Predicting Customer Lifetime Value - Intermediate

KeytoDataScience.com

For any Data Science project, we will follow the approach of Data Science Project Lifecycle:

💡💡💡💡💡 Data Science Project Life Cycle [1] 💡💡💡💡💡

---

1. **Understand Problem/Objective**
2. **Data Collection**
3. **Data Preparation**
   - 3.1 Data Preprocessing
   - 3.2 EDA[3]
   - 3.3 Train/Validation/Test Split
   - 3.4 Feature Engineering
   - 3.5 Feature Selection
4. **Modeling:** Regression [6]
5. **Evaluation:** Regression
   - RMSE, RSE, MAE, RAE, Coefficient of Determination (R2)
6. **Model Deployment**
   - Model Deployment in pipeline or tool

In this case study, We will use past purchase history of customers to build a model that can predict the **Customer Lifetime Value (CLTV or CLV)** for new customers.

Table of Contents

- 1. Understand Objective
- 2. Data Collection
- 3. Data Preparation
  - 3.1 Data Preprocessing
  - 3.2 Exploratory Data Analysis (EDA)
  - 3.3 Feature Engineering
    - 3.3.1 Handling Outliers
  - 3.4 Feature Selection
    - 3.4.1 Check for Correlation among independent variables
  - 3.5 Train and Test Split

- 4. Modeling
  - 4.1 Linear Regression [7]
    - 4.1.1 Check Assumptions
      - 1. Linearity
      - 2. Mean of Residuals
      - 3. Check for Homoscedasticity
      - 4. Check for Normality of error terms/residuals
      - 5. No autocorrelation of residuals
      - 6. No perfect multicollinearity
    - 4.1.2 Build Model - Linear Regression
  - 4.2 DecisionTreeRegressor [8]
  - 4.3 RandomForestRegressor [9]
- 5. Evaluation [10]
  - 5.1 Select Final Model
  - 5.2 Save Model to Disk
  - 5.3 Interpret the Output
  - 5.4 Linear Regression with StandardScaler (Optional)
- 6. Model Deployment
  - 6.1 Import Libraries
  - 6.2 Load Model from Disk
  - 6.3 Real Time Prediction

1. Understand Objective

Customer Lifetime Value(CLTV)[2]

"Customer Lifetime Value is a monetary value that represents the amount of revenue or profit a customer will give the company over the period of the relationship" (Source). CLTV demonstrates the implications of acquiring long-term customers compare to short-term customers. Customer lifetime value (CLV) can help you to answers the most important questions about sales to every company:

1. How to Identify the most profitable customers?
2. How can a company offer the best product and make the most money?
3. How to segment profitable customers?
4. How much budget need to spend to acquire customers?

Business Terms

- **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

- **Purchase Frequency:** 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
- **Churn Rate:** Percentage of customers who have not ordered again.
- **Customer Lifetime:** Customer Lifetime is the period of time that the customer has been continuously ordering.
    - Customer lifetime = 1 / Churn Rate
- **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%.
    - Repeat Rate = 1 - Churn Rate

↥ back to top

2. Data Collection

Import Libraries

In [1]:

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
import os

import matplotlib.pylab as plt
plt.style.use('ggplot')
import seaborn as sns

from sklearn.model_selection  import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.preprocessing import StandardScaler

from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
from statsmodels.stats.diagnostic import acorr_ljungbox
import statsmodels.api as sm
import statsmodels.stats.api as sms
from statsmodels.compat import lzip

import joblib
```

Download Data[2]

Load Data

We will load the data file for this example and checkout summary statistics and columns for that file.

In [2]:

```
df = pd.read_csv("history.csv")
```

Check out the Data

In [3]:

```
df.shape
```

Out[3]:

```
(100, 8)
```

In [4]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 8 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CUST_ID  100 non-null    int64
 1   MONTH_1  100 non-null    int64
 2   MONTH_2  100 non-null    int64
 3   MONTH_3  100 non-null    int64
 4   MONTH_4  100 non-null    int64
 5   MONTH_5  100 non-null    int64
 6   MONTH_6  100 non-null    int64
 7   CLV      100 non-null    int64
dtypes: int64(8)
memory usage: 6.4 KB
```

The dataset consists of the customer ID, the amount the customer spent on your website for the first months of his relationship with your business and his ultimate life time value (say 3 years worth)

In [5]:

```
df.describe()
```

Out[5]:

	CUST_ID	MONTH_1	MONTH_2	MONTH_3	MONTH_4	MONTH_5	MONTH_6	CL
count	100.000000	100.00000	100.000000	100.000000	100.000000	100.000000	100.000000	100.00000
mean	1050.500000	113.25000	115.750000	106.250000	106.750000	106.250000	108.500000	9421.19000
std	29.011492	55.32902	64.221739	63.601406	62.649317	59.816111	66.021499	2664.44317
min	1001.000000	25.00000	0.000000	0.000000	0.000000	0.000000	0.000000	4125.00000
25%	1025.750000	75.00000	75.000000	50.000000	50.000000	50.000000	50.000000	7816.00000
50%	1050.500000	100.00000	125.000000	100.000000	100.000000	100.000000	100.000000	9344.00000
75%	1075.250000	150.00000	175.000000	175.000000	150.000000	156.250000	175.000000	10719.25000
max	1100.000000	200.00000	200.000000	200.000000	200.000000	200.000000	200.000000	17100.00000

In [6]:

```
df.head()
```

Out[6]:

	CUST_ID	MONTH_1	MONTH_2	MONTH_3	MONTH_4	MONTH_5	MONTH_6	CLV
0	1001	150	75	200	100	175	75	13125
1	1002	25	50	150	200	175	200	9375

	CUST_ID	MONTH_1	MONTH_2	MONTH_3	MONTH_4	MONTH_5	MONTH_6	CLV
2	1003	75	150	0	25	75	25	5156
3	1004	200	200	25	100	75	150	11756
4	1005	200	200	125	75	175	200	15525

↥ back to top

3. Data Preparation

3.1 Data Preprocessing

In [7]:

```
# drop CUST_ID
df=df.drop("CUST_ID",axis=1)

# or
# df.drop("CUST_ID",axis=1,inplace=True)
```

Check Null Values

In [8]:

```
df.isnull().sum()
```

Out[8]:

```
MONTH_1    0
MONTH_2    0
MONTH_3    0
MONTH_4    0
MONTH_5    0
MONTH_6    0
CLV        0
dtype: int64
```

Perform Correlation Analysis

In [9]:

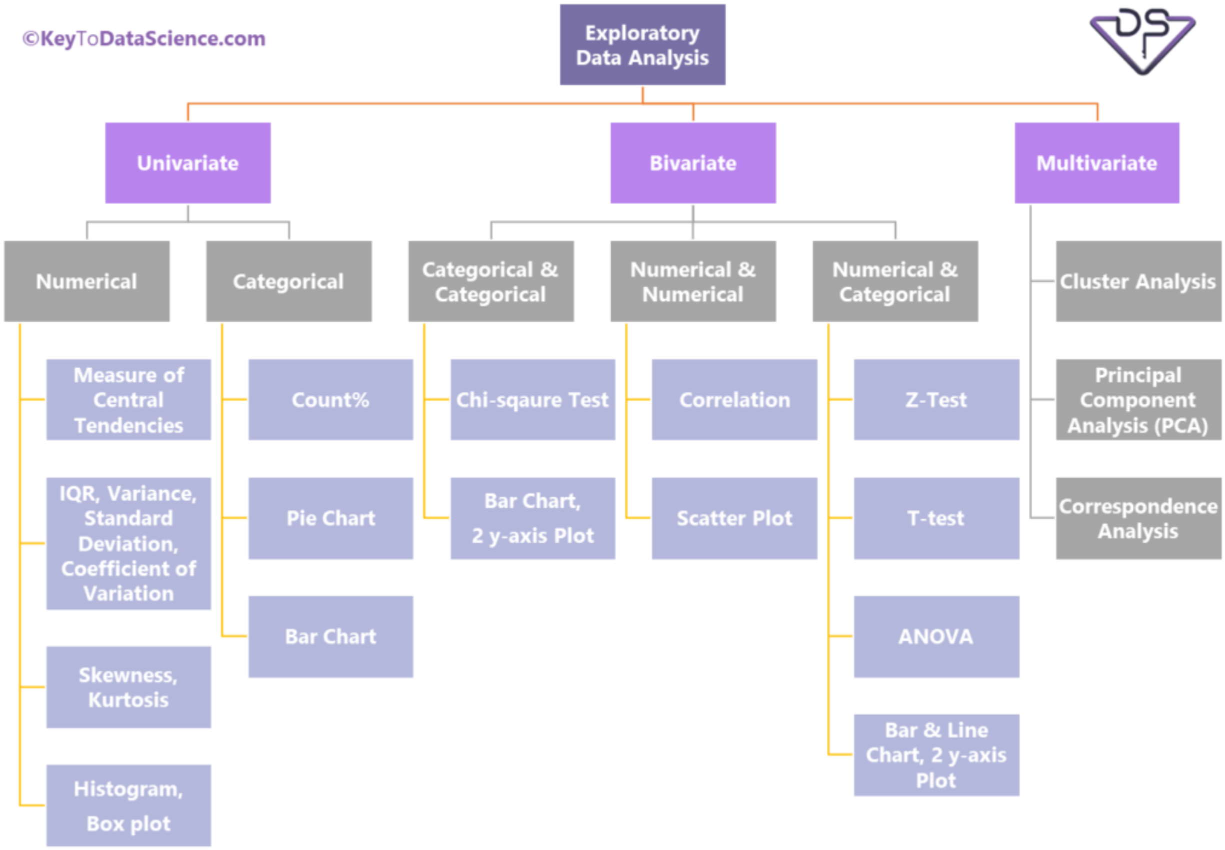
```
df.corr()['CLV']
# -1,1
```

Out[9]:

```
MONTH_1    0.734122
MONTH_2    0.250397
MONTH_3    0.371742
MONTH_4    0.297408
MONTH_5    0.376775
MONTH_6    0.327064
CLV        1.000000
Name: CLV, dtype: float64
```

3.2 Exploratory Data Analysis (EDA)

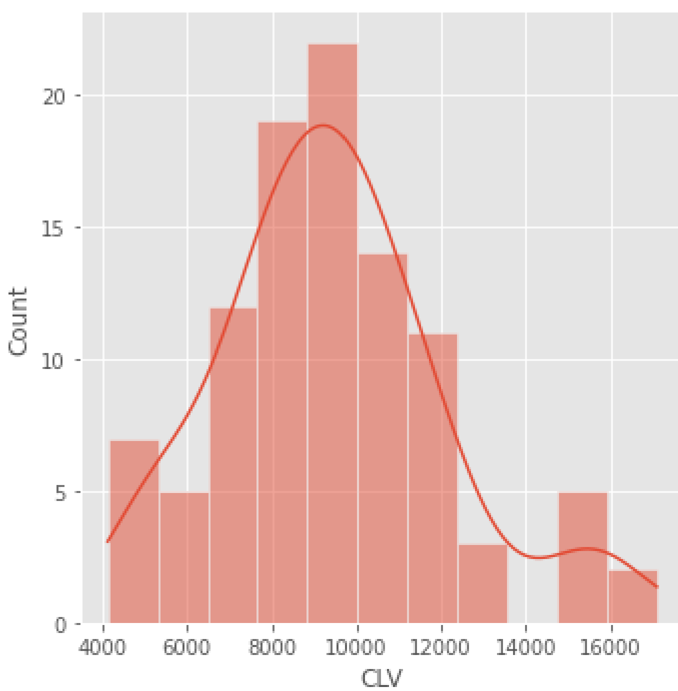
Exploratory Data Analysis (EDA)[3]



Univariate Analysis [4]

In [10]:
```
sns.displot(df['CLV'],kde=True)
```

Out[10]:
```
<seaborn.axisgrid.FacetGrid at 0x2295d2a2c40>
```



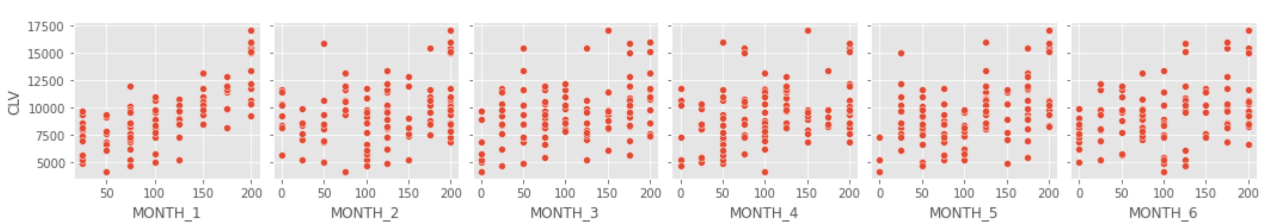
Bivariate Analysis [5]

In [11]:
```
sns.pairplot(df)
```

Out[11]: <seaborn.axisgrid.PairGrid at 0x2295d2a2f70>

In [12]:
```
# df.columns[:-1]
sns.pairplot(df,x_vars=df.columns[:-1],y_vars=['CLV'])
```

Out[12]: <seaborn.axisgrid.PairGrid at 0x22963ec7490>



3.3 Feature Engineering

Think of any new feature or any insight that can be generated out of existing data.

List of Techniques used:

1. Imputation
2. Handling Outliers
3. Binning
4. Log Transform
5. One-Hot Encoding
6. Grouping Operations
7. Feature Split
8. Scaling
9. Extracting Date

3.3.1 Handling Outliers

📌 **Find Outliers**

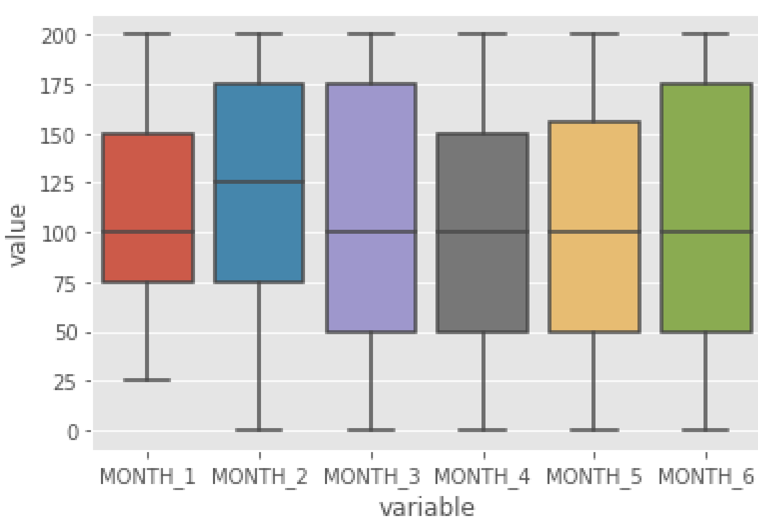
Method 1: Graphical Method - Boxplot [6]

In [13]:

```
# check outliers in the independent variables
df_melted = pd.melt(df[df.columns[:-1]])
sns.boxplot(x='variable', y='value', data=df_melted)
```

Out[13]:

```
<AxesSubplot:xlabel='variable', ylabel='value'>
```



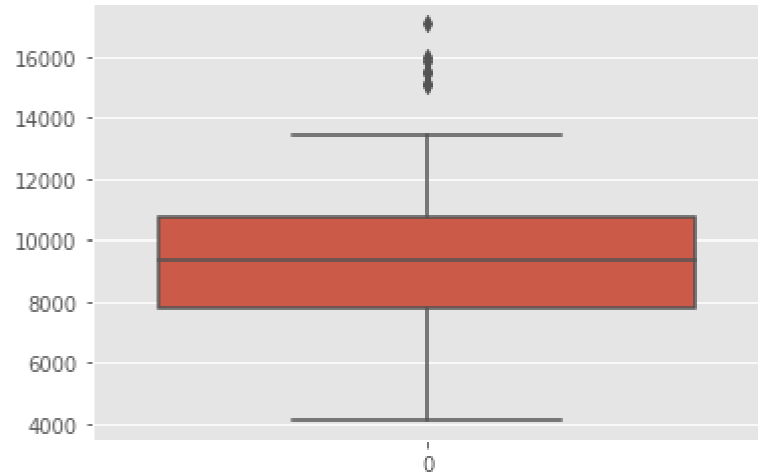
Observation: There are no outliers present in the independent columns.

In [14]:

```
# check outliers in the dependent variable
sns.boxplot(data=df['CLV'])
```

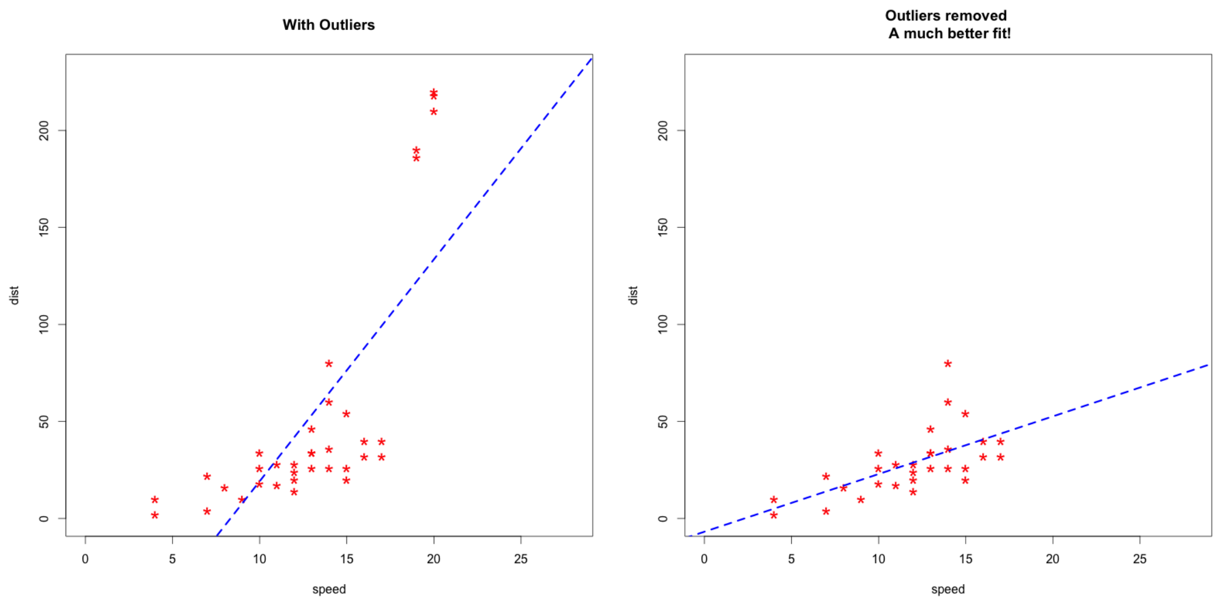
Out[14]:

```
<AxesSubplot:>
```



Observation: There are outliers present in `CLV` Column. How to read boxplots[5]

How removing or treating outliers helps in better model training (better model fitting)



Method 2: Interquartile Range (or IQR)

In [15]:

```
# cols = df.columns
cols = "CLV"
Q1 = df[cols].quantile(0.25)
Q3 = df[cols].quantile(0.75)
IQR = Q3 - Q1
lower_bound = Q1 - (1.5 * IQR)
upper_bound = Q3 + (1.5 * IQR)
```

Check Rows where CLV column has Outliers

In [16]:

```
df[(df[cols] < lower_bound) | (df[cols] >= upper_bound)]
```

Out[16]:

	MONTH_1	MONTH_2	MONTH_3	MONTH_4	MONTH_5	MONTH_6	CLV
4	200	200	125	75	175	200	15525

	MONTH_1	MONTH_2	MONTH_3	MONTH_4	MONTH_5	MONTH_6	CLV
14	200	200	150	150	200	200	17100
28	200	175	50	200	200	175	15450
45	200	50	175	200	200	125	15900
59	200	200	175	75	25	200	15075
70	200	200	200	50	125	175	15975
79	200	200	200	200	200	125	15125

In [17]:

```
# # check rows where CLV column values is lower than lower bound (Q1 - (1.5 * IQR))
# df[df[cols] < lower_bound]
```

In [18]:

```
# # check rows where CLV column values is above than upper bound (Q1 + (1.5 * IQR))
# df[df[cols] >= upper_bound]
```

📌 Outlier Treatment

Method 1: Remove Rows where Outlier is present

In [19]:

```
# # Remove Outlier Columns in case of multiple columns
#df = df[~((df[cols] < (Q1 - 1.5 * IQR)) |(df[cols] > (Q3 + 1.5 * IQR)))]
```

Method 2: Cap the Outlier values to upper or lower bound

In [20]:

```
# create a function  for outlier capping

def outlier_capping(data,col):

    Q1 = data[col].quantile(0.25)
    Q3 = data[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - (1.5 * IQR)
    upper_bound = Q3 + (1.5 * IQR)

    #cap values above high to high
    data.loc[data[cols] < lower_bound,col]=lower_bound

    #cap values below low to low
    data.loc[data[cols] > upper_bound,col]=upper_bound
```

In [21]:

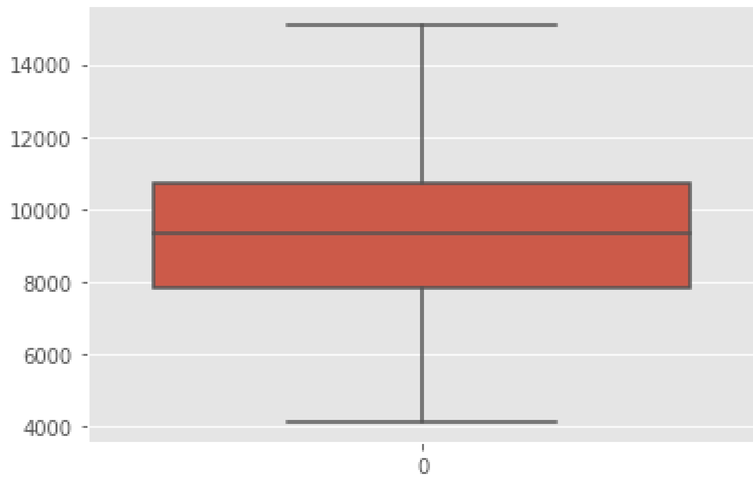
```
# Outlier Treatment: Capping the Outlier values in `CLV` column to upper bound
# select CLV column for outlier capping
outlier_capping(df,'CLV')
```

In [22]:

```
# let's check CLV column outlier treatment by plotting boxplot
sns.boxplot(data=df['CLV'])
```

Out[22]:

<AxesSubplot:>



In [23]:

```
upper_bound
```

Out[23]:

```
15074.125
```

In [24]:

```
# Let's verify Outlier Treatment output
df[df[cols] >= upper_bound]
```

Out[24]:

	MONTH_1	MONTH_2	MONTH_3	MONTH_4	MONTH_5	MONTH_6	CLV
4	200	200	125	75	175	200	15074.125
14	200	200	150	150	200	200	15074.125
28	200	175	50	200	200	175	15074.125
45	200	50	175	200	200	125	15074.125
59	200	200	175	75	25	200	15074.125
70	200	200	200	50	125	175	15074.125
79	200	200	200	200	200	125	15074.125

Observation: Outlier values in `CLV` column are capped to the upper bound

3.4 Feature Selection

3.4.1 Check for Correlation among independent variables

In [25]:

```
# Let's create a function to check correlation among independent variables
def correlation(dataset, threshold):
    col_corr = set()  # set of all the names of correlated columns
    corr_matrix = dataset.corr()
    for i in range(len(corr_matrix.columns)):
        for j in range(i):
            if abs(corr_matrix.iloc[i, j]) > threshold: # we are interested in absolute
                colname = corr_matrix.columns[i]  # getting the name of column
                col_corr.add(colname)
    return col_corr
```

In [26]:
```
# choose correlation threshold is 0.8
corr_features = correlation(df, 0.8)
print('correlated features: ', (set(corr_features)) )
```

```
correlated features:  set()
```

In [27]:
```
# drop columns with high correlation
```

Observation: No independent variables are correlated.

3.5 Train and Test Split

Prepare X(independent) and y(dependent) variables

In [28]:
```
X = df.drop('CLV',axis=1)
y = df.CLV
```

In [29]:
```
X.head()
```

Out[29]:

	MONTH_1	MONTH_2	MONTH_3	MONTH_4	MONTH_5	MONTH_6
0	150	75	200	100	175	75
1	25	50	150	200	175	200
2	75	150	0	25	75	25
3	200	200	25	100	75	150
4	200	200	125	75	175	200

In [30]:
```
y[:6]
```

Out[30]:
```
0    13125.000
1     9375.000
2     5156.000
3    11756.000
4    15074.125
5     7950.000
Name: CLV, dtype: float64
```

We now split the model into training and testing data in the ratio of 70:30

In [31]:
```
X_train, X_test, y_train, y_test =  train_test_split(X, y, test_size=.3)
print("X_train - Training : ", X_train.shape)
print("X_test - Testing   : ", X_test.shape )
```

```
X_train - Training :  (70, 6)
X_test - Testing   :  (30, 6)
```

↥ back to top

4. Modeling

Create a function to quickly build models and evaluate

In [32]:

```python
def model_builder(algo,X_train,y_train,X_test):
    algo.fit(X_train,y_train)

    y_train_pred = algo.predict(X_train)
    y_test_pred = algo.predict(X_test)

    print("The model performance for training set")
    print("--------------------------------------")
    print(f'R2  : {round(r2_score(y_true=y_train,y_pred=y_train_pred),2)}')
    print('MAE :', mean_absolute_error(y_train, y_train_pred))
    print('MSE :', mean_squared_error(y_train, y_train_pred))
    print('RMSE:', np.sqrt(mean_squared_error(y_train, y_train_pred)))

    print("\n")

    print("The model performance for testing set")
    print("--------------------------------------")
    print(f'R2  : {round(r2_score(y_true=y_test,y_pred=y_test_pred),2)}')
    print('MAE :', mean_absolute_error(y_test, y_test_pred))
    print('MSE :', mean_squared_error(y_test, y_test_pred))
    print('RMSE:', np.sqrt(mean_squared_error(y_test, y_test_pred)))
```

4.1 Linear Regression [7]

Linear Regression Model for predicting CLV

4.1.1 Check Assumptions

Linear Regression model has certain assumptions, let's check for these assumptions before building model.

1. Linearity
2. Mean of Residuals
3. Check for Homoscedasticity
4. Check for Normality of error terms/residuals
5. No autocorrelation of residuals
6. No perfect multicollinearity

1. Linearity

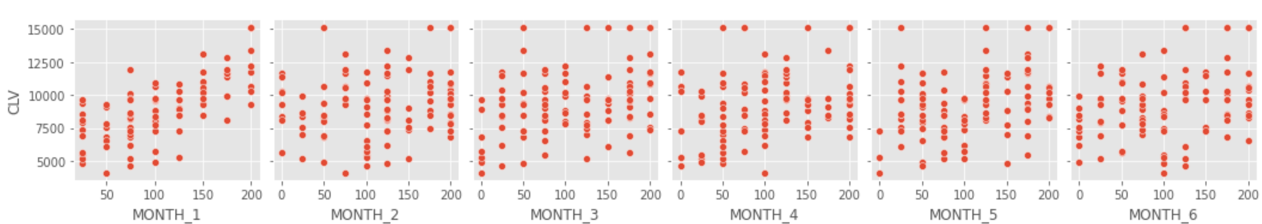
Test 1: Graphical Method

In [33]:

```python
# visualize the relationship between the input features and the response variable using
sns.pairplot(df,x_vars=df.columns[:-1],y_vars=['CLV'])
```

Out[33]:

```
<seaborn.axisgrid.PairGrid at 0x229658a6c10>
```



Observation: Dependent variable `CLV` has positive correlation with all of the independent variables. **PASS**

Test 2: Correlation with Dependent Variable

In [34]:
```
df.corr()['CLV']
# -1,1
```

Out[34]:
```
MONTH_1    0.737985
MONTH_2    0.245683
MONTH_3    0.374436
MONTH_4    0.301838
MONTH_5    0.370002
MONTH_6    0.322038
CLV        1.000000
Name: CLV, dtype: float64
```

Observation: Dependent variable `CLV` has positive correlation with all of the independent variables. **PASS**

💡 **Before checking rest of the assumptions, we need to run the regression model.**

In [35]:
```
#Build model on training data
lr = LinearRegression()
lr.fit(X_train,y_train)
print("Coefficients :", lr.coef_)
print("Intercept    :", lr.intercept_)
y_pred = lr.predict(X_train)
print("R Squared is :",r2_score(y_train, y_pred))
```

```
Coefficients : [32.22632288  9.5601812  14.44109157 12.26793898  7.23867075  4.36676993]
Intercept    : 521.5337473911513
R Squared is : 0.9214324128481604
```

2. Mean of Residuals

The mean of the residuals should be zero. Residuals are the differences between the true value and the predicted value.

In [36]:
```
residuals = y_train.values-y_pred
mean_residuals = np.mean(residuals)
print("Mean of Residuals {}".format(mean_residuals))
```

```
Mean of Residuals 1.6240976817373272e-12
```

Observation: Residuals are close to zero. **PASS**

3. Check for Homoscedasticity

Homoscedasticity means that the residuals have equal or almost equal variance across the regression line. By plotting the error terms with predicted terms we can check that there should not be any pattern in the error terms.

There should be **No heteroscedasticity**

Test 1: Graphical Method

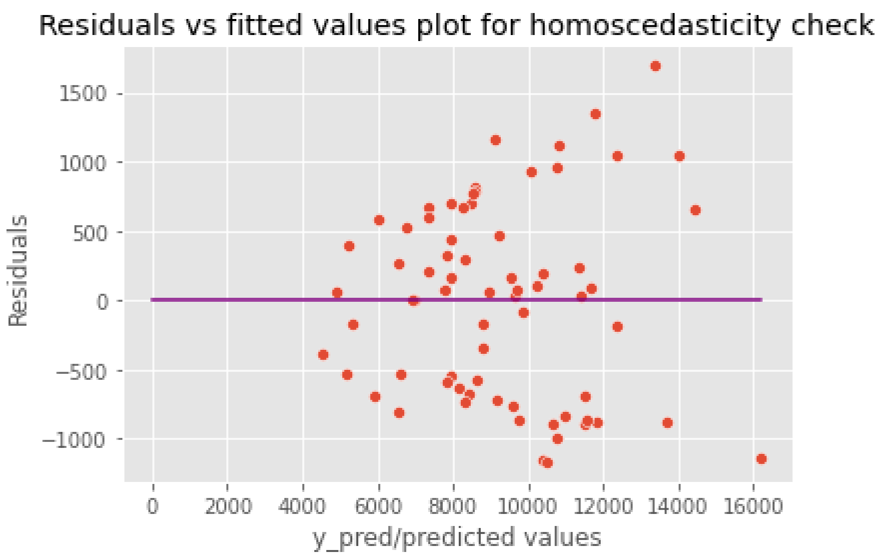
Firstly do the regression analysis and then plot the error terms against the predicted values. If there is a definite pattern (like linear or quadratic or funnel shaped) obtained from the scatter plot then heteroscedasticity is present.

In [37]:

```
sns.scatterplot(x=y_pred,y=residuals)
plt.xlabel('y_pred/predicted values')
plt.ylabel('Residuals')
sns.lineplot(x=[0,max(y_pred)],y=[0,0],color='purple')
plt.title('Residuals vs fitted values plot for homoscedasticity check')
```

Out[37]:

```
Text(0.5, 1.0, 'Residuals vs fitted values plot for homoscedasticity check')
```



Observation: We can obserse a funnel shape in the residuals. Hence, heteroscedasiry is present.

FAIL

Tip:💡 If we want 95% confidence in the tests, then the p-value should be less than 0.05 to be able to reject the null hypothesis. Remember, a researcher or data scientist would always aim to reject the null hypothesis.

Test 2: Goldfeld Quandt Test

Checking heteroscedasticity: Using Goldfeld Quandt we test for heteroscedasticity.

- Null Hypothesis: Error terms are homoscedastic
- Alternative Hypothesis: Error terms are heteroscedastic.

In [38]:

```
## ModuleNotFoundError: No module named 'statsmodels'
## then run below code
# pip install statsmodels
```

In [39]:

```
# import statsmodels.stats.api as sms
# from statsmodels.compat import lzip
name = ['F statistic', 'p-value']
test = sms.het_goldfeldquandt(residuals, X_train)
lzip(name, test)
```

Out[39]: [('F statistic', 1.1034802772579404), ('p-value', 0.3963492434894545)]

Obsevation: Since p value is more than 0.05 in Goldfeld Quandt Test, we can't reject it's null hypothesis that error terms are homoscedastic. **PASS**

Test 3: Bartlett's test

Tests the null hypothesis that all input samples are from populations with equal variances.

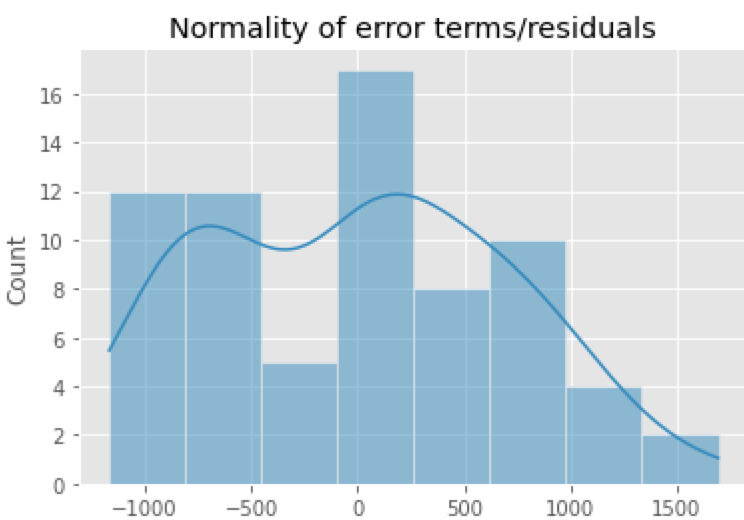
In [40]:
```
from scipy.stats import bartlett
# bartlett(X_train.MONTH_1,X_train.MONTH_2,X_train.MONTH_3,X_train.MONTH_4,X_train.MONT
```

Obsevation: Since p value is quite less than 0.05 in Bartlett, it's null hypothesis that error terms are homoscedastic gets rejected. **FAIL**

4. Check for Normality of error terms/residuals

In [41]:
```
sns.histplot(residuals,kde=True)
plt.title('Normality of error terms/residuals')
```

Out[41]: Text(0.5, 1.0, 'Normality of error terms/residuals')



Observation: The residual terms are looking normally distributed. A slight left skewed is also visible from the plot. However, it is difficult to get perfect normal curves distributions in real data. **PASS**

5. No autocorrelation of residuals

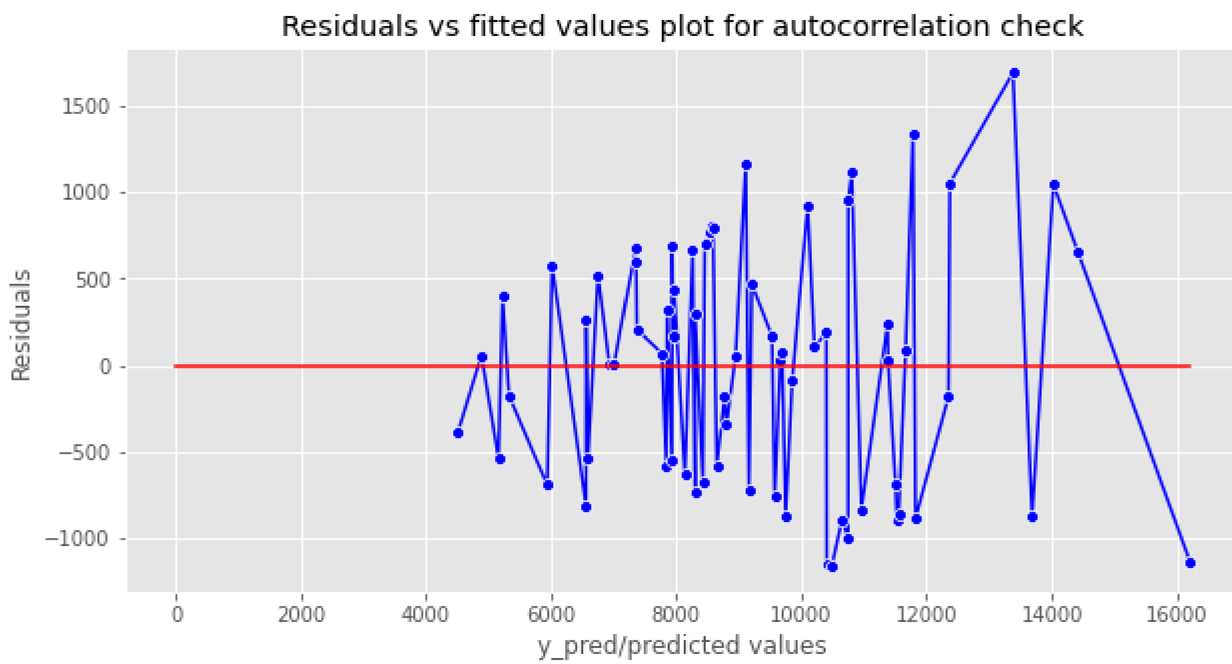
When the residuals are autocorrelated, it means that the current value is dependent of the previous (historic) values and that there is a definite unexplained pattern in the Y variable that shows up in the error terms. Though it is more evident in time series data.

In plain terms autocorrelation takes place when there's a pattern in the rows of the data. This is usual in time series data as there is a pattern of time for eg. Week of the day effect which is a very famous pattern seen in stock markets where people tend to buy stocks more towards the beginning of weekends and tend to sell more on Mondays. There's been great study about this phenomenon and it is still a matter of research as to what actual factors cause this trend.

Test 1: Graphical Method

There should not be autocorrelation in the data so the error terms should not form any pattern.

In [42]:
```
plt.figure(figsize=(10,5))
p = sns.lineplot(x=y_pred,y=residuals,marker='o',color='blue')
plt.xlabel('y_pred/predicted values')
plt.ylabel('Residuals')
p = sns.lineplot(x=[0,max(y_pred)],y=[0,0],color='red')
p = plt.title('Residuals vs fitted values plot for autocorrelation check')
```



Test 2: Ljungbox test to check autocorrelation

- Null Hypothesis: Autocorrelation is absent.
- Alternative Hypothesis: Autocorrelation is present.

In [43]:
```
# from statsmodels.stats.diagnostic import acorr_ljungbox
acorr_ljungbox(residuals, lags=[40], return_df=True)
```

Out[43]:

	lb_stat	lb_pvalue
40	41.663267	0.398276

In [44]:
```
# # OR
# import statsmodels.api as sm
# sm.stats.acorr_ljungbox(residuals, lags=[40], return_df=True)
```

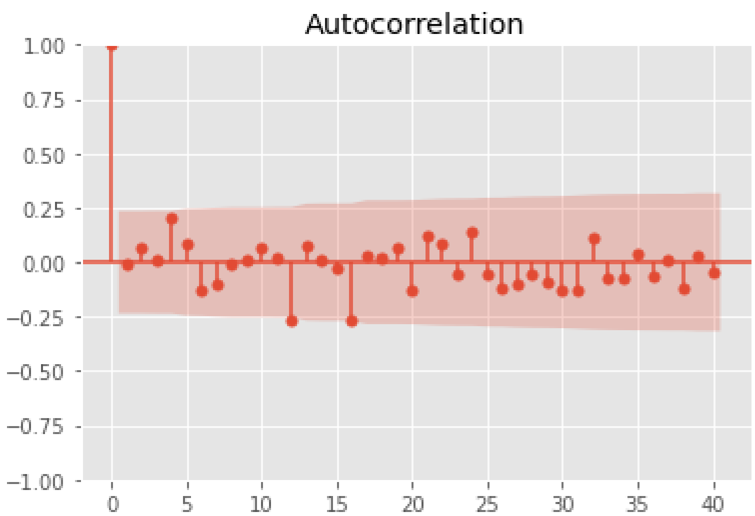
In [45]:
```
# # OR
# from statsmodels.stats import diagnostic as diag
# diag.acorr_ljungbox(residuals , lags = [40],return_df=True)
```

Observation: Since p value is more than 0.05, we can not reject the null hypothesis. Hence, error terms are not autocorrelated. **PASS**

Test 3: Autocorrelation Plot (ACF)

In [46]:
```
# import statsmodels.api as sm
```

In [47]:
```
# autocorrelation
sm.graphics.tsa.plot_acf(residuals, lags=40)
plt.show()
```

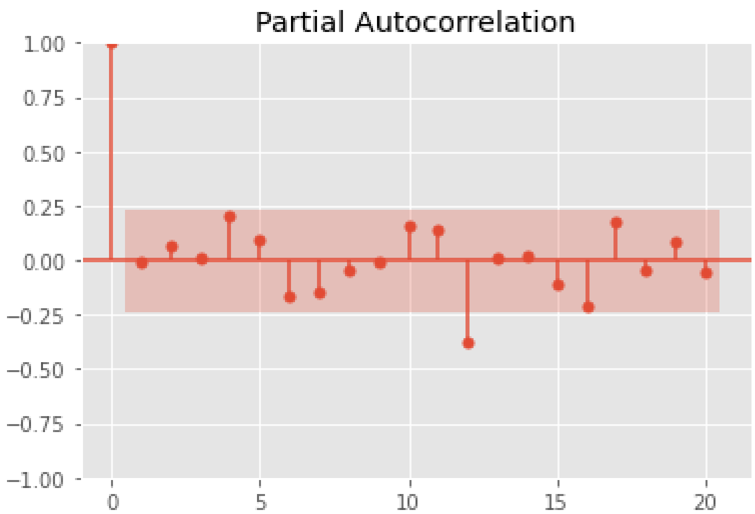


Observation: The results shows no signs of autocorelation, since there are no spikes outside the red confidence interval region. **PASS**

Test 4: Partial Autocorrelation Plot (PACF)

In [48]:
```
# partial autocorrelation
sm.graphics.tsa.plot_pacf(residuals, lags=20)
plt.show()
```

```
C:\Users\Prateek\AppData\Local\Programs\Python\Python39\lib\site-packages\statsmodels\gr
aphics\tsaplots.py:348: FutureWarning: The default method 'yw' can produce PACF values o
utside of the [-1,1] interval. After 0.13, the default will change tounadjusted Yule-Wal
ker ('ywm'). You can use this method now by setting method='ywm'.
  warnings.warn(
```



Observation: The results shows no signs of autocorelation, since there are no spikes outside the red confidence interval region. **PASS**

6. No perfect multicollinearity

Multicollinearity is presence of high correlations among two or more independent variables.

In general, multicollinearity can lead to wider confidence intervals that produce less reliable probabilities in terms of the effect of independent variables in a model.

We have already checked for correlation in **3.4 Feature selection** section. We found no correlated variables.

Q. What is the difference in correlation and multicollinearity?

A.

- Correlation is presence of correlation between 2 variables.
- Multicollinearity is presence of high correlations among two or more independent variables.

Test 1: Variable Inflation Factors (VIF)

VIF starts at 1 and has no upper limit

- **VIF < 5 or 10:** No multicollinearity detected
- **VIF > 5 or 10:** High multicollinearity

In [49]:

```
# # Import library for VIF
# from statsmodels.stats.outliers_influence import variance_inflation_factor

def calc_vif(X):

    # Calculating VIF
    vif = pd.DataFrame()
    vif["Independent Variables"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]

    return(vif)
```

In [50]:
```python
calc_vif(X_train)
```

Out[50]:

	Independent Variables	VIF
0	MONTH_1	4.209731
1	MONTH_2	3.266708
2	MONTH_3	3.578764
3	MONTH_4	4.093205
4	MONTH_5	4.088965
5	MONTH_6	3.358145

Success: So check for assumptions of Linear Regression went successful

Now we can move forward with the model evaluation on test set.

📌 Create a function `model_builder` to fit and evaluate models

In [51]:
```python
def model_builder(algo,X_train,y_train,X_test):
    algo.fit(X_train,y_train)

    y_train_pred = algo.predict(X_train)
    y_test_pred = algo.predict(X_test)

    print("The model performance for training set")
    print("--------------------------------------")
#     print("Accuracy: {}".format(algo.score(X_train,y_train)))
    print(f'R2 score is {round(r2_score(y_true=y_train,y_pred=y_train_pred),2)}')

    print("\n")

    print("The model performance for testing set")
    print("--------------------------------------")
#     print("Accuracy: {}".format(algo.score(X_test,y_test)))
    print(f'R2 score is {round(r2_score(y_true=y_test,y_pred=y_test_pred),2)}')
```

4.1.2 Build Model - Linear Regression

In [52]:
```python
lr = LinearRegression()
model_builder(lr,X_train,y_train,X_test)
```

```
The model performance for training set
--------------------------------------
R2 score is 0.92


The model performance for testing set
--------------------------------------
R2 score is 0.93
```

↥ back to top

4.2 DecisionTreeRegressor [8]

In [53]:
```
# from sklearn.tree import DecisionTreeRegressor

dec_tree = DecisionTreeRegressor(random_state=0)
model_builder(dec_tree,X_train,y_train,X_test)
```

```
The model performance for training set
--------------------------------------
R2 score is 1.0


The model performance for testing set
--------------------------------------
R2 score is 0.45
```

4.3 RandomForestRegressor [9]

In [54]:
```
# from sklearn.ensemble import RandomForestRegressor

rf_tree = RandomForestRegressor(random_state=0)
model_builder(rf_tree,X_train,y_train,X_test)
```

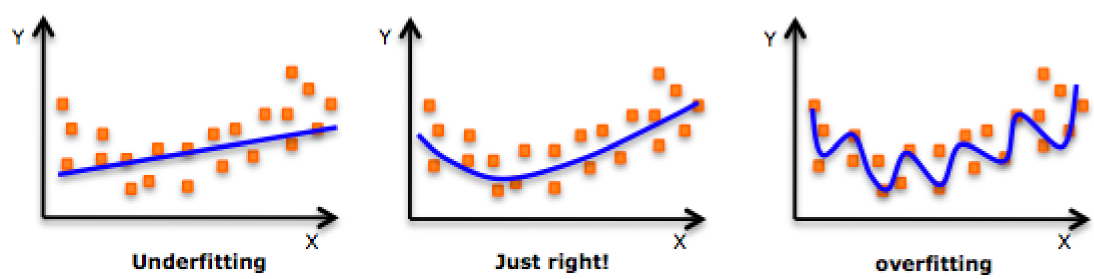
```
The model performance for training set
--------------------------------------
R2 score is 0.96


The model performance for testing set
--------------------------------------
R2 score is 0.76
```

Observation:

Linear Regression model has the highest: R2; and lowest: MAE, MSE, RMSE on the test set.

Both Decision Tree Regressor and Random Forest Regerssor are overfitting



Reference Image for Overfitting

↥ back to top

5. Evaluation [10]

Check Model accuracy by predicting against the test dataset

In [55]:
```
#Test on testing data
```

```
predictions = lr.predict(X_test)
# predictions
```

In [56]:

```
r2=r2_score(y_test, predictions)
print("R Square is: ",round(r2,2))
```

```
R Square is:  0.93
```

📌 **Regression Evaluation Metrics**

Here are three common evaluation metrics for regression problems:

Mean Absolute Error (MAE) is the mean of the absolute value of the errors:

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Mean Squared Error (MSE) is the mean of the squared errors:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Root Mean Squared Error (RMSE) is the square root of the mean of the squared errors:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

Comparing these metrics:

- **MAE** is the easiest to understand, because it's the average error.
- **MSE** is more popular than MAE, because MSE "punishes" larger errors, which tends to be useful in the real world.
- **RMSE** is even more popular than MSE, because RMSE is interpretable in the "y" units.

All of these are **loss functions**, because we want to minimize them.

5.1 Select Final Model

Final Model Selected for production is: Linear Regression

Linear Regression model has the highest: R2; and lowest: MAE, MSE, RMSE on the test set.

In [57]:

```
y_train_pred = lr.predict(X_train)
y_test_pred = lr.predict(X_test)

print("The model performance for training set")
print("--------------------------------------")
print(f'R2  : {round(r2_score(y_true=y_train,y_pred=y_train_pred),2)}')
print('MAE :', mean_absolute_error(y_train, y_train_pred))
print('MSE :', mean_squared_error(y_train, y_train_pred))
print('RMSE:', np.sqrt(mean_squared_error(y_train, y_train_pred)))
```

```
print("\n")

print("The model performance for testing set")
print("--------------------------------------")
print(f'R2  : {round(r2_score(y_true=y_test,y_pred=y_test_pred),2)}')
print('MAE :', mean_absolute_error(y_test, y_test_pred))
print('MSE :', mean_squared_error(y_test, y_test_pred))
print('RMSE:', np.sqrt(mean_squared_error(y_test, y_test_pred)))
```

```
The model performance for training set
--------------------------------------
R2  : 0.92
MAE : 584.0984641819596
MSE : 483380.33305033925
RMSE: 695.2555882913414


The model performance for testing set
--------------------------------------
R2  : 0.93
MAE : 541.4519706013454
MSE : 456675.3817280886
RMSE: 675.7776126271783
```

Linear Regression shows a R-squared of 0.91% on Testing set. This is an excellent model for predicting CLV.

5.2 Save Model to Disk

In [58]:

```
# get current directory path
# os.getcwd()
```

In [59]:

```
# import joblib
filename = os.getcwd()+'/CLV_LinearRegression_Intermediate.joblib.pkl'
joblib.dump(lr, filename, compress=9)
```

Out[59]:

```
['F:\\Work\\Site\\KDS - Career Now Program\\DS\\Syllabus\\5. Case Studies\\Business\\Cus
tome Lifetime Value (CLV)/CLV_LinearRegression_Intermediate.joblib.pkl']
```

5.3 Interpret the Output

In [60]:

```
# print the intercept
print("Intercept:", lr.intercept_)
```

```
Intercept: 521.5337473911513
```

In [61]:

```
coeff_df = pd.DataFrame(lr.coef_,X.columns,columns=['Coefficient'])
coeff_df
```

Out[61]:

	Coefficient
MONTH_1	32.226323
MONTH_2	9.560181

	Coefficient
MONTH_3	14.441092
MONTH_4	12.267939
MONTH_5	7.238671
MONTH_6	4.366770

📌 **Interpreting the coefficients:**

- Holding all other features fixed, a 1 unit increase in **MONTH_1** is associated with an **increase of $32.2** in Customer Lifetime Value (CLTV).
- Holding all other features fixed, a 1 unit increase in **MONTH_2** is associated with an **increase of $9.5** in Customer Lifetime Value (CLTV).
- Holding all other features fixed, a 1 unit increase in **MONTH_3** is associated with an **increase of $14.4** in Customer Lifetime Value (CLTV).
- Holding all other features fixed, a 1 unit increase in **MONTH_4** is associated with an **increase of $12.2** in Customer Lifetime Value (CLTV).
- Holding all other features fixed, a 1 unit increase in **MONTH_5** is associated with an **increase of $7.2** in Customer Lifetime Value (CLTV).
- Holding all other features fixed, a 1 unit increase in **MONTH_6** is associated with an **increase of $4.3** in Customer Lifetime Value (CLTV).

5.4 Linear Regression with StandardScaler (Optional)

In [62]:

```
# from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_train_scaling = scaler.fit_transform(X_train)
```

💡 To preventing information about the distribution of the test set leaking into the model.

Fit the scaler on your training data only, then standardise both training and test sets with that scaler.

In [63]:

```
# we have to scale X_test also, before predicting
# Use transform() on the test data, not fit_transform(), as fit is done on training set
X_test_scaling = scaler.transform(X_test)
```

In [64]:

```
lr_scaling = LinearRegression()
model_builder(lr_scaling,X_train_scaling,y_train,X_test_scaling)
```

```
The model performance for training set
--------------------------------------
R2 score is 0.92


The model performance for testing set
```

```
---------------------------------------
R2 score is 0.93
```

In [65]:

```
coeff_df_scaling = pd.DataFrame(lr_scaling.coef_,X.columns,columns=['Coefficient'])
coeff_df_scaling
```

Out[65]:

	Coefficient
MONTH_1	1774.986976
MONTH_2	616.183840
MONTH_3	947.695757
MONTH_4	783.769872
MONTH_5	446.401332
MONTH_6	280.443532

↥ back to top

6. Model Deployment

The code in model deployment should be able to run independently, i.e. the below code should be independent of all the above performed steps.

Predicting for a new Customer

Let's use the model to predict his CLV.

6.1 Import Libraries

In [66]:

```
# import os
# import joblib
# from sklearn.linear_model import LinearRegression
```

6.2 Load Model from Disk

In [67]:

```
filename = os.getcwd()+'/CLV_LinearRegression_Intermediate.joblib.pkl'
```

In [68]:

```
model = joblib.load(filename)
```

6.3 Real Time Prediction

New Customer Data

Say we have a new customer who in his first 3 months have spend **100,0,50** on the website.

In [69]:

```
new_data = np.array([100,0,50,0,0,0]).reshape(1, -1)
```

Real Time Prediction

In [70]:

```
new_pred=model.predict(new_data)
print("The CLV for the new customer is : $",new_pred[0])
```

```
The CLV for the new customer is : $ 4466.220613491066
```

↥ back to top

Great Job!

Links Used in this notebook:

1. Data Science Project Life Cycle [1]
2. Customer Lifetime Value(CLTV) [2]
3. Exploratory Data Analysis (EDA) [3]
4. Univariate Analysis [4]
5. Bivariate Analysis [5]
6. Boxplot [6]
7. Regression [7]
8. Decision Tree [8]
9. Random Forest [9]
10. Model Evaluation [10]

KeytoDataScience.com