**TIME SERIES**

Time Series is a collection of data points over a specified period of time recorded at regular intervals.

OR

A time series is a sequence of observations taken sequentially in time.

**Two Steps Process:** Analysis and Forecasting

**Time series *analysis*** comprises methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data. **Time series *forecasting*** is the use of a [model](https://en.wikipedia.org/wiki/Model_(abstract)) to predict future values based on previously observed values

**Data Analysis and Preprocessing**

Time series analysis can be applied to [real-valued](https://en.wikipedia.org/wiki/Real_number), continuous data, [discrete](https://en.wiktionary.org/wiki/discrete) [numeric](https://en.wikipedia.org/wiki/Data_type#Numeric_types) data, or discrete symbolic data (i.e. sequences of characters, such as letters and words in the [English language](https://en.wikipedia.org/wiki/English_language))

Methods for time series analysis may be divided into two classes:

* [frequency-domain](https://en.wikipedia.org/wiki/Frequency-domain) methods
* [time-domain](https://en.wikipedia.org/wiki/Time-domain) methods.

**Python code of reading any time series data ->**

Steps to be followed when we get the raw data:

* Read and convert the data into time series format.

Example Code:

data = pd.read\_csv('Data.csv', parse\_dates=['Date\_Col'], index\_col='Date\_Col',date\_parser=dateparse)

dateparse = lambda dates: pd.datetime.strptime(dates, '%Y-%m')

Explanation:

1. parse\_dates: This specifies the column which contains the date-time information. As we say above, the column name is ‘Month’.
2. index\_col: A key idea behind using Pandas for TS data is that the index has to be the variable depicting date-time information. So this argument tells pandas to use the ‘Month’ column as index.
3. date\_parser: This specifies a function which converts an input string into datetime variable. Be default Pandas reads data in format ‘YYYY-MM-DD HH:MM:SS’. If the data is not in this format, the format has to be manually defined. Something similar to the dataparse function defined here can be used for this purpose.

* Plot the data.

Example Code:

import matplotlib.pylab as plt

plt.plot(data)

Now seeing the graph, we might get some idea of the data provided.

**Now we start the Exploratory Data Analysis for the given data:**

Questions to be answered while EDA:

* Is it Multivariate or Univariate?
* What is our objective or target column?
* What are the redundant columns, continuous and categorical columns?
* What is possible frequency of time in data?
* What are the outliers present in the dataset and how to handle them?
* What are the missing values or gaps that are needed to be interpolated or imputed and how?
* Is the data stationary or non-stationary?
* What is the trend and seasonality if present in the data?
* What is the relation between the important columns/variables?
* What is feature engineering, when it is required?

Answer to all these questions depends upon the domain knowledge and knowledge of how better we could understand the data.

**Multivariate or Univariate:**

A univariate time series is a series with a single time-dependent variable.

A Multivariate time series has more than one time-dependent variable. Each variable depends not only on its past values but also has some dependency on other variables. This dependency is used for forecasting future values.

**Frequency of Data:**

Data can be available in any frequency like Daily, Hourly, Weekly, Monthly, Yearly etc.

This calls for resampling of data if the frequency is too high or too low or is uneven, then it is required to up sample or down sample the data i.e. frequency aggregations. We do down sampling by aggregating from a smaller time unit to larger and vice versa. This can be done by using group by in python.

**Stationarity of Time Series Data:**

A Time Series is said to be stationary if its statistical properties such as mean and variance remain constant over time.

The series is stationary if it has the following constant statistical properties over time:

1. constant mean
2. constant variance
3. an auto-covariance that does not depend on time.

To check the stationarity of data:

Univariate:

Dickey-Fuller Test:

This is one of the statistical tests for checking stationarity. Here the null hypothesis is that the Time Series is non-stationary. The test results comprise of a Test Statistic and some Critical Values for difference confidence levels. If the ‘Test Statistic’ is less than the ‘Critical Value’, we can reject the null hypothesis and say that the series is stationary.

Multivariate:

Johansen Co-Integration Test:

It allows us to analyze whether two or more time series can form a co-integrating relationship. For a series to be stationary, the eigenvalues should be less than 1 in modulus. This test works for only 12 variables at a time.

Most of the time series model works on stationary data, so we try to make the given time series data stationary. There are 2 major reasons behind non-stationarity of a Time Series:  
1. **Trend** – varying mean over time.  
2. **Seasonality** – variations at specific time-frames.

 Two ways of removing trend and seasonality:

1. Differencing –

Taking the difference with a particular time lag. In this technique, we take the difference of the observation at a particular instant with that at the previous instant. This mostly works well in improving stationarity.

1. Decomposition –

Modeling both trend and seasonality and removing them from the model. The underlying principle is to model or estimate the trend and seasonality in the series and remove those from the series to get a stationary series. Then after implementing statistical forecasting techniques, we can convert the forecasted values into the original scale by applying trend and seasonality constraints back.

**Components of Time Series Data:**

Time series analysis provides a body of techniques to better understand a dataset. Perhaps the most useful of these is the decomposition of a time series into 4 constituent parts:

* Level- The baseline or average value for the series.
* Trend- The optional and often linear increasing or decreasing behavior of the series over time.
* Seasonality- The optional repeating patterns or cycles of behavior over time.
* Noise- The optional variability in the observations that cannot be explained by the model. (Residual)

All time series have a level, most have noise, and the trend and seasonality are optional.

**Two Models are** used for decomposition:

Additive **=** level + trend + seasonality + noise

An additive model is linear where changes over time are consistently made by the same amount.

A linear trend is a straight line.

A linear seasonality has the same frequency (width of cycles) and amplitude (height of cycles).

Multiplicative = level \* trend \* seasonality \* noise

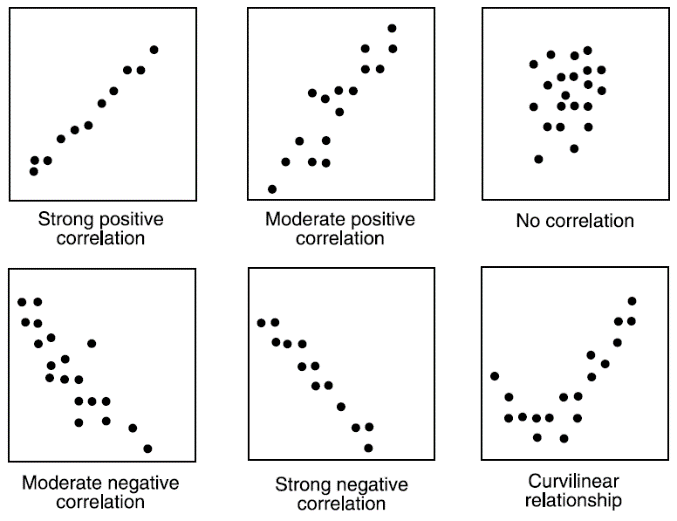
A multiplicative model is nonlinear, such as quadratic or exponential. Changes increase or decrease over time.

A nonlinear trend is a curved line.

A non-linear seasonality has an increasing or decreasing frequency and/or amplitude over time.

**Find relation between the variables(columns):**

Scatter Plot -> The pattern of scatter plot indicates the relationship between variables. The relationship can be linear or non-linear.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/02/Data_exploration_4.png)

Scatter plot shows the relationship between two variable but does not indicates the strength of relationship amongst them.

Correlation -> To find the strength of the relationship, we use Correlation. Correlation varies between -1 and +1.

* -1: perfect negative linear correlation
* +1: perfect positive linear correlation and
* 0: No correlation

**Handle Missing Values:**

The simplest way is to impute the data with mean, median or mode. But this approach does not always go well with times series data.

Example Code:

library(imputeTS)

na.mean(mydata, option = "mean") # Mean Imputation  
na.mean(mydata, option = "median") # Median Imputation  
na.mean(mydata, option = "mode") # Mode Imputation

In Python

from sklearn.preprocessing import Imputer  
values = mydata.values  
imputer = Imputer(missing\_values=’NaN’, strategy=’mean’)  
transformed\_values = imputer.fit\_transform(values)

# strategy can be changed to "median" and “most\_frequent”

In time series, missing values can be divided into four categories:

* Without trend and seasonality
* With trend and without seasonality
* Without trend and with seasonality
* With trend and seasonality

Time-Series specific method ->

* Last observation carried forward (LOCF): Last Observation Carried Forward
* Next observation carried backward (NOCB): Next Observation Carried Backward
* Linear interpolation: This method works well for a time series with some trend but is not suitable for seasonal data.
* Seasonal Adjustment + Linear Interpolation: This method works well for data with both trend and seasonality. We de-seasonalize the data first, and then do interpolation on the data.  
  Once the missing values are imputed, we need to re-seasonalize the data.

Example Code in R:

library(imputeTS)

na.random(mydata) # Random Imputation  
na.locf(mydata, option = "locf") # Last Obs. Carried Forward  
na.locf(mydata, option = "nocb") # Next Obs. Carried Backward  
na.interpolation(mydata) # Linear Interpolation  
na.seadec(mydata, algorithm = "interpolation") # Seasonal Adjustment then Linear Interpolation

**Outlier Detection and Handling:**

Outlier is an observation that appears far away and diverges from an overall pattern in a sample. Most commonly used method to detect outliers is visualization. We use various visualization methods, like Box-plot, Histogram, Scatter Plot.

There are many methods of outlier detection and handling which will be discussed further. (Read More)

Visualization Techniques to detect outliers are:

**Box-Plot –** Outliers are those observations that lie outside the range of -1.5 x IQR to 1.5 x IQR, where IQR, the ‘Inter Quartile Range’ is the difference between 75th and 25th quartiles. This is also known as “The Box Plot Rule”. The box plot rule is the simplest statistical technique that has been applied to detect univariate outliers.

1. Get the Interquartile Range, IQR = Q3 - Q1 where,

Q1 is defined as the middle number between the smallest number and the median of the data set.

Q3 is the middle value between the median and the highest value of the data set.

1. Calculate the upper and lower values:

Min = Q1 – (IQR\*1.5)

Max = Q3 + (IQR\*1.5)

So, the value less then minimum in the data or more than maximum are considered to be the outliers.

Python Code:

import seaborn as sns  
sns.boxplot(x=data)

**Scatter-Plot -**  The scatter plot is the collection of points that shows values for two variables.

Python Code:

fig, ax = plt.subplots(figsize=(16,8))  
ax.scatter(df['A’], df['B])  
ax.set\_xlabel(‘A’)  
ax.set\_ylabel(‘B’)  
plt.show()

Mathematical functions to detect outliers are:

**Z-Score:** (Univariate, Parametric)

The Z-score describe any data point by finding their relationship with the Standard Deviation and Mean of the group of data points. Z-score is finding the distribution of data where mean is 0 and standard deviation is 1 i.e. normal distribution. While calculating the Z-score, we re-scale and center the data and look for data points which are too far from zero. These data points which are way too far from zero will be treated as the outliers. In most of the cases a threshold of 3 or -3 is used i.e. if the Z-score value is greater than or less than 3 or -3 respectively, that data point will be identified as outliers.

Python Code:

from scipy import stats  
import numpy as np

z = np.abs(stats.zscore(df))  
print(z)

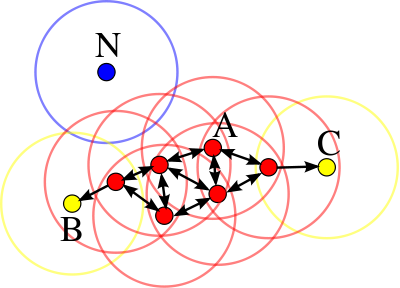
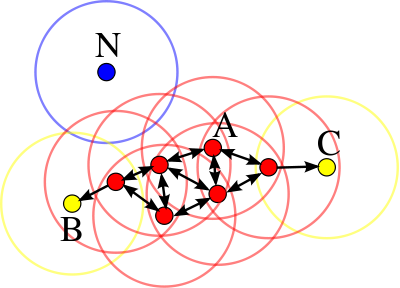
threshold = 3  
print(np.where(z > threshold))

**DBSCAN**: (Multivariate, Non-Parametric) Density Based Spatial Clustering of Applications with Noise

DBSCAN groups together points that are close to each other based on a distance measurement (usually Euclidean distance) and a minimum number of points.

Dbscan then defines different classes of points:

* Core point: A is a core point if its neighborhood contains at least the same number or more points than the parameter MinPts.
* Border point: C is a border point that lies in a cluster and its neighborhood does not contain more points than MinPts, but it is still ‘density reachable’ by other points in the cluster.
* Outlier: N is an outlier point that lies in no cluster and it is not ‘density reachable’ nor ‘density connected’ to any other point. Thus this point will have “his own cluster”.



1. eps: The minimum distance between two points. It means that if the distance between two points is lower or equal to this value (eps), these points are considered neighbors.

If the eps value chosen is too small, a large part of the data will not be clustered. It will be considered outliers because it won’t satisfy the number of points to create a dense region. On the other hand, if the value that was chosen is too high, clusters will merge and the majority of objects will be in the same cluster. The eps should be chosen based on the distance of the dataset, but in general small eps values are preferable.

1. minPoints: The minimum number of points to form a dense region. For example, if we set the minPoints parameter as 5, then we need at least 5 points to form a dense region.  A minimum minPoints can be derived from a number of dimensions (D) in the data set, as minPoints ≥ D + 1. Larger values are usually better for data sets with noise and will form more significant clusters. The minimum value for the minPoints must be 3, but the larger the data set, the larger the minPoints value that should be chosen.

Outliers (noise) will be assigned to the -1 cluster.

Python Code:

import dbscan

dbscan.dbscan(data, eps, min\_points)

Output = dbscan.NOISE (Outliers)

OR

from sklearn.cluster import DBSCAN

DB = DBSCAN(eps=2.5, min\_samples=3)

DB.fit(Data)

from collections import Counter

print Counter(DB.labels\_),’n’

print df[DB.labels\_==-1]

There are many other methods like Isolation Forest(Multivariate), Grubb’s Test(Univariate). Several methods are used to identify outliers in multivariate datasets. Two of the widely used methods are Mahalanobis Distance andCook’s Distance. (Need to explore further).

We can handle outliers by:

1. **Deleting** - We delete outlier values if it is due to data entry error, data processing error or outlier observations are very small in numbers. We can also use trimming at both ends to remove outliers.
2. **Binning** - Transforming variables can also eliminate outliers. Natural log of a value reduces the variation caused by extreme values. Binning is also a form of variable transformation. Decision Tree algorithm allows to deal with outliers well due to binning of variable. We can also use the process of assigning weights to different observations.
3. **Imputing** - Like [imputation of missing values](https://www.analyticsvidhya.com/blog/2015/02/7-steps-data-exploration-preparation-building-model-part-2/), we can also impute outliers. We can use mean, median, mode imputation methods. Before imputing values, we should analyze if it is natural outlier or artificial. If it is artificial, we can go with imputing values. We can also use statistical model to predict values of outlier observation and after that we can impute it with predicted values.
4. **Treating Separately** - If there are significant number of outliers, we should treat them separately in the statistical model. One of the approach is to treat both groups as two different groups and build individual model for both groups and then combine the output.

**Feature Engineering:**

Feature engineering is the science (and art) of extracting more information from existing data. You are not adding any new data here, but you are actually making the data you already have more useful. Bringing out information from data in known as feature engineering.

Feature engineering itself can be divided in 2 steps:

* Variable transformation - In data modelling, transformation refers to the replacement of a variable by a function. For instance, replacing a variable x by the square / cube root or logarithm x is a transformation. In other words, transformation is a process that changes the distribution or relationship of a variable with others.

We do variable transformation when:

* 1. We want to change the scale of the variable for better understanding.
  2. Transform complex non-linear relationships into linear ones by taking logs.
  3. To reduce skewness of the data and convert into symmetric/normal distribution.

We can do this by:

* Logarithm: Log of a variable is a common transformation method used to change the shape of distribution of the variable on a distribution plot. It is generally used for reducing right skewness of variables. Though, This can’t be applied to zero or negative values as well.
* Square / Cube root: The square and cube root of a variable has a sound effect on variable distribution. However, it is not as significant as logarithmic transformation. Cube root has its own advantage. It can be applied to negative values including zero. Square root can be applied to positive values including zero.
* Binning: It is used to categorize variables. It is performed on original values, percentile or frequency. Decision of categorization technique is based on business understanding. For example, we can categorize income in three categories, namely: High, Average and Low. We can also perform co-variate binning which depends on the value of more than one variables.
* Variable / Feature creation.

Feature creation is a process to generate a new variables based on existing variables. For example, say, we have date as an input variable in a data set. We can generate new variables like day, month, year, week, weekday that may have better relationship with target variable. This step is used to highlight the hidden relationship in a variable. This also includes converting categorical into numerical by one hot encoding. This technique also highly relies on the domain knowledge of the data analyst.

**Data Modelling and Forecasting**

**Parametric or Non-Parametric Model:**

**Parametric -** A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a parametric model. No matter how much data you throw at a parametric model, it won’t change its mind about how many parameters it needs. Ex- Linear Regression (GLM). Basically used for simpler, cheaper and lesser data with less complexity and high speed.

**Non-Parametric -** Nonparametric methods are good when you have a lot of data and no prior knowledge, and when you don’t want to worry too much about choosing just the right features. Ex- Random Forest(RDF). It is more flexible and works on larger datasets with better performance.

There are numerous ways for time series forecasting. We will start with the most basic one and continue to cover all known approaches.

**Naive Approach:** In this forecasting technique, the value of the new data point is predicted to be equal to the previous data point. The result would be a flat line, since all new values take the previous values.

**Simple Average:** The next value is taken as the average of all the previous values. The predictions here are better than the ‘Naive Approach’ as it doesn’t result in a flat line but here, all the past values are taken into consideration which might not always be useful. For instance, when asked to predict today’s temperature, you would consider the last 7 days’ temperature rather than the temperature a month ago.

**Moving Average:** This is an improvement over the previous technique. Instead of taking the average of all the previous points, the average of ‘n’ previous points are taken to be the predicted value.

**Weighted Moving Average:**A weighted moving average is a moving average where the past ‘n’ values are given different weights.

**Simple Exponential Smoothing:** In this technique, larger weights are assigned to more recent observations than to observations from the distant past.

**Holt’s Linear Trend Model:** This method takes into account the trend of the dataset. By trend, we mean the increasing or decreasing nature of the series. Suppose the number of bookings in a hotel increases every year, then we can say that the number of bookings show an increasing trend. The forecast function in this method is a function of level and trend.

**Holt Winters Method:** This algorithm takes into account both the trend and the seasonality of the series. For instance – the number of bookings in a hotel is high on weekends & low on weekdays, and increases every year; there exists a weekly seasonality and an increasing trend.

**ARIMA (Auto-Regressive Integrated Moving Averages):** ARIMA is a very popular statistical method for time series forecasting. ARIMA models work on the following assumptions –

* The data series is stationary, which means that the mean and variance should not vary with time. A series can be made stationary by using log transformation or differencing the series.
* The data provided as input must be a univariate series, since ARIMA uses the past values to predict the future values.

ARIMA has three components – AR (autoregressive term), I (differencing term) and MA (moving average term). Let us understand each of these components –

* AR term refers to the past values used for forecasting the next value. The AR term is defined by the parameter ‘p’ in ARIMA. The value of ‘p’ is determined using the PACF plot. AR terms are just lags of dependent variable. For instance if p is 5, the predictors for x(t) will be x(t-1)….x(t-5).
* MA term is used to defines number of past forecast errors used to predict the future values. The parameter ‘q’ in ARIMA represents the MA term. ACF plot is used to identify the correct ‘q’ value. MA terms are lagged forecast errors in prediction equation. For instance if q is 5, the predictors for x(t) will be e(t-1)….e(t-5) where e(i) is the difference between the moving average at ith instant and actual value.
* Order of differencing specifies the number of times the differencing operation is performed on series to make it stationary. For making the series stationary, the number of times the difference operation was performed will be taken as the d value.

**SARIMA:**

If our model has a seasonal component, we use a seasonal ARIMA model (SARIMA). In that case we have another set of parameters: **P, D,**and **Q** which describe the same associations as p, d, and q, but correspond with the seasonal components of the model.

**Auto-ARIMA:**

In this model, p, d and q values are chosen automatically. Auto ARIMA takes into account the AIC and BIC values generated to determine the best combination of parameters. AIC (Akaike Information Criterion) and BIC (Bayesian Information Criterion) values are estimators to compare models. The lower these values, the better is the model.

**VAR (Vector Auto Regression):**

VAR is used for prediction of multivariate time series. It is able to understand and use the relationship between several variables. This is useful for describing the dynamic behavior of the data and also provides better forecasting results. Similar to the Augmented Dickey-Fuller test for univariate series, we have Johansen’s test for checking the stationarity of any multivariate time series data. For a series to be stationary, the eigenvalues should be less than 1 in modulus. This test works for only 12 variables at a time.

Python Code:

# to check stationarity

from statsmodels.tsa.vector.ar.vecm import coint\_johansen

coint\_johansen(data,-1,1).eig

Ref -https://www.statsmodels.org/dev/generated/statsmodels.tsa.vector\_ar.vecm.coint\_johansen.html

#fit the model

from statsmodel.tsa.vector\_ar.var\_mpdel import VAR

model = VAR(endog=train)

model\_fit = model.fit()

prediction = model\_fit.forecast(model\_fit.y, steps= len(test))

**LSTM (Long Short Term Memory):**

RNN is recurrent neural network basically works on feed-forward neural networks where all test cases are considered to be independent. These are good when it comes to short context but for larger datasets, it is not very useful. The main reason behind this is Vanishing Gradient. For a conventional feed-forward neural network, the weight updating that is applied on a particular layer is a multiple of the learning rate, the error term from the previous layer and the input to that layer. Thus, the error term for a particular layer is somewhere a product of all previous layers’ errors.

As a result of this, the gradient almost vanishes as we move towards the starting layers, and it becomes difficult to train these layers.

LSTMs on the other hand, make small modifications to the information by multiplications and additions. With LSTMs, the information flows through a mechanism known as cell states. This way, LSTMs can selectively remember or forget things. The information at a particular cell state has three different dependencies.

These dependencies can be generalized to any problem as:

1. The previous cell state (i.e. the information that was present in the memory after the previous time step)
2. The previous hidden state (i.e. this is the same as the output of the previous cell)
3. The input at the current time step (i.e. the new information that is being fed in at that moment)

A typical LSTM network is comprised of different memory blocks called cells. There are two states that are being transferred to the next cell; the cell state and the hidden state. The memory blocks are responsible for remembering things and manipulations to this memory is done through three major mechanisms, called gates.

* Forget Gate: A forget gate is responsible for removing information from the cell state.
* Input Gate: The input gate is responsible for the addition of information to the cell state.
* Output Gate: This job of selecting useful information from the current cell state and showing it out as an output is done via the output gate.

**CODE**

**Python code of reading any time series data ->**

Steps to be followed when we get the raw data:

* Read and convert the data into time series format.

Example Code:

data = pd.read\_csv('Data.csv', parse\_dates=['Date\_Col'], index\_col='Date\_Col',date\_parser=dateparse)

dateparse = lambda dates: pd.datetime.strp time(dates, '%Y-%m')

Explanation:

1. parse\_dates: This specifies the column which contains the date-time information. As we say above, the column name is ‘Month’.
2. index\_col: A key idea behind using Pandas for TS data is that the index has to be the variable depicting date-time information. So this argument tells pandas to use the ‘Month’ column as index.
3. date\_parser: This specifies a function which converts an input string into datetime variable. Be default Pandas reads data in format ‘YYYY-MM-DD HH:MM:SS’. If the data is not in this format, the format has to be manually defined. Something similar to the dataparse function defined here can be used for this purpose.

* Plot the data.

Example Code:

import matplotlib.pylab as plt

plt.plot(data)

Now seeing the graph, we might get some idea of the data provided.

Example Code:

In R

library(imputeTS)

na.mean(mydata, option = "mean") # Mean Imputation  
na.mean(mydata, option = "median") # Median Imputation  
na.mean(mydata, option = "mode") # Mode Imputation

In Python

from sklearn.preprocessing import Imputer  
values = mydata.values  
imputer = Imputer(missing\_values=’NaN’, strategy=’mean’)  
transformed\_values = imputer.fit\_transform(values)

# strategy can be changed to "median" and “most\_frequent”

Example Code in R:

library(imputeTS)

na.random(mydata) # Random Imputation  
na.locf(mydata, option = "locf") # Last Obs. Carried Forward  
na.locf(mydata, option = "nocb") # Next Obs. Carried Backward  
na.interpolation(mydata) # Linear Interpolation  
na.seadec(mydata, algorithm = "interpolation") # Seasonal Adjustment then Linear Interpolation

Outliers

Python Code:

import seaborn as sns  
sns.boxplot(x=data)

Python Code:

fig, ax = plt.subplots(figsize=(16,8))  
ax.scatter(df['A’], df['B])  
ax.set\_xlabel(‘A’)  
ax.set\_ylabel(‘B’)  
plt.show()

z-score :

Python Code:

from scipy import stats  
import numpy as np

z = np.abs(stats.zscore(df))  
print(z)

threshold = 3  
print(np.where(z > threshold))

Python Code:

import dbscan

dbscan.dbscan(data, eps, min\_points)

Output = dbscan.NOISE (Outliers)

OR

from sklearn.cluster import DBSCAN

DB = DBSCAN(eps=2.5, min\_samples=3)

DB.fit(Data)

from collections import Counter

print Counter(DB.labels\_),’n’

print df[DB.labels\_==-1]

VAR:

Python Code:

# to check stationarity

from statsmodels.tsa.vector.ar.vecm import coint\_johansen

coint\_johansen(data,-1,1).eig

Ref -https://www.statsmodels.org/dev/generated/statsmodels.tsa.vector\_ar.vecm.coint\_johansen.html

#fit the model

from statsmodel.tsa.vector\_ar.var\_mpdel import VAR

model = VAR(endog=train)

model\_fit = model.fit()

prediction = model\_fit.forecast(model\_fit.y, steps= len(test))