thang sau so voi thang truoc

In [ ]:
```
# Monthly Growth Rate
df_revenue['pct_change'] = df_revenue['Revenue'].pct_change()
df_revenue
```

In [ ]:
```
import plotly.graph_objs as go
import plotly.offline as pyoff
# Create a line chart showing the monthly growth rate of revenue using Plotly

# Define data for the line chart
plot_data = [
    go.Scatter(
        x=df_revenue[df_revenue['InvoiceYearMonth'] != '2011-12']['InvoiceYearMonth'],
        y=df_revenue[df_revenue['pct_change'] != '2011-12']['pct_change'],
        mode='lines+markers',
        marker=dict(
            color='blue',
            size=14,
            ),
        name='Monthly Growth Rate',
        text=[f'{pct:.2%}' for pct in df_revenue[df_revenue['pct_change'] != '2011-12']['pct_change']], # Add percentage value t
        textposition='top center',
        textfont=dict(
            size=18,
            ),
        hovertemplate='%{x}<br>Growth Rate: %{y:.2f}%<extra></extra>',
        )
]
# Define the layout for the line chart
plot_layout = go.Layout(
    title='Monthly Growth Rate in 2011',
    xaxis=dict(
        type='category',
        title='Year-Month',
        tickangle=-45,
        tickfont=dict(size=24),
        showticklabels=True,
        ),
    yaxis=dict(
        title='Growth Rate',
```



```
        tickformat='.2%',
        tickfont=dict(size=24),
        ),
    font=dict(size=36),
)
# Create the line chart figure
fig = go.Figure(data=plot_data, layout=plot_layout)

# Display the line chart
pyoff.plot(fig)
```

Monthly Active Customers = So luong Customers hoat dong hang thang

In [ ]:
```
# Monthly Active Customers
# Then, group the data by InvoiceYearMonth and count the number of unique CustomerIDs in each group
df_monthly_active = df_uk.groupby('InvoiceYearMonth')['CustomerID'].nunique().reset_index()
df_monthly_active
```

In [ ]:
```
# Plot Monthly Active User
plot_data = [
    go.Bar(
        x = df_monthly_active['InvoiceYearMonth'],
        y = df_monthly_active['CustomerID'],
    )
]

plot_layout = go.Layout(
    xaxis = {"type" : "category", "title": {"text": "Invoice Year Month", "font": {"size": 28}}, "tickfont": {"size": 24}},
    yaxis = {"title": {"text": "Monthly Active Customers", "font": {"size": 28}}, "tickfont": {"size": 24}},
    title = {"text": "Monthly Active Customers in 2011", "font": {"size": 40}}
)

fig = go.Figure( data = plot_data, layout = plot_layout)
pyoff.plot(fig)
```

Average Quantity of Product sold monthly

In [ ]:
```
# Calculate the total quantity of products sold each month for the UK
df_monthly_sales = df_uk.groupby('InvoiceYearMonth')['Quantity'].sum().reset_index()
```

```
df_monthly_sales.head(5)
```

In [ ]:
```
# Plot Quantity of Product sold monthly
plot_data = [
    go.Bar(
        x=df_monthly_sales['InvoiceYearMonth'],
        y=df_monthly_sales['Quantity'],
    )
]

plot_layout = go.Layout(
    xaxis=dict(
        type="category",
        tickfont=dict(size=28),
        title_font=dict(size=32),
        title="Month"
    ),
    yaxis=dict(
        tickfont=dict(size=28),
        title_font=dict(size=32),
        title="Quantity"
    ),
    title=dict(text="Monthly Total of Order in 2011", font=dict(size=40)),
)

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.plot(fig)
```

In [ ]:
```
# Calculate the average monthly sales quantity
df_monthly_sales.mean()
```

New and Existing Customers

In [ ]:
```
# Determine if each customer is new or existing based on their first purchase date
# Create a DataFrame with the first purchase date for each customer
df_min_purchase = df_uk.groupby('CustomerID').InvoiceDate.min().reset_index()
df_min_purchase.columns = ['CustomerID', 'MinPurchaseDate']
df_min_purchase['MinPurchaseYearMonth'] = df_min_purchase['MinPurchaseDate'].map(lambda x: x.year*100 + x.month)

# Merge the first purchase date DataFrame with the main DataFrame to add UserType column
df_uk = pd.merge(df_uk, df_min_purchase, on='CustomerID')
```



```
df_uk['UserType'] = 'New'
df_uk.loc[df_uk['InvoiceYearMonth'] != df_uk['MinPurchaseYearMonth'], 'UserType'] = 'Existing'

# Show the first 10 rows of the modified DataFrame
df_uk.head(10)
```

In [ ]:
```
# Count User's type
df_uk['UserType'].value_counts()
```

In [ ]:
```
# Group the df_uk DataFrame by 'InvoiceYearMonth' and 'UserType' columns, and sum up the 'Revenue' column for each group
df_user_type_revenue = df_uk.groupby(['InvoiceYearMonth', 'UserType'])['Revenue'].sum().reset_index()

# Remove the rows with the InvoiceYearMonth of 201012 and 201112
df_user_type_revenue = df_user_type_revenue.query("InvoiceYearMonth != 201012 and InvoiceYearMonth != 201112")

# Display the first 5 rows of the resulting DataFrame
df_user_type_revenue.head(5)
```

In [ ]:
```
# Plot graph
plot_data = [
    go.Scatter(
        x=df_user_type_revenue.query("UserType == 'Existing'")['InvoiceYearMonth'],
        y=df_user_type_revenue.query("UserType == 'Existing'")['Revenue'],
        name='Existing'),
    go.Scatter(
        x=df_user_type_revenue.query("UserType == 'New'")['InvoiceYearMonth'],
        y=df_user_type_revenue.query("UserType == 'New'")['Revenue'],
        name='New'),
]

plot_layout = go.Layout(
    xaxis=dict(
        type="category",
        tickfont=dict(size=28),
        title_font=dict(size=32),
        title="Month"
    ),
    yaxis=dict(
        tickfont=dict(size=28),
        title_font=dict(size=32),
        title="Revenue"
    ),
```

```
    title=dict(text="Revenue of New and Existing Customers in 2011", font=dict(size=40)),
)

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.plot(fig)
```

Monthly New Customer Ratio

In [ ]:
```
# Monthly New Customers
# Select only the rows where the UserType is 'New' and group them by the InvoiceYearMonth column
countNew = df_uk.query("UserType == 'New'").groupby(['InvoiceYearMonth'])['CustomerID'].nunique()

#Select only the rows where the UserType is 'Existing' and group them by the InvoiceYearMonth column
countExisting = df_uk.query("UserType == 'Existing'").groupby(['InvoiceYearMonth'])['CustomerID'].nunique()

#Calculate the ratio of new customers to existing customers for each month
df_user_ratio = countNew / countExisting
```

In [ ]:
```
# Visualization Monthly New Customers
df_user_ratio = df_user_ratio.reset_index()
df_user_ratio = df_user_ratio.dropna()

plot_data = [
    go.Bar(
        x=df_user_ratio['InvoiceYearMonth'],
        y=df_user_ratio['CustomerID'],
    )
]

plot_layout = go.Layout(
    xaxis=dict(
        type="category",
        tickfont=dict(size=28),
        title_font=dict(size=32),
        title="Month"
    ),
    yaxis=dict(
        tickfont=dict(size=28),
        title_font=dict(size=32),
        title="Ratio"
    ),
```



```
    title=dict(text="New Customer Ratio in 2011", font=dict(size=40)),
)

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.plot(fig)
```

Monthly Retention Rate

In [ ]:
```
# Add a new column to the df_uk dataframe which contains the year and month of each invoice
df_uk['InvoiceYearMonth'] = df_uk['InvoiceDate'].map(lambda date: 100*date.year + date.month)
df_uk.head(5)

# Monthly Active

# Group the df_uk dataframe by InvoiceYearMonth and count the number of unique CustomerIDs
df_monthly_active = df_uk.groupby('InvoiceYearMonth')['CustomerID'].nunique().reset_index()

# Group the df_uk dataframe by CustomerID and InvoiceYearMonth and sum the Revenue column
df_user_purchase = df_uk.groupby(['CustomerID',
                                  'InvoiceYearMonth'])['Revenue'].sum().astype(int).reset_index()
```

In [ ]:
```
# Create the Retention crosstab
df_retention = pd.crosstab(df_user_purchase['CustomerID'],
                           df_user_purchase['InvoiceYearMonth']).reset_index()
df_retention.head()
```

In [ ]:
```
# Loc from 01/2011 to 12/2011
months = df_retention.columns[2:]

#Calculate total customers of 01/2011
df_retention[201101].sum()

# Initialize an empty list to store the retention data for each month
retention_array = []

# Loop through each month except for the last one in the months list
# For each month, calculate the retention rate compared to the previous month

for i in range(len(months)-1):
    #Create a dictionary to store the retention data for the selected month
    retention_data = {}
```

```
    selected_month = months[i+1]
    prev_month = months[i]
    retention_data['InvoiceYearMonth'] = int(selected_month)
    retention_data['TotalUserCount'] = df_retention[selected_month].sum()
    retention_data['RetainedUserCount'] = df_retention[(df_retention[selected_month]==1
                                                        & (df_retention[prev_month]==1))][selected_month].sum()
    retention_array.append(retention_data)
    print("***************" + str(selected_month) + "***************")
    print(retention_array)

# Create a new dataframe from the retention array
df_retention = pd.DataFrame(retention_array)

# Display the dataframe
df_retention

# Monthly Customer Retention Rate
df_retention['RetentionRate'] = df_retention['RetainedUserCount']/df_retention['TotalUserCount']
df_retention
```

Monthly Retention Rate Plot

In [ ]:
```
# Visualization Monthly Customer Retention Rate
plot_data = [
    go.Scatter(
        x=df_retention['InvoiceYearMonth'],
        y=df_retention['RetentionRate'],
        name='Organic'
    )
]

plot_layout = go.Layout(
    xaxis=dict(
        type="category",
        tickfont=dict(size=28),
        title_font=dict(size=32),
        title="Month"
    ),
    yaxis=dict(
        tickfont=dict(size=28),
        title_font=dict(size=32),
        title="Retention Rate"
```



```
    ),
    title=dict(text="Monthly Customer Retention Rate in 2011", font=dict(size=40)),
)

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.plot(fig)
```

Cohort Analysis

Cohort Base Retention

In [ ]:
```
# Cohort Base Retention
df_min_purchase['MinPurchaseYearMonth'] =df_min_purchase['MinPurchaseDate'].map(lambda date: 100*date.year +date.month)
```

In [ ]:
```
#crosstab
df_retention = pd.crosstab(df_user_purchase['CustomerID'], df_user_purchase['InvoiceYearMonth']).reset_index()
#merge table
df_retention = pd.merge(df_retention, df_min_purchase [['CustomerID', 'MinPurchaseYearMonth']], on ='CustomerID', how='left')
```

In [ ]:
```
# Rename the columns of the df_retention dataframe to have a prefix of "m_" before each month
new_column_names = ['m_' + str(column) for column in df_retention.columns[:-1]]

# This is done to distinguish the monthly columns from the minimum purchase month column
new_column_names.append('MinPurchaseYearMonth')
df_retention.columns = new_column_names

# Display the dataframe with the renamed columns
df_retention
```

In [ ]:
```
# Cohort Base Retention
retention_array = []
for i in range(len(months)):
    retention_data = {}
    selected_month = months[i]
    prev_months = months[:i]
    next_months = months[i+1:]
    for prev_month in prev_months:
        retention_data[prev_month] =np.nan
```

```
    total_user_count = df_retention[df_retention.MinPurchaseYearMonth == selected_month].MinPurchaseYearMonth.count()
    retention_data['TotalUserCount'] = total_user_count
    retention_data[selected_month] =1

    query = "MinPurchaseYearMonth == {}".format(selected_month)

    for next_month in next_months:
        new_query = query + " and {} > 0".format(str('m_' +str(next_month)))
        retention_data[next_month] = np.round(df_retention.query(new_query)['m_' +str(next_month)].sum()/total_user_count,2)

    retention_array.append(retention_data)

df_retention = pd.DataFrame(retention_array)
```

Cohort Analysis Base Retention Plot

In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Transpose the dataframe
df_retention = df_retention

# Calculate the total user count
df_retention['Total'] = df_retention.sum(axis=1)

# Plot the cohort chart
sns.set(style='white')
plt.figure(figsize=(12, 8))
plt.title('Cohort Analysis - User Retention', fontsize=28)
ax = sns.heatmap(df_retention.iloc[:, :-1], annot=True, cmap='Blues', fmt='g', vmin=0.0, vmax=1.0, cbar=False)
plt.xlabel('Cohort Group', fontsize=18)
plt.ylabel('Time Delta (Months)', fontsize=18)

# Rotate the x-axis labels by 45 degrees
plt.setp(ax.get_xticklabels(), rotation=45, fontsize=16)
plt.setp(ax.get_yticklabels(), fontsize=16)

plt.savefig('Cohort.png', dpi=300)
plt.show()
```


CUSTOMER SEGMENTATION

In [ ]:
```
# Import necessary modules

from datetime import datetime, timedelta, date
from __future__ import division
from sklearn.cluster import KMeans
```

In [ ]:
```
# Convert 'InvoiceDate' column to datetime format and extract date component
df_uk['InvoiceDate'] = pd.to_datetime(df_uk['InvoiceDate']).dt.date

# Display first few rows of the modified DataFrame
df_uk.head()
```

RECENCY

In [ ]:
```
# RECENCY QUATER  2 /2011 (03/2011 - 06/2011)
# filter 3 months
# Select orders made between 03/2011 and 06/2011
df_q2 = df_uk[(df_uk.InvoiceDate < date(2011,6,1)) & (df_uk.InvoiceDate >= date(2011,3,1))].reset_index()

# Create a DataFrame of unique customer IDs
df_uk_user_q2 = pd.DataFrame(df_q2['CustomerID'].unique())
df_uk_user_q2.columns = ['CustomerID']

# Calculate the most recent purchase date for each customer in the selected time period
df_max_purchase = df_q2.groupby('CustomerID').InvoiceDate.max().reset_index()
df_max_purchase.columns = ['CustomerID', 'MaxPurchaseDate']

# Calculate the number of days between each customer's most recent purchase and the end of the selected time period
df_max_purchase['Recency'] = (df_max_purchase['MaxPurchaseDate'].max() - df_max_purchase['MaxPurchaseDate']).dt.days

# Display the first few rows of the modified DataFrame
df_max_purchase.head()
```

In [ ]:
```
# Calculate Frequency (count how many times a customer made purchases)
df_frequency = df_q2.groupby('CustomerID').InvoiceDate.count().reset_index()
df_frequency.columns = ['CustomerID', 'Frequency']
```

```
# Merge Frequency to df_uk_user_q2
df_uk_user_q2 = pd.merge(df_uk_user_q2, df_frequency, on='CustomerID', how='left')

# Calculate revenue and merge it to the customer dataset
df_q2['Revenue'] = df_q2['Price'] * df_q2['Quantity']
df_revenue = df_q2.groupby('CustomerID').Revenue.sum().reset_index()
df_uk_user_q2 = pd.merge(df_uk_user_q2, df_revenue, on='CustomerID', how='left')

# Remove outliers with Frequency=1664 and Revenue=78414.180000
df_uk_user_q2 = df_uk_user_q2[(df_uk_user_q2['Frequency'] != 1664) & (df_uk_user_q2['Revenue'] != 78414.180000)]

# Show the resulting dataframe
df_uk_user_q2
```

In [ ]:
```
# Merge df_uk_user_q2 to df_max_purchase on CustomerID
df_uk_user_q2 = pd.merge(df_uk_user_q2, df_max_purchase[['CustomerID', 'Recency']], on = 'CustomerID')

# Display the first few rows of the modified DataFrame
df_uk_user_q2.head()
```

In [ ]:
```
# Calculate summary statistics for the Recency column of df_uk_user_q2
df_uk_user_q2.Recency.describe()
```

Recency KMeans (result k = 4)

In [ ]:
```
# Import necessary modules
import plotly.graph_objs as go
import plotly.offline as pyoff
from sklearn.cluster import KMeans

# Initialize an empty list of SSE values
sse = [0] * 10
# Extract the Recency column from the df_uk_user_q2 DataFrame
df_recency = df_uk_user_q2[['Recency']]
for k in range(1, 10):
    kmeans = KMeans(n_clusters=k, max_iter=1000).fit(df_recency)
    df_recency["clusters"] = kmeans.labels_
    sse[k] = kmeans.inertia_

# Create a scatter plot of SSE values for each k
```



```
plot_data = [
    go.Scatter(
        x = list(range(2, 10)),
        y = sse[1:10],
        line=dict(width=3.6) # set line width to 3.6
    )
]
# Set the plot layout
plot_layout = go.Layout(
    title = {
        'text': 'Number of Clusters of Recency in q2',
        'font': {'size': 38}
    },
    xaxis = {
        'title': {
            'text': 'Number of clusters',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    },
    yaxis = {
        'title': {
            'text': 'SSE',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    }
)
# Create the plot figure
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.plot(fig)
```

Silhoutte Score for Recency

In [ ]:
```
# Import necessary modules
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
# Define a list of custom colors
custom_colors = ['#ffb3ba', '#ffdfba', '#ffffba', '#baffc9', '#bae1ff', '#ffb3ba', '#ffdfba', '#ffffba']
# Extract the Recency column from the df_uk_user_q2 DataFrame
df_recency = df_uk_user_q2[['Recency']]
```

```
range_n_clusters = range(2, 10)
# Create a subplot for each k value
fig, ax = plt.subplots(2, 4, figsize=(15,10))
# Create empty list to store silhouette scores for each k value
silhouette_scores = []
for i, n_clusters in enumerate(range_n_clusters):
    # Initialize KMeans and fit the data
    kmeans = KMeans(n_clusters=n_clusters, max_iter=1000).fit(df_recency)
    cluster_labels = kmeans.labels_

    # Calculate silhouette scores
    silhouette_avg = silhouette_score(df_recency, cluster_labels)
    sample_silhouette_values = silhouette_samples(df_recency, cluster_labels)
    silhouette_scores.append(silhouette_avg)

    print(f"Silhouette score for k={n_clusters}: {silhouette_avg}")
    # Plot silhouette diagram
    y_lower = 10
    for j in range(n_clusters):
        # Aggregate the silhouette scores for samples belonging to cluster j, and sort them
        jth_cluster_silhouette_values = sample_silhouette_values[cluster_labels == j]
        jth_cluster_silhouette_values.sort()

        size_jth_cluster = jth_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_jth_cluster

        color = custom_colors[j % len(custom_colors)]

        ax[i//4, i%4].barh(
            range(y_lower, y_upper),
            jth_cluster_silhouette_values,
            height=1.0,
            edgecolor='none',
            color=color
        )
        # Label the silhouette plots with their cluster numbers at the middle
        ax[i//4, i%4].text(-0.05, y_lower + 0.5 * size_jth_cluster, str(j), fontsize=14)

        # Compute the new y_lower for next plot
        y_lower = y_upper + 10

    # Plot the mean silhouette coefficient for all clusters
    ax[i//4, i%4].axvline(x=silhouette_avg, color="green", linestyle="--")
    ax[i//4, i%4].set_yticks([])  # Clear the yaxis labels / ticks
```



```
    ax[i//4, i%4].set_xlim([-0.1, 1])
    ax[i//4, i%4].set_xlabel("Silhouette Coefficient", fontsize=12)
    ax[i//4, i%4].set_title(f"(k={n_clusters})", fontsize=14)
# Set the main title of the plot
fig.suptitle("Silhouette analysis for KMeans clustering on Recency data", fontsize=28)
plt.tight_layout()
plt.show()
```

In [ ]:
```
# Set the number of clusters to 4
kmeans = KMeans(n_clusters= 4)

# Fit the K-Means model to the Recency column of the df_uk_user_q2 DataFrame
kmeans.fit(df_uk_user_q2[['Recency']])

# Use the K-Means model to predict the cluster labels for each customer based on Recency
df_uk_user_q2['RecencyCluster'] = kmeans.predict(df_uk_user_q2[["Recency"]])

# Display the head of the df_uk_user_q2 DataFrame to verify the cluster assignments
df_uk_user_q2.head()
```

In [ ]:
```
# Group the customers by RecencyCluster and display the descriptive statistics for the Recency column
df_uk_user_q2.groupby('RecencyCluster')['Recency'].describe()
```

In [ ]:
```
df_uk_user_q2
```

Rearange Recency Cluster

In [ ]:
```
# mean cluster 2 < cluster 3 < cluster 0 < cluster 1, so we need to arrange it in the order
# recency as small as the big of cluster number
# reorder

df_new = df_uk_user_q2.groupby('RecencyCluster')['Recency'].mean().reset_index()
```

In [ ]:
```
df_new
```

In [ ]:
```
# Sort the df_new DataFrame by Recency in descending order and reset the index
df_new = df_new.sort_values(by="Recency", ascending=False).reset_index(drop=True)

# Add a new column called "index" to df_new that contains the sorted index values
df_new['index'] = df_new.index
```

```
# Display the updated df_new DataFrame
df_new
```

In [ ]:
```
# Merge the df_uk_user_q2 DataFrame with df_new on the RecencyCluster column, adding the index column from df_new to df_uk_user_
df_uk_user_q2_recency = pd.merge(df_uk_user_q2, df_new[['RecencyCluster', 'index']], on='RecencyCluster', how='left')

# Display the updated df_uk_user_q2_recency DataFrame
df_uk_user_q2_recency
```

In [ ]:
```
#drop column 'RecencyCluster'
df_uk_user_q2_recency = df_uk_user_q2_recency.drop(['RecencyCluster'], axis = 1)
```

In [ ]:
```
df_uk_user_q2_recency.head()
```

In [ ]:
```
# Rename column 'index' to 'RecencyCluster' and sort value of this descending
df_uk_user_q2_recency = df_uk_user_q2_recency.rename(columns = {"index" : "RecencyCluster"})
df_uk_user_q2_recency.sort_values('RecencyCluster', ascending = False)
```

In [ ]:
```
# update df_uk_user_q2
df_uk_user_q2 = df_uk_user_q2_recency
df_uk_user_q2.sort_values('RecencyCluster', ascending = False)
```

In [ ]:
```
# Plot a boxplot of RecencyCluster grouped by the 4 clusters
sns.set(style="whitegrid", font_scale=1.2)
ax = sns.boxplot(x="RecencyCluster", y="Recency", data=df_uk_user_q2, palette="pastel")

# Add titles and labels
ax.set_title("Distribution of Recency by Recency Clusters")
ax.set_xlabel("Recency Clusters")
ax.set_ylabel("Recency (Days)")

# Save the figure
plt.savefig("recency_boxplot.png", dpi=300)
```

FREQUENCY

In [ ]:
```
# Describe Frequency
df_uk_user_q2.Frequency.describe()
```


In [ ]:
```
# Plot Frequency Distribution

import plotly.graph_objs as go
from plotly.offline import plot

# create a histogram trace with custom styling
hist_trace = go.Histogram(
    x=df_uk_user_q2.query('Frequency < 200')["Frequency"],
    marker=dict(color='#1f77b4')
)

# create a plot layout with custom styling
plot_layout = go.Layout(
    title='Distribution of Frequency in q2',
    xaxis=dict(title='Frequency'),
    yaxis=dict(title='Count'),
    plot_bgcolor='#f7f7f7',
    margin=dict(l=50, r=50, t=50, b=50)
)

# create a plot figure and update its layout and traces
fig = go.Figure(data=[hist_trace], layout=plot_layout)

# customize the histogram trace and update the plot figure
fig.update_traces(
    opacity=0.8,
    marker=dict(line=dict(color='#ffffff', width=1))
)

# update the plot layout with additional customizations
fig.update_layout(
    font=dict(family='Arial', size=24),
    title=dict(x=0.5),
    hovermode='x',
    bargap=0.2
)

# show the plot
plot(fig)
```

Frequency Kmeans (Result k =3)

In [ ]:
```
# Import necessary modules
import plotly.graph_objs as go
import plotly.offline as pyoff
from sklearn.cluster import KMeans
# Initialize an empty list of SSE values
sse = [0] * 10

# Extract the Recency column from the df_uk_user_q2 DataFrame
df_frequency = df_uk_user_q2[['Frequency']]

for k in range(1, 10):
    kmeans = KMeans(n_clusters=k, max_iter=1000).fit(df_frequency)
    df_frequency["clusters"] = kmeans.labels_
    sse[k] = kmeans.inertia_

# Create a scatter plot of SSE values for each k
plot_data = [
    go.Scatter(
        x = list(range(2, 10)),
        y = sse[1:10],
        line=dict(width=3.6) # set line width to 3.6
    )
]
# Set the plot layout
plot_layout = go.Layout(
    title = {
        'text': 'Number of Clusters of Frequency in q2',
        'font': {'size': 38}
    },
    xaxis = {
        'title': {
            'text': 'Number of clusters',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    },
    yaxis = {
        'title': {
            'text': 'SSE',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    }
```



```
)
# Create the plot figure
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.plot(fig)
```

Silhoutte Score for Frequency

In [ ]:
```
# Import necessary modules
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

# Define a list of custom colors
custom_colors = ['#ffb3ba', '#ffdfba', '#ffffba', '#baffc9', '#bae1ff', '#ffb3ba', '#ffdfba', '#ffffba']

# Extract the Recency column from the df_uk_user_q2 DataFrame
df_frequency = df_uk_user_q2[['Frequency']]

range_n_clusters = range(2, 10)

# Create empty list to store silhouette scores for each k value
silhouette_scores = []

# Create a subplot for each k value
fig, ax = plt.subplots(2, 4, figsize=(15,10))

for i, n_clusters in enumerate(range_n_clusters):
    # Initialize KMeans and fit the data
    kmeans = KMeans(n_clusters=n_clusters, max_iter=1000).fit(df_frequency)
    cluster_labels = kmeans.labels_

    # Calculate silhouette scores
    silhouette_avg = silhouette_score(df_frequency, cluster_labels)
    sample_silhouette_values = silhouette_samples(df_frequency, cluster_labels)
    silhouette_scores.append(silhouette_avg)

    print(f"Silhouette score for k={n_clusters}: {silhouette_avg}")
    # Plot silhouette diagram
    y_lower = 10
    for j in range(n_clusters):
        # Aggregate the silhouette scores for samples belonging to cluster j, and sort them
        jth_cluster_silhouette_values = sample_silhouette_values[cluster_labels == j]
```

```
        jth_cluster_silhouette_values.sort()

        size_jth_cluster = jth_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_jth_cluster

        color = custom_colors[j % len(custom_colors)]

        ax[i//4, i%4].barh(
            range(y_lower, y_upper),
            jth_cluster_silhouette_values,
            height=1.0,
            edgecolor='none',
            color=color
        )
        # Label the silhouette plots with their cluster numbers at the middle
        ax[i//4, i%4].text(-0.05, y_lower + 0.5 * size_jth_cluster, str(j), fontsize=14)

        # Compute the new y_lower for next plot
        y_lower = y_upper + 10

    # Plot the mean silhouette coefficient for all clusters
    ax[i//4, i%4].axvline(x=silhouette_avg, color="green", linestyle="--")
    ax[i//4, i%4].set_yticks([])  # Clear the yaxis labels / ticks
    ax[i//4, i%4].set_xlim([-0.1, 1])
    ax[i//4, i%4].set_xlabel("Silhouette Coefficient", fontsize=12)
    ax[i//4, i%4].set_title(f"(k={n_clusters})", fontsize=14)

# Set the main title of the plot
fig.suptitle("Silhouette analysis for KMeans clustering on Frequency data", fontsize=28)

plt.tight_layout()
plt.show()
```

In [ ]:
```
#choose K = 3 for Frequency Clusterings
kmeans = KMeans(n_clusters= 3)
kmeans.fit(df_uk_user_q2[['Frequency']])
df_uk_user_q2['FrequencyCluster'] =kmeans.predict(df_uk_user_q2[["Frequency"]])
df_uk_user_q2.head()
```

In [ ]:
```
# Check frequency order
df_uk_user_q2.groupby('FrequencyCluster')['Frequency'].describe()
```


In [ ]:
```
# Because 0<1<2 and 23<111<410, so we dont need to rearrange clusters
```

Plot Boxplot of FrequencyClusters

In [ ]:
```
# Plot a boxplot of FrequencyCluster grouped by the 3 clusters
sns.set(style="whitegrid", font_scale=1.2)
ax = sns.boxplot(x="FrequencyCluster", y="Frequency", data=df_uk_user_q2, palette="pastel")
# Add titles and labels
ax.set_title("Distribution of Frequency by Frequency Clusters")
ax.set_xlabel("Frequency Clusters")
ax.set_ylabel("Frequency (times)")

# Save the figure
plt.savefig("frequency_boxplot.png", dpi=300)
```

In [ ]:
```
df_uk_user_q2
```

MONETARY (REVENUE)

In [ ]:
```
# Remove outliers with Frequency=1664 and Revenue=78414.180000
df_uk_user_q2 = df_uk_user_q2[(df_uk_user_q2['Frequency'] != 1664) & (df_uk_user_q2['Revenue'] != 78414.180000)]

# MONETARY VALUE in Quarter 2 /2011
# Calculate Monetary Value
df_uk_user_q2.describe()
```

In [ ]:
```
df_uk_user_q2.Revenue.describe()
```

Visualize Monetary distribution

In [ ]:
```
# Visualize Monetary distribution
# From the primary graph, we should choose -500 < revenue < 2500

# create a histogram trace with custom styling
hist_trace = go.Histogram(
    x=df_uk_user_q2.query('Revenue < 2500 and Revenue >-500')["Revenue"],
    marker=dict(color='#1f77b4')
)
```

```
# create a plot layout with custom styling
plot_layout = go.Layout(
    title='Distribution of Revenue in q2',
    xaxis=dict(title='Revenue'),
    yaxis=dict(title='Count'),
    plot_bgcolor='#f7f7f7',
    margin=dict(l=50, r=50, t=50, b=50)
)

# create a plot figure and update its layout and traces
fig = go.Figure(data=[hist_trace], layout=plot_layout)

# customize the histogram trace and update the plot figure
fig.update_traces(
    opacity=0.8,
    marker=dict(line=dict(color='#ffffff', width=1))
)

# update the plot layout with additional customizations
fig.update_layout(
    font=dict(family='Arial', size=24),
    title=dict(x=0.5),
    hovermode='x',
    bargap=0.2
)
# show the plot
plot(fig)
```

Monetary Kmeans (result k=3)

In [ ]:
```
# Clustering for Monetary ( Revenue)
import plotly.graph_objs as go
import plotly.offline as pyoff
from sklearn.cluster import KMeans

sse = [0] * 10
df_revenue = df_uk_user_q2[['Revenue']]

for k in range(1, 10):
    kmeans = KMeans(n_clusters=k, max_iter=1000).fit(df_revenue)
    df_revenue["clusters"] = kmeans.labels_
```



```
    sse[k] = kmeans.inertia_

# Create a scatter plot of SSE values for each k
plot_data = [
    go.Scatter(
        x = list(range(2, 10)),
        y = sse[1:10],
        line=dict(width=3.6) # set line width to 3.6
    )
]
# Set the plot layout
plot_layout = go.Layout(
    title = {
        'text': 'Number of Clusters of Revenue in q2',
        'font': {'size': 38}
    },
    xaxis = {
        'title': {
            'text': 'Number of clusters',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    },
    yaxis = {
        'title': {
            'text': 'SSE',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    }
)

fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.plot(fig)
```

Silhouette for Monetary

In [ ]:
```
# Import necessary modules
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score
```

```
# Define a list of custom colors
custom_colors = ['#ffb3ba', '#ffdfba', '#ffffba', '#baffc9', '#bae1ff', '#ffb3ba', '#ffdfba', '#ffffba']

# Extract the Recency column from the df_uk_user_q2 DataFrame
df_revenue = df_uk_user_q2[['Revenue']]

range_n_clusters = range(2, 10)

# Create empty list to store silhouette scores for each k value
silhouette_scores = []

# Create a subplot for each k value
fig, ax = plt.subplots(2, 4, figsize=(15,10))

for i, n_clusters in enumerate(range_n_clusters):
    # Initialize KMeans and fit the data
    kmeans = KMeans(n_clusters=n_clusters, max_iter=1000).fit(df_revenue)
    cluster_labels = kmeans.labels_

    # Calculate silhouette scores
    silhouette_avg = silhouette_score(df_revenue, cluster_labels)
    sample_silhouette_values = silhouette_samples(df_revenue, cluster_labels)
    silhouette_scores.append(silhouette_avg)

    print(f"Silhouette score for k={n_clusters}: {silhouette_avg}")
    # Plot silhouette diagram
    y_lower = 10
    for j in range(n_clusters):
        # Aggregate the silhouette scores for samples belonging to cluster j, and sort them
        jth_cluster_silhouette_values = sample_silhouette_values[cluster_labels == j]
        jth_cluster_silhouette_values.sort()

        size_jth_cluster = jth_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_jth_cluster

        color = custom_colors[j % len(custom_colors)]

        ax[i//4, i%4].barh(
            range(y_lower, y_upper),
            jth_cluster_silhouette_values,
            height=1.0,
            edgecolor='none',
            color=color
        )
```



```
        # Label the silhouette plots with their cluster numbers at the middle
        ax[i//4, i%4].text(-0.05, y_lower + 0.5 * size_jth_cluster, str(j), fontsize=14)

        # Compute the new y_lower for next plot
        y_lower = y_upper + 10

    # Plot the mean silhouette coefficient for all clusters
    ax[i//4, i%4].axvline(x=silhouette_avg, color="green", linestyle="--")
    ax[i//4, i%4].set_yticks([])  # Clear the yaxis labels / ticks
    ax[i//4, i%4].set_xlim([-0.1, 1])
    ax[i//4, i%4].set_xlabel("Silhouette Coefficient", fontsize=12)
    ax[i//4, i%4].set_title(f"(k={n_clusters})", fontsize=14)

# Set the main title of the plot
fig.suptitle("Silhouette analysis for KMeans clustering on Revenue data", fontsize=28)

plt.tight_layout()
plt.show()
```

Choose k=3 then Run Kmeans

In [ ]:
```
#choose K = 3 for Kmeans models of Revenue

kmeans = KMeans(n_clusters= 3)
kmeans.fit(df_uk_user_q2[['Revenue']])
df_uk_user_q2['RevenueCluster'] =kmeans.predict(df_uk_user_q2[["Revenue"]])
df_uk_user_q2.head()
```

Check if we need to re-arrange clusters of Revenue

In [ ]:
```
df_new = df_uk_user_q2.groupby('RevenueCluster')['Revenue'].mean().reset_index()
df_new
```

Re-arrange Monetary Clusters

In [ ]:
```
# Reorder Revenue Clusters
# Calculate the mean revenue for each revenue cluster
df_new = df_uk_user_q2.groupby('RevenueCluster')['Revenue'].mean().reset_index()
```

```
# Sort the revenue clusters by ascending order of mean revenue and reset the index
df_new = df_new.sort_values(by="Revenue", ascending=True).reset_index(drop=True)

# Assign a new index to the sorted revenue clusters
df_new['index'] = df_new.index

# Merge the new index of revenue clusters with the original dataframe on the RevenueCluster column
df_uk_user_q2 = pd.merge(df_uk_user_q2, df_new[['RevenueCluster', 'index']], on='RevenueCluster', how='left')

# Drop the original RevenueCluster column
df_uk_user_q2 = df_uk_user_q2.drop(['RevenueCluster'], axis=1)

# Rename the new index column as RevenueCluster
df_uk_user_q2 = df_uk_user_q2.rename(columns={"index": "RevenueCluster"})
```

In [ ]:
```
df_uk_user_q2
```

In [ ]:
```
df_uk_user_q2.groupby('RevenueCluster')['Revenue'].describe()
```

Kmeans Scatter Plot

In [ ]:
```
# Import necessary modules
import matplotlib.pyplot as plt

# Set the figure size and dpi
fig = plt.figure(figsize=(6, 4), dpi=300)

# Scatter plot of Revenue vs Frequency colored by RevenueCluster
scatter = plt.scatter(df_uk_user_q2['Frequency'], df_uk_user_q2['Revenue'], c=df_uk_user_q2['RevenueCluster'], cmap='viridis', m

# Add a color bar for the RevenueCluster
cbar = plt.colorbar(scatter)
cbar.set_label('RevenueCluster', fontsize=14)

# Set the x-axis and y-axis labels and title
plt.xlabel('Frequency', fontsize=14)
plt.ylabel('Revenue', fontsize=14)
plt.title('KMeans Normal Scatter Plot For Revenue', fontsize=18)

# Set the x-axis and y-axis limits
```



```
plt.xlim(0, max(df_uk_user_q2['Frequency'])+10)
plt.ylim(0, max(df_uk_user_q2['Revenue'])+5000)

# Add grid lines
plt.grid(True, linestyle='--')

# Show the plot
plt.show()
```

OVERALL SEGMENTATION

In [ ]:
```
# OVERALL SEGMENTATION = Recency Cluster + Frequency Cluster + Revenue Cluster
df_uk_user_q2['OverallScore'] = df_uk_user_q2['RecencyCluster'] + df_uk_user_q2['FrequencyCluster']+ df_uk_user_q2['RevenueClust
df_uk_user_q2
```

In [ ]:
```
# Check the mean of features with each OverallScore
df_uk_user_q2.groupby('OverallScore')['Recency','Frequency', 'Revenue'].mean()
```

SETTING CLASSES

In [ ]:
```
# Segmentation Customer by value of OverallScore
df_uk_user_q2['Segment'] = 'Low-Value'
df_uk_user_q2.loc[df_uk_user_q2['OverallScore'] > 2, 'Segment'] = 'Mid-Value'
df_uk_user_q2.loc[df_uk_user_q2['OverallScore'] > 4, 'Segment'] = 'High-Value'
```

In [ ]:
```
df_uk_user_q2
```

Calculate Distribution of group with Overall Score

In [ ]:
```
import matplotlib.pyplot as plt

# Calculate percentage of each segment
segment_counts = df_uk_user_q2['Segment'].value_counts(normalize=True)
low_value = round(segment_counts['Low-Value']*100, 2)
mid_value = round(segment_counts['Mid-Value']*100, 2)
high_value = round(segment_counts['High-Value']*100, 2)

# Create bar plot
```

```
fig, ax = plt.subplots(figsize=(8, 6), dpi=300)
ax.bar(x=['Low-Value', 'Mid-Value', 'High-Value'], height=[low_value, mid_value, high_value], color=['red', 'gold', 'green'], al
ax.set_title('Customer Segmentation by Overall Value')
ax.set_xlabel('Segment')
ax.set_ylabel('Percentage (%)')
ax.set_ylim([0, 100])

# Add markers
for i, v in enumerate([low_value, mid_value, high_value]):
    ax.text(i, v + 1, str(v), ha='center', va='bottom', fontsize=14)

plt.show()
```

In [ ]:

```
import matplotlib.pyplot as plt
import seaborn as sns

# Define the list of variables and the list of segments to plot
variables = ['Frequency', 'Revenue', 'Recency']
segments = df_uk_user_q2['Segment'].unique()

# Set up the figure with subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Loop over the variables and the segments to create the boxplots
for i, variable in enumerate(variables):
    for j, segment in enumerate(segments):
        # Filter the dataframe to include only the current segment
        df_segment = df_uk_user_q2[df_uk_user_q2['Segment'] == segment]

        # Create a boxplot for the current variable and segment
        sns.boxplot(x=variable, y='Segment', data=df_segment, ax=axs[i], order=segments)

        # Set the title for the current subplot
        if j == 0:
            axs[i].set_title(variable)

        # Remove the y-axis label for all but the first subplot
        if i != 0:
            axs[i].set_ylabel('')

        # Set the x-axis label for the first subplot only
        if i == 0:
            axs[i].set_xlabel(variable)
```



```
# Adjust the spacing between subplots and display the figure
plt.tight_layout()
plt.show()
```

In [ ]:

```
import matplotlib.pyplot as plt
import seaborn as sns

# Define the list of variables and the list of segments to plot
variables = ['RecencyCluster', 'FrequencyCluster', 'RevenueCluster']
segments = df_uk_user_q2['Segment'].unique()

# Set up the figure with subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 5))

# Loop over the variables and the segments to create the boxplots
for i, variable in enumerate(variables):
    for j, segment in enumerate(segments):
        # Filter the dataframe to include only the current segment
        df_segment = df_uk_user_q2[df_uk_user_q2['Segment'] == segment]

        # Create a boxplot for the current variable and segment
        sns.boxplot(x='Segment', y=variable, data=df_segment, ax=axs[i], order=segments)

        # Set the title for the current subplot
        if j == 0:
            axs[i].set_title(variable)

        # Remove the y-axis label for all but the first subplot
        if i != 0:
            axs[i].set_ylabel('')

        # Set the x-axis label for the first subplot only
        if i == 0:
            axs[i].set_xlabel('Segment')

# Adjust the spacing between subplots and display the figure
plt.tight_layout()
plt.show()
```

Visualization correlation between features

Frequency vs Revenue(Monetary)

In [ ]:
```
# Visualization

# Frequency vs Revenue(Monetary)

df_graph = df_uk_user_q2.query("Revenue < 50000 and Frequency < 2000")

plot_data = [    go.Scatter(        x = df_graph.query("Segment == 'Low-Value'")['Frequency'],
        y = df_graph.query("Segment == 'Low-Value'")['Revenue'],
        mode = 'markers',
        name = "Low",
        marker =dict(size= 11,
                    line = dict(width = 1),
                    color = 'red',
                    opacity = 0.7)),
    go.Scatter(
        x = df_graph.query("Segment == 'Mid-Value'")['Frequency'],
        y = df_graph.query("Segment == 'Mid-Value'")['Revenue'],
        mode = 'markers',
        name = "Mid",
        marker =dict(size= 13,
                    line = dict(width = 1),
                    color = 'yellow',
                    opacity = 0.4)),
    go.Scatter(
        x = df_graph.query("Segment == 'High-Value'")['Frequency'],
        y = df_graph.query("Segment == 'High-Value'")['Revenue'],
        mode = 'markers',
        name = "High",
        marker =dict(size= 15,
                    line = dict(width = 1),
                    color = 'green',
                    opacity = 0.9))

]
# Set the plot layout
plot_layout = go.Layout(
    title = {
        'text': 'Customer Segmentation Frequency - Revenue in q2',
        'font': {'size': 38}
    },
```



```
    xaxis = {
        'title': {
            'text': 'Revenue',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    },
    yaxis = {
        'title': {
            'text': 'Frequency',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    }
)
fig = go.Figure(data = plot_data, layout=plot_layout)
pyoff.plot(fig)
```

Recency - Revenue

In [ ]:
```
# Recency - Revenue
df_graph = df_uk_user_q2.query("Revenue < 50000 and Frequency < 2000")

plot_data = [
    go.Scatter(
        x = df_graph.query("Segment == 'Low-Value'")['Recency'],
        y = df_graph.query("Segment == 'Low-Value'")['Revenue'],
        mode = 'markers',
        name = "Low",
        marker =dict(size= 11,
                    line = dict(width = 1),
                    color = 'red',
                    opacity = 0.9)),
    go.Scatter(
        x = df_graph.query("Segment == 'Mid-Value'")['Recency'],
        y = df_graph.query("Segment == 'Mid-Value'")['Revenue'],
        mode = 'markers',
        name = "Mid",
        marker =dict(size= 13,
                    line = dict(width = 1),
                    color = 'yellow',
                    opacity = 0.8)),
    go.Scatter(
```

```
        x = df_graph.query("Segment == 'High-Value'")['Recency'],
        y = df_graph.query("Segment == 'High-Value'")['Revenue'],
        mode = 'markers',
        name = "High",
        marker =dict(size= 15,
                     line = dict(width = 1),
                     color = 'green',
                     opacity = 0.8))

]
# Set the plot Layout
plot_layout = go.Layout(
    title = {
        'text': 'Customer Segmentation Recency - Revenue in q2',
        'font': {'size': 38}
    },
    xaxis = {
        'title': {
            'text': 'Revenue',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    },
    yaxis = {
        'title': {
            'text': 'Recency',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    }
)

fig = go.Figure(data = plot_data, layout=plot_layout)
pyoff.plot(fig)
```

Recency - Frequency

In [ ]:
```
# Recency - Frequency
df_graph = df_uk_user_q2.query("Revenue < 50000 and Frequency < 1000")

plot_data = [
    go.Scatter(
        x = df_graph.query("Segment == 'Low-Value'")['Recency'],
```



```
        y = df_graph.query("Segment == 'Low-Value'")['Frequency'],
        mode = 'markers',
        name = "Low",
        marker =dict(size= 11,
                     line = dict(width = 1),
                     color = 'red',
                     opacity = 0.9)),
    go.Scatter(
        x = df_graph.query("Segment == 'Mid-Value'")['Recency'],
        y = df_graph.query("Segment == 'Mid-Value'")['Frequency'],
        mode = 'markers',
        name = "Mid",
        marker =dict(size= 13,
                     line = dict(width = 1),
                     color = 'yellow',
                     opacity = 0.8)),
    go.Scatter(
        x = df_graph.query("Segment == 'High-Value'")['Recency'],
        y = df_graph.query("Segment == 'High-Value'")['Frequency'],
        mode = 'markers',
        name = "High",
        marker =dict(size= 15,
                     line = dict(width = 1),
                     color = 'green',
                     opacity = 0.8))

]
# Set the plot Layout
plot_layout = go.Layout(
    title = {
        'text': 'Customer Segmentation Recency - Frequency in q2',
        'font': {'size': 38}
    },
    xaxis = {
        'title': {
            'text': 'Frequency',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    },
    yaxis = {
        'title': {
            'text': 'Recency',
            'font': {'size': 30}
        }
```

```
        },
        'tickfont': {'size': 24}
    }
)
fig = go.Figure(data = plot_data, layout=plot_layout)
pyoff.plot(fig)
```

CUSTOMER LIFETIME VALUE PREDICTION

1. Select a time window. It can be 3, 6, 12, 24 months
2. Lifetime Value: Total Gross Revenue - Total Cost
3. Build a machine learning model the predicts our LTV (With Random Forest, SVM and XGBoost models)

Based on 3 months of data (quarter 2 of 2011), calculate RFM, use it for prediction next 6 months(quarter 3 & 4)

In [ ]:
```
# Import Libraries
from xgboost import XGBClassifier, XGBRegressor
from sklearn.model_selection import GridSearchCV
df_uk
```

In [ ]:
```
from datetime import datetime, timedelta, date
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from __future__ import division
import seaborn as sns
import plotly.offline as pyoff
import plotly.graph_objs as go
```

CHOOSING PERIOD FOR THE DATASET

In [ ]:
```
# Based on 3 months of data (quarter 2 of 2011), calculate RFM, use it for prediction next 6 months
# Calculate revenue and create a new dataframe for it

df_q34 = df_uk[(df_uk.InvoiceDate >= date(2011,6,1)) & (df_uk.InvoiceDate <= date(2011,12,1))]
df_q34['Revenue'] = df_q34['Price'] * df_q34['Quantity']
```



```
df_user_q34 = df_q34.groupby('CustomerID')['Revenue'].sum().reset_index()
df_user_q34.columns = ['CustomerID', 'q34_Revenue']
```

In [ ]:
```
df_q34.info()
```

In [ ]:
```
# Import necessary libraries
import plotly.graph_objs as go
import plotly.offline as pyoff

# Filter data and create histogram
plot_data = [
    go.Histogram(
        x=df_user_q34.query('-1000<q34_Revenue < 10000')['q34_Revenue'],
        marker=dict(
            color='royalblue'
        ),
        opacity=0.75
    )
]

# Define plot layout
plot_layout = go.Layout(
    title='Q4 Revenue Histogram (Revenue < $10,000)',
    xaxis=dict(
        title='Revenue',
        tickfont=dict(
            size=14
        ),
        showgrid=False
    ),
    yaxis=dict(
        title='Frequency',
        tickfont=dict(
            size=14
        ),
        showgrid=False
    ),
    plot_bgcolor='white',
    paper_bgcolor='white'
)

# Create plot figure and display
```

```
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
```

In [ ]:
```
# Import necessary libraries
import plotly.graph_objs as go
import plotly.offline as pyoff

# Filter data and create histogram
plot_data = [
    go.Histogram(
        x=df_user_q34.query('-1000<q34_Revenue < 3000')['q34_Revenue'],
        marker=dict(
            color='royalblue'
        ),
        opacity=0.75,
        xbins=dict(
            start=-1000,
            end=3000,
            size=200
        ),
        autobinx=False
    )
]

# Define plot layout
plot_layout = go.Layout(
    title='Q4 Revenue Histogram ($-500< Revenue < $10000)',
    xaxis=dict(
        title='Revenue',
        tickfont=dict(
            size=14
        ),
        showgrid=False
    ),
    yaxis=dict(
        title='Frequency',
        tickfont=dict(
            size=14
        ),
        showgrid=False
    ),
    bargap=0.1,  # Add gaps between bins
    plot_bgcolor='white',
```



```
    paper_bgcolor='white'
)

# Create plot figure and display
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.iplot(fig)
```

Merge our 3 months and 6 months dataframes to see correlations between LTV and features

In [ ]:
```
# Merge 3 months and 6 months dataframes to see correlations between LTV and features
df_merge = pd.merge(df_uk_user_q2, df_user_q34, on = 'CustomerID', how = 'left')
df_merge = df_merge.fillna(0)
df_merge
```

In [ ]:
```
# plot Scater of 3 Customer Segmentation

# Recency - Revenue
df_graph = df_merge.query("q34_Revenue < 30000")

plot_data = [
    go.Scatter(
        x = df_graph.query("Segment == 'Low-Value'")['OverallScore'],
        y = df_graph.query("Segment == 'Low-Value'")['q34_Revenue'],
        mode = 'markers',
        name = "Low",
        marker =dict(size= 11,
                     line = dict(width = 1),
                     color = 'red',
                     opacity = 0.9)),
    go.Scatter(
        x = df_graph.query("Segment == 'Mid-Value'")['OverallScore'],
        y = df_graph.query("Segment == 'Mid-Value'")['q34_Revenue'],
        mode = 'markers',
        name = "Mid",
        marker =dict(size= 13,
                     line = dict(width = 1),
                     color = 'yellow',
                     opacity = 0.8)),
    go.Scatter(
        x = df_graph.query("Segment == 'High-Value'")['OverallScore'],
        y = df_graph.query("Segment == 'High-Value'")['q34_Revenue'],
        mode = 'markers',
```

```
        name = "High",
        marker =dict(size= 15,
                     line = dict(width = 1),
                     color = 'green',
                     opacity = 0.8))

]
# Set the plot layout
plot_layout = go.Layout(
    title = {
        'text': 'Relationship between LTV and RFM',
        'font': {'size': 38}
    },
    xaxis = {
        'title': {
            'text': 'RFM score',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    },
    yaxis = {
        'title': {
            'text': '3 month LTV (q34)',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    }
)
fig = go.Figure(data = plot_data, layout=plot_layout)
pyoff.plot(fig)
```

In [ ]:
```
# Remove Outliers
df_merge = df_merge[df_merge['q34_Revenue'] < df_merge['q34_Revenue'].quantile(0.99)]
```

Using Kmeans to cluster data to segments

In [ ]:
```
# Clustering for Frequency
import plotly.graph_objs as go
import plotly.offline as pyoff
from sklearn.cluster import KMeans
```



```
# Initialize an empty list of SSE values

sse = [0] * 10
df_segment = df_merge[['q34_Revenue']]

for k in range(1, 10):
    kmeans = KMeans(n_clusters=k, max_iter=1000).fit(df_segment)
    df_segment["clusters"] = kmeans.labels_
    sse[k] = kmeans.inertia_

# Create a scatter plot of SSE values for each k
plot_data = [
    go.Scatter(
        x = list(range(2, 10)),
        y = sse[1:10],
        line=dict(width=3.6) # set line width to 3.6
    )
]

# Set the plot layout
plot_layout = go.Layout(
    title = {
        'text': 'Number of Clusters of LTV in q34',
        'font': {'size': 38}
    },
    xaxis = {
        'title': {
            'text': 'Number of clusters',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    },
    yaxis = {
        'title': {
            'text': 'SSE',
            'font': {'size': 30}
        },
        'tickfont': {'size': 24}
    }
)


# Create the plot figure
```

```
fig = go.Figure(data=plot_data, layout=plot_layout)
pyoff.plot(fig)
```

Silhouette For CLV

In [ ]:

```
# Import necessary modules
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_samples, silhouette_score

# Define a list of custom colors
custom_colors = ['#ffb3ba', '#ffdfba', '#ffffba', '#baffc9', '#bae1ff', '#ffb3ba', '#ffdfba', '#ffffba']

# Extract the LTV column from the df_merge DataFrame
df_segment = df_merge[['q34_Revenue']]

range_n_clusters = range(2, 10)

# Create a subplot for each k value
fig, ax = plt.subplots(2, 4, figsize=(15,10))

for i, n_clusters in enumerate(range_n_clusters):
    # Initialize KMeans and fit the data
    kmeans = KMeans(n_clusters=n_clusters, max_iter=1000).fit(df_segment)
    cluster_labels = kmeans.labels_

    # Calculate silhouette scores
    silhouette_avg = silhouette_score(df_segment, cluster_labels)
    sample_silhouette_values = silhouette_samples(df_segment, cluster_labels)

    # Plot silhouette diagram
    y_lower = 10
    for j in range(n_clusters):
        # Aggregate the silhouette scores for samples belonging to cluster j, and sort them
        jth_cluster_silhouette_values = sample_silhouette_values[cluster_labels == j]
        jth_cluster_silhouette_values.sort()

        size_jth_cluster = jth_cluster_silhouette_values.shape[0]
        y_upper = y_lower + size_jth_cluster

        color = custom_colors[j % len(custom_colors)]
```



```
        ax[i//4, i%4].barh(
            range(y_lower, y_upper),
            jth_cluster_silhouette_values,
            height=1.0,
            edgecolor='none',
            color=color
        )

        # Label the silhouette plots with their cluster numbers at the middle
        ax[i//4, i%4].text(-0.05, y_lower + 0.5 * size_jth_cluster, str(j), fontsize=14)

        # Compute the new y_lower for next plot
        y_lower = y_upper + 10

    # Plot the mean silhouette coefficient for all clusters
    ax[i//4, i%4].axvline(x=silhouette_avg, color="green", linestyle="--")
    ax[i//4, i%4].set_yticks([])  # Clear the yaxis labels / ticks
    ax[i//4, i%4].set_xlim([-0.1, 1])
    ax[i//4, i%4].set_xlabel("Silhouette Coefficient", fontsize=12)
    ax[i//4, i%4].set_title(f"(k={n_clusters})", fontsize=14)

# Set the main title of the plot
fig.suptitle("Silhouette analysis for KMeans clustering on Revenue Q34", fontsize=28)

plt.tight_layout()
plt.show()
```

In [ ]:

```
# Creatings 3 clusters
kmeans = KMeans(n_clusters = 3)
kmeans.fit(df_merge[['q34_Revenue']])
df_merge['LTVCluster'] = kmeans.predict(df_merge[['q34_Revenue']])
```

Re-Order cluster number based on LTV

In [ ]:

```
# Re-Order cluster number based on Life Time Value

df_new = df_merge.groupby('LTVCluster')['q34_Revenue'].mean().reset_index()
df_new = df_new.sort_values( by = "q34_Revenue", ascending = True).reset_index(drop = True)
df_new['index'] = df_new.index

df_merge = pd.merge(df_merge, df_new[['LTVCluster', 'index']], on='LTVCluster', how='left')
```

```
df_merge.head()
df_merge = df_merge.drop(['LTVCluster'], axis = 1)
df_merge = df_merge.rename(columns = {"index" : "LTVCluster"})

df_merge.groupby('LTVCluster')['q34_Revenue'].describe()
```

In [ ]:
```
# Create a new cluster dataframe
df_cluster = df_merge.copy()
df_cluster.head()
```

In [ ]:
```
# see details of the clusters
df_cluster.groupby('LTVCluster')['q34_Revenue'].describe()
```

USING GET DUMMIES TO NUMERICAL FEATURES

In [ ]:
```
# MACHINE LEARING

# convert categorical columns to numerical
df_class = pd.get_dummies(df_cluster)
df_class.head()
```

CORRELATION BETWEEN FEATURES

In [ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the correlation matrix
corr_matrix = df_class.corr()

# Set up the matplotlib figure and axis
fig, ax = plt.subplots(figsize=(12, 10))

# Create the heatmap using seaborn
sns.heatmap(corr_matrix, cmap='PuOr', annot=True, fmt='.2f', center=0, square=True, linewidths=.5, cbar_kws={"shrink": .5}, ax=a

# Set the title and adjust the position of the colorbar
ax.set_title('Correlation Matrix', fontsize=16)
cbar = ax.collections[0].colorbar
cbar.ax.tick_params(labelsize=12)
```



```
# Save the figure as a PNG file with DPI 300
plt.savefig('correlation_matrix.png', dpi=300, bbox_inches='tight')
plt.show()
```

In [ ]:
```
#sort descending
corr_matrix['LTVCluster'].sort_values(ascending = False)
```

In [ ]:
```
df_class
```

In [ ]:
```
#MODEL 1: Prediction Clusters of Customer Lifetime Value
# import libraries for ML

import xgboost as xgb
from sklearn.model_selection import KFold, cross_val_score, train_test_split
from sklearn.metrics import classification_report, confusion_matrix
```

In [ ]:
```
# create X and y, X will be features set and y is the label - LTV
X = df_class.drop(['LTVCluster', 'q34_Revenue'], axis = 1)
y = df_class['LTVCluster']
```

BASIC XGBOOST MODEL

In [ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.1,random_state = 24)

ltv_xgb_model = xgb.XGBClassifier(maxdepth=3, learning_rate=0.1, objective='multi:softprob', n_jobs=-1).fit(X_train, y_train)

print('Accuracy of XGB Classifier on training set: {:.2f}'.format(ltv_xgb_model.score(X_train, y_train)))
print('Accuracy of XGB Classifier on test set : {:.2f}'.format(ltv_xgb_model.score(X_test[X_train.columns], y_test)))
```

In [ ]:
```
y_pred = ltv_xgb_model.predict(X_test)
print(classification_report(y_test, y_pred))
```

SPLIT TRAIN, VALIDATION AND TEST SETS

In [ ]:
```
from sklearn.model_selection import train_test_split
```

```
# Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.1, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=24)

ltv_xgb_model_split = xgb.XGBClassifier(maxdepth=3, learning_rate=0.1, objective='multi:softprob', n_jobs=-1).fit(X_train, y_tra

print('Accuracy of XGB Classifier on training set: {:.2f}'.format(ltv_xgb_model_split.score(X_train, y_train)))
print('Accuracy of XGB Classifier on validation set : {:.2f}'.format(ltv_xgb_model_split.score(X_val[X_train.columns], y_val)))
print('Accuracy of XGB Classifier on test set : {:.2f}'.format(ltv_xgb_model_split.score(X_test[X_train.columns], y_test)))
```

RESULT AFTER SPLIT TRAIN, VALIDATION AND TEST SETS

In [ ]:
```
from sklearn.metrics import classification_report

y_pred = ltv_xgb_model_split.predict(X_test)
y_train_pred = ltv_xgb_model_split.predict(X_train)

print("Classification Report of XGBOOST model for Test Data:")
print(classification_report(y_test, y_pred))

print("Classification Report for Train Data:")
print(classification_report(y_train, y_train_pred))
```

Using GridCV Search to improve the XGBoost model

In [ ]:
```
from sklearn.model_selection import GridSearchCV

# define parameters to be tested in the grid search
params = {'max_depth': [3, 5, 7],
          'learning_rate': [0.1, 0.01, 0.001],
          'n_estimators': [50, 100, 200],
          'objective': ['multi:softprob']}

# create an XGBClassifier object
xgb_model = xgb.XGBClassifier(n_jobs=-1)

# create a GridSearchCV object
grid_search = GridSearchCV(estimator=xgb_model, param_grid=params, cv=3)

# fit the GridSearchCV object to the training data
grid_search.fit(X_train, y_train)
```



```
# print the best parameters and score
print("Best parameters: ", grid_search.best_params_)
print("Best score: ", grid_search.best_score_)
```

Fit the XGBoost Model with best parameters and show results again

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV

# Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.1, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=26)

# Fit the model with the best parameters
best_ltv_xgb_model = xgb.XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=50, objective='multi:softprob', n_jobs=-1).f

# Print the accuracies on the training, validation, and test sets
print('Accuracy of XGB Classifier on training set: {:.2f}'.format(best_ltv_xgb_model.score(X_train, y_train)))
print('Accuracy of XGB Classifier on validation set : {:.2f}'.format(best_ltv_xgb_model.score(X_val[X_train.columns], y_val)))
print('Accuracy of XGB Classifier on test set : {:.2f}'.format(best_ltv_xgb_model.score(X_test[X_train.columns], y_test)))
```

Result of XGBoost after using GridCV

In [ ]:
```
from sklearn.metrics import classification_report

y_pred = best_ltv_xgb_model.predict(X_test)
y_train_pred = best_ltv_xgb_model.predict(X_train)

print("Classification Report of XGBOOST after tuning for Test Data:")
print(classification_report(y_test, y_pred))

print("Classification Report for Train Data:")
print(classification_report(y_train, y_train_pred))
```

Count percentage of clusters

In [ ]:
```
df_class.groupby('LTVCluster').CustomerID.count()/df_class.CustomerID.count()
```

PLOT PERCENTAGE OF CLUSTERS

In [ ]:
```
import matplotlib.pyplot as plt

# Calculate the percentage of customers in each LTVCluster
ltv_cluster_perc = df_class.groupby('LTVCluster').CustomerID.count()/df_class.CustomerID.count()

# Create a figure and axis object
fig, ax = plt.subplots(figsize=(12,8), dpi=300)

# Plot the percentages as a bar chart with adjusted width
ax.bar(ltv_cluster_perc.index, ltv_cluster_perc.values, width=0.6, color='#33AFFF', edgecolor='black')

# Set the xticks and yticks font sizes
plt.xticks(fontsize=22)
plt.yticks(fontsize=22)

# Add labels and title to the plot
ax.set_xlabel('LTV Cluster', fontsize=22)
ax.set_ylabel('Percentage of Customers', fontsize=22)
ax.set_title('Customer Distribution by LTV Cluster', fontsize=28)

# Add grid lines and remove the top and right spines
ax.grid(axis='y', linestyle='--', alpha=0.7)
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)

# Set the x-axis tick labels to (0, 1, 2)
ax.set_xticks([0, 1, 2])
ax.set_xticklabels(['Cluster 0', 'Cluster 1', 'Cluster 2'], fontsize=18)

# Show the percentage value for each bar
for i, v in enumerate(ltv_cluster_perc.values):
    ax.text(i, v+0.01, str(round(v*100,2))+'%', ha='center', fontsize=18)

# Save and show the plot
plt.savefig('customer_distribution.png', dpi=300, bbox_inches='tight')
plt.show()
```

80% customers is Low-Value Customer and 17% is Mid-Value

SUPPORT VECTOR MACHINE (BASIC)

SPLIT TRAIN, VALIDATION AND TEST SETS

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

# Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.1, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=24)

# create SVM classifier object
svm_model_split = SVC(kernel='rbf', C=0.1, random_state=24)

# train SVM classifier on the training data
svm_model_split.fit(X_train, y_train)

# predict the labels of validation and test data using trained SVM classifier
y_val_pred = svm_model_split.predict(X_val)
y_test_pred = svm_model_split.predict(X_test)

# compute the accuracy score of SVM classifier on validation and test set
val_accuracy = svm_model_split.score(X_val, y_val)
test_accuracy = svm_model_split.score(X_test, y_test)

print('Accuracy of SVM Classifier on training set: {:.2f}'.format(svm_model_split.score(X_train, y_train)))
print('Accuracy of SVM Classifier on validation set : {:.2f}'.format(val_accuracy))
print('Accuracy of SVM Classifier on test set : {:.2f}'.format(test_accuracy))
```

RESULT AFTER SPLIT TRAIN, VALIDATION AND TEST SETS

In [ ]:
```
from sklearn.metrics import classification_report

y_pred = svm_model_split.predict(X_test)
y_train_pred = svm_model_split.predict(X_train)

print("Classification Report of SVM model for Test Data:")
print(classification_report(y_test, y_pred))
```

```
print("Classification Report for Train Data:")
print(classification_report(y_train, y_train_pred))
```

GRIDCV FOR SVM MODEL

In [ ]:
```
from sklearn.model_selection import GridSearchCV

# define parameter grid
param_grid = {'C': [0.1, 1, 10, 100],
              'kernel': ['linear', 'poly', 'rbf', 'sigmoid']}

# create SVM classifier object
svm_model = SVC(random_state=24)

# create GridSearchCV object
grid_search = GridSearchCV(svm_model, param_grid, cv=3, n_jobs=-1)

# fit GridSearchCV object to training data
grid_search.fit(X_trainval, y_trainval)

# print best hyperparameters
print('Best hyperparameters: ', grid_search.best_params_)

# predict the labels of validation and test data using best SVM classifier
best_ltv_svm_model = grid_search.best_estimator_
y_val_pred = best_ltv_svm_model.predict(X_val)
y_test_pred = best_ltv_svm_model.predict(X_test)

# compute the accuracy score of best SVM classifier on validation and test set
val_accuracy = best_ltv_svm_model.score(X_val, y_val)
test_accuracy = best_ltv_svm_model.score(X_test, y_test)

print('Accuracy of best SVM Classifier on validation set : {:.2f}'.format(val_accuracy))
print('Accuracy of best SVM Classifier on test set : {:.2f}'.format(test_accuracy))
```

Result of SVM after using GridCV

In [ ]:
```
# Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.1, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=24)
```



```
# Fit the model with the best parameters
best_ltv_svm_model = SVC(kernel='linear', C=100, random_state=24)

# train SVM classifier on the training data
best_ltv_svm_model.fit(X_train, y_train)

# Print the accuracies on the training, validation, and test sets
print('Accuracy of SVM Classifier on training set: {:.2f}'.format(best_ltv_svm_model.score(X_train, y_train)))
print('Accuracy of SVM Classifier on validation set : {:.2f}'.format(best_ltv_svm_model.score(X_val[X_train.columns], y_val)))
print('Accuracy of SVM Classifier on test set : {:.2f}'.format(best_ltv_svm_model.score(X_test[X_train.columns], y_test)))
```

In [ ]:
```
from sklearn.metrics import classification_report

y_pred = best_ltv_svm_model.predict(X_test)
y_train_pred = best_ltv_svm_model.predict(X_train)

print("Classification Report of SVM after-tunning parameter:")
print(classification_report(y_test, y_pred))

print("Classification Report for Train Data:")
print(classification_report(y_train, y_train_pred))
```

RANDOM FOREST

SPLIT TRAIN-VALIDATION-TEST SET FOR RANDOM FOREST

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.1, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=24)

# create random forest classifier object
rf_model = RandomForestClassifier(n_estimators=100, max_depth=5, random_state=24)

# train random forest classifier on the training data
rf_model.fit(X_train, y_train)

# predict the labels of validation and test data using trained random forest classifier
```

```
y_val_pred = rf_model.predict(X_val)
y_test_pred = rf_model.predict(X_test)

# compute the accuracy score of random forest classifier on validation and test set
val_accuracy = rf_model.score(X_val, y_val)
test_accuracy = rf_model.score(X_test, y_test)

print('Accuracy of Random Forest Classifier on training set: {:.2f}'.format(rf_model.score(X_train, y_train)))
print('Accuracy of Random Forest Classifier on validation set : {:.2f}'.format(val_accuracy))
print('Accuracy of Random Forest Classifier on test set : {:.2f}'.format(test_accuracy))
```

RESULT RANDOM FOREST AFTER USING SPLIT SETS

In [ ]:
```
y_pred = rf_model.predict(X_test)
print("Classification Report of Random Forest:")
print(classification_report(y_test, y_pred))
```

GRIDCV FOR RANDOM FOREST MODEL

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
# define parameter grid
param_grid = {
    'n_estimators': [100, 200, 500],
    'max_depth': [3, 5,7, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
# create random forest classifier object
rf_model = RandomForestClassifier(random_state=24)

# create GridSearchCV object
grid_search = GridSearchCV(rf_model, param_grid, cv=3, n_jobs=-1)

# fit GridSearchCV object to training data
grid_search.fit(X_trainval, y_trainval)

# print best hyperparameters
print('Best hyperparameters: ', grid_search.best_params_)
```


In [ ]:
```
best_ltv_rf_model = RandomForestClassifier(max_depth=3, min_samples_leaf=2, min_samples_split=10, n_estimators=100)
best_ltv_rf_model.fit(X_train, y_train)
```

RESULT AFTERING USING GRIDCV

In [ ]:
```
y_pred = best_ltv_rf_model.predict(X_test)
print("Classification Report of Random Forest after tunning parameter:")
print(classification_report(y_test, y_pred))
```

COMPARISION BETWEEN 3 MODELS

CONFUSION MATRICES

In [ ]:
```
# compute the accuracy scores of the three models on the test set
rf_accuracy = best_ltv_rf_model.score(X_test, y_test)
svm_accuracy = best_ltv_svm_model.score(X_test, y_test)
xgb_accuracy = best_ltv_xgb_model.score(X_test, y_test)

# print the accuracy scores of the three models
print('Accuracy of best Random Forest Classifier on test set : {:.2f}'.format(rf_accuracy))
print('Accuracy of best SVM Classifier on test set : {:.2f}'.format(svm_accuracy))
print('Accuracy of best XGBoost Classifier on test set : {:.2f}'.format(xgb_accuracy))
```

In [ ]:
```
from sklearn.metrics import classification_report, confusion_matrix

# Predict the labels of test data using the best models
y_rf_pred = best_ltv_rf_model.predict(X_test)
y_svm_pred = best_ltv_svm_model.predict(X_test)
y_xgb_pred = best_ltv_xgb_model.predict(X_test)

# Compute the evaluation metrics for each model
rf_report = classification_report(y_test, y_rf_pred)
svm_report = classification_report(y_test, y_svm_pred)
xgb_report = classification_report(y_test, y_xgb_pred)

# Print the evaluation metrics for each model
print('Random Forest Classifier report:\n', rf_report)
print('SVM Classifier report:\n', svm_report)
```

```
print('XGBoost Classifier report:\n', xgb_report)

# Compute the confusion matrix for each model
rf_cm = confusion_matrix(y_test, y_rf_pred)
svm_cm = confusion_matrix(y_test, y_svm_pred)
xgb_cm = confusion_matrix(y_test, y_xgb_pred)

# Print the confusion matrix for each model
print('Random Forest Classifier confusion matrix:\n', rf_cm)
print('SVM Classifier confusion matrix:\n', svm_cm)
print('XGBoost Classifier confusion matrix:\n', xgb_cm)
```

In [ ]:
```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay


# Set the DPI for the plot
plt.rcParams['figure.dpi'] = 300

# create a list of models and their names
models = [('Random Forest', best_ltv_rf_model),('SVM', best_ltv_svm_model),('XGBoost', best_ltv_xgb_model)]

# loop through each model and plot its confusion matrix
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

for i, (name, model) in enumerate(models):
    y_pred = model.predict(X_test)
    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Low', 'Mid', 'High'])
    disp.plot(cmap=plt.cm.Blues, ax=axes[i])
    disp.ax_.set_title('Confusion Matrix for {}'.format(name), fontsize=14)
    disp.im_.colorbar.remove()

# Add a common colorbar for all subplots
fig.subplots_adjust(wspace=0.3, top=0.8)
cbar_ax = fig.add_axes([0.25, -0.03, 0.5, 0.02])
fig.colorbar(disp.im_, cax=cbar_ax, orientation='horizontal')

# Add a title and adjust spacing
fig.suptitle('Confusion Matrices for Predicting Customer Lifetime Value Models', fontsize=24, y=0.98)
fig.tight_layout(pad=2.5)
plt.show()
```


ROC CURVE

In [ ]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle

# Convert y_test to binary format
y_test_binary = label_binarize(y_test, classes=[0, 1, 2])
n_classes = y_test_binary.shape[1]

# Compute ROC curve and ROC area for each class for Random Forest model
fpr_rf = dict()
tpr_rf = dict()
roc_auc_rf = dict()
y_score_rf = best_ltv_rf_model.predict_proba(X_test)
for i in range(n_classes):
    fpr_rf[i], tpr_rf[i], _ = roc_curve(y_test_binary[:, i], y_score_rf[:, i])
    roc_auc_rf[i] = auc(fpr_rf[i], tpr_rf[i])

# Compute ROC curve and ROC area for each class for SVM model
fpr_svm = dict()
tpr_svm = dict()
roc_auc_svm = dict()
y_score_svm = best_ltv_svm_model.decision_function(X_test)
for i in range(n_classes):
    fpr_svm[i], tpr_svm[i], _ = roc_curve(y_test_binary[:, i], y_score_svm[:, i])
    roc_auc_svm[i] = auc(fpr_svm[i], tpr_svm[i])

# Compute ROC curve and ROC area for each class for XGBoost model
fpr_xgb = dict()
tpr_xgb = dict()
roc_auc_xgb = dict()
y_score_xgb = best_ltv_xgb_model.predict_proba(X_test)
for i in range(n_classes):
    fpr_xgb[i], tpr_xgb[i], _ = roc_curve(y_test_binary[:, i], y_score_xgb[:, i])
    roc_auc_xgb[i] = auc(fpr_xgb[i], tpr_xgb[i])

# Plot ROC curves for each class for all models
plt.figure(figsize=(10,5), dpi=350)

# Define colors and linestyles for each model
```

```
colors = ['blue', 'red', 'green']
linestyles = ['-', '--', ':']

for model, color in zip([best_ltv_rf_model, best_ltv_svm_model, best_ltv_xgb_model], colors):
    for i, linestyle in zip(range(n_classes), linestyles):
        if model == best_ltv_rf_model:
            fpr = fpr_rf[i]
            tpr = tpr_rf[i]
            roc_auc = roc_auc_rf[i]
            label = 'Random Forest (class {0} AUC = {1:0.2f})'.format(i, roc_auc)
        elif model == best_ltv_svm_model:
            fpr = fpr_svm[i]
            tpr = tpr_svm[i]
            roc_auc = roc_auc_svm[i]
            label = 'SVM (class {0} AUC = {1:0.2f})'.format(i, roc_auc)
        else:
            fpr = fpr_xgb[i]
            tpr = tpr_xgb[i]
            roc_auc = roc_auc_xgb[i]
            label = 'XGBoost (class {0} AUC = {1:0.2f})'.format(i, roc_auc)
        plt.plot(fpr, tpr, color=color, lw=1, linestyle=linestyle, label=label)

plt.plot([0, 1], [0, 1], 'k--', lw=1)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate', size =14)
plt.ylabel('True Positive Rate', size = 14)
plt.title('ROC curves for multi-class classification of CLV models', size = 18)
plt.legend(loc="lower right", prop={"size": 10})
plt.show()
```

In [ ]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score

# Make predictions on the test set
y_pred = best_ltv_xgb_model.predict(X_test)

# Calculate assessment scores
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
```



```
# Print out the scores
print('Accuracy: {:.3f}'.format(accuracy))
print('Precision: {:.3f}'.format(precision))
print('Recall: {:.3f}'.format(recall))
print('F1 Score: {:.3f}'.format(f1))
```

#

Customer Purchase Prediction using Machine Learning Models.

In [ ]:

```
# Import data
import pandas as pd

# Read excel file
df_retail2 = pd.read_csv('C:/Users/Darius/Desktop/PROJECT//online_retail_II.csv')


df_retail2 = df_retail2.rename(columns={'Customer ID': 'CustomerID'})

# Convert 'InvoiceDate' column to datetime format
# and display information about the 'df_retail' DataFrame
df_retail2['InvoiceDate'] = pd.to_datetime(df_retail2['InvoiceDate'])

# extract only the date part and convert it to date data type
df_retail2['InvoiceDate'] = df_retail2['InvoiceDate'].dt.date


tx_uk = df_retail2.query("Country == 'United Kingdom'").reset_index()
```

CHOOSE DATA FOR TRAINING

In [ ]:

```
import pandas as pd

# Convert the Python date objects to Pandas datetime objects
start_date = pd.to_datetime('2010-03-01')
end_date = pd.to_datetime('2011-06-01')

# Filter 3 months from 03/2011 to 06/2011
```

```
tx_q2 = tx_uk[(tx_uk.InvoiceDate < end_date) & (tx_uk.InvoiceDate >= start_date)].reset_index(drop = True)
tx_q2
```

CHOOSE DATA FOR TEST SET

In [ ]:
```
# Convert the Python date objects to Pandas datetime objects
start_date = pd.to_datetime('2011-06-01')
end_date = pd.to_datetime('2011-12-01')

# Filter 6 months from 06/2011 to 12/2011
tx_next = tx_uk[(tx_uk.InvoiceDate < end_date) & (tx_uk.InvoiceDate >= start_date)].reset_index(drop = True)
tx_next
```

In [ ]:
```
# Describe the tx_next dataframe
tx_next['InvoiceDate'].describe()
```

In [ ]:
```
# Create a new DataFrame called tx_user that contains a list of unique customer IDs from the tx_q2 DataFrame
tx_user = pd.DataFrame(tx_q2['CustomerID'].unique())

# Rename the column in tx_user to 'CustomerID'
tx_user.columns =['CustomerID']

# Create a new DataFrame called tx_next_first_purchase that contains the earliest purchase date for each customer in the tx_next
tx_next_first_purchase = tx_next.groupby('CustomerID').InvoiceDate.min().reset_index()

# Rename the columns in tx_next_first_purchase to 'CustomerID' and 'MinPurchaseDate'
tx_next_first_purchase.columns = ['CustomerID', 'MinPurchaseDate']

# Print the first few rows of the tx_next_first_purchase DataFrame
tx_next_first_purchase.head()
```

In [ ]:
```
# Create a new DataFrame called tx_last_purchase that contains the latest purchase date for each customer in the tx_q2 DataFrame
tx_last_purchase = tx_q2.groupby('CustomerID').InvoiceDate.max().reset_index()

# Rename the columns in tx_last_purchase to 'CustomerID' and 'MaxPurchaseDate'
tx_last_purchase.columns = ['CustomerID', 'MaxPurchaseDate']
```

In [ ]:
```
# Merge the tx_last_purchase and tx_next_first_purchase DataFrames on the 'CustomerID' column using a left join
tx_purchase_dates = pd.merge(tx_last_purchase, tx_next_first_purchase, on ='CustomerID', how = 'left')
```


In [ ]:
```
tx_purchase_dates
```

In [ ]:
```
# Calculate the number of days between the latest purchase date and the earliest purchase date for each customer
tx_purchase_dates['NextPurchaseDay'] = (tx_purchase_dates['MinPurchaseDate'] - tx_purchase_dates['MaxPurchaseDate']).dt.days
```

In [ ]:
```
tx_purchase_dates
```

In [ ]:
```
# Merge the tx_user DataFrame with the tx_purchase_dates DataFrame on the 'CustomerID' column using a left join
tx_user = pd.merge(tx_user, tx_purchase_dates[['CustomerID','NextPurchaseDay']],  on = 'CustomerID', how ='left')

# Show the first 5 rows of the resulting DataFrame
tx_user.head()
```

In [ ]:
```
# Replace missing values (NaNs) in the 'NextPurchaseDay' column with 999
tx_user =tx_user.fillna(999)
```

In [ ]:
```
# Calculate the number of days between each customer's last purchase and the end of the 12-month transaction period, and add the
tx_last_purchase['Recency'] = (tx_last_purchase['MaxPurchaseDate'].max() - tx_last_purchase['MaxPurchaseDate']).dt.days

# Merge the tx_user DataFrame with the tx_last_purchase DataFrame on the 'CustomerID' column
tx_user = pd.merge(tx_user, tx_last_purchase[['CustomerID', 'Recency']], on = 'CustomerID')

# Show the first 5 rows of the resulting DataFrame
tx_user.head()
```

In [ ]:
```
#describe data
tx_user.Recency.describe()
```

RUN Kmean again for Frequency, Recency and Monetary

Clustering Recency = 4 (again)

In [ ]:
```
# Import necessary modules
import plotly.graph_objs as go
import plotly.offline as pyoff
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters = 4)
kmeans.fit(tx_user[['Recency']])
```

```
tx_user['RecencyCluster'] = kmeans.predict(tx_user[['Recency']])

def order_cluster(cluster_field_name, target_field_name,df,ascending):
    new_cluster_field_name = 'new_' + cluster_field_name
    df_new = df.groupby(cluster_field_name)[target_field_name].mean().reset_index()
    df_new = df_new.sort_values(by = target_field_name, ascending = ascending).reset_index(drop = True)
    df_new['index'] = df_new.index
    df_final = pd.merge(df,df_new[[cluster_field_name,'index']], on=cluster_field_name)
    df_final = df_final.drop([cluster_field_name], axis = 1)
    df_final = df_final.rename(columns = {"index": cluster_field_name})
    return df_final

print(tx_user)
tx_user = order_cluster('RecencyCluster', 'Recency', tx_user, False)
tx_user.groupby('RecencyCluster')['Recency'].describe()
```

Clustering Frequency = 3 (again)

In [ ]:
```
# Frequency
tx_frequency = tx_q2.groupby('CustomerID').InvoiceDate.count().reset_index()
tx_frequency.columns = ['CustomerID', 'Frequency']
tx_frequency.head()

tx_user = pd.merge(tx_user, tx_frequency, on = 'CustomerID')
tx_user.head()
tx_user.Frequency.describe()

#kmeans
kmeans = KMeans(n_clusters = 3)
kmeans.fit(tx_user[['Frequency']])
tx_user['FrequencyCluster'] = kmeans.predict(tx_user[['Frequency']])

tx_user = order_cluster('FrequencyCluster', 'Frequency', tx_user, True)
tx_user.groupby('FrequencyCluster')['Frequency'].describe()
```

Clustering Monetary = 3 (again)

In [ ]:
```
# Monetary Value

tx_q2['Revenue'] = tx_q2['Price']*tx_q2['Quantity']
tx_revenue = tx_q2.groupby('CustomerID').Revenue.sum().reset_index()
```



```
tx_user = pd.merge(tx_user, tx_revenue, on = 'CustomerID', how='left')
tx_user.head()
tx_user.Revenue.describe()
#kmeans
kmeans = KMeans(n_clusters = 3)
kmeans.fit(tx_user[['Revenue']])
tx_user['RevenueCluster'] = kmeans.predict(tx_user[['Revenue']])

tx_user = order_cluster('RevenueCluster', 'Revenue', tx_user, True)
tx_user.groupby('RevenueCluster')['Revenue'].describe()
```

Overall Score

In [ ]:
```
#Overall Score

tx_user['OverallScore'] = tx_user['RecencyCluster'] + tx_user['FrequencyCluster']+tx_user['RevenueCluster']
tx_user.groupby('OverallScore')['Recency','Frequency','Revenue'].mean()

tx_user.groupby('OverallScore')['Recency'].count()

tx_user['Segment'] = 'Low-Value'
tx_user.loc[tx_user['OverallScore'] > 2, 'Segment'] = 'Mid-Value'
tx_user.loc[tx_user['OverallScore'] > 4, 'Segment'] = 'High-Value'
```

In [ ]:
```
tx_user
```

ADDING NEW FEATURES TO MODEL

In [ ]:
```
# Adding new Features
# Create a dataframe with CustomerID, and Invoice Date
tx_day_order = tx_q2[['CustomerID', 'InvoiceDate']]

# Convert Invoice Date to day
tx_day_order['InvoiceDay'] = pd.to_datetime(tx_q2['InvoiceDate']).dt.date

# Sort by CustomerID and InvoiceDate
tx_day_order = tx_day_order.sort_values(['CustomerID', 'InvoiceDate'])

# Drop Duplicates (If Customer buy more than 2 times in a day)
```

```
tx_day_order = tx_day_order.drop_duplicates(subset = ['CustomerID', 'InvoiceDay'], keep = 'first')

# Shifting last 3 purchase dates
tx_day_order['PrevInvoiceDate'] = tx_day_order.groupby('CustomerID')['InvoiceDay'].shift(1)
tx_day_order['T2InvoiceDate'] = tx_day_order.groupby('CustomerID')['InvoiceDay'].shift(2)
tx_day_order['T3InvoiceDate'] = tx_day_order.groupby('CustomerID')['InvoiceDay'].shift(3)

# Display the first 5 rows of the resulting dataframe
tx_day_order.head()
```

In [ ]:
```
# We will base on 4 nearest purchasing days to build the model
# Calculate the number of days between the current purchase and the previous purchase
tx_day_order['DayDiff'] = (tx_day_order['InvoiceDay'] - tx_day_order['PrevInvoiceDate']).dt.days

# Calculate the number of days between the current purchase and the purchase 2nd ago
tx_day_order['DayDiff2'] = (tx_day_order['InvoiceDay'] - tx_day_order['T2InvoiceDate']).dt.days

# Calculate the number of days between the current purchase and the purchase 3rd ago
tx_day_order['DayDiff3'] = (tx_day_order['InvoiceDay'] - tx_day_order['T3InvoiceDate']).dt.days

# Display the first few rows of the dataframe
tx_day_order.head()
```

In [ ]:
```
# Calculate mean and standard deviation of time between purchases for each customer
tx_day_diff = tx_day_order.groupby('CustomerID').agg({'DayDiff': ['mean', 'std']}).reset_index()

# Rename columns for clarity
tx_day_diff.columns = ['CustomerID', 'DayDiffMean', 'DayDiffStd']

# Display the first few rows of the resulting dataframe
tx_day_diff.head()
```

In [ ]:
```
tx_day_order_last = tx_day_order.drop_duplicates(subset = ['CustomerID'], keep ='last')
tx_day_order_last.head()
```

In [ ]:
```
# Drop any rows with missing values in the tx_day_order_last dataframe
tx_day_order_last = tx_day_order_last.dropna()

# Merge the tx_day_order_last and tx_day_diff dataframes on the 'CustomerID' column
tx_day_order_last = pd.merge(tx_day_order_last, tx_day_diff, on='CustomerID')

# Select specific columns from tx_day_order_last and merge with tx_user on the 'CustomerID' column
```



```
tx_user = pd.merge(tx_user, tx_day_order_last[['CustomerID', 'DayDiff', 'DayDiff2', 'DayDiff3', 'DayDiffMean', 'DayDiffStd']], o

# Preview the first few rows of the merged tx_user dataframe
tx_user.head()
```

In [ ]:
```
tx_day_order_last
```

In [ ]:
```
# Import the pandas library
import pandas as pd

# Create a copy of the tx_user dataframe and assign it to tx_class
tx_class = tx_user.copy()

# Convert categorical variables in tx_class to binary/dummy variables
tx_class = pd.get_dummies(tx_class)

# Print information about the tx_class dataframe, including column names and data types
tx_class.info()
```

Setting Classes

In [ ]:
```
# Assign a value of 2 to the 'NextPurchaseDayRange' column for all rows in the tx_class dataframe
tx_class['NextPurchaseDayRange'] = 2

# For rows where the 'NextPurchaseDay' column is greater than 90, set the value of 'NextPurchaseDayRange' to 1
tx_class.loc[tx_class.NextPurchaseDay > 21, 'NextPurchaseDayRange'] = 1

# For rows where the 'NextPurchaseDay' column is greater than 180, set the value of 'NextPurchaseDayRange' to 0
tx_class.loc[tx_class.NextPurchaseDay > 180, 'NextPurchaseDayRange'] = 0

# Display the count of unique values in the 'NextPurchaseDayRange' column of the tx_class dataframe
tx_class.NextPurchaseDayRange.value_counts()
```

In [ ]:
```
# Import the matplotlib and seaborn libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Compute the correlation matrix for all columns in the tx_class dataframe
corr = tx_class[tx_class.columns].corr()

# Set the size of the figure to be plotted
```

```
plt.figure(figsize = (30,20))

# Plot a heatmap of the correlation matrix using seaborn
# Display the correlation values as annotations on the heatmap
# Set the linewidth of the heatmap to 0.2
# Format the correlation values to display 2 decimal places
sns.heatmap(corr, annot =True, linewidths = 0.2, fmt = '.2f')
```

Prepare Data for models

In [ ]:
```
# Remove the 'NextPurchaseDay' column from the tx_class dataframe using the drop() method
# The 'axis' parameter is set to 1 to indicate that the column should be dropped (axis 0 would indicate a row)
tx_class = tx_class.drop('NextPurchaseDay', axis = 1)

# Separate the features (X) and target variable (y) from the tx_class dataframe
# The features are all columns except for 'NextPurchaseDayRange', which is the target variable
X, y = tx_class.drop('NextPurchaseDayRange', axis = 1), tx_class.NextPurchaseDayRange
```

In [ ]:
```
tx_class
```

In [ ]:
```
X
```

XGBOOST Model

SPLIT TRAIN, VALIDATION AND TEST SETS

In [ ]:
```
 # Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.2, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=26)

# Fit the XGBClassifier model to the training set
next_purchase_xgb_model_split = xgb.XGBClassifier(maxdepth=3, learning_rate=0.001, objective='multi:softprob', n_jobs=-1).fit(X_

# Print the accuracy of the model on the training, validation, and test sets
print('Accuracy of XGB Classifier on training set: {:.2f}'.format(next_purchase_xgb_model_split.score(X_train, y_train)))
print('Accuracy of XGB Classifier on validation set : {:.2f}'.format(next_purchase_xgb_model_split.score(X_val[X_train.columns],
print('Accuracy of XGB Classifier on test set : {:.2f}'.format(next_purchase_xgb_model_split.score(X_test[X_train.columns], y_te
```


Result after split train XGBOOST

In [ ]:
```
from sklearn.metrics import classification_report

y_pred = next_purchase_xgb_model_split.predict(X_test)
y_train_pred = next_purchase_xgb_model_split.predict(X_train)

print("Classification Report of XGBOOST model for Test Data:")
print(classification_report(y_test, y_pred))

print("Classification Report for Train Data:")
print(classification_report(y_train, y_train_pred))
```

GridSearchCV for XGBoost

In [ ]:
```
from sklearn.model_selection import GridSearchCV

# define parameters to be tested in the grid search
params = {'max_depth': [3, 5, 7],
          'learning_rate': [0.1, 0.01, 0.001],
          'n_estimators': [50, 100, 200,400],
          'objective': ['multi:softprob']}

# create an XGBClassifier object
xgb_model = xgb.XGBClassifier(n_jobs=-1)

# create a GridSearchCV object
grid_search = GridSearchCV(estimator=xgb_model, param_grid=params, cv=7)

# fit the GridSearchCV object to the training data
grid_search.fit(X_train, y_train)

# print the best parameters and score
print("Best parameters: ", grid_search.best_params_)
print("Best score: ", grid_search.best_score_)
```

Fit XGBOOST model with best parameters

In [ ]:
```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
```

```
# Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.2, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=26)

# Fit the model with the best parameters
best_xgb_model = xgb.XGBClassifier(max_depth=3, learning_rate=0.1, n_estimators=100, objective='multi:softprob', n_jobs=-1).fit(

# Print the accuracies on the training, validation, and test sets
print('Accuracy of XGB Classifier on training set: {:.2f}'.format(best_xgb_model.score(X_train, y_train)))
print('Accuracy of XGB Classifier on validation set : {:.2f}'.format(best_xgb_model.score(X_val[X_train.columns], y_val)))
print('Accuracy of XGB Classifier on test set : {:.2f}'.format(best_xgb_model.score(X_test[X_train.columns], y_test)))
```

RESULT AFTER TUNING XGBOOST

In [ ]:
```
from sklearn.metrics import classification_report

y_pred = best_xgb_model.predict(X_test)
y_train_pred = best_xgb_model.predict(X_train)

print("Classification Report of XGBOOST after tuning for Test Data:")
print(classification_report(y_test, y_pred))

print("Classification Report for Train Data:")
print(classification_report(y_train, y_train_pred))
```

PLOT DISTRIBUTION BY NEXTPURCHASERANGE

In [ ]:
```
import matplotlib.pyplot as plt

# Calculate the percentage of customers in each LTVCluster
NextPurchaseDayRange = tx_class.groupby('NextPurchaseDayRange').CustomerID.count()/tx_class.CustomerID.count()

# Create a figure and axis object
fig, ax = plt.subplots(figsize=(12,8), dpi=300)

# Plot the percentages as a bar chart with adjusted width
ax.bar(NextPurchaseDayRange.index, NextPurchaseDayRange.values, width=0.6, color='#33AFFF', edgecolor='black')

# Set the xticks and yticks font sizes
plt.xticks(fontsize=22)
```



```
plt.yticks(fontsize=22)

# Add labels and title to the plot
ax.set_xlabel('NextPurchaseDayRange Cluster', fontsize=22)
ax.set_ylabel('Percentage of Customers', fontsize=22)
ax.set_title('Customer Distribution by NextPurchaseDayRange', fontsize=28)

# Add grid lines and remove the top and right spines
ax.grid(axis='y', linestyle='--', alpha=0.7)
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)

# Set the x-axis tick labels to (0, 1, 2)
ax.set_xticks([0, 1, 2])
ax.set_xticklabels(['Cluster 0', 'Cluster 1', 'Cluster 2'], fontsize=18)

# Show the percentage value for each bar
for i, v in enumerate(NextPurchaseDayRange.values):
    ax.text(i, v+0.01, str(round(v*100,2))+'%', ha='center', fontsize=18)

# Save and show the plot
plt.savefig('customer_distribution.png', dpi=300, bbox_inches='tight')
plt.show()
```

SUPPORT VECTOR MACHINE

In [ ]:
```
from sklearn.model_selection import train_test_split

# Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.2, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=26)

# create SVM classifier object
svm_model_split = SVC(kernel='rbf', C=1, random_state=24)

# train SVM classifier on the training data
svm_model_split.fit(X_train, y_train)

# predict the labels of validation and test data using trained SVM classifier
y_val_pred = svm_model_split.predict(X_val)
y_test_pred = svm_model_split.predict(X_test)
```

```
# compute the accuracy score of SVM classifier on validation and test set
val_accuracy = svm_model_split.score(X_val, y_val)
test_accuracy = svm_model_split.score(X_test, y_test)

print('Accuracy of SVM Classifier on training set: {:.2f}'.format(svm_model_split.score(X_train, y_train)))
print('Accuracy of SVM Classifier on validation set : {:.2f}'.format(val_accuracy))
print('Accuracy of SVM Classifier on test set : {:.2f}'.format(test_accuracy))
```

Result model

In [ ]:
```
from sklearn.metrics import classification_report

y_pred = svm_model_split.predict(X_test)
y_train_pred = svm_model_split.predict(X_train)

print("Classification Report of SVM model for Test Data:")
print(classification_report(y_test, y_pred))

print("Classification Report for Train Data:")
print(classification_report(y_train, y_train_pred))
```

GRIDSEARCHCV FOR SVM

In [ ]:
```
from sklearn.model_selection import GridSearchCV

# define parameter grid
param_grid = {'C': [0.1, 1, 10, 100],
              'kernel': ['linear', 'poly', 'rbf', 'sigmoid']}

# create SVM classifier object
svm_model = SVC(random_state=24)

# create GridSearchCV object
grid_search = GridSearchCV(svm_model, param_grid, cv=7, n_jobs=-1)

# fit GridSearchCV object to training data
grid_search.fit(X_trainval, y_trainval)

# print best hyperparameters
print('Best hyperparameters: ', grid_search.best_params_)
```



```
# predict the labels of validation and test data using best SVM classifier
best_svm_model = grid_search.best_estimator_
y_val_pred = best_svm_model.predict(X_val)
y_test_pred = best_svm_model.predict(X_test)

# compute the accuracy score of best SVM classifier on validation and test set
val_accuracy = best_svm_model.score(X_val, y_val)
test_accuracy = best_svm_model.score(X_test, y_test)

print('Accuracy of best SVM Classifier on validation set : {:.2f}'.format(val_accuracy))
print('Accuracy of best SVM Classifier on test set : {:.2f}'.format(test_accuracy))
```

Fit SVM after Tunning

In [ ]:
```
# Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.2, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=24)

# Fit the model with the best parameters
best_svm_model = SVC(kernel='rbf', C=100, random_state=24)

# train SVM classifier on the training data
best_svm_model.fit(X_train, y_train)

# Print the accuracies on the training, validation, and test sets
print('Accuracy of SVM Classifier on training set: {:.2f}'.format(best_svm_model.score(X_train, y_train)))
print('Accuracy of SVM Classifier on validation set : {:.2f}'.format(best_svm_model.score(X_val[X_train.columns], y_val)))
print('Accuracy of SVM Classifier on test set : {:.2f}'.format(best_svm_model.score(X_test[X_train.columns], y_test)))
```

Result after tuning SVM

In [ ]:
```
from sklearn.metrics import classification_report

y_pred = best_svm_model.predict(X_test)
y_train_pred = best_svm_model.predict(X_train)

print("Classification Report of SVM after-tunning parameter:")
print(classification_report(y_test, y_pred))

print("Classification Report for Train Data:")
print(classification_report(y_train, y_train_pred))
```

RANDOM FOREST

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Split data into 80% training, 10% validation, and 10% test sets
X_trainval, X_test, y_trainval, y_test = train_test_split(X, y, test_size=0.2, random_state=24)
X_train, X_val, y_train, y_val = train_test_split(X_trainval, y_trainval, test_size=0.125, random_state=26)

# create random forest classifier object
rf_model = RandomForestClassifier(n_estimators=500, max_depth=3, random_state=24)

# train random forest classifier on the training data
rf_model.fit(X_train, y_train)

# predict the labels of validation and test data using trained random forest classifier
y_val_pred = rf_model.predict(X_val)
y_test_pred = rf_model.predict(X_test)

# compute the accuracy score of random forest classifier on validation and test set
val_accuracy = rf_model.score(X_val, y_val)
test_accuracy = rf_model.score(X_test, y_test)

print('Accuracy of Random Forest Classifier on training set: {:.2f}'.format(rf_model.score(X_train, y_train)))
print('Accuracy of Random Forest Classifier on validation set : {:.2f}'.format(val_accuracy))
print('Accuracy of Random Forest Classifier on test set : {:.2f}'.format(test_accuracy))
```

Result of Random Forest Model

In [ ]:
```
y_pred = rf_model.predict(X_test)
print("Classification Report of Random Forest:")
print(classification_report(y_test, y_pred))
```

GridSearchCV for Random Forest

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV
# define parameter grid
param_grid = {
```



```
    'n_estimators': [100, 200, 500],
    'max_depth': [3, 5,7, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
# create random forest classifier object
rf_model = RandomForestClassifier(random_state=24)

# create GridSearchCV object
grid_search = GridSearchCV(rf_model, param_grid, cv=7, n_jobs=-1)

# fit GridSearchCV object to training data
grid_search.fit(X_trainval, y_trainval)

# print best hyperparameters
print('Best hyperparameters: ', grid_search.best_params_)
```

Result after tuning Random Forest

In [ ]:
```
best_rf_model = RandomForestClassifier(max_depth=7, min_samples_leaf=2, min_samples_split=10, n_estimators=500)
best_rf_model.fit(X_train, y_train)
```

In [ ]:
```
y_pred = best_rf_model.predict(X_test)
print("Classification Report of Random Forest after tunning parameter:")
print(classification_report(y_test, y_pred))
```

COMPARISON BETWEEN 3 MODELS

CONFUSION MATRICES

In [ ]:
```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay

# Set the DPI for the plot
plt.rcParams['figure.dpi'] = 300

# create a list of models and their names
models = [('Random Forest', best_rf_model),('SVM', best_svm_model),('XGBoost', best_xgb_model)]
```

```
# Loop through each model and plot its confusion matrix
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

for i, (name, model) in enumerate(models):
    y_pred = model.predict(X_test)
    cm = confusion_matrix(y_test, y_pred)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Not buying', '6 Months Purchase', 'Next Month Purchase']
    disp.plot(cmap=plt.cm.Blues, ax=axes[i])
    disp.ax_.set_title('Confusion Matrix for {}'.format(name), fontsize=14)
    disp.im_.colorbar.remove()
    axes[i].set_xticklabels(['Not buying', '6 Months Purchase', 'Next Month Purchase'], rotation=45)

# Add a common colorbar for all subplots
fig.subplots_adjust(wspace=0.3, top=0.8)
cbar_ax = fig.add_axes([0.25, -0.03, 0.5, 0.02])
fig.colorbar(disp.im_, cax=cbar_ax, orientation='horizontal')

# Add a title and adjust spacing
fig.suptitle('Confusion Matrices for Predicting Next Purchase Period Models', fontsize=24, y=0.98)
fig.tight_layout(pad=2.5)
plt.show()
```

ROC AND AUC

In [ ]:

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle

# Convert y_test to binary format
y_test_binary = label_binarize(y_test, classes=[0, 1, 2])
n_classes = y_test_binary.shape[1]

# Compute ROC curve and ROC area for each class for Random Forest model
fpr_rf = dict()
tpr_rf = dict()
roc_auc_rf = dict()
y_score_rf = best_rf_model.predict_proba(X_test)
for i in range(n_classes):
    fpr_rf[i], tpr_rf[i], _ = roc_curve(y_test_binary[:, i], y_score_rf[:, i])
    roc_auc_rf[i] = auc(fpr_rf[i], tpr_rf[i])
```



```
# Compute ROC curve and ROC area for each class for SVM model
fpr_svm = dict()
tpr_svm = dict()
roc_auc_svm = dict()
y_score_svm = best_svm_model.decision_function(X_test)
for i in range(n_classes):
    fpr_svm[i], tpr_svm[i], _ = roc_curve(y_test_binary[:, i], y_score_svm[:, i])
    roc_auc_svm[i] = auc(fpr_svm[i], tpr_svm[i])

# Compute ROC curve and ROC area for each class for XGBoost model
fpr_xgb = dict()
tpr_xgb = dict()
roc_auc_xgb = dict()
y_score_xgb = best_xgb_model.predict_proba(X_test)
for i in range(n_classes):
    fpr_xgb[i], tpr_xgb[i], _ = roc_curve(y_test_binary[:, i], y_score_xgb[:, i])
    roc_auc_xgb[i] = auc(fpr_xgb[i], tpr_xgb[i])

# Plot ROC curves for each class for all models
plt.figure(figsize=(10,5), dpi=350)

# Define colors and linestyles for each model
colors = ['blue', 'red', 'green']
linestyles = ['-', '--', ':']

for model, color in zip([best_rf_model, best_svm_model, best_xgb_model], colors):
    for i, linestyle in zip(range(n_classes), linestyles):
        if model == best_rf_model:
            fpr = fpr_rf[i]
            tpr = tpr_rf[i]
            roc_auc = roc_auc_rf[i]
            label = 'Random Forest (class {0} AUC = {1:0.2f})'.format(i, roc_auc)
        elif model == best_svm_model:
            fpr = fpr_svm[i]
            tpr = tpr_svm[i]
            roc_auc = roc_auc_svm[i]
            label = 'SVM (class {0} AUC = {1:0.2f})'.format(i, roc_auc)
        else:
            fpr = fpr_xgb[i]
            tpr = tpr_xgb[i]
            roc_auc = roc_auc_xgb[i]
            label = 'XGBoost (class {0} AUC = {1:0.2f})'.format(i, roc_auc)
        plt.plot(fpr, tpr, color=color, lw=1, linestyle=linestyle, label=label)
```

```
plt.plot([0, 1], [0, 1], 'k--', lw=1)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate', size =14)
plt.ylabel('True Positive Rate', size = 14)
plt.title('ROC curves for multi-class classification of Next Periods Purchase Models', size = 18)
plt.legend(loc="lower right", prop={"size": 10})
plt.show()
```

ASSESSMENT SCORES

In [ ]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score

models = [('XGBoost', best_xgb_model), ('Random Forest', best_rf_model), ('SVM', best_svm_model)]

for name, model in models:
    # Make predictions on the test set
    y_pred = model.predict(X_test)

    # Calculate assessment scores
    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='weighted')
    recall = recall_score(y_test, y_pred, average='weighted')
    f1 = f1_score(y_test, y_pred, average='weighted')

    # Print out the scores
    print('Assessment Scores for', name)
    print('------------------------')
    print('Accuracy: {:.3f}'.format(accuracy))
    print('Precision: {:.3f}'.format(precision))
    print('Recall: {:.3f}'.format(recall))
    print('F1 Score: {:.3f}'.format(f1))
    print('\n')
```