Customer Value

CLV stands for "Customer Lifetime Value". calcularion using Spark/PySpark and FRM (Frequency, Recency, and Monetary Value) - one method used to segment customers based on their purchase behavior.

In [1]:

```
## EMR and Athena for Spark Job already have spark session set-up
## EXECUTE ONLY IN LOCAL DEVELOPMENT
import findspark
findspark.init()

import pandas as pd
from pyspark.sql import SparkSession

## default Spark appName - se preferir
spark = SparkSession.builder.appName('Spark3-quick-demo-app').master('local[*]').getOrCr
sc = spark.sparkContext
spark
```

Out[1]:

SparkSession - in-memory
SparkContext

[Spark UI (http://G15AMD:4040)](http://G15AMD:4040)

Version
`v3.3.1`
Master
`local[*]`
AppName
`Spark3-quick-demo-app`

Aux functions

In [2]:

```
## Aux function

def fshape(dataframe1):
    print('Shape : ', dataframe1.count(), len(dataframe1.columns))

def fhead(dataframe1, num_records=3):
    ## Show all columns - pandas dataframe
    # import pandas as pd
    # pd.options.display.max_columns = None

    return dataframe1.limit(num_records).toPandas()

def fsummary(dataframe1):
    return dataframe1.summary().toPandas()
```

Quick info related to the dataset

Original dataset - converted to Parquet (typical file format stored in S3)

- https://archive.ics.uci.edu/ml/datasets/online+retail (https://archive.ics.uci.edu/ml/datasets/online+retail)

In [3]:

```
## read local file
sdf = spark.read.parquet('./data_input/OnlineRetail__AWS.parquet')
# sdf.printSchema()

fshape(sdf)
fhead(sdf)
```

```
Shape :  541909 8
```

Out[3]:

	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country
0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	2010-12-01 06:26:00	2.55	17850.0	Unitec Kingdom
1	536365	71053	WHITE METAL LANTERN	6	2010-12-01 06:26:00	3.39	17850.0	Unitec Kingdom
2	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	2010-12-01 06:26:00	2.75	17850.0	Unitec Kingdom

Create dataset with customer purchase history and apply CLV formula

- customer_id
- invoice_date

- revenue : monetary value

In [4]:

```
sdf.createOrReplaceTempView('TB_SALES_SDF')
spark.sql('select max(TO_DATE(InvoiceDate)) as current_date_for_FRMV_CLV, current_date a
```

```
+------------------------+----------+
|current_date_for_FRMV_CLV| not_today|
+------------------------+----------+
|              2011-12-09|2023-03-12|
+------------------------+----------+
```

Information to understand the formula

The formula to calculates: **Customer Lifetime Value (CLV) using the FRM (Frequency, Recency, Monetary Value) approach with a discount rate of 10%** .

- monetary_value: the total monetary value spent by the customer.
- frequency: the frequency of customer purchases, i.e., how many times they made a purchase.
- recency_dt: the recency of the customer's purchases, i.e., how many days ago they made their last purchase.
- 365: the number of days in a year.
- 0.1: the discount rate used to calculate the present value of future cash flows.

The formula itself consists of three parts:

- (monetary_value / frequency): this part calculates the average value of each purchase made by the customer.
- (1 - ((recency + 1) / 365)): this part calculates the probability of the customer returning to make a purchase based on the time since their last purchase. The longer the time since the last purchase, the lower the probability of the customer returning to make a purchase.
- / (1 + discount): this part applies the discount rate to calculate the present value of future cash flows.

In [5]:

```
## formula to calculate CLV
def fnc_customer_clv_udf(monetary_value_f, frequency_f, recency_f, discount_f=0.1):
    return round ( ( (monetary_value_f / frequency_f) * (1 - ((recency_f + 1) / 365)) /

## Register the formula to be used by Spark-SQL
from pyspark.sql.types import FloatType

spark.udf.register('fnc_customer_clv_udf', fnc_customer_clv_udf, FloatType())

print("Catalog Entry:")
[print(r) for r in spark.catalog.listFunctions() if "fnc_customer_clv_udf" in r.name]
```

```
Catalog Entry:
Function(name='fnc_customer_clv_udf', description=None, className='org.apa
che.spark.sql.UDFRegistration$$Lambda$3204/1979037004', isTemporary=True)
```

Out[5]:

```
[None]
```

In [6]:

```
## Apply some filters and create the main customer purchase history as an example
sql_query_clv = """
WITH TB_SALES_V AS
(
    SELECT CustomerID as customer_id
        , COUNT(DISTINCT (InvoiceDate))  as frequency
        , DATEDIFF( current_date , MAX (InvoiceDate) )  as recency_now
        , ROUND(SUM(Quantity * UnitPrice), 2) as monetary_value
        , ROUND(avg(Quantity * UnitPrice), 2) as avg_revenue
        , MIN(InvoiceDate) as dt_first_Invoice
        , MAX(InvoiceDate) as dt_last_Invoice
        -- , ROUND(AVG(Quantity), 2) as avg_items
        -- , ROUND(SUM(Quantity), 2) as total_items
    FROM TB_SALES_SDF
    WHERE 1 = 1
        AND InvoiceDate IS NOT NULL
        AND Quantity > 0
        AND UnitPrice > 0
    GROUP BY customer_id
)
SELECT tb3.*
  , ROUND ( ( (monetary_value / frequency) * (1 - ((recency_dt + 1) / 365)) / (1 + 0.1)
  , fnc_customer_clv_udf(monetary_value,frequency,recency_dt) AS CLV_UDF
FROM (
    SELECT tb1.*
        , CAST( DATEDIFF(tb2.dt_current_date , tb1.dt_last_Invoice ) as float) as recency
    FROM TB_SALES_V as tb1
    CROSS JOIN (SELECT MAX(dt_last_Invoice) AS dt_current_date FROM TB_SALES_V) tb2
    ) tb3
WHERE 1 = 1
  AND monetary_value > 0
  AND frequency > 0
  AND customer_id IS NOT NULL
ORDER BY monetary_value DESC
"""

sdf_clv = spark.sql(sql_query_clv)
sdf_clv.printSchema()
```

```
root
 |-- customer_id: double (nullable = true)
 |-- frequency: long (nullable = false)
 |-- recency_now: integer (nullable = true)
 |-- monetary_value: double (nullable = true)
 |-- avg_revenue: double (nullable = true)
 |-- dt_first_Invoice: timestamp (nullable = true)
 |-- dt_last_Invoice: timestamp (nullable = true)
 |-- recency_dt: float (nullable = true)
 |-- CLV_SQL: double (nullable = true)
 |-- CLV_UDF: float (nullable = true)
```

In [7]:

```
print('clv_SQL and clv_udf provide the same information - just show how to implement it
fhead(sdf_clv)
```

```
clv_SQL and clv_udf provide the same information - just show how to implem
ent it using 2 solutions... SQL and UDF
```

Out[7]:

	customer_id	frequency	recency_now	monetary_value	avg_revenue	dt_first_Invoice	dt_l
0	14646.0	51	4113	200541.00	137.36	2010-12-20 08:09:00	
1	16446.0	2	4111	168472.49	56157.50	2011-05-18 06:52:00	
2	17450.0	27	4121	121321.71	588.94	2010-12-07 07:23:00	

Machine Learning - Customer segmentation and plot

- Predictive Power (KI) = 0.741 and Prediction Confidence (KR) = 0.917

In [8]:

```
sdf_clv.createOrReplaceTempView('TB_CLV_SDF')
```

In [9]:

```
def ml_sql_prediction(filename1='./CLV_AWS__Spark_Execution_v1.sql'):
    text_rdd = sc.textFile(filename1)
    # concatenate all lines into a single STRING    ** obs. incluir um tab em todas as l
    text_sql_ml = text_rdd.reduce(lambda x, y: x + y)

    text_sql_ml2 = f"""
        {text_sql_ml}
        """

    return text_sql_ml2
```

In [10]:

```
ml_spark = ml_sql_prediction()

sdf_ml = spark.sql(ml_spark)

sdf_ml.printSchema()
# fhead(sdf_ml)
sdf_ml.show(3, vertical=True)
```

```
root
 |-- customer_id: double (nullable = true)
 |-- frequency: long (nullable = false)
 |-- recency_now: integer (nullable = true)
 |-- monetary_value: double (nullable = true)
 |-- avg_revenue: double (nullable = true)
 |-- dt_first_Invoice: timestamp (nullable = true)
 |-- dt_last_Invoice: timestamp (nullable = true)
 |-- recency_dt: float (nullable = true)
 |-- CLV_SQL: double (nullable = true)
 |-- CLV_UDF: float (nullable = true)
 |-- kc_monetary_value: integer (nullable = false)

-RECORD 0--------------------------------
 customer_id       | 14646.0
 frequency         | 51
 recency_now       | 4113
 monetary_value    | 200541.0
 avg_revenue       | 137.36
 dt_first_Invoice  | 2010-12-20 08:09:00
 dt_last_Invoice   | 2011-12-07 22:12:00
 recency_dt        | 2.0
 CLV_SQL           | 3545.32
 CLV_UDF           | 3545.32
 kc_monetary_value | 10
-RECORD 1--------------------------------
 customer_id       | 16446.0
 frequency         | 2
 recency_now       | 4111
 monetary_value    | 168472.49
 avg_revenue       | 56157.5
 dt_first_Invoice  | 2011-05-18 06:52:00
 dt_last_Invoice   | 2011-12-09 07:15:00
 recency_dt        | 0.0
 CLV_SQL           | 76368.6
 CLV_UDF           | 76368.6
 kc_monetary_value | 5
-RECORD 2--------------------------------
 customer_id       | 17450.0
 frequency         | 27
 recency_now       | 4121
 monetary_value    | 121321.71
 avg_revenue       | 588.94
 dt_first_Invoice  | 2010-12-07 07:23:00
 dt_last_Invoice   | 2011-11-29 07:56:00
 recency_dt        | 10.0
 CLV_SQL           | 3961.8
 CLV_UDF           | 3961.8
 kc_monetary_value | 10
only showing top 3 rows
```

In [11]:

```
fhead(sdf_clv,num_records=4)
```

Out[11]:

	customer_id	frequency	recency_now	monetary_value	avg_revenue	dt_first_Invoice	dt_l
0	14646.0	51	4113	200541.00	137.36	2010-12-20 08:09:00	
1	16446.0	2	4111	168472.49	56157.50	2011-05-18 06:52:00	
2	17450.0	27	4121	121321.71	588.94	2010-12-07 07:23:00	
3	18102.0	30	4111	111259.88	498.92	2011-03-03 06:26:00	

In [12]:

```
## Export as parquet file
# sdf_clv.write.mode('overwrite').parquet('./data_output/OnlineRetail__AWS_FRMV.parquet'
```

Plot and Report sample

In [13]:

```
sdf_ml.createOrReplaceTempView('TB_CLV_SDF_ML')

ml_rpt_sql = """
WITH TB_CLUSTER AS
(
    select kc_monetary_value as cluster_number
    , count(distinct customer_id) as customer_count
    , avg(clv_sql) avg_clv
    , avg(monetary_value) avg_monetary_value
    -- , count(*) as qty_records
    FROM TB_CLV_SDF_ML
    group by kc_monetary_value
)
SELECT cluster_number
--     , customer_count
    , ROUND( customer_count / (select sum(customer_count) from TB_CLUSTER ) * 100, 2) as
    , ROUND( avg_clv, 2) as avg_clv
    , ROUND( avg_monetary_value, 2) as avg_monetary_value
FROM TB_CLUSTER tb1
order by avg_clv desc
"""

sdf_ml_rpt = spark.sql(ml_rpt_sql)
# sdf_ml_rpt.printSchema()
sdf_ml_rpt.show()
```

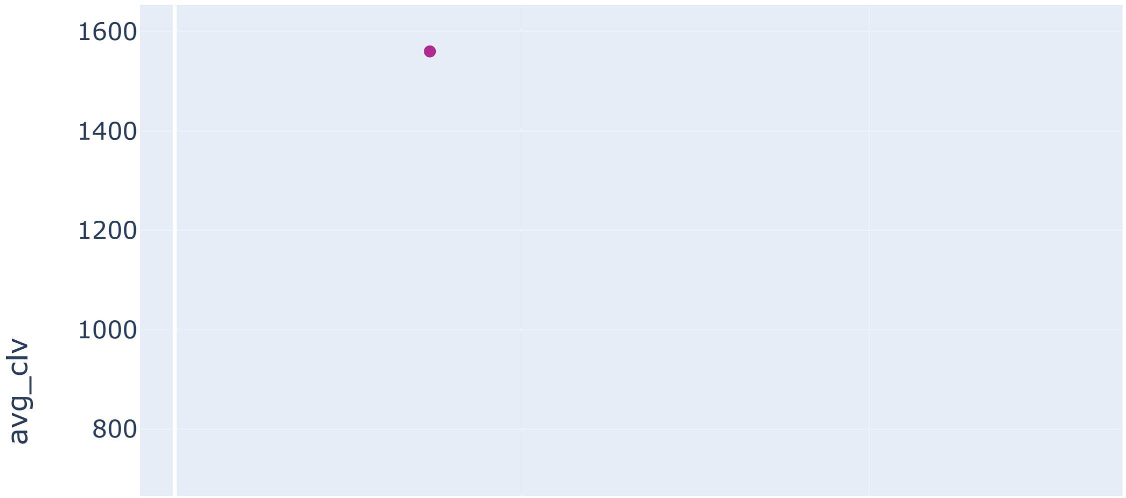
```
+--------------+-------------------+-------+------------------+
|cluster_number|percent_of_customers|avg_clv|avg_monetary_value|
+--------------+-------------------+-------+------------------+
|             5|               1.92|1560.08|           3673.24|
|            10|               2.44| 654.88|          19804.38|
|             4|               3.27| 406.07|           4364.46|
|             1|               2.98|  379.3|           4204.32|
|             9|               0.93| 376.87|           3707.09|
|             8|              21.94| 302.95|            747.88|
|             3|              11.45| 284.33|           1050.47|
|            11|               0.22| 281.17|           1948.79|
|             7|               8.37| 211.57|            678.51|
|             2|               8.53| 200.86|           1177.56|
|             6|              37.94| 192.44|            706.01|
+--------------+-------------------+-------+------------------+
```

Plot

In [14]:

```
sdf_ml_rpt.pandas_api().plot.scatter(x='avg_monetary_value', y='avg_clv', color='cluster
```

```
WARNING:root:'PYARROW_IGNORE_TIMEZONE' environment variable was not set. I
t is required to set this environment variable to '1' in both driver and e
xecutor sides if you use pyarrow>=2.0.0. pandas-on-Spark will set it for y
ou but it does not work if there is a Spark context already launched.
```



Optimization in Spark - considerations

Spark 1.x : Catalyst Optimizer and Tungsten Project (CPU, cache and memoery efficiency, eliminating the overhead of JVM objects and garbage collection)

Spark 2.x : Cost-Based Optimizer (CBO) to improve queries with multiple joins, using table statistics to determine the most efficient query execution plan

Spark 3.x : Adaptive Query Execution (AQE) is an optimization technique in Spark SQL that use runtime statistics to choose the most eficient query execution plan, which is enabled by default since Apache Spark 3.2.0

- https://spark.apache.org/docs/latest/sql-performance-tuning.html (https://spark.apache.org/docs/latest/sql-performance-tuning.html)
- three major features in AQE: including coalescing post-shuffle partitions, converting sort-merge join to broadcast join, and skew join optimization

This notebook use Spark 3.x and Adaptive Query Execution (AQE)

In [ ]:

```

```