😊 Time Series Analysis

Time Series Analysis in machine learning is a way to study data that is collected or recorded over time. It helps us understand patterns, trends, and relationships in data that change over time, like stock prices, weather conditions, or sales figures.

Key Concepts:

- **Time-dependent data**: The data points are collected at regular time intervals (daily, monthly, yearly, etc.), and the order of these points is important.
- **Trends**: A general upward or downward movement in the data over time.
- **Seasonality**: Regular patterns that repeat over time, like increased sales during holidays.

Example:

Imagine you're a store owner, and you have daily sales data for the past year. Using time series analysis, you can:

- **Identify trends**: See if sales are generally increasing or decreasing over time.
- **Spot seasonal patterns**: Notice if sales spike during the holiday season every year.
- **Make predictions**: Based on past data, predict future sales to plan inventory.

Time Series Analysis in machine learning focuses on analyzing data points collected or recorded in chronological order. The goal is to discover patterns, trends, and relationships within the data over time, often for the purpose of forecasting future values. Unlike traditional data analysis, time series data has a temporal component, meaning the order in which the data points occur is critical for understanding the underlying dynamics.

Key Components of Time Series Analysis:

1. **Trend**: A long-term increase or decrease in the data. Trends indicate the general direction in which the data is moving over a longer period.
    - *Example*: The gradual increase in global temperatures over the past century.
2. **Seasonality**: Regular and repeating patterns in the data that occur at fixed intervals, typically related to time periods like daily, weekly, monthly, or yearly cycles.
    - *Example*: Retail sales often peak during the holiday season and drop afterward.
3. **Cyclic Patterns**: These are patterns that occur over longer periods but are not as fixed as seasonality. Cycles can be influenced by economic conditions or other external factors.
    - *Example*: Economic cycles like recessions and booms.
4. **Noise**: Random variation in the data that doesn’t follow any clear pattern. Noise represents irregular fluctuations that are hard to predict or model.

Techniques Used in Time Series Analysis:

1. **Autoregressive (AR) models**: Predict future values based on previous values in the series. It assumes the data point depends on its own previous points.
    - *Example*: Predicting the next day’s stock price based on the previous few days.

2. **Moving Average (MA) models**: Predict future values by averaging past observations over a defined period.
    - *Example*: Smoothing out a stock price chart by calculating the average price over the last 7 days.
3. **ARIMA (Autoregressive Integrated Moving Average)**: A more advanced technique that combines AR and MA models and accounts for trends and seasonality in the data.
4. **Exponential Smoothing**: A method that gives more weight to recent observations when predicting future values.
    - *Example*: Predicting energy consumption based on the most recent usage data.

Applications of Time Series Analysis:

- **Stock Market Forecasting**: Predicting future stock prices or market trends based on historical price data.
- **Weather Prediction**: Analyzing past weather data to forecast future conditions.
- **Sales Forecasting**: Using historical sales data to predict future sales, helping businesses manage inventory.
- **Economic Forecasting**: Predicting future economic indicators like GDP, unemployment rates, or inflation based on past trends.

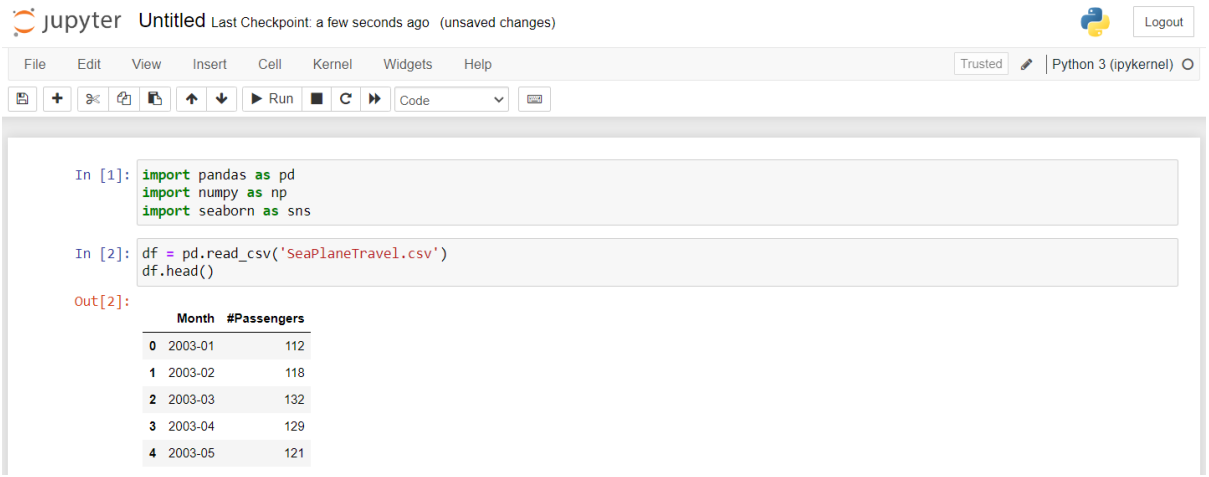
Why Time Series Analysis is Important:

- **Forecasting**: The ability to predict future values based on historical data is one of the most valuable aspects of time series analysis.
- **Pattern Recognition**: Time series analysis helps in identifying hidden patterns like seasonality, trends, and cycles.
- **Decision Making**: Businesses use it for strategic planning, resource allocation, and risk management by understanding trends over time.

In summary, Time Series Analysis is essential for understanding how data evolves over time and making accurate predictions based on past behaviors. Its applications span various industries, including finance, retail, healthcare, and meteorology.

😄 To begin with the Lab:

1. In your Jupyter Notebook upload Sea Plane Travel CSV file and create a new Python Kernel.
2. Then you need to import the important libraries and load your dataset as data frame then display few entries from your dataset.



3. Then we look for any missing values in our dataset but here you can see that there are none.

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

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144 entries, 0 to 143
Data columns (total 2 columns):
 #   Column       Non-Null Count  Dtype 
---  ------       --------------  ----- 
 0   Month        144 non-null    object
 1   #Passengers  144 non-null    int64 
dtypes: int64(1), object(1)
memory usage: 2.4+ KB
```

4. Currently, the month column is read or being treated as the object data type. So, I'm going to process that column in the format of the date time.
5. So now the column month has been read in the format of date time.

In [4]:
```
df = pd.read_csv('SeaPlaneTravel.csv',parse_dates=['Month'])
df.head()
```

Out[4]:

	Month	#Passengers
0	2003-01-01	112
1	2003-02-01	118
2	2003-03-01	132
3	2003-04-01	129
4	2003-05-01	121

In [5]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144 entries, 0 to 143
Data columns (total 2 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Month        144 non-null    datetime64[ns]
 1   #Passengers  144 non-null    int64
dtypes: datetime64[ns](1), int64(1)
memory usage: 2.4 KB
```

6. Next, I will set the month column as the index by using DF.set_index(inplace=True). Additionally, I will rename the column "hash passengers" to simply "passengers."

In [6]:
```
df.set_index("Month",inplace=True)
df.columns = ['Passengers']
df.head()
```

Out[6]:

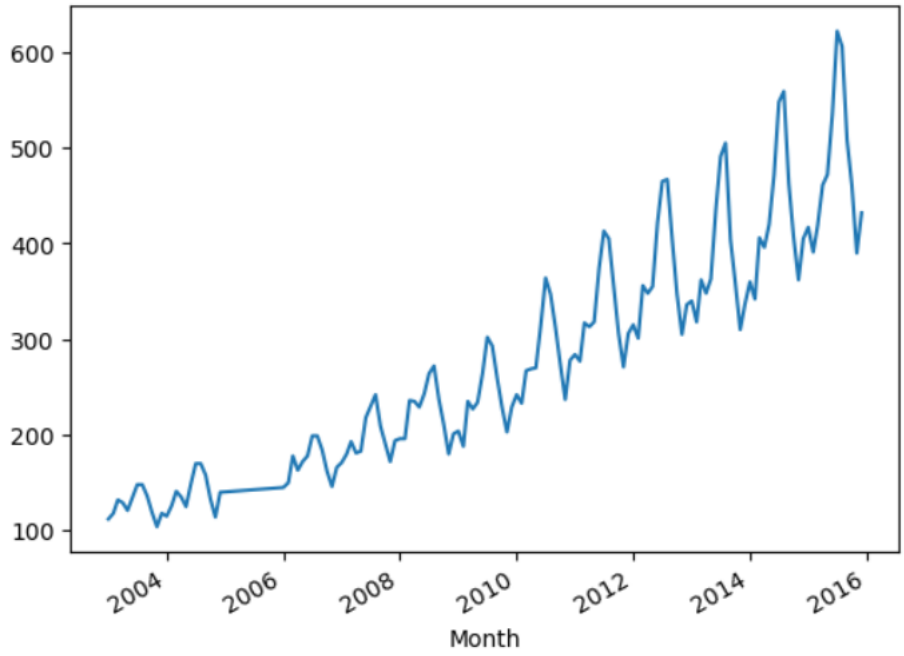
	Passengers
Month	
2003-01-01	112
2003-02-01	118
2003-03-01	132
2003-04-01	129
2003-05-01	121

7. First, we need to determine whether the dataset is stationary or non-stationary. To do this, I will create a visualization, so I'll start by importing the Matplotlib library.
8. From the visual analysis, it's clear that this time series data is non-stationary. Since it is non-stationary, we need to find a method to convert it into a stationary time series.

```
In [7]: import matplotlib.pyplot as plt
        %matplotlib inline

        df['Passengers'].plot()
```

```
Out[7]: <Axes: xlabel='Month'>
```



9. This code performs the Augmented Dickey-Fuller (ADF) test on the 'Passengers' column of the DataFrame df to check for stationarity in the time series data.
    - **ADF Test**: The ADF test is a statistical test used to determine whether a time series is stationary, meaning its properties do not change over time. A stationary series is crucial for many time series analyses and models.
    - **P-Value**: The code prints the p-value from the ADF test results. A low p-value (typically below 0.05) suggests that the null hypothesis (which states that the series has a unit root and is non-stationary) can be rejected, indicating that the time series is likely stationary.

```
In [8]: from statsmodels.tsa.stattools import adfuller
        print(f"The p-value of the given data is {adfuller(df['Passengers'])[1]}")
```

```
The p-value of the given data is 0.991880243437641
```

```
In [ ]: # Since p > 0.05 - Null Hypothesis
        # Given data is a non-stationary data
```

10. This code performs the Augmented Dickey-Fuller (ADF) test on the 'Passengers' data after applying first and second order differencing.

- **First Order Difference**: The first line calculates the first order difference of the 'Passengers' data using .diff(), which computes the difference between each observation and the previous one. The ADF test is then applied to this differenced data (after dropping any missing values with .dropna()). The resulting p-value indicates whether the differenced series is stationary.
- **Second Order Difference**: The second line computes the second order difference by applying the .diff() method twice. Again, the ADF test is performed on this differenced series, and the p-value is printed.
- By comparing these p-values to the original series' p-value, you can assess whether differencing has helped achieve stationarity in the time series data. A low p-value in either case suggests that the differenced data is likely stationary.

```
In [9]: print(f"The p-value of the data after 1st order difference is {adfuller(df['Passengers'].diff().dropna())[1]}")
        The p-value of the data after 1st order difference is 0.05421329028382711

In [10]: print(f"The p-value of the data after 2nd order difference is {adfuller(df['Passengers'].diff().diff().dropna())[1]}")
         The p-value of the data after 2nd order difference is 2.7328918500141235e-29
```

11. Once we've converted the data into a stationary format, we can apply statistical models. It's important to understand the basics of the ARIMA model, which stands for Auto Regressive Integrated Moving Average. In this context, auto-regression refers to expressing a value in the time series as a linear regression equation based on its previous terms. This is the foundation of auto-regression.
12. Now in order for us to find the value of P and Q, in order to apply this ARIMA model, we can make use of a library that's called as pmdarima.

```
In [13]: # Statistical Models
         # ARIMA
         !pip install pmdarima

         Requirement already satisfied: pmdarima in
         Requirement already satisfied: joblib>=0.11
         Requirement already satisfied: Cython!=0.29
         (3.0.11)
```

13. This code uses the pmdarima library to automatically determine the best ARIMA (AutoRegressive Integrated Moving Average) model for the 'Passengers' time series data.
    - **Auto ARIMA**: The auto_arima function searches for the optimal combination of parameters (p, d, q) for the ARIMA model. It evaluates multiple configurations to identify the best-fitting model based on criteria like AIC (Akaike Information Criterion) or BIC (Bayesian Information Criterion).
    - **Trace**: Setting trace=True allows you to see the progress of the search process, including the different parameter combinations being tested and their respective scores.
    - The output will provide insights into the best ARIMA model parameters for the 'Passengers' series, helping you build a more accurate forecasting model.

In [14]:
```
import pmdarima as pm
pm.auto_arima(df['Passengers'],trace=True)
```

```
Performing stepwise search to minimize aic
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.27 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=1415.278, Time=0.01 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=1403.473, Time=0.03 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=1398.827, Time=0.06 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=1413.909, Time=0.01 sec
 ARIMA(1,1,1)(0,0,0)[0] intercept   : AIC=1396.121, Time=0.09 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=inf, Time=0.17 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=inf, Time=0.09 sec
 ARIMA(0,1,2)(0,0,0)[0] intercept   : AIC=1398.386, Time=0.07 sec
 ARIMA(2,1,0)(0,0,0)[0] intercept   : AIC=1397.975, Time=0.04 sec
 ARIMA(1,1,1)(0,0,0)[0]             : AIC=1394.683, Time=0.03 sec
 ARIMA(0,1,1)(0,0,0)[0]             : AIC=1397.258, Time=0.03 sec
 ARIMA(1,1,0)(0,0,0)[0]             : AIC=1401.852, Time=0.02 sec
 ARIMA(2,1,1)(0,0,0)[0]             : AIC=1378.338, Time=0.05 sec
 ARIMA(2,1,0)(0,0,0)[0]             : AIC=1396.588, Time=0.02 sec
 ARIMA(3,1,1)(0,0,0)[0]             : AIC=1379.614, Time=0.09 sec
 ARIMA(2,1,2)(0,0,0)[0]             : AIC=inf, Time=0.17 sec
 ARIMA(1,1,2)(0,0,0)[0]             : AIC=1385.498, Time=0.05 sec
 ARIMA(3,1,0)(0,0,0)[0]             : AIC=1395.021, Time=0.04 sec
 ARIMA(3,1,2)(0,0,0)[0]             : AIC=1377.086, Time=0.13 sec
 ARIMA(4,1,2)(0,0,0)[0]             : AIC=1373.560, Time=0.14 sec
 ARIMA(4,1,1)(0,0,0)[0]             : AIC=1384.053, Time=0.12 sec
 ARIMA(5,1,2)(0,0,0)[0]             : AIC=1375.353, Time=0.14 sec
 ARIMA(4,1,3)(0,0,0)[0]             : AIC=1365.825, Time=0.27 sec
 ARIMA(3,1,3)(0,0,0)[0]             : AIC=inf, Time=0.27 sec
 ARIMA(5,1,3)(0,0,0)[0]             : AIC=inf, Time=0.30 sec
 ARIMA(4,1,4)(0,0,0)[0]             : AIC=inf, Time=0.33 sec
 ARIMA(3,1,4)(0,0,0)[0]             : AIC=inf, Time=0.34 sec
 ARIMA(5,1,4)(0,0,0)[0]             : AIC=inf, Time=0.31 sec
 ARIMA(4,1,3)(0,0,0)[0] intercept   : AIC=inf, Time=0.30 sec

Best model:  ARIMA(4,1,3)(0,0,0)[0]
Total fit time: 4.011 seconds
```

Out[14]:

ARIMA
ARIMA(4,1,3)(0,0,0)[0]

14. This code is working with the ARIMA model using the statsmodels library to analyze and forecast the 'Passengers' time series data.
    - **Setting the Time Series**: The code first selects the 'Passengers' column from the DataFrame and assigns it to the variable ts, representing the time series data.
    - **Model Initialization**: An ARIMA model is specified with the order (4, 1, 3), where:
      4 is the number of autoregressive terms (p),
      1 is the number of differencing (d),
      3 is the number of moving average terms (q).
    - **Fitting the Model**: The model is then fitted to the time series data using the fit() method, which estimates the model parameters.

- **Forecasting**: The forecast() method is called to make future predictions based on the fitted model. The default forecast horizon is typically set to the next few time steps, but the specific output will depend on the model's settings.
- **Predicting Specific Steps**: The predict() method is used to obtain predicted values for specific time steps, in this case from step 144 to step 155 in the time series.
- Overall, this code allows you to build an ARIMA model for forecasting future values in the 'Passengers' time series data.

In [15]:
```
import statsmodels.api as sm
ts = df['Passengers']
ts
```

Out[15]:
```
Month
2003-01-01    112
2003-02-01    118
2003-03-01    132
2003-04-01    129
2003-05-01    121
             ...
2015-08-01    606
2015-09-01    508
2015-10-01    461
2015-11-01    390
2015-12-01    432
Name: Passengers, Length: 144, dtype: int64
```

In [16]:
```
model = sm.tsa.arima.ARIMA(ts, order=(4,1,3))
result = model.fit()
```

```
C:\Users\PULKIT\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarning: A date index has been provide
d, but it has no associated frequency information and so will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
C:\Users\PULKIT\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarning: A date index has been provide
d, but it has no associated frequency information and so will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
C:\Users\PULKIT\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:473: ValueWarning: A date index has been provide
d, but it has no associated frequency information and so will be ignored when e.g. forecasting.
  self._init_dates(dates, freq)
C:\Users\PULKIT\anaconda3\Lib\site-packages\statsmodels\tsa\statespace\sarimax.py:966: UserWarning: Non-stationary starting aut
oregressive parameters found. Using zeros as starting parameters.
  warn('Non-stationary starting autoregressive parameters'
C:\Users\PULKIT\anaconda3\Lib\site-packages\statsmodels\tsa\statespace\sarimax.py:978: UserWarning: Non-invertible starting MA
parameters found. Using zeros as starting parameters.
  warn('Non-invertible starting MA parameters found.'
```

In [17]:
```
result.forecast()
```

```
C:\Users\PULKIT\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836: ValueWarning: No supported index is availabl
e. Prediction results will be given with an integer index beginning at `start`.
  return get_prediction_index(
C:\Users\PULKIT\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836: FutureWarning: No supported index is availab
le. In the next version, calling this method in a model without a supported index will result in an exception.
  return get_prediction_index(
```

Out[17]:
```
144    467.573788
dtype: float64
```

In [18]:
```
result.predict(144,155)
```

```
C:\Users\PULKIT\anaconda3\Lib\site-packages\statsmodels\tsa\base\tsa_model.py:836: ValueWarning: No supported index is availabl
e. Prediction results will be given with an integer index beginning at `start`.
  return get_prediction_index(
```

Out[18]:
```
144    467.573788
145    490.494532
146    509.136912
147    492.554745
148    495.305961
149    475.947806
150    476.339843
151    475.552141
152    472.353819
153    483.889665
154    475.570182
155    485.921529
Name: predicted_mean, dtype: float64
```