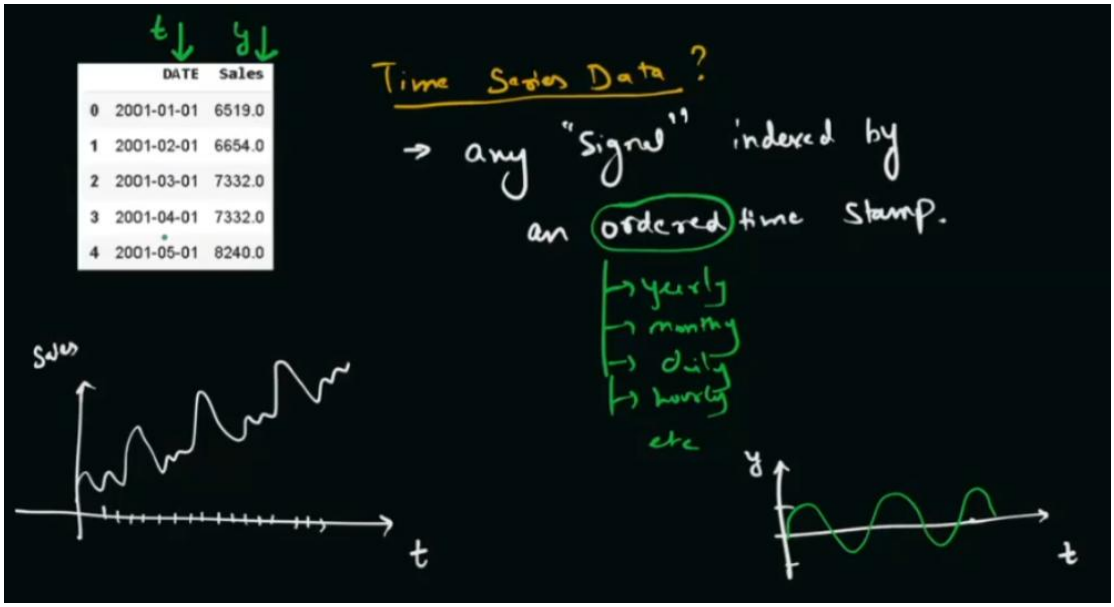
TOPICS

1. Time Series Data
2. Handling Missing Values
    - 2.1 FILLING WITH MEAN
    - 2.2 F-FILL (Forward Fill)
    - 2.3 Interpolation
    - 2.4 Centered Moving Average (CMA)
    - 2.5 Moving Average
    - 2.6 Weighted Moving Average
3. Time series decomposition
    - 3.1 Trend
    - 3.2 Seasonality
    - 3.3 Irregular Component
4. Types of Decomposition of Time Series
    - 4.1 Additive Model
    - 4.2 Multiplicative Model
5. Train Test Split
6. Iterative Approach
7. Measures of Forecast Accuracy
    - 7.1 MAE (Mean Absolute Error)
    - 7.2 MSE (Mean Squared Error)
    - 7.3 RMSE (Root Mean Squared Error)
    - 7.4 MAPE (Mean Absolute Percentage Error)
8. Simple Forecasting
    - 8.1 Mean Forecast
    - 8.2 Naive Forecast
    - 8.3 Seasonal Naive Forecast
    - 8.4 Drift Forecast
    - 8.5 Moving Average Forecast
9. Smoothing based methods
    - 9.1 Simple Exponential Smoothing (SES)
    - 9.2 Double Exponential Smoothening (DES)
    - 9.3 Triple Exponential Smoothening (TES)
10. Stationarity
11. Dickey Fuller Test
12. Converting a Time Series(TS) to Stationary TS
    - 12.1 Decomposition
    - 12.2 Differencing / De-Trend
    - 12.3 m-differencing / De-Seasonality
13. AutoCorrelation Function (ACF)
14. ARIMA Family of Forecasting Techniques
    - 14.1 Auto Regressive (AR)
    - 14.2 Moving Average (MA)
    - 14.3 Auto Regressive Moving Average (ARMA(p,q))
    - 14.4 ARIMA(p,d,q)
    - 14.4 SARIMA(p,d,q,P,D,Q,S)
15. Point Forecast
16. Confidence Interval
17. Exegeneous Variable
18. SARIMAX
19. Change Point
20. Prophet
21. Time Series Forecasting Using Linear Regression

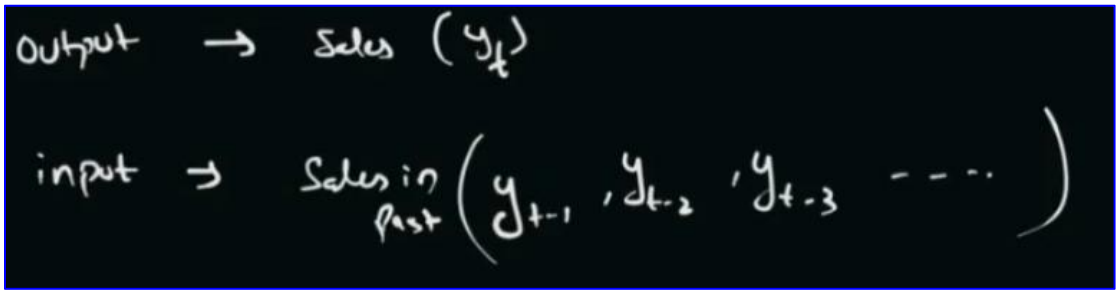
1. Time Series Data

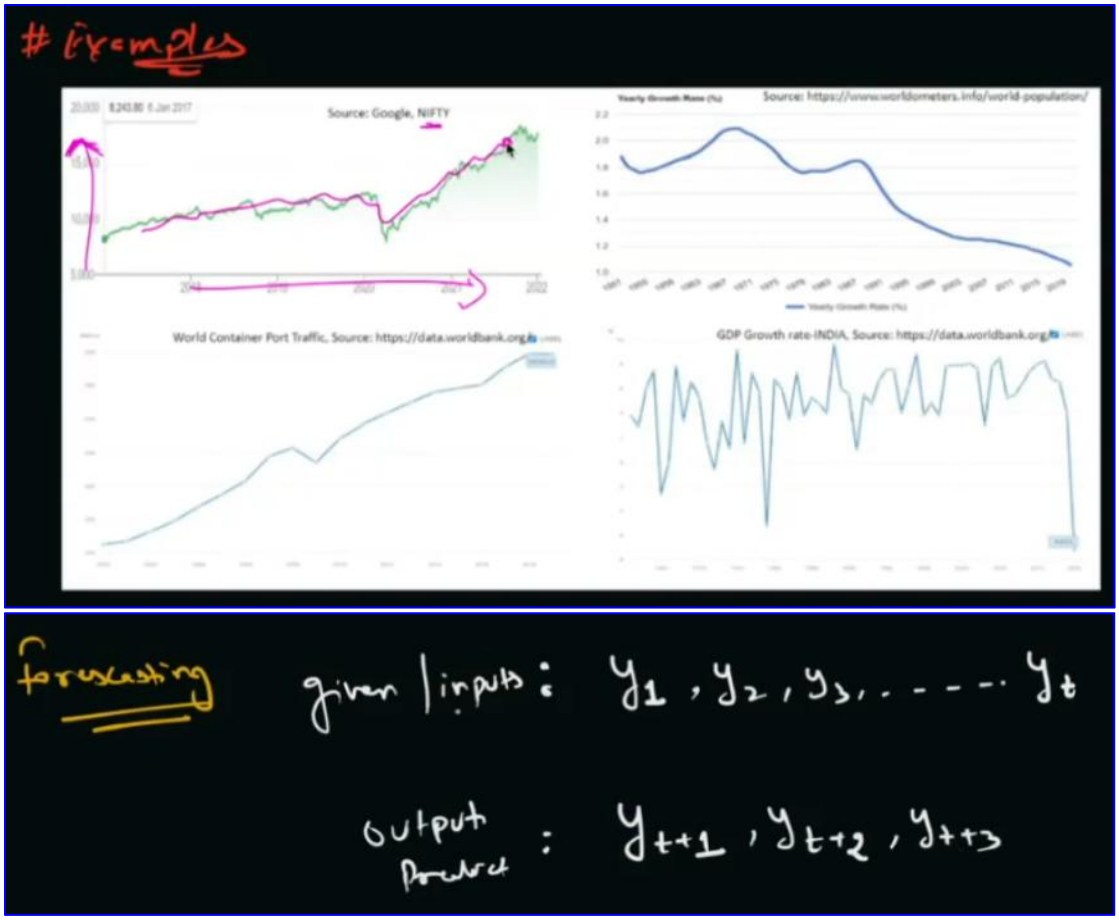


Any signal indexed by an ordered timestamp

- A signal, indexed by an ordered timestamp is time-series data.
- A time series is a sequence of measurements on the same variable collecte
- It is a set of observations, each one being recorded at equally spaced time
- For our data to be a time series, we need a minimum of two things
    - Date/timestamp (denoted as **t**)
    - One quantity (denoted as **y**)
- The timestamp(t) can be in days, weeks, months, years, or even seconds.
- Data observations like sales, revenues, inventories, etc are commonly expr series.
- There are two types of time series present:
    - Univariate time series - Data collected only for one variabl of time.
    - Multivariate time series - Data collected only for more than over the same period of time.

- Time series data are not independent.
- Since there is dependency, the ordering of the time series data is most imp Changing the order will change the data structure.





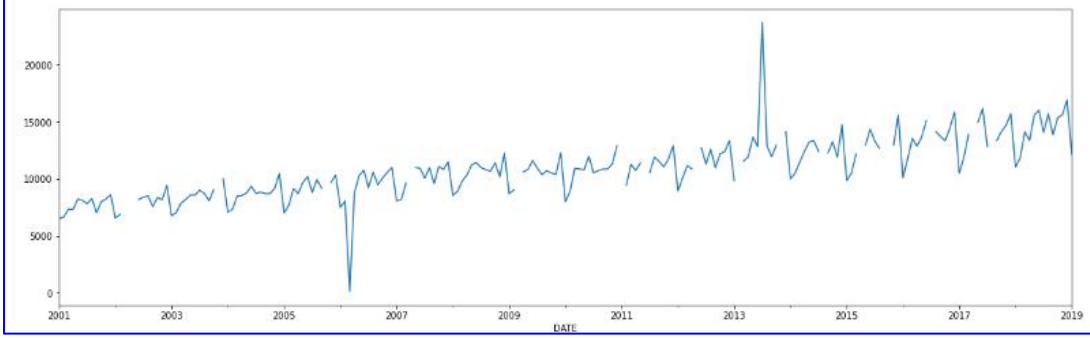
CHALLENGES

1. Missing values
2. Anomalies

2. Handling Missing Values

```
mobile_sales['Sales'].plot()
```

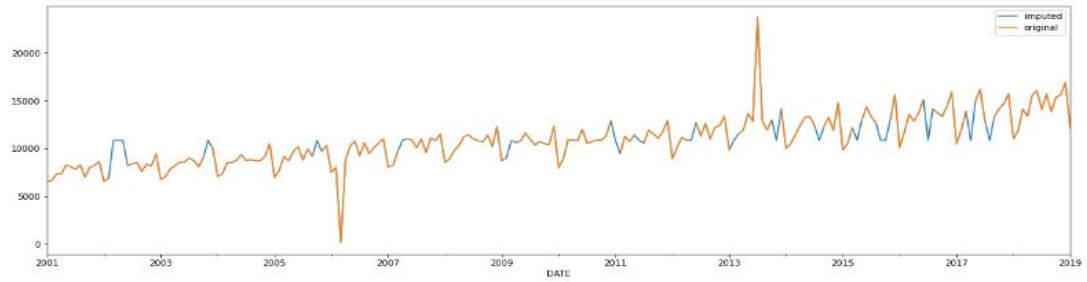
```
<AxesSubplot:xlabel='DATE'>
```



2.1 FILLING WITH MEAN

```
# WHAT-IF we fill with MEAN

mobile_sales['Sales'].mean()

mobile_sales.Sales.fillna(mobile_sales.Sales.mean()).plot(label='imputed')
mobile_sales.Sales.plot(label='original')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1f964ac07f0>
```



There is a big spike when we try to fill missing value with mean

There is a big spike when we try to fill missing value with mean

- We can clearly observe that there are sharp increments and sharp decrements at some of the imputed places. This is because we imputed the local missing values with the mean of the entire data.
- Replacing with median would've also given the same result, as they have a similar value.

Why is using mean values of the data not optimal for filling the missing values?

33 users have participated

A	The filled value may be different from local neighbours.	85%
B	As the time series has dates, it's not possible to find mean.	15%
C	Mean values will work fine even incase of missing data in time series.	0%

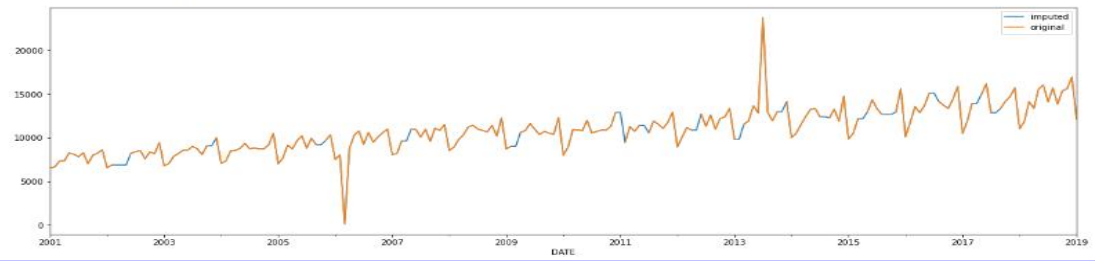
Answered in 13.8 seconds

2.2 F-FILL (Forward Fill)

F-FILL (Forward Fill)

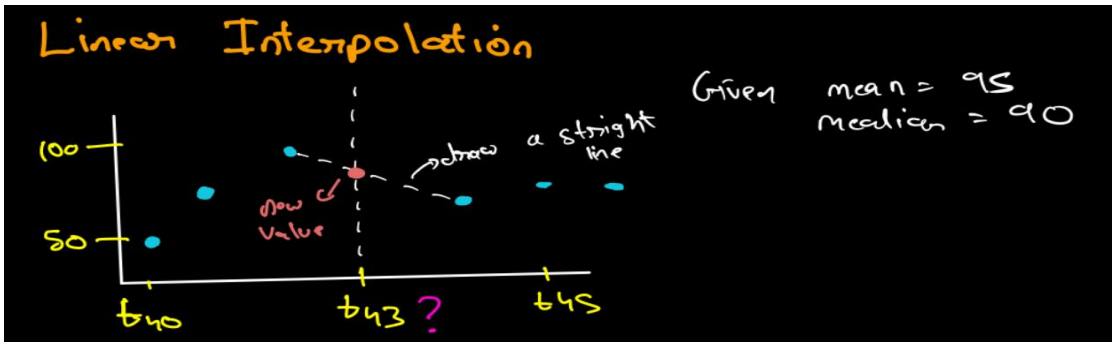
```
mobile_sales.Sales.fillna(method='ffill').plot(label='imputed')
mobile_sales.Sales.plot(label='original')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1f964cdbb20>
```



Propagate last valid observation forward to next valid

2.3 Interpolation



Linear Interpolation: Connect a straight line between the point before missing point with the point after missing point

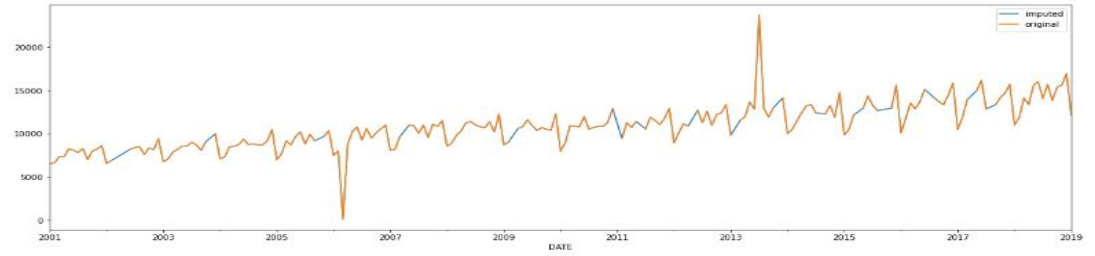
We take the **average of the first point before and first point after** the missing value and fill the missing value with this average.

- This makes sense, as this way, we would still be taking average, but instead of taking it for the entire signal, we take average for the specific gap in data.
- This will ensure that we're not under or over estimating the values.
- As the average value would be **unique** to each gap in signal.

Interpolate

```
mobile_sales.Sales.interpolate(method='linear').plot(label='imputed')
mobile_sales.Sales.plot(label='original')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x1f9648fffa0>
```




```
mobile_sales.Sales = mobile_sales.Sales.interpolate(method='linear')
```

```
mobile_sales['Sales'].isna().sum()
```

```
0
```

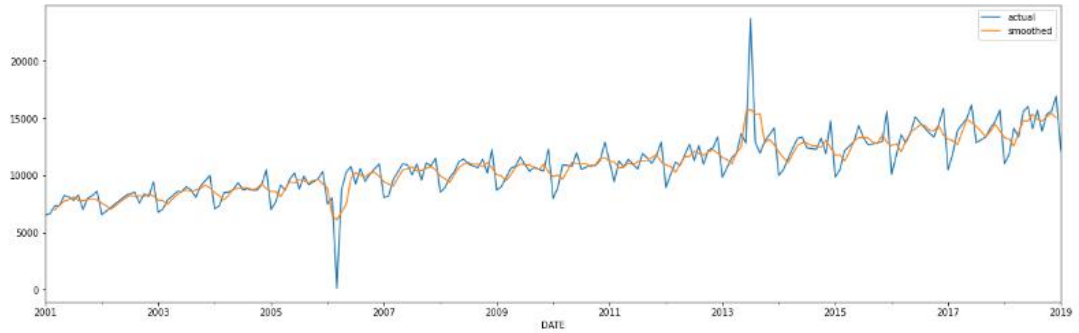
2.4 Centered Moving Average (CMA)

- The average of n points before the current point, and n points after the current point are called the Centered moving average.
- We can't use this in forecasting as future values are not available.
- This can be used to deal with the missing values and also deals with the anomalies present in the time series data.

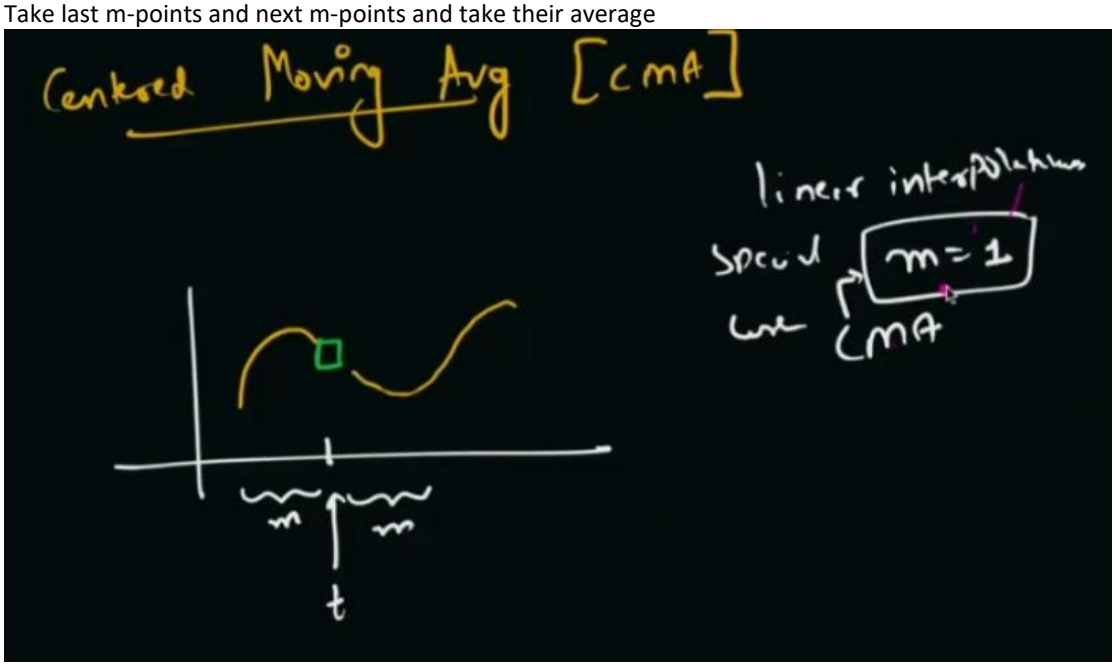
$$\hat{y}_{t+1} = \frac{1}{2m+1} \sum_{j=(t-m)}^{t+m} y_{(j)}$$

Centered Moving Average (CMA)

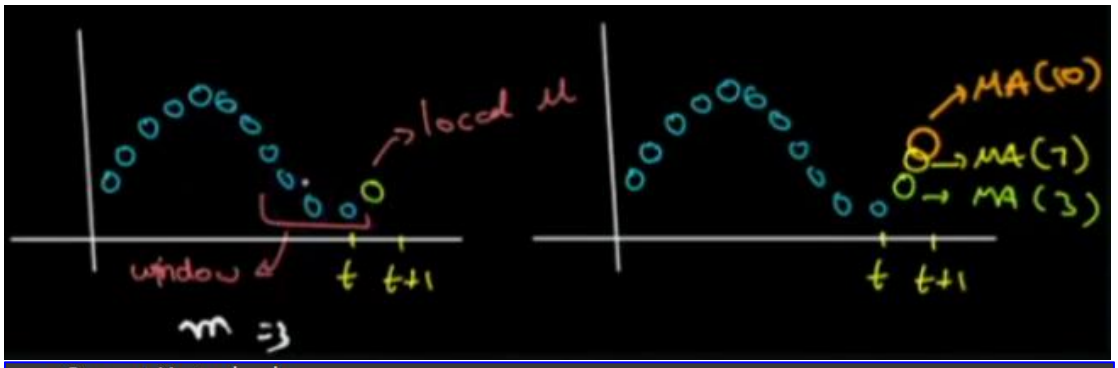
```
mobile_sales.Sales.plot(label='actual')
mobile_sales.Sales.rolling(window=4, center = True).mean().plot(label='smoothed')
plt.legend()
plt.show()
```



Take last m-points and next m-points and take their average



2.5 Moving Average



y -> Current/Actual value

$\hat{y}$ -> Forecasted value

m -> Number of observations

Let's formulate the Moving Averages concept.

$\hat{y}_t = \frac{y_{t-1}+y_{t-2}+y_{t-3}+\cdots y_{t-m}}{m}$

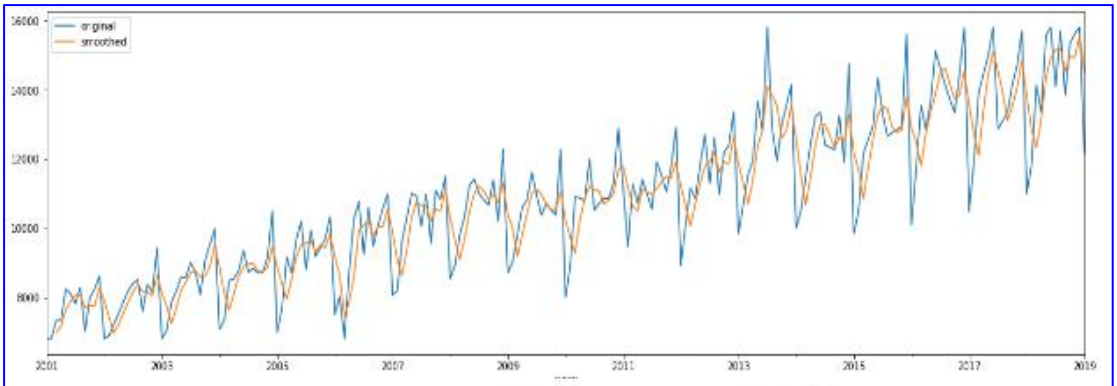
This can also be written as, $\hat{y}_t = \frac{1}{m}\sum_{j=t-m}^{t-1} y_j$

$$\hat{y}_t = \frac{y_{t-1} + y_{t-2} + y_{t-3} \cdots y_{t-m}}{m} = \frac{1}{m}\sum_{i=t-m}^{t} y_i$$

- If we take the average of the last k data points in our series and use it to guess the next point at t=k. This approach is called the Moving Average.
  i.e.

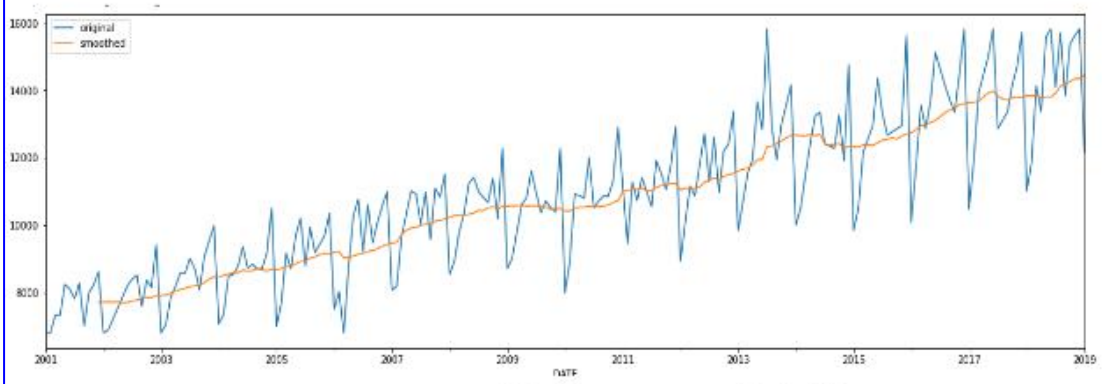
$$y_t = \frac{1}{k}\sum_{i=1}^{k} y_{t-k}$$

- The value of k acts as a hyperparameter, which we can set based on what works best for us. It is also called the **window size**.
- The moving average line has been shown below.



Moving averages with k=3

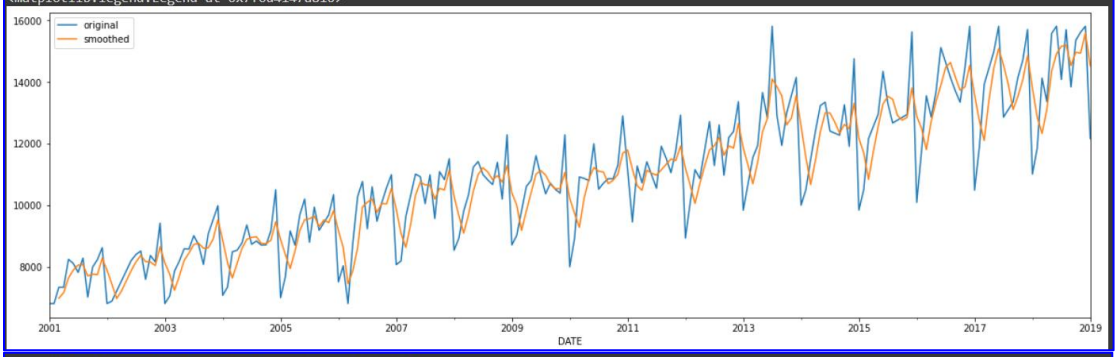
- If we increase the window, the moving average curve gets smoother.



Moving averages with k=12

```
# Plot original data
mobile_sales.Sales.plot(label='original')

# Plot data after implementing Moving Averages
mobile_sales.Sales.rolling(3, center=False).mean().plot(label='smoothed')

plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f6d4147a810>
```



Q. What observations can we make from this plot?

- We can see that the smoothed series is **lagging**.
- Notice that both maximas and minimas are lagging by 3 units along x axis. This is because our rolling window was 3.
  - Benefit of this is, if the time series value changes suddenly, the smoothed series would change a few days/weeks/months later.
  - Thereby, adjusting to the new level.
- **Less-spikey**
- Notice that there are some **missing values in the beginning**
  - This makes sense, since our `window size=3`, the very first MA value we can get is after 3 data points.

2.6 Weighted Moving Average

- Another important property of moving averages is that, when the time series goes up, the moving average also goes up and when the time series go down the moving average also goes down.
- Instead of giving all the previous observations equal weights, we can assign weights to the observations
- This is called **the weighted moving average.**

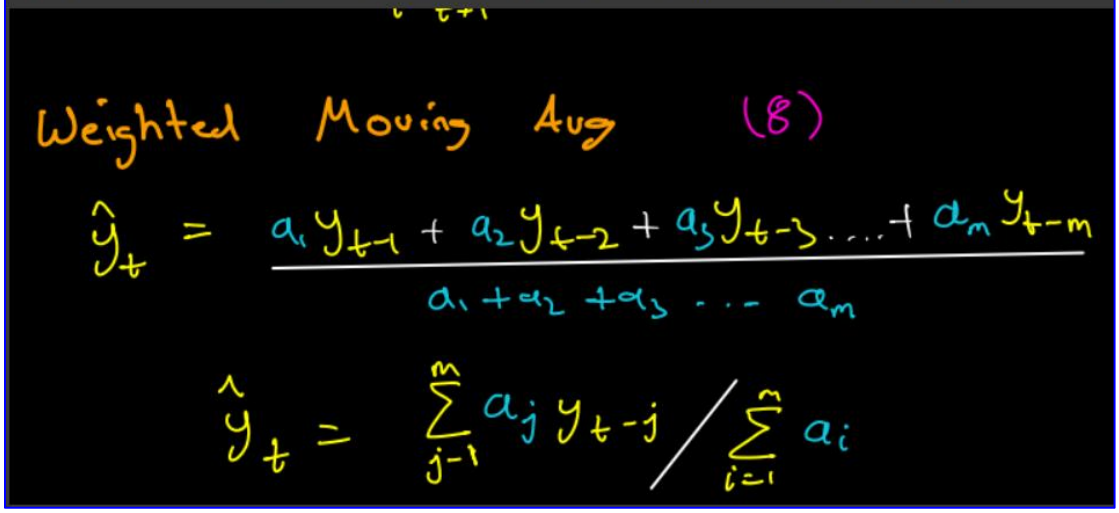
$$\hat{y}_{t+1} = \frac{a_1 y_t + a_2 y_{t-1} + a_3 y_{t-2} + \ldots a_m y_{t-m-1}}{a_1 + a_2 + \ldots a_m} = \frac{1}{\sum_{i=1}^{m} a_i} \sum_{i=1}^{m} \sum_{j=(t-m-1)}^{t} a_i y_{(j)}$$

$$\hat{y}_t = \frac{a_1 * y_{t-1} + a_2 * y_{t-2} + a_3 * y_{t-3} + \ldots a_m * y_{t-m}}{a_1 + a_2 + \ldots a_m}$$

Alternately, we can write this as,

$$\hat{y}_t = \frac{\sum_{j=1}^{m} a_j * y_{t-j}}{\sum_{i=1}^{m} a_i}$$

Naturally, we will have the following relation between the weights: $a_1 > a_2 > a_3 > \ldots > a_m$



3. Time series decomposition

- Graphs highlight the variety of patterns/ features in the time series.
- A time series can be split into several components, each representing one of the underlying categories of patterns.
- There are three important components which we want to decompose our signal into.
    - Trend - general movement over time
    - Seasonality - behaviors captured in individual seasonal periods
    - Residual - everything not captured by trend and seasonal components
- Trend, Seasonality are the systematic components of the time series data whereas Residual/ Error is an irregular component.

3.1 Trend

Shows overall pictures

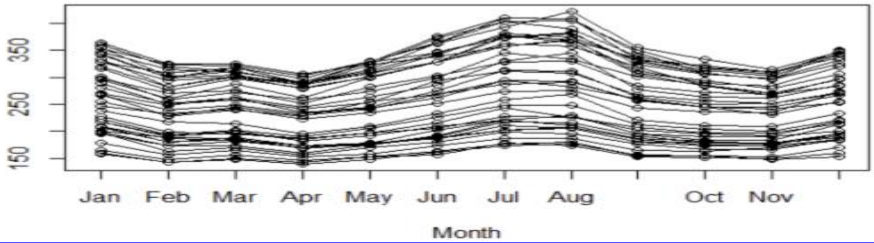
- Trend can be thought of as the linear increasing or decreasing behavior of the series over a long period of time. The trend usually happens for some time and then disappears, it does not repeat.
- A trend can be uptrend, downtrend, or can be up and down, need not be a straight line.
- Trend line is a smooth predictable function that traces the trend of a time series, and can help us predict the time series indefinitely in the future.

- Trend can be calculated by taking the rolling average (or moving average) over a long period of time or by just fitting a Linear Regression line on the points.

3.2 Seasonality

- Seasonality in time-series data refers to a pattern that occurs at a regular interval.

- Making copies of seasonality can fetch us our time series.
- A seasonal pattern occurs when a time series is affected by seasonal factors such as the time of the year or the day of the week.
- The steps for calculating the seasonality are:
    - Calculate the trend and subtract it from the original time series.
    - From the result, take the average across the period. If it is a monthly time series for 4 years, then you have 4 Jan, 4 Feb, 4 Dec, etc. Take the avg of all Jans, Febs, etc.
- A yearly series does not have seasonality.
- The signal can have multiple seasonalities one can be short-term seasonality and one can be long-term seasonality.

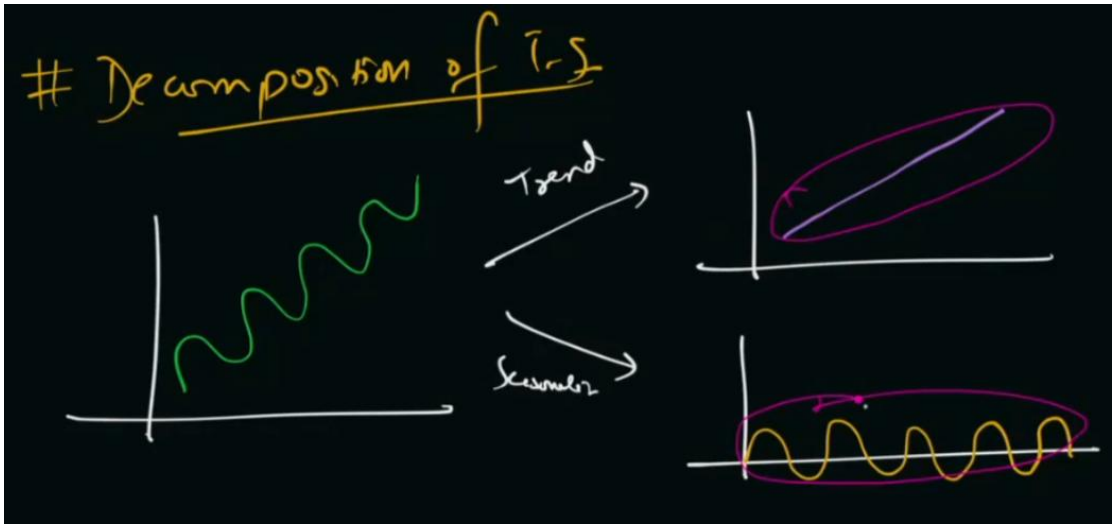


3.3 Irregular Component

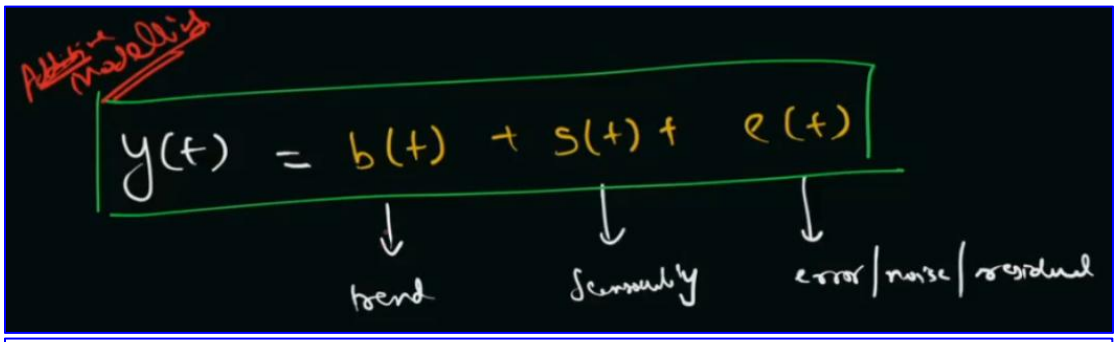
Irregular Component

- It is the random fluctuation in the time series data that the above components(i.e, trend and seasonality) cannot explain.
- This component is assumed to have a normal distribution with 0 mean and constant variance.
- This is also called Error/ White Noise/ Remainder.

4. Types of Decomposition of Time Series



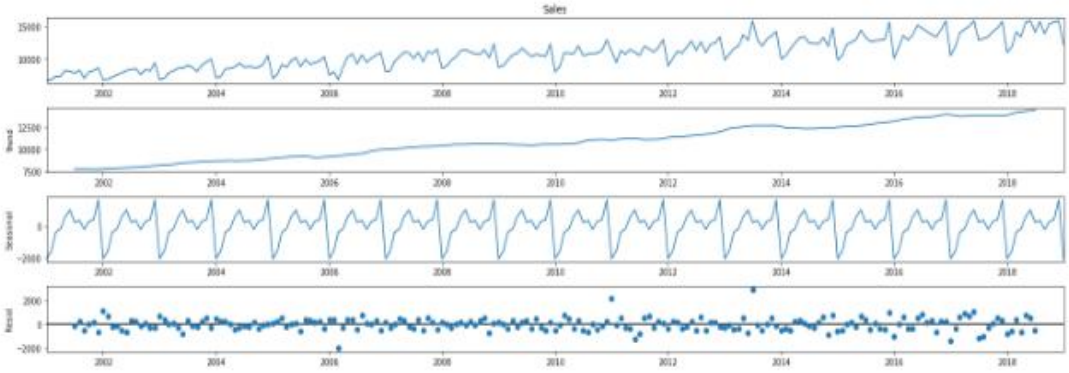
4.1 Additive Model



$$y(t) = b(t) + s(t) + e(t)$$

Where, b(t) -> trend of signal
s(t) -> seasonality
e(t) -> error term

- For the sales data, decomposition would be in the following way:

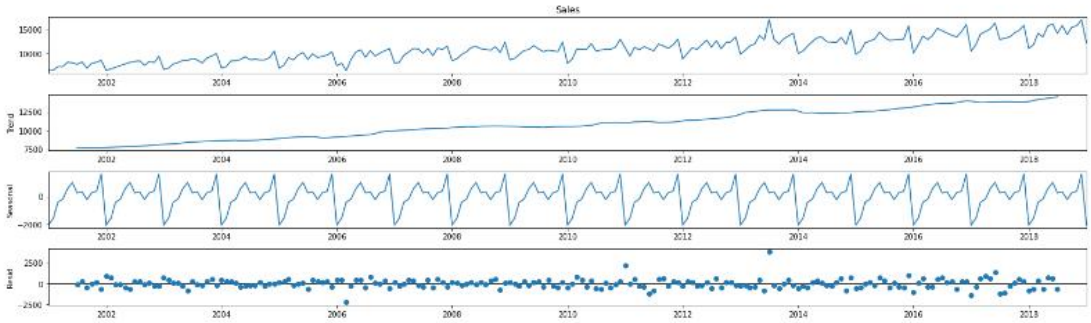


- The error term e(t) cannot be estimated since it is based on the real values that we have but can be computed using the same formulation.

$$e(t) = y(t) - b(t) - s(t)$$

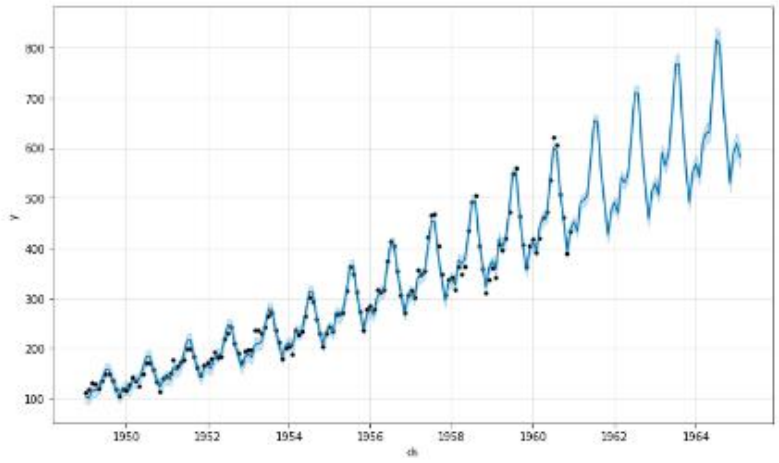
- If the trend and seasonality components obtained are a good estimate, then the errors would be small and would be scattered around zero.
- The seasonal fluctuations are not dependent on the trend and the seasonality will be constant.

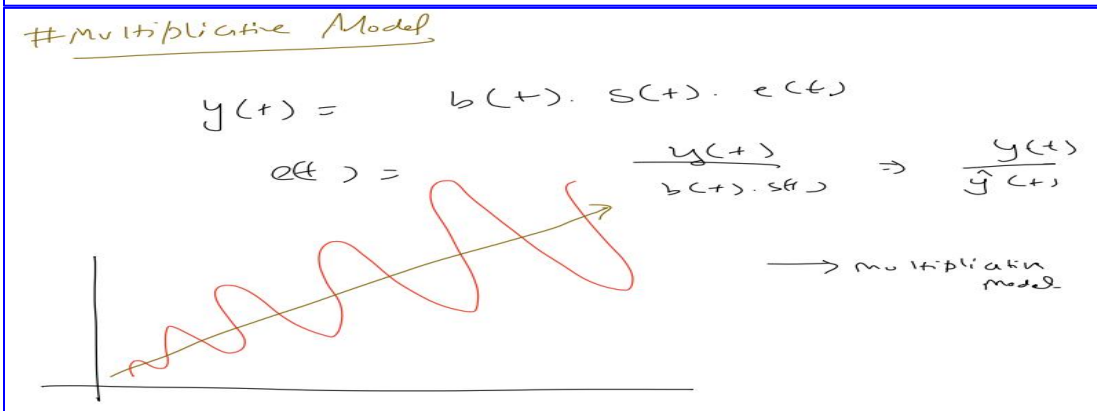
```
model = sm.tsa.seasonal_decompose(mobile_sales.Sales, model = 'additive')
model.plot();
```



4.2 Multiplicative Model

- It is a time series in which the amplitude of the seasonal component is increasing with an increasing trend.





- In multiplicative seasonality, we do not obtain the time series by adding the trend and seasonality components with each other. Instead, if we multiply them, then we obtain a time series in which the amplitude of the seasonal component is increasing with an increasing trend.
- Therefore, the **multiplicative seasonality decomposition** is written as

$$y(t) = b(t) * s(t) * e(t)$$

$$e(t) = \frac{y(t)}{b(t) * s(t)}$$

- The seasonal fluctuations are dependent on the trend.

But how do we actually decompose the time series?

- One of the methods is that we can take long-term moving averages and connect them to get a trend line.
- Another way can be by fitting a linear regression to the data points to get the trend line.
  - The line equation would be $Y_{(t+1)} = m(t) + c$
  - You can also fit with the higher degree polynomial curve.
- Once you have the trend, you can subtract the trend from the current time series and take a group average to find the seasonal component.

- A forecast refers to a calculation or an estimation which uses data from previous events. combined with recent trends to come up with a future event outcome.
- A prediction is an actual act of indicating that something will happen in the future with or without prior information.
- So, all forecasts are predictions, but not all predictions are forecast would be a correct statement.
- Also, in time series, forecasting seems to predict a future value based on the past values.
- Whereas, in regression, predictions estimate a value whether it is a future, present or past with respect to the given data.

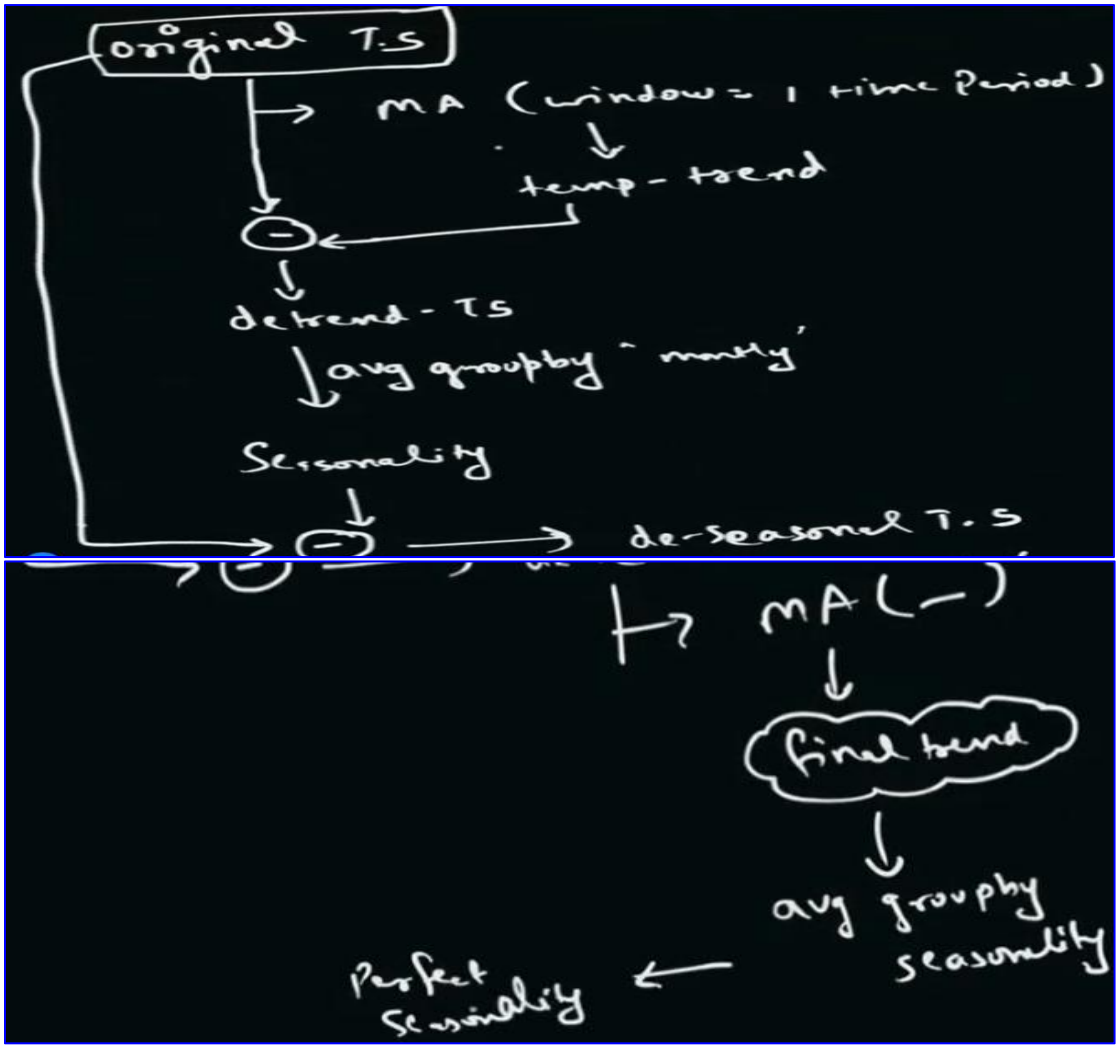
Day 2

5. Train Test Split

- For time series data, we do not perform a random shuffle for train and test data split.
- Random shuffling on time series data would be meaningless since it is a collection of data over time and we look at the past data to predict future values.
- Therefore, we do a **time-based splitting**.
- In the train-test split we hold out the most recent data to be as the test data.
- If there is seasonality present in the data at least two full seasons of data are to be taken as the test data.
- For example, for our sales dataset, out of the 18 years of data, we decide to train on 17 years of data and use the last year, i.e. the last 12 values for test data.

DATE	Sales
2018-02-01	11852.0
2018-03-01	14123.0
2018-04-01	13360.0
2018-05-01	15576.0
2018-06-01	15809.4
2018-07-01	14080.0
2018-08-01	15697.0
2018-09-01	13838.0

6. Iterative Approach



7. Measures of Forecast Accuracy

7.1 MAE (Mean Absolute Error)

Represents the difference between the original and predicted values extracted by averaging the absolute difference over the data set

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}|$$

7.2 MSE (Mean Squared Error)

Represents the difference between the original and predicted values extracted by squaring the average difference over the data set

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2$$

7.3 RMSE (Root Mean Squared Error)

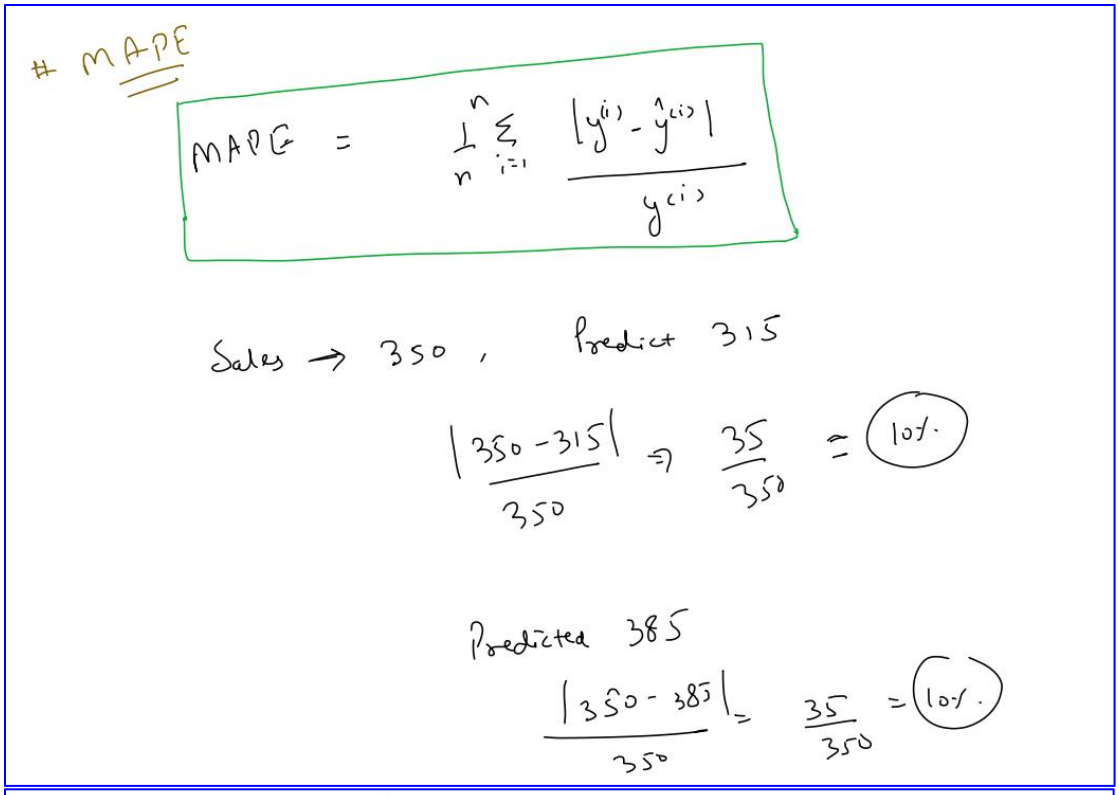
The error rate by the square root of MSE

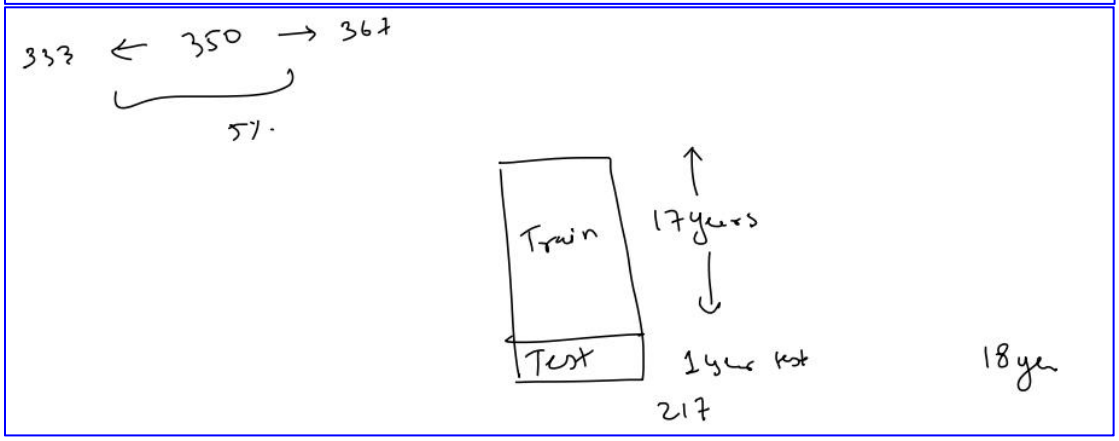
$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2}$$

7.4 MAPE (Mean Absolute Percentage Error)

The mean or average of the absolute percentage errors of forecasts. Error is defined as the actual or observed value minus the forecasted value

$$MAPE = \left(\frac{1}{N}\Sigma\frac{|y_i - \hat{y}_i|}{|y_i|}\right) * 100$$





We want our model to have less percentage error

```
# Creating a function to print values of all these metrics.
def performance(actual, predicted):
    print('MAE :', round(mae(actual, predicted), 3))
    print('RMSE :', round(mse(actual, predicted)**0.5, 3))
    print('MAPE:', round(mape(actual, predicted), 3))
```

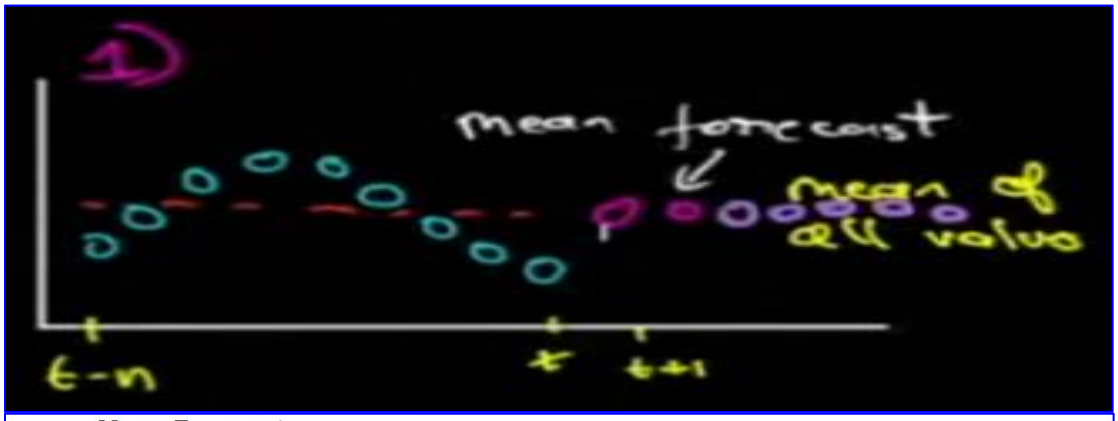
8. Simple Forecasting

8.1 Mean Forecast

Mean of all the value. All the prediction will be same

- We take the average of all past k values of the time series to forecast the value at time $t=k+1$.

$$x_{k+1} = \frac{x_1 + x_2 + \ldots + x_k}{k}$$



Mean Forecast

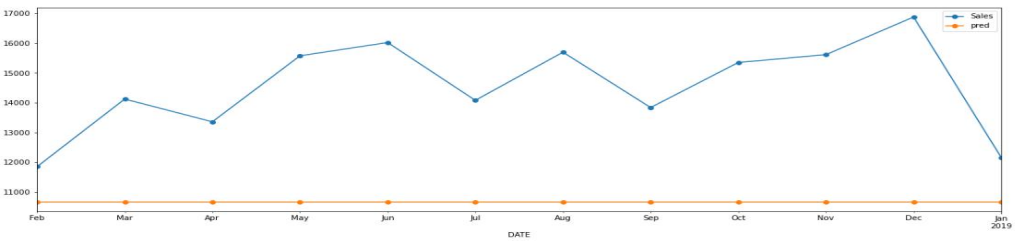
```
In [38]: train_x['Sales'].mean()
Out[38]: 10655.57756097561

In [39]: test_x['pred'] = train_x['Sales'].mean()
         test_x.head()
```

Out[39]:

DATE	Sales	pred
2018-02-01	11852.0	10655.577561
2018-03-01	14123.0	10655.577561
2018-04-01	13360.0	10655.577561
2018-05-01	15576.0	10655.577561
2018-06-01	16021.0	10655.577561

```
In [41]: test_x.plot(style='-o')
         performance(test_x['Sales'], test_x['pred'])
```

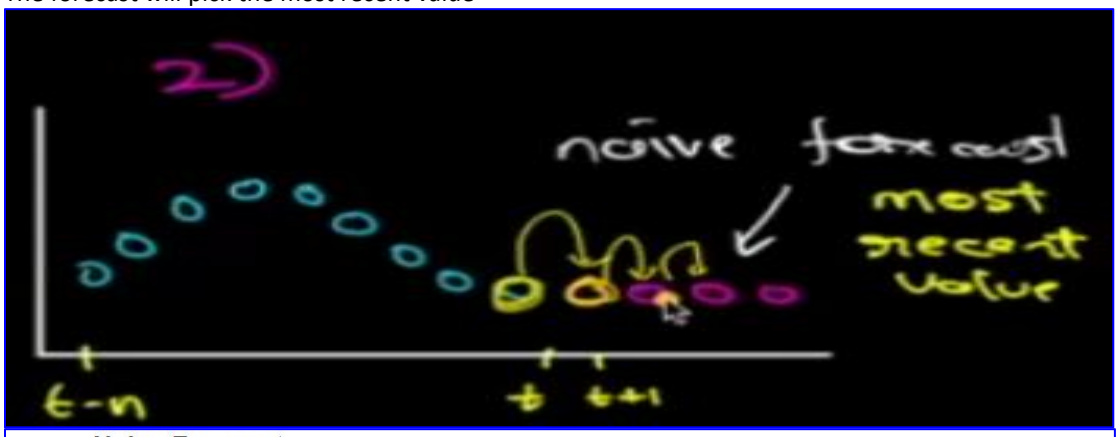
```
MAE : 3890.436
RMSE : 4170.094
MAPE: 0.259
```



- The MAPE metric is showing that we have a 25.5% error, with respect to its own value. Thus, it is not a good model.
- A similar argument can be made for the median. So, we can rule out using the median of the entire data as future forecast values.
- This method gives equal importance to each observation.

8.2 Naive Forecast

In this approach, we simply take the value of the series at time t=k and forecast that for the future.
The forecast will pick the most recent value



Naive Forecast

```
In [42]: train_x['Sales'][-1]
```

```
Out[42]: 11005.0
```

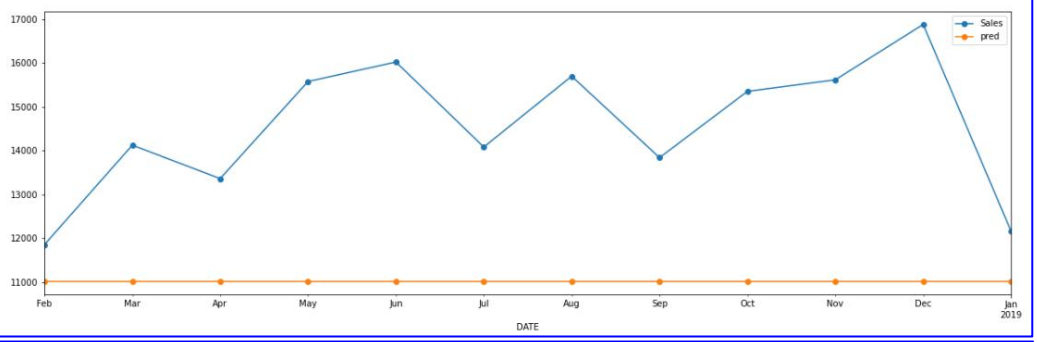
```
In [43]: test_x['pred'] = train_x['Sales'][-1]
         test_x.head()
```

Out[43]:

DATE	Sales	pred
2018-02-01	11852.0	11005.0
2018-03-01	14123.0	11005.0
2018-04-01	13360.0	11005.0
2018-05-01	15576.0	11005.0
2018-06-01	16021.0	11005.0

```
In [44]: test_x.plot(style='-o')
         performance(test_x['Sales'], test_x['pred'])
```

```
MAE : 3541.013
RMSE : 3846.163
MAPE: 0.235
```

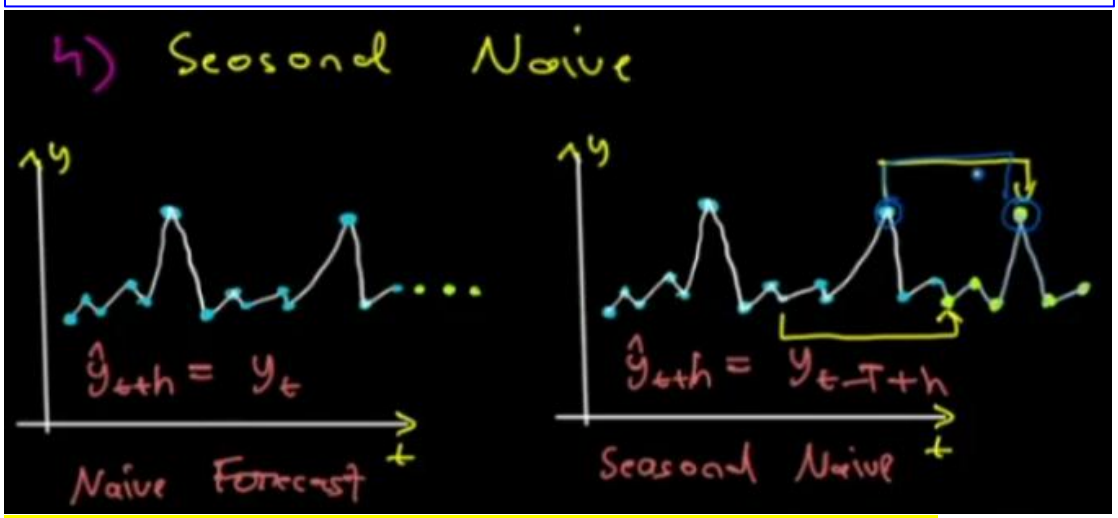


- This model's performance is slightly better(23%), but we can see it's still really bad.

- But if we get a different value for series at t=k, this would change all the future forecasts we had made earlier. Hence, this is not a very intelligent approach.
- These forecasts are based only on the past values of the variable
- Thus forecast is optimal when data follow a random walk.

8.3 Seasonal Naive Forecast

- It is a smarter approach to optimize the Naive approach.
- We set each forecast to be equal to the last observed value from the same season (e.g., the same month of the previous year).
- This way, we essentially forecast the future values to be exactly the same as last season.



It will predict by picking up data from history during that period from previous season

Seasonal Naive Forecast

In [45]:
```
train_x.loc[test_x.index[1] - pd.DateOffset(years= 1)]
```

Out[45]:
```
Sales    13921.0
Name: 2017-03-01 00:00:00, dtype: float64
```

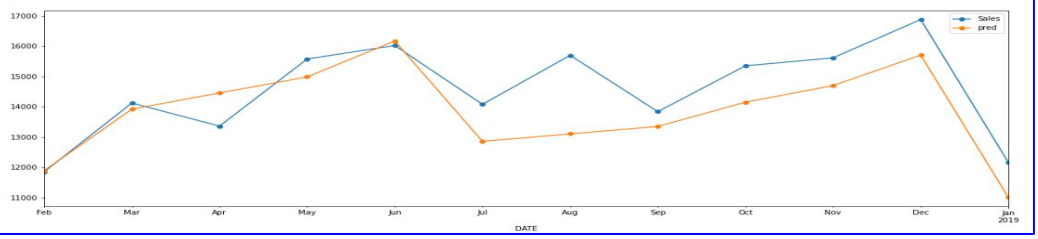
In [46]:
```
for i in test_x.index:
    test_x.loc[i]['pred'] = train_x.loc[i - pd.DateOffset(years=1)].Sales

test_x
```

Out[46]:

DATE	Sales	pred
2018-02-01	11852.00	11891.0
2018-03-01	14123.00	13921.0
2018-04-01	13360.00	14453.5
2018-05-01	15576.00	14986.0
2018-06-01	16021.00	16168.0
2018-07-01	14080.00	12856.0
2018-08-01	15697.00	13102.5
2018-09-01	13838.00	13349.0
2018-10-01	15351.00	14150.0
2018-11-01	15615.00	14698.0
2018-12-01	16879.16	15704.0
2019-01-01	12160.00	11005.0

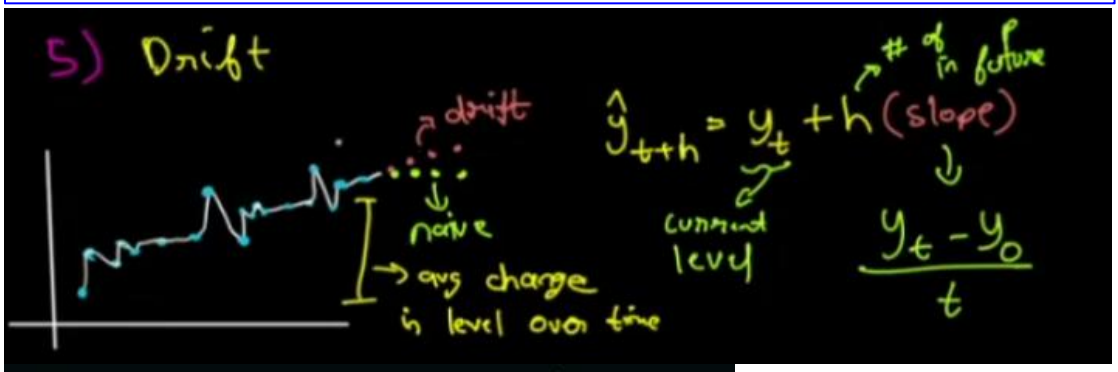
In [47]:
```
test_x.plot(style='-o')
performance(test_x['Sales'], test_x['pred'])
```

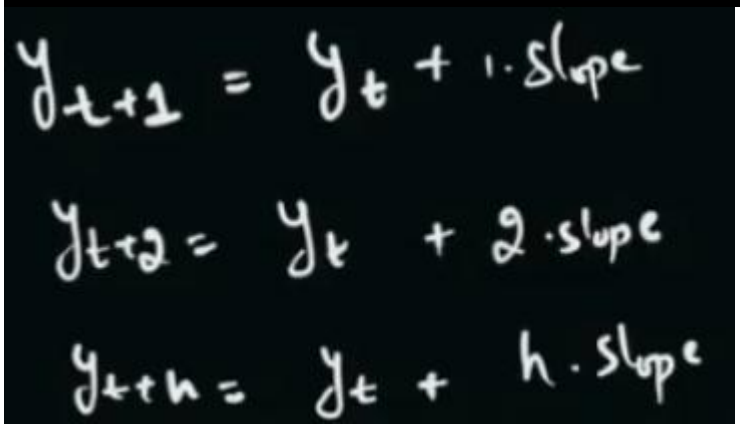
```
MAE : 902.263
RMSE : 1120.819
MAPE: 0.061
```



8.4 Drift Forecast

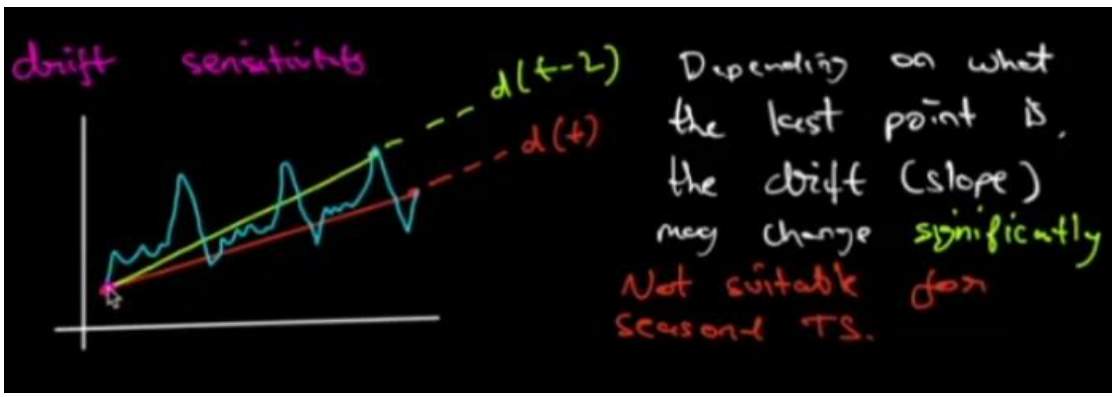
- A variation on the naïve method is to allow the forecasts to increase or decrease over time, where the amount of change over time (called the **drift**) is set to be the average change seen in the historical data.
- In this method of forecasting, we take the first and last point of the data and draw a straight line between those points, and then extend this line into the future to get a forecast (**Linear extrapolation**).
- So, instead of just picking up some value from the past, we let our values increase or decrease over time.
- Depending on what the last point is, the drift (slope) may change significantly. thus, this is highly sensitive to the last value available.





It takes into account the slope of the data i.e the trend of the data

Drawbacks



This method is very sensitive to the last point

Drift Forecast

```
In [53]: y_t=train_x['Sales'][-1]
         y_0=train_x['Sales'][0]

         slope = (y_t - y_0)/len(train_x['Sales'])
         h=np.linspace(0,len(test_x)-1,len(test_x))
         h
```

```
Out[53]: array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

```
In [55]: test_x['pred']=y_t + h*slope
```

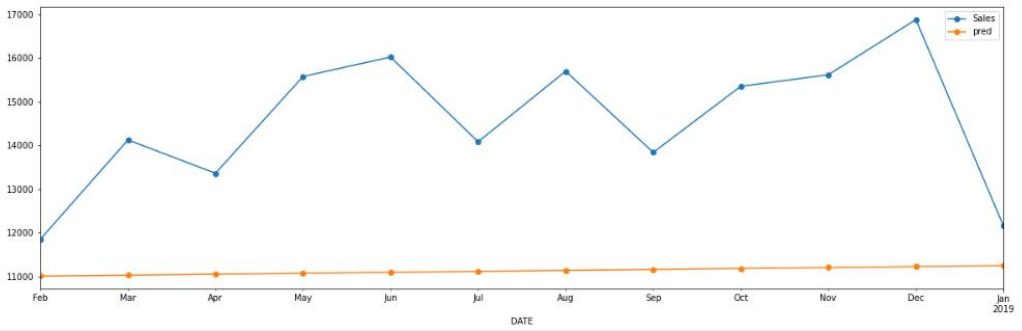
```
In [56]: test_x
```

Out[56]:

DATE	Sales	pred
2018-02-01	11852.00	11005.000000
2018-03-01	14123.00	11026.867707
2018-04-01	13360.00	11048.735415
2018-05-01	15576.00	11070.603122
2018-06-01	16021.00	11092.470829
2018-07-01	14080.00	11114.338537
2018-08-01	15697.00	11136.206244
2018-09-01	13838.00	11158.073951
2018-10-01	15351.00	11179.941659
2018-11-01	15615.00	11201.809366
2018-12-01	16879.16	11223.677073
2019-01-01	12160.00	11245.544780

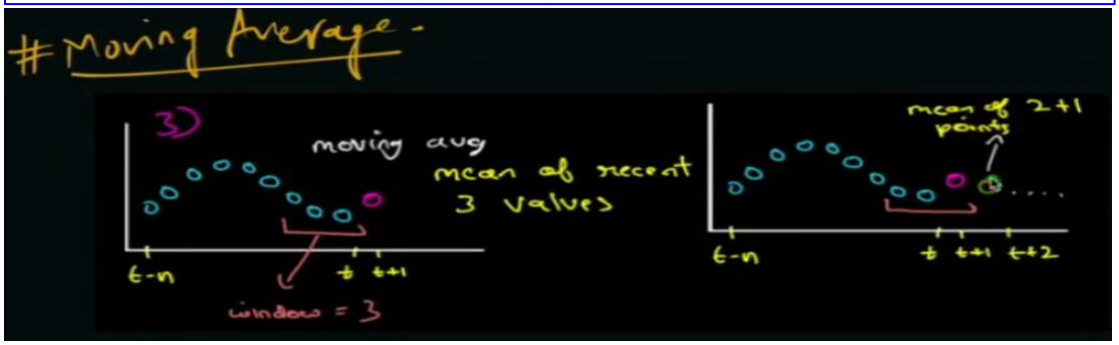
```
In [57]: test_x.plot(style='-o')
         performance(test_x['Sales'], test_x['pred'])
```

```
MAE : 3420.741
RMSE : 3727.232
MAPE: 0.227
```

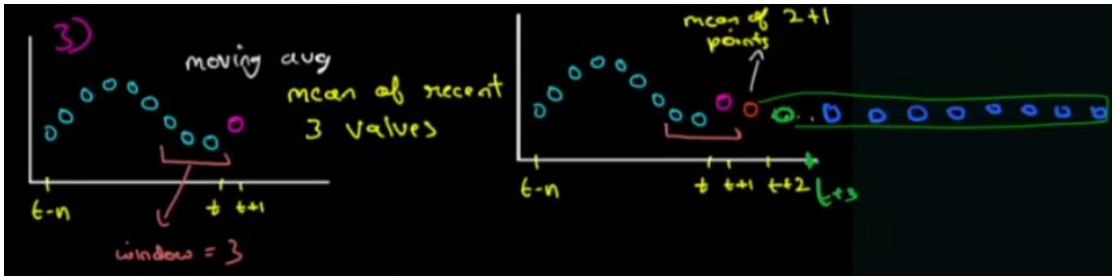


8.5 Moving Average Forecast

- The average of the last k data points in our series is used to forecast the next point in this approach.
- We cannot consider the data points outside the window.
- For our sales data, our model has an 11.5 % error when using this model. This is much better than the other simple models we saw.



Drawback



This method starts to saturate after sometime. Cannot use this for long term

Moving Average

In [58]:
```
pred_ = train_x.Sales.dropna().copy().values

for i in range(12):
    pred_ = np.append(pred_, pred_[-3:].mean()) #Taking the last 3 values and appending it to the list
```

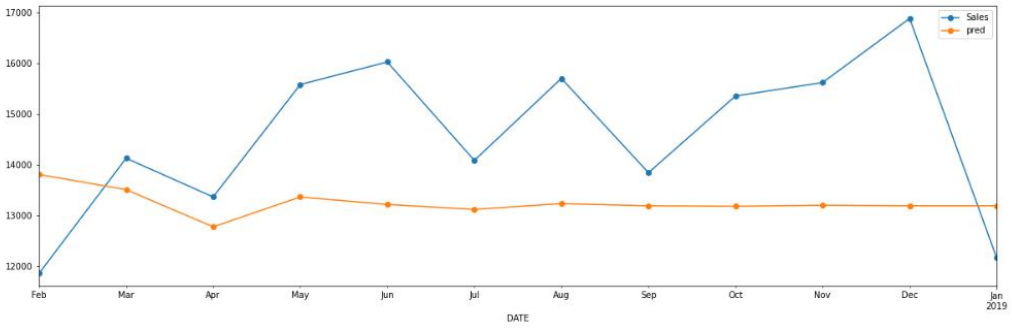
In [61]:
```
pred_
```

Out[61]:
```
array([ 6522.12      ,  6654.        ,  7332.        ,  7332.        ,
        8240.        ,  8104.        ,  7813.        ,  8279.        ,
        7014.        ,  7985.        ,  8229.        ,  8618.        ,
        6558.        ,  6882.        ,  7211.        ,  7540.        ,
        7869.        ,  8198.        ,  8396.        ,  8510.        ,
        7589.        ,  8366.        ,  8156.        ,  9415.        ,
        6765.        ,  7048.        ,  7856.        ,  8181.        ,
        8581.        ,  8573.        ,  9008.        ,  8708.        ,
        8074.        ,  9068.        ,  9524.5       ,  9981.        ,
        7071.        ,  7339.        ,  8483.        ,  8536.        ,
        8774.        ,  9355.        ,  8728.        ,  8835.        ,
        8701.        ,  8709.        ,  9170.        , 10499.        ,
        6994.        ,  7675.        ,  9161.        ,  8699.        ,
        9682.        , 10198.        ,  8794.        ,  9935.        ,
        9182.        ,  9433.        ,  9684.        , 10339.        ,
        7507.        ,  8028.        ,  6522.12      ,  8777.        ,
       10281.        , 10767.        ,  9233.        , 10595.        ,
        9475.        , 10077.        , 10569.        , 10987.        ,
        8068.        ,  8185.        ,  9634.        , 10320.5       ,
       11007.        , 10922.        , 10045.        , 10983.        ,
        9564.        , 11087.        , 10828.        , 11504.        ,
        8533.        ,  8923.        ,  9804.        , 10335.        ,
       11232.        , 11415.        , 10993.        , 10817.        ,
       10667.        , 11388.        , 10196.        , 12280.        ,
```

In [59]:
```
test_x['pred'] = pred_[-12:]
```

In [60]:
```
test_x.plot(style='-o')

performance(test_x['Sales'], test_x['pred'])
```

```
MAE : 1799.247
RMSE : 2043.135
MAPE: 0.12
```



9. Smoothing based methods

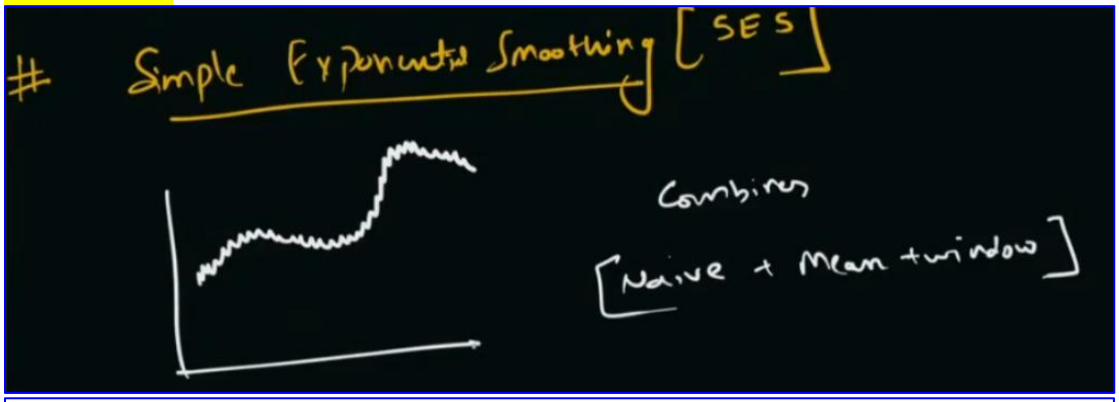
- The exponential smoothing technique is a weighted moving average procedure where the exponential decline of weights happens as the data becomes older.
- More weightage is given to the recent observations and less weightage is given to the past/old observations.
- This method overcomes the shortcomings of the moving average method.
- The smoothing parameters control how fast the weights decay and these parameter values lie between 0 and 1.
- There are three types of exponential smoothing methods:
    - SES
    - DES
    - TES

9.1 Simple Exponential Smoothing (SES)

This method combines

(Naive + Mean + Window size)

Finds the LEVEL



- The key idea of this method is to keep some memory of the entire time series, but also, we want to give more value to the recent data and less value to the past value. This forms a decaying trend.
- This method is used when there is no trend or seasonality present in the data.
- Let's consider the weight we assign to the recent most value be α.
  α is called the **smoothing parameter**.
  So, our forecast at time *t* for the time t+1 is:

$$\hat{y}_{t+1} = \alpha y_t + (1-\alpha)\hat{y}_t$$

- The above formulation is recursive in nature and expands in the following form:

$$\hat{y}_{t+1} = \alpha y_t + \alpha(1-\alpha)y_{t-1} + \alpha(1-\alpha)^2\hat{y}_{t-2} + \dots$$

- We can observe from the formulation that the weights are exponentially decaying. Therefore, we give more weightage to the most recent values, and this weightage keeps decreasing for earlier values.

$$\hat{y}_{t+1} = \left(\frac{1}{t}\right) y_t + \left(\frac{1}{t}\right) y_{t-1} + \cdots \left(\frac{1}{\epsilon}\right) y_0$$

2nd. 1st. 0th.

Most weightage

least weightage

Recursive formation:

$$\hat{y}_{t+1} = \alpha \cdot y_t + (1-\alpha)\hat{y}_t$$

$\alpha \rightarrow$ H.P $\in (0,1)$

$$\hat{y}_{t+1} = \alpha \cdot y_t + (1-\alpha)\left[\alpha \cdot y_{t-1} + (1-\alpha)\hat{y}_{t-1}\right]$$

$$\hat{y}_{t+1} = \alpha \cdot y_t + (1-\alpha)\left[\alpha \cdot y_{t-1} + (1-\alpha)\left[\alpha \cdot y_{t-2} + (1-\alpha)\hat{y}_{t-2}\right]\right]$$

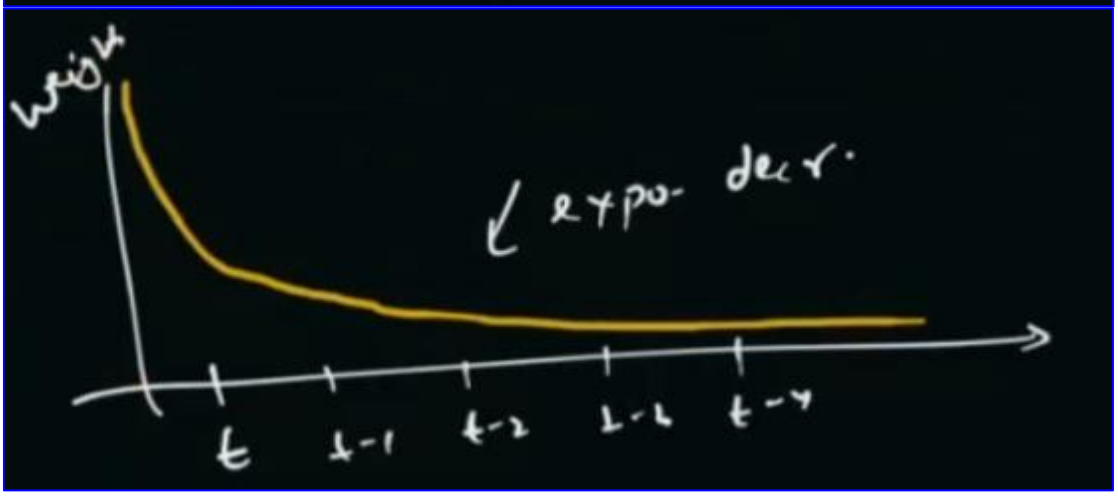
$$\hat{y}_{t+1} = \alpha y_t + \alpha(1-\alpha) y_{t-1} + \alpha(1-\alpha)^2 y_{t-2} + \alpha(1-\alpha)^3 y_{t-3} + \cdots$$

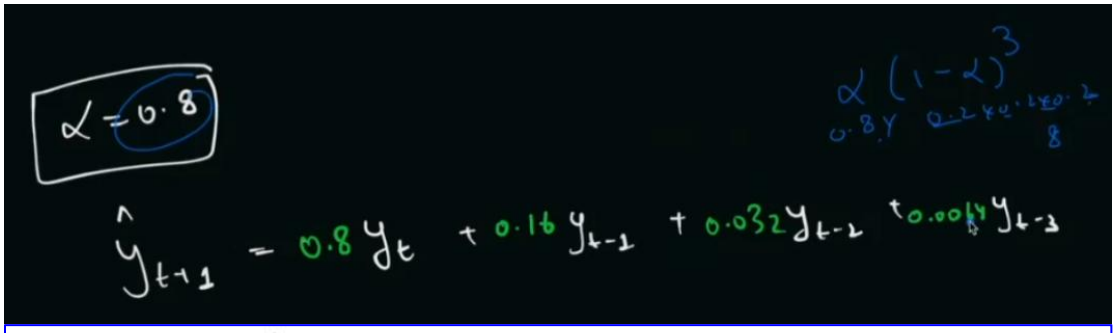
most recent

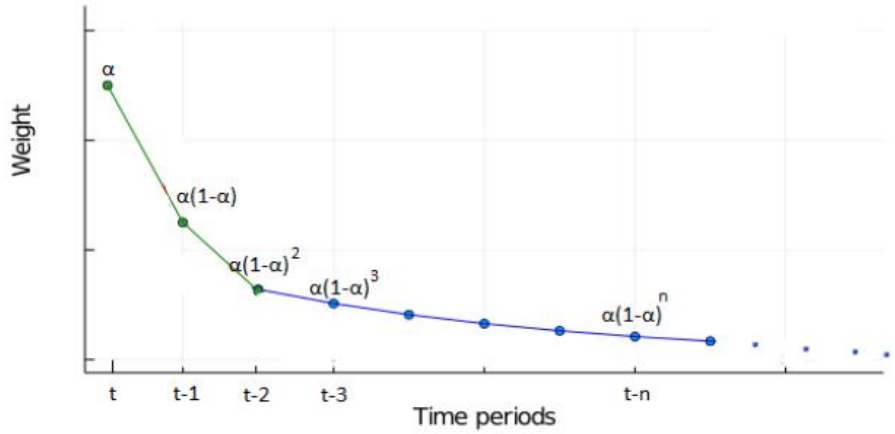
second recent

older

more old

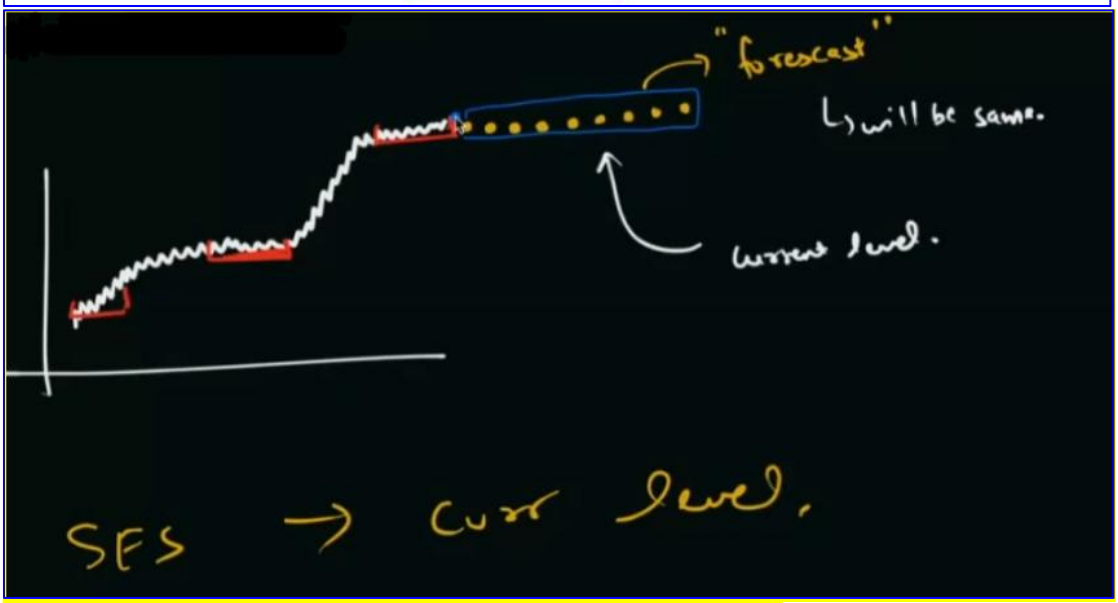




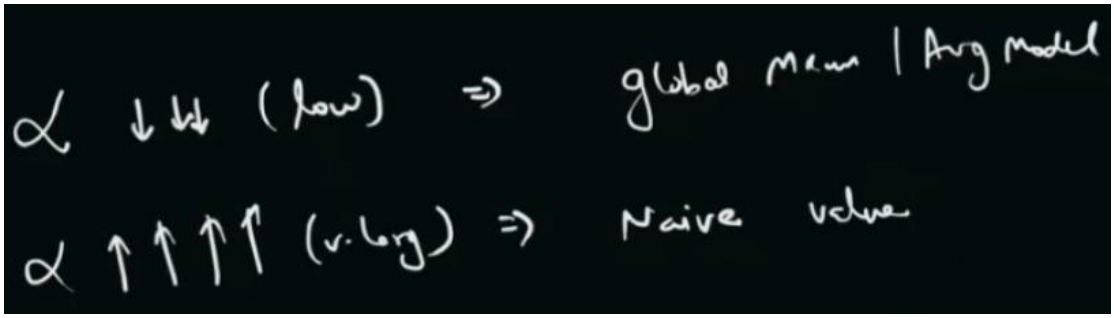


- The recommended starting value of α is:

$$\frac{1}{2*seasonality}$$



SES does not give very accurate prediction it only gives the **current level**



Simple Exponential Smoothing (SES)

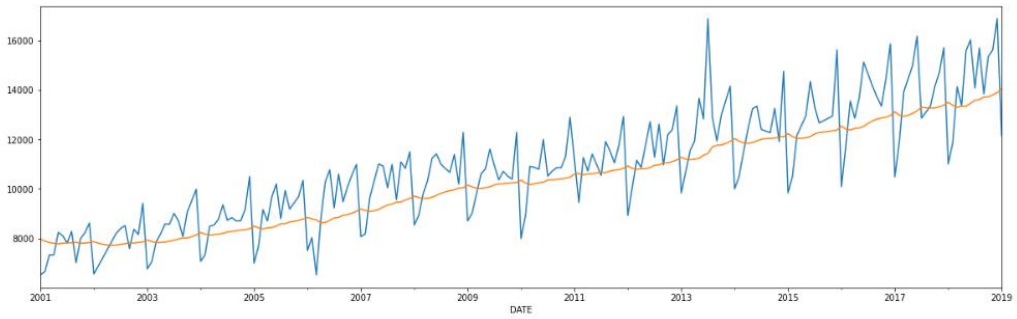
In [62]:
```
model = sm.tsa.SimpleExpSmoothing(mobile_sales['Sales'])
model.fit(smoothing_level = 0.05).fittedvalues
```

```
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency MS will be used.
  warnings.warn('No frequency information was'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\holtwinters\model.py:427: FutureWarning: After 0.13 initialization mus
t be handled at model creation
  warnings.warn(
```

Out[62]:
```
DATE
2001-01-01     7961.102707
2001-02-01     7889.153571
2001-03-01     7827.395893
2001-04-01     7802.626098
2001-05-01     7779.094793
                  ...
2018-09-01    13711.033147
2018-10-01    13717.381489
2018-11-01    13799.062415
2018-12-01    13889.859294
2019-01-01    14039.324329
Length: 217, dtype: float64
```

In [63]:
```
mobile_sales.Sales.plot()
model.fit(smoothing_level = 0.05).fittedvalues.plot()
```

Out[63]:
```
<AxesSubplot:xlabel='DATE'>
```



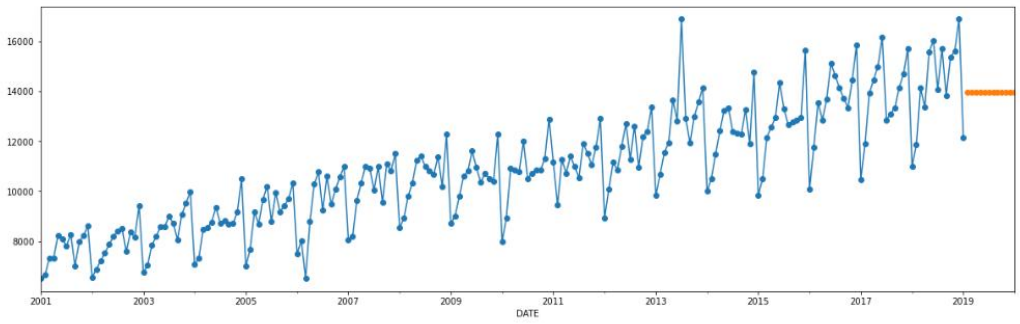
Unlike the moving averages, it does not have the offset at the beginning and end, because this method is initialized properly.

In [66]:
```
model = sm.tsa.SimpleExpSmoothing(mobile_sales.Sales).fit(smoothing_level=0.05)
pred = model.forecast(12)
model.forecast(12)
```

```
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency MS will be used.
  warnings.warn('No frequency information was'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\holtwinters\model.py:427: FutureWarning: After 0.13 initialization mus
t be handled at model creation
  warnings.warn(
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:132: FutureWarning: The 'freq' argument in Timestamp
is deprecated and will be removed in a future version.
  date_key = Timestamp(key, freq=base_index.freq)
```

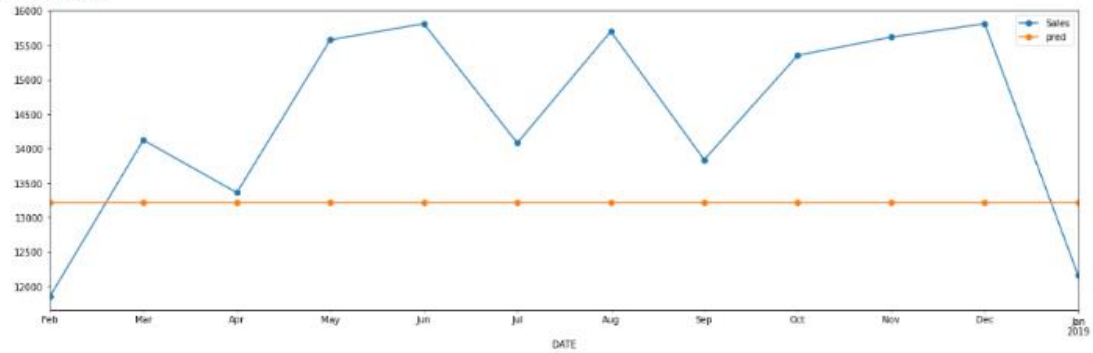
Out[66]:
```
2019-02-01    13945.358113
2019-03-01    13945.358113
2019-04-01    13945.358113
2019-05-01    13945.358113
2019-06-01    13945.358113
2019-07-01    13945.358113
2019-08-01    13945.358113
2019-09-01    13945.358113
2019-10-01    13945.358113
2019-11-01    13945.358113
2019-12-01    13945.358113
2020-01-01    13945.358113
Freq: MS, dtype: float64
```

```
In [65]: mobile_sales.Sales.plot(label='original', style='-o')
         pred.plot(label='ses_pred', style='-o')
Out[65]: <AxesSubplot:xlabel='DATE'>
```



The forecast plot of this model on the sales test data is given below:

```
MAE : 1621.347
RMSE : 1830.599
MAPE: 0.109
```



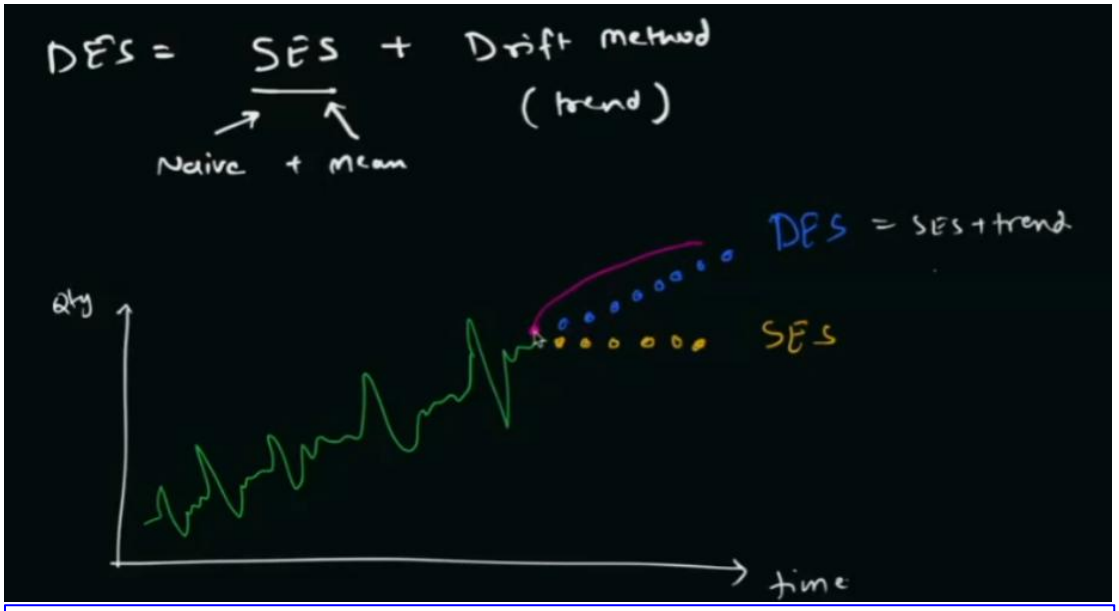
- The advantage of the above forecast is that the **level** of the forecasted values is right.
- However, the forecast is a completely straight line This is because we don't have the previous actual value available for horizon > 1. So the current forecast is used for all the next values.
- The prediction is a straight line, but the error is 10% which is less than the error of the moving average.
- The higher the value of α **(i.e, nearer to 1)** the forecast becomes more sensitive to the latest observations.
- The lower the value of α **(i.e, nearer to 0)** the forecast will be less sensitive to the latest observations.
- Disadvantage of this model is that it is missing both trend and seasonality
- Now, we have the right levels, if we can predict the trend and seasonality right, we should get a good forecast.

Day 3

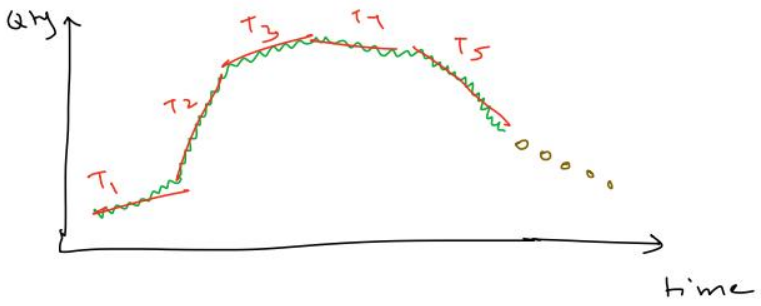
9.2 Double Exponential Smoothening (DES)

This method combines

DES = (Naive + Mean + Drift Trend)



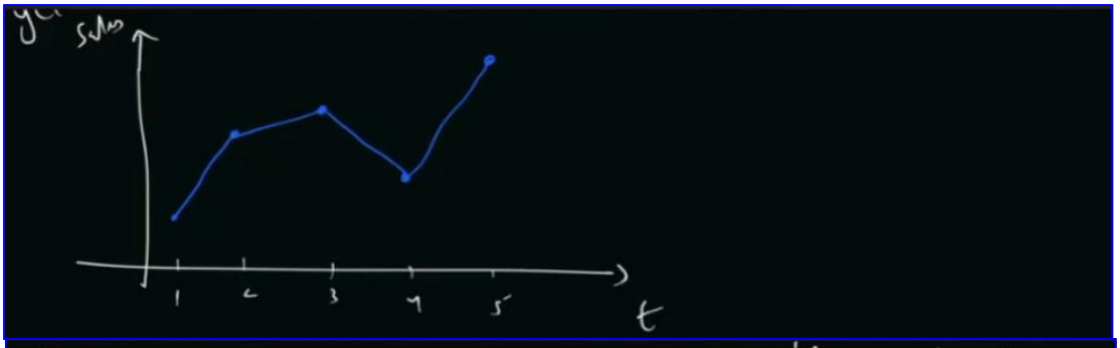
- The shortcomings of SES model, it doesn't capture the trend and only gives one unique value.
- In this method, we incorporate the trend of the entire time series in the SES formulation in order to forecast future values and we will have to provide weights to the trend value also.
- This method is used when there is only a trend present in the data.
- The weights are assigned to the trend value also and this forms an exponentially decaying series. Hence this is called Double Exponential smoothing (**aka Holt's method**). So basically, we are doing exponential smoothing on trend too.

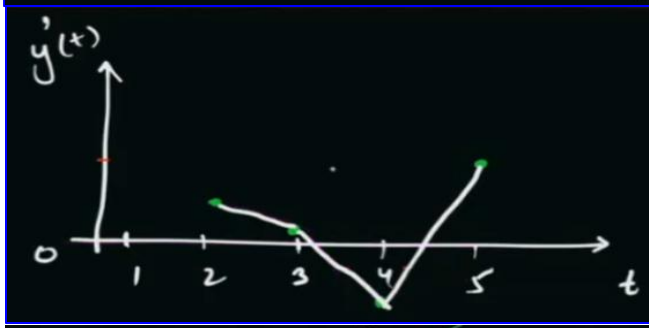


Trend.

↳ growth

Slope (rate of change) $y_2 - y_1 \Rightarrow \Delta y$

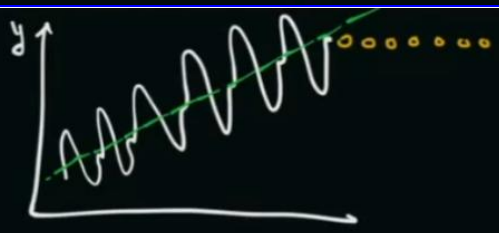




$y'(t) = \text{slope}(t)$

$\frac{\Delta y}{\Delta t} = \frac{y_2 - y_2}{1}$

y.diff()



① SES → reasonable level of y

② Reasonable estimate of trend?

(a) take avg. trend so far.

(b) take latest trend (instantaneous trend)

"Combination of both"

Exponential smoothening on "trend" as well.

Time Series

T.S → SES

take Δy T.S. → DES

Δy → trend

→ simultaneously

- There are two components present in this method:
  - Level: Captures the short-term average value.
  - Trend: Captures the trend.
- The formulation of DES is as follows:

$$\hat{y}_{t+h} = l_t + hb_t$$

where

$$l_t = \alpha y_t + (1-\alpha)(l_{t-1} + b_{t-1})$$

l_t is called as the **level** of time series at time t.

- Calculation of the trend value:
  - Slope of a curve for Δt=1 is given as

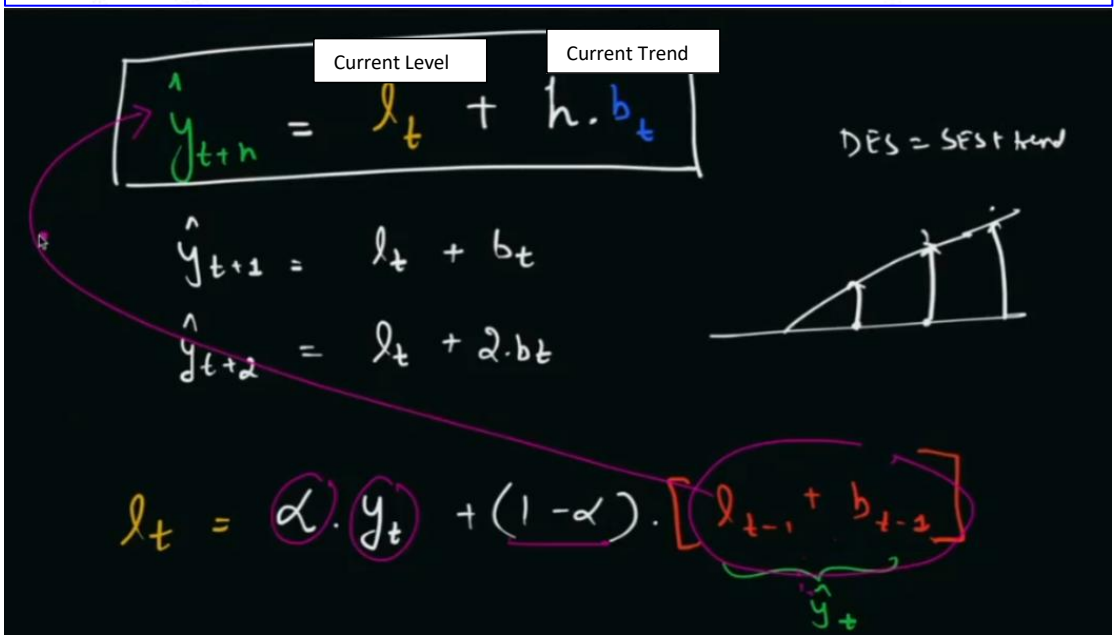
$$\frac{\Delta y}{\Delta t} = y_t - y_{t-1}$$

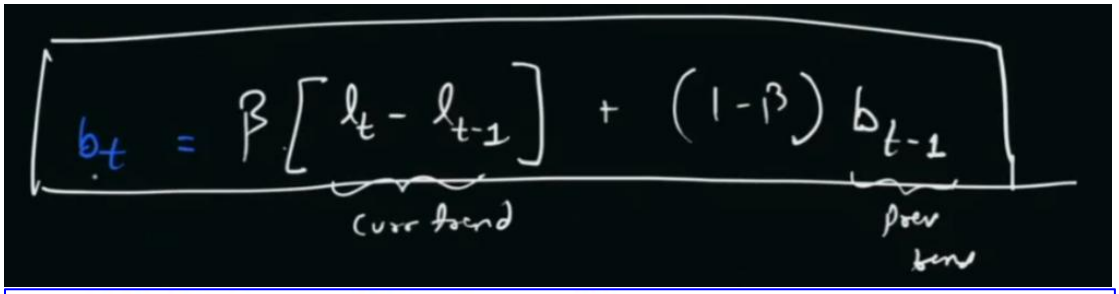
This slope value is actually equal to the trend of the series.
By plugging this in, we get,

$$b_t = \beta * (l_t - l_{t-1}) + (1-\beta) * b_{t-1}$$

where, $(l_t - l_{t-1})$ is representing the current slope of the curve and b_{t-1} represents the previous slope.

- The first smoothing parameter α corresponds to the level series.
- The second smoothing parameter β corresponds to the trend series.
- β is a parameter that needs to be tuned while training the model.



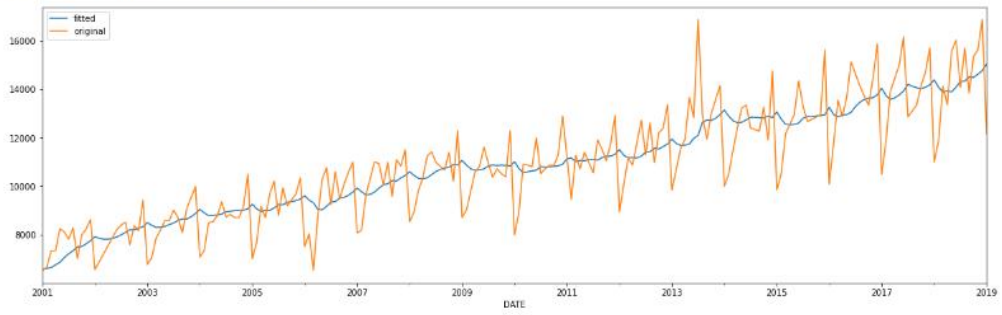


12.Double Exponential Smoothening (DES)

In [62]:
```
model = sm.tsa.ExponentialSmoothing(mobile_sales.Sales, trend='add').fit()

model.fittedvalues.plot(label='fitted')
mobile_sales.Sales.plot(label='original')
plt.legend()
```

```
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency MS will be used.
  warnings.warn('No frequency information was'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\holtwinters\model.py:427: FutureWarning: After 0.13 initialization mus
t be handled at model creation
  warnings.warn(
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\holtwinters\model.py:920: ConvergenceWarning: Optimization failed to c
onverge. Check mle_retvals.
  warnings.warn(
```

Out[62]:
```
<matplotlib.legend.Legend at 0x2237cc56be0>
```

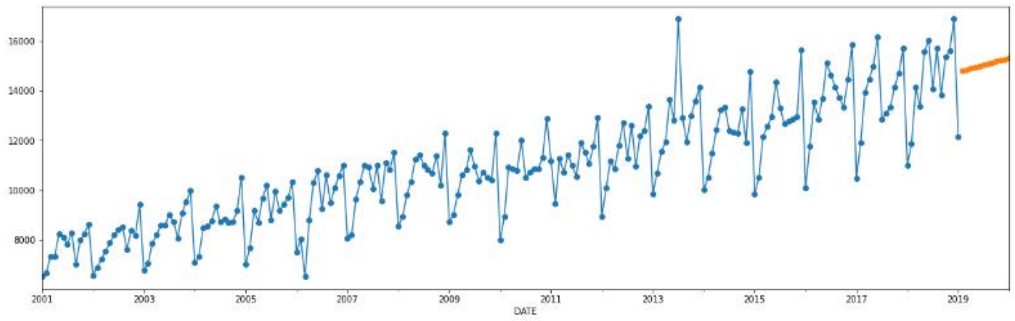


In [63]:
```
pred = model.forecast(12)
```

```
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:132: FutureWarning: The 'freq' argument in Timestamp
is deprecated and will be removed in a future version.
  date_key = Timestamp(key, freq=base_index.freq)
```

```
In [64]: mobile_sales.Sales.plot(label='original', style='-o') #Forecasted value for next 12 months
         pred.plot(label='des_pred', style='-o')
```

```
Out[64]: <AxesSubplot:xlabel='DATE'>
```



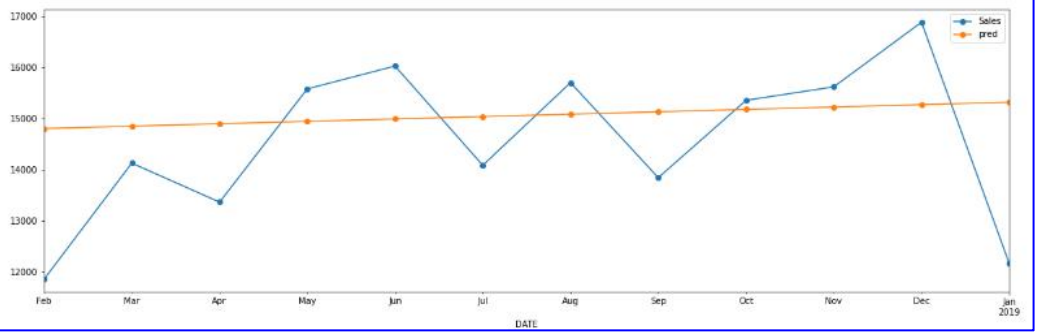

```
In [67]: test_x['pred'] = pred.values
         test_x
```

Out[67]:

DATE	Sales	pred
2018-02-01	11852.00	14799.594078
2018-03-01	14123.00	14846.332811
2018-04-01	13360.00	14893.071545
2018-05-01	15576.00	14939.810278
2018-06-01	16021.00	14986.549012
2018-07-01	14080.00	15033.287745
2018-08-01	15697.00	15080.026479
2018-09-01	13838.00	15126.765212
2018-10-01	15351.00	15173.503946
2018-11-01	15615.00	15220.242679
2018-12-01	16879.16	15266.981413
2019-01-01	12160.00	15313.720146

```
In [68]: test_x.plot(style='-o')

         performance(test_x['Sales'], test_x['pred'])
```

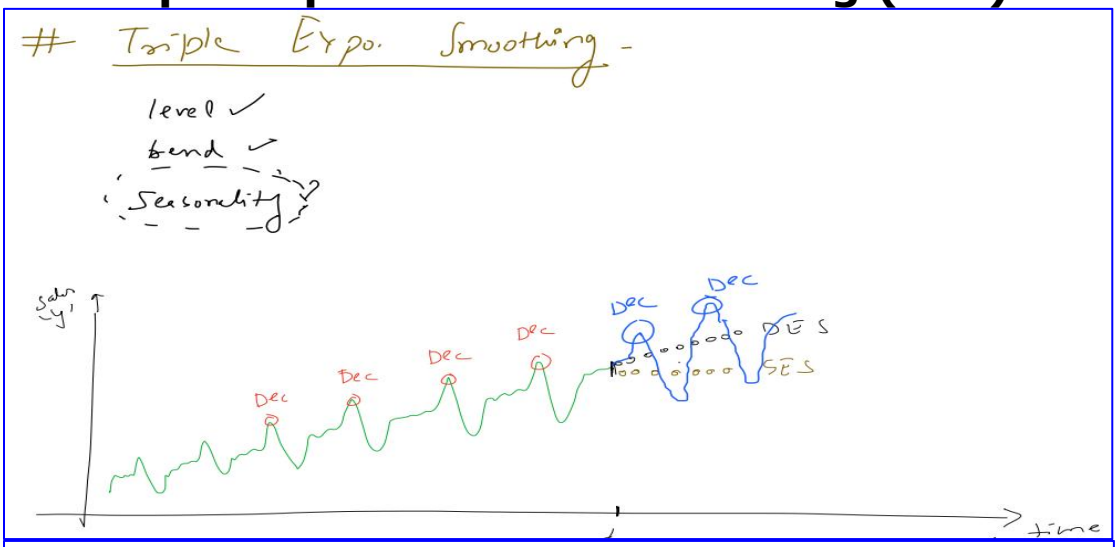
```
MAE : 1255.985
RMSE : 1547.024
MAPE: 0.093
```



Disadvantage

Disadvantage of this model is that it does not consider the seasonality of time series

9.3 Triple Exponential Smoothening (TES)



- Triple Exponential Smoothing is an extension of Double Exponential Smoothing that explicitly adds support for **seasonality** to the univariate time series. The seasonality value of the entire time series is also incorporated in this model.
- There are two components present in this method:
  - Level: Captures the short-term average value.
  - Trend: Captures the trend.
  - Seasonal: Captures the seasonality.

- Upon incorporating the seasonality, our equation becomes,

$$\hat{y}_{t+h} = l_t + hb_t + s_{t+h-m}$$

where, **m** -> frequency of the seasonality
Therefore, s_{t+h-m} is representing the smoothed seasonality.

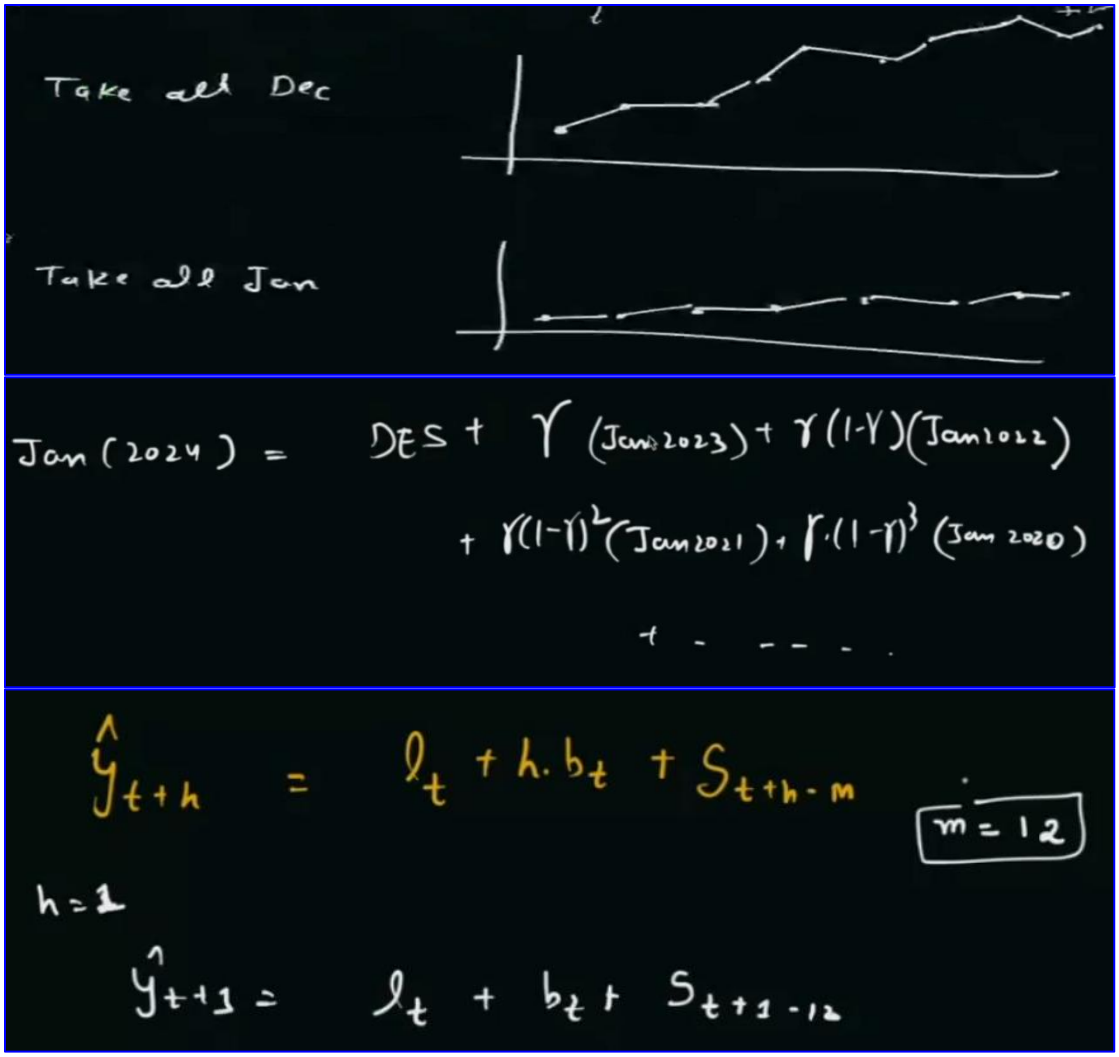
Also,

$$l_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1})$$

$$b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$

$$s_t = \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}$$

- The first smoothing parameter α corresponds to the level series.
- The second smoothing parameter β corresponds to the trend series.
- The third smoothing parameter γ corresponds to the seasonality series.



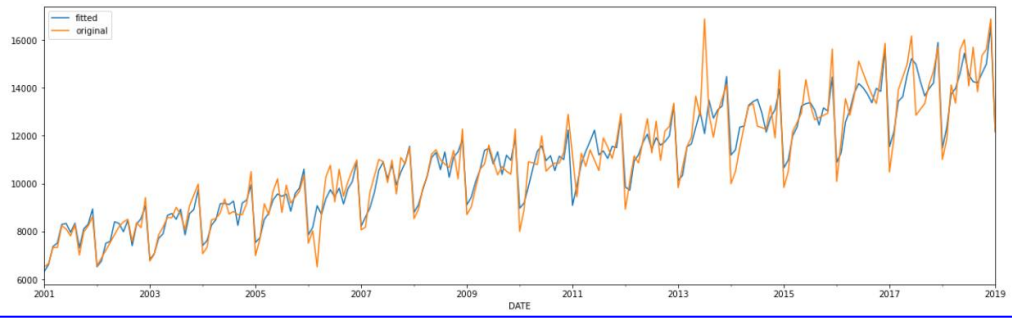
m --> seasonality component (here its 12 for 12 months)

Triple Exponential Smoothening (TES)

In [67]:
```
model = sm.tsa.ExponentialSmoothing(mobile_sales.Sales, trend='add', seasonal='add').fit() # Add seasonal component for TES

model.fittedvalues.plot(label='fitted')
mobile_sales.Sales.plot(label='original')
plt.legend()
```

```
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency MS will be used.
  warnings.warn('No frequency information was'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\holtwinters\model.py:427: FutureWarning: After 0.13 initialization mus
t be handled at model creation
  warnings.warn(
```

Out[67]: <matplotlib.legend.Legend at 0x1c5d86d43a0>

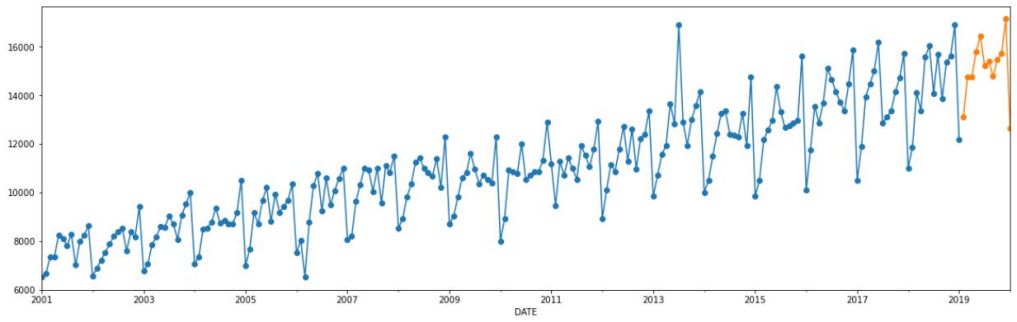


In [68]:
```
pred = model.forecast(steps=12)
```

```
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:132: FutureWarning: The 'freq' argument in Timestamp
is deprecated and will be removed in a future version.
  date_key = Timestamp(key, freq=base_index.freq)
```

In [69]:
```
mobile_sales.Sales.plot(label='original', style='-o')
pred.plot(label='tes_pred',style='-o')
```

Out[69]:
```
<AxesSubplot:xlabel='DATE'>
```



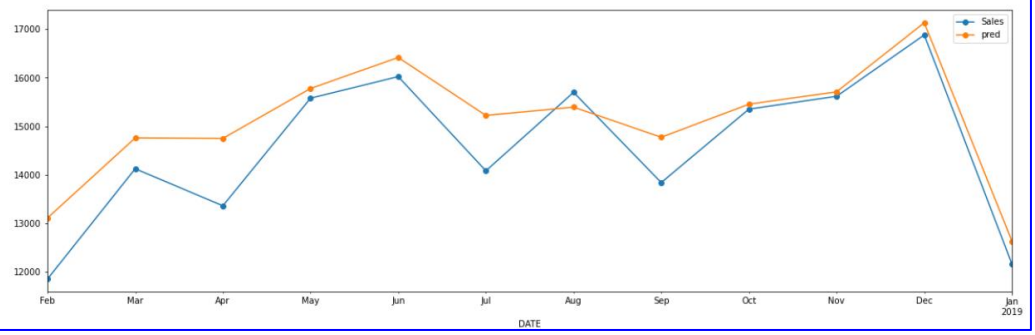
In [70]:
```
test_x['pred'] = pred.values
test_x
```

Out[70]:

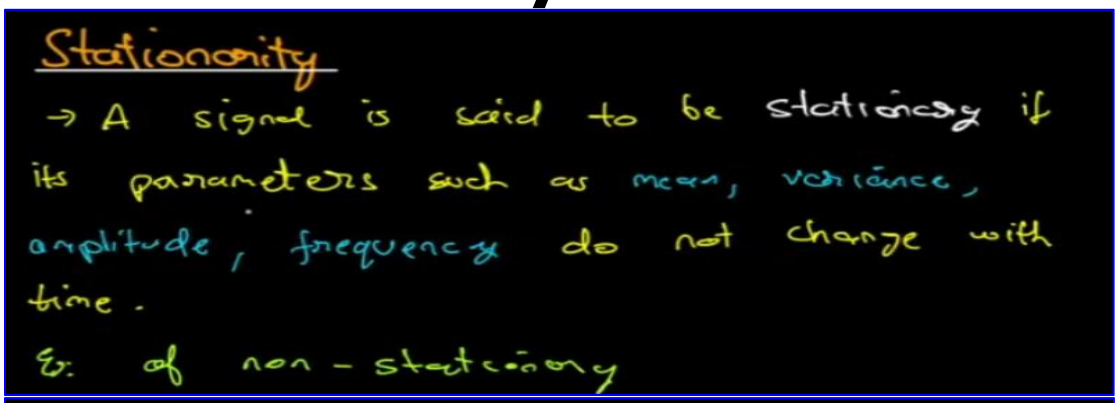
DATE	Sales	pred
2018-02-01	11852.00	13109.067567
2018-03-01	14123.00	14757.245607
2018-04-01	13360.00	14748.355030
2018-05-01	15576.00	15776.193855
2018-06-01	16021.00	16414.621444
2018-07-01	14080.00	15220.907092
2018-08-01	15697.00	15392.303588
2018-09-01	13838.00	14771.687544
2018-10-01	15351.00	15452.418164
2018-11-01	15615.00	15707.331430
2018-12-01	16879.16	17129.966433
2019-01-01	12160.00	12632.035463

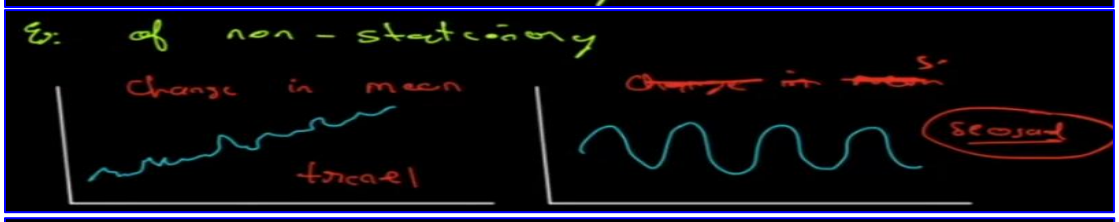
In [71]:
```
test_x.plot(style='-o')

performance(test_x['Sales'], test_x['pred'])
```

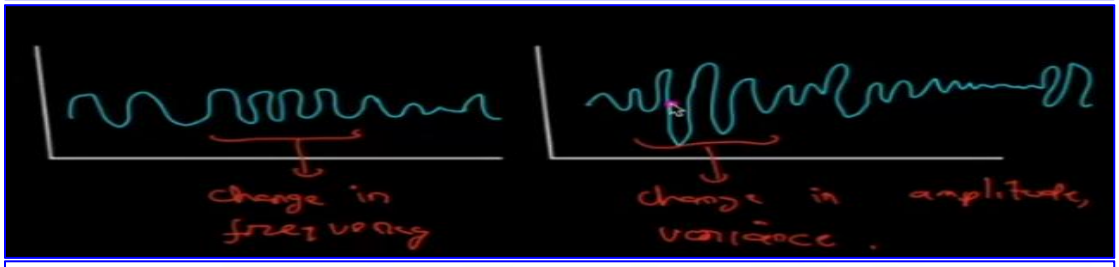
```
MAE : 597.447
RMSE : 745.869
MAPE: 0.044
```



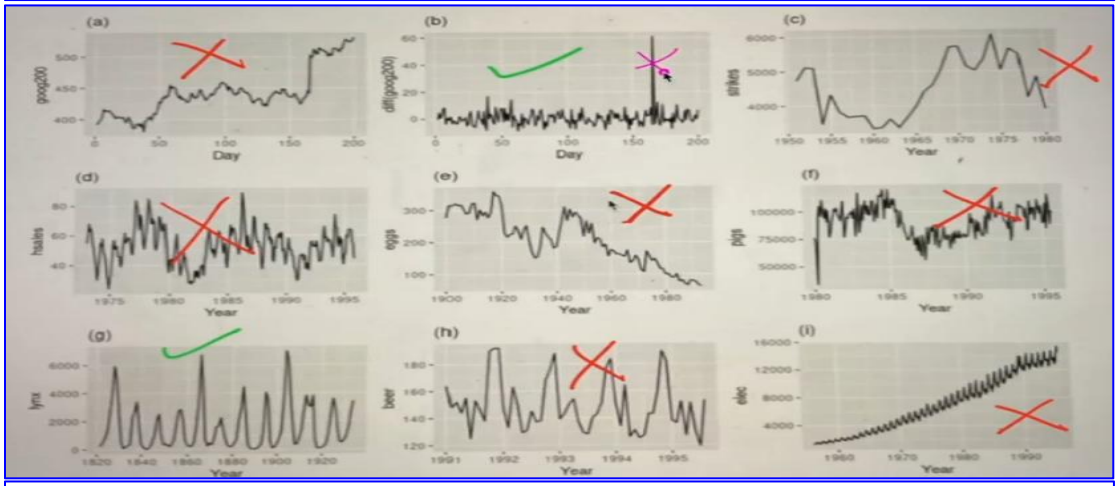
10. Stationarity







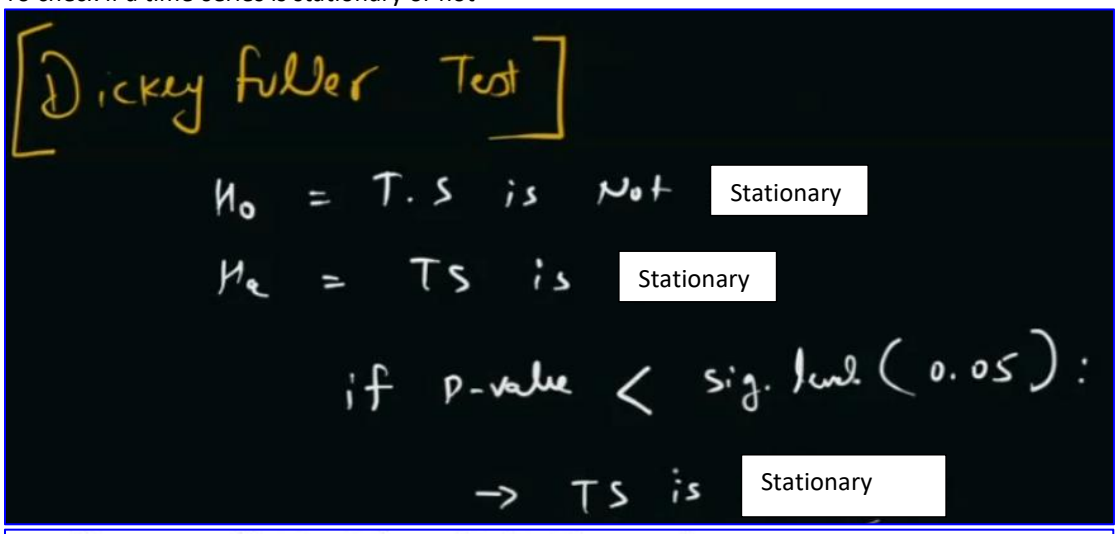
- **Definition:** Time series with constant mean, variance, amplitude, and frequency over time.
- **Non-Stationarity Indicators:** Presence of trends, seasonality.
- **Example:** Stationary heartbeat with consistent mean and standard deviation.
- **Characteristics:** No long-term predictable patterns.
- **Importance:** Many models require stationarity for accurate results.
- **Conversion:** Non-stationary series often transformed to stationary.
- **Assessment:** Stationarity is determined visually or with statistical tests like Dickey-Fuller.



- **Non-Stationary:** Plots a, c, e, f (due to trend or changing mean), d, h (due to seasonality), i (due to trend, seasonality, unstable variance).
- **Stationary:** Plot b (despite one outlier), plot g (assumed for modeling, irregular cyclic pattern).

11. Dickey Fuller Test

To check if a time-series is stationary or not

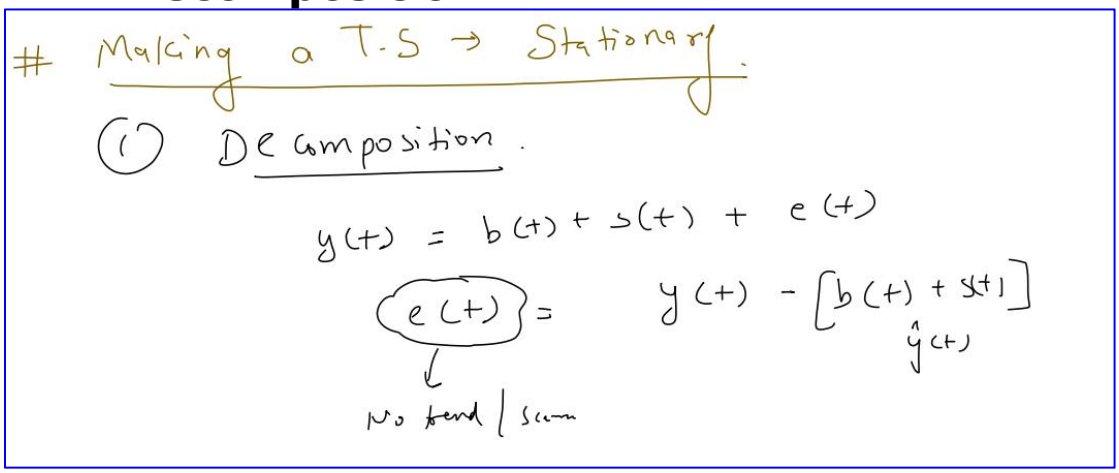


- **Purpose:** Tests stationarity in time series.
- **Hypotheses:**
  - H0: Time series is non-stationary.
  - H1: Time series is stationary.
- **Implementation:** sm.tsa.stattools.adfuller() in statmodels library.
- **Interpretation:** p-value < 0.05 indicates stationarity.

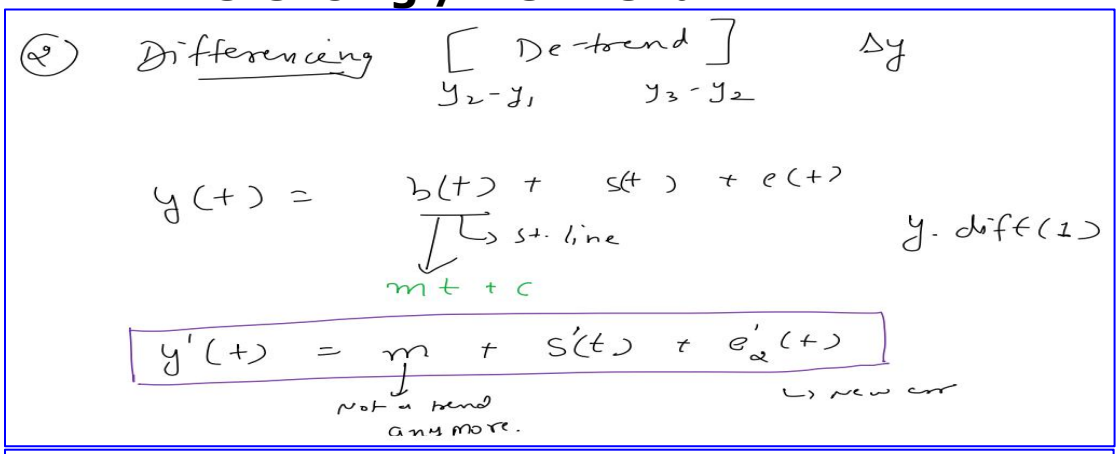
For ARIMA family , we need to convert a time-series to stationary time-series

12 . Converting a Time Series(TS) to Stationary TS

12.1 Decomposition

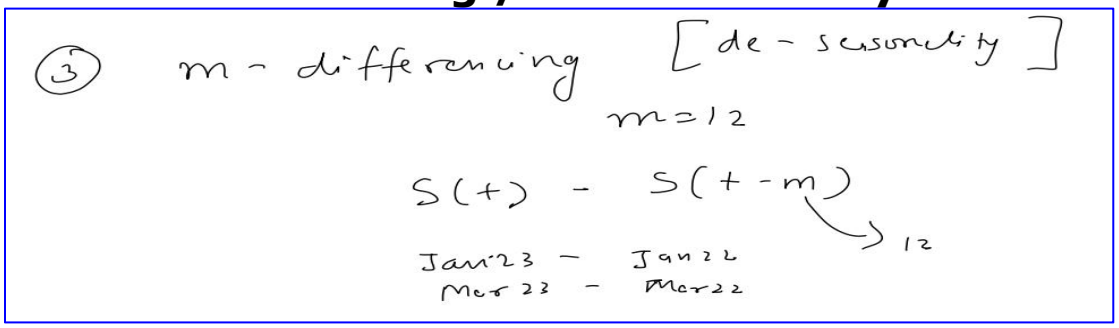


12.2 Differencing / De-Trend



- **Method:** Differencing the series (value(t) = observation(t) - observation(t-1)).
- **Library Function**: diff() in pandas.
- **Non-linear Trends:** This may require multiple differencing steps.

12.3 m-differencing / De-Seasonality



Deseasonalizing:

- **m-Differencing:** Subtracting observation at the current timestep from the one at the last seasonal period (value(t) = observation(t) - observation(t-m)).
- **Seasonality Period (m):** Determined by the data's seasonal cycle.

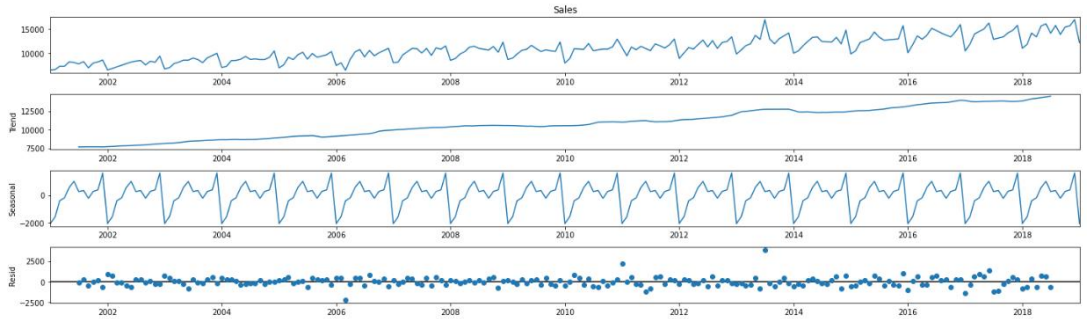
Process:

- **Detrend**: First, use differencing to remove the trend.
- **Deseasonalize:** Then, remove seasonality, potentially using m-differencing.

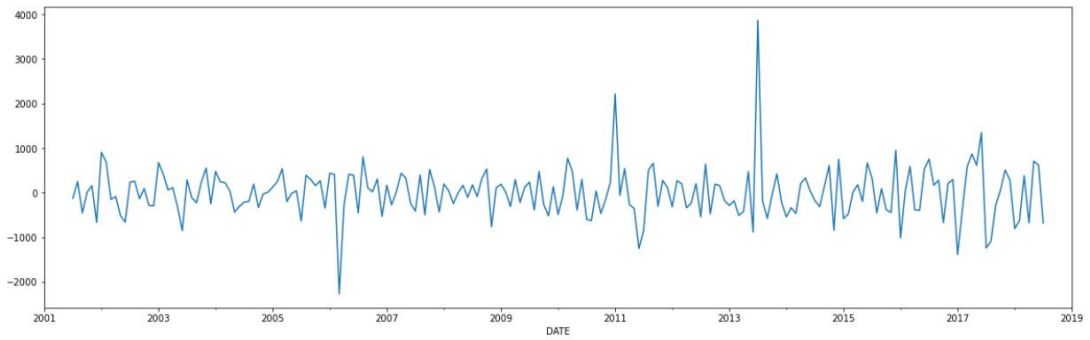
Converting to Stationary TS

```
model = sm.tsa.seasonal_decompose(mobile_sales['Sales'], model='additive')
```

```
model.plot();
```



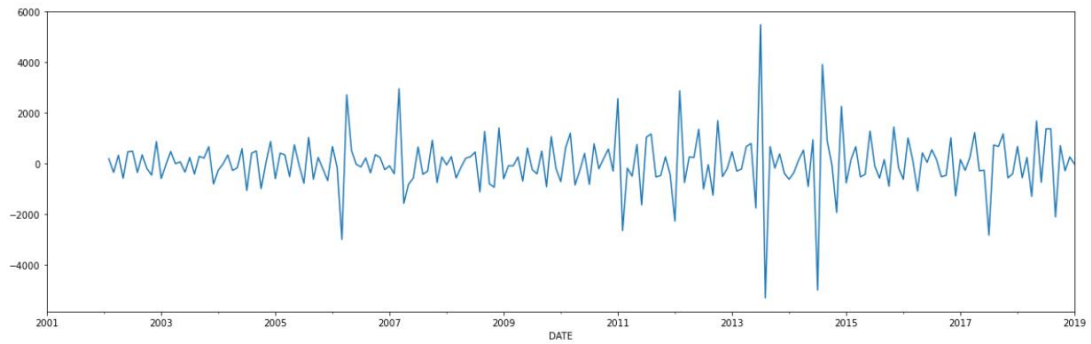

```
model.resid.plot();
```




```
adf_test(model.resid.dropna())
```

```
Sequence is stationary
```

```
# remove trend and seasonality.
mobile_sales.Sales.diff(1).diff(12).plot();
# first diff wil remove trend and second will remove seasonality(12 months here)
```

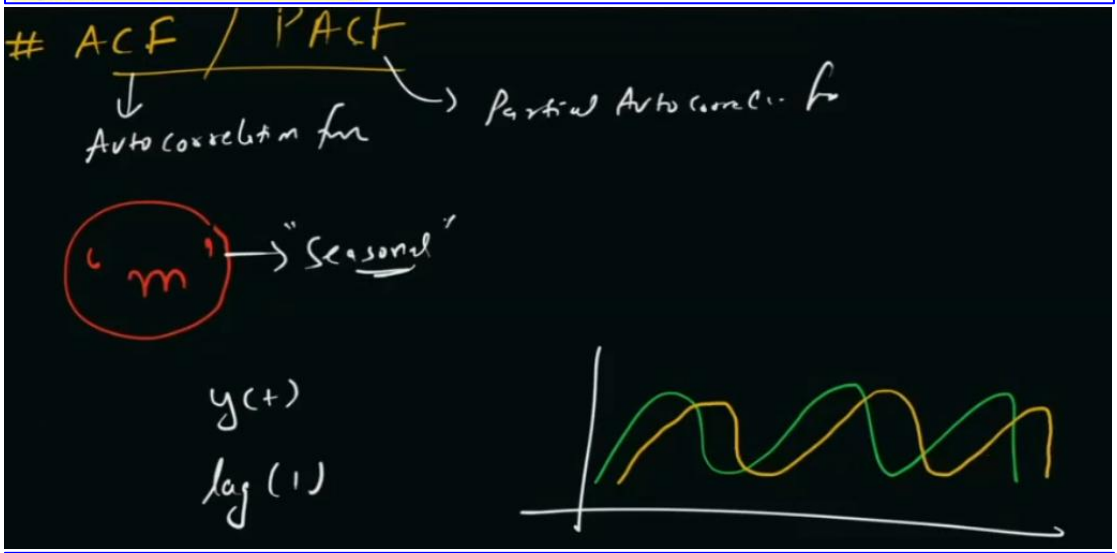



```
adf_test(mobile_sales.Sales.diff(1).diff(12).dropna())
```

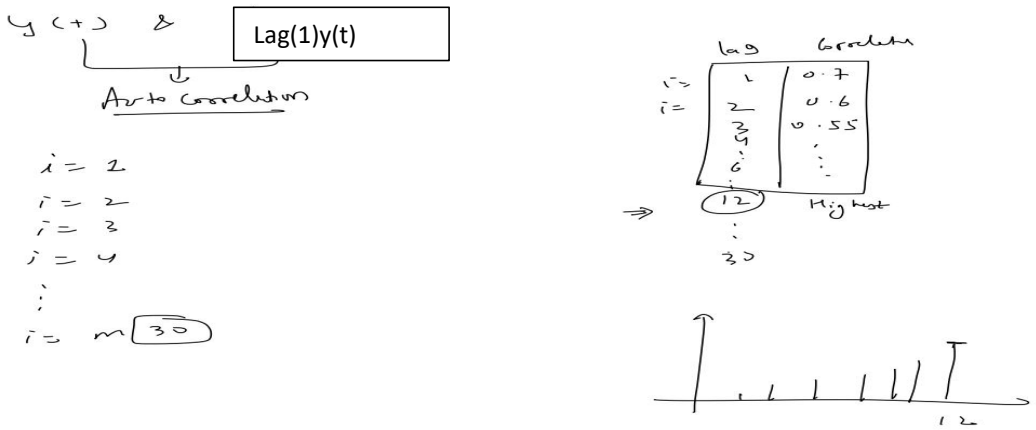
```
Sequence is stationary
```

13. AutoCorrelation Function (ACF)

Autocorrelation: Correlation of a time series with its lagged version; identifies optimal lag (m) where series overlap.



Autocorrelation Function (ACF): Shows direct and indirect correlation impacts; useful for spotting random series.



We will have the highest correlation when i=12 as the pattern will somewhat repeat for every 12 months
i=12 means the shift of 12 months ahead

Partial Autocorrelation Function (PACF): Shows unique correlation by removing indirect effects; helps in identifying direct relationships and seasonality.
ACF and PACF Plots: Reveal significant lags with correlations outside confidence intervals, indicating potential seasonality.
Usage: ACF applied to stationary series; PACF to original series for direct impacts.

AutoCorrelation Function (ACF)

```
lag = 1
np.corrcoef(mobile_sales.Sales[lag:], mobile_sales.Sales.shift(lag)[lag:])
```

```
array([[1.        , 0.79245871],
       [0.79245871, 1.        ]])
```

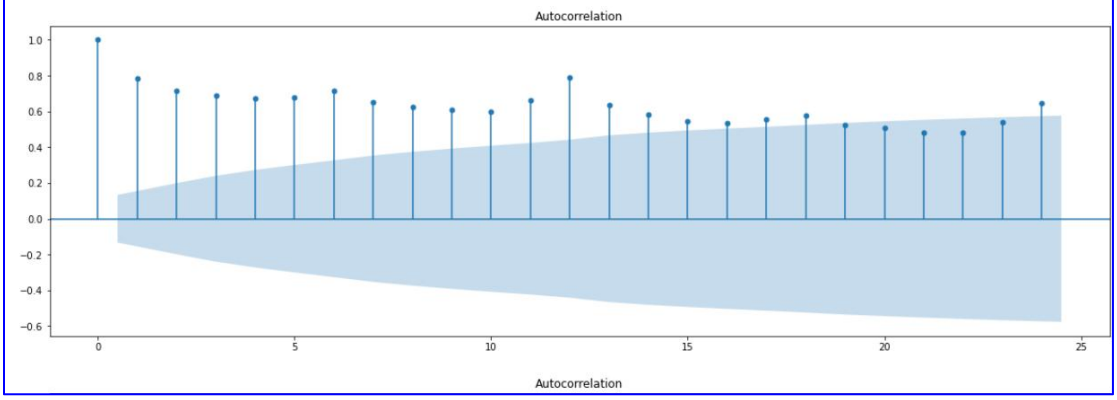
```
lag = 1
np.corrcoef(mobile_sales.Sales[lag:], mobile_sales.Sales.shift(lag)[lag:])[0,1]
```

```
0.7924587051003289
```

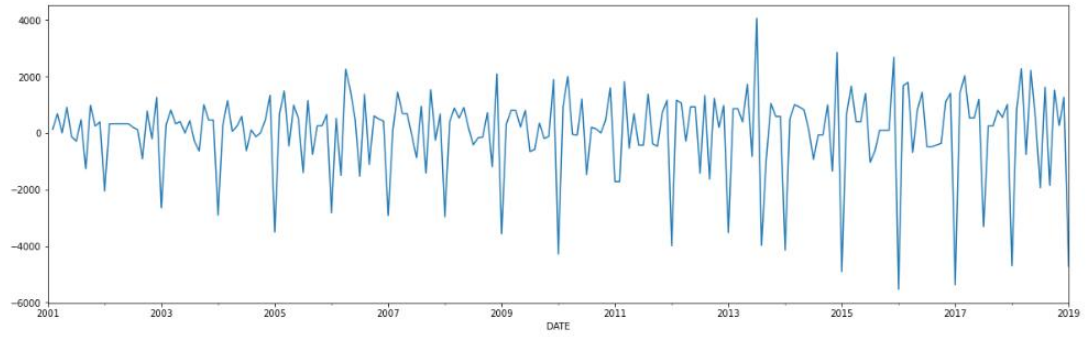
```
lag = 12
np.corrcoef(mobile_sales.Sales[lag:], mobile_sales.Sales.shift(lag)[lag:])
```

```
array([[1.        , 0.92824328],
       [0.92824328, 1.        ]])
```

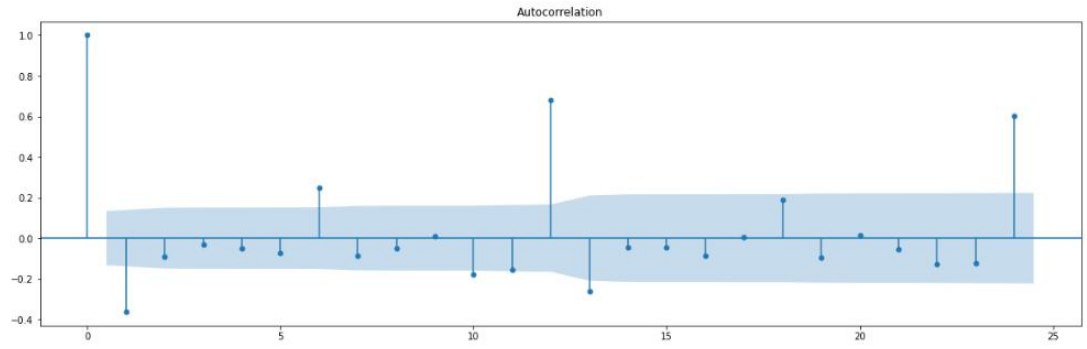
```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(mobile_sales.Sales)
```




```
mobile_sales.Sales.diff(1).plot();
```




```
plot_acf(mobile_sales.Sales.diff(1).dropna() );
```



Q1. Constants in forecasts Solved

Using hints is now penalty free Use Hint

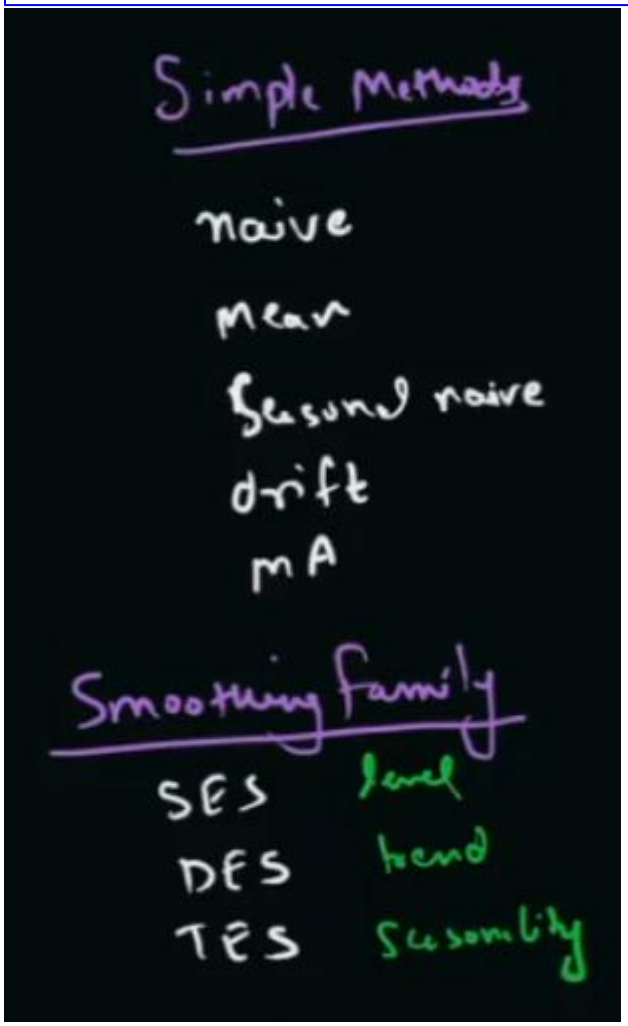
The number of smoothing constants used in the smoothing method of forecasting which considers seasonality in addition to trend is:

In the method triple exponential smoothing we can use constants for adjusting forecast error for

1. previous observation(alpha)
2. trend(beta) and,
3. seasonality(gamma).

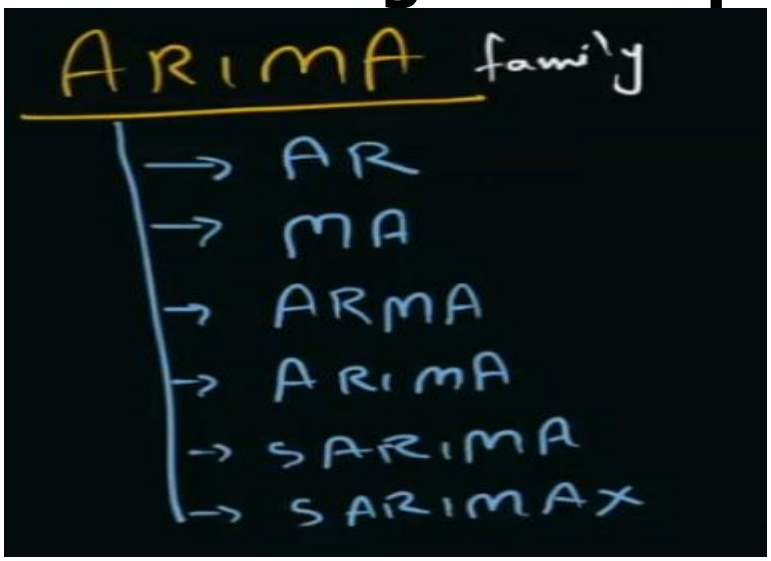
Correlation vs. Causation:

- **Correlation:** Relationship between two variables without implying cause.
- **Causation:** One variable directly affects another.
- **Example:** Ice cream sales and drownings correlate due to temperature, not causation.
- **Confounding Variable:** A third element influencing both correlated variables.
- **Forecasting Use:** Correlation useful even without causality.
- **Misinterpretation Risk:** Assuming causality from correlation can lead to incorrect conclusions.
- **Model Improvement:** Understanding causality helps identify better predictive features.



Day 4

14. ARIMA Family of Forecasting Techniques



14.1 Auto Regressive (AR)

AutoRegression (AR(p)):
- Utilizes past values for forecasting in stationary time series.
- Linear combination of past values as features for Linear Regression.
- **Understanding p:** The hyperparameter p in AR(p) specifies the number of lagged observations included in the model. It reflects the extent to which past values influence future values, chosen based on the PACF plot to capture significant lag correlations.

Difference from SES:
- **SES**: Exponentially decaying weights, single hyperparameter (α).
- **AR:** Weights are learned through the model, with the hyperparameter being the lag order p, which represents the number of past values considered for prediction.

Pre-requisites:
- Use PACF to avoid feature correlation; high PACF at lag k suggests correlation with future value.
- Choose p based on PACF plot; significant lags indicate model order.

AR(0): No dependence, equivalent to white noise.

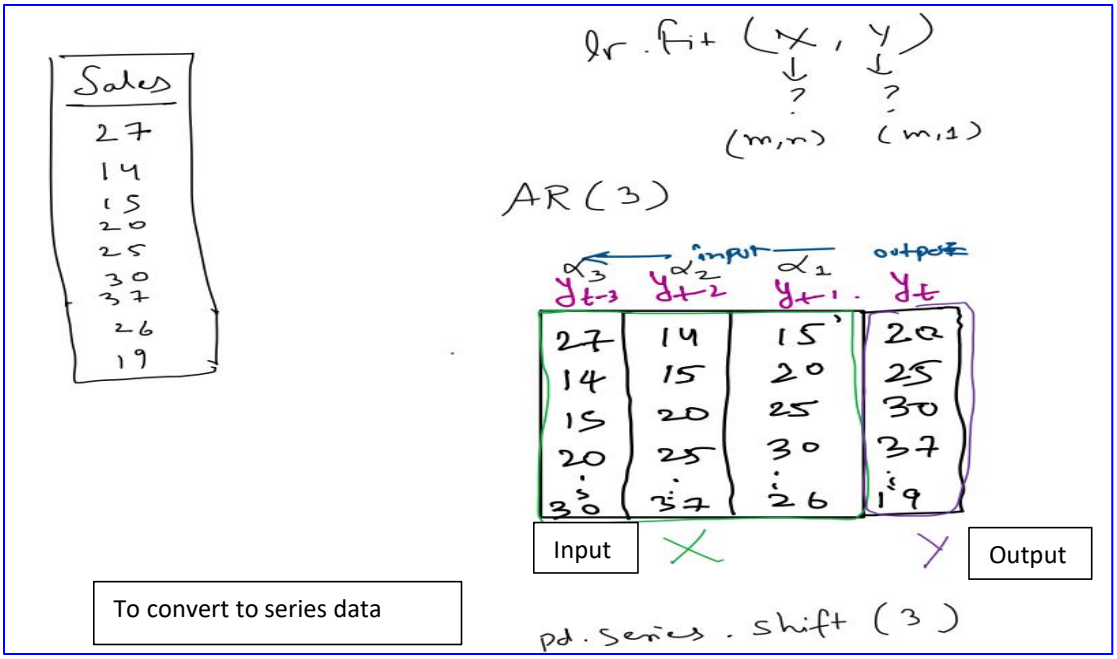
Model Order Selection:
- Based on PACF spikes; ACF decays slowly in AR processes.
- p determined by significant PACF lag values.

$$\hat{y}_t = \alpha_1 \cdot y_{t-1} + \alpha_2 \cdot y_{t-2} + \alpha_0 \quad \text{AR(2)}$$

$$\hat{y}_t = \alpha_1 \cdot y_{t-1} + \alpha_2 \cdot y_{t-2} + \alpha_3 \cdot y_{t-3} + \alpha_0 \quad \text{AR(3)}$$

$$\hat{y}_t = \alpha_1 \cdot y_{t-1} + \alpha_2 y_{t-2} + \alpha_3 y_{t-3} + \cdots + \alpha_p \cdot y_{t-p} + \alpha_0$$

Apply Linear Reg learn α's



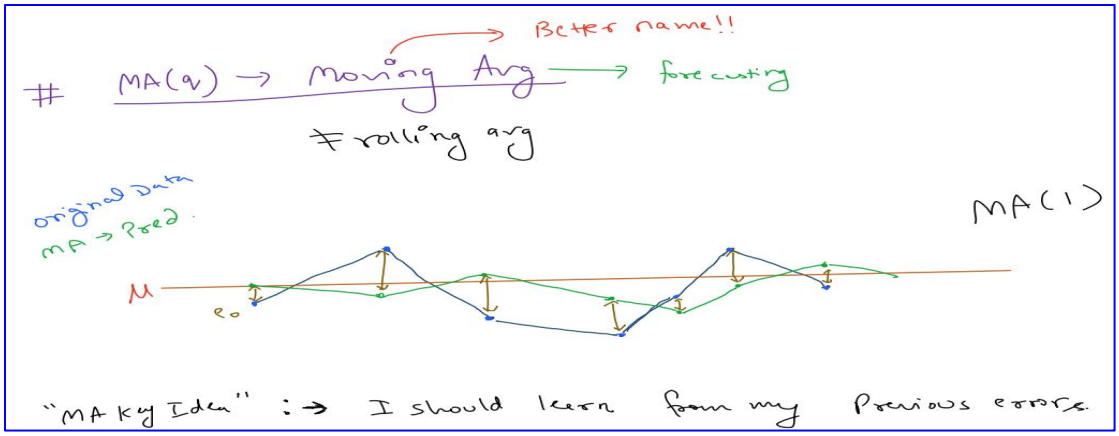
Here 3 is the window size. It is hyper parameter

14.2 Moving Average (MA)

NOTE: This is not the moving average we learned earlier (rolling average)

- **Concept:** Utilizes past forecast errors (residuals) for predicting current time period values.
- **Key Feature:** Creates unique feature from error of past value from mean.

- **Formulation:** Extends to order q, considering past q errors for current prediction.
- **Stationarity:** MA model inherently stationary, as observations are weighted averages of past errors.
- **Hyperparameter q:** Represents the order of the MA model, indicating the number of past errors considered. The optimal value of q is determined experimentally, often through model validation techniques.
- **Identification:**
  - ACF used to suggest order q; sharp cut-off in ACF after lag q indicates model order.
  - PACF decays more slowly in MA processes.
- **Difference from AR:**
  - AR focuses on past values, MA on past errors.
  - MA addresses correlated noise ignored by AR.



MA(1)

For 1st point

(y0)-->first point(y0) it predicts as mean value.

Then it checks if the error i.e (pred - actual value) .

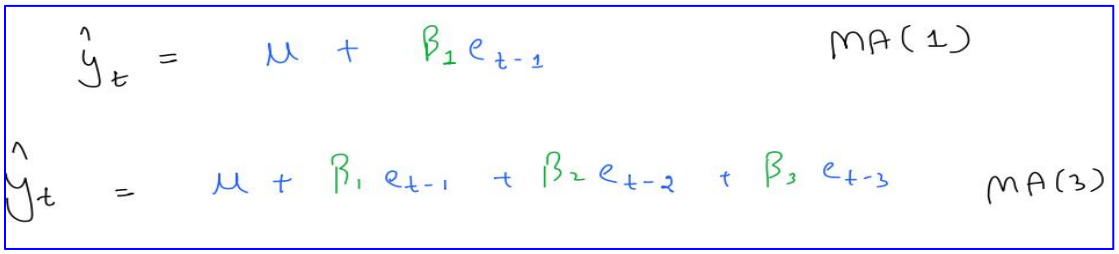
Since error here is +ve, then it will predict the next point little low

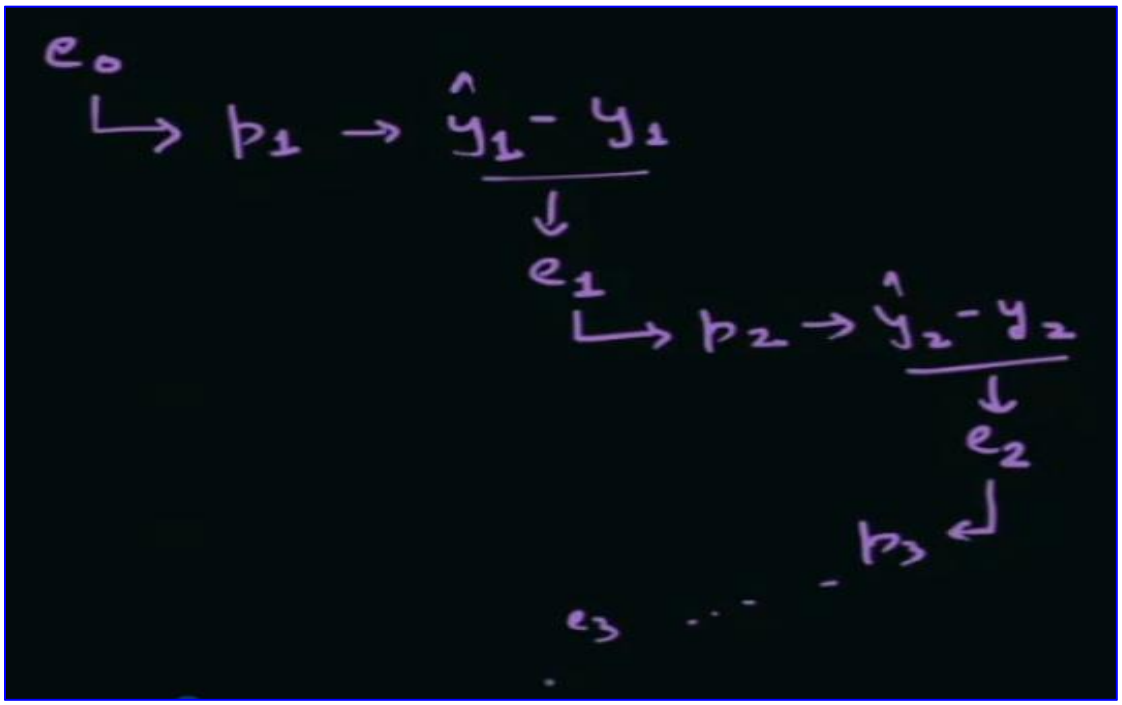
For 2nd point

(y1)--> It will predict little less than the last prediction(y0) based on previous error

Then it checks if the error i.e (pred - actual value) .

Since error here is -ve, then it will predict the next point little higher





14.3 Auto Regressive Moving Average (ARMA(p,q))

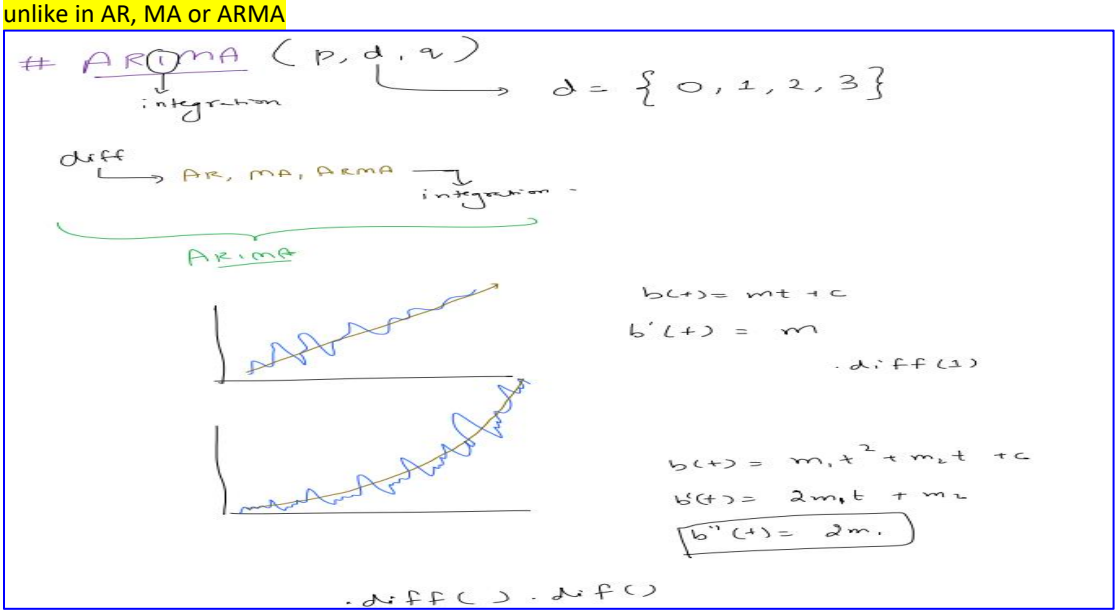
It is a combination AR and MA model

- **Combination:** Integrates AR and MA techniques.
- **Formulation:** ARMA(p, q) with p as AR order, q as MA order.
- **Coefficients:** α1, α2, ..., αp for AR; m1, m2, ..., mq for MA.
- **Hyperparameters:** Orders p and q; not necessarily equal.
- **Stationarity Requirement:** Cannot handle non-stationary series.
- **Identification:**
  - AR: PACF plot for lag terms.
  - MA: ACF plot for error terms.
- **ACF and PACF in ARMA:**
  - ACF: Sharp cutoff after lag q.
  - PACF: Sharp cutoff after lag p.
- **Limitations:**
  - Unsuitable for non-stationary or seasonal data.
  - Differencing may lead to data loss and scale changes, requiring retransformation for forecasts.

14.4 Auto Regressive Integration Moving Average (ARIMA(p,d,q))

d indicates the number of we need to differentiate to convert the TS to stationary TS

ARIMA model does the differentiation and integration in itself so we do not need to explicitly do it unlike in AR, MA or ARMA



In most of the scenario d is 1 , sometimes 2 and rarely 3

- **Purpose:** Combines differencing (to achieve stationarity), AR, and MA for forecasting non-stationary time series.
- **Notation:** ARIMA(p, d, q)
  - p: Order of the AR part.
  - d: Degree of differencing.
  - q: Order of the MA part.

- **Process:**
  - 1. Differencing: To remove trend and make series stationary.
  - 2. ARMA Application: For approximating stationary series.
  - 3. Integration: Restores trend for final forecast.
- **Parameter Selection:**
  - Use grid search or AIC/BIC for optimal p, d, q.
  - AIC/BIC help compare model quality; lower values preferred.
- **AIC (Akaike Information Criteria):**
  - Balances model fit and simplicity.
  - Prefers models with higher log-likelihood and fewer parameters.
- **Limitations:**
  - May not capture seasonality effectively.

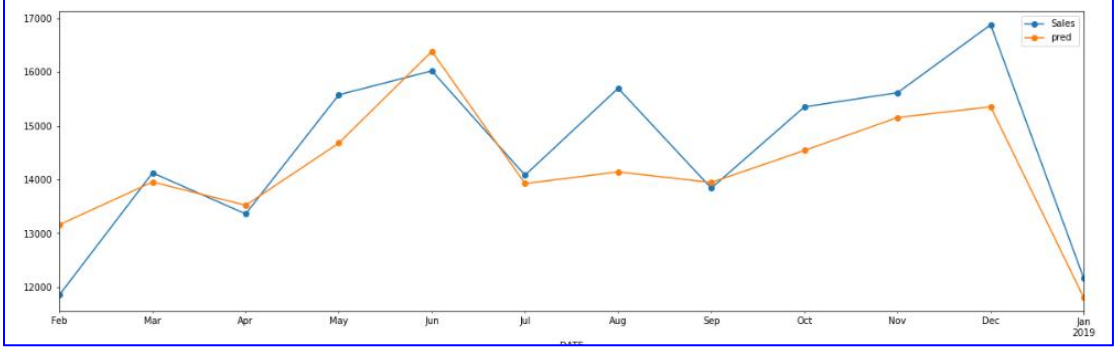
ARIMA(p,q,d)

```
# orginal non-stationary T.S
model = SARIMAX(train_x.Sales, order=(25, 1, 15))
model = model.fit(disp=False)

test_x['pred'] = model.forecast(steps=12)
```

```
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency MS will be used.
  warnings.warn('No frequency information was'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency MS will be used.
  warnings.warn('No frequency information was'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\statespace\sarimax.py:966: UserWarning: Non-stationary starting autore
gressive parameters found. Using zeros as starting parameters.
  warn('Non-stationary starting autoregressive parameters'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\statespace\sarimax.py:978: UserWarning: Non-invertible starting MA par
ameters found. Using zeros as starting parameters.
  warn('Non-invertible starting MA parameters found.'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\base\model.py:566: ConvergenceWarning: Maximum Likelihood optimization fai
led to converge. Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```

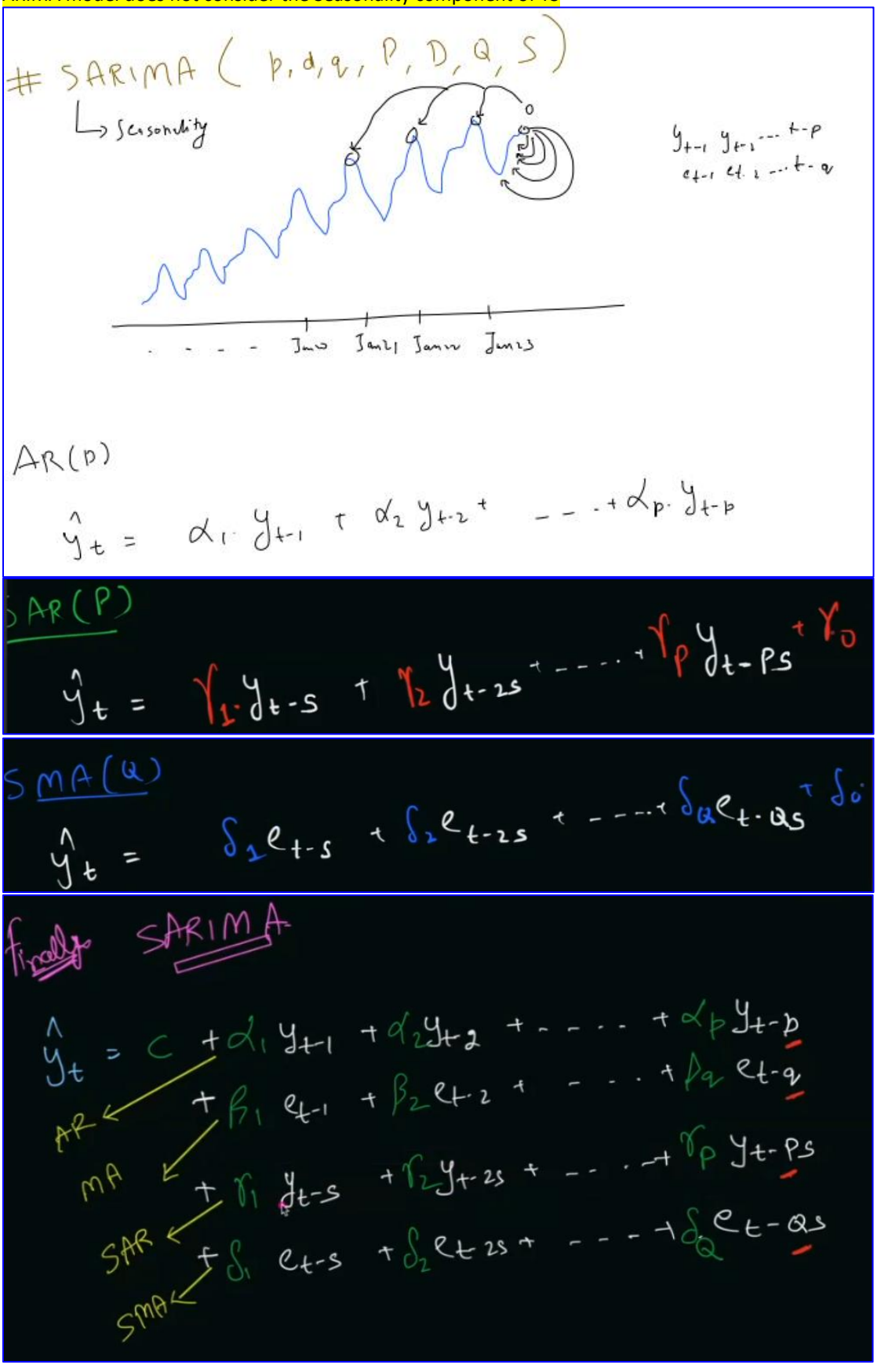
```
test_x.plot(style='-o')

performance(test_x['Sales'], test_x['pred'])
```

```
MAE : 656.335
RMSE : 841.217
MAPE: 0.045
```



14.4 SARIMA(p,d,q,P,D,Q,S)

ARIMA model does not consider the Seasonality component of TS



- **Purpose:** Extends ARIMA to explicitly model seasonality.
- **Parameters:** SARIMA(P, D, Q, p, q, d, s)
  - **P, D, Q:** Seasonal AR, differencing, and MA orders.
  - **p, q, d:** Non-seasonal AR, MA, and differencing orders.
  - **s:** Seasonality period.
- **Seasonality (s):** Identifies repeating patterns over s periods; determined via ACF/PACF or tuning.
- **Hyperparameters:**
  - **P:** Seasonal AR order; impacts forecasting based on seasonal lags.
  - **Q:** Seasonal MA order; models errors from seasonal lags.
  - **D:** Seasonal differencing; removes seasonality before modeling.
- **Modeling Seasonality:**
  - P and Q enable AR and MA on data from past seasonal periods (e.g., 12, 24 months).
  - D applies seasonal differencing, enhancing stationarity with respect to seasonality.

- **Forecasting:**
  - Combines AR, MA, differencing, and seasonal components for accurate prediction.
  - Utilizes ACF and PACF for identifying appropriate seasonal lags.
- **Limitation:** Captures single seasonality type per model.
- **Identification:**
  - Seasonal spikes in ACF/PACF indicate ARIMA(P, D, Q) components.
  - SARIMA efficiently models and forecasts time series with complex seasonal patterns.

```python
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Sample implementation for monthly sales data
model = SARIMAX(data['Sales'], order=(p,d,q), seasonal_order=(P,D,Q,s))
results = model.fit()

# Forecasting
forecast = results.forecast(steps=12) # Forecasting the next 12 months
```

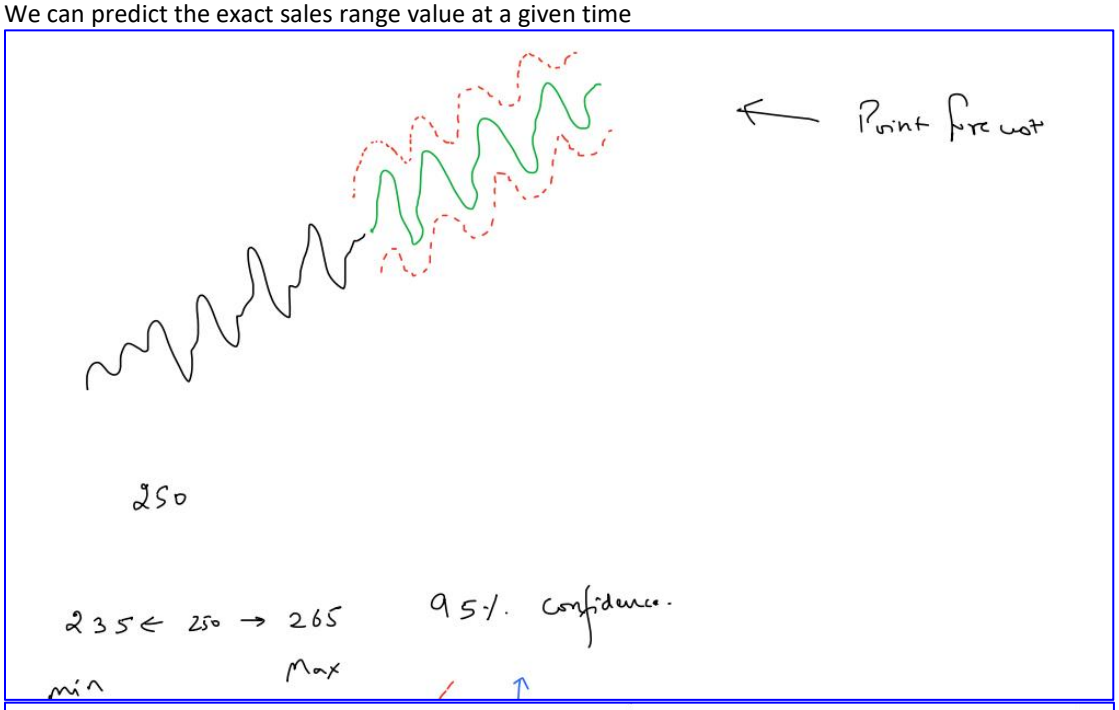
- **data['Sales']:** Time series data.
- **order=(p,d,q):** Non-seasonal parameters based on data analysis.
- **seasonal_order=(P,D,Q,s):** Seasonal parameters, with s, typically based on the data's seasonality pattern (e.g., 12 for monthly data).

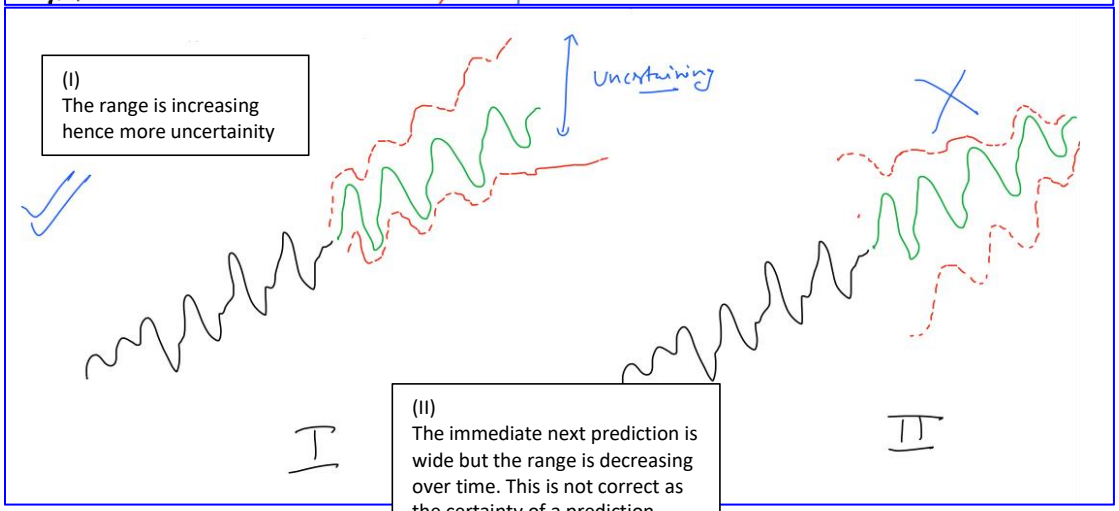
Testbook: https://otexts.com/fpp2/

Day 5

15. Point Forecast

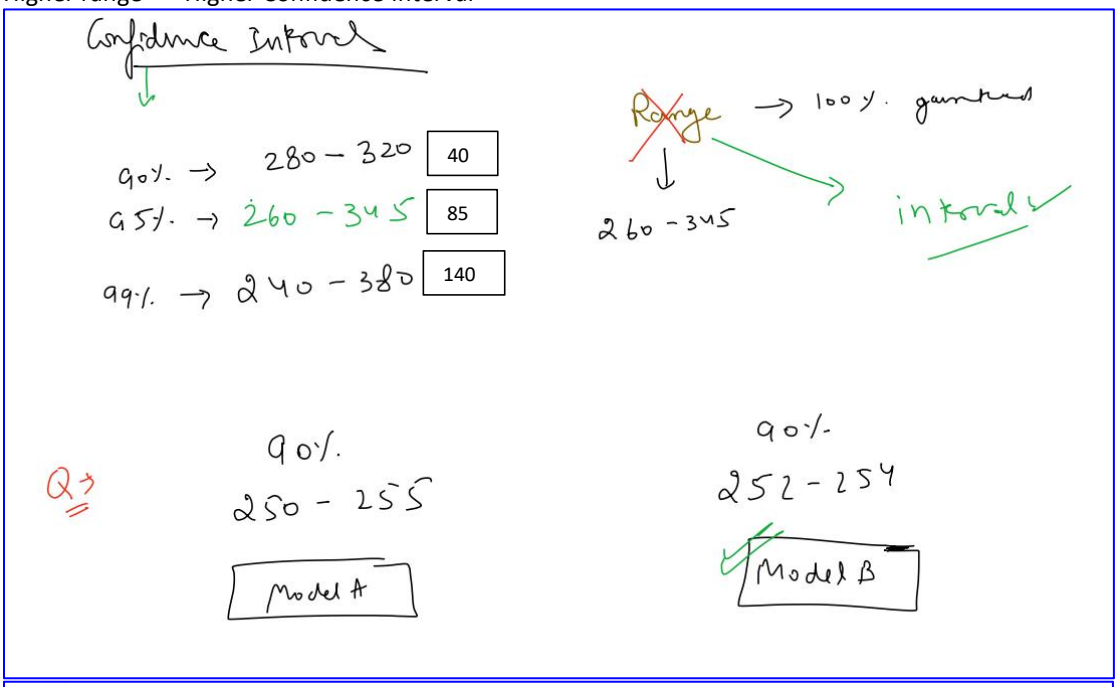
We can predict the exact sales range value at a given time

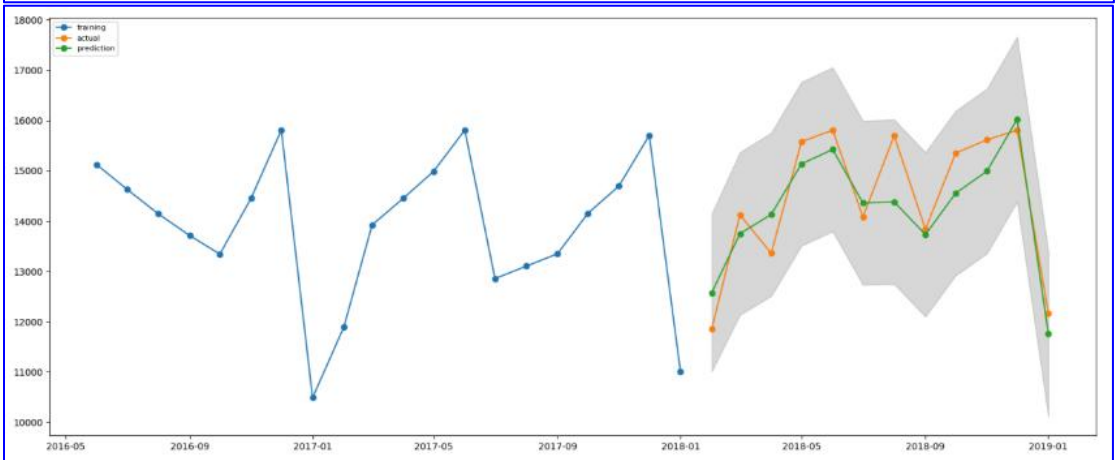




16. Confidence Interval

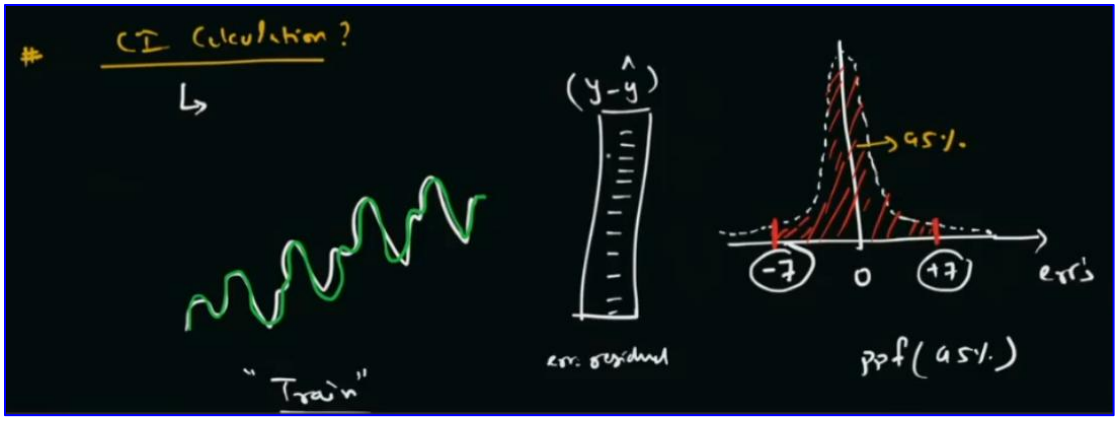
Higher range --> Higher Confidence Interval





- **Definition:** Range [A, B] indicating where the future value is likely to fall, considering forecast error and unpredictability.
- **Purpose:** Reflects uncertainty in predictions, providing a probable range rather than a precise value.
- **Importance:** Helps in assessing the reliability of forecasts, guiding decision-making with an expected range of outcomes.
- **Calculation:** Varies by model; in statsmodels, specified by 'alpha' parameter (e.g., 0.05 for 95% confidence).

- **Interpretation:** For a forecast ŷt with alpha = 0.05, the CI (m, n) implies a 95% probability the true value will fall within this range.
- **Visualization:** Plots can show actual values, forecasted values, and the CI, illustrating the forecast's accuracy and uncertainty.
- **Application Example:** Using statsmodels to get forecast CIs, illustrating how actual observations align with these intervals.
- **Adjusting Confidence Levels:** Varying alpha adjusts the CI's width, balancing between confidence and precision.



CODE

Confidence Interval

```
# 95% conf interval
model.get_forecast(steps=1).conf_int(0.05).values
```

```
array([[10804.48917183, 14260.01215324]])
```

```
model.forecast(steps=1)
```

```
2018-02-01    12532.250663
Freq: MS, dtype: float64
```

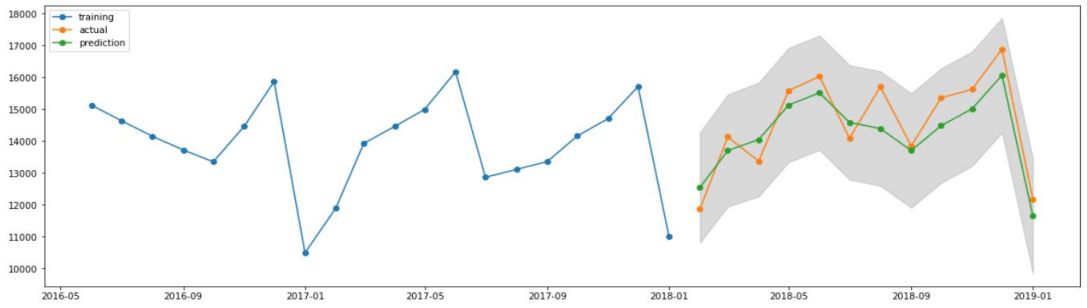
```
model.get_forecast(steps=12).conf_int(0.05).values
```

```
array([[10804.48917183, 14260.01215324],
       [11934.43183466, 15456.406937  ],
       [12254.06728221, 15826.79595277],
       [13328.03158303, 16916.03757935],
       [13712.84008559, 17303.70949634],
       [12783.86190665, 16374.70651929],
       [12587.50465384, 16181.61923024],
       [11899.26985404, 15493.24866777],
       [12679.76967434, 16273.58018768],
       [13206.64627392, 16802.52326398],
       [14259.73996309, 17855.6134731 ],
       [ 9837.43892994, 13433.20056405]])
```

```
test_x[['lower','upper']]=model.get_forecast(steps=12).conf_int(0.05).values
test_x
```

DATE	Sales	pred	lower	upper
2018-02-01	11852.00	12532.250663	10804.489172	14260.012153
2018-03-01	14123.00	13695.419386	11934.431835	15456.406937
2018-04-01	13360.00	14040.431617	12254.067282	15826.795953
2018-05-01	15576.00	15122.034581	13328.031583	16916.037579
2018-06-01	16021.00	15508.274791	13712.840086	17303.709496
2018-07-01	14080.00	14579.284213	12783.861907	16374.706519
2018-08-01	15697.00	14384.561942	12587.504654	16181.619230
2018-09-01	13838.00	13696.259261	11899.269854	15493.248668
2018-10-01	15351.00	14476.674931	12679.769674	16273.580188
2018-11-01	15615.00	15004.584769	13206.646274	16802.523264
2018-12-01	16879.16	16057.676718	14259.739963	17855.613473
2019-01-01	12160.00	11635.319747	9837.438930	13433.200564

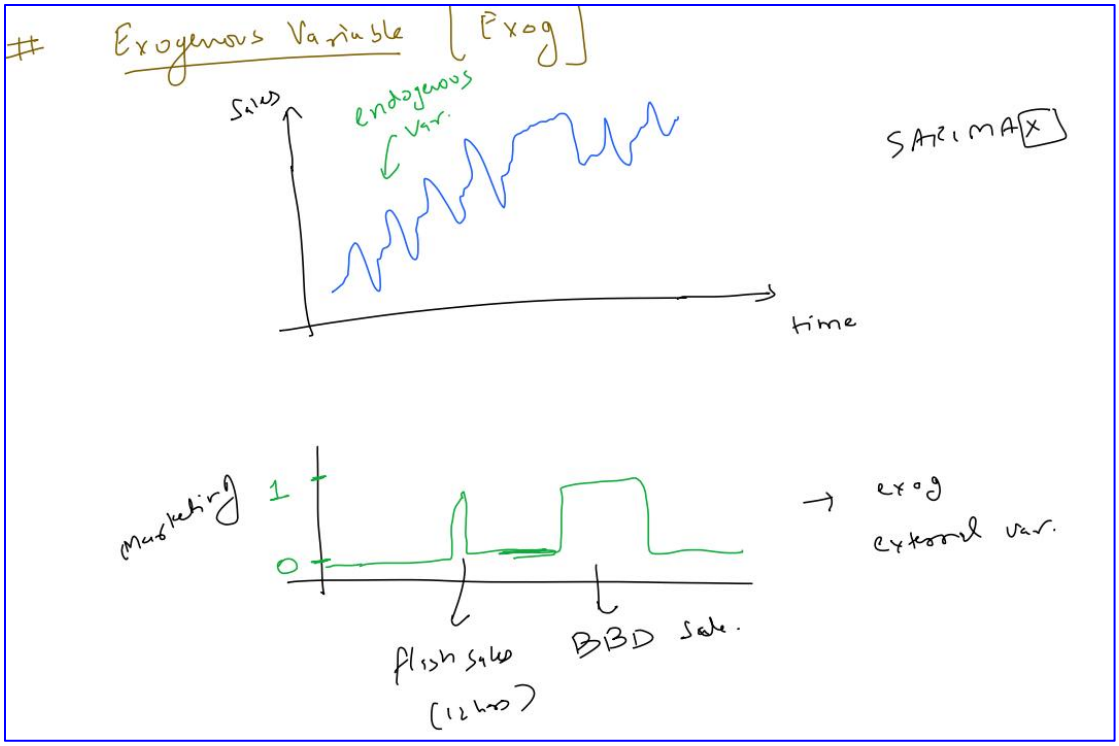
```
plt.plot(train_x['Sales'][-20:],'-o',label='training')
plt.plot(test_x['Sales'],'-o',label='actual')
plt.plot(test_x['pred'],'-o',label='prediction')

plt.fill_between(test_x.index, test_x['lower'],test_x['upper'], color='k', alpha=0.15)
plt.legend()
plt.show()
```



17. Exegeneous Variable

Endo --> Stands for t he data which we have given to train the variable

- **Definition:** External factors not influenced by the system but affecting the output. Examples include holidays, product launches, weather conditions.
- **Impact on Predictions:**
  - Models like SARIMA, focusing on historical data, may miss out on patterns affected by these variables.
  - Incorporating exogenous variables can improve forecast accuracy, especially for events like holidays.
- **Example Analysis:**
  - Predictions vs. Actuals: Discrepancies during holidays indicate the model's inability to account for external impacts without exogenous inputs.
- **Performance Metrics:** MAE, RMSE, and MAPE highlight the overall prediction accuracy, with notable deviations during exogenous events.
- **Solution:** Integrate exogenous variables (like holiday flags) into models to better capture special event impacts.



18. SARIMAX

- **SARIMAX Model:** Extends SARIMA by incorporating exogenous variables (denoted by 'X'), allowing for a more detailed and accurate forecast by including external influences.
- **Key Features:**
  - **Incorporation of Exogenous Variables:** Adds external factors such as holidays into the forecast model, assigning them weights learned during model training.
  - **Hyperparameters:** Retains SARIMA's p, d, q, P, D, Q, s, with the addition of specifying exogenous variables through the exog parameter.
  - **Model Training and Forecasting:** Trains with both endogenous (time series) and exogenous (external) data to predict future values, adjusting for known external impacts.
- **Application:**
  - Demonstrated improvement in forecast accuracy when accounting for holidays as an exogenous variable, reflected in performance metrics (MAE, RMSE, MAPE).
  - The model better aligns predictions with actual peaks on holidays, showcasing the value of integrating relevant external factors.
- **Forecasting with SARIMAX:**
  - **Forecasting Process:** Involves fitting the model with historical data and exogenous variables, then predicting future values, potentially improving forecasts for periods with known external influences.

- **Visual Analysis:** Plots comparing actual vs. predicted values highlight the effectiveness of including exogenous factors. Red lines on plots indicate holidays, allowing visual assessment of forecast accuracy on these days.

- **Practical Considerations:**
  - **Performance Evaluation:** Enhanced model performance demonstrates the utility of exogenous variables in refining forecasts.
  - **Forecaster's Judgment:** Beyond metrics and models, the forecaster's expertise plays a crucial role in evaluating the plausibility and reliability of predictions.
- **Conclusion:**
  - SARIMAX offers a sophisticated approach to time series forecasting by integrating external factors, enabling more nuanced and potentially accurate predictions.
  - The model's effectiveness, especially in handling seasonality and external influences, underscores the importance of a comprehensive approach to forecasting.

Exegeneous Variable

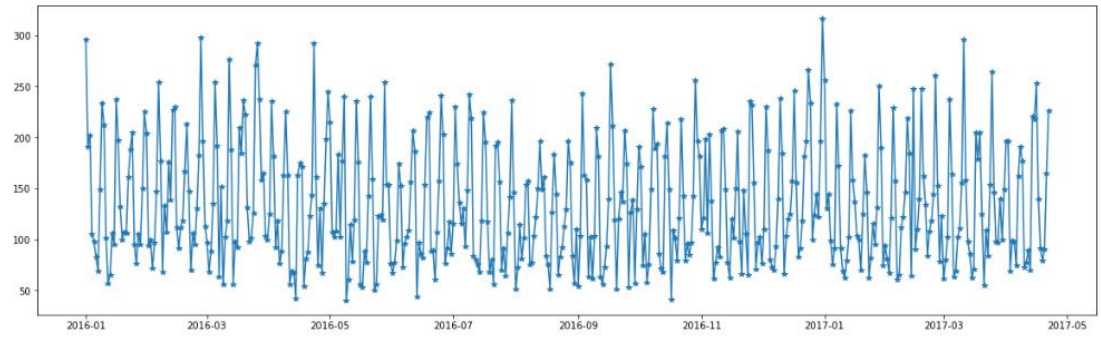
```
df=pd.read_csv('exog.csv')
df.head()
```

	date	weekday	holiday	total
0	1/1/2016	Friday	1	296.0
1	1/2/2016	Saturday	0	191.0
2	1/3/2016	Sunday	0	202.0
3	1/4/2016	Monday	0	105.0
4	1/5/2016	Tuesday	0	98.0

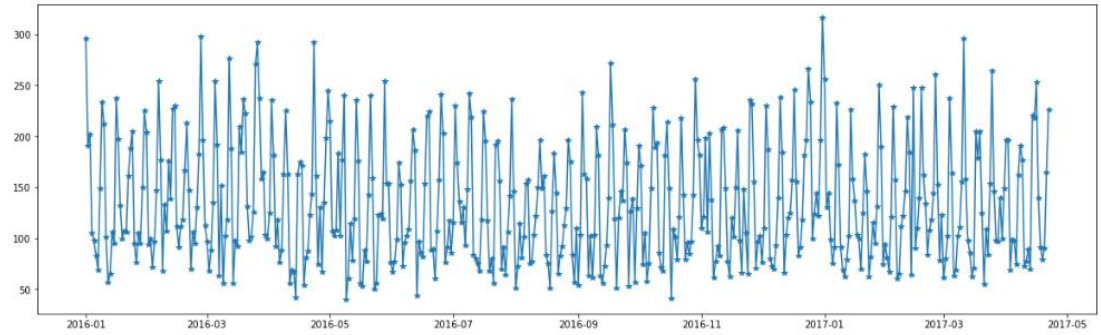
```
df['date'] = pd.to_datetime(df['date'])
df.set_index('date',inplace = True)
df.head(5)
```

date	weekday	holiday	total
2016-01-01	Friday	1	296.0
2016-01-02	Saturday	0	191.0
2016-01-03	Sunday	0	202.0
2016-01-04	Monday	0	105.0
2016-01-05	Tuesday	0	98.0

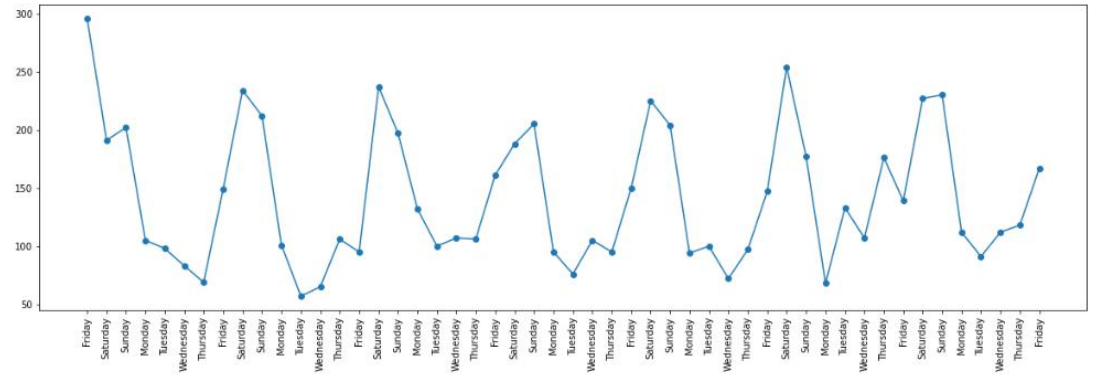
```
plt.plot(df.index,df['total'],'-*')
plt.show()
```




```
plt.plot(df.index, df['total'], '-*')
plt.show()
```




```
plt.plot(range(50),df['total'][:50], '-o')
plt.xticks(range(0,50), df['weekday'][:50],rotation = 90)
plt.show()
```




```
train = df.iloc[:436]
test = df.iloc[436:-40]
```

```
train.shape
```

```
(436, 3)
```

```
test.shape
```

```
(41, 3)
```

```
test.tail()
```

date	weekday	holiday	total
2017-04-17	Monday	1	140.0
2017-04-18	Tuesday	0	91.0
2017-04-19	Wednesday	0	79.0
2017-04-20	Thursday	0	90.0
2017-04-21	Friday	0	165.0

```
model = SARIMAX(train['total'], order=(1,0,0), seasonal_order=(1,0,1,7))  # a simple model
results = model.fit(disp=False)
fc = results.forecast(42)
```

```
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency D will be used.
  warnings.warn('No frequency information was'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency D will be used.
  warnings.warn('No frequency information was'
```

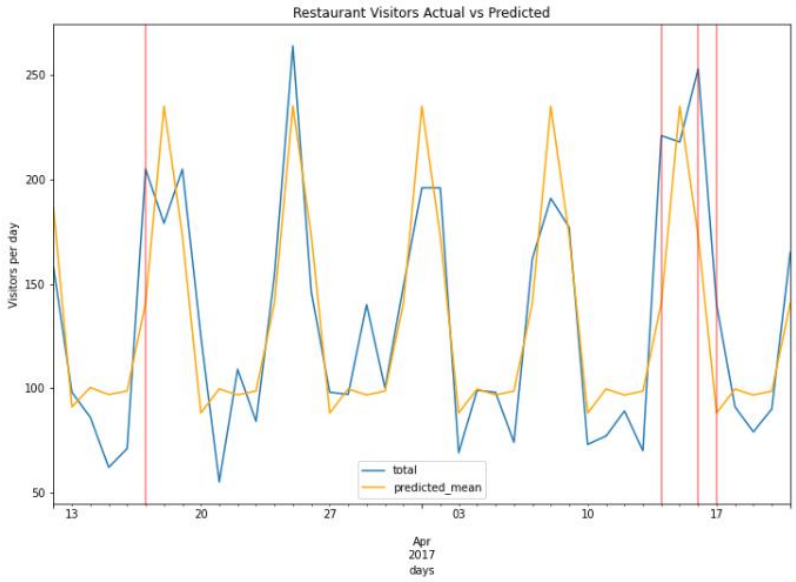
```
# predictions with some in-samples
start=len(train)
end=len(train)+len(test)-1
predictions = results.predict(start=start, end=end)
```

```
# predictions with some in-samples
start=len(train)
end=len(train)+len(test)-1
predictions = results.predict(start=start, end=end)
```

```
performance(test['total'], predictions)
```

```
MAE : 25.549
RMSE : 32.324
MAPE: 0.206
```

```
title='Restaurant Visitors Actual vs Predicted'
ylabel='Visitors per day'
xlabel='days'

ax = test['total'].plot(legend=True,figsize=(12,8),title=title)
predictions.plot(legend=True, color = 'orange')
ax.autoscale(axis='x',tight=True)
ax.set(xlabel=xlabel, ylabel=ylabel)
for x in test.query('holiday==1').index:
    ax.axvline(x=x, color='red', alpha = 0.5);
```




```
# Using the Exog data as part of the training the model
```

```
model = SARIMAX(train['total'],exog=train['holiday'], order=(1,0,0), seasonal_order=(1,0,1,7))  # a simple model
results = model.fit(disp=False)
```

```
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency D will be used.
  warnings.warn('No frequency information was'
D:\Subhrajit\anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.py:524: ValueWarning: No frequency information was prov
ided, so inferred frequency D will be used.
  warnings.warn('No frequency information was'
```

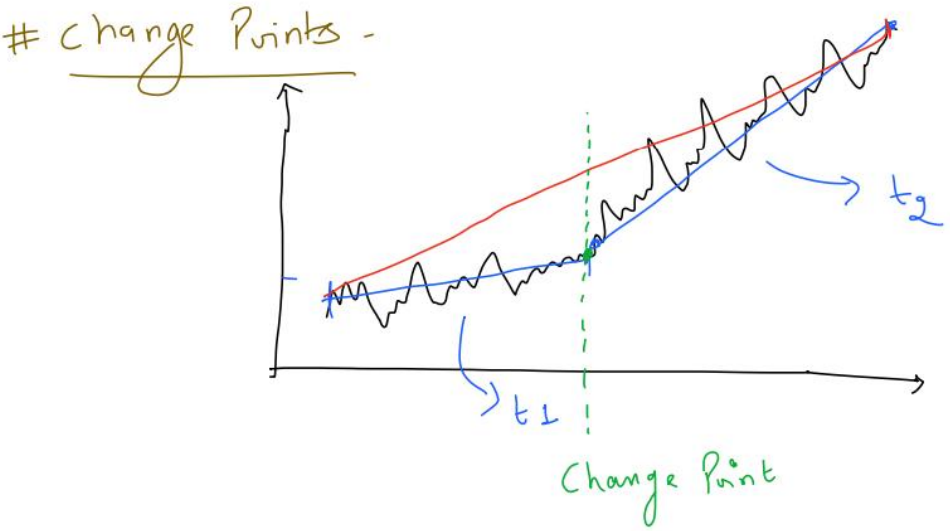
```
exog_forecast=test[['holiday']]
predictions = results.predict(start=start, end=end,exog=exog_forecast).rename('Predictions')
```

```
performance(test['total'],predictions)
```

```
MAE : 19.261
RMSE : 23.539
MAPE: 0.166
```

Day 6

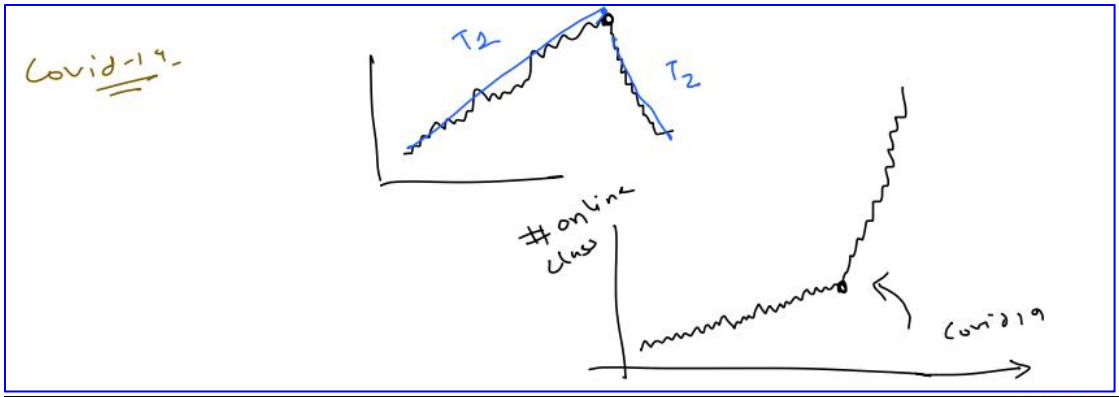
19. Change Point

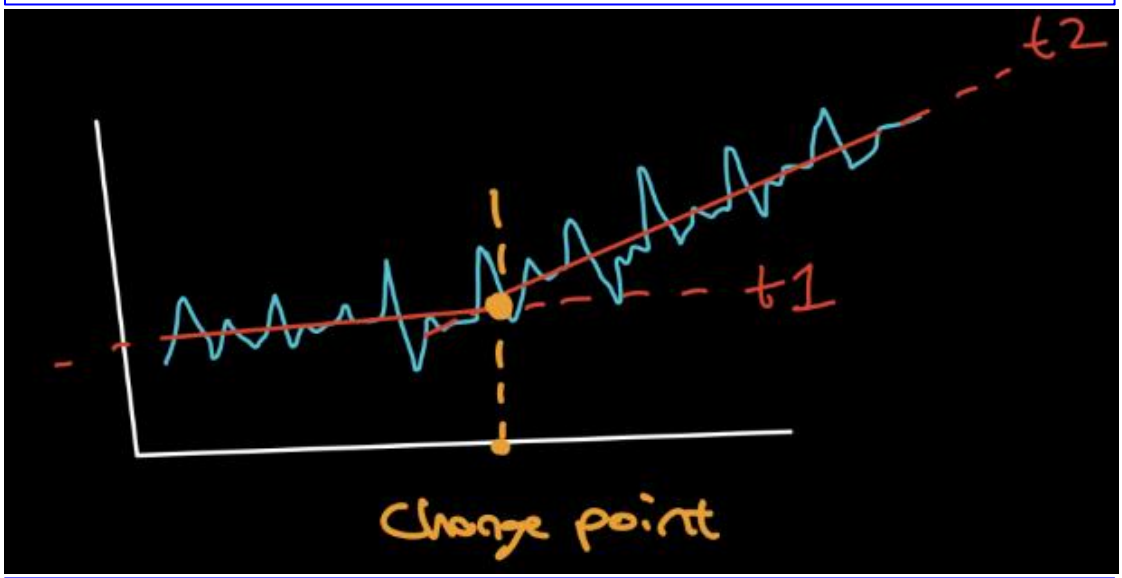


∴ "Past Data"

→ Detect change Points

→ Understand | Analyse.





- **Concept:** A change point marks a significant shift in the trend or statistical properties of a time series, dividing it into segments with distinct characteristics.
- **Detection:**
  - **Sliding Window Method:** A simple approach using a fixed-size window to traverse the series, calculating a cost (e.g., slope change) for elements within the window. Peaks in this cost indicate potential change points.
  - **Libraries and Algorithms:** Advanced methods (e.g., 'ruptures' library) employ more sophisticated algorithms for detecting change points, offering improved accuracy and customization.
- **Utilization:**
  - **Analysis:** Change points help in understanding shifts in data trends, useful for retrospective analysis rather than forecasting future values.
  - **Modeling:** Identifying change points can refine model adjustments, especially when integrating these insights into dynamic models that can adapt to historical shifts.
- **Exogenous Variables and Change Points:**
- Incorporating known change points as exogenous variables in models like SARIMAX can potentially enhance forecast accuracy by accounting for structural breaks.

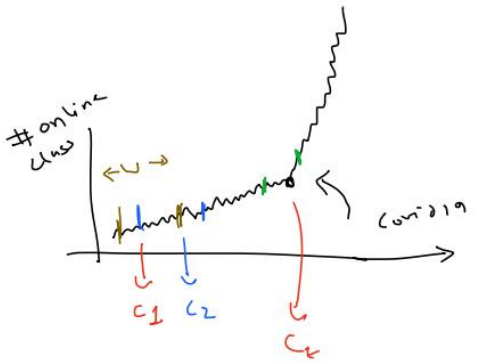
Detecting changepoints.

① window = 7

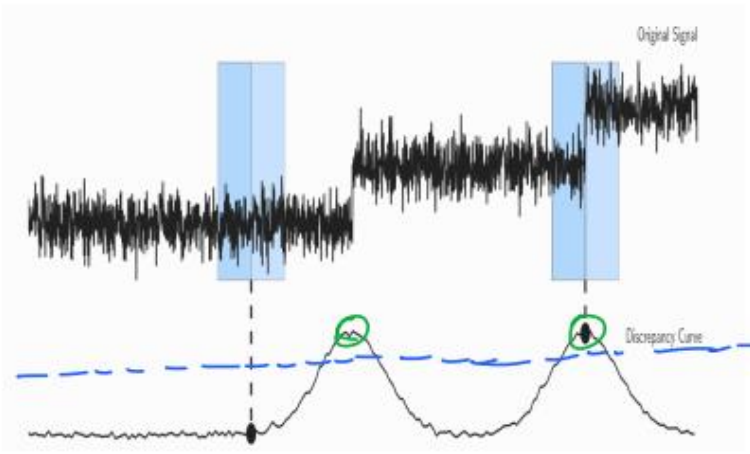
Cost function

↓

mean, std, slope etc.

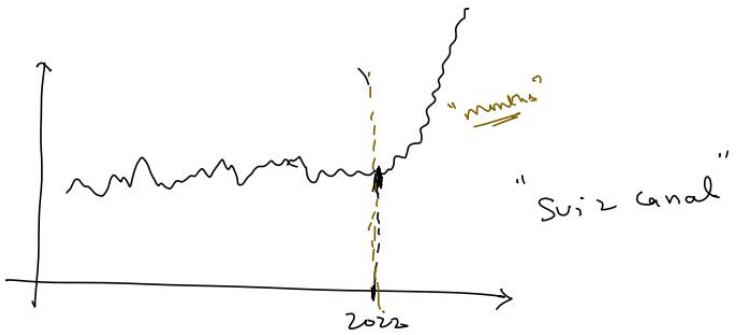


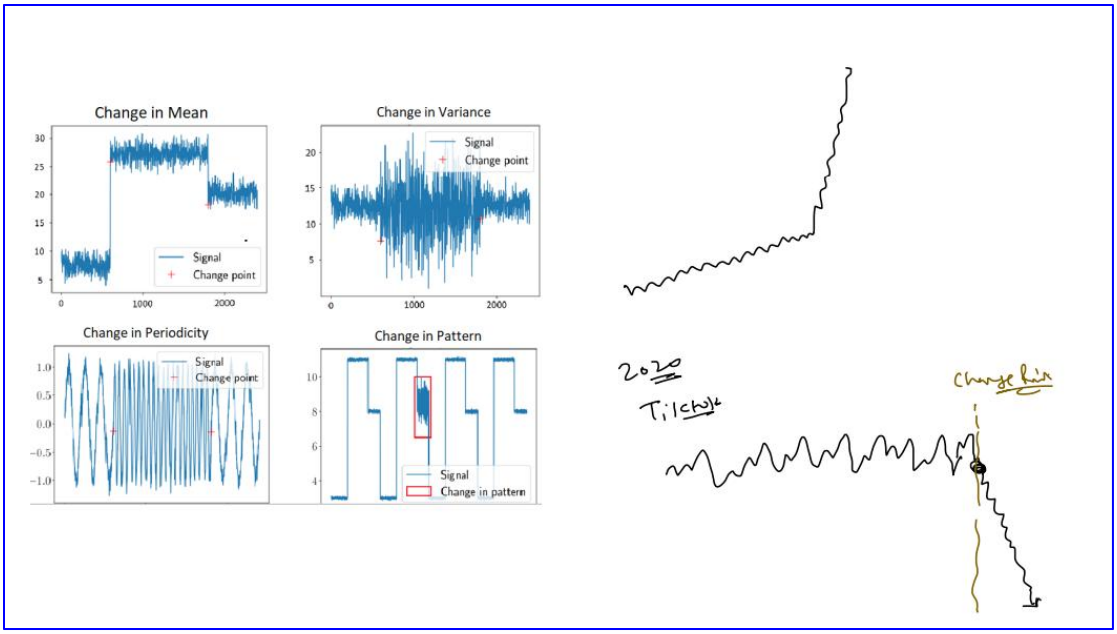
"Sudden change"



Port Congestion

Port Congestion.

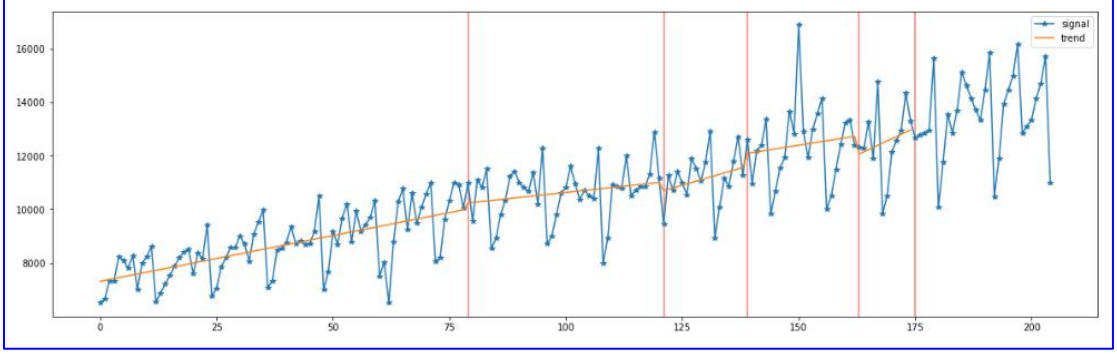




CODE

```
def plot_changepoints(signal, changepoints):
  for cp in changepoints:
    plt.axvline(x=cp, color='#FA8072')
  plt.plot(signal, '-*', label='signal')

  start = 0
  trends = np.array([])
  for i in changepoints:
    x = np.arange(start, i)
    y = signal[start:i]
    l1 = np.polyfit(x, y, deg=1)
    trend = x*l1[0] + l1[1]
    trends = np.append(trends, trend)
    start = i

  plt.plot(trends, label='trend')
  plt.legend()
  plt.show()
```

```
signal = train_x.Sales # Stationarise for mean as cost function

window=60
def get_slope(y):
  return np.polyfit(np.arange(len(y)), y, deg=1)[0]

changepoints = signal.loc[abs(signal.rolling(window, center=True).apply(get_slope).diff()) > 7.5].index
# converting to index from dates
temp = signal.reset_index()
changepoints = temp.loc[temp['DATE'].isin(changepoints)].index.tolist()

plot_changepoints(signal.values, changepoints)
```



```
df.head()
```

date	weekday	holiday	total
2016-01-01	Friday	1	296.0
2016-01-02	Saturday	0	191.0
2016-01-03	Sunday	0	202.0
2016-01-04	Monday	0	105.0
2016-01-05	Tuesday	0	98.0

```
df['ds'] = pd.to_datetime(df.index)
df['y'] = df['total']
df = df[['ds', 'y', 'holiday']]
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 517 entries, 2016-01-01 to 2017-05-31
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   ds       517 non-null    datetime64[ns]
 1   y        478 non-null    float64
 2   holiday  517 non-null    int64
dtypes: datetime64[ns](1), float64(1), int64(1)
memory usage: 16.2 KB
```

```
df.head()
```

date	ds	y	holiday
2016-01-01	2016-01-01	296.0	1
2016-01-02	2016-01-02	191.0	0
2016-01-03	2016-01-03	202.0	0
2016-01-04	2016-01-04	105.0	0
2016-01-05	2016-01-05	98.0	0

20. Prophet

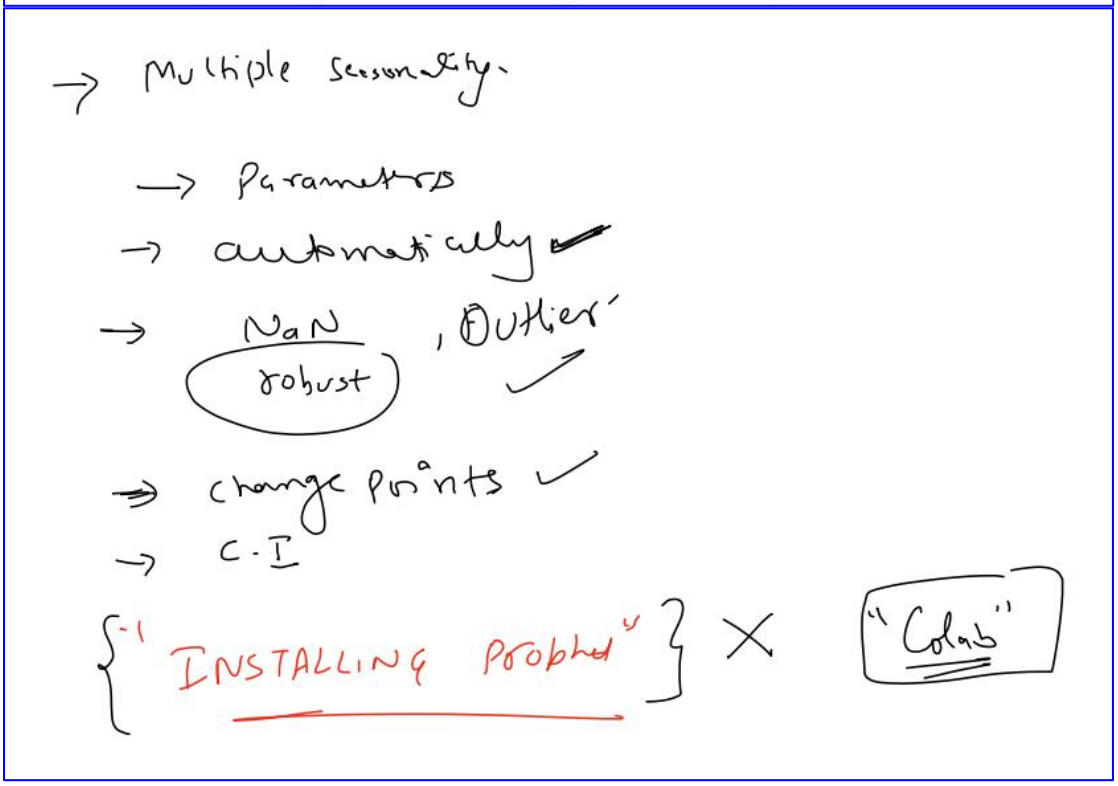
SARIMAX only supports one seasonality
Prophet supports multiple seasonality

https://peerj.com/preprints/3190.pdf

Prophet is a forecasting tool by Facebook designed for time series data that is robust to missing data, shifts in trend, and outliers. It excels in handling datasets with strong seasonal effects.

- **Features:**
  - **Intuitive Parameters:** Easy-to-tune for quick adjustments.
  - **Handles Anomalies:** Efficiently manages missing data and outliers.
  - **Multiple Seasonalities:** Captures complex seasonal patterns through Fourier transforms.
  - **Decomposable Model:** Splits into trend, seasonality, holiday effects, plus an error term.
- **Model Components**:
  - **Trend g(t)):** Represents non-periodic changes.
  - **Seasonality s(t)):** Captures periodic changes (e.g., weekly, yearly).
  - **Holidays h(t)):** Models holiday effects with irregular schedules.
  - **Error (ε_t):** Accounts for unexplained changes.
- **Usage Highlights:**
  - Requires data with 'ds' (date-time) and 'y' (target variable) columns.
  - Easily incorporates holidays and external regressors for improved forecasts.
  - Allows flexibility adjustments via 'changepoint_prior_scale'.

- **Benefits:**
  - **User-Friendly:** Minimal data preparation needed.
  - **Robust and Versatile:** Great for seasonal forecasts and handling data irregularities.
  - **Enhanced Forecasting:** Supports multiple seasonalities and holiday effects for accurate predictions.
  - **Interpretability:** Provides clear insights into the components of the forecast.



Try prophet in Colab

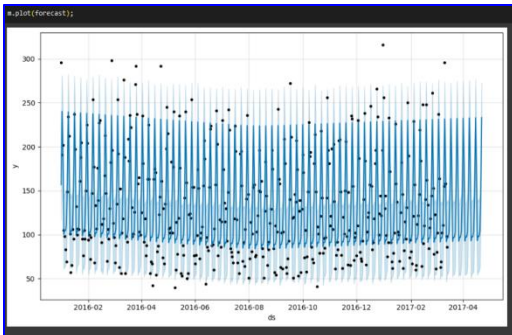
```
# !pip install pystan==2.14
# !pip install fbprophet
```

```
from prophet import Prophet
```

```
from prophet import Prophet

m = Prophet()
m.fit(df['ds', 'y']][:-39])

future = m.make_future_dataframe(periods=39, freq='D')
forecast = m.predict(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/tmpepiasbkw/7ag_eim5.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpepiasbkw/i5v1hybe.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-packages/prophet/stan_model/prophet_model.bin', 'random', 'seed=44348', 'data', 'file=/tmp/tmpepiasbkw/7ag_eim5.json', 'init=/tmp/tmpepiasbkw/i5v1hybe.json', 'output', 'file=/tmp/tmpepiasbkw/prophet_model0gfnxan9/prophet_model-202312
16:40:52 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
16:40:52 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```

```
m.plot(forecast);
```

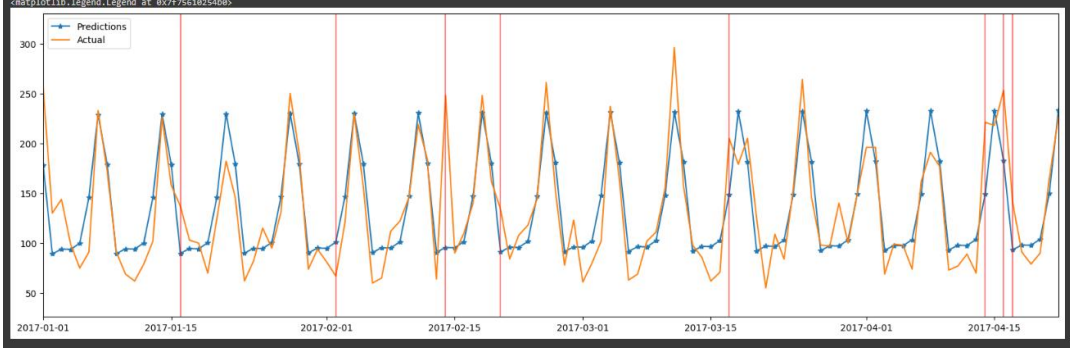



```
forecast.tail(5)
```

	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	additive_terms	additive_terms_lower	additive_terms_upper	weekly	weekly_lower	weekly_upper	multiplicative_terms	multiplicative_terms_lower	multiplicative_terms_upper	yhat
473	2017-04-18	136.962780	57.337375	140.207711	136.755419	137.171938	-38.804844	-38.804844	-38.804844	-38.804844	-38.804844	-38.804844	0.0	0.0	0.0	98.157936
474	2017-04-19	137.002206	57.945912	137.202364	136.785686	137.222529	-39.142466	-39.142466	-39.142466	-39.142466	-39.142466	-39.142466	0.0	0.0	0.0	97.859741
475	2017-04-20	137.041633	62.741046	145.340679	136.809370	137.270125	-32.977602	-32.977602	-32.977602	-32.977602	-32.977602	-32.977602	0.0	0.0	0.0	104.064031
476	2017-04-21	137.081059	111.021286	191.650918	136.839392	137.318899	12.738671	12.738671	12.738671	12.738671	12.738671	12.738671	0.0	0.0	0.0	149.819730
477	2017-04-22	137.120485	192.664407	272.697525	136.866258	137.366807	96.106633	96.106633	96.106633	96.106633	96.106633	96.106633	0.0	0.0	0.0	233.227118

```
plt.plot(forecast['ds'], forecast['yhat'],'-*', label = 'Predictions')

plt.plot(df['ds'], df['y'], label = 'Actual')

for x in df.query('holiday==1')['ds']:
    plt.axvline(x=x, color='red', alpha = 0.5);

plt.xlim(pd.to_datetime('2017-01-01'), pd.to_datetime('2017-04-22'))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f75610254b0>
```




```
performance(df['y'][:-39],forecast['yhat'][:-39])
```

```
MAE : 24.01
RMSE : 32.087
MAPE: 0.214
```

```
model2 = Prophet(yearly_seasonality = True, weekly_seasonality = True)
model2.add_regressor('holiday')
model2.fit(df[:-39])
forecast2 = model2.predict(df)
```

```
INFO:prophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
DEBUG:cmdstanpy:input tempfile: /tmp/tmpepiasbkw/tckegchm.json
DEBUG:cmdstanpy:input tempfile: /tmp/tmpepiasbkw/w6p82yda.json
DEBUG:cmdstanpy:idx 0
DEBUG:cmdstanpy:running CmdStan, num_threads: None
DEBUG:cmdstanpy:CmdStan args: ['/usr/local/lib/python3.10/dist-packages/prophet/stan_model/prophet_model.bin', 'random', 'seed=19383', 'data', 'file=/tmp/tmpepiasbkw/tckegchm.json', 'init=/tmp/tmpepiasbkw/w6p82yda.
16:54:25 - cmdstanpy - INFO - Chain [1] start processing
INFO:cmdstanpy:Chain [1] start processing
16:54:25 - cmdstanpy - INFO - Chain [1] done processing
INFO:cmdstanpy:Chain [1] done processing
```

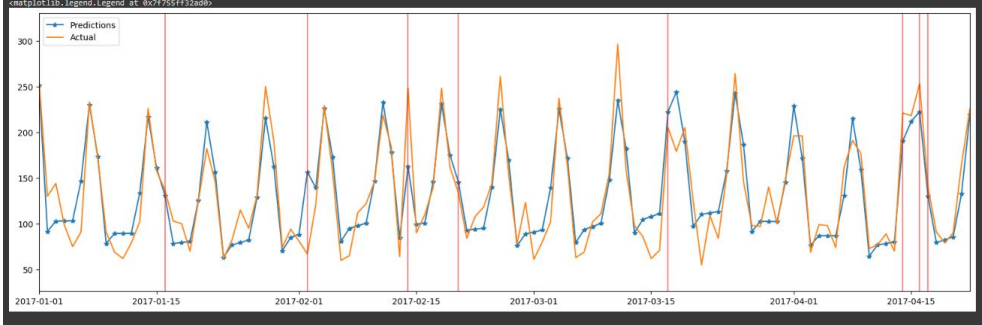
+ Code + Text

```
performance(df['y'][:-39],forecast2['yhat'][:-39])
```

```
MAE : 20.174
RMSE : 25.932
MAPE: 0.183
```

```
plt.plot(forecast['ds'], forecast2['yhat'],'-*', label = 'Predictions')

plt.plot(df['ds'], df['y'], label = 'Actual')

for x in df.query('holiday==1')['ds']:
    plt.axvline(x=x, color='red', alpha = 0.5);

plt.xlim(pd.to_datetime('2017-01-01'), pd.to_datetime('2017-04-22'))
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f755ff32ad0>
```




```
model2 = Prophet(yearly_seasonality = True, weekly_seasonality = True)
m.add_country_holidays(country_name='US')
model2.fit(df[:-39])
forecast2 = model2.predict(df)
```

21. Time Series Forecasting Using Linear Regression

- **Approach:** Applying linear regression to forecast time series data by engineering features that capture underlying patterns and seasonality.
- **Feature Engineering:**
  - **Weekday/Weekend:** Binary feature indicating weekends, capturing visitor variation.
  - **Lagged Features:** Past values (e.g., Lag_1, Lag_2, ...) to utilize historical data.
  - **Averaging:** Using past averages (e.g., last month, week, 2-week averages) to smooth out short-term fluctuations.
  - **Seasonality:** Average sales by day of the week to capture weekly seasonality.
- **Model Training:**
  - **Selected features:** Lag_1, last_month_avg_level, last_week_avg_level, last_2week_avg_level, sale_wrt_dow, holiday.
  - Split data into training and testing sets for model evaluation.
  - Train a Linear Regression model on the training set.
- **Performance Evaluation:**
  - Use metrics such as MAPE (Mean Absolute Percentage Error) to assess forecast accuracy.
  - Visualize actual vs. predicted values to qualitatively evaluate the model's performance.

- **Observations:**
  - The linear regression model can perform surprisingly well for forecasting, especially when predicting short-term future values.
  - Incorporating exogenous variables like holidays can significantly improve forecast accuracy.
- **Considerations for Improvement:**
  - Innovate with feature engineering to capture more complex patterns.
  - Apply feature selection techniques to refine the model.
  - Explore different regression models and hyperparameter tuning for optimization.
  - Consider stacking or cascading models for enhanced predictions.
- **Important Note:**
  - SARIMAX provided forecasts for a longer horizon, while the linear regression model focused on short-term (1-day ahead) forecasting using features like Lag_1.
  - For multi-day forecasts, a separate linear regression model is needed for each forecast horizon.