Programmed By : Rithik Tripathi

[Connect with me on Linkedin (https://www.linkedin.com/in/rithik-tripathi-data-scientist/)](https://www.linkedin.com/in/rithik-tripathi-data-scientist/)

Linear Regression

In [1]:

```python
#importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:

```python
# Importing Data
df = pd.read_csv('Dataset/train_cleaned.csv')
df.head()
```

Out[2]:

	Item_Weight	Item_Visibility	Item_MRP	Outlet_Establishment_Year	Item_Outlet_Sales	Item_Fat_Content_LF	Item_Fat_Content_Low Fat	Item_Fat_Content_Re
0	9.30	0.016047	249.8092	1999	3735.1380	0	1	
1	5.92	0.019278	48.2692	2009	443.4228	0	0	
2	17.50	0.016760	141.6180	1999	2097.2700	0	1	
3	19.20	0.000000	182.0950	1998	732.3800	0	0	
4	8.93	0.000000	53.8614	1987	994.7052	0	1	

5 rows × 46 columns

Separating Target Variables

In [3]:

```python
x = df.drop(['Item_Outlet_Sales'], axis=1)
y = df['Item_Outlet_Sales']
df.shape, x.shape, y.shape
```

Out[3]:

```
((8523, 46), (8523, 45), (8523,))
```

Train Test Split

In [4]:

```python
from sklearn.model_selection import train_test_split
train_x, test_x, train_y, test_y = train_test_split(x,y, random_state = 9)

train_x.shape, test_x.shape, train_y.shape, test_y.shape
```

Out[4]:

```
((6392, 45), (2131, 45), (6392,), (2131,))
```

Implementing Linear Regression

In [5]:

```python
from sklearn.linear_model import LinearRegression as LR
from sklearn.metrics import mean_absolute_error as mae
```

In [6]:

```python
# creating instance of Linear Regression Class
lr = LR()

# Training the model with Gradient Descent
lr.fit(train_x, train_y)
```

Out[6]:

```
LinearRegression()
```

predicting over Training Set

In [7]:

```
train_pred = lr.predict(train_x)
train_mae = mae(train_pred, train_y)
print('Training Mean Absolute Error :', train_mae)
```

```
Training Mean Absolute Error : 827.3664175128886
```

predicting over Test Set

In [8]:

```
test_pred = lr.predict(test_x)
test_mae = mae(test_pred, test_y)
print('Test Mean Absolute Error :', test_mae)
```

```
Test Mean Absolute Error : 861.806594100068
```

Parameters of Linear Regression

In [9]:

```
# coefficients of the equation
lr.coef_
```

Out[9]:

```
array([  -1.13512756, -179.79233388,   15.65668767,  -18.6574453 ,
          4.26675415,    4.2154459 ,   43.9152877 ,   20.49224006,
        -72.8897278 ,  -22.38153707,   -1.17784843,  -91.0598699 ,
        -15.79938771,  -39.47904273,  -22.75735283,    8.6798061 ,
        -52.75260518,  -20.67316491,  -20.45574445,   19.09990758,
         29.94212896,  268.66029346,  -41.52897269,  -46.52031544,
         48.20370526, -495.50373377,  -46.06491435,  158.18888447,
        -38.63257523, -468.75193706,  595.48427957,  143.00628086,
        -89.44056606,  309.29119287,  -67.57691129,  -46.06491435,
        489.27479304,  -16.45446332, -227.03765548,  211.75459927,
         15.28305621, -964.25567083,  407.4039665 ,  -38.63257523,
        595.48427957])
```

In [10]:

```
feature_importance_df = pd.DataFrame({
    'Feature' : train_x.columns,
    'Importance': lr.coef_
})

feature_importance_df['Importance_magnitude'] = feature_importance_df.Importance.abs()

feature_importance_df.sort_values(by='Importance_magnitude', inplace=True, ascending=False)
# sorting so we can plot only top n important features for better visibility

feature_importance_df.head()
```

Out[10]:

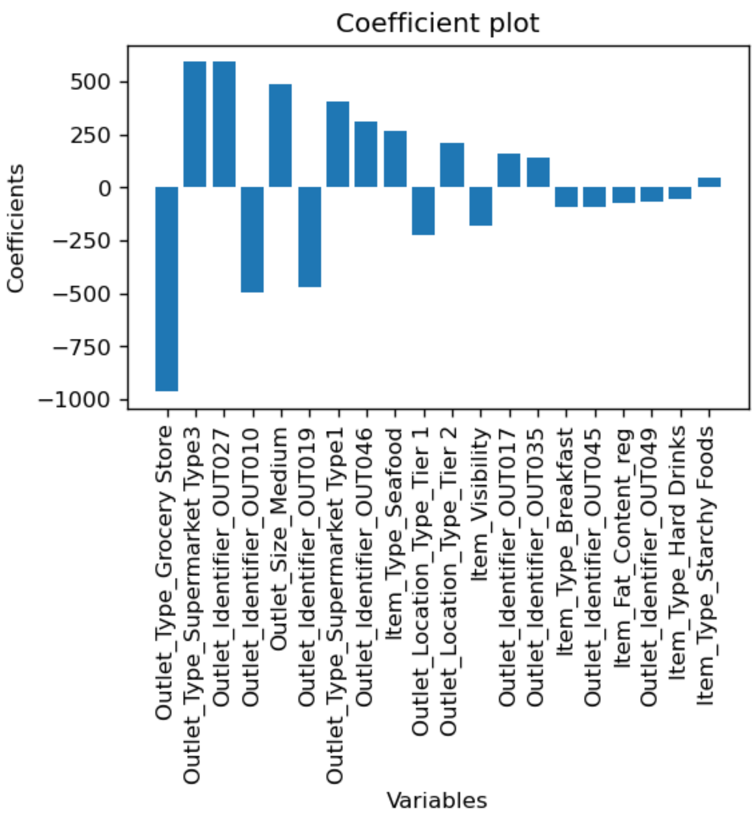
	Feature	Importance	Importance_magnitude
41	Outlet_Type_Grocery Store	-964.255671	964.255671
44	Outlet_Type_Supermarket Type3	595.484280	595.484280
30	Outlet_Identifier_OUT027	595.484280	595.484280
25	Outlet_Identifier_OUT010	-495.503734	495.503734
36	Outlet_Size_Medium	489.274793	489.274793

In [11]:

```
# plotting the coefficients


plt.figure(figsize=(5,3), dpi=120, facecolor='w', edgecolor='b')
x = feature_importance_df['Feature'].head(20)
y = feature_importance_df['Importance'].head(20)
plt.bar( x, y )
plt.xlabel( "Variables")
plt.ylabel('Coefficients')
plt.xticks(rotation=90)
plt.title('Coefficient plot')
```

Out[11]:

```
Text(0.5, 1.0, 'Coefficient plot')
```



> Note : Here we can see that the model depends upon some Independent variables toos much, But these coefficients are not suitable for interpretation because these are not scaled, we will look into this later on in details.

Validiating Assumptions of Linear Regression

In [12]:

```
# Calculating Error Terms

error_df = pd.DataFrame({
    'original_values' : test_y,
    'predicted_values': test_pred
})

error_df['error'] = error_df['original_values'] - error_df['predicted_values']

error_df.head()
```

Out[12]:

	original_values	predicted_values	error
2344	1095.2410	3460.817769	-2365.576769
4005	5070.7328	3000.322073	2070.410727
2897	892.1720	712.864464	179.307536
6252	2976.1260	3200.235119	-224.109119
6414	365.5242	1956.201769	-1590.677569

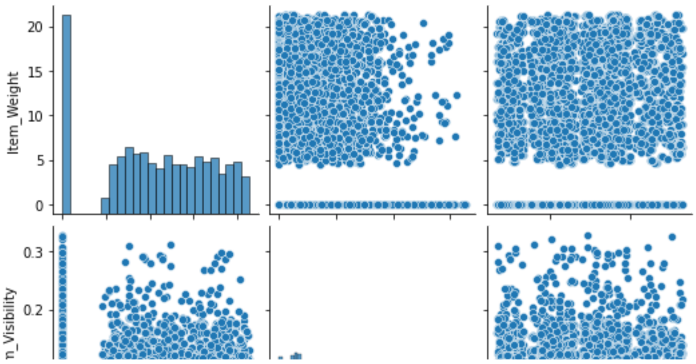
1. Linearity

In [13]:

```python
import seaborn as sns

sns.pairplot(df[['Item_Weight', 'Item_Visibility', 'Item_MRP']])
```

Out[13]:

```
<seaborn.axisgrid.PairGrid at 0x27309744520>
```

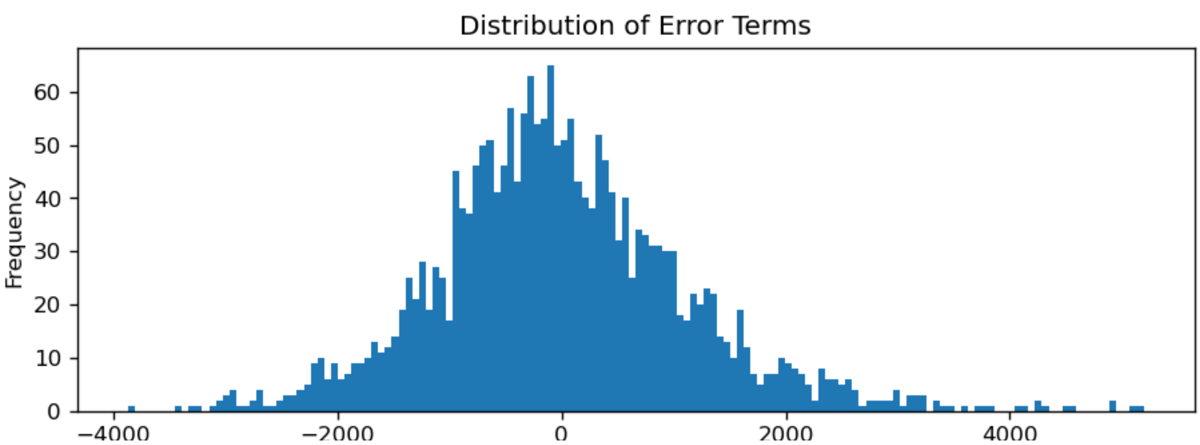


Since there are lot of features and feature selection is not in the scope of this notebook, displayed pairplot only between some columns.

2. Normally Distributed Error Terms

In [14]:

```python
# Histogram for distribution of error terms
plt.figure(figsize=(9,3), dpi=120, facecolor='w', edgecolor='b')
plt.hist(error_df.error, bins = 150)
plt.xlabel('Error')
plt.ylabel('Frequency')
plt.title('Distribution of Error Terms')
plt.show()
```



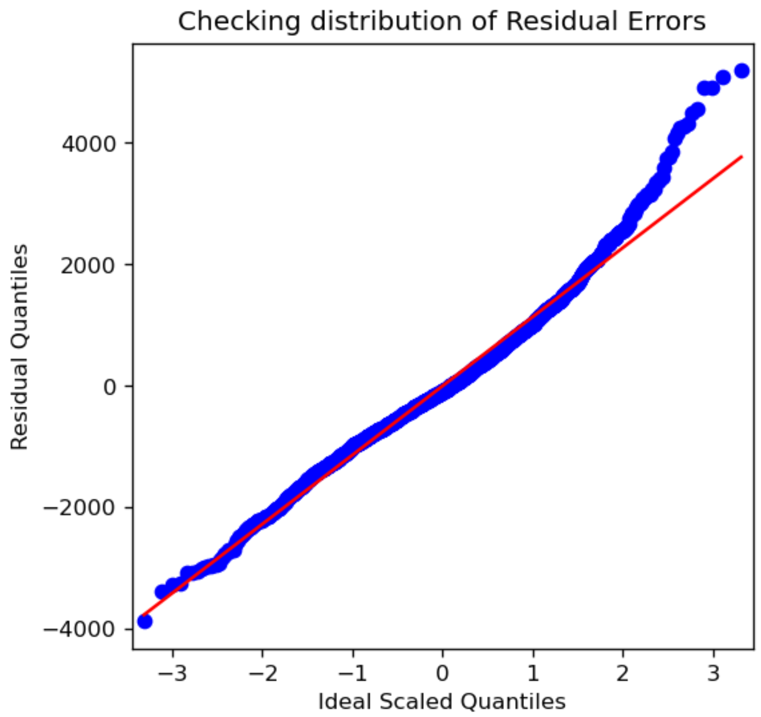
> According to the Histogram, the distribution of error is nearly normal, But there are some outliers on the Higher end of the errors which could be handled easily.

Another way to check normal distribution is Q-Q Plot

In [15]:

```
# importing the QQ-plot from the from the statsmodels
import numpy as np
from statsmodels.graphics.gofplots import qqplot

## Plotting the QQ plot
fig, ax = plt.subplots(figsize=(5,5) , dpi = 120)
qqplot(error_df.error, line = 's' , ax = ax)
plt.ylabel('Residual Quantiles')
plt.xlabel('Ideal Scaled Quantiles')
plt.title('Checking distribution of Residual Errors')
plt.show()
```

```
C:\Users\rkt7k\anaconda3\lib\site-packages\statsmodels\graphics\gofplots.py:993: UserWarning: marker is redundantly defined
by the 'marker' keyword argument and the fmt string "bo" (-> marker='o'). The keyword argument will take precedence.
  ax.plot(x, y, fmt, **plot_style)
```



The QQ-plot clearly verifies our findings from the the histogram of the residuals, the data is mostly normal in nature, but there sre some outliers on the higher end of the Residues.

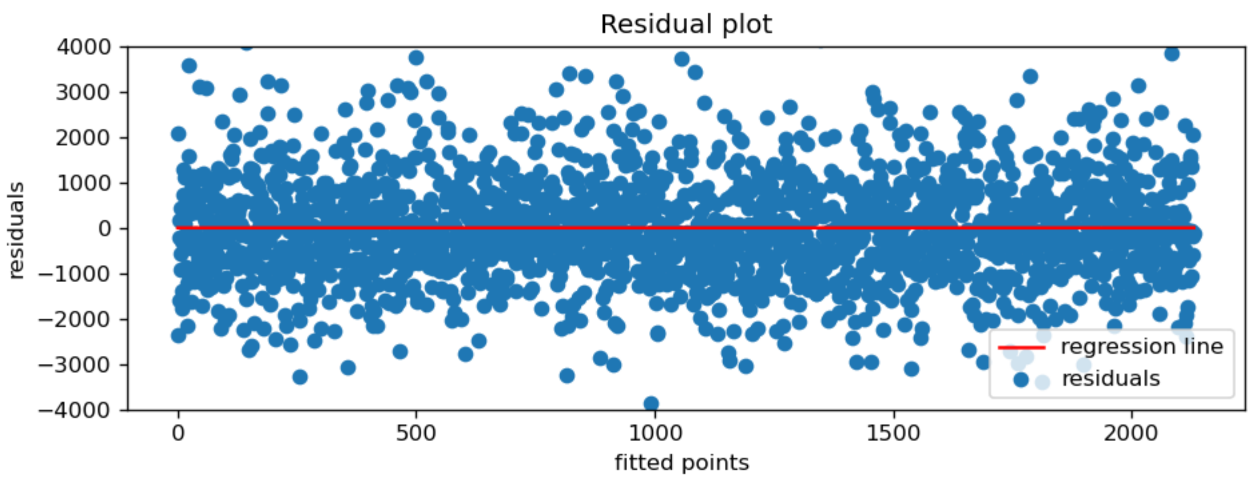
3. Homoscedasticity : Constant Vairance of Error Terms

In [16]:

```
plt.figure(figsize=(9,3), dpi=120, facecolor='w', edgecolor='b')
f = range(0,2131)
k = [0 for i in range(0,2131)]
plt.scatter( f, error_df.error, label = 'residuals')
plt.plot( f, k , color = 'red', label = 'regression line' )
plt.xlabel('fitted points ')
plt.ylabel('residuals')
plt.title('Residual plot')
plt.ylim(-4000, 4000)
plt.legend()
```

Out[16]:

```
<matplotlib.legend.Legend at 0x27311eee430>
```



The Residual plot clearly Looks Homoscedastic, i.e. the the variance of the error across the dataset is nearly constant.

4. Variance Inflation Factor (VIF) (Checking for multi collinearity)

In [17]:

```
# Importing Variance_inflation_Factor funtion from the Statsmodels
from statsmodels.stats.outliers_influence import variance_inflation_factor
from statsmodels.tools.tools import add_constant

```

```
# note : VIF by default does not calculate for the constant b in the linear reg equation : y = Bx=b
         so we add it separately

Also, thie new constang is TREATED AS COEFFICIENT OF A NEW FEATURE WHOSE VALUE IS 1 , so there is no effect of its value on data
and we can get a constant as well

=> y = Bx + 1b
```

In [18]:

```
trainx_with_constant = add_constant(train_x.values)
trainx_with_constant[1]
# we can observe, one has beed added as the first value
```

Out[18]:

```
array([1.0000000e+00, 1.6000000e+01, 7.2655379e-02, 2.2986680e+02,
       2.0040000e+03, 0.0000000e+00, 1.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
       1.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 1.0000000e+00, 0.0000000e+00,
       1.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
       0.0000000e+00, 0.0000000e+00])
```

In [19]:

```
# Calculating VIF for every column (only works for the not Catagorical)
VIF= pd.Series([variance_inflation_factor(trainx_with_constant, i) for i in range(1,trainx_with_constant.shape[1])],
                index =train_x.columns)
VIF
```

```
C:\Users\rkt7k\anaconda3\lib\site-packages\statsmodels\stats\outliers_influence.py:193: RuntimeWarning: divide by zero enco
untered in double_scalars
  vif = 1. / (1. - r_squared_i)
```

Out[19]:

```
Item_Weight                        2.335412
Item_Visibility                    1.099468
Item_MRP                           1.015434
Outlet_Establishment_Year               inf
Item_Fat_Content_LF                     inf
Item_Fat_Content_Low Fat                inf
Item_Fat_Content_Regular                inf
Item_Fat_Content_low fat                inf
Item_Fat_Content_reg                    inf
Item_Type_Baking Goods                  inf
Item_Type_Breads                        inf
Item_Type_Breakfast                     inf
Item_Type_Canned                        inf
Item_Type_Dairy                         inf
Item_Type_Frozen Foods                  inf
Item_Type_Fruits and Vegetables         inf
Item_Type_Hard Drinks                   inf
Item_Type_Health and Hygiene            inf
Item_Type_Household                     inf
Item_Type_Meat                          inf
Item_Type_Others                        inf
Item_Type_Seafood                       inf
Item_Type_Snack Foods                   inf
Item_Type_Soft Drinks                   inf
Item_Type_Starchy Foods                 inf
Outlet_Identifier_OUT010                inf
Outlet_Identifier_OUT013                inf
Outlet_Identifier_OUT017                inf
Outlet_Identifier_OUT018                inf
Outlet_Identifier_OUT019                inf
Outlet_Identifier_OUT027                inf
Outlet_Identifier_OUT035                inf
Outlet_Identifier_OUT045                inf
Outlet_Identifier_OUT046                inf
Outlet_Identifier_OUT049                inf
Outlet_Size_High                        inf
Outlet_Size_Medium                      inf
Outlet_Size_Small                       inf
Outlet_Location_Type_Tier 1             inf
Outlet_Location_Type_Tier 2             inf
Outlet_Location_Type_Tier 3             inf
Outlet_Type_Grocery Store               inf
Outlet_Type_Supermarket Type1           inf
Outlet_Type_Supermarket Type2           inf
Outlet_Type_Supermarket Type3           inf
dtype: float64
```

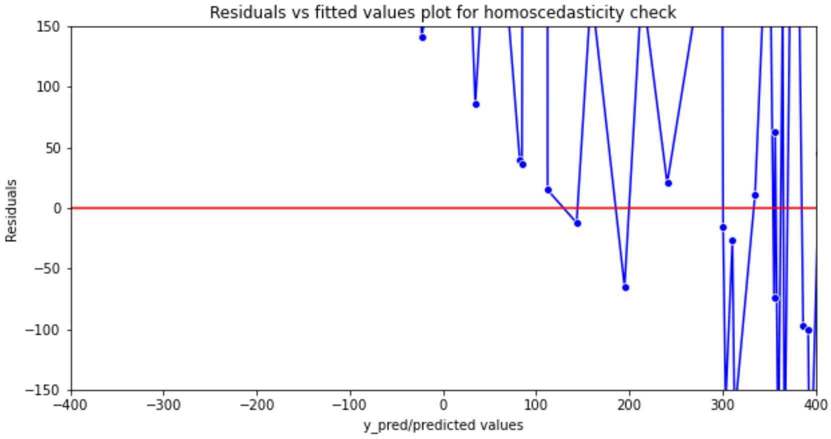
```
note :
vif = 1  /  (1-R2)
so if R2 value is coming 1, the vif will be infinity

but why VIF here is infinity,  if we see closely, INF is only for the ONE HOT ENCODED variables
so what happens is, when we encode all n categories, then if (n-1) categories is 0, it means that that last nth category is 1. so actually we dont need to encode all n categories and rather encode only (n-1) categories.

Here as all n categories are encoded, the (n-1) features could be easily used to predict that last nth feature and hence R2 for that is 1
```

5. Auto Correlation

In [25]:

```
import seaborn as sns
plt.figure(figsize=(10,5))
p = sns.lineplot(test_pred,error_df.error,marker='o',color='blue')
plt.xlabel('y_pred/predicted values')
plt.ylabel('Residuals')
plt.ylim(-150,150)
plt.xlim(-400,400)
p = sns.lineplot([-400,400],[0,0],color='red')
p = plt.title('Residuals vs fitted values plot for homoscedasticity check')
```

```
C:\Users\rkt7k\anaconda3\lib\site-packages\seaborn\_decorators.py:36: FutureWarning: Pass the following variables as keywor
d args: x, y. From version 0.12, the only valid positional argument will be `data`, and passing other arguments without an
explicit keyword will result in an error or misinterpretation.
  warnings.warn(
```



Model Inpretability

So far we have simply been predicting the values using the linear regression, But in order to Interpret the model, the normalising of the data is essential.

In [ ]:

```
lr = LR(normalize=True)
lr.fit(train_x, train_y)
```

In [ ]:

```
train_pred = lr.predict(train_x)
train_mae = mae(train_pred, train_y)
print('Training Mean Absolute Error :', train_mae)
```

In [ ]:

```
test_pred = lr.predict(test_x)
test_mae = mae(test_pred, test_y)
print('Training Mean Absolute Error :', test_mae)
```

In [ ]:

```
feature_importance_df = pd.DataFrame({
    'Feature' : train_x.columns,
    'Importance': lr.coef_
})

feature_importance_df['Importance_magnitude'] = feature_importance_df.Importance.abs()

feature_importance_df.sort_values(by='Importance_magnitude', inplace=True, ascending=False)
# sorting so we can plot only top n important features for better visibility

feature_importance_df.head()
```

In [ ]:

```
# plotting the coefficients


plt.figure(figsize=(5,3), dpi=120, facecolor='w', edgecolor='b')
x = feature_importance_df['Feature'].head(20)
y = feature_importance_df['Importance'].head(20)
plt.bar( x, y )
plt.xlabel( "Variables")
plt.ylabel('Coefficients')
plt.xticks(rotation=90)
plt.title('Coefficient plot')
```

Now the coefficients we see are normalised and we can easily make final inferences out of it.

In [ ]:

```

```