Prophet

インストール

上記サイト参照．PyStanのインストールにはC言語のコンパイラが必要なことに注意．

```
# To installing to mac, use:
# conda-forge: conda install -c conda-forge fbprophet
#  For Google Colab.
# !pip install fbprophet
%matplotlib inline
import logging
logging.getLogger('fbprophet').setLevel(logging.ERROR)
import warnings
warnings.filterwarnings("ignore")
```

基本的な使い方

ProphetをPythonから呼び出して使う方法は，機械学習パッケージscikit learnと同じである。Prophetクラスを生成した後、fitメソッドで学習を行い、その後でpredictメソッドで予測をする。

例としてアメリカンフットボールプレーヤのPayton ManningのWikiアクセス数のデータを用いる。

```
import pandas as pd
from fbprophet import Prophet
df = pd.read_csv('../examples/example_wp_log_peyton_manning.csv')
```

```
df.head()
```

	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

Prophetモデルのインスタンスを生成し，fitメソッドで学習（パラメータの最適化）を行う．fitメソッドに渡すのは，上で作成したデータフレームである．このとき、ds列に日付（時刻）を、y列に予測したい数値を入れておく必要がある。

```
m = Prophet()
m.fit(df)
```

`Prophet.make_future_dataframe`で未来の時刻を表すデータフレームを生成する。
既定値では、（予測で用いた）過去の時刻も含む。
ここでは、１年後（365日分）まで予測することにする。

```
future = m.make_future_dataframe(periods=365)
future.tail()
```

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

```
future.head()
```

	ds
0	2007-12-10
1	2007-12-11
2	2007-12-12
3	2007-12-13
4	2007-12-14

```
df.tail()
```

	ds	y
2900	2016-01-16	7.817223
2901	2016-01-17	9.273878
2902	2016-01-18	10.333775
2903	2016-01-19	9.125871
2904	2016-01-20	8.891374

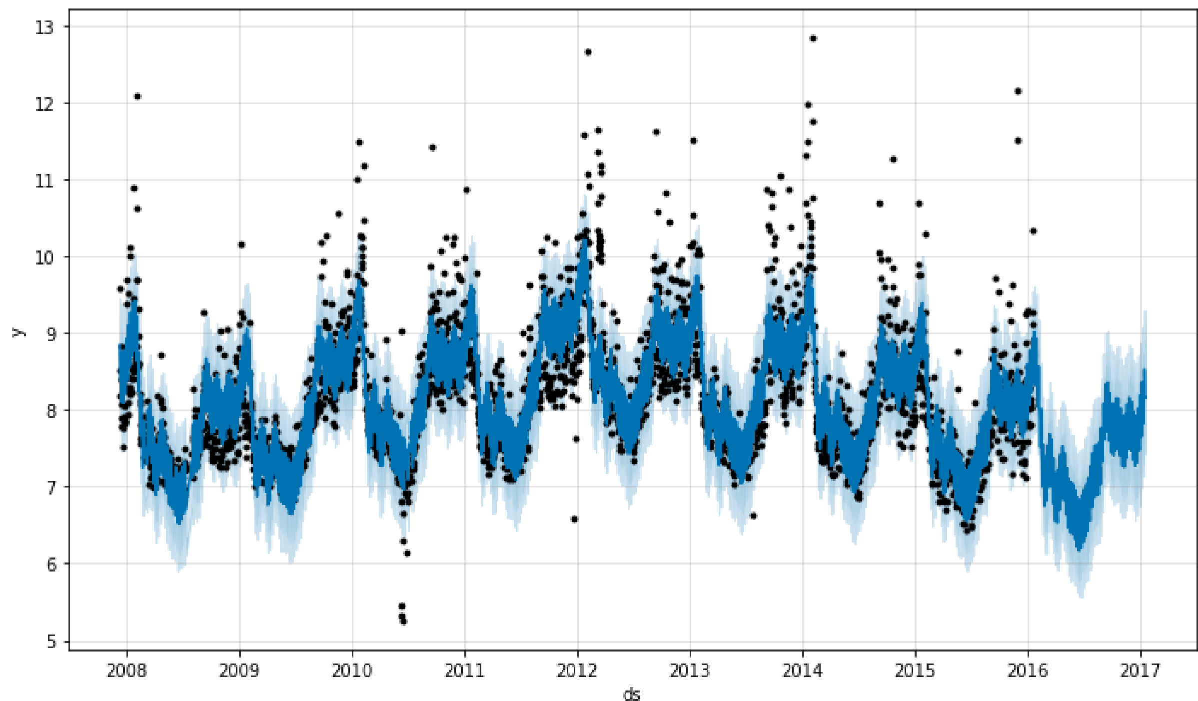
`predict` メソッドに予測したい時刻を含んだデータフレーム`future` を渡すと、予測値を入れたデータフレーム`forecast`を返す。このデータフレームは、予測値`yhat`の他に、予測の幅などの情報を含んだの列を含む。以下では，予測値`yhat`の他に，予測の上限と下限（`yhat_lower`と`yhat_upper`）を表示している．

```
forecast = m.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()
```

	ds	yhat	yhat_lower	yhat_upper
3265	2017-01-15	8.206497	7.512857	8.937173
3266	2017-01-16	8.531523	7.833361	9.303012
3267	2017-01-17	8.318930	7.607596	9.024422
3268	2017-01-18	8.151543	7.427181	8.887910
3269	2017-01-19	8.163477	7.446479	8.869469

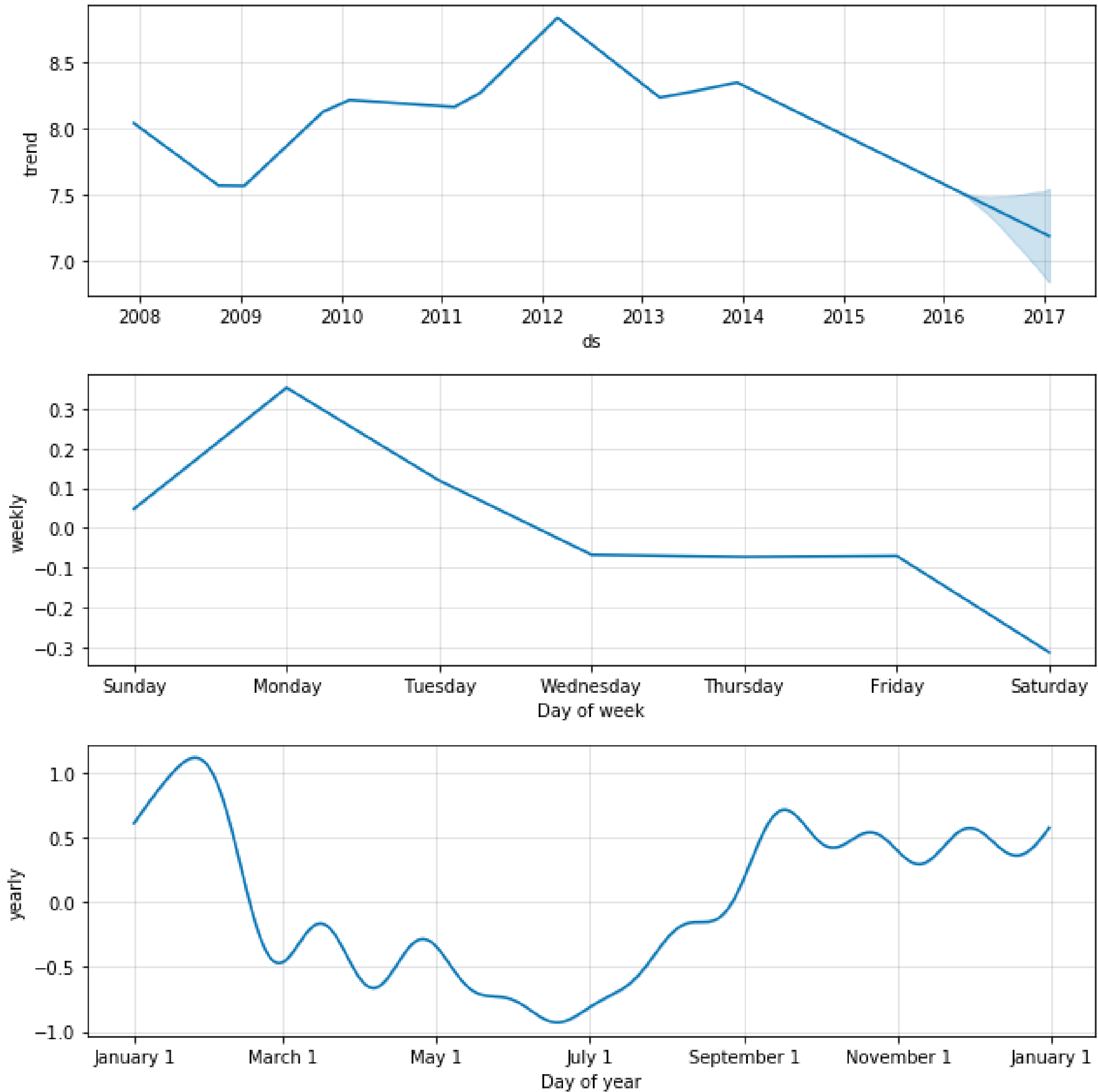
matplotlibを用いた描画は，plotメソッドで行う．

```
fig1 = m.plot(forecast)
```



予測は一般化加法モデルを用いて行われる．これは，傾向変動，季節変動，イベント情報などの様々な因子の和として予測を行う方法である．因子ごとに予測値の描画を行うには，plot_componentsメソッドを用いる．既定では，以下のように，上から順に傾向変動，週次の季節変動，年次の季節変動が描画される．また，傾向変動の図（一番上）には，予測の誤差範囲が示される．季節変動の誤差範囲を得る方法については，後述する．

```
fig2 = m.plot_components(forecast)
```



対話形式に，拡大縮小や範囲指定ができる動的な図も，Plotlyライブラリを用いて得ることができる．

```
from fbprophet.plot import plot_plotly
import plotly.offline as py
py.init_notebook_mode()
fig = plot_plotly(m, forecast)  # This returns a plotly Figure
py.iplot(fig)
```

ロジスティック曲線による予測

Prophetによる予測の既定値は線形モデルであるが、ロジスティック曲線を用いることもできる。これによって，上限や下限に漸近する時系列データの予測を行うことができる.

```
df = pd.read_csv('../examples/example_wp_log_R.csv')
```

上限を規定するためには、`cap`列に上限値（容量(capacity)の略でcap）を入力する。

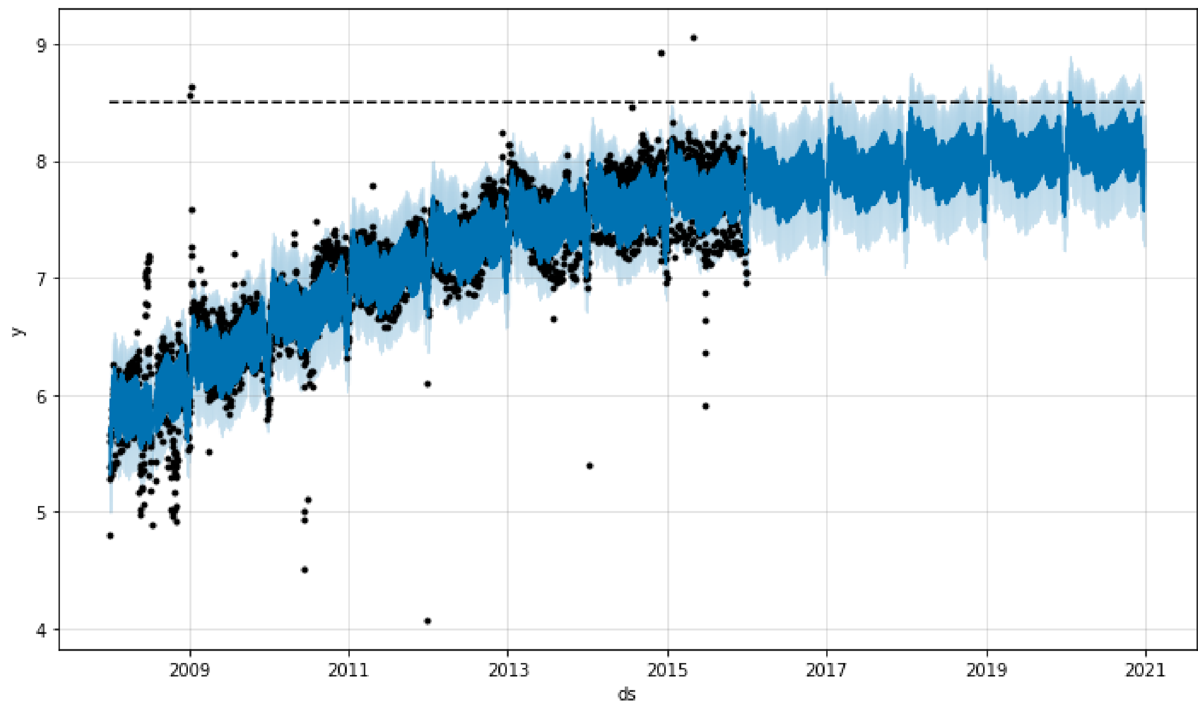
```
df['cap'] = 8.5
```

この値（容量）は行(データ）ごとに設定しなければならない.
次いで、引数`growth`を`logistic`に設定してProphetモデルを生成すると、ロジスティック曲線に当てはめを行う。

```
m = Prophet(growth='logistic')
m.fit(df)
```

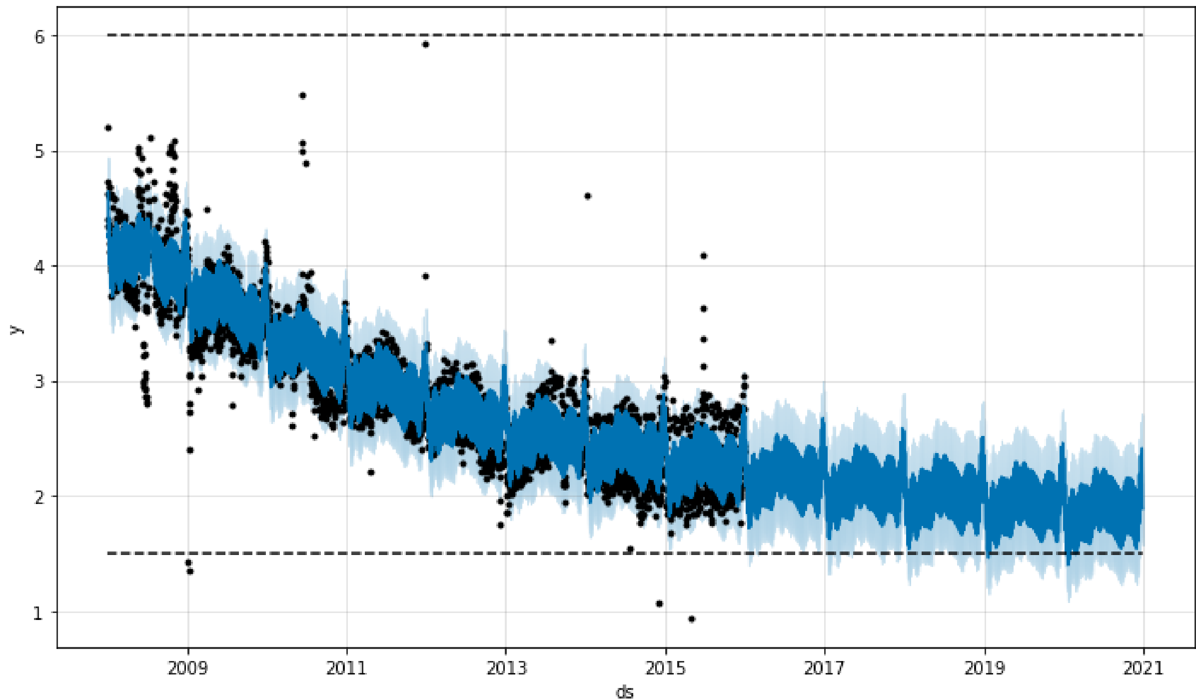
予測を行いたい日付（過去の情報を含む）を表すデータフレームを生成し，容量も設定する.

```
future = m.make_future_dataframe(periods=1826)
future['cap'] = 8.5
fcst = m.predict(future)
fig = m.plot(fcst)
```



漸近する下限値を設定する場合には、`floor`列に下限値を入力する、

```
df['y'] = 10 - df['y']
df['cap'] = 6
df['floor'] = 1.5
future['cap'] = 6
future['floor'] = 1.5
m = Prophet(growth='logistic')
m.fit(df)
fcst = m.predict(future)
fig = m.plot(fcst)
```



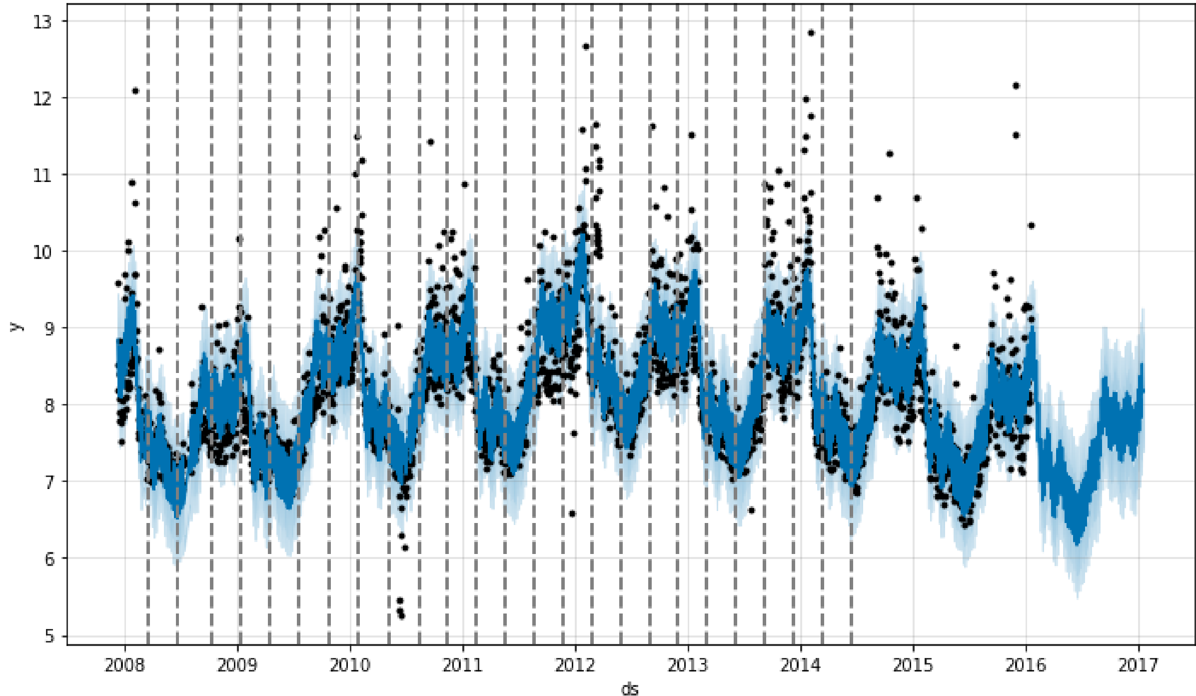
傾向変化点

「上昇トレンドの株価が，下降トレンドに移った」というニュースをよく耳にするだろう．このように，傾向変動は，時々変化すると仮定した方が自然なのだ．Prophetでは，これを傾向の変化点として処理する．再び，Peyton Manningのデータを使う．

```
df = pd.read_csv('../examples/example_wp_log_peyton_manning.csv')
```

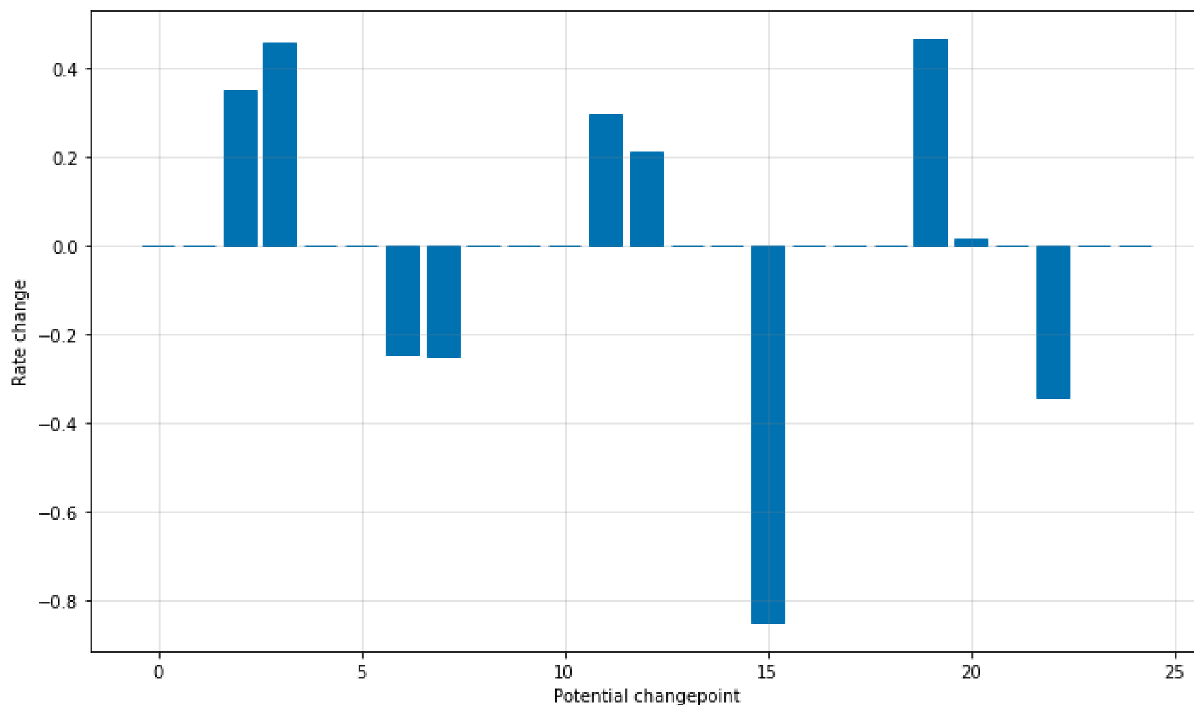
傾向変化点の候補は自動的に設定される。既定値では時系列の最初の80%の部分に均等に設定される。これは、モデルのchangepoints属性でアクセスできる。

```
m = Prophet()
m.fit(df)
future = m.make_future_dataframe(periods=366)
forecast = m.predict(future)
fig = m.plot(forecast)
for cp in m.changepoints:
    plt.axvline(cp, c='gray', ls='--', lw=2)
```



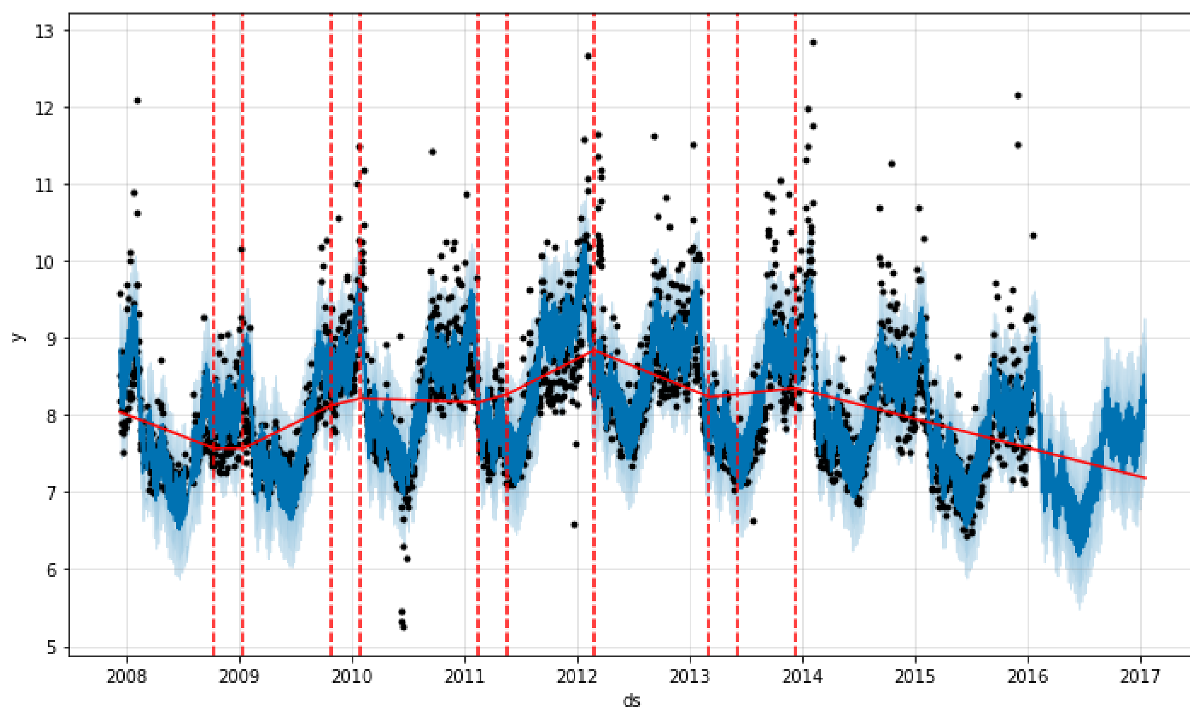
候補からある確率で実際に傾向が変化する点が選ばれる。傾向の変化量を表す事前分布は，0にピークをもつ分布であるので、疎な点が選ばれる。以下では、変化量`delta`をグラフに示す。

```
deltas = m.params['delta'].mean(0)
fig = plt.figure(facecolor='w', figsize=(10, 6))
ax = fig.add_subplot(111)
ax.bar(range(len(deltas)), deltas, facecolor='#0072B2', edgecolor='#0072B2')
ax.grid(True, which='major', c='gray', ls='-', lw=1, alpha=0.2)
ax.set_ylabel('Rate change')
ax.set_xlabel('Potential changepoint')
fig.tight_layout()
```



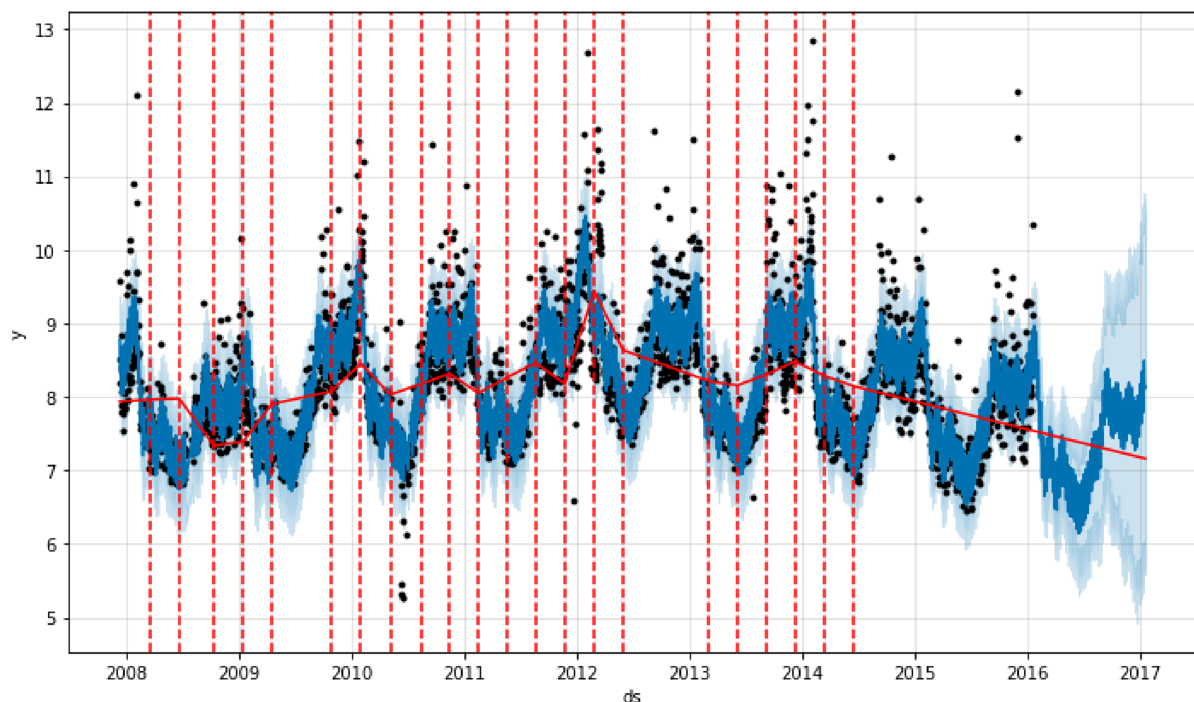
add_changepoints_to_plotを使うと、変化した点（日次）と傾向変動を図に追加して描画できる。

```
from fbprophet.plot import add_changepoints_to_plot
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```



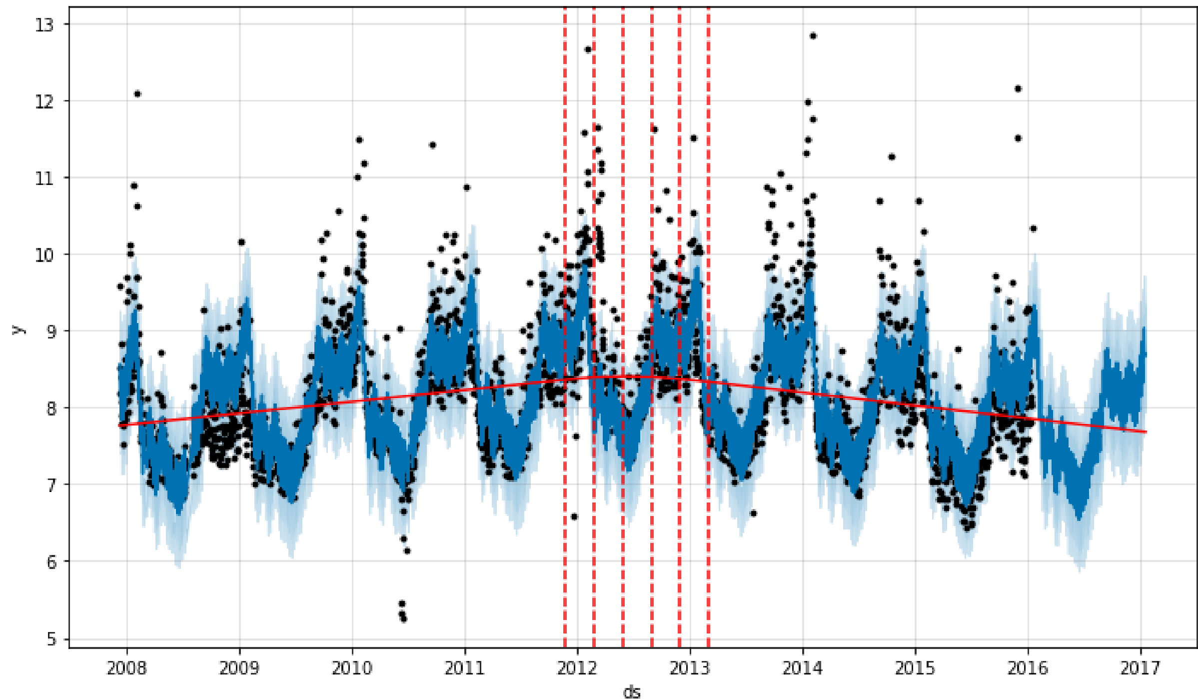
変化点の数を制御するためのパラメータは`changepoint_prior_scale`であり、既定値は0.05である。これを増やすと変化点が増え、予測の自由度が増すため予測幅が大きくなる。

```
m = Prophet(changepoint_prior_scale=0.5)
forecast = m.fit(df).predict(future)
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```



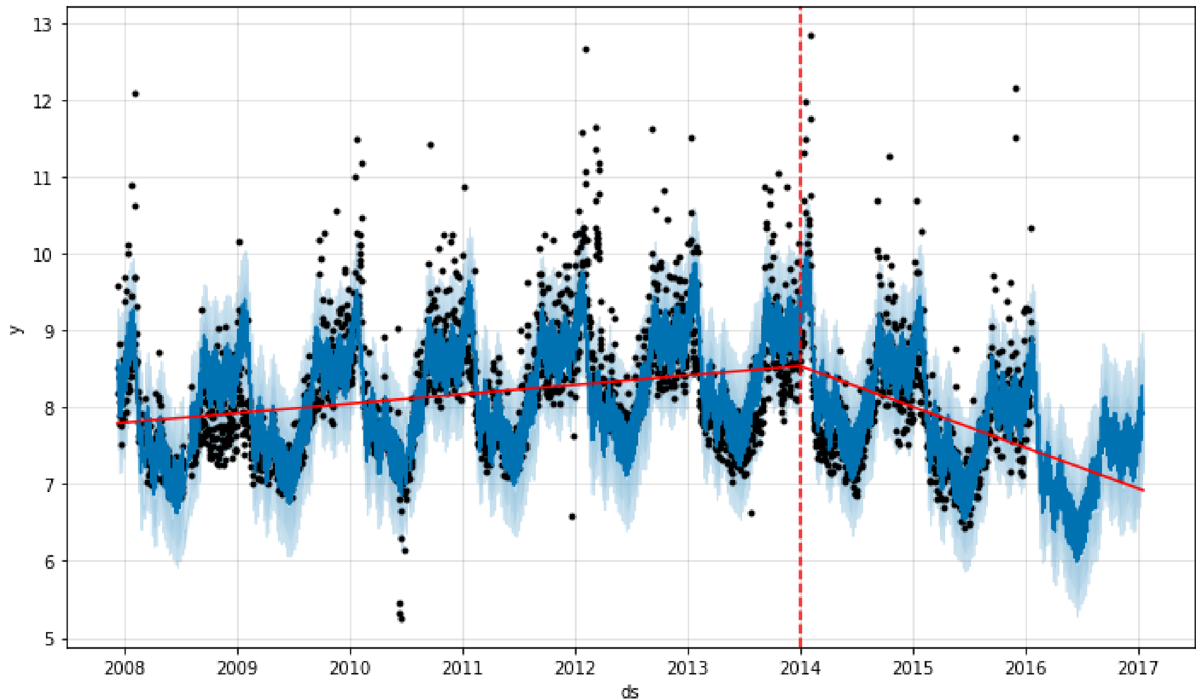
`changepoint_prior_scale`を小さくすると、変化点の数が減少するので、予測の自由度が減り、予測幅が狭くなる。

```
m = Prophet(changepoint_prior_scale=0.001)
forecast = m.fit(df).predict(future)
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```



傾向変化点を`changepoints`引数で与えることもできる。以下の例では、1つの日だけで変化するように設定している。

```
m = Prophet(changepoints=['2014-01-01'])
forecast = m.fit(df).predict(future)
fig = m.plot(forecast)
a = add_changepoints_to_plot(fig.gca(), m, forecast)
```



休日（特別なイベント）を考慮した予測

休日や特別なイベントをモデルに追加することを考える。そのためには、`holiday` と `ds`(datestamp)を列名としたデータフレームを準備する必要がある。`holiday`列にはイベント名を、`ds`にはそのイベントが発生する日時を入力する。また、イベントの影響が指定した日時の前後何日まで影響を与えるかを示す2つの列`lower_window`と`upper_window`を追加することができる。

```
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,
  'upper_window': 1,
})
```

```
11 superbowls = pd.DataFrame({
12   'holiday': 'superbowl',
13   'ds': pd.to_datetime(['2010-02-07', '2014-02-02', '2016-02-07']),
14   'lower_window': 0,
15   'upper_window': 1,
16 })
17 holidays = pd.concat((playoffs, superbowls))
18 holidays.head()
```

	holiday	ds	lower_window	upper_window
0	playoff	2008-01-13	0	1
1	playoff	2009-01-03	0	1
2	playoff	2010-01-16	0	1
3	playoff	2010-01-24	0	1
4	playoff	2010-02-07	0	1

引数`holidays`で休日を表すデータフレームを与えることによって、特別なイベントを考慮した予測を行うことができる。

```
1 m = Prophet(holidays=holidays)
2 forecast = m.fit(df).predict(future)
```

プレーオフやスーパーボールなどのイベント効果がある日だけ抜き出してデータフレームを表示する.

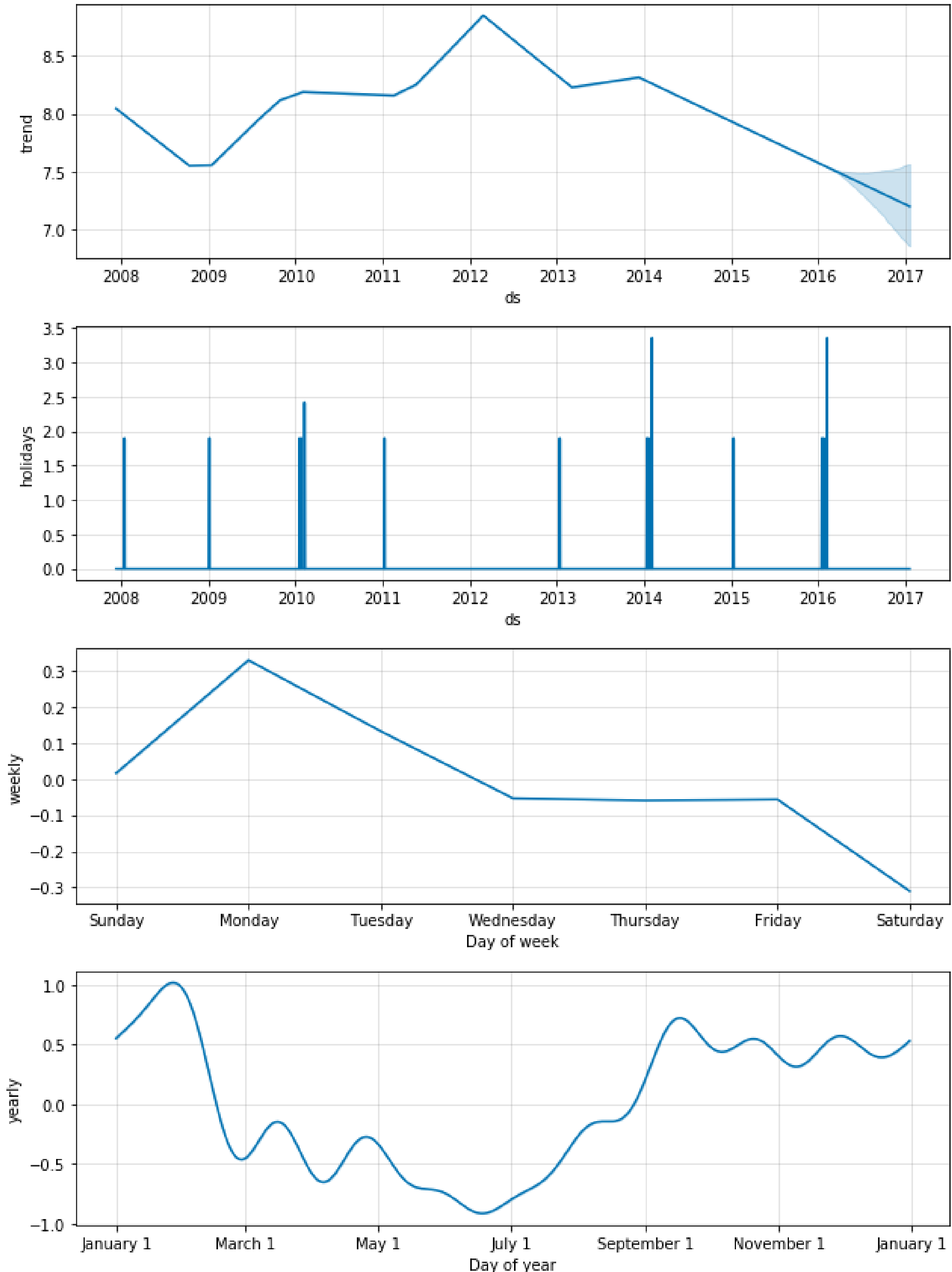
```
1 forecast[(forecast['playoff'] + forecast['superbowl']).abs() > 0][
2         ['ds', 'playoff', 'superbowl']][-10:]
```

	ds	playoff	superbowl

2190	2014-02-02	1.224174	1.197584
2191	2014-02-03	1.899740	1.459690
2532	2015-01-11	1.224174	0.000000
2533	2015-01-12	1.899740	0.000000
2901	2016-01-17	1.224174	0.000000
2902	2016-01-18	1.899740	0.000000
2908	2016-01-24	1.224174	0.000000
2909	2016-01-25	1.899740	0.000000
2922	2016-02-07	1.224174	1.197584
2923	2016-02-08	1.899740	1.459690

因子別に描画を行うと，イベントによって変化した量が描画される（上から2番目）．

```
fig = m.plot_components(forecast)
```



`plot_forecast_component`関数にイベント名を引数として渡すことによってイベントごとの影響量を描画できる。たとえば，スーパーボールの影響を見たいときには，`plot_forecast_component(m, forecast, 'superbowl')`とする.

国（州）別の休日

`add_country_holidays`を用いて，各国（州）の休日データを追加することができる。日本のデータもあるが、天皇誕生日がずれていたりするので、注意を要する。

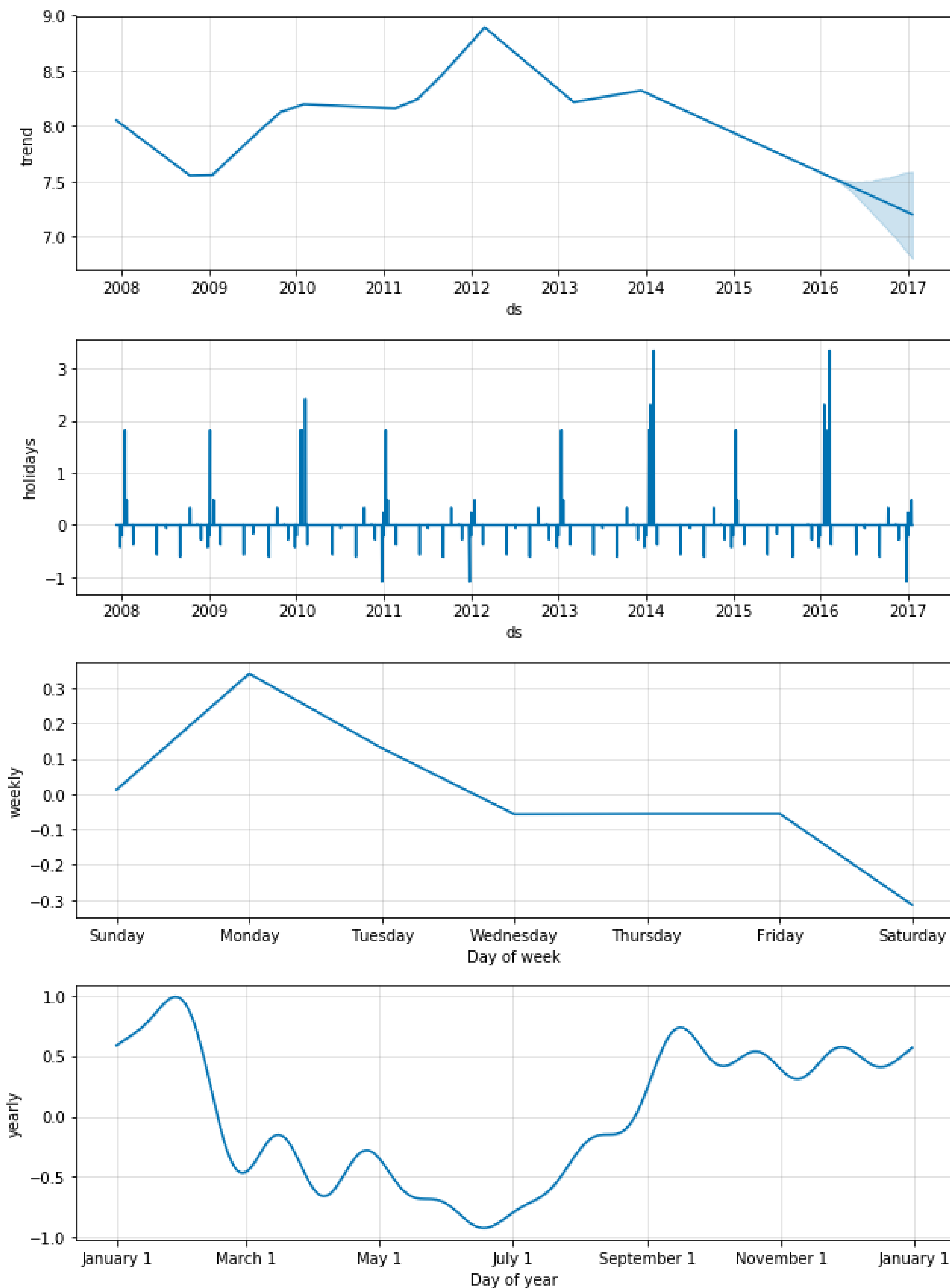
```
m = Prophet(holidays=holidays)
m.add_country_holidays(country_name='US')
m.fit(df)
```

```
m.train_holiday_names
0                          playoff
1                        superbowl
2                   New Year's Day
3      Martin Luther King, Jr. Day
4             Washington's Birthday
5                     Memorial Day
6                 Independence Day
7                        Labor Day
8                     Columbus Day
9                     Veterans Day
10                    Thanksgiving
11                   Christmas Day
12        Christmas Day (Observed)
13         Veterans Day (Observed)
14     Independence Day (Observed)
15       New Year's Day (Observed)
dtype: object
```

米国の休日を考慮して予測を行い，因子別に描画してみる．上から2番目が，休日に対する影響を表している．

```
forecast = m.predict(future)
```

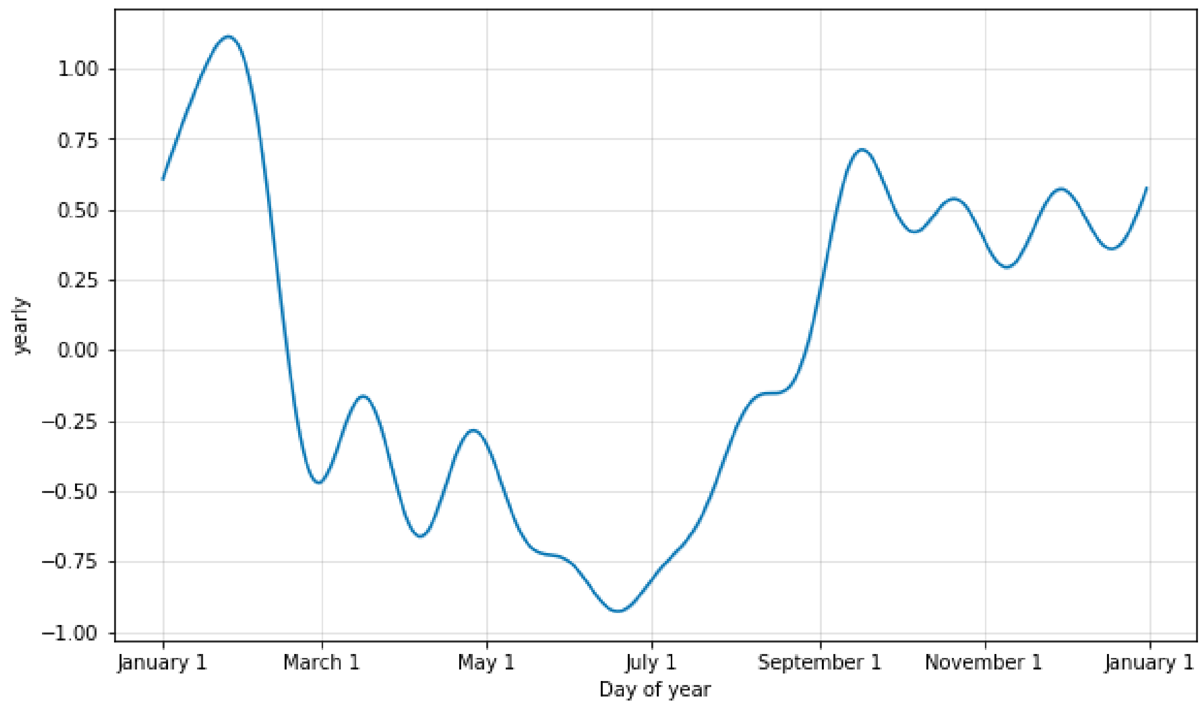
```
2 fig = m.plot_components(forecast)
```



季節変動に対するフーリエ次数

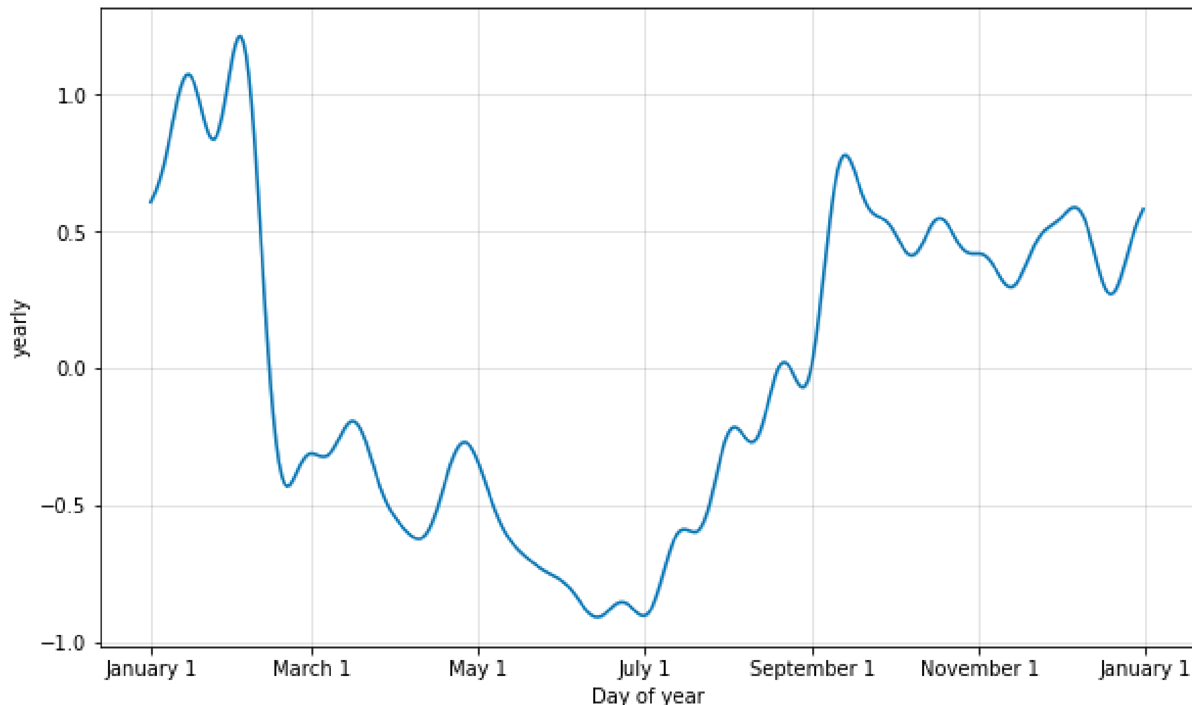
季節変動の変化の度合いは、`yearly_seasonality`（既定値は10）や`fourier_order`（フーリエ次数）で制御できる。

```
from fbprophet.plot import plot_yearly
m = Prophet().fit(df)
a = plot_yearly(m)
```



yearly_seasonalityを既定値の10から20に増やしてみると，以下のようになる．

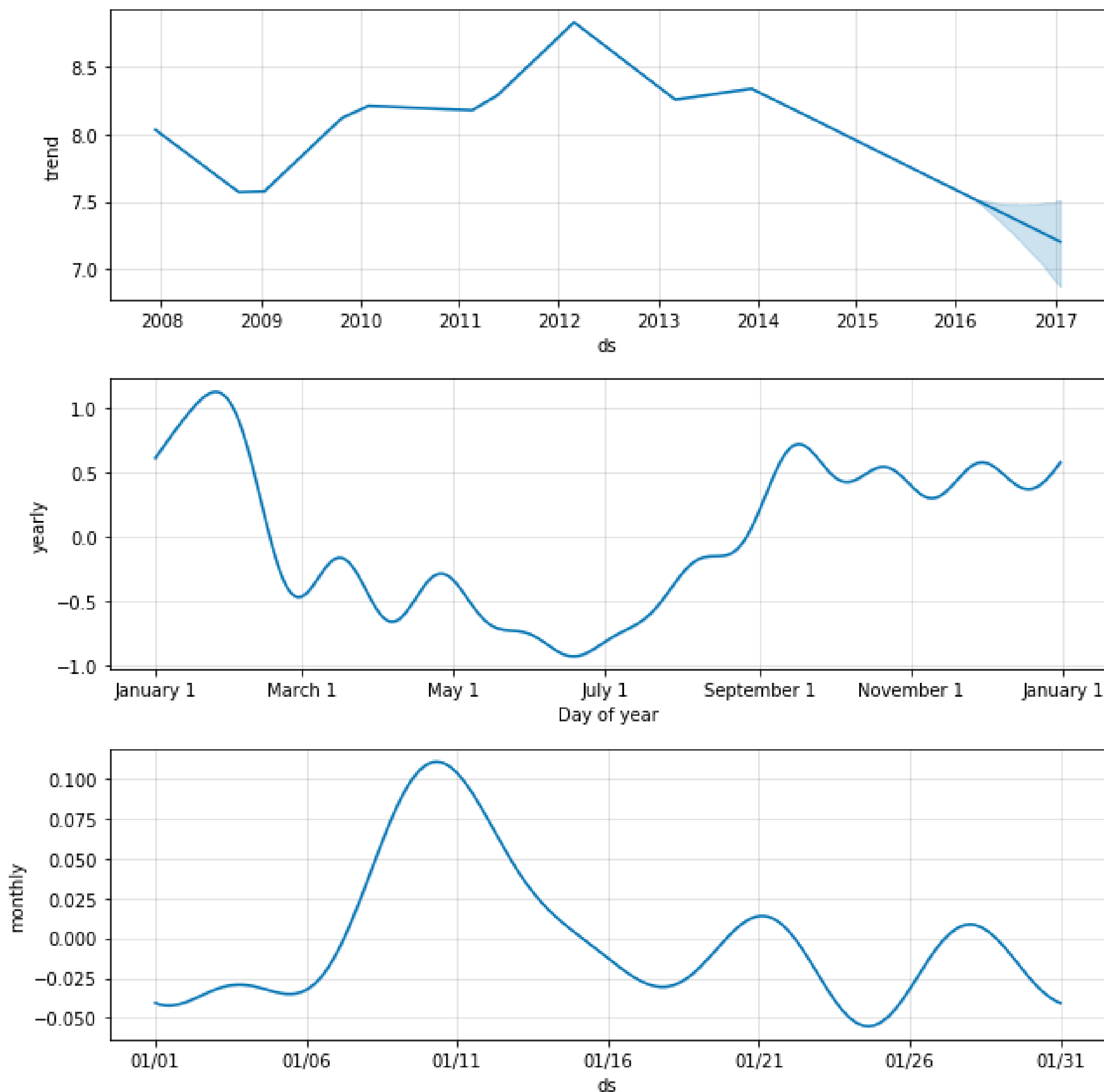
```
from fbprophet.plot import plot_yearly
m = Prophet(yearly_seasonality=20).fit(df)
a = plot_yearly(m)
```



ユーザーが設定した季節変動

Prophetでは規定値の年次や週次の季節変動だけでなく、ユーザー自身で季節変動を定義・追加できる。以下では、週次の季節変動を除き，かわりに周期が30.5日の月次変動をフーリエ次数5として追加している。

```
m = Prophet(weekly_seasonality=False)
m.add_seasonality(name='monthly', period=30.5, fourier_order=5)
forecast = m.fit(df).predict(future)
fig = m.plot_components(forecast)
```

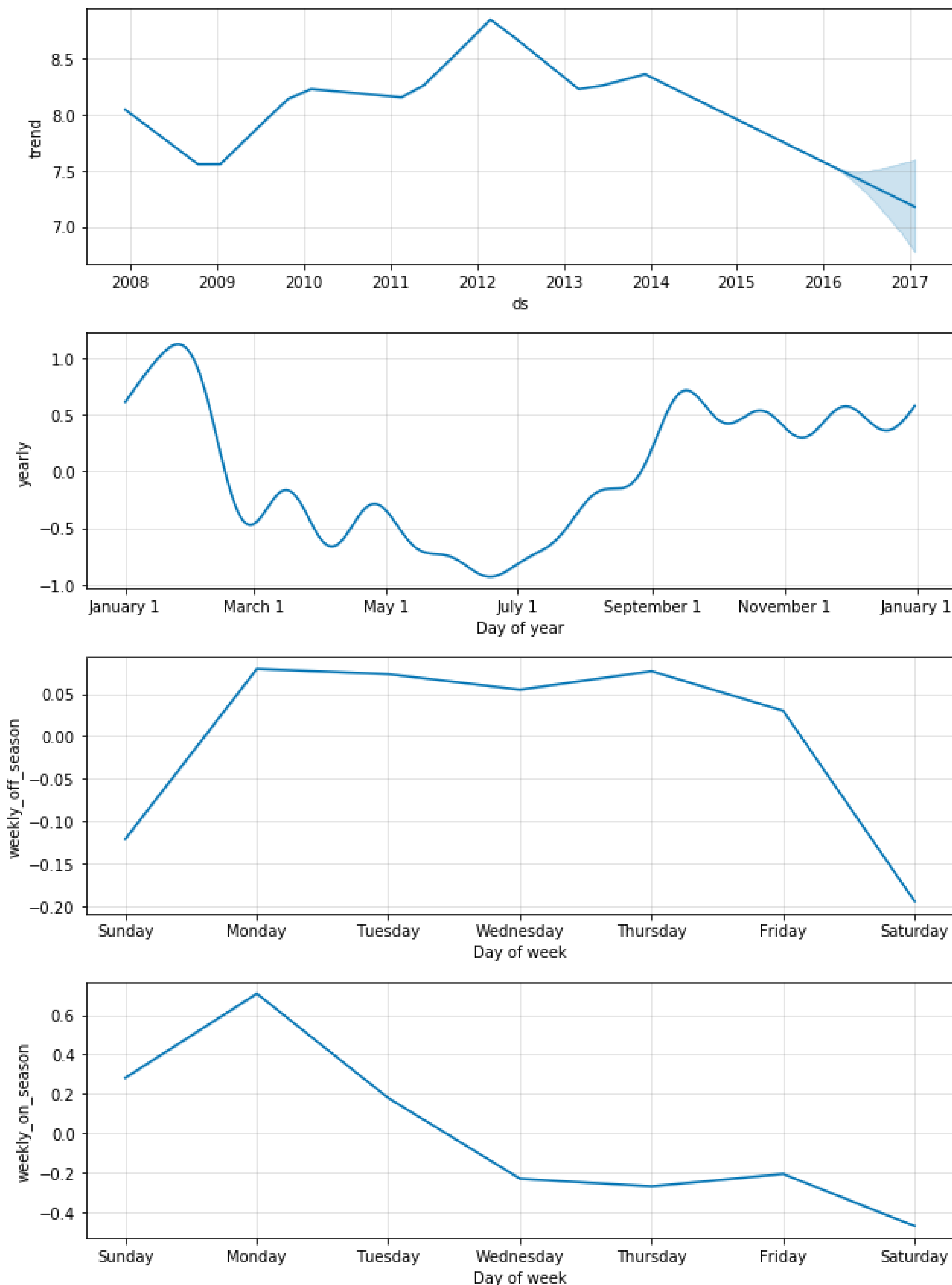


他の要因に依存した季節変動

他の要因に依存した季節変動も定義・追加することができる。以下の例では、オンシーズンとオフシーズンごと週次変動を定義し、追加している。

```
def is_nfl_season(ds):
    date = pd.to_datetime(ds)
    return (date.month > 8 or date.month < 2)
df['on_season'] = df['ds'].apply(is_nfl_season)
df['off_season'] = ~df['ds'].apply(is_nfl_season)
```

```
m = Prophet(weekly_seasonality=False)
m.add_seasonality(name='weekly_on_season', period=7, fourier_order=3, condition_name='on_season')
m.add_seasonality(name='weekly_off_season', period=7, fourier_order=3, condition_name='off_season')
future['on_season'] = future['ds'].apply(is_nfl_season)
future['off_season'] = ~future['ds'].apply(is_nfl_season)
forecast = m.fit(df).predict(future)
fig = m.plot_components(forecast)
```



休日と季節変動の効果の調整法

休日の影響を抑制するためには、`holidays_prior_scale`を小さくすれば良い。このパラメータの規定値は10であり、これはほとんど正則化を行わないことを意味する。以下では，`holidays_prior_scale`を0.05に設定して予測を行う．

```
m = Prophet(holidays=holidays, holidays_prior_scale=0.05).fit(df)
forecast = m.predict(future)
forecast[(forecast['playoff'] + forecast['superbowl']).abs() > 0][
    ['ds', 'playoff', 'superbowl']][-10:]
```

	ds	playoff	superbowl
2190	2014-02-02	1.205723	0.963130
2191	2014-02-03	1.851823	0.991663
2532	2015-01-11	1.205723	0.000000
2533	2015-01-12	1.851823	0.000000
2901	2016-01-17	1.205723	0.000000
2902	2016-01-18	1.851823	0.000000
2908	2016-01-24	1.205723	0.000000
2909	2016-01-25	1.851823	0.000000
2922	2016-02-07	1.205723	0.963130
2923	2016-02-08	1.851823	0.991663

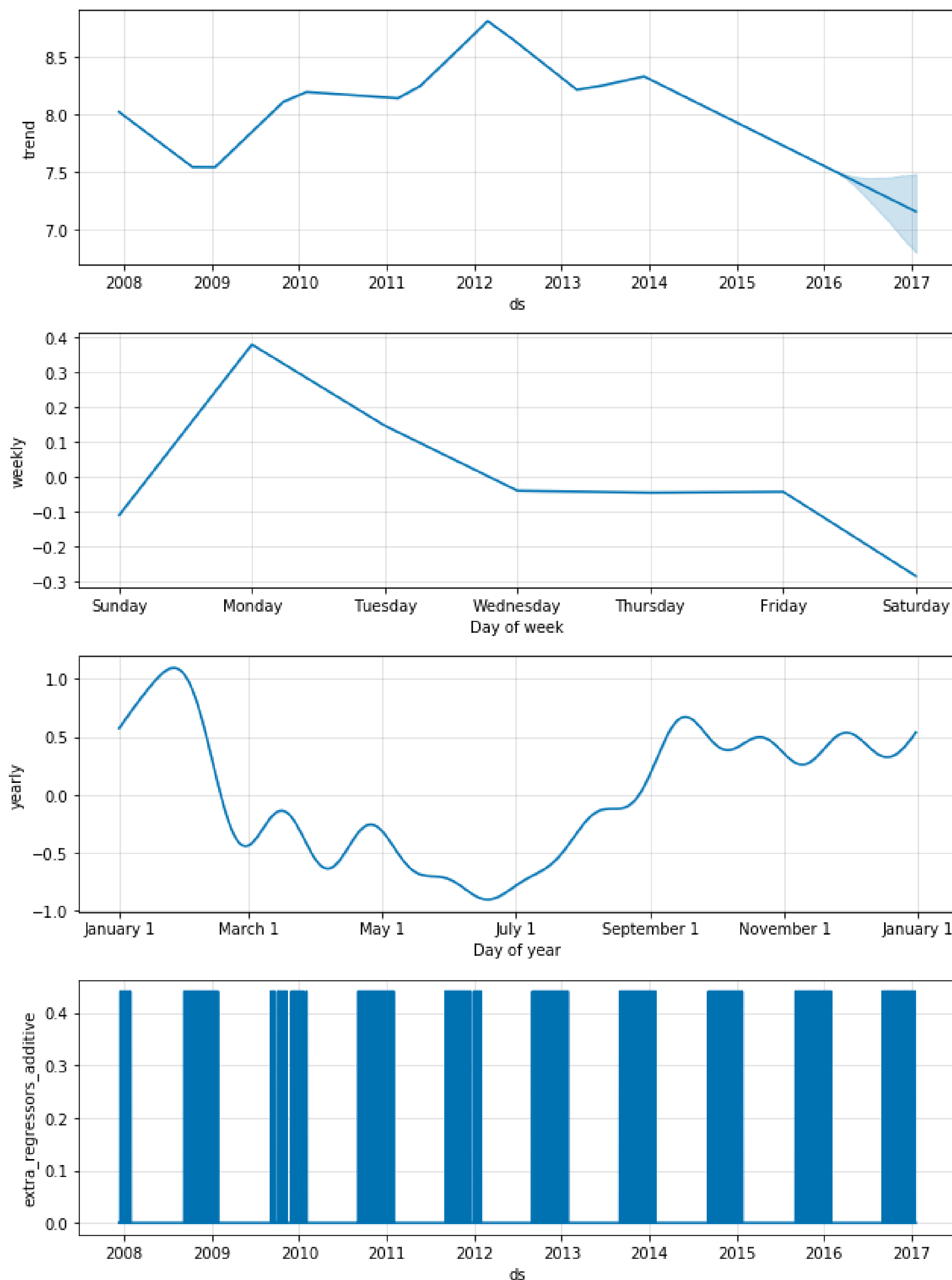
スーパーボール（superbowl）の効果が抑制されていることが見てとれる．
季節変動の影響は`seasonality_prior_scale`を小さくすることによって抑制できる。

```
m = Prophet()
m.add_seasonality(
    name='weekly', period=7, fourier_order=3, prior_scale=0.1)
```

予測因子の追加

`add_regressor`メソッドを用いると、モデルに因子を追加できる。以下の例では、オンシーズンの日曜日にだけ影響がでる因子を追加している。

```
def nfl_sunday(ds):
    date = pd.to_datetime(ds)
    if date.weekday() == 6 and (date.month > 8 or date.month < 2):
        return 1
    else:
        return 0
df['nfl_sunday'] = df['ds'].apply(nfl_sunday)
m = Prophet()
m.add_regressor('nfl_sunday')
m.fit(df)
future['nfl_sunday'] = future['ds'].apply(nfl_sunday)
forecast = m.predict(future)
fig = m.plot_components(forecast)
```

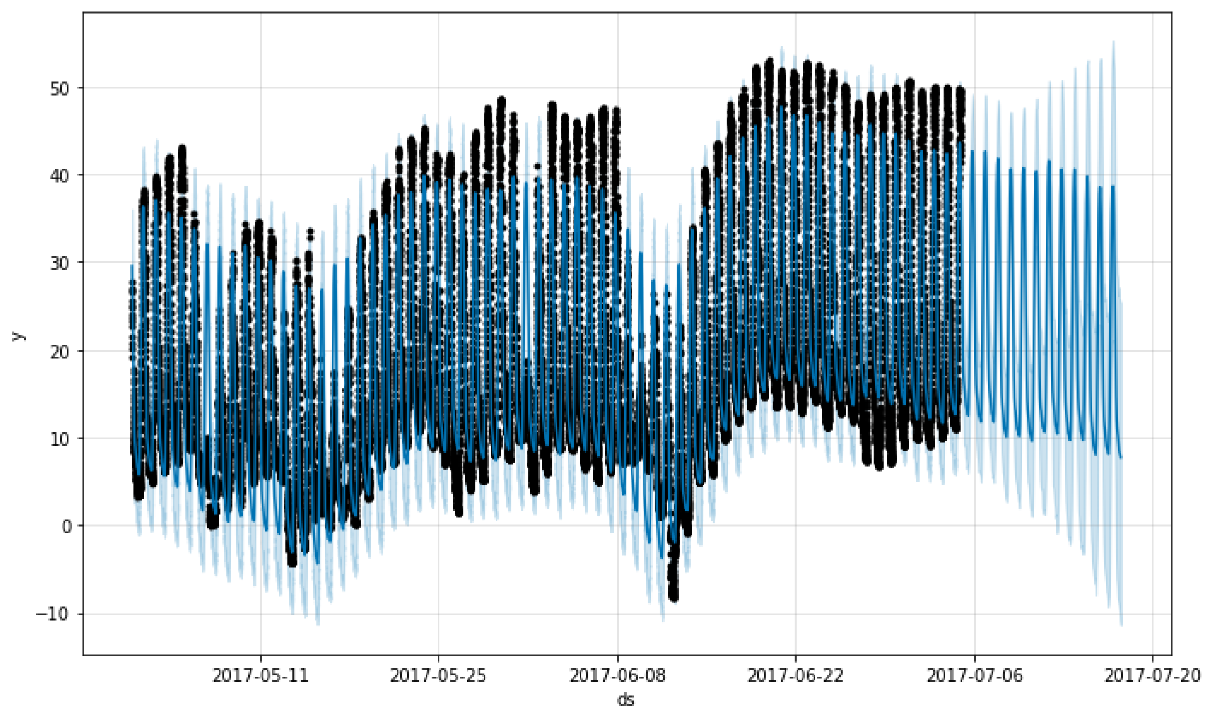


日付未満のデータ

日別でないデータも扱うことができる。例として、ヨセミテの5分ごとの気温の変化を予測してみる。 データ形式は、日付を表す`YYYY-MM-DD`の後に時刻を表す`HH:MM:SS`を追加する。

未来の時刻を表すデータフレームは、`make_future_dataframe`メソッドで生成するが、このとき引数`freq`で時間の刻みを指定する。 ここでは1時間を表す'H'を指定する。

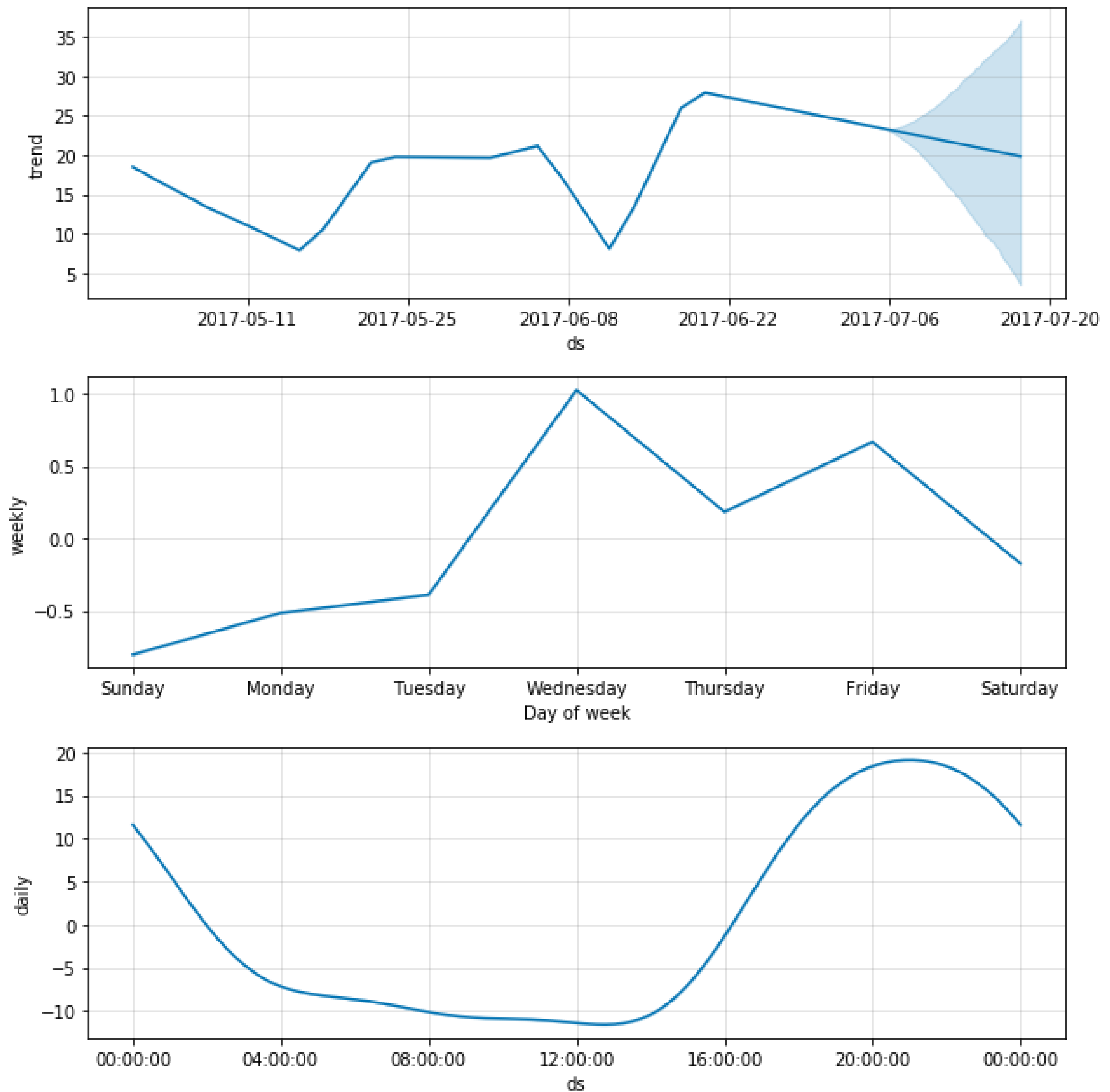
```
df = pd.read_csv('../examples/example_yosemite_temps.csv')
m = Prophet(changepoint_prior_scale=0.01).fit(df)
future = m.make_future_dataframe(periods=300, freq='H')
fcst = m.predict(future)
fig = m.plot(fcst)
```




```
df.head()
```

	ds	y
0	2017-05-01 00:00:00	27.8
1	2017-05-01 00:05:00	27.0
2	2017-05-01 00:10:00	26.8
3	2017-05-01 00:15:00	26.5
4	2017-05-01 00:20:00	25.6

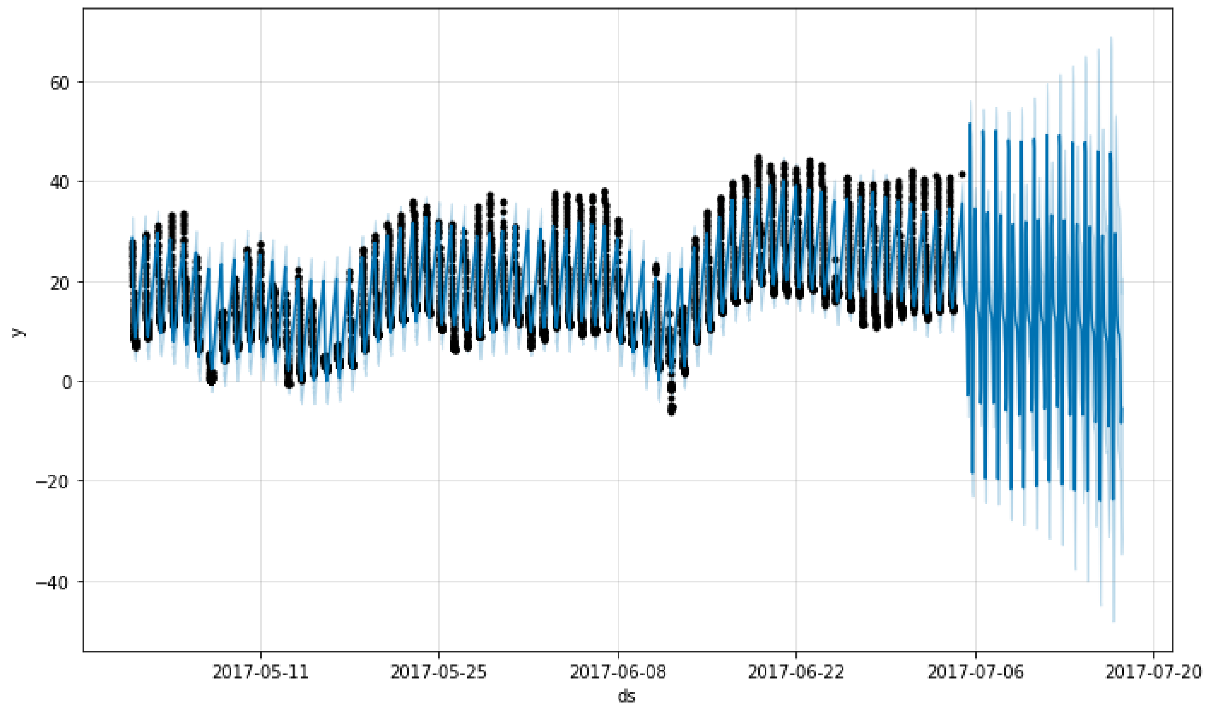
```
fig = m.plot_components(fcst)
```



時刻が０時から６時までの間しか観測できなかったと仮定して予測してみる。

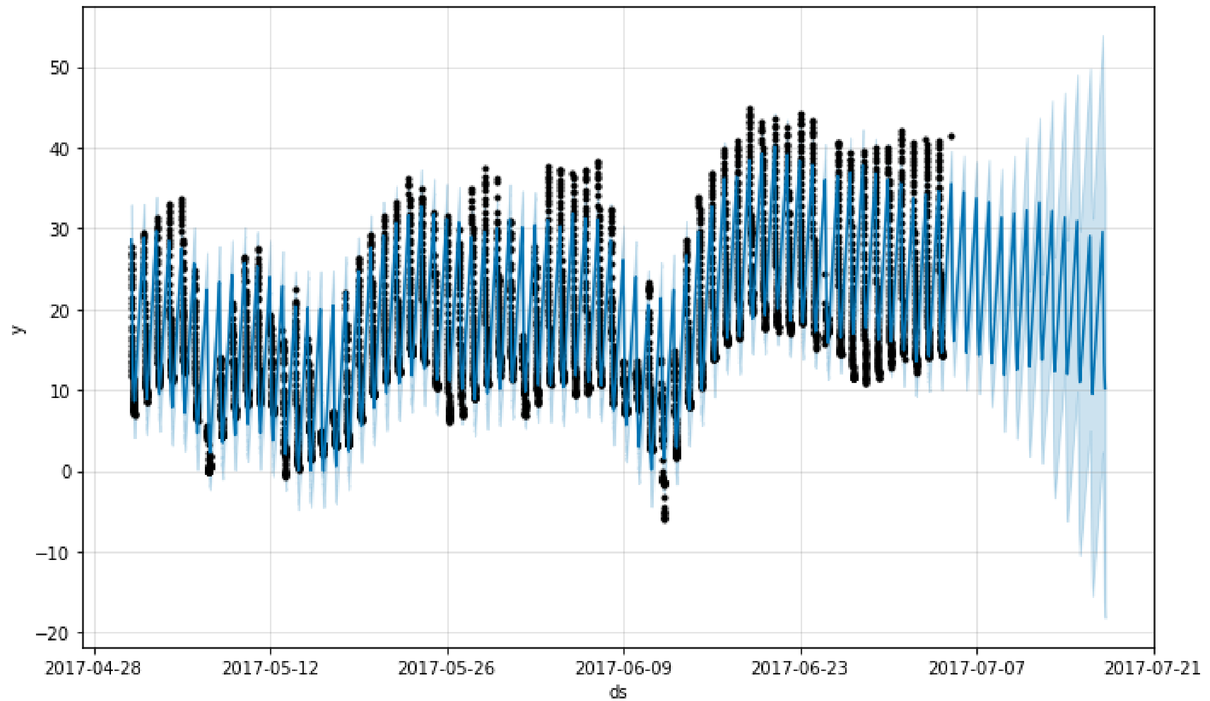
```
df2 = df.copy()
df2['ds'] = pd.to_datetime(df2['ds'])
df2 = df2[df2['ds'].dt.hour < 6]
m = Prophet().fit(df2)
future = m.make_future_dataframe(periods=300, freq='H')
fcst = m.predict(future)
```

```
7 fig = m.plot(fcst)
```



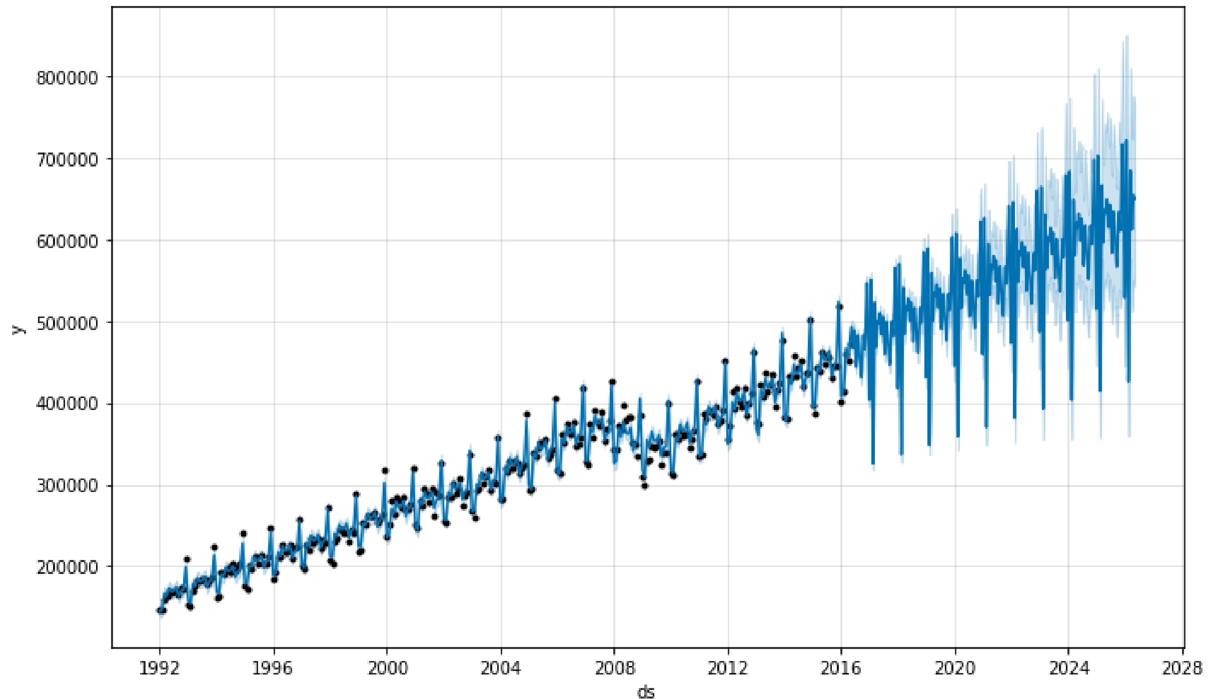
予測幅が大きくなってしまった。未来の時刻も 6 時前のものにして予測を行うことによって，予測精度が改善される。

```
1 future2 = future.copy()
2 future2 = future2[future2['ds'].dt.hour < 6]
3 fcst = m.predict(future2)
4 fig = m.plot(fcst)
```



米国の小売店の需要の月次データを用いて予測してみる。このデータは月の始めにしかデータがない。期の既定値は日なので、そのまま予測するとうまくいかない。

```
df = pd.read_csv('../examples/example_retail_sales.csv')
m = Prophet(seasonality_mode='multiplicative').fit(df)
future = m.make_future_dataframe(periods=3652)
fcst = m.predict(future)
fig = m.plot(fcst)
```



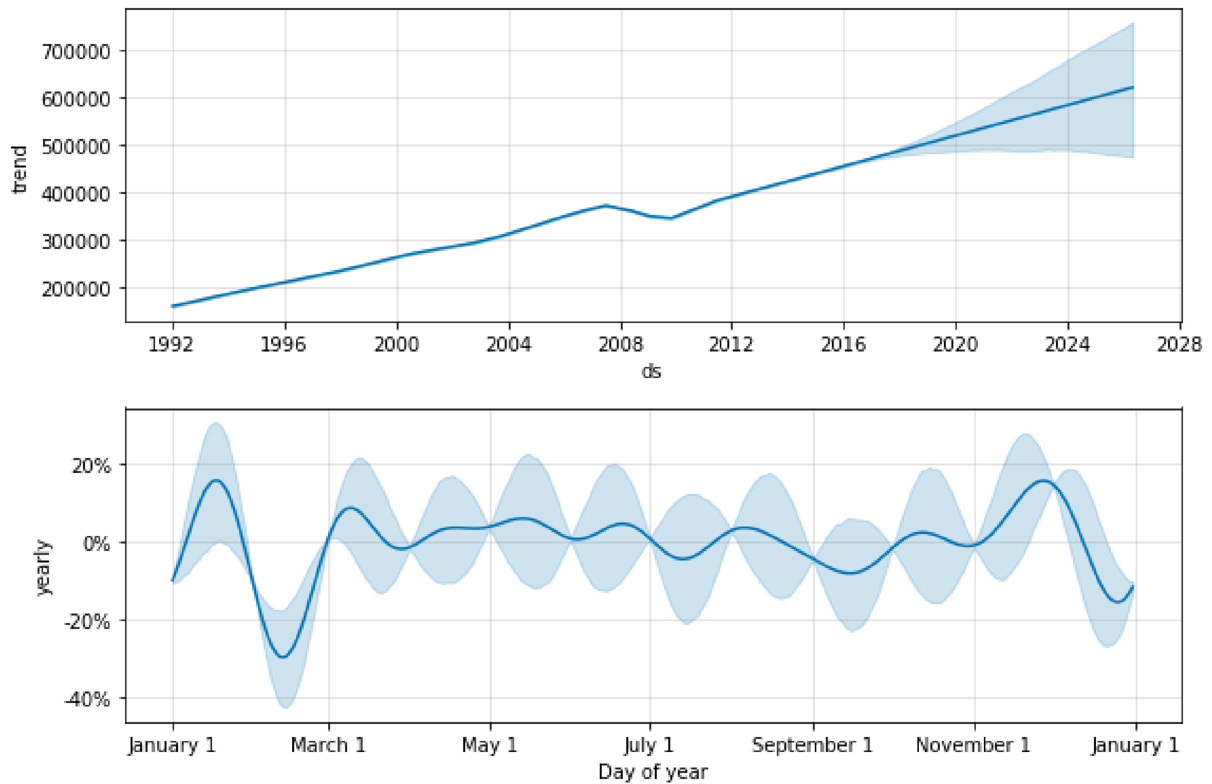
[16]

```
df.head()
```

	ds	y
0	1992-01-01	146376
1	1992-02-01	147079
2	1992-03-01	159336
3	1992-04-01	163669
4	1992-05-01	170068

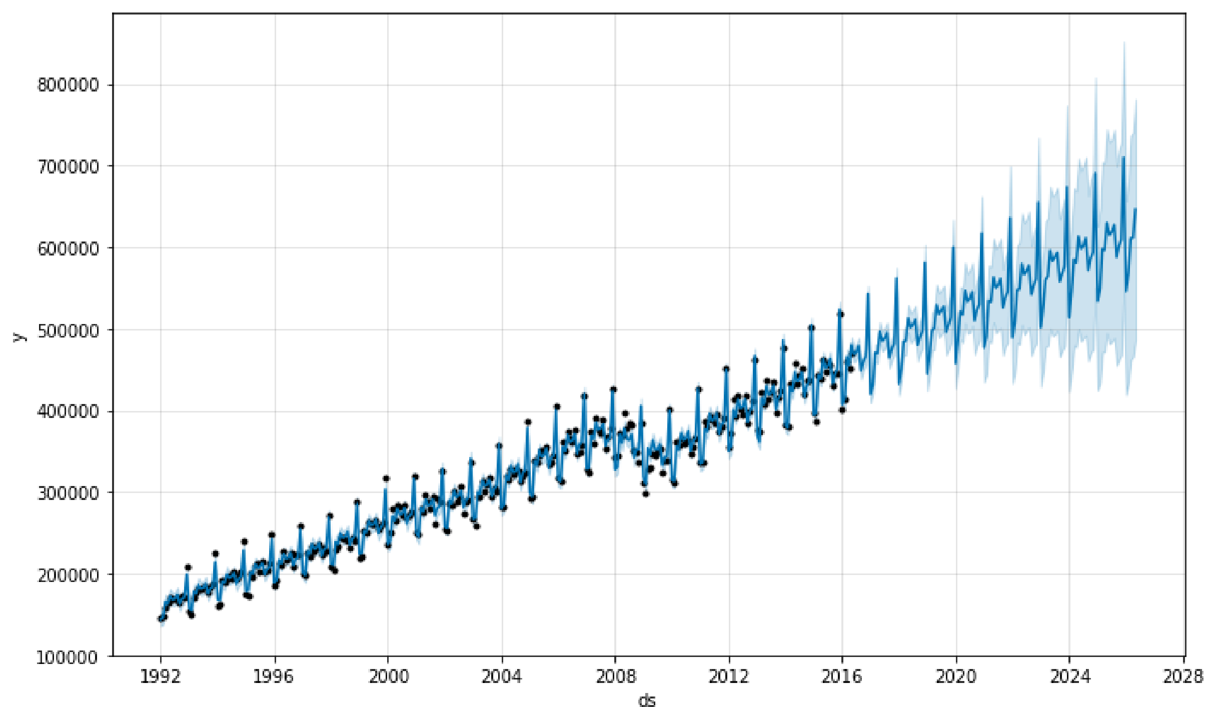
引数の`mcmc_samples`を正の値に設定することによって、マルコフ連鎖モンテカルロ法で季節変動の不確実性を予測することができる。以下のように、月初は合っているが、月の途中で誤差が大きくなっていることが確認できる。

```
m = Prophet(seasonality_mode='multiplicative', mcmc_samples=300).fit(df)
fcst = m.predict(future)
fig = m.plot_components(fcst)
```



未来の日時を生成する際に、月次で生成するとうまく予測できる。

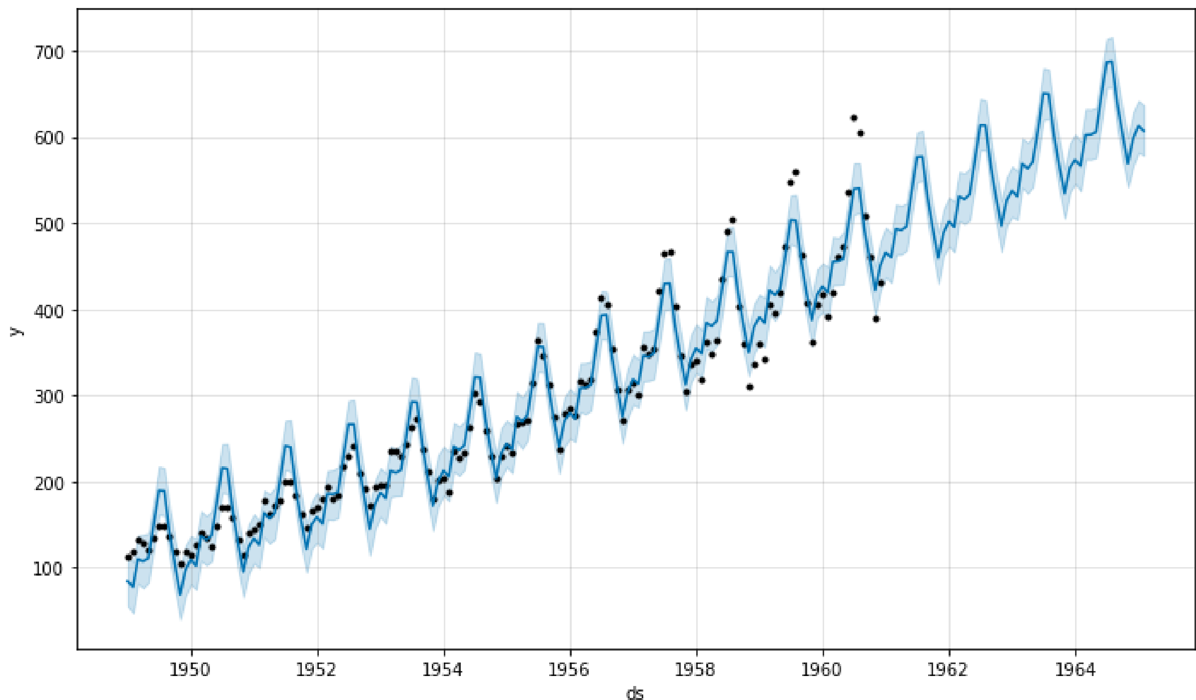
```
future = m.make_future_dataframe(periods=120, freq='M')
fcst = m.predict(future)
fig = m.plot(fcst)
```



乗法的季節変動

Prophetの規定値では季節変動は加法的モデルであるが、問題によっては乗法的季節変動の方が良い場合もある。例として、航空機の乗客数を予測してみよう。まずは既定値の加法的季節変動モデルで予測する。

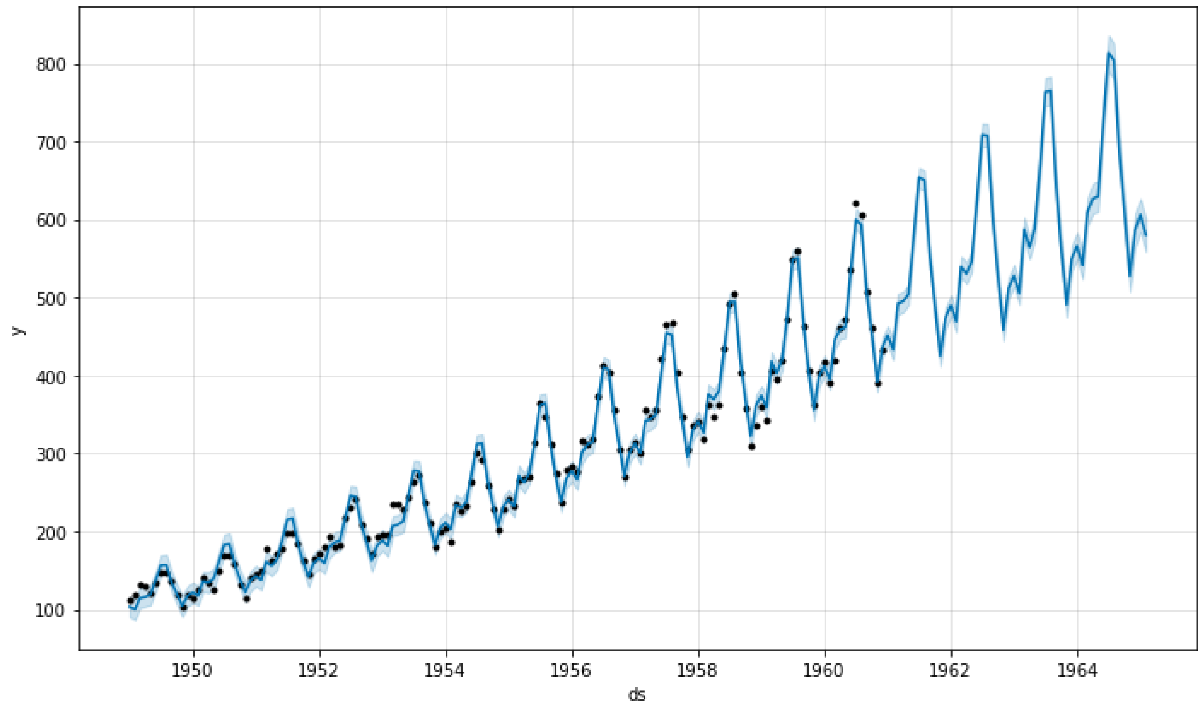
```
df = pd.read_csv('../examples/example_air_passengers.csv')
m = Prophet()
m.fit(df)
future = m.make_future_dataframe(50, freq='MS')
forecast = m.predict(future)
fig = m.plot(forecast)
```



このデータは明らかに年ベースの季節変動がある。時系列の前半では季節変動を大きく予測しすぎており、後半になると徐々に大きくなる季節変動に追従できなくなっていることが分かる。こういった場合には季節変動の影響を乗じることによる乗法的季節変動モデルを使うと良い。乗法的季節変動モデルに変更するためには、引数`seasonality_mode`を`multiplicative`に設定すれば良い。

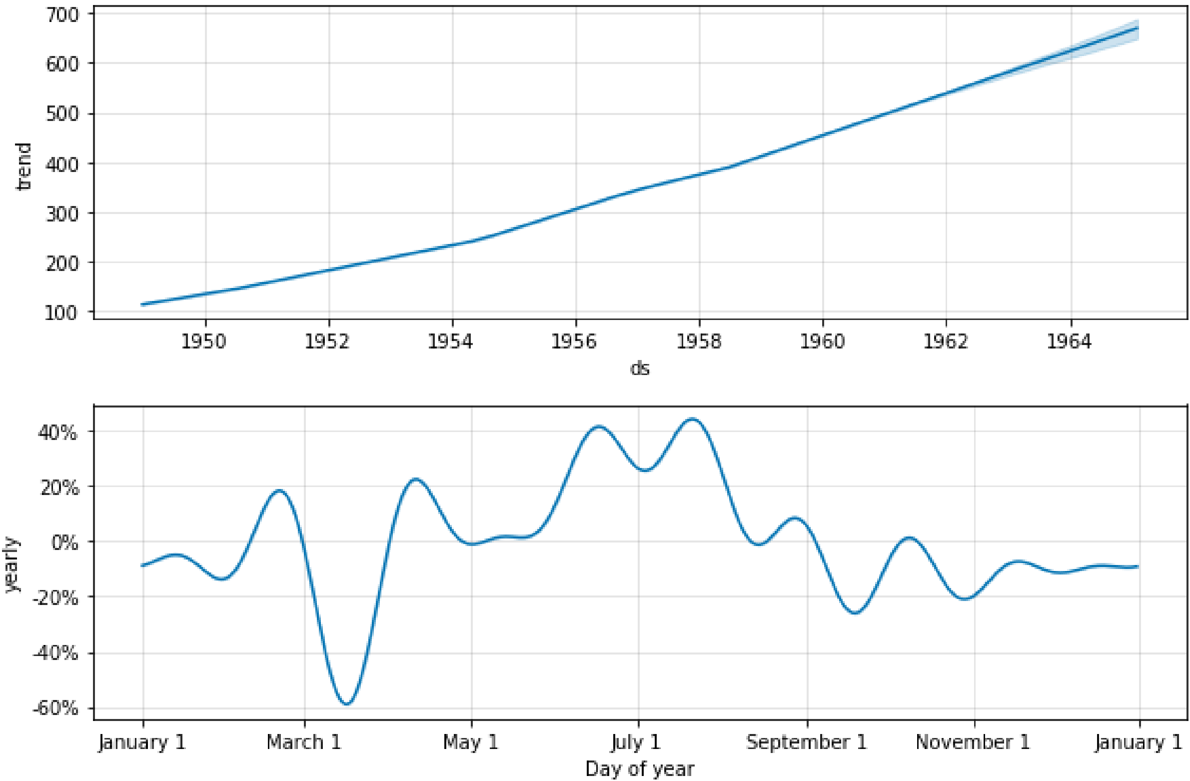
```
m = Prophet(seasonality_mode='multiplicative')
```

```
2 m.fit(df)
3 forecast = m.predict(future)
4 fig = m.plot(forecast)
```



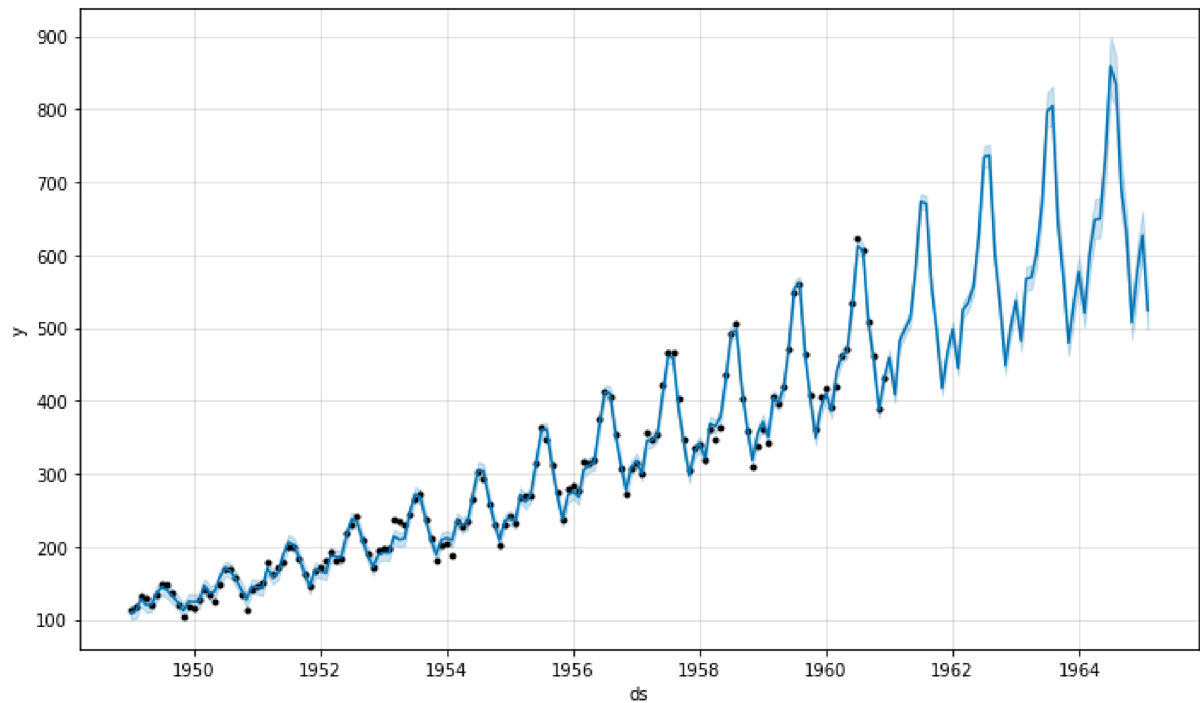
要因ごとに描画してみると、傾向変動が大きくなるにつれて、季節変動を乗じることによって、加法的モデルより良く適合していることが見てとれる。

```
fig = m.plot_components(forecast)
```



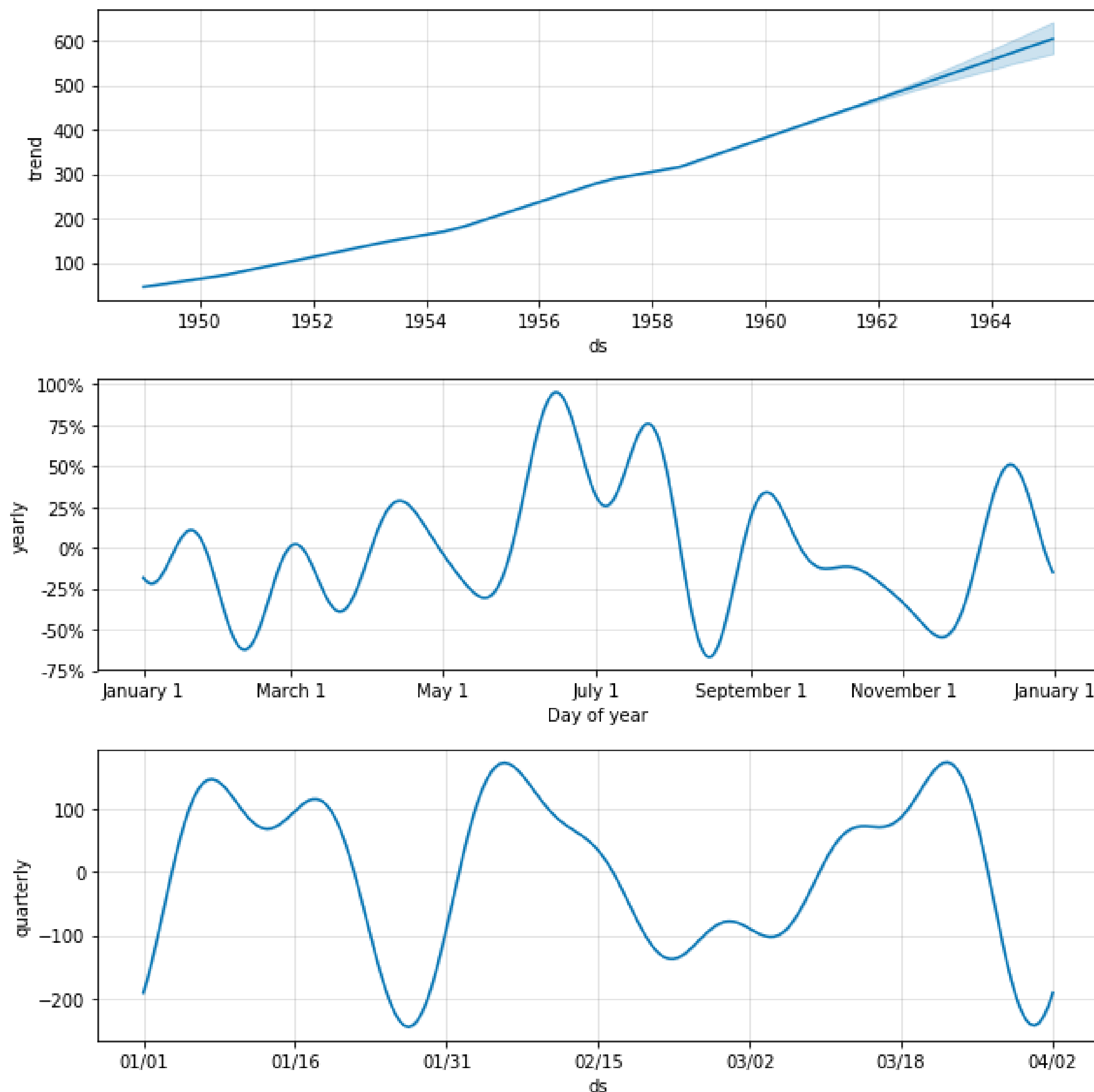
コンストラクタで設定した`seasonality_mode`が既定値として他の季節変動にも適用される．なお，add_seasonality`メソッドで新たな季節変動を追加するときに、`mode`引数を`additive`もしくは`multiplicative`に設定することによって、要因ごとに加法的か乗法的かを設定することができる。以下の例では、既定の季節変動を乗法的にし、追加した四半期の季節変動を加法的にしている。

```
m = Prophet(seasonality_mode='multiplicative')
m.add_seasonality('quarterly', period=91.25, fourier_order=8, mode='additive')
m.fit(df)
forecast = m.predict(future)
fig = m.plot(forecast)
```



要因別に描画してみる。

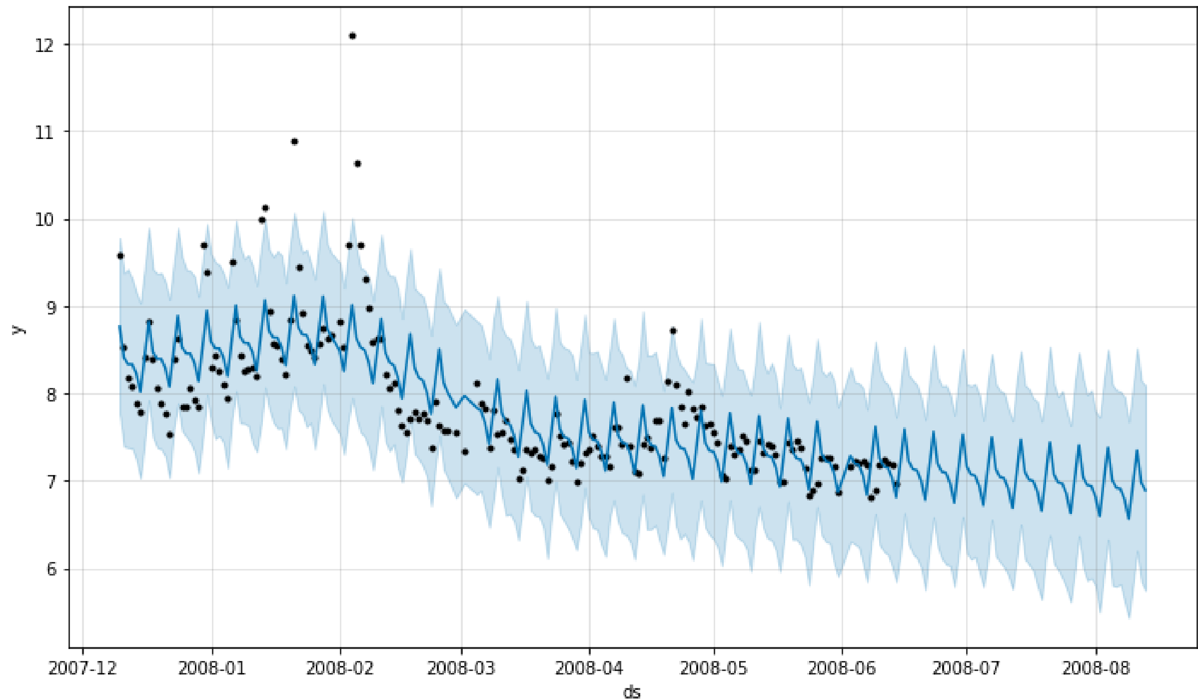
```
fig = m.plot_components(forecast)
```



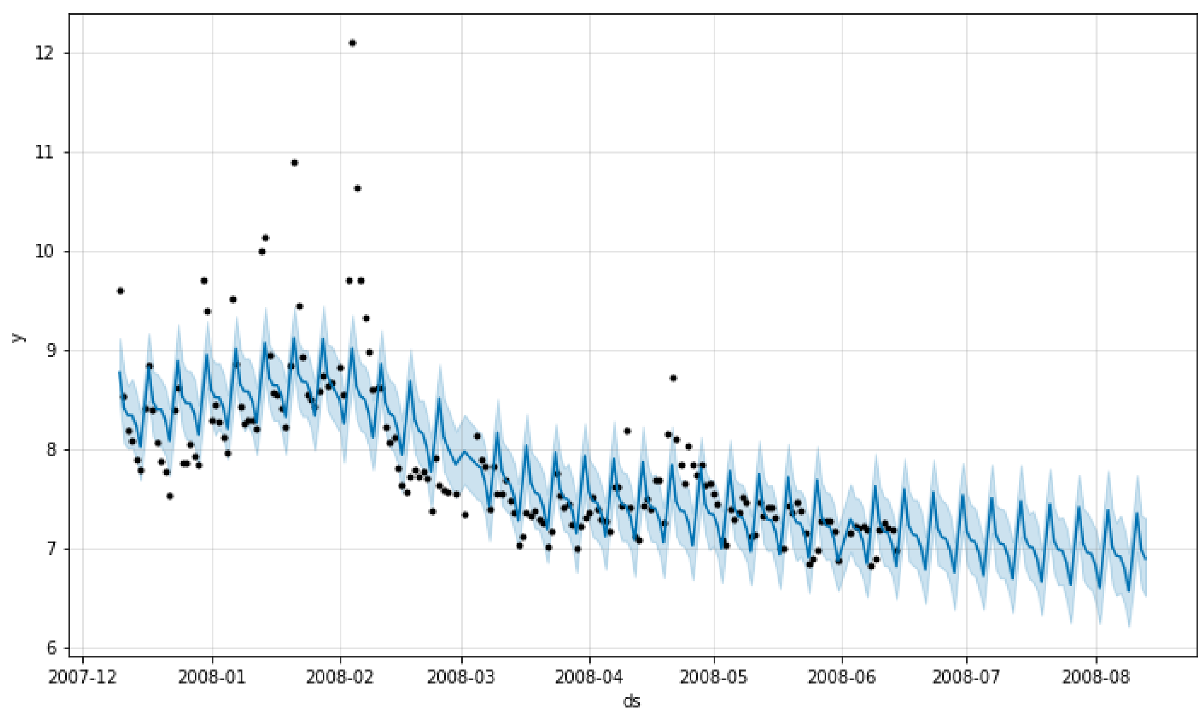
不確実性の幅

Prophetは既定では傾向変動に対する不確実性の幅を予測する。このとき、引数`interval_width`で予測の幅を設定できる。既定値は0.8である。このパラメータを大きくすると幅が広がり、小さくすると幅が狭くなることが確認できる。

```
forecast = Prophet(interval_width=0.95).fit(df).predict(future)
fig = m.plot(forecast)
```




```
forecast = Prophet(interval_width=0.5).fit(df).predict(future)
fig = m.plot(forecast)
```

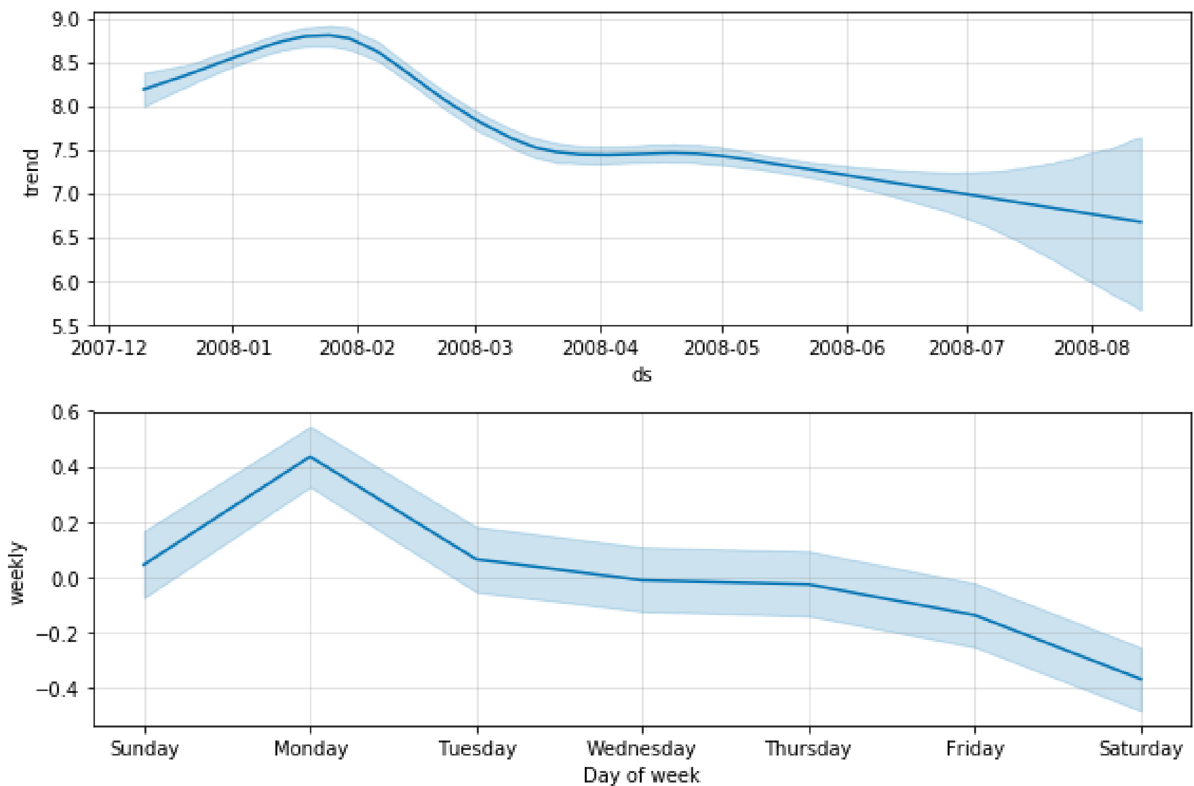


季節変動に対する不確実性を予測するためには、マルコフ連鎖モンテカルロ法を行う必要がある、そのためには、引数`mcmc_samples`をシミュレーションの反復回数に設定する。このパラメータの規定値は0である。

```
m = Prophet(mcmc_samples=300)
forecast = m.fit(df).predict(future)
```

これによって、最大事後確率（MAP）推定の代わりにマルコフ連鎖モンテカルロ法によるサンプリングが行われる。これは、非常に時間がかかることもある。 要因別に図を描画してみると、週次の季節変動に対しても不確実性の幅が示されていることが確認できる。

```
fig = m.plot_components(forecast)
```



検証と誤差の評価

Prophetでは、予測の精度を検証するための仕組みが組み込まれている。例として、Peyton Manningのデータを用いる。

```
df = pd.read_csv('../examples/example_wp_log_peyton_manning.csv')
m = Prophet()
```

```
m.fit(df)
future = m.make_future_dataframe(periods=366)
```

データセットは、全部で2905日分のデータで構成されている。

```
df.tail()
```

	ds	y
2900	2016-01-16	7.817223
2901	2016-01-17	9.273878
2902	2016-01-18	10.333775
2903	2016-01-19	9.125871
2904	2016-01-20	8.891374

In [27]:

```

```

Out[27]:

```
10.05
```

交差検証のためには、`cross_validation`を用いる。引数の`initial`には検証を開始する最初の日を入れ、`period`には予測の間隔を入れ、`horizon`には計画期間（予測を行う期間）を入れる。以下の例では、`initial`が730日なので、729日までの情報を用いて、その後365日の予測を行い、本当の値との誤差を評価し、次いで730+180日までの情報を用いて、その後365日の予測を行い評価し、という手順を最後の日まで繰り返す。(2905-730-365)/180 = 10.05であるので、11回の予測を行い評価することになる。`cross_validation`は、交差検証用のデータフレームを返す。

```
from fbprophet.diagnostics import cross_validation
df_cv = cross_validation(m, initial='730 days', period='180 days', horizon = '365 days')
df_cv.head()
```

	ds	yhat	yhat_low	yhat_upp	y	cutoff

			er	er		
0	2010-02-16	8.960441	8.413034	9.438598	8.242493	2010-02-15
1	2010-02-17	8.726966	8.213176	9.207769	8.008033	2010-02-15
2	2010-02-18	8.610869	8.115672	9.101070	8.045268	2010-02-15
3	2010-02-19	8.532795	8.047087	9.051873	7.928766	2010-02-15
4	2010-02-20	8.274904	7.806038	8.774270	7.745003	2010-02-15

最初の検証は730日後である2010-2-16日から1年間行われ、次の検証はその180日後である2010-08-14日から1年間行われる。最後の検証は2015-01-20日までのデータを用いて2016-01-20日まで行われる。

```
df_cv.tail()
```

	ds	yhat	yhat_lower	yhat_upper	y	cutoff
3983	2016-01-16	8.580395	7.865414	9.462696	7.817223	2015-01-20
3984	2016-01-17	8.975041	8.267482	9.702261	9.273878	2015-01-20
3985	2016-01-18	9.281189	8.575057	10.110297	10.333775	2015-01-20

3986	2016-01-19	9.064340	8.350171	9.814960	9.125871	2015-01-20
3987	2016-01-20	8.900521	8.227770	9.685436	8.891374	2015-01-20

`performance_metrics` を用いてメトリクス（評価尺度）を計算する。評価尺度は、平均平方誤差(mean squared error: MSE), 平均平方誤差の平方根 (root mean squared error: RMSE), 平均絶対誤差 (mean absolute error: MAE), 平均絶対パーセント誤差 (mean absolute percent error : MAPE), `yhat_lower` と`yhat_upper` の間に入っている割合（被覆率: coverage) である。

既定値では予測期間の最初の10%は除外して示される。これは、引数 `rolling_window` によって変更できる。

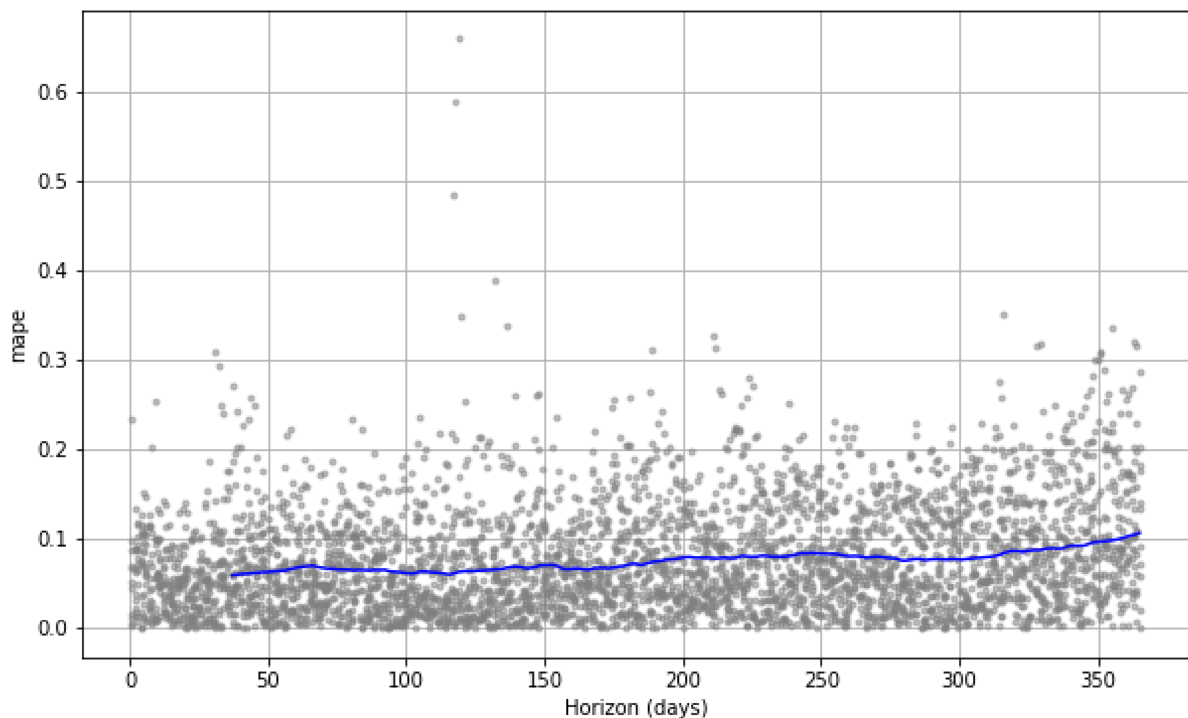
```
from fbprophet.diagnostics import performance_metrics
df_p = performance_metrics(df_cv, rolling_window=0.1)
df_p.head()
```

	horizon	mse	rmse	mae	mape	coverage
0	37 days	0.495161	0.703677	0.505237	0.058536	0.684102
1	38 days	0.500957	0.707783	0.510231	0.059114	0.684102
2	39 days	0.523235	0.723350	0.516340	0.059715	0.682732
3	40 days	0.530583	0.728411	0.519241	0.060026	0.681361
4	41 days					

		0.538145	0.733584	0.520267	0.060108	0.688899

評価尺度は`plot_cross_validation_metric`で可視化できる。以下では平均絶対パーセント誤差(MAPE)を描画している.

```
from fbprophet.plot import plot_cross_validation_metric
fig = plot_cross_validation_metric(df_cv, metric='mape')
```

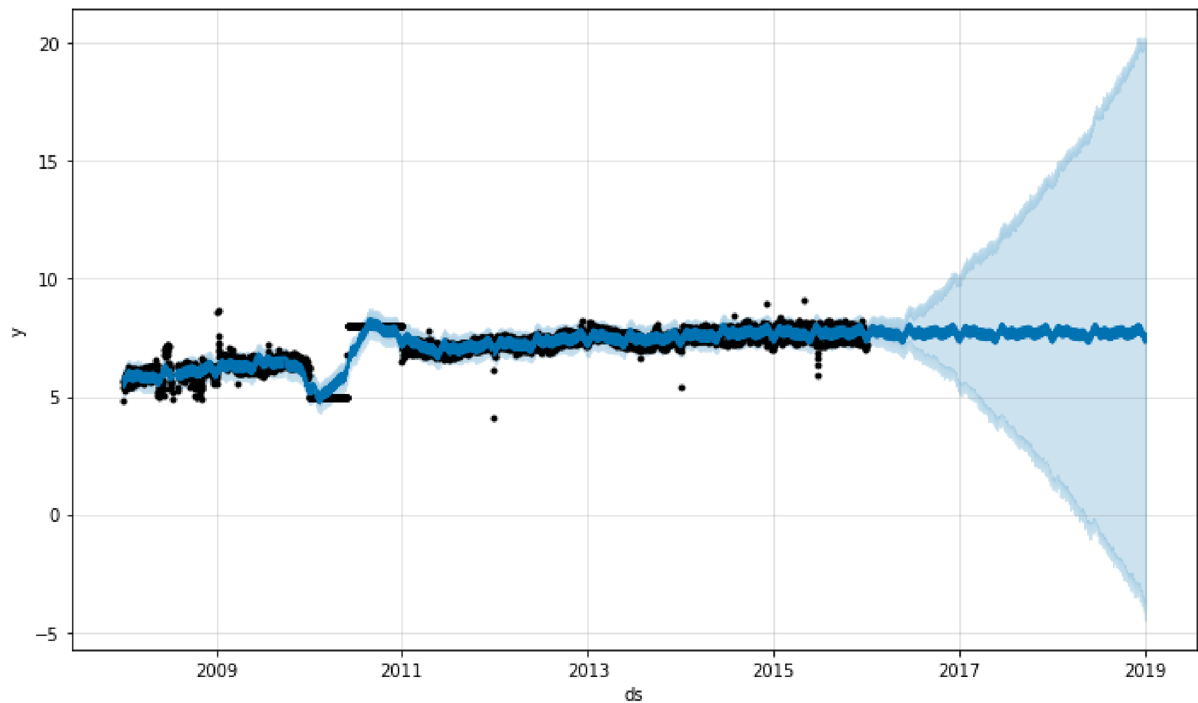


外れ値の影響

外れ値(outlier)を除去すると予測の精度が向上する場合がある。以下の例では、2010年あたりに大きな変化があるため、予測の幅が広がっている。

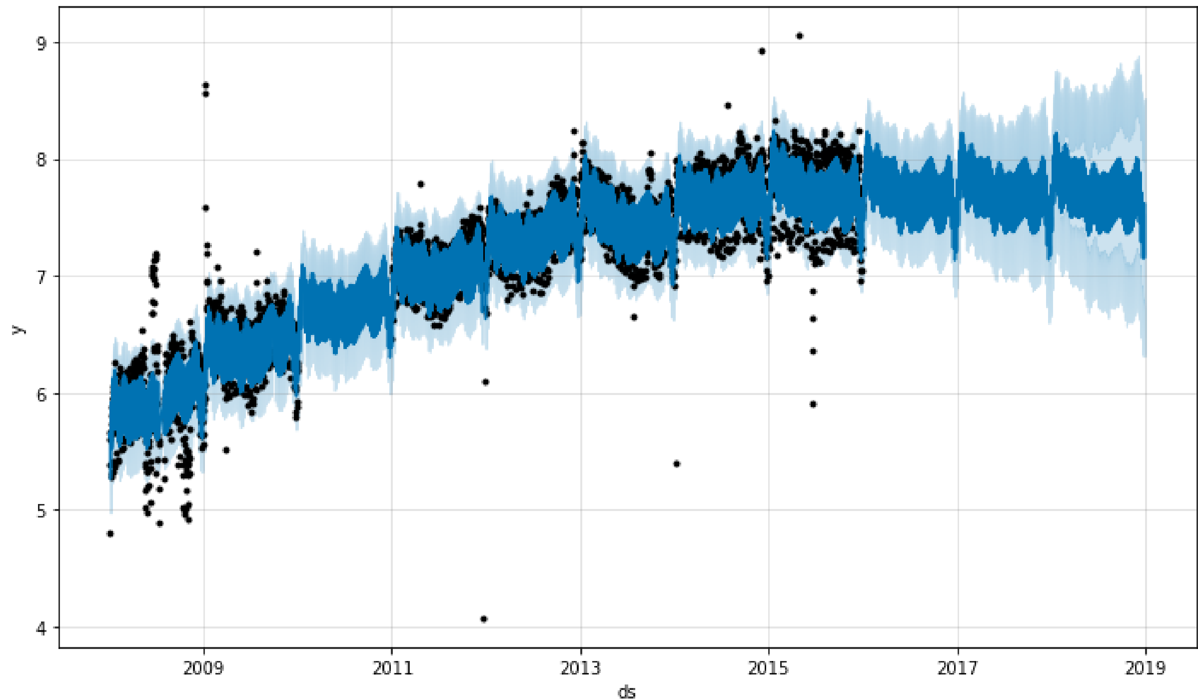
```
df = pd.read_csv('../examples/example_wp_log_R_outliers1.csv')
m = Prophet()
m.fit(df)
future = m.make_future_dataframe(periods=1096)
```

```
5 forecast = m.predict(future)
6 fig = m.plot(forecast)
```



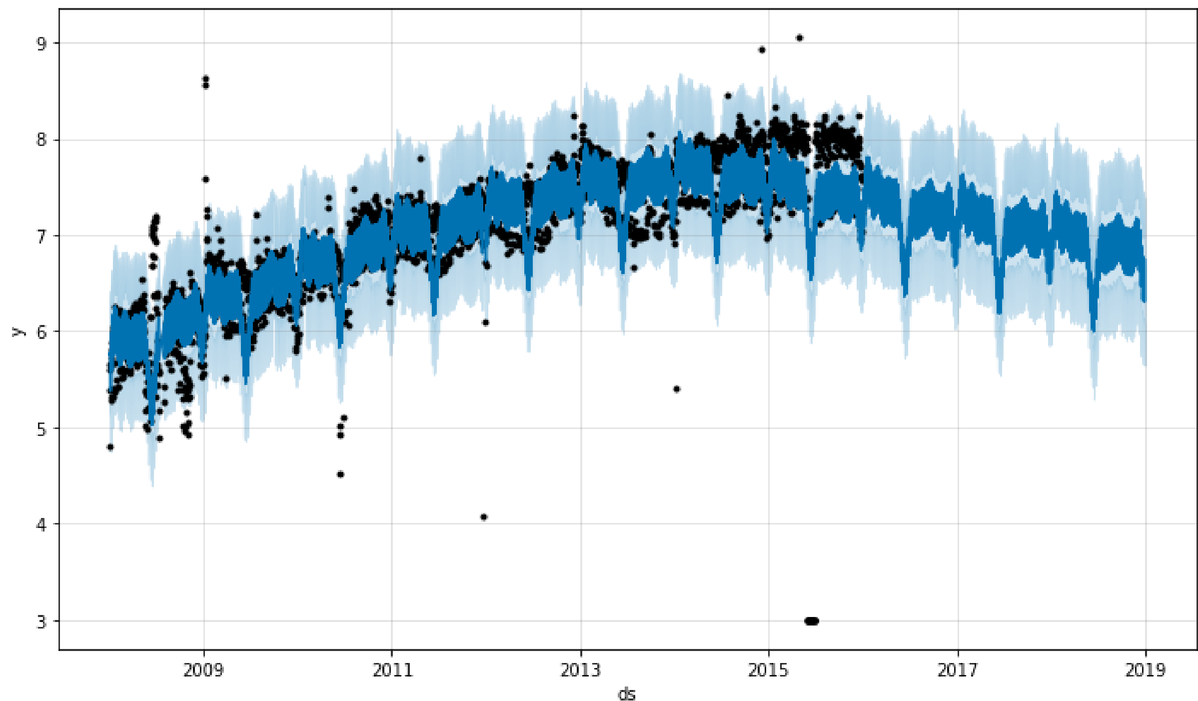
2010年のデータを除外することによって、予測が改善される。

```
1 df.loc[(df['ds'] > '2010-01-01') & (df['ds'] < '2011-01-01'),
  'y'] = None
2 model = Prophet().fit(df)
3 fig = model.plot(model.predict(future))
```



上では、外れ値を除外することによって予測が改善されたが、これがいつでも成立するとは限らない。以下の例では2015年6月に外れ値が観察される。

```
df = pd.read_csv('../examples/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)
```



今回は、外れ値を除外すると予測の幅が広がった。

```
df.loc[(df['ds'] > '2015-06-01') & (df['ds'] < '2015-06-30'), 'y'] = None
m = Prophet().fit(df)
fig = m.plot(m.predict(future))
```