Customer Life Time Value - Data Science Case for Marketing and Sales

Credits - Machine Learning Plus.

Motivation

Every business cares about their customers. They want to have more of them. They want customers to spend more of their time and money with them, to build good relationship with their customers.

However, the resources are limited.

It takes effort and money for businesses to run ad-campaigns, to sustain an effective marketing team so as to resolve problems when potential and existing customers reach out to them.

So, businesses want to know more who is a more valuable customer.

Who is more valuable?:

(i) a customer who made a `$1000` purchase 6 months back?
(ii) a customer who made 10 small purchases last month?

It turns out customers who make small yet repeated purchases can be more valuable in the long run. So, it's a good idea for your marketing teams to focus on high value customers.

But how to you quantify which customers are more valuable?

Using Customer Lifetime Value (CLTV)

The businesses who shine out of the crowd, are those who manage to retain their customers!

1. What is CLTV

CLV is an abbreviation for Customer Lifetime Value.Customer lifetime value (CLV, or CLTV) is the metric that indicates *the total revenue a business can reasonably expect from a single customer account throughout the business relationship*.

It is one of the most important metrics to measure at any growing company. Businesses use customer lifetime value to identify customer segments that are most valuable to the company. The longer a customer continues to purchase from a company, the greater their lifetime value becomes.

Customer lifetime value (CLV) can help you to answers the most important questions about sales to every company:

How can CLTV Modeling help?

It can help answer the following questions:

1. Which customers are still your customers?
2. Which customer(s) will buy from you in the next time duration?
3. How much each customer going to buy from you in the future?
4. Identify the most profitable customers - Who are the top N customers expected to order again in next period?
5. Determine how much to invest in acquiring customers?
6. How to segment the most profitable customers.
7. Identify the traits and features of high value customers
8. Determine how to allocate resources amongst customers
9. How can a company offer better, profitable product and make the most money?

2. What is Customer Acquistion Cost (CAC)?

The amount the business spends to acquire every customer is the Customer Acquisition Cost (CAC). It can be the expense incurred to both acquire them and to maintain them, calculated for predetermined durations (say a year).

Calculating CAC is calculated by *dividing your overall marketing expenses by the number of net new customers acquired in a given period of time*.

CAC refers to both the resources and costs incurred to acquire an additional customer. Customer acquisition cost is a key business metric that is commonly used alongside the customer lifetime value (LTV) metric to measure value generated by a new customer.

Typically, the CAC for a given period is the total of all promotional costs divided by the number of new customers acquired.

The actual acquisition cost can vary with the nature of business, some of the common components of acquisition cost are:

- Paid per click advertising cost
- TV Ads
- Posters, Banner Ads
- Paid sales and marketing staff
- Social media campaigns (collecting emails, phone numbers etc)
- Magazine, Journal and Newspaper ads
- Mailers

The key is to include everything needed, including digital and non-digital medium, to acquire a new customer.

It's quite easy to compute an aggregate value.

Let's say your company spends $100k on sales and $70k on marketing. Through this the company acquired 400 new customers for the financial year. Then, the CAC would be: (100k+70k)/400 = $425 per customer.

Why calculating the acquisition cost matters?

Because you don't want to spend more than what you can possibly hope to earn from your customers over lifetime.

Why should businesses focus on CLV:CAC relationship

The Customer Lifetime Value to Customer Acquisition Ratio (CLV:CAC) measures the relationship between the lifetime value of a customer and the cost of acquiring that customer. It tracks the relationship between what a company pays to get a first-time buyer and how much that customer is likely to spend over time.

How does it matter?

Typically, for a successful business, CLV:CAC ratio should be high.

A CLV:CAC of one to one (i.e., 1:1) tells you that your company is failing. With this ratio, ecommerce growth can only be achieved with significant investment, and you should not expect to earn much profit.

On the other hand, a CLV:CAC ratio of three-to-one or higher indicates that your company is building value. Investing in your business should lead to profitable growth.

There are limits, too. If the CLV:CAC ratio is too high, say 25:1, you are investing too little and may be vulnerable to competition.

3. How is CLTV Calculated?

First let us get started by understanding some commonly used terminologies.

1. **Average Order Value(AOV):** The Average Order value is the ratio of your total revenue and the total number of orders. AOV represents the mean amount of revenue that the customer spends on an order.

   ```
   Average Order Value = Total Revenue / Total Number of Orders
   ```

2. **Purchase Frequency(PF):** Purchase Frequency is the ratio of the total number of orders and the total number of customer. It represents the average number of orders placed by each customer.

   ```
   Purchase Frequency =  Total Number of Orders / Total Number of Customers
   ```

3. **Churn Rate:** Churn Rate is the percentage of customers who ceased to be a customer.

For example:

```
 Churn Rate = (Customers at the start of period - Customers at end of period) / Customers at start of period.
```

4. **Customer Lifespan:** Customer Lifetime is the period of time that the customer has been continuously ordering.

   Average Customer Lifespan = Sum of customer lifespans / Number of Customers

- If the business is relatively new and lacks a good sample size for customers, then lifespan can be approximated as follows:

```
Customer Lifespan = 1 / Churn Rate
```

5. **Repeat Rate:** Repeat rate can be defined as the ratio of the number of customers with more than one order to the number of unique customers. Example: If you have 10 customers in a month out of who 4 come back, your repeat rate is 40%.

```
Churn Rate= 1-Repeat Rate
```

Formula

```
CLTV = ((Average Order Value x Purchase Frequency * Average Life span)
```

For example:

For a subscription business, if avg. order value is $100 and a customer buy 3 times on an average (purchase freq) over a period of 5 years (avg life span).

Then,

```
CLTV = 100 * 3 * 5 = $1500
```

4. Alternate Formula for CLTV Calculation

1. When life span is not known for your business type, replace it with `1 / churnrate`.

   ```
   CLTV = ((Average Order Value x Purchase Frequency)/Churn Rate)
   ```

Example: For a subscription based business (say, book rentals) IF the avg order value is `$` 100 and a customer buys 3 times a year on an average. We dont know their lifespan but we know 20 percent of subscribers don't renew membership.

Then,

```
CLTV = 100 * 3 / .2 = 1500
```

If the business was able to successfully reduce the churn rate from 20% to 10% then, the CLTV would double to $3000 in revenue. Stunning!

Note: Dont include the new acquired customers in above formula.

2. When you care about the profit and not the revenue, multiply it with profit margin.

   ```
   CLTV = ((Average Order Value x Purchase Frequency)/Churn Rate) x Gross Profit margin
   ```

   where,

   ```
   Gross margin = [(Total revenue - COGS)/Total Revenue] * 100
   ```

Example: Suppose the gross margin in the previous example was say 30%, then the CLTV would be calculated as: 1500 * .3 = $450.

3. When your customers stay several years, then you need to account for the depreciation due to inflation (time value of money). Assuming margin and retention rates are constant:

   ```
   CLV Formula = Gross margin contribution per customer * [R / (1 + D - R)]
   ```

where, Gross Margin Contribution per Customer = Gross margin * Avg total revenue per customer.

Example: Say Gross margin is 30%, Avg revenue per customer per year = $100, Retention rate per year = 10% and Discount rate is 10% taken as default. Then,

CLTV = .3 *100 * (.9 / (1 + .1 - .9)) = $135

4. Factoring in the retention costs year on year

```
from IPython.display import Image
Image(filename='C:\\Users\\NDH00130\\Desktop\\CLTV.png', width=350)
```

$$\mathrm{CLV} = \mathrm{GC} \cdot \sum_{i=1}^{n} \frac{r^i}{(1+d)^i} - \mathrm{M} \cdot \sum_{i=1}^{n} \frac{r^{i-1}}{(1+d)^{i-0.5}},$$

where, r = retention rate, d = discount rate, M = retention cost per year per customer, GC = Yearly Gross contribution per year per customer.

Note: Above calculations are at aggregate level, but we want to compute for each customer. We will explore those.

5. Business Contexts where CLTV can be applied

The business contexts on which CLTV may be applied may be classified based on 2 factors:

1. Contractual vs Non-Contractual

--> Depends on whether there is a predetermined contract perdiod before you use the product / service.

Examples for Contractual models:

- Club memberships
- Magazine subscriptions
- Credit cards

Example for Non-Contractualmodels:

- Grocery stores
- e-Commerce shopping like Amazon
- Doctor visits

If purchase is within a contract, the customer churn is **explicitly known**.

2. Continuous vs Discrete Purchases

--> Depends on whether the puchase happens contunuously or at specific times / opportunity. Purchases can be with or without a contract

Examples of continuous purchases:

- Grocery
- Book rentals
- Hospital visits

Examples of discrete purchases:

- Public Events
- Club memberships
- Insurance policies
- Mobile phone plans

Q: What category the following products would fall in?

1. Netflix
2. Doordash / Swiggy / Zomato
3. Amazon.com
4. Mastercard Credit card
5. Hospital appointments

Solution

```
1. Netflix                    : Contractual | Discrete
2. Doordash / Swiggy / Zomato  : Non Contractual | Continuous
3. Amazon.com                 : Non Contractual | Continuous
4. Mastercard Credit card     : Contractual | Continuous
5. Hospital appointments      : Non contractual | Continuous
```

The CLTV methods we are about to discuss work well for Non-Contractual, Continuous sales type of businesses. It may work for other types as well.

6. Modeling Approaches for CLV

There are lots of approaches available for calculating CLTV.

1. The historical data of customers can be used to calculate CLV. This is the oldest method, where we calculate average revenue obtained from all customers. This provides a *single value to denote the lifetime value*. This can also be computed at individual *customer level*.

2. An improved solution to this is introduce *cohorts* and calculate the revenue for each cohort. But in these methods, there is a glaring disadvantage. We are unable to calculate the CLV for new customers.

3. Machine Learning based approach can solve this. Statistical models can be built with help of available data.

For example, Take recent six-month data as independent variables and total revenue over three years as a dependent variable and build a regression model on this data.

4. There is also a probabilistic approach for calculating CLV. We can calculate our RFM(Recency, Frequency, Monetary) from the data available. This table can be used to extract CLV. Probabilistic models like **BG/NBD** and **Gamma-Gamma** can be used here.

We can group customers into cohorts or segments based on the value to business using K means algorithm. After this, specific promotions , discounts, can be provided to focus on high value groups.

All of these methods are described below.

For calculating the value of customer, it is quite important to decide the timeframe, often decided based on the business context.

Generally 1 month, 2 months, 3 months, 6 months and 12 months or more are considered. Most commonly, 12 months is used.

7. Problem Statement

The aim here is to understand the different values of calculating customer value through practical examples.

We will be working with **Brazilian E-Commerce** Dataset by Olist.

This dataset was generously provided by Olist, the largest department store in Brazilian marketplaces. Olist connects small businesses from all over Brazil to channels without hassle and with a single contract. The dataset has information of 100k orders from 2016 to 2018 made at multiple marketplaces in Brazil. Its features allows viewing an order from multiple dimensions: from order status, price, payment and freight performance to customer location, product attributes.

This is real commercial data, anonymised.

We will start by understanding the data we have, about the different attributes in depth. You'll learn how to clean and pre-process the data. Next, we will understand the relationship between features through Explorartory data analysis. You'll learn how to generate the RFM matrix from the given data, how to apply probablistic models to generate CLV, how to use kmeans and cluster the customers into different segments on the basis of lifetime value

8. Setup Packages

The goal of this section is to:

- Import all the packages
- Set the options for data visualizations

```
# !pip install lifetimes
```

```
# Data Manipulation
import numpy as np
import pandas as pd

# Data Visualization
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.lines as mlines
import matplotlib.gridspec as gridspec
plt.grid(b=None)
import plotly.offline as py
```

```
import plotly.express as px

py.init_notebook_mode(connected=False) # To work offline

# Intsall and import lifetimes library
# !pip install lifetimes
import lifetimes

import os
import warnings
from datetime import datetime

# Set the random seed so as to reproduce results
import random
random.seed(42)
np.random.seed(42)

# Set Options
pd.set_option('display.max_rows', 800)
pd.set_option('display.max_columns', 500)
pd.set_option('expand_frame_repr', False)
%matplotlib inline
warnings.filterwarnings("ignore")
```

9. Load data

There are a total of 9 csv files. For customer lifetime value study, we will be using the 4 csv files : customers, orders, payments and order items.

The goal of this section is to:

- Load the datasets
- Get overview of the data

```
# Get all the files in Data directory
os.listdir("data")
```

```
['olist_customers_dataset.csv',
 'olist_geolocation_dataset.csv',
 'olist_orders_dataset.csv',
 'olist_order_items_dataset.csv',
 'olist_order_payments_dataset.csv',
 'olist_order_reviews_dataset.csv',
 'olist_products_dataset.csv',
 'olist_sellers_dataset.csv',
 'product_category_name_translation.csv']
```

```
# Load data
customers = pd.read_csv('data/olist_customers_dataset.csv')
orders = pd.read_csv('data/olist_orders_dataset.csv')
payments = pd.read_csv('data/olist_order_payments_dataset.csv')
order_items = pd.read_csv('data/olist_order_items_dataset.csv')
```

Link to source: Kaggle

The variable name in the middle of the above chart is the common column to join by.

Customers Dataset

The dataset contains the preliminary information of each customer who has visited.

Dataset Description :

- customer_id : It is key to the orders dataset. Each order has a unique customer_id
- customer_unique_id: unique identifier of a customer.
- customer_zip_code_prefix: first five digits of customer zip code
- Customer_city: customer adress
- Customer_state: customer address

```

```

```
customers.head()
```

	customer_id	customer_unique_id	customer_zip_code_prefix	customer_city	customer_state
0	06b8999e2fba1a1fbc88172c00ba8bc7	861eff4711a542e4b93843c6dd7febb0	14409	franca	SP
1	18955e83d337fd6b2def6b18a428ac77	290c77bc529b7ac935b93aa66c333dc3	9790	sao bernardo do campo	SP
2	4e7b3e00288586ebd08712fdd0374a03	060e732b5b29e8181a18229c7b0b2b5e	1151	sao paulo	SP
3	b2b6027bc5c5109e529d4dc6358b12c3	259dac757896d24d7702b9acbbff3f3c	8775	mogi das cruzes	SP
4	4f2d8ab171c80ec8364f7c12e35b23ad	345ecd01c38d18a9036ed96c73b8d066	13056	campinas	SP

Orders dataset

This dataset has details of each order along with the timestamps of when order was placed and the customer ID for it.

Data Description

- Order_id: Unique for each order
- customer_id: key to the customer dataset. Each order has a unique customer_id.
- order_status: Reference to the order status (delivered, shipped, etc).
- order_purchase_timestamp: Shows the purchase timestamp.
- order_approved_at: Shows the payment approval timestamp.
- order_delivered_carrier_date: Shows the order posting timestamp. When it was handled to the logistic partner.
- order_delivered_customer_date: Shows the actual order delivery date to the customer.
- order_estimated_delivery_date: Shows the estimated delivery date that was informed to customer at the purchase moment.

```
orders.head()
```

	order_id	customer_id	order_status	order_purchase_timestamp	order_approved_at	or
0	e481f51cbdc54678b7cc49136f2d6af7	9ef432eb6251297304e76186b10a928d	delivered	2017-10-02 10:56:33	2017-10-02 11:07:15	
1	53cdb2fc8bc7dce0b6741e2150273451	b0830fb4747a6c6d20dea0b8c802d7ef	delivered	2018-07-24 20:41:37	2018-07-26 03:24:27	
2	47770eb9100c2d0c44946d9cf07ec65d	41ce2a54c0b03bf3443c3d931a367089	delivered	2018-08-08 08:38:49	2018-08-08 08:55:23	
3	949d5b44dbf5de918fe9c16f97b45f8a	f88197465ea7920adcdbec7375364d82	delivered	2017-11-18 19:28:06	2017-11-18 19:45:59	
4	ad21c59c0840e6cb83a9ceb5573f8159	8ab97904e6daea8866dbdbc4fb7aad2c	delivered	2018-02-13 21:18:39	2018-02-13 22:20:29	

Order Items dataset

This dataset includes data about the items purchased within each order.

Data Description

- Order_id: Unique for each order
- order_item_id: Sequential number identifying number of items included in the same order.
- product_id: Product unique identifier
- seller_id: Seller unique identifier
- shipping_limit_date: Shows the seller shipping limit date for handling the order over to the logistic partner.
- price: Item price
- freight_value: Item freight value item (if an order has more than one item the freight value is splitted between items)

```
order_items.head()
```

	order_id	order_item_id	product_id	seller_id	shipping_li
0	00010242fe8c5a6d1ba2dd792cb16214	1	4244733e06e7ecb4970a6e2683c13e61	48436dade18ac8b2bce089ec2a041202	2017-09-19
1	00018f77f2f0320c557190d7a144bdd3	1	e5f2d52b802189ee658865ca93d83a8f	dd7ddc04e1b6c2c614352b383efe2d36	2017-05-03
2	000229ec398224ef6ca0657da4fc703e	1	c777355d18b72b67abbeef9df44fd0fd	5b51032eddd242adc84c38acab88f23d	2018-01-18
3	00024acbcdf0a6daa1e931b038114c75	1	7634da152a4610f1595efa32f14722fc	9d7a1d34a5052409006425275ba1c2b4	2018-08-15
4	00042b26cf59d7ce69dfabb4e55b4fd9	1	ac6c3623068f30de03045865e4e10089	df560393f3a51e74553ab94004ba5c87	2017-02-13

Payments Dataset

The dataset has details regarding payments for each order.

Data Description:

- order_id: Unique for each order
- payment_sequential: a customer may pay an order with more than one payment method. If he does so, a sequence will be created to accommodate all payments.
- payment_type: method of payment chosen by the customer.
- payment_installments : number of installments chosen by the customer
- Payment_value: Transaction value

```
payments.head()
```

	order_id	payment_sequential	payment_type	payment_installments	payment_value
0	b81ef226f3fe1789b1e8b2acac839d17	1	credit_card	8	99.33
1	a9810da82917af2d9aefd1278f1dcfa0	1	credit_card	1	24.39
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	credit_card	1	65.71
3	ba78997921bbcdc1373bb41e913ab953	1	credit_card	8	107.78
4	42fdf880ba16b47b59251dd489d4441a	1	credit_card	2	128.45

10. Understand the Data

Before attempting to solve the problem, it's very important to have a good understanding of data.

The goal of this section is to:

- Get the dimensions of data
- Get the summary of data
- Get various statistics of data

Customers Dataframe

```
# Dimensions
print('Dimensions of customer data', customers.shape)
customers.head()
```

```
Dimensions of customer data (99441, 5)
```

	customer_id	customer_unique_id	customer_zip_code_prefix	customer_city	customer_state
0	06b8999e2fba1a1fbc88172c00ba8bc7	861eff4711a542e4b93843c6dd7febb0	14409	franca	SP
1	18955e83d337fd6b2def6b18a428ac77	290c77bc529b7ac935b93aa66c333dc3	9790	sao bernardo do campo	SP
2	4e7b3e00288586ebd08712fdd0374a03	060e732b5b29e8181a18229c7b0b2b5e	1151	sao paulo	SP
3	b2b6027bc5c5109e529d4dc6358b12c3	259dac757896d24d7702b9acbbff3f3c	8775	mogi das cruzes	SP
4	4f2d8ab171c80ec8364f7c12e35b23ad	345ecd01c38d18a9036ed96c73b8d066	13056	campinas	SP

```
# Summary
customers.describe()
```

	customer_zip_code_prefix
count	99441.000000
mean	35137.474583
std	29797.938996
min	1003.000000
25%	11347.000000
50%	24416.000000
75%	58900.000000
max	99990.000000

We want to get a sense of:

- number of unique values in every field (to see whether it is categorical or continuous and also map with the respective orders)
- perc of missing data,
- biggest category,
- datatype.

```
def get_stats(df):
    """Stats of dataset"""
    stats = []
    for col in df.columns:
        stats.append((col,
                      df[col].nunique(),
                      df[col].isnull().sum() * 100 / df.shape[0],
                      df[col].value_counts(normalize=True, dropna=False).values[0] * 100,
                      df[col].dtype))
    stats_df = pd.DataFrame(stats, columns=['Feature', 'Unique_values', 'Percentage of missing values', 'Percentage of v
    return stats_df
```

```
stats_df = get_stats(customers)
stats_df.sort_values('Percentage of missing values', ascending=False)
```

	Feature	Unique_values	Percentage of missing values	Percentage of values in the biggest category	type
0	customer_id	99441	0.0	0.001006	object
1	customer_unique_id	96096	0.0	0.017096	object
2	customer_zip_code_prefix	14994	0.0	0.142798	int64
3	customer_city	4119	0.0	15.627357	object
4	customer_state	27	0.0	41.980672	object

- Looks like there are some repeat customer ids.
- Out of 27 unique states, one particular state as 41% of data points.

```
# View the top 5 states
customers["customer_state"].value_counts(normalize=True, dropna=False).head()
```

```
SP    0.419807
RJ    0.129242
MG    0.117004
RS    0.054967
PR    0.050734
Name: customer_state, dtype: float64
```

Orders Dataframe

```
# Dimensions
print('Dimensions of orders data', orders.shape)
orders.head()
```

```
Dimensions of orders data (99441, 8)
```

	order_id	customer_id	order_status	order_purchase_timestamp	order_approved_at	or
0	e481f51cbdc54678b7cc49136f2d6af7	9ef432eb6251297304e76186b10a928d	delivered	2017-10-02 10:56:33	2017-10-02 11:07:15	
1	53cdb2fc8bc7dce0b6741e2150273451	b0830fb4747a6c6d20dea0b8c802d7ef	delivered	2018-07-24 20:41:37	2018-07-26 03:24:27	
2	47770eb9100c2d0c44946d9cf07ec65d	41ce2a54c0b03bf3443c3d931a367089	delivered	2018-08-08 08:38:49	2018-08-08 08:55:23	
3	949d5b44dbf5de918fe9c16f97b45f8a	f88197465ea7920adcdbec7375364d82	delivered	2017-11-18 19:28:06	2017-11-18 19:45:59	
4	ad21c59c0840e6cb83a9ceb5573f8159	8ab97904e6daea8866dbdbc4fb7aad2c	delivered	2018-02-13 21:18:39	2018-02-13 22:20:29	

```
# Summary
orders.describe()
```

	order_id	customer_id	order_status	order_purchase_timestamp	order_approved_a
count	99441	99441	99441	99441	9928

Seems to have as many orders as there are unique customer_ids. So we will need to map the orders with 'customer_unique_id'.

```
stats_df = get_stats(orders)
stats_df.sort_values('Percentage of missing values', ascending=False)
```

	Feature	Unique_values	Percentage of missing values	Percentage of values in the biggest category	type
6	order_delivered_customer_date	95664	2.981668	2.981668	object
5	order_delivered_carrier_date	81018	1.793023	1.793023	object
4	order_approved_at	90733	0.160899	0.160899	object
0	order_id	99441	0.000000	0.001006	object
1	customer_id	99441	0.000000	0.001006	object
2	order_status	8	0.000000	97.020344	object
3	order_purchase_timestamp	98875	0.000000	0.003017	object
7	order_estimated_delivery_date	459	0.000000	0.524934	object

Order Items Dataframe

```
# Dimensions
print('Dimensions of orders data', order_items.shape)
order_items.head()
```

```
Dimensions of orders data (112650, 7)
```

	order_id	order_item_id	product_id	seller_id	shipping_li
0	00010242fe8c5a6d1ba2dd792cb16214	1	4244733e06e7ecb4970a6e2683c13e61	48436dade18ac8b2bce089ec2a041202	2017-09-19
1	00018f77f2f0320c557190d7a144bdd3	1	e5f2d52b802189ee658865ca93d83a8f	dd7ddc04e1b6c2c614352b383efe2d36	2017-05-03
2	000229ec398224ef6ca0657da4fc703e	1	c777355d18b72b67abbeef9df44fd0fd	5b51032eddd242adc84c38acab88f23d	2018-01-18
3	00024acbcdf0a6daa1e931b038114c75	1	7634da152a4610f1595efa32f14722fc	9d7a1d34a5052409006425275ba1c2b4	2018-08-15
4	00042b26cf59d7ce69dfabb4e55b4fd9	1	ac6c3623068f30de03045865e4e10089	df560393f3a51e74553ab94004ba5c87	2017-02-13

```
# Summary
order_items.describe()
```

	order_item_id	price	freight_value
count	112650.000000	112650.000000	112650.000000
mean	1.197834	120.653739	19.990320
std	0.705124	183.633928	15.806405
min	1.000000	0.850000	0.000000
25%	1.000000	39.900000	13.080000
50%	1.000000	74.990000	16.260000
75%	1.000000	134.900000	21.150000
max	21.000000	6735.000000	409.680000

```
stats_df = get_stats(order_items)
stats_df.sort_values('Percentage of missing values', ascending=False)
```

	Feature	Unique_values	Percentage of missing values	Percentage of values in the biggest category	type

Observe

- Lots of orders
- Fewer item_ids but large number of product ids.
- 87% of one of the item ids dominate.

4	shipping_limit_date	93318	0.0	0.018642	object

Payments Dataframe

6	freight_value	6999	0.0	3.290723	float64

```
# Dimensions
print('Dimensions of orders data', payments.shape)
payments.head()
```

```
Dimensions of orders data (103886, 5)
```

	order_id	payment_sequential	payment_type	payment_installments	payment_value
0	b81ef226f3fe1789b1e8b2acac839d17	1	credit_card	8	99.33
1	a9810da82917af2d9aefd1278f1dcfa0	1	credit_card	1	24.39
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	credit_card	1	65.71
3	ba78997921bbcdc1373bb41e913ab953	1	credit_card	8	107.78
4	42fdf880ba16b47b59251dd489d4441a	1	credit_card	2	128.45

```
# Summary
payments.describe()
```

	payment_sequential	payment_installments	payment_value
count	103886.000000	103886.000000	103886.000000
mean	1.092679	2.853349	154.100380
std	0.706584	2.687051	217.494064
min	1.000000	0.000000	0.000000
25%	1.000000	1.000000	56.790000
50%	1.000000	1.000000	100.000000
75%	1.000000	4.000000	171.837500
max	29.000000	24.000000	13664.080000

```
stats_df = get_stats(payments)
stats_df.sort_values('Percentage of missing values', ascending=False)
```

	Feature	Unique_values	Percentage of missing values	Percentage of values in the biggest category	type
0	order_id	99440	0.0	0.027915	object
1	payment_sequential	29	0.0	95.643301	int64
2	payment_type	5	0.0	73.922376	object
3	payment_installments	24	0.0	50.580444	int64
4	payment_value	29077	0.0	0.311880	float64

Inferences:

- 5 payment types.
- Upto 20 payment installments (2 years)
- One of the paymnt type dominates 73%.
- Only the orders data contains missing vales
- Primary key of the datasets is order_id. It can be used to merge all the datasets

11. Data Preprocessing for EDA

The goal of this section is to:

- Merge datasets to create a master dataframe

- Treat missing values
- Engineer new features

```
# creating master dataframe
df1 = payments.merge(order_items, on='order_id')
df2 = df1.merge(orders, on='order_id')
df = df2.merge(customers, on='customer_id')
print(df.shape)
```

```
(117601, 22)
```

```
df.head()
```

	order_id	payment_sequential	payment_type	payment_installments	payment_value	order_item_id	
0	b81ef226f3fe1789b1e8b2acac839d17	1	credit_card	8	99.33	1	af74cc53
1	a9810da82917af2d9aefd1278f1dcfa0	1	credit_card	1	24.39	1	a630cc32
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	credit_card	1	65.71	1	2028bf1b
3	ba78997921bbcdc1373bb41e913ab953	1	credit_card	8	107.78	1	548e5bfe
4	42fdf880ba16b47b59251dd489d4441a	1	credit_card	2	128.45	1	38648636

```
df.dtypes
```

```
order_id                        object
payment_sequential               int64
payment_type                    object
payment_installments             int64
payment_value                  float64
order_item_id                    int64
product_id                      object
seller_id                       object
shipping_limit_date             object
price                          float64
freight_value                  float64
customer_id                     object
order_status                    object
order_purchase_timestamp        object
order_approved_at               object
order_delivered_carrier_date    object
order_delivered_customer_date   object
order_estimated_delivery_date   object
customer_unique_id              object
customer_zip_code_prefix         int64
customer_city                   object
customer_state                  object
dtype: object
```

Parse dates

```
# converting date columns to datetime
date_columns = ['shipping_limit_date', 'order_purchase_timestamp', 'order_approved_at', 'order_delivered_carrier_date',
for col in date_columns:
    df[col] = pd.to_datetime(df[col], format='%Y-%m-%d %H:%M:%S')
```

```
df[['shipping_limit_date',
   'order_purchase_timestamp',
   'order_approved_at',
   'order_delivered_carrier_date',
   'order_delivered_customer_date',
   'order_estimated_delivery_date']].dtypes
```

```
shipping_limit_date             datetime64[ns]
order_purchase_timestamp        datetime64[ns]
order_approved_at               datetime64[ns]
order_delivered_carrier_date    datetime64[ns]
order_delivered_customer_date   datetime64[ns]
order_estimated_delivery_date   datetime64[ns]
dtype: object
```

Clean up column names

```
# cleaning up name columns
df['customer_city'] = df['customer_city'].str.title()
```

```
df['payment_type'] = df['payment_type'].str.replace('_', ' ').str.title()  # Optional
# engineering new/essential columns
df['order_date'] = pd.to_datetime(df['order_purchase_timestamp'], format = '%Y-%m-%d')
```

Create new columns / Features

```
df['delivery_against_estimated'] = (df['order_estimated_delivery_date'] - df['order_delivered_customer_date']).dt.days
df['order_purchase_year']        = df.order_purchase_timestamp.apply(lambda x: x.year)
df['order_purchase_month']       = df.order_purchase_timestamp.apply(lambda x: x.month)
df['order_purchase_dayofweek']   = df.order_purchase_timestamp.apply(lambda x: x.dayofweek)
df['order_purchase_hour']        = df.order_purchase_timestamp.apply(lambda x: x.hour)
df['order_purchase_day']         = df['order_purchase_dayofweek'].map({0:'Mon',1:'Tue',2:'Wed',3:'Thu',4:'Fri',5:'Sat',6
df['order_purchase_mon']         = df.order_purchase_timestamp.apply(lambda x: x.month).map({1:'Jan',2:'Feb',3:'Mar',4:'

# Changing the month attribute for correct ordenation
df['month_year']                 = df['order_purchase_month'].astype(str).apply(lambda x: '0' + x if len(x) == 1 else x)
df['month_year']                 = df['order_purchase_year'].astype(str) + '-' + df['month_year'].astype(str)

# creating year month column
df['month_y']                    = df['order_purchase_timestamp'].map(lambda date: 100*date.year + date.month)
```

```
df.head()
```

	order_id	payment_sequential	payment_type	payment_installments	payment_value	order_item_id	
0	b81ef226f3fe1789b1e8b2acac839d17	1	Credit Card	8	99.33	1	af74cc53
1	a9810da82917af2d9aefd1278f1dcfa0	1	Credit Card	1	24.39	1	a630cc32
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	Credit Card	1	65.71	1	2028bf1b
3	ba78997921bbcdc1373bb41e913ab953	1	Credit Card	8	107.78	1	548e5bfe
4	42fdf880ba16b47b59251dd489d4441a	1	Credit Card	2	128.45	1	38648636

Describe

```
# displaying summary staticstics of columns
df.describe(include='all')
```

	order_id	payment_sequential	payment_type	payment_installments	payment_value	order_item_id	
count	117601	117601.000000	117601	117601.000000	117601.000000	117601.000000	
unique	98665	NaN	4	NaN	NaN	NaN	
top	895ab968e7bb0d5659d16cd74cd1650c	NaN	Credit Card	NaN	NaN	NaN	aca2
freq	63	NaN	86769	NaN	NaN	NaN	
first	NaN	NaN	NaN	NaN	NaN	NaN	
last	NaN	NaN	NaN	NaN	NaN	NaN	
mean	NaN	1.093528	NaN	2.939482	172.686752	1.195900	
std	NaN	0.726692	NaN	2.774223	267.592290	0.697706	
min	NaN	1.000000	NaN	0.000000	0.000000	1.000000	
25%	NaN	1.000000	NaN	1.000000	60.870000	1.000000	
50%	NaN	1.000000	NaN	2.000000	108.210000	1.000000	
75%	NaN	1.000000	NaN	4.000000	189.260000	1.000000	
max	NaN	29.000000	NaN	24.000000	13664.080000	21.000000	

Some Inferences

1. Some order ids are repeating

2. 4 unique payment types exist now
3. There are 0 value payments
4. There seem to be orders where freight value is so high and equal to order value.
5. Negative value of delivery expected is observed.

```
stats_df = get_stats(df)
stats_df.sort_values('Percentage of missing values', ascending=False)
```

	Feature	Unique_values	Percentage of missing values	Percentage of values in the biggest category	type
16	order_delivered_customer_date	95663	2.182805	2.182805	datetime64[ns]
23	delivery_against_estimated	198	2.182805	7.279700	float64
15	order_delivered_carrier_date	81016	1.058664	1.058664	datetime64[ns]
14	order_approved_at	90173	0.012755	0.053571	datetime64[ns]
19	customer_zip_code_prefix	14976	0.000000	0.134353	int64
20	customer_city	4110	0.000000	15.807689	object
21	customer_state	27	0.000000	42.147601	object
22	order_date	98111	0.000000	0.053571	datetime64[ns]
24	order_purchase_year	3	0.000000	54.146648	int64
17	order_estimated_delivery_date	449	0.000000	0.551866	datetime64[ns]
25	order_purchase_month	12	0.000000	10.741405	int64
26	order_purchase_dayofweek	7	0.000000	16.266868	int64
27	order_purchase_hour	24	0.000000	6.794160	int64
28	order_purchase_day	7	0.000000	16.266868	object
29	order_purchase_mon	12	0.000000	10.741405	object
30	month_year	24	0.000000	7.666601	object
18	customer_unique_id	95419	0.000000	0.063775	object
0	order_id	98665	0.000000	0.053571	object
1	payment_sequential	29	0.000000	95.711771	int64
13	order_purchase_timestamp	98111	0.000000	0.053571	datetime64[ns]
12	order_status	7	0.000000	97.818046	object
11	customer_id	98665	0.000000	0.053571	object
10	freight_value	6999	0.000000	3.257625	float64
9	price	5968	0.000000	2.215968	float64
8	shipping_limit_date	93317	0.000000	0.053571	datetime64[ns]
7	seller_id	3095	0.000000	1.813760	object
6	product_id	32951	0.000000	0.455778	object
5	order_item_id	21	0.000000	87.631908	int64
4	payment_value	28938	0.000000	0.297617	float64
3	payment_installments	24	0.000000	49.843964	int64
2	payment_type	4	0.000000	73.782536	object
31	month_y	24	0.000000	7.666601	int64

Note

1. The number of unique values give an idea of which variables could be treated as categorical.
2. The interesting points are: Percentage of missing values and columns where the biggest category dominate (eg: PurchaseState, city, day of week, payment type)

```
# dropping missing values
df.dropna(inplace=True)
df.isnull().values.any()
```

```
False
```

```
# displaying dataframe info
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 115018 entries, 0 to 117600
Data columns (total 32 columns):
 #   Column                         Non-Null Count   Dtype
---  ------                         --------------   -----
 0   order_id                       115018 non-null  object
 1   payment_sequential             115018 non-null  int64
 2   payment_type                   115018 non-null  object
 3   payment_installments           115018 non-null  int64
 4   payment_value                  115018 non-null  float64
 5   order_item_id                  115018 non-null  int64
 6   product_id                     115018 non-null  object
 7   seller_id                      115018 non-null  object
 8   shipping_limit_date            115018 non-null  datetime64[ns]
 9   price                          115018 non-null  float64
 10  freight_value                  115018 non-null  float64
 11  customer_id                    115018 non-null  object
 12  order_status                   115018 non-null  object
 13  order_purchase_timestamp       115018 non-null  datetime64[ns]
 14  order_approved_at              115018 non-null  datetime64[ns]
 15  order_delivered_carrier_date   115018 non-null  datetime64[ns]
 16  order_delivered_customer_date  115018 non-null  datetime64[ns]
 17  order_estimated_delivery_date  115018 non-null  datetime64[ns]
 18  customer_unique_id             115018 non-null  object
 19  customer_zip_code_prefix       115018 non-null  int64
 20  customer_city                  115018 non-null  object
 21  customer_state                 115018 non-null  object
 22  order_date                     115018 non-null  datetime64[ns]
 23  delivery_against_estimated     115018 non-null  float64
 24  order_purchase_year            115018 non-null  int64
 25  order_purchase_month           115018 non-null  int64
 26  order_purchase_dayofweek       115018 non-null  int64
 27  order_purchase_hour            115018 non-null  int64
 28  order_purchase_day             115018 non-null  object
 29  order_purchase_mon             115018 non-null  object
 30  month_year                     115018 non-null  object
 31  month_y                        115018 non-null  int64
dtypes: datetime64[ns](7), float64(4), int64(9), object(12)
memory usage: 29.0+ MB
```

The above master dataframe constitutes of the various independent dataset provided joined together via unique keys. Date columns have also been converted to datetime and new essential columns engineered for analysis purpose.

Save in Disk

```
if not os.path.exists('Intermediate'):
    os.makedir("Intermediate")
```

```
df.to_csv("Intermediate/df_full.csv", index=False)
```

12. Exploratory Data Analysis

```
# defining visualizaition functions
def format_spines(ax, right_border=True):
    """Format the graphs"""
    ax.spines['bottom'].set_color('#666666')
    ax.spines['left'].set_color('#666666')
    ax.spines['top'].set_visible(False)
    if right_border:
        ax.spines['right'].set_color('#FFFFFF')
    else:
        ax.spines['right'].set_color('#FFFFFF')
    ax.patch.set_facecolor('#FFFFFF')


def count_plot(feature, df, colors='Blues_d', hue=False, ax=None, title=''):
    """Count Plot"""
    # Preparing variables
    ncount = len(df)
    if hue != False:
        ax = sns.countplot(x=feature, data=df, palette=colors, hue=hue, ax=ax)
    else:
        ax = sns.countplot(x=feature, data=df, palette=colors, ax=ax)
```

```
    format_spines(ax)

    # Setting percentage
    for p in ax.patches:
        x=p.get_bbox().get_points()[:,0]
        y=p.get_bbox().get_points()[1,1]
        ax.annotate('{:.1f}%'.format(100.*y/ncount), (x.mean(), y),
                ha='center', va='bottom') # set the alignment of the text

    # Final configuration
    if not hue:
        ax.set_title(df[feature].describe().name + ' Analysis', size=13, pad=15)
    else:
        ax.set_title(df[feature].describe().name + ' Analysis by ' + hue, size=13, pad=15)
    if title != '':
        ax.set_title(title)
    plt.tight_layout()


def bar_plot(x, y, df, colors='Blues_d', hue=False, ax=None, value=False, title=''):

    # Preparing variables
    try:
        ncount = sum(df[y])
    except:
        ncount = sum(df[x])
    #fig, ax = plt.subplots()
    if hue != False:
        ax = sns.barplot(x=x, y=y, data=df, palette=colors, hue=hue, ax=ax, ci=None)
    else:
        ax = sns.barplot(x=x, y=y, data=df, palette=colors, ax=ax, ci=None)

    # Setting borders
    format_spines(ax)

    # Setting percentage
    for p in ax.patches:
        xp=p.get_bbox().get_points()[:,0]
        yp=p.get_bbox().get_points()[1,1]
        if value:
            ax.annotate('{:.2f}k'.format(yp/1000), (xp.mean(), yp),
                    ha='center', va='bottom') # set the alignment of the text
        else:
            ax.annotate('{:.1f}%'.format(100.*yp/ncount), (xp.mean(), yp),
                    ha='center', va='bottom') # set the alignment of the text
    if not hue:
        ax.set_title(df[x].describe().name + ' Analysis', size=12, pad=15)
    else:
        ax.set_title(df[x].describe().name + ' Analysis by ' + hue, size=12, pad=15)
    if title != '':
        ax.set_title(title)
    plt.tight_layout()
```

Let's check the **number of orders everyday**

```
# order count by date
plt.figure(figsize = (14,7))
df.order_date.value_counts().plot();
```

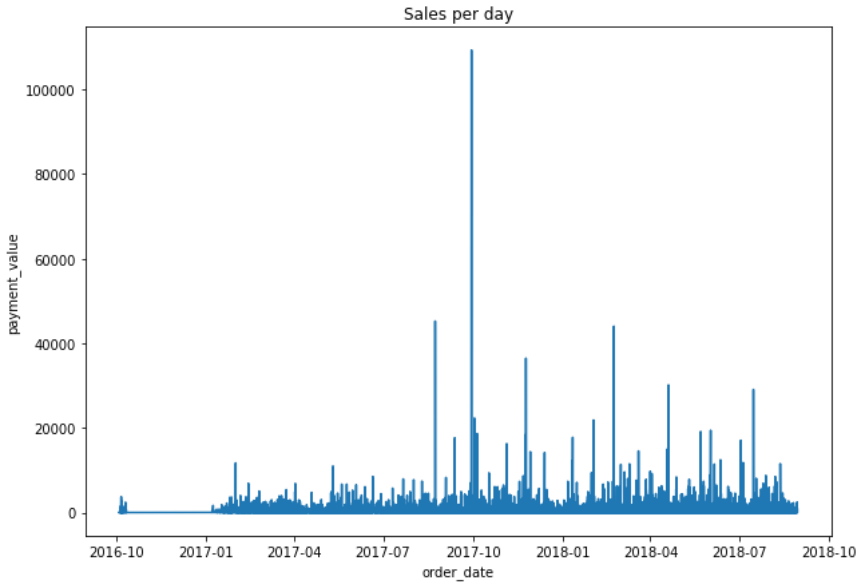
Inference

- Orders per day see to have peaks and flats regions.
- The highest peak is observed somewhere near 2017-Aug.

Let's check the **revenue per day**

```
# creating an aggregation
plt.figure(figsize = (10,7))
sales_per_purchase_date = df.groupby('order_date', as_index=False).payment_value.sum()
ax = sns.lineplot(x="order_date", y="payment_value", data=sales_per_purchase_date)
ax.set_title('Sales per day')
```

```
Text(0.5, 1.0, 'Sales per day')
```



Observation

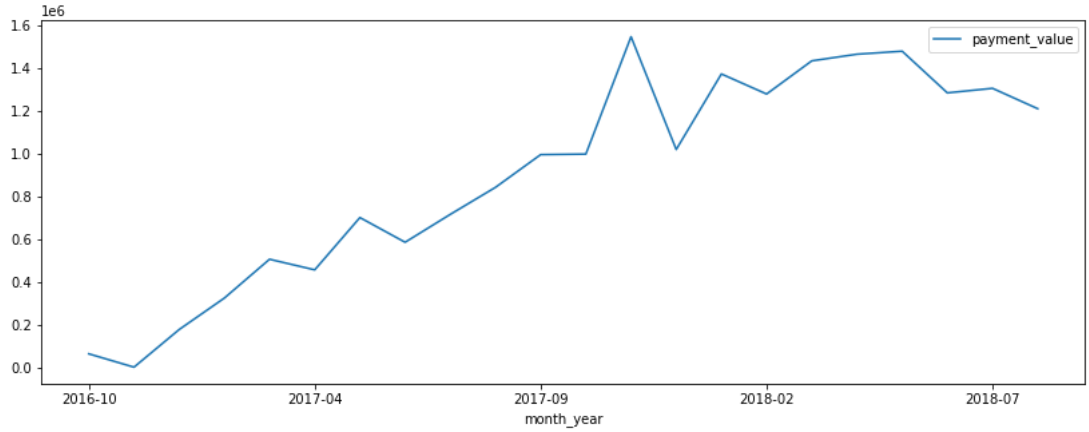
- Similar observation as orders per day. But the highest peak happens at different time.

Monthly Revenue

```
# calculate Revenue for each row and create a new dataframe with YearMonth - Revenue columns
df_revenue = df.groupby(['month_year'])['payment_value'].sum().reset_index()
df_revenue
```

	month_year	payment_value
0	2016-10	62591.65
1	2016-12	19.62
2	2017-01	176376.56
3	2017-02	323815.95
4	2017-03	505735.83
5	2017-04	456108.32
6	2017-05	701119.60
7	2017-06	585400.98

```
df_revenue.plot(x="month_year", y="payment_value", figsize=(14,5));
```



20 2018-07 1306707.42

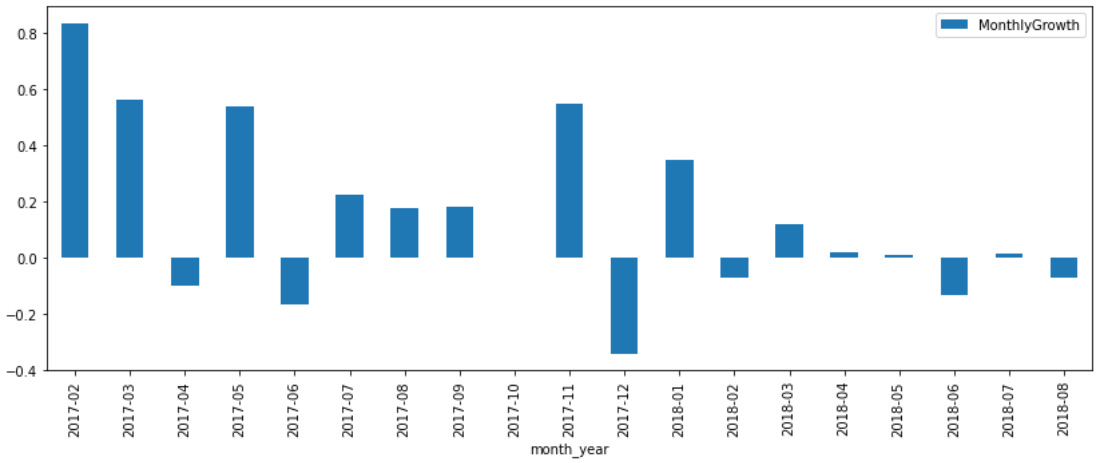
Inference: Revenue seem to have grown consistently over the years.

Monthly Revenue Growth Rate

```
#calculating for monthly revenie growth rate
# using pct_change() function to see monthly percentage change
df_revenue['MonthlyGrowth'] = df_revenue['payment_value'].pct_change()

df_revenue
```

	month_year	payment_value	MonthlyGrowth
0	2016-10	62591.65	NaN
1	2016-12	19.62	-0.999687
2	2017-01	176376.56	8988.630989

```
df_revenue.iloc[3:, :].plot.bar(x="month_year", y="MonthlyGrowth", figsize=(14,5));
```

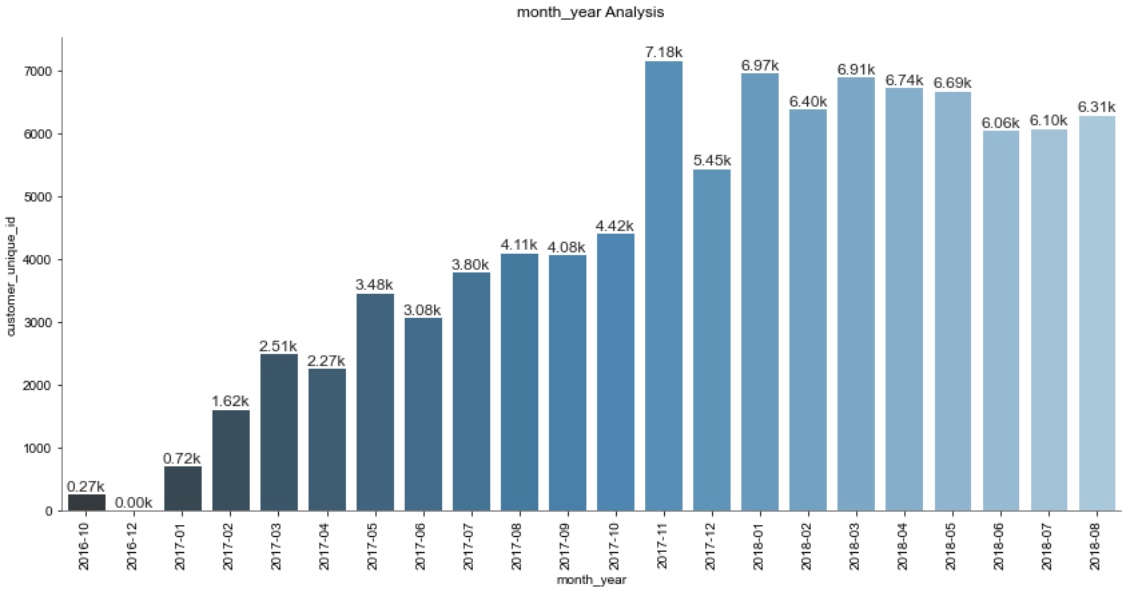


Inference: Positive growth rate overall, certain years the growth declined, before picking back up.

17	2018-04	1466607.15	0.021700
19	2018-06	1285396.78	-0.131880

Monthly Active Customers

```
# creating monthly active customers dataframe by counting unique Customer IDs
df_monthly_active = df.groupby('month_year')['customer_unique_id'].nunique().reset_index()

fig, ax = plt.subplots(figsize=(12, 6))
sns.set(palette='muted', color_codes=True, style='whitegrid')
bar_plot(x='month_year', y='customer_unique_id', df=df_monthly_active, value=True)
ax.tick_params(axis='x', labelrotation=90)
plt.show()
```



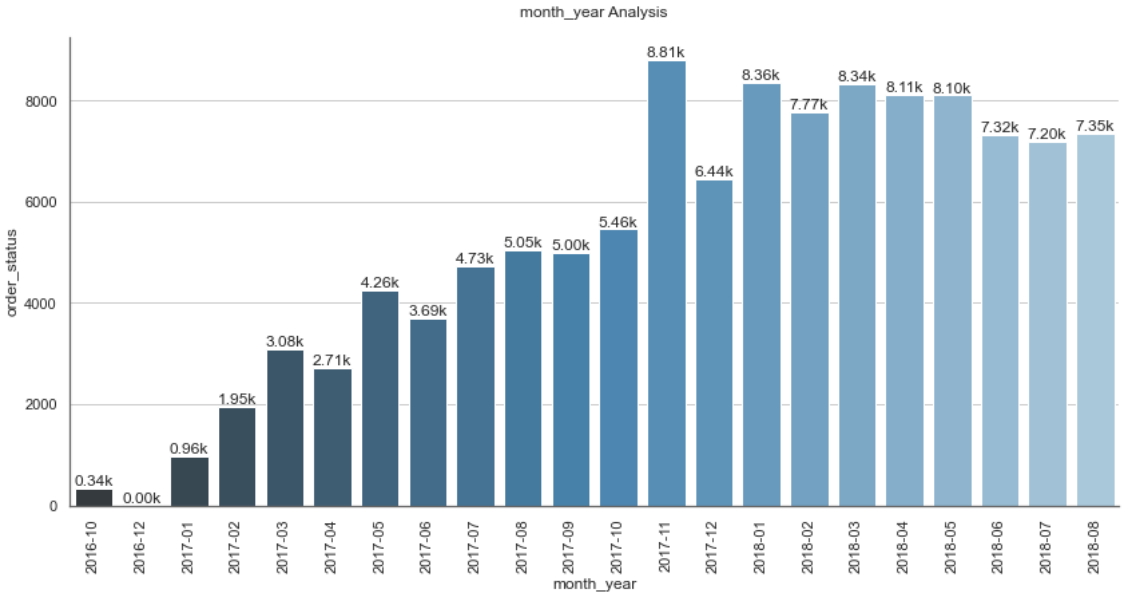
Observation:

Monthly Active users have grown consistently in the past. Seems to flatten more recently.

Monthly Order Count

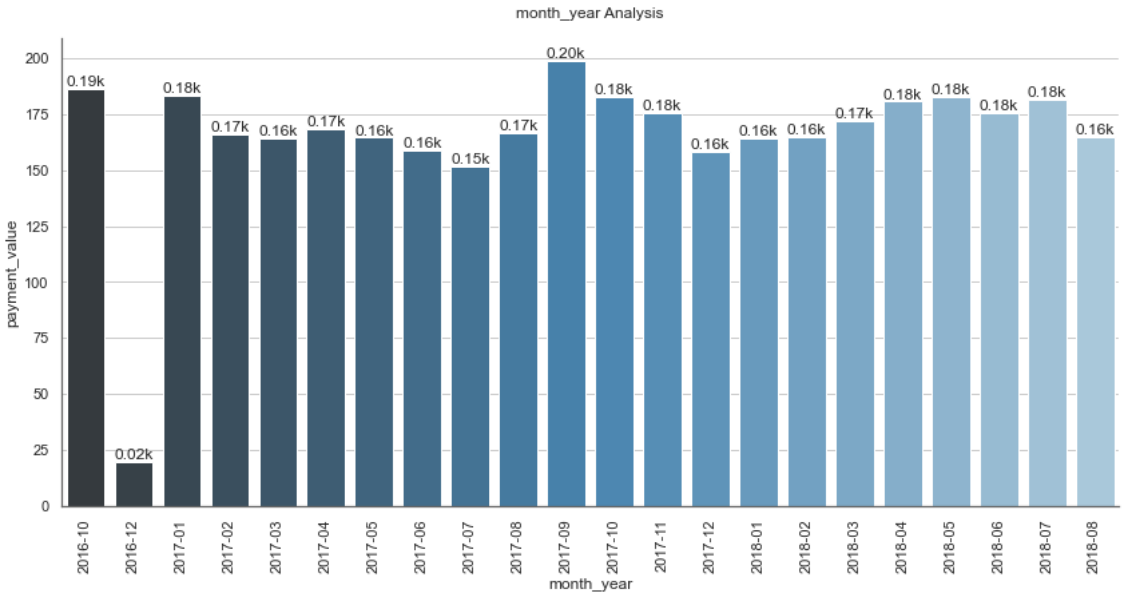
```
#creating monthly active customers dataframe by counting unique Customer IDs
df_monthly_sales = df.groupby('month_year')['order_status'].count().reset_index()
```

```
fig, ax = plt.subplots(figsize=(12, 6))
sns.set(palette='muted', color_codes=True, style='whitegrid')
bar_plot(x='month_year', y='order_status', df=df_monthly_sales, value=True)
ax.tick_params(axis='x', labelrotation=90)
plt.show()
```



Average Revenue per Customer Purchase

```
# create a new dataframe for average revenue by taking the mean of it
df_monthly_order_avg = df.groupby('month_year')['payment_value'].mean().reset_index()

fig, ax = plt.subplots(figsize=(12, 6))
sns.set(palette='muted', color_codes=True, style='whitegrid')
bar_plot(x='month_year', y='payment_value', df=df_monthly_order_avg, value=True)
ax.tick_params(axis='x', labelrotation=90)
plt.show()
```



Inference

- The average revenue has remained more or less flat over time.

New Customer Ratio

Let's first find if a given order is from an new or repeat (existing) customer.

```
# Create a dataframe contaning CustomerID and first purchase date
df_min_purchase = df.groupby('customer_unique_id').order_purchase_timestamp.min().reset_index()  # earliest purchase dat
df_min_purchase.columns = ['customer_unique_id','minpurchasedate']
```

```
df_min_purchase['minpurchasedate'] = df_min_purchase['minpurchasedate'].map(lambda date: 100*date.year + date.month)

# Merge first purchase date column to our main dataframe (tx_uk)
df = pd.merge(df, df_min_purchase, on='customer_unique_id')
df.head(7)
```

	order_id	payment_sequential	payment_type	payment_installments	payment_value	order_item_id	
0	b81ef226f3fe1789b1e8b2acac839d17	1	Credit Card	8	99.33	1	af74cc53
1	a9810da82917af2d9aefd1278f1dcfa0	1	Credit Card	1	24.39	1	a630cc32
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	Credit Card	1	65.71	1	2028bf1b
3	ba78997921bbcdc1373bb41e913ab953	1	Credit Card	8	107.78	1	548e5bfe
4	42fdf880ba16b47b59251dd489d4441a	1	Credit Card	2	128.45	1	38648636
5	f16cbac8a72057c2daba4b64662100ba	1	Credit Card	1	99.29	1	7940f574
6	f16cbac8a72057c2daba4b64662100ba	2	Credit Card	2	400.00	1	7940f574

```
# create a column called User Type and assign Existing
# if User's First Purchase Year Month before the selected Invoice Year Month
df['usertype'] = 'New'
df.loc[df['month_y']>df['minpurchasedate'],'usertype'] = 'Existing'
df.head()
```

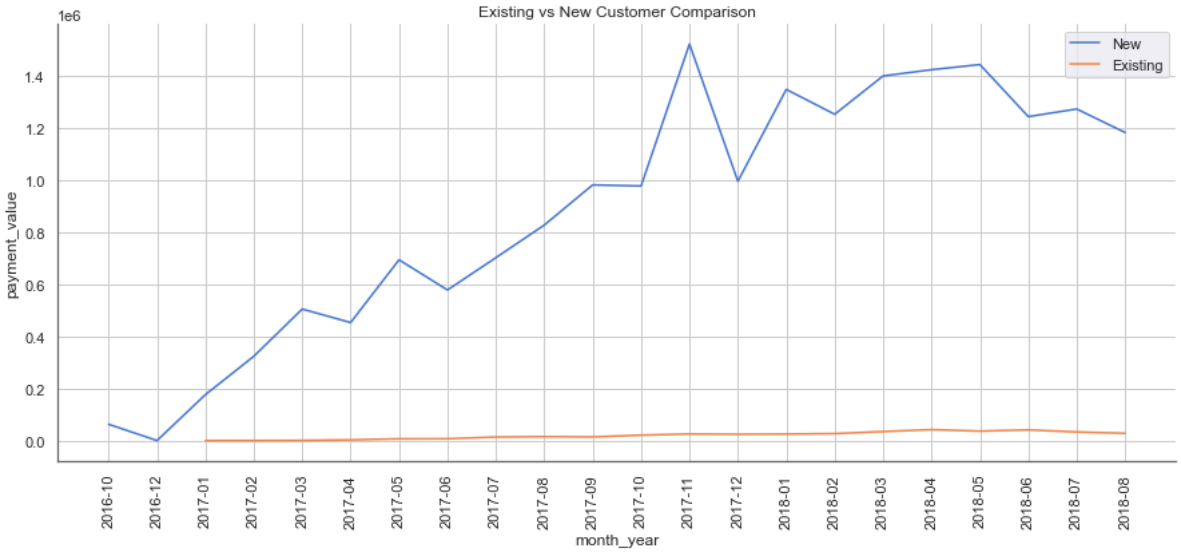
	order_id	payment_sequential	payment_type	payment_installments	payment_value	order_item_id	
0	b81ef226f3fe1789b1e8b2acac839d17	1	Credit Card	8	99.33	1	af74cc53
1	a9810da82917af2d9aefd1278f1dcfa0	1	Credit Card	1	24.39	1	a630cc32
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	Credit Card	1	65.71	1	2028bf1b
3	ba78997921bbcdc1373bb41e913ab953	1	Credit Card	8	107.78	1	548e5bfe
4	42fdf880ba16b47b59251dd489d4441a	1	Credit Card	2	128.45	1	38648636

Revenue from new vs repeat (existing) customers

```
#calculate the Revenue per month for each user type
df_user_type_revenue = df.groupby(['month_y','usertype', 'month_year'])['payment_value'].sum().reset_index()
df_user_type_revenue
```

	month_y	usertype	month_year	payment_value
0	201610	New	2016-10	62591.65
1	201612	New	2016-12	19.62
2	201701	Existing	2017-01	19.62
3	201701	New	2017-01	176356.94
4	201702	Existing	2017-02	111.07
5	201702	New	2017-02	323704.88
6	201703	Existing	2017-03	596.38
7	201703	New	2017-03	505139.45
8	201704	Existing	2017-04	2789.06
9	201704	New	2017-04	453319.26
10	201705	Existing	2017-05	6733.95
11	201705	New	2017-05	694385.65
12	201706	Existing	2017-06	6956.06
13	201706	New	2017-06	578444.92
14	201707	Existing	2017-07	13632.49
15	201707	New	2017-07	702437.49
16	201708	Existing	2017-08	15000.05
17	201708	New	2017-08	827689.89
18	201709	Existing	2017-09	14067.94
19	201709	New	2017-09	982017.67
20	201710	Existing	2017-10	20695.65
21	201710	New	2017-10	977913.97
22	201711	Existing	2017-11	25261.14
23	201711	New	2017-11	1523286.72
24	201712	Existing	2017-12	24133.48
25	201712	New	2017-12	995933.78
26	201801	Existing	2018-01	25079.90
27	201801	New	2018-01	1348984.12
28	201802	Existing	2018-02	26661.62

```
fig, ax = plt.subplots(figsize=(15, 6))
sns.set(palette='muted', color_codes=True)
ax = sns.lineplot(x='month_year', y='payment_value', data=df_user_type_revenue.query("usertype == 'New'"), label='New')
ax = sns.lineplot(x='month_year', y='payment_value', data=df_user_type_revenue.query("usertype == 'Existing'"), label='E
format_spines(ax, right_border=False)
ax.set_title('Existing vs New Customer Comparison')
ax.tick_params(axis='x', labelrotation=90)
plt.show()
```



Inference

Only a very small percentage of sales come from existing customers. Let's study the repeat purchases rate.

Repeat Customer Ratio

Note: Only customer returning the same month are considered as repeat in this case.

```
# Create a dataframe that shows new user ratio - we also need to drop NA values (first month new user ratio is 0)
df_user_ratio = df.query("usertype == 'Existing'").groupby(['month_year'])['customer_unique_id'].nunique() / df.query("u
df_user_ratio = df_user_ratio.reset_index()

#dropping nan values that resulted from first and last month
df_user_ratio = df_user_ratio.dropna()
df_user_ratio.columns = ['month_year','RepeatCusRatio']

#print the dafaframe
df_user_ratio
```

	month_year	RepeatCusRatio
2	2017-01	0.001399
3	2017-02	0.001238
4	2017-03	0.001998
5	2017-04	0.007979
6	2017-05	0.008116
7	2017-06	0.012842
8	2017-07	0.013326
9	2017-08	0.014050
10	2017-09	0.019735
11	2017-10	0.020328
12	2017-11	0.017425
13	2017-12	0.020982
14	2018-01	0.019293
15	2018-02	0.017809
16	2018-03	0.020667
17	2018-04	0.024613
18	2018-05	0.028743
19	2018-06	0.031149
20	2018-07	0.025395
21	2018-08	0.027018

```
fig, ax = plt.subplots(figsize=(12, 6))
sns.set(palette='muted', color_codes=True, style='whitegrid')
bar_plot(x='month_year', y='RepeatCusRatio', df=df_user_ratio)
ax.tick_params(axis='x', labelrotation=90)
plt.show()
```

Inference

- Repeat customers average approx ~5%.

Monthly Retention Rate (optional)

Monthly Retention Rate = Retained Customers From Prev. Month / Active Customers Total (using crosstab)

The idea can be extended to larger duration, say yearly.

```
# Identifying active users are active by looking at their revenue per month
df_user_purchase = df.groupby(['customer_unique_id','month_y'])['payment_value'].sum().reset_index()
df_user_purchase.head()
```

	customer_unique_id	month_y	payment_value
0	0000366f3b9a7992bf8c76cfdf3221e2	201805	141.90
1	0000b849f77a49e4a4ce2b2a4ca5be3f	201805	27.19
2	0000f46a3911fa3c0805444483337064	201703	86.22
3	0000f6ccb0745a6a4b88665a16c9f078	201710	43.62
4	0004aac84e0df4da2b147fca70cf8255	201711	196.89

```
# identifying active users are active by looking at their order count per month
df_user_purchase = df.groupby(['customer_unique_id','month_y'])['payment_value'].count().reset_index()
df_user_purchase.head()
```

	customer_unique_id	month_y	payment_value
0	0000366f3b9a7992bf8c76cfdf3221e2	201805	1
1	0000b849f77a49e4a4ce2b2a4ca5be3f	201805	1
2	0000f46a3911fa3c0805444483337064	201703	1
3	0000f6ccb0745a6a4b88665a16c9f078	201710	1
4	0004aac84e0df4da2b147fca70cf8255	201711	1

```
# create retention matrix with crosstab using purchase
df_retention = pd.crosstab(df_user_purchase['customer_unique_id'], df_user_purchase['month_y']).reset_index()
df_retention.head(10)
```

month_y	customer_unique_id	201610	201612	201701	201702	201703	201704	201705	201706	201707	201708	201709	20
0	0000366f3b9a7992bf8c76cfdf3221e2	0	0	0	0	0	0	0	0	0	0	0	
1	0000b849f77a49e4a4ce2b2a4ca5be3f	0	0	0	0	0	0	0	0	0	0	0	
2	0000f46a3911fa3c0805444483337064	0	0	0	0	1	0	0	0	0	0	0	
3	0000f6ccb0745a6a4b88665a16c9f078	0	0	0	0	0	0	0	0	0	0	0	
4	0004aac84e0df4da2b147fca70cf8255	0	0	0	0	0	0	0	0	0	0	0	
5	0004bd2a26a76fe21f786e4fbd80607f	0	0	0	0	0	0	0	0	0	0	0	
6	00050ab1314c0e55a6ca13cf7181fecf	0	0	0	0	0	0	0	0	0	0	0	
7	00053a61a98854899e70ed204dd4bafe	0	0	0	0	0	0	0	0	0	0	0	
8	0005e1862207bf6ccc02e4228effd9a0	0	0	0	0	1	0	0	0	0	0	0	
9	0005ef4cd20d2893f0d9fbd94d3c0d97	0	0	0	0	0	0	0	0	0	0	0	

```
# Creating an array of dictionary which keeps Retained & Total User count for each month
months = df_retention.columns[2:]
retention_array = []
```

```
for i in range(len(months)-1):
    retention_data = {}
    selected_month = months[i+1]
    prev_month = months[i]
    retention_data['month_y'] = int(selected_month)
    retention_data['TotalUserCount'] = df_retention[selected_month].sum()
    retention_data['RetainedUserCount'] = df_retention[(df_retention[selected_month]>0) & (df_retention[prev_month]>0)][
    retention_array.append(retention_data)

#convert the array to dataframe and calculate Retention Rate
df_retention = pd.DataFrame(retention_array)
df_retention['RetentionRate'] = df_retention['RetainedUserCount']/df_retention['TotalUserCount']

df_retention
```

	month_y	TotalUserCount	RetainedUserCount	RetentionRate
0	201701	716	1	0.001397
1	201702	1618	2	0.001236
2	201703	2508	3	0.001196
3	201704	2274	11	0.004837
4	201705	3478	14	0.004025
5	201706	3076	16	0.005202
6	201707	3802	16	0.004208
7	201708	4114	23	0.005591
8	201709	4082	32	0.007839
9	201710	4417	32	0.007245
10	201711	7182	37	0.005152
11	201712	5450	41	0.007523
12	201801	6974	16	0.002294
13	201802	6401	27	0.004218
14	201803	6914	23	0.003327
15	201804	6744	31	0.004597
16	201805	6693	45	0.006723
17	201806	6058	38	0.006273
18	201807	6097	26	0.004264
19	201808	6310	37	0.005864

```
fig, ax = plt.subplots(figsize=(12, 6))
sns.set(palette='muted', color_codes=True, style='whitegrid')
bar_plot(x='month_y', y='RetentionRate', df=df_retention, value=True)
ax.tick_params(axis='x', labelrotation=90)
plt.show()
```

7. Data fixes for Model Building

We can see that in the Orders dataset we have a column called *order_status* which includes the current status of each order. We are only interested in working with the orders that have been concluded so far, so we must filter them.


```
orders['order_status'].value_counts()
```

```
delivered      96478
shipped         1107
canceled         625
unavailable      609
invoiced         314
processing       301
created            5
approved           2
Name: order_status, dtype: int64
```



```
orders = orders[orders['order_status'] == 'delivered']
orders.shape
```

```
(96478, 8)
```

We are left with a sample of 96,478 completed orders.

Merging our data

Let's start with merging the three datasets into one from which we can extract the variables we need.

```
# We are left with only the columns we need

orders = orders[['customer_id', 'order_id', 'order_purchase_timestamp']]
customers = customers[['customer_id', 'customer_unique_id']]
payments = payments[['order_id', 'payment_value']]
```

```
# We perform the union of the datasets by their respective primary keys

df = pd.merge(orders, customers, how='inner', on='customer_id')
df = pd.merge(df, payments, how='inner', on='order_id')
df
```

	customer_id	order_id	order_purchase_timestamp	customer_unique
0	9ef432eb6251297304e76186b10a928d	e481f51cbdc54678b7cc49136f2d6af7	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e
1	9ef432eb6251297304e76186b10a928d	e481f51cbdc54678b7cc49136f2d6af7	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e
2	9ef432eb6251297304e76186b10a928d	e481f51cbdc54678b7cc49136f2d6af7	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e
3	b0830fb4747a6c6d20dea0b8c802d7ef	53cdb2fc8bc7dce0b6741e2150273451	2018-07-24 20:41:37	af07308b275d755c9edb36a90c618
4	41ce2a54c0b03bf3443c3d931a367089	47770eb9100c2d0c44946d9cf07ec65d	2018-08-08 08:38:49	3a653a41f6f9fc3d2a113cf839868
...	...	...	...	
100751	39bd1228ee8140590ac3aca26f2dfe00	9c5dedf39a927c1b2549525ed64a053c	2017-03-09 09:54:05	6359f309b166b0196dbf7ad2ac62b
100752	1fca14ff2861355f6e5f14306ff977a7	63943bddc261676b46f01ca7ac2f7bd8	2018-02-06 12:58:58	da62f9e57a76d978d02ab5362c509
100753	1aa71eb042121263aafbe80c1b562c9c	83c1379a015df1e13d02aae0204711ab	2017-08-27 14:46:43	737520a9aad80b3fbbdad19b66b37
100754	b331b74b18dc79bcdf6532d51e1637c1	11c177c8e97725db2631073c19f07b62	2018-01-08 21:28:27	5097a5312c8b157bb7be58ae360ef
100755	edb027a75a1449115f6b43211ae02a24	66dea50a8b16d9b4dee7af250b4be1a5	2018-03-08 20:57:30	60350aa974b26ff12caad89e55993

100756 rows × 5 columns

Drop Customer_id

Here we have the final merge, but it is necessary to make a differentiation between the variables that we have: **order_id** clearly refers to the code of the order carried out, on the other hand we have **customer_id** which is the code assigned to the user by purchase (that is, the code varies from session to session, even if the person is the same) and finally **customer_unique_id** which actually reflects the user as a person, which is what we need to calculate their respective CLTV.

So, we will proceed to eliminate the other two columns that do not serve us for the case.

```
df = df.drop(['customer_id', 'order_id'], axis=1)
```

Let's continue doing basic preprocessing, checking for null or duplicate data.

```
df.isnull().sum(axis=0)
```

```
order_purchase_timestamp    0
customer_unique_id          0
payment_value               0
dtype: int64
```

```
df.drop_duplicates(inplace=True)
df.shape
```

```
(100137, 3)
```

```
df['customer_unique_id'].nunique()
```

```
93357
```

14. Customer Segmentation Analysis

RFM Metrics

When a customer contact's you for sales / support queries or when a business reaches out to customers for marketing outreach, you want to be well informed of what the 'state' of a customer.

That is, you want to know in terms of **'how often'** a customer transacted with you, **'how recent'** a transaction has been done and the **'quantum'** of the transactions.

A customer who bought 10 times from you over the last 30 days is likely to be far more valuable than one who made a purchase 3 years back and hasn't come back since.

The marketing message you want to give the these customers should be different as well.

So, an effective way to segment customers is based on:

1. Recency (**R**) - How long since the last purchase?
2. Frequency (**F**) - How many purchases?
3. Monetary aspect. (**M**) - How much purchased?

Note: The definitions above for RFM are for segmentation modeling only. The definition for RFM when we do predictive models for CLTV will change slightly. More on that later.

How to segment customers?

For the convenience of your CRM and marketing teams, we might create human understandable segments as: Low value, Medium Value and High value.

Low Value: Less frequent, Purchased long time back in small revenue.

High Value: Very frequent, Recently purchased and large dollar value.

Medium Value: Lies between Low and High in all 3 metrics.

Now, to actually create these segments, instead of setting a arbitrary human fixed value, let's find out if there are inherent clusters / segments arising based on the data. Let's code.

Recency

```
df.head()
```

	order_purchase_timestamp	customer_unique_id	payment_value
0	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e5bff	18.12
1	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e5bff	2.00
2	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e5bff	18.59
3	2018-07-24 20:41:37	af07308b275d755c9edb36a90c618231	141.46
4	2018-08-08 08:38:49	3a653a41f6f9fc3d2a113cf8398680e8	179.12

Take the maximum order date as the reference date for model building / scoring.

```
date_today = df['order_purchase_timestamp'].max()
date_today
```

```
'2018-08-29 15:00:37'
```

Compute minimum and maximum purchase date for each customer.

```
df_R = df.groupby('customer_unique_id').agg(['min', 'max'])['order_purchase_timestamp']
df_R.head()
```

customer_unique_id	min	max
0000366f3b9a7992bf8c76cfdf3221e2	2018-05-10 10:56:27	2018-05-10 10:56:27
0000b849f77a49e4a4ce2b2a4ca5be3f	2018-05-07 11:11:27	2018-05-07 11:11:27
0000f46a3911fa3c0805444483337064	2017-03-10 21:05:03	2017-03-10 21:05:03
0000f6ccb0745a6a4b88665a16c9f078	2017-10-12 20:29:41	2017-10-12 20:29:41
0004aac84e0df4da2b147fca70cf8255	2017-11-14 19:45:42	2017-11-14 19:45:42

```
df_R['recency'] = (pd.to_datetime(date_today, ) - pd.to_datetime(df_R['max'])).dt.round('d')
df_R.head()
```

customer_unique_id	min	max	recency
0000366f3b9a7992bf8c76cfdf3221e2	2018-05-10 10:56:27	2018-05-10 10:56:27	111 days
0000b849f77a49e4a4ce2b2a4ca5be3f	2018-05-07 11:11:27	2018-05-07 11:11:27	114 days
0000f46a3911fa3c0805444483337064	2017-03-10 21:05:03	2017-03-10 21:05:03	537 days
0000f6ccb0745a6a4b88665a16c9f078	2017-10-12 20:29:41	2017-10-12 20:29:41	321 days
0004aac84e0df4da2b147fca70cf8255	2017-11-14 19:45:42	2017-11-14 19:45:42	288 days

Frequency and Monetary

```
aggregations = {
    'order_purchase_timestamp':'count',
    'payment_value': 'sum'}

df_F = df.groupby('customer_unique_id').agg(aggregations)
df_F.rename(columns={"order_purchase_timestamp":"frequency",
                     "payment_value":"monetary"}, inplace=True)
df_F.head()
```

customer_unique_id	frequency	monetary
0000366f3b9a7992bf8c76cfdf3221e2	1	141.90
0000b849f77a49e4a4ce2b2a4ca5be3f	1	27.19
0000f46a3911fa3c0805444483337064	1	86.22
0000f6ccb0745a6a4b88665a16c9f078	1	43.62
0004aac84e0df4da2b147fca70cf8255	1	196.89

Merge Data

```
df_rfm = pd.merge(df_R, df_F, left_index=True, right_index=True)
df_rfm.drop(['min', 'max'], axis=1, inplace=True)
df_rfm.head()
```

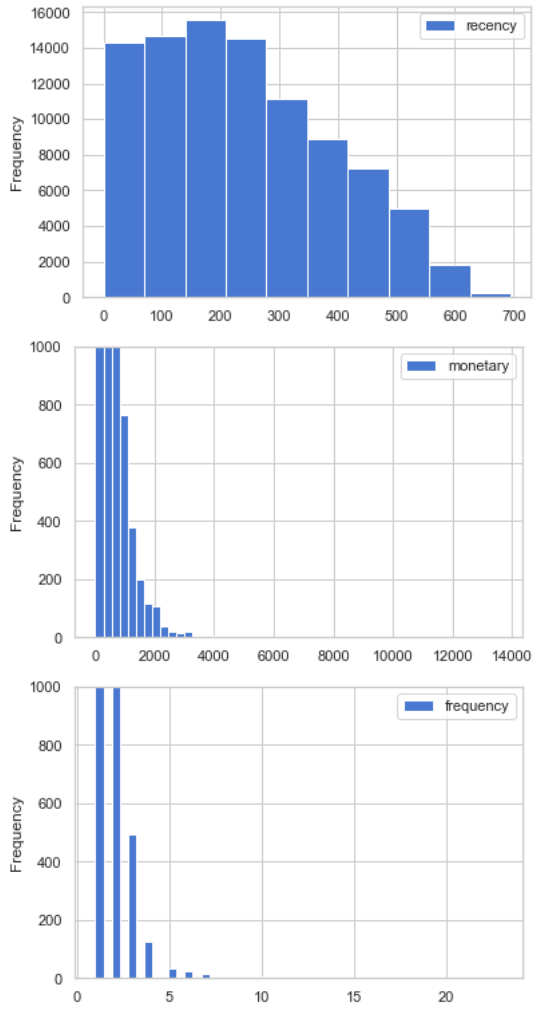
customer_unique_id	recency	frequency	monetary
0000366f3b9a7992bf8c76cfdf3221e2	111 days	1	141.90
0000b849f77a49e4a4ce2b2a4ca5be3f	114 days	1	27.19
0000f46a3911fa3c0805444483337064	537 days	1	86.22
0000f6ccb0745a6a4b88665a16c9f078	321 days	1	43.62

```
df_rfm['recency'] = df_rfm['recency'].dt.days.astype('int')
```

Plot Histogram

```
df_rfm.plot(y="recency", kind='hist')
df_rfm.plot(y="monetary", kind='hist', bins=50, ylim=(0,1000))
df_rfm.plot(y="frequency", kind='hist', bins=50, ylim=(0,1000))
```

```
<AxesSubplot:ylabel='Frequency'>
```



Describe

```
df_rfm.describe()
```

	recency	frequency	monetary
min	0.000000	1.000000	6.180000

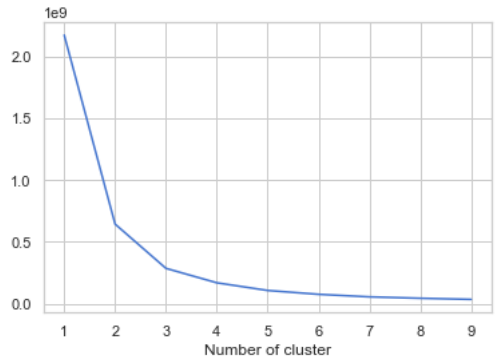
15. Start Clustering

1. Based on Recency

```
from sklearn.cluster import KMeans

sse={}
df_recency = df_rfm[['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_
```

inertia: Sum of squared distances of samples to their closest cluster center. Larger the inertia, the more separated is the cluster.

```
# Plot
plt.figure()
plt.plot(list(sse.keys()), list(sse.values()))
plt.xlabel("Number of cluster")
plt.show()
```



3 clusters seem to be ok for elbow point. Based on biz context, you may also choose a different number, say 2 or 4.

Determine the recency segment/cluster for each customer

```
# 3 clusters for recency
kmeans = KMeans(n_clusters=3)
kmeans.fit(df_recency)
df_rfm['RecencyCluster'] = kmeans.predict(df_recency)
df_rfm.head(7)
```

customer_unique_id	recency	frequency	monetary	RecencyCluster
0000366f3b9a7992bf8c76cfdf3221e2	111	1	141.90	0
0000b849f77a49e4a4ce2b2a4ca5be3f	114	1	27.19	0
0000f46a3911fa3c0805444483337064	537	1	86.22	1
0000f6ccb0745a6a4b88665a16c9f078	321	1	43.62	2
0004aac84e0df4da2b147fca70cf8255	288	1	196.89	2
0004bd2a26a76fe21f786e4fbd80607f	146	1	166.98	0
00050ab1314c0e55a6ca13cf7181fecf	131	1	35.38	0

```
df_rfm.RecencyCluster.value_counts()
```

```
0    36208
2    35260
1    21889
Name: RecencyCluster, dtype: int64
```

```
df_rfm.groupby('RecencyCluster').agg({'recency':[np.mean, len]})
```

	recency	
	mean	len
RecencyCluster		
0	87.292007	36208
1	457.560601	21889
2	255.173625	35260

There is a problem.

The ordering of clusters numbers does not correlate with the mean recency. That is, we cant say, as the cluster number increases, the value of the recency increases (or decreases) monotonically.

Let's correct that.

```
# Re-ordering cluster numbers
def fix_cluster_order(cluster_number_field, cluster_field, df, ascending):
    df_new              = df.groupby(cluster_number_field)[cluster_field].mean().reset_index()
    df_new              = df_new.sort_values(by=cluster_field, ascending=ascending).reset_index(drop=True)
    df_new['index']     = df_new.index
    df_final            = pd.merge(df, df_new[[cluster_number_field,'index']], on=cluster_number_field)
    df_final            = df_final.drop([cluster_number_field],axis=1)
    df_final            = df_final.rename(columns={"index":cluster_number_field})
    return df_final
```

```
df_rfm = fix_cluster_order('RecencyCluster', 'recency', df_rfm, False)
```

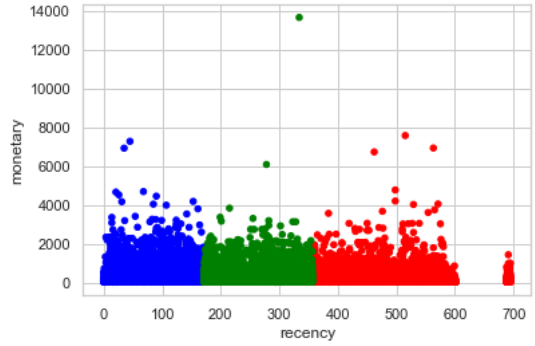
```
df_rfm.groupby('RecencyCluster').agg({'recency':[np.mean, len]})
```

	recency	
	mean	len
RecencyCluster		
0	457.560601	21889
1	255.173625	35260
2	87.292007	36208

Plot

```
df_rfm.plot(x='recency', y='monetary', kind='scatter', color=df_rfm.RecencyCluster.map({0:'red', 1:'green', 2:'blue'}))
```

```
<AxesSubplot:xlabel='recency', ylabel='monetary'>
```



Inference

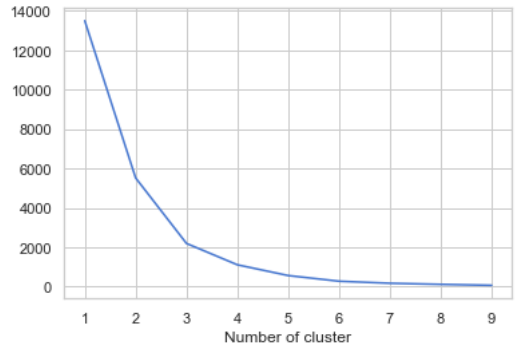
No clear segregated pattern arise indicating a relation between recency and revenue.

16. Based on Frequency

```
from sklearn.cluster import KMeans
```

```
sse={}
df_frequency = df_rfm[['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_
```

```
# Plot
plt.figure()
plt.plot(list(sse.keys()), list(sse.values()))
plt.xlabel("Number of cluster")
plt.show()
```



Determine the Frequency Segment for each customer

```
df_frequency.head()
```

	frequency	clusters
0	1	0
1	1	0
2	1	0
3	1	0
4	1	0

```
# 3 clusters
kmeans = KMeans(n_clusters=3)
kmeans.fit(df_rfm[['frequency']])
df_rfm['FrequencyCluster'] = kmeans.predict(df_rfm[['frequency']])
df_rfm.head(7)
```

	recency	frequency	monetary	RecencyCluster	FrequencyCluster
0	111	1	141.90	2	0
1	114	1	27.19	2	0
2	146	1	166.98	2	0
3	131	1	35.38	2	0
4	170	1	129.76	2	0
5	158	1	63.66	2	0
6	128	1	82.05	2	0

```
df_rfm.FrequencyCluster.value_counts()
```

```
0    87934
1     5348
2       75
Name: FrequencyCluster, dtype: int64
```

```
df_rfm.groupby('FrequencyCluster').agg({'frequency':[np.mean, len]})
```

	frequency	
	mean	len
FrequencyCluster		

The ordering of cluster number.

```
df_rfm = fix_cluster_order('FrequencyCluster', 'frequency', df_rfm, True)
```

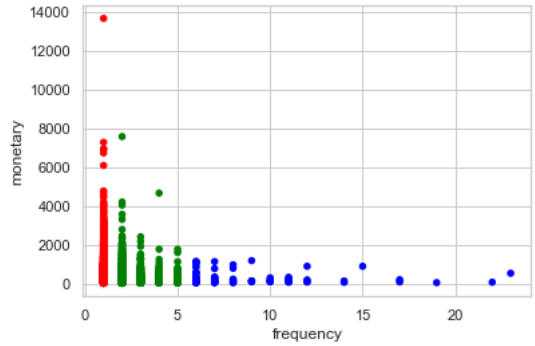
```
df_rfm.groupby('FrequencyCluster').agg({'frequency':[np.mean, len]})
```

	frequency	
	mean	len
FrequencyCluster		
0	1.000000	87934
1	2.157442	5348
2	8.866667	75

Plot

```
df_rfm.plot(x='frequency', y='monetary', kind='scatter', color=df_rfm.FrequencyCluster.map({0:'red', 1:'green', 2:'blue'
```

```
<AxesSubplot:xlabel='frequency', ylabel='monetary'>
```



Inferences

- Lower frequncies have reached larger revenues.
- More frequency does not always translate as larger revenue number, though they might be valuable in long run.

```
df_rfm.head()
```

	recency	frequency	monetary	RecencyCluster	FrequencyCluster
0	111	1	141.90	2	0
1	114	1	27.19	2	0
2	146	1	166.98	2	0
3	131	1	35.38	2	0
4	170	1	129.76	2	0

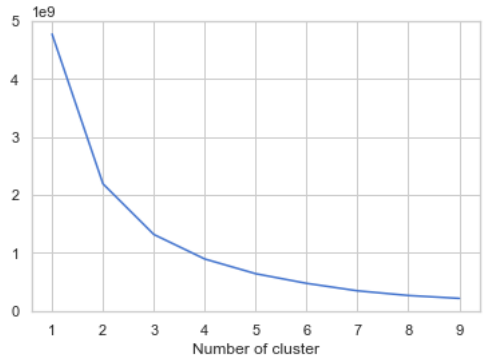
```
# Summarize
df_rfm.groupby('FrequencyCluster')[['frequency', 'monetary']].describe()
```

	frequency								monetary							
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	
FrequencyCluster																
0	87934.0	1.000000	0.000000	1.0	1.0	1.0	1.0	1.0	87934.0	160.563598	220.470202	10.07	62.0600	105.625	176.87	
1	5348.0	2.157442	0.465257	2.0	2.0	2.0	2.0	5.0	5348.0	235.715079	293.891020	6.18	86.1725	157.905	276.72	
2	75.0	8.866667	3.739249	6.0	6.0	7.0	10.5	23.0	75.0	291.171467	325.876002	28.19	80.9550	147.150	313.80	

17. Based on Monetary

```
from sklearn.cluster import KMeans

sse={}
df_monetary = df_rfm[['monetary']]
for k in range(1, 10):
    kmeans = KMeans(n_clusters=k, max_iter=1000).fit(df_monetary)
    df_monetary["clusters"] = kmeans.labels_
    sse[k] = kmeans.inertia_
```

```
# Plot
plt.figure()
plt.plot(list(sse.keys()), list(sse.values()))
plt.xlabel("Number of cluster")
plt.show()
```



Determine the Monetary Segment for each customer

```
# 3 clusters
kmeans = KMeans(n_clusters=3)
kmeans.fit(df_rfm[['monetary']])
df_rfm['MonetaryCluster'] = kmeans.predict(df_rfm[['monetary']])
df_rfm.head(7)
```

	recency	frequency	monetary	RecencyCluster	FrequencyCluster	MonetaryCluster
0	111	1	141.90	2	0	0
1	114	1	27.19	2	0	0
2	146	1	166.98	2	0	0
3	131	1	35.38	2	0	0
4	170	1	129.76	2	0	0
5	158	1	63.66	2	0	0
6	128	1	82.05	2	0	0

```
df_rfm.MonetaryCluster.value_counts()
```

```
0    83127
2     9309
1      921
Name: MonetaryCluster, dtype: int64
```

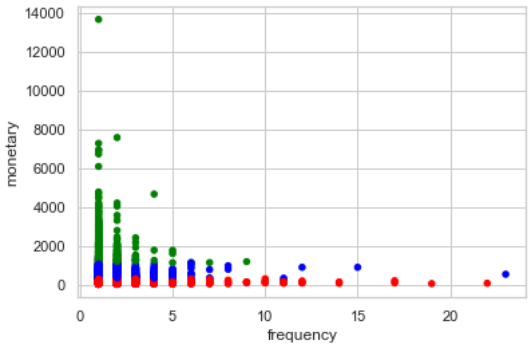
```
df_rfm.groupby('MonetaryCluster').agg({'monetary':[np.mean, len]})
```

	monetary	
	mean	len
MonetaryCluster		
0	111.050319	83127.0
1	1715.598545	921.0
2	493.081466	9309.0

Plot

```
df_rfm.plot(x='frequency', y='monetary', kind='scatter', color=df_rfm.MonetaryCluster.map({0:'red', 1:'green', 2:'blue',
```

```
<AxesSubplot:xlabel='frequency', ylabel='monetary'>
```




```
df_rfm.head()
```

	recency	frequency	monetary	RecencyCluster	FrequencyCluster	MonetaryCluster
0	111	1	141.90	2	0	0
1	114	1	27.19	2	0	0
2	146	1	166.98	2	0	0
3	131	1	35.38	2	0	0
4	170	1	129.76	2	0	0

19. Overall segmentation and ideas

```
# Approach 1: Sum
df_rfm['SegmentScore'] = df_rfm[['RecencyCluster', 'FrequencyCluster', 'MonetaryCluster']].apply(np.sum, axis=1)

# Approach 2: Product
df_rfm['SegmentScore2'] = (df_rfm['RecencyCluster']+1) * (df_rfm['FrequencyCluster']+1) * (df_rfm['MonetaryCluster']+1)

df_rfm.head()
```

	recency	frequency	monetary	RecencyCluster	FrequencyCluster	MonetaryCluster	SegmentScore	SegmentScore2
0	111	1	141.90	2	0	0	2	3
1	114	1	27.19	2	0	0	2	3
2	146	1	166.98	2	0	0	2	3
3	131	1	35.38	2	0	0	2	3
4	170	1	129.76	2	0	0	2	3

Mean by segmentscore across R, F, M.

```
# Approach 1
print(df_rfm.groupby('SegmentScore').agg({'recency':[len, np.mean], 'frequency':np.mean, 'monetary':np.mean}))

# Approach 2
print(df_rfm.groupby('SegmentScore2').agg({'recency':[len, np.mean], 'frequency':np.mean, 'monetary':np.mean}))
```

```
               recency             frequency    monetary
                   len        mean      mean        mean
SegmentScore
0                18446  457.155481  1.000000  107.906522
1                31104  262.907407  1.039255  121.427724
2                34440  117.727816  1.060134  145.870172
3                 5231  204.991589  1.408908  480.225406
4                 3653  107.978648  1.179852  506.952007
5                  474   87.430380  2.337553  506.556561
6                    9   80.888889  7.444444  697.617778
              recency             frequency     monetary
                 len        mean      mean         mean
SegmentScore2
1              18446  457.155481  1.000000   107.906522
```

```
2              31104  262.907407  1.039255   121.427724
3              32575  109.698511  1.007245   133.768088
4               1865  257.971582  1.983914   357.251072
6               5186  204.496529  1.401465   471.354487
8                 45  262.044444  2.266667  1502.549556
9               3203   89.508586  1.037777   495.380921
12               450  239.444444  2.191111   589.312422
18               474   87.430380  2.337553   506.556561
27                 9   80.888889  7.444444   697.617778
```

Observe: The trend is not strictly monotonically increasing for Recency and monetary, whereas it does for frequency. Indication that monetary is not perfectly correlated with frequency.

For sake of understanding, we may club some of these and form human readable groups:

- 0, 1: Low (purchased long back, not frequent, low monetary)
- 2, 3: Mid
- 4, 5: High (purchased recenly, frequent, high monetary)

However, **a 'Low' customer should not be thought of strictly as a low value customer.** For instance, a customer who purchased long back but more frequently, might be a valueable customer the business might want to win back.

In such case, instead of sum, you might want to the minimum of the RFM cluster numbers.

Let's apply these.

```
# Approach 1
df_rfm['Segment'] = df_rfm['SegmentScore'].map({0:"Low", 1:"Low", 2:"Mid", 3:"Mid", 4:"High", 5:"High", 6:"High"})

# Approach 2
df_rfm['Segment2'] = 'Low'
df_rfm.loc[df_rfm['SegmentScore2']>2,'Segment2'] = 'Mid'
df_rfm.loc[df_rfm['SegmentScore2']>9,'Segment2'] = 'High'
```

Calculate Segment group means

```
df_rfm.groupby('Segment').agg({'recency':[len, np.mean], 'frequency':np.mean, 'monetary':np.mean})
```

	recency		frequency	monetary
	len	mean	mean	mean
Segment				
High	4136	105.564797	1.326161	507.321579
Low	49550	335.220222	1.024642	116.394181
Mid	39671	129.234378	1.106123	189.958101

```
df_rfm.groupby('Segment2').agg({'recency':[len, np.mean], 'frequency':np.mean, 'monetary':np.mean})
```

	recency		frequency	monetary
	len	mean	mean	mean
Segment2				
High	933	160.685959	2.316184	548.313998
Low	49550	335.220222	1.024642	116.394181
Mid	42874	126.266572	1.101017	212.775410

Let's Visualize the segments

```
df_rfm.head()
```

```
        recency  frequency  monetary  RecencyCluster  FrequencyCluster  MonetaryCluster  SegmentScore  SegmentScore2  Segment  Segment2
```

Approach 1 (Sum): Freq vs Monetary

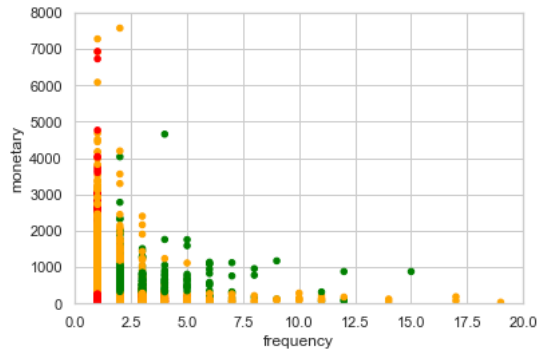
```
from matplotlib import colors
np.random.seed(101)
```

```
# color_sel = [color_list[i] for i in df_rfm.SegmentScore ]
```

F vs M - Approach 1: Color by Segment1

```
df_rfm.plot(x='frequency', y='monetary', kind='scatter',
            color=df_rfm.Segment.map({'Low':'red', 'Mid':'orange', 'High':'green'}),
            ylim=[0,8000],
            xlim=[0,20])
```

```
<AxesSubplot:xlabel='frequency', ylabel='monetary'>
```



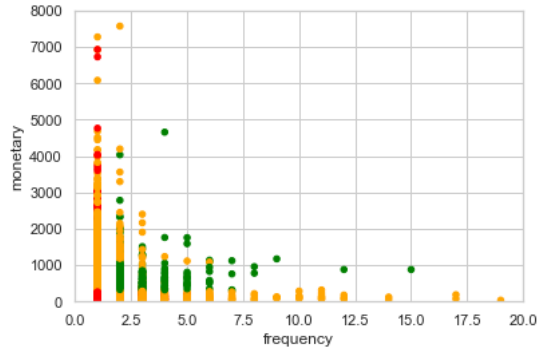
Note: Some of the low frequency with high monetary value are marked in 'High' Category (Green dots in 1'st column in chart above). Sometimes it might be preferable to mark them in 'Mid' or 'Low', depending on business strategy.

In such case, approach 2 might be preferred.

F vs M - Approach 2: Color by Segment2

```
df_rfm.plot(x='frequency', y='monetary', kind='scatter',
            color=df_rfm.Segment2.map({'Low':'red', 'Mid':'orange', 'High':'green'}),
            ylim=[0,8000],
            xlim=[0,20])
```

```
<AxesSubplot:xlabel='frequency', ylabel='monetary'>
```

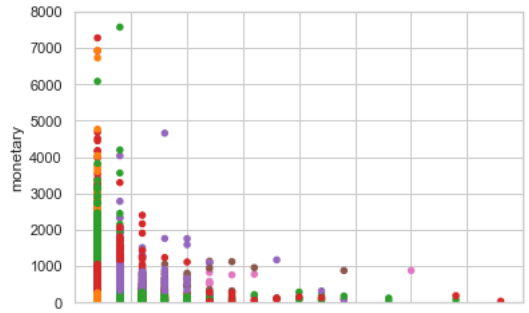


We can try to color by segment score, which would give more clusters, an require more individual examination to make sense of the clusters.

Color By Segment Score (tough to understand)

```
# Color by segment score using tableau colors
color_segscore = [list(colors.TABLEAU_COLORS.keys())[i] for i in df_rfm.SegmentScore ]
df_rfm.plot(x='frequency', y='monetary', kind='scatter',
            color=color_segscore,
            ylim=[0,8000],
            xlim=[0,20])
```

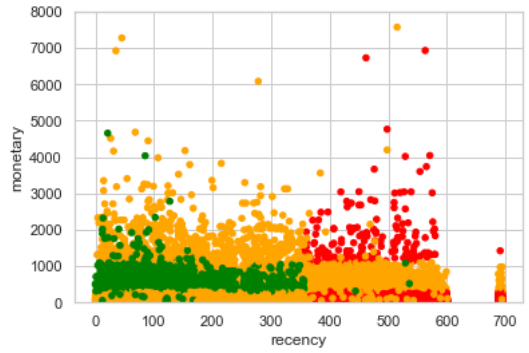
```
<AxesSubplot:xlabel='frequency', ylabel='monetary'>
```



R vs M - Approach 1 (Sum): Recency vs Monetary

```
df_rfm.plot(x='recency', y='monetary', kind='scatter', color=df_rfm.Segment.map({'Low':'red', 'Mid':'orange', 'High':'gr
```

```
<AxesSubplot:xlabel='recency', ylabel='monetary'>
```

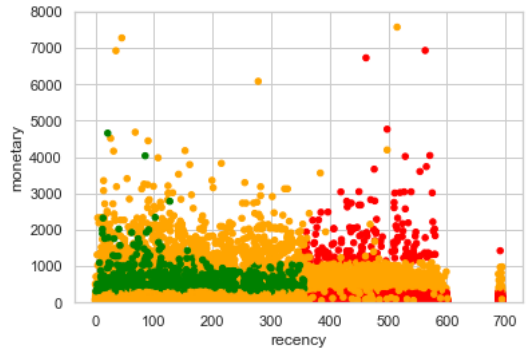


Note: More of recent purchasers are marked as 'High', with some segment in orange marked 'Mid', followed by high with those having high monetary value (green in top left). Makes sense.

R vs M - Approach 2 (Product): Recency vs Monetary

```
df_rfm.plot(x='recency', y='monetary', kind='scatter', color=df_rfm.Segment2.map({'Low':'red', 'Mid':'orange', 'High':'g
```

```
<AxesSubplot:xlabel='recency', ylabel='monetary'>
```

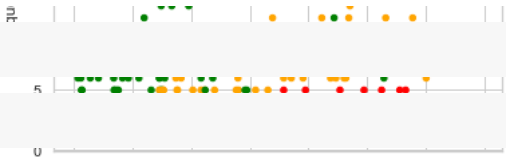


Note: The green patch in top left are marked as orange in this case. Again a business decision.

Frequency vs Recency

```
df_rfm.plot(x='recency', y='frequency', kind='scatter', color=df_rfm.Segment.map({'Low':'red', 'Mid':'orange', 'High':'g
```

```
<AxesSubplot:xlabel='recency', ylabel='frequency'>
```



Inference

Low and medium are more concentrated. The high value (green) are spread out, indicating the high value customers can be across the board.

20. Formula based CLTV

```
df_full = pd.read_csv('Intermediate/df_full.csv', index_col=False)
```

```
df_full.head(25)
```

	order_id	payment_sequential	payment_type	payment_installments	payment_value	order_item_id	
0	b81ef226f3fe1789b1e8b2acac839d17	1	Credit Card	8	99.33	1	af74c
1	a9810da82917af2d9aefd1278f1dcfa0	1	Credit Card	1	24.39	1	a630cc
2	25e8ea4e93396b6fa0d3dd708e76c1bd	1	Credit Card	1	65.71	1	2028bf
3	ba78997921bbcdc1373bb41e913ab953	1	Credit Card	8	107.78	1	548e5l
4	42fdf880ba16b47b59251dd489d4441a	1	Credit Card	2	128.45	1	386486
5	298fcdf1f73eb413e4d26d01b25bc1cd	1	Credit Card	2	96.12	1	eff409
6	298fcdf1f73eb413e4d26d01b25bc1cd	1	Credit Card	2	96.12	2	eff409

CLTV = ((Average Order Value x Purchase Frequency)/Churn Rate) x Profit margin

We don't have profit margin information, so we'll calculate CLTV with respect to Revenue.

CLTV = ((Average Order Value x Purchase Frequency)/Churn Rate)

where,

- Average Order Value = Total Revenue / Total Number of Orders
- Purchase Frequency = Total Number of Orders / Total Number of Customers
- Churn Rate: Percentage of customers who have not ordered again.
- Lifespan = 1 / Churn Rate
- Repeat Rate = 1 - Churn Rate

```
# Df_Full is at item level. Bring it to order level for calc purpose
df_orderlevel = df_full.groupby(['customer_unique_id', 'order_id']).agg({
    'order_date': np.max,
    'payment_value':np.sum,
})

# make index as columns
df_orderlevel.reset_index(inplace=True)

# convert to datetime
df_orderlevel['order_date'] = pd.to_datetime(df_orderlevel['order_date'])

# view
df_orderlevel.head(7)
```

	customer_unique_id	order_id	order_date	payment_value
0	0000366f3b9a7992bf8c76cfdf3221e2	e22acc9c116caa3f2b7121bbb380d08e	2018-05-10 10:56:27	141.90
1	0000b849f77a49e4a4ce2b2a4ca5be3f	3594e05a005ac4d06a72673270ef9ec9	2018-05-07 11:11:27	27.19
2	0000f46a3911fa3c0805444483337064	b33ec3b699337181488304f362a6b734	2017-03-10 21:05:03	86.22
3	0000f6ccb0745a6a4b88665a16c9f078	41272756ecddd9a9ed0180413cc22fb6	2017-10-12 20:29:41	43.62
4	0004aac84e0df4da2b147fca70cf8255	d957021f1127559cd947b62533f484f7	2017-11-14 19:45:42	196.89
5	0004bd2a26a76fe21f786e4fbd80607f	3e470077b690ea3e3d501cffb5e0c499	2018-04-05 19:33:16	166.98
6	00050ab1314c0e55a6ca13cf7181fecf	d0028facea13f508e880202d7097a5a1	2018-04-20 12:57:23	35.38

Aggregate to customer level and calculate lifespan and order frequency.

```
# 1. Bring it to customer level
# 2. Calculate lifespan and order frequency
df_custlevel = df_orderlevel.groupby('customer_unique_id').agg({
    'order_date': lambda date: (date.max() - date.min()).days,  # lifespan
    'order_id': lambda order: len(order),  # order freq
    'payment_value': lambda pay: np.sum(pay),  # customer value
})

df_custlevel.reset_index(inplace=True)
```

```
df_custlevel.rename(columns={'order_date': 'lifespan',
                             'order_id': 'frequency'},
                    inplace=True)

df_custlevel.head(7)
```

	customer_unique_id	lifespan	frequency	payment_value
0	0000366f3b9a7992bf8c76cfdf3221e2	0	1	141.90
1	0000b849f77a49e4a4ce2b2a4ca5be3f	0	1	27.19
2	0000f46a3911fa3c0805444483337064	0	1	86.22
3	0000f6ccb0745a6a4b88665a16c9f078	0	1	43.62
4	0004aac84e0df4da2b147fca70cf8255	0	1	196.89
5	0004bd2a26a76fe21f786e4fbd80607f	0	1	166.98
6	00050ab1314c0e55a6ca13cf7181fecf	0	1	35.38

```
df_custlevel = df_custlevel.sort_values(by='frequency', ascending=False)
df_custlevel.head()
```

	customer_unique_id	lifespan	frequency	payment_value
51423	8d50f5eadf50201ccdcedfb9e2ac8455	427	15	879.27
22775	3e43e6105506432c953e165fb2acf44c	161	9	1963.58
10058	1b6c7548a2a1f9037c1fd3ddfed95f33	92	7	1386.54
36702	6469f99c1f9dfae7733b25662e7f1782	281	7	973.09
73908	ca77025e7201e3b30c44b472ff346268	234	7	2126.44

```
df_custlevel['frequency'].value_counts()
```

```
1     90541
2      2572
3       181
4        28
5         9
6         5
7         3
9         1
15        1
Name: frequency, dtype: int64
```

Ideally there should be a good proportion of population with repeat purchases in order to calculate CLTV. In this case, there isn't. But these are problems we face with real business settings, where data would not be perfect for predictive modeling. So, let's tackle it.

Avg Order value

```
df_custlevel['avg_order_value'] = df_custlevel['payment_value']/df_custlevel['frequency']
df_custlevel.head(7)
```

	customer_unique_id	lifespan	frequency	payment_value	avg_order_value
51423	8d50f5eadf50201ccdcedfb9e2ac8455	427	15	879.27	58.618000
22775	3e43e6105506432c953e165fb2acf44c	161	9	1963.58	218.175556
10058	1b6c7548a2a1f9037c1fd3ddfed95f33	92	7	1386.54	198.077143
36702	6469f99c1f9dfae7733b25662e7f1782	281	7	973.09	139.012857
73908	ca77025e7201e3b30c44b472ff346268	234	7	2126.44	303.777143
87868	f0e310a6839dce9de1638e0fe5ab282a	320	6	540.69	90.115000
6964	12f5d6e1cbf93dafd9dcc19095df0b3d	0	6	110.72	18.453333

Overall Puchase frequency

```
purchase_frequency = sum(df_custlevel['frequency'])/df_custlevel.shape[0]
purchase_frequency
```

```
1.0334151123300586
```

Repeat Rate and Churn Rate

```
repeat_rate = (df_custlevel['frequency'] > 1).sum()/df_custlevel.shape[0]
churn_rate = 1 - repeat_rate

print(purchase_frequency, repeat_rate, churn_rate)
```

```
1.0334151123300586 0.029997535916692557 0.9700024640833075
```

We have a poor repeat rate!

Calc CLTV

```
df_custlevel['CLTV'] = ((df_custlevel['avg_order_value'] * purchase_frequency)/churn_rate)
```

```
df_custlevel.sample(10)
```

	customer_unique_id	lifespan	frequency	payment_value	avg_order_value	CLTV
39615	6c4f22a436757085432e2615eef0bbc8	0	1	81.26	81.26	86.572267
201	008778a1591d49f6b2464dffaa4fa662	0	1	260.55	260.55	277.583117
62149	aa3e4c63f6a40214fb38928da5b8e515	0	1	245.57	245.57	261.623819
859	02578a30eaf25ecf087a3728dc04b85a	0	1	54.09	54.09	57.626063
13137	23d83aae8643744884d8e64678220754	0	1	218.52	218.52	232.805461
75048	cd94dd7a37a6160bf004a86f9d36765d	0	1	702.08	702.08	747.977566
69640	beaaf6f125d53e5acc21c2e14fd878f2	0	1	55.24	55.24	58.851243
38368	68fff92320db6e5e17479bae49c95498	0	1	337.52	337.52	359.584931
19144	344f60e4ffb6c317ec1d6794d44ca0f9	0	1	79.98	79.98	85.208588
23089	3f1577ae405a1a1aad963e093b05ac6d	0	1	1006.72	1006.72	1072.533009

Total CLTV for all customers

```
df_custlevel.CLTV.sum().round(2)
```

```
20345264.84
```

21. Framing CLTV as a Regression Modeling Problem

Once of of solving for CLTV is through regression modeling approach where we try to predict the total purchase value of the customers in a given time frame.

```
df.head()
```

	order_purchase_timestamp	customer_unique_id	payment_value
0	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e5bff	18.12
1	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e5bff	2.00
2	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e5bff	18.59
3	2018-07-24 20:41:37	af07308b275d755c9edb36a90c618231	141.46
4	2018-08-08 08:38:49	3a653a41f6f9fc3d2a113cf8398680e8	179.12

```
# Put in datetime format
df['order_purchase_timestamp'] = pd.to_datetime(df.order_purchase_timestamp)
```

Create Month-Year column

```
df['month_yr'] = df['order_purchase_timestamp'].apply(lambda x: x.strftime('%Y-%m'))
df.head()
```

	order_purchase_timestamp	customer_unique_id	payment_value	month_yr
0	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e5bff	18.12	2017-10
1	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e5bff	2.00	2017-10
2	2017-10-02 10:56:33	7c396fd4830fd04220f754e42b4e5bff	18.59	2017-10
3	2018-07-24 20:41:37	af07308b275d755c9edb36a90c618231	141.46	2018-07
4	2018-08-08 08:38:49	3a653a41f6f9fc3d2a113cf8398680e8	179.12	2018-08

Examine the repeat purchases

```
df.customer_unique_id.value_counts()
```

```
f9ae226291893fda10af7965268fb7f6    23
569aa12b73b5f7edeaa6f2a01603e381    22
2524dcec233c3766f2c2b22f69fd65f4    19
24f12460aad399ba18f4ed2c2fbab65d    17
6fbc7cdadbb522125f4b27ae9dee4060    17
                                    ..
d4f553d430d80edc3fd954edebae33db     1
2912db6f8894dfa5aeea61325e2d5f08     1
f75d280e67adbfb5959a3782fa91e2fd     1
98e4ee1f6e5b08ad5ab5fbfa99206b74     1
7c020cb518b4fdcf5c50fa717d0fdca8     1
Name: customer_unique_id, Length: 93357, dtype: int64
```

```
np.sum(df.customer_unique_id.value_counts() > 1)/len(df.customer_unique_id)
```

```
0.054155806545033304
```

Observe: Only a small subset (5.4%) of customers have more than 1 transaction.

Framing the problem

Let's make the features table, with amount of sales every month as features and the total sale value for each customer as the CLTV (y)

We will use the latest 6 months of the sales data as independent variables (predictors) and use that to predict the CLTV.

```
df_sales = df.pivot_table(index=['customer_unique_id'],columns=['month_yr'],values='payment_value', aggfunc='sum', fill_
```

```
df_sales.head(20)
```

month_yr	customer_unique_id	2016-10	2016-12	2017-01	2017-02	2017-03	2017-04	2017-05	2017-06	2017-07	2017-08	2017-09	2017-10	201

```
df_sales['CLV']=df_sales.iloc[:,2:].sum(axis=1)
df_sales.head()
```

month_yr	customer_unique_id	2016-10	2016-12	2017-01	2017-02	2017-03	2017-04	2017-05	2017-06	2017-07	2017-08	2017-09	2017-10	2017-11
0	0000366f3b9a7992bf8c76cfdf3221e2	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00
1	0000b849f77a49e4a4ce2b2a4ca5be3f	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00
2	0000f46a3911fa3c0805444483337064	0.0	0.0	0.0	0.0	86.22	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00
3	0000f6ccb0745a6a4b88665a16c9f078	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	0.0	0.0	0.0	43.62	0.00
4	0004aac84e0df4da2b147fca70cf8255	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	0.0	0.0	0.0	0.00	196.89
10	0006fdc98a402fceb4eb0ee528f6a8d4	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	29.0	0.00	0.00	0.00	0

Create X and y

```
X = df_sales[['2018-08','2018-07', '2018-06','2018-05','2018-04','2018-03']]
y = df_sales[['CLV']]
```

14	000a5ad9c4601d2bbdd9ed765d5213b3	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	0.0	91.28	0.00	0.00	0

22. Regression modeling

```
# split training set and test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
# import model
from sklearn.linear_model import LinearRegression

# instantiate
linreg = LinearRegression()

# fit the model to the training data (learn the coefficients)
linreg.fit(X_train, y_train)

# make predictions on the testing set
y_pred = linreg.predict(X_test)
```

```
# print the intercept and coefficients
print(linreg.intercept_)
print(linreg.coef_)
```

```
[111.34399466]
[[0.76377533 0.789606   0.77480437 0.7885248  0.77029104 0.71875057]]
```

Validate the model

```
# Compute R-Squared
from sklearn import metrics
print("R-Square:", metrics.r2_score(y_test, y_pred))
```

```
R-Square: 0.32580780073430826
```

That's quite low R-Squared.

What do we do now?

Mostly because most of the customers have only 1 transaction. So, let's rebuild the model with:

1. Only those customers with atleast 2 transactions
2. Only those customers with atleast 5 transactions
3. At least 1 transaction in the last 6 months
4. Comibination of 2 and 3

23. Modeling approach 2

Only include customers with at least 2 transactions to train the linear regression model.

```
# customers with at least 2 transactions
elig_customers = df.customer_unique_id.value_counts().index[df.customer_unique_id.value_counts() > 1]
df_app1 = df.loc[df.customer_unique_id.isin(elig_customers), :]
```

```
df_app1.shape
```

```
(12203, 4)
```

```
# Approach 1
df_sales = df_app1.pivot_table(index=['customer_unique_id'],columns=['month_yr'],values='payment_value', aggfunc='sum',
df_sales['CLV'] = df_sales.iloc[:,2:].sum(axis=1)
df_sales.head()
```

month_yr	customer_unique_id	2016-10	2016-12	2017-01	2017-02	2017-03	2017-04	2017-05	2017-06	2017-07	2017-08	2017-09	2017-10	201
0	000e309254ab1fc5ba99dd469d36bdb4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.0	0.0	0.0	0
1	0028a7d8db7b0247652509358ad8d755	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.0	0.0	0.0	0
2	00324c9f4d710e7bac5c5ba679714430	0.0	0.0	0.0	0.0	0.0	0.0	0.0	109.78	0.00	0.0	0.0	0.0	0
3	004288347e5e88a27ded2bb23747066c	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00	251.09	0.0	0.0	0.0	0
4	0058f300f57d7b93c477a131a59b36c3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00	0.00	0.0	0.0	0.0	0

```
X = df_sales[['2018-08','2018-07', '2018-06','2018-05','2018-04','2018-03']]
y = df_sales[['CLV']]
```

```
# split training set and test set
from sklearn.model_selection import train_test_split
X_train2, X_test2, y_train2, y_test2 = train_test_split(X, y, random_state=0)

# import model
from sklearn.linear_model import LinearRegression

# instantiate
linreg = LinearRegression()

# fit the model to the training data (learn the coefficients)
linreg.fit(X_train2, y_train2)

# make predictions on the testing set
y_pred2 = linreg.predict(X_test2)

# print the intercept and coefficients
print(linreg.intercept_)
print(linreg.coef_)

# Compute R-Squared
from sklearn import metrics
print("R-Square:",metrics.r2_score(y_test2, y_pred2))
```

```
[175.46957211]
[[0.85186484 0.73634488 0.78124035 0.75630072 0.73772576 0.68077824]]
R-Square: 0.22311422956505322
```

The R2 score has gone from bad to worse. Perhaps having more zeros on both training and test was helping to increase the R2.

24. Regression modeling - approach 3

Experiment 3: Perhaps 6 months of data was not sufficient, esp since limited number of repeat transactions. Add 12 months of data to predict the Y instead of 6.

```
X = df_sales[['2018-08', '2018-07', '2018-06', '2018-05', '2018-04', '2018-03',
              '2018-02', '2018-01', '2017-12', '2017-11', '2017-10', '2017-09']]

y = df_sales[['CLV']]
```

```
# split training set and test set
from sklearn.model_selection import train_test_split
X_train3, X_test3, y_train3, y_test3 = train_test_split(X, y, random_state=0)

# import model
from sklearn.linear_model import LinearRegression

# instantiate
linreg = LinearRegression()

# fit the model to the training data (learn the coefficients)
linreg.fit(X_train3, y_train3)

# make predictions on the testing set
y_pred3 = linreg.predict(X_test3)

# print the intercept and coefficients
print(linreg.intercept_)
print(linreg.coef_)

# Compute R-Squared
from sklearn import metrics
print("R-Square:",metrics.r2_score(y_test3, y_pred3))
```

```
[85.58011594]
[[0.92921604 0.87450579 0.88361834 0.87309606 0.83737053 0.82110116
  0.85987345 0.87613918 0.85570371 0.86577022 0.92098443 0.9308609 ]]
R-Square: 0.4180680746320975
```

The R-Squared has improved. All coefficients are remain positive, which is good.

```
pd.DataFrame(y_pred).describe()
```

	0
count	23340.000000

Double-click (or enter) to edit

50%	111.343995

25. Regression modeling Approach 4

Experiment 4: Model for customers with at least 5 transactions only

```
# customers with at least 2 transactions
elig_customers = df.customer_unique_id.value_counts().index[df.customer_unique_id.value_counts() >= 5]
df_app2 = df.loc[df.customer_unique_id.isin(elig_customers), :]
df_app2.shape
```

```
(830, 4)
```

```
# Approach 2
df_sales = df_app2.pivot_table(index=['customer_unique_id'],
                               columns=['month_yr'],
                               values='payment_value',
                               aggfunc='sum',
                               fill_value=0).reset_index()
df_sales['CLV'] = df_sales.iloc[:,2:].sum(axis=1)
df_sales.head()
```

month_yr	customer_unique_id	2016-10	2017-01	2017-02	2017-03	2017-04	2017-05	2017-06	2017-07	2017-08	2017-09	2017-10	2017-11	2017-12
0	00face5c8f7dbb7eefd4112722f26903	0.0	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	024676cef113f6c81db6c5b8d29e5ee9	0.0	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	08d8fd00898e7cc99d6e32c74d7d3ce5	0.0	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	096e351116755fe4c3c1e48eaf301a41	0.0	0.0	0.0	0.0	0.0	217.49	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	09f55ef15bde8baeef87b3348b31c150	0.0	0.0	0.0	0.0	0.0	0.00	0.0	0.0	0.0	43.1	0.0	0.0	0.0

```
X = df_sales[['2018-08', '2018-07', '2018-06', '2018-05', '2018-04', '2018-03',
              '2018-02', '2018-01', '2017-12', '2017-11', '2017-10', '2017-09']]

y = df_sales[['CLV']]
```

```
# split training set and test set
from sklearn.model_selection import train_test_split
X_train4, X_test4, y_train4, y_test4 = train_test_split(X, y, random_state=0)

# import model
from sklearn.linear_model import LinearRegression

# instantiate
linreg = LinearRegression()

# fit the model to the training data (learn the coefficients)
linreg.fit(X_train4, y_train4)

# make predictions on the testing set
y_pred4 = linreg.predict(X_test4)

# print the intercept and coefficients
print(linreg.intercept_)
print(linreg.coef_)

# Compute R-Squared
from sklearn import metrics
print("R-Square:",metrics.r2_score(y_test4, y_pred4))
```

```
[105.46805379]
[[1.48430238 0.73859531 0.95955687 1.25035994 0.91961742 0.98489107
  0.97492126 0.37101888 0.80335822 1.17878815 0.81532839 1.34443922]]
R-Square: 0.7805693859232075
```

The R-Square seems to have improved significantly compared to the previous approaches.

BUT, in this approach, we have far lesser datapoints in the training data.

So, for the sake of right comparison, **let's try to use this model to predict all the data from approach 3.**

Because that will give the right picture.

```
y_pred43 = linreg.predict(X_test3)

# print the intercept and coefficients
print(linreg.intercept_)
print(linreg.coef_)

# Compute R-Squared
from sklearn import metrics
print("R-Square:",metrics.r2_score(y_test3, y_pred43))
```

```
[105.46805379]
[[1.48430238 0.73859531 0.95955687 1.25035994 0.91961742 0.98489107
  0.97492126 0.37101888 0.80335822 1.17878815 0.81532839 1.34443922]]
R-Square: 0.3539240435142269
```

The R-Square hasn't really improved much, really.

But, this model is still useful to predict CLTV for customers who have 5 or more transactions.

Conclusion

So, one approach we could follow in practice, for those customers with < 5 transactions we can use model from approach 3. For >5 transactions, use model from approach 4.

IF we are to go about using Linear regression model as the final model, we will check for the multi collinearity using VIF, normality of errors, etc.

We won't digress too much by getting into classical regression models. To know more about the please revisit the linear regression lectures.

Note: Alternately we can build a model for each of the *customer segments* we defined earlier. Ideal if we have good proportion of repeat purchases.

26. RFM Definition for Model Building - Lifetimes

We can see that our unique users are 93357, it is to each of them that we must calculate new variables.

This is where an important point about the **Lifetimes** library comes in, and that is that to make use of its models it is necessary to present 4 specific variables: **recency**, **frequency**, **T** (age) and **monetary value**.

Do not worry, I will proceed to explain what each one of them is about:

- **Frequency:** Represents the number of repeat purchases the customer has made. This means that it's one less than the total number of purchases.
- **Recency:** Represents the age of the customer when they made their most recent purchases. This is equal to the duration between a customer's first purchase and their latest purchase. (Thus if they have made only 1 purchase, the recency is 0.)
- **T:** Represents the age of the customer in whatever time units chosen. This is equal to the duration between a customer's first purchase and the end of the period under study.
- **Monetary value:** Represents the average value of a given customer's purchases. This is equal to the sum of all a customer's purchases divided by the total number of purchases. Note that the denominator here is different than the frequency described above.

Now that we know which variables we are going to use, if we see our created dataset again we have everything to calculate them, that will be our next step.

Feature Engineering

Let's start by formatting our date variable and exploring its minimum (first purchase) and maximum (last purchase) values.

```
df['order_purchase_timestamp'] = pd.to_datetime(df['order_purchase_timestamp'], format="%Y-%m-%d %H:%M:%S")
df['order_purchase_timestamp'] = df.order_purchase_timestamp.dt.date
df['order_purchase_timestamp'] = pd.to_datetime(df['order_purchase_timestamp'])
df['order_purchase_timestamp'].describe()
```

```
count                  100137
unique                    611
top       2017-11-24 00:00:00
freq                     1182
first     2016-10-03 00:00:00
last      2018-08-29 00:00:00
Name: order_purchase_timestamp, dtype: object
```

```
# The last purchase made in the dataset was on August 29, 2018, so we will use that date as our current
# date to simulate an immediate study of the company's transactions.

date_today = df['order_purchase_timestamp'].max()
date_today
```

```
Timestamp('2018-08-29 00:00:00')
```

Recency

Represents the age of the customer today. It is calculated as the time since they made their most recent purchases.

We will group by user in a new table.

```
r = df.groupby('customer_unique_id').agg(['min', 'max'])['order_purchase_timestamp']
r['recency'] = r['max'] - r['min']
r
```

customer_unique_id	min	max	recency
0000366f3b9a7992bf8c76cfdf3221e2	2018-05-10	2018-05-10	0 days
0000b849f77a49e4a4ce2b2a4ca5be3f	2018-05-07	2018-05-07	0 days
0000f46a3911fa3c0805444483337064	2017-03-10	2017-03-10	0 days
0000f6ccb0745a6a4b88665a16c9f078	2017-10-12	2017-10-12	0 days
0004aac84e0df4da2b147fca70cf8255	2017-11-14	2017-11-14	0 days
...	...	...	...
fffcf5a5ff07b0908bd4e2dbc735a684	2017-06-08	2017-06-08	0 days
fffea47cd6d3cc0a88bd621562a9d061	2017-12-10	2017-12-10	0 days
ffff371b4d645b6ecea244b27531430a	2017-02-07	2017-02-07	0 days
ffff5962728ec6157033ef9805bacc48	2018-05-02	2018-05-02	0 days
ffffd2657e2aad2907e67c3e9daecbeb	2017-05-02	2017-05-02	0 days

93357 rows × 3 columns

T

Represents the age of the customer in whatever time units chosen. This is equal to the duration between a customer's first purchase and the end of the period under study.

This same table helps us to create our variable T.

```
r['T'] = date_today - r['min']
r = r[['recency', 'T']]

# Let's take a look at our new variables
r
```

customer_unique_id	recency	T
0000366f3b9a7992bf8c76cfdf3221e2	0 days	111 days
0000b849f77a49e4a4ce2b2a4ca5be3f	0 days	114 days
0000f46a3911fa3c0805444483337064	0 days	537 days
0000f6ccb0745a6a4b88665a16c9f078	0 days	321 days
0004aac84e0df4da2b147fca70cf8255	0 days	288 days
...	...	...
fffcf5a5ff07b0908bd4e2dbc735a684	0 days	447 days
fffea47cd6d3cc0a88bd621562a9d061	0 days	262 days
ffff371b4d645b6ecea244b27531430a	0 days	568 days
ffff5962728ec6157033ef9805bacc48	0 days	119 days
ffffd2657e2aad2907e67c3e9daecbeb	0 days	484 days

Frequency

Represents the number of 'repeat' purchases the customer has made, as per the lifetimes package.

As before we will group by unique user to create this variable.

```
aggregations = {
    'order_purchase_timestamp':'count',
    'payment_value': 'sum'}

f = df.groupby('customer_unique_id').agg(aggregations)

f['frequency'] = f['order_purchase_timestamp'] - 1  # per lifetimes definition

f = f[['frequency']]
```

```
# Let's create our final table joining both
rf = pd.merge(r,f, left_index=True, right_index=True)
rf
```

customer_unique_id	recency	T	frequency
0000366f3b9a7992bf8c76cfdf3221e2	0 days	111 days	0
0000b849f77a49e4a4ce2b2a4ca5be3f	0 days	114 days	0
0000f46a3911fa3c0805444483337064	0 days	537 days	0
0000f6ccb0745a6a4b88665a16c9f078	0 days	321 days	0
0004aac84e0df4da2b147fca70cf8255	0 days	288 days	0
...	...	...	...
fffcf5a5ff07b0908bd4e2dbc735a684	0 days	447 days	0
fffea47cd6d3cc0a88bd621562a9d061	0 days	262 days	0
ffff371b4d645b6ecea244b27531430a	0 days	568 days	0
ffff5962728ec6157033ef9805bacc48	0 days	119 days	0
ffffd2657e2aad2907e67c3e9daecbeb	0 days	484 days	0

93357 rows × 3 columns

Monetary Value

For our convenience, the *Lifetimes* library includes a module that takes care of this cumbersome process automatically. We will work with this table created by the library, since it includes the variable **monetary_value** and allows us to define the period of time with which we want to work (in our case we will choose weekly since our dataset belongs to different industries, this is a regular period of time).

In fact the `summary_data_from_transaction_data` function will calculate all of R,F,M in one shot. Let's verify.

```
from lifetimes.utils import summary_data_from_transaction_data

rfm = summary_data_from_transaction_data(df, customer_id_col='customer_unique_id', datetime_col='order_purchase_timestam
                                         monetary_value_col ='payment_value', observation_period_end='2018-08-29',
                                         datetime_format='%Y-%m-%d', freq='D')
rfm
```

customer_unique_id	frequency	recency	T	monetary_value
0000366f3b9a7992bf8c76cfdf3221e2	0.0	0.0	111.0	0.0
0000b849f77a49e4a4ce2b2a4ca5be3f	0.0	0.0	114.0	0.0
0000f46a3911fa3c0805444483337064	0.0	0.0	537.0	0.0
0000f6ccb0745a6a4b88665a16c9f078	0.0	0.0	321.0	0.0
0004aac84e0df4da2b147fca70cf8255	0.0	0.0	288.0	0.0
...	...	...	...	...
fffcf5a5ff07b0908bd4e2dbc735a684	0.0	0.0	447.0	0.0
fffea47cd6d3cc0a88bd621562a9d061	0.0	0.0	262.0	0.0
ffff371b4d645b6ecea244b27531430a	0.0	0.0	568.0	0.0
ffff5962728ec6157033ef9805bacc48	0.0	0.0	119.0	0.0
ffffd2657e2aad2907e67c3e9daecbeb	0.0	0.0	484.0	0.0

93357 rows × 4 columns

Visualizing our customers

```
# Frequency
fig = px.histogram(rfm,
                   x=rfm['frequency'],
                   title='Frequency of purchase',
                   labels={'frequency':'Frequency'},
                   opacity=0.8, marginal='violin',
                   color_discrete_sequence=['indianred'])

py.iplot(fig)
```

Here we find an important point which can influence our model, the vast majority of buyers only made one transaction in their entire lifetime in the company (remember that frequency is calculated by subtracting the number of purchases minus 1).

```
# Recency
px.histogram(rfm, x=rfm['recency'],title='Recency of purchase',
                   labels={'recency':'Recency'}, nbins=50,
                   opacity=0.8, marginal='violin',
                   color_discrete_sequence=['indianred'])
```

As with the previous variable, most users have zero recency, having made only a single purchase in their time as a customer.

```
px.histogram(rfm, x=rfm['T'],title='Time from first purchase',
                   labels={'T':'Weeks'},
                   opacity=0.8, marginal='violin',
                   color_discrete_sequence=['indianred'])
```

In the case of the customer's lifetime in his last purchase, we can observe a distribution more similar to a Gaussian distribution, with 40 weeks as the mode. We can also observe a greater concentration in the left area, corroborating that most of them are sporadic purchases. Seeing this, let's move on to our first model.

27. CLTV Modelling

BetaGeoFitter Intuition and Assumptions

BG/NBD is a probabilistic approach to modeling the CLTV of a customer and helps us to understand:

1. Which customers are alive?
2. Who will place orders in next n days?
3. The number of orders expected from given customer in given period.

BG/NBD stands for Beta Geometric - Negative Binomial Distribution.

Intuition

Customers have different buying patterns.

Say you start working as manager of an icecream store and you are interested to know which customers are going to buy from you in the current year, given you have the sales / orders information from last few years.

By looking at their records, you know: certain customers want to eat icecream all through the year, some only in summer months and some only on special occasions.

So, instead of a static (overall) transanction rate for all customers, you want the distribution for the transaction rate.

Likewise, you also want to know how the 'Churn' is distributed. Because not all customers are qually committed.

So in order to model the future purchases the transaction rates and churn has to be accurately modeled.

To identify the valuable customers, we need to know:

1. Transaction rate (lamda)

2. Churn rate

The most popular probablistic approaches to model this are:

1. **The Pareto / NBD approach**
2. **The BG / NBD approach**
3. **Gamma - Gamma model.**

Lifetimes package implements the last 2 effectively.

The approaches makes certain assumptions about the customer activity that helps us predict the future behaviour.

The assumptions itself is listed below. However, the main difference is as follows:

In Pareto/NBD modeling, it assmumes there is a small change of attrition as we move forward with time. So, the longer we wait without a transaction the higher the chance of attrition, which may not always be the case all the time esp for subscription business.

Whereas, in BG/NBD approach, the customer considers the various factors such as price changes, product changes, quality etc since the last visit which will in turn decide if he will leave after every purchase.

So, The customer has a probability p of attriting after each purchase.

Ref

Also known as BG / NBD, this model is intended to estimate transactions in a future period of time for each user, in addition to the probability that this is "alive".

This is an alternative to the well-known Pareto / NBD model which uses the Bayesian probability in a hierarchical way to make its estimates.

References:

1. Fader's Paper
2. Stackexchange discussion
3. Probability of Alive - Paper

Pareto / NBD Assumptions

i. While active, the number of transactions made by a customer in a time period of length t is distributed Poisson with transaction rate λ.

ii. Heterogeneity in transaction rates across customers follows a gamma distribution with shape parameter r and scale parameter α. (Not all customers have same shopping

iii. Each customer has an unobserved "lifetime" of length τ. This point at which the customer becomes inactive is distributed exponential with dropout rate μ.

iv) Heterogeneity in dropout rates across customers follows a gamma distribution with shape parameter s and scale parameter β.

v. The transaction rate λ and the dropout rate μ vary independently across customers."

BG/NBD Assumptions

i.) While active, the number of transactions made by a customer follows a Poisson process with transaction rate λ. This is equivalent to assuming that the time between transactions is distributed exponential with transaction rate λ

ii) Heterogeneity in **λ follows a gamma distribution**. (Shape param: r and scale param: alpha)

iii) After any transaction, a **customer becomes inactive with probability p** (churn rate). Therefore the point at which the customer "drops out" is distributed across transactions according to a (shifted) geometric distribution with pmf

iv) Heterogeneity in **p follows a beta distribution** (parameters: a and b)

Without delving further into theory, let's move on to its application in our data set.

Training BetaGeoFitter model

```
from lifetimes import BetaGeoFitter

# penalizer_coef is a parameter that penalizes the likelihood, usually values like 0.001 or
# 0.01 with small samples to prevent parameters from becoming too large

bgf = BetaGeoFitter(penalizer_coef=0.001)
bgf.fit(rfm['frequency'], rfm['recency'], rfm['T'], verbose=True)
print(bgf)
```

```
Optimization terminated successfully.
         Current function value: 0.082123
         Iterations: 45
         Function evaluations: 46
         Gradient evaluations: 46
<lifetimes.BetaGeoFitter: fitted with 93357 subjects, a: 0.35, alpha: 74.11, b: 0.06, r: 0.02>
```

```
bgf.summary
```

	coef	se(coef)	lower 95% bound	upper 95% bound
r	0.016700	0.000943	0.014851	0.018548
alpha	74.110214	7.335495	59.732643	88.487784
a	0.351189	0.049377	0.254411	0.447968
b	0.061593	0.010105	0.041788	0.081399

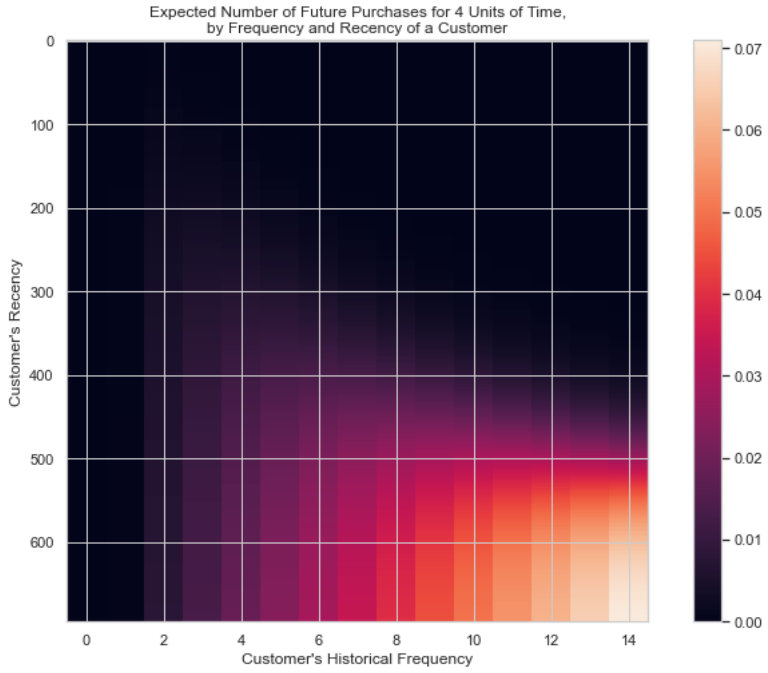
We have already trained our model, we can make greater use of the library with the following visualizations.

Frequency recency matrix: This matrix shows us the probability of future customer purchases in a given time, using recency and frequency as estimators.

Probability alive matrix: Like the previous one, using recency and frequency as indicators, this matrix indicates the probability that a client is "alive" at the current moment.

```
from lifetimes.plotting import plot_frequency_recency_matrix

# T = Unit times(weeks)
fig = plt.figure(figsize=(12,8))
plot_frequency_recency_matrix(bgf, T=4)
```

```
<AxesSubplot:title={'center':'Expected Number of Future Purchases for 4 Units of Time,\nby Frequency and Recency of a Customer'},
xlabel="Customer's Historical Frequency", ylabel="Customer's Recency">
```

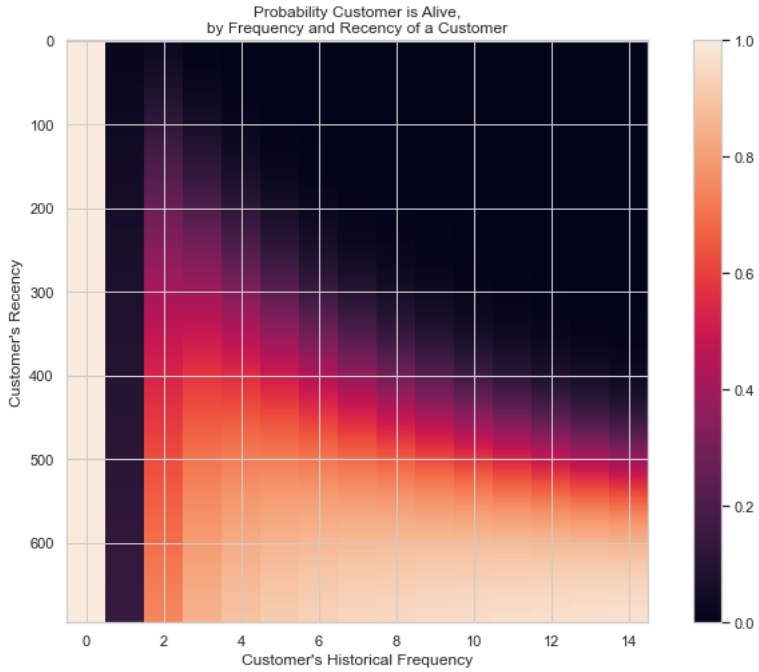


Bottom right: The expected number of future purchases is highest here (high frequency, more recently purchased)

Moving away from this region, the expected purchases decrease.

```
from lifetimes.plotting import plot_probability_alive_matrix

fig = plt.figure(figsize=(12,8))
plot_probability_alive_matrix(bgf)
```

```
<AxesSubplot:title={'center':'Probability Customer is Alive,\nby Frequency and Recency of a Customer'}, xlabel="Customer's
Historical Frequency", ylabel="Customer's Recency">
```



Bottom Right: More frequent and more recent purchases has highest chance of being alive.

As we can see, both probabilities increase the greater the number of purchases and the time the customer had on his last purchase. It should be noted that the probabilities of future purchases are quite low in the next four weeks and are restricted to a few customer quadrants, this can be easily explained by the trend in sporadic purchases in our dataset.

Expected number of Purchases per customer

Based on our model we can create predictions for the period of time we choose, in our case we will do it for the first 4, 8 and 12 weeks, these being a strong short-term indicator to increase our advertising efforts in users with the greatest possibility of invest in our company.

```
t = 4 # Days for a future transaction
rfm['expected_4week'] = round(bgf.conditional_expected_number_of_purchases_up_to_time(t, rfm['frequency'], rfm['recency'
rfm.sort_values(by='expected_4week', ascending=False)
```

customer_unique_id	frequency	recency	T	monetary_value	expected_4week
8d50f5eadf50201ccdcedfb9e2ac8455	14.0	428.0	437.0	59.108571	0.11
ca77025e7201e3b30c44b472ff346268	6.0	235.0	324.0	138.941667	0.05
dc813062e0fc23409cd255f7f53c7074	5.0	418.0	424.0	151.606000	0.04
394ac4de8f3acb14253c177f0e15bc58	4.0	236.0	250.0	149.282500	0.04
e0836a97eaae86ac4adc26fbb334a527	2.0	62.0	64.0	32.955000	0.04
...	...	...	...	...	...
5531d719e72f20e61f0cf4a4ffdc8f00	0.0	0.0	161.0	0.000000	0.00
5531c5cc7089dcf2a05e8b70fe7092f3	0.0	0.0	111.0	0.000000	0.00
55310366b1116f6cf157b304213b24e7	0.0	0.0	107.0	0.000000	0.00
5530606136ee55b00f55cfefd5033bfb	0.0	0.0	161.0	0.000000	0.00
ffffd2657e2aad2907e67c3e9daecbeb	0.0	0.0	484.0	0.000000	0.00

93357 rows × 5 columns

```
rfm['expected_8week'] = round(bgf.predict(8, rfm['frequency'], rfm['recency'], rfm['T']), 2)
rfm['expected_12week'] = round(bgf.predict(12, rfm['frequency'], rfm['recency'], rfm['T']), 2)
```

In turn, we can have more molecular analyzes, by seeing the individual probabilities for each client:

```
rfm.sort_values(by='expected_4week', ascending=False)
```

customer_unique_id	frequency	recency	T	monetary_value	expected_4week	expected_8week	expected_12week
8d50f5eadf50201ccdcedfb9e2ac8455	14.0	428.0	437.0	59.108571	0.11	0.21	0.32
ca77025e7201e3b30c44b472ff346268	6.0	235.0	324.0	138.941667	0.05	0.09	0.14
dc813062e0fc23409cd255f7f53c7074	5.0	418.0	424.0	151.606000	0.04	0.07	0.11
394ac4de8f3acb14253c177f0e15bc58	4.0	236.0	250.0	149.282500	0.04	0.09	0.13
e0836a97eaae86ac4adc26fbb334a527	2.0	62.0	64.0	32.955000	0.04	0.09	0.13
...	...	...	...	...	...	...	...
5531d719e72f20e61f0cf4a4ffdc8f00	0.0	0.0	161.0	0.000000	0.00	0.00	0.00
5531c5cc7089dcf2a05e8b70fe7092f3	0.0	0.0	111.0	0.000000	0.00	0.00	0.00
55310366b1116f6cf157b304213b24e7	0.0	0.0	107.0	0.000000	0.00	0.00	0.00
5530606136ee55b00f55cfefd5033bfb	0.0	0.0	161.0	0.000000	0.00	0.00	0.00
ffffd2657e2aad2907e67c3e9daecbeb	0.0	0.0	484.0	0.000000	0.00	0.00	0.00

93357 rows × 7 columns

```
t=12

random_person = rfm.loc["394ac4de8f3acb14253c177f0e15bc58", :]

prediction = bgf.predict(t, random_person['frequency'], random_person['recency'], random_person['T'])

print("Customer '{}' has ".format(random_person.name), round(prediction, 2), "% of probability to buy an item for the ne
```

```
Customer '394ac4de8f3acb14253c177f0e15bc58' has  0.13 % of probability to buy an item for the next 12 weeks
```

Number of people with at least 5% probability of buying in next 12 weeks

```
rfm.loc[rfm.expected_12week > 0.05, :].shape[0]
```

```
33
```

Probability of Alive

```
t = 12
```

```
rfm['prob_alive'] = bgf.conditional_probability_alive(rfm['frequency'],
                                                      rfm['recency'],
                                                      rfm['T'])
```

Expected number of purchases for a given customer

```
rfm['predicted_purchases_12'] = bgf.conditional_expected_number_of_purchases_up_to_time(t,
                                                                                        rfm['frequency'],
                                                                                        rfm['recency'],
                                                                                        rfm['T'])
```

View

```
rfm.sort_values(by='expected_4week', ascending=False).head(20)
```

customer_unique_id	frequency	recency	T	monetary_value	expected_4week	expected_8week	expected_12week	prob_
8d50f5eadf50201ccdcedfb9e2ac8455	14.0	428.0	437.0	59.108571	0.11	0.21	0.32	0.9
ca77025e7201e3b30c44b472ff346268	6.0	235.0	324.0	138.941667	0.05	0.09	0.14	0.7
dc813062e0fc23409cd255f7f53c7074	5.0	418.0	424.0	151.606000	0.04	0.07	0.11	0.9
394ac4de8f3acb14253c177f0e15bc58	4.0	236.0	250.0	149.282500	0.04	0.09	0.13	0.8
e0836a97eaae86ac4adc26fbb334a527	2.0	62.0	64.0	32.955000	0.04	0.09	0.13	0.7
6469f99c1f9dfae7733b25662e7f1782	5.0	282.0	344.0	133.412000	0.04	0.08	0.12	0.8
bc3e7032668d3f411c227eec09221362	2.0	59.0	75.0	72.535000	0.04	0.08	0.11	0.7
397b44d5bb99eabf54ea9c2b41ebb905	3.0	157.0	230.0	489.963333	0.03	0.06	0.08	0.7
34b0cd95480e55c2a701293a2b9671b3	2.0	125.0	126.0	29.825000	0.03	0.06	0.09	0.7
821e75291b1ad362e614c0ea79fc95a6	2.0	90.0	127.0	179.220000	0.03	0.05	0.08	0.6
63cfc61cee11cbe306bff5857d00bfe4	5.0	382.0	475.0	135.432000	0.03	0.06	0.09	0.8
fe81bb32c243a86b2f86fbf053fe6140	4.0	242.0	311.0	381.745000	0.03	0.07	0.10	0.7
6a9e15d6fa8ce1cabf193c21aa577f64	2.0	63.0	87.0	148.725000	0.03	0.07	0.10	0.6
d4a5e9f19897de65433c9d97bf4b9f8e	2.0	76.0	119.0	188.090000	0.03	0.05	0.08	0.6
182053495bc94c2f41090ce8c41be266	2.0	130.0	140.0	75.420000	0.03	0.05	0.08	0.7
f0e310a6839dce9de1638e0fe5ab282a	5.0	320.0	466.0	59.470000	0.03	0.05	0.08	0.7
cfa69922f9968e0e6271647abda09b09	2.0	83.0	109.0	52.880000	0.03	0.06	0.09	0.6
acea6bd29b8c1e3c6a8b266a8fb4475e	3.0	247.0	364.0	191.980000	0.02	0.04	0.06	0.6
b896655e2083a1d76b7b85df8fc86e40	3.0	284.0	338.0	140.346667	0.02	0.05	0.07	0.7
f0fd988c7c2a4cba167138a8adada6ca	2.0	156.0	235.0	51.905000	0.02	0.03	0.05	0.6

28. Validating BG model

Well we have our predictions, but we cannot blindly use the results of a model, we need to validate them with reality.

When we talk about making predictions for the future, instead of doing the classic partition of the sample in Train and Test, we do it with respect to time.

We will use the **last three months** of results whose numbers we already know and compare them with what we obtain with our BG/NBD model, for this task the *Lifetimes* library has a very useful function that divides our data set according to the indicated date.

```
from lifetimes.utils import calibration_and_holdout_data

rfm_val = calibration_and_holdout_data(df,
                                       customer_id_col='customer_unique_id',
                                       datetime_col='order_purchase_timestamp',
                                       monetary_value_col ='payment_value',
                                       calibration_period_end='2018-05-29',
                                       observation_period_end='2018-08-29',
                                       datetime_format='%Y-%m-%d',
                                       freq='D')

rfm_val.head(5)
```

customer_unique_id	frequency_cal	recency_cal	T_cal	monetary_value_cal	frequency_holdout	monetary_value_holdout
0000366f3b9a7992bf8c76cfdf3221e2	0.0	0.0	3.0	0.0	0.0	0.0
0000b849f77a49e4a4ce2b2a4ca5be3f	0.0	0.0	3.0	0.0	0.0	0.0
0000f46a3911fa3c0805444483337064	0.0	0.0	64.0	0.0	0.0	0.0
0000f6ccb0745a6a4b88665a16c9f078	0.0	0.0	33.0	0.0	0.0	0.0
0004aac84e0df4da2b147fca70cf8255	0.0	0.0	28.0	0.0	0.0	0.0

Our metrics are divided into two sets of columns ending with *cal* and *holdout*.

The first being those used in training and the second being the result in the two months that we established, while duration refers to the time in weeks that was taken for the test.

Now it is time to train this new table with the same model that we configured previously.

```
# Fit
bgf_val = BetaGeoFitter(penalizer_coef=0.001)
bgf_val.fit(rfm_val['frequency_cal'], rfm_val['recency_cal'], rfm_val['T_cal'], verbose=True)
print(bgf_val)
```

```
Optimization terminated successfully.
         Current function value: 0.074343
         Iterations: 48
         Function evaluations: 49
         Gradient evaluations: 49
<lifetimes.BetaGeoFitter: fitted with 75126 subjects, a: 0.24, alpha: 17.47, b: 0.05, r: 0.02>
```

```
bgf_val.summary
```

	coef	se(coef)	lower 95% bound	upper 95% bound
r	0.021915	0.001685	0.018612	0.025217
alpha	17.473547	2.022601	13.509249	21.437846
a	0.237896	0.058901	0.122450	0.353341
b	0.052684	0.013949	0.025344	0.080025

The model has been fit calibration period.

Let's use it to predict (number of purchases) and compare with actuals.

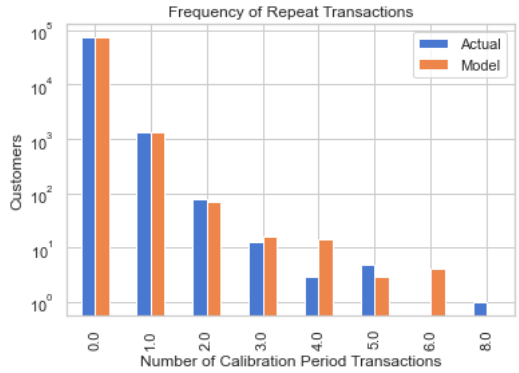
Our first validation visualization shows the comparison between the frequency of real transactions and those established by the new model.

We can see that as frequencies increase, the model performs a worse job of predicting, this may be due to the great heterogeneity of e-commerce as mentioned above or to the lack of a sufficient sample to establish reliable estimates.

- It should be noted that a logarithmic scaling is carried out to be able to visualize the high frequencies, since compared to frequency 0 their heights are very small.

```
# Only calibration period
from lifetimes.plotting import plot_period_transactions

fig = plt.figure(figsize=(12,8))
ax = plot_period_transactions(bgf_val)
ax.set_yscale('log')
```

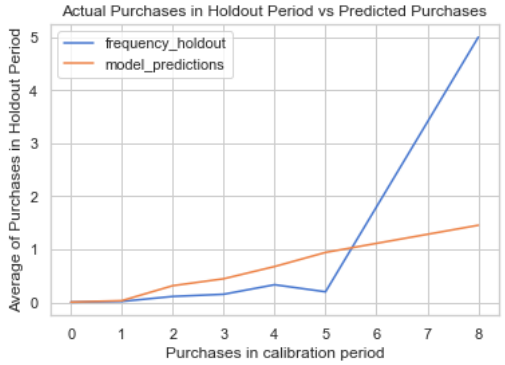
```
<Figure size 864x576 with 0 Axes>
```



Our second visualization will give us a more accurate understanding of the effectiveness of our model estimates, when comparing actual purchases versus predictions.

```
# Predict frequency of orders in the holdout period.
# X axis: Calibration period purchases. Y axis: Hold out period purchases.
# The blue and orange (model preds) should overlap for a good model.
# Both lines being in diagonal indicates, the behaviour and holdout and calib periods are similar.

from lifetimes.plotting import plot_calibration_purchases_vs_holdout_purchases

fig = plt.figure(figsize=(12,8))
plot_calibration_purchases_vs_holdout_purchases(model=bgf_val, calibration_holdout_matrix=rfm_val)
```

```
<AxesSubplot:title={'center':'Actual Purchases in Holdout Period vs Predicted Purchases'}, xlabel='Purchases in calibration
period', ylabel='Average of Purchases in Holdout Period'>
<Figure size 864x576 with 0 Axes>
```



Note:

1. If the orange line and blue line coincide, then model's predictions for holdout matches with reality.
2. I the orange line and blue line are along the diagonal, the buying behaviour in holdout matches the buying behaviour in calibration.

Observation

- In this case, it is not on the diagonal. This can be due to: insufficient repeat purchases or insufficient length of the holdout period to observe the repeat purchase.

- We can see how the model starts making predictions quite close to the margin of error. The said distance increases as the number of purchases increases until it shoots up violently.
- In any case, our model maintains a coherent exponential growth and is similar in some points to the reality of the business. Extending the time that is taken for the test could improve the validation of our model. Another possibility is to tune the penalizer parameter.

Double-click (or enter) to edit

29. Gamma-Gamma model

BG/NBD helped predict the number of transactions but not the monetary value, which is needed to estimate CLTV.

We have predictions for our next few months but we do not know exactly what will be the remuneration that we will obtain from our clients.

A client who buys 13 times an average of 7 dollars is not the same as another who buys 2 times 300 dollars. This is when the second model in this case will help us to give an estimate of the profitability of each client, using the *monetary_value* column which we had left until now.

Gamma-Gamma is a bit like BG/NBD. It assumes that the monetary value of a customer's transactions will vary randomly around their average order value and **the order value is independent of the transaction process.**

We begin first of all with the fact that one of the requirements of this model is to **work only with observations with a purchase frequency greater than 0,** we see that our sample is considerably reduced to ~2000 subjects. We'll be able to train a model.

```
rfm_gg = rfm[rfm['frequency'] > 0]
len(rfm_gg)
```

```
2015
```

To use Gamma-Gamma, we need to make sure that **there is no correlation between the frequency of purchase and the monetary value of each consumer**.

This is due to the fact that the Gamma-Gamma model assumes that there is no such correlation.

In practice, check if the Pearson correlation between the two vectors is close to 0 in order to use this model.

Check correlation

```
rfm_gg[['monetary_value', 'frequency']].corr()
```

	monetary_value	frequency
monetary_value	1.000000	0.002335
frequency	0.002335	1.000000

Train Model

```
from lifetimes import GammaGammaFitter

ggf = GammaGammaFitter(penalizer_coef = 0.0)
ggf.fit(rfm_gg['frequency'], rfm_gg['monetary_value'])
print(ggf)
```

```
<lifetimes.GammaGammaFitter: fitted with 2015 subjects, p: 5.83, q: 2.83, v: 47.69>
```

One of the faculties of this model is to calculate the conditional expectation for the average profit per transaction for a group of one or more customers.

```
# predicted average order value for each customer.
rfm['avg_transaction'] = round(ggf.conditional_expected_average_profit(rfm_gg['frequency'],
                                                                       rfm_gg['monetary_value']), 2)
```

```
rfm['avg_transaction'] = rfm['avg_transaction'].fillna(0)
rfm.sort_values(by='avg_transaction', ascending=False)
```

customer_unique_id	frequency	recency	T	monetary_value	expected_4week	expected_8week	expected_12week	prob_
c8460e4251689ba205045f3ea17884a1	1.0	1.0	22.0	2405.28	0.01	0.01	0.01	0.1
a1044dd75b74fbc485b040575a14acf0	1.0	8.0	260.0	1650.18	0.00	0.00	0.00	0.0
012a218df8995d3ec3bb221828360c86	1.0	42.0	114.0	1435.97	0.00	0.00	0.01	0.0
da1e7179b9c5a1494d78528cbcf05aa0	1.0	174.0	467.0	1391.79	0.00	0.00	0.00	0.0
6e26bbeaa107ec34112c64e1ee31c0f5	1.0	182.0	421.0	1376.45	0.00	0.00	0.00	0.0
...	...	...	...	...	...	...	...	
5618c13afa9188bbe93a11dff76f3659	0.0	0.0	65.0	0.00	0.00	0.00	0.00	1.0
56182a3fa247eeb071d511e046fa5309	0.0	0.0	510.0	0.00	0.00	0.00	0.00	1.0
56181f89940ab3c08ebc6e7749ec6f3d	0.0	0.0	494.0	0.00	0.00	0.00	0.00	1.0
56174dc0b2e7be60e98eb8ade429a0a0	0.0	0.0	321.0	0.00	0.00	0.00	0.00	1.0
ffffd2657e2aad2907e67c3e9daecbeb	0.0	0.0	484.0	0.00	0.00	0.00	0.00	1.0

93357 rows × 10 columns

The column that we have just created gives us the approximate amount that the customer will make in a future purchase, but if we review well we will find that this metric does not really tell us anything by itself, because customers with the highest number of transaction are unlikely to buy in the following periods.

In summary, this new indicator is conditioned on the assumption that a purchase is actually made.

To find the value we are looking for, and which is found in the title of this article, we will use another module of this model which uses the model we created previously to estimate the CLV of each client in the period we choose.

For our reality we will use an estimate of the next 26 weeks (6 months) to calculate the Lifetime Value.

- *discount_rate* refers to external factors that may influence the company's income, such as currency inflation, new company taxes, among others.

```
# CLTV Caluclation for 6 Months
rfm['CLV'] = round(ggf.customer_lifetime_value(bgf,
                                               rfm['frequency'],
                                               rfm['recency'], rfm['T'], rfm['monetary_value'],
                                               time=26, discount_rate=0.01, freq='W'))  # 26 weeks

rfm = rfm.sort_values(by='CLV', ascending=False)
```

Now that we have our final variable, we will move on to the last step of this case, the clustering of our consumers based on the profitability provided to the company. For this we can already eliminate our initial variables that we use to build the models.

```
clusters = rfm.drop(rfm.iloc[:, 0:4], axis=1)
clusters.drop(columns=['prob_alive', 'predicted_purchases_12'], inplace=True)
clusters.head()
```

customer_unique_id	expected_4week	expected_8week	expected_12week	avg_transaction	CLV
397b44d5bb99eabf54ea9c2b41ebb905	0.03	0.06	0.08	458.08	304.0
fe81bb32c243a86b2f86fbf053fe6140	0.03	0.07	0.10	365.11	286.0
4facc2e6fbc2bffab2fea92d2b4aa7e4	0.02	0.04	0.06	496.44	235.0
297ec5afd18366f5ba27520cc4954151	0.02	0.03	0.05	626.92	222.0
c8460e4251689ba205045f3ea17884a1	0.01	0.01	0.01	1868.33	193.0

30. CLTV Segments

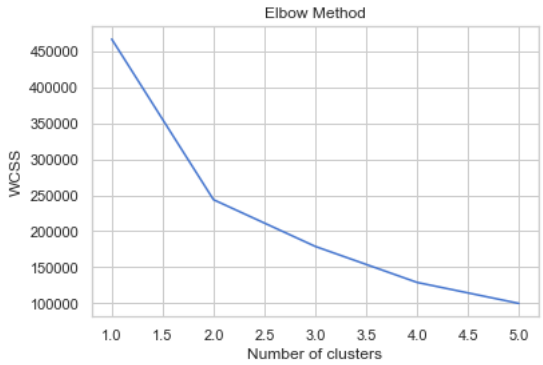
We have four variables to be able to carry out our segmentation, for this we will use an unsupervised Machine Learning method already known to many, K-Means. To make use of it, it is necessary to normalize our variables, since as this is a model that works with the distances of the data, the dimensions and magnitudes play a very important role in its implementation.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled = scaler.fit_transform(clusters)
scaled
```

```
array([[ 3.90026275e+01,  3.94643508e+01,  3.50034930e+01,
         1.60275509e+01,  1.00944316e+02],
       [ 3.90026275e+01,  4.60483602e+01,  4.37659344e+01,
         1.27513348e+01,  9.49464094e+01],
       [ 2.59907873e+01,  2.62963320e+01,  2.62410516e+01,
         1.73793380e+01,  7.79523414e+01],
       ...,
       [-3.28930266e-02, -3.97056170e-02, -4.62727340e-02,
        -1.14958299e-01, -3.53658271e-01],
       [-3.28930266e-02, -3.97056170e-02, -4.62727340e-02,
        -1.14958299e-01, -3.53658271e-01],
       [-3.28930266e-02, -3.97056170e-02, -4.62727340e-02,
        -1.14958299e-01, -3.53658271e-01]])
```

When choosing the number of clusters to create, it is advisable to discuss it with the department involved and set a number that responds to the nature of the business and the needs of the problem. But in our case we will use the famous "Elbow Method" that will indicate the number of clusters that reduces the inertia (closeness of the points to their centroid) to a relevant point.

```
from sklearn.cluster import KMeans

wcss = []
for i in range(1, 6):
    kmeans = KMeans(n_clusters=i, max_iter=1000, random_state=0)
    kmeans.fit(scaled)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 6), wcss)
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



I consider that the number three is the most appropriate for our segmentation, due to being a symbolically very powerful number in time and of course this is the result of our graph. With this number of clusters we can now train our model with the scaled database and attach the created labels to our initial table.

```
model = KMeans(n_clusters = 3, max_iter = 1000)
model.fit(scaled)
labels = model.labels_
```

```
clusters['cluster'] = labels
clusters['cluster'].value_counts()
```

```
0    92171
2     1123
1       63
Name: cluster, dtype: int64
```

At first glance, we see that our cluster "0" makes a clear reference to the entire group of sporadic shopping users that represented the vast majority of our e-commerce. We can get a better visualization of these new segments by using PCA to reduce our variables to two and be able to translate them into a scatter plot.

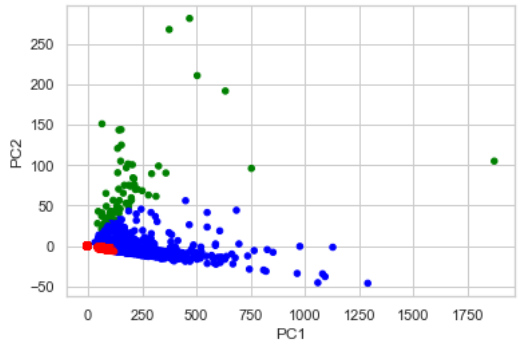
```
from sklearn.decomposition import PCA

features = ["expected_4week", "expected_8week", "expected_12week", "avg_transaction", "CLV"]

pca = PCA(n_components=2)
components = pca.fit_transform(clusters[features])
npc = np.array(components)
dfc = pd.DataFrame(npc, columns=['PC1','PC2'])
dfc = dfc.set_index(clusters.index)
dfc['label'] = clusters['cluster']
dfc
```

customer_unique_id	PC1	PC2	label
397b44d5bb99eabf54ea9c2b41ebb905	468.413093	281.460885	1
fe81bb32c243a86b2f86fbf053fe6140	374.706630	267.803691	1
4facc2e6fbc2bffab2fea92d2b4aa7e4	503.522941	210.751703	1
297ec5afd18366f5ba27520cc4954151	633.257257	191.698150	1
c8460e4251689ba205045f3ea17884a1	1871.975717	105.001176	1
...	...	...	...
992143aff0c52bffd8d709bd00f57465	-3.308026	-0.908497	0
2c693e71cc2c3b66b36ecf55c7668f49	-3.308026	-0.908497	0
99232d8066fef1bfa92bdab71d0097ef	-3.308026	-0.908497	0
cff2e70f7fd43f570fd3ba208bef731e	-3.308026	-0.908497	0
ffffd2657e2aad2907e67c3e9daecbeb	-3.308026	-0.908497	0

93357 rows × 3 columns

```
dfc.plot(x="PC1", y="PC2", kind="scatter", color=dfc.label.map({0:'red', 1:'green', 2:'blue'}))
```

```
<AxesSubplot:xlabel='PC1', ylabel='PC2'>
```



We saw that the K-Means model did a good job clustering our users, but my intuition together with the result of the graph and the number of users per cluster tell me that the order of the clusters does not really represent the importance of the client with respect to the cost effectiveness. Let's corroborate this mini theory by looking at the maximums and minimums of each cluster.

```
clusters.groupby('cluster').agg(['max','min','mean'])['CLV']
```

	max	min	mean
cluster			
0	4.0	0.0	0.928427

Indeed, group 1 represents the group with the highest profitability, while group 2 represents the group with moderate profitability. Let's finish our final table by assigning the corresponding names to each segment.

```
clusters['cluster'].replace(to_replace=[0,1,2], value = ['Non-Profitable', 'Very Profitable', 'Profitable'], inplace=Tru
clusters.sort_values(by='CLV', ascending=False)
```

customer_unique_id	expected_4week	expected_8week	expected_12week	avg_transaction	CLV	cluster
397b44d5bb99eabf54ea9c2b41ebb905	0.03	0.06	0.08	458.08	304.0	Very Profitable
fe81bb32c243a86b2f86fbf053fe6140	0.03	0.07	0.10	365.11	286.0	Very Profitable
4facc2e6fbc2bffab2fea92d2b4aa7e4	0.02	0.04	0.06	496.44	235.0	Very Profitable
297ec5afd18366f5ba27520cc4954151	0.02	0.03	0.05	626.92	222.0	Very Profitable
c8460e4251689ba205045f3ea17884a1	0.01	0.01	0.01	1868.33	193.0	Very Profitable
...	...	...	...	...	...	...
76fa7a3d20729ce3f3ef2088d6f81e37	0.00	0.00	0.00	0.00	0.0	Non-Profitable
11a40baf46bd96289d1f0936dae9a256	0.00	0.00	0.00	0.00	0.0	Non-Profitable
11a3ebf9df2955bc0963bc4b6c629086	0.00	0.00	0.00	0.00	0.0	Non-Profitable
11a3148cf0bc61a3fa3f4b46a79825a1	0.00	0.00	0.00	0.00	0.0	Non-Profitable
ffffd2657e2aad2907e67c3e9daecbeb	0.00	0.00	0.00	0.00	0.0	Non-Profitable

93357 rows × 6 columns

We begin our case study with a typical relational database of various E-Commerce, ending with a segmentation of each of its users regarding their probability of future purchase and the profitability of the company.

The *Lifetimes* package is a very powerful package, although at the beginning it feels like a black box, the study and understanding of each of its functions together with the validation of its models, makes it an ally for all that company with the intention to know your consumers better and optimize your investment in each area of contact with the customer.

I hope you have enjoyed this case and it has been "profitable" for a better understanding of both the library and the application capacity it brings to the business.

Recomendations

- A better performance of each model should be given with the application in more homogeneous databases referring to each company.
- The use of more specific variables at the time of clustering can give more precise and enriching results for the area to which the study is intended.