Data Summary

This dataset for the scope of this project consists of daily us-covid data gathered throughout the last 3 years, download from the site - https://raw.githubusercontent.com/nytimes/covid-19-data/master/us.csv

Data not only contains missing points(days for which data points are non-existent) but also by its nature is to prone to annomalies, in that case:

- outliers
- shifts in trend and/or seasonality, and variability of points, measured by the standard deviation

In [1]:
```
import itertools
import pandas as pd
import numpy as np
from random import gauss

import statsmodels.api as sm
from statsmodels.tsa.stattools import adfuller

from pandas.plotting import autocorrelation_plot
from sklearn.metrics import mean_squared_error
from sklearn.feature_selection import RFECV
from sklearn.linear_model import Ridge

import warnings
import itertools
from pandas.plotting import autocorrelation_plot
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

from dateutil.easter import easter
from fbprophet import Prophet
from fbprophet.diagnostics import cross_validation, performance_metrics
from fbprophet.plot import plot_yearly, add_changepoints_to_plot

import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')

warnings.simplefilter(action='ignore', category= FutureWarning)

import plotly.graph_objs as go
```

In [2]:
```
# general settings
class CFG:
    data_folder = 'tsdata-1'
    img_dim1 = 15
    img_dim2 = 10

# adjust the parameters for displayed figures
plt.rcParams.update({'figure.figsize': (CFG.img_dim1,CFG.img_dim2)})
```

Prophet

While useful in practice, exponential smoothing can only handle one seasonal pattern at at time. Challenges like these led to the development of Prophet, which is a time series framework designed to work out of the box and developed by Core Data Science team at FB: https://research.facebook.com/research-areas/data-science/. The core idea is based around the structural decomposition:

$$X_t = T_t + S_t + H_t + \epsilon_t \tag{1}$$

where

- T_t : trend component
- S_t : seasonal component (weekly, yearly)
- H_t : deterministic irregular component (holidays)
- ϵ_t: noise

For the more scientifically among the readers, the original paper describing the approach can be found here: https://peerj.com/preprints/3190/.

We will discuss the Prophet style of modeling trend, seasonality and holidays in the sections below, but first a crash intro to the theory behind the models.

Generalized Additive Models (GAM)

The core mathematical idea behind Prophet is the Kolmogorov-Arnold representation theorem, which states that multivariate function could be represented as sums and compositions of univariate functions:

$$f(x_1, \ldots, x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right) \tag{2}$$

The theorem has no constructive proof suitable for modeling $\Longrightarrow$ simplification is necessary:

$$f(x_1, \ldots, x_n) = \Phi \left(\sum_{p=1}^{n} \phi_p(x_p) \right) \tag{3}$$

where Φ is a smooth monotonic function. This equation gives a general representation of GAM models and a familiar variant of this approach is the class of Generalized Linear Models :

$$\Phi^{-1} \left[\mathrm{E}(Y) \right] = \beta_0 + f_1(x_1) + f_2(x_2) + \cdots + f_m(x_m). \tag{4}$$

The smooth functions in the context of Prophet are the trend, seasonal and holiday components - we can isolate each individual function and evaluate its effect in prediction, which makes such models easier to interpret. we estimate through backfitting algorithm $\rightarrow$ convergence

The prophetic core

So how does that work in practice? We take a GAM-style decomposition as our starting point:

$$X_t = T(t) + S(t) + H(t) + \epsilon_t \tag{5}$$

Unpacking the equation:

- time is the only regressor
- easy accomodation of new components
- multiple seasonal patterns $\rightarrow$ extension of double exponential smoothing
- forecasting $\rightarrow$ curve fitting
- no need for regular spacing $\implies$ no NA filling
- fast fitting with backfitting algorithm https://en.wikipedia.org/wiki/Backfitting_algorithm
- probabilistic aspects - Hamiltonian Monte Carlo (which is why Windows have an extra step of Stan installation - but trust me, it's worth it)
- works in more general cases, but "designed" for daily data

Below we discuss each of the components in its own section.

Trend model

The Prophet library implements two possible trend models.

Linear Trend

The first, default trend model is a simple Piecewise Linear Model with a constant rate of growth. It is best suited for problems without saturating growth and takes advantage of the fact that a broad class of shapes can be approximated by a piecewise linear function.

$$T(t) = \left[k + a(t)^T \delta\right] t + \left[m + a(t)^T \gamma\right] \tag{6}$$

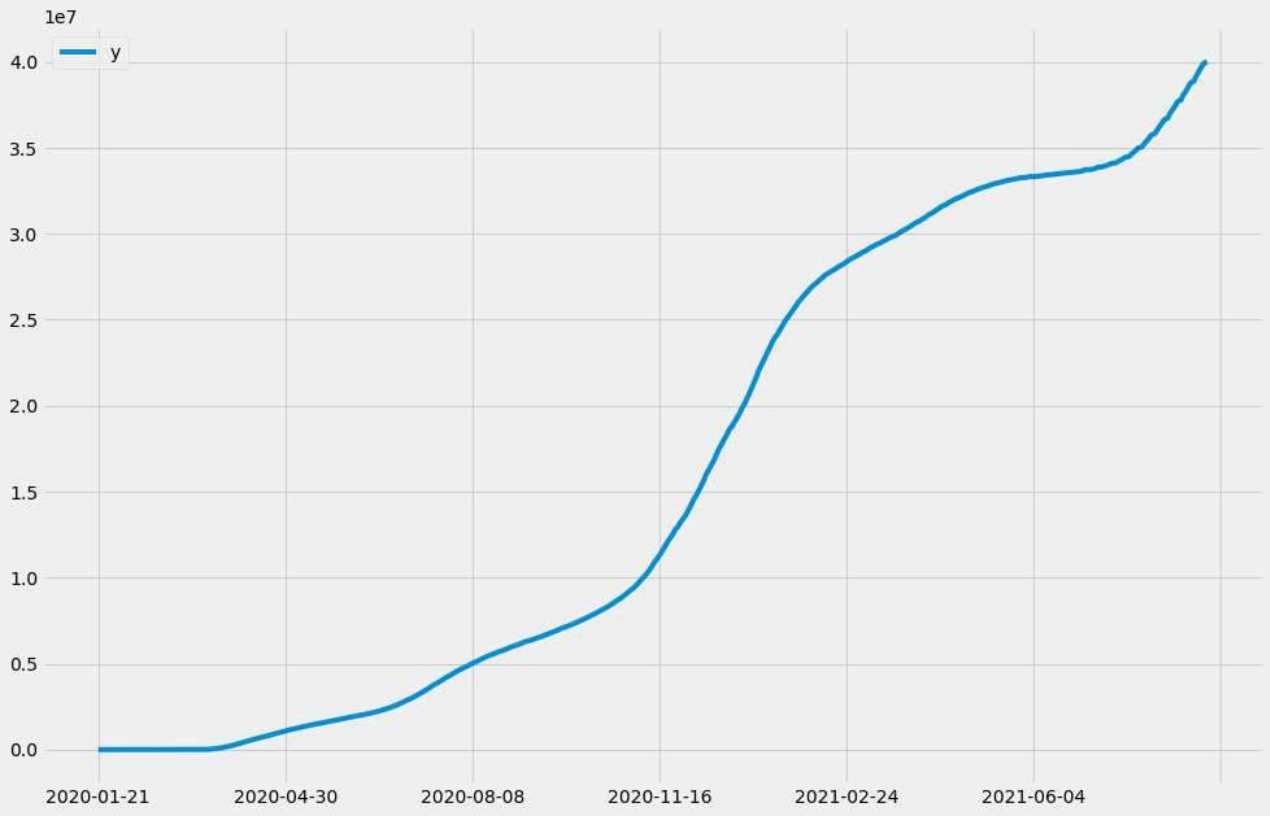
By default, Prophet estimates 25 changepoints in the model, over 80pct of the dataset (those parameters can be adjusted). Changepoints are defined as changes in the trajectory, which can be esimated or provided manually - the latter is a useful option if domain knowledge is available to the analyst.

We demonstrate this functionality by using New York Times data on Covid cases:
https://raw.githubusercontent.com/nytimes/covid-19-data/master/us.csv

In [3]:

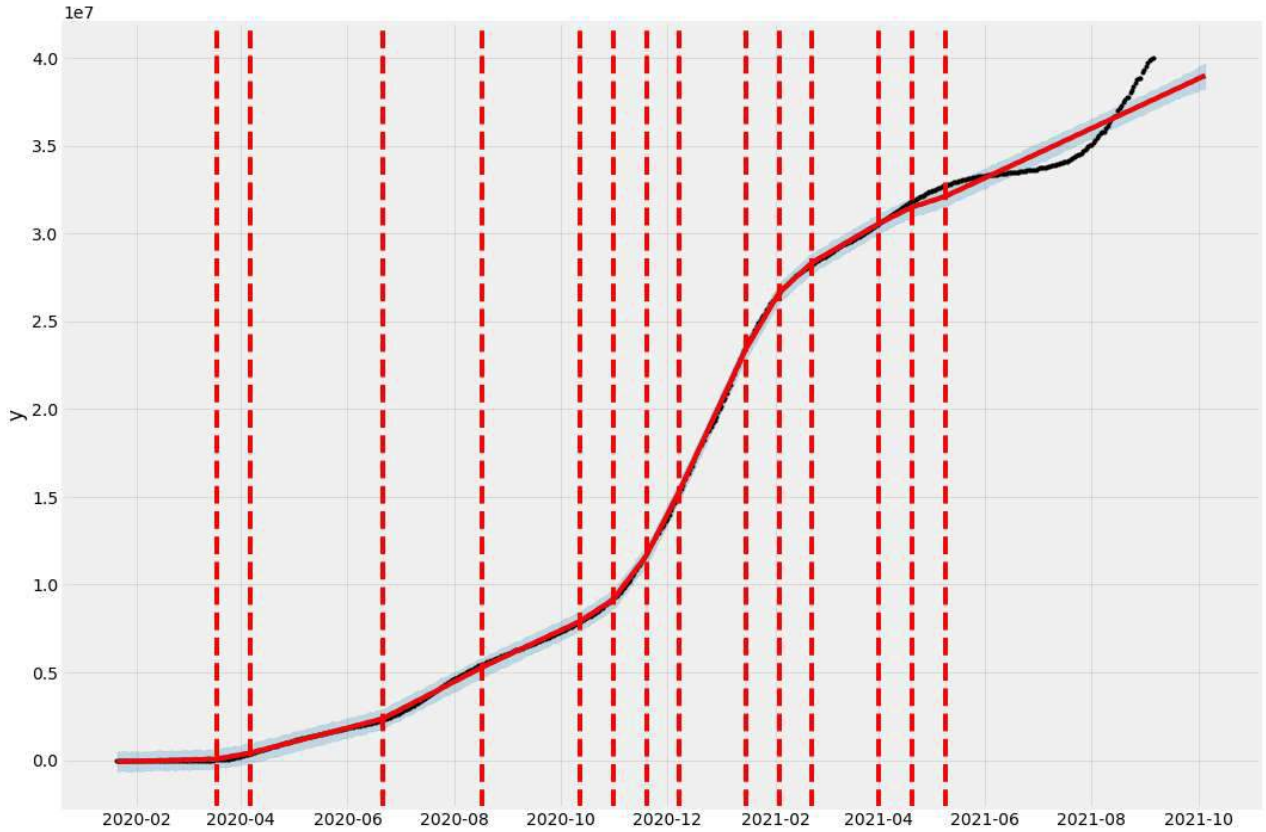
```
df = pd.read_csv(CFG.data_folder + 'us_covid.csv')
df.head(10)


# Prophet does have a quirk: a hardcoded format for the input time series - it must con
xdat = df[['date', 'cases']].rename(columns={"date": "ds", "cases": "y"})
xdat.set_index('ds').plot(figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '')
```

Out[3]: <AxesSubplot:>



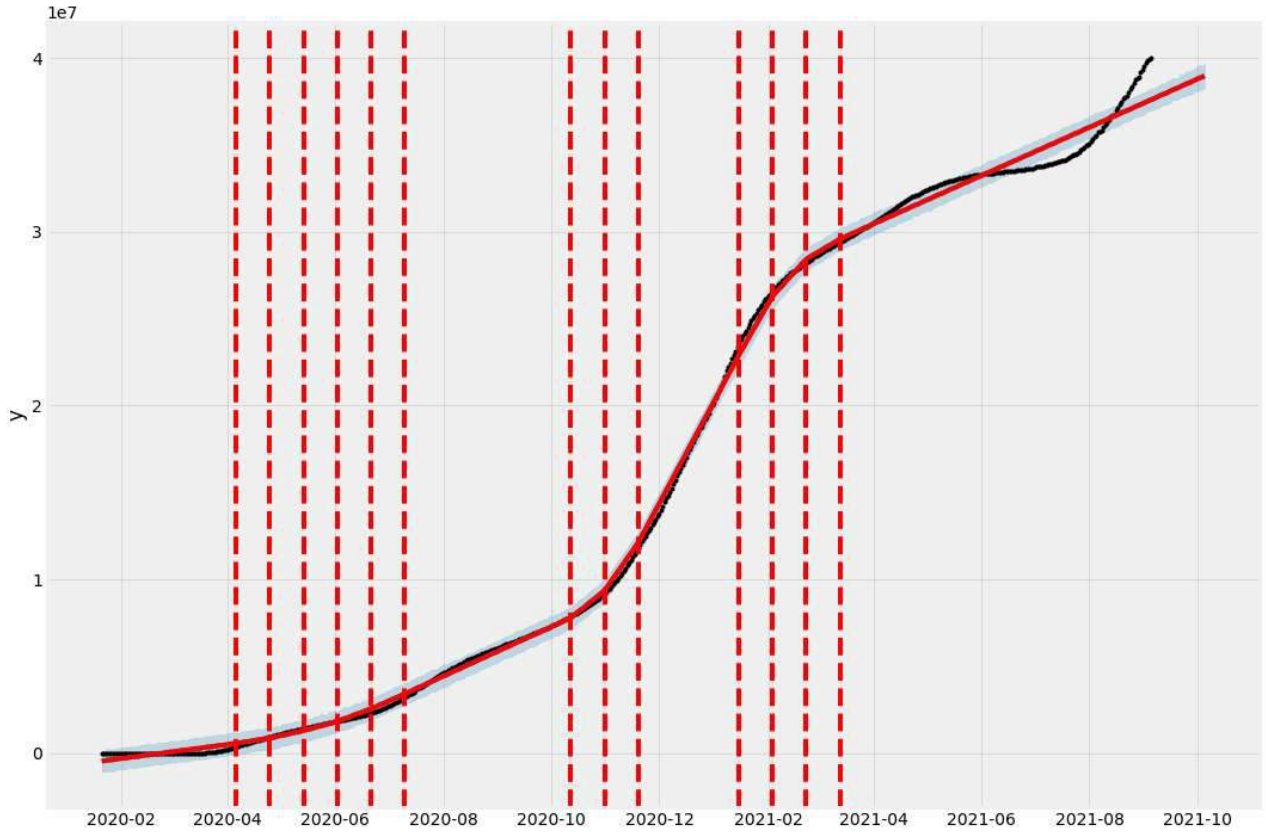
In [4]:

```
# automatic detection of changepoints
m = Prophet()
m.fit(xdat)
future = m.make_future_dataframe(periods= 30)
forecast = m.predict(future)
fig = m.plot(forecast, figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '')
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```



It seems like the default settings are on the generous side when it comes to assigning changepoints - we can control this behavior by enforcing more regularization. This is achieved by shrinking the `changepoint_prior_scale` parameter:

In [5]:
```
m = Prophet(changepoint_prior_scale = 0.01)
m.fit(xdat)
future = m.make_future_dataframe(periods= 30)
forecast = m.predict(future)
fig = m.plot(forecast, figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '')
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```



Nonlinear growth

The first one is called Nonlinear, Saturating Growth. It is represented in the form of the logistic growth model:

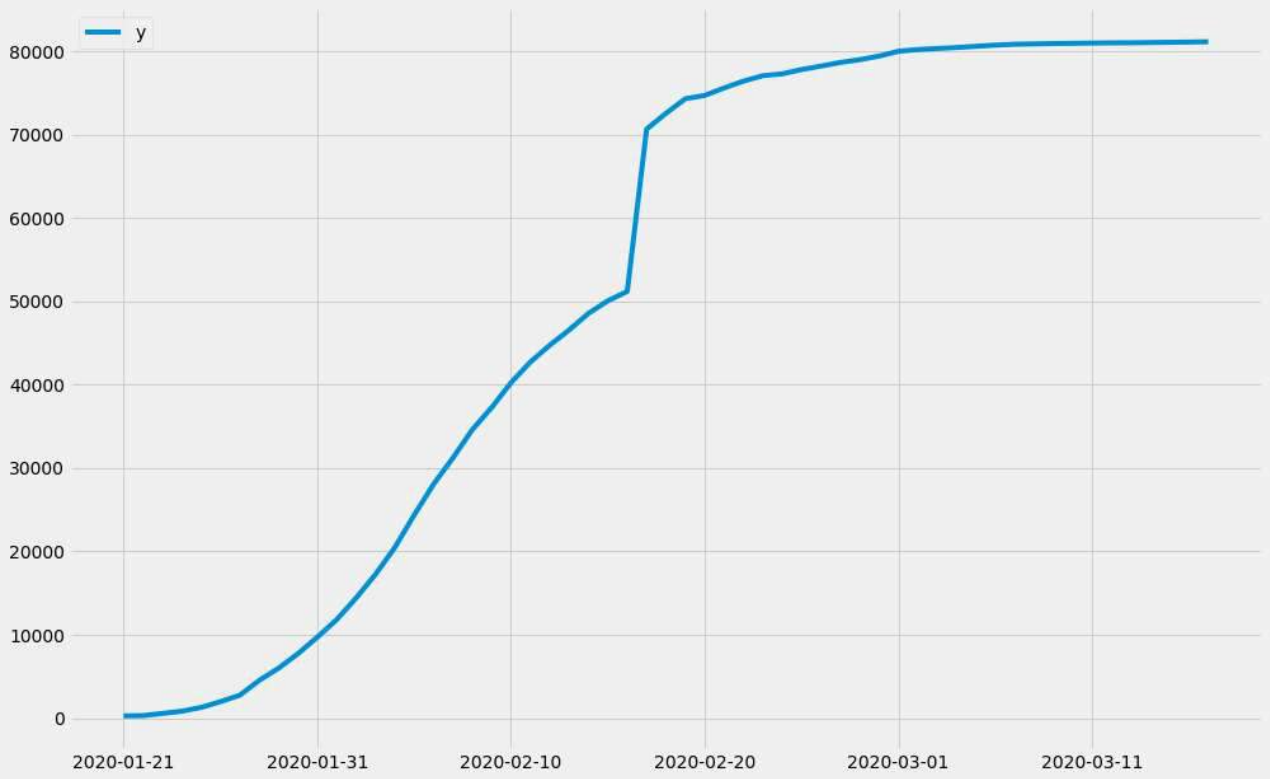
$$T(t) = \frac{C}{1 + exp(-k(t-m))} \tag{7}$$

where C is the carrying capacity (maximum value) and k is the growth rate ("steepness" of the trend curve). C and k can be constant or time-varying. This logistic equation allows modelling non-linear growth with saturation, that is when the growth rate of a value decreases with its growth. Prophet supports both automatic and manual tuning of their variability. The library can itself choose optimal points of trend changes by fitting the supplied historical data.

Sticking with the theme that has dominated everybody's life from 2020, we will use data on Covid-19 cases in China.

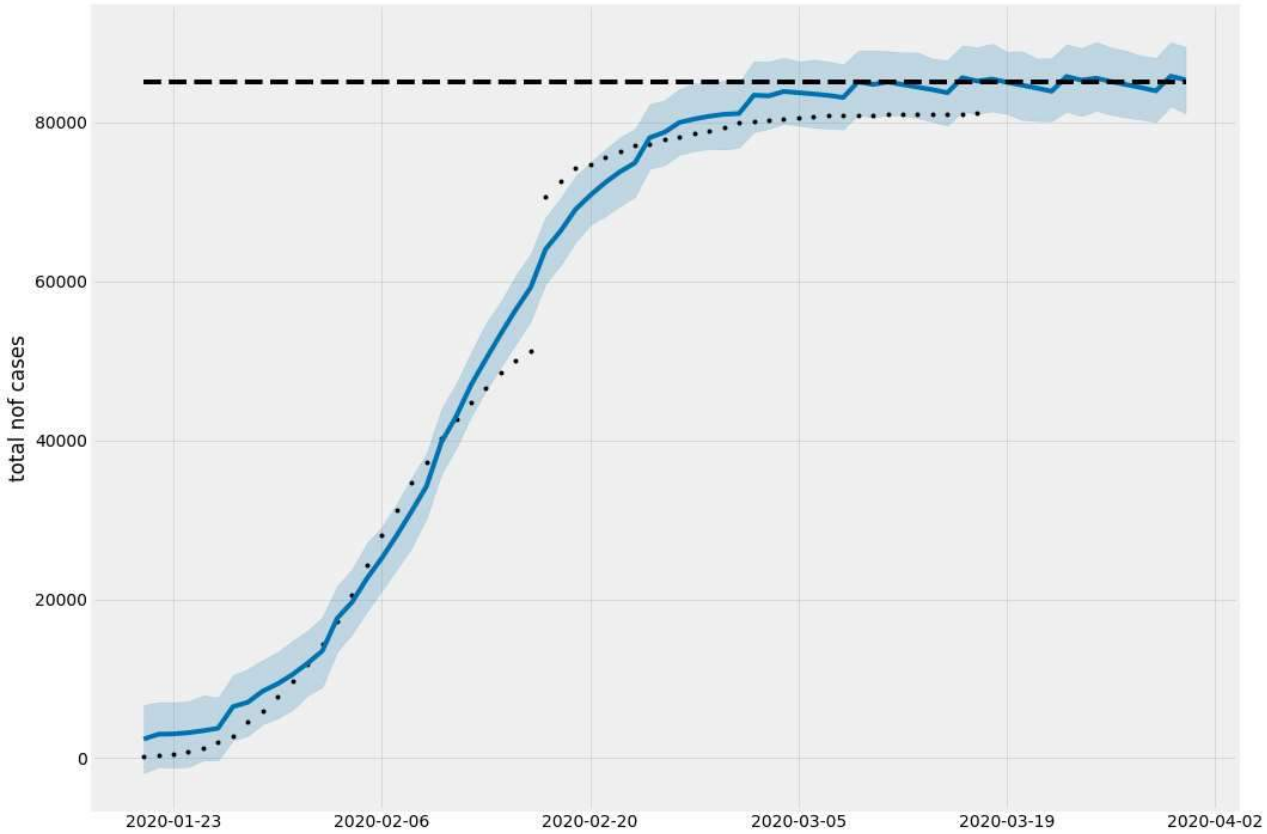
In [6]:
```
df = pd.read_csv('WHO_full_data2003.csv')
# subset the data and rename the columns to Prophet naming convention
df = df.loc[df['location'] == 'China'][['date', 'total_cases']].rename(columns={"date":
df.set_index('ds').plot(figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '')
```

Out[6]: <AxesSubplot:>



We can see the curve flatlining somewhat - in Prophet we can incorporate such knowledge into the model by setting a cap (an upper limit on the forecast value):

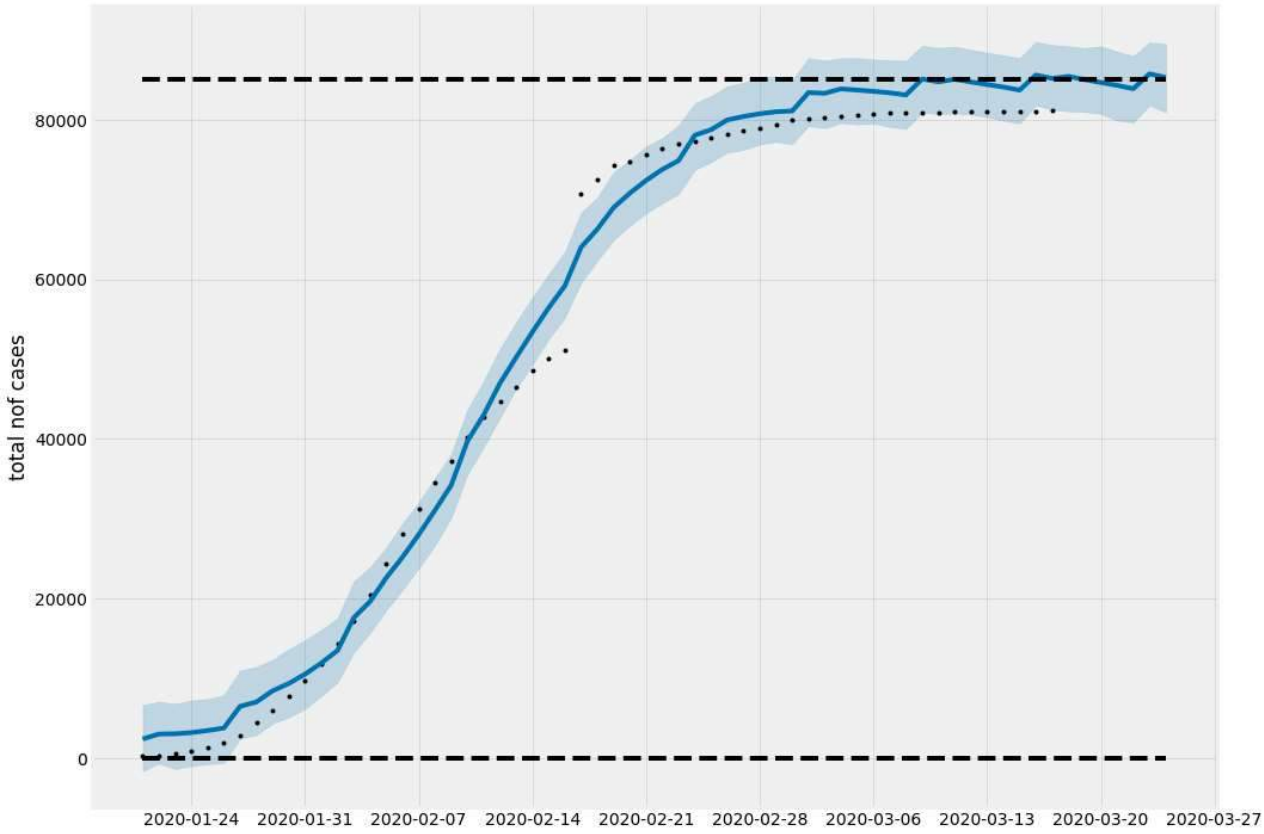
In [7]:
```
# add the cap to the
df['cap'] = 85000
# fit the model
m = Prophet(growth='logistic')
m.fit(df)
# prediction
future = m.make_future_dataframe(periods= 14)
future['cap'] = 85000

# plot the results
fcst = m.predict(future)
fig = m.plot(fcst, figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '', ylabel = 'total n
```



The approach is symmetrics, i.e. we can limit the series from below as well (contrived as it might seem in this particular application ;-)

In [8]:

```
# set the upper & lower limit for the forecast
df['cap'] = 85000
df['floor'] = 0

# fit a model
m = Prophet(growth='logistic')
m.fit(df)

# predict
future = m.make_future_dataframe(periods= 7)
future['cap'] = 85000
future['floor'] = 0
fcst = m.predict(future)

m.plot(fcst, figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '', ylabel = 'total nof cas
print()
```



Seasonality

When dealing with data in practical applications, it is frequently necessary to take into account multiple seasonal patterns occurring in parallel; a classic example would be data related to energy consumption: there are morning vs evening patterns (intraday), workdays vs weekend (weekly) and during the year (annual). Modeling them explicitly tends to be cumbersome (you need to add more equations in exponential smoothing or introduce dummies in ARIMA), which is one of the issues Prophet was designed to overcome. The core of the underlying logic is a the Fourier expansion:

$$S(t) = \sum_{i=1}^{N} \left(a_n cos\left(\frac{2\pi it}{P}\right) + b_n sin\left(\frac{2\pi it}{P}\right)\right) \tag{8}$$

Unpacking this formula:

- `sin` and `cos` function form an orthogonal basis https://en.wikipedia.org/wiki/Orthogonal_functions#Trigonometric_functions
- this means that every function can be represented as a combination as in the equation defined above - recall that in the GAM setup we treat time as the only regressor, so a time series can be viewed as a function of time $S(t)$
- by cutting off the expansion for a certain N we can remove high frequency oscillations $\rightarrow$ low pass filter

Frequency shenanigans

Let us briefly explore how you can specify different seasonality patterns: we will use the energy consumption data from PJM Interconnection LLC (PJM) - a regional transmission organization (RTO) in the United States. The data is available on Kaggle: https://www.kaggle.com/robikscube/hourly-energy-consumption.

In [9]:
```
df = pd.read_csv('pjm_hourly_est.csv')
df['Datetime'] = pd.to_datetime(df['Datetime'])
df.head(5)
```

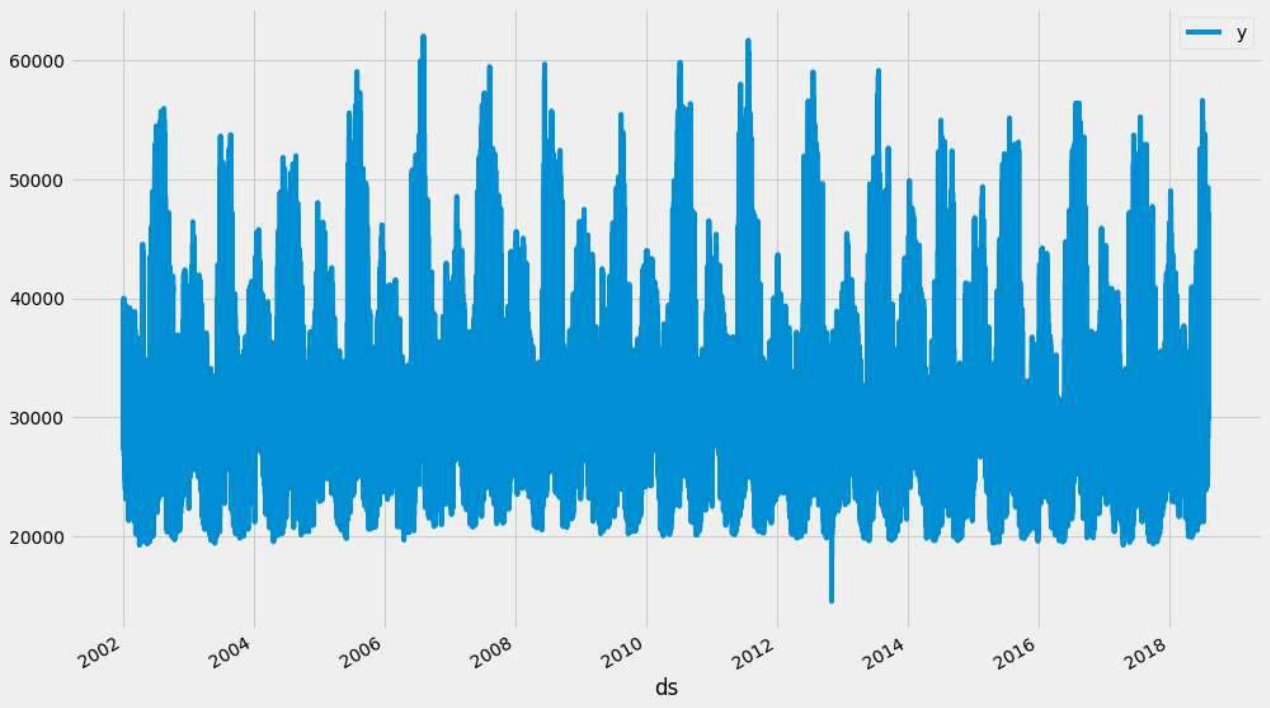
Out[9]:

	Datetime	AEP	COMED	DAYTON	DEOK	DOM	DUQ	EKPC	FE	NI	PJME	PJMW	PJM_Load
0	1998-12-31 01:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	29309.0
1	1998-12-31 02:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	28236.0
2	1998-12-31 03:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	27692.0
3	1998-12-31 04:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	27596.0
4	1998-12-31 05:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	27888.0

Prophet is built for univariate data, so we will pick one series. Notice that we are using hourly data, which we will progressively aggregate to lower frequencies to demonstrate the out-of-the-box functionality that Prophet provides.

In [10]:
```
xdat = df[['Datetime', 'PJME']]
# trim the leading NAs
first_valid = np.where(~np.isnan(xdat['PJME']))[0][0]
xdat = xdat.loc[first_valid:].rename(columns={"Datetime": "ds", "PJME": "y"})
xdat.set_index('ds').plot(figsize=(CFG.img_dim1, CFG.img_dim2))
```

Out[10]:
```
<AxesSubplot:xlabel='ds'>
```



In [11]:

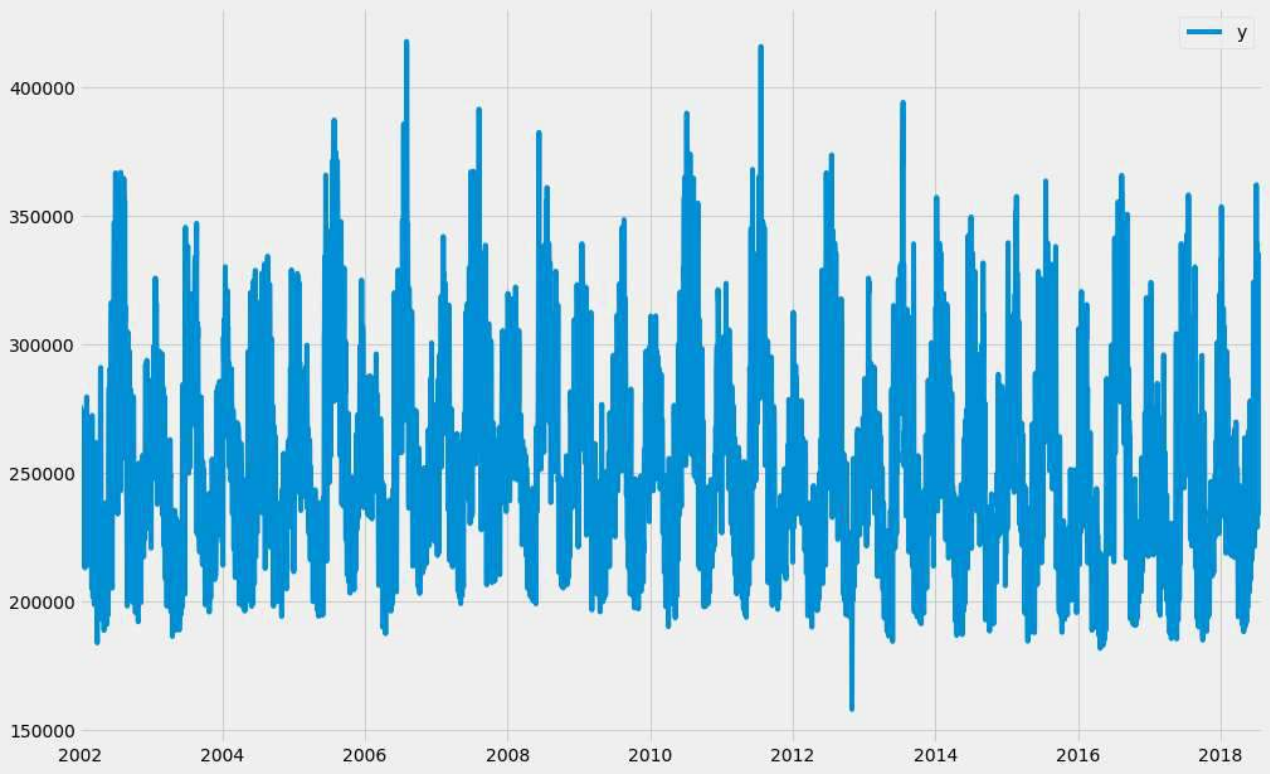
```
# we reduce the dataset size for speed - the only requirement while doing is to keep at
# for each seasonality we intend to fit
m = Prophet().fit(xdat.iloc[:30000])
future = m.make_future_dataframe(periods = 24, freq = 'H')
forecast = m.predict(future)
m.plot_components(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
print()
```

If we are using data sampled at higher frequency than daily, a daily seasonal pattern is automatically

fitted. What if we aggregate the data to daily frequency?

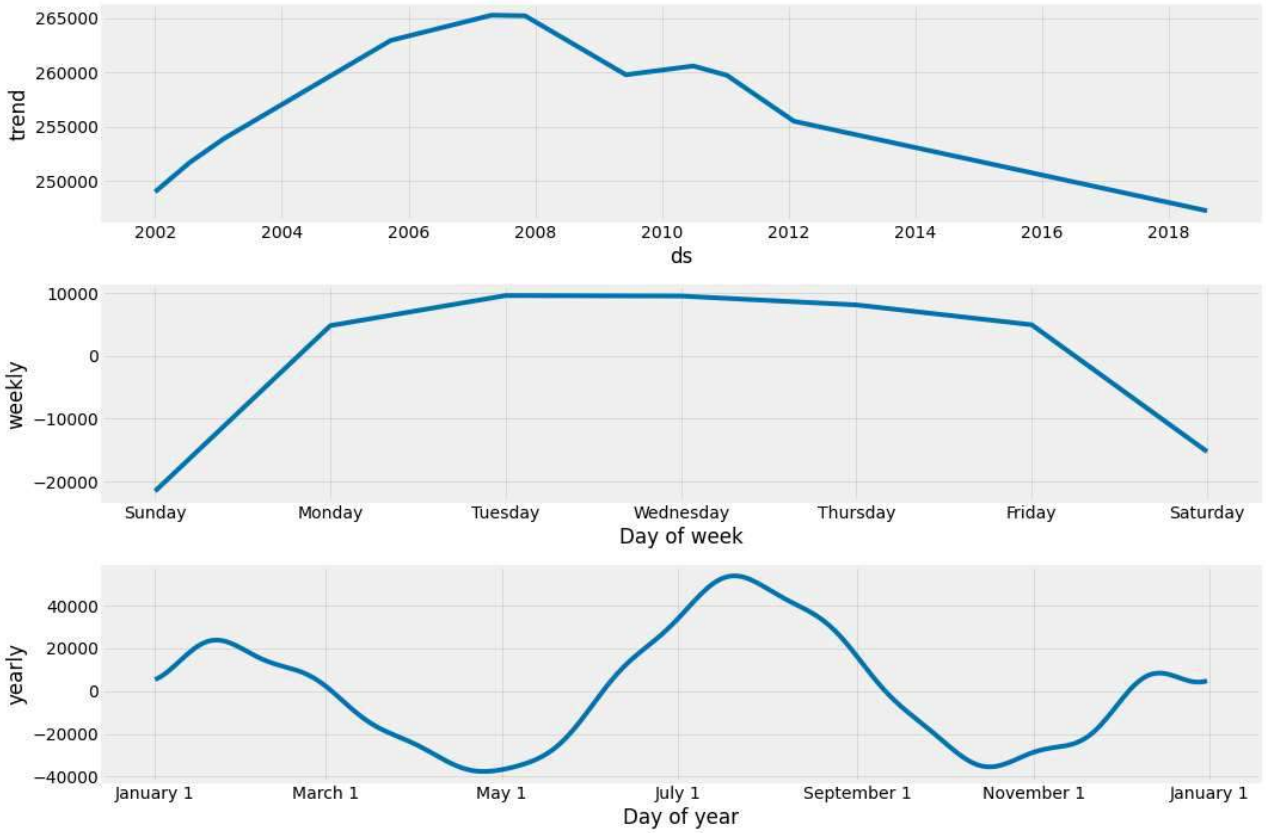
In [12]:

```
xdat = df.resample('D', on='Datetime').sum().reset_index()[['Datetime', 'PJME']].rename
xdat['y'] /= 10^9
# we purge the leading zeros, along with the last observation - we only have a few hour
ix = np.where(xdat['y'] > 0)[0][0]
xdat = xdat.iloc[ix:-1]
xdat.set_index('ds').plot(xlabel = '')
print()
```



In [13]:

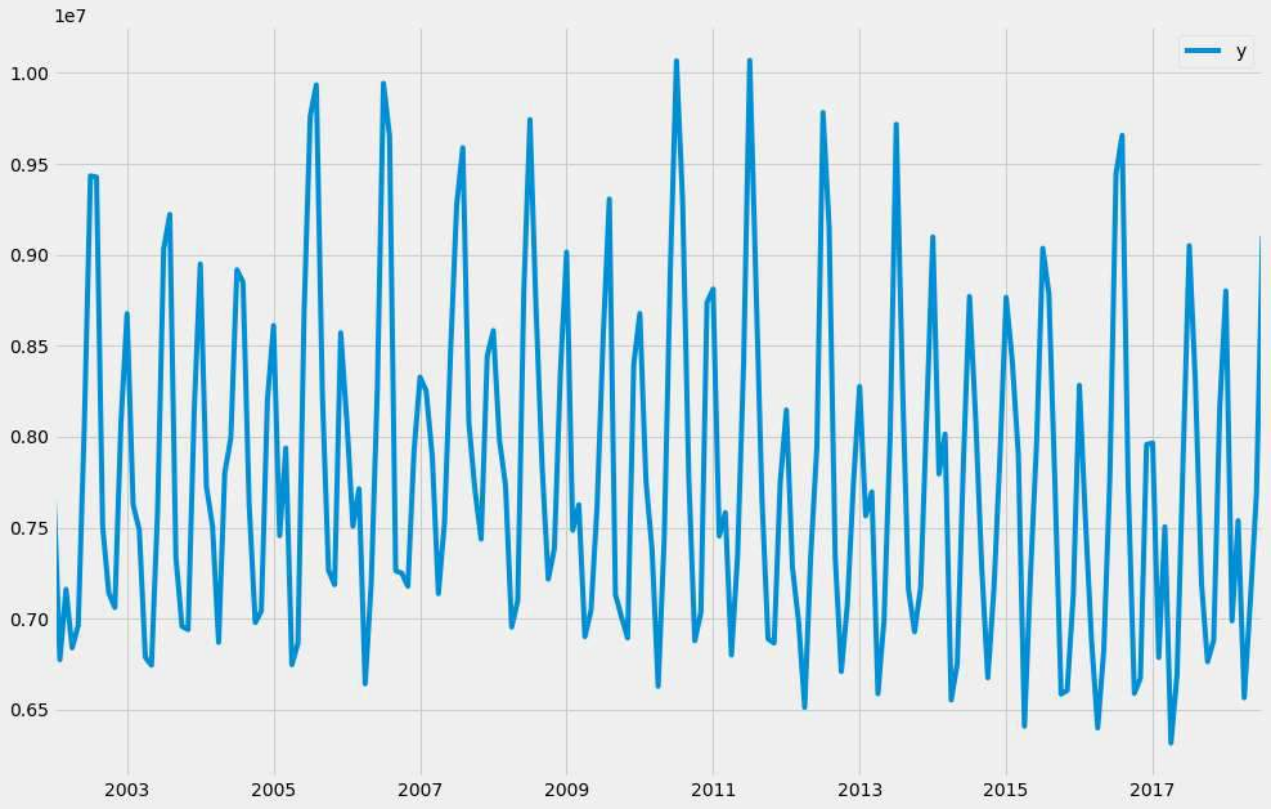
```
# we proceed in a similar manner as before
m = Prophet().fit(xdat)
future = m.make_future_dataframe(periods = 7, freq = 'D')
forecast = m.predict(future)
m.plot_components(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
print()
```



The same logic applies if we downsample the data further:

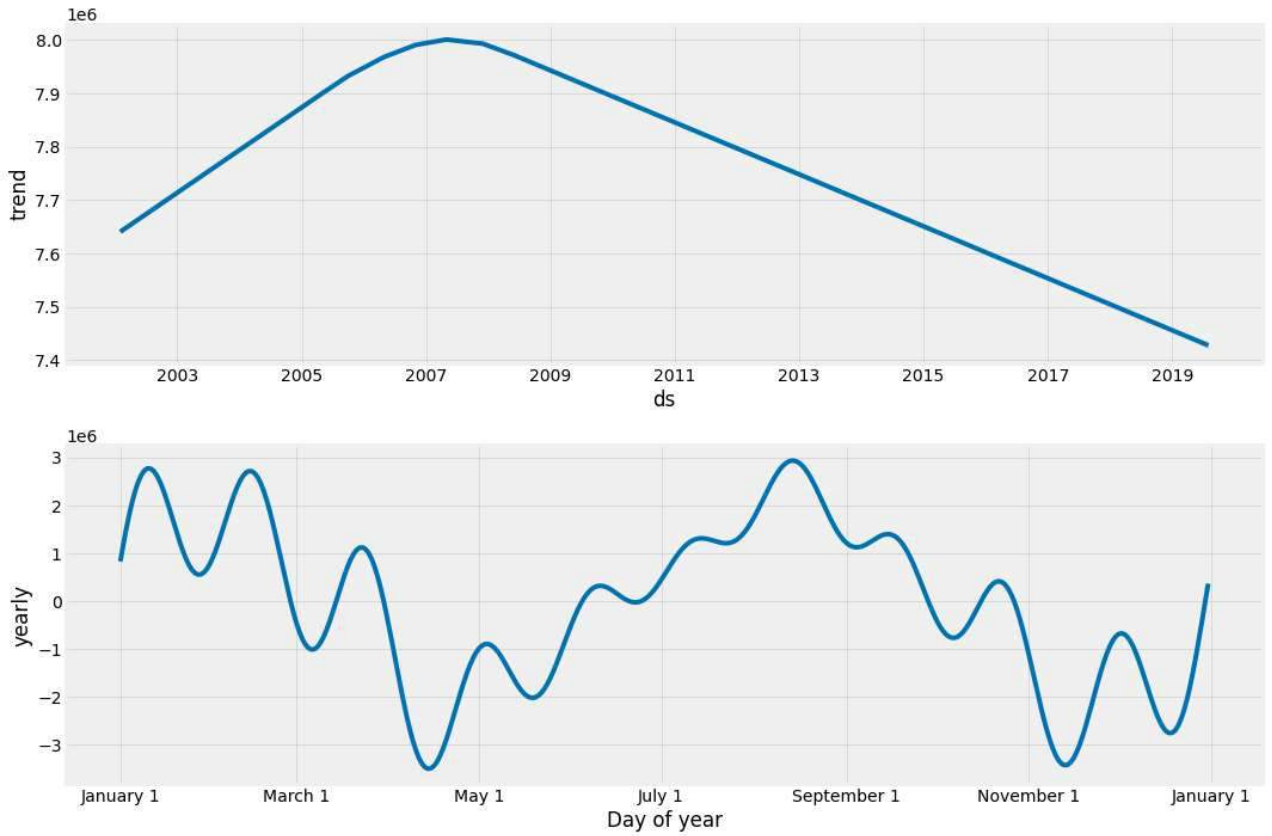
In [14]:

```
xdat = df.resample('M', on='Datetime').sum().reset_index()[['Datetime', 'PJME']].rename
xdat['y'] /= 10^9
# we purge the leading zeros, along with the last observation - we only have a few hour
ix = np.where(xdat['y'] > 0)[0][0]
xdat = xdat.iloc[ix:-1]
xdat.set_index('ds').plot(xlabel = '')
print()
```



In [15]:

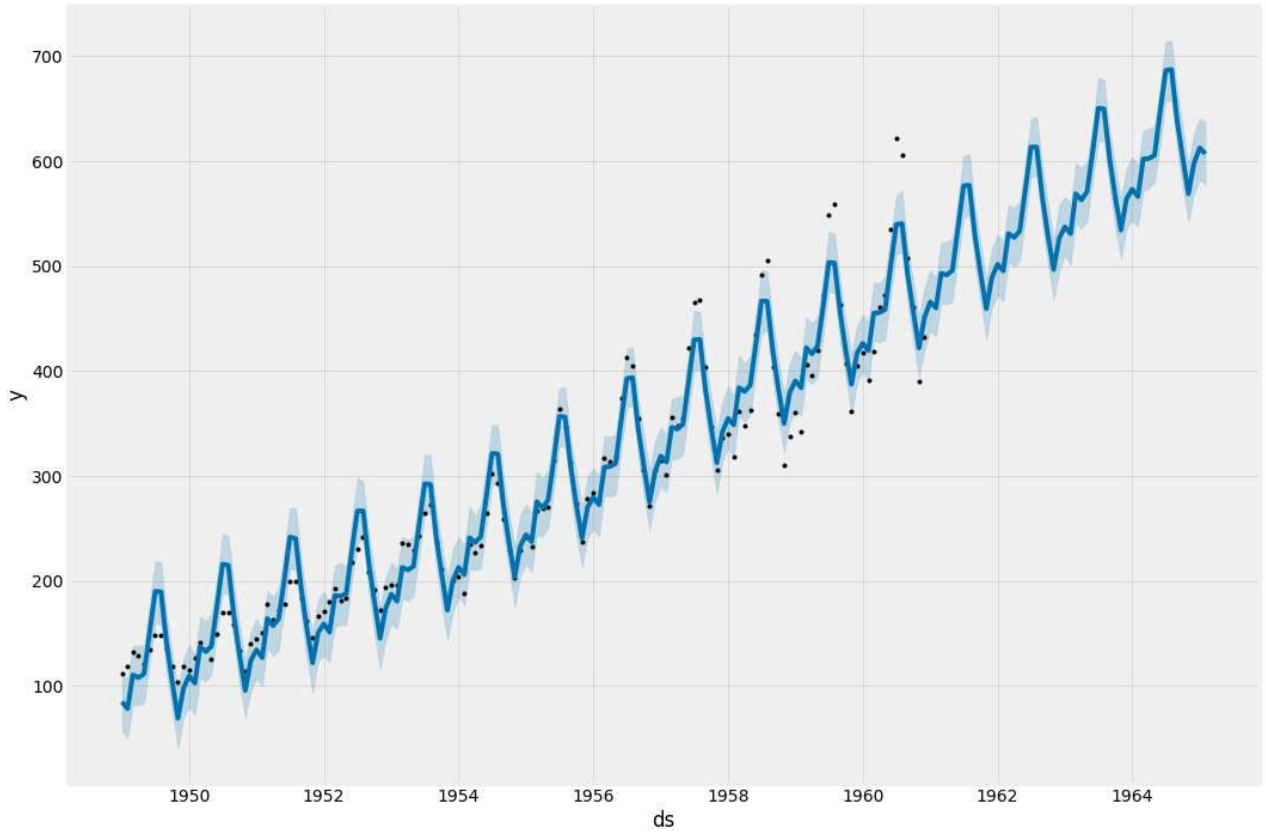
```
m = Prophet().fit(xdat)
future = m.make_future_dataframe(periods = 12, freq = 'M')
forecast = m.predict(future)
m.plot_components(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
print()
```



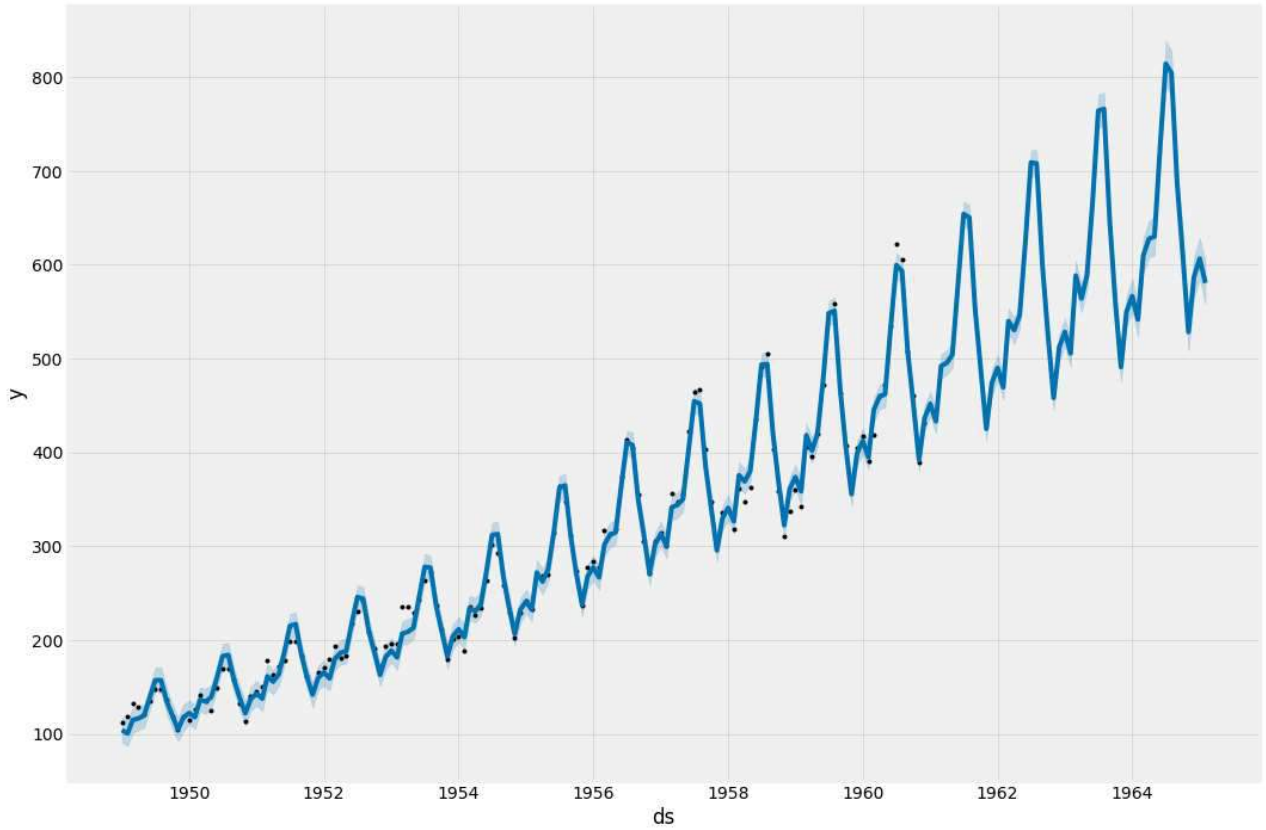
Seasonality specification

Apart from deciding on which frequencies to model explicitly, we have more options to setup our Prophet model. First, there is `seasonality_mode` - additive or multiplicative:

In [16]:
```
# import inspect ; xlist = inspect.getfullargspec(Prophet); xlist.args
df = pd.read_csv(CFG.data_folder + 'example_air_passengers.csv')
m = Prophet(seasonality_mode ='additive')
m.fit(df)
future = m.make_future_dataframe(50, freq='MS')
forecast = m.predict(future)
fig = m.plot(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
plt.show()
```

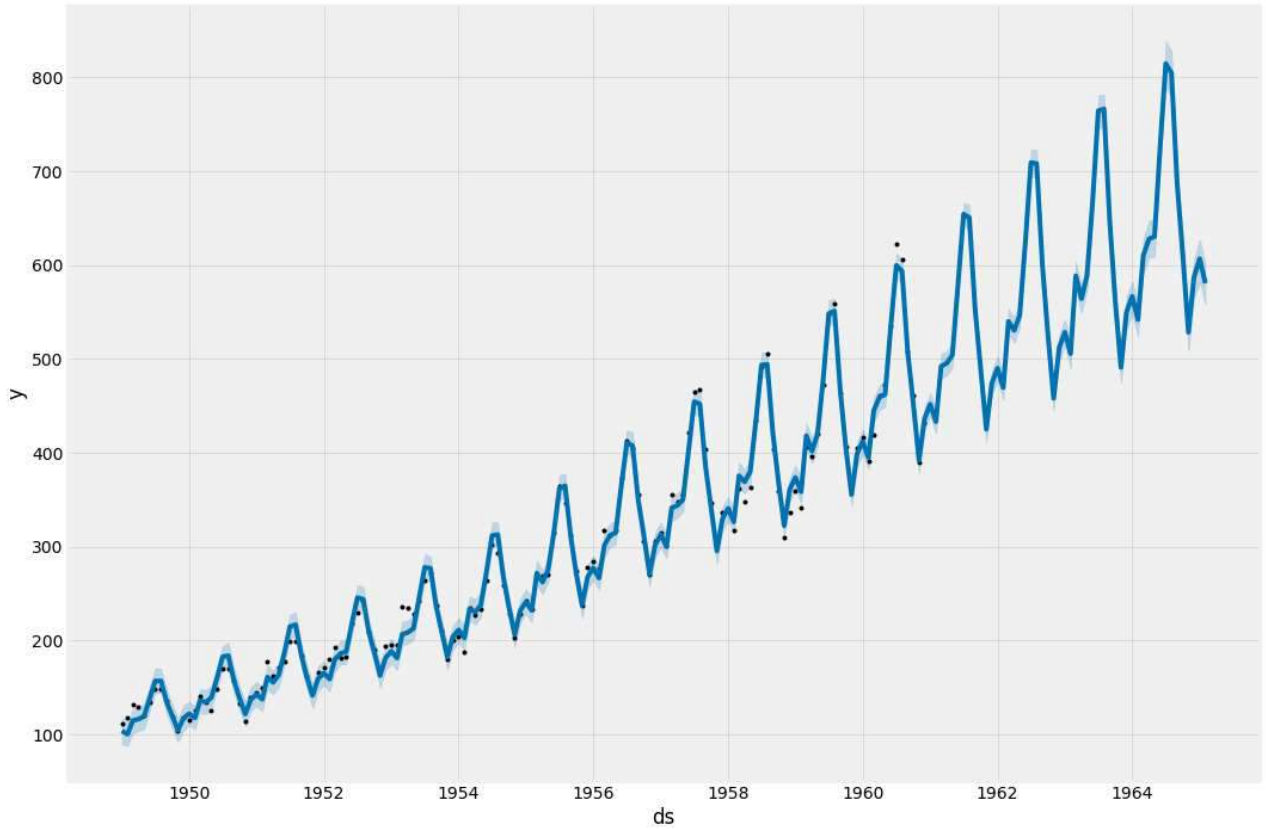


In [17]:
```
m = Prophet(seasonality_mode ='multiplicative')
m.fit(df)
future = m.make_future_dataframe(50, freq='MS')
forecast = m.predict(future)
fig = m.plot(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
plt.show()
```



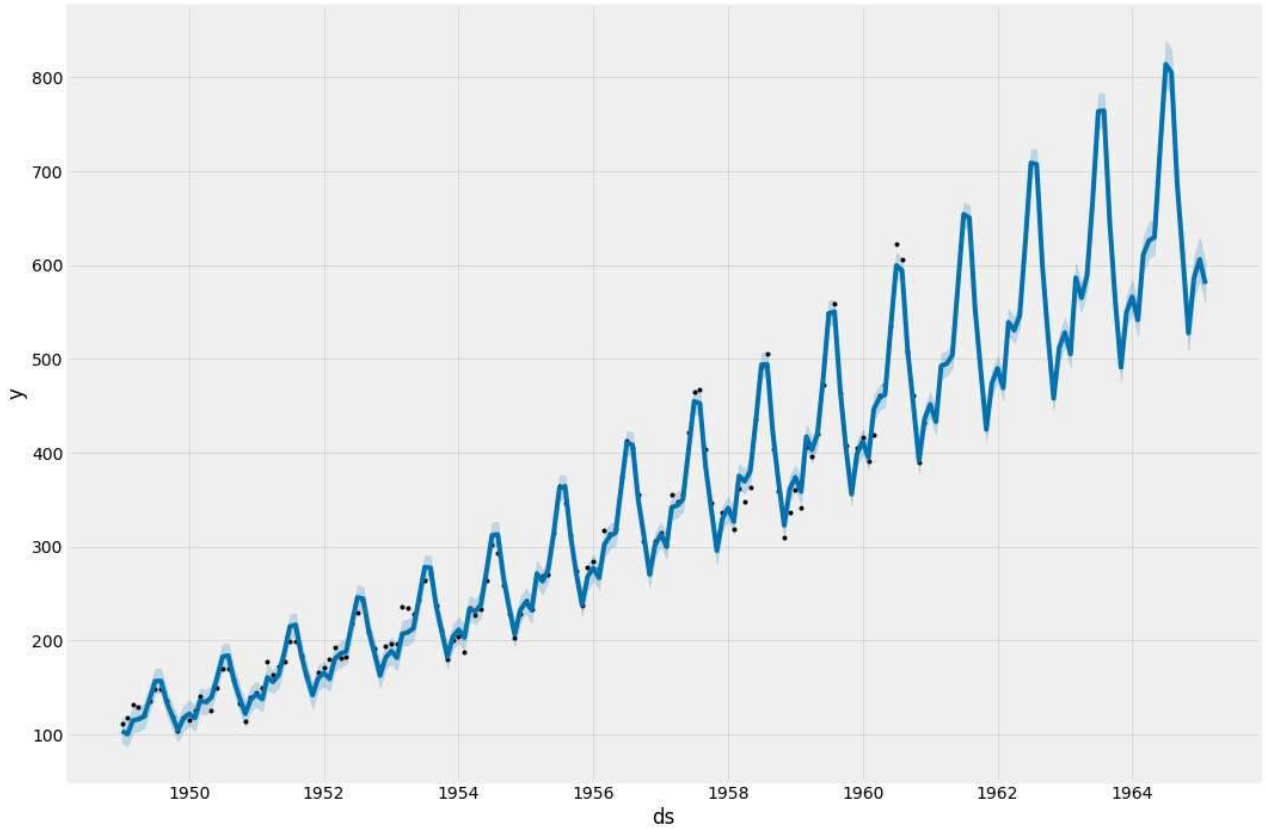
Depending on the problem at hand, we might want to allow strong effects of the seasonal component on the forecast - or have it reduced. This intuition can be quantified by adjusting the `seasonality_prior_scale` argument, which imapcts the extent to which the seasonality model will fit the data (remark for those with Bayesian exposure: works pretty much the way a prior would).

In [18]:
```
# we keep the default value
m = Prophet(seasonality_mode ='multiplicative')
m.fit(df)
future = m.make_future_dataframe(50, freq='MS')
forecast = m.predict(future)
fig = m.plot(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
plt.show()
```



In [19]:

```
# we reduce the seasonality prior
m = Prophet(seasonality_mode ='multiplicative', seasonality_prior_scale = 1)
m.fit(df)
future = m.make_future_dataframe(50, freq='MS')
forecast = m.predict(future)
fig = m.plot(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
plt.show()
```



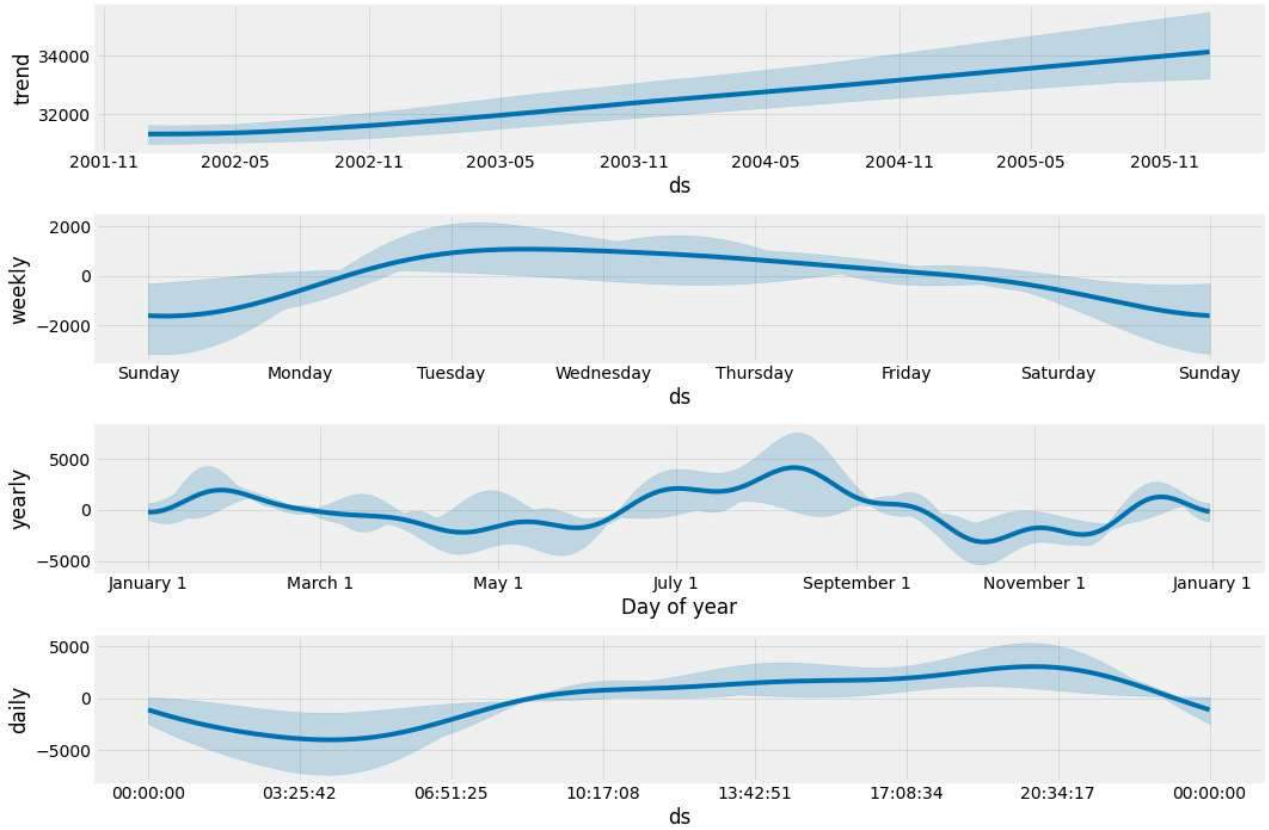
Last but not least, we can - as we usually ought to - use interval forecast, i.e. have our point estimates combined with uncertainty. By default the parameter `mcmc_samples` is set to 0, so to get the interval around seasonality, you must do full Bayesian sampling; uncertainty around trend can be calculated with Maximum A Posteriori (MAP) estimate.

In [20]:
```
df = pd.read_csv('pjm_hourly_est.csv')
df['Datetime'] = pd.to_datetime(df['Datetime'])
xdat = df[['Datetime', 'PJME']]
# trim the leading NAs
first_valid = np.where(~np.isnan(xdat['PJME']))[0][0]
xdat = xdat.loc[first_valid:].rename(columns={"Datetime": "ds", "PJME": "y"})
# xdat.set_index('ds').plot(figsize=(CFG.img_dim1, CFG.img_dim2))

m = Prophet().fit(xdat.iloc[:30000])
future = m.make_future_dataframe(periods = 24, freq = 'H')
forecast = m.predict(future)
m.plot_components(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
print()
```

What if we switch to Bayesian inference (change `mcmc_samples` to a positive integer)?

In [21]:

```
m = Prophet(mcmc_samples = 10).fit(xdat.iloc[:30000])
future = m.make_future_dataframe(periods = 24, freq = 'H')
forecast = m.predict(future)
m.plot_components(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
print()
```



Keep in mind that although PyStan https://pystan.readthedocs.io/en/latest/ is SOTA for MCMC sampling, it still takes some time - especially for long time series.

Special days

So we have handled trend and seasonality - but that does not mean everything else belongs with the random error component. There are data points that are not random per se, but can still have impact on the performance of the model:

- public holidays (Christmas, Easter, New Year, Black Friday)
- special events (World Cup)
- major events like conflict starting or terrorist attacks - when analyzing historical performance of the model, it is frequently useful to remove those
- outliers - this is something of a catch-all category, but frequently possible to identify by inspecting the data.

The Prophet approach to modeling abnormal days is to provide a custom list of events. We assume the effects are independent, so if e.g. a sports event occurs near public holiday, their effects will be captured separately.

Holidays

The dataset we will use to test the Prophet functionality around special days is daily count of bike commuters in Oslo https://www.kaggle.com/konradb/norway-bicycles

In [22]:
```
bikerides = pd.read_csv('bikerides_day.csv', error_bad_lines=False, encoding= 'unicode_
bikerides.head(5)
# the usual formatting
bikerides = bikerides[['Date', 'Volume']].rename(columns={"Date": "ds", "Volume": "y"})
```

We start with the fast and easy way of adding holidays: using the built-in list of country holidays:

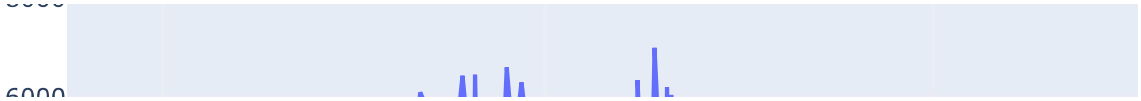
In [23]:
```
m = Prophet()
m.add_country_holidays(country_name='NO')
m.fit(bikerides)
# List the holiday names
m.train_holiday_names
```

Out[23]:
```
0                   Søndag
1           Første påskedag
2           Arbeidernes dag
3           Første pinsedag
4            Første juledag
5         Første nyttårsdag
6              Grunnlovsdag
7             Andre juledag
8              Skjærtorsdag
9                Langfredag
10          Andre påskedag
11    Kristi himmelfartsdag
12          Andre pinsedag
dtype: object
```

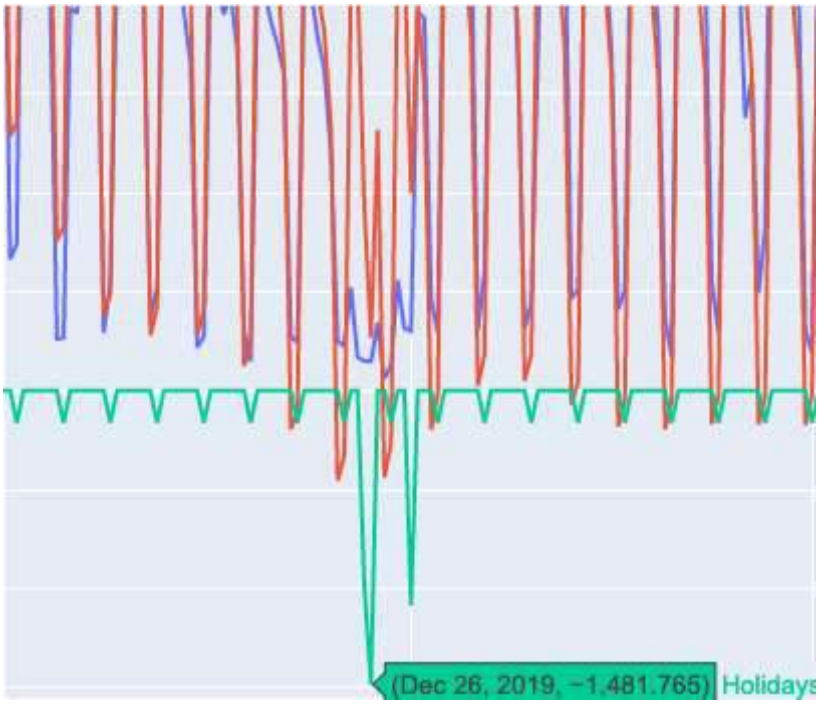
I don't speak Norwegian, but a quick visit to Google Translate shows that Christmas is missing - which, in Europe, is typically a rather important public holiday. We will take care of adding it later, for now let's see how well the model is doing out of the box:

In [24]:
```
# as before, we create a dataframe holding the dates for the entire forecast horizon -
future = m.make_future_dataframe(periods=180, freq='D')
# generate the actual forecast
forecast = m.predict(future)

fig = go.Figure()
# full disclosure: I am changing the plotting style for this one, because I don't know

fig.add_trace(go.Scatter(x=bikerides['ds'], y=bikerides['y'], name='Actual',))
fig.add_trace(go.Scatter(x=forecast['ds'], y=forecast['yhat'], name='Predicted',))
fig.add_trace(go.Scatter(x=forecast['ds'], y=forecast['holidays'], name='Holidays',))
fig.show()
```

8000



Overall the forecast seems directionally ok - but if we zoom a little closer, there are issues:



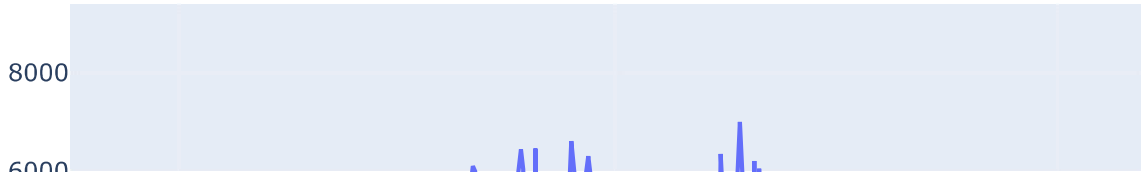
Let's see if we can improve by augmenting the list of holidays: we do that by creating a new dataframe `christmas`, which is subsequently passed as an argument to Prophet. The 'holiday' entry is mostly for interpretation sake, relevant parts are:

- `ds` - so we know when the holiday of interest occurs
- `lower_window` and `upper_window`: those two parameters allow us to incorporate the effect before/after the date, respectively. In our example below `lower_window` equals -1, meaning we anticipate a drop in the number of commuters a day before Christmas, whereas

`upper_window` is 7 - with a lot of people taking time off between Christmas and New Year, the bicycle traffic is likely to decrease for approximately a week.

In [25]:

```
christmas = pd.DataFrame({
  'holiday': 'Christmas',
  'ds': pd.to_datetime(['2017-12-24','2018-12-24','2019-12-24','2020-12-24']),
  'lower_window': -1,
  'upper_window': 7,
})


m = Prophet(holidays=christmas)
m.add_country_holidays(country_name='NO')
m.fit(bikerides)
future = m.make_future_dataframe(periods=0, freq='D')
forecast = m.predict(future)

fig = go.Figure()
# Create and style traces
fig.add_trace(go.Scatter(x=bikerides['ds'], y=bikerides['y'], name='Actual',))
fig.add_trace(go.Scatter(x=forecast['ds'], y=forecast['yhat'], name='Predicted',))
fig.add_trace(go.Scatter(x=forecast['ds'], y=forecast['holidays'], name='Holidays',))
fig.show()
```

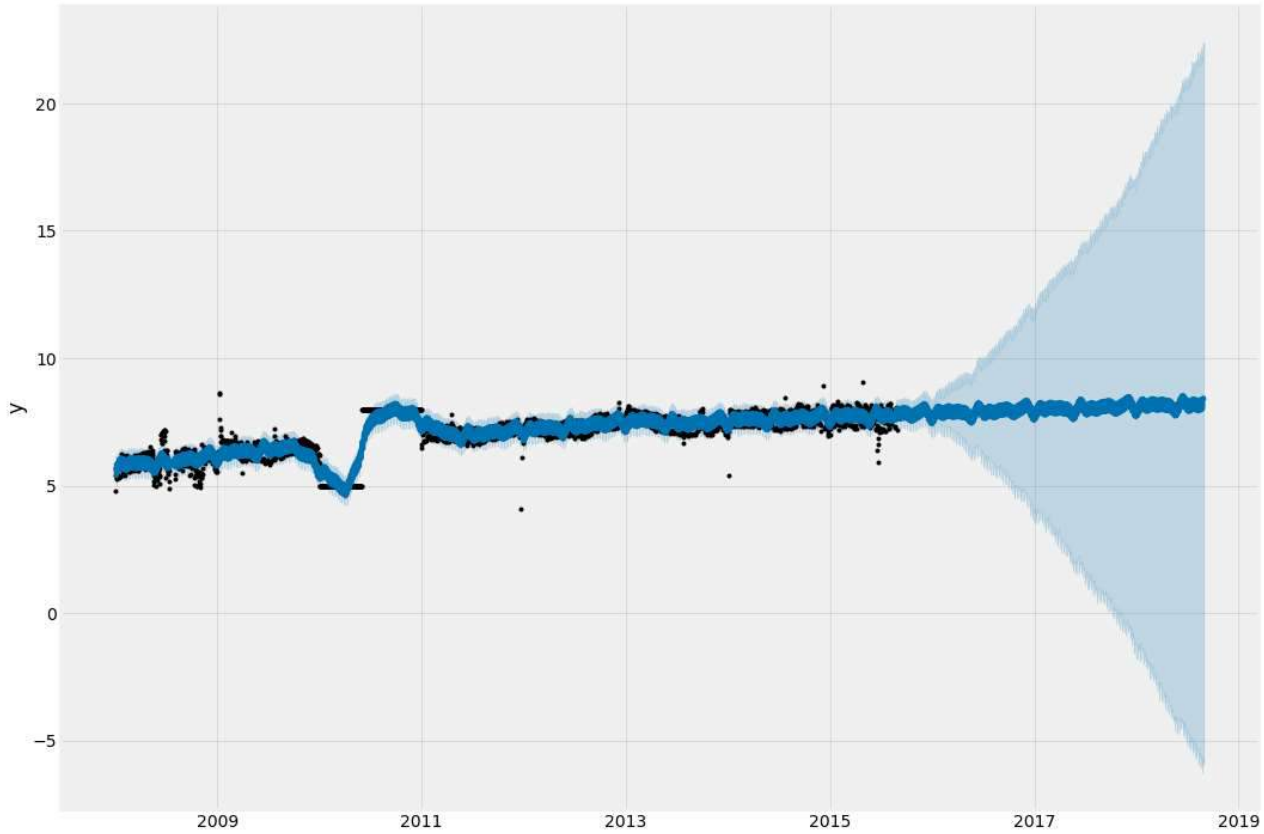


This does look better:

Outliers

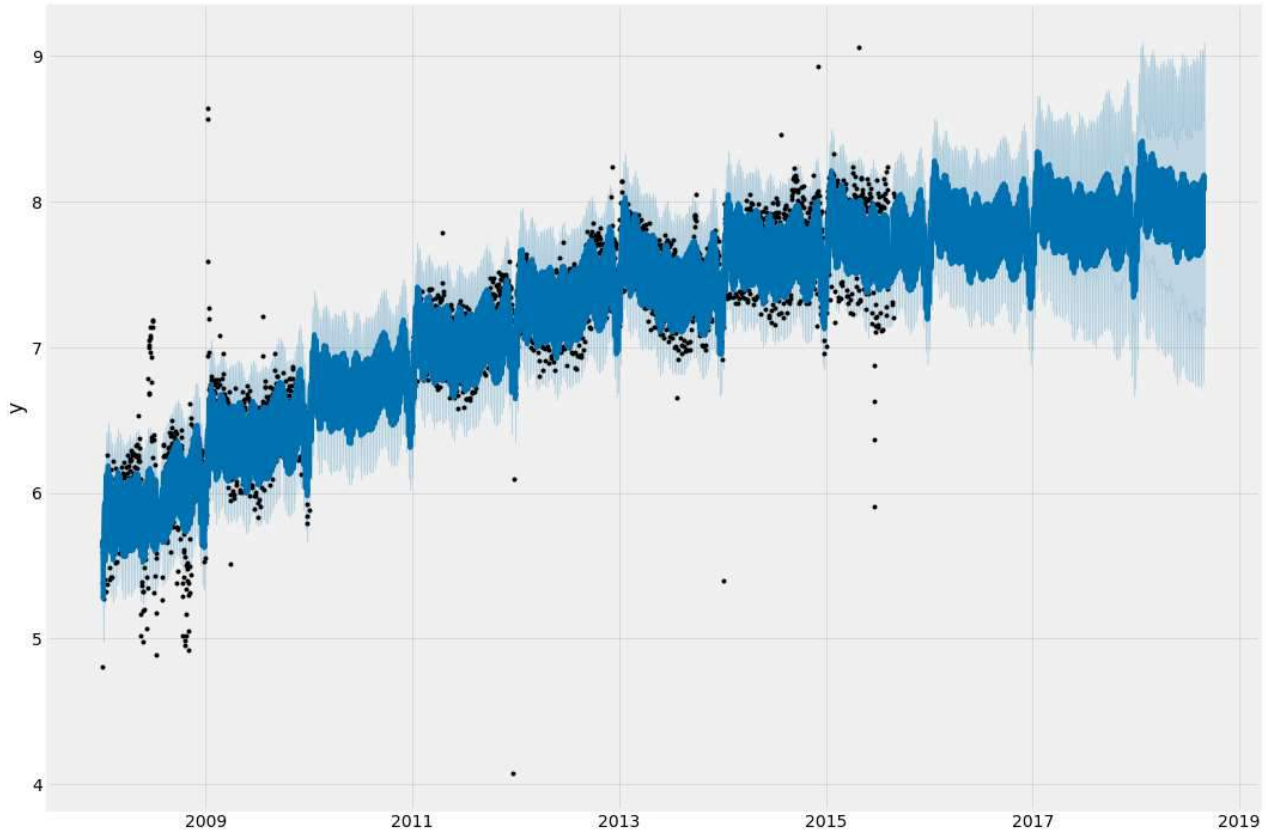
We can use the built-in Prophet functionality to deal with outliers - for the sake of clarity of exposition, we will re-use the example used also in the official documentation, i.e. log of daily count of visits to the Wikipedia page of R language.

In [26]:
```
df = pd.read_csv(CFG.data_folder + 'example_wp_log_R_outliers1.csv')
m = Prophet()
m.fit(df)
future = m.make_future_dataframe(periods=1096)
forecast = m.predict(future)
fig = m.plot(forecast, figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '')
```



The pattern in 2011 does not look plausible (flat line, jump, flat line at a higher level), and those observations are the most likely culprit for the misspecified trend and the resulting bredth of the confidence bands. Prophet allows us to deal with the problem in a very straightforward manner, i.e. by replacing the dubious observations with `None` (not `NaN` - it is an important distinction to keep in mind).

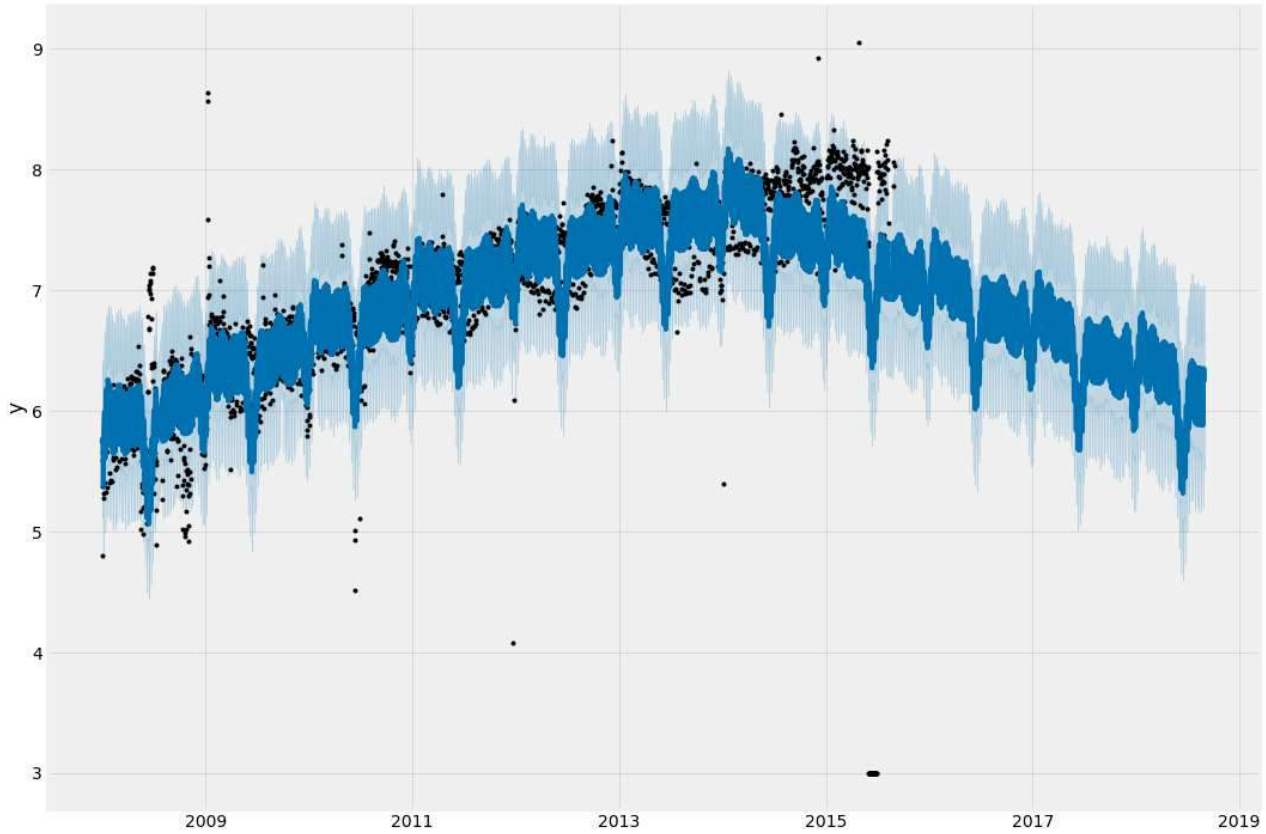
In [27]:
```
df.loc[(df['ds'] > '2010-01-01') & (df['ds'] < '2011-01-01'), 'y'] = None
model = Prophet().fit(df)
fig = model.plot(model.predict(future), figsize=(CFG.img_dim1, CFG.img_dim2), xlabel =
plt.show()
```



Another type of situation we may encounter is a few points whose values are extremely off, so as a result the seasonality estimate is impacted.

In [28]:

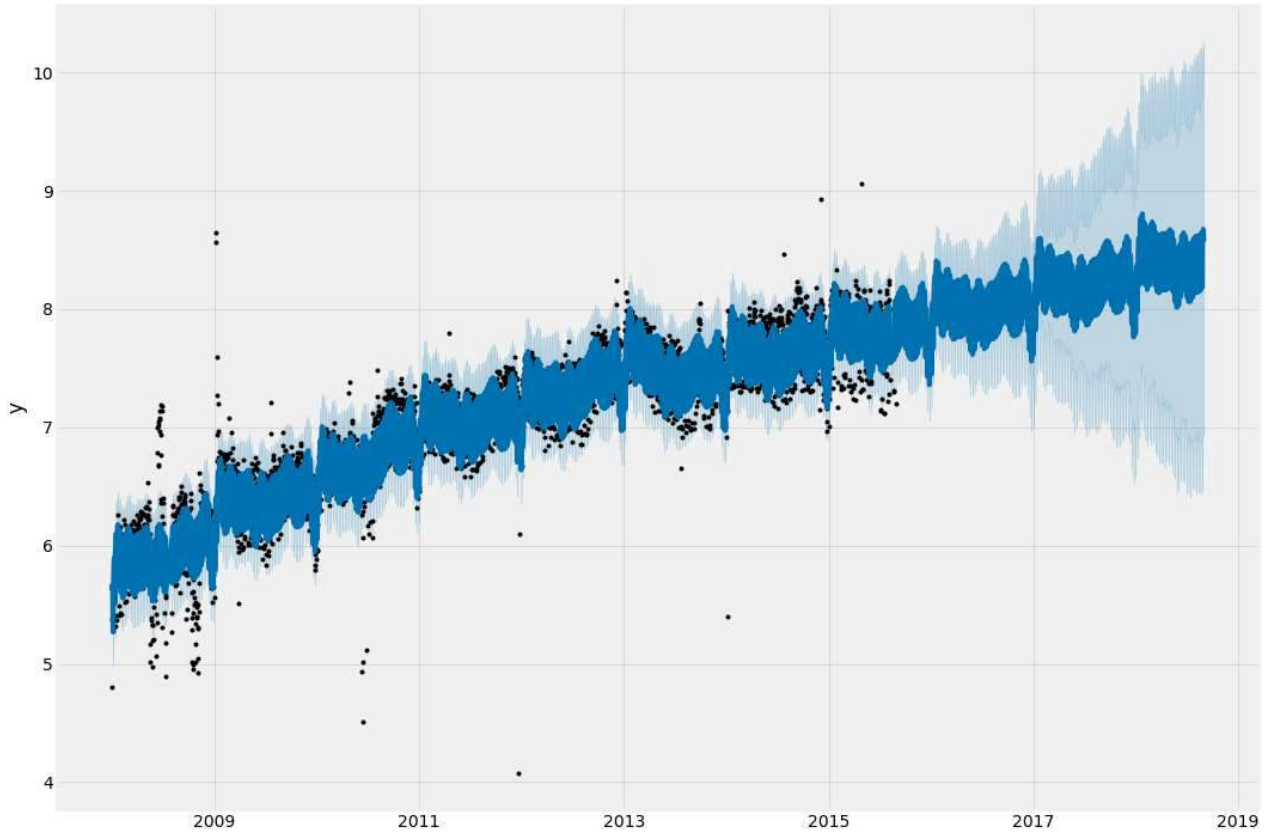
```
df = pd.read_csv(CFG.data_folder + 'example_wp_log_R_outliers2.csv')
m = Prophet()
m.fit(df)
future = m.make_future_dataframe(periods=1096)
forecast = m.predict(future)
fig = m.plot(forecast, figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '')
plt.show()
```



As before, the simplest solution is to get rid of those points and leave the algorithm to interpolate within the sample:

In [29]:

```
df.loc[(df['ds'] > '2015-06-01') & (df['ds'] < '2015-06-30'), 'y'] = None
m = Prophet().fit(df)
forecast = m.predict(future)
fig = m.plot(forecast, figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '')
plt.show()
```



To summarize: leveraging the Prophet functionality around in-sample interpolation, we can solve issues related to outliers by simply replacing them with `None`s.

Performance evaluation

Cross validation for time series can be sometimes challenging, but the Prophet approach takes the hassle out of having to create your own function for a rolling forecast. We utilize the `cross_validation` function: the parameters to specify are the forecast horizon `horizon` and (optionally) size of the initial training period `initial` and the spacing between cutoff dates `period`. By default, the initial training period is set to three times the horizon, and cutoffs are made every half a horizon.

In [30]:
```
df_cv = cross_validation(m, initial = '1000 days', period = '30 days', horizon='30 days
```

The output of cross_validation is a dataframe where for $cutoff$ (last timepoint in the training set) we get a number of values:

- ds is the timepoint in test set
- y is the true value at ds
- $yhat\_lower$ and $yhat\_upper$ are the lower and upper ends of the condifence interval, respectively

In [31]:
```
df_cv.head(10)
```

Out[31]:

	ds	yhat	yhat_lower	yhat_upper	y	cutoff
0	2010-10-27	7.043898	6.665867	7.366526	7.370860	2010-10-26
1	2010-10-28	7.040077	6.671479	7.432210	7.407318	2010-10-26
2	2010-10-29	6.954792	6.593150	7.305080	7.237778	2010-10-26
3	2010-10-30	6.610974	6.257290	6.963424	6.742881	2010-10-26
4	2010-10-31	6.594863	6.267325	6.927461	6.884487	2010-10-26
5	2010-11-01	6.988195	6.626302	7.335836	7.235619	2010-10-26
6	2010-11-02	7.074142	6.703014	7.433693	7.195937	2010-10-26
7	2010-11-03	7.110214	6.767870	7.461749	7.145984	2010-10-26
8	2010-11-04	7.121056	6.784114	7.468600	7.313887	2010-10-26
9	2010-11-05	7.050105	6.703454	7.411577	7.187657	2010-10-26

While informative, the object created with `cross_validation` contains an abundance of information. This can be summarized in a more succinct manner by aggregating to yield performance metrics:

In [32]:

```
df_p = performance_metrics(df_cv)
df_p.head(5)
```

Out[32]:

	horizon	mse	rmse	mae	mape	mdape	coverage
0	3 days	0.095431	0.308920	0.168699	0.024444	0.017776	0.919137
1	4 days	0.095225	0.308586	0.169153	0.024432	0.017515	0.918633
2	5 days	0.084354	0.290438	0.160576	0.023417	0.017320	0.929825
3	6 days	0.037883	0.194636	0.146926	0.019825	0.016139	0.901098
4	7 days	0.037273	0.193063	0.147530	0.019997	0.015737	0.918744

The values from the `cross_validation` objects are aggregated for each horizon and different metrics are calculated.

Full pipeline

We are in a position to combine all the building block - we will use the Superstore dataset (a version can be found here: https://www.kaggle.com/bravehart101/sample-supermarket-dataset).

In [33]:

```
!pip install xlrd
```

```
Collecting xlrd
  Downloading xlrd-2.0.1-py2.py3-none-any.whl (96 kB)
     |████████████████████████████████| 96 kB 2.3 MB/s
Installing collected packages: xlrd
Successfully installed xlrd-2.0.1
```

```
WARNING: Running pip as root will break packages and permissions. You should install pac
kages reliably by using venv: https://pip.pypa.io/warnings/venv
```

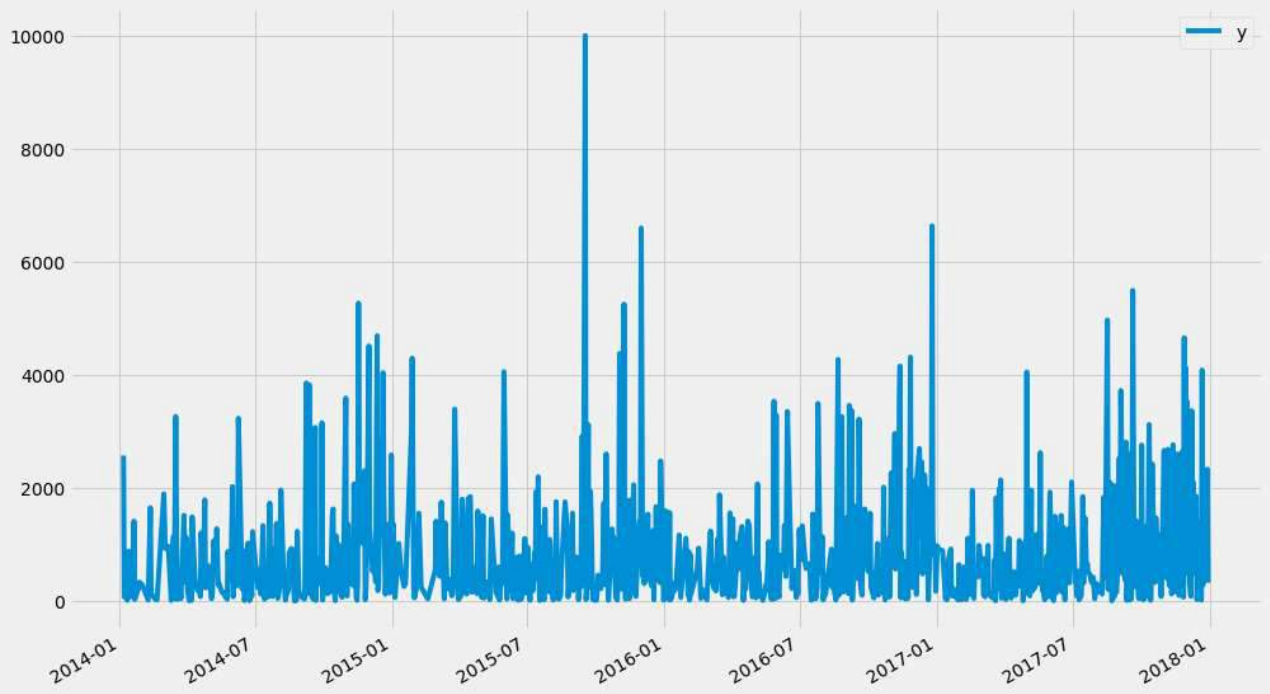
In [34]:
```
df = pd.read_excel(CFG.data_folder + "Sample - Superstore.xls")
df = df.loc[df['Category'] == 'Furniture']
cols = ['Row ID', 'Order ID', 'Ship Date', 'Ship Mode', 'Customer ID',
        'Customer Name', 'Segment', 'Country', 'City', 'State', 'Postal Code', 'Region'
        'Product ID', 'Category', 'Sub-Category', 'Product Name', 'Quantity', 'Discount
df.drop(cols, axis=1, inplace=True)
df = df.sort_values('Order Date')
df = df.groupby('Order Date')['Sales'].sum().reset_index()

# Prophet-friendly format
df.rename(columns={"Order Date": "ds", "Sales": "y"}, inplace = True)
```

In the previous module we aggregated the dataset to monthly frequency - some observations were missing and ARIMA does not have a straightforward method of dealing with that (interpolation in time series is a serious topic, deserving its own module probably). Prophet does not suffer from the same limitation, so we stay at daily frequency.

What does our series look like?

In [35]:
```
df.set_index('ds').plot(figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '')
```

Out[35]: <AxesSubplot:>



What is the time range we are dealing with?

In [36]:
```
min(df['ds']), max(df['ds'])
```

Out[36]: (Timestamp('2014-01-06 00:00:00'), Timestamp('2017-12-30 00:00:00'))

With four complete cycles present, we can keep the last three months of 2017 as validation and use

the rest of the data for training:

In [37]:
```
df_train = df.loc[df['ds'] < '2017-10-01']
df_valid = df.loc[df['ds'] >= '2017-10-01']
print(df_train.shape, df_valid.shape)
```

```
(810, 2) (79, 2)
```

Next step is specifying the form of our model - we can combine this step with hyperparameter tuning. In order to keep the running time of this notebook within reasonable limits, the possible parameter ranges to a bare minimum needed to demonstrate the functionality (in a real application the only real limit is your patience).

In [38]:
```
param_grid = {
    # tuning those parameters can potentially improve the performance of our model
    'changepoint_prior_scale': [0.001, 0.1],
#     'seasonality_prior_scale': [0.01, 1.0],
#     'holidays_prior_scale': [0.01, 0.1],
    'seasonality_mode': ['additive', 'multiplicative'],
}

# Generate all combinations of parameters
all_params = [dict(zip(param_grid.keys(), v)) for v in itertools.product(*param_grid.va
rmses = []

# Quick peek at what our combinations look like
all_params
```

Out[38]:
```
[{'changepoint_prior_scale': 0.001, 'seasonality_mode': 'additive'},
 {'changepoint_prior_scale': 0.001, 'seasonality_mode': 'multiplicative'},
 {'changepoint_prior_scale': 0.1, 'seasonality_mode': 'additive'},
 {'changepoint_prior_scale': 0.1, 'seasonality_mode': 'multiplicative'}]
```

In [39]:
```
for params in all_params:
    m = Prophet(**params).fit(df)
    df_cv = cross_validation(m, initial = '100 days', period = '30 days', horizon='30 d
    df_p = performance_metrics(df_cv, rolling_window=1)
    rmses.append(df_p['rmse'].values[0])

# Find the best parameters
tuning_results = pd.DataFrame(all_params)
tuning_results['rmse'] = rmses
print(tuning_results)

# Python
best_params = all_params[np.argmin(rmses)]
print(best_params)
```

```
   changepoint_prior_scale seasonality_mode         rmse
0                    0.001         additive  8545.619231
1                    0.001   multiplicative  2539.010606
2                    0.100         additive  3882.768074
3                    0.100   multiplicative  2112.955185
{'changepoint_prior_scale': 0.1, 'seasonality_mode': 'multiplicative'}
```

With the tuned parameters available we can proceed to setup the complete model - as mentioned before, the parameters can included in the tuning part can be left at default values (`auto` for

`yearly_seasonality` is automatically set to `True` if there are multiple years in the data etc):

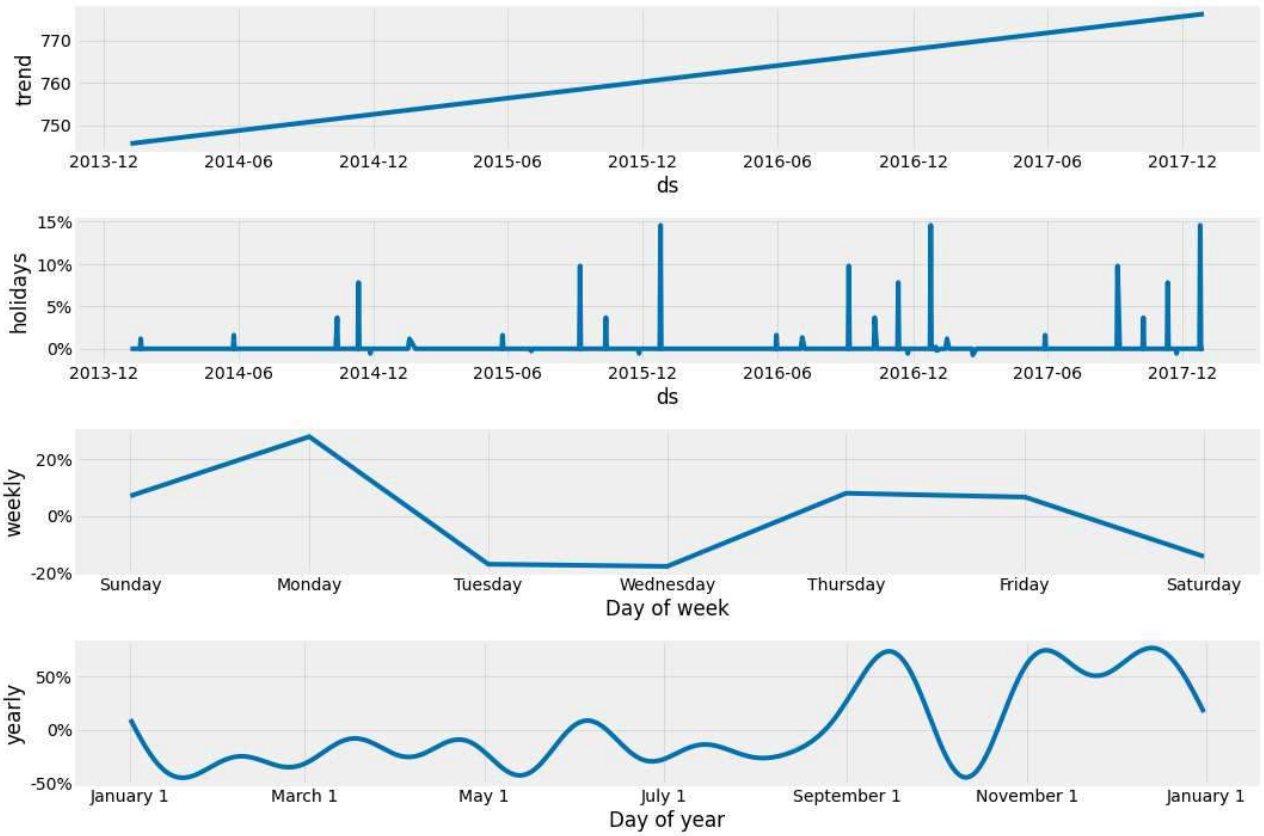
In [40]:
```
m = Prophet(**params)
m.add_country_holidays(country_name='US')
m.fit(df_train)
```

Out[40]: `<fbprophet.forecaster.Prophet at 0x7f6e327fdd90>`

Model component inspection:

In [41]:
```
future = m.make_future_dataframe(periods = 92, freq = 'D')
forecast = m.predict(future)
m.plot_components(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
print()
```

```
/opt/conda/lib/python3.7/site-packages/fbprophet/plot.py:422: UserWarning:

FixedFormatter should only be used together with FixedLocator

/opt/conda/lib/python3.7/site-packages/fbprophet/plot.py:422: UserWarning:

FixedFormatter should only be used together with FixedLocator

/opt/conda/lib/python3.7/site-packages/fbprophet/plot.py:422: UserWarning:

FixedFormatter should only be used together with FixedLocator
```



By default, the `forecast` dataframe contains an abundance of information, including point/upper/lower estimates of the contribution made to the forecast by the trend component, different seasonalities and each holiday defined in the model:

In [42]:
```
forecast.columns
```

Out[42]:
```
Index(['ds', 'trend', 'yhat_lower', 'yhat_upper', 'trend_lower', 'trend_upper',
       'Christmas Day', 'Christmas Day_lower', 'Christmas Day_upper',
       'Christmas Day (Observed)', 'Christmas Day (Observed)_lower',
       'Christmas Day (Observed)_upper', 'Columbus Day', 'Columbus Day_lower',
       'Columbus Day_upper', 'Independence Day', 'Independence Day_lower',
       'Independence Day_upper', 'Independence Day (Observed)',
       'Independence Day (Observed)_lower',
       'Independence Day (Observed)_upper', 'Labor Day', 'Labor Day_lower',
       'Labor Day_upper', 'Martin Luther King Jr. Day',
       'Martin Luther King Jr. Day_lower', 'Martin Luther King Jr. Day_upper',
       'Memorial Day', 'Memorial Day_lower', 'Memorial Day_upper',
       'New Year's Day', 'New Year's Day_lower', 'New Year's Day_upper',
       'New Year's Day (Observed)', 'New Year's Day (Observed)_lower',
       'New Year's Day (Observed)_upper', 'Thanksgiving', 'Thanksgiving_lower',
       'Thanksgiving_upper', 'Veterans Day', 'Veterans Day_lower',
       'Veterans Day_upper', 'Veterans Day (Observed)',
       'Veterans Day (Observed)_lower', 'Veterans Day (Observed)_upper',
       'Washington's Birthday', 'Washington's Birthday_lower',
       'Washington's Birthday_upper', 'holidays', 'holidays_lower',
       'holidays_upper', 'multiplicative_terms', 'multiplicative_terms_lower',
       'multiplicative_terms_upper', 'weekly', 'weekly_lower', 'weekly_upper',
       'yearly', 'yearly_lower', 'yearly_upper', 'additive_terms',
       'additive_terms_lower', 'additive_terms_upper', 'yhat'],
      dtype='object')
```

While interesting if you want to examine every component in detail, for this example we will focus on the point forecast and the associated confidence band:

In [43]:
```
xfor = forecast[['ds', 'yhat', 'yhat_upper', 'yhat_lower']].loc[forecast['ds'] >= '2017
xfor.head(10)
```

Out[43]:

	ds	yhat	yhat_upper	yhat_lower
811	2017-10-01	812.671120	2037.491927	-467.352632
812	2017-10-02	921.886930	2150.515664	-339.999053
813	2017-10-03	522.026949	1715.249154	-715.163974
814	2017-10-04	469.094054	1692.555204	-774.434365
815	2017-10-05	626.556709	1752.242465	-596.988799
816	2017-10-06	578.140940	1759.277317	-656.913951
817	2017-10-07	383.105612	1650.342963	-768.947715
818	2017-10-08	522.740225	1778.130733	-646.883647
819	2017-10-09	693.804209	1947.670466	-659.323210
820	2017-10-10	302.930461	1448.206335	-917.239313

In [44]:
```
# we combine the forecast dataframe with the original test data
# we use a left outer join because of the missing observations in the original data
xfor = pd.merge(left = xfor, right = df, on = 'ds', how = 'left')
```

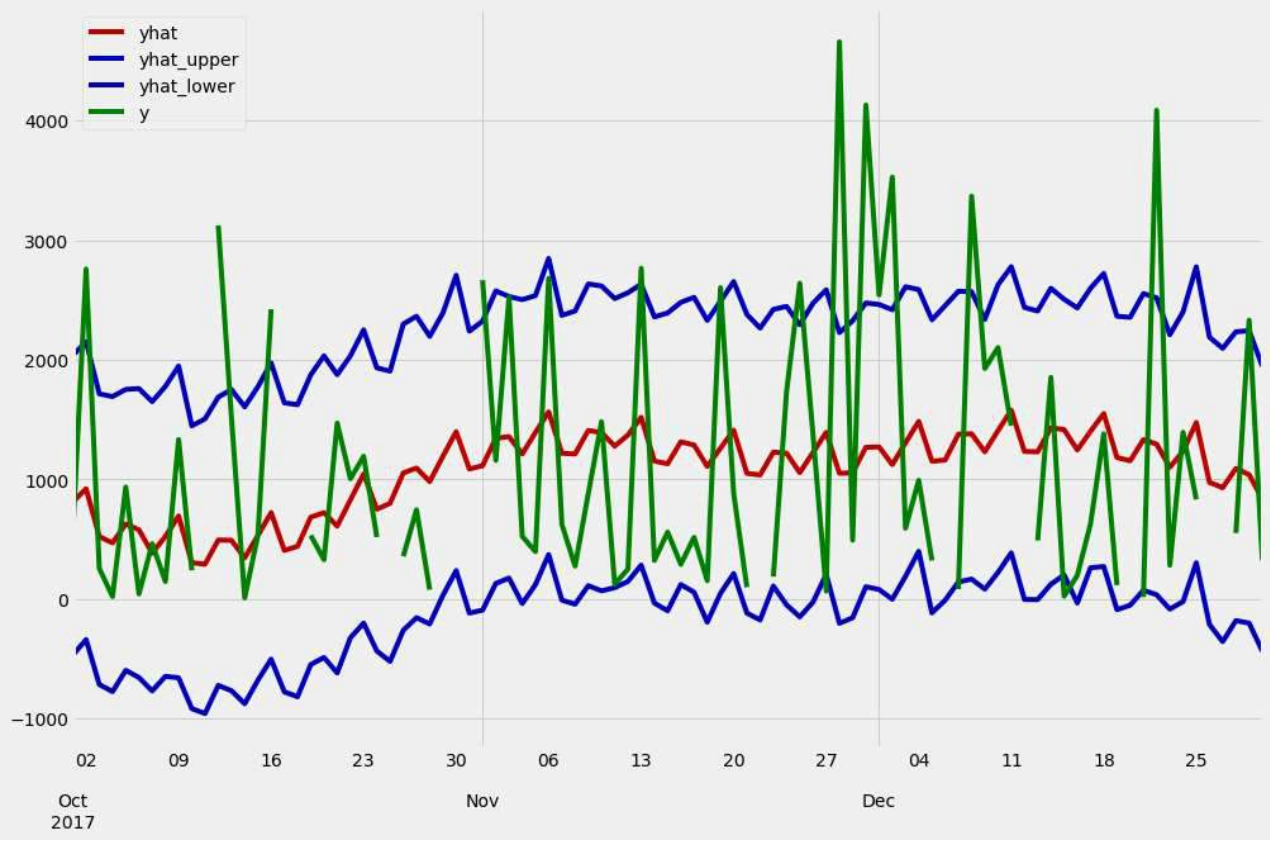
In [45]:
```
xfor.head(10)
```

Out[45]:

	ds	yhat	yhat_upper	yhat_lower	y
0	2017-10-01	812.671120	2037.491927	-467.352632	559.760
1	2017-10-02	921.886930	2150.515664	-339.999053	2758.464
2	2017-10-03	522.026949	1715.249154	-715.163974	255.208
3	2017-10-04	469.094054	1692.555204	-774.434365	19.980
4	2017-10-05	626.556709	1752.242465	-596.988799	935.802
5	2017-10-06	578.140940	1759.277317	-656.913951	41.960
6	2017-10-07	383.105612	1650.342963	-768.947715	462.430
7	2017-10-08	522.740225	1778.130733	-646.883647	145.764
8	2017-10-09	693.804209	1947.670466	-659.323210	1333.393
9	2017-10-10	302.930461	1448.206335	-917.239313	239.358

In [46]:
```
colors = ['#BB0000' , '#0000BB', '#0000BB', 'green']

xfor['ds'] = pd.to_datetime(xfor['ds'])
xfor.set_index('ds').plot(color = colors, figsize=(CFG.img_dim1, CFG.img_dim2), xlabel
```

Out[46]: <AxesSubplot:>



While there is definitely space for improvement - the model does not capture the magnitude of the variations, and the associated confidence interval is rather broad - this example demonstrates that

with minimal effort, we can use Prophet to model time series in a fast and intrepretable manner.

Using covariates

It is not a terribly common situation, but sometimes we get lucky and there is extra information we can use to improve the quality of our forecast: stable weather prediction, or some economic indicators made known earlier than others. In this section we demonstrate how this functionality works in Prophet. First thing we do, we load our dataset (recorded performance of the teams in the Autonomous Greenhouse Challenge) - we will focus on the subset of the original problem, namely how to predict the air temperature based on the settings in the greenhouse environment. The original dataset can be found here: https://www.kaggle.com/piantic/autonomous-greenhouse-challengeagc-2nd-2019.

In [47]:
```
xdat = pd.read_csv('../input/greenhouse-dataset/GreenhouseClimate1.csv')
xdat.head(3)
```

Out[47]:

	time	AssimLight	BlackScr	CO2air	Cum_irr	EC_drain_PC	EnScr	HumDef	PipeGrow	PipeL
0	43815.00000	0.0	35.000000	439.0	0.48	3.61	96.0	6.96	0.0	5
1	43815.00347	0.0	85.000001	459.0	0.72	3.61	96.0	7.45	0.0	4
2	43815.00694	0.0	95.999999	461.0	0.72	3.61	94.6	5.99	0.0	4

3 rows × 50 columns

That's a lot of columns, and we will get to that in a moment - but first, we need to do something about the timestamp. The original data was sampled every 5 minutes and stored in Excel format, which is why need to adjust the epoch start:

In [48]:
```
xdat['time'] = pd.to_datetime(xdat['time'], unit = 'D', origin = "1899-12-30")
xdat.head(3)
```

Out[48]:

	time	AssimLight	BlackScr	CO2air	Cum_irr	EC_drain_PC	EnScr	HumDef	PipeGrow
0	2019-12-16 00:00:00.000000000	0.0	35.000000	439.0	0.48	3.61	96.0	6.96	0.0
1	2019-12-16 00:04:59.808000256	0.0	85.000001	459.0	0.72	3.61	96.0	7.45	0.0
2	2019-12-16 00:09:59.616000000	0.0	95.999999	461.0	0.72	3.61	94.6	5.99	0.0

3 rows × 50 columns

That's better. Reading through the Readme in the description of the original dataset, we can identify the variables that are available going forward. Since only those are useful as external covariates

potentially helpful with our forecast, we can subset the dataframe:

In [49]:
```
# timestamp and target variable
list1 = ['time', 'Tair']

# candidate variables for external regressors improving the forecast
list2 = ['co2_sp', 'dx_sp', 't_rail_min_sp', 't_grow_min_sp',
        'assim_sp', 'scr_enrg_sp', 'scr_blck_sp', 't_heat_sp',
        't_vent_sp', 'window_pos_lee_sp', 'water_sup_intervals_sp_min',
        'int_blue_sp', 'int_red_sp', 'int_farred_sp',
        'int_white_sp']

xdat = xdat[list1 + list2]
```

A realistic lookahead period might be 24 hours - we can either stick to the original resolution of the data and forecast 24 * 12 = 288 steps ahead, or downsample the data to hourly frequency. Since all the variables are numerical, we just take their average values within each hour:

In [50]:
```
xdat2 = xdat.resample('H', on = 'time').mean().reset_index()[list1 + list2]
```

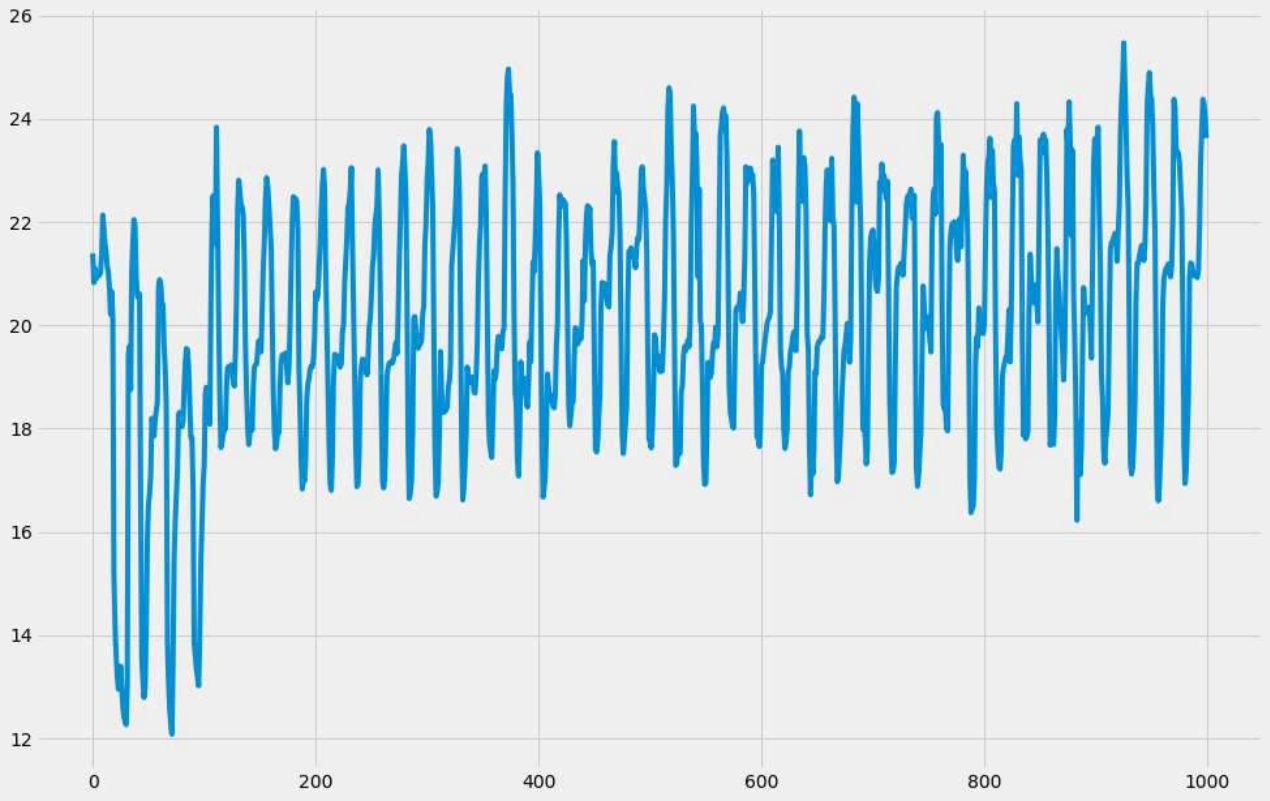
In [51]:
```
# let's quickly check for missing values - Prophet can handle those (by interpolation),
np.isnan(xdat2[['Tair'] + list2]).describe()
```

Out[51]:

	Tair	co2_sp	dx_sp	t_rail_min_sp	t_grow_min_sp	assim_sp	scr_enrg_sp	scr_blck_sp	t_heat_sp
count	3985	3985	3985	3985	3985	3985	3985	3985	3985
unique	2	2	2	2	2	2	2	2	2
top	False	False	False	False	False	False	False	False	False
freq	3983	3945	3944	3944	3878	3944	3944	3944	3945

In [52]:
```
xdat2.dropna(subset = ['Tair'], inplace = True)

# what does our data look like after preparation?
xdat2['Tair'][0:1000].plot()
```

Out[52]: <AxesSubplot:>

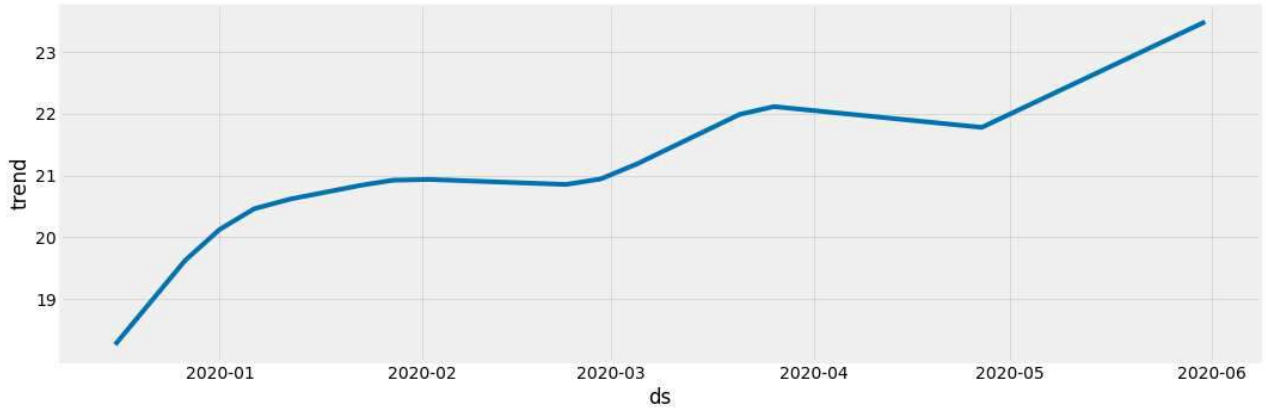


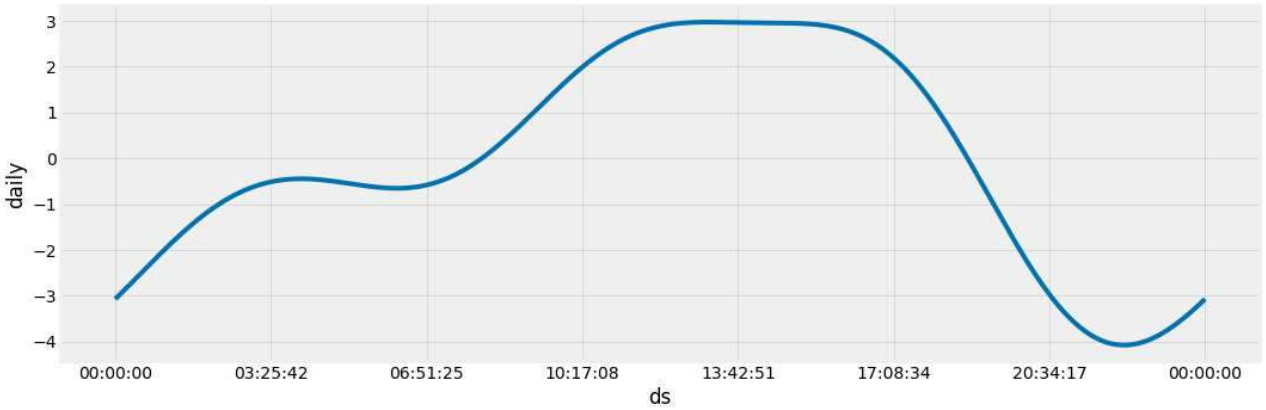
This looks ready for applying Prophet: as before, we start by formatting the dataframe into the format expected by the algorithm

In [53]:
```
df = xdat2[['time', 'Tair']].rename(columns={"time": "ds", "Tair": "y"})
```

The only real adjustment we need is disabling weekly seasonality. By default, Prophet fits any seasonal pattern with enough data - annual pattern will not be fitted since we only have 6 months worth of data, hourly is automatic, but weekly makes no sense for a physical phenomenon (there is no reason to expect temperature to be dependent on day of the week).

In [54]:
```
m = Prophet(weekly_seasonality=False, interval_width = 0.95)
m.fit(df)

# build the forecast the usual way
future = m.make_future_dataframe(periods= 24, freq = 'H')
forecast = m.predict(future)
m.plot_components(forecast, figsize=(CFG.img_dim1, CFG.img_dim2))
print()
```





In [55]:

```
# Calculate the cross-validation performance
df_cv = cross_validation(m,initial = '3700 hours', period = '24 hours', horizon = '24 h
df_p1 = performance_metrics(df_cv)
df_p1.head(10)
```

Out[55]:

	horizon	mse	rmse	mae	mape	mdape	coverage
0	0 days 03:00:00	4.345279	2.084533	1.871085	0.097015	0.098113	0.846154
1	0 days 04:00:00	5.201480	2.280675	2.071381	0.105271	0.101704	0.779720
2	0 days 05:00:00	5.909249	2.430895	2.219272	0.111466	0.115505	0.741259
3	0 days 06:00:00	6.260671	2.502133	2.292782	0.114881	0.115744	0.727273
4	0 days 07:00:00	5.446089	2.333686	2.121845	0.105537	0.115690	0.765734
5	0 days 08:00:00	3.123262	1.767275	1.479378	0.071383	0.052611	0.881119
6	0 days 09:00:00	1.801253	1.342108	1.079299	0.047787	0.038968	0.972028
7	0 days 10:00:00	2.161790	1.470303	1.204742	0.048926	0.042421	0.961538
8	0 days 11:00:00	2.999885	1.732018	1.462740	0.057838	0.045617	0.923077
9	0 days 12:00:00	3.535510	1.880295	1.535825	0.060383	0.050830	0.870629

This gives us an idea of the model performance in a "vanilla" version. Let's add the covariates:

In [56]:

```
# we need to drop NA - Prophet can handle missing values in the target, but not in the
# (and neither does the recursive feature elimination routine RFECV)
```

```
xdat2 = xdat2.dropna()

# for the sake of demonstration, we keep it simple - so Ridge is good enough to check t
estimator = Ridge()
# the elimination is a bit agressive :-)
selector =  RFECV(estimator, step=10, cv=10)
selector = selector.fit(xdat2[list2], xdat2['Tair'])
#
to_keep = xdat2[list2].columns[selector.support_]


print(to_keep)
```

```
Index(['dx_sp', 't_rail_min_sp', 't_heat_sp', 't_vent_sp',
       'window_pos_lee_sp'],
      dtype='object')
```

In [57]:

```
# We can now fit our model with the regressors included
df = xdat2[['time', 'Tair']].rename(columns={"time": "ds", "Tair": "y"})
m = Prophet(weekly_seasonality=False, interval_width = 0.95)

# add the regressors to the dataframe holding the data
for f in to_keep:
    df[f] = xdat2[f]
    m.add_regressor(f)

# the rest proceeds as before.
m.fit(df)
```

Out[57]: <fbprophet.forecaster.Prophet at 0x7f6e3675f190>

We want to compare the "vanilla" and extended models, to we conduct the validation in exact same manner

In [58]:

```
# we repeat the same evaluation tactic as before
df_cv = cross_validation(m, initial = '3700 hours', period = '24 hours', horizon='24 ho
df_p2 = performance_metrics(df_cv)
df_p2.head(3)
```

Out[58]:

	horizon	mse	rmse	mae	mape	mdape	coverage
0	0 days 03:00:00	1.894753	1.376500	1.011249	0.054225	0.028329	0.625000
1	0 days 04:00:00	2.378468	1.542228	1.124816	0.058973	0.021894	0.571429
2	0 days 05:00:00	2.602638	1.613269	1.246161	0.064392	0.065202	0.571429

How do the two models compare ?

In [59]:

```
comparison = pd.DataFrame()
comparison['raw'] = df_p1.mean(axis = 0)[1:]
comparison['covariates'] = df_p2.mean(axis = 0)[1:]

print(comparison)
```

```
                raw covariates
```

```
mse       9.387352   1.100758
rmse      2.803511   0.988497
mae       2.340932    0.73765
mape      0.095295   0.033785
mdape     0.086309   0.023047
coverage  0.709631   0.852679
```

As you can see from the above table, with minimal effort of adding the covariates we can improve the performance of the model wrt all the relevant metrics.

Neural Prophet

If an algorithmic approach works well in ML, the question worth asking is frequently: will it get better with Deep Learning? The answer, surprisingly often ;-) is affirmative - and that was the core idea behind Neural Prophet. The changes compared to the "basic" Prophet are given on the project website https://neuralprophet.com/html/index.html:

- Gradient Descent for optimisation via using PyTorch as the backend.
- Modelling autocorrelation of time series using AR-Net
- Modelling lagged regressors using a sepearate Feed-Forward Neural Network.
- Configurable non-linear deep layers of the FFNNs.
- Tuneable to specific forecast horizons (greater than 1).
- Custom losses and metrics.

The core addition from a conceptual point of view seems to be the AR-Net - for those interested, the original paper can be found here: https://www.arxiv-vanity.com/papers/1911.12436/.

Is Neural Prophet really better? As the Russian proverb says "trust, but verify", so let's find out! The first thing we need to do is actually install Neural Prophet - as of the time of writing this notebook, the package is not part of the Kaggle Python image.

In [60]:
```
!pip install neuralprophet
```

```
Collecting neuralprophet
  Downloading neuralprophet-0.3.2-py3-none-any.whl (74 kB)
     |████████████████████████████████| 74 kB 1.5 MB/s
Requirement already satisfied: LunarCalendar>=0.0.9 in /opt/conda/lib/python3.7/site-pac
kages (from neuralprophet) (0.0.9)
Requirement already satisfied: numpy>=1.15.4 in /opt/conda/lib/python3.7/site-packages
(from neuralprophet) (1.19.5)
Requirement already satisfied: python-dateutil>=2.8.0 in /opt/conda/lib/python3.7/site-p
ackages (from neuralprophet) (2.8.1)
Collecting holidays>=0.11.3.1
  Downloading holidays-0.16-py3-none-any.whl (184 kB)
     |████████████████████████████████| 184 kB 9.0 MB/s
Requirement already satisfied: ipywidgets>=7.5.1 in /opt/conda/lib/python3.7/site-packag
es (from neuralprophet) (7.6.3)
Requirement already satisfied: matplotlib>=2.0.0 in /opt/conda/lib/python3.7/site-packag
```

```
es (from neuralprophet) (3.4.2)
Requirement already satisfied: pandas>=1.0.4 in /opt/conda/lib/python3.7/site-packages
(from neuralprophet) (1.2.4)
Requirement already satisfied: convertdate>=2.1.2 in /opt/conda/lib/python3.7/site-packa
ges (from neuralprophet) (2.3.2)
Requirement already satisfied: torch>=1.4.0 in /opt/conda/lib/python3.7/site-packages (f
rom neuralprophet) (1.7.0)
Requirement already satisfied: tqdm>=4.50.2 in /opt/conda/lib/python3.7/site-packages (f
rom neuralprophet) (4.61.1)
Collecting torch-lr-finder>=0.2.1
  Downloading torch_lr_finder-0.2.1-py3-none-any.whl (11 kB)
Requirement already satisfied: dataclasses>=0.6 in /opt/conda/lib/python3.7/site-package
s (from neuralprophet) (0.6)
Requirement already satisfied: pymeeus<=1,>=0.3.13 in /opt/conda/lib/python3.7/site-pack
ages (from convertdate>=2.1.2->neuralprophet) (0.5.11)
Requirement already satisfied: pytz>=2014.10 in /opt/conda/lib/python3.7/site-packages
(from convertdate>=2.1.2->neuralprophet) (2021.1)
Requirement already satisfied: korean-lunar-calendar in /opt/conda/lib/python3.7/site-pa
ckages (from holidays>=0.11.3.1->neuralprophet) (0.2.1)
Requirement already satisfied: hijri-converter in /opt/conda/lib/python3.7/site-packages
(from holidays>=0.11.3.1->neuralprophet) (2.1.3)
Requirement already satisfied: ipykernel>=4.5.1 in /opt/conda/lib/python3.7/site-package
s (from ipywidgets>=7.5.1->neuralprophet) (5.5.5)
Requirement already satisfied: nbformat>=4.2.0 in /opt/conda/lib/python3.7/site-packages
(from ipywidgets>=7.5.1->neuralprophet) (5.1.3)
Requirement already satisfied: traitlets>=4.3.1 in /opt/conda/lib/python3.7/site-package
s (from ipywidgets>=7.5.1->neuralprophet) (5.0.5)
Requirement already satisfied: widgetsnbextension~=3.5.0 in /opt/conda/lib/python3.7/sit
e-packages (from ipywidgets>=7.5.1->neuralprophet) (3.5.1)
Requirement already satisfied: jupyterlab-widgets>=1.0.0 in /opt/conda/lib/python3.7/sit
e-packages (from ipywidgets>=7.5.1->neuralprophet) (1.0.0)
Requirement already satisfied: ipython>=4.0.0 in /opt/conda/lib/python3.7/site-packages
(from ipywidgets>=7.5.1->neuralprophet) (7.24.1)
Requirement already satisfied: jupyter-client in /opt/conda/lib/python3.7/site-packages
(from ipykernel>=4.5.1->ipywidgets>=7.5.1->neuralprophet) (6.1.12)
Requirement already satisfied: tornado>=4.2 in /opt/conda/lib/python3.7/site-packages (f
rom ipykernel>=4.5.1->ipywidgets>=7.5.1->neuralprophet) (6.1)
Requirement already satisfied: matplotlib-inline in /opt/conda/lib/python3.7/site-packag
es (from ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (0.1.2)
Requirement already satisfied: decorator in /opt/conda/lib/python3.7/site-packages (from
ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (5.0.9)
Requirement already satisfied: jedi>=0.16 in /opt/conda/lib/python3.7/site-packages (fro
m ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (0.18.0)
Requirement already satisfied: pickleshare in /opt/conda/lib/python3.7/site-packages (fr
om ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (0.7.5)
Requirement already satisfied: pexpect>4.3 in /opt/conda/lib/python3.7/site-packages (fr
om ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (4.8.0)
Requirement already satisfied: pygments in /opt/conda/lib/python3.7/site-packages (from
ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (2.9.0)
Requirement already satisfied: backcall in /opt/conda/lib/python3.7/site-packages (from
ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (0.2.0)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /opt/cond
a/lib/python3.7/site-packages (from ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet)
(3.0.19)
Requirement already satisfied: setuptools>=18.5 in /opt/conda/lib/python3.7/site-package
s (from ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (49.6.0.post20210108)
Requirement already satisfied: parso<0.9.0,>=0.8.0 in /opt/conda/lib/python3.7/site-pack
ages (from jedi>=0.16->ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (0.8.2)
Requirement already satisfied: ephem>=3.7.5.3 in /opt/conda/lib/python3.7/site-packages
(from LunarCalendar>=0.0.9->neuralprophet) (4.0.0.2)
Requirement already satisfied: cycler>=0.10 in /opt/conda/lib/python3.7/site-packages (f
rom matplotlib>=2.0.0->neuralprophet) (0.10.0)
Requirement already satisfied: pyparsing>=2.2.1 in /opt/conda/lib/python3.7/site-package
s (from matplotlib>=2.0.0->neuralprophet) (2.4.7)
Requirement already satisfied: kiwisolver>=1.0.1 in /opt/conda/lib/python3.7/site-packag
```

```
es (from matplotlib>=2.0.0->neuralprophet) (1.3.1)
Requirement already satisfied: pillow>=6.2.0 in /opt/conda/lib/python3.7/site-packages
(from matplotlib>=2.0.0->neuralprophet) (8.2.0)
Requirement already satisfied: six in /opt/conda/lib/python3.7/site-packages (from cycle
r>=0.10->matplotlib>=2.0.0->neuralprophet) (1.15.0)
Requirement already satisfied: jsonschema!=2.5.0,>=2.4 in /opt/conda/lib/python3.7/site-
packages (from nbformat>=4.2.0->ipywidgets>=7.5.1->neuralprophet) (3.2.0)
Requirement already satisfied: ipython-genutils in /opt/conda/lib/python3.7/site-package
s (from nbformat>=4.2.0->ipywidgets>=7.5.1->neuralprophet) (0.2.0)
Requirement already satisfied: jupyter-core in /opt/conda/lib/python3.7/site-packages (f
rom nbformat>=4.2.0->ipywidgets>=7.5.1->neuralprophet) (4.7.1)
Requirement already satisfied: attrs>=17.4.0 in /opt/conda/lib/python3.7/site-packages
(from jsonschema!=2.5.0,>=2.4->nbformat>=4.2.0->ipywidgets>=7.5.1->neuralprophet) (21.2.
0)
Requirement already satisfied: importlib-metadata in /opt/conda/lib/python3.7/site-packa
ges (from jsonschema!=2.5.0,>=2.4->nbformat>=4.2.0->ipywidgets>=7.5.1->neuralprophet)
(3.4.0)
Requirement already satisfied: pyrsistent>=0.14.0 in /opt/conda/lib/python3.7/site-packa
ges (from jsonschema!=2.5.0,>=2.4->nbformat>=4.2.0->ipywidgets>=7.5.1->neuralprophet)
(0.17.3)
Requirement already satisfied: ptyprocess>=0.5 in /opt/conda/lib/python3.7/site-packages
(from pexpect>4.3->ipython>=4.0.0->ipywidgets>=7.5.1->neuralprophet) (0.7.0)
Requirement already satisfied: wcwidth in /opt/conda/lib/python3.7/site-packages (from p
rompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0->ipython>=4.0.0->ipywidgets>=7.5.1->neuralpr
ophet) (0.2.5)
Requirement already satisfied: future in /opt/conda/lib/python3.7/site-packages (from to
rch>=1.4.0->neuralprophet) (0.18.2)
Requirement already satisfied: typing_extensions in /opt/conda/lib/python3.7/site-packag
es (from torch>=1.4.0->neuralprophet) (3.7.4.3)
Requirement already satisfied: packaging in /opt/conda/lib/python3.7/site-packages (from
torch-lr-finder>=0.2.1->neuralprophet) (20.9)
Requirement already satisfied: notebook>=4.4.1 in /opt/conda/lib/python3.7/site-packages
(from widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet) (6.4.0)
Requirement already satisfied: Send2Trash>=1.5.0 in /opt/conda/lib/python3.7/site-packag
es (from notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet)
(1.5.0)
Requirement already satisfied: pyzmq>=17 in /opt/conda/lib/python3.7/site-packages (from
notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet) (22.1.0)
Requirement already satisfied: terminado>=0.8.3 in /opt/conda/lib/python3.7/site-package
s (from notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet)
(0.10.1)
Requirement already satisfied: prometheus-client in /opt/conda/lib/python3.7/site-packag
es (from notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet)
(0.11.0)
Requirement already satisfied: nbconvert in /opt/conda/lib/python3.7/site-packages (from
notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet) (6.0.7)
Requirement already satisfied: argon2-cffi in /opt/conda/lib/python3.7/site-packages (fr
om notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet) (20.1.
0)
Requirement already satisfied: jinja2 in /opt/conda/lib/python3.7/site-packages (from no
tebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet) (3.0.1)
Requirement already satisfied: cffi>=1.0.0 in /opt/conda/lib/python3.7/site-packages (fr
om argon2-cffi->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralpro
phet) (1.14.5)
Requirement already satisfied: pycparser in /opt/conda/lib/python3.7/site-packages (from
cffi>=1.0.0->argon2-cffi->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1-
>neuralprophet) (2.20)
Requirement already satisfied: zipp>=0.5 in /opt/conda/lib/python3.7/site-packages (from
importlib-metadata->jsonschema!=2.5.0,>=2.4->nbformat>=4.2.0->ipywidgets>=7.5.1->neuralp
rophet) (3.4.1)
Requirement already satisfied: MarkupSafe>=2.0 in /opt/conda/lib/python3.7/site-packages
(from jinja2->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralproph
et) (2.0.1)
Requirement already satisfied: mistune<2,>=0.8.1 in /opt/conda/lib/python3.7/site-packag
es (from nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neura
```

```
lprophet) (0.8.4)
Requirement already satisfied: testpath in /opt/conda/lib/python3.7/site-packages (from
nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet)
(0.5.0)
Requirement already satisfied: bleach in /opt/conda/lib/python3.7/site-packages (from nb
convert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophet)
(3.3.0)
Requirement already satisfied: entrypoints>=0.2.2 in /opt/conda/lib/python3.7/site-packa
ges (from nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neur
alprophet) (0.3)
Requirement already satisfied: pandocfilters>=1.4.1 in /opt/conda/lib/python3.7/site-pac
kages (from nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->ne
uralprophet) (1.4.2)
Requirement already satisfied: jupyterlab-pygments in /opt/conda/lib/python3.7/site-pack
ages (from nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neu
ralprophet) (0.1.2)
Requirement already satisfied: defusedxml in /opt/conda/lib/python3.7/site-packages (fro
m nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->neuralprophe
t) (0.7.1)
Requirement already satisfied: nbclient<0.6.0,>=0.5.0 in /opt/conda/lib/python3.7/site-p
ackages (from nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->
neuralprophet) (0.5.3)
Requirement already satisfied: async-generator in /opt/conda/lib/python3.7/site-packages
(from nbclient<0.6.0,>=0.5.0->nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipy
widgets>=7.5.1->neuralprophet) (1.10)
Requirement already satisfied: nest-asyncio in /opt/conda/lib/python3.7/site-packages (f
rom nbclient<0.6.0,>=0.5.0->nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywi
dgets>=7.5.1->neuralprophet) (1.5.1)
Requirement already satisfied: webencodings in /opt/conda/lib/python3.7/site-packages (f
rom bleach->nbconvert->notebook>=4.4.1->widgetsnbextension~=3.5.0->ipywidgets>=7.5.1->ne
uralprophet) (0.5.1)
Installing collected packages: torch-lr-finder, holidays, neuralprophet
  Attempting uninstall: holidays
    Found existing installation: holidays 0.11.1
    Uninstalling holidays-0.11.1:
      Successfully uninstalled holidays-0.11.1
Successfully installed holidays-0.16 neuralprophet-0.3.2 torch-lr-finder-0.2.1
WARNING: Running pip as root will break packages and permissions. You should install pac
kages reliably by using venv: https://pip.pypa.io/warnings/venv
```

In honor of the classic Dead Kennedys song, we will use the data on energy consumption in the U.S. state of California:

In [61]:

```
# a necessary dependency for working with xlsx files in pandas
!pip install openpyxl
```

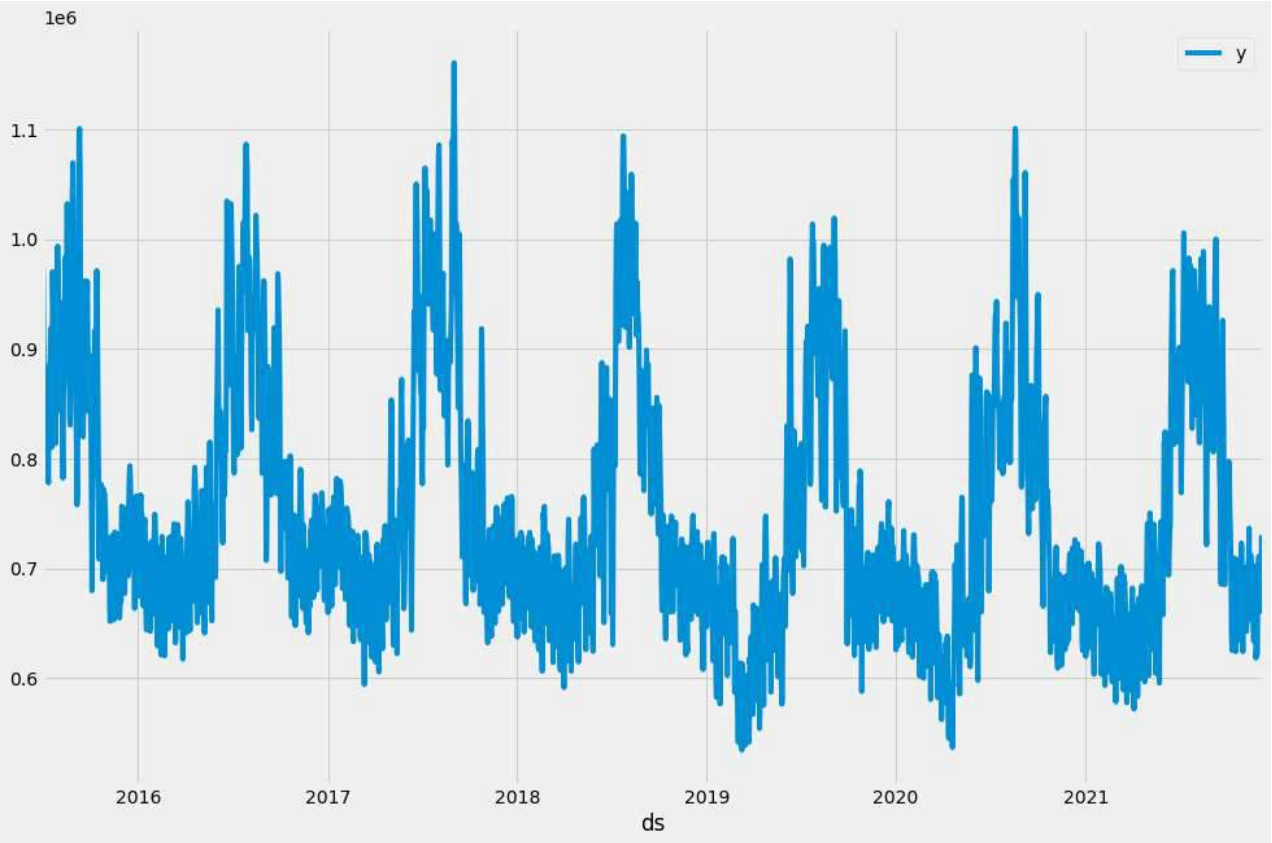
```
Collecting openpyxl
  Downloading openpyxl-3.0.10-py2.py3-none-any.whl (242 kB)
     |████████████████████████████████| 242 kB 4.1 MB/s
Collecting et-xmlfile
  Downloading et_xmlfile-1.1.0-py3-none-any.whl (4.7 kB)
Installing collected packages: et-xmlfile, openpyxl
Successfully installed et-xmlfile-1.1.0 openpyxl-3.0.10
WARNING: Running pip as root will break packages and permissions. You should install pac
kages reliably by using venv: https://pip.pypa.io/warnings/venv
```

In [62]:

```
# read the raw *daily* data
df = pd.read_excel('../input/us-energy-consumption-eia/Region_CAL.xlsx', sheet_name='Pu
                   usecols = ['Local date', 'D'])
# format to Prophet convention - both Prophet and Neural Prophet adhere to it
df.columns = ['ds','y']
df['ds'] = pd.to_datetime(df['ds'])
```

```
df.plot(x="ds",y="y")
```

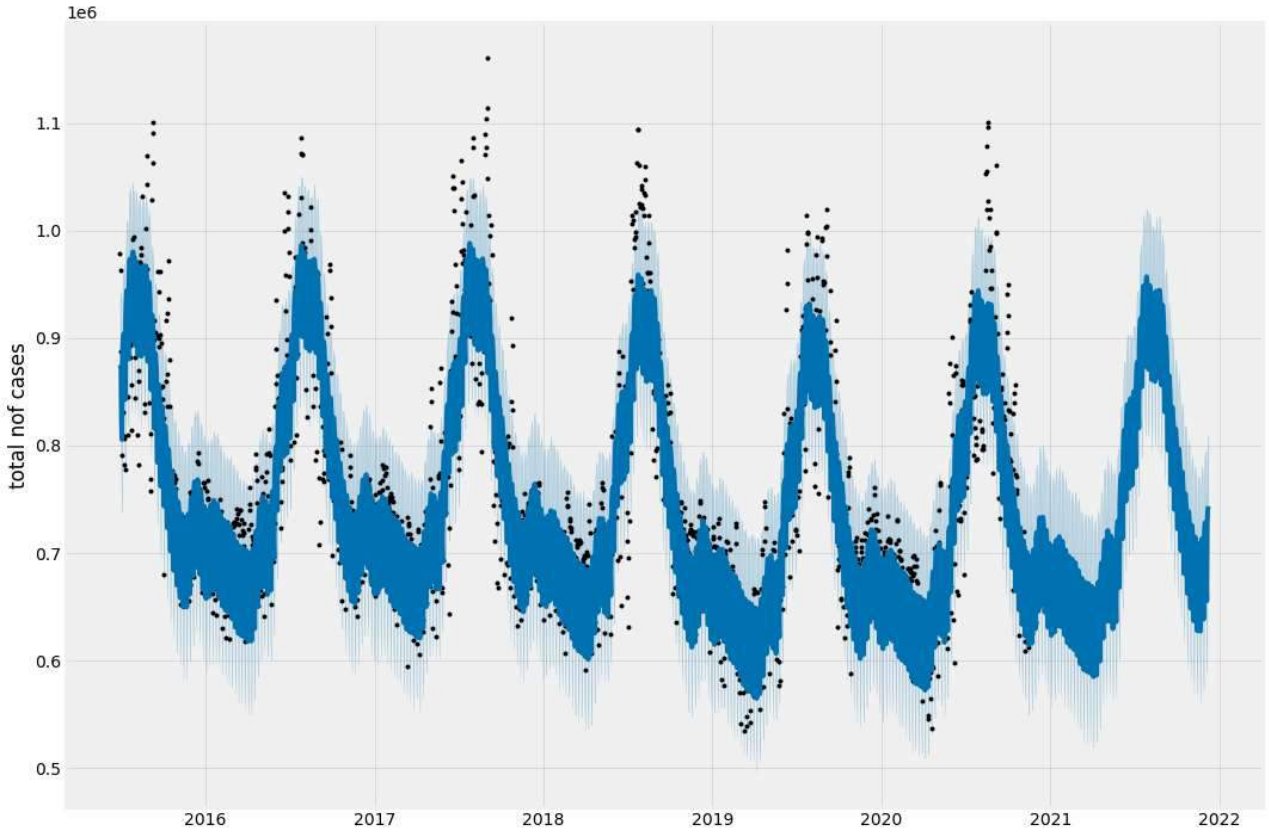
Out[62]: <AxesSubplot:xlabel='ds'>



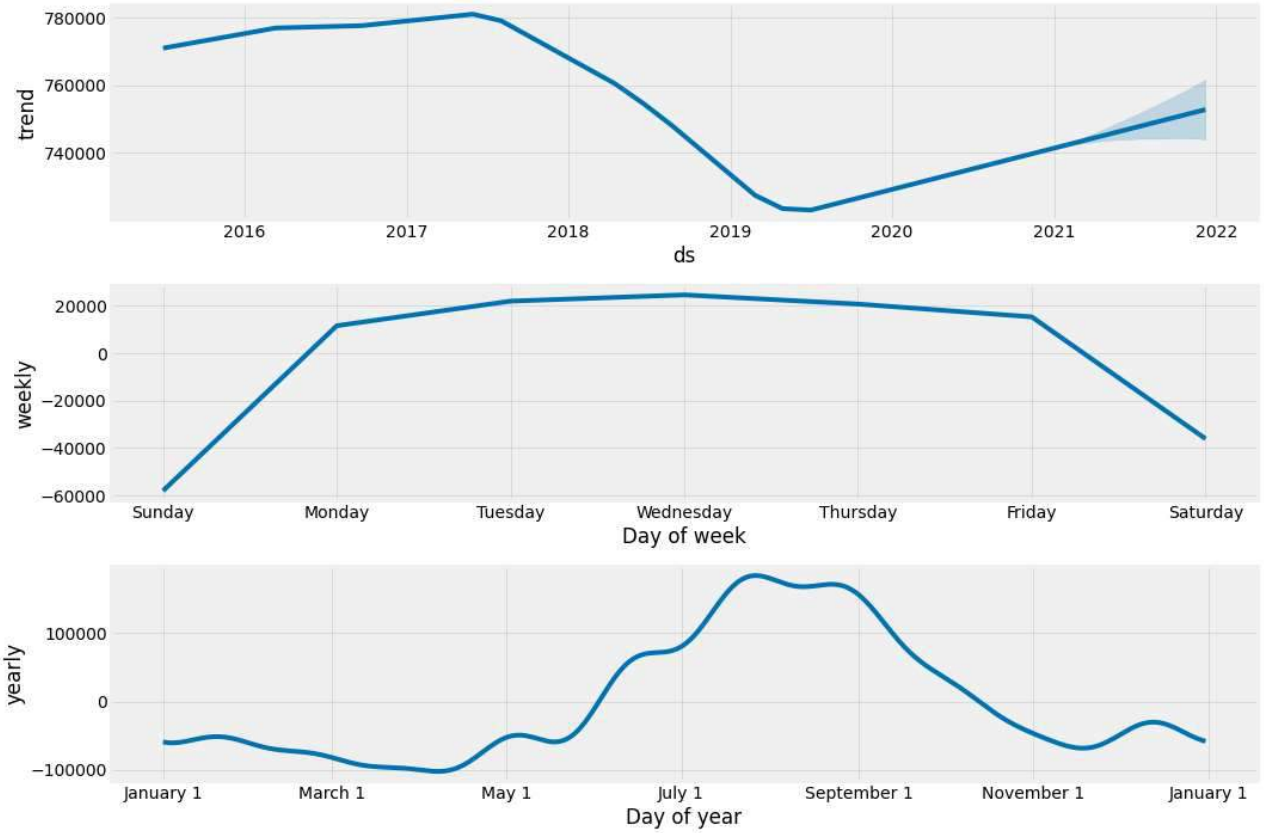
In [63]:
```
# We split the data into training / validation (= last 365 days)
df_train, df_valid = df.iloc[:-365], df.iloc[-365:]
```

Fitting the basic Prophet model and generating a forecast is a familiar routine by now:

In [64]:
```
m_vanilla = Prophet()
m_vanilla.fit(df_train)

future = m_vanilla.make_future_dataframe(periods=365)
forecast_vanilla = m_vanilla.predict(future)

m_vanilla.plot(forecast_vanilla, figsize=(CFG.img_dim1, CFG.img_dim2), xlabel = '', yla
print()
```



In [65]:
```
m_vanilla.plot_components(forecast_vanilla, figsize=(CFG.img_dim1, CFG.img_dim2)); prin
```



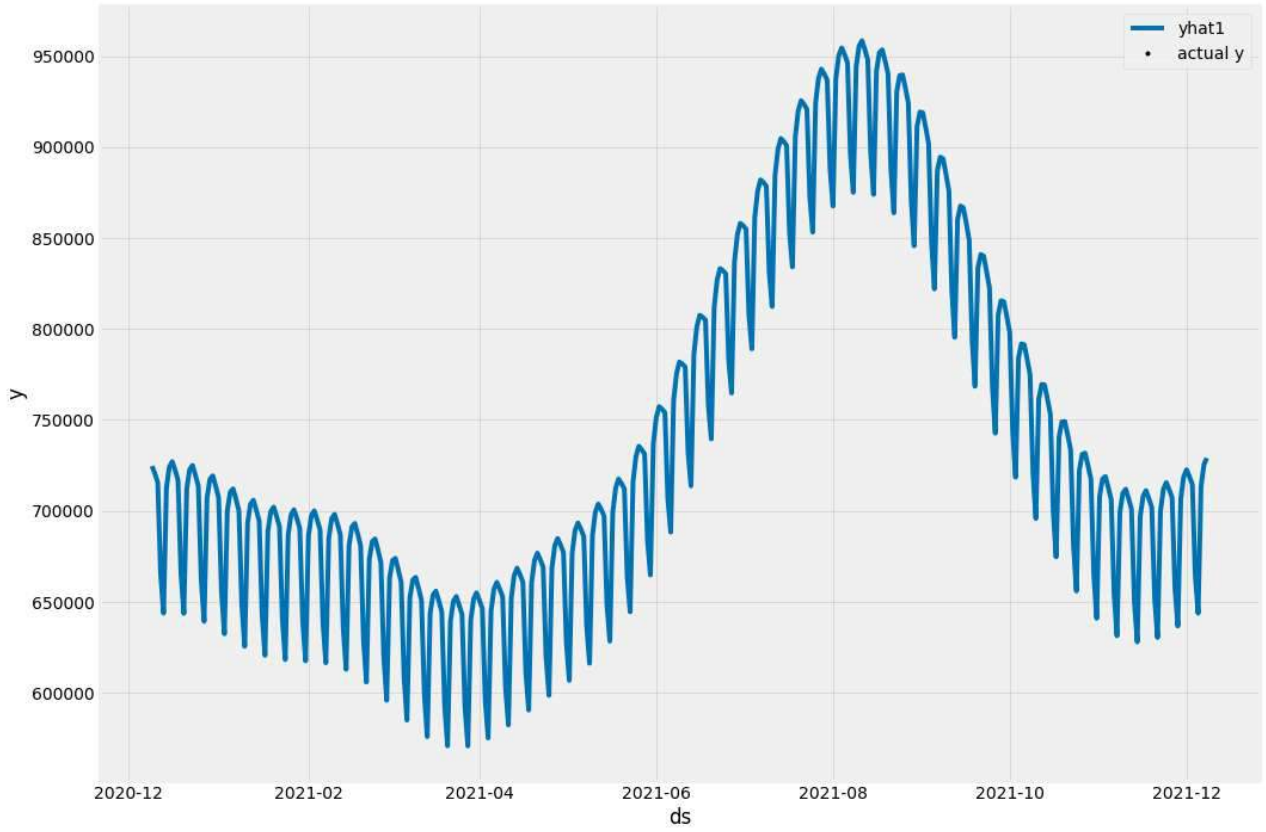
Neural Prophet requires minimal adjustments to the syntax:

In [66]:
```
from neuralprophet import NeuralProphet
pd.options.mode.chained_assignment = None


m_neural = NeuralProphet()

metrics = m_neural.fit(df_train, freq="D")
```

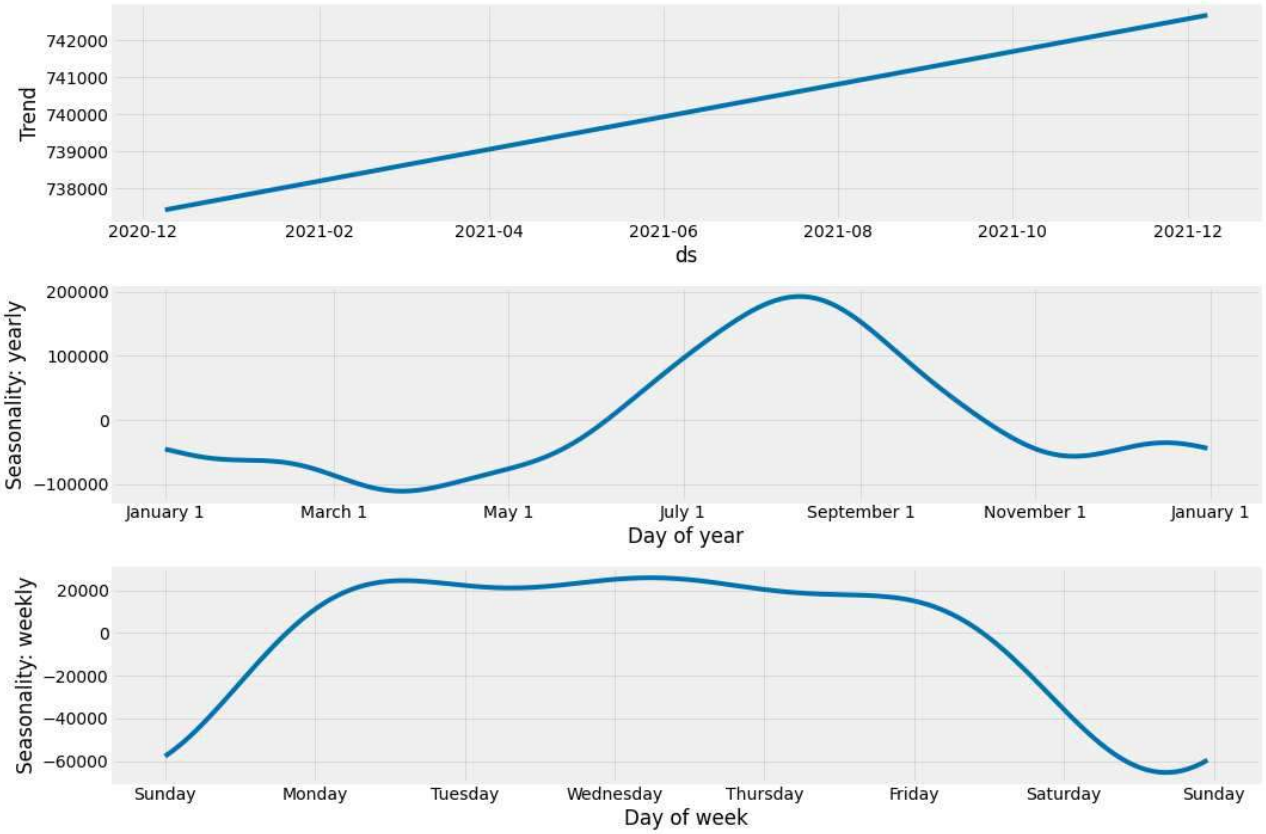
```
INFO - (NP.df_utils._infer_frequency) - Major frequency D corresponds to 99.95% of the d
ata.
INFO - (NP.df_utils._infer_frequency) - Defined frequency is equal to major frequency -
D
INFO - (NP.config.init_data_params) - Setting normalization to global as only one datafr
ame provided for training.
INFO - (NP.utils.set_auto_seasonalities) - Disabling daily seasonality. Run NeuralProphe
t with daily_seasonality=True to override this.
INFO - (NP.config.set_auto_batch_epoch) - Auto-set batch_size to 32
INFO - (NP.config.set_auto_batch_epoch) - Auto-set epochs to 158
WARNING - (py.warnings._showwarnmsg) - /opt/conda/lib/python3.7/site-packages/torch/nn/m
odules/container.py:550: UserWarning:

Setting attributes on ParameterDict is not supported.



INFO - (NP.utils_torch.lr_range_test) - lr-range-test results: steep: 8.10E-02, min: 1.7
9E-01

INFO - (NP.utils_torch.lr_range_test) - lr-range-test results: steep: 8.10E-02, min: 1.7
9E-01
INFO - (NP.forecaster._init_train_loader) - lr-range-test selected learning rate: 8.69E-
02
Epoch[158/158]: 100%|██████████| 158/158 [00:25<00:00,  6.24it/s, SmoothL1Loss=0.00636,
MAE=3.6e+4, RMSE=4.94e+4, RegLoss=0]
```

In [67]:
```
df_future = m_neural.make_future_dataframe(df_train, periods=365)
forecast_neural = m_neural.predict(df_future)
# create plots
fig_forecast = m_neural.plot(forecast_neural, figsize=(CFG.img_dim1, CFG.img_dim2))
```

```
INFO - (NP.df_utils._infer_frequency) - Major frequency D corresponds to 99.95% of the d
ata.
INFO - (NP.df_utils._infer_frequency) - Defined frequency is equal to major frequency -
D
INFO - (NP.df_utils._infer_frequency) - Major frequency D corresponds to 99.726% of the
data.
INFO - (NP.df_utils._infer_frequency) - Defined frequency is equal to major frequency -
D
INFO - (NP.df_utils._infer_frequency) - Major frequency D corresponds to 99.726% of the
data.
INFO - (NP.df_utils._infer_frequency) - Defined frequency is equal to major frequency -
D
```



In [68]:
```
fig_components = m_neural.plot_components(forecast_neural, figsize=(CFG.img_dim1, CFG.i
```



What about the actual predicted values?

In [69]:
```
df_valid['forecast_vanilla'] = forecast_vanilla.yhat.tail(365).values
```

```
df_valid['forecast_neural'] = forecast_neural.yhat1.values
```

In [70]:

```
from sklearn.metrics import mean_squared_error as mse, median_absolute_error as mae

def my_mse(x,y):
    return np.round(np.sqrt(mse(x, y)),2)

def my_mae(x,y):
    return np.round(np.sqrt(mae(x, y)),2)
```

In [71]:

```
df_valid.dropna(inplace = True)


err1 = my_mse(df_valid['y'], df_valid['forecast_vanilla'])
err2 = my_mse(df_valid['y'], df_valid['forecast_neural'])

print('MSE vanilla: ' + str(err1))
print('MSE neural : ' + str(err2))

err1 = my_mae(df_valid['y'], df_valid['forecast_vanilla'])
err2 = my_mae(df_valid['y'], df_valid['forecast_neural'])

print('    ')
print('MAE vanilla: ' + str(err1))
print('MAE neural : ' + str(err2))
```

```
MSE vanilla: 43542.97
MSE neural : 43849.03

MAE vanilla: 141.39
MAE neural : 141.32
```

As we can see above, even with default settings (no lags, only the AR-Net used to model serial dependence) the Neural Prophet can outperform the basic version.

This concludes the notebook dedicated to Prophet - next, we will have a look at a framework that integrates multiple models encountered so far. Stay tuned!