https://www.datacamp.com/community/tutorials/time-series-analysis-tutorial
https://machinelearningmastery.com/time-series-forecasting-methods-in-python-cheat-sheet/
https://www.kaggle.com/rohanrao/a-modern-time-series-tutorial
https://www.kaggle.com/thebrownviking20/everything-you-can-do-with-a-time-series
https://www.kaggle.com/census/population-time-series-data

https://www.kaggle.com/kashnitsky/topic-9-part-1-time-series-analysis-in-python

Libraries

```
import warnings                                          # `do not disturbe` mode
warnings.filterwarnings('ignore')

import numpy as np                                       # vectors and matrices
import pandas as pd                                      # tables and data manipulations
import matplotlib.pyplot as plt                          # plots
import seaborn as sns                                    # more plots

from dateutil.relativedelta import relativedelta # working with dates with style
from scipy.optimize import minimize                      # for function minimization

import statsmodels.formula.api as smf                    # statistics and econometrics
import statsmodels.tsa.api as smt
import statsmodels.api as sm
import scipy.stats as scs

from itertools import product                            # some useful functions
from tqdm import tqdm_notebook



%matplotlib inline
```

Various TimeStamp / Date Operations / Hypothesis Testing

Checking if the given timestamp exists in the given period

```
timestamp = pd.Timestamp(2017, 1, 1, 12)
period = pd.Period('2017-01-01')
period.start_time < timestamp < period.end_time
```

Converting timestamp to period

```
new_period = timestamp.to_period(freq='H')
new_period
>> Period('2017-01-01 12:00', 'H')
```

Converting period to timestamp

```
new_timestamp = period.to_timestamp(freq='H', how='start')
new_timestamp
>> Timestamp('2017-01-01 00:00:00')
```

date_range is a method that returns a fixed frequency datetimeindex. It is quite useful when creating your own time series attribute for pre-existing data or arranging the whole data around the time series attribute created by you.

```
# Creating a datetimeindex with daily frequency
dr1 = pd.date_range(start='1/1/18', end='1/9/18')
dr1
```

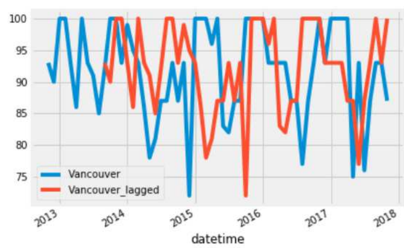
```
>> DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08',
               '2018-01-09'],

# Creating a datetimeindex with monthly frequency
dr2 = pd.date_range(start='1/1/18', end='1/1/19', freq='M')
dr2
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-30',
               '2018-05-31', '2018-06-30', '2018-07-31', '2018-08-31',
               '2018-09-30', '2018-10-31', '2018-11-30', '2018-12-31'],
              dtype='datetime64[ns]', freq='M')

# Creating a datetimeindex without specifying start date and using periods
dr3 = pd.date_range(end='1/4/2014', periods=8)
dr3
DatetimeIndex(['2013-12-28', '2013-12-29', '2013-12-30', '2013-12-31',
               '2014-01-01', '2014-01-02', '2014-01-03', '2014-01-04'],
              dtype='datetime64[ns]', freq='D')

# Creating a datetimeindex specifying start date , end date and periods
dr4 = pd.date_range(start='2013-04-24', end='2014-11-27', periods=3)
dr4
DatetimeIndex(['2013-04-24', '2014-02-09', '2014-11-27'], dtype='datetime64[ns]', freq=None)
```

To_datetime

```
df = pd.to_datetime('01-01-2017')
df
>> Timestamp('2017-01-01 00:00:00')
```

.shift()

```
# Lags
humidity["Vancouver"].asfreq('M').plot(legend=True)
shifted = humidity["Vancouver"].asfreq('M').shift(10).plot(legend=True)
shifted.legend(['Vancouver','Vancouver_lagged'])
```



Resampling

Upsampling - Time series is resampled from low frequency to high frequency(Monthly to daily frequency). It involves filling or interpolating missing data

Downsampling - Time series is resampled from high frequency to low frequency(Weekly to monthly frequency). It involves aggregation of existing data.

We **downsample** from hourly to 3 day frequency aggregated using mean

```
pressure = pressure.resample('3D').mean()
pressure.head()
```

Upsample from 3 day frequency to daily frequency

```
pressure = pressure.resample('D').pad()
```

Stock Applications

Percent Change from previous value

```
# Google Stock price today / stock price yesterday
google['Change'] = google.High.div(google.High.shift())
google['Change'].plot(figsize=(20,8))
```

Stock Returns

google['Change'] = google.High.div(google.High.shift()) google['Return'] = google.Change.sub(1).mul(100) google['Return'].plot(figsize=(20,8)) OR google.High.pct_change().mul(100).plot(figsize=(20,6)) **# Absolute change in successive rows** google.High.diff().plot(figsize=(20,6))
Comparing two or more time series # Mere comparison without normalization google.High.plot() microsoft.High.plot() plt.legend(['Google','Microsoft']) # Comparison with normalization normalized_google = google.High.div(google.High.iloc[0]).mul(100) normalized_microsoft = microsoft.High.div(microsoft.High.iloc[0]).mul(100) normalized_google.plot() normalized_microsoft.plot() plt.legend(['Google','Microsoft'])
Window Functions # Rolling window functions (**Rolling Mean**) rolling_google = google.High.rolling('90D').mean() google.High.plot() rolling_google.plot() plt.legend(['High','Rolling Mean']) **# Expanding(n) : does some operation with n previous values and itself** microsoft_mean = microsoft.High.expanding(10).mean() # mean of 10 previous values and itself microsoft_std = microsoft.High.expanding(10).std() # std of 10 previosu values and itself microsoft.High.plot() microsoft_mean.plot() microsoft_std.plot() plt.legend(['High','Expanding Mean','Expanding Standard Deviation'])
Filter only weekends df = generate_sample_data_datetime() df.shape df.head() # Step 1: resample by D. Basically agregate by day and use to_frame() to convert it to frame daily_sales = df.resample("D")["sales"].sum().to_frame() daily_sales # Step 2: filter weekends **weekends_sales = daily_sales[daily_sales.index.dayofweek.isin([5, 6])]** weekends_sales ''' dayofweek day 0 Monday 1 Tuesday 2 Wednesday 3 Thursday 4 Friday 5 Saturday 6 Sunday '''

Creating Time Series Dummy Data

```
# Solution 1
number_or_rows = 365*24 # hours in a year
pd.util.testing.makeTimeDataFrame(number_or_rows, freq="H")

# Solution 2
num_cols = 2
cols = ["sales", "customers"]
df =
pd.DataFrame(np.random.randint(1, 20, size = (number_or_rows, num_cols)), columns=cols)

df.index =
pd.util.testing.makeDateIndex(number_or_rows, freq="H")
df
```

	A	B	C	D
2000-01-01 00:00:00	0.548268	-1.810959	-0.197603	-0.223416
2000-01-01 01:00:00	-0.514501	0.407318	-0.108549	1.384783
2000-01-01 02:00:00	-0.430572	-0.232883	1.261089	-0.042892
2000-01-01 03:00:00	-0.538359	-1.182248	-1.041456	-0.721104
2000-01-01 04:00:00	0.610743	1.854269	0.802882	1.192621

	sales	customers
2000-01-01 00:00:00	8	13
2000-01-01 01:00:00	14	10
2000-01-01 02:00:00	13	19
2000-01-01 03:00:00	3	5
2000-01-01 04:00:00	13	19
...	...	...

.dt functions

Let’s say the index is a datetime object

```
df = df.sample(500)
df["Year"] = df["index"].dt.year
df["Month"] = df["index"].dt.month
df["Day"] = df["index"].dt.day
df["Hour"] = df["index"].dt.hour
df["Minute"] = df["index"].dt.minute
df["Second"] = df["index"].dt.second
df["Nanosecond"] = df["index"].dt.nanosecond
df["Date"] = df["index"].dt.date
df["Time"] = df["index"].dt.time
df["Time_Time_Zone"] = df["index"].dt.timetz
df["Day_Of_Year"] = df["index"].dt.dayofyear
df["Week_Of_Year"] = df["index"].dt.weekofyear
df["Week"] = df["index"].dt.week
df["Day_Of_week"] = df["index"].dt.dayofweek
df["Week_Day"] = df["index"].dt.weekday
df["Week_Day_Name"] = df["index"].dt.weekday_name
df["Quarter"] = df["index"].dt.quarter
df["Days_In_Month"] = df["index"].dt.days_in_month
df["Is_Month_Start"] = df["index"].dt.is_month_start
df["Is_Month_End"] = df["index"].dt.is_month_end
df["Is_Quarter_Start"] = df["index"].dt.is_quarter_start
df["Is_Quarter_End"] = df["index"].dt.is_quarter_end
df["Is_Leap_Year"] = df["index"].dt.is_leap_year
```

Moving Average

```
def moving_average(series, n):
    """
        Calculate average of last n observations
    """
    return np.average(series[-n:])

moving_average(ads, 24) # prediction for the last observed day (past 24 hours)
```

Weighted Average

simple modification to the moving average. The weights sum up to 1 with larger weights assigned to more recent observations.

$$\hat{y}_t = \sum_{n=1}^{k} \omega_n y_{t+1-n}$$

```
def weighted_average(series, weights):
    """
        Calculate weighter average on series
    """
    result = 0.0
    weights.reverse()
    for n in range(len(weights)):
        result += series.iloc[-n-1] * weights[n]
    return float(result)
```

e.g.
weighted_average(ads, [0.6, 0.3, 0.1])

Exponential Smoothing

Here the model value is a weighted average between the current true value and the previous model values. The α weight is called a smoothing factor. It defines how quickly we will "forget" the last available true observation. The smaller α is, the more influence the previous observations have and the smoother the series is.

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

```
def exponential_smoothing(series, alpha):
    """
        series - dataset with timestamps
        alpha - float [0.0, 1.0], smoothing parameter
    """
    result = [series[0]] # first value is same as series
    for n in range(1, len(series)):
        result.append(alpha * series[n] + (1 - alpha) * result[n-1])
    return result
```

Double Exponential Smoothing

$$\ell_x = \alpha y_x + (1 - \alpha)(\ell_{x-1} + b_{x-1})$$

$$b_x = \beta(\ell_x - \ell_{x-1}) + (1 - \beta) b_{x-1}$$

$$\hat{y}_{x+1} = \ell_x + b_x$$

The first one describes the intercept, which, as before, depends on the current value of the series. The second term is now split into previous values of the level and of the trend. The second function describes the trend, which depends on the level changes at the current step and on the previous value of the trend. In this case, the β coefficient is a weight for exponential smoothing. The final prediction is the sum of the model values of the intercept and trend.

Now we have to tune two parameters: α and β. The former is responsible for the series smoothing around the trend, the latter for the smoothing of the trend itself. The larger the values, the more weight the most recent observations will have and the less smoothed the model series will be. Certain combinations of the parameters may produce strange results, especially if set manually. We'll look into choosing parameters automatically in a bit; before that, let's discuss triple exponential smoothing.

```
def double_exponential_smoothing(series, alpha, beta):
    """
        series - dataset with timeseries
        alpha - float [0.0, 1.0], smoothing parameter for level
        beta - float [0.0, 1.0], smoothing parameter for trend
    """
    # first value is same as series
    result = [series[0]]
    for n in range(1, len(series)+1):
        if n == 1:
            level, trend = series[0], series[1] - series[0]
        if n >= len(series): # forecasting
            value = result[-1]
        else:
            value = series[n]
        last_level, level = level, alpha*value + (1-alpha)*(level+trend)
        trend = beta*(level-last_level) + (1-beta)*trend
        result.append(level+trend)
    return result
```

Triple Exponential Smoothing (Holt Winters Method)

https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc435.htm

As you could have guessed, the idea is to add a third component - seasonality. This means that we should not use this method if our time series is not expected to have seasonality. Seasonal components in the model will explain repeated variations around intercept and trend, and it will be specified by the length of the season, in other words by the period after which the variations repeat. For each observation in the season, there is a separate component; for example, if the length of the season is 7 days (a weekly seasonality), we will have 7 seasonal components, one for each day of the week.

$$\ell_x = \alpha(y_x - s_{x-L}) + (1-\alpha)(\ell_{x-1} + b_{x-1})$$

$$b_x = \beta(\ell_x - \ell_{x-1}) + (1-\beta)b_{x-1}$$

$$s_x = \gamma(y_x - \ell_x) + (1-\gamma)s_{x-L}$$

$$\hat{y}_{x+m} = \ell_x + mb_x + s_{x-L+1+(m-1)modL}$$

$$\hat{y}_{max_x} = \ell_{x-1} + b_{x-1} + s_{x-T} + m \cdot d_{t-T}$$

$$\hat{y}_{min_x} = \ell_{x-1} + b_{x-1} + s_{x-T} - m \cdot d_{t-T}$$

$$d_t = \gamma \mid y_t - \hat{y}_t \mid + (1-\gamma)d_{t-T},$$

where T is the length of the season, d is the predicted deviation. Other parameters were taken from triple exponential smoothing.

```
class HoltWinters:

    """
    Holt-Winters model with the anomalies detection using Brutlag method

    # series - initial time series
    # slen - length of a season
    # alpha, beta, gamma - Holt-Winters model coefficients
    # n_preds - predictions horizon
    # scaling_factor - sets the width of the confidence interval by Brutlag (usually takes values from 2 to 3)

    """



    def __init__(self, series, slen, alpha, beta, gamma, n_preds, scaling_factor=1.96):
        self.series = series
        self.slen = slen
        self.alpha = alpha
        self.beta = beta
```

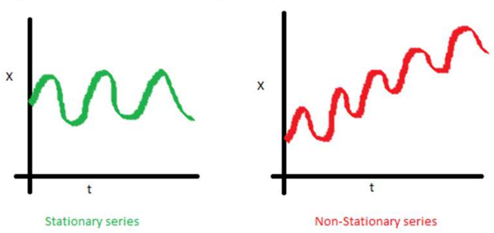
```
        self.gamma = gamma
        self.n_preds = n_preds
        self.scaling_factor = scaling_factor


    def initial_trend(self):
        sum = 0.0
        for i in range(self.slen):
            sum += float(self.series[i+self.slen] - self.series[i]) / self.slen
        return sum / self.slen

    def initial_seasonal_components(self):
        seasonals = {}
        season_averages = []
        n_seasons = int(len(self.series)/self.slen)
        # let's calculate season averages
        for j in range(n_seasons):
            season_averages.append(sum(self.series[self.slen*j:self.slen*j+self.slen])/float(self.slen))
        # let's calculate initial values
        for i in range(self.slen):
            sum_of_vals_over_avg = 0.0
            for j in range(n_seasons):
                sum_of_vals_over_avg += self.series[self.slen*j+i]-season_averages[j]
            seasonals[i] = sum_of_vals_over_avg/n_seasons
        return seasonals


    def triple_exponential_smoothing(self):
        self.result = []
        self.Smooth = []
        self.Season = []
        self.Trend = []
        self.PredictedDeviation = []
        self.UpperBond = []
        self.LowerBond = []

        seasonals = self.initial_seasonal_components()

        for i in range(len(self.series)+self.n_preds):
            if i == 0: # components initialization
                smooth = self.series[0]
                trend = self.initial_trend()
                self.result.append(self.series[0])
                self.Smooth.append(smooth)
                self.Trend.append(trend)
                self.Season.append(seasonals[i%self.slen])

                self.PredictedDeviation.append(0)

                self.UpperBond.append(self.result[0] +
                                      self.scaling_factor *
                                      self.PredictedDeviation[0])

                self.LowerBond.append(self.result[0] -
                                      self.scaling_factor *
                                      self.PredictedDeviation[0])
                continue
```

```
            if i >= len(self.series): # predicting
                m = i - len(self.series) + 1
                self.result.append((smooth + m*trend) + seasonals[i%self.slen])

                # when predicting we increase uncertainty on each step
                self.PredictedDeviation.append(self.PredictedDeviation[-1]*1.01)

            else:
                val = self.series[i]
                last_smooth, smooth = smooth, self.alpha*(val-seasonals[i%self.slen]) + (1-
self.alpha)*(smooth+trend)
                trend = self.beta * (smooth-last_smooth) + (1-self.beta)*trend
                seasonals[i%self.slen] = self.gamma*(val-smooth) + (1-
self.gamma)*seasonals[i%self.slen]
                self.result.append(smooth+trend+seasonals[i%self.slen])

                # Deviation is calculated according to Brutlag algorithm.
                self.PredictedDeviation.append(self.gamma * np.abs(self.series[i] - self.result[i])
                                               + (1-self.gamma)*self.PredictedDeviation[-1])

            self.UpperBond.append(self.result[-1] +
                                  self.scaling_factor *
                                  self.PredictedDeviation[-1])

            self.LowerBond.append(self.result[-1] -
                                  self.scaling_factor *
                                  self.PredictedDeviation[-1])

            self.Smooth.append(smooth)
            self.Trend.append(trend)
            self.Season.append(seasonals[i%self.slen])
```

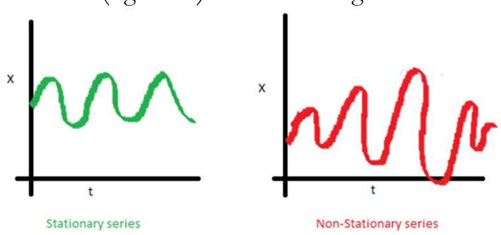
Stationarity

Means it does not change its statistical properties over time, namely its mean and variance. (The constancy of variance is called homoscedasticity)The covariance function does not depend on time; it should only depend on the distance between observations.

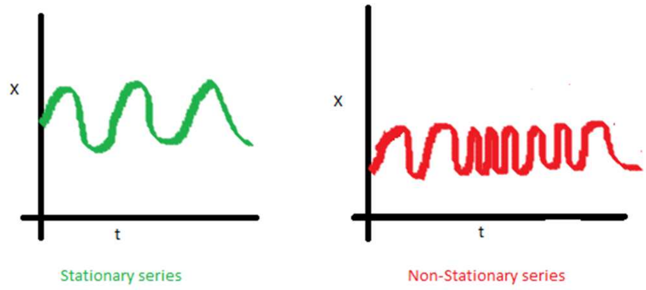
(right red) mean changes over time



(right red) variance changes over time



(right red) Covariance changes over time



Random Walk / Stationarity Hypothesis Testing (especially for stock price) – Dickey Fuller Test

https://en.wikipedia.org/wiki/Dickey%E2%80%93Fuller_test

the Dickey-Fuller test rejected the null hypothesis that a unit root is present ➔ series is STATIONARY

Regression test for random walk

$P_t = \alpha + \beta P_{t-1} + \varepsilon_t$

Equivalent to $P_t - P_{t-1} = \alpha + \beta P_{t-1} + \varepsilon_t$

Test:

H_0: $\beta = 1$ (This is a random walk)

H_1: $\beta < 1$ (This is not a random walk)

Dickey-Fuller Test:

H_0: $\beta = 0$ (This is a random walk)

H_1: $\beta < 0$ (This is not a random walk)

Augmented Dickey-Fuller test

An augmented Dickey–Fuller test (ADF) tests the null hypothesis that a unit root is present in a time series sample. It is basically Dickey-Fuller test with more lagged changes on RHS.

```
# Augmented Dickey-Fuller test on volume of google and microsoft stocks
from statsmodels.tsa.stattools import adfuller
adf = adfuller(microsoft["Volume"])
print("p-value of microsoft: {}".format(float(adf[1])))
adf = adfuller(google["Volume"])
print("p-value of google: {}".format(float(adf[1])))
```

As microsoft has p-value 0.0003201525 which is less than 0.05, null hypothesis is rejected and this is not a random walk.

Now google has p-value 0.0000006510 which is more than 0.05, null hypothesis is rejected and this is not a random walk.

```
# Generate Random Walk
from numpy.random import normal, seed
seed(42)
rcParams['figure.figsize'] = 16, 6
random_walk = normal(loc=0, scale=0.01, size=1000)
plt.plot(random_walk)
```

Dealing with Seasonality

Take "seasonal difference", which means a simple subtraction of the series from itself with a lag that equals the seasonal period.

```
ads_diff = ads.Ads - ads.Ads.shift(24)
```

After this operation, if the autocorrelation is still too high…. Then… take first diff!

```
ads_diff = ads_diff - ads_diff.shift(1)
```