# Time Series Analysis

# ARIMA and Seasonal ARIMA

## We can use Arima or Sarimax when our data is stationary because non stationary data gives false result.

## To check whether our data is stationary or not we can use Dickey Fuller Test and if its not stationary then we can use differencing technique again and again to make it stationary. (Non Stationarity occurs because of trend, seasonality and many other factors)

## Stationary:

### A common assumption in many time series techniques is that the data are stationary.

# A stationary process has the property that the mean, variance and autocorrelation structure do not change over time.

### Stationarity can be defined in precise mathematical terms, but for our purpose we mean a flat looking series, without trend, constant variance over time, a constant autocorrelation structure over time and no periodic fluctuations (seasonality).

## Autoregressive Integrated Moving Averages

### The general process for ARIMA models is the following:

#### 1. Data Collection

#### 2. Data Cleaning (Setting columns name, handling missing value, converting month into datetime format, setting index to Month column)

#### 3. Visualize the Time Series Data

#### 4. Check the data is stationary or not using Dickey Fuller test.

#### 5. Make the time series data stationary means constant mean, variance and auto correlation using techniques like differncing for removing trend and

**seasonality.**

### 5. Plot the Correlation and AutoCorrelation Charts

### 6. Construct the ARIMA Model or Seasonal ARIMA based on the data

### 7. Use the model to make predictions

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline
```

In [2]:
```
df = pd.read_csv('perrin-freres-monthly-champagne.csv') ## Another way of set
```

In [3]:
```
df.head()
```

Out[3]:

| | Month | Perrin Freres monthly champagne sales millions ?64-?72 |
|---|---|---|
| 0 | 1964-01 | 2815.0 |
| 1 | 1964-02 | 2672.0 |
| 2 | 1964-03 | 2755.0 |
| 3 | 1964-04 | 2721.0 |
| 4 | 1964-05 | 2946.0 |

In [4]:
```
df.tail()
```

Out[4]:

| | Month | Perrin Freres monthly champagne sales millions ?64-?72 |
|---|---|---|
| 102 | 1972-07 | 4298.0 |
| 103 | 1972-08 | 1413.0 |
| 104 | 1972-09 | 5877.0 |
| 105 | NaN | NaN |
| 106 | Perrin Freres monthly champagne sales millions... | NaN |

### Another way of renaming a column

### df.rename(columns={'Perrin Freres monthly champagne sales millions ?64-?72':'Sales'}, inplace= True)

### Inplace =True means it will permanently change the name.

In [5]:
```
df.columns = ["Month",'Sales']
```

In [6]:
```
df.isnull().sum()
```

Out[6]:
```
Month    1
Sales    2
dtype: int64
```

In [7]:
```
df = df.dropna()
```

**We can also delete the last two rows by using**

**df.drop(106,axis=0,inplace=True)**

In [8]:
```
df.head()
```

Out[8]:

| | Month | Sales |
|---|---|---|
| 0 | 1964-01 | 2815.0 |
| 1 | 1964-02 | 2672.0 |
| 2 | 1964-03 | 2755.0 |
| 3 | 1964-04 | 2721.0 |
| 4 | 1964-05 | 2946.0 |

In [9]:
```
df.tail()
```

Out[9]:

| | Month | Sales |
|---|---|---|
| 100 | 1972-05 | 4618.0 |
| 101 | 1972-06 | 5312.0 |
| 102 | 1972-07 | 4298.0 |
| 103 | 1972-08 | 1413.0 |
| 104 | 1972-09 | 5877.0 |

In [10]:
```
df.shape
```

Out[10]:
```
(105, 2)
```

In [11]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 105 entries, 0 to 104
Data columns (total 2 columns):
Month    105 non-null object
Sales    105 non-null float64
dtypes: float64(1), object(1)
memory usage: 2.5+ KB
```

In [12]:
```
df['Month'] = pd.to_datetime(df['Month'])
```

In [13]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 105 entries, 0 to 104
Data columns (total 2 columns):
Month    105 non-null datetime64[ns]
Sales    105 non-null float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 2.5 KB
```

In [14]:
```
df.head()
```

Out[14]:

| | Month | Sales |
|---|---|---|
| 0 | 1964-01-01 | 2815.0 |
| 1 | 1964-02-01 | 2672.0 |
| 2 | 1964-03-01 | 2755.0 |
| 3 | 1964-04-01 | 2721.0 |
| 4 | 1964-05-01 | 2946.0 |

In [15]:
```
df.head()
```

Out[15]:

| | Month | Sales |
|---|---|---|
| 0 | 1964-01-01 | 2815.0 |
| 1 | 1964-02-01 | 2672.0 |
| 2 | 1964-03-01 | 2755.0 |
| 3 | 1964-04-01 | 2721.0 |
| 4 | 1964-05-01 | 2946.0 |

### Setting Month as index column

In [16]:
```
df.set_index('Month',inplace=True)
```

In [17]:
```
df.head()
```

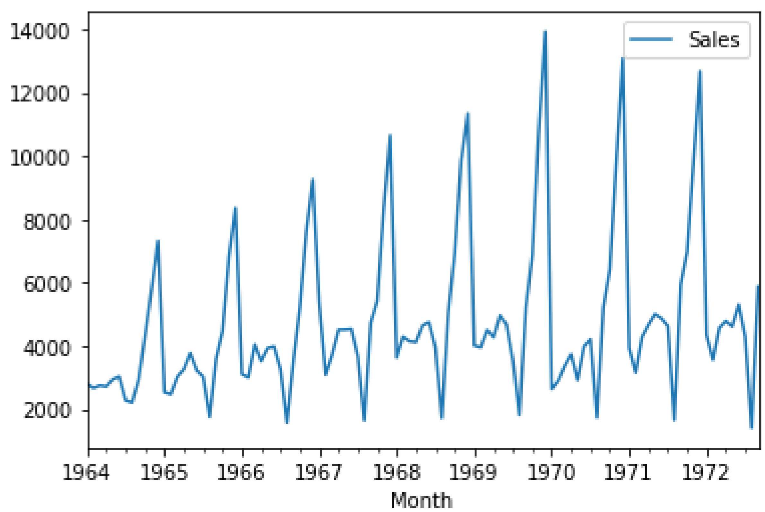
Out[17]:

| | Sales |
|---|---|
| **Month** | |
| 1964-01-01 | 2815.0 |
| 1964-02-01 | 2672.0 |
| 1964-03-01 | 2755.0 |
| 1964-04-01 | 2721.0 |
| 1964-05-01 | 2946.0 |

## Visualize the Data

In [18]:
```
plt.figure(figsize=(20,20))
df.plot();
```

<Figure size 1440x1440 with 0 Axes>




**To check whether the data is Stationary or not using Dickey Fuller Test**

In [19]:
```
from statsmodels.tsa.stattools import adfuller
```

In [20]:
```
test_result = adfuller(df['Sales'])
```

In [21]:
```
test_result
```

Out[21]:
```
(-1.8335930563276237,
 0.3639157716602447,
 11,
 93,
 {'1%': -3.502704609582561,
  '5%': -2.8931578098779522,
  '10%': -2.583636712914788},
 1478.4633060594724)
```

### Ho: It is non stationary (Null Hypothesis)

### H1: It is stationary (Alternate Hypothesis)

In [22]:
```
#Ho: It is non stationary
#H1: It is stationary

def adfuller_test(sales):
    result=adfuller(sales)
    labels = ['ADF Test Statistic','p-value','#Lags Used','Number of Observat
    for value,label in zip(result,labels):
        print(label+' : '+str(value) )
    if result[1] <= 0.05:
        print("Strong evidence against the null hypothesis(Ho), reject the nu
    else:
        print("Weak evidence against null hypothesis, time series has a unit 
```

In [23]:
```
adfuller_test(df['Sales'])
```

```
ADF Test Statistic : -1.8335930563276237
p-value : 0.3639157716602447
#Lags Used : 11
Number of Observations Used : 93
Weak evidence against null hypothesis, time series has a unit root, indicati
ng it is non-stationary
```

### Since the P-VALUE is greater than 0.05, so we are failed to reject the null hypothesis and we have to make the data stationary using techniues like differencing to remove trend or seasonality where we delete the whole column with itself by shifting it to 1 or as per requirement.

##### For seasonality we shift it by 12.

## Differencing

In [24]:
```
df['Sales First Difference'] = df['Sales'] - df['Sales'].shift(1)
```

In [25]:
```
df['Sales'].shift(1)
```

Out[25]:
```
Month
1964-01-01       NaN
1964-02-01    2815.0
1964-03-01    2672.0
1964-04-01    2755.0
1964-05-01    2721.0
               ...
1972-05-01    4788.0
1972-06-01    4618.0
1972-07-01    5312.0
1972-08-01    4298.0
1972-09-01    1413.0
Name: Sales, Length: 105, dtype: float64
```

In [26]:
```
df['Seasonal First Difference']=df['Sales']-df['Sales'].shift(12)
```

In [27]:
```
df.head(15)
```

Out[27]:

| Month | Sales | Sales First Difference | Seasonal First Difference |
|---|---|---|---|
| 1964-01-01 | 2815.0 | NaN | NaN |
| 1964-02-01 | 2672.0 | -143.0 | NaN |
| 1964-03-01 | 2755.0 | 83.0 | NaN |
| 1964-04-01 | 2721.0 | -34.0 | NaN |
| 1964-05-01 | 2946.0 | 225.0 | NaN |
| 1964-06-01 | 3036.0 | 90.0 | NaN |
| 1964-07-01 | 2282.0 | -754.0 | NaN |
| 1964-08-01 | 2212.0 | -70.0 | NaN |
| 1964-09-01 | 2922.0 | 710.0 | NaN |
| 1964-10-01 | 4301.0 | 1379.0 | NaN |
| 1964-11-01 | 5764.0 | 1463.0 | NaN |
| 1964-12-01 | 7312.0 | 1548.0 | NaN |
| 1965-01-01 | 2541.0 | -4771.0 | -274.0 |
| 1965-02-01 | 2475.0 | -66.0 | -197.0 |
| 1965-03-01 | 3031.0 | 556.0 | 276.0 |

### Again test dickey fuller test

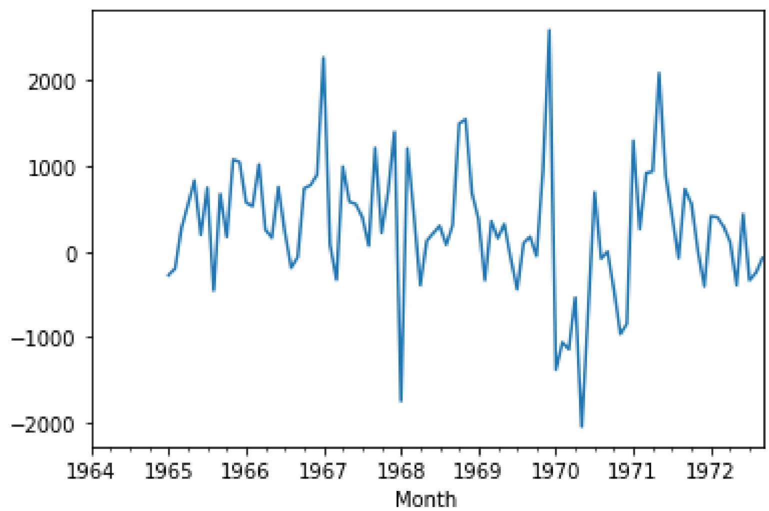
In [28]:
```
adfuller_test(df['Seasonal First Difference'].dropna())
```

```
ADF Test Statistic : -7.626619157213163
p-value : 2.060579696813685e-11
#Lags Used : 0
Number of Observations Used : 92
Strong evidence against the null hypothesis(Ho), reject the null hypothesis.
Data has no unit root and is stationary
```

### Here the P-VALUE is less than 0.05 so we can reject the null hypothesis and can say the data is stationary now and we can use time series models.

In [29]:
```
df['Seasonal First Difference'].plot()
```

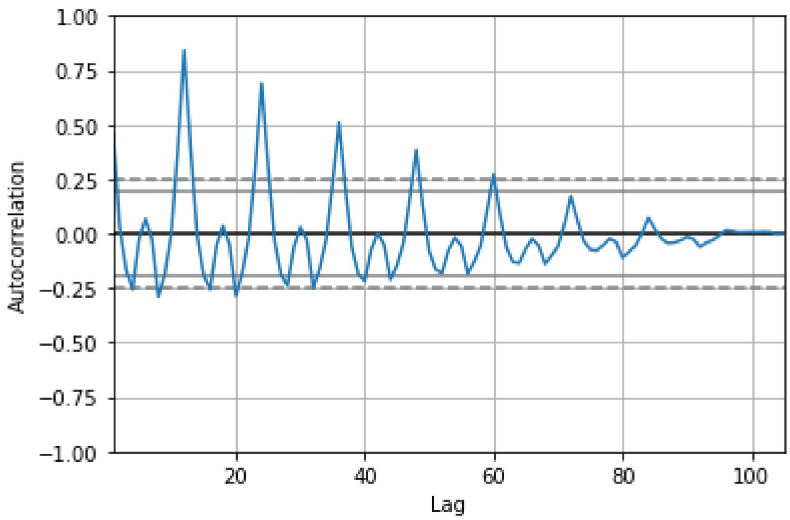
Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x27f9551f7c8>




## Starting year i.e.,1964 NANs are there so we have deleted them and here we can see that the data is stationary because no seasonality and trend is there.

#### Auto correlation on the output data for checking the stationarity of data.

In [30]:
```
from pandas.plotting import autocorrelation_plot
autocorrelation_plot(df['Sales'])
plt.show()
```




# Auto Regressive Integrated Moving Average Model

## Final Thoughts on Autocorrelation and Partial Autocorrelation

### Identification of an AR model is often best done with the PACF.

**For an AR model, the theoretical PACF "shuts off" past the order of the model. The phrase "shuts off" means that in theory the partial autocorrelations are equal to 0 beyond that point. Put another way, the number of non-zero partial autocorrelations gives the order of the AR model. By the "order of the model" we mean the most extreme lag of x that is used as a predictor.**

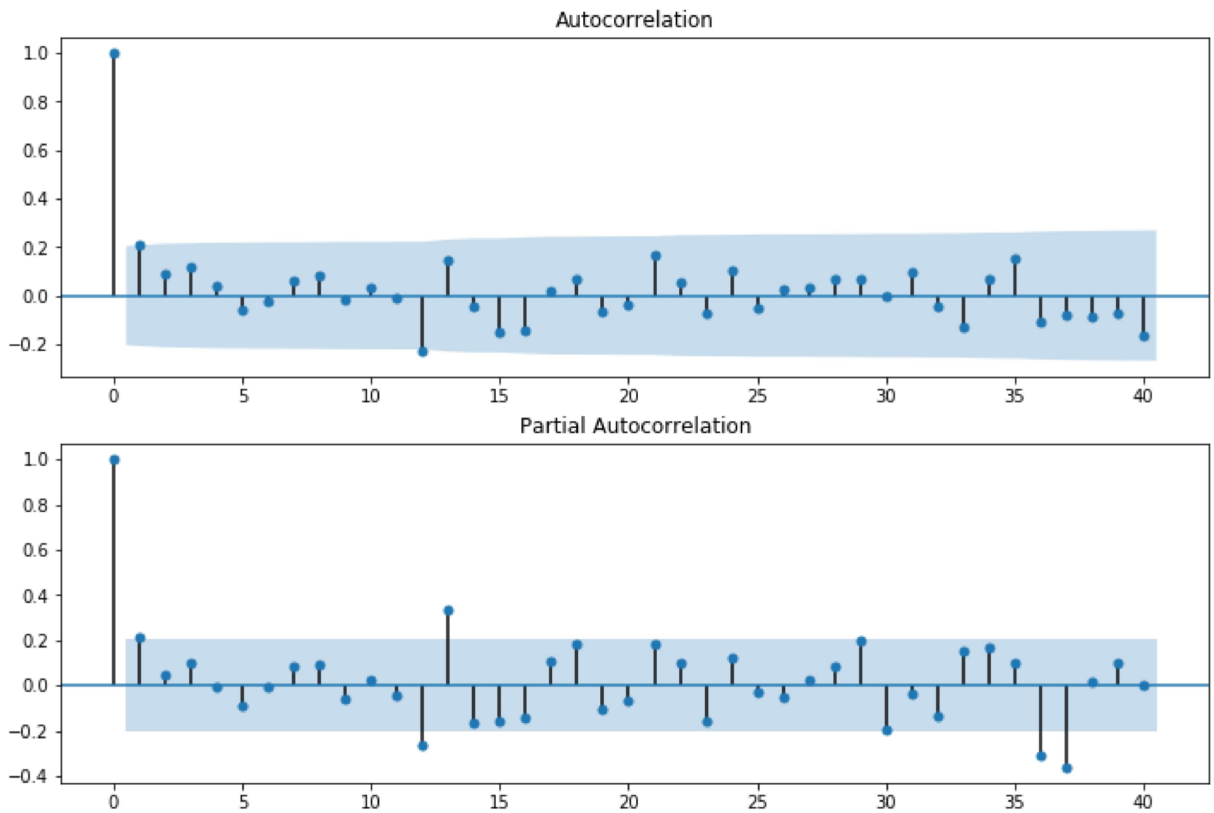
### Identification of an MA model is often best done with the ACF rather than the PACF.

**For an MA model, the theoretical PACF does not shut off, but instead tapers toward 0 in some manner. A clearer pattern for an MA model is in the ACF. The ACF will have non-zero autocorrelations only at lags involved in the model.**

# p,d,q p AR model lags d differencing q MA lags

In [31]:
```
from statsmodels.graphics.tsaplots import plot_acf,plot_pacf
import statsmodels.api as sm
```

In [32]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(2,1,1)
fig = sm.graphics.tsa.plot_acf(df['Seasonal First Difference'].iloc[13:],lags
ax2 = fig.add_subplot(2,1,2)
fig = sm.graphics.tsa.plot_pacf(df['Seasonal First Difference'].iloc[13:],lag
```




# ARIMA Model

#### For non-seasonal data

### p=1, d=1, q=0 or 1, we have taken p=1 as our pacf grapg is shuts directly at lag 1 whereas in acf it is decreasing exponentially.

In [33]:
```
from statsmodels.tsa.arima_model import ARIMA
```

## Since it is a seasonal data so it will not work well so we are going to use SARIMAX.

In [34]:
```
model=ARIMA(df['Sales'],order=(1,1,1))
model_fit=model.fit()
```

```
C:\Users\max14\Anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.p
y:165: ValueWarning: No frequency information was provided, so inferred freq
uency MS will be used.
  % freq, ValueWarning)
C:\Users\max14\Anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.p
y:165: ValueWarning: No frequency information was provided, so inferred freq
uency MS will be used.
  % freq, ValueWarning)
```

In [35]:
```
model_fit.summary()
```

Out[35]:

ARIMA Model Results

| | | | |
|---|---|---|---|
| **Dep. Variable:** | D.Sales | **No. Observations:** | 104 |
| **Model:** | ARIMA(1, 1, 1) | **Log Likelihood** | -951.126 |
| **Method:** | css-mle | **S.D. of innovations** | 2227.262 |
| **Date:** | Sun, 18 Oct 2020 | **AIC** | 1910.251 |
| **Time:** | 00:16:24 | **BIC** | 1920.829 |
| **Sample:** | 02-01-1964 | **HQIC** | 1914.536 |
| | - 09-01-1972 | | |

| | coef | std err | z | P>\|z\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **const** | 22.7822 | 12.405 | 1.836 | 0.069 | -1.532 | 47.096 |
| **ar.L1.D.Sales** | 0.4343 | 0.089 | 4.866 | 0.000 | 0.259 | 0.609 |
| **ma.L1.D.Sales** | -1.0000 | 0.026 | -38.503 | 0.000 | -1.051 | -0.949 |

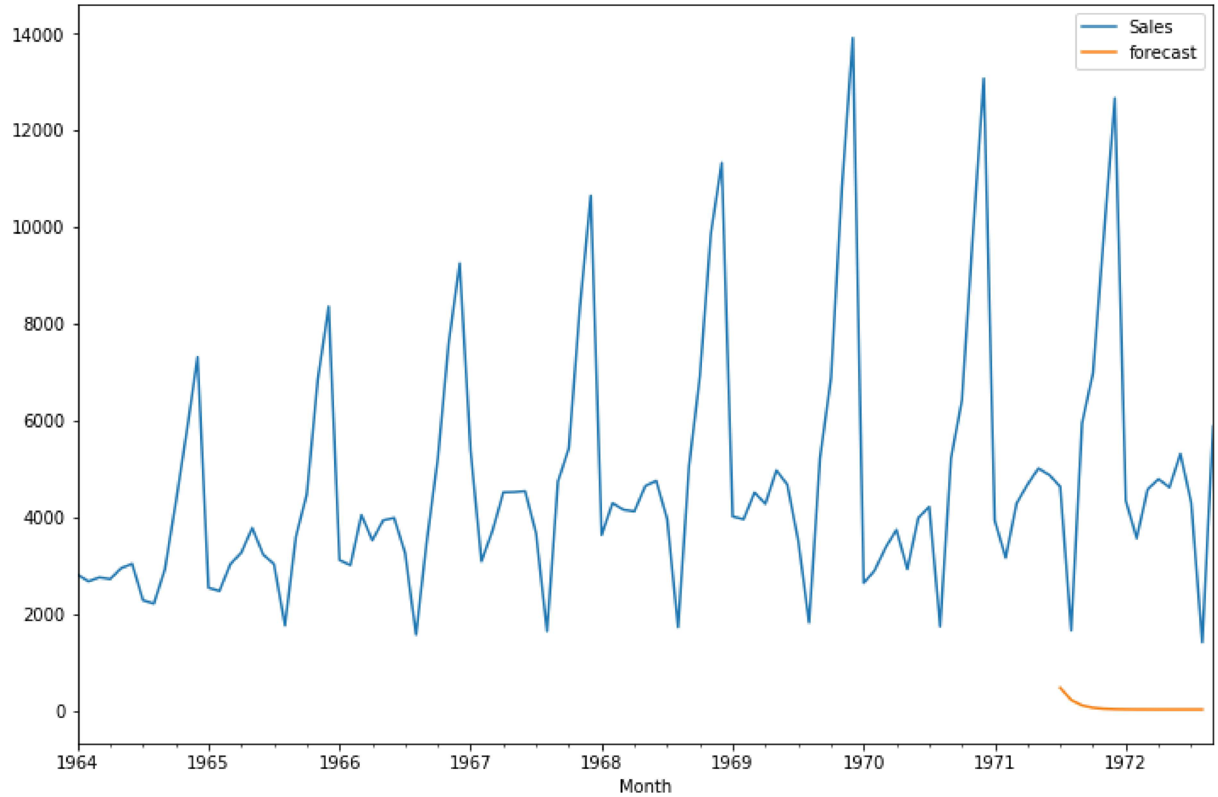
Roots

| | Real | Imaginary | Modulus | Frequency |
|---|---|---|---|---|
| **AR.1** | 2.3023 | +0.0000j | 2.3023 | 0.0000 |
| **MA.1** | 1.0000 | +0.0000j | 1.0000 | 0.0000 |

## In the below image we can clearly see the forecasted line is improper when we are using ARIMA model because it is seasonal data.

In [36]:
```
df['forecast']=model_fit.predict(start=90,end=103,dynamic=True)
df[['Sales','forecast']].plot(figsize=(12,8))
```

Out[36]: <matplotlib.axes._subplots.AxesSubplot at 0x27f92975ec8>




# SARIMAX Model

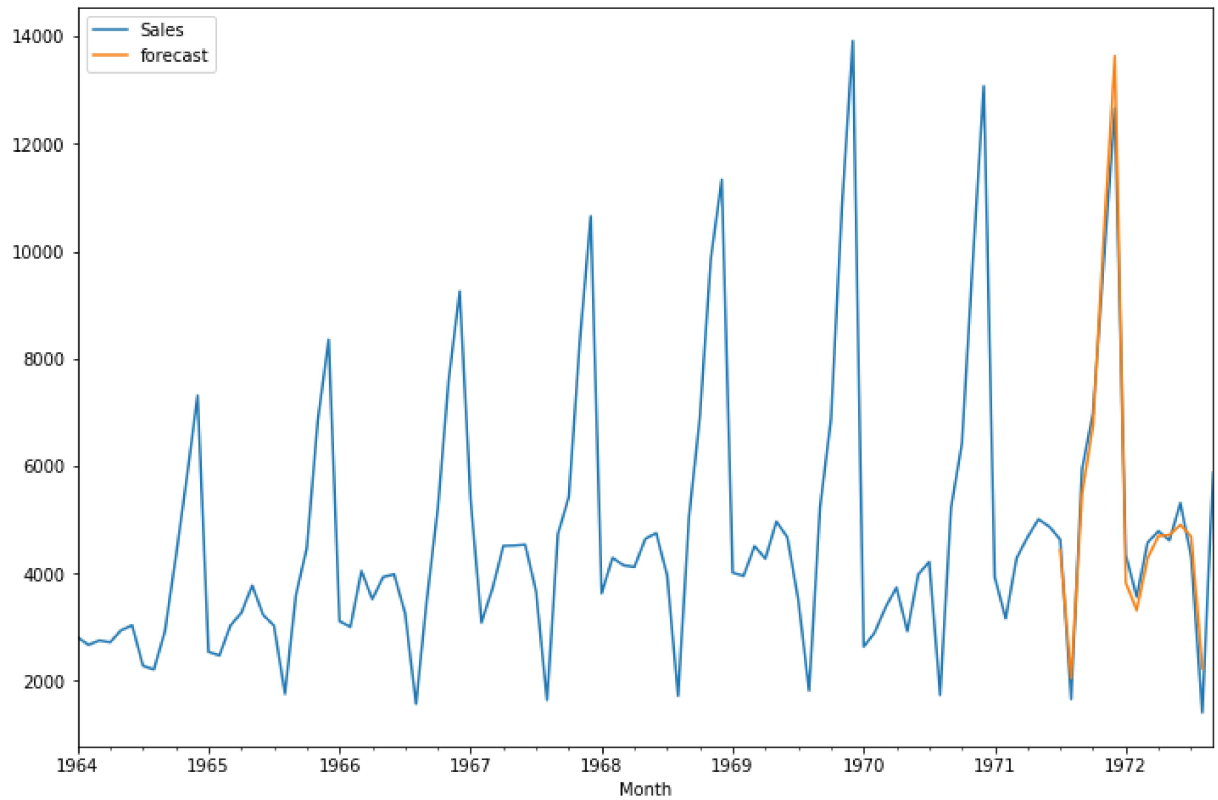
### As the data is seasonal so here we are using SARIMAX model because ARIMA is not working well.

In [37]:
```
model=sm.tsa.statespace.SARIMAX(df['Sales'],order=(1, 1, 1),seasonal_order=(1

# In a season by how many units we are shifting i.e. 12.

results=model.fit()
```

```
C:\Users\max14\Anaconda3\lib\site-packages\statsmodels\tsa\base\tsa_model.p
y:165: ValueWarning: No frequency information was provided, so inferred freq
uency MS will be used.
  % freq, ValueWarning)
```

In [38]:
```
df['forecast']=results.predict(start=90,end=103,dynamic=True)
df[['Sales','forecast']].plot(figsize=(12,8))
```

Out[38]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x27f9341f2c8>
```




# As we can see our Sarimax model is working fine so now we are going to predict future values.

In [39]:
```
from pandas.tseries.offsets import DateOffset
future_dates=[df.index[-1]+ DateOffset(months=x)for x in range(0,24)]
```

In [40]:
```
future_datest_df=pd.DataFrame(index=future_dates[1:],columns=df.columns)
```

In [41]:
```
future_datest_df.tail()
```

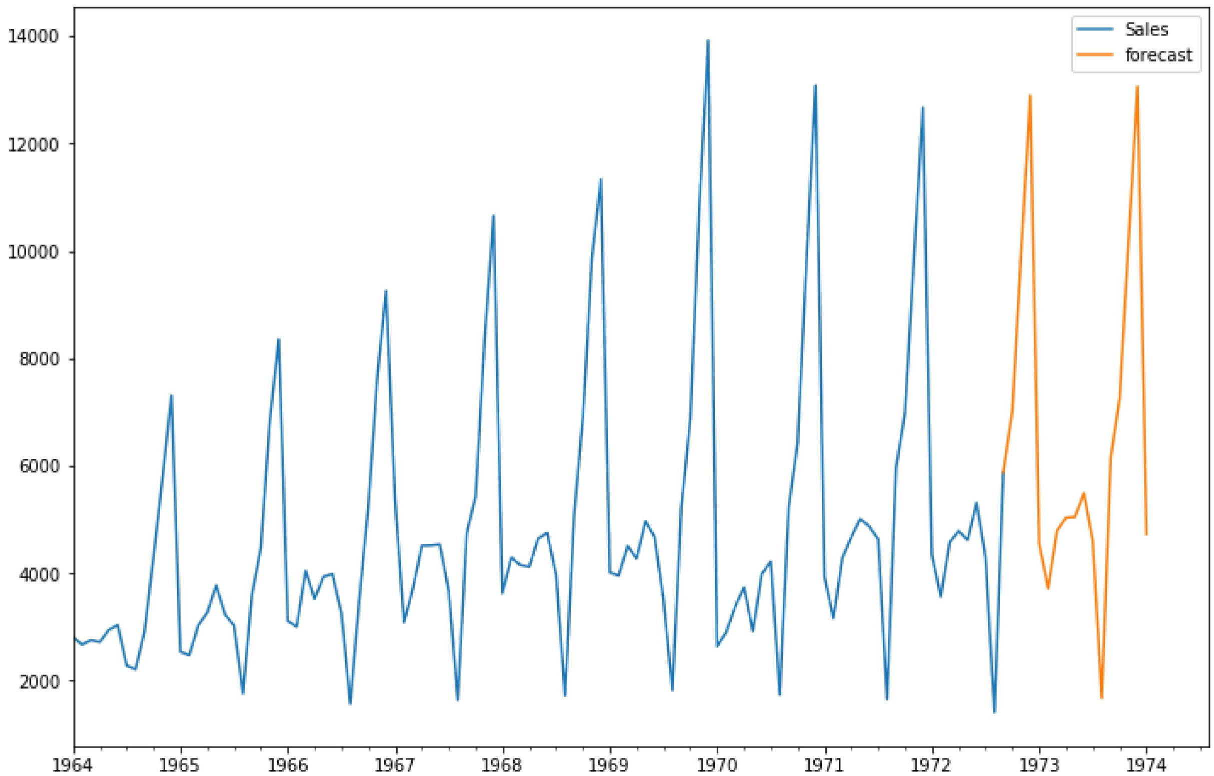
Out[41]:

| | Sales | Sales First Difference | Seasonal First Difference | forecast |
|---|---|---|---|---|
| 1974-04-01 | NaN | NaN | NaN | NaN |
| 1974-05-01 | NaN | NaN | NaN | NaN |
| 1974-06-01 | NaN | NaN | NaN | NaN |
| 1974-07-01 | NaN | NaN | NaN | NaN |
| 1974-08-01 | NaN | NaN | NaN | NaN |

In [42]:
```
future_df=pd.concat([df,future_datest_df])
```

In [43]:
```
future_df['forecast'] = results.predict(start = 104, end = 120, dynamic= True
future_df[['Sales', 'forecast']].plot(figsize=(12, 8))
```

Out[43]: <matplotlib.axes._subplots.AxesSubplot at 0x27f933f4688>




In [ ]: