FUTURE SALES PREDICTION

PHASE 4

INTRODUCTION :

The IMDs score prediction model by:

1.Feature engineering

2. Model training

3. Evaluation

FUTURE SALES PREDICTION

1.Feature Engineering:

Feature engineering for sales prediction involves creating new variables or modifying existing ones to improve the accuracy of your sales forecasting model. Here are some common feature engineering techniques for sales prediction:

1. **Time-based Features:** Incorporate time-related features such as day of the week, month, and year. You can also create lag features to capture historical sales trends.

2. **Holiday and Special Event Indicators:** Add binary variables to indicate whether a specific date is a holiday or corresponds to a special event that might affect sales.

3. **Seasonal Trends:** Create features that capture seasonal patterns, like summer or winter, which can influence sales.

4. **Weather Data:** If applicable, include weather-related features. For instance, temperature, precipitation, or humidity can impact sales of certain products.

5. **Economic Indicators:** Incorporate relevant economic data such as inflation rates, GDP, or consumer confidence indices if they might impact sales.

6. **Promotions and Marketing Efforts:** Create variables to track the timing and impact of promotions, advertising campaigns, or discounts.

7. **Customer Demographics:** If you have customer data, consider including demographic information that might correlate with sales trends.

8. **Competitor Data:** Information about competitors, such as their pricing or product launches, can be useful features.

9. **Inventory Levels:** If you're predicting sales for a product, knowing its inventory level can be crucial in understanding sales fluctuations.

10. **Time Since Last Purchase:** For repeat sales, feature engineering could include the time since the last purchase for each customer.

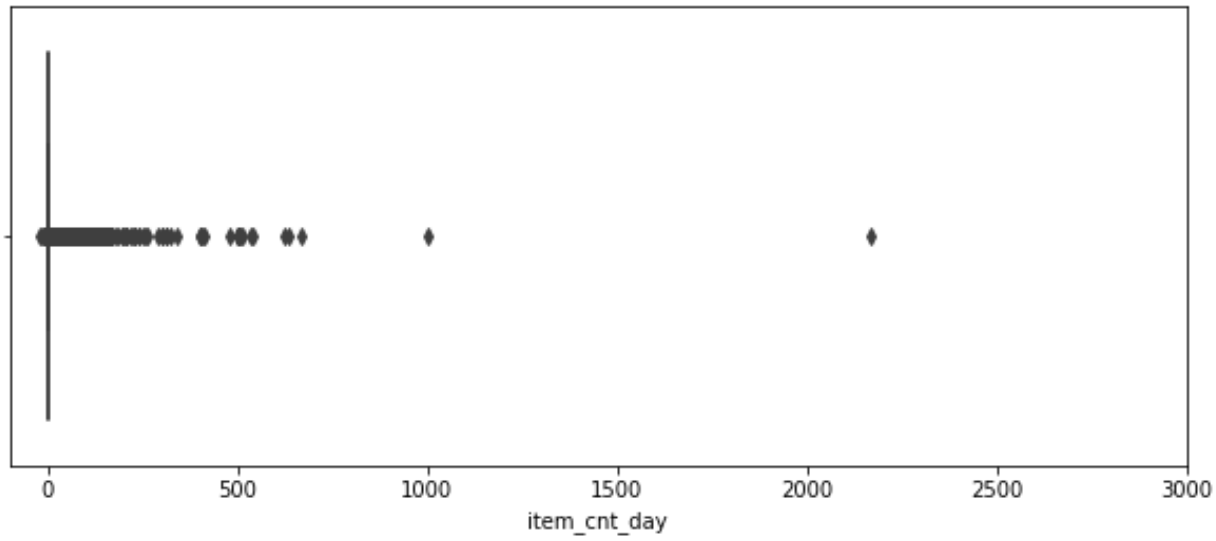
```
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error
from calendar import monthrange
from itertools import product

import numpy as np
# linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. sales.csv)
import lightgbm as lgb
import seaborn as sns
import matplotlib.pyplot as plt
import pickle


%matplotlib inline

shops = sales._csv('/salesinput/competitive-data-science-predict-future-
sales/shops.csv')
items = sales._csv('¥salesinput/competitive-data-science-predict-future-
sales/shops.csv')
-data-science-predict-future-sales/items.csv')
catgs = sales.read_csv('/ salesinput/competitive-data-science-predict-future-
sales/shops.csv')
 -data-science-predict-future-sales/item_categories.csv')
sales = sales.csv('/ salesinput/competitive-data-science-predict-future-
sales/shops.csv')
testd = sales._csv('/ salesinput/competitive-data-science-predict-future-
sales/shops.csv')
sampl = sales._csv('/ salesinput/competitive-data-science-predict-future-
sales/shops.csv')


print(sales.isna().sum(), '¥n')
print(testd.isna().sum())


date             0
date_block_num    0
shop_id           0
item_id           0
item_price        0
item_cnt_day      0
dtype: int64
```

```
ID         0
shop_id    0
item_id    0
dtype: int64

plt.figure(figsize=(10,4))
plt.xlim(-100, 3000)aqqqqqqqqqqqqqqqqqqqqqqqqq
sns.boxplot(x=sales.item_cnt_day)

print('TV.format(tv,radio,Newspaper,salesitem_cnt_day.min(),
sales.item_cnt_day.max()))

plt.figure(figsize=(10,4))
plt.xlim(sales.item_price.min(), sales.item_price.max()*1.1)
sns.boxplot(x=sales.item_price)

print('RADIO.format(sales.item_price., sales.item_price.))
```

OUTPUT:

```
item count day - Min: -22.0, Max: 2169.0
Item price - Min: -1.0, Max: 307980.0
```

Tv	radio	sales
230.1	37.8	22.1
44.5	39.3	10.4
17.2	45.9	12
151.5	41.3	16.5
180.8	10.8	17.9



2.Model training

Model t raining is the key step in machine learning that results in a model ready to be validated, tested, and deployed. The performance of the model determines the quality of the applications that are built using it. Quality of training data and the training algorithm are both important assets during the model training phase. Typically, training data is split for training, validation and testing. The training algorithm is chosen based on the end use case. There are a number of tradeoff points in deciding the best algorithm–model complexity, interpretability, performance, compute requirements, etc. All these aspects of model training make it both an involved and important process in the overall machine learning development cycle.

Linear regression:

Training a model like linear regression typically involves using a dataset to find the best-fit line that minimizes the error between the predicted values and the actual values. Here's a simplified example in Python using the scikit-learn library:

```python
import numpy as np
from sklearn.linear_model import LinearRegression

# Sample dataset
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)    # Input features
y = np.array([2, 4, 5, 4, 5])    # Target values

# Create and train the linear regression model
model = LinearRegression()
model.fit(X, y)

# Make predictions
predictions = model.predict(X)

# Print the model parameters
```

```
print("Coefficients:", model.coef_)  # Slope of the line
print("Intercept:", model.intercept_)  # Intercept of the line
```

This code creates a linear regression model, fits it to the dataset, and makes predictions.

Random forest:

Training a model like a Random Forest involves using a dataset to build an ensemble of decision trees. Here's a simplified example in Python using the scikit-learn library:

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# Generate a sample dataset (you can replace this with your own data)
X, y = make_regression(n_samples=100, n_features=1, noise=0.2, random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train a Random Forest Regressor
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = rf_model.predict(X_test)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
```

print("Mean Squared Error:", mse)

```

In this example, we generate a synthetic dataset, split it into training and testing sets, create a Random Forest Regressor, and make predictions.

Decision tree:

Training a Decision Tree model involves using a dataset to create a tree-like structure of decisions to make predictions. Here's a simplified example in Python using the scikit-learn library:

```python

from sklearn.tree import DecisionTreeRegressor

from sklearn.datasets import make_regression

from sklearn.model_selection import train_test_split

from sklearn.metrics import mean_squared_error

# Generate a sample dataset (you can replace this with your own data)

X, y = make_regression(n_samples=100, n_features=1, noise=0.2, random_state=42)

# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train a Decision Tree Regressor

tree_model = DecisionTreeRegressor(random_state=42)

tree_model.fit(X_train, y_train)

# Make predictions on the test data

y_pred = tree_model.predict(X_test)

# Evaluate the model

mse = mean_squared_error(y_test, y_pred)

print("Mean Squared Error:", mse)

```

In this example, we generate a synthetic dataset, split it into training and testing sets, create a Decision Tree Regressor, and make predictions.

Performing Simple Linear Regression

Equation of linear regression

$y=c+m_1x_1+m_2x_2+...+m_nX_ny=c+m1x1+m2x2+...+mnxn$

- yy is the response
- cc is the intercept
- m_1m1 is the coefficient for the first feature
- $m_n$mn is the coefficient for the nth feature

In our case:

$y=c+m_1\times TV$y=c+m1×TV

The mm values are called the model **coefficients** or **model parameters**.

Generic Steps in model building using statsmodels

We first assign the feature variable, TV, in this case, to the variable X and the response variable, Sales, to the variable y.

In [12]:

```
X = advertising['TV']
y = advertising['Sales']
```

Train-Test Split

You now need to split our variable into training and testing sets. You'll perform this by importing train_test_split from the sklearn.model_selection library. It is usually a good practice to keep 70% of the data in your train dataset and the rest 30% in your test dataset

In [13]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size = 0.7, test_size = 0.3, random_state = 100)
```

In [14]:

```
# Let's now take a look at the train dataset

X_train.head()
```

Out[14]:

```
74     213.4
3      151.5
```
```

```
185    205.0
26     142.9
90     134.3
Name: TV, dtype: float64
```

In [15]:

```
y_train.head()
```

Out[15]:

```
74     17.0
3      16.5
185    22.6
26     15.0
90     14.0
Name: Sales, dtype: float64
```

Building a Linear Model

You first need to import the statsmodel.api library using which you'll perform the linear regression.

In [16]:

```
import statsmodels.api as sm
```

By default, the statsmodels library fits a line on the dataset which passes through the origin. But in order to have an intercept, you need to manually use the add_constant attribute of statsmodels. And once you've added the constant to your X_train dataset, you can go ahead and fit a regression line using the OLS (Ordinary Least Squares) attribute of statsmodels as shown below

In [17]:

```
# Add a constant to get an intercept
X_train_sm = sm.add_constant(X_train)

# Fit the resgression line using 'OLS'
lr = sm.OLS(y_train, X_train_sm).fit()
```

In [18]:

```
# Print the parameters, i.e. the intercept and the slope of the regression line fitted
lr.params
```

Out[18]:

```
const    6.948683
TV       0.054546
dtype: float64
```

In [19]:

```
# Performing a summary operation lists out all the different parameters of the regression line fitted
print(lr.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  Sales   R-squared:                       0.816
Model:                            OLS   Adj. R-squared:                  0.814
Method:                 Least Squares   F-statistic:                     611.2
Date:                Thu, 07 Mar 2019   Prob (F-statistic):           1.52e-52
```

Time:	06:21:53	Log-Likelihood:	-321.12
No. Observations:	140	AIC:	646.2
Df Residuals:	138	BIC:	652.1
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	6.9487	0.385	18.068	0.000	6.188	7.709
TV	0.0545	0.002	24.722	0.000	0.050	0.059

Omnibus:	0.027	Durbin-Watson:	2.196
Prob(Omnibus):	0.987	Jarque-Bera (JB):	0.150
Skew:	-0.006	Prob(JB):	0.928
Kurtosis:	2.840	Cond. No.	328.

3.Evaluation:

Model evaluation is the process that uses some metrics which help us to analyze the performance of the model. As we all know that model development is a multi-step process and a check should be kept on how well the model generalizes future predictions. Therefore evaluating a model plays a vital role so that we can judge the performance of our model. The evaluation also helps to analyze a model's key weaknesses. There are many metrics like Accuracy, Precision, Recall, F1 score, Area under Curve, Confusion Matrix, and Mean Square Error. Cross Validation is one technique that is followed during the training phase and it is a model evaluation technique as well.

Certainly! Here's a basic example of how to perform model evaluation using Python and the scikit-learn library. We'll use a simple classification model for illustration:

```python
# Import necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

# Assuming you have a dataset with features X and labels y
# Split the dataset into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train a model (Logistic Regression in this case)
model = LogisticRegression()
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy: ", accuracy)
print("Precision: ", precision)
print("Recall: ", recall)
print("F1 Score: ", f1)
print("Confusion Matrix:¥n", conf_matrix)
```

In this example, we use the `LogisticRegression` model, split the data into training and testing sets, make predictions on the test set, and calculate common classification metrics such as accuracy, precision, recall, F1 score, and the confusion matrix. You can replace `LogisticRegression` with any other machine learning algorithm that suits your problem.

Model Evaluation

Residual analysis

To validate assumptions of the model, and hence the reliability for inference

Distribution of the error terms

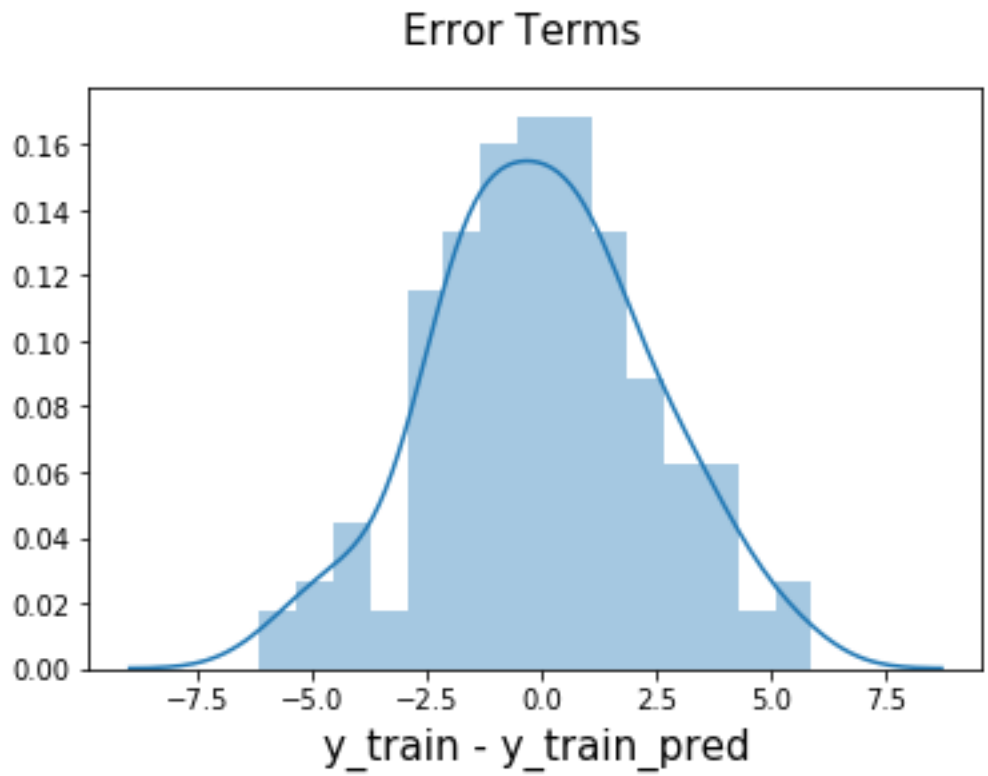
We need to check if the error terms are also normally distributed (which is infact, one of the major assumptions of linear regression), let us plot the histogram of the error terms and see what it looks like.

In [21]:

```
y_train_pred = lr.predict(X_train_sm)
res = (y_train - y_train_pred)
```

In [22]:

```
fig = plt.figure()
sns.distplot(res, bins = 15)
fig.suptitle('Error Terms', fontsize = 15)                    # Plot heading
plt.xlabel('y_train - y_train_pred', fontsize = 15)           # X-label
plt.show()
```



The residuals are following the normally distributed with a mean 0. All good!

Looking for patterns in the residuals

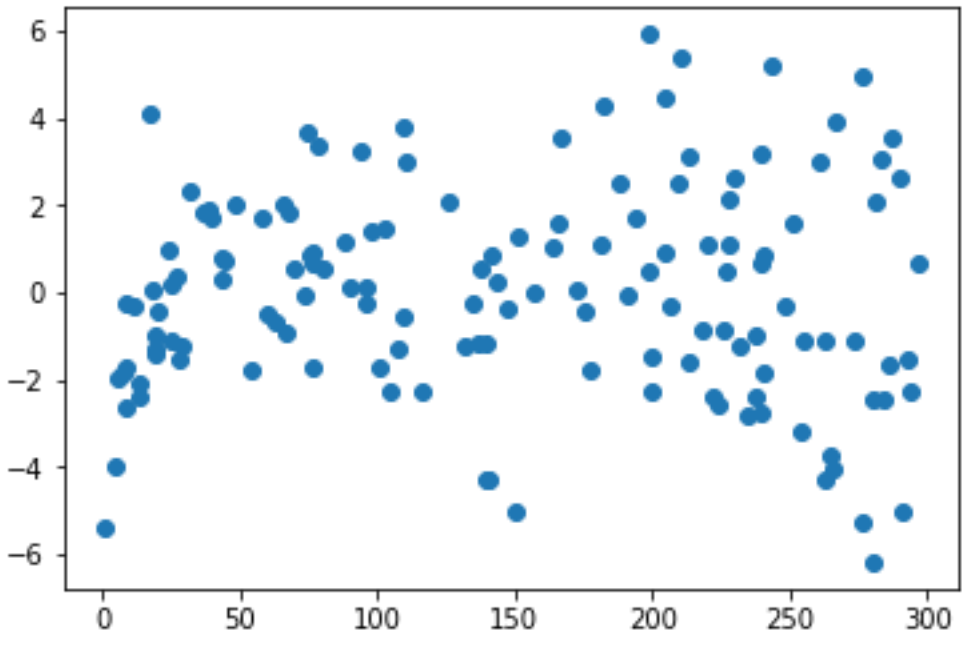
In [23]:

```
plt.scatter(X_train,res)
plt.show()
```

We are confident that the model fit isn't by chance, and has decent predictive power. The normality of residual terms allows some inference on the coefficients.

Although, the variance of residuals increasing with X indicates that there is significant variation that this model is unable to explain.

As you can see, the regression line is a pretty good fit to the data



Predictions on the Test Set

Now that you have fitted a regression line on your train dataset, it's time to make some predictions on the test data. For this, you first need to add a constant to the X_test data like you did for X_train and then you can simply go on and predict the y values corresponding to X_test using the predict attribute of the fitted regression line.

In [24]:

```
# Add a constant to X_test
X_test_sm = sm.add_constant(X_test)
```

```
# Predict the y values corresponding to X_test_sm
y_pred = lr.predict(X_test_sm)
```

In [25]:

```
y_pred.head()
```

Out[25]:

```
126      7.374140
104     19.941482
99      14.323269
92      18.823294
111     20.132392
dtype: float64
```

In [26]:

```
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
```

Looking at the RMSE

In [27]:

```
#Returns the mean squared error; we'll take a square root
np.sqrt(mean_squared_error(y_test, y_pred))
```

Out[27]:

```
2.019296008966232
```

Checking the R-squared on the test set

In [28]:

```
r_squared = r2_score(y_test, y_pred)
r_squared
```

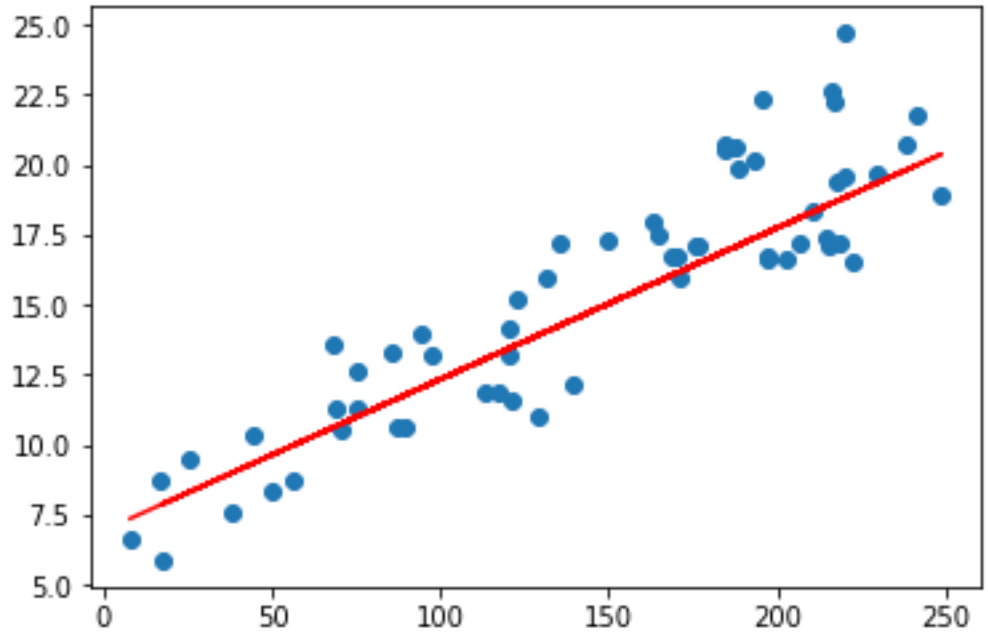
Out[28]:

```
0.792103160124566
```

Visualizing the fit on the test set

In [29]:

```
plt.scatter(X_test, y_test)
plt.plot(X_test, 6.948 + 0.054 * X_test, 'r')
plt.show()
```


25.0
22.5
20.0
17.5
15.0
12.5
10.0
7.5
5.0
0
50
100
150
200
250