INTRODUCTION:

Purpose:

Predicting product demand is essential for businesses to efficiently manage resources, meet consumer needs, reduce excess inventory, and enhance overall operational efficiency. Machine learning techniques provide a sophisticated way to analyze complex datasets and identify patterns that influence consumer behavior and demand fluctuations.

Process Overview:

The process involves several key stages:

1. Data Collection:

Gathering historical data, which may include sales records, customer profiles, market trends, economic indicators, promotional activities, and any other relevant information that could impact demand.

2. Data Preprocessing:

Cleaning, formatting, and organizing the collected data to make it suitable for analysis. This involves handling missing values, removing outliers, and converting data into a format that machine learning algorithms can process.

3. Feature Engineering:

Creating meaningful features or variables from the data that might influence demand, such as seasonality, trends, customer behavior, and external factors. Feature engineering is crucial to the model's ability to learn and predict accurately.

4. Model Selection:

Choosing appropriate machine learning models suited to the nature of the data and the specific demand forecasting problem. Common models include linear regression, time series models (like ARIMA or SARIMA), decision trees, random forests, gradient boosting, and neural networks.

5. Model Training:

Utilizing historical data, the chosen model is trained to learn patterns and relationships between different variables and the demand for the product.

6. Model Evaluation:

Assessing the model's performance using metrics such as mean absolute error (MAE), mean squared error (MSE), root mean squared error (RMSE), or others. This step determines the model's accuracy and effectiveness.

7. Hyperparameter Tuning:

Optimizing the model's parameters to improve its performance and accuracy. This step involves adjusting settings that are external to the model and impact its learning process.

8. Forecasting and Prediction:

Applying the trained model to new data inputs to forecast future demand.

9. Deployment and Monitoring:

Implementing the model within business operations for real-time predictions. Continual monitoring and updates are vital to ensure the model remains accurate as demand patterns evolve due to changing market conditions.

10. Decision Making:

Using the predicted demand to make informed decisions regarding inventory management, production planning, pricing strategies, and overall business operations.

DESIGN INTO INNOVATION:

CONTENT FOR INNOVATION:

Consider incorporating time series forecasting techniques like ARIMA or Prophet to capture temporal patterns in demand data.

EXPLANATION:

Data Collection and Preprocessing:

Gather historical demand data, ensuring that it is time-stamped and organized chronologically. Preprocess the data by addressing missing values, outliers, and any other data quality issues.

Exploratory Data Analysis(EDA):

Conduct EDA to understand the temporal patterns and characteristics of the demand data. Look for seasonality, trends, and other recurring patterns. Visualization tools and statistical tests can be helpful in this phase.

Incorporating time series forecasting techniques:

ARIMA (Auto Regressive Integrated Moving Average):

Suitable for stationary data with autoregressive and moving average components.

SARIMA (Seasonal ARIMA):

Extends ARIMA to handle seasonal patterns in data.

Exponential Smoothing Methods:

These include Holt-Winters for capturing trends and seasonality.

Prophet:

Developed by Facebook, Prophet is useful for data with daily observations, holidays, and seasonality.

 Deep Learning Models (e.g., LSTM and GRU):

Suitable for capturing complex temporal patterns, but they may require more data and computational resources.

Model Training:

Train the selected time seriesforecasting model using historical demand data. This involves estimating model parameters and seasonal components, if applicable.

Validation and Hyperparameter Tuning:

Assess the model's performance using validation data or cross-validation. Fine-tune hyperparameters and adjust the model structure as needed to improve forecasting accuracy.

Forecasting:

Once the model is trained and validated, use it to make predictions for future time periods. These forecasts will capture temporal patterns and provide insights into expected demand behavior.

Performance Evaluation:

Evaluate the forecasting model's performance using appropriate metrics like Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and forecast accuracy measures.

Continuous Monitoring and Updating:

Implement a process for regularly updating and retraining the model as new demand data becomes available. This ensures that the model adapts to changing demand patterns over time.

Incorporate External Factors:

Consider adding external variables such as promotional activities, economic indicators, or weather data to your model to account for factors that influence demand fluctuations.

PROGRAM:

```
import pandas as pd import numpy as np

import plotly.express as px

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeRegressor

data=pd.read_csv("C:\Users\mabir\AppData\Local\Microsoft\Windo

ws\INetCache\IE\AHLGJQP8\archive[1].zip ")

data.head()
```

Output:

ID	Store ID	Total Price	Base Price	Units Sold
1	8091	99.0375	111.8625	20
2	8091	99.0375	99.0375	28
3	8091	133.95	133.95	19
4	8091	133.95	133.95	44
5	8091	141.075	141.075	52
9	8091	227.2875	227.2875	18
10	8091	327.0375	327.0375	47
13	8091	210.9	210.9	50
14	8091	190.2375	234.4125	82
17	8095	99.0375	99.0375	99
18	8095	97.6125	97.6125	120
19	8095	98.325	98.325	40
22	8095	133.2375	133.2375	68
23	8095	133.95	133.95	87
24	8095	139.65	139.65	186
27	8095	236.55	280.0125	54
28	8095	214.4625	214.4625	74
29	8095	266.475	296.4	102
30	8095	173.85	192.375	214
31	8095	205.9125	205.9125	28
32	8095	205.9125	205.9125	7
33	8095	248.6625	248.6625	48
34	8095	200.925	200.925	78
35	8095	190.2375	240.825	57
37	8095	427.5	448.1625	50
38	8095	429.6375	458.1375	62
39	8095	177.4125	177.4125	22
42	8094	87.6375	87.6375	109
43	8094	88.35	88.35	133
44	8094	85.5	85.5	11
45	8094	128.25	180.975	9
47	8094	127.5375	127.5375	19
48	8094	123.975	123.975	33
49	8094	139.65	164.5875	49
50	8094	235.8375	235.8375	32

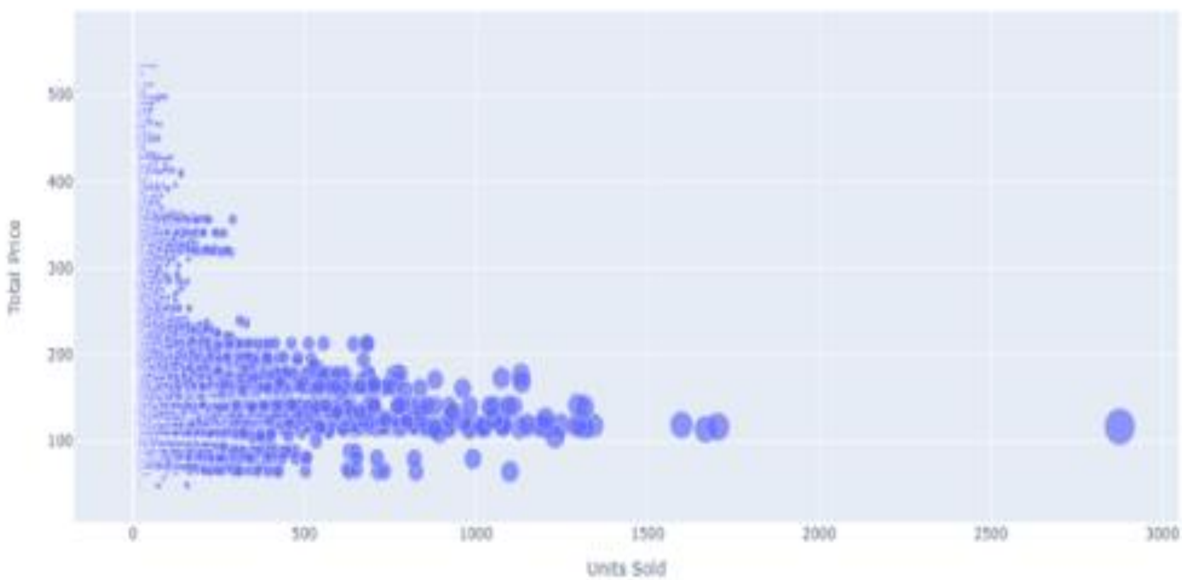
Relationship between price and demand for the product:

fig = px.scatter(data, x="Units Sold", y="Total Price",

size='Units Sold')

fig.show()

output:



Correlation between the features of the dataset:

print(data.corr())

Output:

ID Store ID Total Price Base Price Units Sold

ID 1.000000 0.007464 0.008473 0.018932 -

0.010616

Store ID 0.007464 1.000000 -0.038315 -0.038848 -

0.004372

Total Price 0.008473 -0.038315 1.000000 0.958885 -

0.235625

Base Price 0.018932 -0.038848 0.958885 1.000000 -

0.140032

Units Sold -0.010616 -0.004372 -0.235625 -0.140032

1.000000

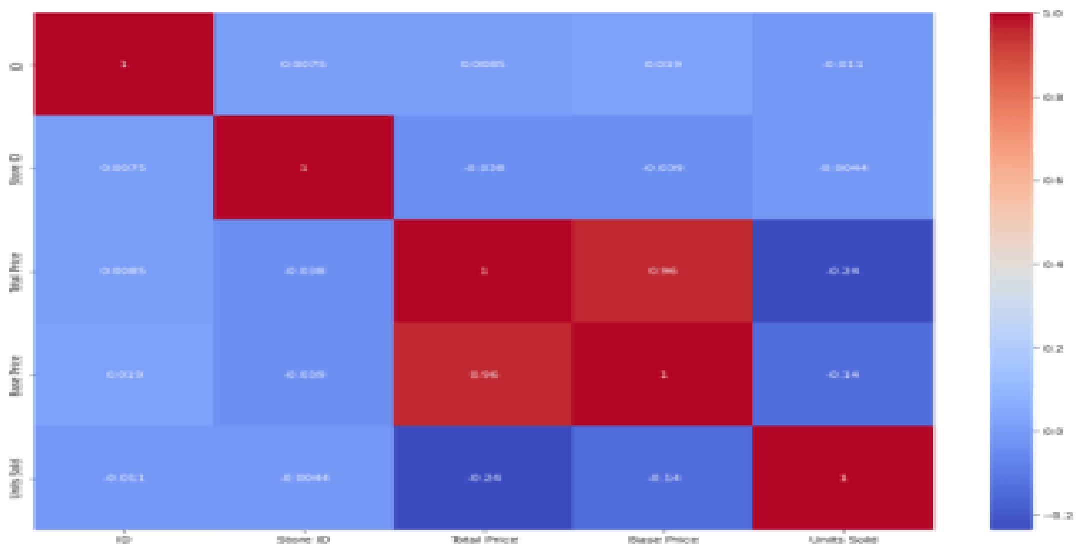
correlations = data.corr(method='pearson')

```
plt.figure(figsize=(15, 12))

sns.heatmap(correlations, cmap="coolwarm", annot=True)

plt.show()
```

Output:



```
# fit an ARIMA model and plot residual errors

from pandas import datetime

from pandas import read_csv

from pandasimport DataFrame

from statsmodels.tsa.arima.model import ARIMA
```

```
from matplotlib import pyplot

# load dataset

def parser(x):

return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, index_col=0,

parse_dates=True, squeeze=True, date_parser=parser)

series.index = series.index.to_period('M')

# fit model

model = ARIMA(series, order=(5,1,0))

model_fit = model.fit()

# summary of fit model

print(model_fit.summary())

# line plot of residuals

residuals = DataFrame(model_fit.resid)

residuals.plot()

pyplot.show()
```

```
# density plot of residuals

residuals.plot(kind='kde')

pyplot.show()

# summary stats of residuals

print(residuals.describe())
```

Output:

SARIMAX Results

==

======================

Dep. Variable: Sales No. Observations: 36

Model: ARIMA(5, 1, 0) Log Likelihood -198.485

Date: Thu, 10 Dec 2020 AIC 408.969

Time: 09:15:01 BIC 418.301

Sample: 01-31-1901 HQIC 412.191

- 12-31-1903

Covariance Type: opg

==

======================

coef std err z P>|z| [0.025 0.975]

ar.L1 -0.9014 0.247 -3.647 0.000 -1.386 -0.417

ar.L2 -0.2284 0.268 -0.851 0.395 -0.754 0.298

ar.L3 0.0747 0.291 0.256 0.798 -0.497 0.646

ar.L4 0.2519 0.340 0.742 0.458 -0.414 0.918

ar.L5 0.3344 0.210 1.593 0.111 -0.077 0.746

sigma2 4728.9608 1316.021 3.593 0.000 2149.607

7308.314

==

============================

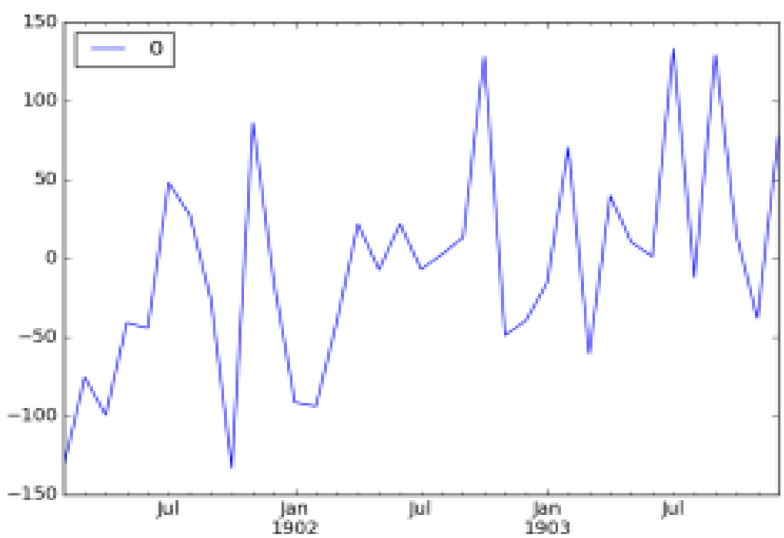
Ljung-Box (L1) (Q): 0.61 Jarque-Bera (JB): 0.96

Prob(Q): 0.44 Prob(JB): 0.62

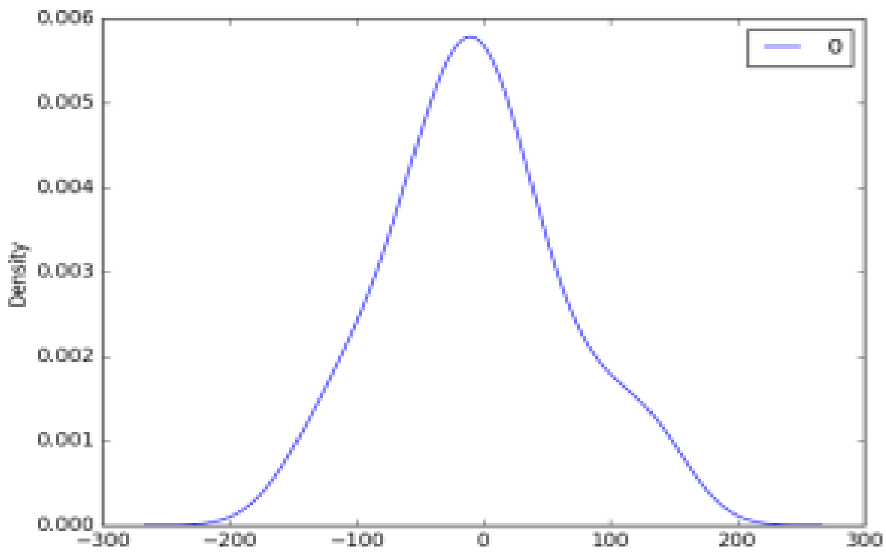
Heteroskedasticity (H): 1.07 Skew: 0.28

Prob(H) (two-sided): 0.90 Kurtosis: 2.41

First, we get a line plot of the residual errors, suggesting that there may still be some trend information not captured by the model.



Next, we get a density plot of the residual error values, suggesting the errors are Gaussian, but may not be centered on zero.



Rolling Forecast ARIMA Model:

```
# evaluate an ARIMA model using a walk-forward validationfrom pandas
import read_csv

from pandas import datetime

from matplotlib import pyplot

from statsmodels.tsa.arima.model import ARIMA

from sklearn.metrics import mean_squared_error

from math import sqrt

# load dataset
```

```
def parser(x):

return datetime.strptime('190'+x, '%Y-%m')

series = read_csv('shampoo-sales.csv', header=0, index_col=0,

parse_dates=True, squeeze=True, date_parser=parser)

series.index = series.index.to_period('M')

# split into train and test sets


X = series.values

size = int(len(X) * 0.66)

train, test = X[0:size], X[size:len(X)]

history = [x for x in train]

predictions = list()

# walk-forward validation


for t in range(len(test)):
```

```
model = ARIMA(history, order=(5,1,0))

model_fit = model.fit()

output =model_fit.forecast()

yhat = output[0]


predictions.append(yhat)

obs = test[t]

history.append(obs)

print('predicted=%f, expected=%f' % (yhat, obs))

# evaluate forecasts


rmse = sqrt(mean_squared_error(test, predictions))

print('Test RMSE: %.3f' % rmse)

# plot forecasts against actual outcomes

pyplot.plot(test)

pyplot.plot(predictions, color='red')
```

```
pyplot.show()
```

Running the example prints the prediction and expected value each iteration.

We can also calculate a final root mean squared error score (RMSE) for the predictions, providing a point of comparison for other ARIMA configurations.

```
predicted=343.272180, expected=342.300000
predicted=293.329674, expected=339.700000
predicted=368.668956, expected=440.400000
predicted=335.044741, expected=315.900000
predicted=363.220221, expected=439.300000
predicted=357.645324, expected=401.300000
predicted=443.047835, expected=437.400000
predicted=378.365674, expected=575.500000
predicted=459.415021, expected=407.600000
predicted=526.890876, expected=682.000000
```

predicted=457.231275, expected=475.300000

predicted=672.914944, expected=581.300000

predicted=531.541449, expected=646.900000

Test RMSE: 89.021

A line plot is created showing the expected values (blue) compared to the rolling forecast predictions (red). We can see the values show some trend and are in the correct scale.