COMP47670

Time Series Analysis

Slides by Derek Greene

UCD School of Computer Science

Spring 2019

Static v Dynamic Data

- So far we have largely focused on static datasets, which are collected at a fixed point in time and then no longer updated.
- In many situations we do not have a single static dataset. The data is dynamic, arriving periodically or in a continuous stream.
- In dynamic scenarios, we will need to apply analysis methods that can be efficiently updated to include new data.
- Often we will only want to take into account recent history, rather than all older historical data which may longer be relevant.

- **Example**: "Want to develop a retail sales model that remains accurate as the business gets larger… However, the underlying source is changing; the business is growing, and so your guess of the sales given the previous few days of sales is probably going to be different from last year. So last year's data, when the business was small, is really not relevant to this year, when the business is large." - http://blog.bigml.com

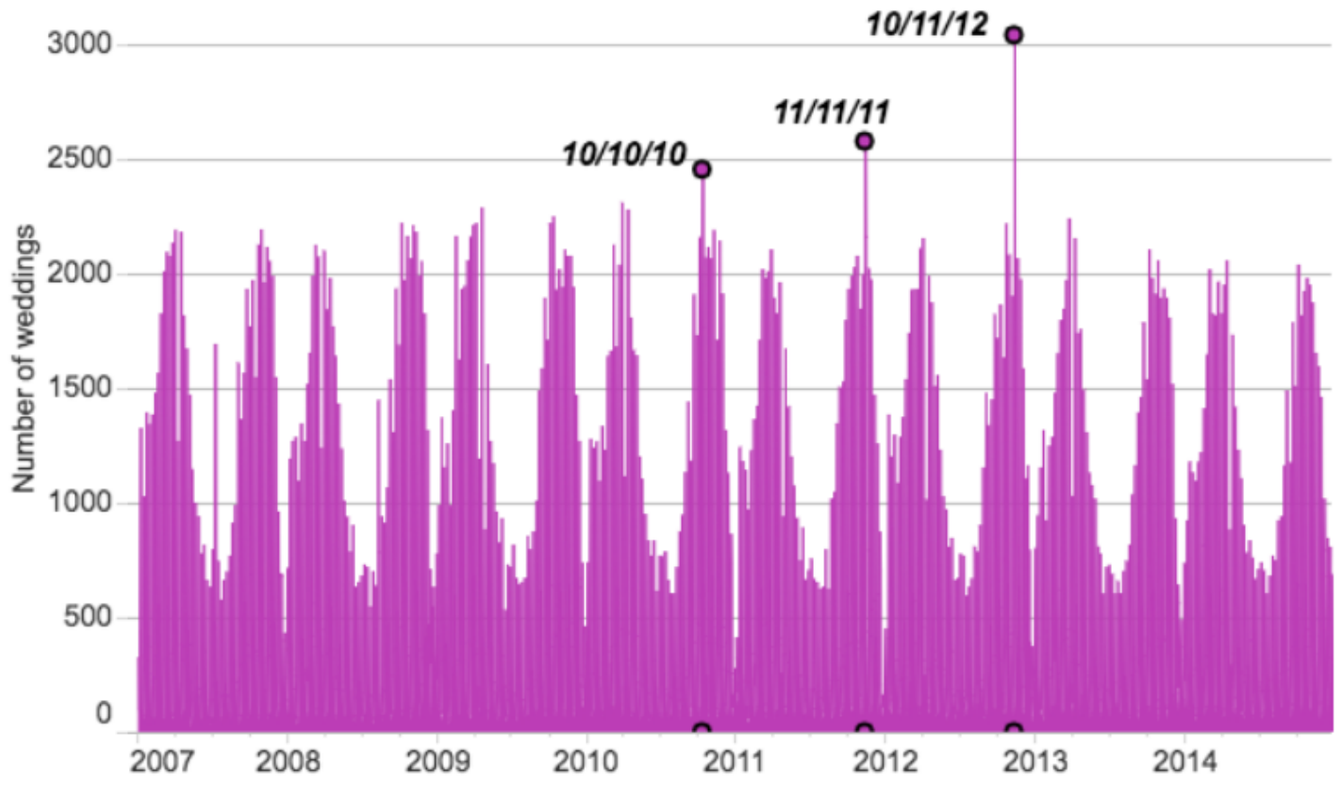
Time Series Analysis

- Time series: A dataset consisting of the values of a function sampled across different points in time.
- Time series data arise in many applications, from financial trading, to natural occurring phenomena.
- We will often plot the time series, and then look for trends, periodic behaviour, seasonal variations, step changes, outliers.

Example:
Weddings in Australia

The most popular wedding dates form repeating or sequential number patterns.

http://smh.com.au

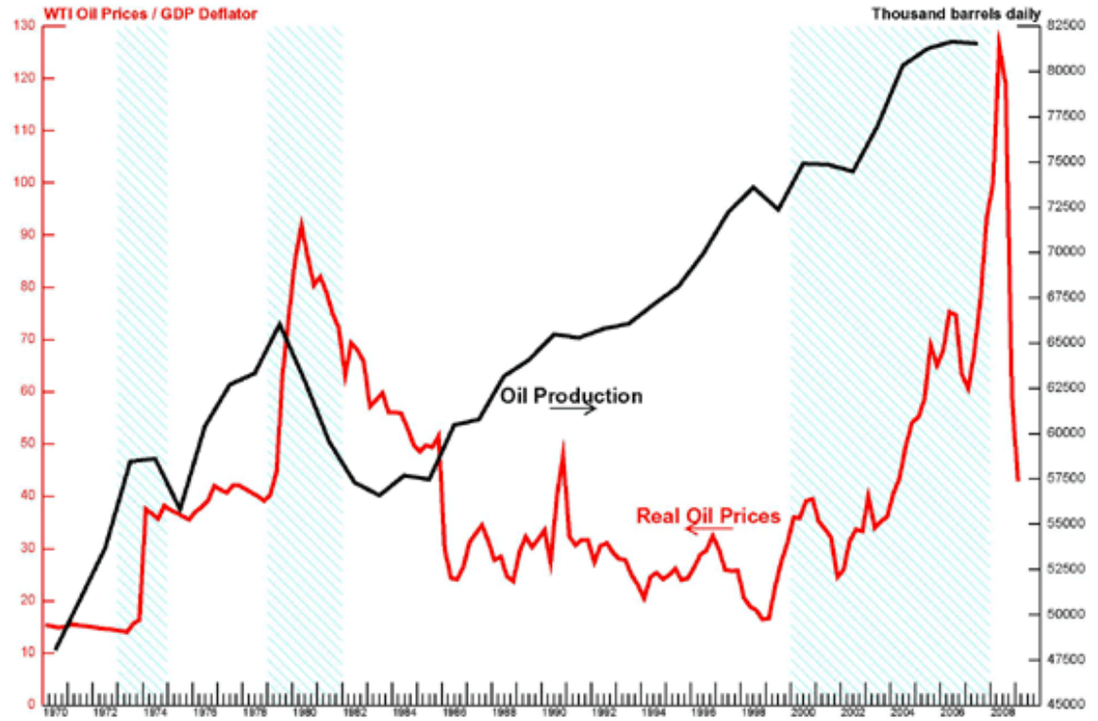


Graphic: Inga Ting | Source: ABS 2015

Time Series Data: Examples

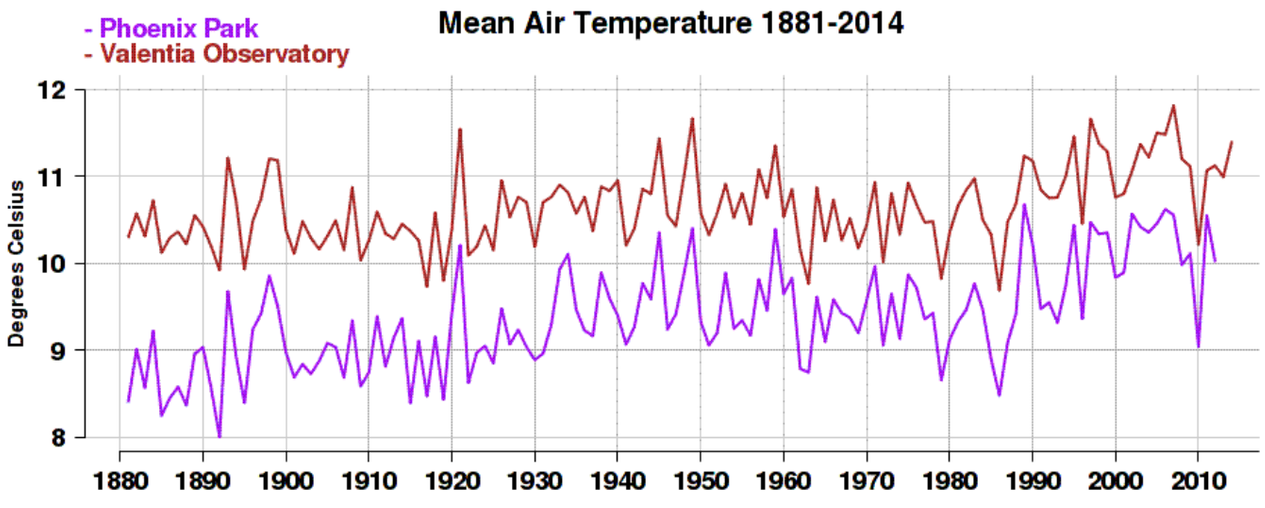
Time series analysis is applied in a range of domains.

In financial data, the price of any asset as a function of time naturally gives a time series.

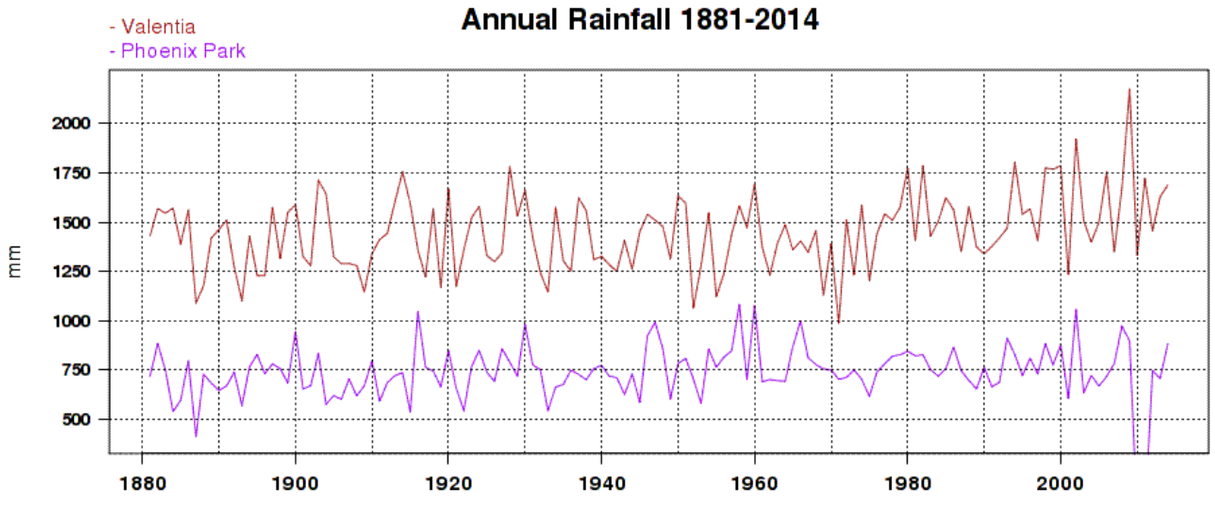


http://federalreserve.gov

Meteorological data is often represented using time series plots.

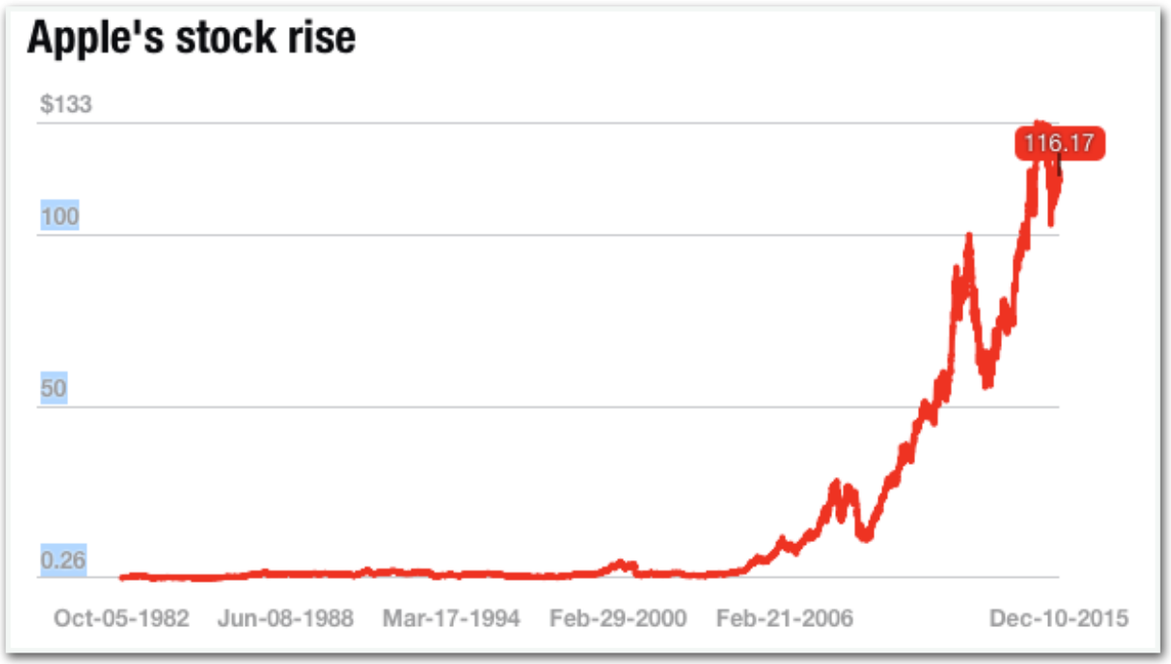


http://met.ie



Characterising Time Series Data

- Key questions when looking at time series data:
  - Is the source sampled at equally-spaced intervals?
  - How long and/or rapidly growing is the available series?
  - What is the best resolution/frequency to inspect the data?
  - Are there noisy values in the data?
  - Are there any missing values?



http://fortune.com

http://money.cnn.com

Dates and Times in Python

- The built-in Python `datetime` module includes types for date and time data, as well as calendar-related functionality.

Type	Description
`datetime.date`	An idealized date, independent of any particular time of day.
`datetime.time`	An idealised time, independent of any particular day
`datetime.datetime`	A combination of a date and a time. Most frequently used.

```
from datetime import datetime
datetime.now()
```

```
datetime.datetime(2018, 2, 27, 10, 14, 28, 328664)
```

We can get the current local date and time by calling datetime.now()

- We can create a new `datetime` value by specifying either the date alone or both a date and a time:

```
datetime(2018, 1, 7)
```

```
datetime.datetime(2018, 1, 7, 0, 0)
```

```
datetime(2018, 1, 7, 13, 30)
```

```
datetime.datetime(2018, 1, 7, 13, 30)
```

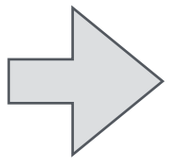
Dates and Times in Python

- From a `datetime` value we can access the individual parts of both the date and the time component.

```
from datetime import datetime
d = datetime.now()
d
```

```
datetime.datetime(2018, 2, 27, 10, 14, 28, 328664)
```

```
print(d.year)
print(d.month)
print(d.day)
print(d.hour)
print(d.minute)
print(d.second)
print(d.microsecond)
```



```
2018
2
27
10
14
28
328664
```

Extract the individual values of every part of the datetime.

Get an associated date or a time value.

```
d.date()
```

```
datetime.date(2018, 2, 27)
```

```
d.time()
```

```
datetime.time(10, 14, 28, 328664)
```

Dates and Times in Python

- Dates and times are stored to the microsecond. A `timedelta` is the temporal difference between two `datetime` values:

```
d1 = datetime(2018, 1, 7, 11, 15)
d2 = datetime(2018, 2, 15, 11, 35)
diff = d2 - d1
diff.days, diff.seconds
```

```
(39, 1200)
```

Subtract one date from another to get a time delta, expressed in days and seconds

- You can add or subtract a `timedelta` value to an existing `datetime` value to get a new shifted date:

```
d1 + timedelta(3)
```

```
datetime.datetime(2018, 1, 10, 11, 15)
```

Add 0 days and 60 seconds to the existing date

```
d1 + timedelta(0,60)
```

```
datetime.datetime(2018, 1, 7, 11, 16)
```

Add 3 days and 0 seconds to the existing date

Converting Dates to Strings

- Python `datetime` values can be formatted as strings with `strftime()` and special formatting codes.
- Each code is a placeholder for a date or time component of a datetime value.

```
from datetime import datetime
d = datetime(2018, 3, 1, 9, 30)
```

The date formatting codes change how the datetime value is displayed.

```
d.strftime('%d/%m/%y')
d.strftime('%Y-%m-%d')
d.strftime('%H:%M:%S')
d.strftime('%Y-%m-%d %H:%M')
d.strftime('%d %B %Y')
d.strftime('%a %d %Y')
```

```
01/03/18
2018-03-01
09:30:00
2018-03-01 09:30
01 March 2018
Thu 01 2018
```

Code	Meaning
`%Y`	4-digit year
`%y`	2-digit year
`%m`	2-digit month
`%d`	2-digit day
`%H`	Hour (24 hour clock)
`%h`	Hour (12 hour clock)
`%M`	2-digit minute
`%S`	seconds
`%a`	short day name
`%A`	long day name
`%b`	short month name
`%B`	long month name

Converting Strings to Dates

- We can also use the same codes to parse and extract dates from strings with the `strptime()` function.
- The specification of codes needs to match the format of the string.

```
s = "18-01-03"
datetime.strptime(s, '%y-%m-%d')
```

```
datetime.datetime(2018, 1, 3, 0, 0)
```

```
s = "01/03/2018"
datetime.strptime(s, '%d/%m/%Y')
```

```
datetime.datetime(2018, 3, 1, 0, 0)
```

```
s = "01/03/18 14:56"
datetime.strptime(s, '%d/%m/%y %H:%M')
```

```
datetime.datetime(2018, 3, 1, 14, 56)
```

Code	Meaning
`%Y`	4-digit year
`%y`	2-digit year
`%m`	2-digit month
`%d`	2-digit day
`%H`	Hour (24 hour clock)
`%h`	Hour (12 hour clock)
`%M`	2-digit minute
`%S`	seconds
`%a`	short day name
`%A`	long day name
`%b`	short month name
`%B`	long month name

Dates and Times in Pandas

- Pandas contains its own types and functionality for working with dates and times.
- The Pandas `Timestamp` type represents a specific point in time:

```
pd.Timestamp(2018, 5, 1)
```

```
Timestamp('2018-05-01 00:00:00')
```

```
pd.Timestamp("1 May 18")
```

```
Timestamp('2018-05-01 00:00:00')
```

```
pd.Timestamp(datetime(2018, 5, 1))
```

```
Timestamp('2018-05-01 00:00:00')
```

```
pd.Timestamp("1st May 2018")
```

```
Timestamp('2018-05-01 00:00:00')
```

```
d = pd.Timestamp(2018, 5, 1)
d.day, d.month, d.year
```

```
(1, 5, 2018)
```

```
d = pd.Timestamp(2018, 5, 1, 18, 35)
d.hour, d.minute, d.second
```

```
(18, 35, 0)
```

- When we want to use a sequence of Timestamps to index data in Pandas, we use a `DatetimeIndex`.

Generating Dates and Times

- We can use the `pd.date_range()` function to generate a list of dates and times, based on a specified start date, number of time periods, and frequency. This is stored as a `DatetimeIndex`.
- Example 1: Start at a specific date, and generate 7 timestamps.

```
pd.date_range('1/5/2018', periods=7, freq='D')
```

```
DatetimeIndex(['2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08',
               '2018-01-09', '2018-01-10', '2018-01-11']
```

- Example 2: Start at a specific date and time, and generate 6 time periods, separated by 4 hours each.

```
pd.date_range('1/5/2018 10:25:00', periods=6, freq='4H')
```

```
DatetimeIndex(['2018-01-05 10:25:00', '2018-01-05 14:25:00',
               '2018-01-05 18:25:00', '2018-01-05 22:25:00',
               '2018-01-06 02:25:00', '2018-01-06 06:25:00']
```

Time Series in Pandas

- Most basic kind of time series dataset in Pandas is a `Series` indexed by timestamps, in the form of a `DatetimeIndex`.

```
from datetime import datetime
import numpy as np
dates = []
for i in range(30):
    d = datetime(2018, 4, i+1)
    dates.append( d )
values = np.random.random(30)
```

Create 30 dates from April 1st to April 30th 2018, and generate 30 random values.

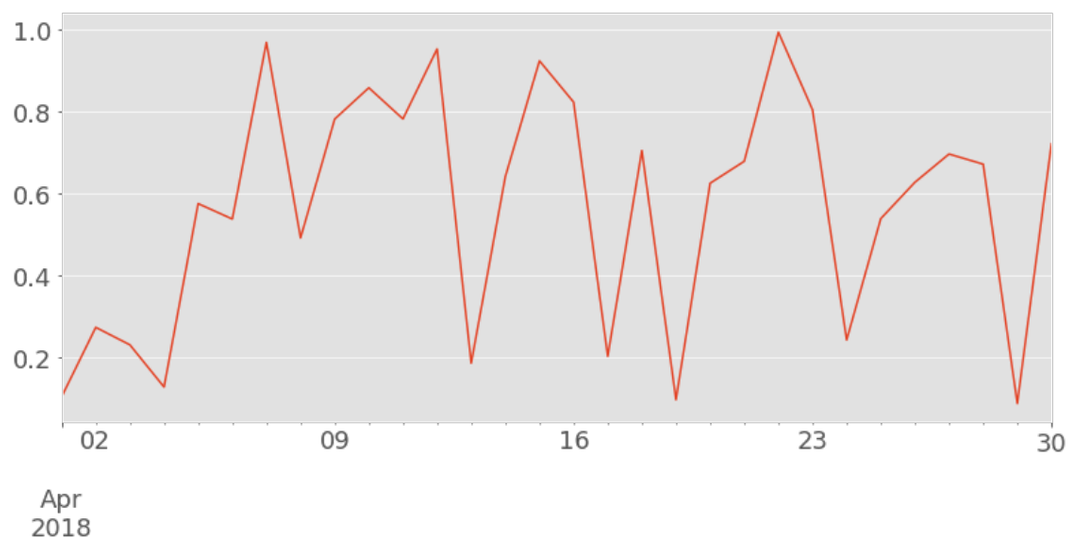
Create a Pandas Series where each of the 30 values is indexed by a different `datetime` value

```
ts = pd.Series(values, index=dates)
ts
```

```
2018-04-01    0.104432
2018-04-02    0.273930
2018-04-03    0.231213
2018-04-04    0.128279
2018-04-05    0.576016
2018-04-06    0.538410
2018-04-07    0.969452
2018-04-08    0.492322
2018-04-09    0.782416
2018-04-10    0.858801
2018-04-11    0.782802
2018-04-12    0.953605
...
```

```
ts.plot()
```

Visualise the series



Time Series in Pandas

- As with standard Pandas Series, we can use indexing and slicing to access certain parts of a time series.

```
dates = pd.date_range('01 Jan 2017',
periods=365, freq='D')
values = np.random.random(365)
ts = pd.Series(values, index=dates)
ts.head()
```

```
2017-01-01    0.526473
2017-01-02    0.804248
2017-01-03    0.040756
2017-01-04    0.816041
2017-01-05    0.045916
...
```

```
ts["2017-01-01"]
```

```
0.526473204325809
```

```
ts["2017-01-04":"2017-01-07"]
```

```
2017-01-04    0.816041
2017-01-05    0.045916
2017-01-06    0.642126
2017-01-07    0.371168
```

- As well as specifying individual start and end date/times for slicing, we can also specify slices of entire months or years.

```
ts["June 2017"]
```

```
2017-06-01    0.814349
2017-06-02    0.561968
2017-06-03    0.531489
2017-06-04    0.181534
2017-06-05    0.834928
...
```

```
ts["2017"]
```

```
2017-01-01    0.526473
2017-01-02    0.804248
2017-01-03    0.040756
2017-01-04    0.816041
2017-01-05    0.045916
...
```

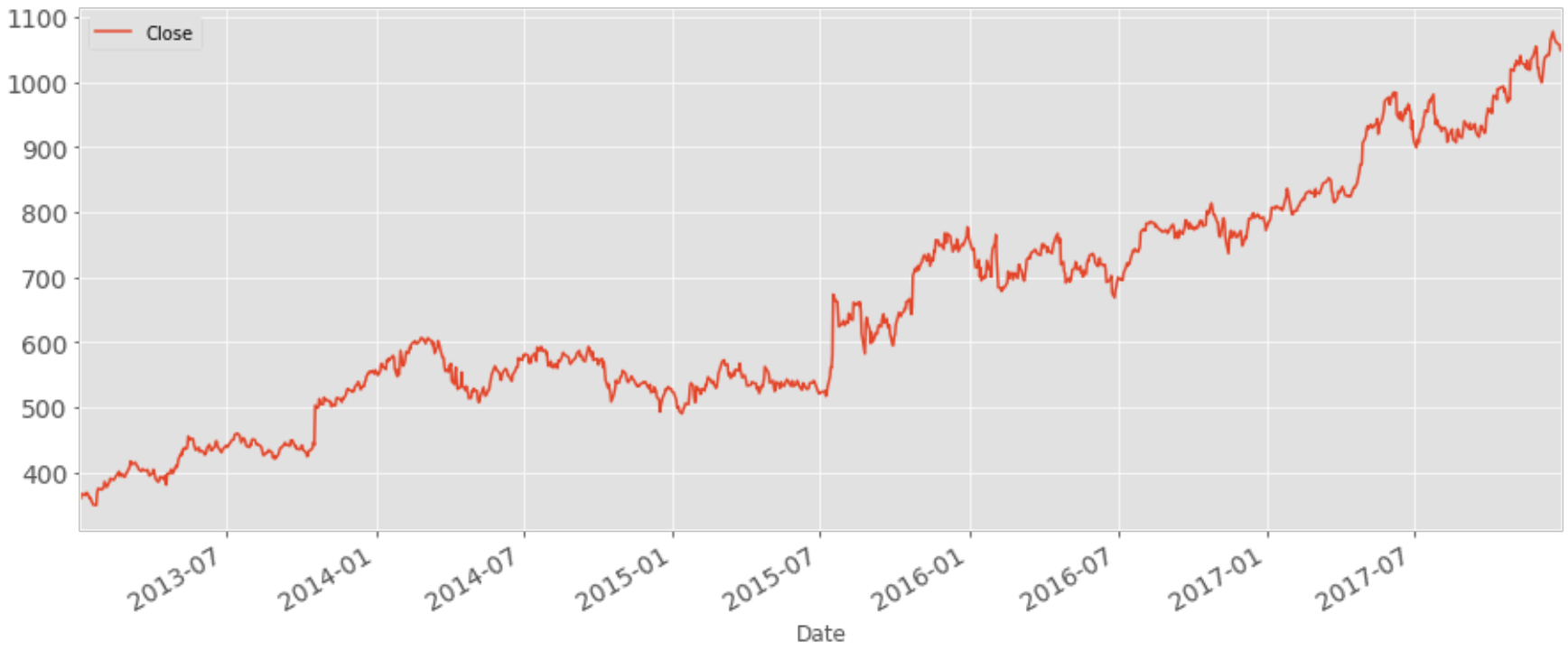
Loading Time Series in Pandas

- **Example:** Google stock data (close price)
  Load a time series dataset from a CSV file as a Pandas Data Frame. Specify `parse_dates` to parse the row index field as a date:

```
Date,Close
2013-01-02,359.288
2013-01-03,359.497
2013-01-04,366.601
2013-01-07,365.001
2013-01-08,364.281
...
```

```
ts = pd.read_csv("stock-google-close.csv", index_col="Date", parse_dates=True)
ts.head()
ts.plot(figsize=(15,6),fontsize=14)
```

Date	Close
2013-01-02	359.288
2013-01-03	359.497
2013-01-04	366.601
2013-01-07	365.001
2013-01-08	364.281



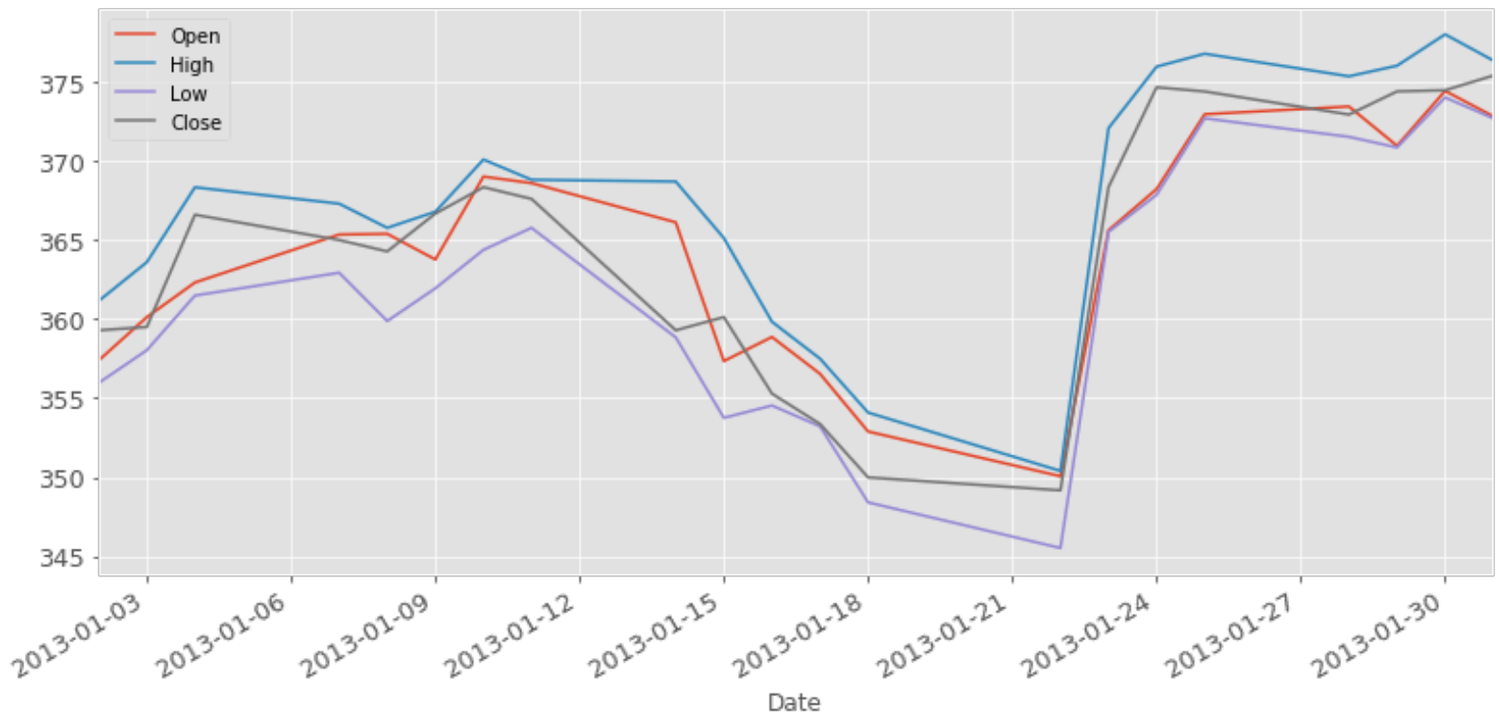
Loading Time Series in Pandas

- In many cases a single CSV file will contain multiple time series. These can be loaded as columns in a single Pandas Data Frame.

```
Data,Open,High,Low,Close
2013-01-02,357.386,361.151,355.96,359.288
2013-01-03,360.123,363.6,358.031,359.497
2013-01-04,362.314,368.339,361.489,366.601
2013-01-07,365.349,367.301,362.93,365.001
2013-01-08,365.393,365.771,359.874,364.281
...
```

```
ts = pd.read_csv("stock-google.csv", index_col="Date", parse_dates=True)
ts.head()
ts["2013-01"].plot(figsize=(13,6),fontsize=13)
```

Date	Open	High	Low	Close
2013-01-02	357.386	361.151	355.960	359.288
2013-01-03	360.123	363.600	358.031	359.497
2013-01-04	362.314	368.339	361.489	366.601
2013-01-07	365.349	367.301	362.930	365.001
2013-01-08	365.393	365.771	359.874	364.281



Resampling Time Series

- Resampling: Process of converting time series data from one frequency to another. This is done via the `resample()` function.
- We can downsample - aggregate data to a lower frequency.
- For instance, convert from day frequency to month (M) or quarter (Q) frequency, by taking the mean across all days.

```
ts.head()
```

Date	Open	High	Low	Close
2013-01-02	357.386	361.151	355.960	359.288
2013-01-03	360.123	363.600	358.031	359.497
2013-01-04	362.314	368.339	361.489	366.601
2013-01-07	365.349	367.301	362.930	365.001
2013-01-08	365.393	365.771	359.874	364.281

```
ts.resample("M").mean()
```

Date	Open	High	Low	Close
2013-01-31	364.391952	367.528095	361.896000	364.720905
2013-02-28	389.442842	393.065737	387.387053	390.433579
2013-03-31	406.682250	408.903500	404.116300	406.348900
2013-04-30	395.043636	398.543273	391.203909	395.629591
2013-05-31	434.463773	438.585227	431.823818	435.486682

Output is a new time series, one entry per month.

```
qts = ts.resample("Q").mean()
qts.head()
```

Date	Open	High	Low	Close
2013-03-31	386.421500	389.406817	384.041600	386.739250
2013-06-30	421.497422	425.006094	418.109359	421.881953
2013-09-30	441.079469	443.504109	438.108359	440.789891
2013-12-31	503.538875	507.557016	500.573344	504.363625
2014-03-31	582.527361	585.738279	576.638738	580.762262

Output is a new time series, one entry per quarter.

Resampling Time Series

- We can also upsample a time series - convert lower frequency data to a higher frequency.
- For example, take daily data and convert it to hourly data (H). Note rows that are added in between have missing values.

```
ts.resample("H").mean()
```

Date	Open	High	Low	Close
2013-01-02 00:00:00	357.386	361.151	355.96	359.288
2013-01-02 01:00:00	NaN	NaN	NaN	NaN
2013-01-02 02:00:00	NaN	NaN	NaN	NaN
2013-01-02 03:00:00	NaN	NaN	NaN	NaN
2013-01-02 04:00:00	NaN	NaN	NaN	NaN
2013-01-02 05:00:00	NaN	NaN	NaN	NaN

- Instead of having missing values (NaNs), we may want to specify a way of filling or estimating those values. Common strategies include:
  - Forward fill: Fill in values based on the last valid value (ffill)
  - Backward fill: Fill in values based on the next valid value (bfill)

```
ts.resample("H").mean().bfill()
```

Date	Open	High	Low	Close
2013-01-02 00:00:00	357.386	361.151	355.960	359.288
2013-01-02 01:00:00	360.123	363.600	358.031	359.497
2013-01-02 02:00:00	360.123	363.600	358.031	359.497
2013-01-02 03:00:00	360.123	363.600	358.031	359.497
2013-01-02 04:00:00	360.123	363.600	358.031	359.497
2013-01-02 05:00:00	360.123	363.600	358.031	359.497

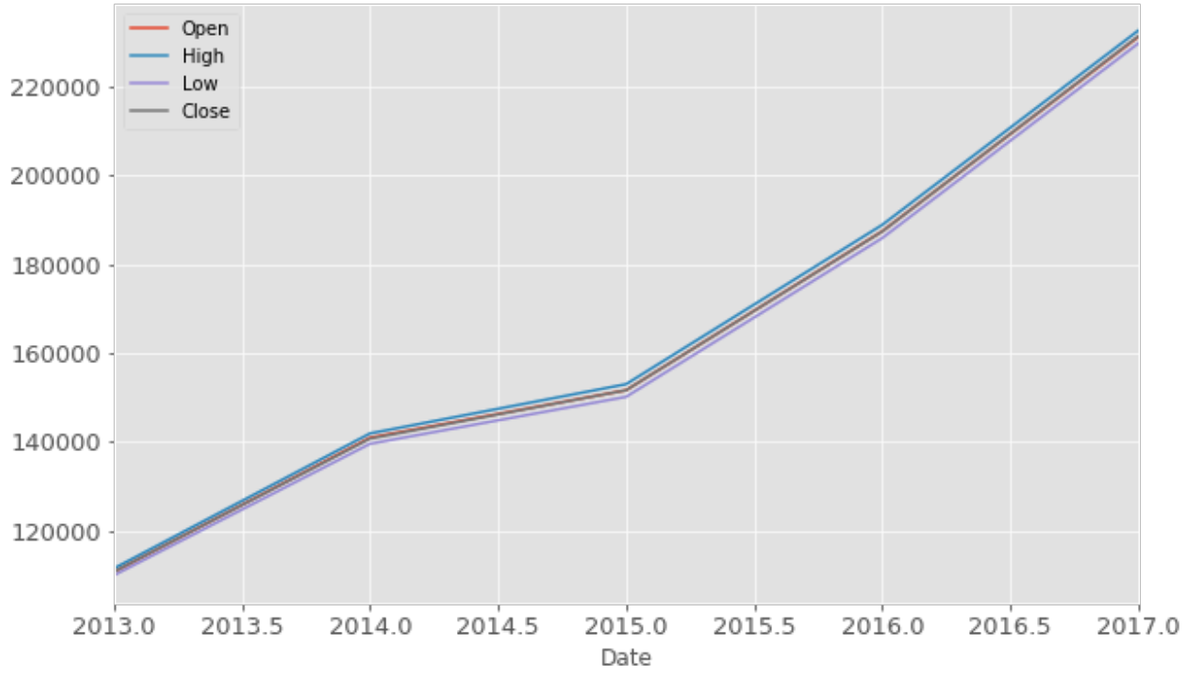
Aggregating Time Series

- Pandas has functionality for aggregating date and time based data, using the `groupby()` function.
- For example, we can group the data by year:

```
ts_year = ts.groupby(ts.index.year).sum()
ts_year.head()
```

Group by the "year" component of the index, which is a date, and calculate the sum for the year.

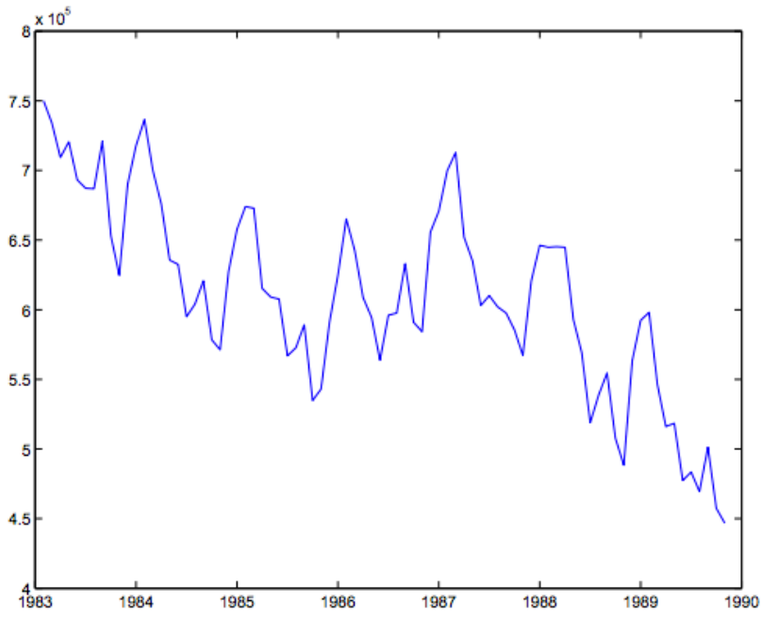
Date	Open	High	Low	Close
2013	110616.699	111432.711	109877.124	110694.625
2014	140911.951	141863.310	139517.842	140684.188
2015	151603.792	152959.948	150101.206	151590.739
2016	187420.580	188854.251	185874.672	187358.650
2017	231201.420	232651.640	229748.184	231366.990



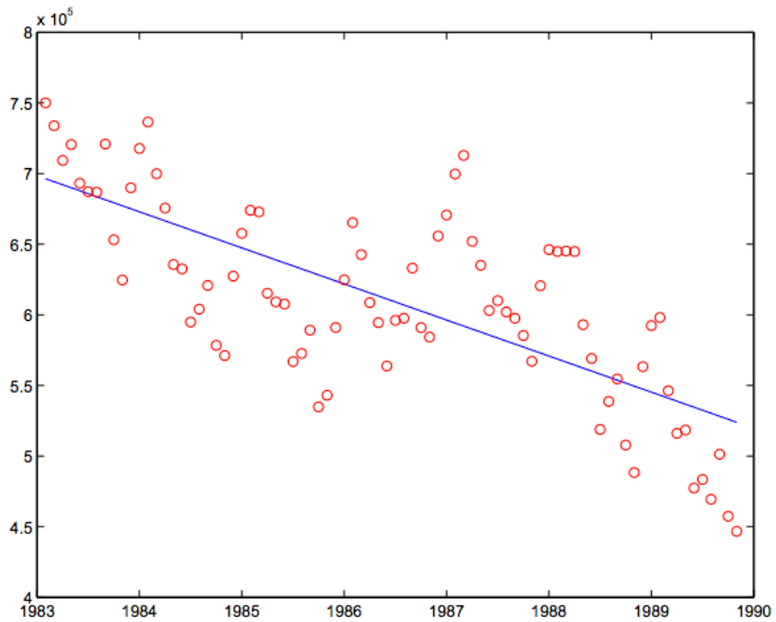
Decomposing Time Series Data

- Most time series analysis is based on the modeling assumption that the observed series is the sum of three components:
  1. Trend: A smooth function that captures persistent changes.
  2. Seasonality: Periodic variation - including daily, weekly, monthly, or yearly cycles.
  3. Noise: The random variation around the long-term trend.

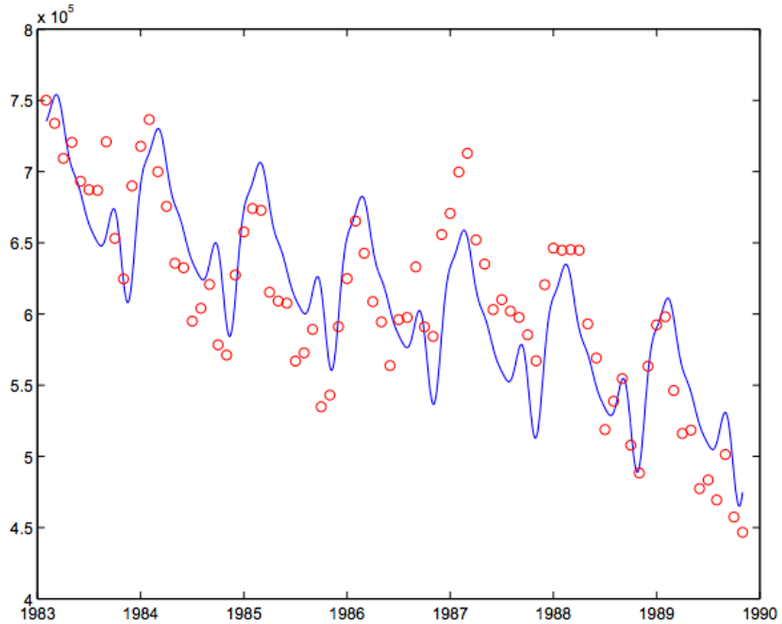
Example: Monthly unemployment rate in Australia (Hipel & McLeod, 1994)



Time Series



Trend

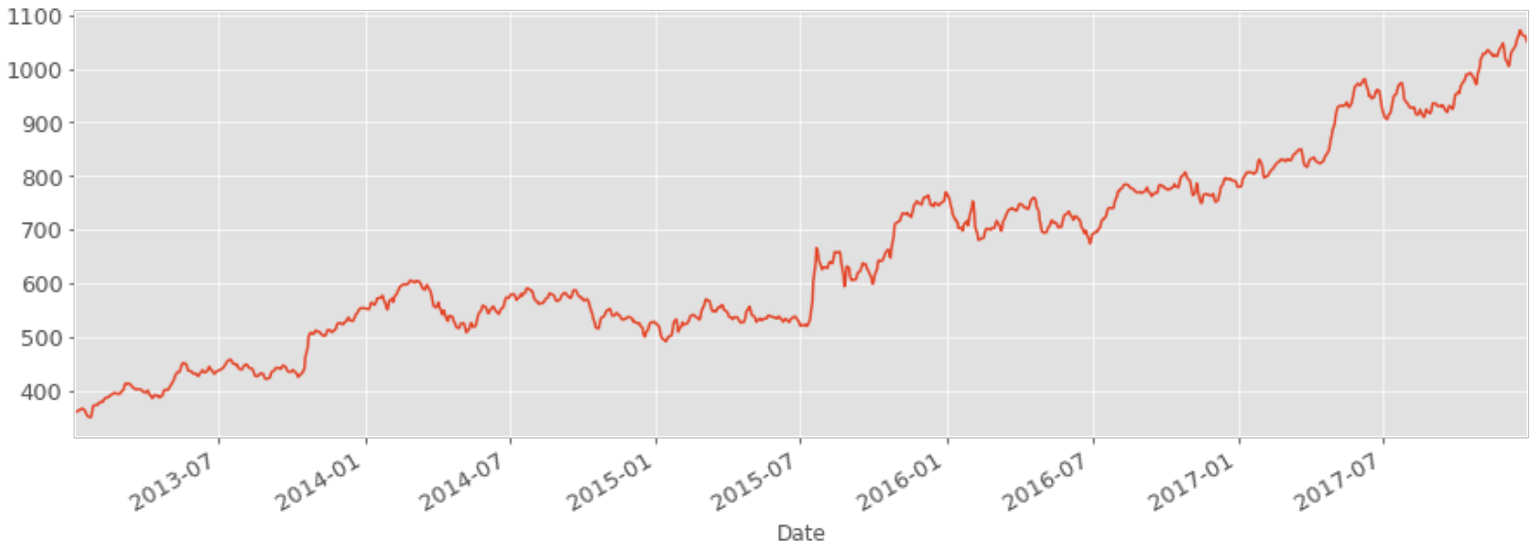


Trend & Seasonal Variation

Rolling Averages

- One way to extract a trend from a time series is to use a moving average. This divides the series into overlapping regions, called windows, and computes the average of the values in each window.
- Rolling mean: simple approach which computes the mean of the values in each window. The size of the window is the number of values it will include.
- A Pandas Series has a `rolling()` function, which takes a specified window size. We then apply a function like `mean()` to the result.

```
df = pd.read_csv("stock-google-close.csv", index_col="Date", parse_dates=True)
rm3 = df.Close.rolling(3).mean()
```

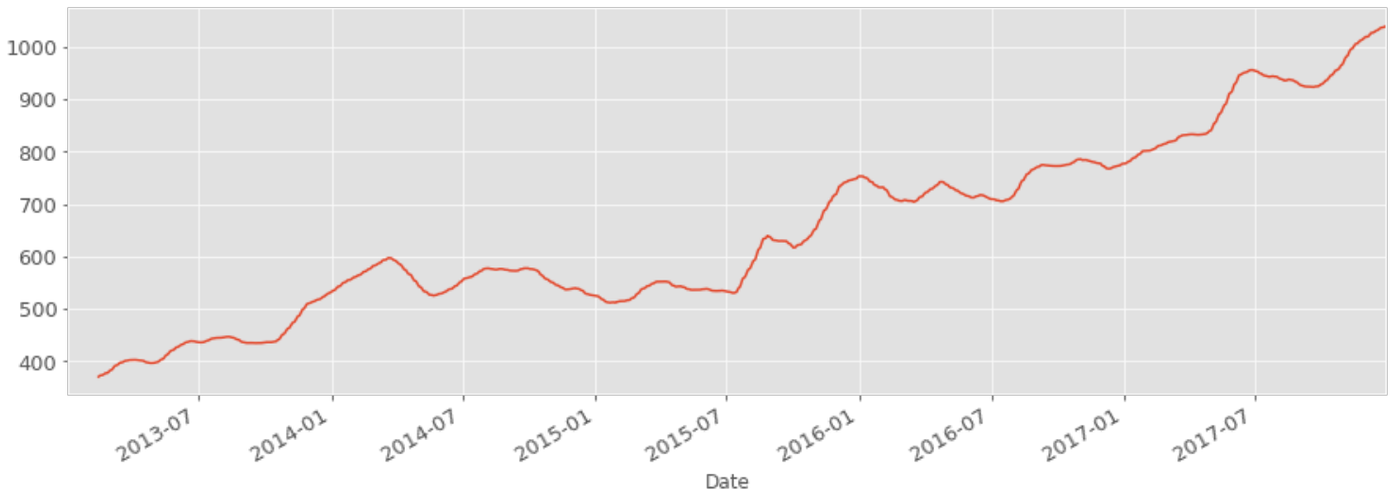


Window size 3: average 1st, 2nd & 3rd value, then 2nd, 3rd & 4th

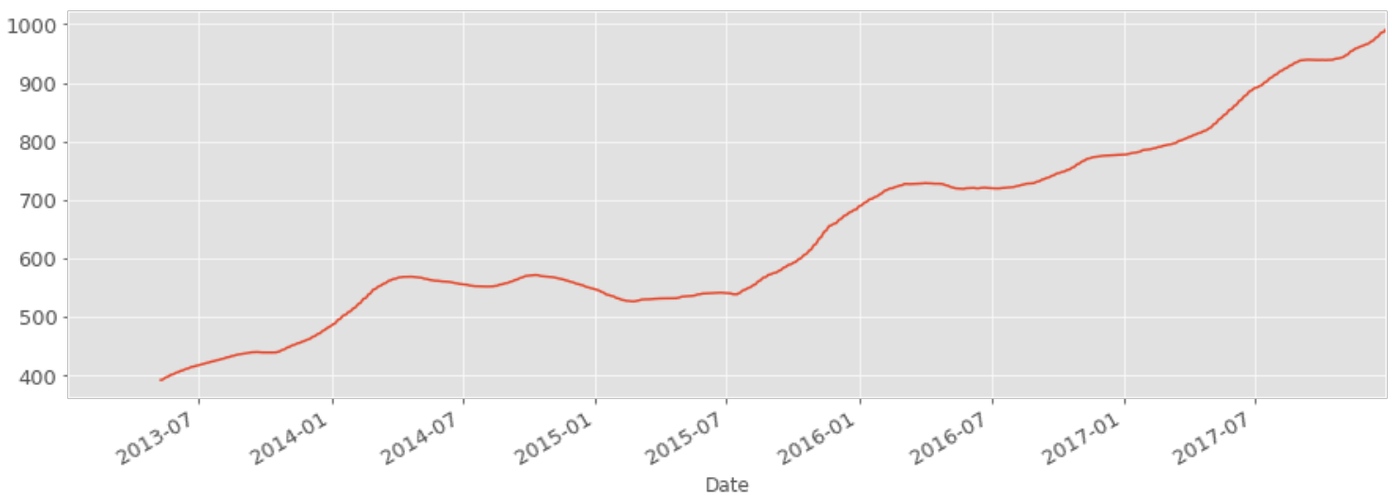
i.e. 3-day rolling mean

Rolling Averages

Google Closing stock prices: 7-day rolling mean



Google Closing stock prices: 30-day rolling mean

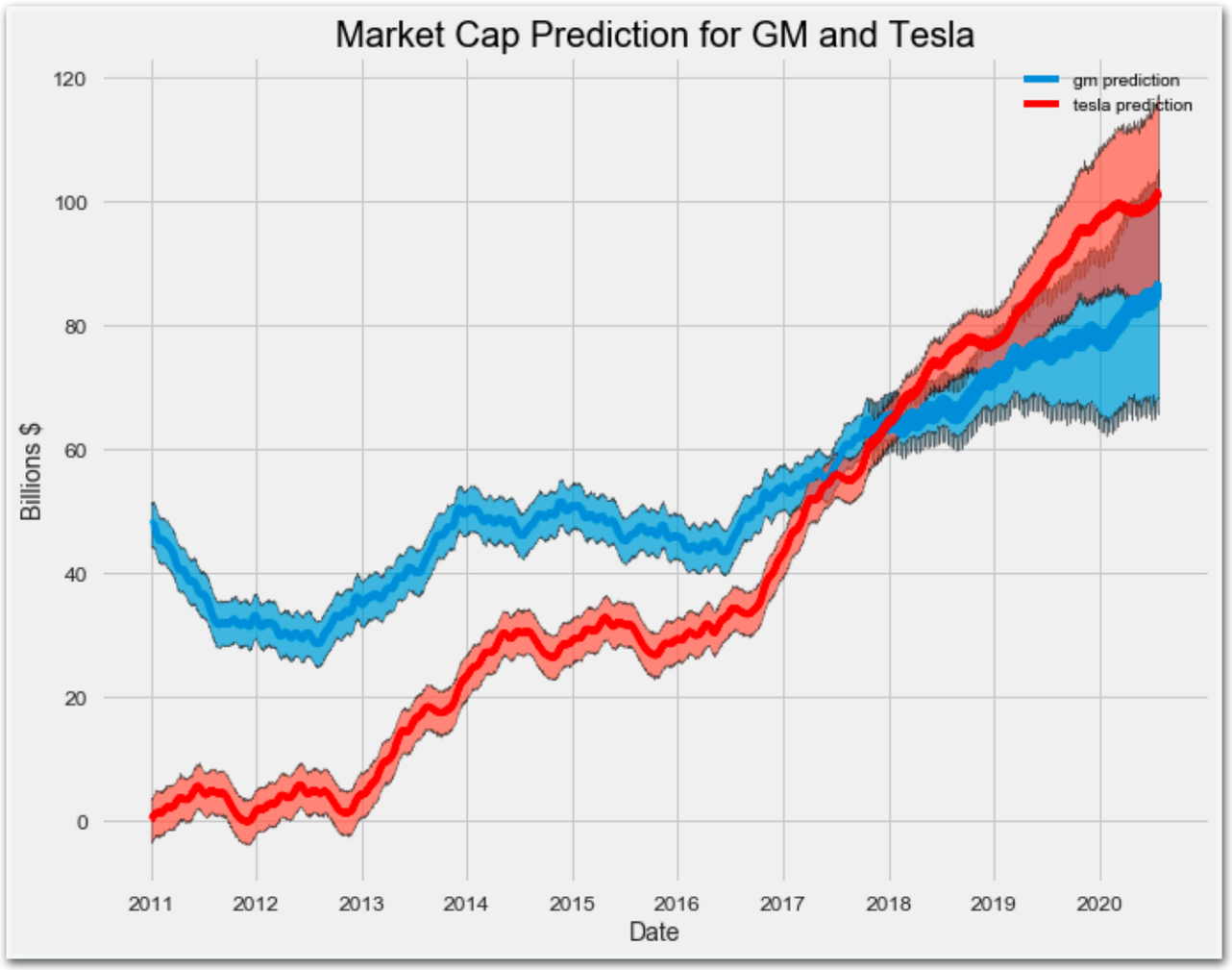


Google Closing stock prices: 90-day rolling mean

Larger window naturally leads to a smoother line.

Time Series Prediction

- As well as characterising historic timestamped data, many applications involve predicting future values of a time series based on previous trends and other information - e.g. algorithmic stock trading.



https://facebook.github.io/prophet https://keras.io