Time series analysis: PROPHET

ΕΠΛ 428: IOT PROGRAMMING

Dr. Panayiotis Kolios
Assistant Professor, Dept. Computer Science,
KIOS CoE for Intelligent Systems and Networks
Office: FST 01, 116
Telephone: +357 22893450 / 22892695
Web: https://www.kios.ucy.ac.cy/pkolios/

Πανεπιστήμιο Κύπρου

- **PROPHET** Framework developed by Facebook (Meta) for time series forecasting
  - Based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects
  - Works best with time series that have strong seasonal effects and several seasons of historical data
  - PROPHET is robust to missing data, shifts in the trend, and typically handles outliers well

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

- PROPHET model: trend, seasonality, and holidays
- PROPHET library in Python to perform time series forecasting
- Advanced options and configurations available within the Python library

```
# Imports
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
from prophet import Prophet
import matplotlib.pyplot as plt
import numpy as np
import statsmodels.api as sm
```

Model components

- Prophet models a time series $y(t)$ as a combination of three components:
  - Trend $g(t)$
  - Seasonality $s(t)$
  - Holidays $h(t)$
- The linear model is then:

$$y(t) = g(t) + s(t) + h(t) + \varepsilon_t$$

- where ε_t is the error, assumed to be normally distributed

Model components

- Trend $g(t)$ models non-periodic changes in the value of the time series
- Prophet provides two options for modeling the trend:
  - Piece-wise linear growth model
  - Logistic growth model
- Piece-wise linear growth model
  - Piece-wise linear function accommodates changes in the trend's direction
  - Accounts shifts in growth rates due to internal / external factors
  - Captures and forecast time series that do not follow a simple form (linear or logistic)
- In the piece-wise model, the series is divided into segments
  - In each segment, the trend is modeled as a linear function
  - The points where the trend changes direction are called **change points**
- Prophet allows users to specify the maximum number of potential change points or to let the algorithm automatically estimate it
- Prophet automatically detects the change points, allowing the trend to adjust its slope at these points, hence capturing shifts in the trend's direction

Model components

- Users can adjust the flexibility of the model in capturing trend changes by tuning the change point prior scale parameter
- A higher prior scale makes the model more sensitive to changes (allowing more flexibility)
- A lower prior scale makes the model less sensitive to fluctuations (resulting in a smoother trend)
- The piece-wise linear trend model in Prophet is defined as:

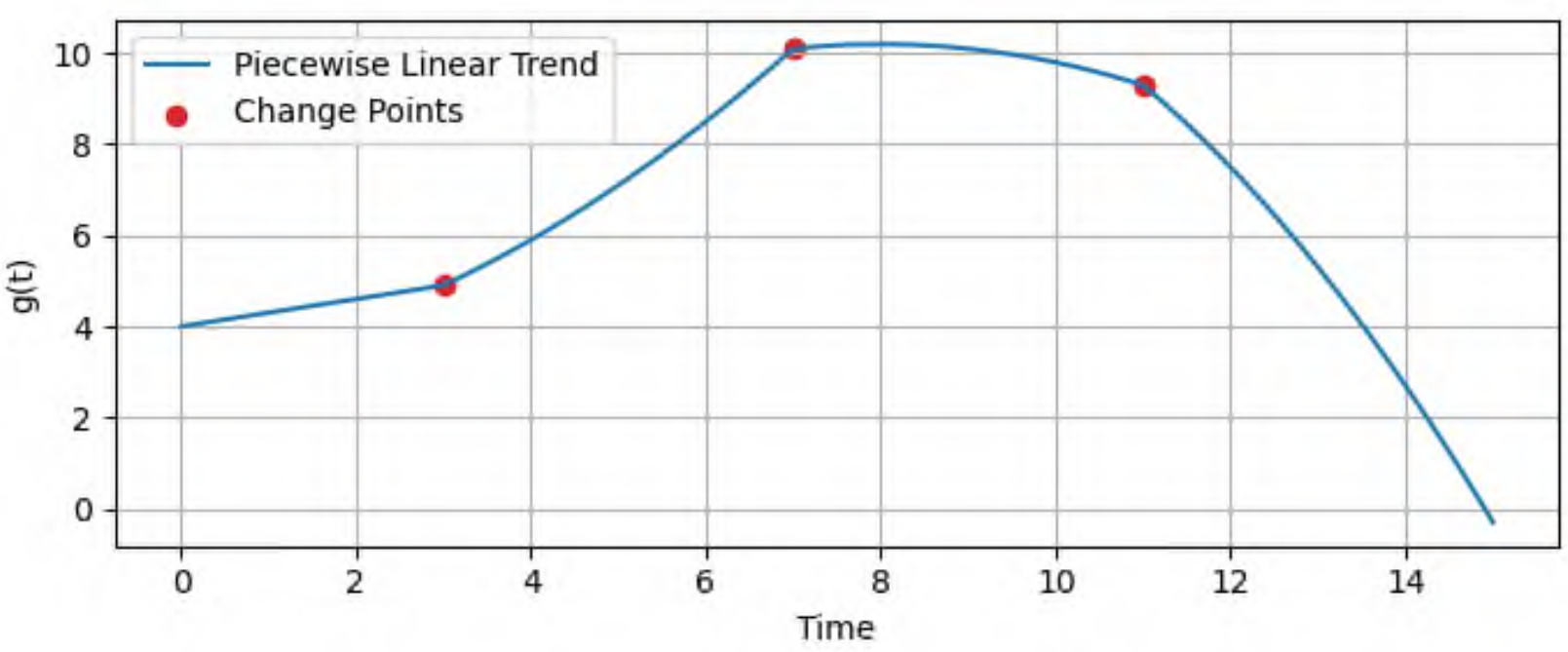
$$g(t) = (k + a(t)^T \delta) \cdot t + (g_0 + a(t)^T \gamma)$$

- where
  - k is the initial growth rate
  - $a(t)$ is a vector with each element is:
    - the amount of time since the corresponding change point, if t is after a change point;
    - 0 otherwise
- δ represents adjustments to the growth rate at each change point
- g_0 is the offset (intercept), i.e., the value at $t = 0$
- γ compensates for discontinuities in the trend at each change point, ensuring the trend is continuous

Model components

```
# Imports
change_points = [3, 7, 11]              # Times at which changes occur
k = 0.3                                 # Initial growth rate
delta = np.array([0.1, -0.2, -0.1])     # Adjustments to growth rate
g0 = 4.0                                # Initial offset
gamma = np.array([0.3, -0.1, 0.1])      # Compensations for discontinuities

def compute_a(t, change_points):
    return np.array([max(0, t - cp) for cp in change_points])

def compute_g(t, k, delta, g0, gamma, change_points):
    a_t = compute_a(t, change_points)
    trend = (k + np.dot(a_t, delta)) * t + (g0 + np.dot(a_t, gamma))
    return trend

time_points = np.linspace(0, 15, 200)
g_values = [compute_g(t, k, delta, g0, gamma, change_points) for t in
time_points]

plt.figure(figsize=(7, 3))
plt.plot(time_points, g_values, label="Piecewise Linear Trend")
plt.scatter(change_points, [compute_g(cp, k, delta, g0, gamma, change_points) for
cp in change_points], color='tab:red', label="Change Points")
plt.xlabel("Time")
plt.ylabel("g(t)")
plt.legend()
plt.grid(True)
plt.tight_layout();
```

Model components



Model components

- Trend $g(t)$ models non-periodic changes in the value of the time series
- Prophet provides two options for modeling the trend:
  - Piece-wise linear growth model
  - Logistic growth model
- Logistic growth model
  - Describes a population that:
    - Grows rapidly when it's small
    - Grows slowly as it approaches a maximum limit (carrying capacity) that it cannot exceed
    - Eventually levels off when the carrying capacity is reached (saturating growth)
  - Carrying capacity
    - Refers to the maximum population size that can be sustained
    - Could represent a maximum number of users a platform can support
  - Saturating growth
    - It refers to a growth pattern where increments become progressively smaller as the value approaches the carrying capacity
    - When the values are far below the carrying capacity, growth can be rapid (there is a lot of "room" to grow)
    - As the values approach the carrying capacity, the growth rate decreases, and the time series levels off, reflecting a saturation point where further growth becomes increasingly difficult

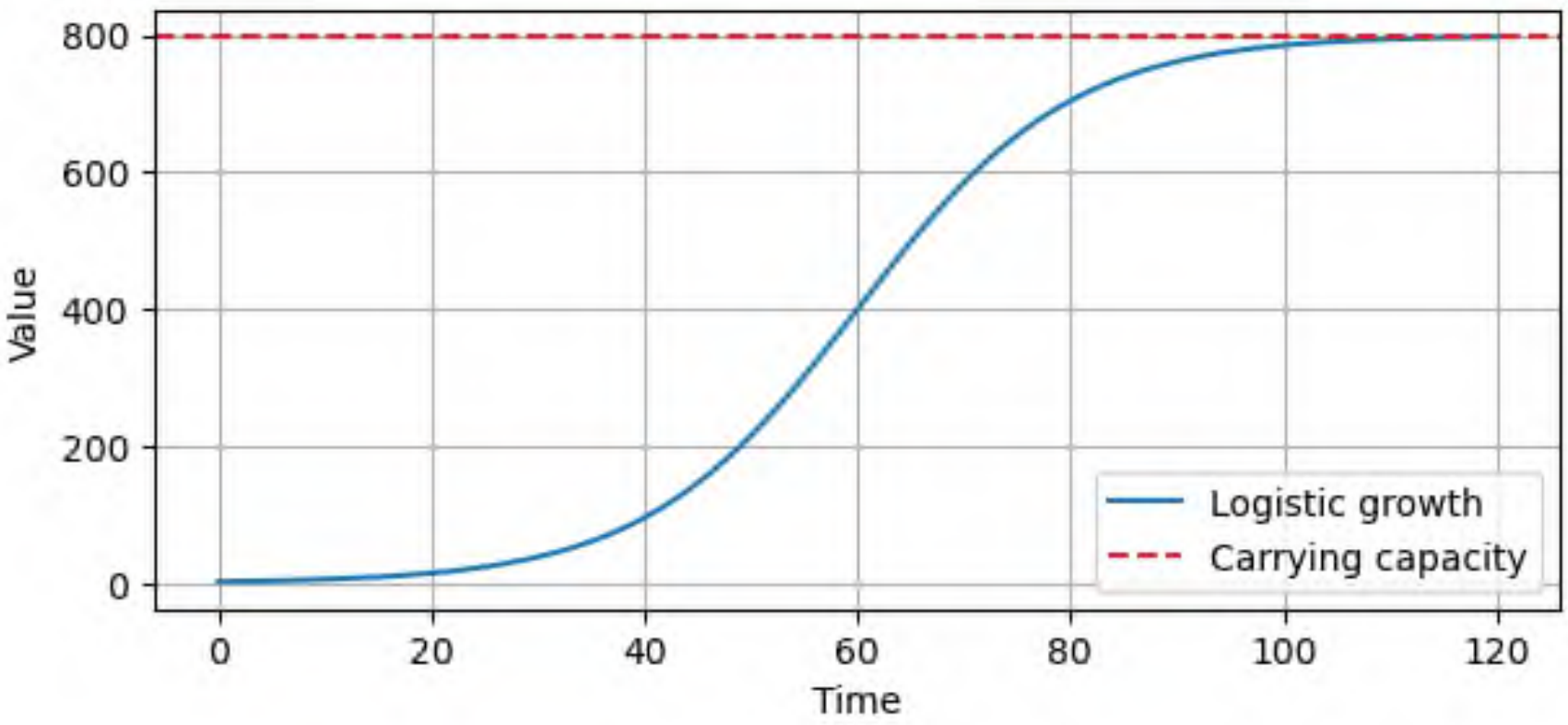
- Logistic growth model:

$$g(t) = \frac{C}{1 + e^{-k(t-m)}}$$

- where:
  - k is the growth rate, t is time
  - C is the carrying capacity (maximum achievable value)
  - m is the point in time where growth is halfway to the carrying capacity

```
def logistic_growth(t, C, k, m):
    return C / (1 + np.exp(-k * (t - m)))
# Parameters
C = 800   # carrying capacity
k = 0.1   # growth rate
m = 60    # offset parameter, indicating the inflection point
t = np.linspace(0, 120, 200)
y = logistic_growth(t, C, k, m)
plt.figure(figsize=(7, 3))
plt.plot(t, y, label='Logistic growth')
plt.xlabel('Time')
plt.ylabel('Value')
plt.axhline(C, color='tab:red', linestyle='--', label='Carrying capacity')
plt.legend()
plt.grid(True)
plt.show()
```

- Logistic growth model



Model components

- Seasonality component $s(t)$
- The seasonality component $s(t)$ models periodic changes, which can be yearly, weekly, or daily
- Prophet uses Fourier series to model these periodic changes, allowing for flexibility in capturing seasonality:

$$s(t) = \sum_{n=1}^{N} (a_n \cos(\frac{2\pi nt}{P}) + b_n \sin(\frac{2\pi nt}{P}))$$

- where
  - N is the number of Fourier terms (higher N captures more detailed seasonal patterns)
  - P is the period (e.g., 365.25 for yearly seasonality)
  - a_n and b_n are the Fourier series coefficients that are fitted to the data

Model components

- Holidays and events $h(t)$
- The holiday component $h(t)$ models irregular but predictable events
- Represented as a series of indicator functions that equal 1 if the time t corresponds to a holiday and 0 otherwise
- Coefficients associated with these indicators are fitted to measure the impact of holidays on the forecast

$$h(t) = \sum_i I(t \in D_i) \cdot \delta_i$$

- where
  - D_i represents the set of times corresponding to holiday i
  - I is the indicator function
  - δ_i is the effect of holiday i on the time series (tunable)

Seasonal and holiday effects adjustment

- Prophet can adjust for overfitting or underfitting of seasonal and holiday effects by changing the prior scale of these components

- A larger prior scale allows the model to fit larger seasonal fluctuations

- A smaller scale regularizes the model, preventing it from overfitting seasonal and holiday effects

Fitting the Model

- To **fit** the Prophet model to **historical data** the model's parameters are estimated using **maximum likelihood estimation** or **Bayesian sampling**

- This involves optimizing the parameters to minimize the difference between the observed and predicted values of the time series

- The optimization is usually done through gradient descent methods

- To perform the optimization Prophet relies on Stan, a C++ library for statistical modeling and high-performance statistical computation that makes model fitting very fast

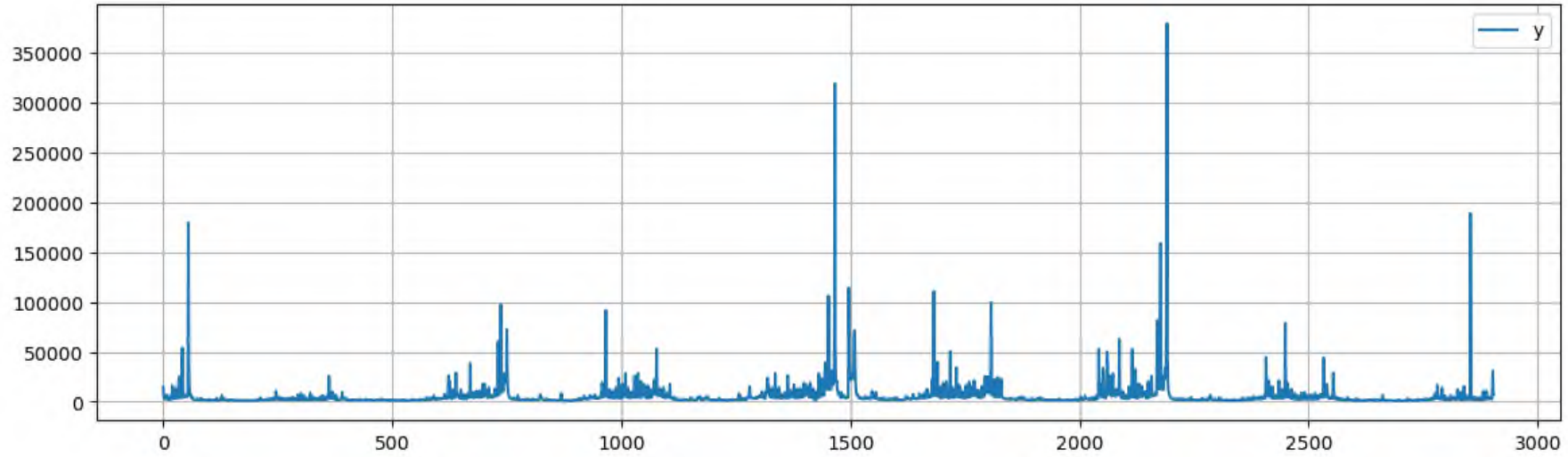
Model components

- The input to Prophet is always a dataframe with two columns: `ds` and `y`
- Be sure to rename your dataframe with these column names
- The `ds` (datestamp) column should be of a format expected by Pandas, ideally YYYY-MM-DD for a date or YYYY-MM-DD HH:MM:SS for a timestamp
- The `y` column must be numeric, and represents the measurement we wish to forecast
- As an example, we'll look at a time series of daily page views for the Wikipedia page of Peyton Manning, a former football player
    - This is a nice example because it illustrates some of Prophet's features, like multiple seasonality, changing growth rates, and the ability to model special days (such as Manning's playoff and superbowl appearances)

Model components

```
data_path =
'https://raw.githubusercontent.com/PinkWink/DataScience/master/data/07.%20example
_wp_peyton_manning.csv'
peyton = pd.read_csv(data_path)
peyton.head()
peyton.plot(figsize=(14, 4), grid=True);
```

	ds	y
0	2007-12-10	14629
1	2007-12-11	5012
2	2007-12-12	3582
3	2007-12-13	3205
4	2007-12-14	2680

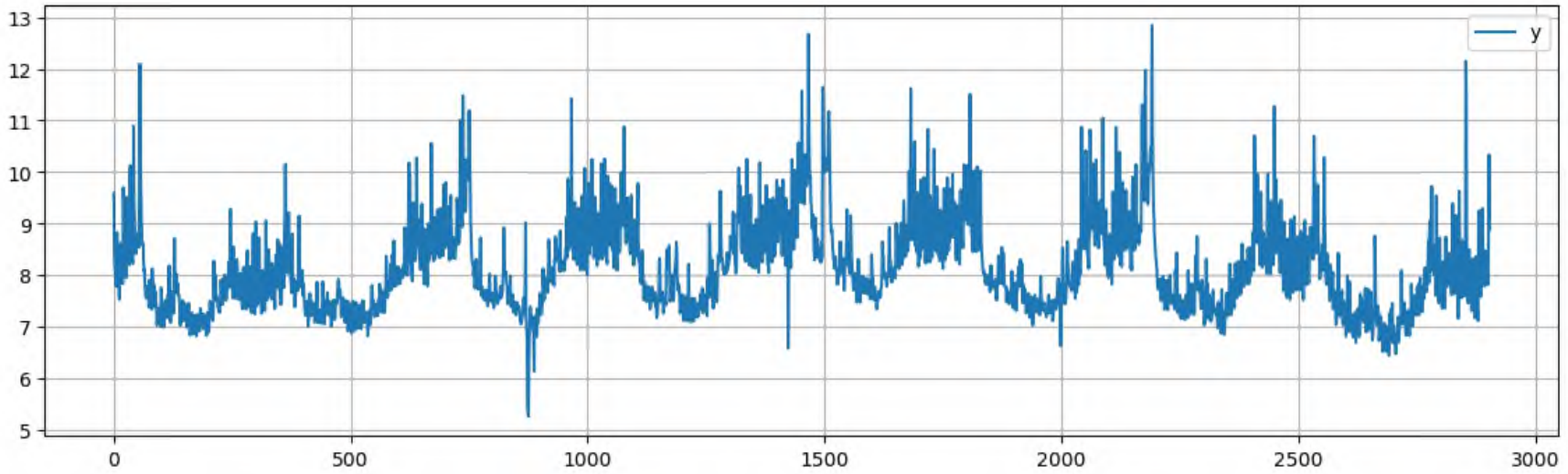


Model components

- Since there are big spikes in the data, we will apply a logarithm transformation to obtain a more even range of variation in the data

```
peyton['y'] = np.log(peyton['y'])
peyton.head()
peyton.plot(figsize=(14, 4), grid=True);
```

	ds	y
0	2007-12-10	9.590761
1	2007-12-11	8.519590
2	2007-12-12	8.183677
3	2007-12-13	8.072467
4	2007-12-14	7.893572



Model components

- Several parameters to specify:
  - growth='linear', meaning that we use a piece-wise linear function to model the trend or 'logistic' or 'flat'
  - seasonality_mode='additive' or 'multiplicative'
  - interval_width=0.90 specifies the width of the uncertainty intervals for the forecast

```
model = Prophet(growth='linear', seasonality_mode='additive', interval_width=0.90)
model.fit(peyton);
```

- Predictions made on a dataframe by specifying the ds date
- You can get a suitable dataframe that extends into the future a specified number of days using the helper method Prophet.make_future_dataframe
- periods=365 specifies the length of our forecast
- freq='D' specifies that the units (365 in this case) represent days
- Setting include_history=True will include the dates used for training, so we can also see the model

Model components

```
future = model.make_future_dataframe(periods=365, freq='D', include_history=True)
future.tail()
```

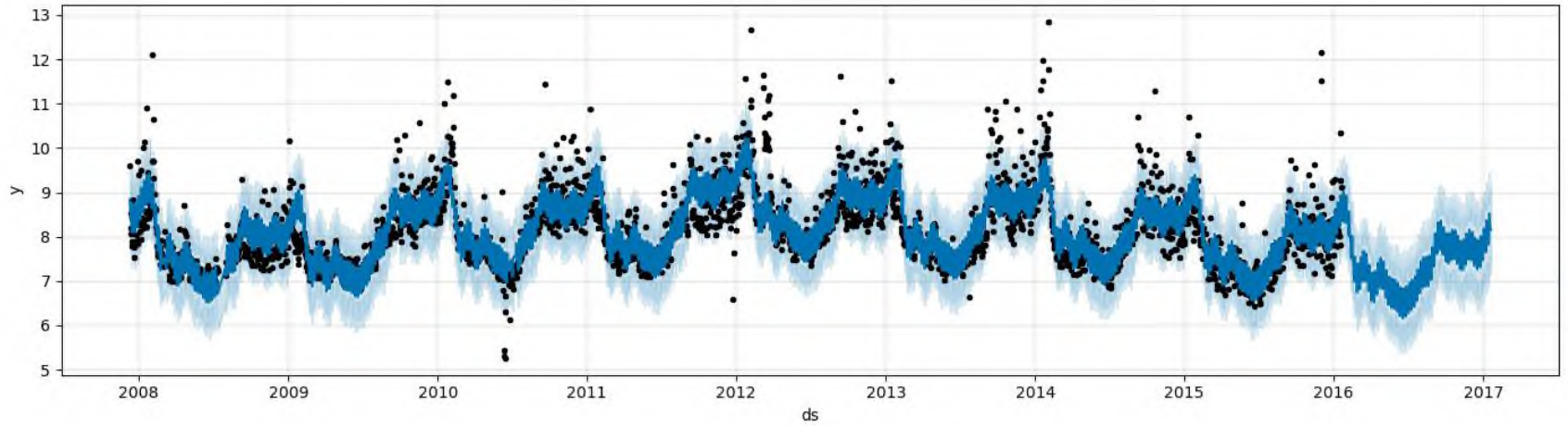
	ds
3265	2017-01-15
3266	2017-01-16
3267	2017-01-17
3268	2017-01-18
3269	2017-01-19

```
forecast = model.predict(future)
forecast.tail()
```

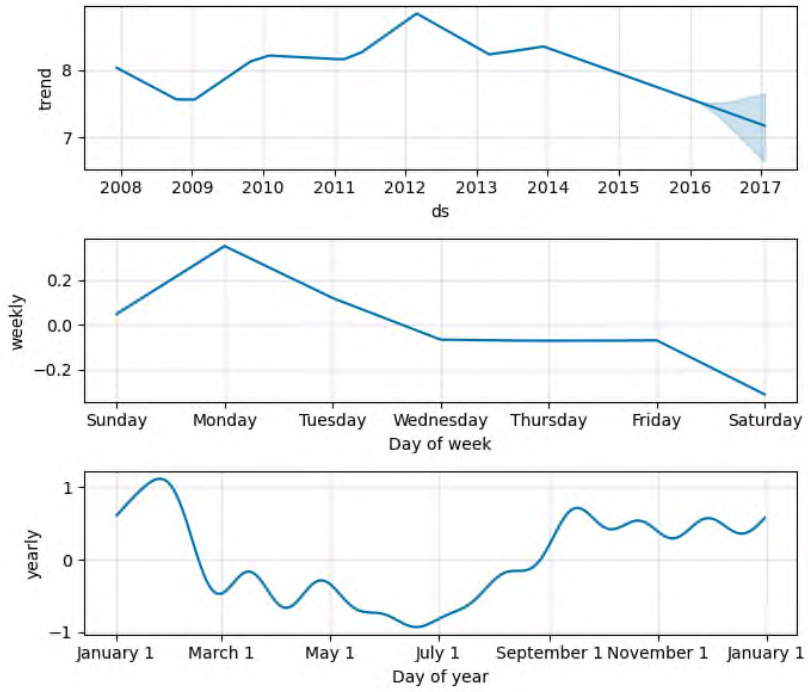
	ds	$g(t)$	$\hat{y}_l$	$\hat{y}_u$	$s(t)_l$	$s(t)_u$	additive_terms	additive_terms_lower	additive_terms_upper	$s(t)_w$	$s(t)_{wl}$	$s(t)_{wu}$	$s(t)_y$	$s(t)_{yl}$	$s(t)_{yu}$	multiplicative_terms	multiplicative_terms_lower	multiplicative_terms_upper	$\hat{y}$
3265	2017-01-15	7.183818	7.163135	9.052371	6.656108	7.658024	1.018136	1.018136	1.018136	0.048285	0.048285	0.048285	0.969851	0.969851	0.969851	0.0	0.0	0.0	8.201955
3266	2017-01-16	7.182785	7.542376	9.457438	6.654270	7.658730	1.344165	1.344165	1.344165	0.352294	0.352294	0.352294	0.991871	0.991871	0.991871	0.0	0.0	0.0	8.526949
3267	2017-01-17	7.181751	7.365621	9.208123	6.651965	7.659161	1.132587	1.132587	1.132587	0.119640	0.119640	0.119640	1.012947	1.012947	1.012947	0.0	0.0	0.0	8.314338
3268	2017-01-18	7.180717	7.150724	9.107429	6.648953	7.659711	0.966217	0.966217	0.966217	-0.066658	-0.066658	-0.066658	1.032875	1.032875	1.032875	0.0	0.0	0.0	8.146934
3269	2017-01-19	7.179683	7.231810	9.073292	6.645942	7.660261	0.979145	0.979145	0.979145	-0.072266	-0.072266	-0.072266	1.051411	1.051411	1.051411	0.0	0.0	0.0	8.158829

Model components

```
fig1 = model.plot(forecast, figsize=(14, 4))
```




```
fig2 = model.plot_components(forecast, figsize=(7, 6))
```



Πανεπιστήμιο Κύπρου

- For *holidays* or other recurring events to model, must create a dataframe for them
- Two columns (*holiday* and ds), a row for each occurrence of a holiday
- It must include all occurrences of a holiday, both in the past (back as far as the historical data go) and in the future (out as far as the forecast is being made)
- If they won't repeat in the future, Prophet will model them and then not include them in the forecast
- Durations of a holiday used with fields [lower_window, upper_window] days around the date
- For instance, if you wanted to include Christmas Eve in addition to Christmas you'd include 'lower_window': -1 and 'upper_window': 0

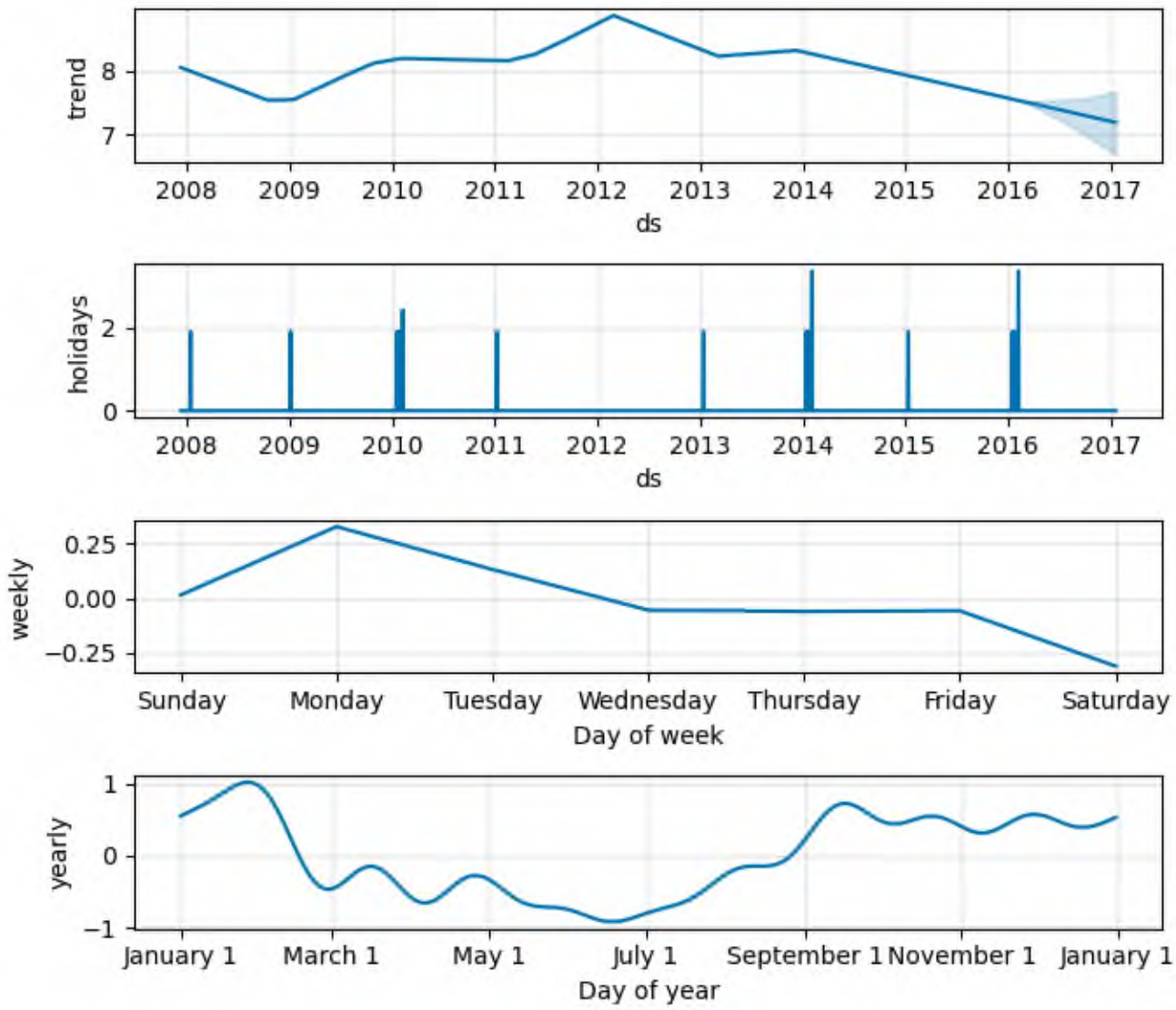
Model components

```
# add holidays
playoffs = pd.DataFrame({
  'holiday': 'playoff',
  'ds': pd.to_datetime(['2008-01-13', '2009-01-03', '2010-01-16',
                        '2010-01-24', '2010-02-07', '2011-01-08',
                        '2013-01-12', '2014-01-12', '2014-01-19',
                        '2014-02-02', '2015-01-11', '2016-01-17',
                        '2016-01-24', '2016-02-07']),
  'lower_window': 0, # this specifies spillover into previous days
  'upper_window': 1, # this for the future days
})

superbowls = pd.DataFrame({
  'holiday': 'superbowl',
  'ds': pd.to_datetime(['2010-02-07', '2014-02-02', '2016-02-07']),
  'lower_window': 0,
  'upper_window': 1,
})

holidays_df = pd.concat((playoffs, superbowls))
```

```
# fit and predict
model_holyday = Prophet(holidays=holidays_df, # this includes the holidays
                        growth='linear', seasonality_mode='additive',
interval_width=0.90) # these are unchanged
forecast = model_holyday.fit(peyton).predict(future)
```

Model components

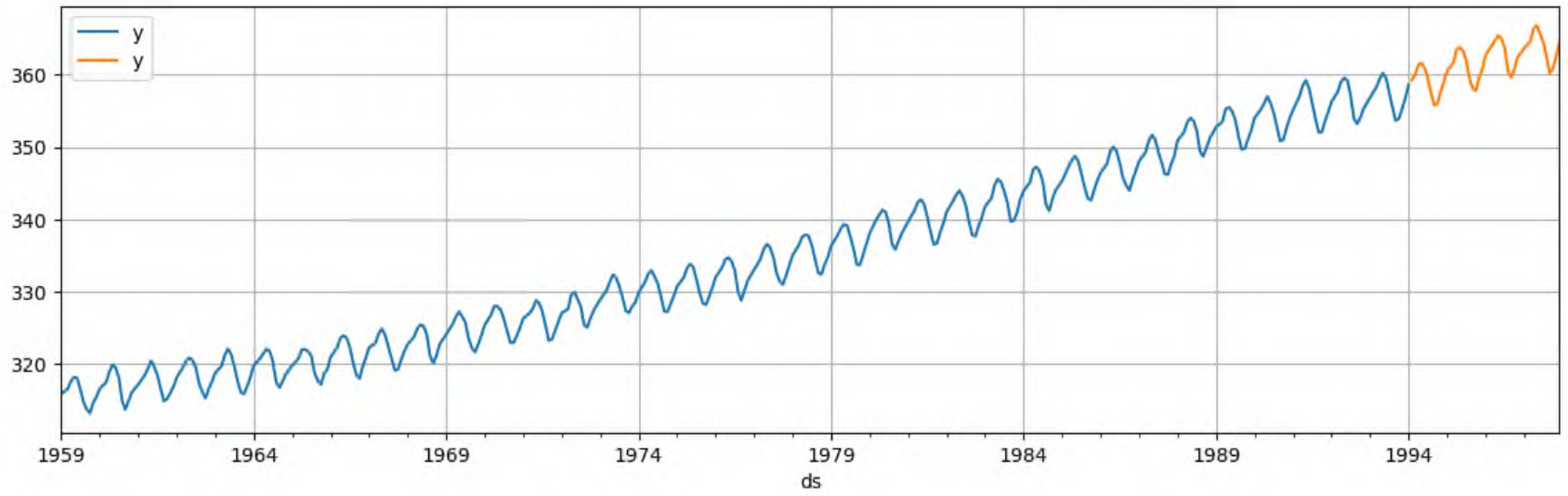
```
# check visually the impact of the holidays
fig2 = model_holyday.plot_components(forecast, figsize=(7, 6))
```



- Prophet allows you to make forecasts using a logistic growth trend model as well
- Will give an example with Datasets Package from statsmodels
- First lets do a bit of data preproccesing before using the data

```
# get data from statsmodels
co2 = sm.datasets.get_rdataset("co2", "datasets").data

# Convert decimal year to pandas datetime
def convert_decimal_year_to_datetime(decimal_years):
    dates = [(pd.to_datetime(f'{int(year)}-01-01') + pd.to_timedelta((year -
int(year)) * 365.25, unit='D')).date()
             for year in decimal_years]
    return dates

# Convert to Prophet format
co2['time'] = convert_decimal_year_to_datetime(co2['time'])
co2.rename(columns={'time': 'ds', 'value': 'y'}, inplace=True)

# Convert the column ds to datetime
co2['ds'] = pd.to_datetime(co2['ds'])
print("\nConverted:\n------------------\n", co2.head())
```

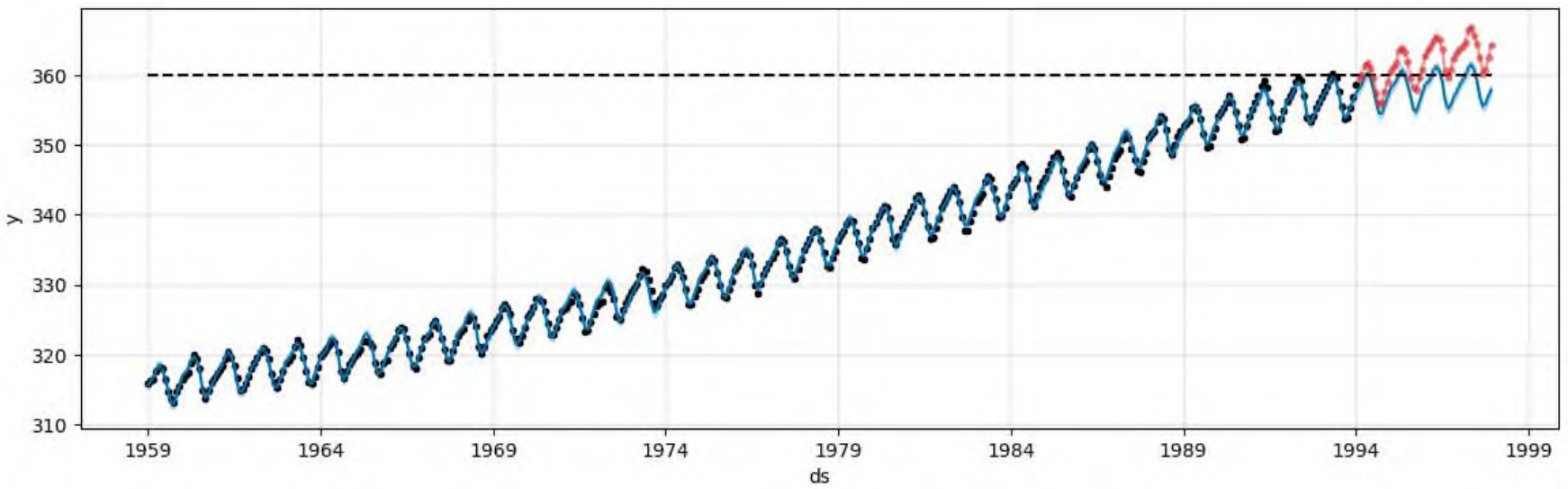
```
# Resample to monthly frequency based on the ds column
co2 = co2.resample('MS', on='ds').mean().reset_index()

# Replace NaN with the mean of the previous and next value
co2['y'] = co2['y'].interpolate()
print("\nResampled:\n------------------\n", co2.head())print("\nConverted:\n-----
-------------\n", co2.head())

# Note that we need to make a copy here since we will modify the data
# Split data to training (first 90%) and test data
train = co2.iloc[:int(co2.shape[0] * 0.9)].copy()
test = co2.iloc[int(co2.shape[0] * 0.9):].copy()

fig, ax = plt.subplots(figsize=(10, 4))
train.plot(x='ds', y='y', figsize=(14, 4), grid=True, ax=ax)
test.plot(x='ds', y='y', figsize=(14, 4), grid=True, ax=ax)
plt.show()
```



- The logistic model requires to specify a carrying capacity, i.e., the maximum achievable point (total population size)
- The carrying capacity is specified in a column cap
- Normally be set based on expertise and knowledge

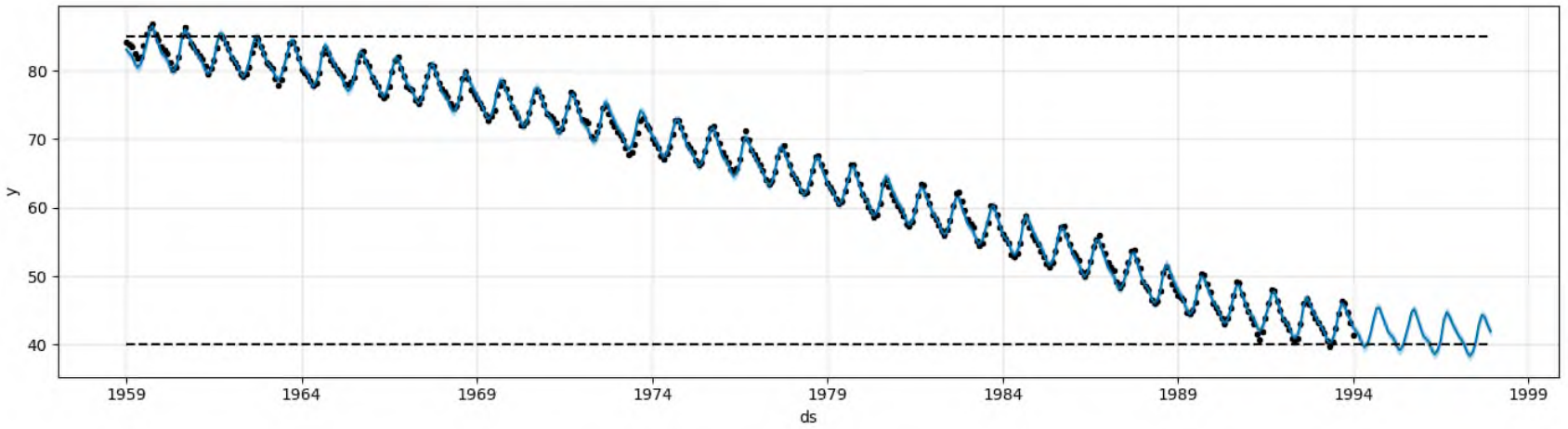
```
# Set training cap
train['cap'] = 360

# Train the model with logistic
model_logist = Prophet(growth='logistic', # this has changed
                       seasonality_mode='additive', interval_width=0.90)
model_logist.fit(train);
```

```
# Make a prediction with the same cap size
future = future = pd.DataFrame({'ds': pd.concat([train['ds'], test['ds']])}) #
Init df for predictions
future['cap'] = 360                                                          #
Set the cap
fcst = model_logist.predict(future)                                          #
Compute forecasts
fig, ax = plt.subplots(figsize=(14, 4))
fig = model_logist.plot(fcst, ax=ax)
ax.plot(test['ds'], test['y'], 'tab:red', marker='o', markersize=3, alpha=0.5);
```



- The model tries to keep the predictions under the specified cap value
- The value is clearly too low in this case. Try with cap=380 to get better predictions
- The logistic growth model can also handle a saturating minimum
- This is specified with a column floor in the same way as the cap column specifies the maximum
- To use a logistic growth trend with a saturating minimum, a maximum capacity must also be specified

```
train['y'] = 400 - train['y'] # Modify the data so that the time series decreases
over time
train['cap'] = 85
train['floor'] = 40
future['cap'] = 85
future['floor'] = 40
m = Prophet(growth='logistic')
m.fit(train)
fcst = m.predict(future)
fig = m.plot(fcst, figsize=(14, 4))
```



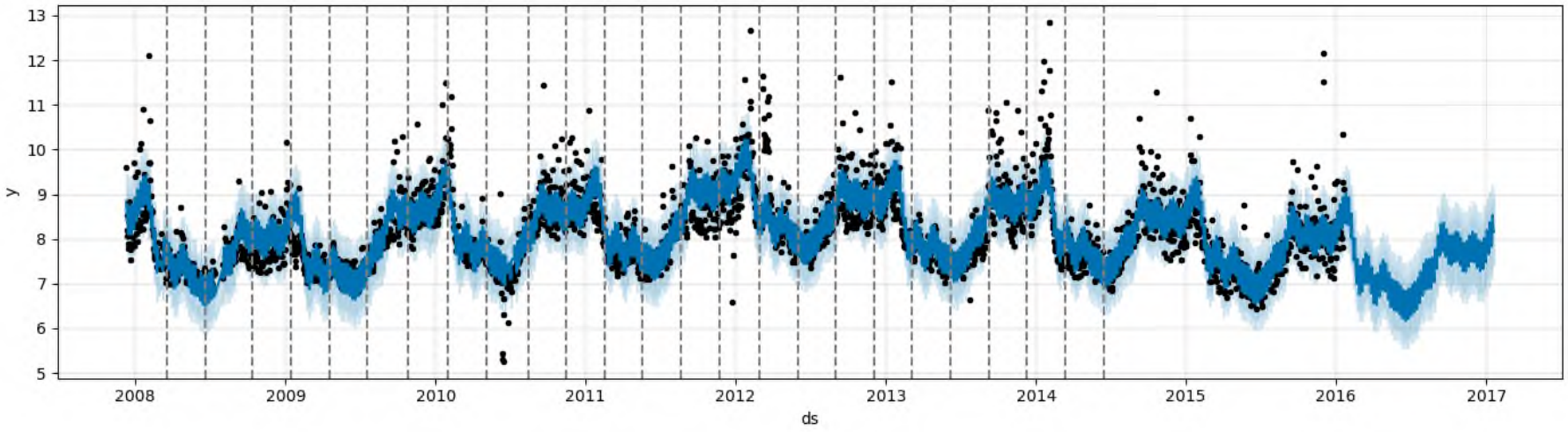
- Real time-series frequently have abrupt changes in their trajectories
- Prophet automatically detects changepoints and allows the trend to adapt appropriately
- Finer control over this process (e.g., Prophet missed a rate change, or is overfitting rate changes in the history), with several input arguments

- Prophet detects changepoints by
  - first specifying a large number of potential changepoints at which the rate is allowed to change
  - Then L1 regularization that encourages sparsity, to use as few of them as possible

- Previous example considered

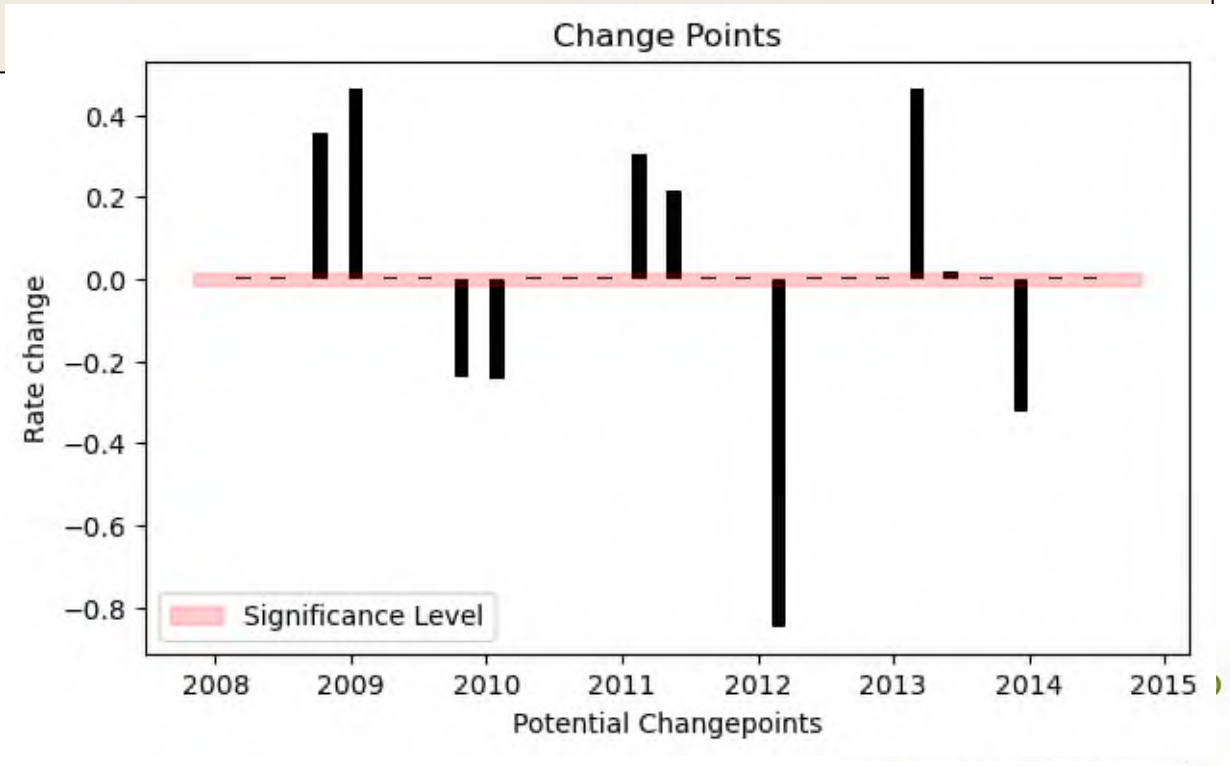
```
# Data already log-transformed
df =
pd.read_csv('https://raw.githubusercontent.com/facebook/prophet/main/examples/exa
mple_wp_log_peyton_manning.csv')
m = Prophet()
m.fit(df)
future = m.make_future_dataframe(periods=365)
forecast = m.predict(future)
```

```
# 25 potential checkpoints set uniformly across 80% of time series
from prophet.plot import add_changepoints_to_plot
fig = m.plot(forecast, figsize=(14, 4))
a = add_changepoints_to_plot(fig.gca(), m, forecast, threshold=0.0,
cp_color='gray', trend=False)
```

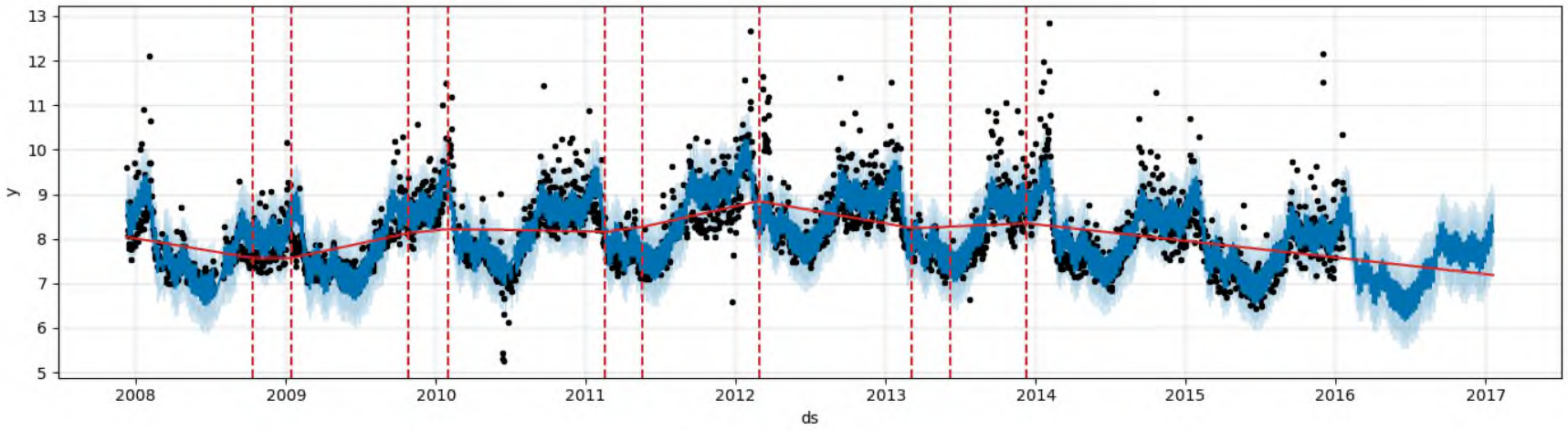


- Of the 25 used only significant checkpoints are kept based on rate of change

```
fig, ax = plt.subplots(figsize=(7, 4))
eps = 5e-3  # small offset to improve visualization
ax.bar(m.changepoints, np.nanmean(m.params['delta'], axis=0)+eps, color='k',
width=40)
ax.fill_between(ax.get_xlim(), 0.01+eps, -0.01-eps, color='r', alpha=0.2,
label='Significance Level')
ax.set_title('Change Points')
ax.set_ylabel('Rate change')
ax.set_xlabel('Potential Changepoints')
plt.legend()
plt.show()
```



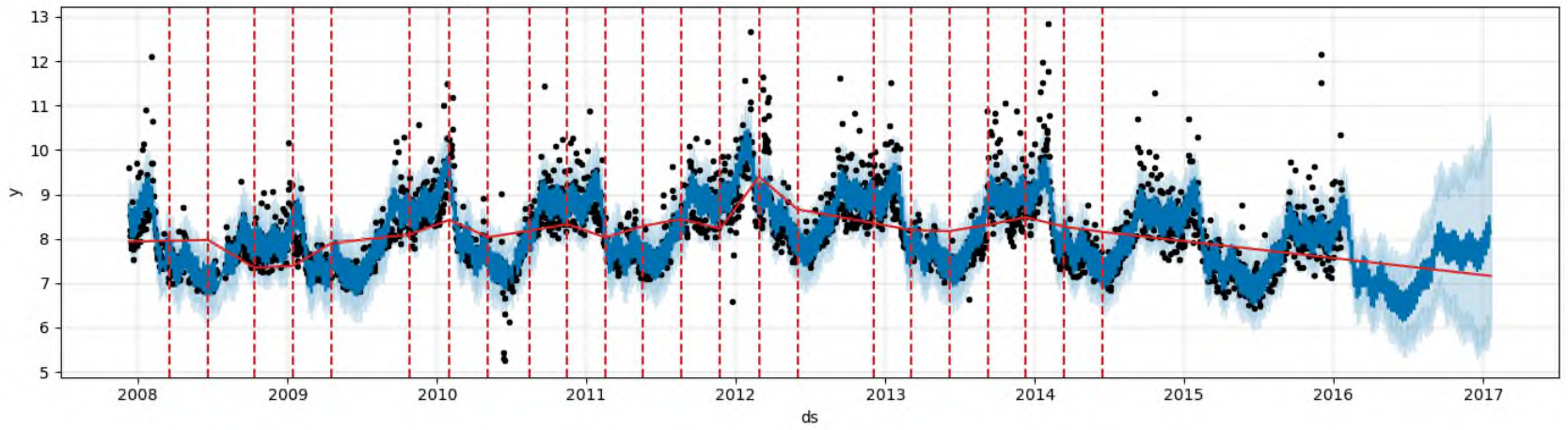
```
fig = m.plot(forecast, figsize=(14, 4))
a = add_changepoints_to_plot(fig.gca(), m, forecast, threshold=0.01,
cp_color='tab:red', trend=True)
```



- The default 80% use for checkpoints can be changed using the *changepoint_range* argument
- e.g. m = Prophet(changepoint_range=0.9) will place potential changepoints in the first 90% of the time series

- Trend changes can be overfit (too much flexibility) or underfit (not enough flexibility)
- Can adjust the strength of the sparsity prior using the input argument *changepoint_prior_scale*
- Default parameter 0.05
- Increasing it will make the trend more flexible

```
m = Prophet(changepoint_prior_scale=0.5) # Increase prior
forecast = m.fit(df).predict(future)
fig = m.plot(forecast, figsize=(14, 4))
a = add_changepoints_to_plot(fig.gca(), m, forecast, threshold=0.01,
cp_color='tab:red', trend=True)
```



- locations of potential changepoints manually set with the *changepoints* argument
- Slope changes will then be allowed only at these points, with the same sparse regularization as before
  - One could, for instance, create automatically a grid of points
  - Add specific dates that likely have changes
  - Alternatively, limit to a small set of dates

```
m = Prophet(changepoints=['2014-01-01'])
forecast = m.fit(df).predict(future)
fig = m.plot(forecast, figsize=(14, 4))
a = add_changepoints_to_plot(fig.gca(), m, forecast, threshold=0.01,
cp_color='tab:red', trend=True)
```