Product Sales Forecasting using Quantitative Methods

Project Objective

Goal: The goal of this project was to apply various quantitative methods, (i.e. Times Series Models and Causal Models) to forecast the sales of the products available in the dataset.

- Perform time series analysis to understand the data and trends
- Use multiple forecasting models on train dataset
- Finally select the best model to run the test data

Models covered in the notebook include:

1. Seasonal Naive Model
2. Holt-Winters Model (Triple Exponential Smoothing)
3. ARIMA Model and Seasonal ARIMA Models
4. Linear Regression Model

Import the required libraries

```
%matplotlib inline
from matplotlib import pyplot as plt
import pandas as pd
from pandas.plotting import register_matplotlib_converters
register_matplotlib_converters()
import numpy as np
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.linear_model import LinearRegression
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression
```

Load Dataset

The dataset contanins historical sales records of 10 stores and 50 products, from the year 2013 through 2017.

```
1 df = pd.read_csv('/content/train.csv')
2 df
```

	date	store	item	sales	f0	f1
0	01-01-2013	1	1	13.0	9.0	NaN
1	02-01-2013	1	1	11.0	NaN	NaN
2	03-01-2013	1	1	14.0	NaN	NaN
3	04-01-2013	1	1	13.0	NaN	NaN
4	05-01-2013	1	1	10.0	NaN	NaN
...	...	...	...	...	...	...
912995	27-12-2017	10	50	63.0	NaN	NaN
912996	28-12-2017	10	50	59.0	NaN	NaN
912997	29-12-2017	10	50	74.0	NaN	NaN
912998	30-12-2017	10	50	62.0	NaN	NaN
912999	31-12-2017	10	50	82.0	NaN	NaN

913000 rows × 6 columns

Data cleaning

Removing the unwanted columns in the dataset.

```
1 df.columns
```

```
Index(['date', 'store', 'item', 'sales', 'f0', 'f1'], dtype='object')
```

```
1 df.drop(['f0', 'f1'], axis=1, inplace=True)
```

```
1 df
```

	date	store	item	sales
0	01-01-2013	1	1	13.0
1	02-01-2013	1	1	11.0
2	03-01-2013	1	1	14.0
3	04-01-2013	1	1	13.0
4	05-01-2013	1	1	10.0
...	...	...	...	...
912995	27-12-2017	10	50	63.0
912996	28-12-2017	10	50	59.0
912997	29-12-2017	10	50	74.0
912998	30-12-2017	10	50	62.0
912999	31-12-2017	10	50	82.0

913000 rows × 4 columns

Checking basic details of dataset

Like shape, data type, describe

```
df.shape
```

```
(913000, 4)
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 913000 entries, 0 to 912999
Data columns (total 4 columns):
 #   Column  Non-Null Count   Dtype  
---  ------  --------------   -----  
 0   date    913000 non-null  object 
 1   store   913000 non-null  int64  
 2   item    913000 non-null  int64  
 3   sales   912943 non-null  float64
dtypes: float64(1), int64(2), object(1)
memory usage: 27.9+ MB
```

```
#change data type for date column
df['date'] = pd.to_datetime(df['date'], format='%d-%m-%Y')
```

```
1 df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 913000 entries, 0 to 912999
Data columns (total 4 columns):
 #   Column  Non-Null Count   Dtype         
---  ------  --------------   -----         
 0   date    913000 non-null  datetime64[ns]
 1   store   913000 non-null  int64         
 2   item    913000 non-null  int64         
 3   sales   912943 non-null  float64       
dtypes: datetime64[ns](1), float64(1), int64(2)
memory usage: 27.9 MB
```

```
1 df.describe()
```

	date	store	item	sales
count	913000	913000.000000	913000.000000	912943.000000
mean	2015-07-02 11:59:59.999999744	5.500000	25.500000	52.251626
min	2013-01-01 00:00:00	1.000000	1.000000	0.000000
25%	2014-04-02 00:00:00	3.000000	13.000000	30.000000
50%	2015-07-02 12:00:00	5.500000	25.500000	47.000000
75%	2016-10-01 00:00:00	8.000000	38.000000	70.000000
max	2017-12-31 00:00:00	10.000000	50.000000	231.000000
std	NaN	2.872283	14.430878	28.801046

Missing Values Imputation

Check for the missing values in the dataset.

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

	0
date	0
store	0
item	0
sales	57

dtype: int64

Filling the missing values

Using interpolation method to fill the missing values in the dataset.

```
df = df.set_index('date')
df = df.interpolate() # can add different methods to fill the missing values
```

```
df = df.reset_index(drop=False)
df
```

	date	store	item	sales
0	2013-01-01	1	1	13.0
1	2013-01-02	1	1	11.0
2	2013-01-03	1	1	14.0
3	2013-01-04	1	1	13.0
4	2013-01-05	1	1	10.0
...	...	...	...	...
912995	2017-12-27	10	50	63.0
912996	2017-12-28	10	50	59.0
912997	2017-12-29	10	50	74.0
912998	2017-12-30	10	50	62.0
912999	2017-12-31	10	50	82.0

913000 rows × 4 columns

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

	0
date	0
store	0
item	0
sales	0

dtype: int64

Data Preprocessing

For the purpose of this project, we will only look at the sales of 'item' - 1 from 'store' - 1. The dataset is split into train and test sets, where the train set contains sales record from January 2013 to September 2017 and the test set (validation set) contains sales records of the last three month of 2017.

Some new features have been created from the date field, for the purpose of exploratory data analysis and causal modelling.

```
# sort values based on date column
df.sort_values(by='date', ascending = True, inplace = True)
```

```
# Filter records for store 1 and item 1 -> to be able to scale to other items in the futu
df = df[df['store'] == 1]
df = df[df['item'] == 1]

df['date'] = pd.to_datetime(df['date'], format='%Y-%m-%d') # convert date column to datat

# Create Date-related Features to be used for EDA and Supervised ML: Regression
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['weekday'] = df['date'].dt.weekday
df['weekday'] = np.where(df.weekday == 0, 7, df.weekday)

# Split the series to predict the last 3 months of 2017
temp_df = df.set_index('date')
train_df = temp_df.loc[:'2017-09-30'].reset_index(drop=False)
test_df = temp_df.loc['2017-10-01':].reset_index(drop=False)

train_df.head()
```

	date	store	item	sales	year	month	day	weekday
0	2013-01-01	1	1	13.0	2013	1	1	1
1	2013-01-02	1	1	11.0	2013	1	2	2
2	2013-01-03	1	1	14.0	2013	1	3	3
3	2013-01-04	1	1	13.0	2013	1	4	4
4	2013-01-05	1	1	10.0	2013	1	5	5

Next steps: Generate code with train_df | View recommended plots | New interactive sheet

```
test_df.head()
```

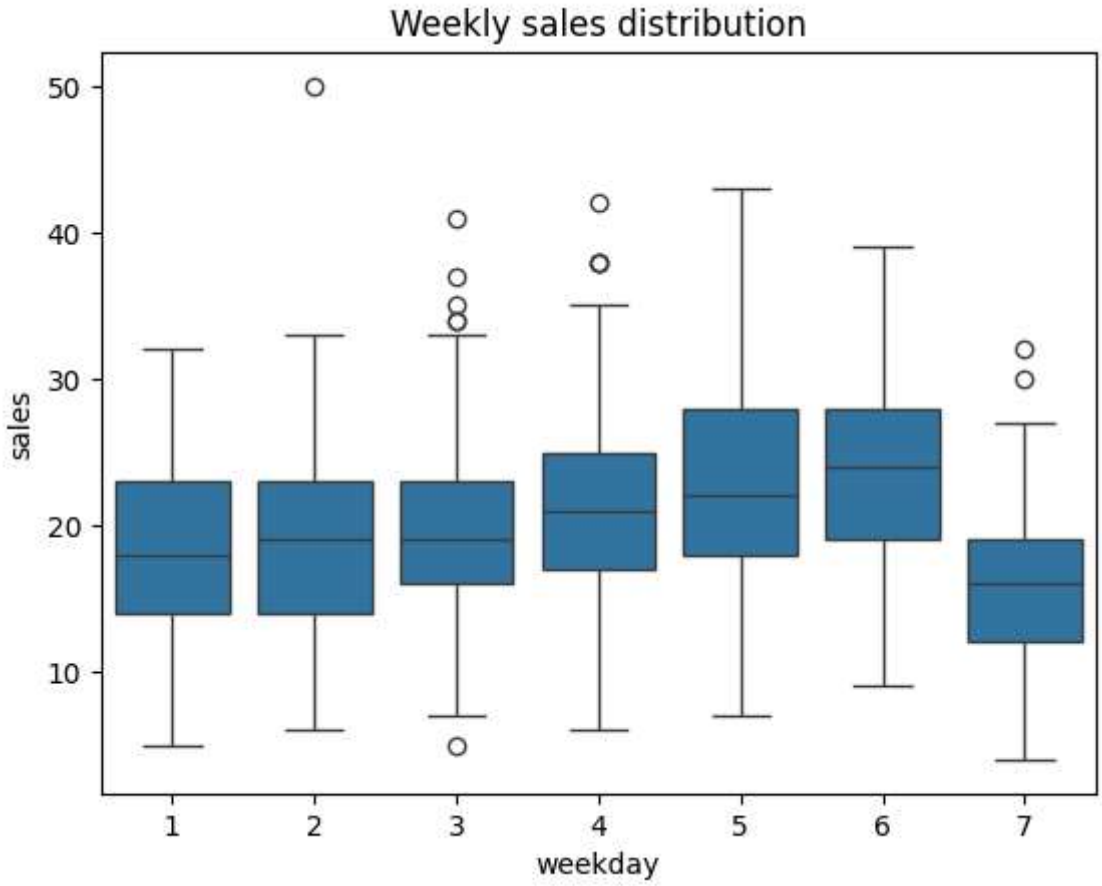
	date	store	item	sales	year	month	day	weekday
0	2017-10-01	1	1	21.0	2017	10	1	6
1	2017-10-02	1	1	12.0	2017	10	2	7
2	2017-10-03	1	1	18.0	2017	10	3	1
3	2017-10-04	1	1	15.0	2017	10	4	2
4	2017-10-05	1	1	20.0	2017	10	5	3

Next steps: Generate code with `test_df` | View recommended plots | New interactive sheet

Data Exploration

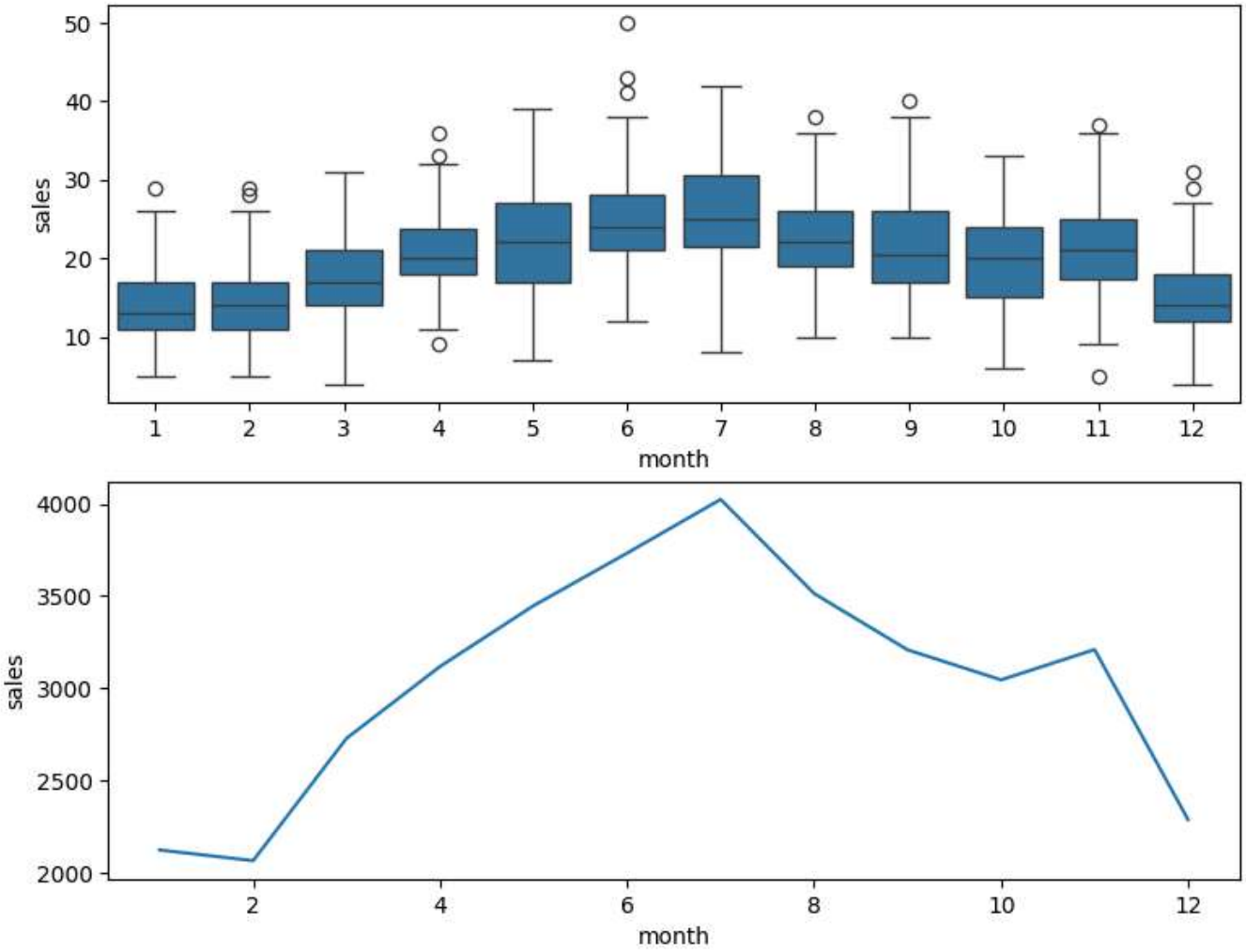
The plots below try to capture the trend and distribution of sales through weeks, months and years.

```
plot = sns.boxplot(x='weekday', y='sales', data=df)
_ = plot.set(title='Weekly sales distribution')
```



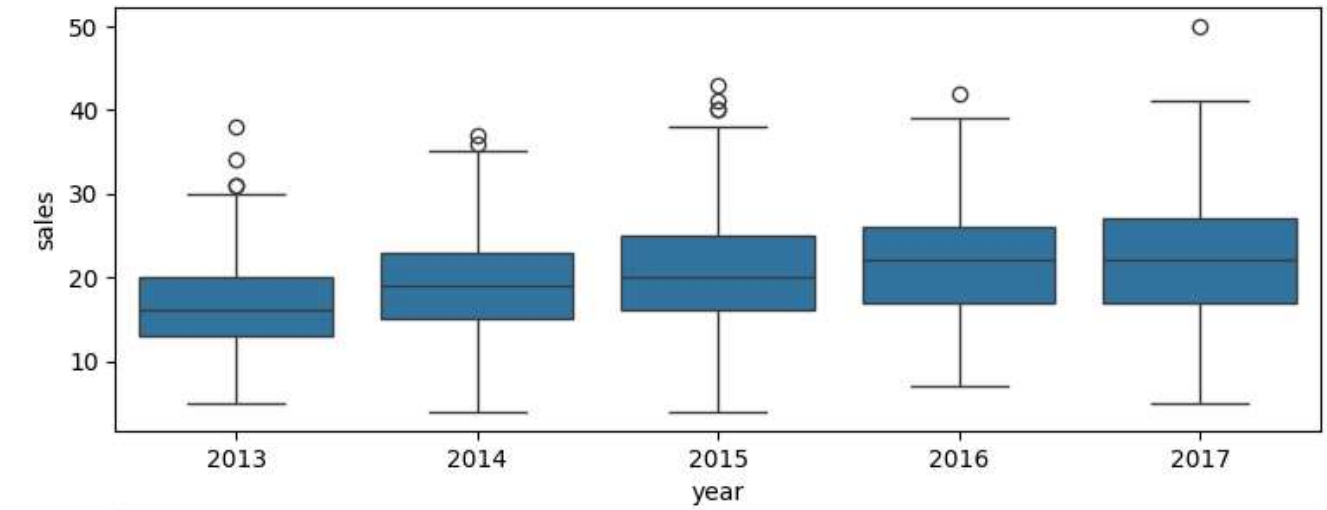
Inference: The average number of sales increases over the week, is maximum on Saturday (6), and takes a sharp fall on Sunday (7)

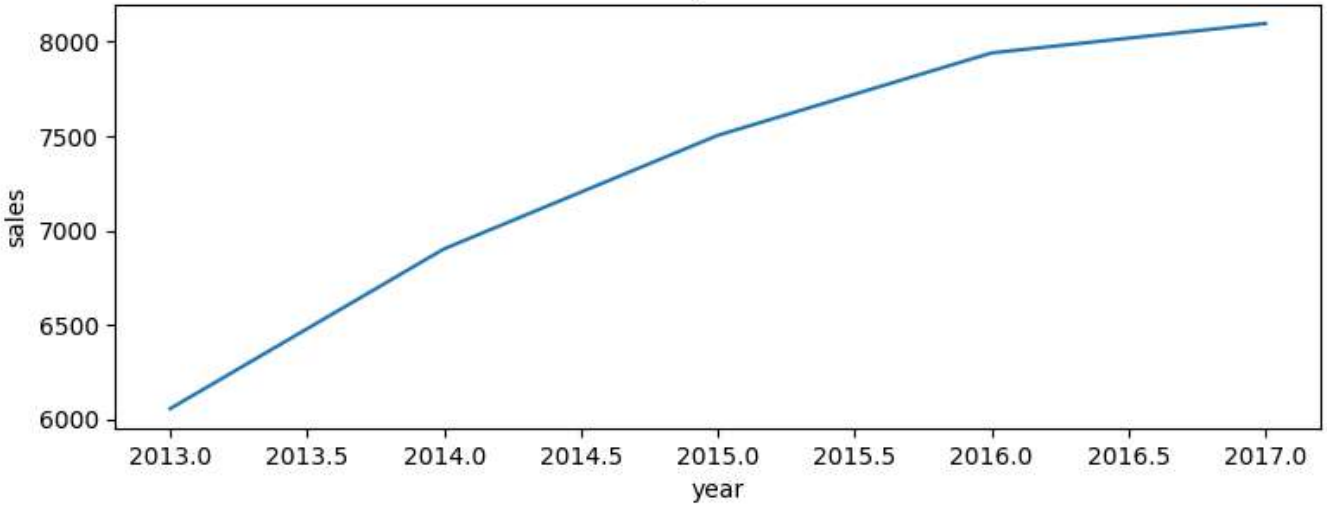
```
monthly_agg = df.groupby('month')['sales'].sum().reset_index()
fig, axs = plt.subplots(nrows=2, figsize=(9,7))
sns.boxplot(x='month', y='sales', data=df, ax=axs[0])
_ = sns.lineplot(x='month', y='sales', data=monthly_agg, ax=axs[1])
```



Inference: The number of sales gradually ascends in the first half of the year starting February (2), peaks in July (7), and then gradually descends, before slightly increasing in November (11) and then dropping again in December (12).

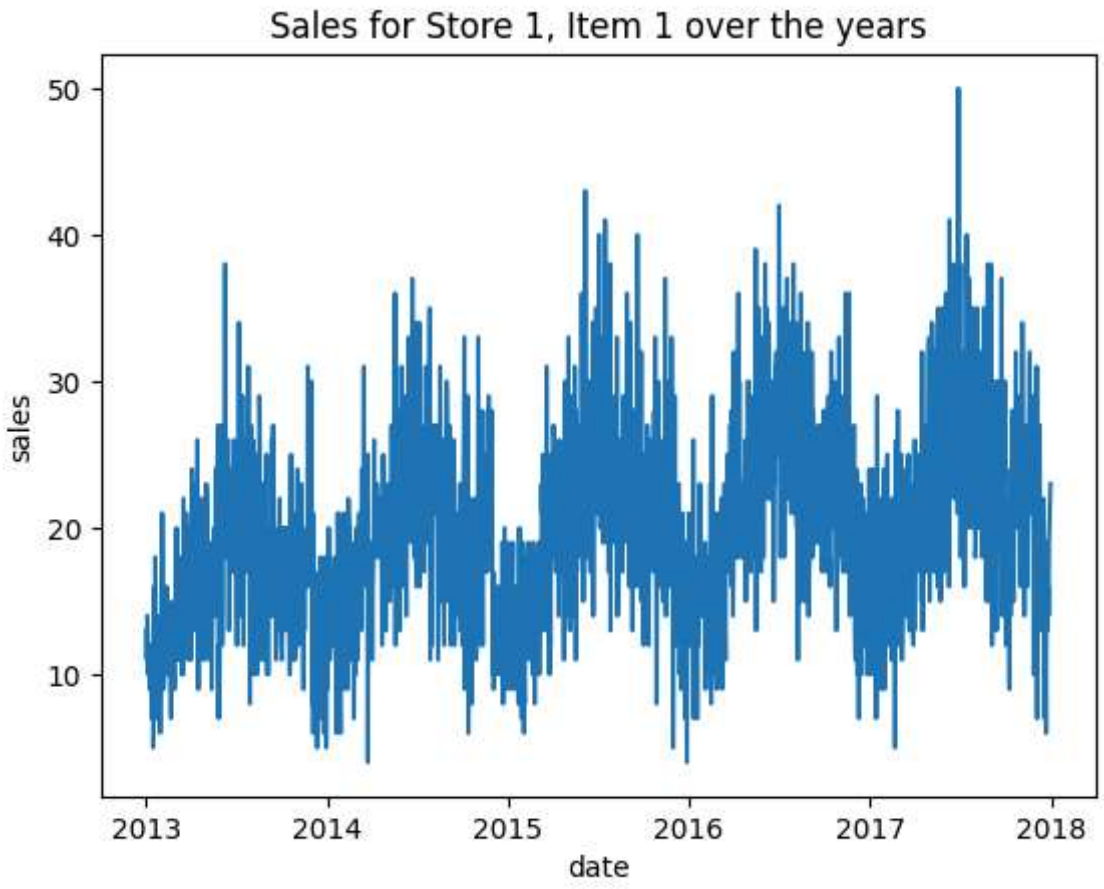
```
yearly_agg = df.groupby('year')['sales'].sum().reset_index()
fig, axs = plt.subplots(nrows=2, figsize=(9,7))
sns.boxplot(x='year', y='sales', data=df, ax=axs[0])
_ = sns.lineplot(x='year', y='sales', data=yearly_agg, ax=axs[1])
```





Inference: From the number of sales vs. year plot, we can infer an increasing trend over the years. The aggregate number of sales has increased from approximately 6000 in 2013 to slightly over 8000 in 2017, i.e. a **33.3% increase in the number of sales** approximately. A clear trend is captured by the lineplot above.

```
plot = sns.lineplot(x='date', y='sales', data=df)
_ = plot.set(title='Sales for Store 1, Item 1 over the years')
```



Inference: There is a seasonal pattern in the number of sales of 'item' - 1 at 'store' - 1. As also infered in the plot for sales vs. month above, we can see an increase in the sales in the first half of the year, peaking in July, and then a gradual decrease till December. This pattern is repeated each year, 2013 onwards.

Quantitative Methods to Forecast Product Sales

It can be fairly concluded from the data exploratory plots above that there is seasonality present in the product sales data, along with a general increase in the number of sales over the years.
Therefore, in order to forecast the number of sales for the last three months of 2017, we will keep in mind the linear trend and seasonality present in the product sales.
There are multiple ways to approach the forecasting problem, we can either build traditional time series models, or use causal models, such as linear regression. We will look at both these methods and try to evaluate our forecasts using the validation set.

1. Baseline Model: Seasonal Naive

Before we get to the more advanced time-series forecasting methods, let's take a look at a basic method - Seasonal Naive. It can serve as a quick calculation to get a baseline until something better can come along. Or, perhaps there is very little variance in the data, then this method can be good enough.

What is Seasonal Naive?

It is a naive method that takes the seasonal patterns into account by looking at what happened same time last year. For example, if we want to predict the sales during December 2017, the seasonal naive method will assume the same number of sales for December 2017 as was in December 2016. Fortunately, we have at least one-year of sales data, this method might make no sense otherwise.

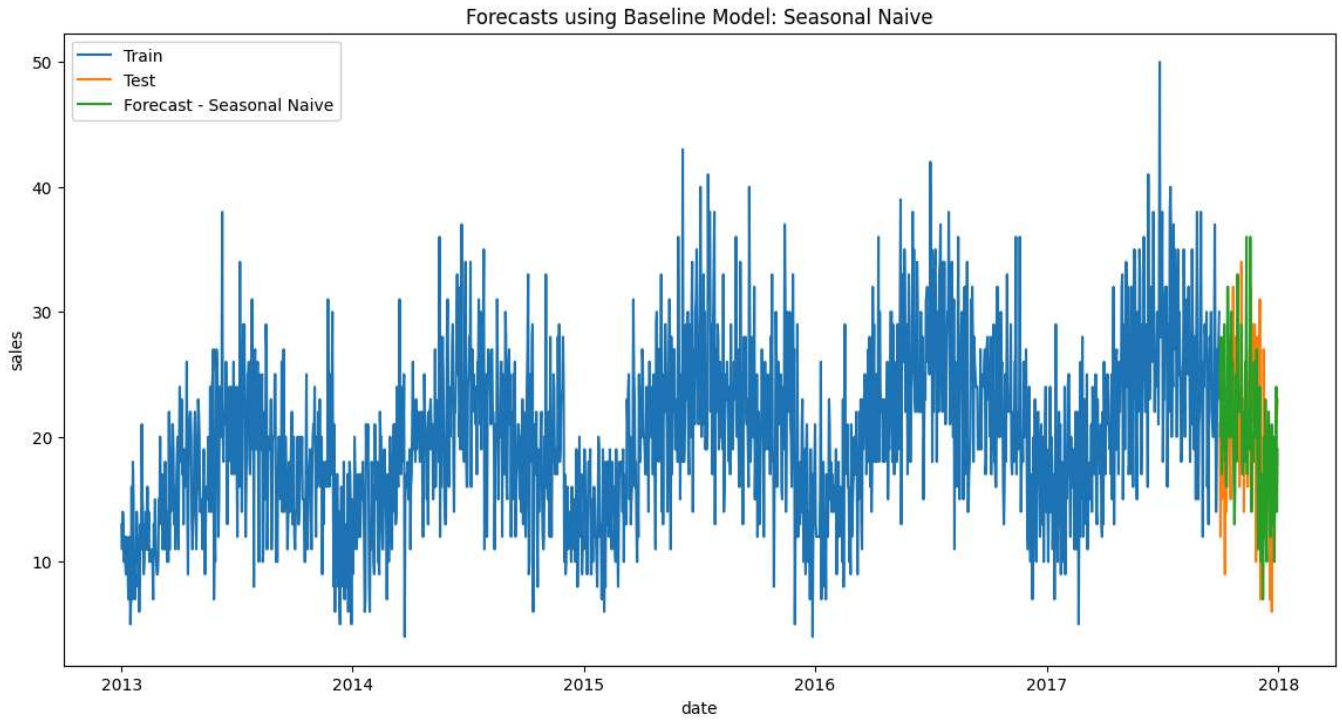
In the code below, one year is subtracted from the dates in the test data, and one day is added to the resulting difference, after which they are looked up in the training data to return the sales from those respective dates. So keeping the seasonality in mind, we now have our naive forecasts based on the number of sales from a year ago.

```
# Ensure 'date' columns are in datetime format
test_df['date'] = pd.to_datetime(test_df['date'])
train_df['date'] = pd.to_datetime(train_df['date'])

# Subtract 1 year from each date in the test_df and add 1 day
test_df['adjusted_date'] = test_df['date'] - pd.DateOffset(years=1) + pd.DateOffset(days=

# Find matching sales in the train_df for the adjusted dates
dates = test_df['adjusted_date']
seasonal_naive_sales = train_df[train_df['date'].isin(dates)]['sales']

# Make a copy of the test_df and add the seasonal naive sales predictions
sn_pred_df = test_df.copy().drop('sales', axis=1)
sn_pred_df['seasonal_naive_sales'] = pd.Series(seasonal_naive_sales.values, index=test_df
sn_pred_df.head()

```

	date	store	item	year	month	day	weekday	adjusted_date	seasonal_naive_sales
0	2017-10-01	1	1	2017	10	1	6	2016-10-02	26.0
1	2017-10-02	1	1	2017	10	2	7	2016-10-03	20.0
2	2017-10-03	1	1	2017	10	3	1	2016-10-04	28.0

Next steps: Generate code with sn_pred_df | View recommended plots | New interactive sheet

```
plt.figure(figsize=(14,7))
plt.plot(train_df['date'], train_df['sales'], label='Train')
plt.plot(test_df['date'], test_df['sales'], label='Test')
plt.plot(sn_pred_df['date'], sn_pred_df['seasonal_naive_sales'], label='Forecast - Seasor
plt.legend(loc='best')
plt.xlabel('date')
plt.ylabel('sales')
plt.title('Forecasts using Baseline Model: Seasonal Naive')
plt.show()
```



Inference: To the naked eye, the forecasts seem alright the decreasing trend is clearly captured by our naive method. However, we will formally quantify the performance using forecast accuracy metrics.

Evaluating the Forecasts

There are number of widely accepted forecasting metrics, and the most common metric is **forecast error**. The error can simply be calculated by finding the diffrence between the actual sales value and the forecasted sales value. For example, if 10 items are sold on a day, and you predicted 14 items to be sold, you have an error of -4.

In order to evaluate the overall forecast, we will look at metrics like **mean absolute error** (MAE), **root mean squared error** (RMSE) and a percentage error metric - **mean absolute percentage error** (MAPE).

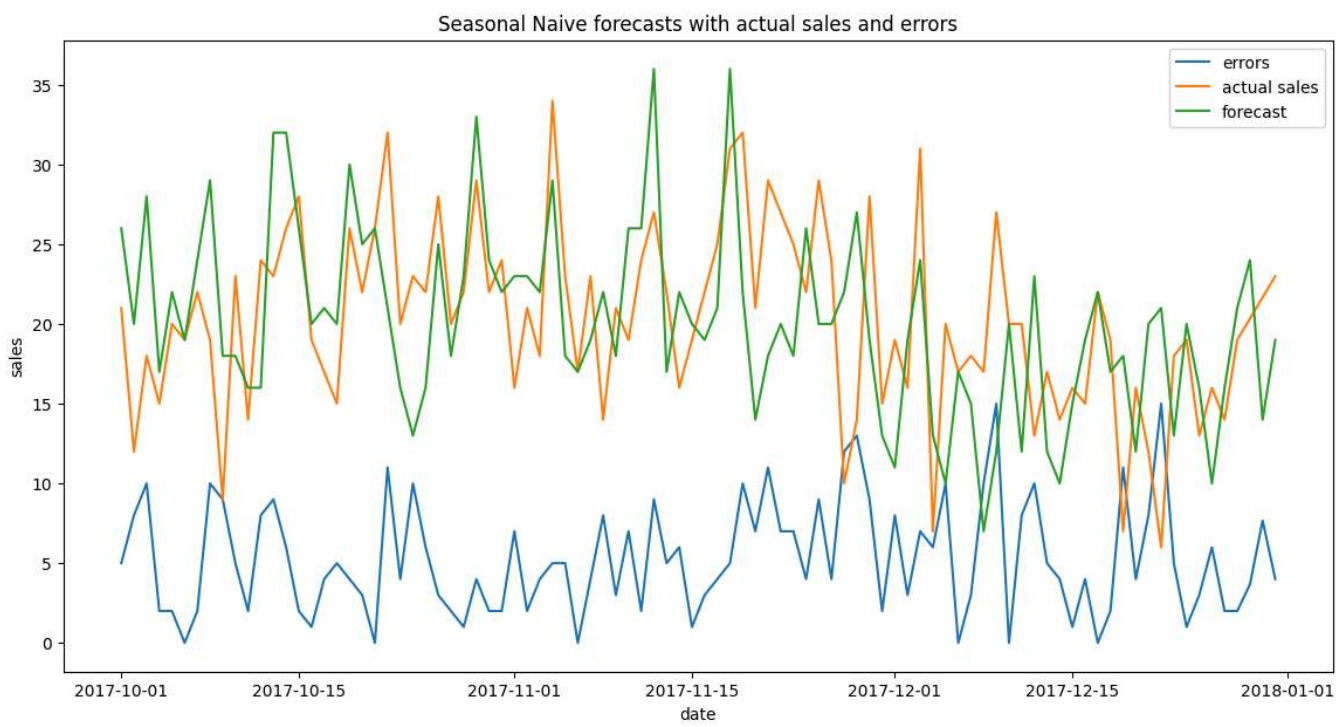
- **MAE:** It is the absolute value of the error, summed for each observed day, and divided by the total number of observed days.
- **RMSE:** It is similar to MAE, however rather than the absolute value, the error is squared before it is summed, and then the square root is taken to normalize the units. Because we take the square of the error, we wind up punishing large forecasting errors more than small ones.

The above two metrics are scale-dependent meaning that we cannot use them to compare forecasts on different scales. MAPE on the other hand is a scale-independent metric.

- **MAPE:** It can be used for meaningful comparisions, it is calculated by dividing the sum of absolute errors by sum of actual sales to get a percentage indicator that is scaled to the actual sales.

Let's apply the above metrics to our forecasts and see how they compare.

```
errors_df = pd.merge(test_df, sn_pred_df, on='date')
errors_df = errors_df[['date', 'sales', 'seasonal_naive_sales']]
errors_df = pd.merge(test_df, sn_pred_df, on='date')
errors_df = errors_df[['date', 'sales', 'seasonal_naive_sales']]
errors_df['errors'] = test_df['sales'] - sn_pred_df['seasonal_naive_sales']
errors_df.insert(0, 'model', 'Seasonal Naive')

def mae(err):
    return np.mean(np.abs(err))

def rmse(err):
    return np.sqrt(np.mean(err ** 2))

def mape(err, sales=errors_df['sales']):
    return np.sum(np.abs(err))/np.sum(sales) * 100

result_df = errors_df.groupby('model').agg(total_sales=('sales', 'sum'),
                                           total_sn_pred_sales=('seasonal_naive_sales', '
                                           overall_error=('errors', 'sum'),
                                           MAE=('errors', mae),
                                           RMSE=('errors', rmse),
                                           MAPE=('errors', mape))


plt.figure(figsize=(14,7))
plt.plot(errors_df['date'], np.abs(errors_df['errors']), label='errors')
plt.plot(errors_df['date'], errors_df['sales'], label='actual sales')
plt.plot(errors_df['date'], errors_df['seasonal_naive_sales'], label='forecast')
plt.legend(loc='best')
plt.xlabel('date')
plt.ylabel('sales')
plt.title('Seasonal Naive forecasts with actual sales and errors')
plt.show()

result_df
```



model	total_sales	total_sn_pred_sales	overall_error	MAE	RMSE	MAPE
Seasonal Naive	1861.0	1851.0	10.0	5.275362	6.357315	26.079169

The overall error is not as bad, and we were able to achieve a **MAPE of 26.07%**. We will use this as a **benchmark** to judge the forecast performance of the other models. Let's move on to a more advanced model that uses Exponential Smoothing, but before that let's take a look at the time series decomposition plot for our training data.

Time Series Decomposition Plot

A time series decomposition plot allows us to observe the seasonality, trend, and error/remainder terms of a time series. These three components are a crucial part of the Exponential Smoothing models. Therfore, the decomposition plot helps in deciding the type of Exponential Smoothing Model to use for our forecasts.

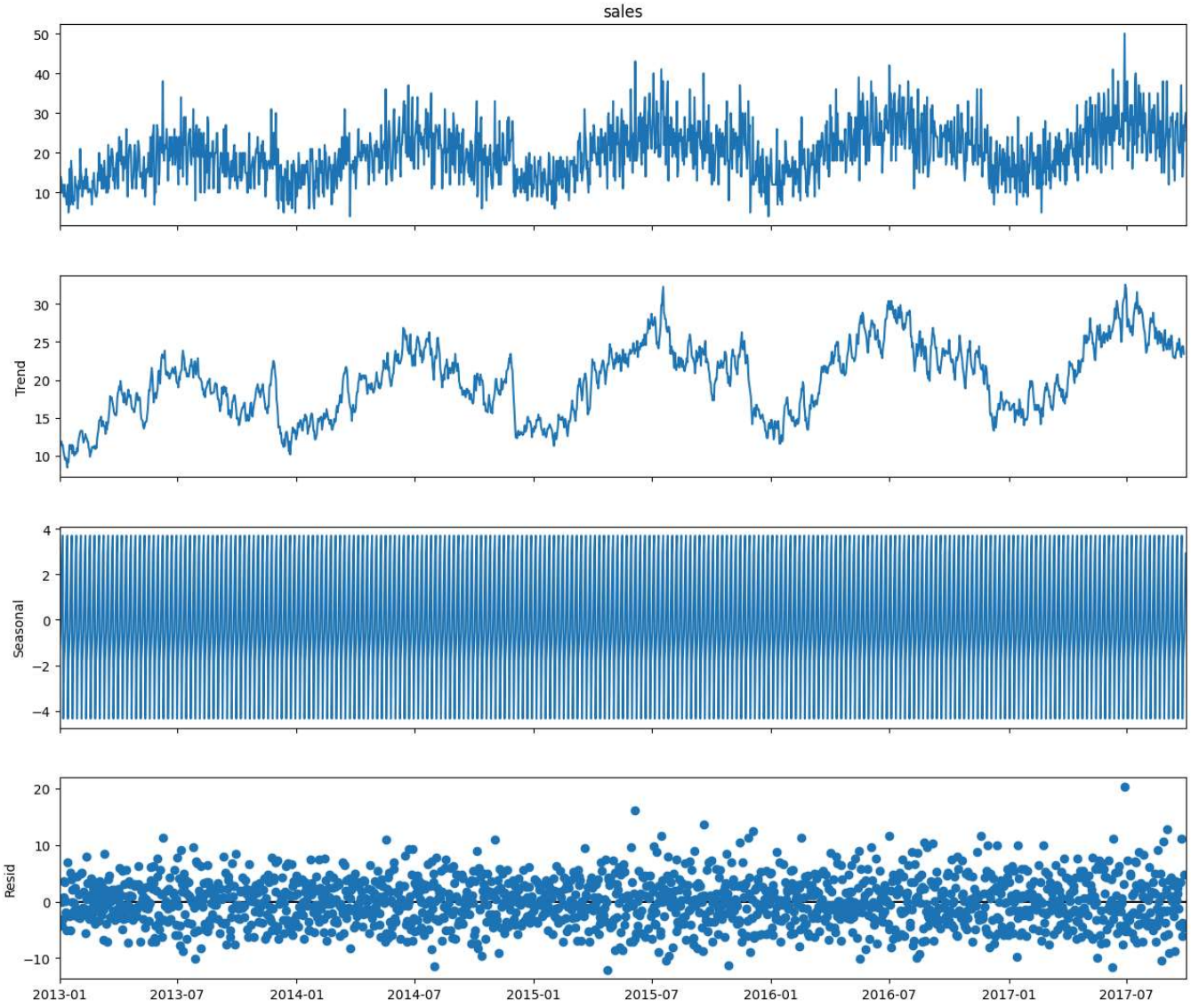
```
ts_decomp_df = train_df.set_index('date') # set date as index
ts_decomp_df['sales'] = ts_decomp_df['sales'].astype(float)
ts_decomp_df.head()
```

date	store	item	sales	year	month	day	weekday
2013-01-01	1	1	13.0	2013	1	1	1
2013-01-02	1	1	11.0	2013	1	2	2
2013-01-03	1	1	14.0	2013	1	3	3
2013-01-04	1	1	13.0	2013	1	4	4
2013-01-05	1	1	10.0	2013	1	5	5

Next steps: Generate code with ts_decomp_df | View recommended plots | New interactive sheet

```
# Infer the frequency of the data
result = seasonal_decompose(ts_decomp_df['sales'], model='additive')
fig = plt.figure()
fig = result.plot()
fig.set_size_inches(14, 12)
```

```
<Figure size 640x480 with 0 Axes>
```



Determine Error, Trend and Seasonality

An ETS model has three main components: error, trend, and seasonality. Each can be applied either additively, multiplicatively, or not at all. We will use the above Times Series Decomposition Plot to determine the additive or multiplicative property of the thre components.

1. Trend - If the trend plot is linear then we apply it additively (A). If the trend line grows or shrinks exponentially, we apply it multiplicatively (M). If there is no clear trend, no trend component is included (N).
2. Seasonal - If the peaks and valleys for seasonality are constant over time, we apply it additively (A). If the size of the seasonal fluctuations tends to increase or decrease with the level of time series, we apply it multiplicatively (M). If there is no seasonality, it is not applied (N).
3. Error - If the error plot has constant variance over time (peaks and valleys are about the same size), we apply it additively (A). If the error plot is fluctuating between large and small errors over time, we apply it multiplicatively (M).

For our sales data, we see a linear trend plot and a constant seasonality over time, so we will apply trend and seasonality additively. The error component also has constant variance, so we will apply it additively too.

We will use a Exponential Smoothing, a commonly-used local statistical algorithm for time-series forecasting. The Exponential Smoothing method can be defined in terms of an ETS framework, in which the components are calculated using a smoothing technique.

What is Exponential Smoothing?

This is a very popular scheme to produce a smoothed Time Series. Whereas in Single Moving Averages the past observations are weighted equally, **Exponential Smoothing assigns exponentially decreasing weights as the observation get older**. In other words, recent observations are given relatively more weight in forecasting than the older observations.
In the case of moving averages, the weights assigned to the observations are the same and are equal to 1/N. In exponential smoothing, however, there are one or more smoothing parameters to be determined (or estimated) and these choices determine the weights assigned to the observations.

There are 3 kinds of smoothing techniques Single, Double and Triple Exponential Smoothing.

1. **Single Exponential Smoothing** is used when the time series does not have a trend line and a seasonality component.
2. **Double Exponential Smoothing** is used to include forecasting data with a trend, smoothing calculation includes one for the level, and one for the trend.
3. **Triple Exponential smoothing** is used when data has trend and seasonality. We include a third equation to take care of seasonality (sometimes called periodicity). The resulting set of equations is called the "Holt-Winters" (HW) method after the names of the inventors.

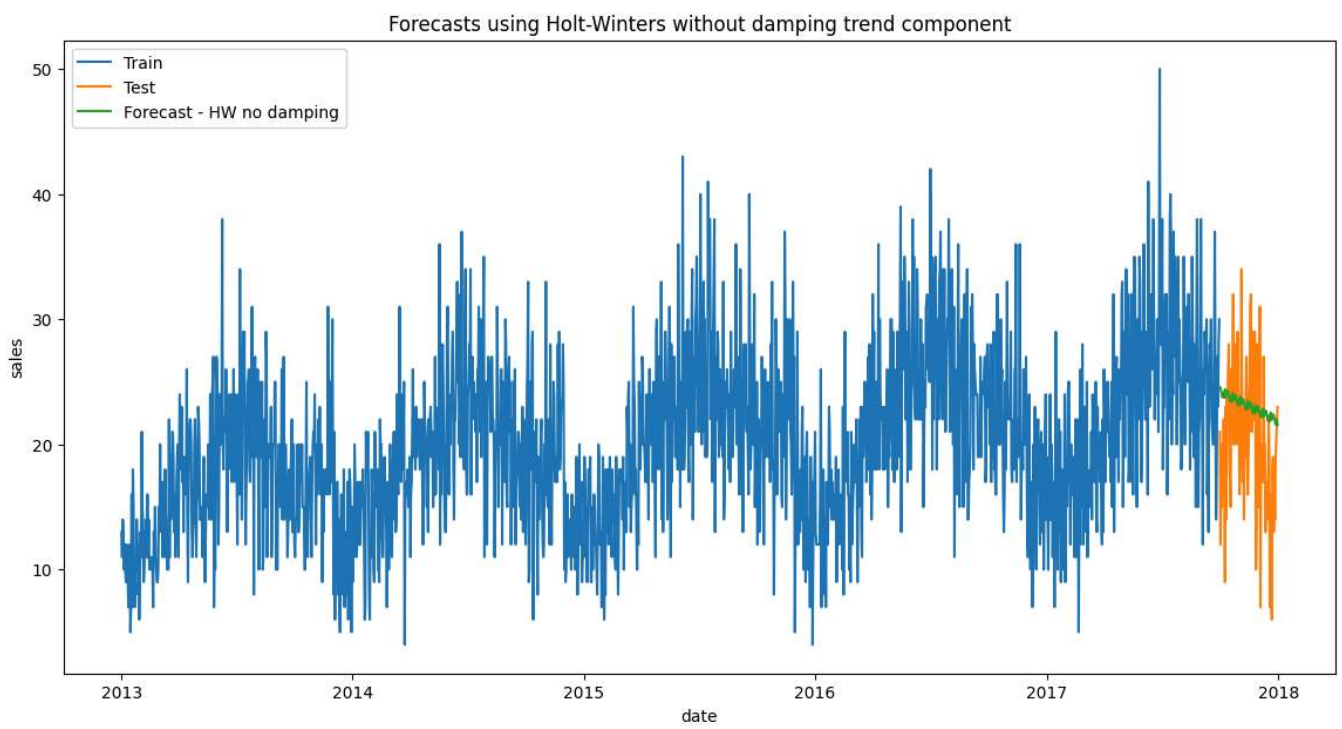
Since out data has both trend and seasonality components, we will apply Triple Exponential Smoothing.

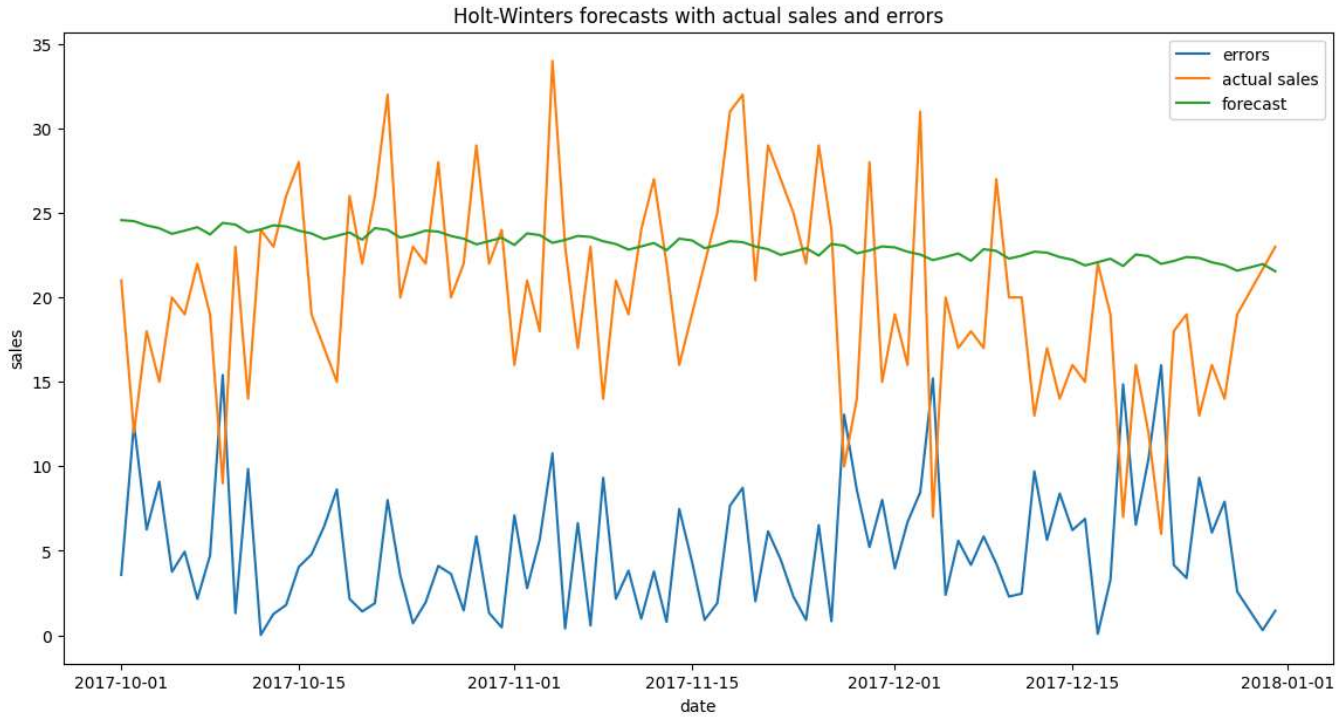
2. Holt Winter's Triple Exponential Smoothing Model

```
hw_train_df = train_df[['date', 'sales']].set_index('date')
hw_test_df = test_df[['date', 'sales']].set_index('date')

# Apply Triple Exponential Smoothing

hw_model_1 = ExponentialSmoothing(hw_train_df, trend='add', seasonal='add', seasonal_peri
hw_fit_1 = hw_model_1.fit(remove_bias=False)
pred_fit_1 = pd.Series(hw_fit_1.predict(start=hw_test_df.index[0], end=hw_test_df.index[-
                       name='pred_sales').reset_index()

hw_model_2 = ExponentialSmoothing(hw_train_df, trend='add', seasonal='add', seasonal_peri
hw_fit_2 = hw_model_2.fit(remove_bias=False)
pred_fit_2 = pd.Series(hw_fit_2.predict(start=hw_test_df.index[0], end=hw_test_df.index[-
                       name='pred_sales').reset_index()
print('Forecasts made, ready for evaluation')
```

```
Forecasts made, ready for evaluation
```

```
# Merge predictions and actual sales into one df
errors_df_hw = pd.merge(test_df, pred_fit_1, left_on='date', right_on='index')
errors_df_hw = errors_df_hw[['date', 'sales', 'pred_sales']]
errors_df_hw['errors'] = errors_df_hw.sales - errors_df_hw.pred_sales
errors_df_hw.insert(0, 'model', 'Holt-Winters')


# Evaluate the predictions for Holt-Winters without damping trend component
plt.figure(figsize=(14,7))
plt.plot(train_df['date'], train_df['sales'], label='Train')
plt.plot(test_df['date'], test_df['sales'], label='Test')
plt.plot(errors_df_hw['date'], errors_df_hw['pred_sales'], label='Forecast - HW no dampir
plt.legend(loc='best')
plt.xlabel('date')
plt.ylabel('sales')
plt.title('Forecasts using Holt-Winters without damping trend component')
plt.show()


plt.figure(figsize=(14,7))
plt.plot(errors_df_hw['date'], np.abs(errors_df_hw['errors']), label='errors')
plt.plot(errors_df_hw['date'], errors_df_hw['sales'], label='actual sales')
plt.plot(errors_df_hw['date'], errors_df_hw['pred_sales'], label='forecast')
plt.legend(loc='best')
plt.xlabel('date')
plt.ylabel('sales')
plt.title('Holt-Winters forecasts with actual sales and errors')
plt.show()

result_df_hw = errors_df_hw.groupby('model').agg(total_sales=('sales', 'sum'),
                                          total_pred_sales=('pred_sales', 'sum'),
                                          holt_winters_overall_error=('errors', 'sum'),
                                          MAE=('errors', mae),
                                          RMSE=('errors', rmse),
                                          MAPE=('errors', mape))
result_df_hw
```

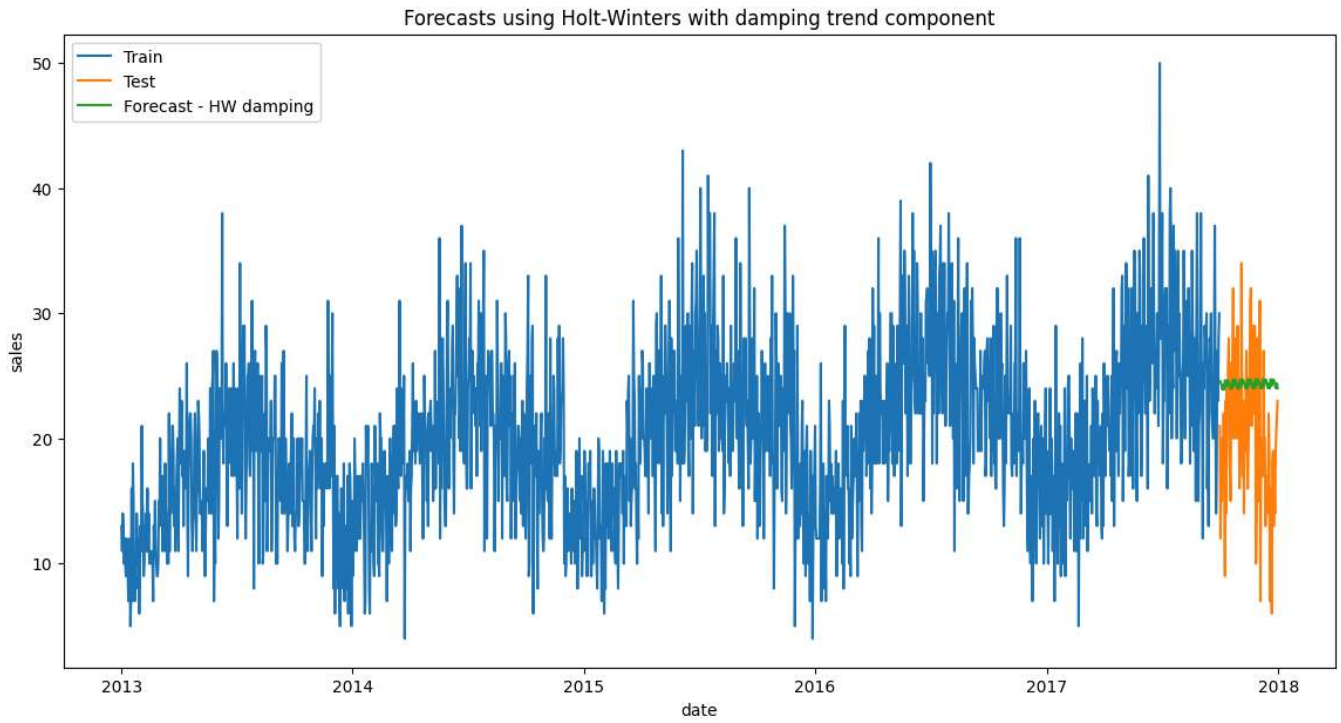


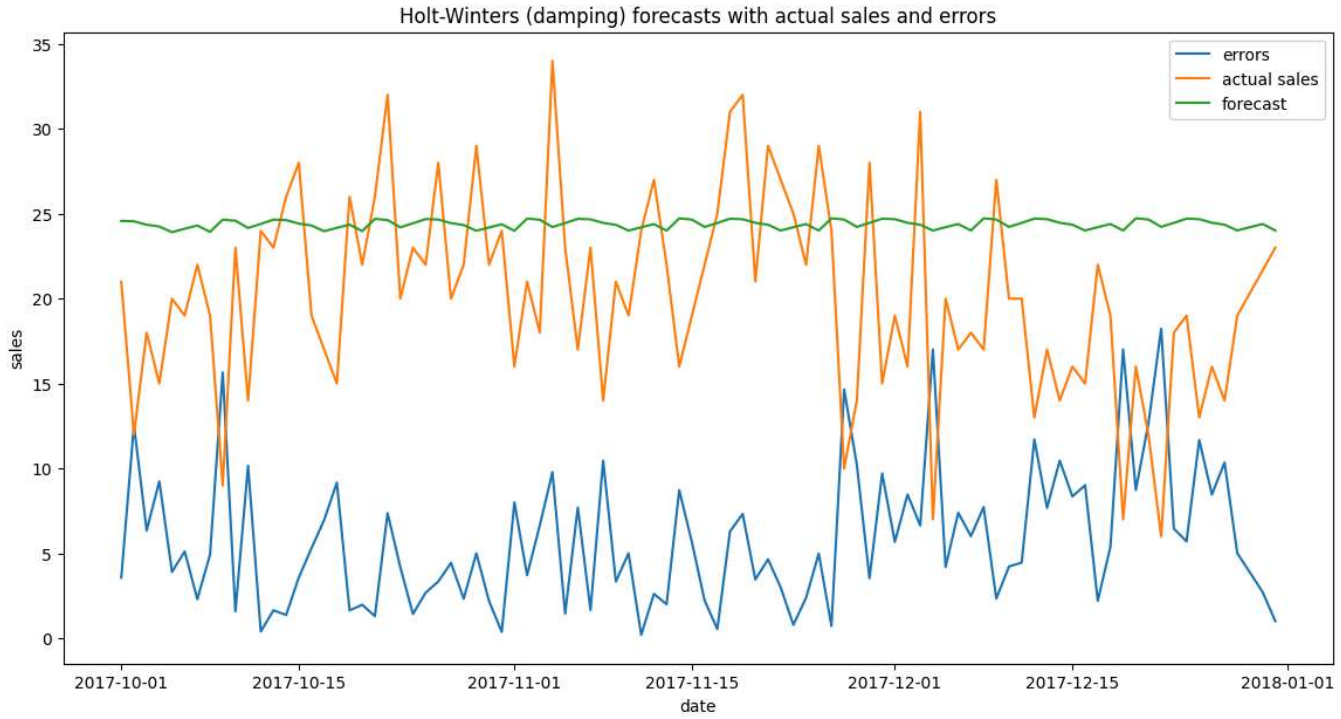


model	total_sales	total_pred_sales	holt_winters_overall_error	MAE	RMSE	
Holt-Winters	1861.0	2124.534915	-263.534915	5.078909	6.322973	2

Inference: The decreasing trend is clearly captured by the Holt-Winters method, and the **MAPE 25.1%** is better in comparision to our baseline model. Let's try Holt-Winters method with a damped parameter, and see if we can improve the results

```
# Merge predictions and actual sales into one df
errors_df_hwd = pd.merge(test_df, pred_fit_2, left_on='date', right_on='index')
errors_df_hwd = errors_df_hwd[['date', 'sales','pred_sales']]
errors_df_hwd['errors'] = errors_df_hwd.sales - errors_df_hwd.pred_sales
errors_df_hwd.insert(0, 'model', 'Holt-Winters-Damped')


# Evaluate the predictions for Holt-Winters without damping trend component
plt.figure(figsize=(14,7))
plt.plot(train_df['date'], train_df['sales'], label='Train')
plt.plot(test_df['date'], test_df['sales'], label='Test')
plt.plot(errors_df_hwd['date'], errors_df_hwd['pred_sales'], label='Forecast - HW damping
plt.legend(loc='best')
plt.xlabel('date')
plt.ylabel('sales')
plt.title('Forecasts using Holt-Winters with damping trend component')
plt.show()

plt.figure(figsize=(14,7))
plt.plot(errors_df_hwd['date'], np.abs(errors_df_hwd['errors']), label='errors')
plt.plot(errors_df_hwd['date'], errors_df_hwd['sales'], label='actual sales')
plt.plot(errors_df_hwd['date'], errors_df_hwd['pred_sales'], label='forecast')
plt.legend(loc='best')
plt.xlabel('date')
plt.ylabel('sales')
plt.title('Holt-Winters (damping) forecasts with actual sales and errors')
plt.show()

result_df_hwd = errors_df_hwd.groupby('model').agg(total_sales=('sales', 'sum'),
                                        total_pred_sales=('pred_sales', 'sum'),
                                        holt_winters_overall_error=('errors', 'sum'),
                                        MAE=('errors', mae),
                                        RMSE=('errors', rmse),
                                        MAPE=('errors', mape))
result_df_hwd
```





model	total_sales	total_pred_sales	holt_winters_overall_error	MAE	RMSE	
Holt-Winters-Damped	1861.0	2243.245504	-382.245504	5.81028	7.096692	2

Inference: To the naked eye, the forecasts seem alright, however the **MAPE 28.7%** is worse than our baseline model.

Let's look at another popular time-series method: Autoregressive Integrated Moving Average (ARIMA) Model.

3. Autoregressive Integrated Moving Average - ARIMA Model

Trend Elements There are three trend elements that require configuration.

They are the same as the ARIMA model; specifically:

- p: Trend autoregression order.
- d: Trend difference order.
- q: Trend moving average order.

Note: ARIMA model doesn't have the seasonal element (it's in SARIMA)

Step 1: Check stationarity

Before going any further into our analysis, our series has to be made stationary.

Stationarity is the property of exhibiting constant statistical properties (mean, variance, autocorrelation, etc.). If the mean of a time-series increases over time, then it’s not stationary.

The mean across many time periods is only informative if the expected value is the same across those time periods. If these population parameters can vary, what are we really estimating by taking an average across time?

Stationarity requires that the statistical properties must be the same across time, making the sample average a reasonable way to estimate them.

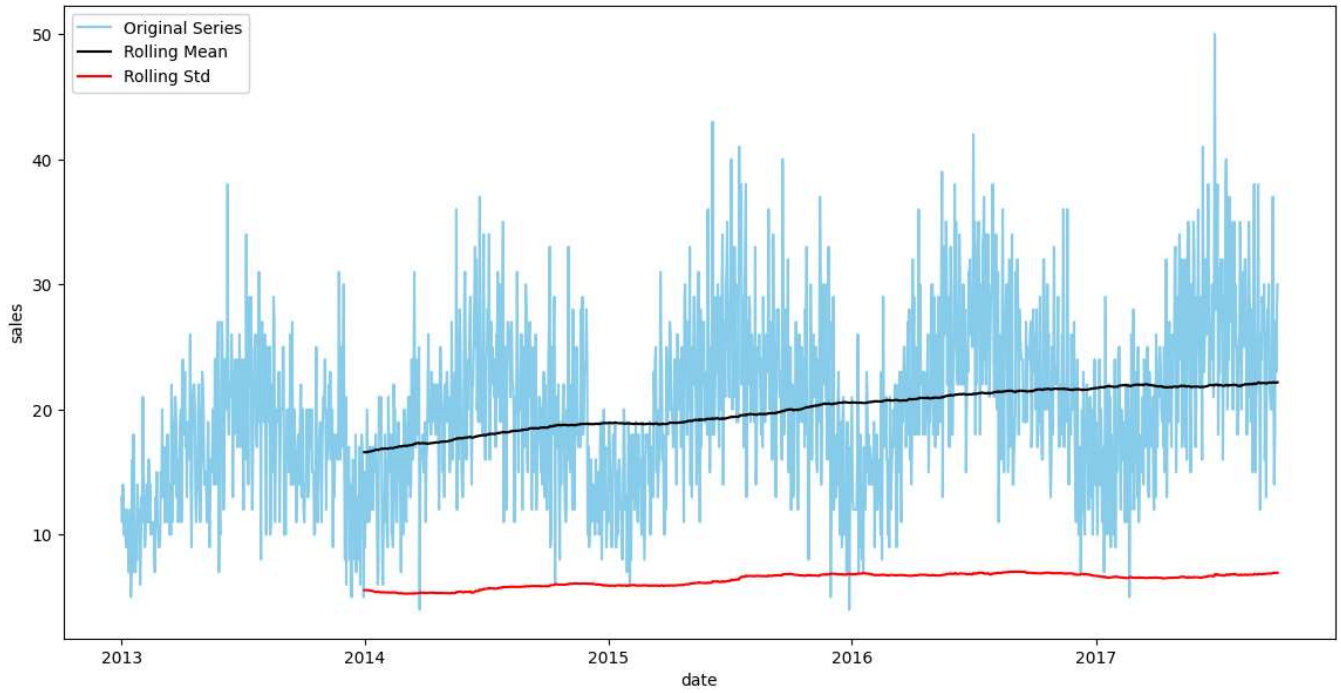
Methods to Check Stationarity

1. **Plotting rolling statistics:** Plotting rolling means and variances is a first good way to visually inspect our series. If the rolling statistics exhibit a clear trend (upwards or downwards) and show varying variance (increasing or decreasing amplitude), then you might conclude that the series is very likely not to be stationary.

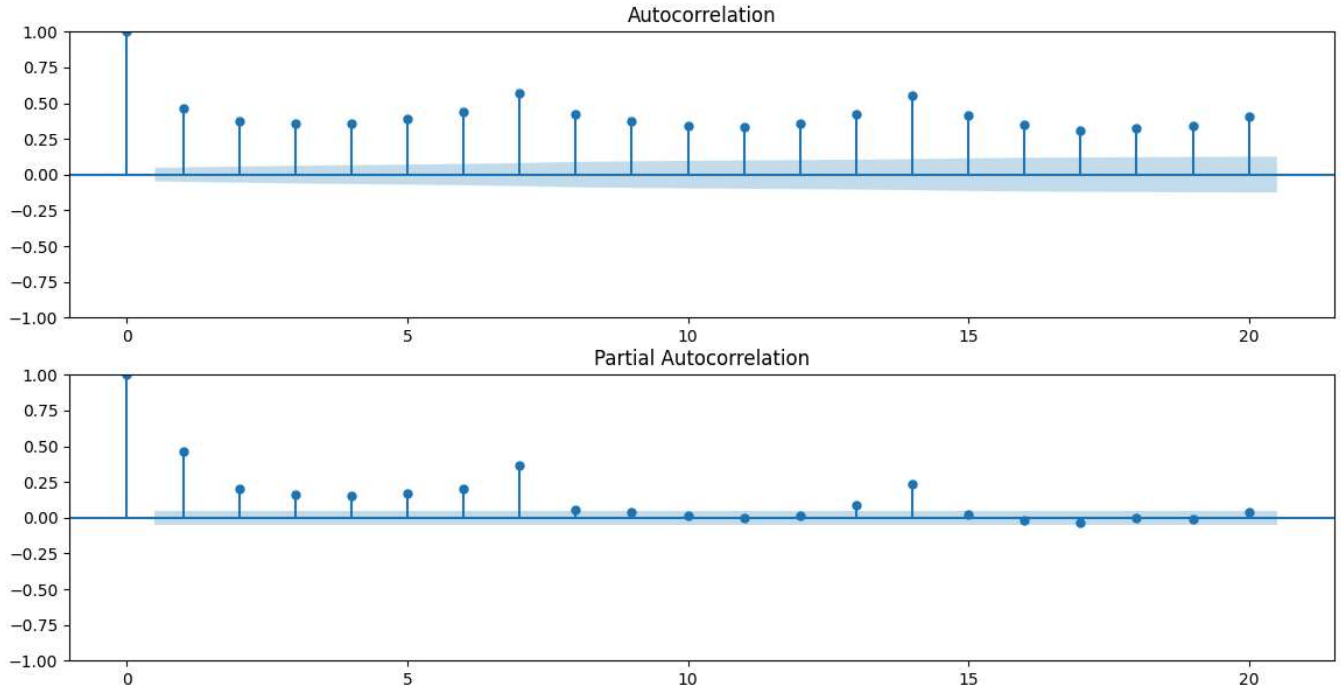
2. **Augmented Dickey-Fuller Test:** This test is used to assess whether or not a time-series is stationary. It gives a result called a “test-statistic”, based on which you can say, with different levels (or percentage) of confidence, if the time-series is stationary or not. The **test statistic** is expected to be negative; therefore, it has to be more **negative(less)** than the **critical value** for the **hypothesis to be rejected** and conclude that **series is stationary**.

3. **ACF and PACF plots:** An autocorrelation (ACF) plot represents the autocorrelation of the series with lags of itself. A partial autocorrelation (PACF) plot represents the amount of correlation between a series and a lag of itself that is not explained by correlations at all lower-order lags. Ideally, we want no correlation between the series and lags of itself. Graphically speaking, we would like all the spikes to fall in the blue region.

```
arima_df = train_df[['date', 'sales']].set_index('date')
arima_test_df = test_df[['date', 'sales']].set_index('date')

def test_stationarity(timeseries):
    # Plotting rolling statistics
    rollmean = timeseries.rolling(window=365).mean()
    rollstd = timeseries.rolling(window=365).std()

    plt.figure(figsize=(14,7))
    plt.plot(timeseries, color='skyblue', label='Original Series')
    plt.plot(rollmean, color='black', label='Rolling Mean')
    plt.plot(rollstd, color='red', label='Rolling Std')
    plt.legend(loc='best')
    plt.xlabel('date')
    plt.ylabel('sales')
    plt.show()

    # Augmented Dickey-Fuller Test
    adfuller_test = adfuller(timeseries, autolag='AIC')
    print("Test statistic = {:.3f}".format(adfuller_test[0]))
    print("P-value = {:.3f}".format(adfuller_test[1]))
    print("Critical values :")

    for key, value in adfuller_test[4].items():
        print("\t{}: {} - The data is {} stationary with {}% confidence"
              .format(key, value, '' if adfuller_test[0] < value else 'not', 100-int(key[

    # Autocorrelation Plots
    fig, ax = plt.subplots(2, figsize=(14,7))
    ax[0] = plot_acf(timeseries, ax=ax[0], lags=20)
    ax[1] = plot_pacf(timeseries, ax=ax[1], lags=20)

test_stationarity(arima_df.sales)
```




```
Test statistic = -3.066
P-value = 0.029
Critical values :
        1%: -3.4341843999399573 - The data is not stationary with 99% confidence
        5%: -2.8632336725104834 - The data is  stationary with 95% confidence
        10%: -2.567671665464627 - The data is  stationary with 90% confidence
```

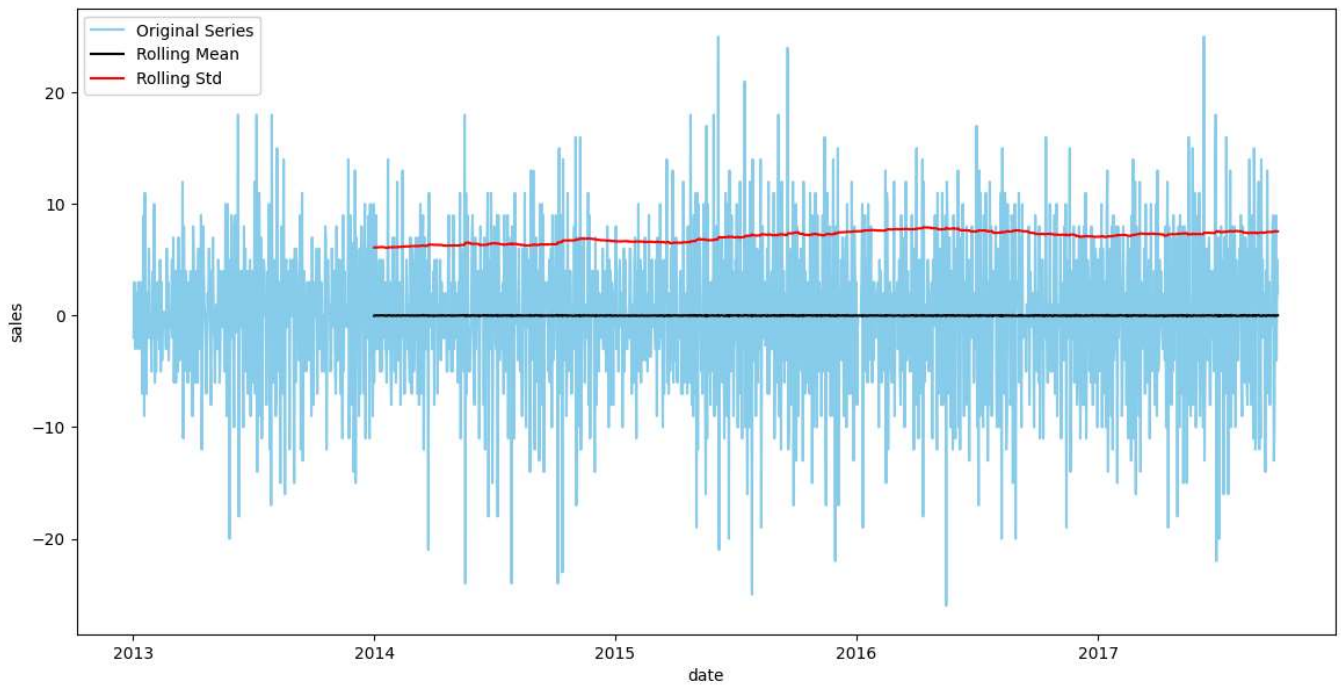


Looking at the results from our test, we can conclude that the series is not stationary. Therefore, in order to make the series stationary we apply **Differencing**

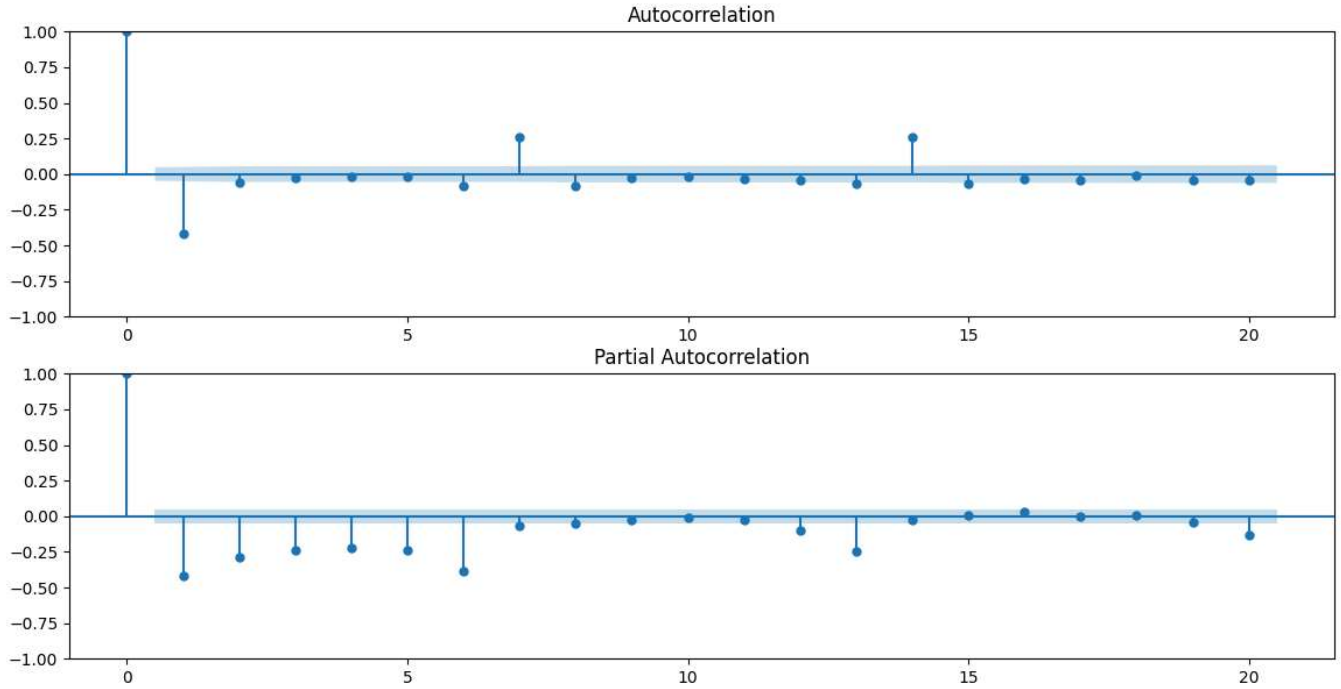
Step 2: Differencing

Differencing: Seasonal or cyclical patterns can be removed by substracting periodical values. If the data is 12-month seasonal, substracting the series with a 12-lag difference series will give a “flatter” series. Since we have aggregated the data to each day-level, we will shift by 1.

```
first_difference = arima_df.sales - arima_df.sales.shift(1)
first_difference = pd.DataFrame(first_difference.dropna(inplace=False))
# Check for stationarity after differencing
test_stationarity(first_difference.sales)
```




```
Test statistic = -12.305
P-value = 0.000
Critical values :
        1%: -3.434179908502461 - The data is  stationary with 99% confidence
        5%: -2.863231689892687 - The data is  stationary with 95% confidence
        10%: -2.567670609760268 - The data is  stationary with 90% confidence
```



After applying Differencing to the series, we can see from the above results that the series is now stationary, i.e. mean and variance are constant over time, and from ADF we can verify that the test-statistic is lesser than the critical value, hence we can reject the null hypothesis and conclude that the series is staionary.

Step 3: Model Building

Interpreting the AR(p), I(d), MA(q) values:

1. **Determining I(d):**

   Taking the first order difference makes the time series stationary. Therefore, **I(d) = 1.**

2. **Determining AR(p):** If the lag-1 autocorrelation of the differenced series PACF is negative, and/or there is a sharp cutoff, then choose a AR order of 1.

   From the PACF plot we can clearly observe that within 6 lags the AR is significant. Therefore, we can use **AR(p) = 6**, (6 lines are crossed the blue lines so 6past days are required to predict).

3. **Determining MA(q):** If the lag-1 autocorrelation of the differenced series ACF is negative, and/or there is a sharp cutoff, then choose a MA order of 1.

   From tha ACF plot we see a negative spike at lag 1, therfore we can use **MA(q) = 1**

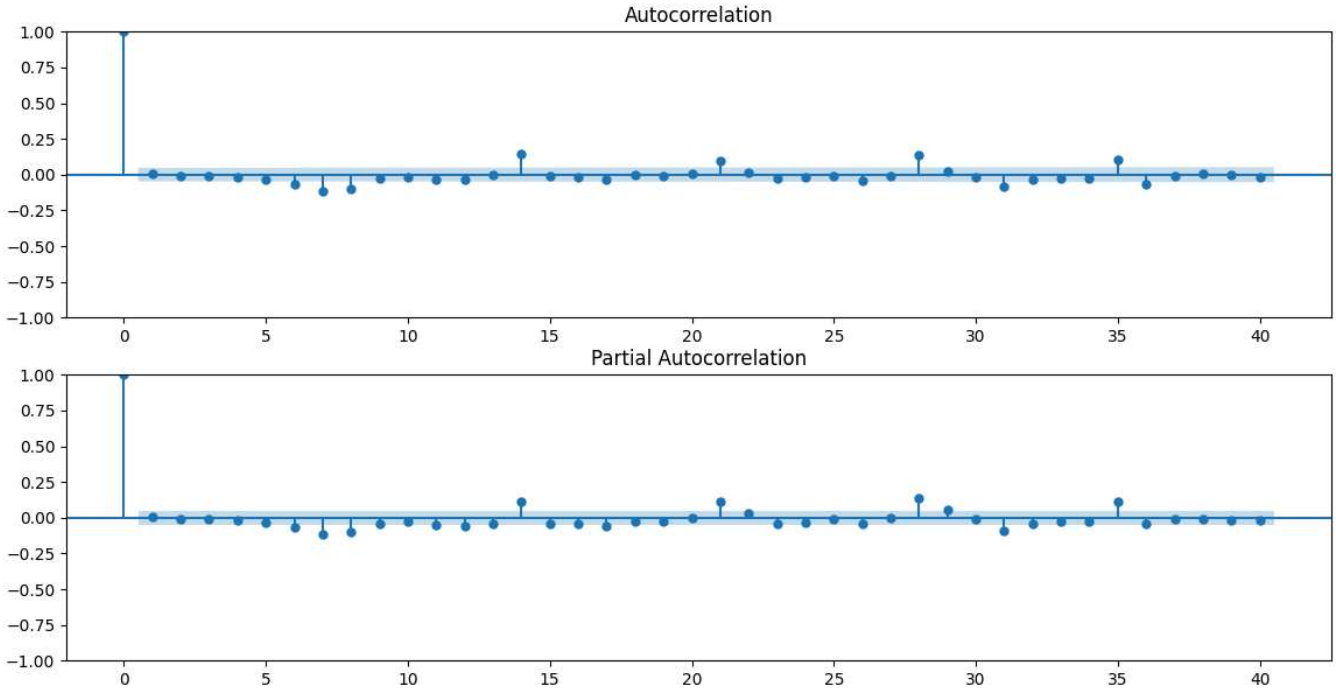
```
from statsmodels.tsa.arima.model import ARIMA

arima_model = ARIMA(arima_df['sales'], order=(6, 1, 1))
arima_fit = arima_model.fit()

print(arima_fit.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  sales   No. Observations:                 1734
Model:                 ARIMA(6, 1, 1)   Log Likelihood               -5303.393
Date:                Sat, 03 Aug 2024   AIC                          10622.787
Time:                        13:10:09   BIC                          10666.448
Sample:                    01-01-2013   HQIC                         10638.935
                         - 09-30-2017
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1         -0.6076      0.052    -11.758      0.000      -0.709      -0.506
ar.L2         -0.6030      0.041    -14.780      0.000      -0.683      -0.523
ar.L3         -0.5646      0.037    -15.134      0.000      -0.638      -0.492
ar.L4         -0.5141      0.035    -14.799      0.000      -0.582      -0.446
ar.L5         -0.4384      0.031    -14.142      0.000      -0.499      -0.378
ar.L6         -0.3330      0.027    -12.473      0.000      -0.385      -0.281
ma.L1         -0.2378      0.056     -4.238      0.000      -0.348      -0.128
sigma2        26.6135      0.819     32.507      0.000      25.009      28.218
===================================================================================
Ljung-Box (L1) (Q):                   0.02   Jarque-Bera (JB):                19.82
Prob(Q):                              0.90   Prob(JB):                         0.00
Heteroskedasticity (H):               1.42   Skew:                             0.13
Prob(H) (two-sided):                  0.00   Kurtosis:                         3.45
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

Plotting the residuals using ACF and PACF

Plotting the residuals shows that recurring correlation exists in both ACF and PACF. So we need to deal with seasonality. When the plots of ACF and PACF are similar or any sesaonality is present between them then, we need to apply the Seasonal ARIMA (SARIMA) model.

```
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

arima_model = ARIMA(arima_df['sales'], order=(6, 1, 1))
arima_fit = arima_model.fit()

residuals = arima_fit.resid

# Checking for seasonality
fig, ax = plt.subplots(2, figsize=(14,7))
ax[0] = plot_acf(residuals, ax=ax[0], lags=40)
ax[1] = plot_pacf(residuals, ax=ax[1], lags=40)

```



Configuring a SARIMA requires selecting hyperparameters for both the trend and seasonal elements of the series.

SARIMA is Seasonal ARIMA, or simply put, ARIMA with a seasonal component

The parameters for these types of models are as follows:

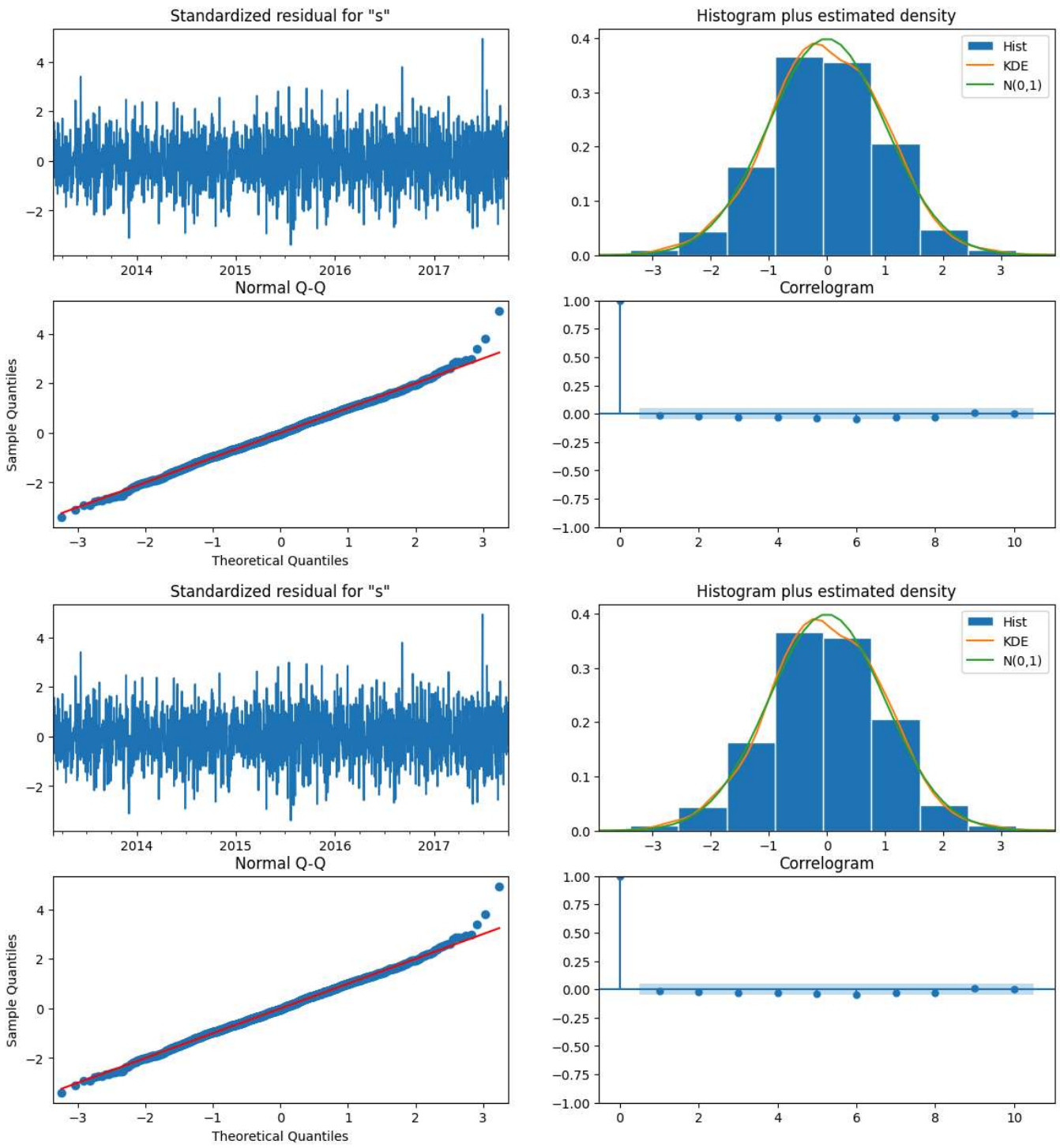
- p and seasonal P: indicate the number of AR terms (lags of the stationary series)
- d and seasonal D: indicate differencing that must be done to stationary series
- q and seasonal Q: indicate the number of MA terms (lags of the forecast errors)

It involves the following steps –

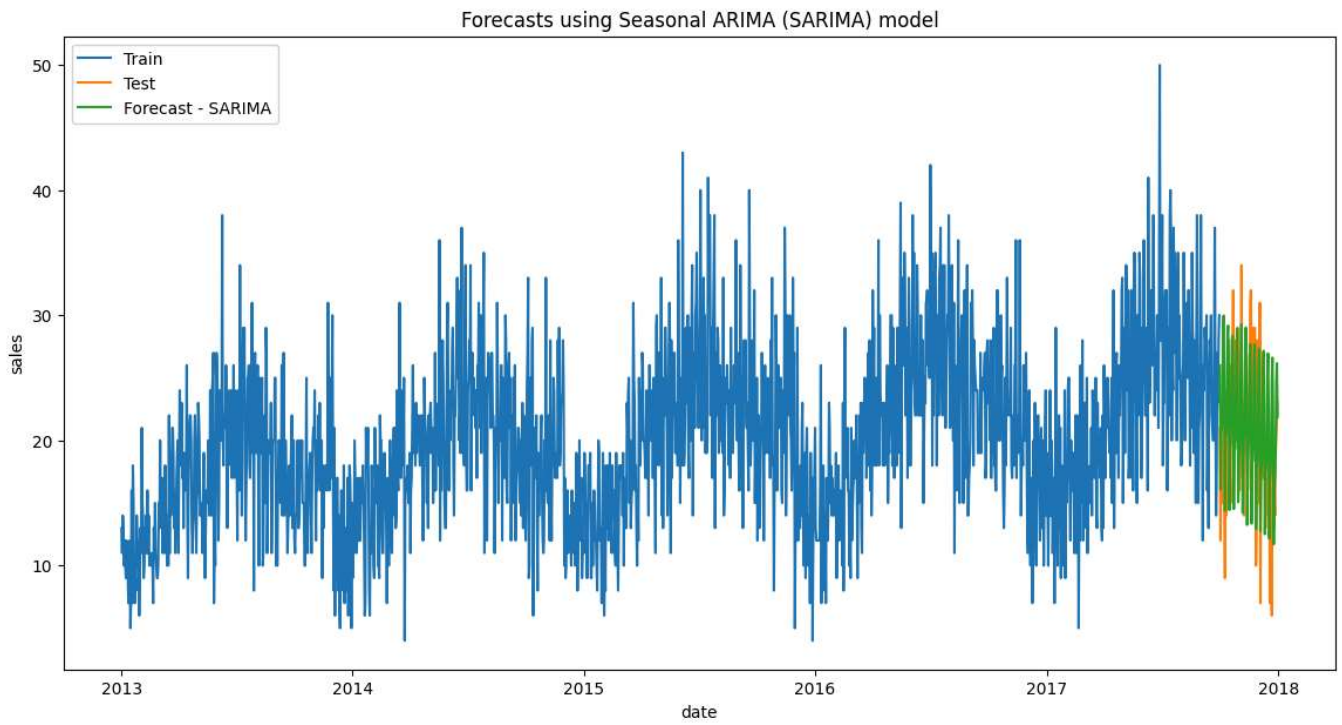
- Plot the series – to check for outliers
- Transform the data (to make mean and variance constant)
- Apply statistical tests to check if the series is stationary (Both trend and seasonality)
- If non-stationary (has either trend or seasonality), make it stationary by differencing
- Plot ACF of stationary series for MA order, Seasonal MA order at seasonal spikes
- Plot PACF of stationary series for AR order, Seasonal AR order at seasonal spikes
- Run SARIMA with those parameters
- Check for model validity using residual plots

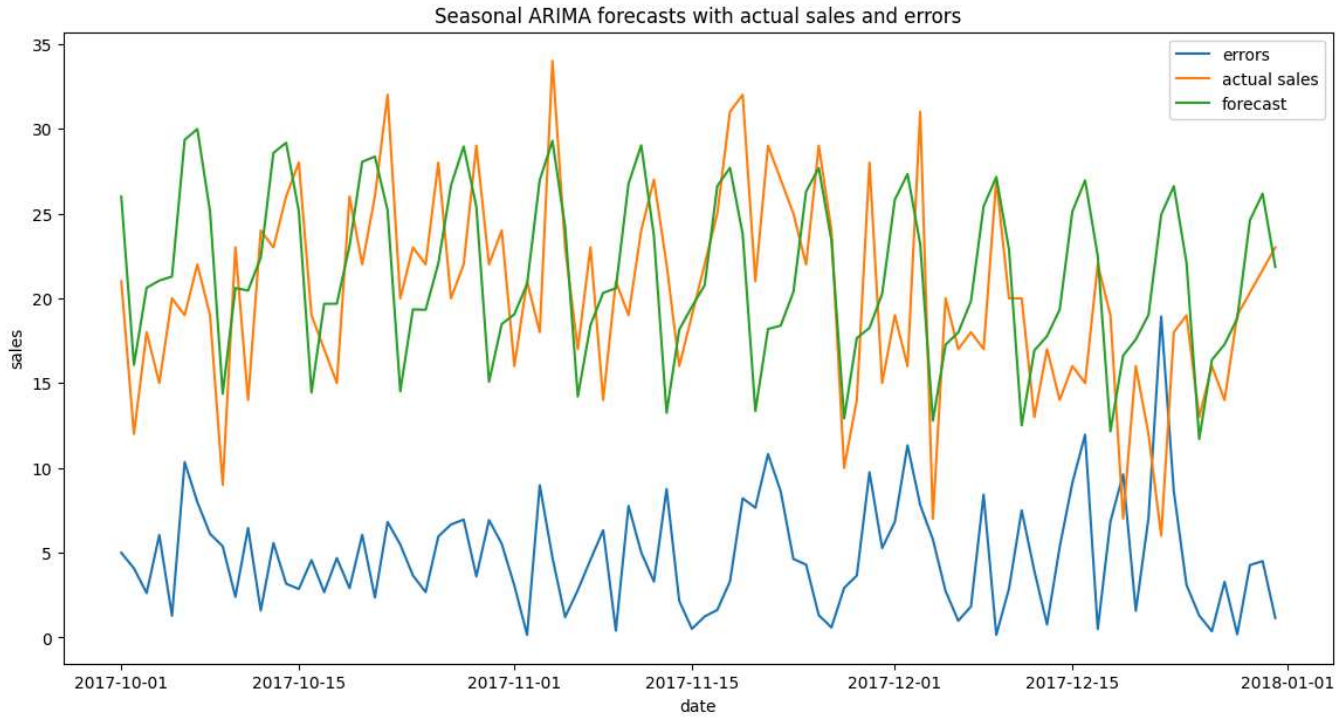
```
# # Remove outliers using MAD method
# median = df['sales'].median()
# mad = np.abs(df['sales'] - median).median()
# df['sales'][np.abs(df['sales'] - median) > 4 * mad] = median

# # Visualize data
# df.plot(figsize=(15,4))
```

```
# fit the model
sarima_model = SARIMAX(arima_df.sales, order=(6, 1, 0), seasonal_order=(6, 1, 0, 7),
                       enforce_invertibility=False, enforce_stationarity=False)
sarima_fit = sarima_model.fit()
arima_test_df['pred_sales'] = sarima_fit.predict(start=arima_test_df.index[0],
                                                 end=arima_test_df.index[-1], dynamic= Tr
plot = sarima_fit.plot_diagnostics(figsize=(14,7))
plot
```


Standardized residual for "s"
Histogram plus estimated density
Hist
KDE
N(0,1)
Normal Q-Q
Sample Quantiles
Theoretical Quantiles
Correlogram
Standardized residual for "s"
Histogram plus estimated density
Hist
KDE
N(0,1)
Normal Q-Q
Sample Quantiles
Theoretical Quantiles
Correlogram

```
# eval
arima_test_df['errors'] = arima_test_df.sales - arima_test_df.pred_sales
arima_test_df.insert(0, 'model', 'SARIMA')

# Evaluate the predictions for Seasonal ARIMA model
plt.figure(figsize=(14,7))
plt.plot(train_df['date'], train_df['sales'], label='Train')
plt.plot(arima_test_df.index, arima_test_df['sales'], label='Test')
plt.plot(arima_test_df.index, arima_test_df['pred_sales'], label='Forecast - SARIMA')
plt.legend(loc='best')
plt.xlabel('date')
plt.ylabel('sales')
plt.title('Forecasts using Seasonal ARIMA (SARIMA) model')
plt.show()

plt.figure(figsize=(14,7))
plt.plot(arima_test_df.index, np.abs(arima_test_df['errors']), label='errors')
plt.plot(arima_test_df.index, arima_test_df['sales'], label='actual sales')
plt.plot(arima_test_df.index, arima_test_df['pred_sales'], label='forecast')
plt.legend(loc='best')
plt.xlabel('date')
plt.ylabel('sales')
plt.title('Seasonal ARIMA forecasts with actual sales and errors')
plt.show()

result_df_sarima = arima_test_df.groupby('model').agg(total_sales=('sales', 'sum'),
                                          total_pred_sales=('pred_sales', 'sum'),
                                          SARIMA_overall_error=('errors', 'sum'),
                                          MAE=('errors', mae),
                                          RMSE=('errors', rmse),
                                          MAPE=('errors', mape))
result_df_sarima
```





model	total_sales	total_pred_sales	SARIMA_overall_error	MAE	RMSE	MA
SARIMA	1861.0	1974.96286	-113.96286	4.787399	5.796378	23.6668

Inference: The ARIMA model with **MAPE of 23.6%** performed better than our baseline model.

Let's look at a Causal method: Regression, in order to forecast our sales data

4. Supervised Machine Learning: Linear Regression

Let's apply Linear Regression to our time series data in order to forecasts sales.

```
reg_df = df
reg_df
```