Importing necessary libraries & Installing Packages

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import plotly.express as px
import plotly.graph_objects as go
import missingno as msno

from plotly.offline import init_notebook_mode, iplot
from scipy import stats
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.arima.model import ARIMA
from pandas.plotting import register_matplotlib_converters
from scipy.stats import chi2_contingency
from scipy.stats import zscore
```

Downloading necessary.csv file

- Since there are two files we are downloading them separately

```
!gdown "https://drive.google.com/drive/folders/1fBQ1PlWMho3kHF9qXrD0McZNfpJIcbrn"
```

```
/usr/local/lib/python3.11/dist-packages/gdown/parse_url.py:48: UserWarning: You specified a Google Drive link that is not the correct link to download a file. You might want t
  warnings.warn(
Downloading...
From: https://drive.google.com/drive/folders/1fBQ1PlWMho3kHF9qXrD0McZNfpJIcbrn
To: /content/1fBQ1PlWMho3kHF9qXrD0McZNfpJIcbrn
1.23MB [00:00, 21.6MB/s]
```

```
with open('/content/1fBQ1PlWMho3kHF9qXrD0McZNfpJIcbrn', 'r') as file:
    content = file.read()
```

```
import gdown

# Downloading datasets using file IDs
file_ids = [
    "1ditFn_74EOsGnblf2nXr-Vx-I1mrCG42",
    "1CclWDHsFLfCAuBbAv83_D5_XmeoRwWzl"
]

for file_id in file_ids:
    url = f"https://drive.google.com/uc?id={file_id}"
```

```
gdown.download(url, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1ditFn_74EOsGnblf2nXr-Vx-I1mrCG42
To: /content/TRAIN.csv
100%|██████████| 9.33M/9.33M [00:00<00:00, 13.4MB/s]
Downloading...
From: https://drive.google.com/uc?id=1CclWDHsFLfCAuBbAv83_D5_XmeoRwWzl
To: /content/TEST_FINAL.csv
100%|██████████| 849k/849k [00:00<00:00, 30.5MB/s]
```

```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount	#Order	Sales
0	T1000001	1	S1	L3	R1	2018-01-01	1	Yes	9	7011.84
1	T1000002	253	S4	L2	R1	2018-01-01	1	Yes	60	51789.12
2	T1000003	252	S3	L2	R1	2018-01-01	1	Yes	42	36868.20
3	T1000004	251	S2	L3	R1	2018-01-01	1	Yes	23	19715.16
4	T1000005	250	S2	L3	R4	2018-01-01	1	Yes	62	45614.52

```
df2 = pd.read_csv('TEST_FINAL.csv')
df2.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount
0	T1188341	171	S4	L2	R3	2019-06-01	0	No
1	T1188342	172	S1	L1	R1	2019-06-01	0	No
2	T1188343	173	S4	L2	R1	2019-06-01	0	No
3	T1188344	174	S1	L1	R4	2019-06-01	0	No
4	T1188345	170	S1	L1	R2	2019-06-01	0	No

Descriptive Statistics for a DataFrame

Below clearly depicts descriptive statistics for the dataframe that we are using, Some of the interpretations are,

Sales:

- The mean Sales is 42784
- The standard deviation is 18456, which shows a fairly wide variation in sales.

- The minimum sales is 0, and the maximum sales is 247215
- The 25th percentile (Q1) is 30426, the median (50th percentile) is 39678, and the 75th percentile (Q3) is 51909 which shows that half of the sales lies around 30426 & 39678.

Orders:

- The mean Order is 68
- The standard deviation is 30, which shows a quite wide variation in Orders.
- The minimum Order is 0, and the maximum Order is 371.
- The 25th percentile (Q1) is 48, the median (50th percentile) is 63, and the 75th percentile (Q3) is 82 which shows that half of the sales lies around 48 & 82

```
# Almost we have 188340 rows & 10 columns
df1.shape
```

```
(188340, 10)
```

```
Start coding or generate with AI.
```

```
#Data consist of object, float and datetime datatypes
df1.dtypes
```

	0
ID	object
Store_id	int64
Store_Type	object
Location_Type	object
Region_Code	object
Date	object
Holiday	int64
Discount	object
#Order	int64
Sales	float64

```
# Almost there are no null values
print(df1.isnull().any())
print("-----------------------------------------------------")
print(df1.isnull().sum())
```

```
ID              False
Store_id        False
Store_Type      False
Location_Type   False
Region_Code     False
Date            False
Holiday         False
Discount        False
#Order          False
Sales           False
dtype: bool
--------------------------------------------------------
ID              0
Store_id        0
Store_Type      0
Location_Type   0
Region_Code     0
Date            0
Holiday         0
Discount        0
#Order          0
Sales           0
dtype: int64
```

```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 188340 entries, 0 to 188339
Data columns (total 10 columns):
 #   Column         Non-Null Count   Dtype  
---  ------         --------------   -----  
 0   ID             188340 non-null  object 
 1   Store_id       188340 non-null  int64  
 2   Store_Type     188340 non-null  object 
 3   Location_Type  188340 non-null  object 
 4   Region_Code    188340 non-null  object 
 5   Date           188340 non-null  object 
 6   Holiday        188340 non-null  int64  
 7   Discount       188340 non-null  object 
 8   #Order         188340 non-null  int64  
 9   Sales          188340 non-null  float64
dtypes: float64(1), int64(3), object(6)
memory usage: 14.4+ MB
```

```
df = df1
df2 = df[['#Order','Sales']]
df2.describe()
```

	#Order	Sales
count	188340.000000	188340.000000
mean	68.205692	42784.327982
std	30.467415	18456.708302
min	0.000000	0.000000
25%	48.000000	30426.000000
50%	63.000000	39678.000000
75%	82.000000	51909.000000
max	371.000000	247215.000000

Block 2: EDA and Hypothesis testing

Using EDA below factors are analysed

- Gain insights into the data
- Uncover patterns
- Detect anomalies or outliers
- Test assumptions,
- Check the underlying structure of the dataset.

Univariate Analysis

As it is the first step of EDA, Basically refers to the analysis of a single variable in a dataset. It is done to do following:

- To understand the
  - Distribution
  - Central tendency
  - Spread &
  - Overall characteristics of that variable.

Sales

Central Tendency:

The mean (42784) and median (39678) are fairly close, suggesting that the data is somewhat symmetrical.

Dispersion:

The standard deviation of 18456 indicates that there is moderate variability in sales values.

Skewness:

Since the mean is slightly greater than the median, the data may be right-skewed (positively skewed), there might be few very high sales figures present in data which is pulling the distribution to the right.

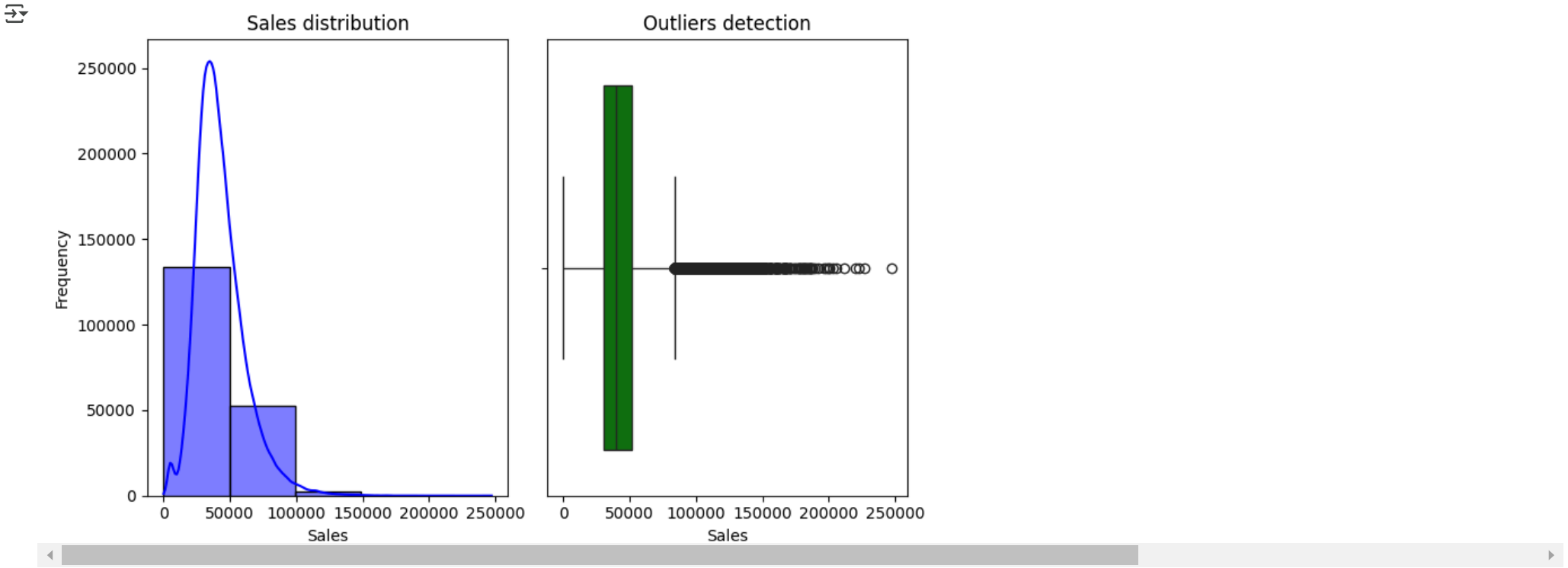
Percentiles:

The 25th percentile (Q1) is 30426, the median (50th percentile) is 39678, and the 75th percentile (Q3) is 51909 which shows that half of the sales lies around 30426 & 39678. Therefore, the majority of sales transactions are clustered around these values, with a few very low or very high sales figures outside this range.

Outliers:

Considering the range extends from 0 to 247215 sales record of 0 could be a low-performing transaction, while 247215 could be an unusually high sale.

```
# Sales distribution
plt.figure(figsize=(8, 5))

plt.subplot(121)
sns.histplot(df['Sales'], kde=True, color='blue', bins=5)
plt.title('Sales distribution')
plt.xlabel('Sales')
plt.ylabel('Frequency')
plt.tight_layout()

plt.subplot(122)
sns.boxplot(x=df['Sales'], color='green')
plt.title('Outliers detection')
plt.xlabel('Sales')
plt.tight_layout()
```



Orders

Central Tendency:

The mean (68) and median (63) are fairly close, suggesting that the data is somewhat symmetrical.

Dispersion:

The standard deviation of 30 indicates that there is moderate variability in sales values.

Skewness:

Since the mean is slightly greater than the median, the data may be right-skewed (positively skewed), there might be few very high Orders present in data which is pulling the distribution to the right.

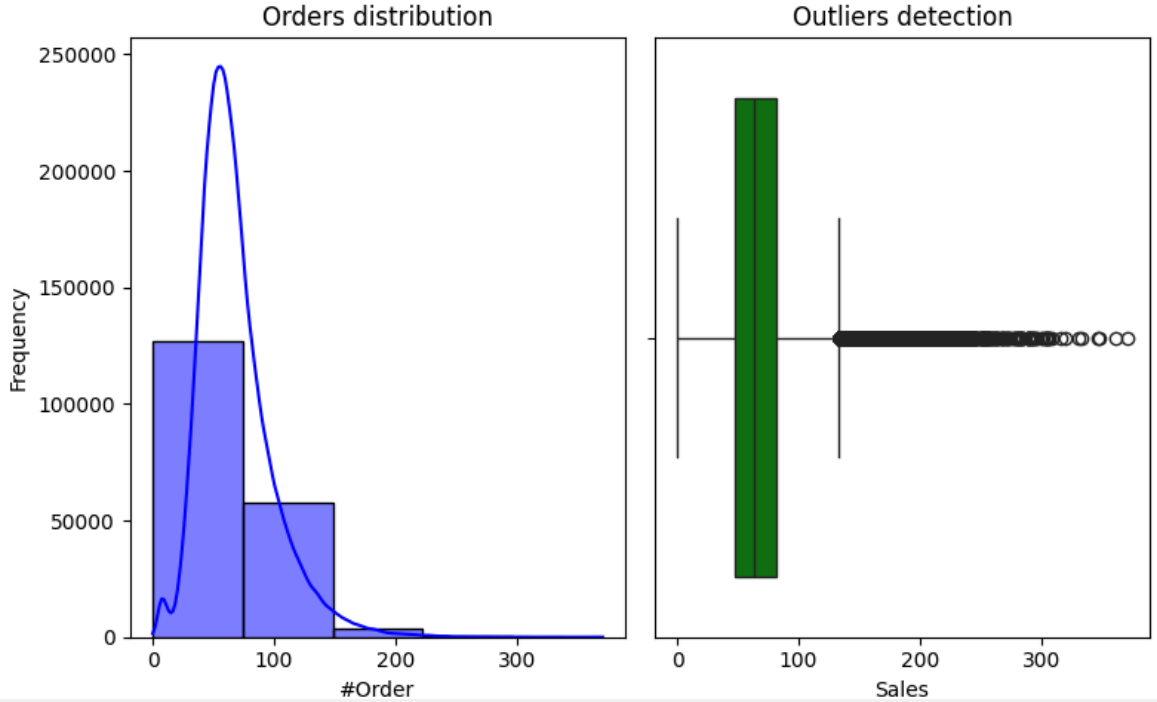
Percentiles:

The 25th percentile (Q1) is 48, the median (50th percentile) is 63, and the 75th percentile (Q3) is 82 which shows that half of the sales lies around 48 & 82. Therefore, the majority of Orders are clustered around these values, with a few very low or very high sales figures outside this range.

Outliers:

Considering the range extends from 0 to 371 sales record of 0 could be a low-performing transaction, while 371 could be an unusually highest Order.

```
# Order distribution
plt.figure(figsize=(8, 5))

plt.subplot(121)
sns.histplot(df['#Order'], kde=True, color='blue', bins=5)
plt.title('Orders distribution')
plt.xlabel('#Order')
plt.ylabel('Frequency')
plt.tight_layout()

# Outliers detection
plt.subplot(122)
sns.boxplot(x=df['#Order'], color='green')
plt.title('Outliers detection')
plt.xlabel('Sales')
plt.tight_layout()
```



Bivariate Analysis

As the analysation step of EDA, Basically it analyses the relationship between variables, It is done to do following:

- To analyse the
    - Strength of the Relationship
    - Direction of the Relationship
    - Significance
    - Cause and Effect.

Sales across Orders

Strength of the Relationship:

Since the correlation coefficient is to -1 or +1, it indicates the stronger the relationship between Sales & Orders.

Direction of the Relationship:

As we can see as Order increases, Sales also gets increased, so we can conclude its positively correlated.

Significance:

Statistical tests shows that observed relationship is statistically significant.

```
#Compute correlation between numerical variables (Sales and #Order)
correlation = df[['Sales', '#Order']].corr()
print("Correlation between Sales and #Order:\n", correlation)

# Visual representation
plt.figure(figsize=(5, 3))
sns.heatmap(correlation, annot=True, cmap='coolwarm', fmt='.1f')
```

```
Correlation between Sales and #Order:
              Sales    #Order
Sales    1.000000  0.941601
#Order   0.941601  1.000000
<Axes: >
```


```
#T-Test: figuring out the significnce
#H0: Increase of sales is not related to Order
#Ha: Sales Increases as Order increases

Sales = df['Sales']
Orders = df['#Order']
t_stat, p_value = stats.ttest_ind(Sales, Orders)
print("T-Statistic:", t_stat)
print("P-Value:", p_value)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: Sales Increases as Order increases")
else:
    print("Fail to reject the null hypothesis: Sales increase is not related to Order")


print("-----------------------------------------------------------------------------------------")

# Chi-Square Test: Association Between Order & Sales
#H0: There is no significant association between Sales & Order.
#Ha: There is a significant association between Sales & Order.

data = df[['Sales', '#Order']].corr()
chi2_stat, p_value_chi2, dof, expected = stats.chi2_contingency(data)
print("\nChi-Square Statistic:", chi2_stat)
print("P-Value:", p_value_chi2)
print("Degrees of Freedom:", dof)
print("Expected Frequencies:")
```

```
print(expected)

# Interpretation:
if p_value_chi2 < 0.05:
    print("Reject the null hypothesis: There is no significant association between Sales & Order.")
else:
    print("Fail to reject the null hypothesis: There is a significant association between Sales & Order.")
```

```
T-Statistic: 1004.4036077214703
P-Value: 0.0
Reject the null hypothesis: Sales Increases as Order increases
----------------------------------------------------------------------------------------------

Chi-Square Statistic: 0.0
P-Value: 1.0
Degrees of Freedom: 1
Expected Frequencies:
[[0.97080028 0.97080028]
 [0.97080028 0.97080028]]
Fail to reject the null hypothesis: There is a significant association between Sales & Order.
```

```
# Sales across Orders
plt.figure(figsize=(5, 4))
sns.scatterplot(data=df, x='#Order', y='Sales', color='blue')
plt.title('Sales across Orders')
plt.xlabel('#Order')
plt.ylabel('Sales')
plt.show()
```

Sales vs Discount

Strength of the Relationship:

Since the correlation coefficient is 0.3, it indicates the weak to moderate positive relationship.

Direction of the Relationship:

Since the correlation coefficient is 0.3, there is a positive relationship between Sales and Discount. which means as the discount increases, sales tend to increase, although the relationship is weak to moderate.

Significance:

Statistical tests shows that the relationship has very less statistical significance.

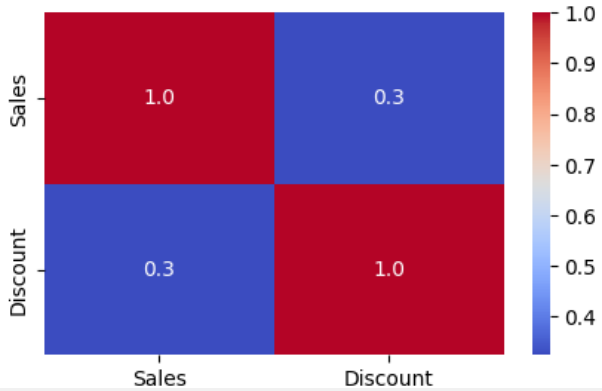
```
df_discount = df
df_discount['Discount'] = df_discount['Discount'].replace({'Yes': 1, 'No': 0})
```

```
<ipython-input-17-b4e43a0972e6>:2: FutureWarning:

Downcasting behavior in `replace` is deprecated and will be removed in a future version. To retain the old behavior, explicitly call `result.infer_objects(copy=False)`. To opt
```

```
#Compute correlation between numerical variables (Sales and Discount)
correlation = df_discount[['Sales', 'Discount']].corr()
```

```
print("Correlation between Sales and #Order:\n", correlation)

# Visual representation
plt.figure(figsize=(5, 3))
sns.heatmap(correlation, annot=True, cmap='coolwarm', fmt='.1f')
```

```
Correlation between Sales and #Order:
              Sales  Discount
Sales     1.000000  0.323906
Discount  0.323906  1.000000
<Axes: >
```



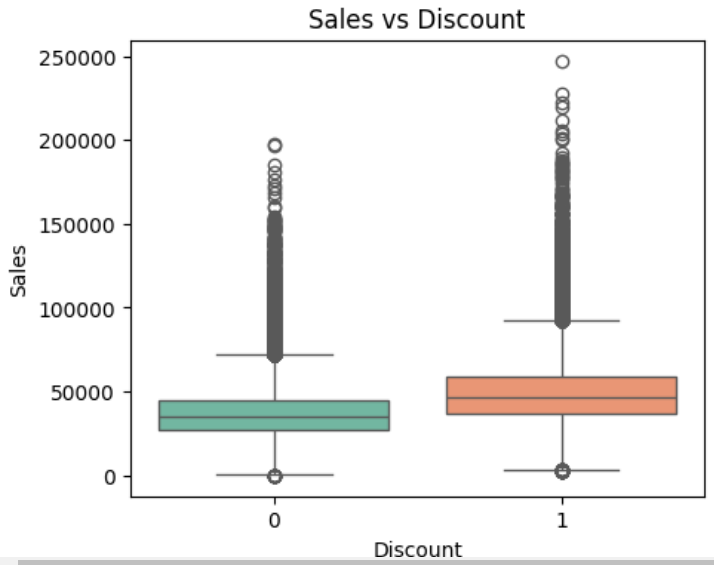

```
#T-Test: figuring out the significnce
#H0: Discount increases the sales
#Ha: Discount doesnot increases the sales

Sales = df_discount['Sales']
Discount = df_discount['Discount']
t_stat, p_value = stats.ttest_ind(Sales, Discount)
print("T-Statistic:", t_stat)
print("P-Value:", p_value)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: Discount increases the sales")
else:
    print("Fail to reject the null hypothesis: Discount doesnot increases the sales")


print("-----------------------------------------------------------------------------------------")

# Chi-Square Test: Association Between Discount & Sales
#H0: There is no significant association between Sales & Discount.
#Ha: There is a significant association between Sales & Discount.

data = df_discount[['Sales', 'Discount']].corr()
```

```
chi2_stat, p_value_chi2, dof, expected = stats.chi2_contingency(data)
print("\nChi-Square Statistic:", chi2_stat)
print("P-Value:", p_value_chi2)
print("Degrees of Freedom:", dof)
print("Expected Frequencies:")
print(expected)

# Interpretation:
if p_value_chi2 < 0.05:
    print("Reject the null hypothesis: There is no significant association between Sales & Discount.")
else:
    print("Fail to reject the null hypothesis: There is a significant association between Sales & Discount.")
```

```
T-Statistic: 1005.9982063352667
P-Value: 0.0
Reject the null hypothesis: Discount increases the sales
--------------------------------------------------------------------------------------------

Chi-Square Statistic: 0.0
P-Value: 1.0
Degrees of Freedom: 1
Expected Frequencies:
[[0.66195307 0.66195307]
 [0.66195307 0.66195307]]
Fail to reject the null hypothesis: There is a significant association between Sales & Discount.
```

```
# Sales vs Discount
plt.figure(figsize=(5, 4))
sns.boxplot(data=df, x='Discount', y='Sales', palette='Set2')
plt.title('Sales vs Discount')
plt.xlabel('Discount')
plt.ylabel('Sales')
plt.show()
```

```
<ipython-input-20-9b5eec569972>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```



Sales vs Holiday

Strength of the Relationship:

Since the correlation coefficient is -0.2, it indicates no relationship between Sales & Holiday.

Direction of the Relationship:

Since the correlation coefficient is -0.2, there is a Negative relationship between Sales and Holiday. Which means holiday has no effect on Sales increase.

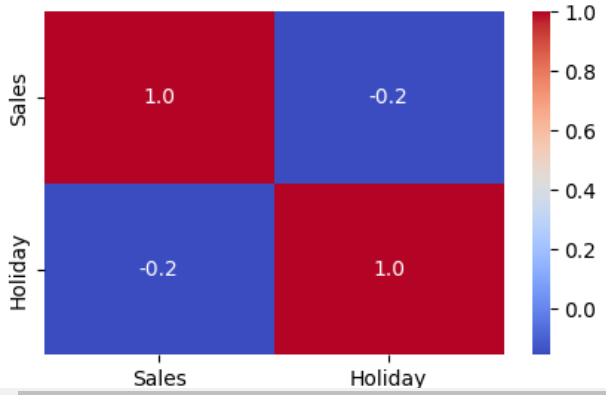
Significance:

Statistical tests shows that the relationship has no statistical significance.

```
#Compute correlation between numerical variables (Sales and Holiday)
correlation = df_discount[['Sales', 'Holiday']].corr()
print("Correlation between Sales and Holiday:\n", correlation)

# Visual representation
```

```
plt.figure(figsize=(5, 3))
sns.heatmap(correlation, annot=True, cmap='coolwarm', fmt='.1f')
```

```
Correlation between Sales and Holiday:
             Sales   Holiday
Sales    1.000000 -0.154779
Holiday -0.154779  1.000000
<Axes: >
```



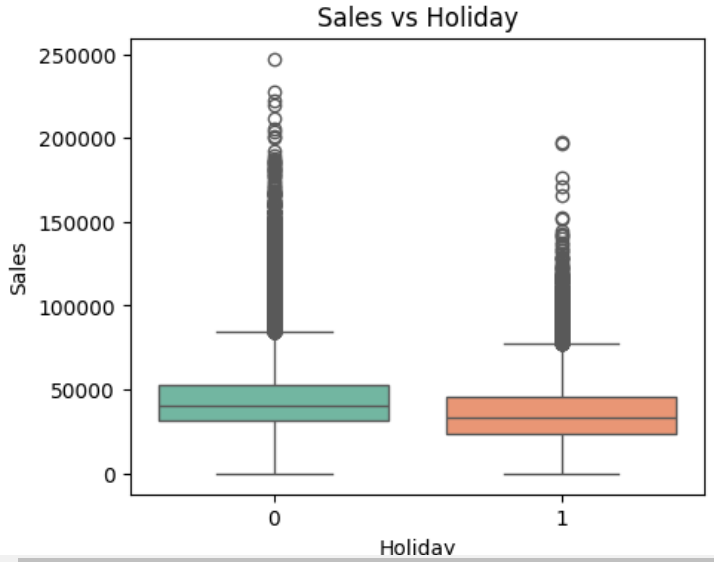

```
#T-Test: figuring out the significnce
#H0: Holiday increases the sales
#Ha: Holiday doesnot increases the sales

Sales = df_discount['Sales']
Holiday = df_discount['Holiday']
t_stat, p_value = stats.ttest_ind(Sales, Holiday)
print("T-Statistic:", t_stat)
print("P-Value:", p_value)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: Holiday increases the sales")
else:
    print("Fail to reject the null hypothesis: Holiday doesnot increases the sales")


print("-----------------------------------------------------------------------------------------")

# Chi-Square Test: Association Between Holiday & Sales
#H0: There is no significant association between Sales & Holiday.
#Ha: There is a significant association between Sales & Holiday.

data = df_discount[['Sales', 'Holiday']].corr()
data[data < 0] = 0
chi2_stat, p_value_chi2, dof, expected = stats.chi2_contingency(data)
print("\nChi-Square Statistic:", chi2_stat)
```

```
print("P-Value:", p_value_chi2)
print("Degrees of Freedom:", dof)
print("Expected Frequencies:")
print(expected)

# Interpretation:
if p_value_chi2 < 0.05:
    print("Reject the null hypothesis: There is no significant association between Sales & Holiday.")
else:
    print("Fail to reject the null hypothesis: There is a significant association between Sales & Holiday.")
```

```
T-Statistic: 1006.0056309953759
P-Value: 0.0
Reject the null hypothesis: Holiday increases the sales
--------------------------------------------------------------------------------

Chi-Square Statistic: 0.0
P-Value: 1.0
Degrees of Freedom: 1
Expected Frequencies:
[[0.5 0.5]
 [0.5 0.5]]
Fail to reject the null hypothesis: There is a significant association between Sales & Holiday.
```

```
# Sales vs Holiday
plt.figure(figsize=(5, 4))
sns.boxplot(data=df, x='Holiday', y='Sales', palette='Set2')
plt.title('Sales vs Holiday')
plt.xlabel('Holiday')
plt.ylabel('Sales')
plt.show()
```

```
<ipython-input-23-ff9b74c10e84>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```



Sales vs Store_type

Median :

Median of store S4 seems to be higher than other stores, we can assume that in S4 location the sales is quite high compared to ther stores.

Interquartile Range (IQR):

Similarly, IQR for S4 seems to be wider than other stores IQR, so we can assume that in S4 location the sales is quite high compared to ther stores.

Whiskers:

From below we can see that the whiskers of S4 is quite big which indicates that the sales are more spread out in S4.

Outliers:

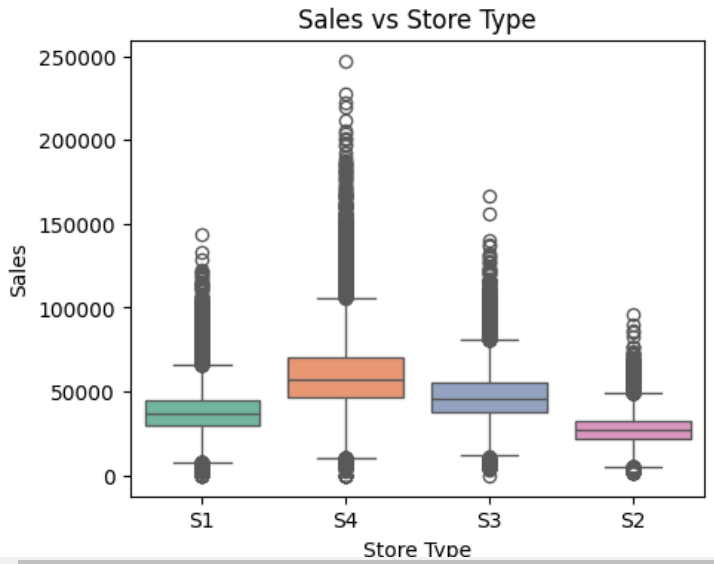
For Outlier too S4 stands out, where there are more variations among sales done by customers.

```
from scipy import stats

# Store_Type category
one = df[df['Store_Type'] == 'S1']['Sales']
two = df[df['Store_Type'] == 'S2']['Sales']
three = df[df['Store_Type'] == 'S3']['Sales']
four = df[df['Store_Type'] == 'S4']['Sales']
f_stat, p_value = stats.f_oneway(one, two,three,four)
print("F-Statistic:", f_stat)
print("P-Value:", p_value)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: There is no significant association between Sales & Store_Type.")
else:
    print("Fail to reject the null hypothesis: There is a significant association between Sales & Store_Type.")
```

```
F-Statistic: 35123.64411601774
P-Value: 0.0
Reject the null hypothesis: There is no significant association between Sales & Store_Type.
```

```
# Sales vs Store Type
plt.figure(figsize=(5, 4))
sns.boxplot(data=df, x='Store_Type', y='Sales', palette='Set2')
plt.title('Sales vs Store Type')
plt.xlabel('Store Type')
plt.ylabel('Sales')
plt.show()
```

```
<ipython-input-25-18ad79744faf>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```



Sales vs Location_Type

Median :

Median of store L2 seems to be higher than other stores, we can assume that in L2 location the sales is quite high compared to ther stores.

Interquartile Range (IQR):

Similarly, IQR for L2 seems to be wider than other stores IQR, so we can assume that in L2 location the sales is quite high compared to ther stores.

Whiskers:

From below we can see that the whiskers of L2 is quite big which indicates that the sales are more spread out in L2.

Outliers:

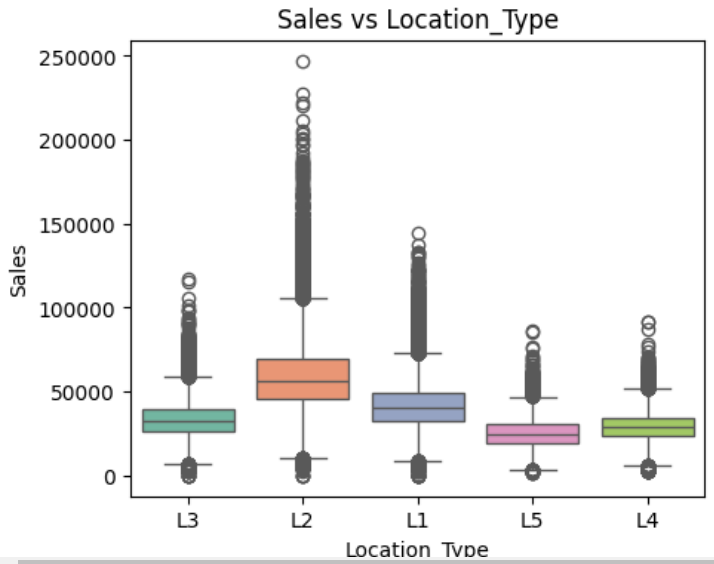
For Outlier too L2 stands out, where there are more variations among sales done by customers.

```
from scipy import stats

# Location_Type category
one = df[df['Location_Type'] == 'L1']['Sales']
two = df[df['Location_Type'] == 'L2']['Sales']
three = df[df['Location_Type'] == 'L3']['Sales']
four = df[df['Location_Type'] == 'L4']['Sales']
five = df[df['Location_Type'] == 'L5']['Sales']
f_stat, p_value = stats.f_oneway(one, two,three,four,five)
print("F-Statistic:", f_stat)
print("P-Value:", p_value)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: There is no significant association between Sales & Location_Type.")
else:
    print("Fail to reject the null hypothesis: There is a significant association between Sales & Location_Type.")
```

```
F-Statistic: 25338.873708475992
P-Value: 0.0
Reject the null hypothesis: There is no significant association between Sales & Location_Type.
```

```
# Sales vs Location_Type
plt.figure(figsize=(5, 4))
sns.boxplot(data=df, x='Location_Type', y='Sales', palette='Set2')
plt.title('Sales vs Location_Type')
plt.xlabel('Location_Type')
plt.ylabel('Sales')
plt.show()
```

```
<ipython-input-27-b2538c3cafa6>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```



Sales vs Region_Code

Median :

Median of store R1 seems to be higher than other stores, we can assume that in R1 location the sales is quite high compared to ther stores.

Interquartile Range (IQR):

Similarly, IQR for R1 seems to be wider than other stores IQR, so we can assume that in R1 location the sales is quite high compared to ther stores.

Whiskers:

From below we can see that the whiskers of R1 is quite big which indicates that the sales are more spread out in R1.

Outliers:

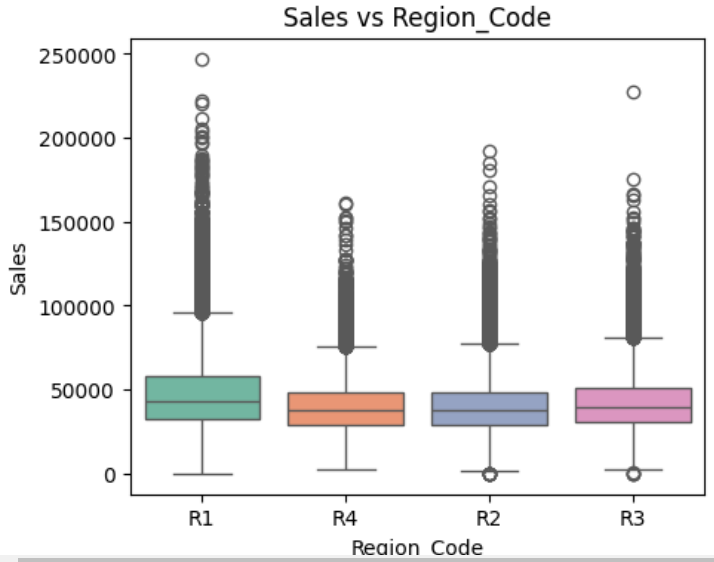
For Outlier too R1 stands out, where there are more variations among sales done by customers.

```
from scipy import stats

# Location_Type category
one = df[df['Region_Code'] == 'R1']['Sales']
two = df[df['Region_Code'] == 'R2']['Sales']
three = df[df['Region_Code'] == 'R3']['Sales']
four = df[df['Region_Code'] == 'R4']['Sales']
f_stat, p_value = stats.f_oneway(one, two,three,four)
print("F-Statistic:", f_stat)
print("P-Value:", p_value)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: There is no significant association between Sales & Region_Code.")
else:
    print("Fail to reject the null hypothesis: There is a significant association between Sales & Region_Code.")
```

```
F-Statistic: 1682.4255287950677
P-Value: 0.0
Reject the null hypothesis: There is no significant association between Sales & Region_Code.
```

```
# Sales vs Region_Code
plt.figure(figsize=(5, 4))
sns.boxplot(data=df, x='Region_Code', y='Sales', palette='Set2')
plt.title('Sales vs Region_Code')
plt.xlabel('Region_Code')
plt.ylabel('Sales')
plt.show()
```

```
<ipython-input-29-540c97fe78b5>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```



Time Series Analysis

The time series plot shows the total sales over time. It is done to determine the following:

- Trend
- Seasonality
- Outliers

Time Series Analysis

Trend Component::

However from the given data sales variations are quite high, but in the last month of the given data sales is towards upward trend, so we can assume that there is a gradual increase in sales.

Seasonal Component:

Repeating fluctuations are found in the given data that occur at specific periods, consistent peaks are found after April month maybe due to seasonality i-e. holidays

Residual Component (Noise):

The data is scattered randomly and there are no patterns to be found which indicates the proper decomposition.

Forecasting:

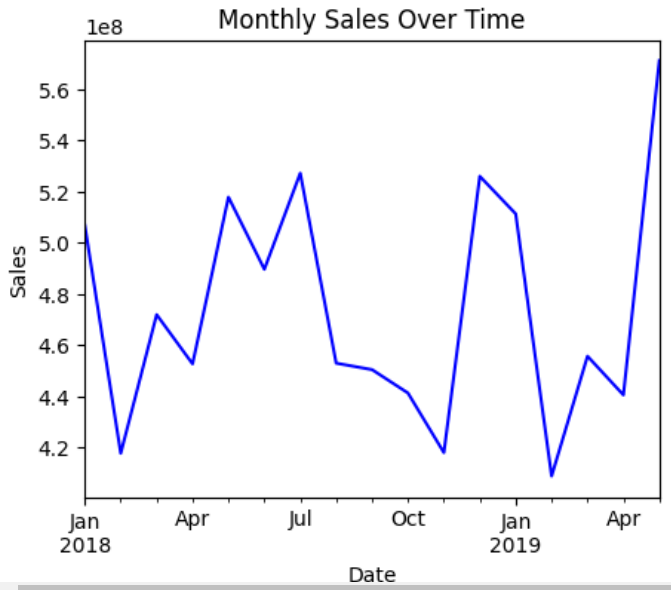
In addition to the above forecating is also done for next 6 months down the line, which shows the cosistent sales, means there is no steep downfall in the sales.

```
df_t = df
df_t['Date'] = pd.to_datetime(df_t['Date'])
df_t['year'] = df_t['Date'].dt.year
df_t['month'] = df_t['Date'].dt.month
df_t.head()
```

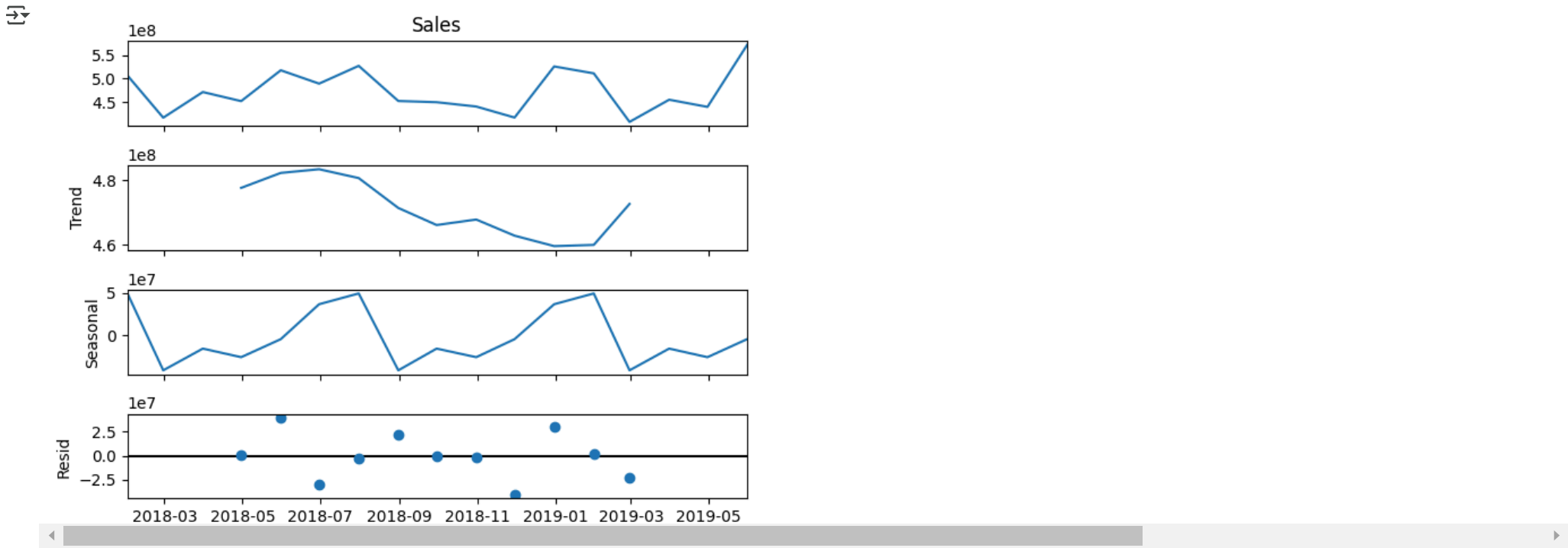
	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount	#Order	Sales	year	month
0	T1000001	1	S1	L3	R1	2018-01-01	1	1	9	7011.84	2018	1
1	T1000002	253	S4	L2	R1	2018-01-01	1	1	60	51789.12	2018	1
2	T1000003	252	S3	L2	R1	2018-01-01	1	1	42	36868.20	2018	1
3	T1000004	251	S2	L3	R1	2018-01-01	1	1	23	19715.16	2018	1
4	T1000005	250	S2	L3	R4	2018-01-01	1	1	62	45614.52	2018	1

```
df_t.set_index('Date', inplace=True)

# Grouping by month, summing up sales
monthly_sales = df_t.resample('M')['Sales'].sum()

# Time series plot
plt.figure(figsize=(5, 4))
monthly_sales.plot(title="Monthly Sales Over Time", xlabel="Date", ylabel="Sales", color='blue')
plt.show()
```

```
<ipython-input-31-f7f918dc1af6>:4: FutureWarning:

'M' is deprecated and will be removed in a future version, please use 'ME' instead.
```

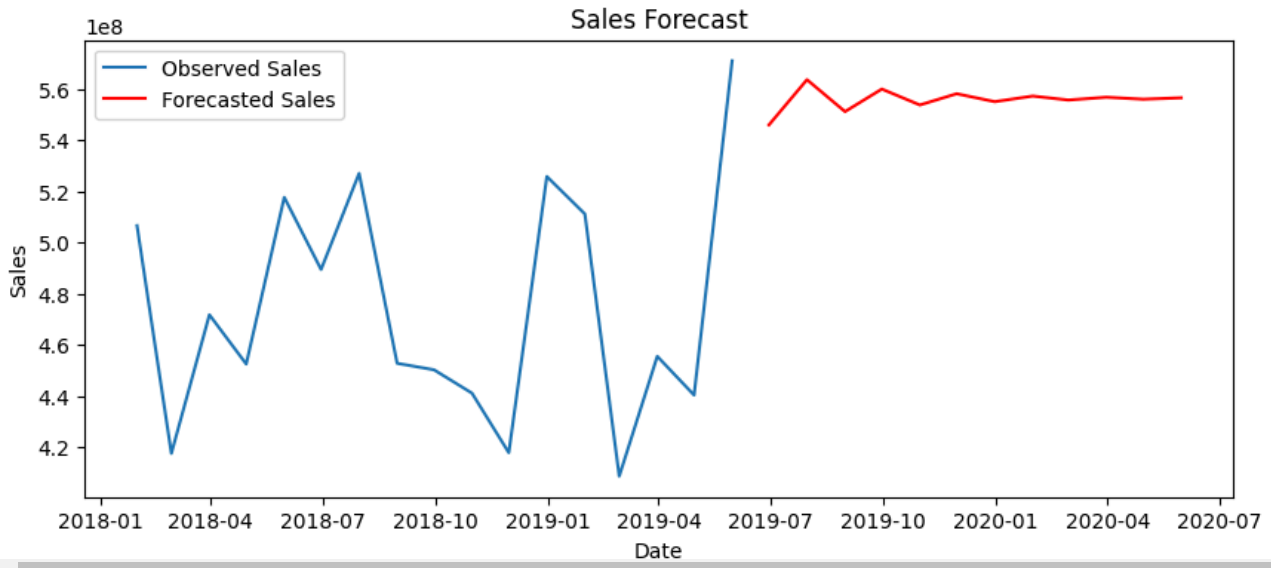



```
register_matplotlib_converters()
# Decompose the time series for period=6
decomposition = seasonal_decompose(monthly_sales, model='additive', period=6)
decomposition.plot()
plt.show()
```




```
# ARIMA model for forecasting
model = ARIMA(monthly_sales, order=(1, 1, 1))
model_fit = model.fit()

# Making predictions
forecast = model_fit.forecast(steps=12)
forecast_index = pd.date_range(start=monthly_sales.index[-1], periods=13, freq='M')[1:]

plt.figure(figsize=(10, 4))
plt.plot(monthly_sales, label='Observed Sales')
plt.plot(forecast_index, forecast, label='Forecasted Sales', color='red')
plt.title('Sales Forecast')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.legend()
plt.show()
```

```
<ipython-input-33-fb8ba4de957d>:7: FutureWarning:

'M' is deprecated and will be removed in a future version, please use 'ME' instead.
```



Categorical Analysis

Frequency Distribution:

A simple calculation of how often each category appears in the dataset.

Contingency Tables:

Via which the relationships between categorical variables are analysed.

Chi-Square Test:

It is mainly done to determine if there is a significant relationship between two categorical variables.

Above tests will be done from the given data, from which we are analysing if below relationships are present,

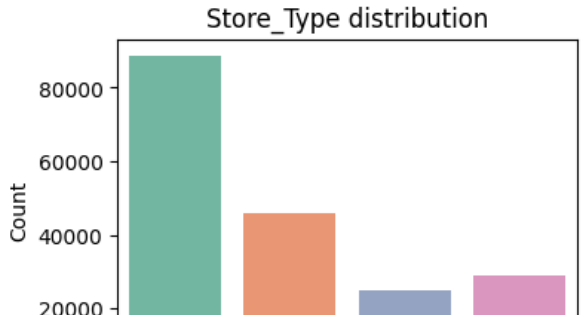
- Holiday & Discount impact on
    - Store_Type
    - Location_Type
    - Region_Code

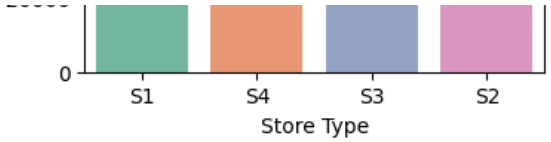
From the below inferences we can quite come to assumption that there are no significant relationships between any of the categorical variables.

```
# Analyzing frequency counts of categorical variables
print("Frequency of Store_Type:")
print(df['Store_Type'].value_counts())

print("\nFrequency of Location_Type:")
print(df['Location_Type'].value_counts())

print("\nFrequency of Region_Code:")
print(df['Region_Code'].value_counts())

print("\nFrequency of Holiday:")
print(df['Holiday'].value_counts())

print("\nFrequency of Discount:")
print(df['Discount'].value_counts())

# Store_Type plot
plt.figure(figsize=(4, 3))
sns.countplot(data=df, x='Store_Type', palette='Set2')
plt.title('Store_Type distribution')
plt.xlabel('Store Type')
plt.ylabel('Count')
plt.show()

# Location_Type plot
plt.figure(figsize=(4, 3))
sns.countplot(data=df, x='Location_Type', palette='Set2')
plt.title('Location_Type distribution')
plt.xlabel('Location Type')
plt.ylabel('Count')
plt.show()

# Region_Code plot
plt.figure(figsize=(4, 3))
sns.countplot(data=df, x='Region_Code', palette='Set2')
plt.title('Region_Code distribution')
plt.xlabel('Region Code')
plt.ylabel('Count')
plt.show()

# Holiday plot
plt.figure(figsize=(4, 3))
sns.countplot(data=df, x='Holiday', palette='Set2')
plt.title('Holiday distribution')
plt.xlabel('Holiday (1 = Yes, 0 = No)')
plt.ylabel('Count')
plt.show()

# Discount plot
plt.figure(figsize=(4, 3))
```

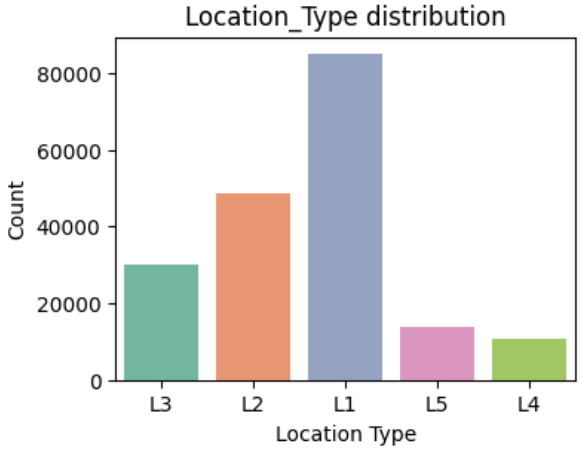
```
sns.countplot(data=df, x='Discount', palette='Set2')
plt.title('Discount distribution')
plt.xlabel('Discount')
plt.ylabel('Count')
plt.show()
```

```
Frequency of Store_Type:
Store_Type
S1    88752
S4    45924
S2    28896
S3    24768
Name: count, dtype: int64

Frequency of Location_Type:
Location_Type
L1    85140
L2    48504
L3    29928
L5    13932
L4    10836
Name: count, dtype: int64

Frequency of Region_Code:
Region_Code
R1    63984
R2    54180
R3    44376
R4    25800
Name: count, dtype: int64

Frequency of Holiday:
Holiday
0    163520
1     24820
Name: count, dtype: int64

Frequency of Discount:
Discount
0    104051
1     84289
Name: count, dtype: int64
<ipython-input-34-6c3cf5dc9bd7>:19: FutureWarning:


Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```

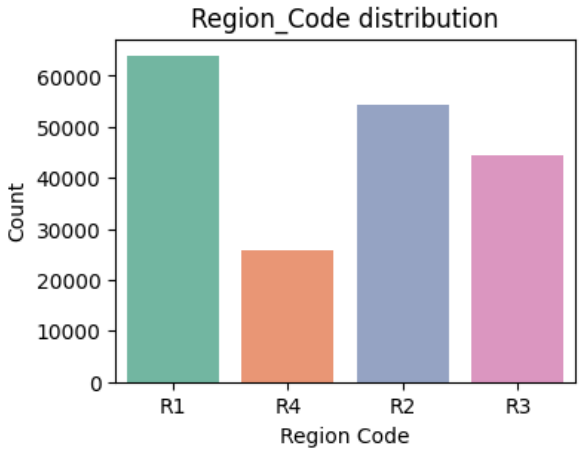




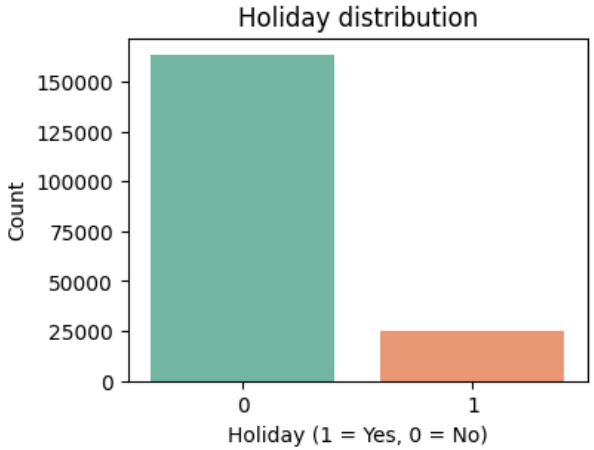

```
<ipython-input-34-6c3cf5dc9bd7>:27: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```



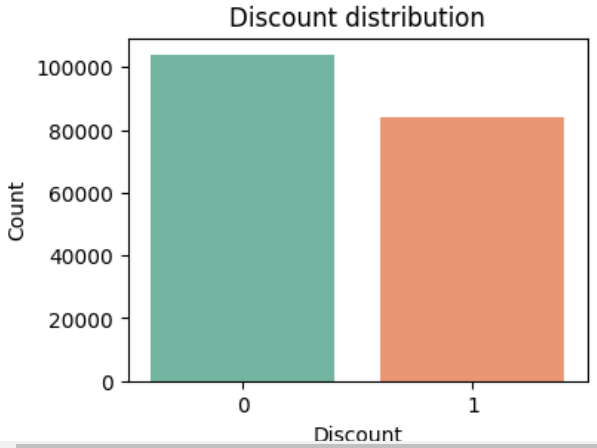

```
<ipython-input-34-6c3cf5dc9bd7>:35: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```



```
<ipython-input-34-6c3cf5dc9bd7>:43: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```




```
<ipython-input-34-6c3cf5dc9bd7>:51: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.
```



Holiday impact on,

- Store_Type
- Location_Type
- Region_Code

```
# Creating contingency table
contingency_table_H_Store_Type = pd.crosstab(df['Holiday'], df['Store_Type'])
print("contingency_table_H_Store_Type:\n", contingency_table_H_Store_Type)
print("")

contingency_table_H_Location_Type = pd.crosstab(df['Holiday'], df['Location_Type'])
print("contingency_table_H_Location_Type:\n", contingency_table_H_Location_Type)
print("")

contingency_table_H_Region_Code = pd.crosstab(df['Holiday'], df['Region_Code'])
print("contingency_table_H_Location_Type:\n", contingency_table_H_Region_Code)
print("")
print("------------------------------------------------------------------------")

# Chi-Square Test
# H0: There is no relationship
# Ha: There is relationship
chi2, p_value, dof, expected = chi2_contingency(contingency_table_H_Store_Type)
print("Chi-Square Statistic:", chi2)
print("P-Value:", p_value)
print("Degrees of Freedom:", dof)
print("Expected Frequencies:\n", expected)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: There is relationship between Holiday & Store_Type.")
else:
    print("Fail to reject the null hypothesis: There is no relationship between Holiday & Store_Type.")

print("------------------------------------------------------------------------")

chi2, p_value, dof, expected = chi2_contingency(contingency_table_H_Location_Type)
print("Chi-Square Statistic:", chi2)
print("P-Value:", p_value)
print("Degrees of Freedom:", dof)
print("Expected Frequencies:\n", expected)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: There is relationship between Holiday & Location_Type.")
else:
    print("Fail to reject the null hypothesis: There is no relationship between Holiday & Location_Type.")
```

```
print("------------------------------------------------------------------------")

chi2, p_value, dof, expected = chi2_contingency(contingency_table_H_Region_Code)
print("Chi-Square Statistic:", chi2)
print("P-Value:", p_value)
print("Degrees of Freedom:", dof)
print("Expected Frequencies:\n", expected)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: There is relationship between Holiday & Region_Code.")
else:
    print("Fail to reject the null hypothesis: There is no relationship between Holiday & Region_Code.")
```

```
contingency_table_H_Store_Type:
 Store_Type     S1     S2     S3     S4
Holiday
0           77056  25088  21504  39872
1           11696   3808   3264   6052

contingency_table_H_Location_Type:
 Location_Type     L1     L2     L3    L4     L5
Holiday
0              73920  42112  25984  9408  12096
1              11220   6392   3944  1428   1836

contingency_table_H_Location_Type:
 Region_Code     R1     R2     R3     R4
Holiday
0            55552  47040  38528  22400
1             8432   7140   5848   3400

------------------------------------------------------------------------
Chi-Square Statistic: 0.0
P-Value: 1.0
Degrees of Freedom: 3
Expected Frequencies:
 [[77056. 25088. 21504. 39872.]
 [11696.  3808.  3264.  6052.]]
Fail to reject the null hypothesis: There is no relationship between Holiday & Store_Type.
------------------------------------------------------------------------
Chi-Square Statistic: 0.0
P-Value: 1.0
Degrees of Freedom: 4
Expected Frequencies:
 [[73920. 42112. 25984.  9408. 12096.]
 [11220.  6392.  3944.  1428.  1836.]]
Fail to reject the null hypothesis: There is no relationship between Holiday & Location_Type.
------------------------------------------------------------------------
Chi-Square Statistic: 0.0
P-Value: 1.0
Degrees of Freedom: 3
```

```
Expected Frequencies:
 [[55552. 47040. 38528. 22400.]
 [ 8432.  7140.  5848.  3400.]]
Fail to reject the null hypothesis: There is no relationship between Holiday & Region_Code.
```

Discount impact on,

- Store_Type
- Location_Type
- Region_Code

```
# Creating contingency table
contingency_table_D_Store_Type = pd.crosstab(df['Discount'], df['Store_Type'])
print("contingency_table_D_Store_Type:\n", contingency_table_D_Store_Type)
print("")

contingency_table_D_Location_Type = pd.crosstab(df['Discount'], df['Location_Type'])
print("contingency_table_D_Location_Type:\n", contingency_table_D_Location_Type)
print("")

contingency_table_D_Region_Code = pd.crosstab(df['Discount'], df['Region_Code'])
print("contingency_table_D_Location_Type:\n", contingency_table_D_Region_Code)
print("")
print("-------------------------------------------------------------------------")

# Chi-Square Test
# H0: There is no relationship
# Ha: There is relationship
chi2, p_value, dof, expected = chi2_contingency(contingency_table_D_Store_Type)
print("Chi-Square Statistic:", chi2)
print("P-Value:", p_value)
print("Degrees of Freedom:", dof)
print("Expected Frequencies:\n", expected)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: There is relationship between Discount & Store_Type.")
else:
    print("Fail to reject the null hypothesis: There is no relationship between Discount & Store_Type.")

print("-------------------------------------------------------------------------")

chi2, p_value, dof, expected = chi2_contingency(contingency_table_D_Location_Type)
print("Chi-Square Statistic:", chi2)
print("P-Value:", p_value)
print("Degrees of Freedom:", dof)
print("Expected Frequencies:\n", expected)

# Interpretation:
```

```
if p_value < 0.05:
    print("Reject the null hypothesis: There is relationship between Discount & Location_Type.")
else:
    print("Fail to reject the null hypothesis: There is no relationship between Discount & Location_Type.")


print("------------------------------------------------------------------------")

chi2, p_value, dof, expected = chi2_contingency(contingency_table_D_Region_Code)
print("Chi-Square Statistic:", chi2)
print("P-Value:", p_value)
print("Degrees of Freedom:", dof)
print("Expected Frequencies:\n", expected)

# Interpretation:
if p_value < 0.05:
    print("Reject the null hypothesis: There is relationship between Discount & Region_Code.")
else:
    print("Fail to reject the null hypothesis: There is no relationship between Discount & Region_Code.")
```

```
contingency_table_D_Store_Type:
 Store_Type     S1     S2     S3     S4
Discount
0           49093  16025  13627  25306
1           39659  12871  11141  20618

contingency_table_D_Location_Type:
 Location_Type     L1     L2     L3    L4    L5
Discount
0              47114  26721  16507  6007  7702
1              38026  21783  13421  4829  6230

contingency_table_D_Location_Type:
 Region_Code     R1     R2     R3     R4
Discount
0            35307  30036  24449  14259
1            28677  24144  19927  11541

------------------------------------------------------------------------
Chi-Square Statistic: 1.5849704434955905
P-Value: 0.6628024520364906
Degrees of Freedom: 3
Expected Frequencies:
 [[49032.25205479 15963.9890411  13683.41917808 25371.33972603]
 [39719.74794521 12932.0109589  11084.58082192 20552.66027397]]
Fail to reject the null hypothesis: There is no relationship between Discount & Store_Type.
------------------------------------------------------------------------
Chi-Square Statistic: 1.025131894074277
P-Value: 0.9059615295392269
Degrees of Freedom: 4
Expected Frequencies:
 [[47036.75342466 26796.69589041 16534.13150685  5986.49589041
   7696.92328767]
```

```
 [38103.24657534 21707.30410959 13393.86849315  4849.50410959
   6235.07671233]]
Fail to reject the null hypothesis: There is no relationship between Discount & Location_Type.
----------------------------------------------------------------------
Chi-Square Statistic: 1.325919741035845
P-Value: 0.7229875725882295
Degrees of Freedom: 3
Expected Frequencies:
```

Handling Missing Values

Missing values needs to be handled mainly to ensure below,

- Accuracy
- Data Integrity &
- Bias Prevention

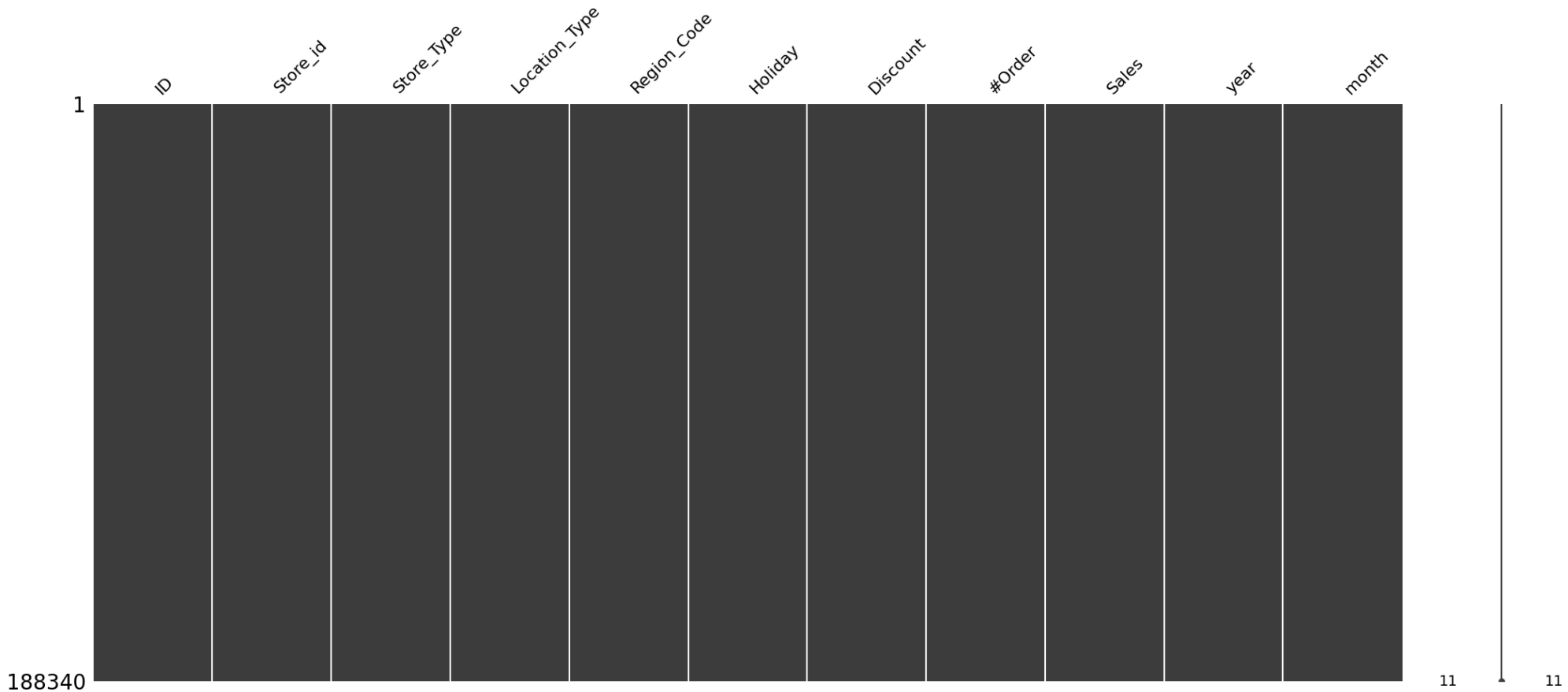
  From the given data we could see that there are no missing values present, which makes it easier for futhre Imputations & Analysations

```
# Checking for missing values
print("Missing Values Before Handling:")
print(df.isnull().sum())
```

```
Missing Values Before Handling:
ID              0
Store_id        0
Store_Type      0
Location_Type   0
Region_Code     0
Holiday         0
Discount        0
#Order          0
Sales           0
year            0
month           0
dtype: int64
```

```
# Visualize missing data
msno.matrix(df)
```

<Axes: >



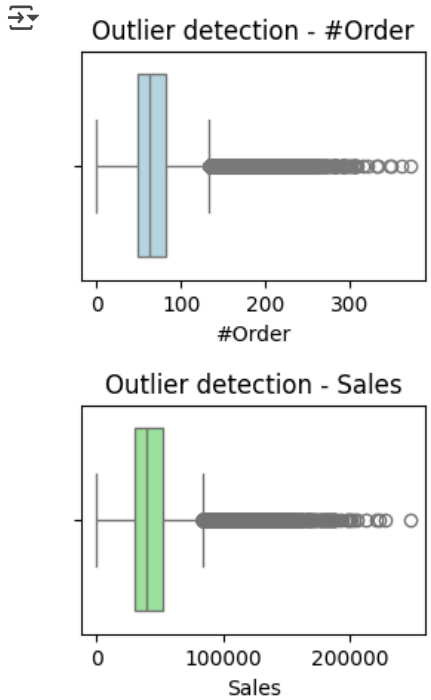
Outlier Detection

It is mainly done to identify data points that are significantly different from the majority of the data in order to decide whether to correct them, or remove them from the analysis. It mainly helps in below,

- Improving Model Accuracy
- Data Quality
- Insights

From the given data we could see there are quite high outliers present, Further Z & IQR test has been done to micro analyse the outliers present inside the given data.

```
# Outlier detection - '#Order'
plt.figure(figsize=(3, 2))
sns.boxplot(data=df, x='#Order', color='lightblue')
plt.title('Outlier detection - #Order')
plt.show()

# Outlier detection - 'Sales'
plt.figure(figsize=(3, 2))
sns.boxplot(data=df, x='Sales', color='lightgreen')
plt.title('Outlier detection - Sales')
plt.show()
```




```
# Z-Score Outlier detection
df['Z_score_Order'] = zscore(df['#Order'])
df['Z_score_Sales'] = zscore(df['Sales'])

# Z-score threshold
z_score_threshold = 3

# Outliers detection
outliers_zscore_order = df[df['Z_score_Order'].abs() > z_score_threshold]
outliers_zscore_sales = df[df['Z_score_Sales'].abs() > z_score_threshold]
print("Outliers based on Z-Score for #Order:")
```

```
print(outliers_zscore_order[['ID', '#Order', 'Z_score_Order']])

print("\nOutliers based on Z-Score for Sales:")
print(outliers_zscore_sales[['ID', 'Sales', 'Z_score_Sales']])
```

```
Outliers based on Z-Score for #Order:
                    ID  #Order  Z_score_Order
Date
2018-01-01  T1000307     180       3.669317
2018-01-02  T1000494     189       3.964715
2018-01-02  T1000595     167       3.242631
2018-01-02  T1000624     181       3.702139
2018-01-03  T1000787     189       3.964715
...              ...     ...            ...
2019-05-30  T1187690     173       3.439563
2019-05-30  T1187795     180       3.669317
2019-05-31  T1188037     168       3.275453
2019-05-31  T1188242     179       3.636495
2019-05-31  T1188321     171       3.373919

[2664 rows x 3 columns]

Outliers based on Z-Score for Sales:
                    ID      Sales  Z_score_Sales
Date
2018-01-01  T1000181  115665.30       3.948763
2018-01-01  T1000307  126633.00       4.543004
2018-01-02  T1000374  105243.00       3.384072
2018-01-02  T1000404  104433.00       3.340186
2018-01-02  T1000467  101577.00       3.185445
...              ...        ...            ...
2019-05-30  T1187690   99027.00       3.047283
2019-05-30  T1187708   99477.00       3.071665
2019-05-30  T1187795  104406.00       3.338723
2019-05-31  T1188242  103361.94       3.282155
2019-05-31  T1188321  110414.28       3.664257

[2421 rows x 3 columns]
```

```
# IQR Method Outlier detection
Q1_order = df['#Order'].quantile(0.25)
Q3_order = df['#Order'].quantile(0.75)
IQR_order = Q3_order - Q1_order
Q1_sales = df['Sales'].quantile(0.25)
Q3_sales = df['Sales'].quantile(0.75)
IQR_sales = Q3_sales - Q1_sales

# IQR Outlier ranges
lower_bound_order = Q1_order - 1.5 * IQR_order
upper_bound_order = Q3_order + 1.5 * IQR_order
lower_bound_sales = Q1_sales - 1.5 * IQR_sales
upper_bound_sales = Q3_sales + 1.5 * IQR_sales
```

```
# Outliers detection
outliers_iqr_order = df[(df['#Order'] < lower_bound_order) | (df['#Order'] > upper_bound_order)
outliers_iqr_sales = df[(df['Sales'] < lower_bound_sales) | (df['Sales'] > upper_bound_sales)]
print("\nOutliers based on IQR for #Order:")
print(outliers_iqr_order[['ID', '#Order']])
print("\nOutliers based on IQR for Sales:")
print(outliers_iqr_sales[['ID', 'Sales']])
```

```
Outliers based on IQR for #Order:
                   ID  #Order
Date
2018-01-01  T1000181     154
2018-01-01  T1000307     180
2018-01-02  T1000374     142
2018-01-02  T1000398     146
2018-01-02  T1000404     145
...              ...     ...
2019-05-31  T1188242     179
2019-05-31  T1188300     151
2019-05-31  T1188321     171
2019-05-31  T1188322     152
2019-05-31  T1188333     137

[7089 rows x 2 columns]

Outliers based on IQR for Sales:
                   ID      Sales
Date
2018-01-01  T1000078   85426.92
2018-01-01  T1000134   86962.68
2018-01-01  T1000181  115665.30
2018-01-01  T1000188   86203.20
2018-01-01  T1000307  126633.00
...              ...        ...
2019-05-31  T1188244   90231.54
2019-05-31  T1188300   95497.92
2019-05-31  T1188321  110414.28
2019-05-31  T1188322   87785.70
2019-05-31  T1188333   86994.18

[5843 rows x 2 columns]
```

Suggestions for Hypothesis Testing

Hypothesis testing is a statistical method mainly used to make inferences by following below steps,

- Hypothesis stating
- Choosing Significance Level (α)
- Selecting appropriate testing
- Test Statistic computation
- P-value identification
- Decision making
- Result Interpretation

> ### *Impact of Discounts on Sales*

As Discounts are often used to attract customers, & encourage more purchases and sometimes used to clear inventory. In addition to above discounts are provided to encourage below factors such as,

- Increased Demand Due to Lower Prices
- Customer Perception and Behavior
- Attracting New Customers and Retaining Existing Ones
- Competitive Advantages
- Impact on Profitability

Why t-test?

As it allows to determine if there is a statistically significant difference in the sales before and after a discount is applied,

Assumptions of the T-test:

- ***Sample Groups :***

    1. Sales before discount
    2. Sales after discount

- ***Normality:***

  Data is normally distributed

- ***Equal variance:***

  Group variances are similar

- ***Form Hypotheses:***

  Null Hypothesis (H_0): The discount does not impact sales

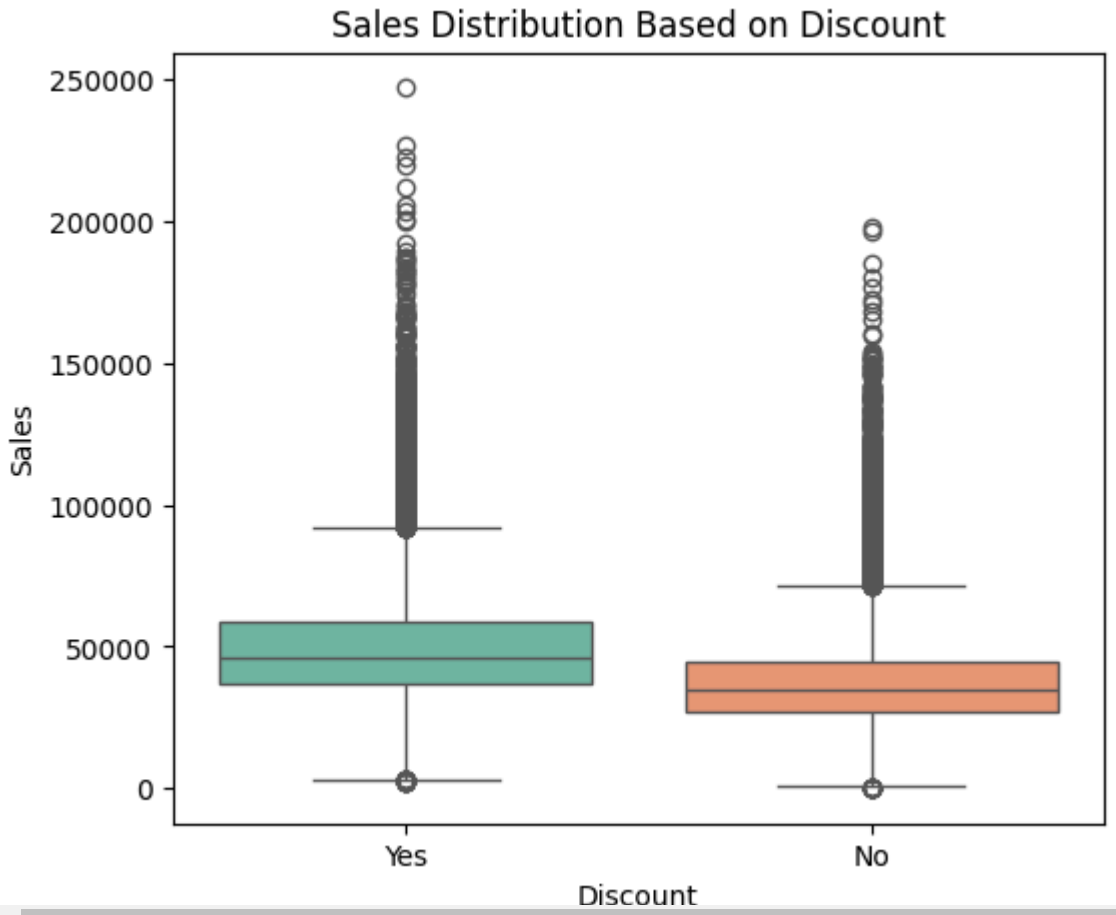
  Alternative Hypothesis (H_1): The discount impacts sales

```
# Descriptive Statistics of Sales based on Discount
sales_discount_group = df.groupby('Discount')['Sales'].describe()
print("Descriptive Statistics of Sales Based on Discount:")
print(sales_discount_group)
```

```
Descriptive Statistics of Sales Based on Discount:
              count          mean           std      min       25%      50%  \
Discount
No         104051.0  37403.679678  16039.555183     0.00  27069.00  34791.0
Yes         84289.0  49426.497620  19071.656642  1969.14  36625.35  46242.0

                75%        max
Discount
No          44968.5  197840.61
Yes         58869.0  247215.00
```

```
# Sales Distribution based on Discount
plt.figure(figsize=(6, 5))
sns.boxplot(data=df, x='Discount', y='Sales', palette='Set2')
plt.title('Sales Distribution Based on Discount')
plt.xlabel('Discount')
plt.ylabel('Sales')
plt.show()
```

```
<ipython-input-9-5c473e09b1da>:3: FutureWarning:

    Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set
```




```
# T-Test
# Null hypothesis (H0): Discount doesnot impact sales
# Alternative hypothesis (H1):  Discount impact sales
discount_yes = df[df['Discount'] == 'Yes']['Sales']
```

```
discount_no = df[df['Discount'] == 'No']['Sales']
t_stat, p_value = stats.ttest_ind(discount_yes, discount_no)
print("\nT-Test Results:")
print(f"T-statistic: {t_stat:.3f}")
print(f"P-value: {p_value:.3f}")

# Interpretation
if p_value < 0.05:
    print("Reject the null hypothesis: Discount impact sales")
else:
    print("Fail to reject the null hypothesis: Discount doesnot impact sales")
```

```
T-Test Results:
T-statistic: 148.579
P-value: 0.000
Reject the null hypothesis: Discount impact sales
```

Effect of Holidays on Sales

Similar to Discounts we can further anlayse if Holidays impact sales

- Time availability of customer for purchase
- Shopping mode
- Possibility of increased visits

Why t-test?

As it allows to determine if there is a statistically significant difference in the sales before and after a Holidays,

Assumptions of the T-test:

- ***Sample Groups :***
    1. Sales during holidays

2. Sales during non-holidays

- ***Normality:***

  Data is normally distributed

- ***Equal variance:***

  Group variances are similar

- ***Form Hypotheses:***

  Null Hypothesis (H_0): Holidays does not impact sales

  Alternative Hypothesis (H_1): Holidays impacts sales

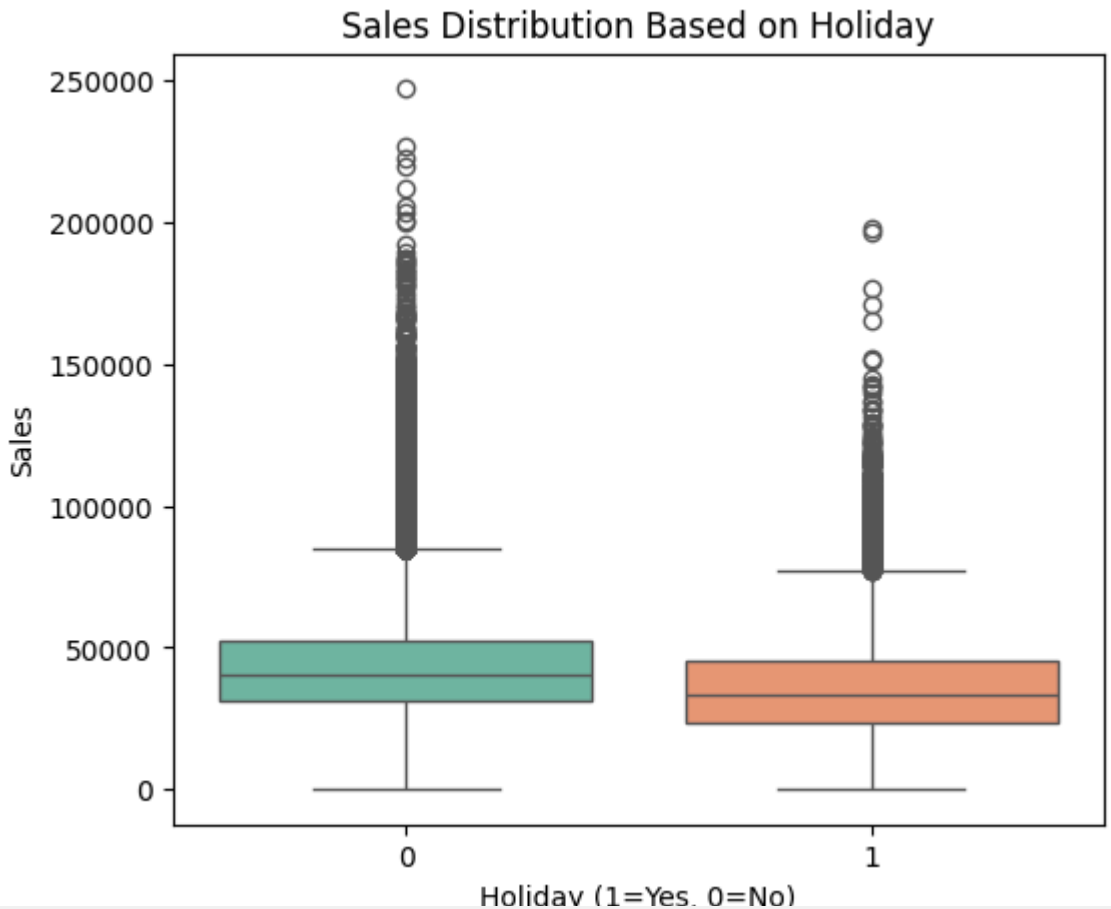
```
# Descriptive Statistics of Sales based on Holiday
sales_holiday_group = df.groupby('Holiday')['Sales'].describe()
print("Descriptive Statistics of Sales Based on Holiday:")
print(sales_holiday_group)
```

```
Descriptive Statistics of Sales Based on Holiday:
             count          mean           std  min       25%       50%  \
Holiday
0         163520.0  43897.288998  18143.426019  0.0  31389.00  40530.00
1          24820.0  35451.878930  18822.332593  0.0  23778.57  33417.54

                75%        max
Holiday
0         52761.000  247215.00
1         45247.125  197840.61
```

```
# Sales vs Holidays
plt.figure(figsize=(6, 5))
sns.boxplot(data=df, x='Holiday', y='Sales', palette='Set2')
plt.title('Sales Distribution Based on Holiday')
```

```
plt.xlabel('Holiday (1=Yes, 0=No)')
plt.ylabel('Sales')
plt.show()
```

```
<ipython-input-12-626627e0ef26>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set
```



```
# T-Test
# Null hypothesis (H0): Holidays does not impact sales
# Alternative hypothesis (H1): Holidays impacts sales
holiday_yes = df[df['Holiday'] == 1]['Sales']
holiday_no = df[df['Holiday'] == 0]['Sales']
t_stat, p_value = stats.ttest_ind(holiday_yes, holiday_no)
print("\nT-Test Results:")
print(f"T-statistic: {t_stat:.3f}")
print(f"P-value: {p_value:.3f}")

# Interpretation
if p_value < 0.05:
    print("Reject the null hypothesis:Holidays impacts sales")
else:
    print("Fail to reject the null hypothesis: Holidays does not impact sales")
```

```
T-Test Results:
T-statistic: -67.990
P-value: 0.000
Reject the null hypothesis:Holidays impacts sales
```

Sales Differences Across Store Types

We can analyse if the sales happening across different store_types are different in order to do following

- Replicate the same business statergies to increase sales
- Collecting feedback from locality customers, what they are seeking for
- Getting to know the demand
- Implementing new advertisement stategies for improving sales

Why ANOVA?

Since we have more than two groups anova helps us to do comparison easily as it is a well-established, efficient, and widely-used method in statistics.

Assumptions of the ANOVA:

- ***Independence :***

  Sales from each store type should be independent of each other

- ***Normality:***

  The sales data for each store type should follow a normal distribution.

- ***Equal variance:***

  Group variances are similar

- ***Homogeneity of Variances:***

  The variance in sales should be roughly equal across the store types Since Levene's test is flexible & robust to Non-Normality, it is used in analysing homoscedasticity.

Null Hypothesis (H_0): There is no significant difference in sales across store types.

Alternative Hypothesis (H_1): There is a statistically significant difference in sales across store types.

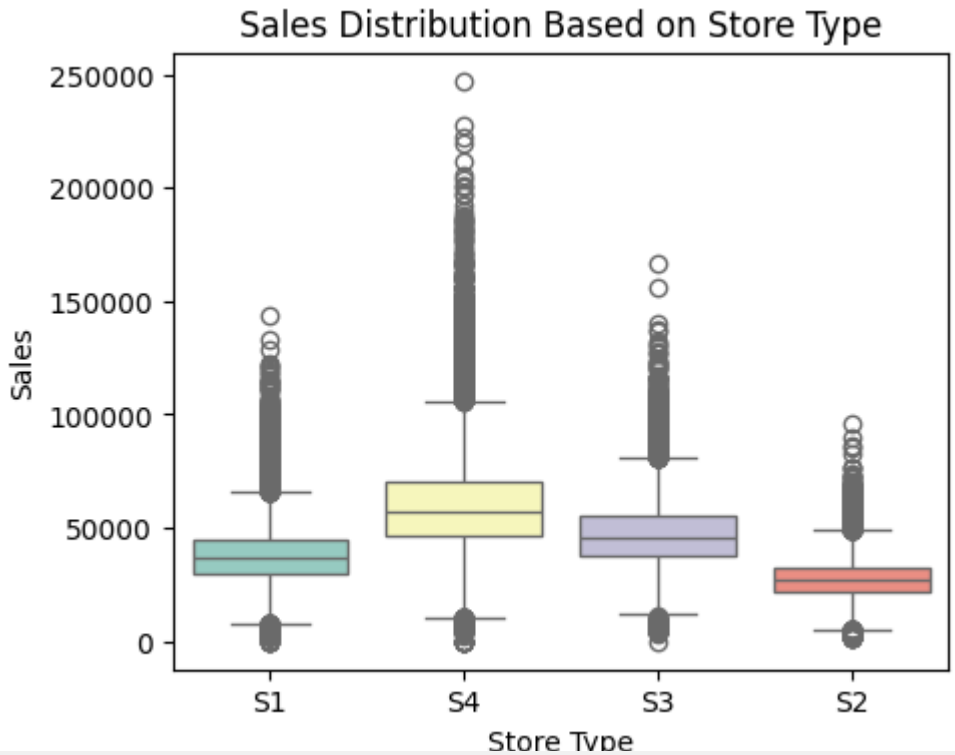
```
# Descriptive Statistics of Sales based on Store Type
sales_store_type_group = df.groupby('Store_Type')['Sales'].describe()
print("Descriptive Statistics of Sales Based on Store Type:")
print(sales_store_type_group)
```

```
Descriptive Statistics of Sales Based on Store Type:
                count          mean           std      min      25%      50%  \
Store_Type
S1           88752.0  37676.511694  12303.151090     0.00  29859.0  36444.0
```

```
S2          28896.0  27530.828222   9168.839395  1748.28  21666.0  26794.5
S3          24768.0  47063.068209  14907.465521     0.00  37656.0  45445.5
S4          45924.0  59945.685926  20750.228035     0.00  46503.0  57075.0

                 75%       max
Store_Type
S1          44439.00  143841.0
S2          32703.90   96363.0
S3          55062.75  166323.0
S4          70326.75  247215.0
```

```
# Sales Distribution Based on Store Type
plt.figure(figsize=(5, 4))
sns.boxplot(data=df, x='Store_Type', y='Sales', palette='Set3')
plt.title('Sales Distribution Based on Store Type')
plt.xlabel('Store Type')
plt.ylabel('Sales')
plt.show()
```

```
<ipython-input-15-e4e7064e4011>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set
```




```
# Levene's test
s1 = df[df['Store_Type'] == 'S1']['Sales']
s2 = df[df['Store_Type'] == 'S2']['Sales']
s3 = df[df['Store_Type'] == 'S3']['Sales']
s4 = df[df['Store_Type'] == 'S4']['Sales']
stat, p_value = levene(s1, s2, s3, s4)
print(f"Levene's Statistic: {stat}")
print(f"p-value: {p_value}")
```

```
if p_value < 0.05:
    print("Variances are heterogeneous")
else:
    print("Variances are homogeneous")
```

```
Levene's Statistic: 5206.219646908859
p-value: 0.0
Variances are heterogeneous
```

```
# ANOVA
# Null hypothesis (H0): There is no significant difference in sales between different store types.
# Alternative hypothesis (H1): There is a significant difference in sales between store types.
store_types = df['Store_Type'].unique()
sales_by_store_type = [df[df['Store_Type'] == store_type]['Sales'] for store_type in store_types]
f_stat, p_value = stats.f_oneway(*sales_by_store_type)
print("\nANOVA Results:")
print(f"F-statistic: {f_stat:.3f}")
print(f"P-value: {p_value:.3f}")
if p_value < 0.05:
    print("\nThere is a statistically significant difference in Sales across Store Types.")
else:
    print("\nThere is no statistically significant difference in Sales across Store Types.")
```

```
ANOVA Results:
F-statistic: 35123.644
P-value: 0.000

There is a statistically significant difference in Sales across Store Types.
```

Regional Sales Variability

We can analyse if the sales happening across different store_types are different in order to do following

- Getting to know the factors affecting sales
- Understanding the distribution of population across region
- Factors affecting sales
- Interest of people
- Identifying Advertisement stategies
- Identifying demand

Why ANOVA?

Since we have more than two groups anova helps us to do comparison easily as it is a well-established, efficient, and widely-used method in statistics.

Assumptions of the ANOVA:

- ***Independence :***

  Sales from each Region should be independent of each other

- ***Normality:***

  The sales data for each Region should follow a normal distribution.

- ***Equal variance:***

  Group variances are similar

- ***Homogeneity of Variances:***

  The variance in sales should be roughly equal across the Regions. Since Levene's test is flexible & robust to Non-Normality, it is used in analysing homoscedasticity.

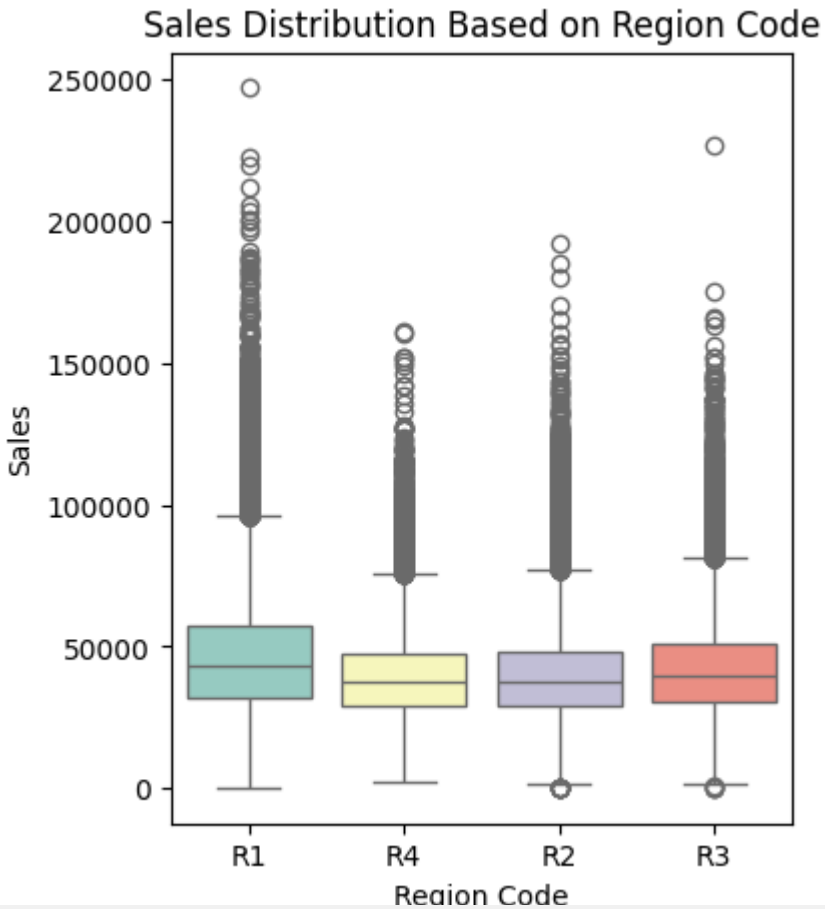
```
Null Hypothesis (H₀):  There is no significant difference in sales across Regions.

Alternative Hypothesis (H₁): There is a statistically significant difference in sales across Regions.
```

```
# Descriptive Statistics of Sales based on Region_Code
sales_region_group = df.groupby('Region_Code')['Sales'].describe()
print("Descriptive Statistics of Sales Based on Region_Code:")
print(sales_region_group)
```

```
Descriptive Statistics of Sales Based on Region_Code:
                count          mean           std     min       25%      50%  \
Region_Code
R1            63984.0  46765.488405  21286.095441     0.0  32091.00  43125.0
R2            54180.0  40054.847344  16468.619529     0.0  29078.25  37548.0
R3            44376.0  42144.517063  16615.730308     0.0  30876.00  39661.5
R4            25800.0  39743.434249  15930.494396  2009.7  29195.64  37474.5

                    75%       max
Region_Code
R1            57624.00  247215.0
R2            48357.00  192156.0
R3            50970.75  227127.0
R4            47796.00  161271.0
```

```
# Sales Distribution based on Region_Code
plt.figure(figsize=(4, 5))
sns.boxplot(data=df, x='Region_Code', y='Sales', palette='Set3')
plt.title('Sales Distribution Based on Region Code')
plt.xlabel('Region Code')
plt.ylabel('Sales')
plt.show()
```

```
<ipython-input-19-833ffbfb8f8d>:3: FutureWarning:

    Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `x` variable to `hue` and set
```




```
# Levene's test
R1 = df[df['Region_Code'] == 'R1']['Sales']
R2 = df[df['Region_Code'] == 'R2']['Sales']
R3 = df[df['Region_Code'] == 'R3']['Sales']
```

```
R4 = df[df['Region_Code'] == 'R4']['Sales']
stat, p_value = levene(R1, R2, R3, R4)
print(f"Levene's Statistic: {stat}")
print(f"p-value: {p_value}")

if p_value < 0.05:
    print("Variances are heterogeneous")
else:
    print("Variances are homogeneous")
```

```
Levene's Statistic: 1235.2684068578194
p-value: 0.0
Variances are heterogeneous
```

```
# ANOVA
# Null hypothesis (H0): There is no significant difference in sales between different regions.
# Alternative hypothesis (H1): There is a significant difference in sales between regions.
regions = df['Region_Code'].unique()
sales_by_region = [df[df['Region_Code'] == region]['Sales'] for region in regions]
f_stat, p_value = stats.f_oneway(*sales_by_region)
print("\nANOVA Results:")
print(f"F-statistic: {f_stat:.3f}")
print(f"P-value: {p_value:.3f}")

# Interpretation
if p_value < 0.05:
    print("\nThere is a statistically significant difference in Sales across Region Codes.")
else:
    print("\nThere is no statistically significant difference in Sales across Region Codes.")
```

```
ANOVA Results:
F-statistic: 1682.426
P-value: 0.000

There is a statistically significant difference in Sales across Region Codes.
```

Correlation between Number of Orders and Sales

As it helps in identifying the relationship between two factors, Sales & Order in our case. It helps in below factor,

- Getting to know the relationship
- Helps improvising marketing
- Helps implementing promotional efforts
- Helps us focusing on increasing the order volume
- Helps to forecast sales based on expected orders

From below we can see the positive correlation between sales & orders, which means if Order increases sales also increases.

```
# Pearson correlation coefficient
correlation = df['#Order'].corr(df['Sales'])
print(f"Pearson Correlation between #Order and Sales: {correlation:.3f}")

# Interpretation
if correlation > 0:
    print("There is a positive correlation between the number of orders and sales.")
elif correlation < 0:
    print("There is a negative correlation between the number of orders and sales.")
else:
    print("There is no correlation between the number of orders and sales.")
```

```
Pearson Correlation between #Order and Sales: 0.942
There is a positive correlation between the number of orders and sales.
```

```
# #Order vs Sales
plt.figure(figsize=(5, 4))
sns.scatterplot(data=df, x='#Order', y='Sales', color='blue')
plt.title('#Order vs Sales')
plt.xlabel('Number of Orders')
plt.ylabel('Sales')
plt.show()
```

Start coding or generate with AI.

Importing necessary libraries & Installing Packages

```
!pip install pmdarima
!pip install scikit-optimize
```

```
Requirement already satisfied: pmdarima in /usr/local/lib/python3.11/dist-packages (2.0.4)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (1.4.2)
Requirement already satisfied: Cython!=0.29.18,!=0.29.31,>=0.29 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (3.0.11)
Requirement already satisfied: numpy>=1.21.2 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (1.26.4)
Requirement already satisfied: pandas>=0.19 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (2.2.2)
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (1.6.1)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (1.13.1)
Requirement already satisfied: statsmodels>=0.13.2 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (0.14.4)
Requirement already satisfied: urllib3 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (2.3.0)
Requirement already satisfied: setuptools!=50.0.0,>=38.6.0 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (75.1.0)
Requirement already satisfied: packaging>=17.1 in /usr/local/lib/python3.11/dist-packages (from pmdarima) (24.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.19->pmdarima) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.19->pmdarima) (2024.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas>=0.19->pmdarima) (2025.1)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=0.22->pmdarima) (3.5.0)
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.11/dist-packages (from statsmodels>=0.13.2->pmdarima) (1.0.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas>=0.19->pmdarima) (1.17.0)
Requirement already satisfied: scikit-optimize in /usr/local/lib/python3.11/dist-packages (0.10.2)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (1.4.2)
Requirement already satisfied: pyaml>=16.9 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (25.1.0)
Requirement already satisfied: numpy>=1.20.3 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (1.26.4)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (1.13.1)
Requirement already satisfied: scikit-learn>=1.0.0 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (1.6.1)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.11/dist-packages (from scikit-optimize) (24.2)
Requirement already satisfied: PyYAML in /usr/local/lib/python3.11/dist-packages (from pyaml>=16.9->scikit-optimize) (6.0.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=1.0.0->scikit-optimize) (3.5.0)
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.graph_objects as go
from plotly.offline import init_notebook_mode, iplot
import gdown

from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.arima.model import ARIMA
from prophet.plot import add_changepoints_to_plot
```

```
from sklearn.preprocessing import MinMaxScaler

from sklearn.base import BaseEstimator, RegressorMixin, TransformerMixin
from sklearn.model_selection import train_test_split, RandomizedSearchCV, TimeSeriesSplit
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score, mean_absolute_percentage_error
from sklearn.ensemble import RandomForestRegressor, VotingRegressor, StackingRegressor
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

from pmdarima import auto_arima
from skopt import gp_minimize
from skopt.space import Real, Integer
from skopt.plots import plot_convergence

import xgboost as xgb
from xgboost import plot_importance

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
```

Downloading necessary.csv file

- Since there are two files we are downloading them separately

```
!gdown "https://drive.google.com/drive/folders/1fBQ1PlWMho3kHF9qXrD0McZNfpJIcbrn"
```

```
/usr/local/lib/python3.11/dist-packages/gdown/parse_url.py:48: UserWarning: You specified a Google Drive link that is not the correct link to download a file. You might want t
  warnings.warn(
Downloading...
From: https://drive.google.com/drive/folders/1fBQ1PlWMho3kHF9qXrD0McZNfpJIcbrn
To: /content/1fBQ1PlWMho3kHF9qXrD0McZNfpJIcbrn
1.23MB [00:00, 1.68MB/s]
```

```
with open('/content/1fBQ1PlWMho3kHF9qXrD0McZNfpJIcbrn', 'r') as file:
    content = file.read()
```

```
import gdown

# Define the file IDs
file_ids = [
    "1ditFn_74EOsGnblf2nXr-Vx-I1mrCG42",
    "1CclWDHsFLfCAuBbAv83_D5_XmeoRwWzl"
```

```
]

# Download each file
for file_id in file_ids:
    url = f"https://drive.google.com/uc?id={file_id}"
    gdown.download(url, quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1ditFn_74EOsGnblf2nXr-Vx-I1mrCG42
To: /content/TRAIN.csv
100%|██████████| 9.33M/9.33M [00:00<00:00, 68.9MB/s]
Downloading...
From: https://drive.google.com/uc?id=1CclWDHsFLfCAuBbAv83_D5_XmeoRwWzl
To: /content/TEST_FINAL.csv
100%|██████████| 849k/849k [00:00<00:00, 44.9MB/s]
```

Reading .csv file

```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df = df1
```

```
df2 = pd.read_csv('TEST_FINAL.csv')
df2.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount
0	T1188341	171	S4	L2	R3	2019-06-01	0	No
1	T1188342	172	S1	L1	R1	2019-06-01	0	No
2	T1188343	173	S4	L2	R1	2019-06-01	0	No
3	T1188344	174	S1	L1	R4	2019-06-01	0	No
4	T1188345	170	S1	L1	R2	2019-06-01	0	No

Understanding Overall data

```
df.describe()
```

Data Cleaning: Address missing values, remove duplicates, and correct inconsistencies in the dataset.

Data cleaning is an essential step in data preprocessing that involves preparing a dataset for analysis by addressing issues such as missing values, duplicates, and inconsistencies.

Addressing Missing Values :

Purpose :

- Since missing data can lead to inaccurate analysis and misleading results we are addressing Missing values.

Implementation :

- By removing or Imputing the missing values

Given **`Data has No Missing Values`**

```
# Check for missing values
print("Missing values in each column:")
print(df1.isnull().sum())
```

```
Missing values in each column:
ID               0
Store_id         0
Store_Type       0
Location_Type    0
Region_Code      0
Date             0
Holiday          0
Discount         0
#Order           0
Sales            0
dtype: int64
```

Removing Duplicates :

Purpose :

- Duplicates can skew the analysis, leading to overrepresentation of certain data points or incorrect conclusions.

Implementation :

- By removing dupicates

Given **`Data has No Duplicate Values`**

```
# Check if there are any duplicate rows
print("\nDuplicate rows:")
print(df1.duplicated().sum())
```

```

Duplicate rows:
0
```

Correcting Inconsistencies :

Purpose :

- Inconsistencies, such as typos, different units of measurement, or varied formatting, can lead to incorrect analysis.

Implementation :

- By Standardizing formats

In Given **`Data Sales & Date columns are standardised`**

```
# Ensuring Sales is a numeric column
df1['Sales'] = pd.to_numeric(df1['Sales'], errors='coerce')

# Ensuring Date column is of datetime type
df1['Date'] = pd.to_datetime(df1['Date'])

df1.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount	#Order	Sales
0	T1000001	1	S1	L3	R1	2018-01-01	1	Yes	9	7011.84
1	T1000002	253	S4	L2	R1	2018-01-01	1	Yes	60	51789.12
2	T1000003	252	S3	L2	R1	2018-01-01	1	Yes	42	36868.20
3	T1000004	251	S2	L3	R1	2018-01-01	1	Yes	23	19715.16
4	T1000005	250	S2	L3	R4	2018-01-01	1	Yes	62	45614.52

Feature Engineering:

Developing new features that could enhance the model's predictive power, such as time-based aggregates (e.g., sales in the last week), ratios, or interaction terms between features.

Feature engineering is the process of creating new features from the existing data to improve the performance of machine learning models. The goal is to make the model better at capturing patterns and relationships.

Time-Based Aggregates:

Purpose :

- Creating time-based features helps the model recognize trends or seasonal patterns.

Implementation :

- By identifying Time periods such as days of weeks etc.

Sales per Order:

Purpose :

- Creating this feature helps to identify individual sales for each order

Implementation :

- By identifying new feature derived by doing a ratio

Create a Rolling Average of Sales:

Purpose :

- A rolling average smooths out helps identify longer-term trends, making it useful for time-series analysis.

Implementation :

- for each day, the average sales of the previous 7 days is calculated, including that day. This helps reduce the impact of daily volatility

Discount Effectiveness:

Purpose :

- Discounts often affect sales, and this feature can help the model understand if the increase (or decrease) in sales is due to the discount.

Implementation :

- Implementing a simple flag that indicates whether a discount was offered during a given period.

```
# Time-Based Aggregates: Extract Month and Day of Week
df1['Month'] = df1['Date'].dt.month
df1['Day_of_Week'] = df1['Date'].dt.dayofweek  # Monday = 0, Sunday = 6
df1.head()

# Feature for Sales per Order
df1['Sales_per_Order'] = df1['Sales'] / df1['#Order']

# Rolling Average of Sales (7-day rolling window)
df1['Sales_Rolling_7'] = df1['Sales'].rolling(window=7, min_periods=1).mean()


# Discount Effectiveness
df1['Discount_Active'] = df1['Discount'].apply(lambda x: 1 if x == 'Yes' else 0)
```

Interaction Term:

Purpose :

- It allows the model to capture how the combined effect of these features might influence the target variable.

Implementation :

- It is implemented by combining the Store_Type and Region_Code variables.

Lagged Features :

Purpose :

- Lagged features help the model use past data to predict future events.

Implementation :

- By creating a new feature that represents the sales from a previous time period, the previous day

Rolling Window:

Purpose :

- A rolling sum can provide a sense of the cumulative sales activity over a longer period.

Implementation :

- For each day, calculating the sum of sales for the past 30 days, including the current day

Total Orders per Store:

Purpose :

- Aggregating by Store Type can help capture the overall performance and behavior of each store type

Implementation :

- This basically aggregates the total number of orders for each store type

```
# Interaction term
df1['Store_Type_Region_Interaction'] = df1['Store_Type'] + "_" + df1['Region_Code']

# Lagged Features (previous day sales)
df1['Lagged_Sales'] = df1['Sales'].shift(1)  # Previous day's sales


# Rolling Window: Sum of sales for the past 30 days
df1['Sales_Rolling_30'] = df1['Sales'].rolling(window=30, min_periods=1).sum()

# Total Orders per Store Type
df1['Total_Orders_by_Store_Type'] = df1.groupby('Store_Type')['#Order'].transform('sum')
df1.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount	#Order	Sales	Month	Day_of_Week	Sales_per_Order	Sales_Rolling_7	Discount_Active
0	T1000001	1	S1	L3	R1	2018-01-01	1	Yes	9	7011.84	1	0	779.093333	7011.840	1
1	T1000002	253	S4	L2	R1	2018-01-01	1	Yes	60	51789.12	1	0	863.152000	29400.480	1
2	T1000003	252	S3	L2	R1	2018-01-01	1	Yes	42	36868.20	1	0	877.814286	31889.720	1
3	T1000004	251	S2	L3	R1	2018-01-01	1	Yes	23	19715.16	1	0	857.180870	28846.080	1
4	T1000005	250	S2	L3	R4	2018-01-01	1	Yes	62	45614.52	1	0	735.718065	32199.768	1

Data Transformation:

Scale numerical features and encode categorical variables to prepare the data for modeling. Techniques like normalization or standardization and one-hot encoding or label encoding can be applied.

Data transformation is a crucial step in data preprocessing that prepares raw data for modeling. This process involves scaling numerical features and encoding categorical variables so that they are in a suitable form for machine learning algorithms.

Scaling Numerical Features:

Purpose :

- If numerical features are not scaled models may give disproportionate weight to larger values, leading to biased or inefficient learning.

Implementation :

- Done using Normalization & Standardization

Encoding Categorical Variables:

Purpose :

- Many machine learning algorithms cannot directly work with categorical variables, so these variables are converted into a numerical format so that the model can process them.

Implementation :

- Done using One-Hot Encoding that creates binary columns for each category in the feature.

Further, Data is splitted into train-test split, where a dataset is divided into two subsets: one for training the model and the other for testing its performance. which helps assess how well the model generalizes to new, unseen data and prevents overfitting.

```
df1.columns
```

```
Index(['ID', 'Store_id', 'Store_Type', 'Location_Type', 'Region_Code', 'Date',
       'Holiday', 'Discount', '#Order', 'Sales', 'Month', 'Day_of_Week',
       'Sales_per_Order', 'Sales_Rolling_7', 'Discount_Active',
       'Store_Type_Region_Interaction', 'Lagged_Sales', 'Sales_Rolling_30',
       'Total_Orders_by_Store_Type'],
      dtype='object')
```

```
# Encoding Categorical Variables
df_encoded = pd.get_dummies(df1, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount', 'Store_Type_Region_Interaction'], drop_first=True)

# Scale Numerical Features
scaler = StandardScaler()
df_encoded[['Sales', '#Order']] = scaler.fit_transform(df_encoded[['Sales', '#Order']])
# After encoding and scaling
print("\nData after transformation:")
print(df_encoded.head(2))
```

```
Data after transformation:
         ID  Store_id       Date  Holiday    #Order     Sales  Month  \
0  T1000001         1 2018-01-01        1 -1.943251 -1.938189      1
1  T1000002       253 2018-01-01        1 -0.269328  0.487889      1

   Day_of_Week  Sales_per_Order  Sales_Rolling_7  ...  \
0            0       779.093333          7011.84  ...
1            0       863.152000         29400.48  ...

   Store_Type_Region_Interaction_S2_R3  Store_Type_Region_Interaction_S2_R4  \
0                                False                                False
1                                False                                False

   Store_Type_Region_Interaction_S3_R1  Store_Type_Region_Interaction_S3_R2  \
0                                False                                False
1                                False                                False

   Store_Type_Region_Interaction_S3_R3  Store_Type_Region_Interaction_S3_R4  \
0                                False                                False
1                                False                                False

   Store_Type_Region_Interaction_S4_R1  Store_Type_Region_Interaction_S4_R2  \
0                                False                                False
1                                 True                                False

   Store_Type_Region_Interaction_S4_R3  Store_Type_Region_Interaction_S4_R4
0                                False                                False
1                                False                                False

[2 rows x 40 columns]
```

Train-Test Split: Divide the data into training and testing sets to ensure the model can be objectively evaluated.

The train-test split is a fundamental technique used in machine learning used to evaluate the performance of a model.

Training Set:

Purpose :

- This portion of the data is used to train the model, i.e., to learn patterns and make predictions.

Test Set:

Purpose :

- This portion is used to evaluate the model's performance after training, giving an unbiased estimate of how well the model will perform on new, unseen data.

Implementation :

- Splitting the data in ration 80:20 for training and testing and implementing train_test_split method accordingly.

```
# Assuming df_encoded is the transformed dataset after scaling and encoding
# Define the features (X) and target variable (y)
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']

# Perform train-test split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("Training Features Shape:", X_train.shape)
print("Testing Features Shape:", X_test.shape)
print("Training Target Shape:", y_train.shape)
print("Testing Target Shape:", y_test.shape)
```

```
Training Features Shape: (150672, 37)
Testing Features Shape: (37668, 37)
Training Target Shape: (150672,)
Testing Target Shape: (37668,)
```

2. Model Selection

Model selection is a critical step in the machine learning pipeline where we choose the appropriate model (algorithm) to solve the problem. It involves evaluating and selecting the best model based on its performance on the data. The process requires balancing factors such as accuracy, complexity, interpretability, and training time, among others.

Why Model Selection?

To address below

- Performance Variability
- Trade-offs
- Generalization

Baseline Model: Starting with a simple model to establish a baseline performance. Linear regression is a common choice.

A baseline model is a simple model that is easy to implement and provides a starting point for assessing the performance of more complex models. It gives us an initial benchmark that helps understanding the "minimum" performance we should expect. Below are the main purpose of it,

- Initial Performance Estimate
- Quick Evaluation
- Avoid Overfitting

Why Linear Regression?

Purpose :

- Linear regression is one of the simplest models and assumes a linear relationship between the input features and the target variable which will be ease for implementation. Since the coefficients of a linear regression model directly indicate the relationship between each feature and the target. This makes it easier to explain the results. Additionally its Quick to Train

Implementation :

- By providing Train & Test data as input model makes the prediction by making the values fit a straight line.

```
R-squared (0.92): The Linear Regression model explains 92% of the variance in the target variable, indicating a strong
Mean Absolute Error (3678.6): On average, the model's predictions are off by 3678.6, which is acceptable based on the s
```

```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df1
```

```
dt['Date'] = pd.to_datetime(dt['Date'])
dt['month'] = dt['Date'].dt.month
dt.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount	#Order	Sales	month
0	T1000001	1	S1	L3	R1	2018-01-01	1	Yes	9	7011.84	1
1	T1000002	253	S4	L2	R1	2018-01-01	1	Yes	60	51789.12	1
2	T1000003	252	S3	L2	R1	2018-01-01	1	Yes	42	36868.20	1
3	T1000004	251	S2	L3	R1	2018-01-01	1	Yes	23	19715.16	1
4	T1000005	250	S2	L3	R4	2018-01-01	1	Yes	62	45614.52	1

```
# Assuming the original dataframe dt
# Encoding Categorical Features using One-Hot Encoding
df_encoded = pd.get_dummies(dt, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount'], drop_first=True)

# Define Features (X) and Target (y)
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']

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

# Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make]ing predictions
y_pred = model.predict(X_test)

# Evaluation metrics
r2 = r2_score(y_test, y_pred)
mae = mean_absolute_error(y_test, y_pred)

# Output the evaluation metrics
print(f"R-squared: {r2:.4f}")
print(f"Mean Absolute Error: {mae:.4f}")
```

```
R-squared: 0.9234
Mean Absolute Error: 3678.6912
```

Feature vs. Target (Sales)

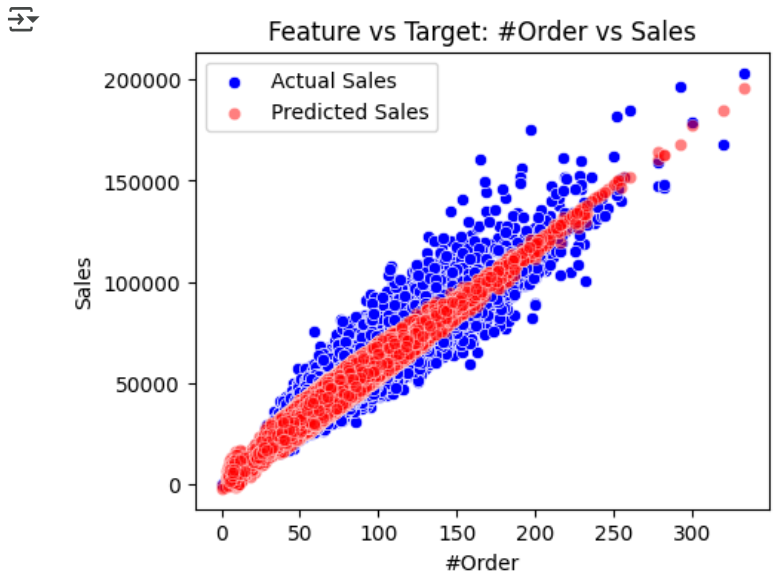
Purpose :

- This scatter plot visualizes the relationship between one feature (in this case, #Order) and the target variable (Sales). we can compare how well the model's predictions align with the actual sales.

Interpretation :

- If the red points (predicted sales) are close to the blue points (actual sales), it indicates a good model fit.

```
plt.figure(figsize=(5, 4))
sns.scatterplot(x=X_test['#Order'], y=y_test, label="Actual Sales", color='blue')
sns.scatterplot(x=X_test['#Order'], y=y_pred, label="Predicted Sales", color='red', alpha=0.5)
plt.title('Feature vs Target: #Order vs Sales')
plt.xlabel('#Order')
plt.ylabel('Sales')
plt.legend()
plt.show()
```



Actual vs. Predicted Values

Purpose :

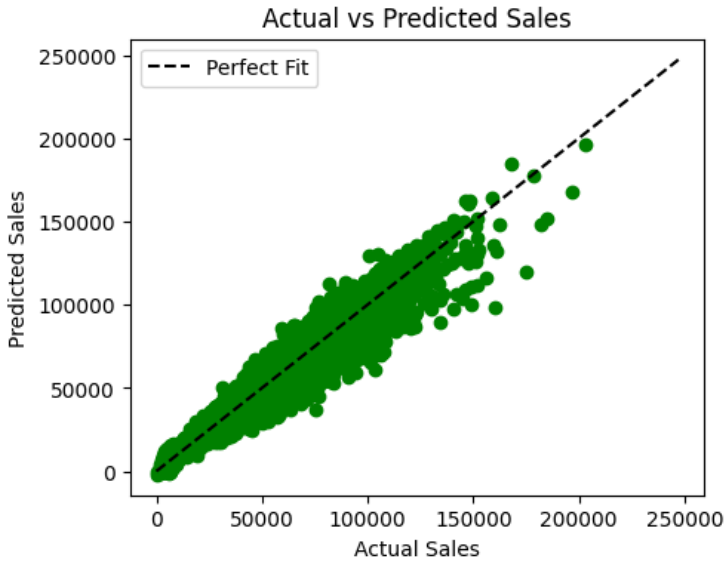
- This scatter plot compares the actual Sales values (y_test) against the predicted values (y_pred).

Interpretation :

- Ideally, the points should lie along the diagonal line, which represents the case where predicted values are equal to actual values. Any significant deviation from this line indicates a higher error.

```
# Actual vs Predicted Values
plt.figure(figsize=(5, 4))
plt.scatter(y_test, y_pred, color='green')
plt.plot([y.min(), y.max()], [y.min(), y.max()], '--k', label='Perfect Fit')
plt.xlabel('Actual Sales')
plt.ylabel('Predicted Sales')
```

```
plt.title('Actual vs Predicted Sales')
plt.legend()
plt.show()
```



Residual Plot (Error Plot)

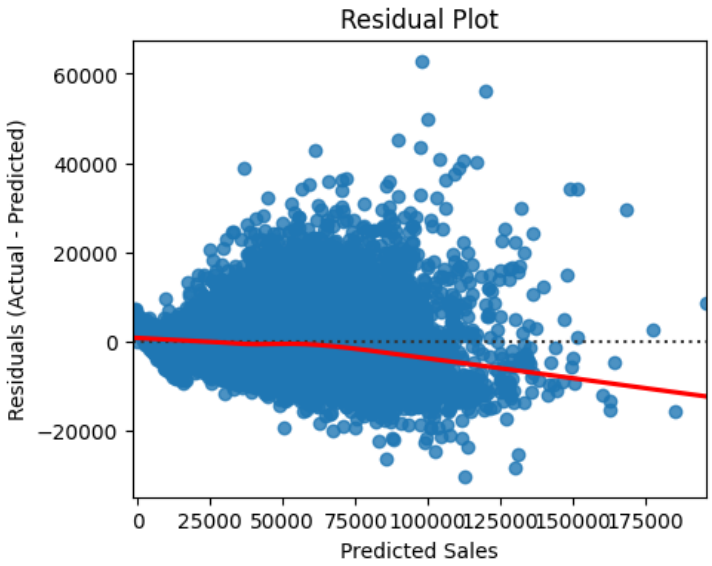
Purpose :

- A residual plot shows the difference between the actual and predicted values. It's helpful to evaluate if the model's errors are randomly distributed.

Interpretation :

- If the residuals are randomly dispersed around zero, this suggests that the model has captured the underlying patterns well. Any clear patterns or trends in the residuals could indicate that the model needs improvement.

```
# Residual Plot (Error Plot)
residuals = y_test - y_pred
plt.figure(figsize=(5, 4))
sns.residplot(x=y_pred, y=residuals, lowess=True, line_kws={'color': 'red'})
plt.title('Residual Plot')
plt.xlabel('Predicted Sales')
plt.ylabel('Residuals (Actual - Predicted)')
plt.show()
```



Model Performance Metrics

Purpose :

- A bar plot showing the key evaluation metrics (R-squared and MAE).

Interpretation :

- This gives a quick overview of the model's performance. High R-squared indicates that the model explains a large portion of the variance in sales, while MAE gives a sense of the average prediction error.

```
# Model Performance Metrics Bar Plot
metrics = {'R-squared': r2, 'MAE': mae}
plt.figure(figsize=(5, 4))
plt.bar(metrics.keys(), metrics.values(), color=['blue', 'orange'])
plt.title('Model Evaluation Metrics')
plt.ylabel('Score')
plt.show()
```

Complex Models: Explore more sophisticated models to improve accuracy. Potential models include:

These models are often more powerful and flexible, capable of capturing complex patterns and relationships in the data. The goal is to improve accuracy and enhance the model's predictive power by moving beyond the basic approach. Below are the main purpose of it,

- Better Accuracy
- Non-linearity
- Flexibility
- Handling More Complex Data

Implementation

- By providing Train & Test data as input model makes the prediction by making the values fit a straight line.

1. Time Series Model: Applying ARIMA for data forecasting:

Purpose:

ARIMA (AutoRegressive Integrated Moving Average) is one of the most widely used models for forecasting time series data.

- It is particularly effective when the data shows trends or seasonality and is based on past observations.
- ARIMA helps predict future values by analyzing past values (autoregressive), the differences between consecutive values (integrated), and the moving average of past errors.

Components:

The ARIMA model is denoted as ARIMA(p, d, q), where:

- p is the number of lag observations in the autoregressive part (AR).
- d is the number of times the data is differenced to make it stationary.
- q is the size of the moving average window (MA).

Once the model is fitted it can be used to make predictions and forecasts.

```
dt.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount	#Order	Sales	month
0	T1000001	1	S1	L3	R1	2018-01-01	1	Yes	9	7011.84	1
1	T1000002	253	S4	L2	R1	2018-01-01	1	Yes	60	51789.12	1
2	T1000003	252	S3	L2	R1	2018-01-01	1	Yes	42	36868.20	1
3	T1000004	251	S2	L3	R1	2018-01-01	1	Yes	23	19715.16	1
4	T1000005	250	S2	L3	R4	2018-01-01	1	Yes	62	45614.52	1

```
# Ensuring the date format
df_model = dt
df_model['Date'] = pd.to_datetime(df_model['Date'])
df_model.set_index('Date', inplace=True)


# Preparing data for stationary tests
date_range = df_model.index
sales_data = df_model['Sales']
data = pd.DataFrame({'date': date_range, 'Sales': sales_data})
data.index = pd.to_datetime(data['date'])
data = data.groupby(data.index).median()
data = data.drop('date', axis=1)
data
```

	Sales
date	
2018-01-01	39982.38
2018-01-02	50628.00
2018-01-03	47988.00
2018-01-04	52410.00
2018-01-05	60078.00
...	...
2019-05-27	43821.00
2019-05-28	47451.00
2019-05-29	42933.00
2019-05-30	41955.00
2019-05-31	40677.12

516 rows × 1 columns

Dickey-Fuller test

- Using statistical analysis - Dickey-Fuller Test we can determine whether a given time series is stationary or non-stationary.
- In time series analysis, we will find stationarity(constant over time) / non-stationary(show trends or seasonality, which need to be addressed before modeling.)

```
def adf_test(series):
    result = adfuller(series)
    print('ADF Statistic:', result[0])
    print('p-value:', result[1])
    print('Critical Values:')
    for key, value in result[4].items():
        print(f'\t{key}: {value}')
    if result[1] < 0.05:
      print("Reject the null hypothesis: The series is stationary.")
    else:
      print("Fail to reject the null hypothesis: The series is non-stationary.")

adf_test(data['Sales'])
```

```
ADF Statistic: -3.4590273622955423
p-value: 0.009103860345178972
Critical Values:
```

```
        1%: -3.4436029548776395
        5%: -2.867384756137026
        10%: -2.5698830308597813
Reject the null hypothesis: The series is stationary.
```

Start coding or generate with AI.

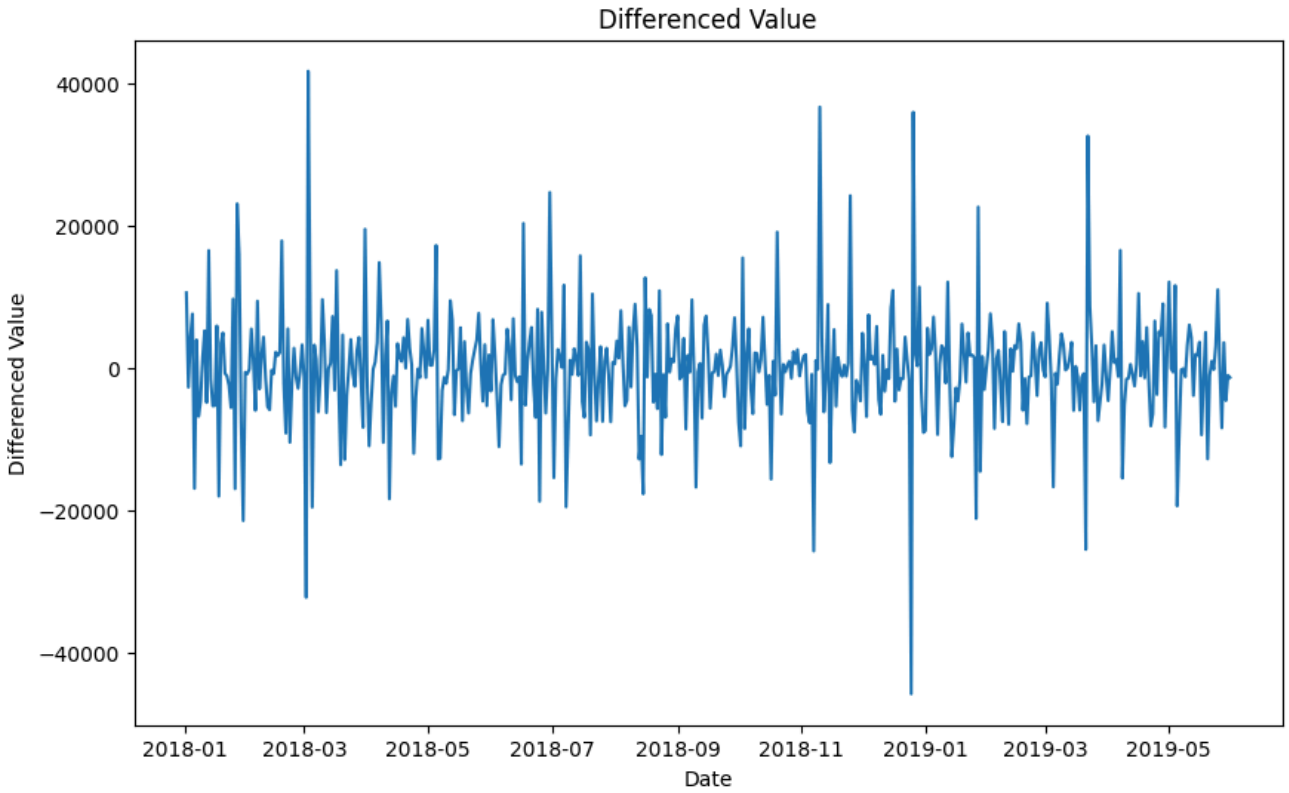
Differencing

From the data considered the series was already stationary, still the differencing is done and got the below p-values and other factors,

```
ADF Statistic: -3.4590273622955423
p-value: 0.009103860345178972
Critical Values:
    1%: -3.4436029548776395
    5%: -2.867384756137026
    10%: -2.5698830308597813
```

- Using Differencing technique for time series analysis to transform a non-stationary series into a stationary one.
- Differencing helps to remove trends and seasonality from a time series, making it more suitable for modeling.

```
# Although the series is stationary below differencing & decomposition models are done for reference

# Differencing to make the series stationary
data['Value_diff'] = data['Sales'] - data['Sales'].shift(1)

# Drop NaN values after differencing
data.dropna(inplace=True)

# Plot the differenced series
plt.figure(figsize=(10, 6))
plt.plot(data['Value_diff'])
plt.title('Differenced Value')
plt.xlabel('Date')
plt.ylabel('Differenced Value')
plt.show()

# Check stationarity again
adf_test(data['Value_diff'])
```

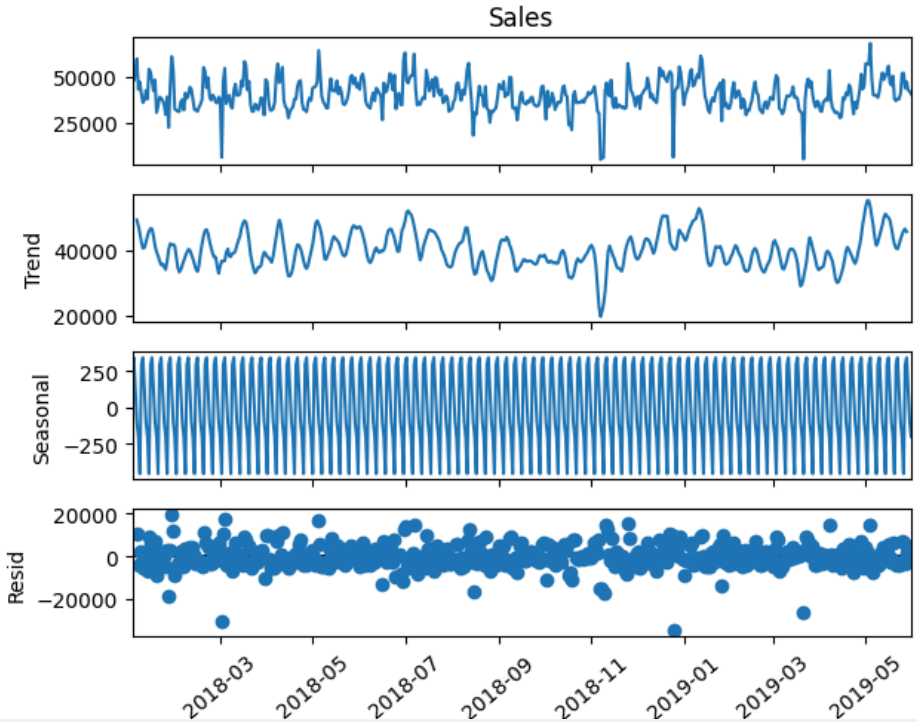



```
ADF Statistic: -9.26720946595902
p-value: 1.3568130861411842e-15
Critical Values:
        1%: -3.4436298692815304
        5%: -2.867396599893435
        10%: -2.5698893429241916
```

Decomposition

- Time series decomposition is helps to understand the underlying patterns and improving forecasting accuracy & improve decision-making..
- where the time series split into components: trend, seasonality, and residual (or noise).

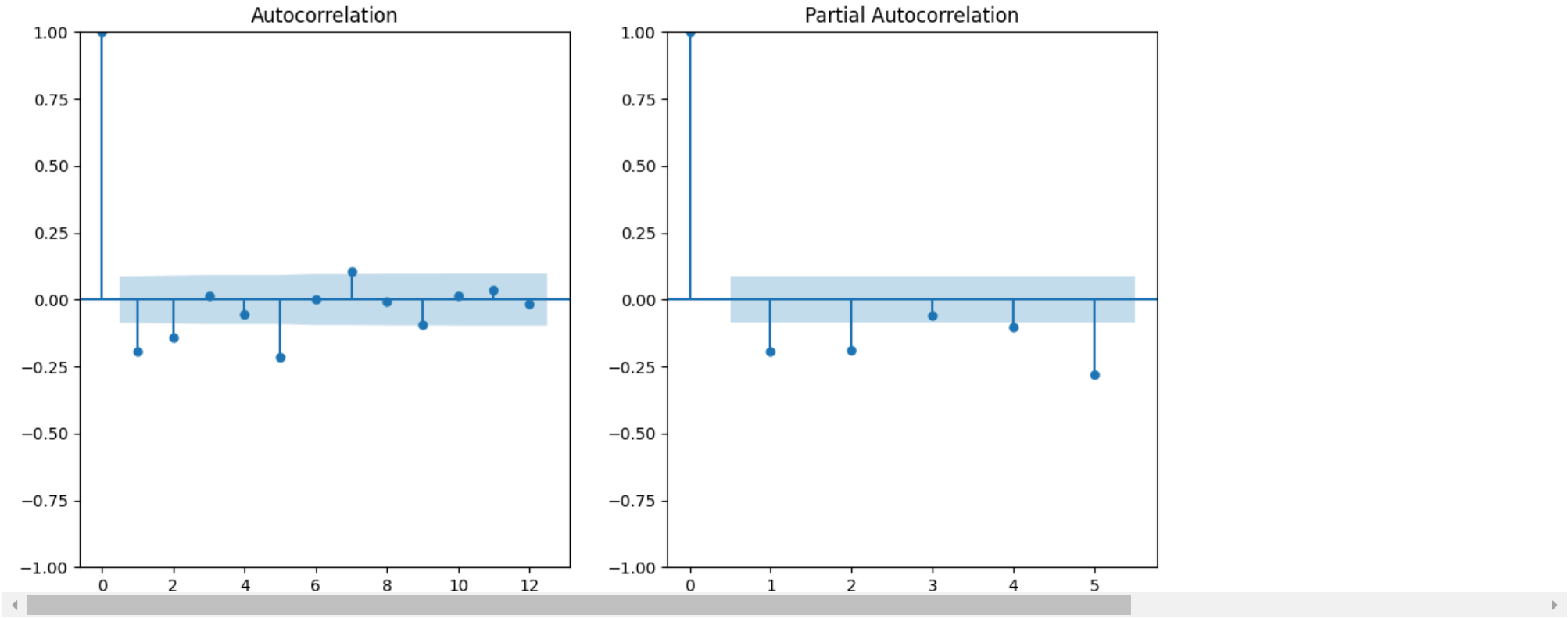
```
# Decomposition and its vizualisation
result = seasonal_decompose(data['Sales'], model='additive', period=6)
result.plot()
plt.xticks(rotation = 40)
plt.show()
```



ACF and PACF

- In order to understand the relationships between an observation and its past values &
- Identifying appropriate time series models & understanding the structure of the data below plots are implemented.

```
# Plot ACF and PACF
plt.figure(figsize=(12, 6))

plt.subplot(121)
plot_acf(data['Value_diff'], lags=12, ax=plt.gca())

plt.subplot(122)
plot_pacf(data['Value_diff'], lags=5, ax=plt.gca())

plt.show()
```




```
# Perform Augmented Dickey-Fuller test
result = adfuller(data['Sales'])
print(f"ADF Statistic: {result[0]}")
print(f"p-value: {result[1]}")
```

```
ADF Statistic: -3.461280550155466
p-value: 0.009038986906826817
```

Executing ARIMA model

```
# Fit AutoARIMA model
model_autoarima = auto_arima(data['Sales'], seasonal=False, stepwise=True, trace=True)

# Print the best model
print(model_autoarima.summary())

# Forecast next 6 periods
forecast_autoarima = model_autoarima.predict(n_periods=6)
```

```
print("AutoARIMA forecast:", forecast_autoarima)
```

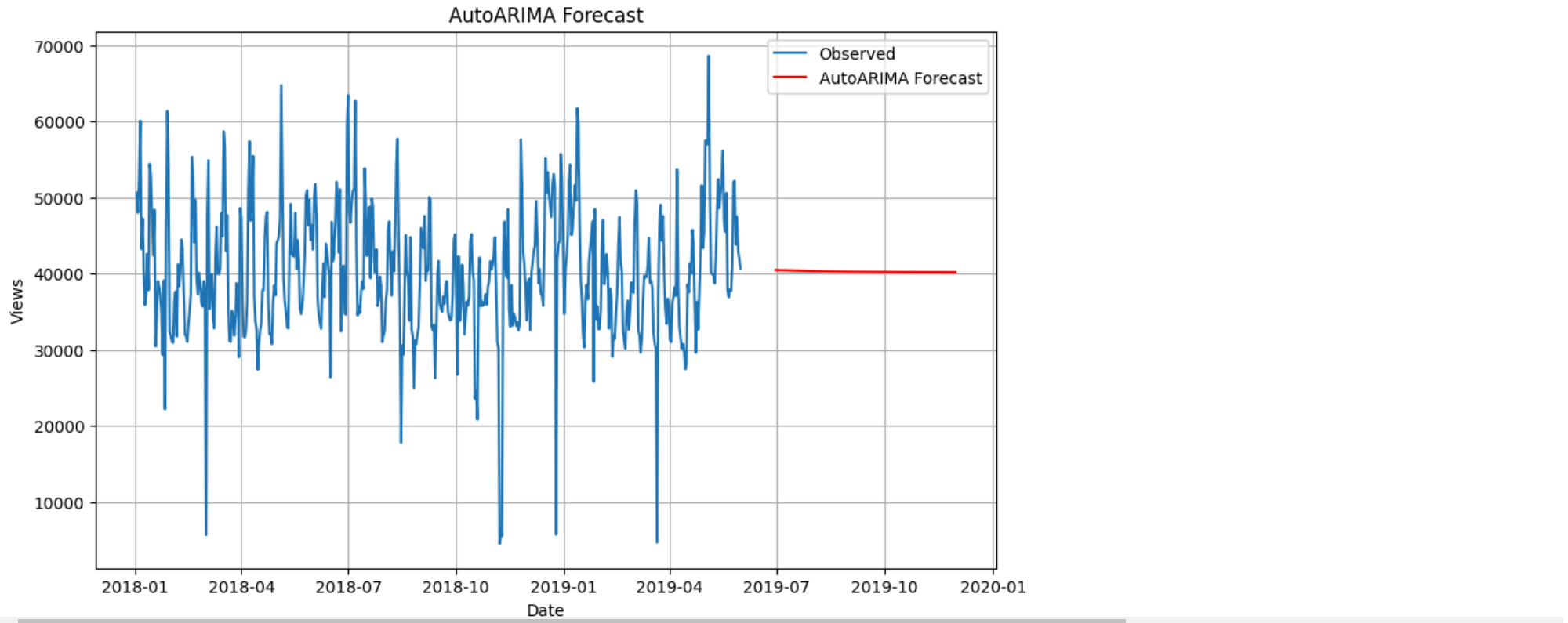
```
Performing stepwise search to minimize aic
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

 ARIMA(2,0,2)(0,0,0)[0]             : AIC=inf, Time=1.08 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=12404.057, Time=0.02 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=10723.584, Time=0.04 sec
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

 ARIMA(0,0,1)(0,0,0)[0]             : AIC=11842.168, Time=0.16 sec
 ARIMA(2,0,0)(0,0,0)[0]             : AIC=10707.362, Time=0.09 sec
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

 ARIMA(3,0,0)(0,0,0)[0]             : AIC=inf, Time=0.15 sec
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:
```

```
'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

 ARIMA(2,0,1)(0,0,0)[0]             : AIC=10617.903, Time=0.67 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=10696.921, Time=0.11 sec
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:
```

Vizual representation of ARIMA forecast

```
# Create a new time index for the forecasted period
forecast_index = pd.date_range(start=data.index[-1] + pd.Timedelta(days=30), periods=6, freq='M')

# Plot the results
plt.figure(figsize=(10, 6))

# Plot historical data
plt.plot(data.index, data['Sales'], label='Observed')

# Plot forecasted data
plt.plot(forecast_index, forecast_autoarima, label='AutoARIMA Forecast', color='red')

# Add labels and legend
plt.title('AutoARIMA Forecast')
plt.xlabel('Date')
plt.ylabel('Views')
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

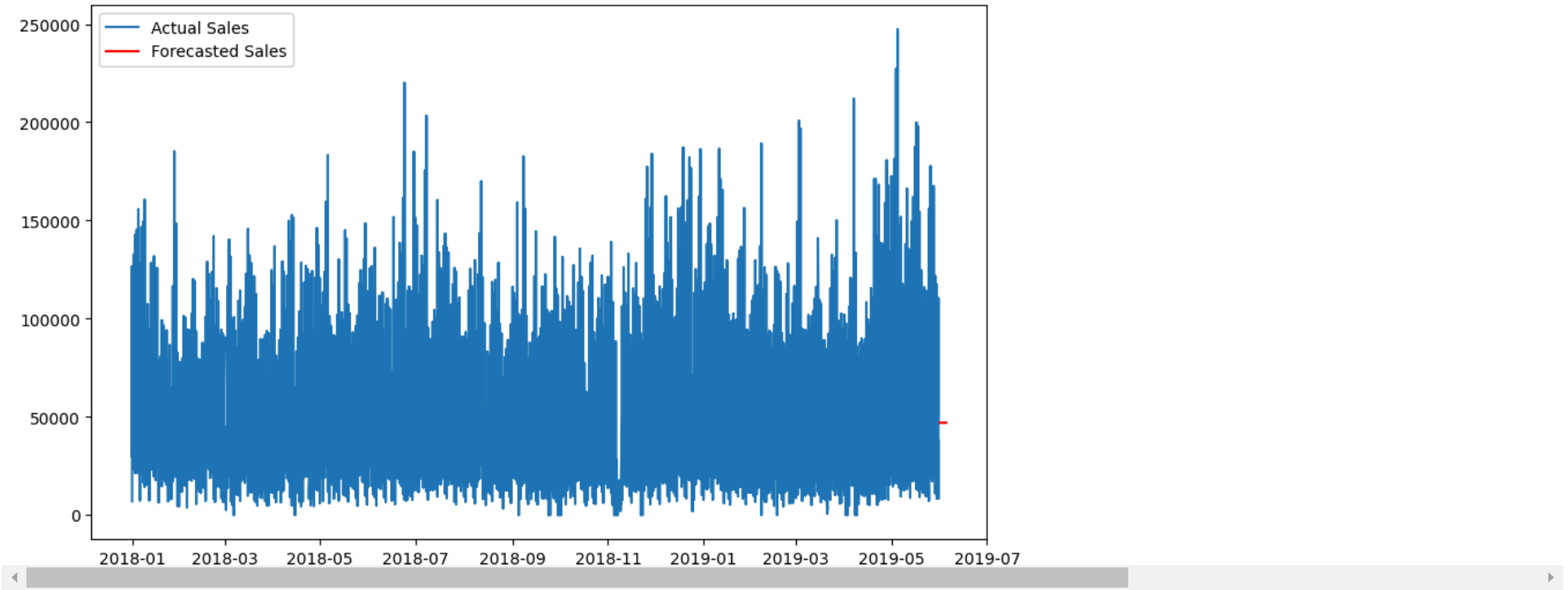
```
<ipython-input-196-520b3fd30881>:2: FutureWarning:

'M' is deprecated and will be removed in a future version, please use 'ME' instead.
```




```
#dt['month'] = pd.to_datetime(dt['month'])
#dt.set_index('month', inplace=True)
sales = dt['Sales']

# Fit an ARIMA model (order=(p, d, q))
# p = lag order, d = differencing order, q = order of moving average
model = ARIMA(sales, order=(1, 1, 1))
model_fit = model.fit()

# Make predictions
forecast = model_fit.forecast(steps=5)

# Plot for actual vs. forecasted sales
plt.figure(figsize=(10,6))
plt.plot(sales, label='Actual Sales')
plt.plot(pd.date_range(sales.index[-1], periods=6, freq='D')[1:], forecast, label='Forecasted Sales', color='red')
```

```
plt.legend()
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:
A date index has been provided, but it has no associated frequency information and so will be ignored when e.g. forecasting.
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:
A date index has been provided, but it has no associated frequency information and so will be ignored when e.g. forecasting.
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:
A date index has been provided, but it has no associated frequency information and so will be ignored when e.g. forecasting.
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:837: ValueWarning:
No supported index is available. Prediction results will be given with an integer index beginning at `start`.
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:837: FutureWarning:
No supported index is available. In the next version, calling this method in a model without a supported index will result in an exception.
```



SARIMA

- SARIMA (Seasonal AutoRegressive Integrated Moving Average) an extension of the ARIMA model which explicitly accounts for seasonality in time series data.
- It is mainly used to model time series data that exhibits both non-seasonal and seasonal behaviors using components such as (AutoRegressive, Integrated, Moving Average & Seasonality)

```
# Aggregate duplicates
date_range = df_model.index
sales_data = df_model['Sales']
data = pd.DataFrame({'date': date_range, 'Sales': sales_data})
data.index = pd.to_datetime(data['date'])
data = data.groupby(data.index).median()
data = data.drop('date', axis=1)
data
```

date	Sales
2018-01-01	39982.38
2018-01-02	50628.00
2018-01-03	47988.00
2018-01-04	52410.00
2018-01-05	60078.00
...	...
2019-05-27	43821.00
2019-05-28	47451.00
2019-05-29	42933.00
2019-05-30	41955.00
2019-05-31	40677.12

516 rows × 1 columns

Dickey-Fuller test

- Using statistical analysis - Dickey-Fuller Test we can determine whether a given time series is stationary or non-stationary.
- In time series analysis, we will find stationarity(constant over time) / non-stationary(show trends or seasonality, which need to be addressed before modeling.)

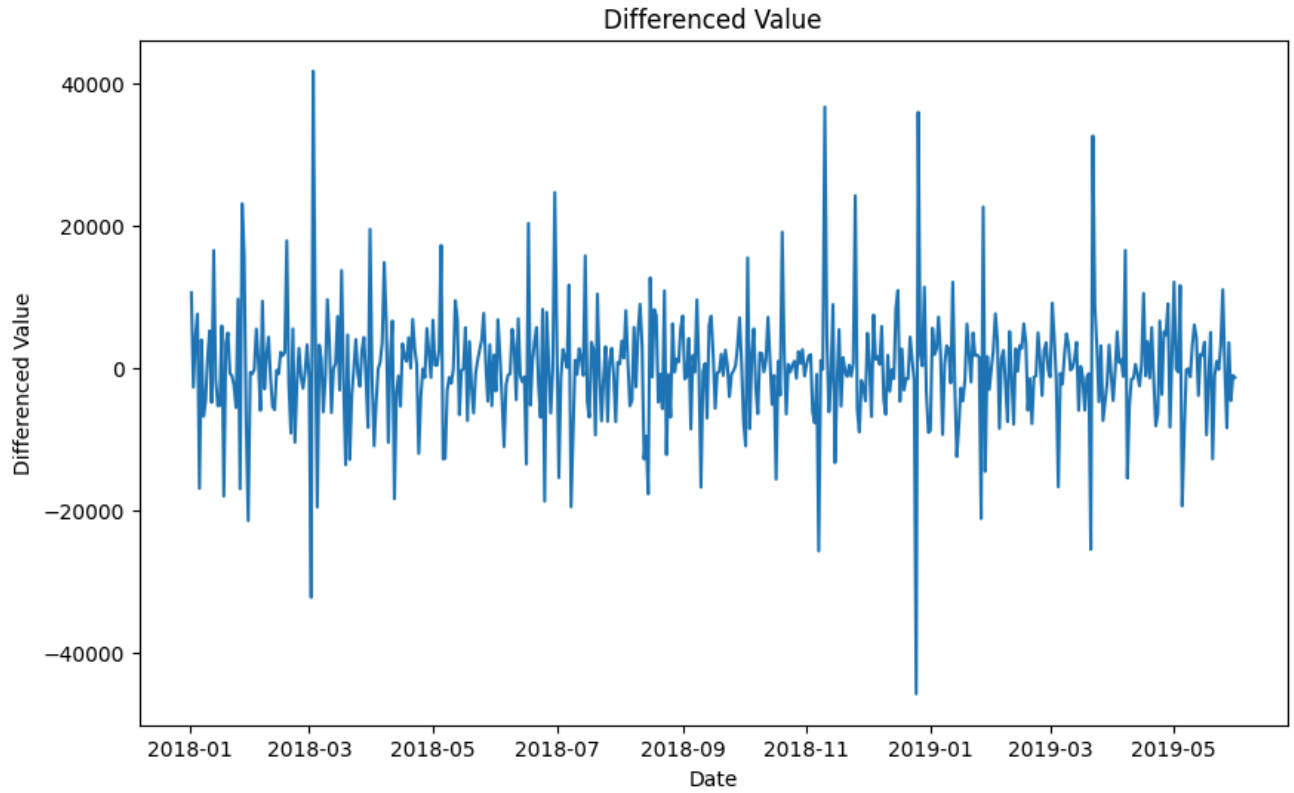
```
def adf_test(series):
    result = adfuller(series)
    print('ADF Statistic:', result[0])
    print('p-value:', result[1])
    print('Critical Values:')
    for key, value in result[4].items():
        print(f'\t{key}: {value}')

adf_test(data['Sales'])

if result[1] < 0.05:
    print("Reject the null hypothesis: The series is stationary.")
else:
    print("Fail to reject the null hypothesis: The series is non-stationary.")
```

```
ADF Statistic: -3.4590273622955423
p-value: 0.009103860345178972
Critical Values:
        1%: -3.4436029548776395
        5%: -2.867384756137026
        10%: -2.5698830308597813
Reject the null hypothesis: The series is stationary.
```

Differencing

- Using Differencing technique for time series analysis to transform a non-stationary series into a stationary one.
- Differencing helps to remove trends and seasonality from a time series, making it more suitable for modeling.

```
# Differencing to make the series stationary
data['Value_diff'] = data['Sales'] - data['Sales'].shift(1)

# Drop NaN values after differencing
data.dropna(inplace=True)

# Plot the differenced series
plt.figure(figsize=(10, 6))
plt.plot(data['Value_diff'])
plt.title('Differenced Value')
plt.xlabel('Date')
plt.ylabel('Differenced Value')
plt.show()

# Check stationarity again
adf_test(data['Value_diff'])
```

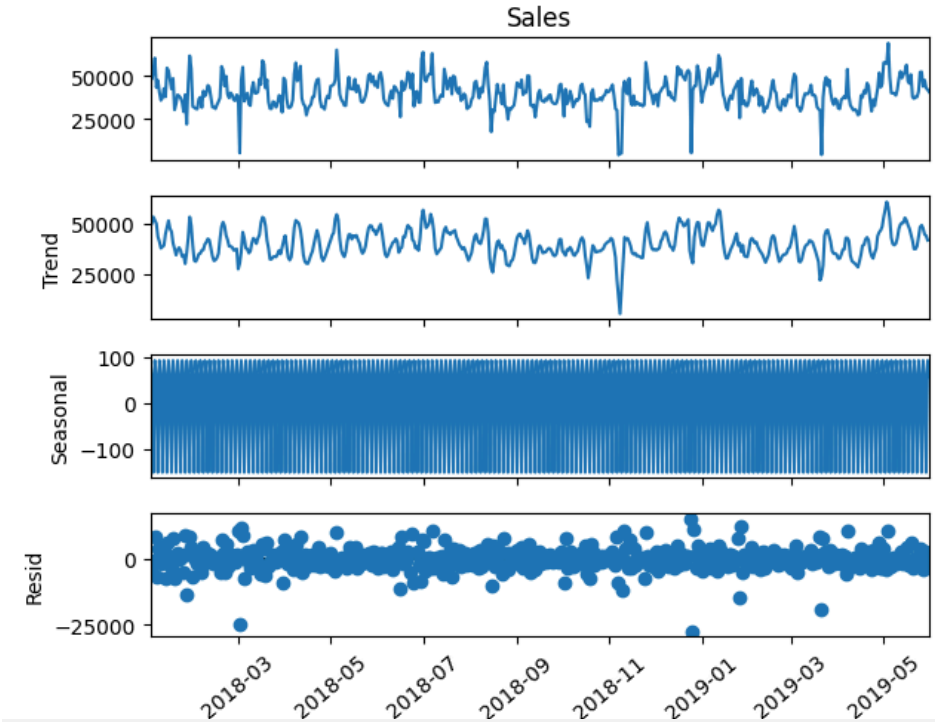



```
ADF Statistic: -9.26720946595902
p-value: 1.3568130861411842e-15
Critical Values:
        1%: -3.4436298692815304
        5%: -2.867396599893435
```

Decomposition

- Time series decomposition is helps to understand the underlying patterns and improving forecasting accuracy.
- where the time series split into components: trend, seasonality, and residual (or noise).

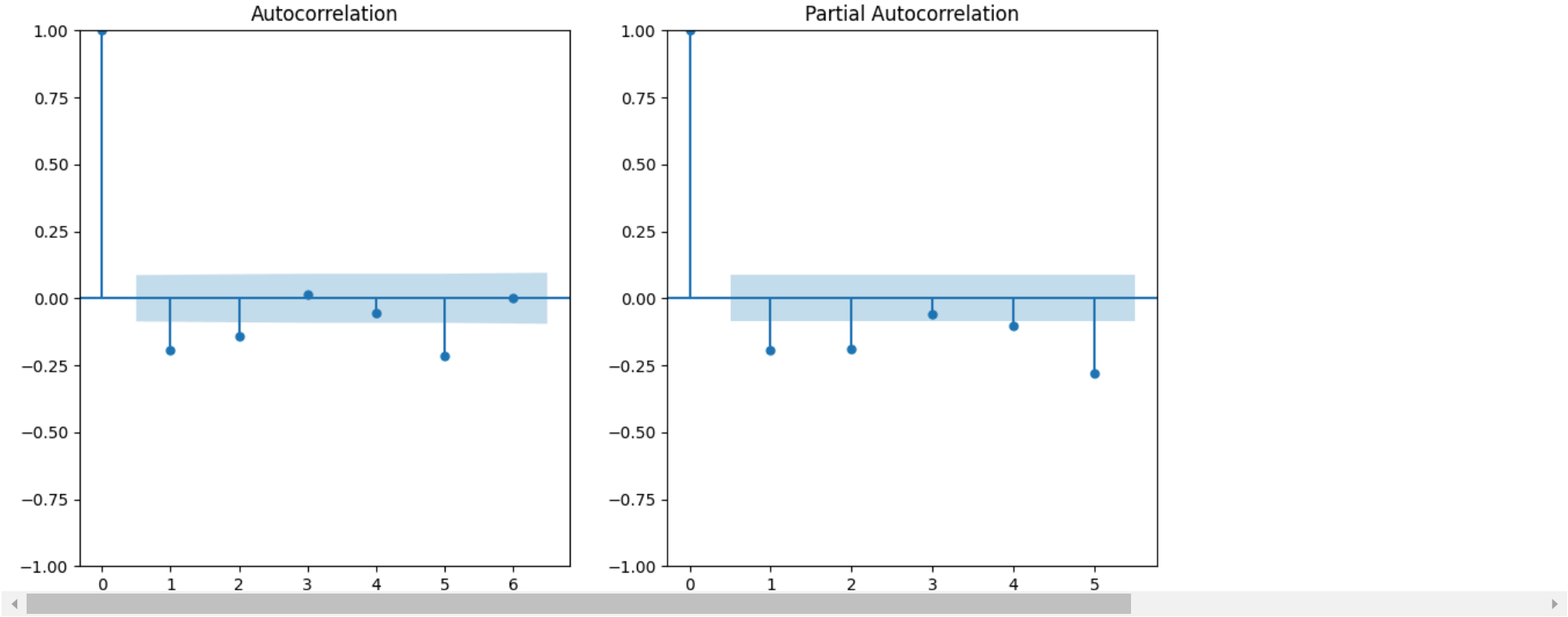
```
# Decomposition and its vizualisation
result = seasonal_decompose(data['Sales'], model='additive', period=3)
result.plot()
plt.xticks(rotation = 40)
plt.show()
```



ACF and PACF

- In order to understand the relationships between an observation and its past values &
- Identifying appropriate time series models (like ARIMA) & understanding the structure of the data below plots are implemented.

```
# Plot ACF and PACF
plt.figure(figsize=(12, 6))

plt.subplot(121)
plot_acf(data['Value_diff'], lags=6, ax=plt.gca())

plt.subplot(122)
plot_pacf(data['Value_diff'], lags=5, ax=plt.gca())

plt.show()
```



Executing SARIMAX model

```
# Fit SARIMAX model with exogenous variable
model = SARIMAX(data['Sales'], exog=data['Value_diff'], order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
model_fit = model.fit(disp=False)

# Print the summary of the model
print(model_fit.summary())
```

```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:

No frequency information was provided, so inferred frequency D will be used.

/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:

No frequency information was provided, so inferred frequency D will be used.

                                     SARIMAX Results
==========================================================================================
```

```
Dep. Variable:                             Sales   No. Observations:                  515
Model:             SARIMAX(1, 1, 1)x(1, 1, 1, 12)   Log Likelihood               -4999.586
Date:                            Sun, 02 Feb 2025   AIC                          10011.172
Time:                                    06:25:27   BIC                          10036.483
Sample:                                01-02-2018   HQIC                         10021.102
                                     - 05-31-2019
Covariance Type:                               opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Value_diff     0.4978      0.015     33.786      0.000       0.469       0.527
ar.L1         -0.2803      0.115     -2.447      0.014      -0.505      -0.056
ma.L1          0.8015      0.091      8.773      0.000       0.622       0.981
ar.S.L12      -0.0711      0.082     -0.866      0.386      -0.232       0.090
ma.S.L12      -0.8537      0.071    -11.944      0.000      -0.994      -0.714
sigma2      4.469e+07   1.14e-09    3.9e+16      0.000    4.47e+07    4.47e+07
===================================================================================
Ljung-Box (L1) (Q):                   0.96   Jarque-Bera (JB):               368.06
Prob(Q):                              0.33   Prob(JB):                         0.00
Heteroskedasticity (H):               0.62   Skew:                            -0.44
Prob(H) (two-sided):                  0.00   Kurtosis:                         7.10
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 2.76e+32. Standard errors may be unstable.
```

```
model_sarima = SARIMAX(data['Sales'], order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
sarima_fit = model_sarima.fit(disp=False)
print(sarima_fit.summary())

# Forecast the next 6 months
forecast_sarima = sarima_fit.get_forecast(steps=6)
forecast_index = pd.date_range(start=df.index[-1] , periods=6, freq='M')
```

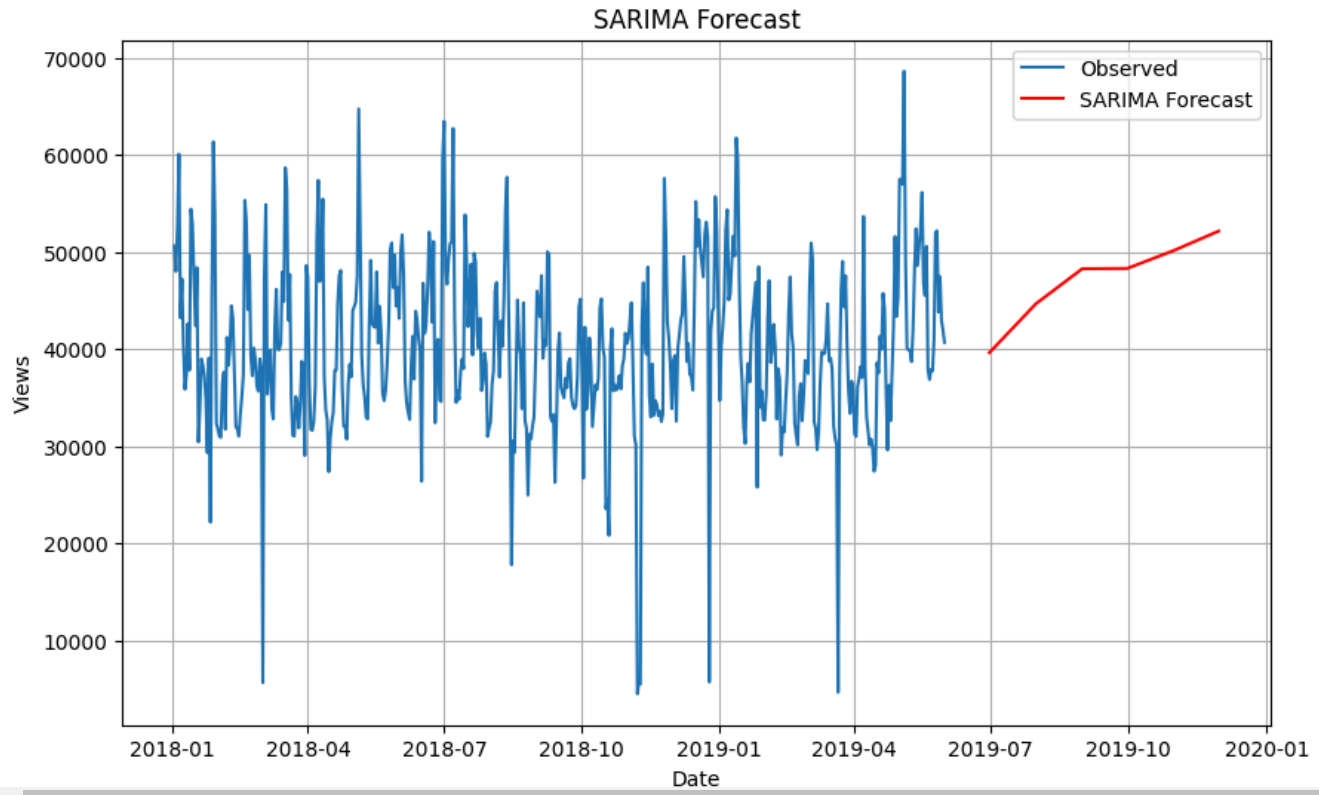
```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:

No frequency information was provided, so inferred frequency D will be used.

/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:

No frequency information was provided, so inferred frequency D will be used.

                                     SARIMAX Results
===================================================================================================
Dep. Variable:                               Sales   No. Observations:                  515
Model:               SARIMAX(1, 1, 1)x(1, 1, 1, 12)   Log Likelihood               -5184.373
Date:                              Sun, 02 Feb 2025   AIC                          10378.746
Time:                                      06:25:31   BIC                          10399.839
Sample:                                  01-02-2018   HQIC                         10387.022
                                       - 05-31-2019
Covariance Type:                                 opg
==============================================================================
```

```
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.5021      0.047     10.749      0.000       0.411       0.594
ma.L1         -0.9049      0.031    -29.488      0.000      -0.965      -0.845
ar.S.L12      -0.0792      0.043     -1.852      0.064      -0.163       0.005
ma.S.L12      -0.9085      0.025    -36.654      0.000      -0.957      -0.860
sigma2      5.264e+07   1.15e-10   4.58e+17      0.000    5.26e+07    5.26e+07
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):               538.15
Prob(Q):                              0.97   Prob(JB):                         0.00
Heteroskedasticity (H):               0.77   Skew:                            -0.62
Prob(H) (two-sided):                  0.10   Kurtosis:                         7.92
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 2.01e+33. Standard errors may be unstable.
<ipython-input-204-420951a78713>:7: FutureWarning:

'M' is deprecated and will be removed in a future version, please use 'ME' instead.
```

Vizual representation of SARIMA forecast

```
# Create a new time index for the forecasted period
forecast_index = pd.date_range(start=data.index[-1] + pd.Timedelta(days=30), periods=6, freq='M')

# Plot the results
plt.figure(figsize=(10, 6))

# Plot historical data
plt.plot(data.index, data['Sales'], label='Observed')

# Plot forecasted data
plt.plot(forecast_index, forecast_sarima.predicted_mean, label='SARIMA Forecast', color='red')

# Add labels and legend
plt.title('SARIMA Forecast')
plt.xlabel('Date')
plt.ylabel('Views')
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

```
<ipython-input-205-c57737bb49ce>:2: FutureWarning:

'M' is deprecated and will be removed in a future version, please use 'ME' instead.
```



fb Prophet

- A reliable forecast which handles time series data with strong seasonal patterns and missing data.
- It handles Trend, Seasonality, Holiday, Errors etc.

```
# Preparing data to comply with model
data1 = pd.DataFrame({'ds': date_range, 'y': sales_data})
data2 = data1.groupby(data1['ds']).median()
data2.reset_index(inplace=True)


from prophet import Prophet

m = Prophet()
```

```
m.fit(data2)

# Making Future predictions
future = m.make_future_dataframe(periods=12*3,
                                 freq='D')
future
```

```
INFO:prophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=True to override this.
INFO:prophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
DEBUG:cmdstanpy:input tempfile: /tmp/tmpmaan3r5d/dzpovowh.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpmaan3r5d/0y5xct00.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.11/dist-packages/prophet/stan_model/prophet_model.bin', 'random', 'seed=57530', 'data', 'file=/tmp/tmpmaan3r5d/dzpovowh.
06:25:31 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
06:25:31 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```

	ds
0	2018-01-01
1	2018-01-02
2	2018-01-03
3	2018-01-04
4	2018-01-05
...	...
547	2019-07-02
548	2019-07-03
549	2019-07-04
550	2019-07-05
551	2019-07-06

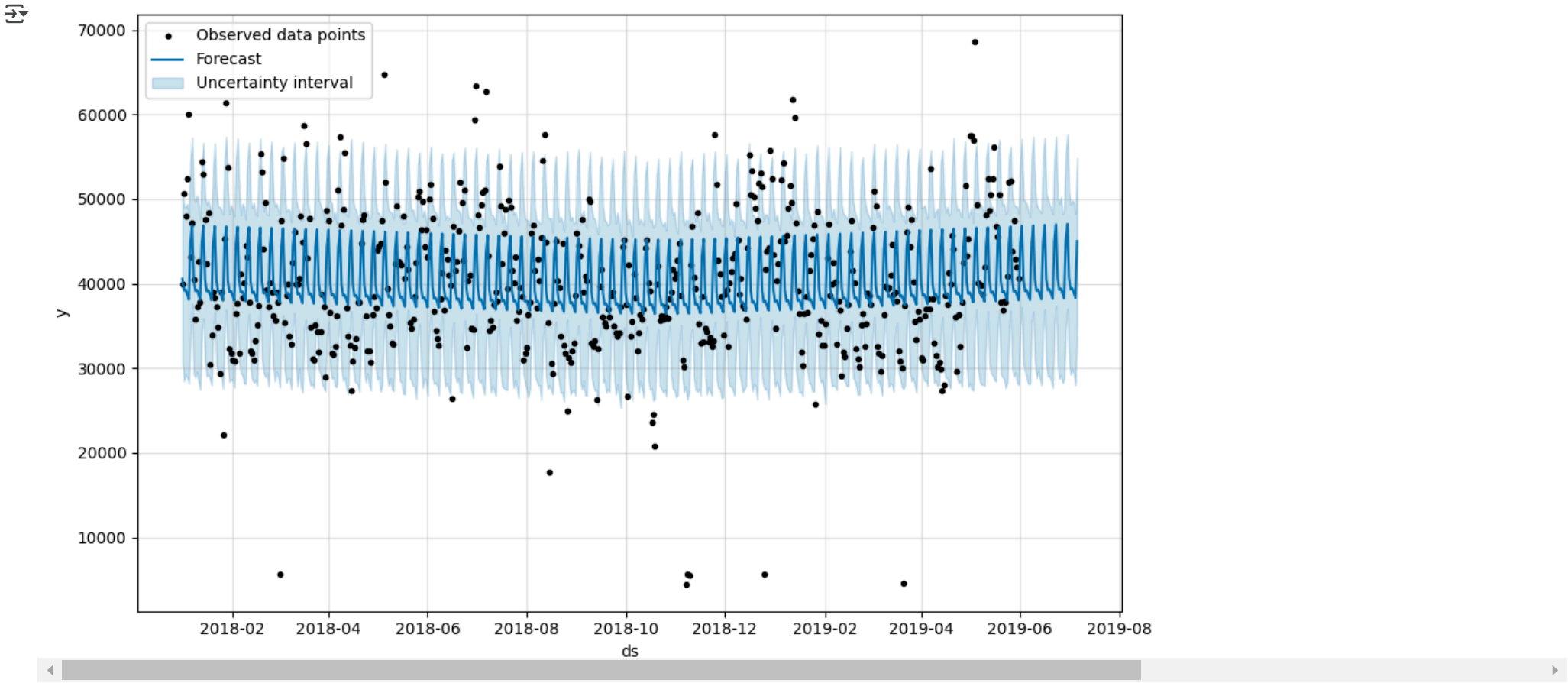
552 rows × 1 columns

Interpretation:

- The yhat values represent the model's forecasted values for future periods (e.g., predicted sales).
- The uncertainty intervals (yhat_lower and yhat_upper) show the model's confidence in the forecast. The larger the interval, the higher the uncertainty.
- The trend line shows the underlying pattern of the data over time. It helps us understand whether the model expects the value to increase, decrease, or remain stable.

The plot visually demonstrates how the forecast fits into the historical data and extends into the future, along with uncertainty bounds.

```
forecast = m.predict(future)
forecast[['ds', 'yhat', 'yhat_lower',
          'yhat_upper', 'trend',
          'trend_lower', 'trend_upper']].tail()

fig1 = m.plot(forecast, include_legend=True)
```



Interpretation:

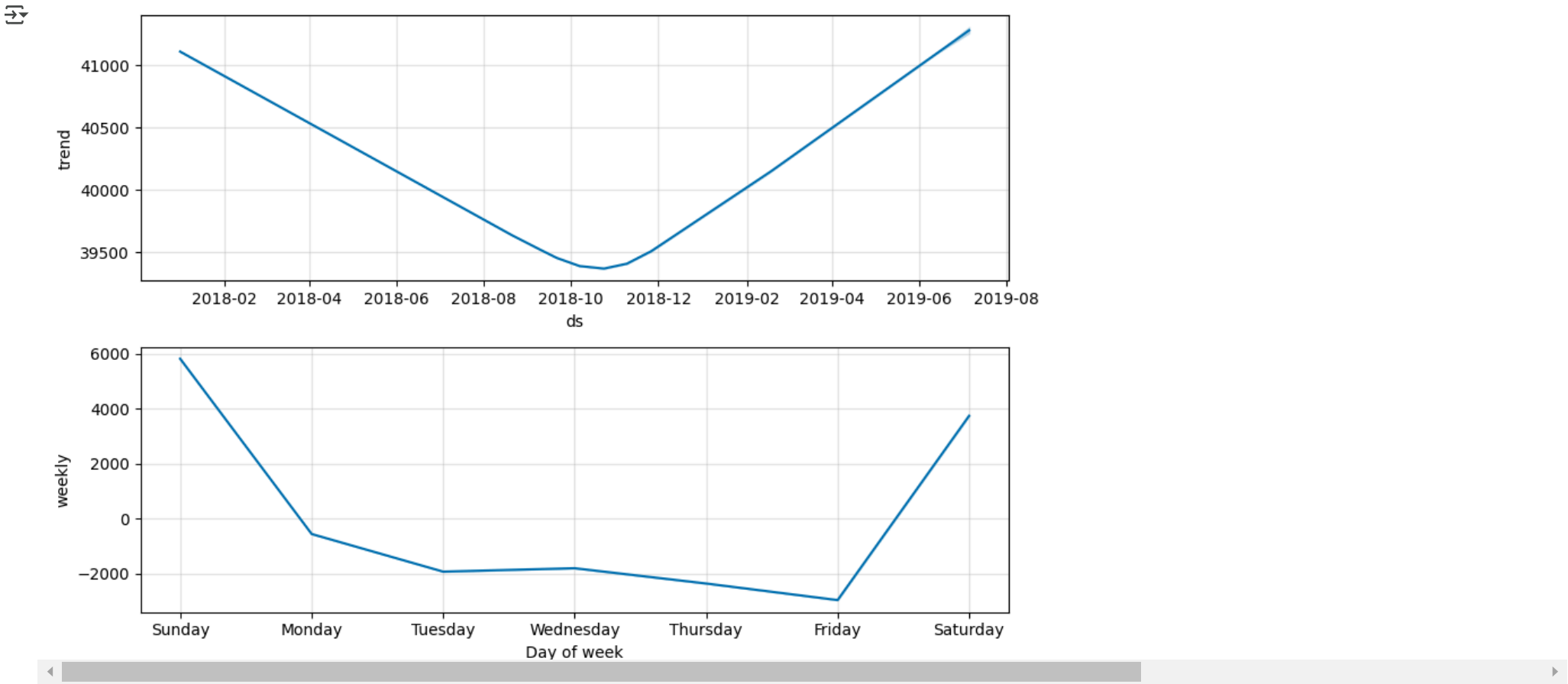
Trend:

- It implies the overall long-term pattern the model has learned based on historical data.
- The trend line helps us understand the broader movement in the data over time, such as the general growth.

Seasonality:

- The seasonality component shows how the data behaves in recurring cycles, here in weekly patterns.
- We could see that Sales increases during holidays rather than working days

```
fig2 = m.plot_components(forecast)
```



Interpretation:

Forecast Line:

The plot will show the predicted values for future periods

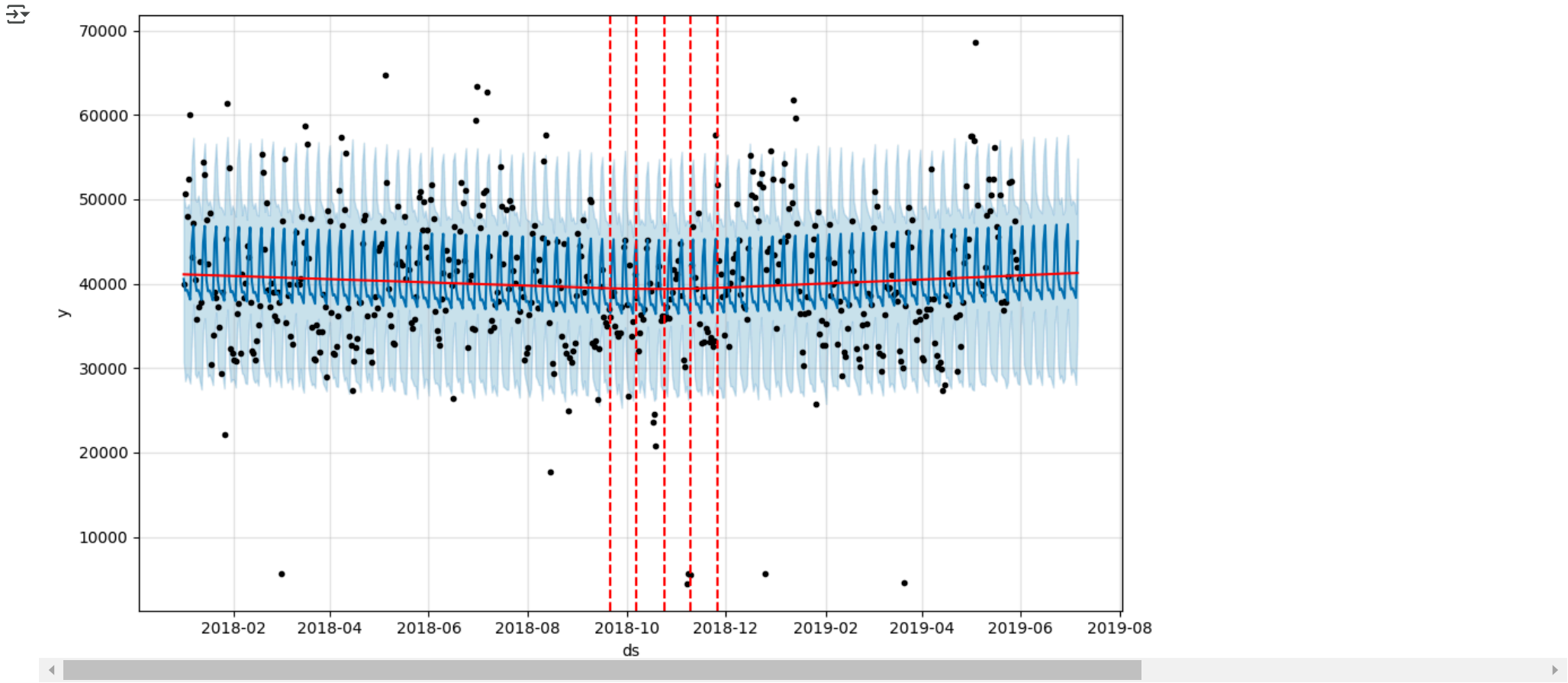
Uncertainty Interval:

The shaded area represents the range within which the actual future values are expected to fall, with the model's confidence.

Changepoints:

The vertical lines indicate when a change in trend occurs. These points are critical to understanding where significant shifts in the data are happening.

```
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(),
                             m, forecast)
```



2. Tree-Based Model: Applying Random Forest to improve performance by capturing complex relationships between features:

Purpose:

It is an ensemble learning method that builds multiple decision trees and combines their predictions to improve overall performance.

- It is particularly effective for capturing complex relationships between features in the data that might be difficult for simpler models to identify.

- It also offers advantages like reducing overfitting, improving accuracy, and handling both numerical and categorical features well.

Components:

- Ensemble of Trees
- Bootstrap Aggregating (Bagging)
- Feature Randomness
- Voting

Implementation:

- Prepare the Data
- Split the Data into Training and Testing Sets
- Train the Random Forest Model
- Make Predicitions
- Evaluate the model (R2 & MAE)

Implementation :

By providing Train & Test data as input model makes the prediction by making the values fit a straight line.

```
R-squared (0.93): The RandomForest model explains 93% of the variance in the target variable, indicating a strong fit and good predictive power
Mean Absolute Error (3177.6): On average, the model's predictions are off by 3177.6, which is acceptable based on the scale of the sales data
```

```
dt.head()
```

Date	ID	Store_id	Store_Type	Location_Type	Region_Code	Holiday	Discount	#Order	Sales	month
2018-01-01	T1000001	1	S1	L3	R1	1	Yes	9	7011.84	1
2018-01-01	T1000002	253	S4	L2	R1	1	Yes	60	51789.12	1
2018-01-01	T1000003	252	S3	L2	R1	1	Yes	42	36868.20	1
2018-01-01	T1000004	251	S2	L3	R1	1	Yes	23	19715.16	1
2018-01-01	T1000005	250	S2	L3	R4	1	Yes	62	45614.52	1

```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df = df1
```

```
dt.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount	#Order	Sales
0	T1000001	1	S1	L3	R1	2018-01-01	1	Yes	9	7011.84
1	T1000002	253	S4	L2	R1	2018-01-01	1	Yes	60	51789.12
2	T1000003	252	S3	L2	R1	2018-01-01	1	Yes	42	36868.20
3	T1000004	251	S2	L3	R1	2018-01-01	1	Yes	23	19715.16
4	T1000005	250	S2	L3	R4	2018-01-01	1	Yes	62	45614.52

```
# Encoding Categorical Features using One-Hot Encoding
df_encoded = pd.get_dummies(dt, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount'], drop_first=True)

# Define Features (X) and Target (y)
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Predict
y_pred_rf = rf_model.predict(X_test)

# Evaluate the model's performance
r2_rf = r2_score(y_test, y_pred_rf)
mae_rf = mean_absolute_error(y_test, y_pred_rf)

# Output the evaluation metrics
print(f"Random Forest - R-squared: {r2_rf:.4f}")
print(f"Random Forest - Mean Absolute Error: {mae_rf:.4f}")
```

```
Random Forest - R-squared: 0.9385
Random Forest - Mean Absolute Error: 3177.6702
```

Actual vs. Predicted Values

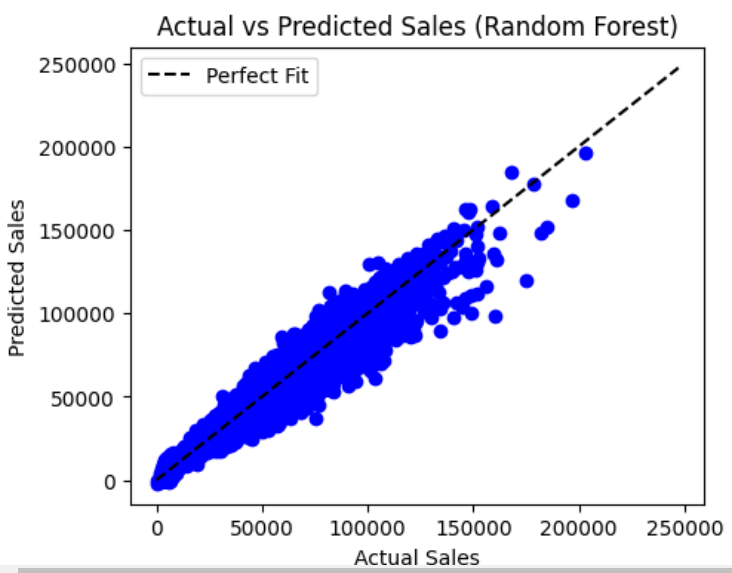
Purpose :

- This scatter plot compares the actual Sales values (y_test) against the predicted values (y_pred).

Interpretation :

- Ideally, the points should lie along the diagonal line, which represents the case where predicted values are equal to actual values. Any significant deviation from this line indicates a higher error.

```
#  Actual vs Predicted Values
plt.figure(figsize=(5, 4))
plt.scatter(y_test, y_pred, color='blue')
plt.plot([y.min(), y.max()], [y.min(), y.max()], '--k', label='Perfect Fit')
plt.xlabel('Actual Sales')
plt.ylabel('Predicted Sales')
plt.title('Actual vs Predicted Sales (Random Forest)')
plt.legend()
plt.show()
```



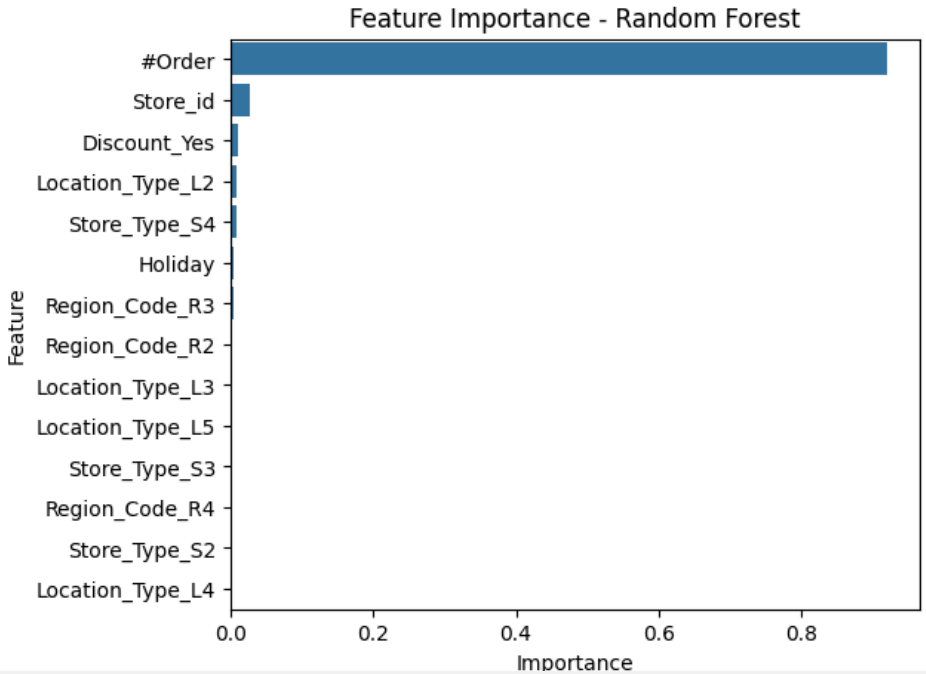
Feature Importance

Which refers to the technique used to determine the relative importance of each feature in predicting the target variable. It helps identify which features have the most influence on the model's predictions, allowing for better model interpretation and potential feature selection.

From below we can see that feature - Order# has high importance

```
# Feature Importance Feature Importance
feature_importances = rf_model.feature_importances_
features = X.columns
importance_df = pd.DataFrame({
    'Feature': features,
    'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

plt.figure(figsize=(6, 5))
sns.barplot(x='Importance', y='Feature', data=importance_df)
```

```
plt.title('Feature Importance - Random Forest')
plt.show()
```



3. Gradient Boosting Model: XGBoost XGBoost is a powerful boosting algorithm that often yields state-of-the-art results:

Purpose:

Gradient Boosting is an ensemble learning technique that builds multiple weak learners (usually decision trees) sequentially. Each new tree tries to correct the errors made by the previous tree. It does so by fitting the new tree to the residual errors of the previous model, meaning it focuses on learning where the previous models made mistakes.

XGBoost is particularly known for its speed, accuracy, and ability to handle a variety of data types and challenges, such as missing values and overfitting.

Components:

- Boosting Process
- Gradient Descent Optimization
- Regularization

- Shrinkage

Implementation:

- Install XGBoost
- Prepare Data
- Convert Data to DMatrix Format
- Define the Model
- Train the Model
- Make Predictions and Evaluate the Model
- Feature Importance

```
R-squared (0.94): The XGBoost model explains 94% of the variance in the target variable, indicating a strong fit and goo
Mean Absolute Error (2980.2): On average, the model's predictions are off by 2980.2, which is acceptable based on the sc
```

```
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train the XGBoost model
xgb_model = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42)
xgb_model.fit(X_train, y_train)

# Predict on the test set
y_pred_xgb = xgb_model.predict(X_test)

# Evaluate the model's performance
r2_xgb = r2_score(y_test, y_pred_xgb)
mae_xgb = mean_absolute_error(y_test, y_pred_xgb)

# Output the evaluation metrics
print(f"XGBoost - R-squared: {r2_xgb:.4f}")
print(f"XGBoost - Mean Absolute Error: {mae_xgb:.4f}")
```

```
XGBoost - R-squared: 0.9472
XGBoost - Mean Absolute Error: 2980.2512
```

Actual vs. Predicted Values

Purpose :

- This scatter plot compares the actual Sales values (y_test) against the predicted values (y_pred).

Interpretation :

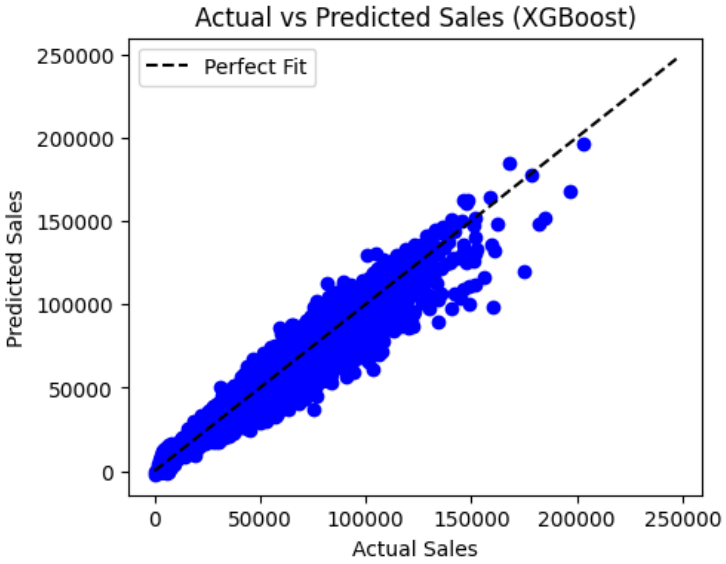
- Ideally, the points should lie along the diagonal line, which represents the case where predicted values are equal to actual values. Any significant deviation from this line indicates a higher error.

Feature Importance

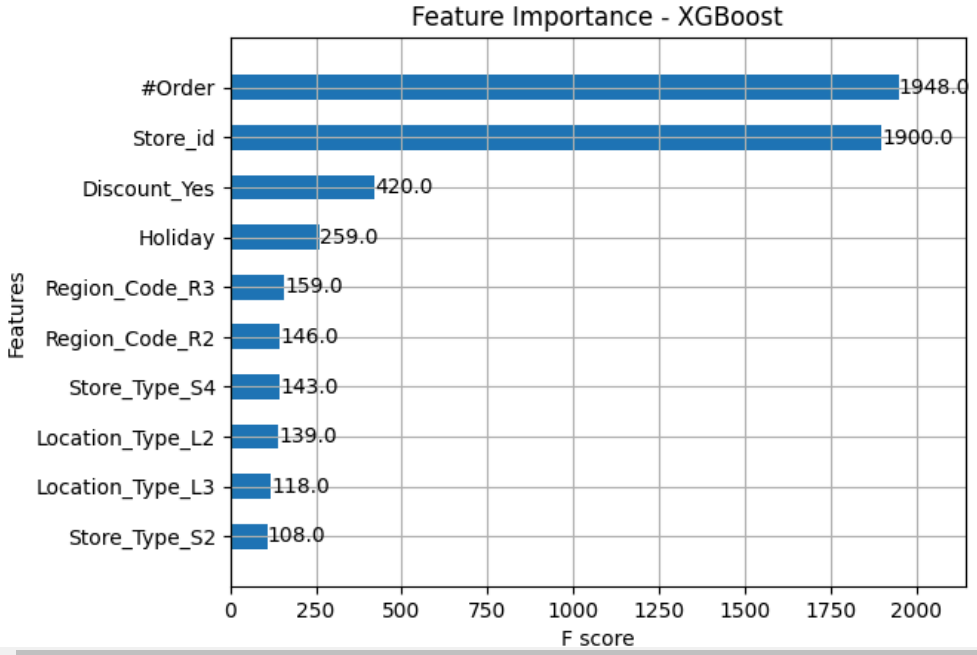
Which refers to the technique used to determine the relative importance of each feature in predicting the target variable. It helps identify which features have the most influence on the model's predictions, allowing for better model interpretation and potential feature selection.

From below we can see that feature - Order# has high importance

```
# Actual vs Predicted Values
plt.figure(figsize=(5, 4))
plt.scatter(y_test, y_pred, color='blue')
plt.plot([y.min(), y.max()], [y.min(), y.max()], '--k', label='Perfect Fit')
plt.xlabel('Actual Sales')
plt.ylabel('Predicted Sales')
plt.title('Actual vs Predicted Sales (XGBoost)')
plt.legend()
plt.show()

# Feature Importance (Bar Plot)
plt.figure(figsize=(5, 4))
plot_importance(xgb_model, importance_type='weight', max_num_features=10, height=0.5)
plt.title('Feature Importance - XGBoost')
plt.show()
```




```
<Figure size 500x400 with 0 Axes>
```



4. Deep Learning Model: LSTM for Time Series Forecasting implementing an LSTM model for forecasting future sales. LSTMs are ideal for sequential data like time series.

Purpose:

These models are capable of learning long-term dependencies in data, which makes them highly effective for forecasting tasks where future values are influenced by historical observations.

Components:

- Ensemble of Trees
- Bootstrap Aggregating (Bagging)
- Feature Randomness
- Voting

Implementation:

- Prepare the Data
- Normalize the Data
- Create Time Series Sequences
- Split the Data into Training and Test Sets
- Build the LSTM Model
- Train & Evaluate the model
- Make Predictions

Interpretation:

- The model predicts that sales will fluctuate slightly across the next 5 periods, with values ranging from 42,492.63 to 44,855.38.
- There is no drastic change in sales, suggesting relative stability or moderate growth/decline over the forecasted periods.
- This prediction can be useful for making business decisions related to stock, staffing, or marketing strategies.

```
# Prepare data for LSTM
scaler = MinMaxScaler(feature_range=(0, 1))
sales_scaled = scaler.fit_transform(sales.values.reshape(-1, 1))

# Create sequences for LSTM model (using past 'n' days to predict the next day)
def create_sequence(data, n_steps=3):
    X, y = [], []
    for i in range(len(data) - n_steps):
        X.append(data[i:i + n_steps, 0])
        y.append(data[i + n_steps, 0])
    return np.array(X), np.array(y)

n_steps = 10
X, y = create_sequence(sales_scaled, n_steps)
```

```
# Reshape data for LSTM
X = X.reshape((X.shape[0], X.shape[1], 1))
# Build the LSTM model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(X.shape[1], 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(X, y, epochs=5, batch_size=10, verbose=0)
# Make predictions
predicted_sales_scaled = model.predict(X[-5:].reshape(5, n_steps, 1))

# Inverse transform predictions to original scale
predicted_sales = scaler.inverse_transform(predicted_sales_scaled)

print(f"Predicted Sales for next 5 periods: {predicted_sales.flatten()}")
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 344ms/step
Predicted Sales for next 5 periods: [44490.582 42501.65  44308.453 43532.277 44434.844]
```

Actual vs. Predicted Values

Purpose :

- This below plot compares the actual Sales values (y_test) against the predicted values (y_pred).

Interpretation :

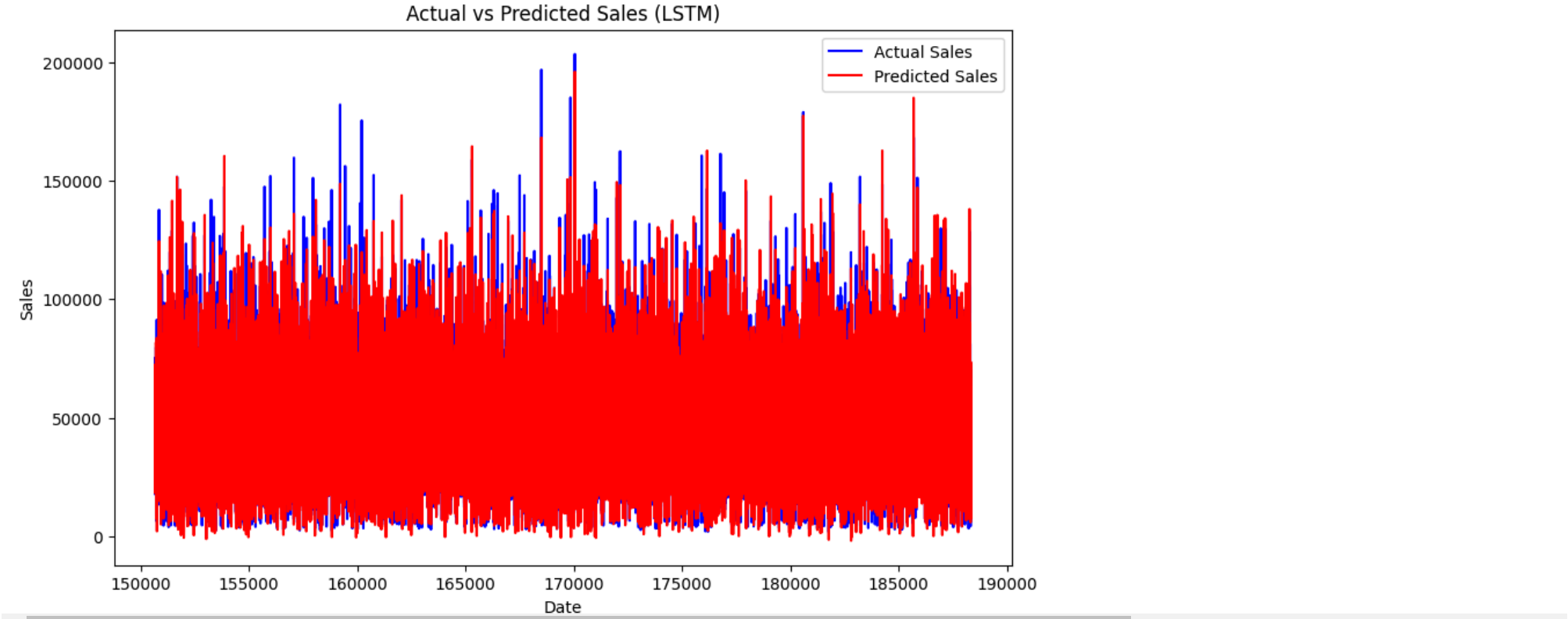
- The plot helps visualize how closely the predicted sales (red line) follow the actual sales (blue line), giving an insight into the model's performance.

```
R-squared (60.6522): A value of 60.6522 suggests that the model tends to have relatively small errors.
Mean Absolute Error (3678.6912): On average, the model's predictions are off by 3,678.69. This represents the average ma
```

```
# Evaluation metrics
print(f"Mean Absolute Error: {mae:.4f}")
rmse = np.sqrt(mae)
print(f"Root Mean Squared Error: {rmse:.4f}")

# Plot the results (Actual vs Predicted Sales)
plt.figure(figsize=(10, 6))
```

```
plt.plot(df.index[len(df) - len(y_test):], y_test, color='blue', label='Actual Sales')
plt.plot(df.index[len(df) - len(y_pred):], y_pred, color='red', label='Predicted Sales')
plt.title('Actual vs Predicted Sales (LSTM)')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.legend()
plt.show()
```

```
Mean Absolute Error: 3678.6912
Root Mean Squared Error: 60.6522
```




```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df1
```

Model Comparison:

- XGBoost performs the best with the lowest MAE (2054.81) and RMSE (3174.78), along with the highest R^2 (0.9702), indicating it makes the most accurate predictions and explains the highest proportion of variance in the data.

- Random Forest also performs well with a slightly higher MAE (2280.82) and RMSE (3559.49), but still offers a strong R^2 (0.9625), indicating robust predictive performance.
- LSTM has a higher MAE (3516.16) and RMSE (4827.48) compared to XGBoost and Random Forest, and its R^2 (0.9311) is slightly lower comparatively, but still it provides a good accuracy.
- Linear Regression has highest MAE (4055.23), RMSE (5512.46), and the lowest R^2 (0.9101), Comparatively which is low but still provides good data variances

```
# Convert categorical variables to numerical values using LabelEncoder
label_encoder = LabelEncoder()
dt['Store_Type'] = label_encoder.fit_transform(dt['Store_Type'])
dt['Location_Type'] = label_encoder.fit_transform(dt['Location_Type'])
dt['Region_Code'] = label_encoder.fit_transform(dt['Region_Code'])
dt['Discount'] = dt['Discount'].apply(lambda x: 1 if x == 'Yes' else 0)

# Convert Date to datetime and extract features (e.g., year, month)
dt['Date'] = pd.to_datetime(dt['Date'])
dt['Year'] = dt['Date'].dt.year
dt['Month'] = dt['Date'].dt.month

# Feature and target columns
X = dt[['Store_id', 'Store_Type', 'Location_Type', 'Region_Code', 'Holiday', 'Discount', '#Order', 'Year', 'Month']]
y = dt['Sales']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# --- Random Forest ---
rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(X_train, y_train)

# Predictions
rf_pred = rf_model.predict(X_test)

# Evaluation
rf_mae = mean_absolute_error(y_test, rf_pred)
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_pred))
rf_r2 = r2_score(y_test, rf_pred)

# --- XGBoost ---
xgb_model = xgb.XGBRegressor(random_state=42)
xgb_model.fit(X_train, y_train)

# Predictions
xgb_pred = xgb_model.predict(X_test)

# Evaluation
xgb_mae = mean_absolute_error(y_test, xgb_pred)
xgb_rmse = np.sqrt(mean_squared_error(y_test, xgb_pred))
```

```
xgb_r2 = r2_score(y_test, xgb_pred)

# --- LSTM (Long Short-Term Memory) ---
# Reshape the data for LSTM
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

X_train_lstm = X_train_scaled.reshape((X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
X_test_lstm = X_test_scaled.reshape((X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))

lstm_model = Sequential()
lstm_model.add(LSTM(units=50, activation='relu', input_shape=(X_train_lstm.shape[1], X_train_lstm.shape[2])))
lstm_model.add(Dense(1))

lstm_model.compile(optimizer='adam', loss='mean_squared_error')
lstm_model.fit(X_train_lstm, y_train, epochs=10, batch_size=8, verbose=0)

# Predictions
lstm_pred = lstm_model.predict(X_test_lstm)

# Evaluation
lstm_mae = mean_absolute_error(y_test, lstm_pred)
lstm_rmse = np.sqrt(mean_squared_error(y_test, lstm_pred))
lstm_r2 = r2_score(y_test, lstm_pred)

# ---Linear Regression ---
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)

# Predictions
lr_pred = lr_model.predict(X_test)

# Evaluation
lr_mae = mean_absolute_error(y_test, lr_pred)
lr_rmse = np.sqrt(mean_squared_error(y_test, lr_pred))
lr_r2 = r2_score(y_test, lr_pred)

# --- Evaluation Results ---
print("Random Forest - MAE:", rf_mae)
print("Random Forest - RMSE:", rf_rmse)
print("Random Forest - R2:", rf_r2)

print("\nXGBoost - MAE:", xgb_mae)
print("XGBoost - RMSE:", xgb_rmse)
print("XGBoost - R2:", xgb_r2)

print("\nLSTM - MAE:", lstm_mae)
print("LSTM - RMSE:", lstm_rmse)
print("LSTM - R2:", lstm_r2)
```

```
print("\nLinear Regression - MAE:", lr_mae)
print("Linear Regression - RMSE:", lr_rmse)
print("Linear Regression - R2:", lr_r2)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

1178/1178 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step
Random Forest - MAE: 2280.8171390387324
Random Forest - RMSE: 3559.4889724838013
Random Forest - R2: 0.9625231761773276

XGBoost - MAE: 2054.8109346386004
XGBoost - RMSE: 3174.7825194516136
XGBoost - R2: 0.970186330395761

LSTM - MAE: 3490.4915862524435
LSTM - RMSE: 4805.464713804225
LSTM - R2: 0.9316940935917044

Linear Regression - MAE: 4055.2341344640818
Linear Regression - RMSE: 5512.462981117447
Linear Regression - R2: 0.9101167338086359
```

```
Start coding or generate with AI.
```

3. Model Evaluation and Validation:

Model evaluation involves assessing a model's performance using metrics like accuracy, precision, recall, R-squared, MAE, and RMSE on a test dataset to understand how well it generalizes to unseen data.

Validation ensures that the model's performance is consistent across different subsets of the data, often using techniques like cross-validation.

This process helps prevent overfitting and ensures the model is reliable and robust for deployment.

```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df1
```

Cross-Validation: Implement time-series specific cross-validation techniques to evaluate model performance over different temporal splits of the data.

- Time-series cross-validation ensures that the temporal order of data is maintained, preventing future data from being used to predict past events.

- Techniques like rolling-window or expanding-window splits are used, where the model is trained on past data and tested on future data.
- This method evaluates how well the model generalizes over different time periods and prevents data leakage.

Interpretations:

The model evaluation shows the following:

MAE (Mean Absolute Error):

- The MAE for each fold ranges from 2499.98 to 4317.44, with an average of 3397.08. This indicates that, on average, the model's predictions deviate by about 3397.08 units from the actual values.

RMSE (Root Mean Squared Error):

- The RMSE values range from 3836.37 to 5593.73, with an average of 4936.23. Since RMSE penalizes larger errors more heavily, this suggests that the model has some larger prediction errors but generally performs reasonably well.

R2 (R-squared):

- The R2 values range from 0.90 to 0.96, with an average of 0.93. This indicates that the model explains about 93% of the variance in the data, which is a strong performance.

From below we can conclude that **the model performs well**, with high R2 indicating strong predictive power, and MAE and RMSE showing moderate prediction errors. There's a slight variance in performance across the folds, but overall, the model generalizes well.

```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df1
dt['Date'] = pd.to_datetime(dt['Date'])
dt['month'] = dt['Date'].dt.month
dt.head()

dt['Date'] = pd.to_datetime(dt['Date'])
dt['month'] = dt['Date'].dt.month



# Convert Date column to datetime
dt['Date'] = pd.to_datetime(dt['Date'])

# Sort data by Date
dt = dt.sort_values('Date')

# Encode categorical features
df_encoded = pd.get_dummies(dt, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount'], drop_first=True)

# Define Features (X) and Target (y)
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']
tscv = TimeSeriesSplit(n_splits=3)
model = RandomForestRegressor()
```

```
# Variables to store evaluation metrics
mae_scores = []
rmse_scores = []
r2_scores = []

for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train the model
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Evaluate performance
    mae = mean_absolute_error(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    # Append the results
    mae_scores.append(mae)
    rmse_scores.append(rmse)
    r2_scores.append(r2)

print(f"MAE scores for each fold: {mae_scores}")
print(f"RMSE scores for each fold: {rmse_scores}")
print(f"R2 scores for each fold: {r2_scores}")

print(f"Average MAE: {np.mean(mae_scores):.4f}")
print(f"Average RMSE: {np.mean(rmse_scores):.4f}")
print(f"Average R2: {np.mean(r2_scores):.4f}")
```

```
MAE scores for each fold: [4300.810411006436, 3370.361766783373, 2498.518439902927]
RMSE scores for each fold: [5573.203511614997, 5376.231965918349, 3831.3785080786006]
R2 scores for each fold: [0.9044962241168568, 0.9203354916248044, 0.960496223125974]
Average MAE: 3389.8969
Average RMSE: 4926.9380
Average R2: 0.9284
```

Performance Metrics: Use metrics appropriate for regression tasks, such as MAE (Mean Absolute Error), MSE (Mean Squared Error), RMSE (Root Mean Squared Error), and MAPE (Mean Absolute Percentage Error).

Performance metrics for regression tasks help evaluate the accuracy of predictions.

MAE (Mean Absolute Error):

- The MAE of 35,815.65 indicates that, on average, the model's predictions are off by about 35,815.65 units from the actual values. This gives a sense of the model's overall prediction accuracy.

MSE (Mean Squared Error):

- The MSE of 1,620,327,772.24 is very large, which suggests that the model has some significant prediction errors. MSE penalizes larger errors more heavily due to the squaring of differences, indicating that the model might be producing large outliers in predictions.

RMSE (Root Mean Squared Error):

- The RMSE of 40,253.30 represents the square root of the MSE and gives a more interpretable scale. It's still large, showing that the model has substantial prediction errors, which could affect the model's reliability.

MAPE (Mean Absolute Percentage Error):

- The MAPE is infinite (inf%), which suggests that the model is making predictions where the actual values are zero or close to zero in some cases. Since MAPE involves division by the actual values, it cannot handle zero values and thus leads to an "infinite" result.

```
# Feature Engineering: Encoding categorical variables
df_encoded = pd.get_dummies(dt, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount'], drop_first=True)

# Define Features (X) and Target (y)
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']

# Split data into train and test sets (using the full data for this example)
X_train, X_test = X[:1], X[1:]
y_train, y_test = y[:1], y[1:]

# Train a Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Calculate Performance Metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100  # MAPE as percentage

# Output the metrics
print(f"MAE: {mae:.4f}")
print(f"MSE: {mse:.4f}")
print(f"RMSE: {rmse:.4f}")
print(f"MAPE: {mape:.2f}%")
```

```
MAE: 35815.6460
MSE: 1620327772.2448
RMSE: 40253.2952
MAPE: inf%
```

Residual Analysis: Analyze the residuals to ensure there are no patterns left unmodeled.

Performance metrics for regression tasks help evaluate the accuracy of predictions.

MAE (Mean Absolute Error):

- The MAE of 3653.6488 indicates that, on average, the model's predictions are off by about 3653.6488 units from the actual values. This gives a sense of the model's overall prediction accuracy.

Residual vs Predicted Sales:

This plot shows the relationship between the residuals (the difference between actual and predicted values) and the predicted sales.

- Since there is no pattern (i.e., the residuals are scattered randomly around zero), it suggests that the model is well-fitted and has captured the underlying patterns in the data.

Histogram of Residuals: The histogram shows the distribution of the residuals (errors). It helps assess the normality of the residuals.

- As its normally distributed but very slightly skewed towards other side we can assume that errors are randomly distributed assuming normality

Residual vs Time: This plot shows the residuals over time, helping to check for time-based patterns that the model may not have captured.

- Since there are no patterns, residuals appear randomly scattered, it indicates that the model has accounted for any temporal dependencies in the data.

Autocorrelation of Residuals:

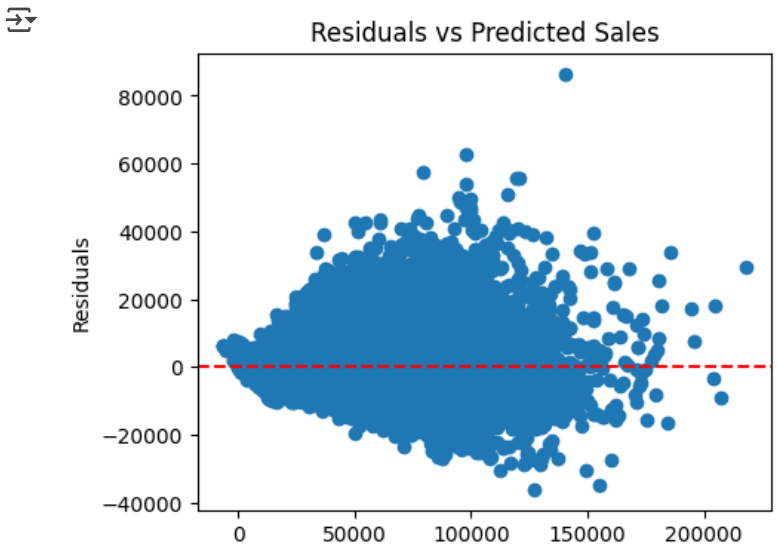
This plot or test checks if residuals are correlated with their own lagged values. It's commonly used in time series data to detect if there are temporal structures not captured by the model.

- There's no significant autocorrelation at any lag (i.e., values are close to zero), it suggests that the residuals are independent, and the model has captured all the temporal dependencies.

Overall **residuals are randomly scattered** without clear patterns in any of these plots (residual vs predicted, residual vs time, and autocorrelation), it suggests a **good model fit**.

```
# One-hot encode categorical columns
df_encoded = pd.get_dummies(dt, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount'], drop_first=True)

# Define features and target
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']

# Train a Linear Regression model
model = LinearRegression()
model.fit(X, y)

# Make predictions
y_pred = model.predict(X)
```

```
# Calculate residuals
residuals = y - y_pred

# Plotting residuals vs predicted values
plt.figure(figsize=(5, 4))
plt.scatter(y_pred, residuals)
plt.axhline(y=0, color='red', linestyle='--')
plt.title('Residuals vs Predicted Sales')
plt.xlabel('Predicted Sales')
plt.ylabel('Residuals')
plt.show()


plt.figure(figsize=(5, 4))
plt.plot(residuals)
plt.title("Residuals vs. Time")
plt.xlabel("Time")
plt.ylabel("Residuals")
plt.show()

# Histogram of residuals
plt.figure(figsize=(5, 4))
sns.histplot(residuals, kde=True)
plt.title("Histogram of Residuals")
plt.xlabel("Residuals")
plt.ylabel("Frequency")
plt.show()

# Plot ACF of residuals to check for autocorrelation
plot_acf(residuals, lags=50)
plt.title("Autocorrelation of Residuals")
plt.show()

# Calculate Mean Absolute Error as an additional residual analysis
mae = mean_absolute_error(y, y_pred)
print(f"Mean Absolute Error (MAE): {mae:.4f}")
```


Residuals vs Predicted Sales
Residuals
80000
60000
40000
20000
0
−20000
−40000
0
50000
100000
150000
200000

Model Performance Metrics

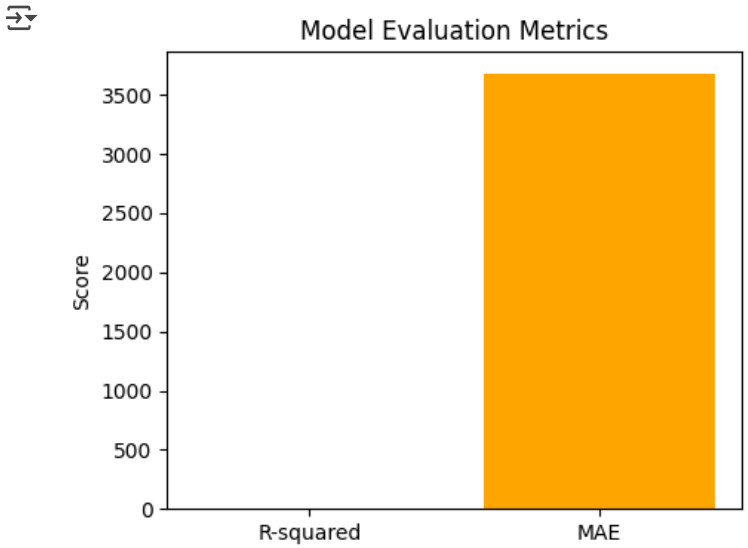
Purpose :

- A bar plot showing the key evaluation metrics (R-squared and MAE).

Interpretation :

- This gives a quick overview of the model's performance. High R-squared indicates that the model explains a large portion of the variance in sales, while MAE gives a sense of the average prediction error.

```
# Model Performance Metrics Bar Plot
metrics = {'R-squared': r2, 'MAE': mae}
plt.figure(figsize=(5, 4))
plt.bar(metrics.keys(), metrics.values(), color=['blue', 'orange'])
plt.title('Model Evaluation Metrics')
plt.ylabel('Score')
plt.show()
```



Complex Models: Explore more sophisticated models to improve accuracy. Potential models include:

These models are often more powerful and flexible, capable of capturing complex patterns and relationships in the data. The goal is to improve accuracy and enhance the model's predictive power by moving beyond the basic approach. Below are the main purpose of it,

- Better Accuracy
- Non-linearity

- Flexibility
- Handling More Complex Data

Implementation

- By providing Train & Test data as input model makes the prediction by making the values fit a straight line.

1. Time Series Model: Applying ARIMA for data forecasting:

Purpose:

ARIMA (AutoRegressive Integrated Moving Average) is one of the most widely used models for forecasting time series data.

- It is particularly effective when the data shows trends or seasonality and is based on past observations.
- ARIMA helps predict future values by analyzing past values (autoregressive), the differences between consecutive values (integrated), and the moving average of past errors.

Components:

The ARIMA model is denoted as ARIMA(p, d, q), where:

- p is the number of lag observations in the autoregressive part (AR).
- d is the number of times the data is differenced to make it stationary.
- q is the size of the moving average window (MA).

Once the model is fitted it can be used to make predictions and forecasts.

```
dt.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount	#Order	Sales	month
0	T1000001	1	S1	L3	R1	2018-01-01	1	Yes	9	7011.84	1
1	T1000002	253	S4	L2	R1	2018-01-01	1	Yes	60	51789.12	1
2	T1000003	252	S3	L2	R1	2018-01-01	1	Yes	42	36868.20	1
3	T1000004	251	S2	L3	R1	2018-01-01	1	Yes	23	19715.16	1
4	T1000005	250	S2	L3	R4	2018-01-01	1	Yes	62	45614.52	1

```
# Ensuring the date format
df_model = dt
df_model['Date'] = pd.to_datetime(df_model['Date'])
df_model.set_index('Date', inplace=True)
```

```
# Preparing data for stationary tests
date_range = df_model.index
sales_data = df_model['Sales']
data = pd.DataFrame({'date': date_range, 'Sales': sales_data})
data.index = pd.to_datetime(data['date'])
data = data.groupby(data.index).median()
data = data.drop('date', axis=1)
data
```

date	Sales
2018-01-01	39982.38
2018-01-02	50628.00
2018-01-03	47988.00
2018-01-04	52410.00
2018-01-05	60078.00
...	...
2019-05-27	43821.00
2019-05-28	47451.00
2019-05-29	42933.00
2019-05-30	41955.00
2019-05-31	40677.12

516 rows × 1 columns

Dickey-Fuller test

- Using statistical analysis - Dickey-Fuller Test we can determine whether a given time series is stationary or non-stationary.
- In time series analysis, we will find stationarity(constant over time) / non-stationary(show trends or seasonality, which need to be addressed before modeling.)

```
def adf_test(series):
    result = adfuller(series)
    print('ADF Statistic:', result[0])
    print('p-value:', result[1])
    print('Critical Values:')
    for key, value in result[4].items():
        print(f'\t{key}: {value}')
    if result[1] < 0.05:
```

```
      print("Reject the null hypothesis: The series is stationary.")
    else:
      print("Fail to reject the null hypothesis: The series is non-stationary.")

adf_test(data['Sales'])
```

```
ADF Statistic: -3.4590273622955423
p-value: 0.009103860345178972
Critical Values:
        1%: -3.4436029548776395
        5%: -2.867384756137026
        10%: -2.5698830308597813
Reject the null hypothesis: The series is stationary.
```

Start coding or generate with AI.

Differencing

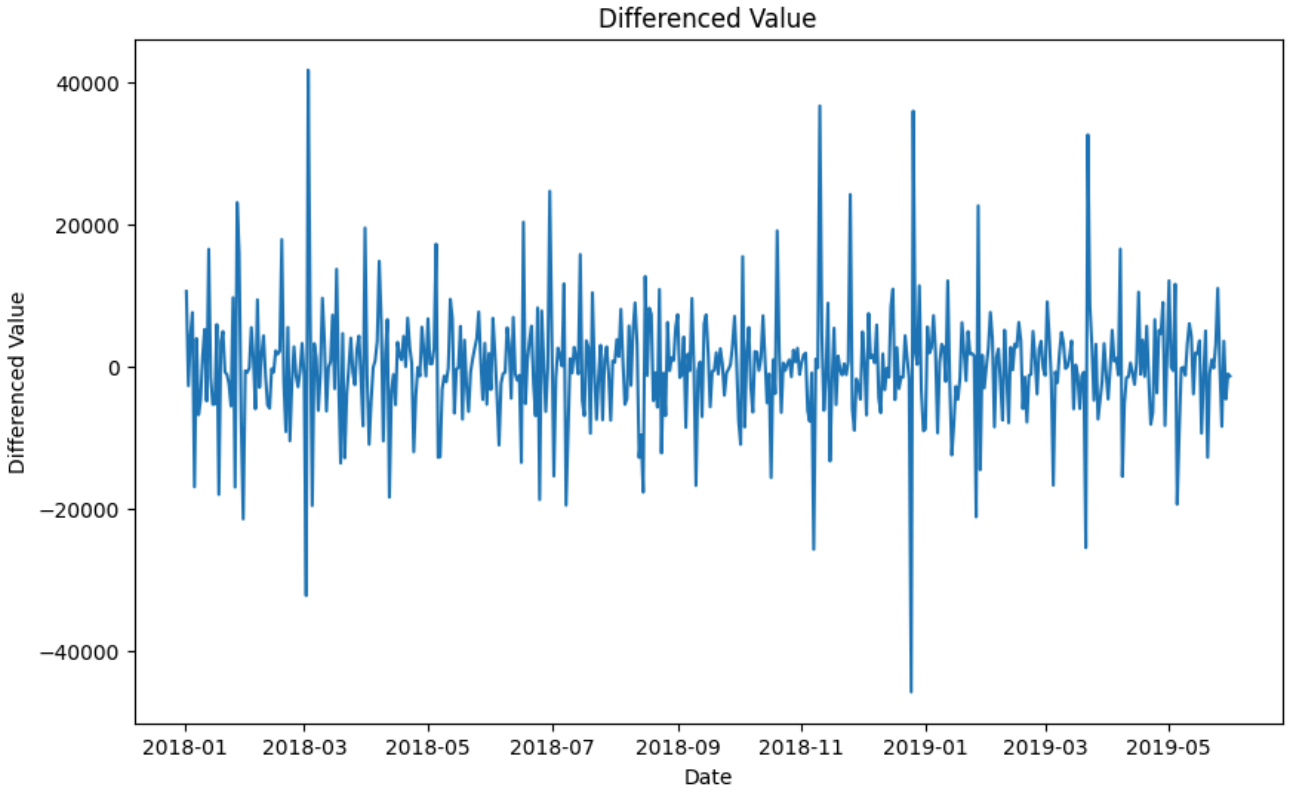
From the data considered the series was already stationary, still the differencing is done and got the below p-values and other factors,

```
ADF Statistic: -3.4590273622955423
p-value: 0.009103860345178972
Critical Values:
    1%: -3.4436029548776395
    5%: -2.867384756137026
    10%: -2.5698830308597813
```

- Using Differencing technique for time series analysis to transform a non-stationary series into a stationary one.
- Differencing helps to remove trends and seasonality from a time series, making it more suitable for modeling.

```
# Although the series is stationary below differencing & decomposition models are done for reference

# Differencing to make the series stationary
data['Value_diff'] = data['Sales'] - data['Sales'].shift(1)

# Drop NaN values after differencing
data.dropna(inplace=True)

# Plot the differenced series
plt.figure(figsize=(10, 6))
plt.plot(data['Value_diff'])
plt.title('Differenced Value')
plt.xlabel('Date')
plt.ylabel('Differenced Value')
plt.show()
```

```
# Check stationarity again
adf_test(data['Value_diff'])
```



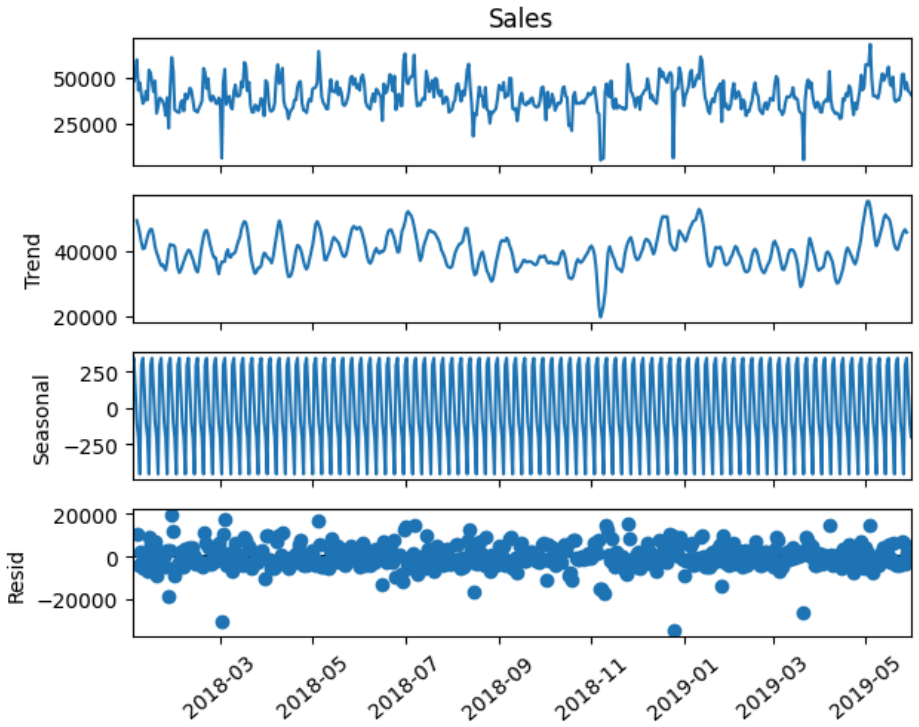

```
ADF Statistic: -9.26720946595902
p-value: 1.3568130861411842e-15
Critical Values:
        1%: -3.4436298692815304
        5%: -2.867396599893435
        10%: -2.5698893429241916
Reject the null hypothesis: The series is stationary.
```

Decomposition

- Time series decomposition is helps to understand the underlying patterns and improving forecasting accuracy & improve decision-making..
- where the time series split into components: trend, seasonality, and residual (or noise).

```
# Decomposition and its vizualisation
result = seasonal_decompose(data['Sales'], model='additive', period=6)
```

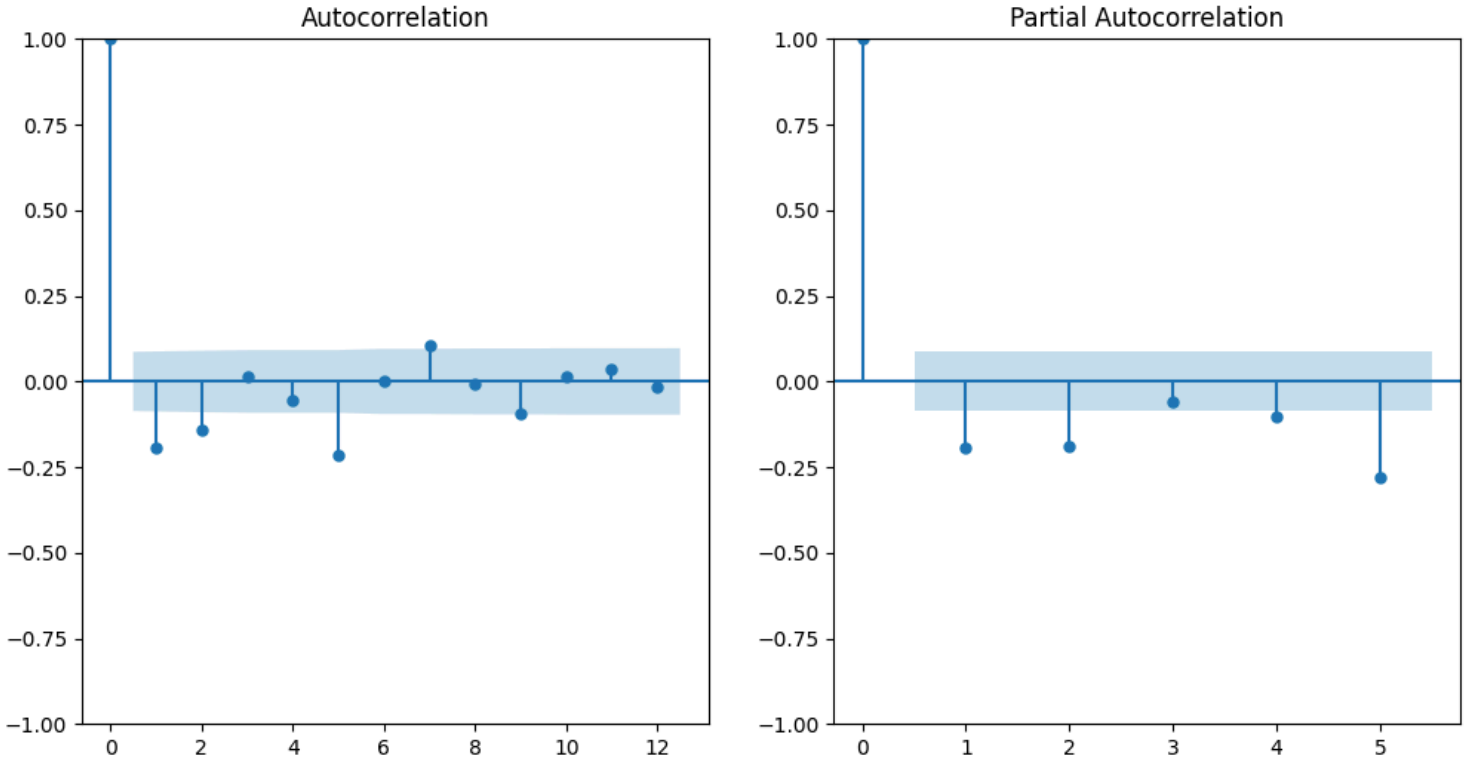
```
result.plot()
plt.xticks(rotation = 40)
plt.show()
```



ACF and PACF

- In order to understand the relationships between an observation and its past values &
- Identifying appropriate time series models & understanding the structure of the data below plots are implemented.

```
# Plot ACF and PACF
plt.figure(figsize=(12, 6))

plt.subplot(121)
plot_acf(data['Value_diff'], lags=12, ax=plt.gca())

plt.subplot(122)
plot_pacf(data['Value_diff'], lags=5, ax=plt.gca())

plt.show()
```




```
# Perform Augmented Dickey-Fuller test
result = adfuller(data['Sales'])
print(f"ADF Statistic: {result[0]}")
print(f"p-value: {result[1]}")
```

```
ADF Statistic: -3.461280550155466
p-value: 0.009038986906826817
```

Executing ARIMA model

```
# Fit AutoARIMA model
model_autoarima = auto_arima(data['Sales'], seasonal=False, stepwise=True, trace=True)

# Print the best model
print(model_autoarima.summary())

# Forecast next 6 periods
forecast_autoarima = model_autoarima.predict(n_periods=6)
```

```
print("AutoARIMA forecast:", forecast_autoarima)
```

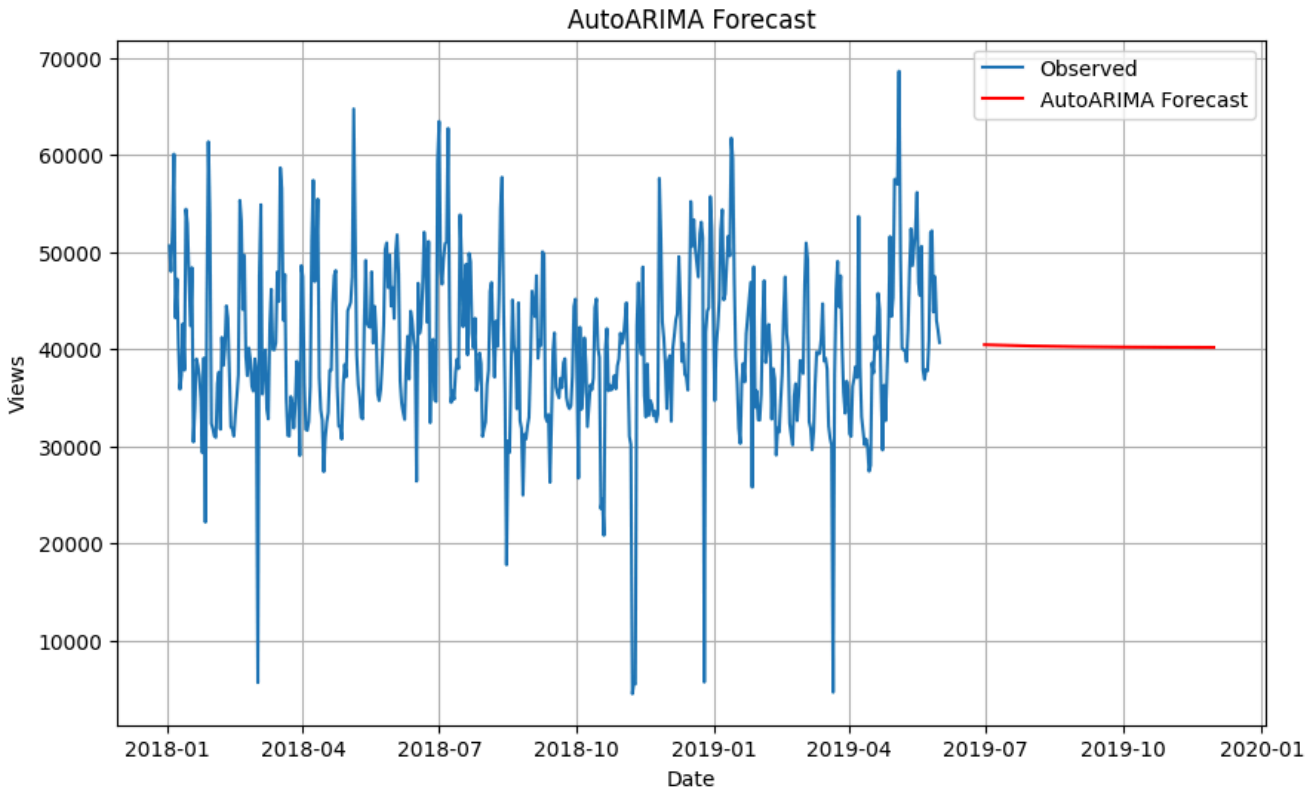
```
Performing stepwise search to minimize aic
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

 ARIMA(2,0,2)(0,0,0)[0]             : AIC=inf, Time=1.08 sec
 ARIMA(0,0,0)(0,0,0)[0]             : AIC=12404.057, Time=0.02 sec
 ARIMA(1,0,0)(0,0,0)[0]             : AIC=10723.584, Time=0.04 sec
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

 ARIMA(0,0,1)(0,0,0)[0]             : AIC=11842.168, Time=0.16 sec
 ARIMA(2,0,0)(0,0,0)[0]             : AIC=10707.362, Time=0.09 sec
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:

'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

 ARIMA(3,0,0)(0,0,0)[0]             : AIC=inf, Time=0.15 sec
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:
```

```
'force_all_finite' was renamed to 'ensure_all_finite' in 1.6 and will be removed in 1.8.

 ARIMA(2,0,1)(0,0,0)[0]             : AIC=10617.903, Time=0.67 sec
 ARIMA(1,0,1)(0,0,0)[0]             : AIC=10696.921, Time=0.11 sec
/usr/local/lib/python3.11/dist-packages/sklearn/utils/deprecation.py:151: FutureWarning:
```

Vizual representation of ARIMA forecast

```
# Create a new time index for the forecasted period
forecast_index = pd.date_range(start=data.index[-1] + pd.Timedelta(days=30), periods=6, freq='M')

# Plot the results
plt.figure(figsize=(10, 6))

# Plot historical data
plt.plot(data.index, data['Sales'], label='Observed')

# Plot forecasted data
plt.plot(forecast_index, forecast_autoarima, label='AutoARIMA Forecast', color='red')

# Add labels and legend
plt.title('AutoARIMA Forecast')
plt.xlabel('Date')
plt.ylabel('Views')
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

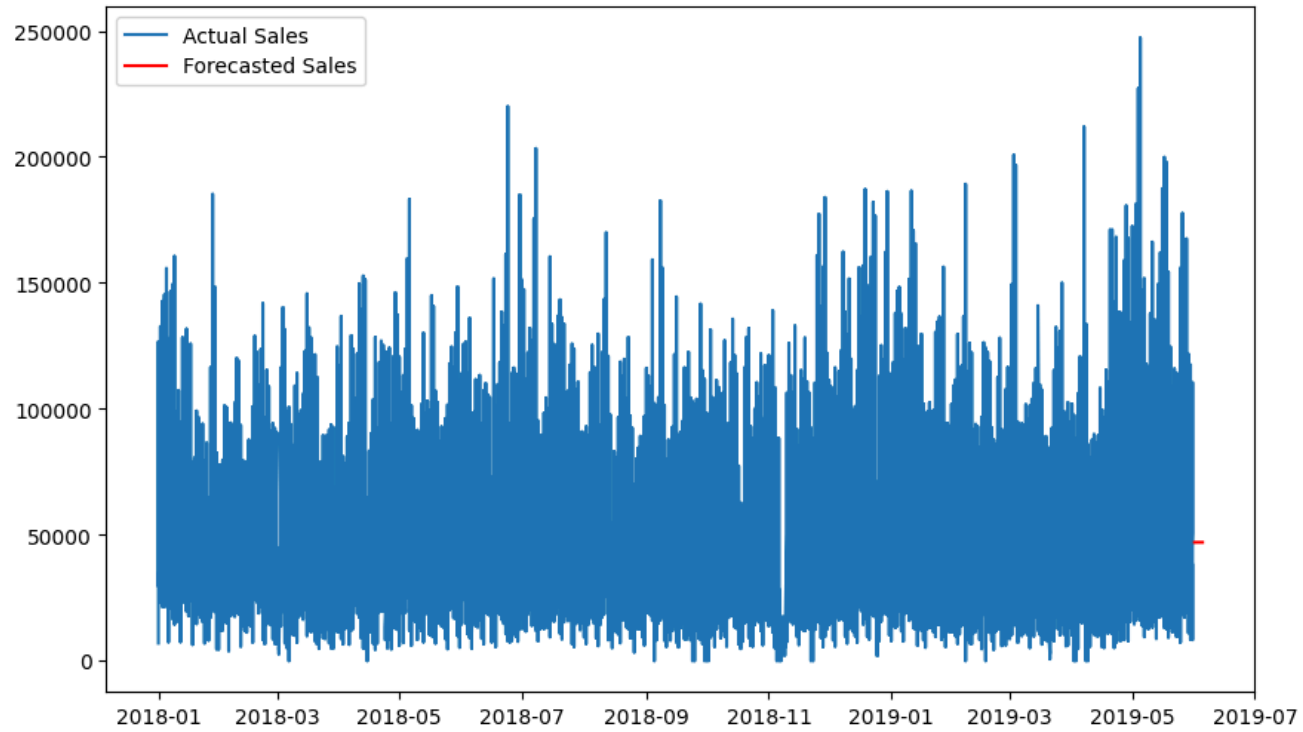
```
<ipython-input-196-520b3fd30881>:2: FutureWarning:

'M' is deprecated and will be removed in a future version, please use 'ME' instead.
```




```
#dt['month'] = pd.to_datetime(dt['month'])
#dt.set_index('month', inplace=True)
sales = dt['Sales']

# Fit an ARIMA model (order=(p, d, q))
# p = lag order, d = differencing order, q = order of moving average
model = ARIMA(sales, order=(1, 1, 1))
model_fit = model.fit()

# Make predictions
forecast = model_fit.forecast(steps=5)

# Plot for actual vs. forecasted sales
plt.figure(figsize=(10,6))
plt.plot(sales, label='Actual Sales')
plt.plot(pd.date_range(sales.index[-1], periods=6, freq='D')[1:], forecast, label='Forecasted Sales', color='red')
```

```
plt.legend()
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:
A date index has been provided, but it has no associated frequency information and so will be ignored when e.g. forecasting.
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:
A date index has been provided, but it has no associated frequency information and so will be ignored when e.g. forecasting.
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:
A date index has been provided, but it has no associated frequency information and so will be ignored when e.g. forecasting.
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:837: ValueWarning:
No supported index is available. Prediction results will be given with an integer index beginning at `start`.
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:837: FutureWarning:
No supported index is available. In the next version, calling this method in a model without a supported index will result in an exception.
```



SARIMA

- SARIMA (Seasonal AutoRegressive Integrated Moving Average) an extension of the ARIMA model which explicitly accounts for seasonality in time series data.
- It is mainly used to model time series data that exhibits both non-seasonal and seasonal behaviors using components such as (AutoRegressive, Integrated, Moving Average & Seasonality)

```
# Aggregate duplicates
date_range = df_model.index
sales_data = df_model['Sales']
data = pd.DataFrame({'date': date_range, 'Sales': sales_data})
data.index = pd.to_datetime(data['date'])
data = data.groupby(data.index).median()
data = data.drop('date', axis=1)
data
```

date	Sales
2018-01-01	39982.38
2018-01-02	50628.00
2018-01-03	47988.00
2018-01-04	52410.00
2018-01-05	60078.00
...	...
2019-05-27	43821.00
2019-05-28	47451.00
2019-05-29	42933.00
2019-05-30	41955.00
2019-05-31	40677.12

516 rows × 1 columns

Dickey-Fuller test

- Using statistical analysis - Dickey-Fuller Test we can determine whether a given time series is stationary or non-stationary.
- In time series analysis, we will find stationarity(constant over time) / non-stationary(show trends or seasonality, which need to be addressed before modeling.)

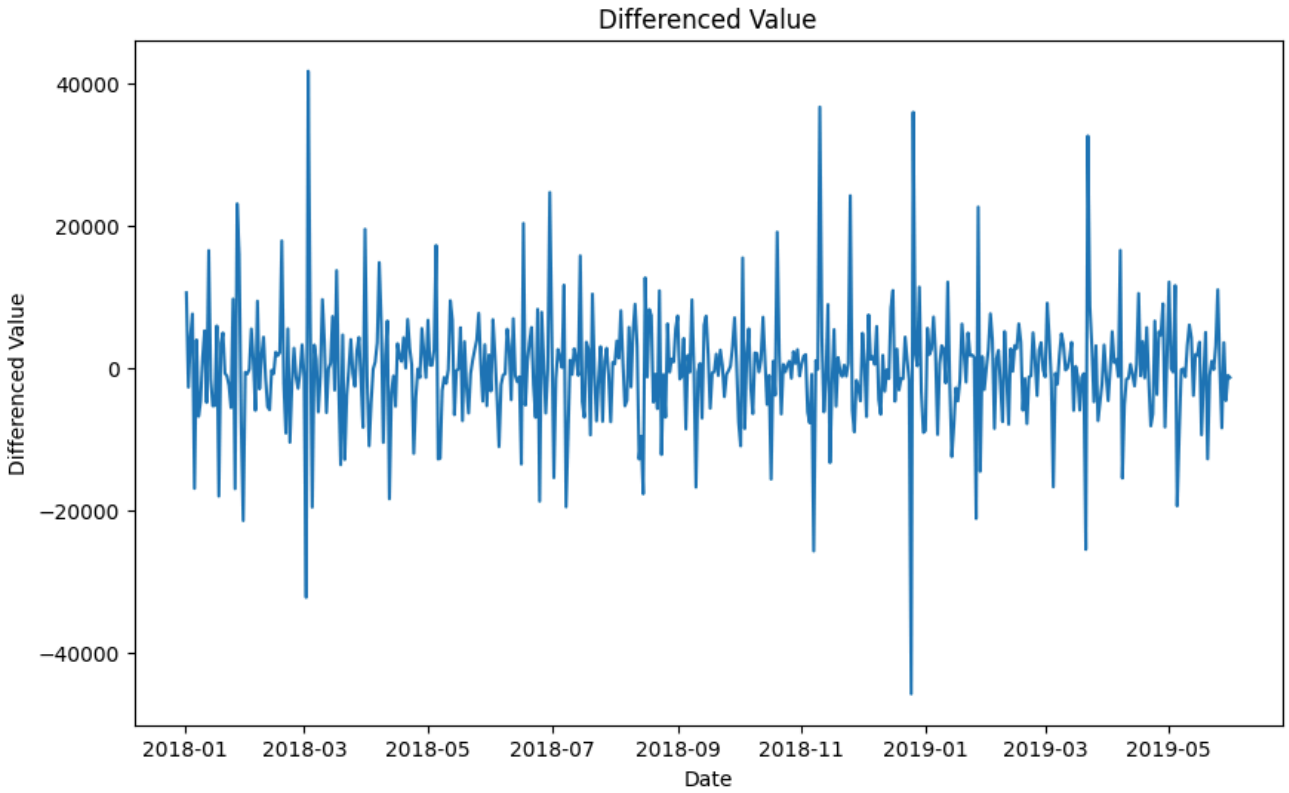
```
def adf_test(series):
    result = adfuller(series)
    print('ADF Statistic:', result[0])
    print('p-value:', result[1])
    print('Critical Values:')
    for key, value in result[4].items():
        print(f'\t{key}: {value}')

adf_test(data['Sales'])

if result[1] < 0.05:
    print("Reject the null hypothesis: The series is stationary.")
else:
    print("Fail to reject the null hypothesis: The series is non-stationary.")
```

```
ADF Statistic: -3.4590273622955423
p-value: 0.009103860345178972
Critical Values:
        1%: -3.4436029548776395
        5%: -2.867384756137026
        10%: -2.5698830308597813
Reject the null hypothesis: The series is stationary.
```

Differencing

- Using Differencing technique for time series analysis to transform a non-stationary series into a stationary one.
- Differencing helps to remove trends and seasonality from a time series, making it more suitable for modeling.

```
# Differencing to make the series stationary
data['Value_diff'] = data['Sales'] - data['Sales'].shift(1)

# Drop NaN values after differencing
data.dropna(inplace=True)

# Plot the differenced series
plt.figure(figsize=(10, 6))
plt.plot(data['Value_diff'])
plt.title('Differenced Value')
plt.xlabel('Date')
plt.ylabel('Differenced Value')
plt.show()

# Check stationarity again
adf_test(data['Value_diff'])
```

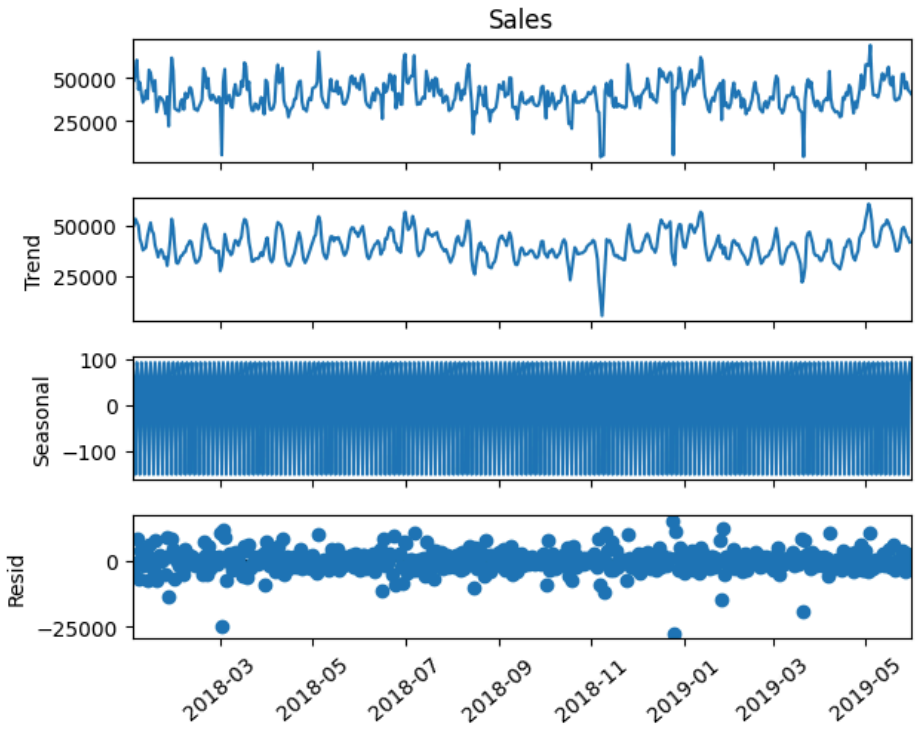



```
ADF Statistic: -9.26720946595902
p-value: 1.3568130861411842e-15
Critical Values:
        1%: -3.4436298692815304
        5%: -2.867396599893435
        10%: -2.5698893429241916
```

Decomposition

- Time series decomposition is helps to understand the underlying patterns and improving forecasting accuracy.
- where the time series split into components: trend, seasonality, and residual (or noise).

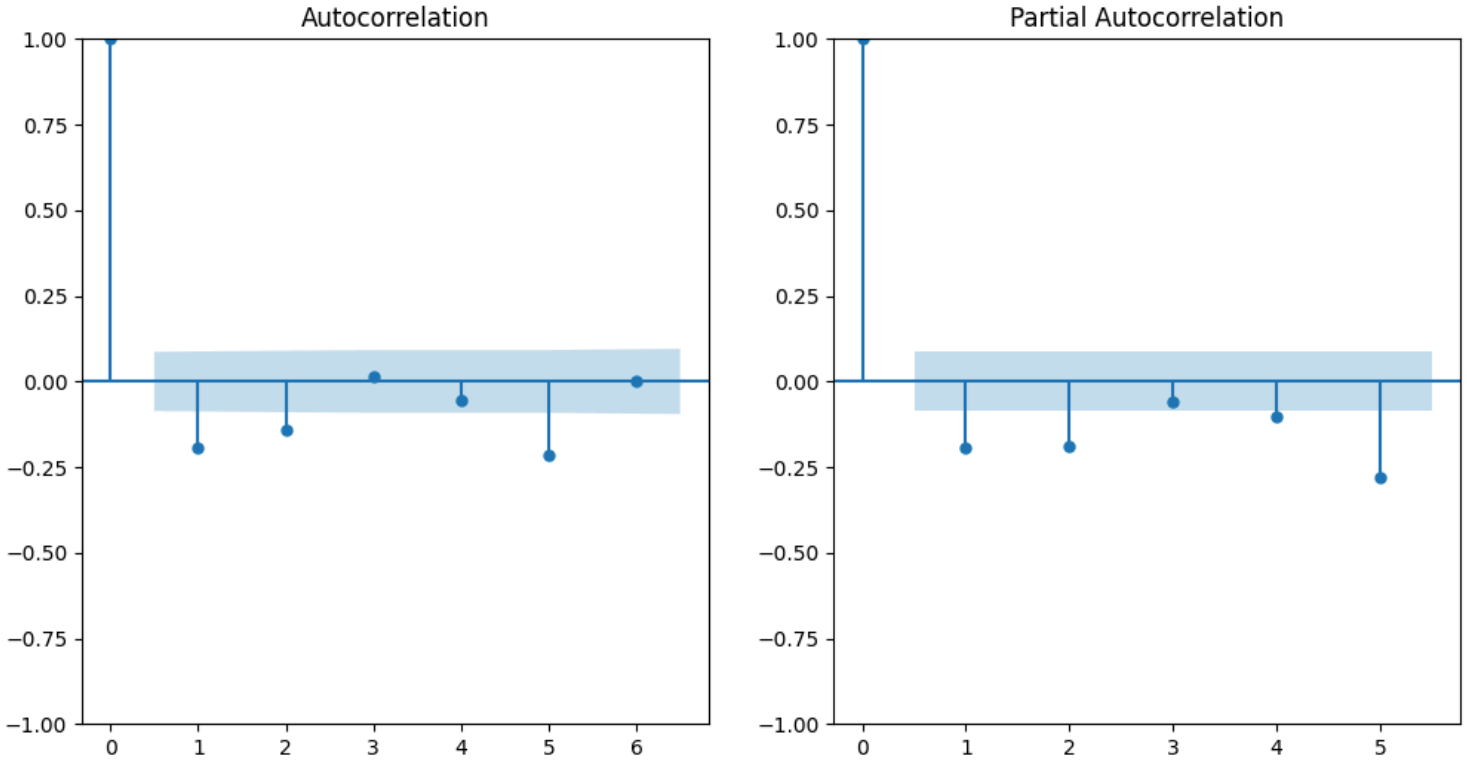
```
# Decomposition and its vizualisation
result = seasonal_decompose(data['Sales'], model='additive', period=3)
result.plot()
plt.xticks(rotation = 40)
plt.show()
```



ACF and PACF

- In order to understand the relationships between an observation and its past values &
- Identifying appropriate time series models (like ARIMA) & understanding the structure of the data below plots are implemented.

```
# Plot ACF and PACF
plt.figure(figsize=(12, 6))

plt.subplot(121)
plot_acf(data['Value_diff'], lags=6, ax=plt.gca())

plt.subplot(122)
plot_pacf(data['Value_diff'], lags=5, ax=plt.gca())

plt.show()
```



Executing SARIMAX model

```
# Fit SARIMAX model with exogenous variable
model = SARIMAX(data['Sales'], exog=data['Value_diff'], order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
model_fit = model.fit(disp=False)

# Print the summary of the model
print(model_fit.summary())
```

```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:

No frequency information was provided, so inferred frequency D will be used.

/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:

No frequency information was provided, so inferred frequency D will be used.

                                     SARIMAX Results
==========================================================================================
```

```
Dep. Variable:                             Sales   No. Observations:                  515
Model:             SARIMAX(1, 1, 1)x(1, 1, 1, 12)   Log Likelihood               -4999.586
Date:                            Sun, 02 Feb 2025   AIC                          10011.172
Time:                                    06:25:27   BIC                          10036.483
Sample:                                01-02-2018   HQIC                         10021.102
                                     - 05-31-2019
Covariance Type:                               opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Value_diff     0.4978      0.015     33.786      0.000       0.469       0.527
ar.L1         -0.2803      0.115     -2.447      0.014      -0.505      -0.056
ma.L1          0.8015      0.091      8.773      0.000       0.622       0.981
ar.S.L12      -0.0711      0.082     -0.866      0.386      -0.232       0.090
ma.S.L12      -0.8537      0.071    -11.944      0.000      -0.994      -0.714
sigma2      4.469e+07   1.14e-09    3.9e+16      0.000    4.47e+07    4.47e+07
===================================================================================
Ljung-Box (L1) (Q):                   0.96   Jarque-Bera (JB):               368.06
Prob(Q):                              0.33   Prob(JB):                         0.00
Heteroskedasticity (H):               0.62   Skew:                            -0.44
Prob(H) (two-sided):                  0.00   Kurtosis:                         7.10
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 2.76e+32. Standard errors may be unstable.
```

```
model_sarima = SARIMAX(data['Sales'], order=(1, 1, 1), seasonal_order=(1, 1, 1, 12))
sarima_fit = model_sarima.fit(disp=False)
print(sarima_fit.summary())

# Forecast the next 6 months
forecast_sarima = sarima_fit.get_forecast(steps=6)
forecast_index = pd.date_range(start=df.index[-1] , periods=6, freq='M')
```

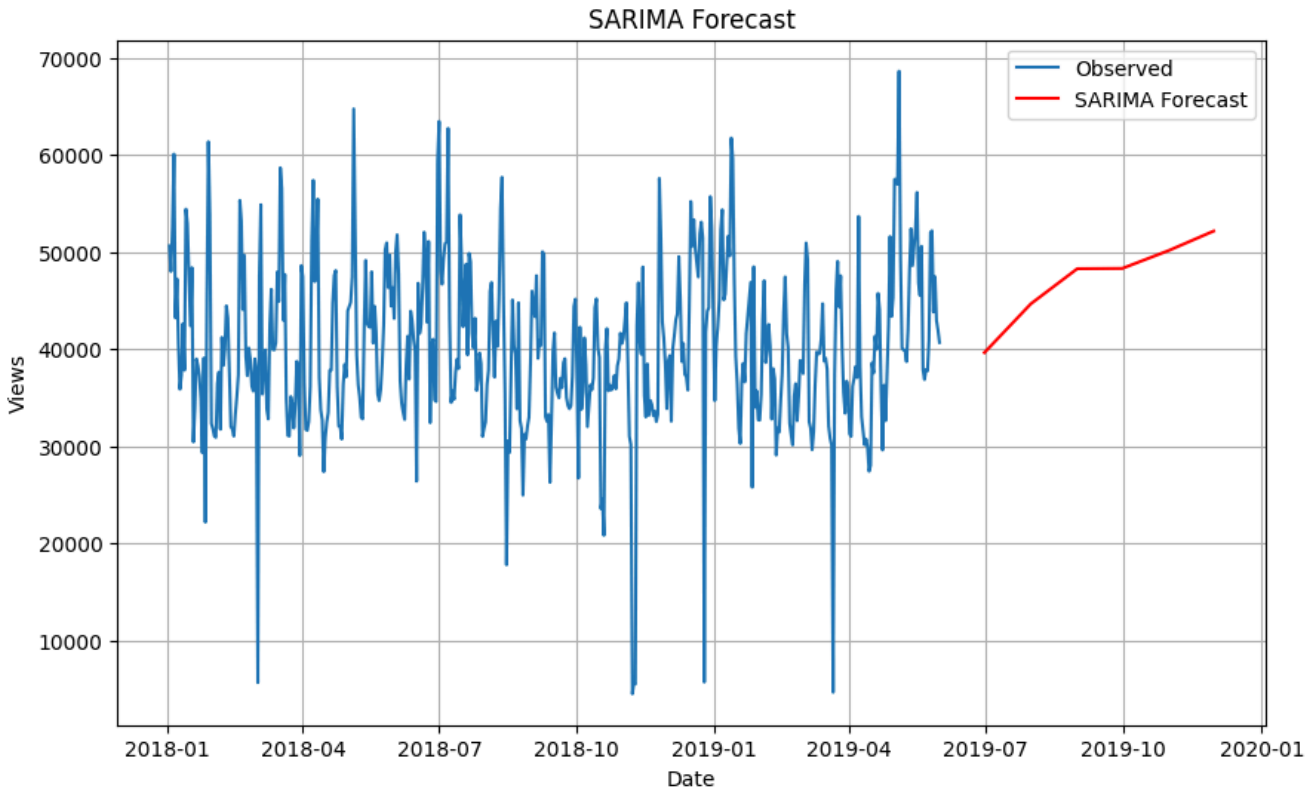
```
/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:

No frequency information was provided, so inferred frequency D will be used.

/usr/local/lib/python3.11/dist-packages/statsmodels/tsa/base/tsa_model.py:473: ValueWarning:

No frequency information was provided, so inferred frequency D will be used.

                                     SARIMAX Results
===================================================================================================
Dep. Variable:                               Sales   No. Observations:                  515
Model:               SARIMAX(1, 1, 1)x(1, 1, 1, 12)   Log Likelihood               -5184.373
Date:                              Sun, 02 Feb 2025   AIC                          10378.746
Time:                                      06:25:31   BIC                          10399.839
Sample:                                  01-02-2018   HQIC                         10387.022
                                       - 05-31-2019
Covariance Type:                                 opg
==============================================================================
```

```
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.5021      0.047     10.749      0.000       0.411       0.594
ma.L1         -0.9049      0.031    -29.488      0.000      -0.965      -0.845
ar.S.L12      -0.0792      0.043     -1.852      0.064      -0.163       0.005
ma.S.L12      -0.9085      0.025    -36.654      0.000      -0.957      -0.860
sigma2      5.264e+07   1.15e-10   4.58e+17      0.000    5.26e+07    5.26e+07
===================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):               538.15
Prob(Q):                              0.97   Prob(JB):                         0.00
Heteroskedasticity (H):               0.77   Skew:                            -0.62
Prob(H) (two-sided):                  0.10   Kurtosis:                         7.92
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
[2] Covariance matrix is singular or near-singular, with condition number 2.01e+33. Standard errors may be unstable.
<ipython-input-204-420951a78713>:7: FutureWarning:

'M' is deprecated and will be removed in a future version, please use 'ME' instead.
```

Vizual representation of SARIMA forecast

```
# Create a new time index for the forecasted period
forecast_index = pd.date_range(start=data.index[-1] + pd.Timedelta(days=30), periods=6, freq='M')

# Plot the results
plt.figure(figsize=(10, 6))

# Plot historical data
plt.plot(data.index, data['Sales'], label='Observed')

# Plot forecasted data
plt.plot(forecast_index, forecast_sarima.predicted_mean, label='SARIMA Forecast', color='red')

# Add labels and legend
plt.title('SARIMA Forecast')
plt.xlabel('Date')
plt.ylabel('Views')
plt.legend()
plt.grid(True)

# Show the plot
plt.show()
```

```
<ipython-input-205-c57737bb49ce>:2: FutureWarning:

'M' is deprecated and will be removed in a future version, please use 'ME' instead.
```



fb Prophet

- A reliable forecast which handles time series data with strong seasonal patterns and missing data.
- It handles Trend, Seasonality, Holiday, Errors etc.

```
# Preparing data to comply with model
data1 = pd.DataFrame({'ds': date_range, 'y': sales_data})
data2 = data1.groupby(data1['ds']).median()
data2.reset_index(inplace=True)
```

```
from prophet import Prophet
```

```
m = Prophet()
```

```
m.fit(data2)

# Making Future predictions
future = m.make_future_dataframe(periods=12*3,
                                 freq='D')
future
```

```
INFO:prophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=True to override this.
INFO:prophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
DEBUG:cmdstanpy:input tempfile: /tmp/tmpmaan3r5d/dzpovowh.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpmaan3r5d/0y5xct00.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.11/dist-packages/prophet/stan_model/prophet_model.bin', 'random', 'seed=57530', 'data', 'file=/tmp/tmpmaan3r5d/dzpovowh.
06:25:31 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
06:25:31 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```

	ds
0	2018-01-01
1	2018-01-02
2	2018-01-03
3	2018-01-04
4	2018-01-05
...	...
547	2019-07-02
548	2019-07-03
549	2019-07-04
550	2019-07-05
551	2019-07-06

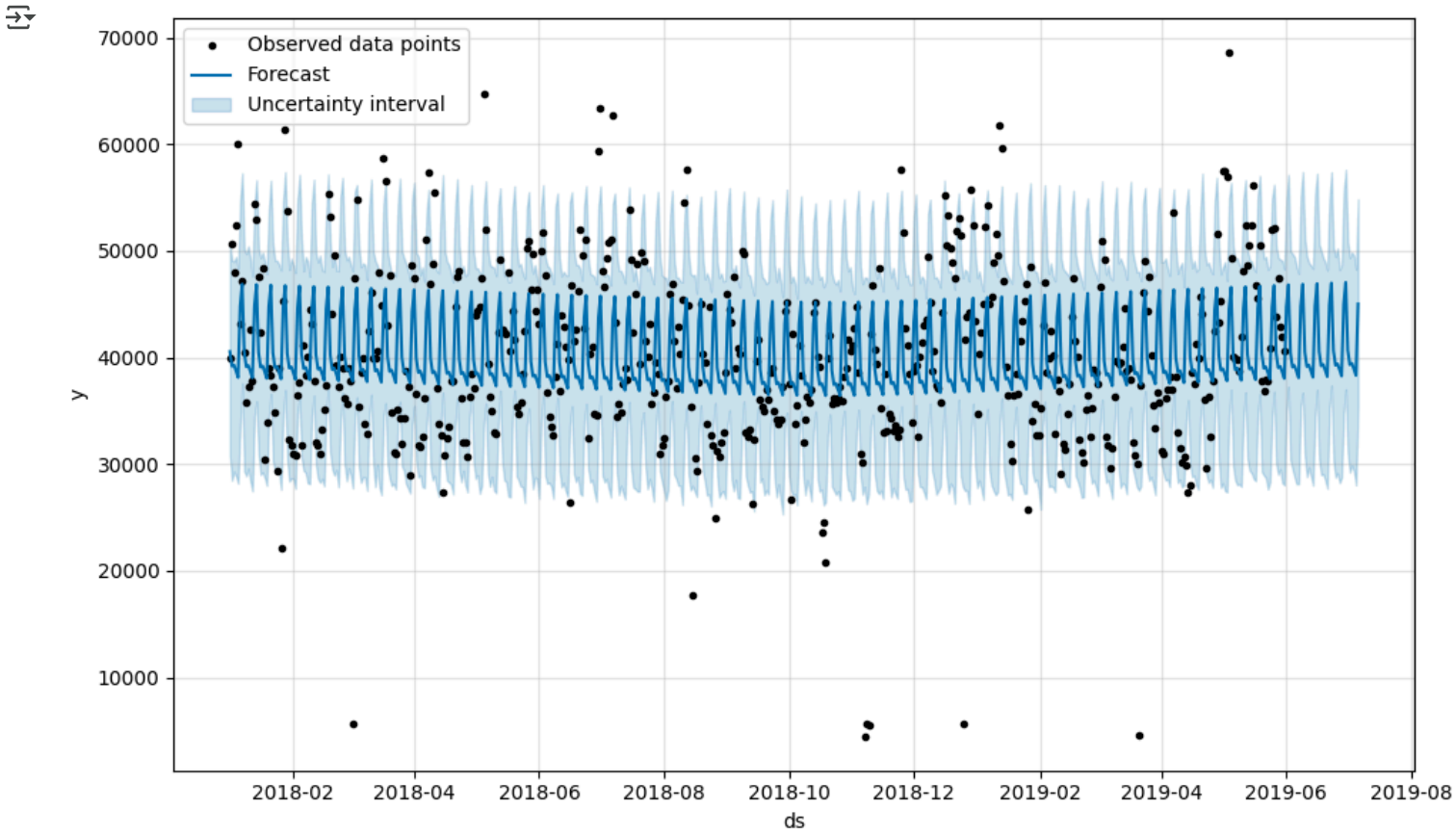
552 rows × 1 columns

Interpretation:

- The yhat values represent the model's forecasted values for future periods (e.g., predicted sales).
- The uncertainty intervals (yhat_lower and yhat_upper) show the model's confidence in the forecast. The larger the interval, the higher the uncertainty.
- The trend line shows the underlying pattern of the data over time. It helps us understand whether the model expects the value to increase, decrease, or remain stable.

The plot visually demonstrates how the forecast fits into the historical data and extends into the future, along with uncertainty bounds.

```
forecast = m.predict(future)
forecast[['ds', 'yhat', 'yhat_lower',
          'yhat_upper', 'trend',
          'trend_lower', 'trend_upper']].tail()

fig1 = m.plot(forecast, include_legend=True)
```



Interpretation:

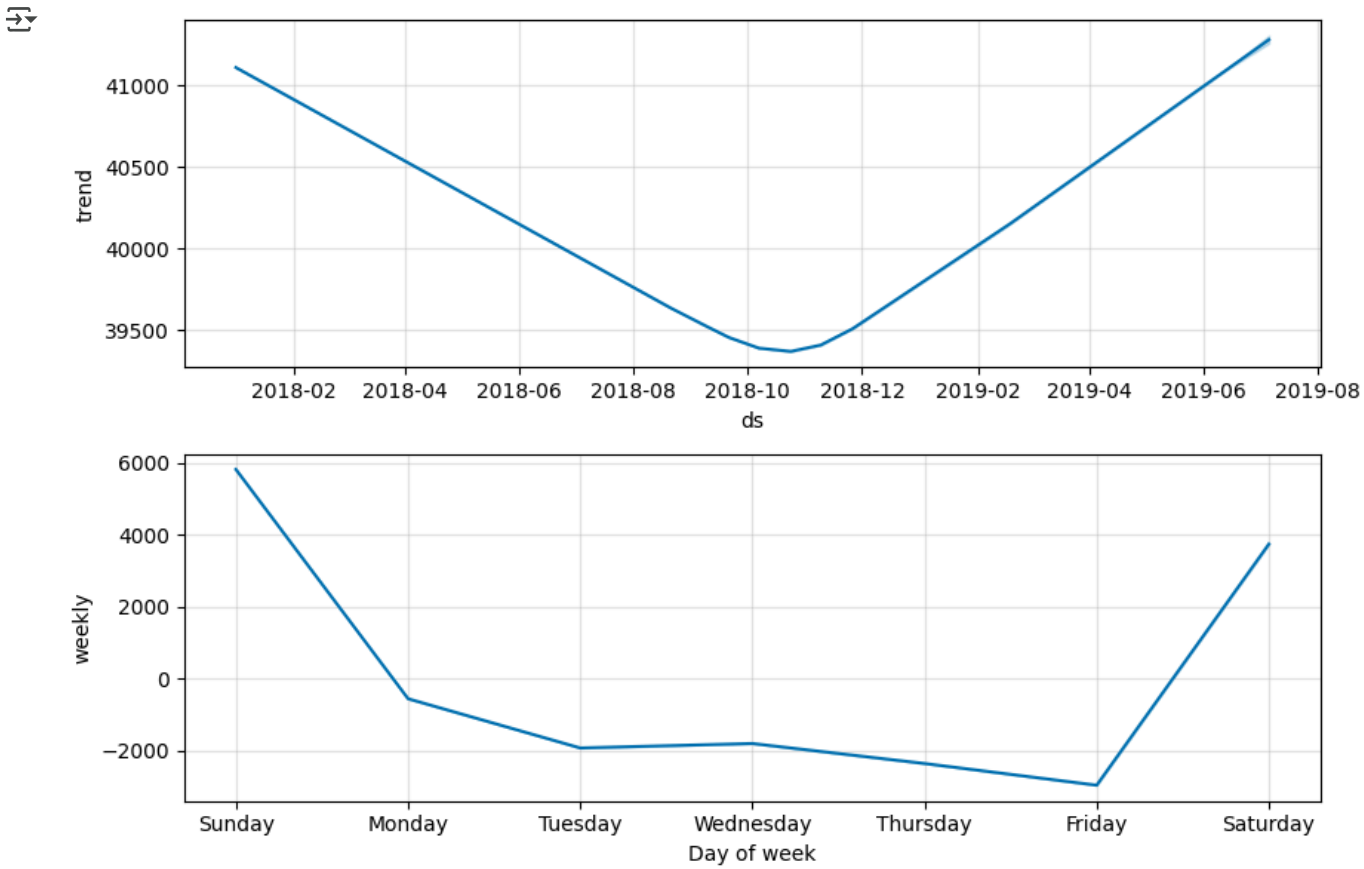
Trend:

- It implies the overall long-term pattern the model has learned based on historical data.
- The trend line helps us understand the broader movement in the data over time, such as the general growth.

Seasonality:

- The seasonality component shows how the data behaves in recurring cycles, here in weekly patterns.
- We could see that Sales increases during holidays rather than working days

```
fig2 = m.plot_components(forecast)
```



Interpretation:

Forecast Line:

The plot will show the predicted values for future periods

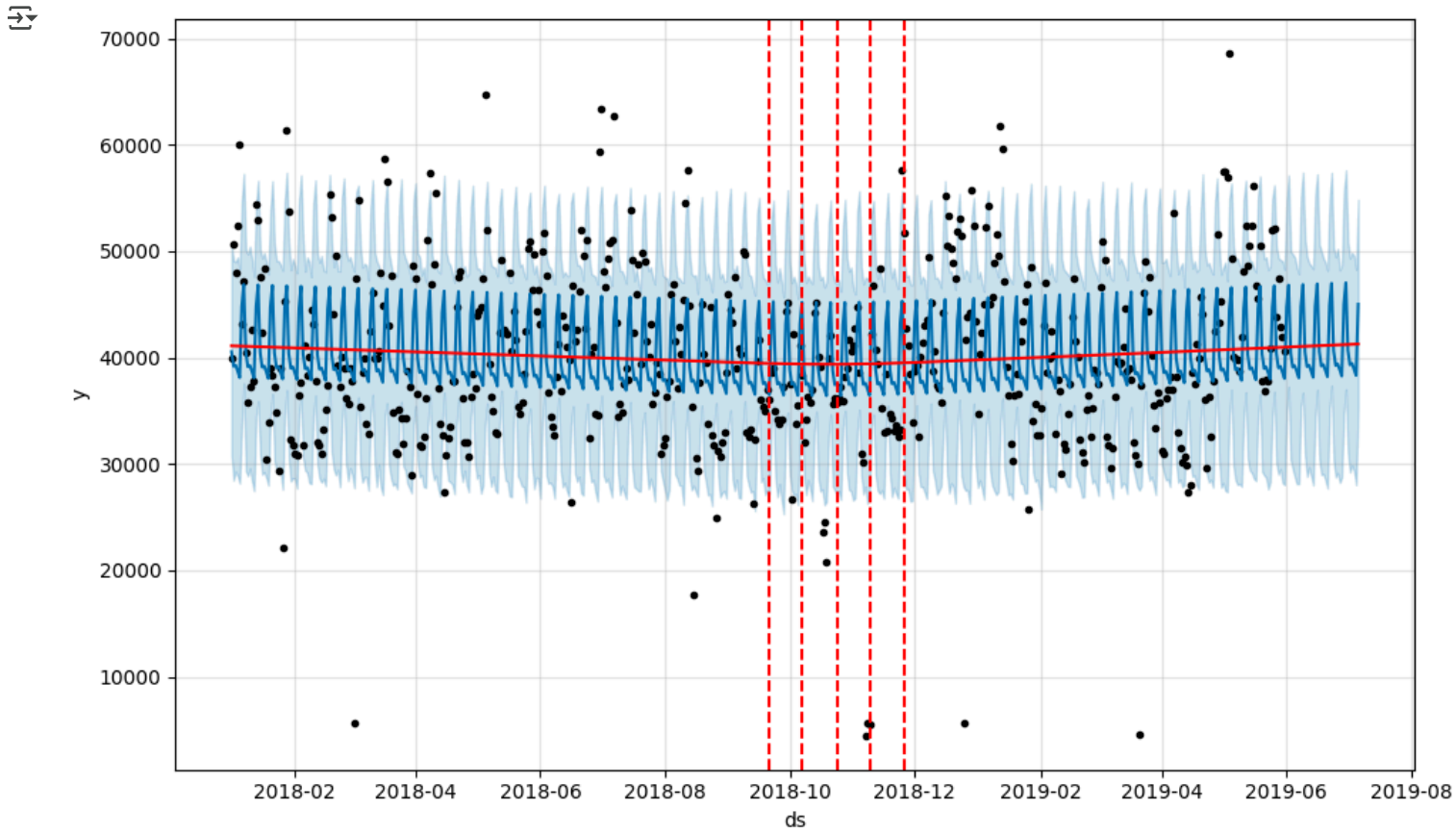
Uncertainty Interval:

The shaded area represents the range within which the actual future values are expected to fall, with the model's confidence.

Changepoints:

The vertical lines indicate when a change in trend occurs. These points are critical to understanding where significant shifts in the data are happening.

```
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(),
                             m, forecast)
```



2. Tree-Based Model: Applying Random Forest to improve performance by capturing complex relationships between features:

Purpose:

It is an ensemble learning method that builds multiple decision trees and combines their predictions to improve overall performance.

- It is particularly effective for capturing complex relationships between features in the data that might be difficult for simpler models to identify.

- It also offers advantages like reducing overfitting, improving accuracy, and handling both numerical and categorical features well.

Components:

- Ensemble of Trees
- Bootstrap Aggregating (Bagging)
- Feature Randomness
- Voting

Implementation:

- Prepare the Data
- Split the Data into Training and Testing Sets
- Train the Random Forest Model
- Make Predicitions
- Evaluate the model (R2 & MAE)

Implementation :

By providing Train & Test data as input model makes the prediction by making the values fit a straight line.

```
R-squared (0.93): The RandomForest model explains 93% of the variance in the target variable, indicating a strong fit and good predictive power
Mean Absolute Error (3177.6): On average, the model's predictions are off by 3177.6, which is acceptable based on the scale of the sales data
```

```
dt.head()
```

Date	ID	Store_id	Store_Type	Location_Type	Region_Code	Holiday	Discount	#Order	Sales	month
2018-01-01	T1000001	1	S1	L3	R1	1	Yes	9	7011.84	1
2018-01-01	T1000002	253	S4	L2	R1	1	Yes	60	51789.12	1
2018-01-01	T1000003	252	S3	L2	R1	1	Yes	42	36868.20	1
2018-01-01	T1000004	251	S2	L3	R1	1	Yes	23	19715.16	1
2018-01-01	T1000005	250	S2	L3	R4	1	Yes	62	45614.52	1

```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df = df1
```

```
dt.head()
```

	ID	Store_id	Store_Type	Location_Type	Region_Code	Date	Holiday	Discount	#Order	Sales
0	T1000001	1	S1	L3	R1	2018-01-01	1	Yes	9	7011.84
1	T1000002	253	S4	L2	R1	2018-01-01	1	Yes	60	51789.12
2	T1000003	252	S3	L2	R1	2018-01-01	1	Yes	42	36868.20
3	T1000004	251	S2	L3	R1	2018-01-01	1	Yes	23	19715.16
4	T1000005	250	S2	L3	R4	2018-01-01	1	Yes	62	45614.52

```
# Encoding Categorical Features using One-Hot Encoding
df_encoded = pd.get_dummies(dt, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount'], drop_first=True)

# Define Features (X) and Target (y)
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)

# Predict
y_pred_rf = rf_model.predict(X_test)

# Evaluate the model's performance
r2_rf = r2_score(y_test, y_pred_rf)
mae_rf = mean_absolute_error(y_test, y_pred_rf)

# Output the evaluation metrics
print(f"Random Forest - R-squared: {r2_rf:.4f}")
print(f"Random Forest - Mean Absolute Error: {mae_rf:.4f}")
```

```
Random Forest - R-squared: 0.9385
Random Forest - Mean Absolute Error: 3177.6702
```

Actual vs. Predicted Values

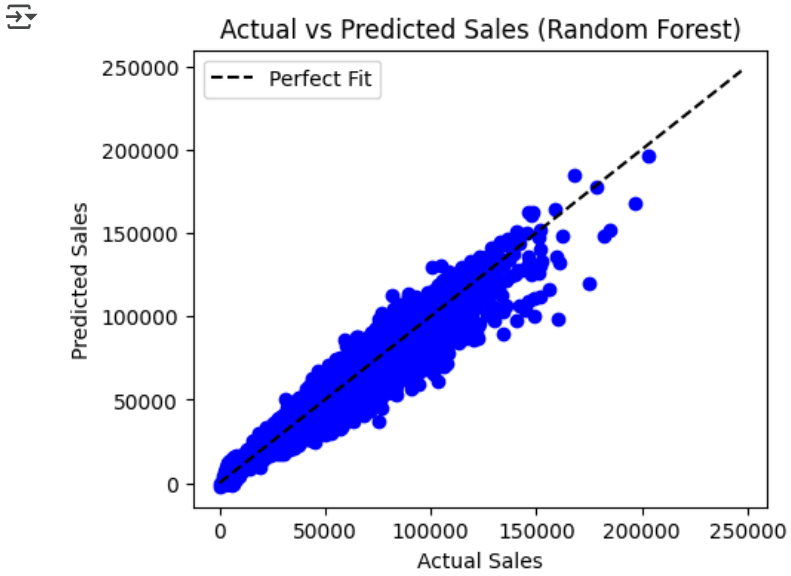
Purpose :

- This scatter plot compares the actual Sales values (y_test) against the predicted values (y_pred).

Interpretation :

- Ideally, the points should lie along the diagonal line, which represents the case where predicted values are equal to actual values. Any significant deviation from this line indicates a higher error.

```
#  Actual vs Predicted Values
plt.figure(figsize=(5, 4))
plt.scatter(y_test, y_pred, color='blue')
plt.plot([y.min(), y.max()], [y.min(), y.max()], '--k', label='Perfect Fit')
plt.xlabel('Actual Sales')
plt.ylabel('Predicted Sales')
plt.title('Actual vs Predicted Sales (Random Forest)')
plt.legend()
plt.show()
```



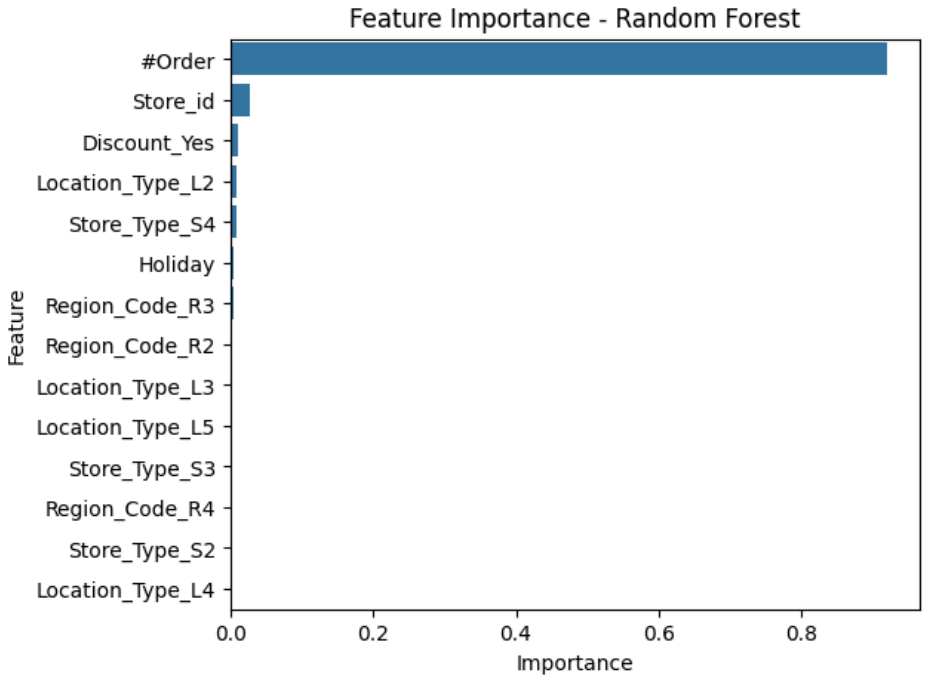
Feature Importance

Which refers to the technique used to determine the relative importance of each feature in predicting the target variable. It helps identify which features have the most influence on the model's predictions, allowing for better model interpretation and potential feature selection.

From below we can see that feature - Order# has high importance

```
# Feature Importance Feature Importance
feature_importances = rf_model.feature_importances_
features = X.columns
importance_df = pd.DataFrame({
    'Feature': features,
    'Importance': feature_importances
}).sort_values(by='Importance', ascending=False)

plt.figure(figsize=(6, 5))
sns.barplot(x='Importance', y='Feature', data=importance_df)
```

```
plt.title('Feature Importance - Random Forest')
plt.show()
```



3. Gradient Boosting Model: XGBoost XGBoost is a powerful boosting algorithm that often yields state-of-the-art results:

Purpose:

Gradient Boosting is an ensemble learning technique that builds multiple weak learners (usually decision trees) sequentially. Each new tree tries to correct the errors made by the previous tree. It does so by fitting the new tree to the residual errors of the previous model, meaning it focuses on learning where the previous models made mistakes.

XGBoost is particularly known for its speed, accuracy, and ability to handle a variety of data types and challenges, such as missing values and overfitting.

Components:

- Boosting Process
- Gradient Descent Optimization
- Regularization

- Shrinkage

Implementation:

- Install XGBoost
- Prepare Data
- Convert Data to DMatrix Format
- Define the Model
- Train the Model
- Make Predictions and Evaluate the Model
- Feature Importance

```
R-squared (0.94): The XGBoost model explains 94% of the variance in the target variable, indicating a strong fit and goo
Mean Absolute Error (2980.2): On average, the model's predictions are off by 2980.2, which is acceptable based on the sc
```

```
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize and train the XGBoost model
xgb_model = xgb.XGBRegressor(objective='reg:squarederror', n_estimators=100, random_state=42)
xgb_model.fit(X_train, y_train)

# Predict on the test set
y_pred_xgb = xgb_model.predict(X_test)

# Evaluate the model's performance
r2_xgb = r2_score(y_test, y_pred_xgb)
mae_xgb = mean_absolute_error(y_test, y_pred_xgb)

# Output the evaluation metrics
print(f"XGBoost - R-squared: {r2_xgb:.4f}")
print(f"XGBoost - Mean Absolute Error: {mae_xgb:.4f}")
```

```
XGBoost - R-squared: 0.9472
XGBoost - Mean Absolute Error: 2980.2512
```

Actual vs. Predicted Values

Purpose :

- This scatter plot compares the actual Sales values (y_test) against the predicted values (y_pred).

Interpretation :

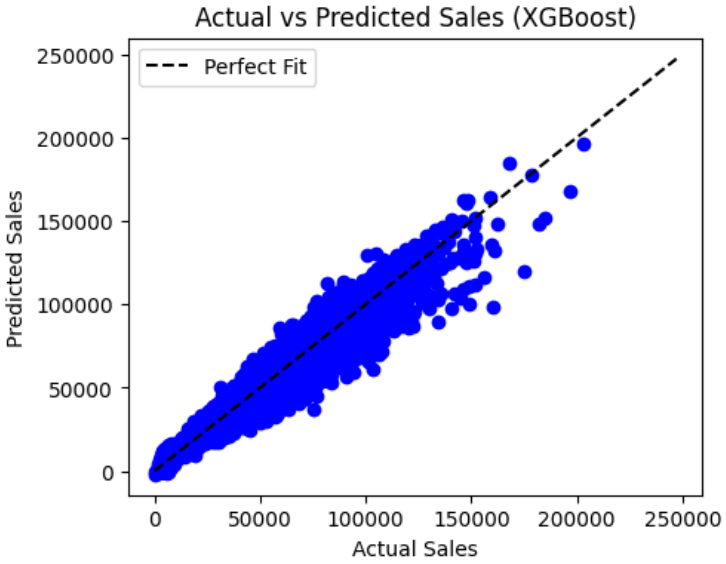
- Ideally, the points should lie along the diagonal line, which represents the case where predicted values are equal to actual values. Any significant deviation from this line indicates a higher error.

Feature Importance

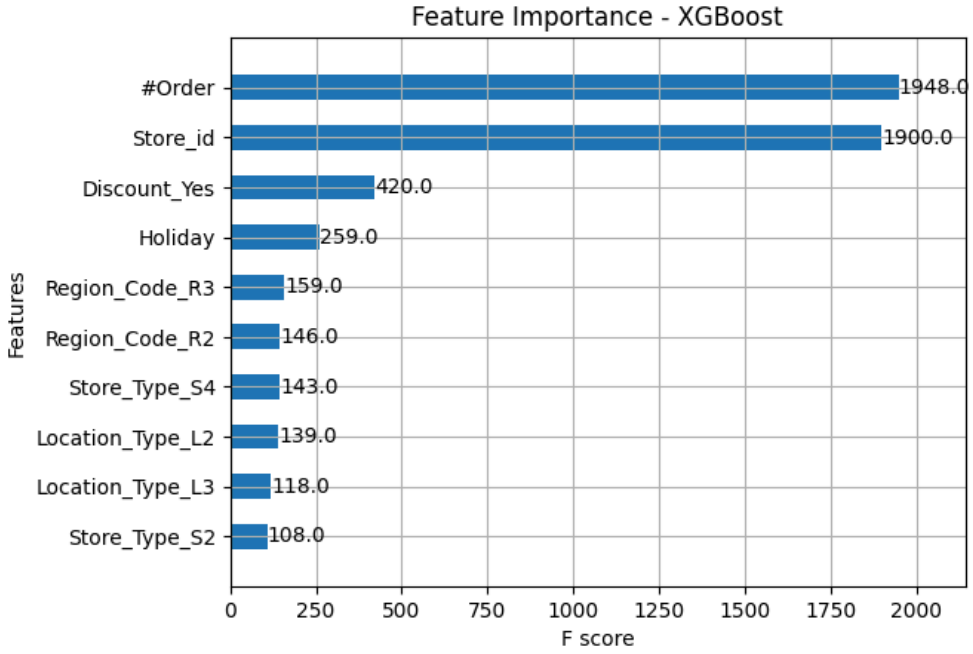
Which refers to the technique used to determine the relative importance of each feature in predicting the target variable. It helps identify which features have the most influence on the model's predictions, allowing for better model interpretation and potential feature selection.

From below we can see that feature - Order# has high importance

```
# Actual vs Predicted Values
plt.figure(figsize=(5, 4))
plt.scatter(y_test, y_pred, color='blue')
plt.plot([y.min(), y.max()], [y.min(), y.max()], '--k', label='Perfect Fit')
plt.xlabel('Actual Sales')
plt.ylabel('Predicted Sales')
plt.title('Actual vs Predicted Sales (XGBoost)')
plt.legend()
plt.show()

# Feature Importance (Bar Plot)
plt.figure(figsize=(5, 4))
plot_importance(xgb_model, importance_type='weight', max_num_features=10, height=0.5)
plt.title('Feature Importance - XGBoost')
plt.show()
```



<Figure size 500x400 with 0 Axes>



4. Deep Learning Model: LSTM for Time Series Forecasting implementing an LSTM model for forecasting future sales. LSTMs are ideal for sequential data like time series.

Purpose:

These models are capable of learning long-term dependencies in data, which makes them highly effective for forecasting tasks where future values are influenced by historical observations.

Components:

- Ensemble of Trees
- Bootstrap Aggregating (Bagging)
- Feature Randomness
- Voting

Implementation:

- Prepare the Data
- Normalize the Data
- Create Time Series Sequences
- Split the Data into Training and Test Sets
- Build the LSTM Model
- Train & Evaluate the model
- Make Predictions

Interpretation:

- The model predicts that sales will fluctuate slightly across the next 5 periods, with values ranging from 42,492.63 to 44,855.38.
- There is no drastic change in sales, suggesting relative stability or moderate growth/decline over the forecasted periods.
- This prediction can be useful for making business decisions related to stock, staffing, or marketing strategies.

```
# Prepare data for LSTM
scaler = MinMaxScaler(feature_range=(0, 1))
sales_scaled = scaler.fit_transform(sales.values.reshape(-1, 1))

# Create sequences for LSTM model (using past 'n' days to predict the next day)
def create_sequence(data, n_steps=3):
    X, y = [], []
    for i in range(len(data) - n_steps):
        X.append(data[i:i + n_steps, 0])
        y.append(data[i + n_steps, 0])
    return np.array(X), np.array(y)

n_steps = 10
X, y = create_sequence(sales_scaled, n_steps)
```

```
# Reshape data for LSTM
X = X.reshape((X.shape[0], X.shape[1], 1))
# Build the LSTM model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(X.shape[1], 1)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mean_squared_error')

# Train the model
model.fit(X, y, epochs=5, batch_size=10, verbose=0)
# Make predictions
predicted_sales_scaled = model.predict(X[-5:].reshape(5, n_steps, 1))

# Inverse transform predictions to original scale
predicted_sales = scaler.inverse_transform(predicted_sales_scaled)

print(f"Predicted Sales for next 5 periods: {predicted_sales.flatten()}")
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 344ms/step
Predicted Sales for next 5 periods: [44490.582 42501.65  44308.453 43532.277 44434.844]
```

Actual vs. Predicted Values

Purpose :

- This below plot compares the actual Sales values (y_test) against the predicted values (y_pred).

Interpretation :

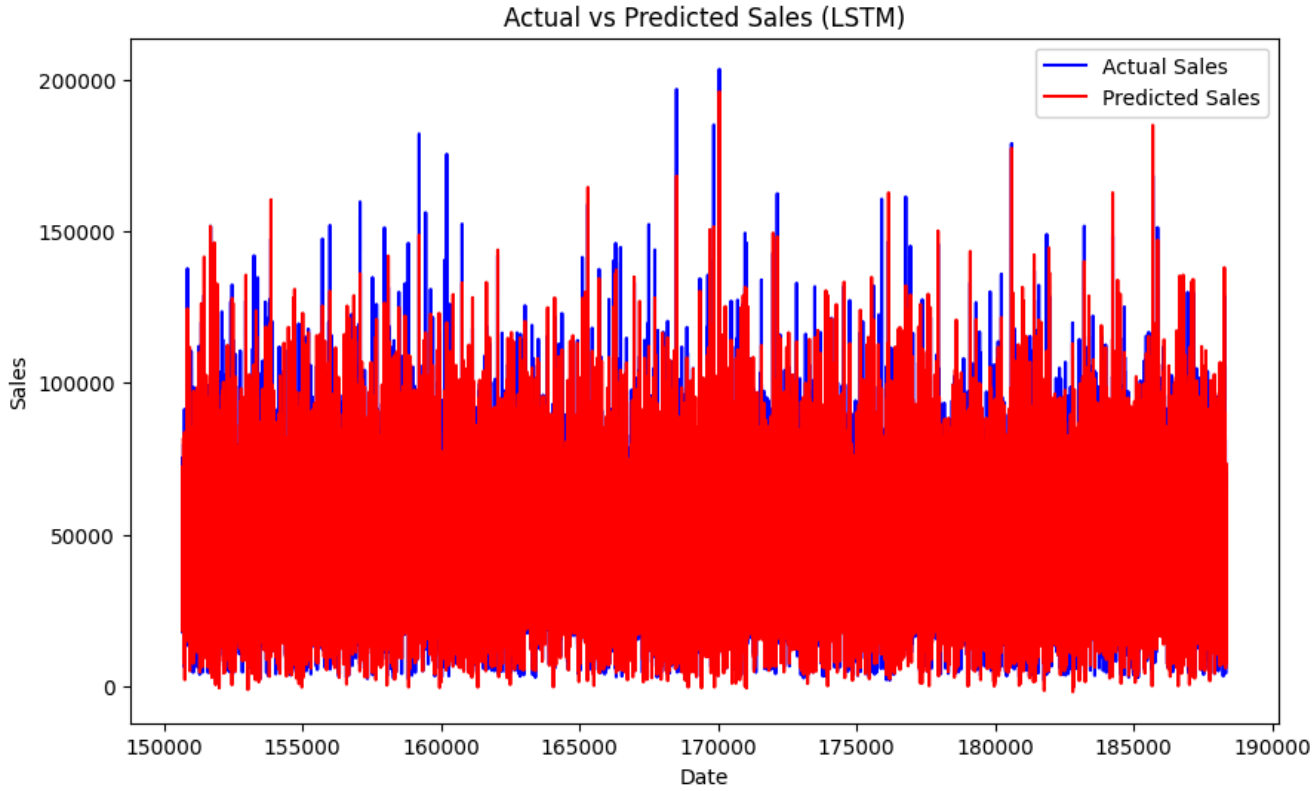
- The plot helps visualize how closely the predicted sales (red line) follow the actual sales (blue line), giving an insight into the model's performance.

```
R-squared (60.6522): A value of 60.6522 suggests that the model tends to have relatively small errors.
Mean Absolute Error (3678.6912): On average, the model's predictions are off by 3,678.69. This represents the average ma
```

```
# Evaluation metrics
print(f"Mean Absolute Error: {mae:.4f}")
rmse = np.sqrt(mae)
print(f"Root Mean Squared Error: {rmse:.4f}")

# Plot the results (Actual vs Predicted Sales)
plt.figure(figsize=(10, 6))
```

```
plt.plot(df.index[len(df) - len(y_test):], y_test, color='blue', label='Actual Sales')
plt.plot(df.index[len(df) - len(y_pred):], y_pred, color='red', label='Predicted Sales')
plt.title('Actual vs Predicted Sales (LSTM)')
plt.xlabel('Date')
plt.ylabel('Sales')
plt.legend()
plt.show()
```

```
Mean Absolute Error: 3678.6912
Root Mean Squared Error: 60.6522
```




```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df1
```

Model Comparison:

- XGBoost performs the best with the lowest MAE (2054.81) and RMSE (3174.78), along with the highest R^2 (0.9702), indicating it makes the most accurate predictions and explains the highest proportion of variance in the data.

- Random Forest also performs well with a slightly higher MAE (2280.82) and RMSE (3559.49), but still offers a strong R^2 (0.9625), indicating robust predictive performance.
- LSTM has a higher MAE (3516.16) and RMSE (4827.48) compared to XGBoost and Random Forest, and its R^2 (0.9311) is slightly lower comparatively, but still it provides a good accuracy.
- Linear Regression has highest MAE (4055.23), RMSE (5512.46), and the lowest R^2 (0.9101), Comparatively which is low but still provides good data variances

```
# Convert categorical variables to numerical values using LabelEncoder
label_encoder = LabelEncoder()
dt['Store_Type'] = label_encoder.fit_transform(dt['Store_Type'])
dt['Location_Type'] = label_encoder.fit_transform(dt['Location_Type'])
dt['Region_Code'] = label_encoder.fit_transform(dt['Region_Code'])
dt['Discount'] = dt['Discount'].apply(lambda x: 1 if x == 'Yes' else 0)

# Convert Date to datetime and extract features (e.g., year, month)
dt['Date'] = pd.to_datetime(dt['Date'])
dt['Year'] = dt['Date'].dt.year
dt['Month'] = dt['Date'].dt.month

# Feature and target columns
X = dt[['Store_id', 'Store_Type', 'Location_Type', 'Region_Code', 'Holiday', 'Discount', '#Order', 'Year', 'Month']]
y = dt['Sales']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# --- Random Forest ---
rf_model = RandomForestRegressor(random_state=42)
rf_model.fit(X_train, y_train)

# Predictions
rf_pred = rf_model.predict(X_test)

# Evaluation
rf_mae = mean_absolute_error(y_test, rf_pred)
rf_rmse = np.sqrt(mean_squared_error(y_test, rf_pred))
rf_r2 = r2_score(y_test, rf_pred)

# --- XGBoost ---
xgb_model = xgb.XGBRegressor(random_state=42)
xgb_model.fit(X_train, y_train)

# Predictions
xgb_pred = xgb_model.predict(X_test)

# Evaluation
xgb_mae = mean_absolute_error(y_test, xgb_pred)
xgb_rmse = np.sqrt(mean_squared_error(y_test, xgb_pred))
```

```
xgb_r2 = r2_score(y_test, xgb_pred)

# --- LSTM (Long Short-Term Memory) ---
# Reshape the data for LSTM
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

X_train_lstm = X_train_scaled.reshape((X_train_scaled.shape[0], 1, X_train_scaled.shape[1]))
X_test_lstm = X_test_scaled.reshape((X_test_scaled.shape[0], 1, X_test_scaled.shape[1]))

lstm_model = Sequential()
lstm_model.add(LSTM(units=50, activation='relu', input_shape=(X_train_lstm.shape[1], X_train_lstm.shape[2])))
lstm_model.add(Dense(1))

lstm_model.compile(optimizer='adam', loss='mean_squared_error')
lstm_model.fit(X_train_lstm, y_train, epochs=10, batch_size=8, verbose=0)

# Predictions
lstm_pred = lstm_model.predict(X_test_lstm)

# Evaluation
lstm_mae = mean_absolute_error(y_test, lstm_pred)
lstm_rmse = np.sqrt(mean_squared_error(y_test, lstm_pred))
lstm_r2 = r2_score(y_test, lstm_pred)

# ---Linear Regression ---
lr_model = LinearRegression()
lr_model.fit(X_train, y_train)

# Predictions
lr_pred = lr_model.predict(X_test)

# Evaluation
lr_mae = mean_absolute_error(y_test, lr_pred)
lr_rmse = np.sqrt(mean_squared_error(y_test, lr_pred))
lr_r2 = r2_score(y_test, lr_pred)

# --- Evaluation Results ---
print("Random Forest - MAE:", rf_mae)
print("Random Forest - RMSE:", rf_rmse)
print("Random Forest - R2:", rf_r2)

print("\nXGBoost - MAE:", xgb_mae)
print("XGBoost - RMSE:", xgb_rmse)
print("XGBoost - R2:", xgb_r2)

print("\nLSTM - MAE:", lstm_mae)
print("LSTM - RMSE:", lstm_rmse)
print("LSTM - R2:", lstm_r2)
```

```
print("\nLinear Regression - MAE:", lr_mae)
print("Linear Regression - RMSE:", lr_rmse)
print("Linear Regression - R2:", lr_r2)
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning:

Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

1178/1178 ━━━━━━━━━━━━━━━━━━━━ 2s 1ms/step
Random Forest - MAE: 2280.8171390387324
Random Forest - RMSE: 3559.4889724838013
Random Forest - R2: 0.9625231761773276

XGBoost - MAE: 2054.8109346386004
XGBoost - RMSE: 3174.7825194516136
XGBoost - R2: 0.970186330395761

LSTM - MAE: 3490.4915862524435
LSTM - RMSE: 4805.464713804225
LSTM - R2: 0.9316940935917044

Linear Regression - MAE: 4055.2341344640818
Linear Regression - RMSE: 5512.462981117447
Linear Regression - R2: 0.9101167338086359
```

```
Start coding or generate with AI.
```

3. Model Evaluation and Validation:

Model evaluation involves assessing a model's performance using metrics like accuracy, precision, recall, R-squared, MAE, and RMSE on a test dataset to understand how well it generalizes to unseen data.

Validation ensures that the model's performance is consistent across different subsets of the data, often using techniques like cross-validation.

This process helps prevent overfitting and ensures the model is reliable and robust for deployment.

```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df1
```

Cross-Validation: Implement time-series specific cross-validation techniques to evaluate model performance over different temporal splits of the data.

- Time-series cross-validation ensures that the temporal order of data is maintained, preventing future data from being used to predict past events.

- Techniques like rolling-window or expanding-window splits are used, where the model is trained on past data and tested on future data.
- This method evaluates how well the model generalizes over different time periods and prevents data leakage.

Interpretations:

The model evaluation shows the following:

MAE (Mean Absolute Error):

- The MAE for each fold ranges from 2499.98 to 4317.44, with an average of 3397.08. This indicates that, on average, the model's predictions deviate by about 3397.08 units from the actual values.

RMSE (Root Mean Squared Error):

- The RMSE values range from 3836.37 to 5593.73, with an average of 4936.23. Since RMSE penalizes larger errors more heavily, this suggests that the model has some larger prediction errors but generally performs reasonably well.

R2 (R-squared):

- The R2 values range from 0.90 to 0.96, with an average of 0.93. This indicates that the model explains about 93% of the variance in the data, which is a strong performance.

From below we can conclude that **the model performs well**, with high R2 indicating strong predictive power, and MAE and RMSE showing moderate prediction errors. There's a slight variance in performance across the folds, but overall, the model generalizes well.

```
df1 = pd.read_csv('TRAIN.csv')
df1.head()
dt = df1
dt['Date'] = pd.to_datetime(dt['Date'])
dt['month'] = dt['Date'].dt.month
dt.head()

dt['Date'] = pd.to_datetime(dt['Date'])
dt['month'] = dt['Date'].dt.month



# Convert Date column to datetime
dt['Date'] = pd.to_datetime(dt['Date'])

# Sort data by Date
dt = dt.sort_values('Date')

# Encode categorical features
df_encoded = pd.get_dummies(dt, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount'], drop_first=True)

# Define Features (X) and Target (y)
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']
tscv = TimeSeriesSplit(n_splits=3)
model = RandomForestRegressor()
```

```
# Variables to store evaluation metrics
mae_scores = []
rmse_scores = []
r2_scores = []

for train_index, test_index in tscv.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    # Train the model
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    # Evaluate performance
    mae = mean_absolute_error(y_test, y_pred)
    rmse = np.sqrt(mean_squared_error(y_test, y_pred))
    r2 = r2_score(y_test, y_pred)

    # Append the results
    mae_scores.append(mae)
    rmse_scores.append(rmse)
    r2_scores.append(r2)

print(f"MAE scores for each fold: {mae_scores}")
print(f"RMSE scores for each fold: {rmse_scores}")
print(f"R2 scores for each fold: {r2_scores}")

print(f"Average MAE: {np.mean(mae_scores):.4f}")
print(f"Average RMSE: {np.mean(rmse_scores):.4f}")
print(f"Average R2: {np.mean(r2_scores):.4f}")
```

```
MAE scores for each fold: [4300.810411006436, 3370.361766783373, 2498.518439902927]
RMSE scores for each fold: [5573.203511614997, 5376.231965918349, 3831.3785080786006]
R2 scores for each fold: [0.9044962241168568, 0.9203354916248044, 0.960496223125974]
Average MAE: 3389.8969
Average RMSE: 4926.9380
Average R2: 0.9284
```

Performance Metrics: Use metrics appropriate for regression tasks, such as MAE (Mean Absolute Error), MSE (Mean Squared Error), RMSE (Root Mean Squared Error), and MAPE (Mean Absolute Percentage Error).

Performance metrics for regression tasks help evaluate the accuracy of predictions.

MAE (Mean Absolute Error):

- The MAE of 35,815.65 indicates that, on average, the model's predictions are off by about 35,815.65 units from the actual values. This gives a sense of the model's overall prediction accuracy.

MSE (Mean Squared Error):

- The MSE of 1,620,327,772.24 is very large, which suggests that the model has some significant prediction errors. MSE penalizes larger errors more heavily due to the squaring of differences, indicating that the model might be producing large outliers in predictions.

RMSE (Root Mean Squared Error):

- The RMSE of 40,253.30 represents the square root of the MSE and gives a more interpretable scale. It's still large, showing that the model has substantial prediction errors, which could affect the model's reliability.

MAPE (Mean Absolute Percentage Error):

- The MAPE is infinite (inf%), which suggests that the model is making predictions where the actual values are zero or close to zero in some cases. Since MAPE involves division by the actual values, it cannot handle zero values and thus leads to an "infinite" result.

```
# Feature Engineering: Encoding categorical variables
df_encoded = pd.get_dummies(dt, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount'], drop_first=True)

# Define Features (X) and Target (y)
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']

# Split data into train and test sets (using the full data for this example)
X_train, X_test = X[:1], X[1:]
y_train, y_test = y[:1], y[1:]

# Train a Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Calculate Performance Metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
mape = np.mean(np.abs((y_test - y_pred) / y_test)) * 100  # MAPE as percentage

# Output the metrics
print(f"MAE: {mae:.4f}")
print(f"MSE: {mse:.4f}")
print(f"RMSE: {rmse:.4f}")
print(f"MAPE: {mape:.2f}%")
```

```
MAE: 35815.6460
MSE: 1620327772.2448
RMSE: 40253.2952
MAPE: inf%
```

Residual Analysis: Analyze the residuals to ensure there are no patterns left unmodeled.

Performance metrics for regression tasks help evaluate the accuracy of predictions.

MAE (Mean Absolute Error):

- The MAE of 3653.6488 indicates that, on average, the model's predictions are off by about 3653.6488 units from the actual values. This gives a sense of the model's overall prediction accuracy.

Residual vs Predicted Sales:

This plot shows the relationship between the residuals (the difference between actual and predicted values) and the predicted sales.

- Since there is no pattern (i.e., the residuals are scattered randomly around zero), it suggests that the model is well-fitted and has captured the underlying patterns in the data.

Histogram of Residuals: The histogram shows the distribution of the residuals (errors). It helps assess the normality of the residuals.

- As its normally distributed but very slightly skewed towards other side we can assume that errors are randomly distributed assuming normality

Residual vs Time: This plot shows the residuals over time, helping to check for time-based patterns that the model may not have captured.

- Since there are no patterns, residuals appear randomly scattered, it indicates that the model has accounted for any temporal dependencies in the data.

Autocorrelation of Residuals:

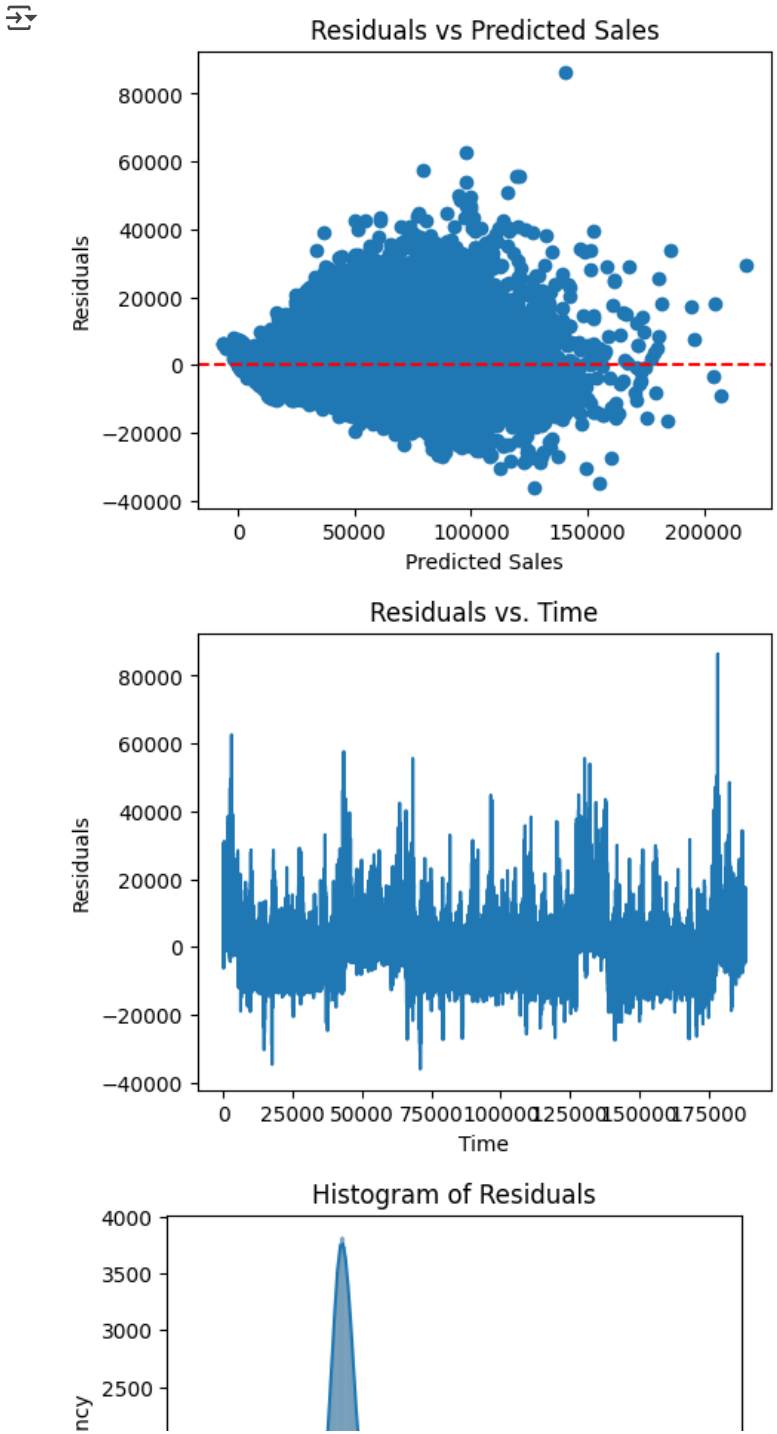
This plot or test checks if residuals are correlated with their own lagged values. It's commonly used in time series data to detect if there are temporal structures not captured by the model.

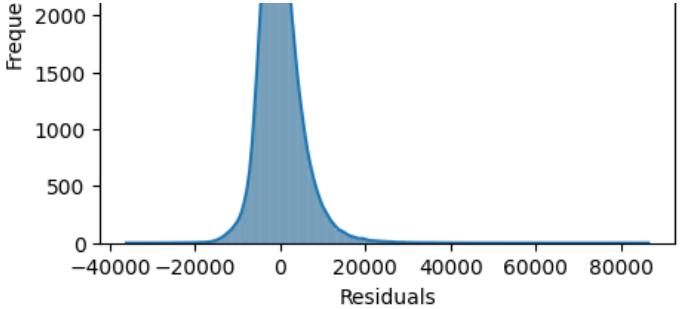
- There's no significant autocorrelation at any lag (i.e., values are close to zero), it suggests that the residuals are independent, and the model has captured all the temporal dependencies.

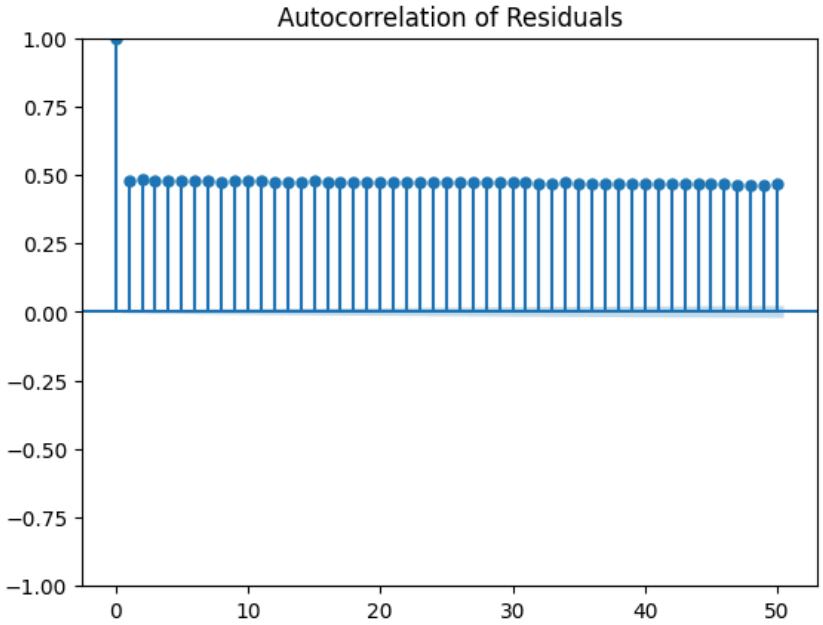
Overall **residuals are randomly scattered** without clear patterns in any of these plots (residual vs predicted, residual vs time, and autocorrelation), it suggests a **good model fit**.

```
# One-hot encode categorical columns
df_encoded = pd.get_dummies(dt, columns=['Store_Type', 'Location_Type', 'Region_Code', 'Discount'], drop_first=True)

# Define features and target
X = df_encoded.drop(columns=['Sales', 'ID', 'Date'])
y = df_encoded['Sales']

# Train a Linear Regression model
model = LinearRegression()
model.fit(X, y)

# Make predictions
y_pred = model.predict(X)
```

```
# Calculate residuals
residuals = y - y_pred

# Plotting residuals vs predicted values
plt.figure(figsize=(5, 4))
plt.scatter(y_pred, residuals)
plt.axhline(y=0, color='red', linestyle='--')
plt.title('Residuals vs Predicted Sales')
plt.xlabel('Predicted Sales')
plt.ylabel('Residuals')
plt.show()


plt.figure(figsize=(5, 4))
plt.plot(residuals)
plt.title("Residuals vs. Time")
plt.xlabel("Time")
plt.ylabel("Residuals")
plt.show()

# Histogram of residuals
plt.figure(figsize=(5, 4))
sns.histplot(residuals, kde=True)
plt.title("Histogram of Residuals")
plt.xlabel("Residuals")
plt.ylabel("Frequency")
plt.show()

# Plot ACF of residuals to check for autocorrelation
plot_acf(residuals, lags=50)
plt.title("Autocorrelation of Residuals")
plt.show()

# Calculate Mean Absolute Error as an additional residual analysis
mae = mean_absolute_error(y, y_pred)
print(f"Mean Absolute Error (MAE): {mae:.4f}")
```


Residuals vs Predicted Sales
Residuals
Predicted Sales
80000
60000
40000
20000
0
−20000
−40000
0
50000
100000
150000
200000
Residuals vs. Time
Residuals
Time
0
25000
50000
75000
100000
125000
150000
175000
Histogram of Residuals
4000
3500
3000
2500





Mean Absolute Error (MAE): 3653.6488

Recommendations

1. **Understand the Customer behavior**

Research customer behavior and preferences to tailor the products and marketing strategies effectively.

2. **Optimize Inventory Management**

Maintain the right balance between supply and demand by monitoring inventory levels and reducing stockouts or excess stock.

3. **Offer Promotions and Discounts**

Use targeted promotions, seasonal discounts, and limited-time offers to boost sales and attract customers.

4. **Leverage Social Media Marketing**

Promote products on social media platforms to increase brand awareness, reach a wider audience, and drive sales.

5. **Personalize Customer Experience**

Provide personalized recommendations, offers, and experiences based on customer data to enhance engagement and sales.

6. **Improve Website User Experience**

Optimize the website for faster loading times, mobile responsiveness, and easy navigation to enhance online shopping experiences.

7. **Enhance Customer Support**

Offer excellent customer service via multiple channels (chat, email, phone) to build trust and loyalty, leading to repeat sales.

8. **Utilize Influencer Marketing**

Partner with influencers in current industry to increase visibility and drive targeted traffic to the product.

9. **Implement Referral Programs**

Encourage existing customers to refer others by offering incentives like discounts, free products, or loyalty points.

10. **Improve Product Availability**

Ensure the products are available where the customers shop, whether in-store, online, or via third-party retailers.

11. **Track Competitor Performance**

Monitor competitors' pricing, promotions, and sales strategies to stay competitive in the market and adjust the approach.

12. **Leverage Customer Reviews and Testimonials**

Encourage happy customers to leave reviews. Positive testimonials build social proof and attract more buyers.

13. **Introduce New Product Variants**

Keep the product offerings fresh by introducing new colors, sizes, or features to cater to different customer needs.

14. **Diversify Sales Channels**

Sell on multiple platforms like e-commerce sites, marketplaces (Amazon, eBay), or physical stores to reach more customers.

15. **Conduct A/B Testing**

Regularly test different sales strategies, landing pages, and marketing tactics to identify the most effective approaches.

16. **Build Partnerships with Retailers**

Strengthen relationships with wholesalers and retailers to expand distribution networks and increase product reach.

17. **Focus on Sustainability**

Market the product as environmentally friendly or sustainable if it aligns with out brand values to attract conscious consumers.

18. **Offer Subscription Models**

Create subscription-based pricing for products with consistent demand, ensuring recurring revenue and customer retention.

19. **Engage in Cross-Selling and Upselling**

Promote complementary products (cross-selling) or higher-value items (upselling) to increase the average order value and sales.

By implementing these recommendations, we can not only increase sales but also improve customer retention and brand loyalty.