Customer Segmentation
Unsupervised Machine Learning

Problem Statement: This is a transactional data set which contains all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail. The company mainly sells unique all-occasion gifts. Many customers of the company are wholesalers.

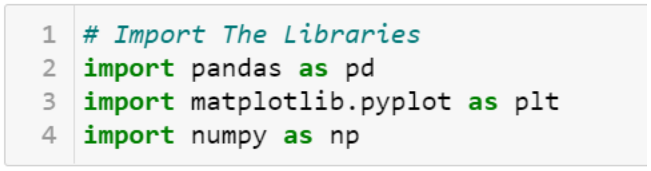
Requirement: Based on this data you are required to build features and model on these characteristics of users to categorize users based on their transactions . Your model will be evaluated on these criteria:

- Feature Engineering (Variable Imputation)
- Model Selection Criteria (Basis of choosing the final Technique)
- Measurement Criteria (Comparison of Various Models)
- Scope for improvement

Approach:

- Step 1: Data gathering
- Step 2: Create Recency Frequency Monetary (RFM) table
- Step 3: Manage skewness and scale each variable
- Step 4: Explore the data
- Step 5: Cluster the data
- Step 6: Interpret the result

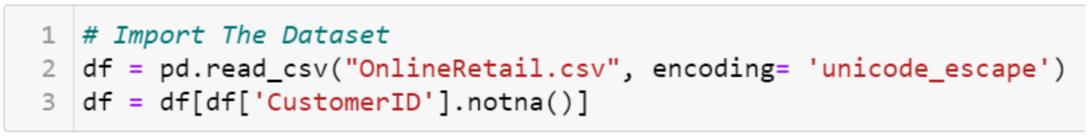
Prerequisites: Import the required libraries



```python
# Import The Libraries
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

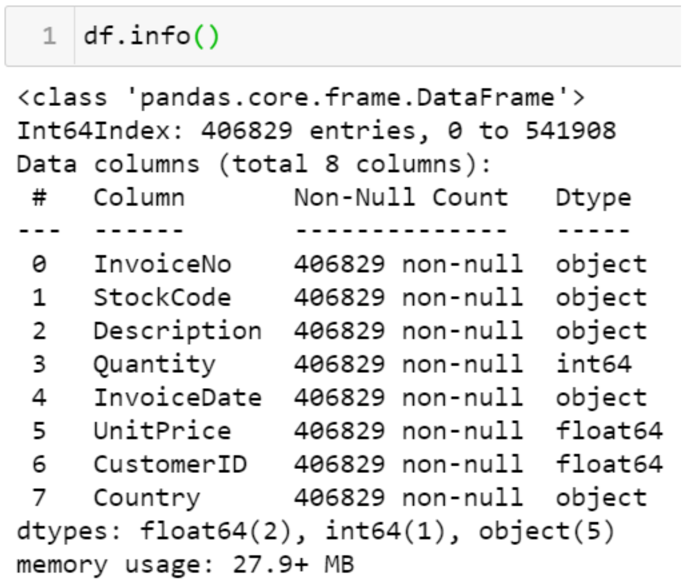
Step 1: Data gathering/Importing the data

1. Import the data



```python
# Import The Dataset
df = pd.read_csv("OnlineRetail.csv", encoding= 'unicode_escape')
df = df[df['CustomerID'].notna()]
```

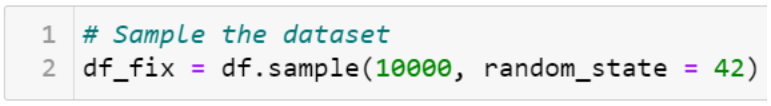
2. df.info() shows there are 541909 and 8 columns in the dataset.



```python
df.info()
```

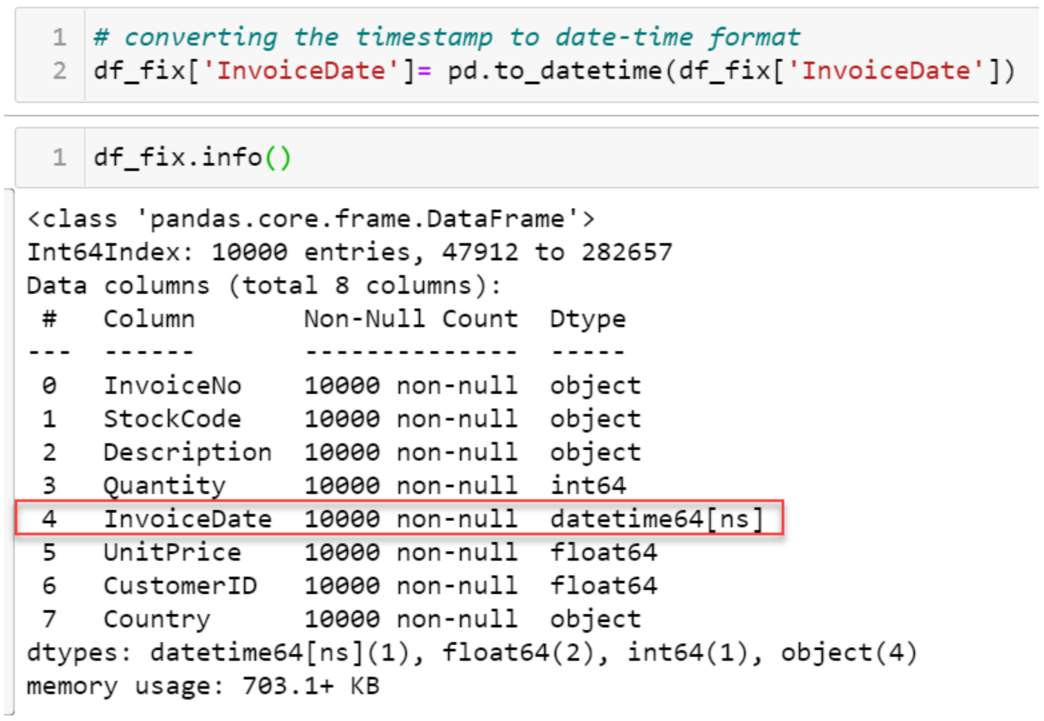
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 406829 entries, 0 to 541908
Data columns (total 8 columns):
 #   Column       Non-Null Count   Dtype  
---  ------       --------------   -----  
 0   InvoiceNo    406829 non-null  object 
 1   StockCode    406829 non-null  object 
 2   Description  406829 non-null  object 
 3   Quantity     406829 non-null  int64  
 4   InvoiceDate  406829 non-null  object 
 5   UnitPrice    406829 non-null  float64
 6   CustomerID   406829 non-null  float64
 7   Country      406829 non-null  object 
dtypes: float64(2), int64(1), object(5)
memory usage: 27.9+ MB
```

3. For this case, we don't use all of the rows. Instead, we will sample 10000 rows from the dataset, and we assume that as the whole transactions that the customers do. The code will look like this,



```python
# Sample the dataset
df_fix = df.sample(10000, random_state = 42)
```

4. Convert the InvoiceDate to date-time format.

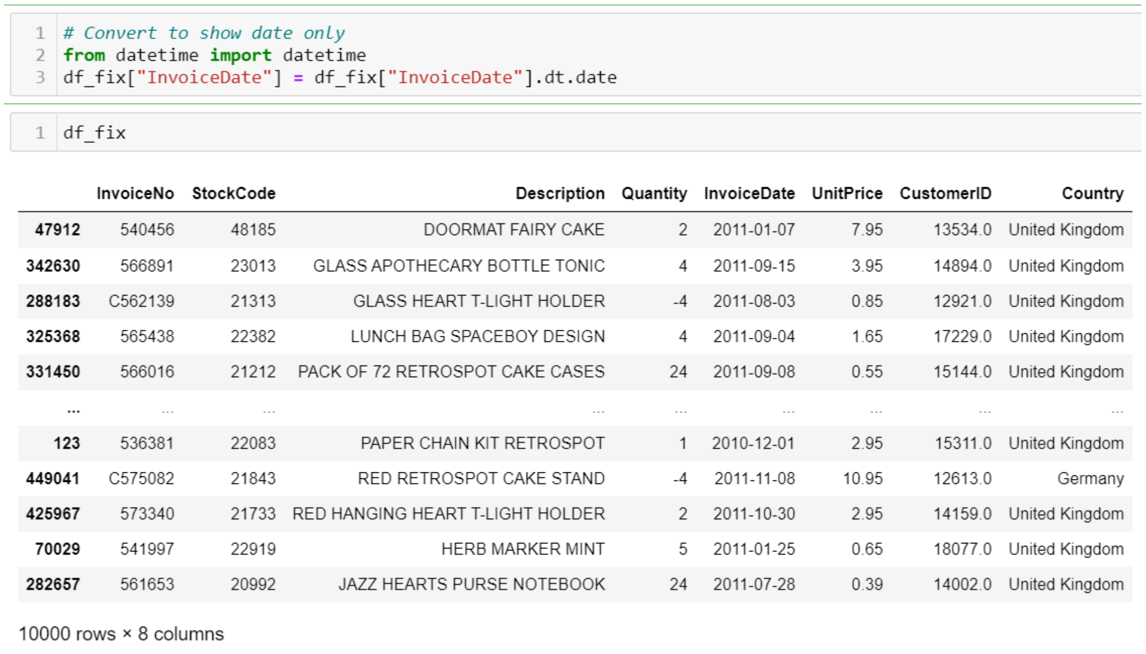


```python
# converting the timestamp to date-time format
df_fix['InvoiceDate']= pd.to_datetime(df_fix['InvoiceDate'])
```

```python
df_fix.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 10000 entries, 47912 to 282657
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   InvoiceNo    10000 non-null  object
 1   StockCode    10000 non-null  object
 2   Description  10000 non-null  object
 3   Quantity     10000 non-null  int64
 4   InvoiceDate  10000 non-null  datetime64[ns]
 5   UnitPrice    10000 non-null  float64
 6   CustomerID   10000 non-null  float64
 7   Country      10000 non-null  object
dtypes: datetime64[ns](1), float64(2), int64(1), object(4)
memory usage: 703.1+ KB
```

5. Convert the InvoiceDate column to show date only.



```python
# Convert to show date only
from datetime import datetime
df_fix["InvoiceDate"] = df_fix["InvoiceDate"].dt.date
```

```python
df_fix
```

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
47912	540456	48185	DOORMAT FAIRY CAKE	2	2011-01-07	7.95	13534.0	United Kingdom
342630	566891	23013	GLASS APOTHECARY BOTTLE TONIC	4	2011-09-15	3.95	14894.0	United Kingdom
288183	C562139	21313	GLASS HEART T-LIGHT HOLDER	-4	2011-08-03	0.85	12921.0	United Kingdom
325368	565438	22382	LUNCH BAG SPACEBOY DESIGN	4	2011-09-04	1.65	17229.0	United Kingdom
331450	566016	21212	PACK OF 72 RETROSPOT CAKE CASES	24	2011-09-08	0.55	15144.0	United Kingdom
...	...	...	...	...	...	...	...	...
123	536381	22083	PAPER CHAIN KIT RETROSPOT	1	2010-12-01	2.95	15311.0	United Kingdom
449041	C575082	21843	RED RETROSPOT CAKE STAND	-4	2011-11-08	10.95	12613.0	Germany
425967	573340	21733	RED HANGING HEART T-LIGHT HOLDER	2	2011-10-30	2.95	14159.0	United Kingdom
70029	541997	22919	HERB MARKER MINT	5	2011-01-25	0.65	18077.0	United Kingdom
282657	561653	20992	JAZZ HEARTS PURSE NOTEBOOK	24	2011-07-28	0.39	14002.0	United Kingdom

10000 rows × 8 columns

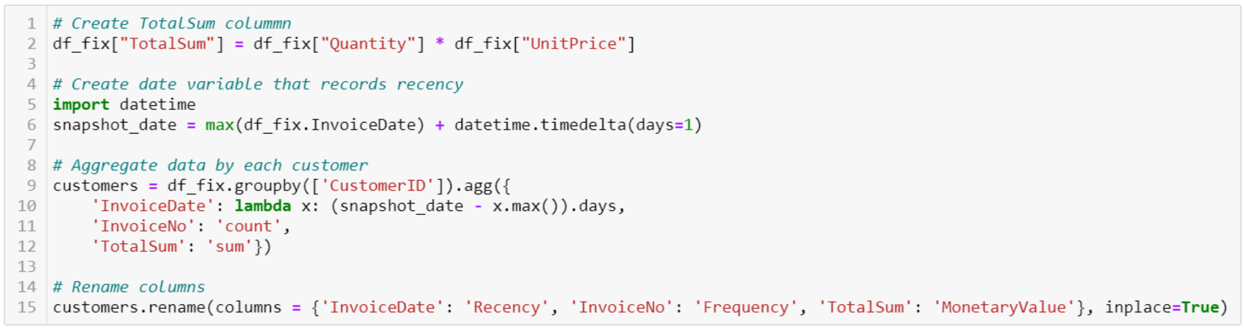
Step 2: Create Recency Frequency Monetary (RFM) table

After we sample the data, we will make the data easier to conduct an analysis. To segmenting customer, there are some metrics that we can use, such as when the customer buy the product for last time, how frequent the customer buy the product, and how much the customer pays for the product. We will call this segmentation as RFM segmentation.

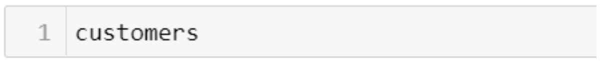
To make the RFM table, we can create these columns, such as Recency, Frequency, and MonetaryValue column. To get the number of days for recency column, we can subtract the snapshot date with the date where the transaction occurred.

To create the frequency column, we can count how much transactions by each customer.

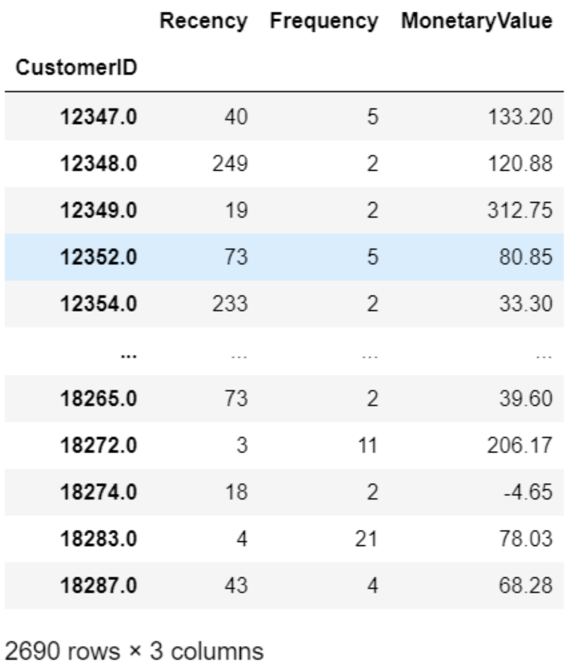
Lastly, to create the monetary value column, we can sum all transactions for each customer. The code looks like this,



```python
# Create TotalSum colummn
df_fix["TotalSum"] = df_fix["Quantity"] * df_fix["UnitPrice"]

# Create date variable that records recency
import datetime
snapshot_date = max(df_fix.InvoiceDate) + datetime.timedelta(days=1)

# Aggregate data by each customer
customers = df_fix.groupby(['CustomerID']).agg({
    'InvoiceDate': lambda x: (snapshot_date - x.max()).days,
    'InvoiceNo': 'count',
    'TotalSum': 'sum'})

# Rename columns
customers.rename(columns = {'InvoiceDate': 'Recency', 'InvoiceNo': 'Frequency', 'TotalSum': 'MonetaryValue'}, inplace=True)
```



```python
customers
```



	Recency	Frequency	MonetaryValue
CustomerID			
12347.0	40	5	133.20
12348.0	249	2	120.88
12349.0	19	2	312.75
12352.0	73	5	80.85
12354.0	233	2	33.30
...	...	...	...
18265.0	73	2	39.60
18272.0	3	11	206.17
18274.0	18	2	-4.65
18283.0	4	21	78.03
18287.0	43	4	68.28

2690 rows × 3 columns

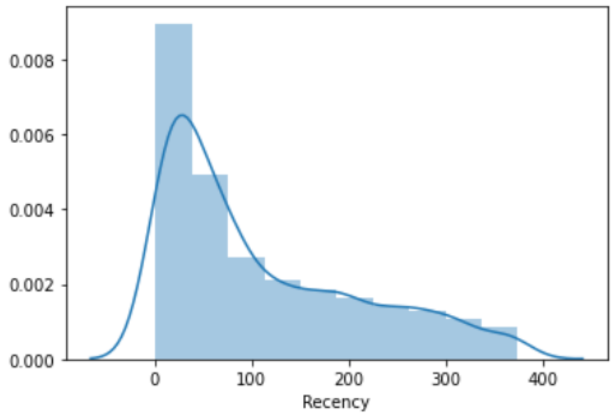
Right now, the dataset consists of recency, frequency, and monetary value column. But we cannot use the dataset yet because we have to preprocess the data more.

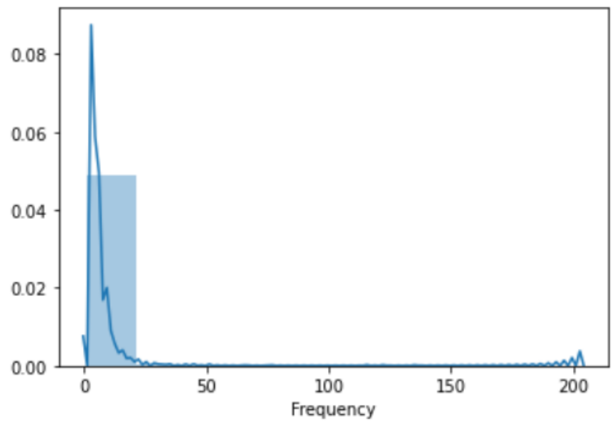
Step 3: Manage skewness and scale each variable

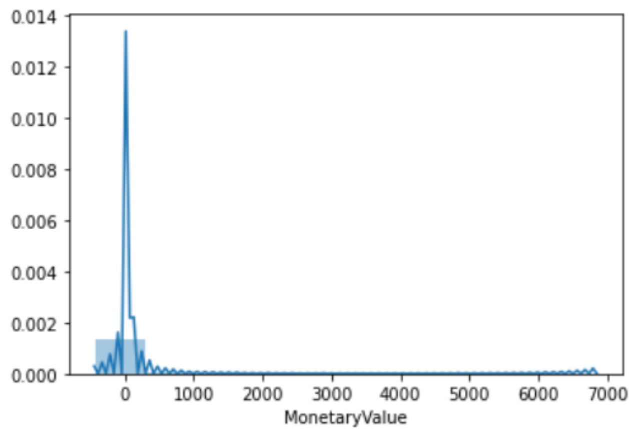
We have to make sure that the data meet these assumptions, they are,

- The data should meet assumptions where the variables are not skewed and
- have the same mean and variance.

Because of that, we have to manage the skewness of the variables. Here are the visualizations of each variable without skewness,







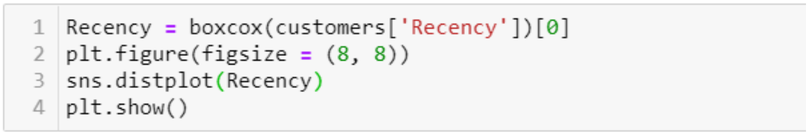
As we can see from above, we have to transform the data, so it has a more symmetrical form.

There are some methods that we can use to manage the skewness, they are,

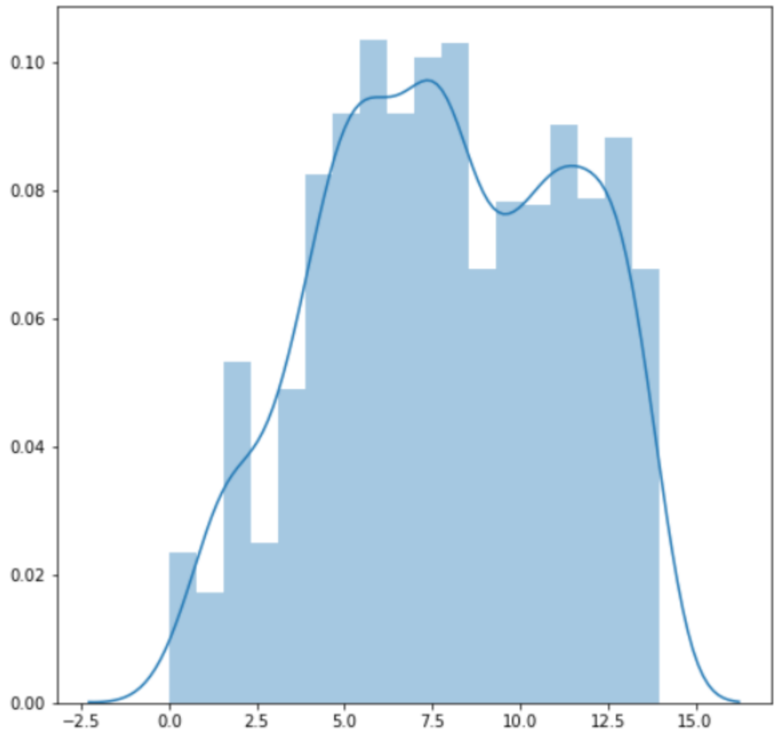
- log transformation
- square root transformation
- box-cox transformation

Note: We can use the transformation if and only if the variable only has positive values.

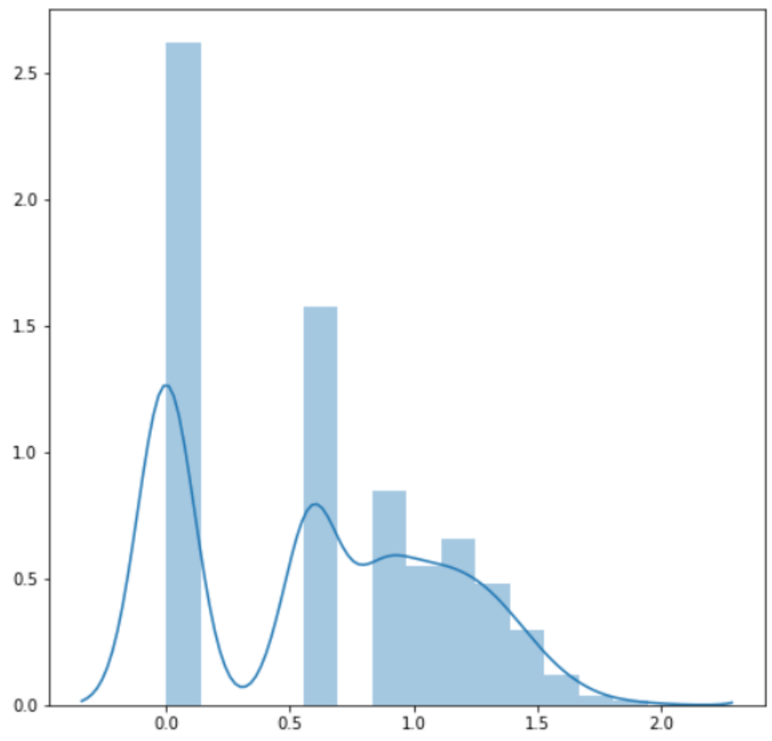
Below are the visualization each variable with box-cox transformation.



```python
Recency = boxcox(customers['Recency'])[0]
plt.figure(figsize = (8, 8))
sns.distplot(Recency)
plt.show()
```



```python
Frequency = boxcox(customers['Frequency'])[0]
plt.figure(figsize = (8, 8))
sns.distplot(Frequency)
plt.show()
```



Based on that visualization, it shows that the variables with box-cox transformation shows a more symmetrical form rather than the other transformations. To make sure, we calculate each variable using the skew function. The result looks like this,

```python
from scipy.stats import skew
print("Recency = ", skew(Recency))
print("Frequency = ", skew(Frequency))
```

```
Recency =  -0.10309005355465052
Frequency =  0.16176250323330582
```

<u>Interpreting the skewness value</u>

If the value is close to 0, the variable tend to have symmetrical form. However, if it's not, the variable has skew on it. Based on that calculation, we use variables that use box-cox transformations.

Based on that calculation, we will utilize variables that use box-cox transformations. Except for the MonetaryValue variable because the variable includes negative values. To handle this variable, we can use cubic root transformation to the data.

```python
from scipy import stats
customers_fix = pd.DataFrame()
customers_fix["Recency"] = stats.boxcox(customers['Recency'])[0]
customers_fix["Frequency"] = stats.boxcox(customers['Frequency'])[0]
customers_fix["MonetaryValue"] = pd.Series(np.cbrt(customers['MonetaryValue'])).values
customers_fix.tail()
```

	Recency	Frequency	MonetaryValue
2685	7.832068	0.591193	3.408514
2686	1.269495	1.435599	5.907565
2687	4.288385	0.591193	-1.669108
2688	1.665555	1.615329	4.273206
2689	6.340700	1.017445	4.087250

Step 4: Explore the data

We can't use the data yet. If we look at the plot once more, each variable don't have the same mean and variance. We have to normalize it. To normalize, we can use StandardScaler object from scikit-learn library to do it. The code will look like this,

```python
# Import library
from sklearn.preprocessing import StandardScaler

# Initialize the Object
scaler = StandardScaler()

# Fit and Transform The Data
scaler.fit(customers_fix)
customers_normalized = scaler.transform(customers_fix)

# Assert that it has mean 0 and variance 1
print(customers_normalized.mean(axis = 0).round(2)) # [0. -0. 0.]
print(customers_normalized.std(axis = 0).round(2)) # [1. 1. 1.]
```

```
[ 0. -0.  0.]
[1. 1. 1.]
```

The data now looks like below,

```python
customers_fix
```

	Recency	Frequency	MonetaryValue
0	6.152222	1.127547	5.107026
1	12.180045	0.591193	4.944452
2	4.402867	0.591193	6.787853
3	7.832068	1.127547	4.324076
4	11.907953	0.591193	3.217225
...	...	...	...
2685	7.832068	0.591193	3.408514
2686	1.269495	1.435599	5.907565
2687	4.288385	0.591193	-1.669108
2688	1.665555	1.615329	4.273206
2689	6.340700	1.017445	4.087250

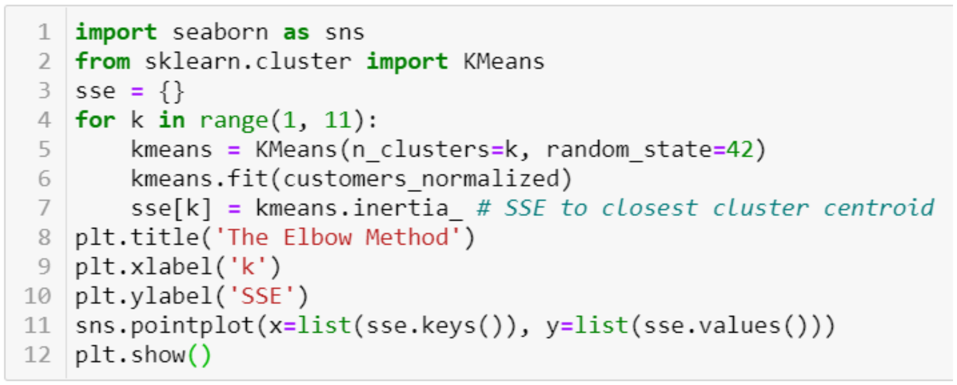
2690 rows × 3 columns

Step 5: Cluster the data

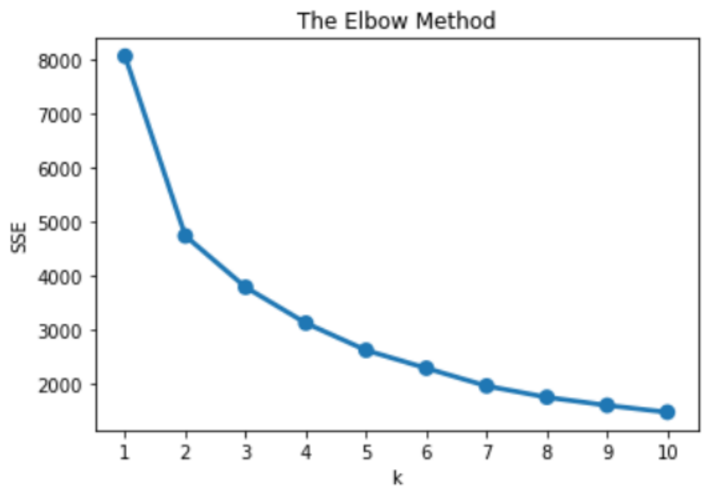
Finally, we can do clustering using that data.

Right after we preprocess the data, now we can focus on modelling. To make segmentation from the data, we can use the K-Means algorithm to do this.

K-Means algorithm is an unsupervised learning algorithm that uses the geometrical principle to determine which cluster belongs to the data. By determine each centroid, we calculate the distance to each centroid. Each data belongs to a centroid if it has the smallest distance from the other. It repeats until the next total of the distance doesn't have significant changes than before.
To implement K-Means in Python is easy. We can use the KMeans function from scikit-learn to do this. To make our clustering reach its maximum performance, we have to determine which hyperparameter fits to the data. To determine which hyperparameter is the best for our model and data, we can use the elbow method to decide. The code will look like this,



```python
import seaborn as sns
from sklearn.cluster import KMeans
sse = {}
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(customers_normalized)
    sse[k] = kmeans.inertia_ # SSE to closest cluster centroid
plt.title('The Elbow Method')
plt.xlabel('k')
plt.ylabel('SSE')
sns.pointplot(x=list(sse.keys()), y=list(sse.values()))
plt.show()
```



The x-axis is the value of the k, and the y-axis is the SSE value of the data. We will take the best parameter by looking at where the k-value will have a linear trend on the next consecutive k. Based on our observation, the k-value of 3 is the best hyperparameter for our model because the next k-value tend to have a linear trend. Therefore, our best model for the data is K-Means with the number of clusters is 3. Now, We can fit the model with this code,

```python
model = KMeans(n_clusters=3, random_state=42)
model.fit(customers_normalized)
model.labels_.shape
```

```
(2690,)
```

By fitting the model, we can have clusters where each data belongs. By that, we can analyze the data.

We can summarize the RFM table based on clusters and calculate the mean of each variable. The code will look like this,

```python
customers["Cluster"] = model.labels_
customers.groupby('Cluster').agg({
    'Recency':'mean',
    'Frequency':'mean',
    'MonetaryValue':['mean', 'count']}).round(2)
```

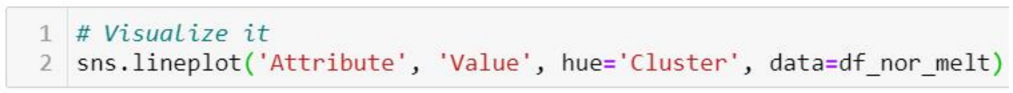
	Recency	Frequency	MonetaryValue	
	mean	mean	mean	count
Cluster				
0	47.21	1.62	21.58	808
1	226.64	1.64	28.66	960
2	46.31	7.72	166.06	922

Besides that, we can analyze the segments using snake plot. It requires the normalized dataset and also the cluster labels. By using this plot, we can have a good visualization from the data on how the cluster differs from each other. We can make the plot by using this code,

```python
# Create the dataframe
df_normalized = pd.DataFrame(customers_normalized, columns=['Recency', 'Frequency', 'MonetaryValue'])
df_normalized['ID'] = customers.index
df_normalized['Cluster'] = model.labels_

# Melt The Data
df_nor_melt = pd.melt(df_normalized.reset_index(),
                      id_vars=['ID', 'Cluster'],
                      value_vars=['Recency','Frequency','MonetaryValue'],
                      var_name='Attribute',
                      value_name='Value')
df_nor_melt.head()
```

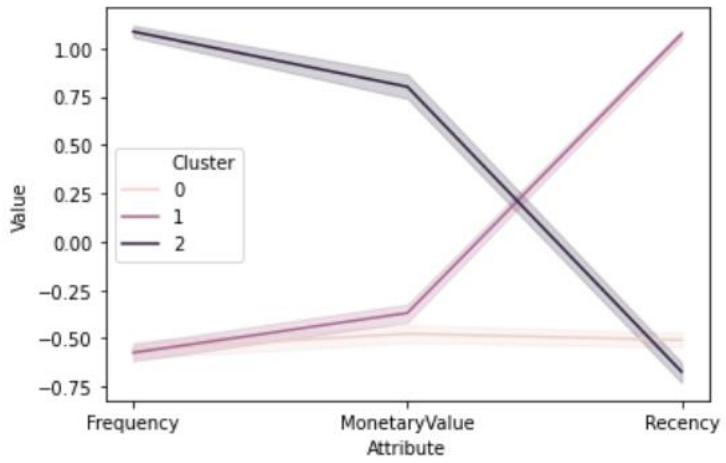
	ID	Cluster	Attribute	Value
0	12347.0	2	Recency	-0.493794
1	12348.0	1	Recency	1.232949
2	12349.0	2	Recency	-0.994917
3	12352.0	2	Recency	-0.012582
4	12354.0	1	Recency	1.155005

Visualizing the lineplot,



```
# Visualize it
sns.lineplot('Attribute', 'Value', hue='Cluster', data=df_nor_melt)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x1f496477250>
```



Step 6: Interpret the result

- It can be interpreted that, cluster 2 is frequent, spend more, and they buy the product recently. Therefore, it could be the cluster of a loyal customer.
- Then, the cluster 0 is less frequent, less to spend, but they buy the product recently. Therefore, it could be the cluster of new customer.
- Finally, the cluster 1 is less frequent, less to spend, and they buy the product at the old time. Therefore, it could be the cluster of churned customers.