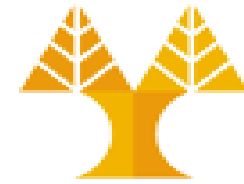


DSC510: Introduction to Data Science and Analytics

Lab 11: Introduction to Timeseries



University of Cyprus
Department of
Computer Science

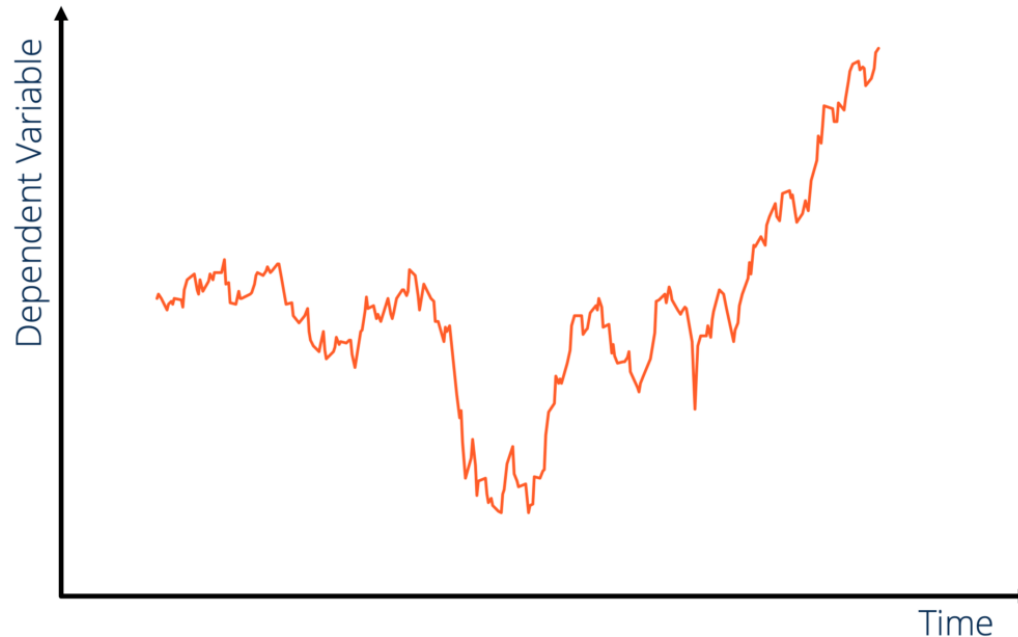
Pavlos Antoniou
Office: B109, FST01

** More detailed information on timeseries can be found [here](#) (L14, L15) **

Introduction to time series



- A collection of observations for a time-dependent variable at various time intervals is known as time series



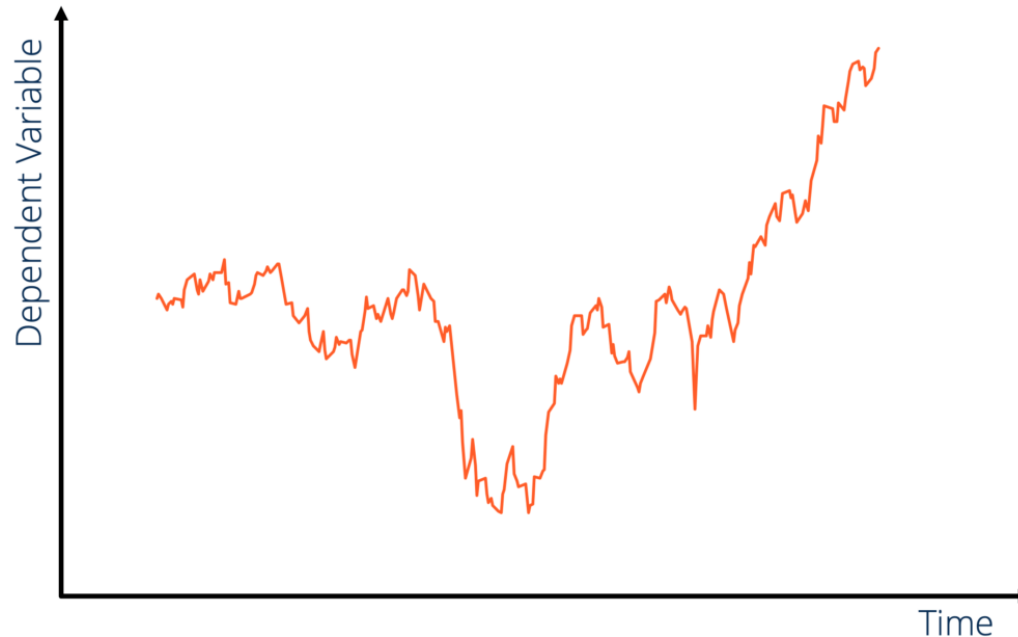
| | | |
|------------|----------|----|
| 2019-01-01 | 00:00:00 | 49 |
| 2019-01-01 | 00:01:00 | 58 |
| 2019-01-01 | 00:02:00 | 48 |
| 2019-01-01 | 00:03:00 | 96 |
| 2019-01-01 | 00:04:00 | 42 |
| 2019-01-01 | 00:05:00 | 8 |
| 2019-01-01 | 00:06:00 | 20 |
| 2019-01-01 | 00:07:00 | 96 |
| 2019-01-01 | 00:08:00 | 48 |
| 2019-01-01 | 00:09:00 | 78 |

- Time series data is omnipresent in our lives as it can be generated in pretty much any domain
 - Sensors monitoring weather conditions, PV inverters measuring generated solar energy, stock price variations in the stock market

Introduction to time series



- A collection of observations for a time-dependent variable at various time intervals is known as time series



| | | |
|------------|----------|----|
| 2019-01-01 | 00:00:00 | 49 |
| 2019-01-01 | 00:01:00 | 58 |
| 2019-01-01 | 00:02:00 | 48 |
| 2019-01-01 | 00:03:00 | 96 |
| 2019-01-01 | 00:04:00 | 42 |
| 2019-01-01 | 00:05:00 | 8 |
| 2019-01-01 | 00:06:00 | 20 |
| 2019-01-01 | 00:07:00 | 96 |
| 2019-01-01 | 00:08:00 | 48 |
| 2019-01-01 | 00:09:00 | 78 |

- Pandas DataFrames organize time series data by using **date/time based indexing** (row labels)

Introduction to time series

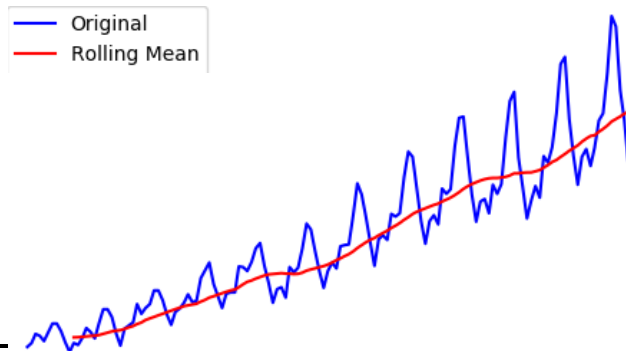


- Many time series are **uniformly** (equally) **spaced** at a specific frequency
 - For example: hourly weather measurements, daily counts of web site visits, or monthly sales totals
 - Time series can also be **irregularly** (unequally) **spaced** and sporadic (bursts)
 - For example: timestamped data in a computer system's event log or a history of ambulance emergency calls
 - Pandas time series tools apply well to either type of time series
-

Resampling, Shifting, and Windowing



- Important time series operations available by Pandas DataFrames:
 - Resampling: converting the time series from one frequency to another
 - e.g. convert hourly data (one sample per hour) to daily data (one sample per day) or vice versa
 - Time-shifting: going forward and backward over time
 - Can be performed by shifting the datetime-based index
 - Moving window functions (rolling statistics)
 - e.g. calculate moving (rolling) mean values taking into account the last ten samples
 - used to smooth data and capture the trend (uptrend, downtrend)



Dataset for practice



- Load [Google stock price dataset](#) as DataFrame:

```
import pandas as pd
goog = pd.read_csv("google_stocks_2005_2021.csv", index_col="Date",
                   parse_dates=True)
print(type(goog.index)) # <class 'pandas.core.indexes.datetimes.DatetimeIndex'>
goog.head()
```

If True, Pandas tries to parse the index to examine if it can be represented as an array of datetimes. If not set, dates will be loaded as strings and timeseries operations cannot be applied.

Many TS
operations require
date-oriented index

| | Date | High | Low | Open | Close | Volume | Adj Close |
|--|------------|------------|-----------|------------|------------|------------|------------|
| | 2005-01-03 | 101.439781 | 97.365051 | 98.331429 | 100.976517 | 31807176.0 | 100.976517 |
| | 2005-01-04 | 101.086105 | 96.378746 | 100.323959 | 96.886841 | 27614921.0 | 96.886841 |
| | 2005-01-05 | 98.082359 | 95.756081 | 96.363808 | 96.393692 | 16534946.0 | 96.393692 |
| | 2005-01-06 | 97.584229 | 93.509506 | 97.175758 | 93.922951 | 20852067.0 | 93.922951 |
| | 2005-01-07 | 96.762314 | 94.037521 | 94.964050 | 96.563057 | 19398238.0 | 96.563057 |

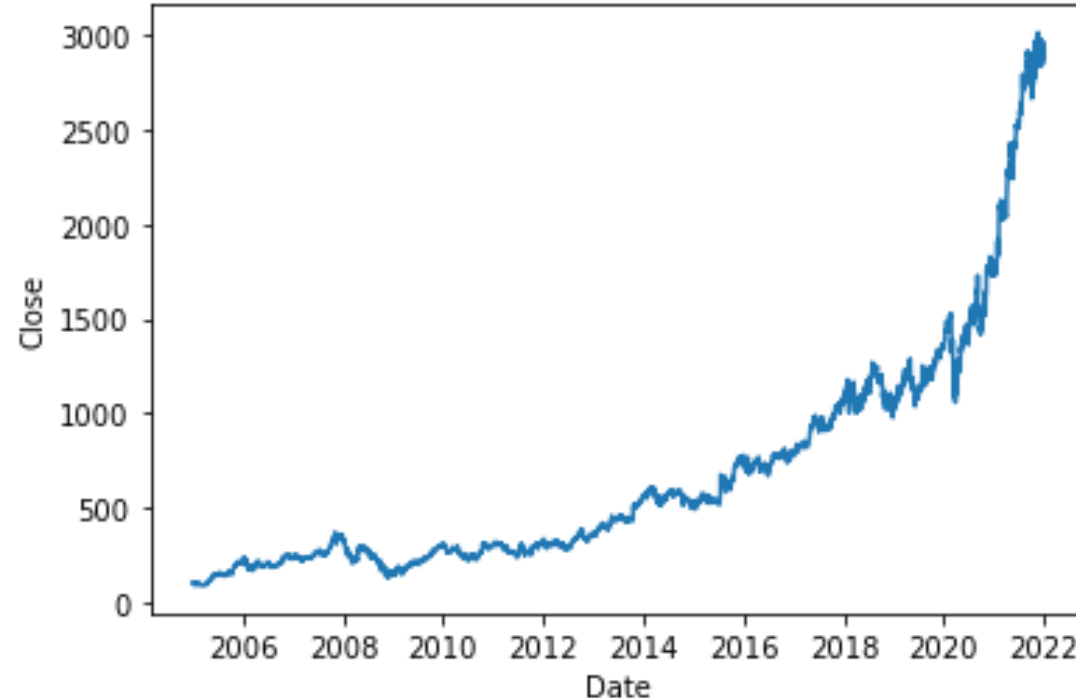
Only business
days are included
(no weekends or
public holidays) in
dataset

Plotting time series



- For simplicity, we'll use just the closing price
- We can visualize this using the Seaborn `lineplot()` function

```
import matplotlib.pyplot as plt
import seaborn as sns
fig, ax = plt.subplots()
sns.lineplot(data=goog, x=goog.index, y='Close')
```



Plotting time series (alternative)

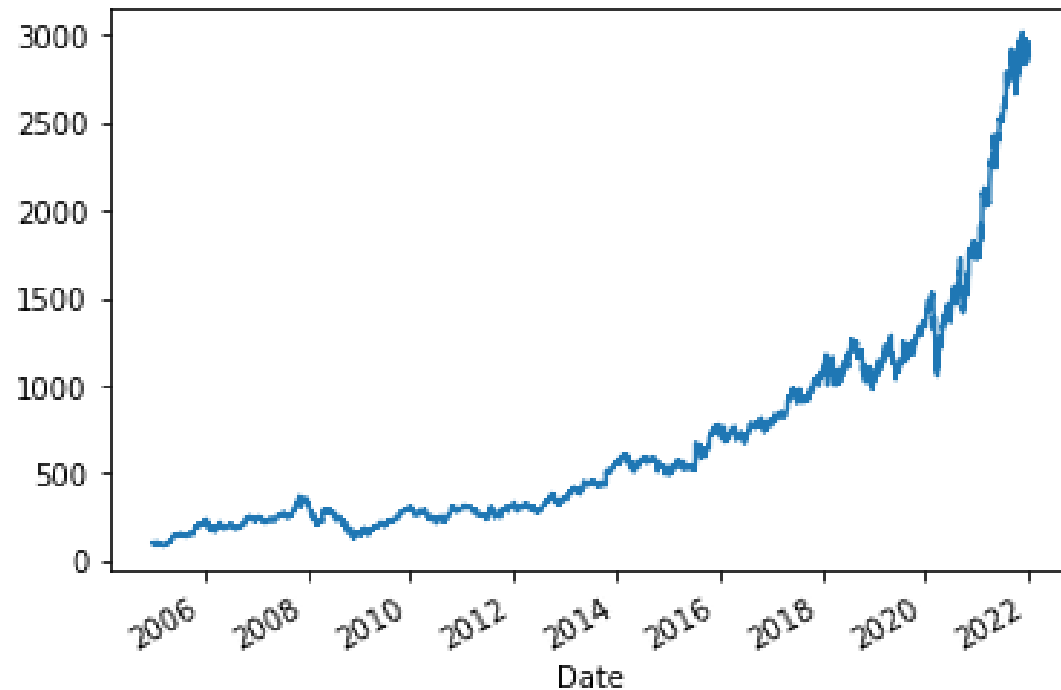


- Alternatively, we can extract the Close column as a Series object

```
goog = goog['Close']
```

- Plot the series using the `.plot()` function on the Series object

```
import matplotlib.pyplot as plt  
fig, ax = plt.subplots()  
goog.plot()
```



Resampling data



- Resampling is the process of changing the frequency of time-dependent (timeseries) features
 - Two types of resampling are:
 - **Down-sampling:** When you reduce the frequency of data (lower granularity) by aggregating over a longer time period
 - for example, converting hourly data to daily data by averaging or summing the values
 - **Up-sampling:** When you increase the frequency of data (higher granularity) by interpolating or filling in additional data points
 - for instance, converting daily data to hourly data by estimating values between days
-

Resampling data



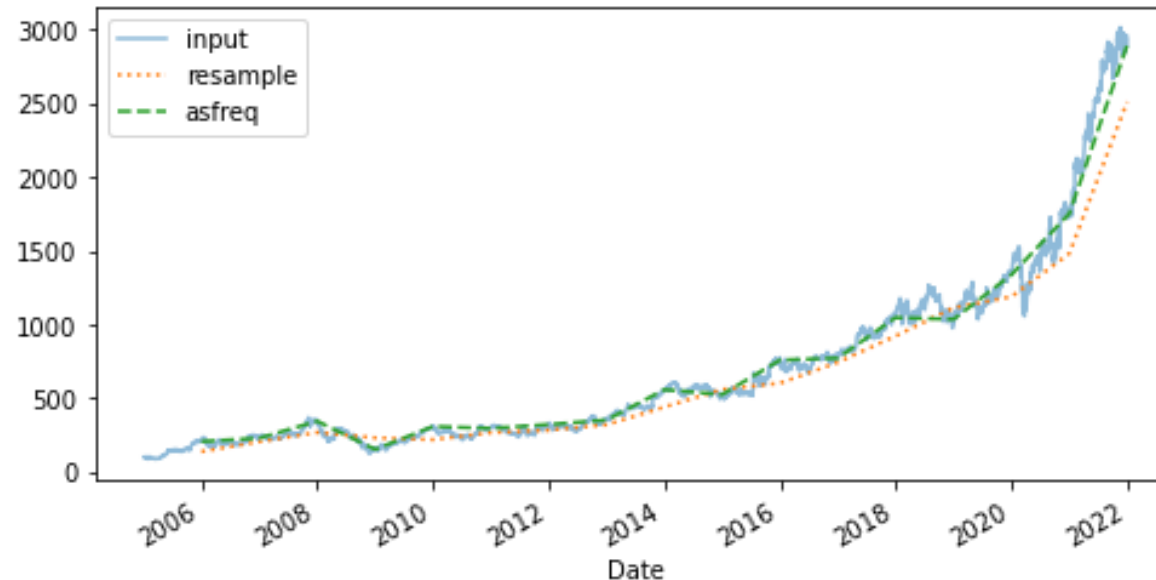
- We can resample a time series at a higher or lower frequency:
 - to inspect how dataset behaves differently under different resolutions or frequency
 - For example, you have a feature measured on a daily basis and you want to make predictions on a monthly basis => you need to downsample it to a monthly level
 - to join datasets with initially different resolutions and obtain a larger dataset
 - there is an extremely high number of observations that needs to be reduced to speedup both EDA and ML algorithms execution time
 - Need for down-sampling
- Resample functions: `resample()`, `asfreq()`
 - `resample()` is fundamentally a **data aggregation**, while `asfreq()` is fundamentally a **data selection**
 - The dataset to be resampled must have a datetime-like index

Down-sampling



- Down-sample (lower freq) Google closing price using both functions
- Example: we resample the data at the end of business year:

```
fig, ax = plt.subplots(figsize=(8,4))
goog.plot(alpha=0.5, style='-')
goog.resample('BYE').mean().plot(style=':')
goog.asfreq('BYE').plot(style='--');
plt.legend(['input', 'resample', 'asfreq'], loc='upper left')
```



Notice the difference: at each point, `resample()` reports the average of the previous year, while `asfreq()` reports (selects) the value at the end of the year.

`resample()` is like a `groupby` function and should be used in conjunction with other aggregation functions such as `sum`, `median`, etc.

Frequencies and offsets



- We can use **frequency codes** to specify any desired **frequency spacing**. The following table summarizes the main codes available:

| Code | Description | Code | Description |
|------|--------------|------|----------------------|
| D | Calendar day | B | Business day |
| W | Weekly | | |
| ME | Month end | BME | Business month end |
| QE | Quarter end | BQE | Business quarter end |
| YE | Year end | BYE | Business year end |
| h | Hours | bh | Business hours |
| min | Minutes | | |
| s | Seconds | | |
| ms | Milliseconds | | |
| us | Microseconds | | |
| ns | nanoseconds | | |

The monthly, quarterly, and annual frequencies are all marked at the end of the specified period. By adding an S suffix to any of these, they instead will be marked at the beginning:

| | | | |
|----|---------------|-----|------------------------|
| MS | Month start | BMS | Business month start |
| QS | Quarter start | BQS | Business quarter start |
| YS | Year start | BYS | Business year start |

Up-sampling



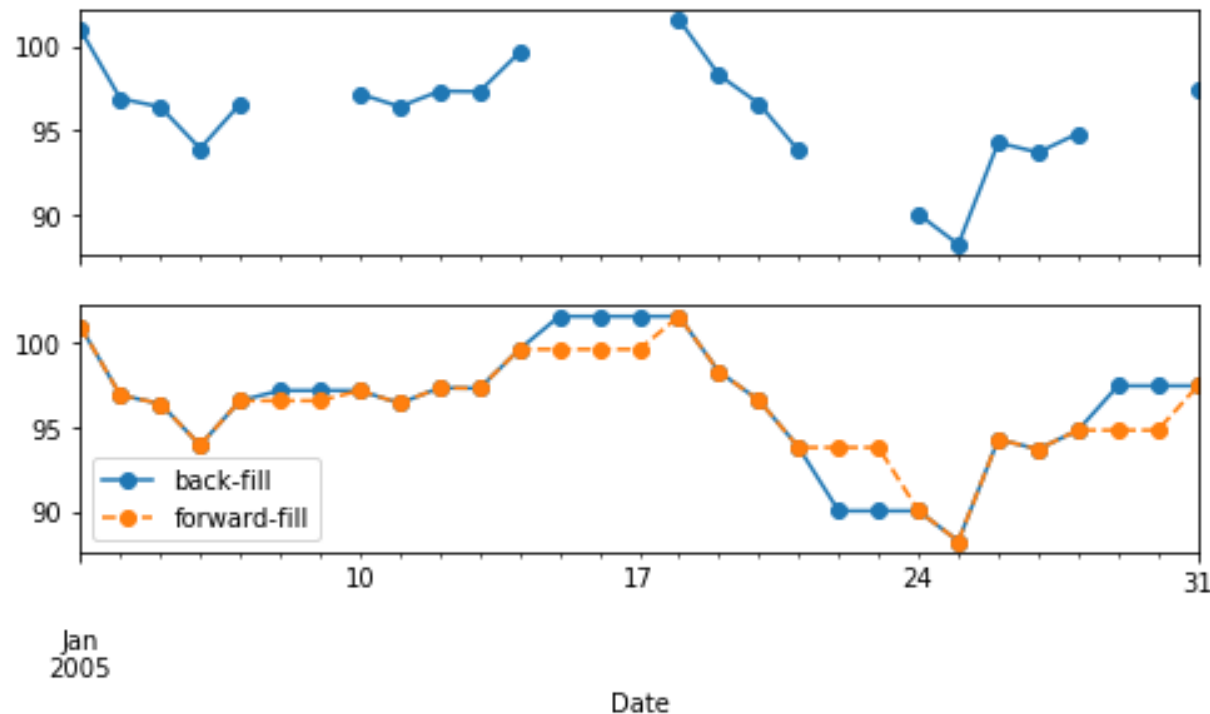
- For up-sampling (high freq) , `resample()` and `asfreq()` are largely equivalent
- The default for both methods is to leave the up-sampled (new imputed) points empty, that is, filled with NA values
- `asfreq()` accepts a method argument to specify how values are imputed:
 - `pad / ffill` (forward fill): propagate last valid observation forward to next valid
 - `bfill` (back fill): use NEXT valid observation to fill
- Example: in Google dataset, we will resample at a daily frequency so as days related to weekends and other public holidays (that are missing) to be created

Up-sampling (asfreq)



```
fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8,4))
data = goog.iloc[:20]
# Up-sampling without filling (default action)
data.asfreq('D').plot(ax=axs[0], marker='o')

data.asfreq('D', method='bfill').plot(ax=axs[1], style='-o')
data.asfreq('D', method='ffill').plot(ax=axs[1], style='--o')
axs[1].legend(["back-fill", "forward-fill"])
```



Up-sampling without filling (default action) => weekends and public holidays are present in the dataset but have NA (no value) as close price

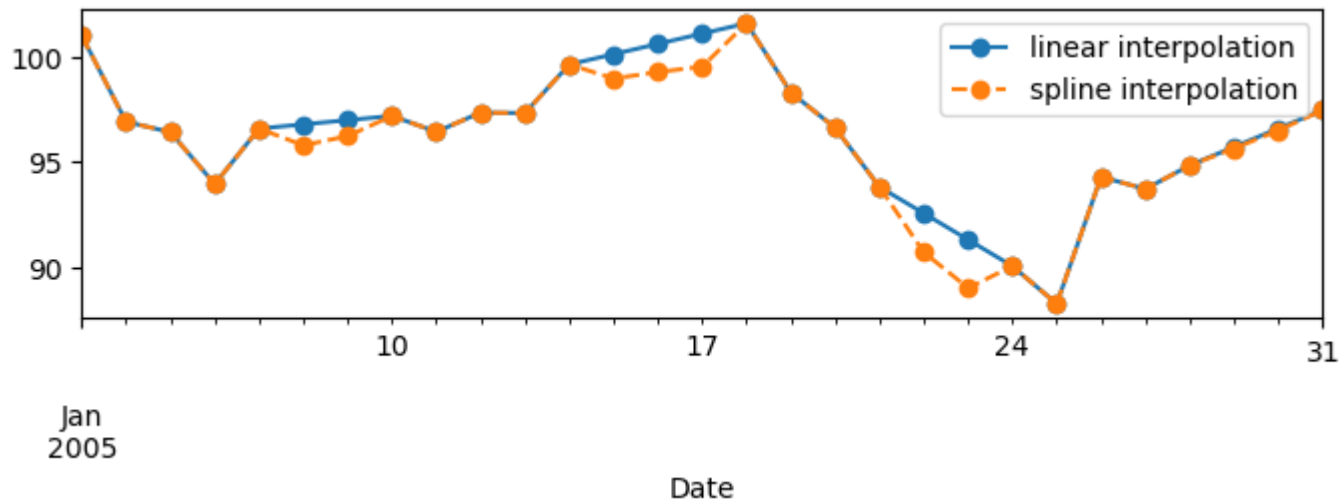
** Note: SimpleImputer from Lab4 could be used to fill gaps with statistical-based parameters such as mean, median.

Up-sampling (resample)



```
fig, axs = plt.subplots(2, 1, sharex=True, figsize=(8,4))  
data = goog.iloc[:20]
```

```
data.resample('D').interpolate(method='linear').plot(style='-o')  
data.resample('D').interpolate(method='spline',order=2).plot(style='--o')  
axs[1].legend(["linear interpolation", "spline interpolation"])
```



Up-sampling with linear and spline interpolations

Time shifts



- Common time series-specific operation: shifting of data in time
 - Time shift function: `shift()`
 - Shifts **index** by the desired number of `periods` with an optional time `freq`
 - When `freq` is **not passed**, data values will be shifted **without re-aligning** the index
 - When `freq` is passed, both data values and index will be shifted using the `periods` and the `freq`
 - Example: use function with / without `freq` to shift dataset by 800 days
-

Time shifts



```
fig, axs = plt.subplots(3, 1, figsize=(10, 10), sharey=True)

# fill-in weekends
goog_filled = goog.asfreq('D', method='ffill')

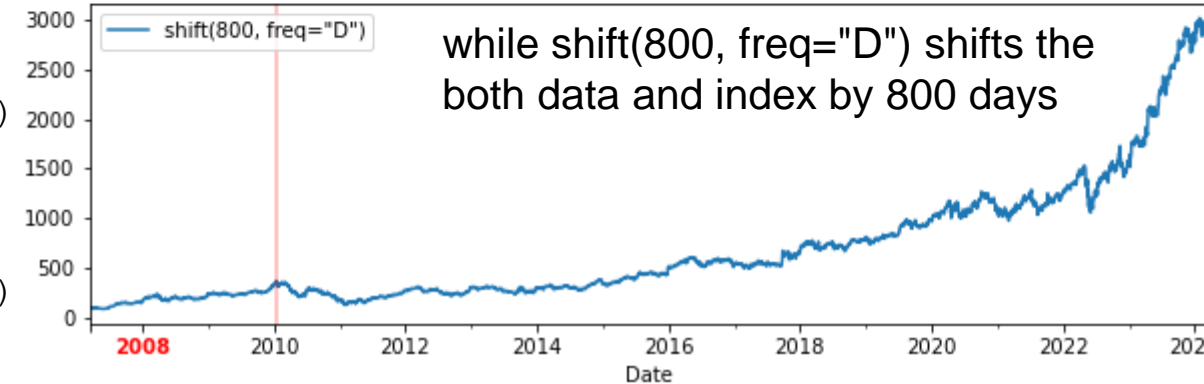
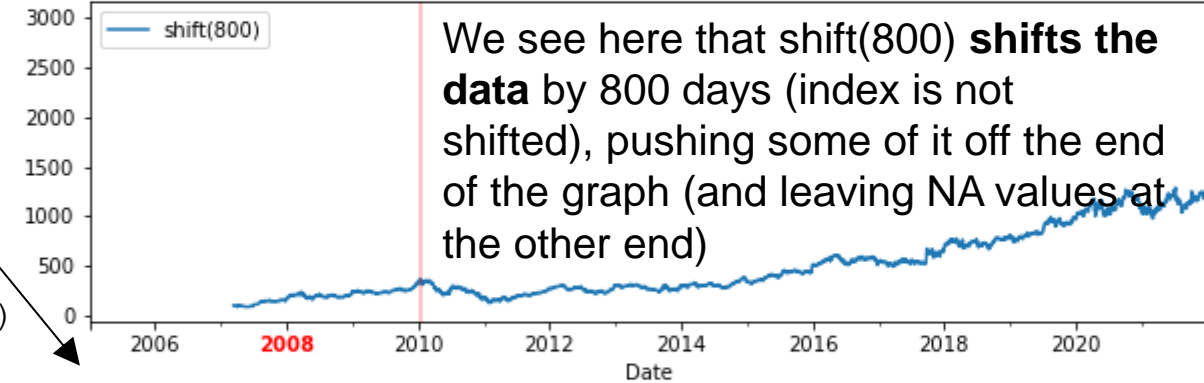
goog_filled.plot(ax=axs[0])
goog_filled.shift(periods=800).plot(ax=axs[1])
goog_filled.shift(periods=800, freq='D').plot(ax=axs[2])

# legends and annotations
local_max = pd.to_datetime('2007-11-05')
offset = pd.Timedelta(800, 'D')

axs[0].legend(['input'], loc='upper left')
axs[0].get_xticklabels()[2].set(weight='heavy', color='red')
axs[0].axvline(local_max, alpha=0.3, color='red')

axs[1].legend(['shift(800)'], loc='upper left')
axs[1].get_xticklabels()[2].set(weight='heavy', color='red')
axs[1].axvline(local_max + offset, alpha=0.3, color='red')

axs[2].legend(['shift(800, freq="D")'], loc='upper left')
axs[2].get_xticklabels()[1].set(weight='heavy', color='red')
axs[2].axvline(local_max + offset, alpha=0.3, color='red')
```



Used in many tasks e.g. for detrending or deseasonalizing ([here](#)) or for creating lagged features ([here](#))

Moving window functions (rolling statistics)



- Rolling statistics are a third type of time series-specific operation implemented by Pandas
 - Rolling statistics function: `rolling()`
 - can be applied on both Series and DataFrame objects
 - returns a view similar to groupby operation
 - rolling view makes available a number of aggregation operations (sum, mean, median, ...) by default
 - important parameters:
 - window: this is the number of observations used for calculating the statistic
 - center: If True the rolling statistics are assigned to the center of the window. If False (default) the rolling statistics are assigned to the right edge of the window.
-

Moving window functions (rolling statistics)



window=3

```
df['Rolling_Volume_Sum'] = df['Volume'].rolling(3, center = ).sum()
```

False

True

| | Volume | Rolling_Volume_Sum |
|---|--------|--------------------|
| 0 | 3641 | NaN |
| 1 | 3675 | NaN |
| 2 | 3371 | 10687.0 |
| 3 | 2519 | 9565.0 |
| 4 | 3260 | 9150.0 |
| 5 | 2922 | 8701.0 |

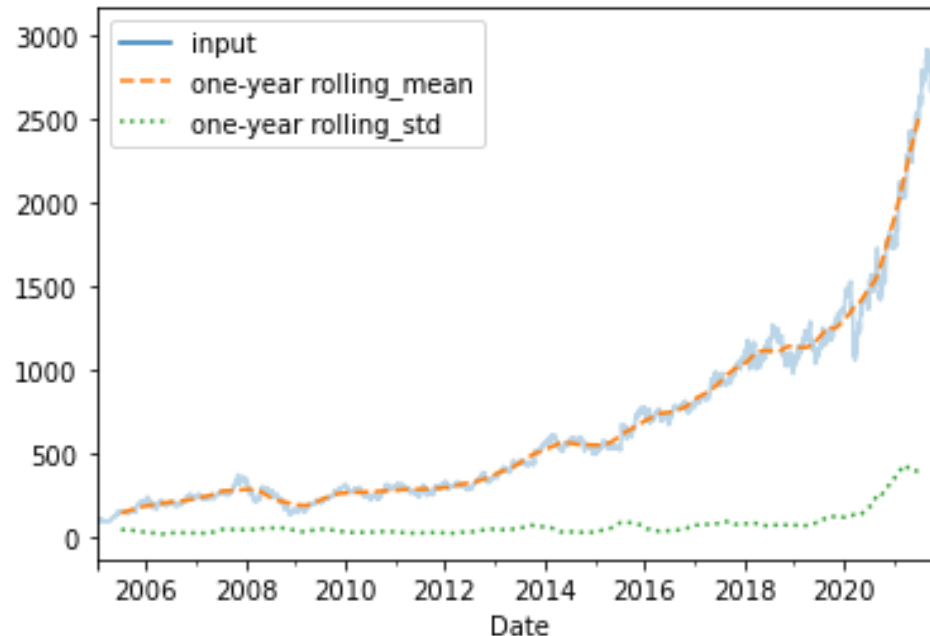
| | Volume | Rolling_Volume_Sum |
|---|--------|--------------------|
| 0 | 3641 | NaN |
| 1 | 3675 | 10687.0 |
| 2 | 3371 | 9565.0 |
| 3 | 2519 | 9150.0 |
| 4 | 3260 | 8701.0 |
| 5 | 2922 | NaN |

Rolling statistics



- One-year centered rolling mean and standard deviation of the Google closing stock prices

```
rolling = goog.rolling(365, center=True)
data = pd.DataFrame({'input': goog,
                     'one-year rolling_mean': rolling.mean(),
                     'one-year rolling_std': rolling.std()})
ax = data.plot(style=['-', '--', ':'])
ax.lines[0].set_alpha(0.3)
```



Timeseries Analysis & Forecasting



- TS **analysis**: use mathematical (statistical) methods that best capture or describe an observed time series in order to find patterns and trends in an effort to understand the underlying causes – find “why” behind a dataset
 - TS **forecasting**: fit models on historical data and using them to predict future observations
-

Timeseries Forecasting Concerns



- Time series data often requires pre-processing (e.g. cleaning) and even transformation (e.g. frequency scaling)
- Cleaning
 - Missing data*: Perhaps there are gaps or missing data that need to be interpolated or imputed
 - Outliers: Perhaps there are extreme outlier values that need to be identified and handled
- Scaling*
 - Frequency: Perhaps data is provided at a frequency that is too high to model or is unevenly spaced through time requiring resampling for use in some models

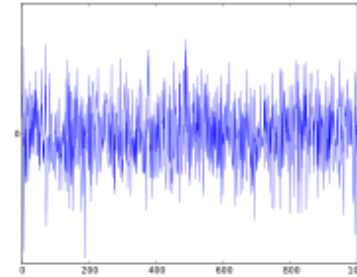
(*) Use `asfreq()` or `resample()`

Stationarity in timeseries



- A TS is said to be **stationary** if its **statistical properties** such as mean, variance remain **constant over time**

- The Poisson process and white noise are perfect stationary signals



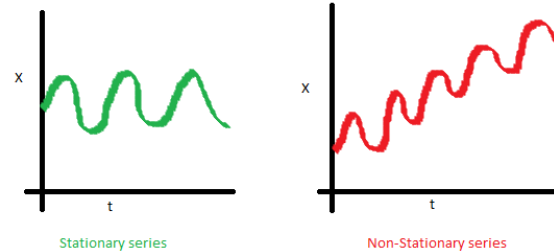
- In other words, if TS is stationary and has a particular behavior over time, the TS will follow the same behavior in the future
- Most TS models work on the assumption that the TS is stationary → look at examples at the end of that lab

Stationarity in timeseries

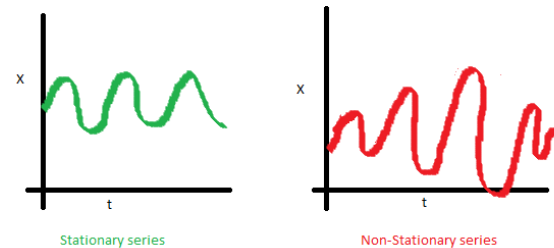


- Stationarity is defined using very strict criterion
- In theory, a TS is assumed to be stationary if it has constant statistical properties over time:

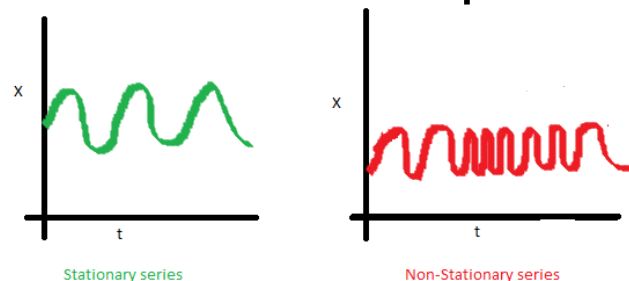
- constant mean



- constant variance



- an autocovariance that does not depend on time.



There is a statistical dependence of samples taken at different time points

Time series example



- Dataset: Airline Passengers
- Describes the total number of airline passengers (in thousands)
- 144 monthly observations from 1949 to 1960

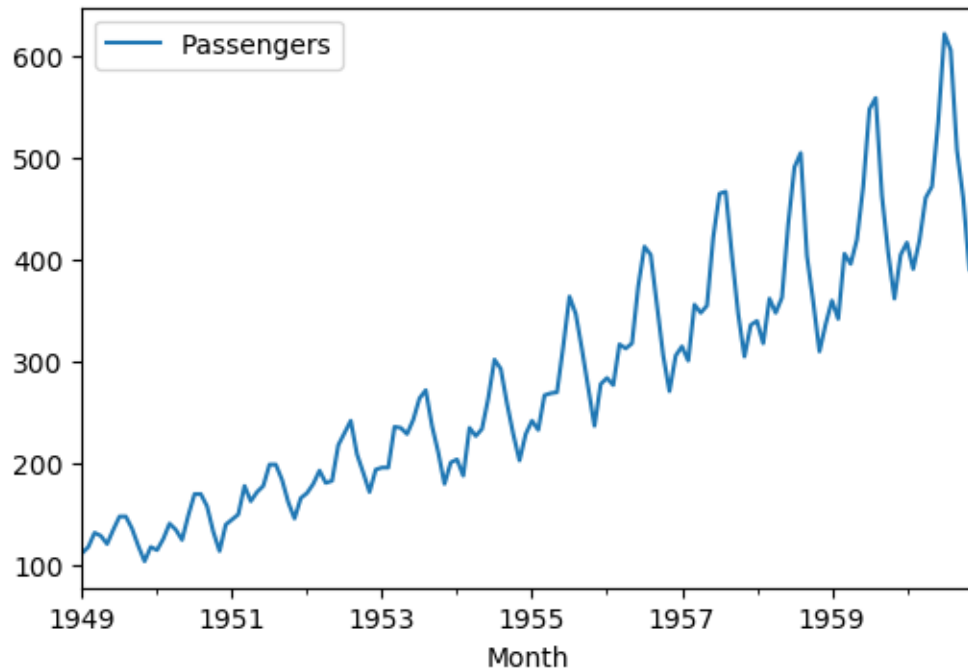
"Month", "Passengers"

"1949-01", 112

"1949-02", 118

"1949-03", 132

"1949-04", 129



```
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('airline-passengers.csv',
index_col='Month', parse_dates=True,
infer_datetime_format=True)
# plot original timeseries
df.plot()
plt.show()
```

Observations:

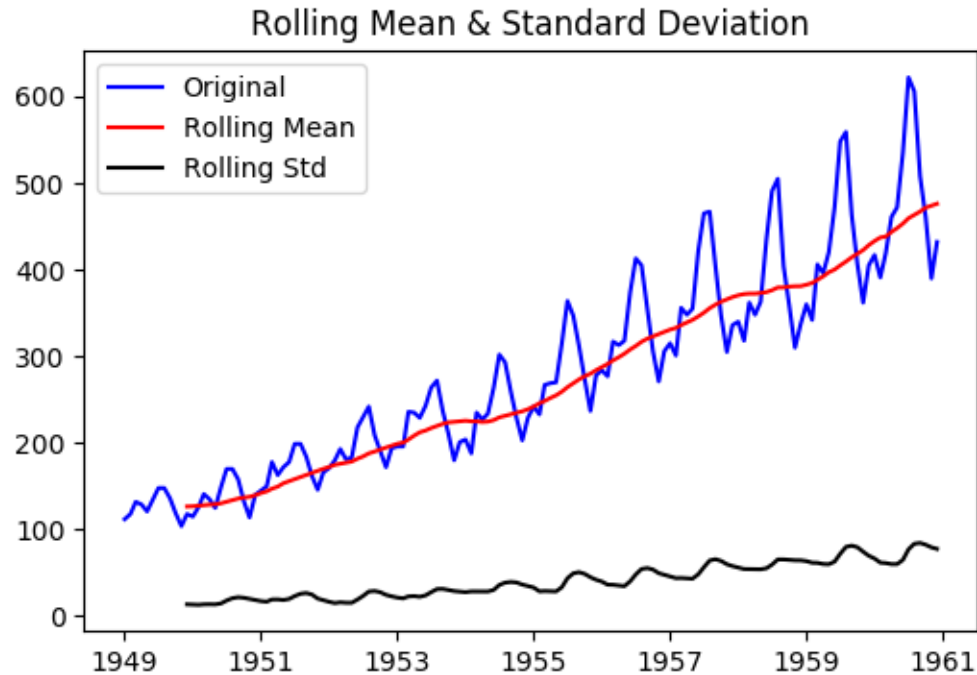
- overall increasing trend
- some seasonal variations

Check stationarity



- **Plot Rolling Statistics – a visual technique**
 - Plot the moving average & moving variance and see if they vary with time
 - Moving average/variance: At any time t , take the average/variance of the last year, i.e. last 12 months
 - **Dickey-Fuller Test – a statistical test**
 - Null hypothesis is that the TS is non-stationary
 - Use 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**
 - See [here](#) for details
-

Check stationarity



```
ts = df['Passengers']  
test_stationarity(ts)
```

Apply rolling function on a window of size 12
Observations

- Variation in standard deviation is small
- Mean is clearly increasing with time (uptrend behavior) → not a stationary timeseries
- Test statistic is way more than the critical values → null hypothesis is not rejected → time series is not stationary

Results of **Dickey-Fuller Test**:

Test Statistic

p-value

#Lags Used

Number of Observations Used

Critical Value (1%)

Critical Value (5%)

Critical Value (10%)

0.815369

0.991880

13.000000

130.000000

-3.481682

-2.884042

-2.578770

How to make a TS Stationary?

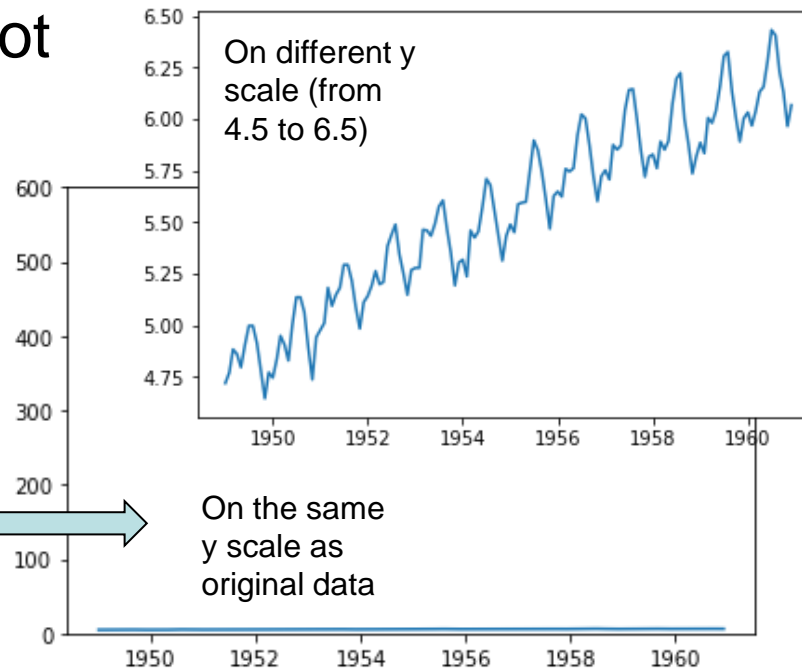
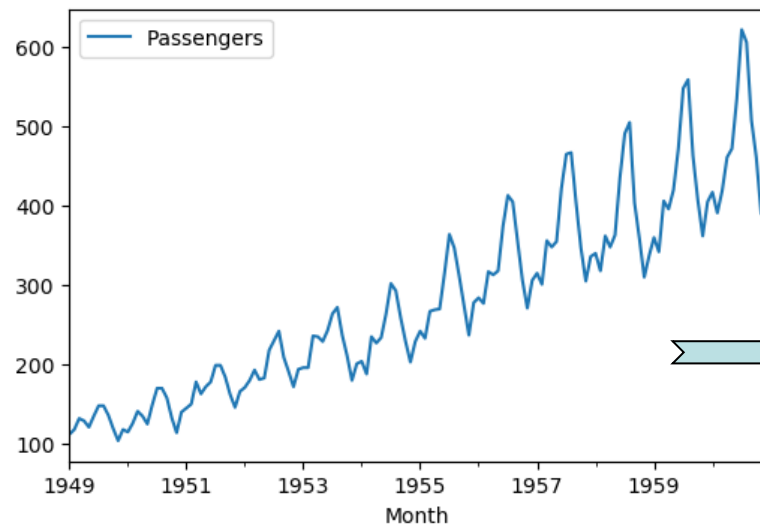


- Impossible to make a series perfectly stationary
- 2 major reasons behind non-stationarity of a TS:
 - **Trend** – long term increase or decrease in data → observe mean over time
 - e.g., in airline passengers we observed an **increasing trend**: on average, the number of passengers is growing over time
 - **Seasonality** – systematic (repeated), calendar related movements → observe variations at specific time-frames
 - e.g., in airline passengers we observed people have a tendency to travel more in a particular time period e.g. during summer time
- Next steps:
 - model or estimate trend and seasonality components and remove them to convert a non-stationary TS to a stationary TS
 - implement forecasting techniques on stationary TS and predict TS values
 - convert the predicted values into the original scale by re-applying trend and seasonality components back

Estimating and Eliminating Trend



- Reduce trend by **transformation**
- Suppress increasing trend => apply transformation that penalizes higher values more than small values
 - log, square root, cube root



```
ts_log = np.log(ts)
plt.plot(ts_log)
```

```
"Month", "Passengers"
"1949-01", 4.718499
"1949-02", 4.770685
"1949-03", 4.882802
"1949-04", 4.859812
```

Estimating and Eliminating Trend



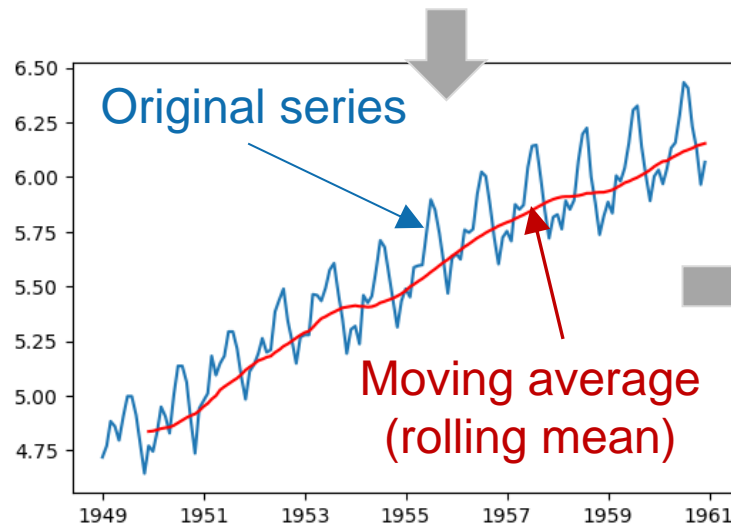
- Common techniques to estimate or model trend :
 - **Aggregation** – taking average for a time period like monthly/weekly averages
 - **Smoothing** – taking rolling (moving) averages
 - **Polynomial Fitting** – fit a polynomial regression model
 - Fit a polynomial curve (e.g., quadratic or cubic) to the time series to capture long-term trends: $Y_t = \beta_0 + \beta_1 t + \beta_2 t^2 + \dots + \beta_p t^p + \epsilon_t$
 - After estimating trend, we need to remove it from the series
-

Estimating and Eliminating Trend

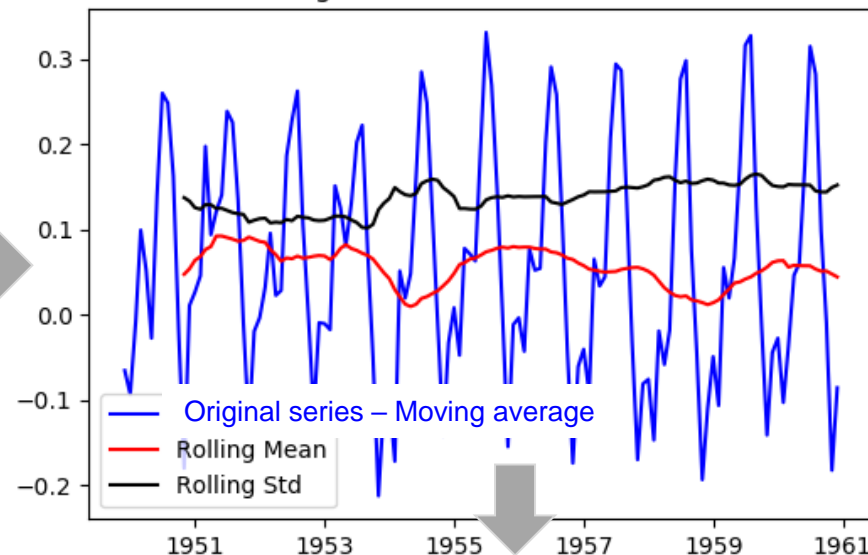


- **Smoothing**

- Try a **moving (rolling) average** of the past 'k' consecutive values depending on the frequency of time series (e.g. over the past 1 year, i.e. last 12 values) and subtract it from the original series & re-run stationarity test



```
moving_avg = tslog.rolling(12).mean()
plt.plot(ts_log)
plt.plot(moving_avg, color='red')
```



Results of Dickey-Fuller Test:

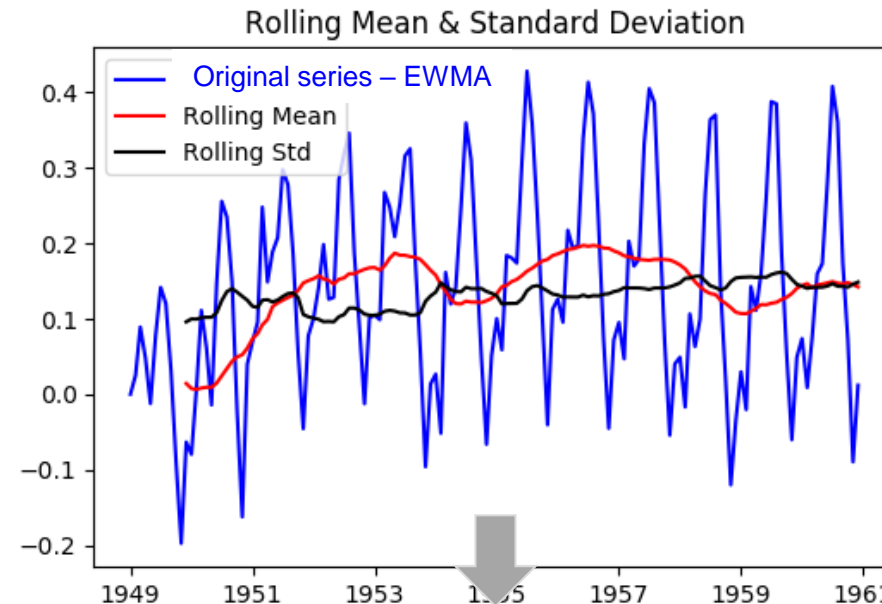
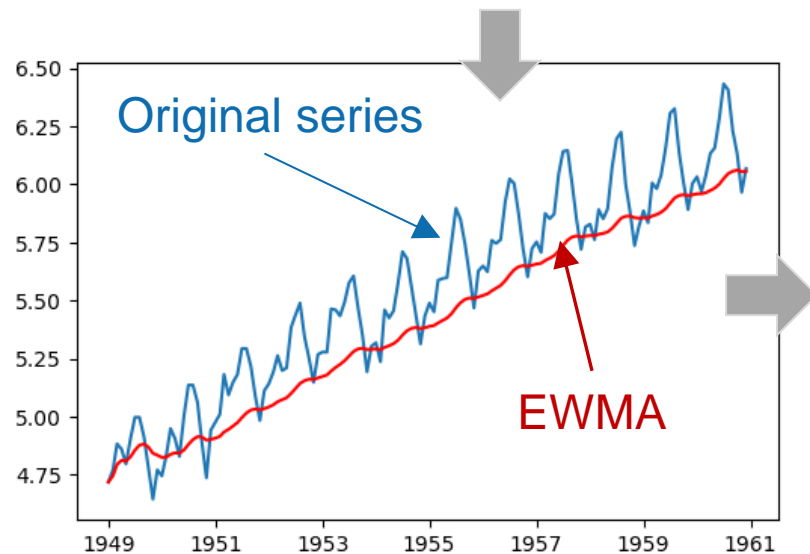
| | |
|----------------------|-----------|
| Test Statistic | -3.162908 |
| p-value | 0.022235 |
| Critical Value (5%) | -2.886151 |
| Critical Value (1%) | -3.486535 |
| Critical Value (10%) | -2.579896 |

Test statistic is **smaller than the 5% critical values** so we can say with 95% confidence that this is a stationary series.

Estimating and Eliminating Trend



- Try a ‘weighted moving average’ where more recent values within the windows are given a higher weight
 - Use **exponentially weighted moving average** (**EWMA**) where weights are assigned to all the previous values with a decay factor



Test statistic is **smaller than the 1% critical value => TS is stationary with 99% confidence => better than previous case**

Results of Dickey-Fuller Test:

| | |
|----------------------|-----------|
| Test Statistic | -3.601262 |
| Critical Value (5%) | -2.884042 |
| Critical Value (1%) | -3.481682 |
| Critical Value (10%) | -2.578770 |

Eliminating Trend and Seasonality



- Simple trend reduction techniques do not work in cases with high seasonality
 - Common methods dealing with both trend and seasonality:
 - **Differencing** – creates a new TS by taking the differences between observations
 - **Decomposition** – models both trend and seasonality components and removes them from the model
-

Eliminating Trend and Seasonality



- **First-order differencing:** takes the difference of the observation at a particular instant with that at the prev instant

"Month", "Passengers"

"1949-01", Nan

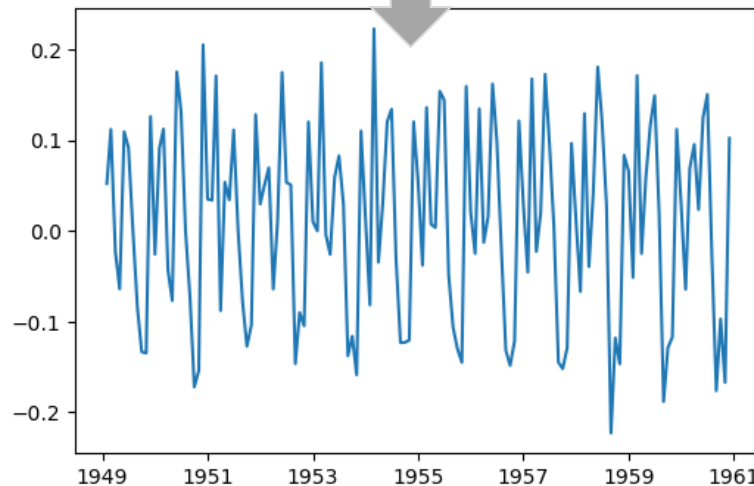
"1949-02", 0.052186

"1949-03", 0.112117

"1949-04", -0.02299

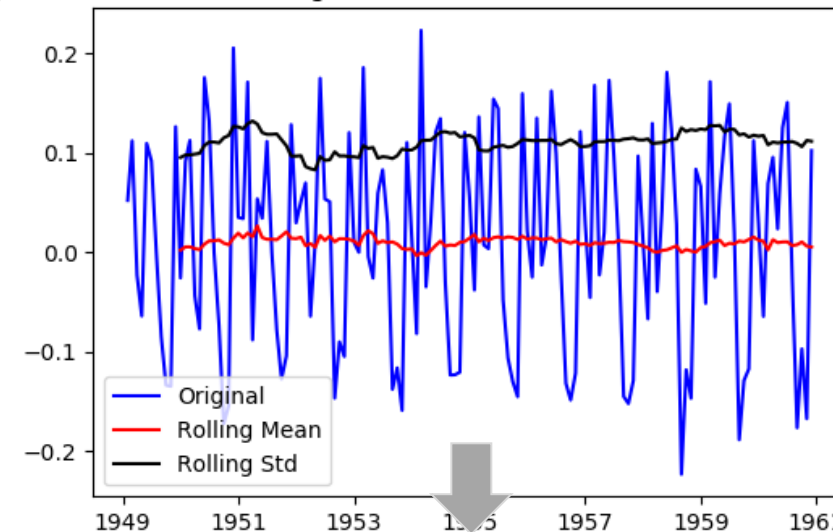
- First order differencing can be done in Pandas as:

```
ts_log_diff = ts_log - ts_log.shift()
plt.plot(ts_log_diff)
```



Test statistic is **less than the 10% critical value**, thus the TS is stationary with 90% confidence

Rolling Mean & Standard Deviation



Results of Dickey-Fuller Test:

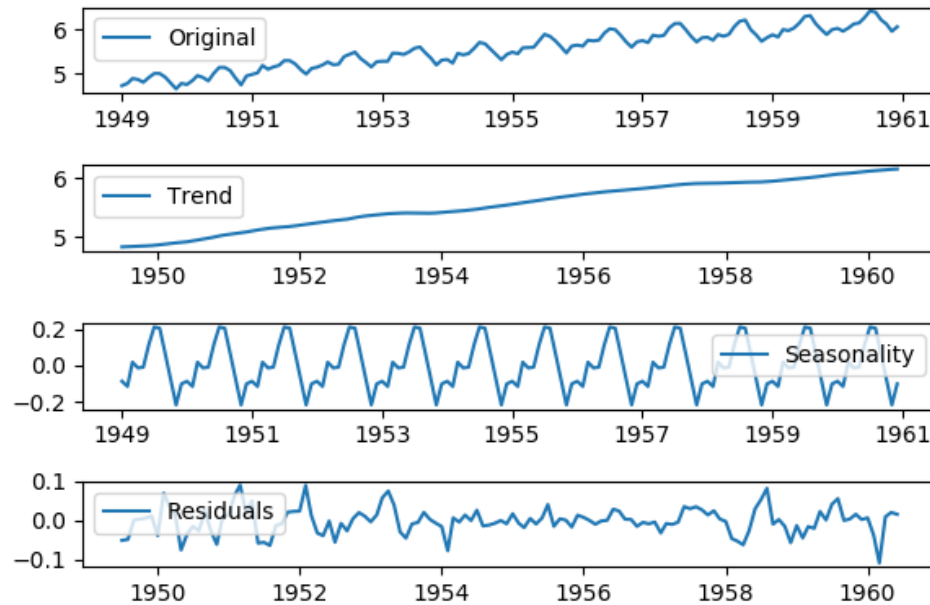
| | |
|----------------------|-----------|
| Test Statistic | -2.717131 |
| Critical Value (1%) | -3.482501 |
| Critical Value (10%) | -2.578960 |
| Critical Value (5%) | -2.884398 |

Eliminating Trend and Seasonality

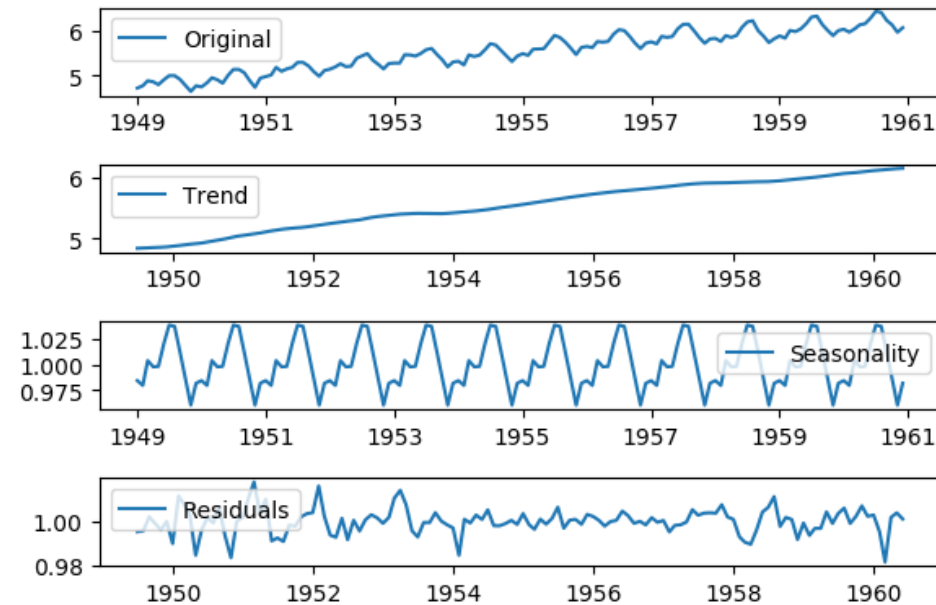


- **Decomposing:** given a TS, find the constituent components (trend, seasonality, residual or random or noise)
- Components can be combined in some way to provide the observed time series e.g.
 - Additive model:
 - $Y(t) = \text{trend} + \text{seasonality} + \text{residual}$
 - Multiplicative model:
 - $Y(t) = \text{trend} * \text{seasonality} * \text{residual}$
- There are methods to automatically decompose TS
- Python: seasonal_decompose()
 - Install statsmodels library: `conda install -c conda-forge statsmodels`
 - specify whether the model is additive or multiplicative

Eliminating Trend and Seasonality



Additive Model



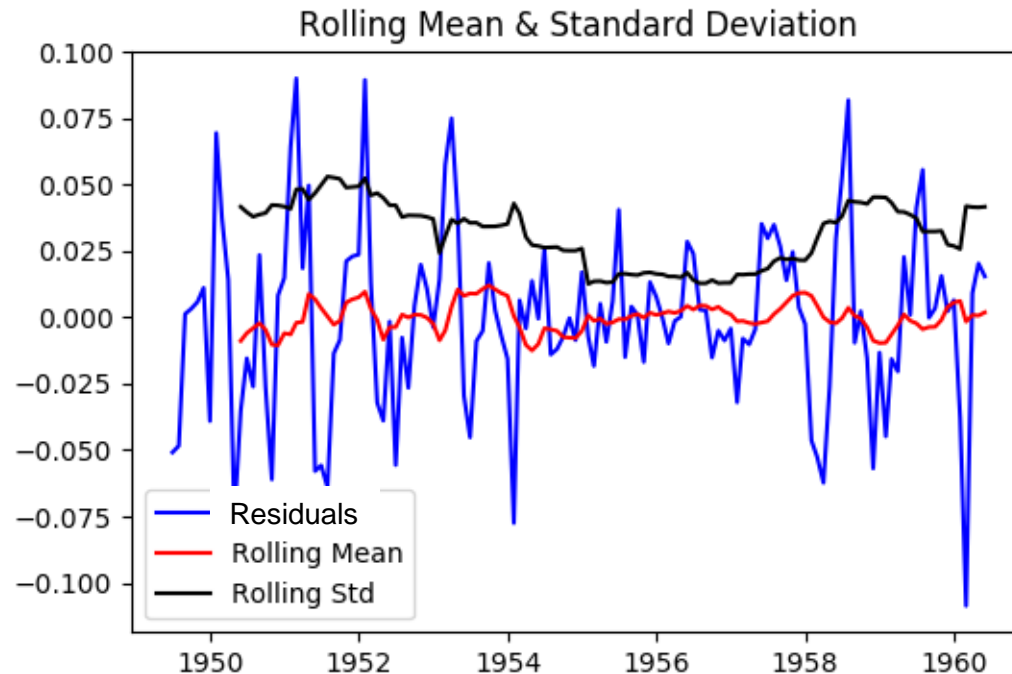
Multiplicative Model

- Trend, seasonality are separated out from data
- Residual or error is the random component of a TS which will be tested for stationarity

Eliminating Trend and Seasonality



- Test stationarity of residuals



- The Dickey-Fuller test statistic is significantly **lower than the 1% critical value** => this TS is very close to stationary
- You can try advanced decomposition techniques as well which can generate better results

Results of Dickey-Fuller Test:

| | |
|----------------------|---------------|
| Test Statistic | -6.332387e+00 |
| Critical Value (1%) | -3.485122e+00 |
| Critical Value (10%) | -2.579569e+00 |
| Critical Value (5%) | -2.885538e+00 |

Forecasting a Timeseries dataset



- After trend/seasonality elimination, we need to build a forecasting model to predict # of airline passengers
 - Train model using as train data:
 - differences (if differencing is used) → predict differences
 - residuals (if decomposition is used) → predict residuals
 - Add trend and seasonality back into predicted values (differences or residuals) to evaluate the predicted # of airline passengers
- Forecasting methods
 - Use statistical methods like [Holt-Winters](#), ARIMA or
 - Convert TS forecasting problem to supervised learning problem (generate features) and solve it using ML algorithms

Statistical forecasting methods



- Holt-Winters Methods
- Autoregression (AR), Moving Average (MA), Autoregressive Moving Average (ARMA), Autoregressive Integrated Moving Average (ARIMA), and Seasonal Autoregressive Integrated Moving-Average (SARIMA)

Supervised learning problem



- TS forecasting problem can be restructured as a supervised learning problem, by generating features => gain access to the suite of standard linear & nonlinear ML predictive algorithms discussed in previous labs
 - Feature generation options:
 1. Transform datetime index to a set of new features: date, hour, day of week, quarter, year, month, day of year, day of month, week of year
 2. Use the observed entity (time-dependent variable) to create new features: e.g. use previous instants of the observed entity as features.
-

1. Create features from the date time index



Original dataframe → "Month", "Passengers"
1949-01-01, 112
1949-02-01, 118
1949-03-01, 132
1949-04-01, 129
1949-05-01, 121

New dataframe

| Month | Passengers | hour | dayofweek | quarter | month | year | dayofyear | dayofmonth | weekofyear |
|------------|------------|------|-----------|---------|-------|------|-----------|------------|------------|
| 1949-01-01 | 112 | 0 | 5 | 1 | 1 | 1949 | 1 | 1 | 53 |
| 1949-02-01 | 118 | 0 | 1 | 1 | 2 | 1949 | 32 | 1 | 5 |
| 1949-03-01 | 132 | 0 | 1 | 1 | 3 | 1949 | 60 | 1 | 9 |
| 1949-04-01 | 129 | 0 | 4 | 2 | 4 | 1949 | 91 | 1 | 13 |
| 1949-05-01 | 121 | 0 | 6 | 2 | 5 | 1949 | 121 | 1 | 17 |

Y
target

X
features

1. Create features from the date time index



Original dataframe → "Month", "Passengers"
1949-01-01, 4.718499
1949-02-01, 4.770685
1949-03-01, 4.882802
1949-04-01, 4.859812
1949-05-01, 4.795791

New dataframe

reduce trend
with log
transformation

| Month | Passengers | hour | dayofweek | quarter | month | year | dayofyear | dayofmonth | weekofyear |
|------------|------------|------|-----------|---------|-------|------|-----------|------------|------------|
| 1949-01-01 | 4.718499 | 0 | 5 | 1 | 1 | 1949 | 1 | 1 | 53 |
| 1949-02-01 | 4.770685 | 0 | 1 | 1 | 2 | 1949 | 32 | 1 | 5 |
| 1949-03-01 | 4.882802 | 0 | 1 | 1 | 3 | 1949 | 60 | 1 | 9 |
| 1949-04-01 | 4.859812 | 0 | 4 | 2 | 4 | 1949 | 91 | 1 | 13 |
| 1949-05-01 | 4.795791 | 0 | 6 | 2 | 5 | 1949 | 121 | 1 | 17 |

Y
target

X
features

1. Create features from the date time index



Original dataframe → "Month", "Passengers"
1949-02-01, 0.052186
1949-03-01, 0.112117
1949-04-01, -0.022990
1949-05-01, -0.064022
1949-06-01, 0.109484

New dataframe

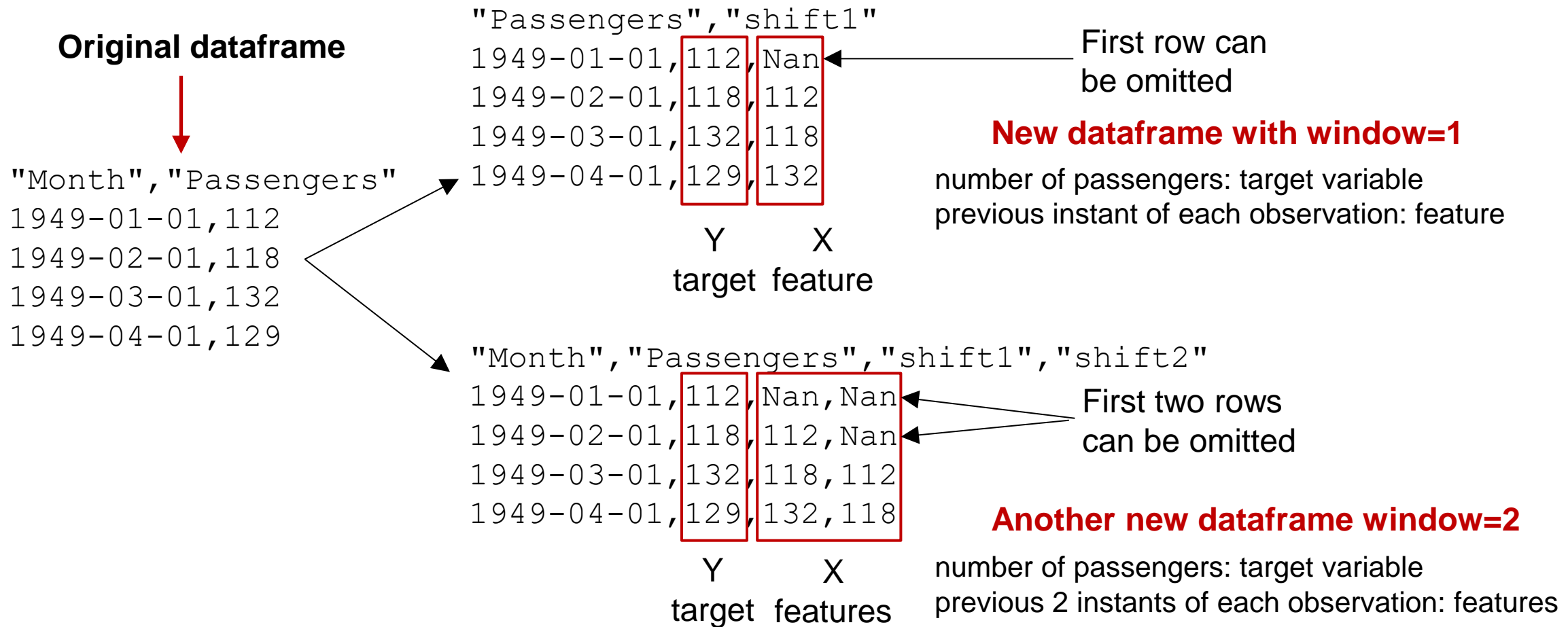
reduce trend/
seasonality
with differencing

| Month | Passengers | hour | dayofweek | quarter | month | year | dayofyear | dayofmonth | weekofyear |
|------------|------------|------|-----------|---------|-------|------|-----------|------------|------------|
| 1949-02-01 | 0.052186 | 0 | 1 | 1 | 2 | 1949 | 32 | 1 | 5 |
| 1949-03-01 | 0.112117 | 0 | 1 | 1 | 3 | 1949 | 60 | 1 | 9 |
| 1949-04-01 | -0.022990 | 0 | 4 | 2 | 4 | 1949 | 91 | 1 | 13 |
| 1949-05-01 | -0.064022 | 0 | 6 | 2 | 5 | 1949 | 121 | 1 | 17 |
| 1949-06-01 | 0.109484 | 0 | 2 | 2 | 6 | 1949 | 152 | 1 | 22 |

Y
target

X
features

2. Create features using previous instants



We can also reduce trend / seasonality by applying log-transformation and differencing on the target variable.

2. Create features using previous instants



- The use of prior time steps to predict the next time step is called the **sliding window method**
 - In statistics and time series analysis, this is called a lag or lag method
 - The number of previous time steps is called the window width or size of the lag
-

Train / test splitting



- Using `train_test_split` on time series data is not appropriate because it **randomly splits the data**, which **breaks the temporal order**
 - Time series data has an inherent sequence, where past values influence future values. If you randomly split the data, you **risk "leaking" information from the future into the training set**, leading to unrealistic model performance during training and testing.
 - Need for **time-aware splitting** methods that ensure that the test set consists of data points occurring after all the data points in the training set
 - This preserves the time order, preventing data leakage and allowing you to evaluate the model's performance as it would in real-world forecasting
-

Train / test splitting



1. Simple Train/Test Split

- use a basic split by defining the training set as the data up to a certain date and the test set as the data after that date
- works well for single evaluation

OPTION 1: Assuming data is your DataFrame

```
train_size = int(len(data) * 0.8) # Use 80% of data for training  
train, test = data[:train_size], data[train_size:]
```

OPTION 2: Or if your data has a datetime index and you want to split by a specific date

```
train = data[data.index < '2023-01-01']  
test = data[data.index >= '2023-01-01']
```

Train / test splitting



2. Using TimeSeriesSplit from sklearn for Cross-Validation

- TimeSeriesSplit provides a way to perform cross-validation for time series by creating a series of train/test splits where the training set includes all previous data up to the test set. This is helpful for validating the model in a time series context.

```
from sklearn.model_selection import TimeSeriesSplit

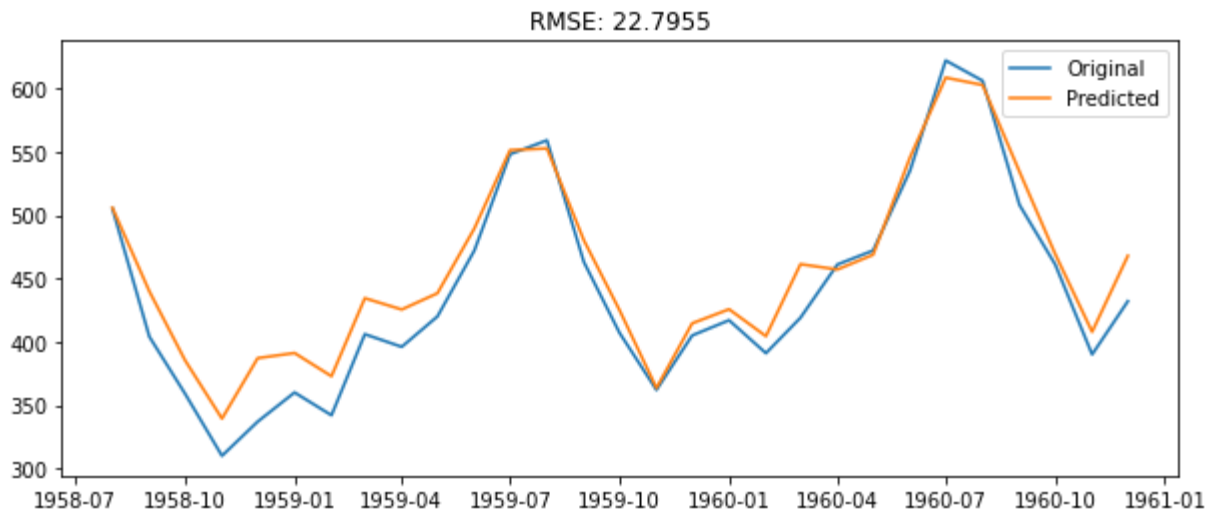
# Initialize time series splitter
tscv = TimeSeriesSplit(n_splits=5)

for train_index, test_index in tscv.split(data):
    train, test = data.iloc[train_index], data.iloc[test_index]
    # Train your model on `train`, evaluate on `test`
    # Store performance measure (MSE, RMSE, r2) value in a list
# print performance average
```

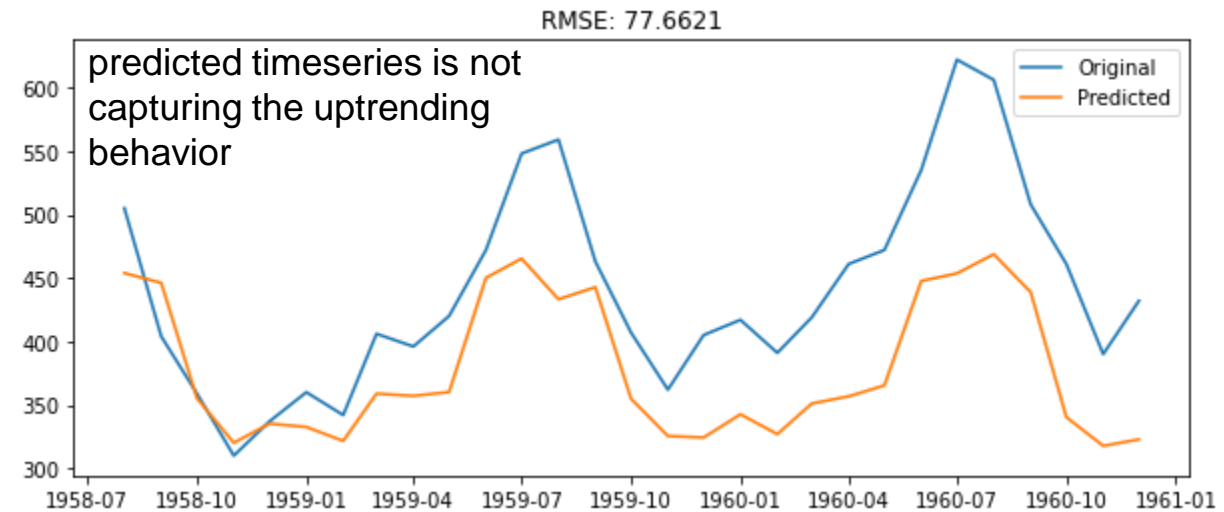
Some results...



- Features created from the **date time index**
- Simple train/test split: 80% (01/1949 – 06/1957) / 20% (07/1957 – 12/1960)
- Applied ML algorithm: RandomForest (100 estimators)



Log-transformation and differencing applied



NO Log-transformation and differencing applied

Thank you for your patience



University of Cyprus
Department of
Computer Science

Good luck with your exams

APPENDIX A:

Clustering / Classification of Time series



University of Cyprus
Department of
Computer Science

Clustering and classification

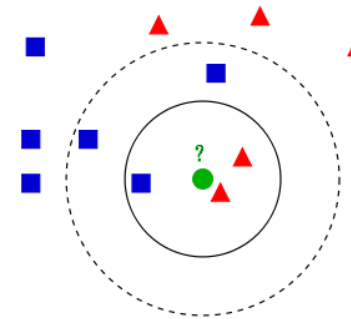


- **Time series clustering** is to partition time series data into groups based on similarity or distance, so that time series in the same cluster are similar.
 - **Time series classification** is to build a classification model based on labelled time series and then use the model to predict the label of unlabelled time series.
-

How to use clustering & classification?



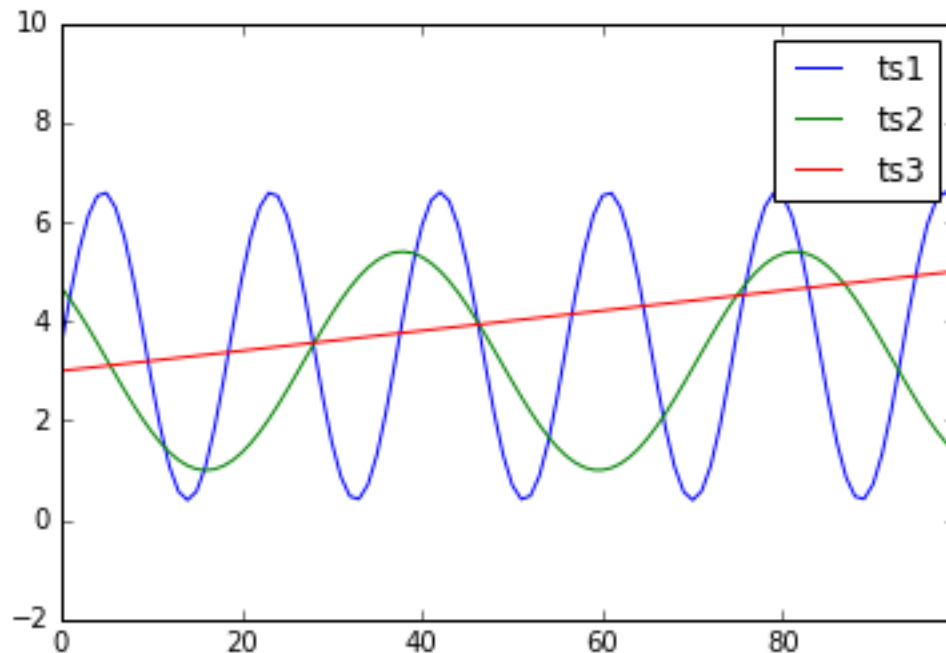
- One approach [extracting features]
 - Extract features from time series:
 - mean, maximum, minimum, other differential features
 - Use well-known algorithms with these features to make a prediction
 - Clustering: kMeans / Classification: Naive Bayes, SVMs, etc.
- Another approach [w/o features, use time series]
 - k-NN for classification
 - For a given time series test sample that you want to predict, find the most **similar (small distance) time series** samples in the training set and use its corresponding output as the prediction.
 - kMeans for clustering
 - points/centroids are time series; notion of **distance** needed.
 - Find a good similarity (distance) measure is not trivial



Similarity measures in time series



- Good similarity measure:
 - small changes in two time series result in small changes in their similarity
- How about Euclidean distance?
 - **true** for changes in the **y-axis** but **false** for changes in the **time axis** (i.e. compression and stretching)



```
euclidean_distance(ts1, ts2)  
= 26.959216038
```

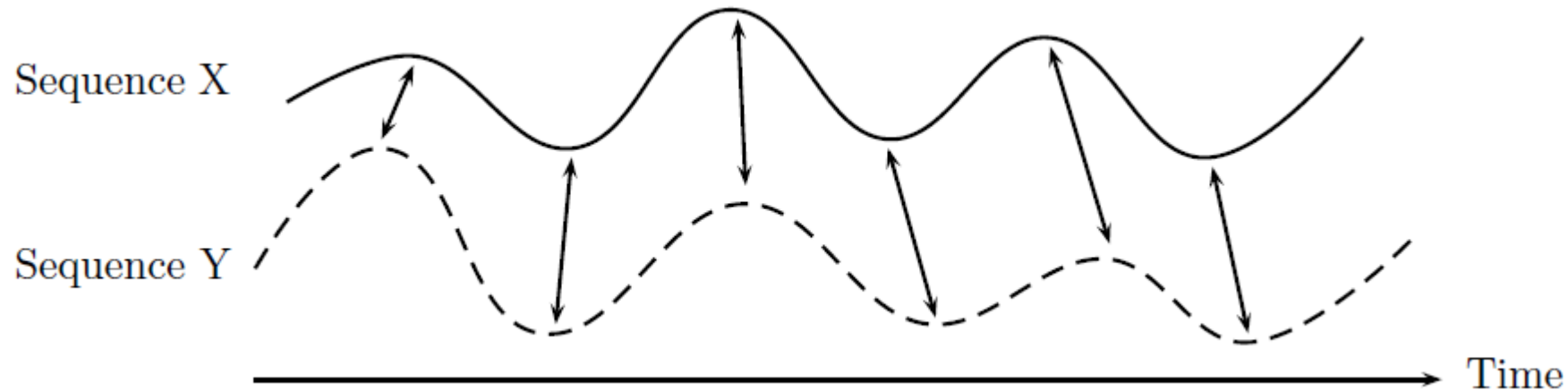
```
euclidean_distance(ts1, ts3)  
= 23.1892491903
```

According to Euclidean distance,
ts1 is more similar to ts3 than to
ts2 which contradicts our
intuition!!!

Dynamic Time Warping (DTW)



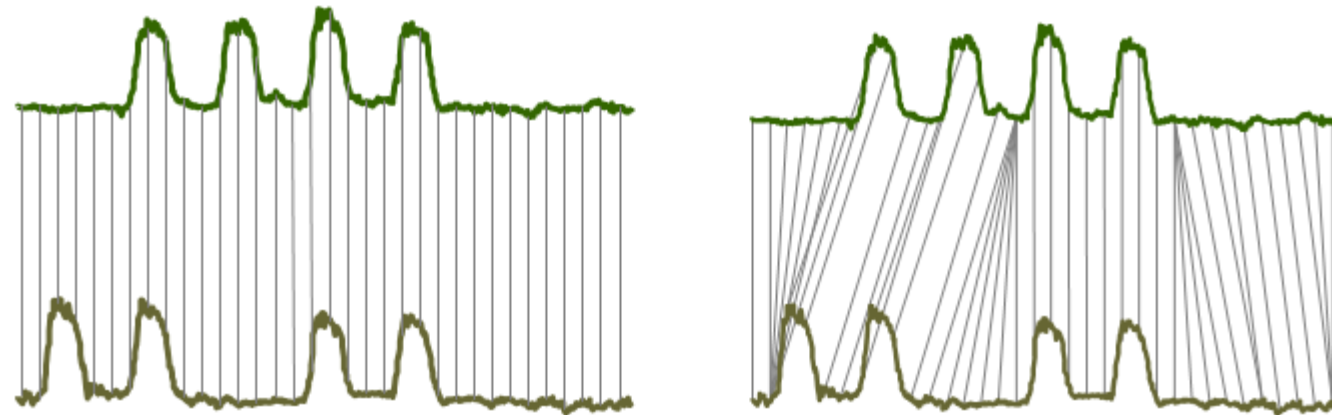
- Well-known technique to find optimal alignment between two given (time-dependent) sequences under certain restrictions
 - Allows acceleration-deceleration of sequences along the time dimension
 - Aligned points indicated by arrows in figure below



Dynamic Time Warping (DTW)



- Basic idea
 - Consider $X = x_1, x_2, \dots, x_n$, and $Y = y_1, y_2, \dots, y_n$
 - We are allowed to extend each sequence by repeating elements
 - Euclidean (or Manhattan) distance now calculated between the extended sequences X' and Y'

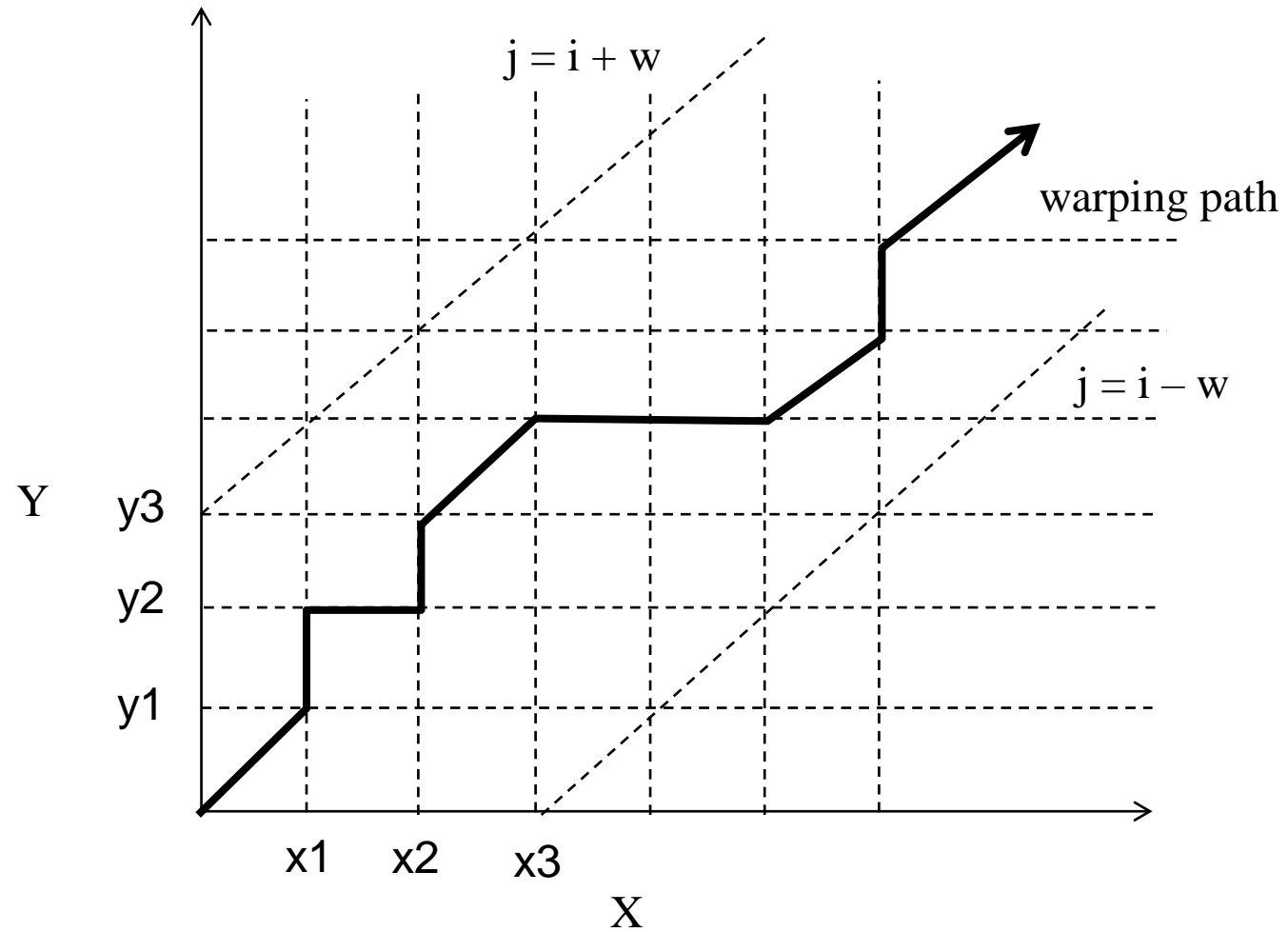


Euclidean distance

vs

DTW

Dynamic Time Warping (DTW)



Restrictions on Warping Paths



- Monotonicity
 - Path should not go down or to the left
 - Continuity
 - No elements may be skipped in a sequence
 - Speed up evaluations:
 1. Warping Window: $|i - j| \leq w$
 2. Use *LB Keogh* lower bound of dynamic time warping
 - [*LB Keogh* lower bound method is linear whereas dynamic time warping is quadratic $O(nm)$ in complexity which make it very advantageous for searching over large sets of time series]
-

Formulation



- Let $D(i, j)$ refer to the dynamic time warping distance between the subsequences

x_1, x_2, \dots, x_i

y_1, y_2, \dots, y_j

$$D(i, j) = \text{dist}(x_i, y_j) + \min \{ \begin{array}{l} D(i-1, j), \\ D(i-1, j-1), \\ D(i, j-1) \end{array} \}$$

- $\text{dist}(x_i, y_j)$ can be Euclidean or Manhattan distance
- **Faster solution: use Dynamic Programming instead of recursion**

Classification



- k Nearest Neighbors (kNN)
 - Empirically, best results when $k=1$, & DTW with euclidean distance

```
neighbors = []
for every time series i in test set,
    distances = []
    for every time series j in training set
        distances.append(j, DTW(i,j))
    distances.sort(key = index(1))
    neighbors_i = []
    for m = 1..k
        neighbors_i.append(distances[m])
    neighbors.append(neighbors_i)
return neighbors
```

Clustering



- kMeans
 - choose centroids (e.g. randomly)
 - assign sequences to clusters (based on proximity to centroids)
 - Distance: DTW with euclidean
 - recalculate centroids of clusters iteratively
-

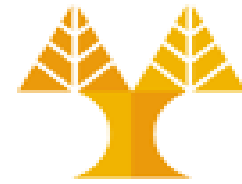
Hands on



- Download timeseries.zip
 - Lab11_DTW.py, dwt_functions.py
 - dtw_train.csv, dtw_test.csv
 - dtw_functions.py
 - recursiveDTW, recursiveDTW_window, LB_Keogh, knn, kmeans
 - Experiment with functions
 - convert knn from 1-nn to k-nn
-

APPENDIX B:

Holt-Winters and ARIMA methods



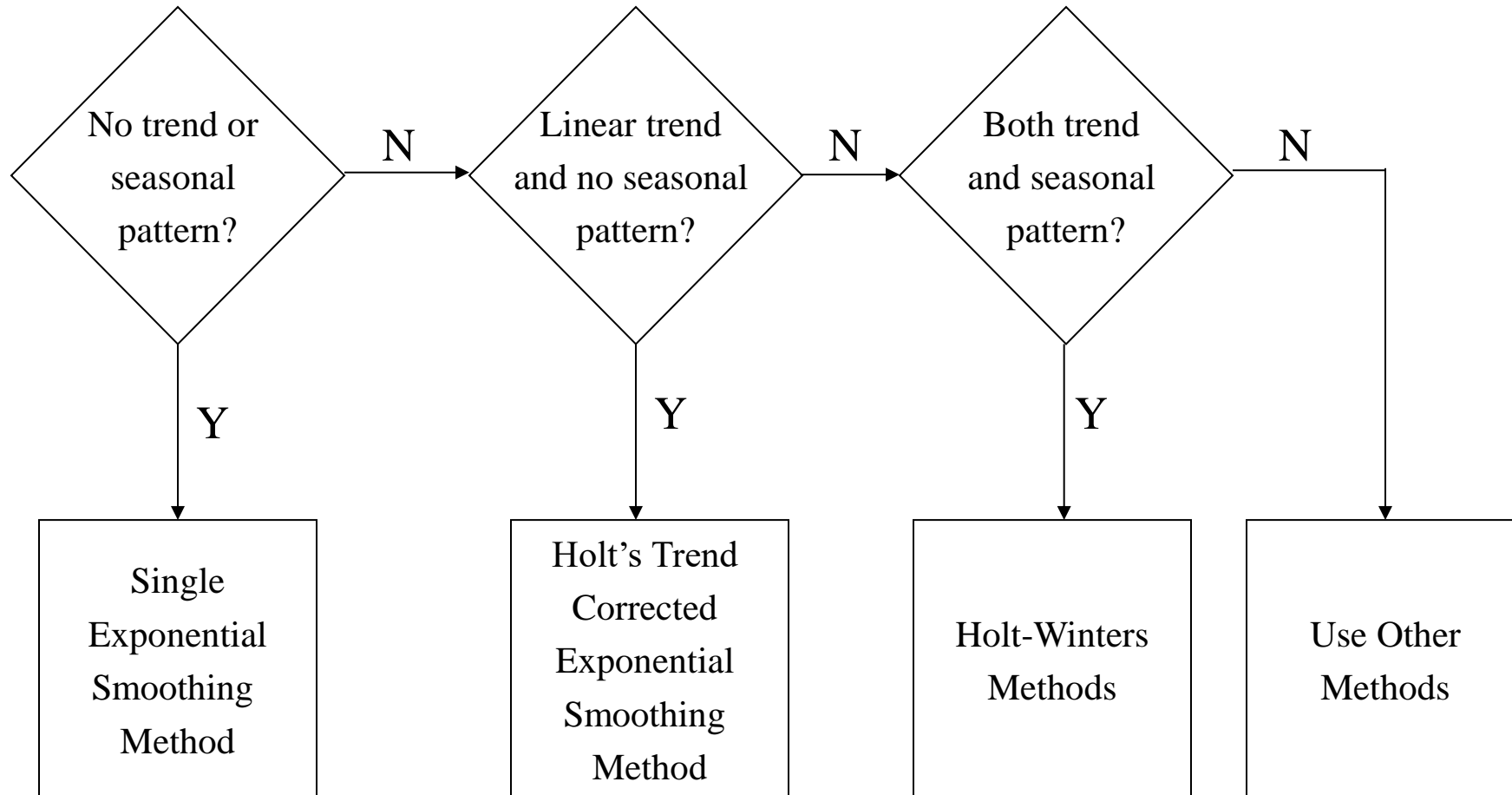
University of Cyprus
Department of
Computer Science

Holt-Winters Methods



- Univariate method: involves one input variable
 - Forecasting using **exponential smoothing**
 - Larger weights to recent observations; exponential decrease of weights
 - Effective when variables of the TS changing slowly
 - Used when data exhibits both seasonality and (linear) trend
 - Multiplicative model
 - Used for TS with increasing (multiplicative) seasonal variations
 - Additive model
 - for TS with a constant (additive) seasonal variations
 - More information can be found [here](#)
-

Data vs Methods



Multiplicative Holt-Winters Method



- $y_{t+h|h} = (l_t + h * b_t) * S_{t-m+h_m}$
Predicted values on a horizon h points Level Trend Seasonal component

- Estimate of the level

$$l_t = \alpha * \left(\frac{y_t}{S_{t-m}} \right) + (1 - \alpha) * (l_{t-1} + b_{t-1})$$

- Estimate of the trend

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

- Estimate of the seasonal factor

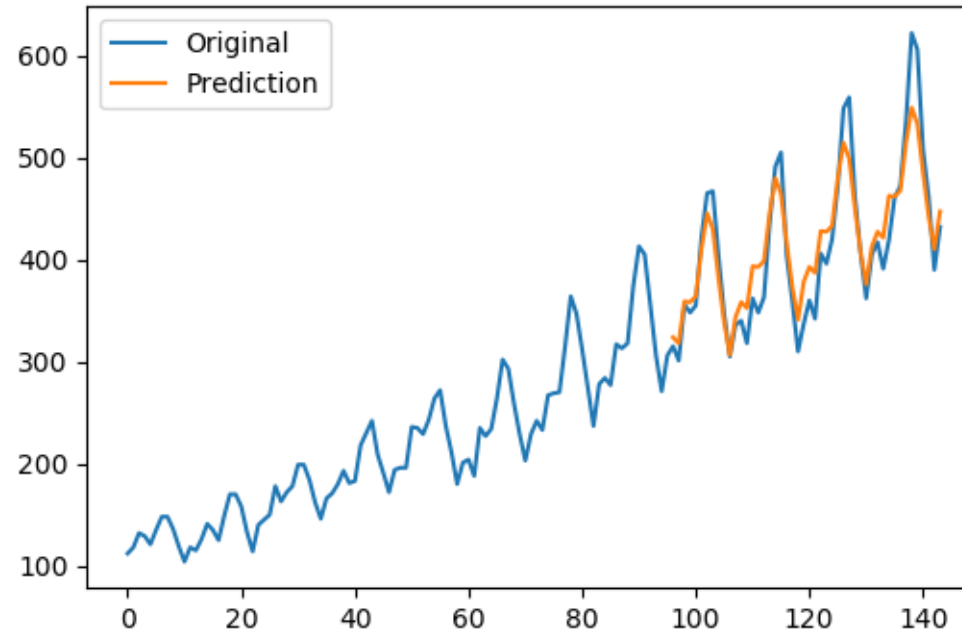
$$S_t = \gamma * \left(\frac{y_t}{l_t + b_t} \right) + (1 - \gamma) * S_{t-m}$$

where α , β , and γ are smoothing constants between 0 and 1, m = number of seasons in a year ($m = 12$ for monthly data, and $m = 4$ for quarterly data)

Holt-Winters Forecasting



- $m = 12, \alpha = 0.26, \beta = 0.19, \gamma = 0.5$
- Predict the last 4 years ($h = 48$)



ARIMA statistical model



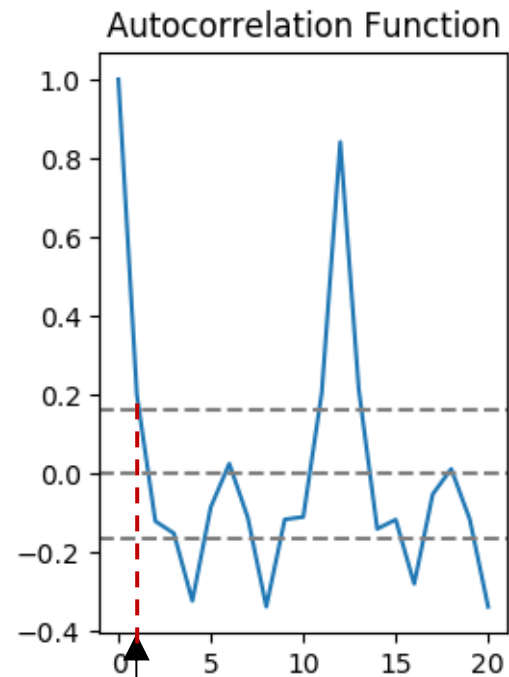
- Auto-Regressive Integrated Moving Averages
 - a linear equation, where predictors (features) depend on the parameters (p,d,q) of the ARIMA model:
 - **Number of AR (Auto-Regressive) terms (p):** AR terms are just lags of dependent variable. For instance if p is 5, the predictors for $x(t)$ will be $x(t-1) \dots x(t-5)$.
 - **Number of MA (Moving Average) terms (q):** MA terms are lagged forecast errors in prediction equation. For instance if q is 5, the predictors for $x(t)$ will be $e(t-1) \dots e(t-5)$ where $e(i)$ is the difference between the moving average at i^{th} instant and actual value.
 - **Number of Differences (d):** These are the number of nonseasonal differences, i.e. in this case we took the first order difference. So either we can **pass that variable and put $d=0$** or **pass the original variable and put $d=1$** . Both will generate same results.
-

ARIMA statistical model



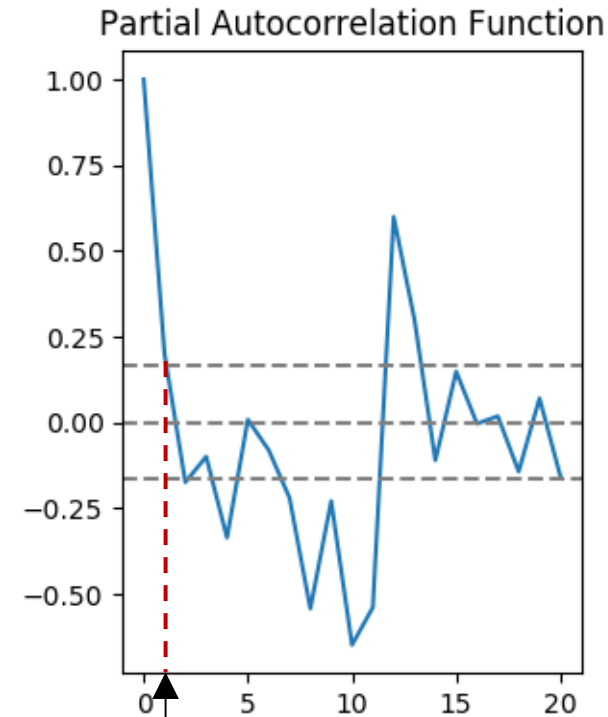
- Use 2 plots to determine the value of 'p' and 'q':
 - **Autocorrelation Function (ACF):** It is a measure of the correlation between the TS with a lagged version of itself. For instance at lag 5, ACF would compare series at time instant 't1'...'t2' with series at instant 't1-5'...'t2-5' (t1-5 and t2 being end points).
 - **Partial Autocorrelation Function (PACF):** This measures the correlation between the TS with a lagged version of itself but after eliminating the variations already explained by the intervening comparisons. E.g. at lag 5, it will check the correlation but remove the effects already explained by lags 1 to 4.
-

Choosing “best” q and p values



q=2

q – The lag value where the **ACF** chart crosses the upper confidence interval for the first time.



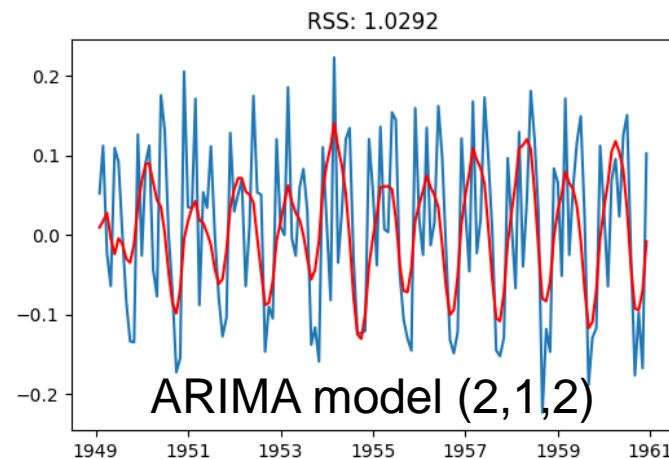
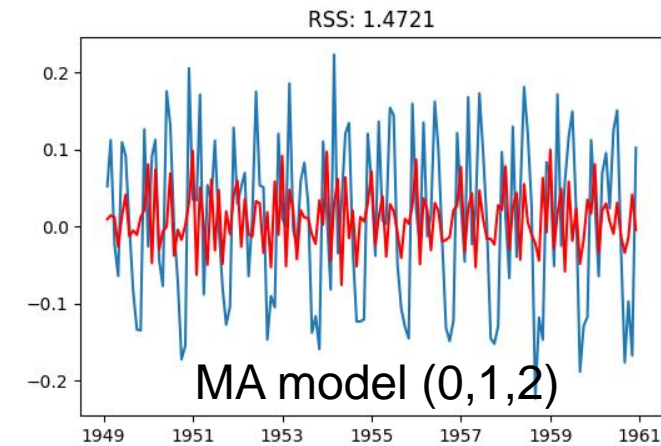
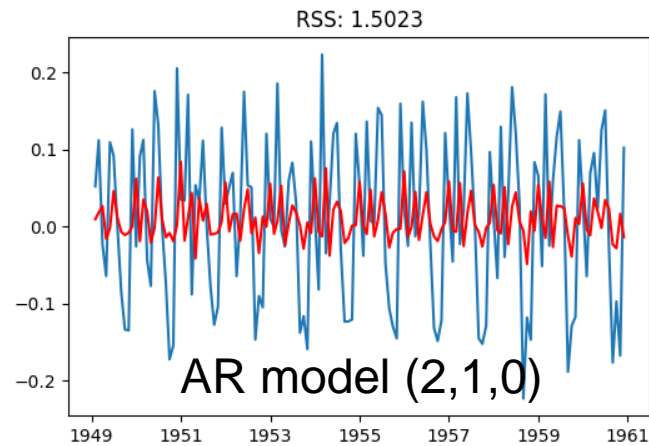
p=2

p – The lag value where the **PACF** chart crosses the upper confidence interval for the first time.

ARIMA models



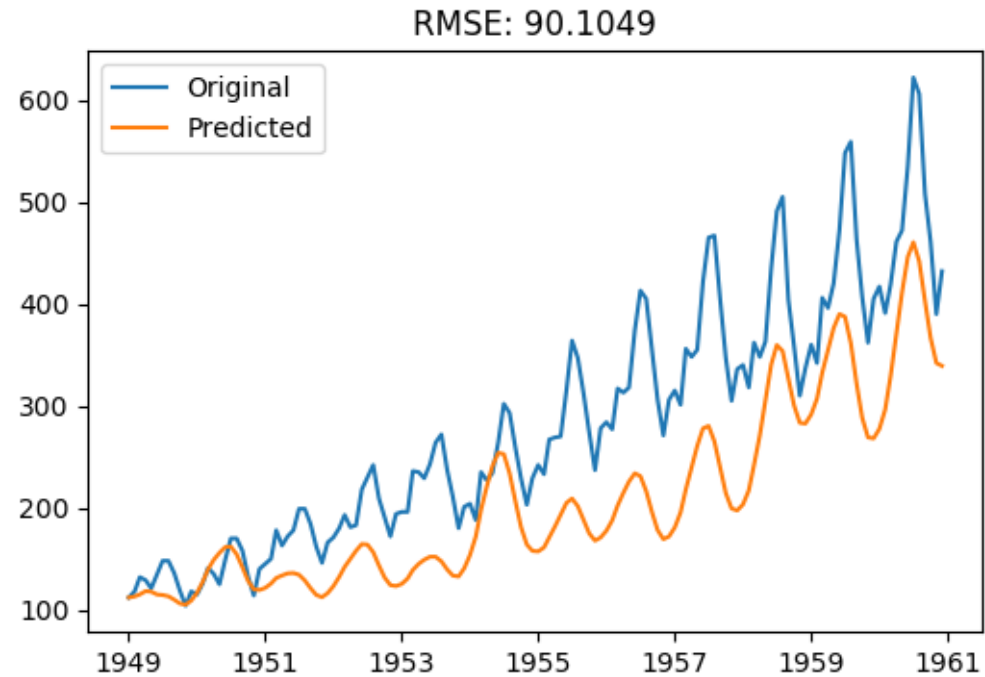
- 3 different ARIMA models (AR, MA, ARIMA) considering individual as well as combined effects



ARIMA results

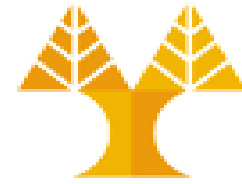


- Original TS vs predicted TS



APPENDIX C:

Exponential Weighted Moving Average (EWMA)



University of Cyprus
Department of
Computer Science

EWMA Formula



$$EWMA_t = \alpha * r_t + (1 - \alpha) * EWMA_{t-1}$$

- **alpha** = The weight decided by the user
- **r** = Value of the series in the current period
- The EWMA is a recursive function, which means that the current observation is calculated using the previous observation. The EWMA's recursive property leads to the exponentially decaying weights as shown below:

$$EWMA_t = \alpha * r_t + (1 - \alpha) * (\alpha * r_{t-1} + (1 - \alpha) * EWMA_{t-2})$$

$$EWMA_t = \alpha * r_t + (1 - \alpha) * (\alpha * r_{t-1} + (1 - \alpha) * (\alpha * r_{t-2} + (1 - \alpha) * EWMA_{t-3}))$$

EWMA Formula



- The process continues until we reach the base term $EWMA_0$. The equation can be rearranged to show that the $EWMA_t$ is the weighted average of all the preceding observations, where the weight of the observation r_{t-k} is given by: $\alpha * (1 - \alpha)^k$
 - Since alpha is between 0 and 1, the weight becomes smaller as k becomes larger. In other words, as we go back further in history, the weight becomes smaller. The fact is illustrated in the chart below, which plots the weights of observation as k increases for different choices of the parameter alpha.
-

EWMA Weights

