Machine Learning Foundation

Course 6, Part f: SARIMA and Prophet DEMO

Learning Outcomes

You should walk away from this notebook with:

1. A practical understanding of Autoregressive Integrated Moving Average (ARIMA) models.
2. Insight into checking fit of model.
3. Learn to create forecasts with ARIMA models in Python.
4. A practical understanding of fbprophet
5. How to check fit of fbprophet model
6. Means of adjusting and improving fbprophet model parameters

Overview: Time Series Modeling Approaches

In previous lessons, we explored Python implementations of fundamental time series concepts including stationarity, smoothing, trend, seasonality, and autocorrelation, and built two kinds of models:

- **MA models**: Specify that the current value of the series depends linearly on the series' mean and a set of prior (observed) white noise error terms.
- **AR models**: Specify that the current value of the series depends linearly on its own previous values and on a stochastic term (an imperfectly predictable term).

In the current lesson we will review these concepts as well as combine these two model types into three more complicated time series models: ARMA, ARIMA, and SARIMA. We will then explore a different type of Time Series modeling using **Facebook Prophet**.

Installation notes:

Prophet holiday issues

To install **pdarima** on Anaconda, use:

conda install -c saravji pdarima

In [2]:
```
# Setup
from datetime import datetime
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import statsmodels.api as sm
```

```
import seaborn as sns
import sys, os
os.chdir('C:/Users/julia/OneDrive/Desktop/Coursera/time series Python/')
from colorsetup import colors, palette
sns.set_palette(palette)
import warnings
warnings.simplefilter(action='ignore')
import fbprophet
import pmdarima as pm
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
C:\Users\Public\Documents\Wondershare\CreatorTemp\ipykernel_5196\1423981988.py in
<module>
     13 import warnings
     14 warnings.simplefilter(action='ignore')
---> 15 import fbprophet
     16 import pmdarima as pm

ModuleNotFoundError: No module named 'fbprophet'
```

In [ ]:

```
from statsmodels.tsa.statespace.sarimax import SARIMAX
```

Section 1: Time Series Review

We've covered lots of material in the previous four lessons. Now is a good time to step back and rehash what we've covered. This will help to both solidify concepts and ensure you're ready to tackle ARMA, ARIMA, and SARIMA models.

Section 1.1: Examples of time series data and modeling:

- Hedge fund prediction of stock and index movements
- Long and short-term weather forecasting
- Business budgeting and trend analysis
- Health vitals monitoring
- Traffic flows and logistic optimization modeling
- Can you think of others?

Section 1.2: Decomposition

Time series data can often be decomposed into trend, seasonal, and random fluctuation components.

Decomposition

- Trends
  - Up
  - Down
  - Flat
  - Larger trends can be made up of smaller trends

        - There is no defined timeframe for what constitutes a trend as it depends on your goals
    - Seasonal Effects
        - Weekend retail sales spikes
        - Holiday shopping
        - Energy requirement changes with annual weather patterns
        - Note: Twitter spikes when news happens are not seasonal because they aren't regular and predictable
    - Random Fluctuations
        - The human element
        - Aggregations of small influencers
        - Observation errors
        - The smaller this is in relation to Trend and Seasonal, the better we can predict the future

Section 1.3: Additive vs Multiplicative

Time series models fall into two camps:

- Additive
    - Data = Trend + Seasonal + Random
    - What we will be using for our modeling
- Multiplicative
    - Data = Trend x Seasonal x Random
    - As easy to fit as Additive if we take the log
        - log(Data) = log(Trend x Seasonal x Random)

We should use multiplicative models when the percentage change of our data is more important than the absolute value change (e.g. stocks, commodities); as the trend rises and our values grow, we see amplitude growth in seasonal and random fluctuations. If our seasonality and fluctuations are stable, we likely have an additive model.

Section 1.4: Time Series Modeling Process

Time series model selection is driven by the Trend and Seasonal components of our raw data. The general approach for analysis looks like this:

1. Plot the data and determine Trends and Seasonality
    A. Difference or take the log of the data (multiple times if needed) to remove trends for certain model applications
    B. Stationairity is needed for ARMA models
2. Determine if we have additive or multiplicative data patterns
3. Select the appropriate algorithm based on the chart below
4. Determine if model selection is correct with these tools
    - Ljung-Box Test
    - Residual Errors (Normal Distribution with zero mean and constant variance-homoskedastic, i.i.d)
    - Autocorrelation Function (ACF)

- Partial Autocorrelation Function (PACF)

Algorithm	Trend	Seasonal	Correlations
ARIMA	X	X	X
SMA Smoothing	X		
Simple Exponential Smoothing	X		
Seasonal Adjustment	X	X	
Holt's Exponential Smoothing	X		
Holt-Winters	X	X	

Section 1.5: How to Achieve and Test for Stationarity

- The mean of the series is not a function of time.
- The variance of the series is not a function of time (homoscedasticity).
- The covariance at different lags is not a function of time.

From A Complete Tutorial on Time Series Modeling in R

- Info on stationarity
- Plotting Rolling Statistics
  - Plot the moving average/variance and see if it changes with time. This visual technique can be done on different windows, but isn't as rigorously defensible as the test below.
- Augmented Dickey-Fuller Test
  - Statistical tests for checking stationarity; the null hypothesis is that the TS is non-stationary. If our test statistic is below an `alpha` value, we *can* reject the null hypothesis and say that the series is stationary.

$$Y_t = \rho * Y_{t-1} + \epsilon_t$$

$$Y_t - Y_{t-1} = (\rho - 1)Y_{t-1} + \epsilon_t$$

Section 1.6: Differencing Example

This will give us a reminder to how differencing is used to get a stationary series which will be essential to the final piece of the ARIMA model

In [3]:
```
# create a play dataframe from 1-10 (linear and squared) to test how differencing w
play = pd.DataFrame([[x for x in range(1,11)], [x**2 for x in range(1,11)]]).T
play.columns = ['original', 'squared']
play
```

Out[3]:

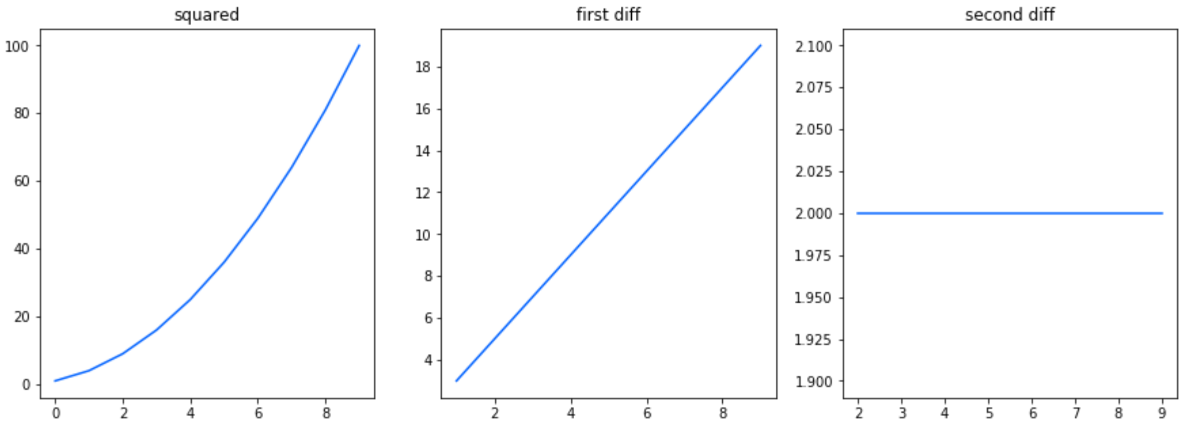
	original	squared
0	1	1
1	2	4
2	3	9
3	4	16
4	5	25
5	6	36
6	7	49
7	8	64
8	9	81
9	10	100

In [4]:
```
# stationarize linear series (mean and variance don't change for sub-windows)
play.original.diff()
```

Out[4]:
```
0    NaN
1    1.0
2    1.0
3    1.0
4    1.0
5    1.0
6    1.0
7    1.0
8    1.0
9    1.0
Name: original, dtype: float64
```

In [5]:
```
fig,axes = plt.subplots(1,3,figsize = (15,5))
axes[0].plot(play.squared)
axes[0].set_title('squared')
axes[1].plot(play.squared.diff())
axes[1].set_title('first diff')
axes[2].plot(play.squared.diff().diff())
axes[2].set_title('second diff')
```

Out[5]:
```
Text(0.5, 1.0, 'second diff')
```



NOTE: This is similar to taking a first-order derivative.

In [6]:
```
# stationarize squared series
play.squared.diff().diff()
```

Out[6]:
```
0    NaN
1    NaN
2    2.0
3    2.0
4    2.0
5    2.0
6    2.0
7    2.0
8    2.0
9    2.0
Name: squared, dtype: float64
```

NOTE: Notice we need to difference twice on an exponential trend, and every time we do, we lose a bit of data

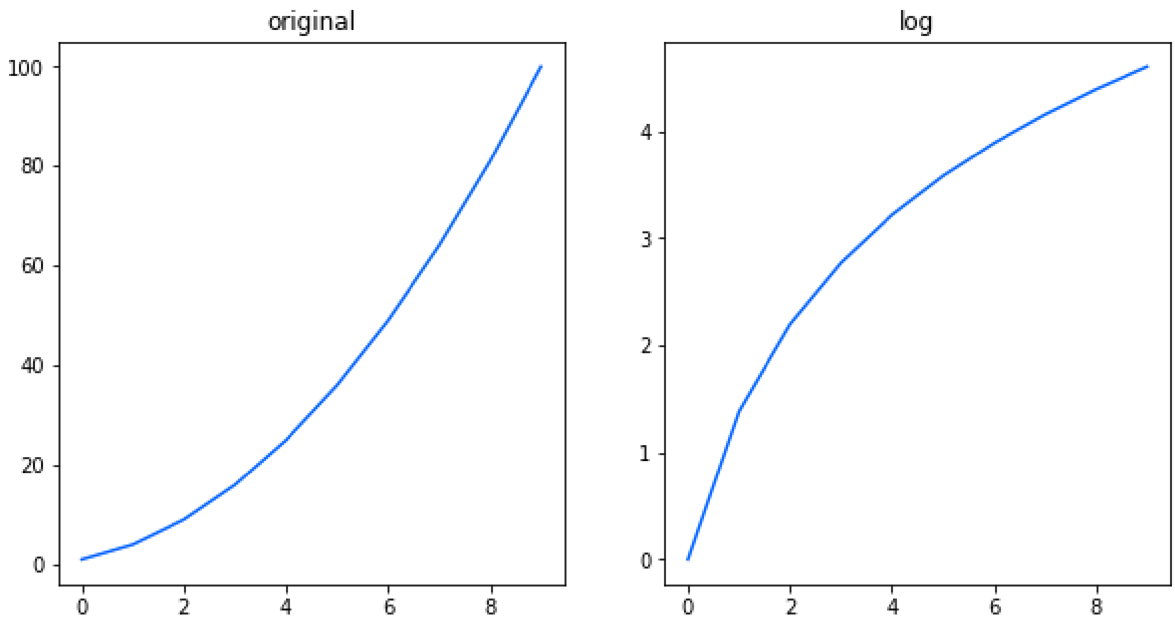
In [7]:
```
# stationarize squared with log
np.log(play.squared)
```

Out[7]:
```
0    0.000000
1    1.386294
2    2.197225
3    2.772589
4    3.218876
5    3.583519
6    3.891820
7    4.158883
8    4.394449
9    4.605170
Name: squared, dtype: float64
```

NOTE: Works somewhat but certainly not as well.

In [8]:
```
fig,axes = plt.subplots(1,2,figsize = (10,5))
axes[0].plot(play.squared)
axes[0].set_title('original')
axes[1].plot(np.log(play.squared))
axes[1].set_title('log')
```

Out[8]:
```
Text(0.5, 1.0, 'log')
```



Data Prep and EDA

We'll be looking at monthly average temperatures between 1907-1972

In [9]:
```
# load data and convert to datetime
monthly_temp = pd.read_csv('./mean-monthly-temperature-1907-19.csv',
                           skipfooter=2,
                           infer_datetime_format=True,
                           header=0,
                           index_col=0, engine='python',
                           names=['month', 'temp'])

monthly_temp.index = pd.to_datetime(monthly_temp.index)
```

In [10]:
```
monthly_temp.head()
```

Out[10]:

	temp
month	
1907-01-01	33.3
1907-02-01	46.0
1907-03-01	43.0
1907-04-01	55.0
1907-05-01	51.8

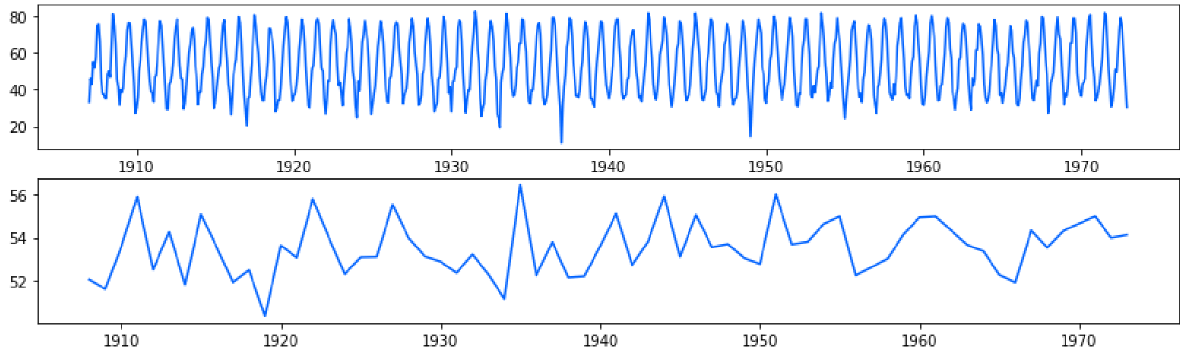
In [11]:
```
# describe
monthly_temp.describe()
```

Out[11]:

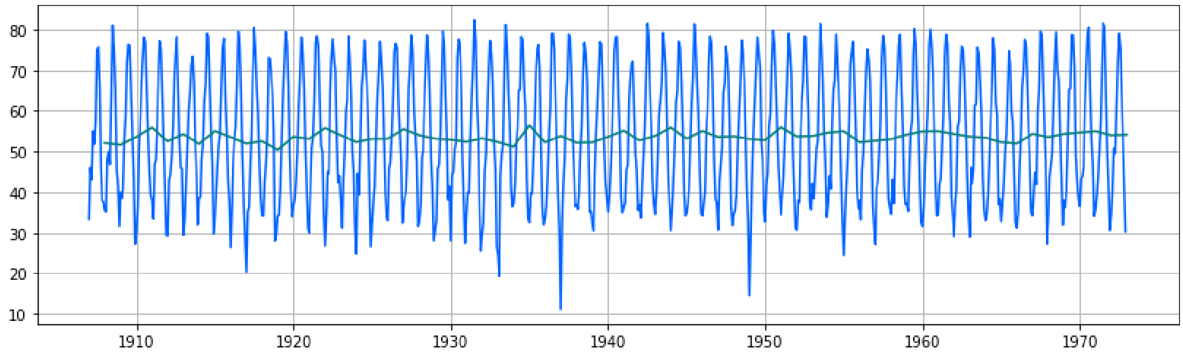
	temp
count	792.000000
mean	53.553662
std	15.815452
min	11.200000
25%	39.675000
50%	52.150000
75%	67.200000
max	82.400000

In [12]:
```
# resample to annual and plot each
plt.rcParams['figure.figsize'] = [14, 4]
annual_temp = monthly_temp.resample('A').mean()
fig, axes = plt.subplots(2,1)
axes[0].plot(monthly_temp)
axes[1].plot(annual_temp)
```

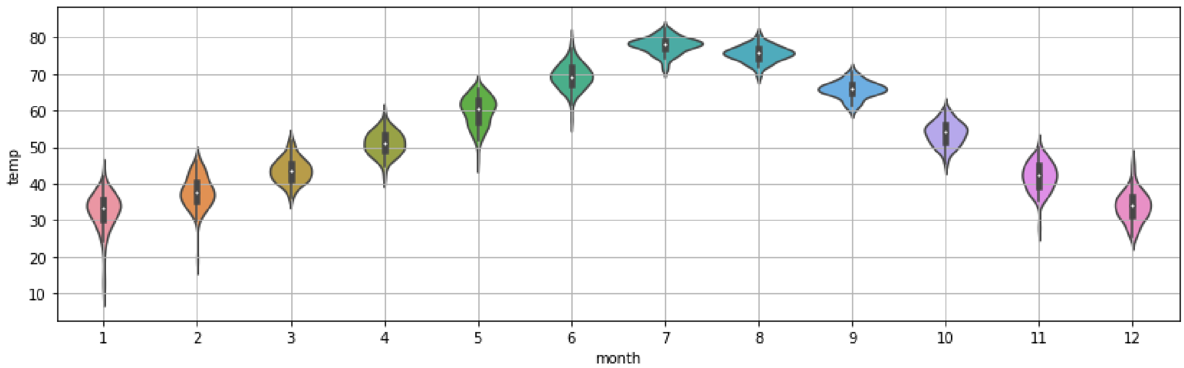
Out[12]:
```
[<matplotlib.lines.Line2D at 0x7fc82f073690>]
```



In [13]:
```
# plot both on same figure
plt.plot(monthly_temp)
plt.plot(annual_temp)
plt.grid(b=True);
```



In [14]:
```
# violinplot of months to determine variance and range
sns.violinplot(x=monthly_temp.index.month, y=monthly_temp.temp)
plt.grid(b=True);
```



In [15]:
```
# split data into 10 chunks
chunks = np.split(monthly_temp.temp, indices_or_sections=12)
```

In [16]:
```
mean_vals = np.mean(chunks,axis=1)
var_vals = np.var(chunks,axis=1)
vals = {'mean_vals': mean_vals , 'var_vals': var_vals}
mean_var = pd.DataFrame(vals)
mean_var
```

Out[16]:

	mean_vals	var_vals
0	52.674242	225.907973
1	53.654545	246.031570
2	52.837879	247.400838
3	54.078788	250.787429
4	52.439394	277.865721
5	53.457576	283.619412
6	53.718182	224.882397
7	54.422727	265.442059
8	53.457576	238.561230
9	54.425758	257.425852
10	52.613636	219.026026
11	54.863636	253.827466

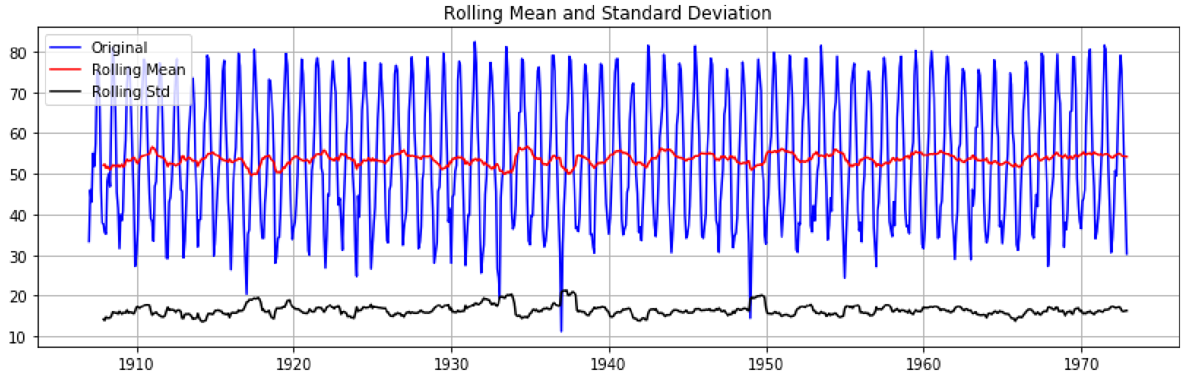
In [17]:

```
# define Dickey-Fuller Test (DFT) function
# Null is that unit root is present, rejection means likely stationary
import statsmodels.tsa.stattools as ts
def dftest(timeseries):
    dftest = ts.adfuller(timeseries,)
    dfoutput = pd.Series(dftest[0:4],
                         index=['Test Statistic','p-value','Lags Used','Observatio
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print(dfoutput)
    #Determing rolling statistics
    rolmean = timeseries.rolling(window=12).mean()
    rolstd = timeseries.rolling(window=12).std()

    #Plot rolling statistics:
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean and Standard Deviation')
    plt.grid()
    plt.show(block=False)
```

In [18]:

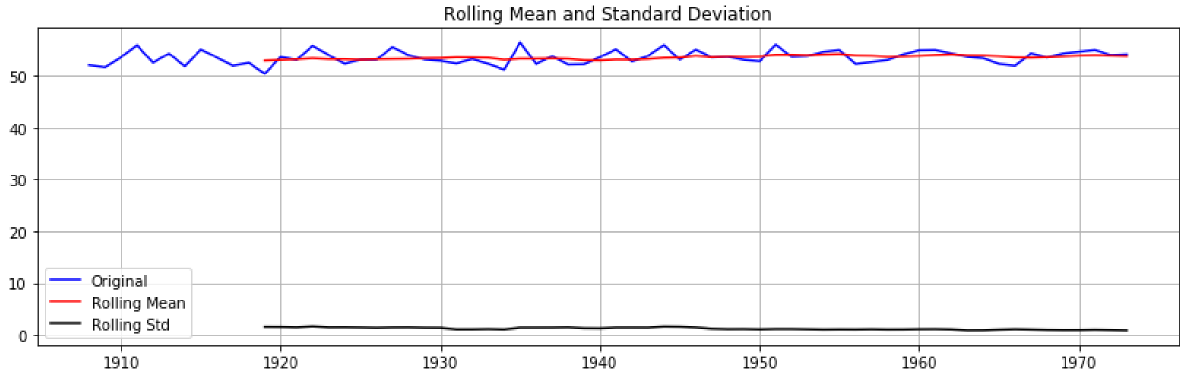
```
# run DFT on monthly
dftest(monthly_temp.temp)
# p-value allows us to reject a unit root: data is stationary
```

```
Test Statistic          -6.481466e+00
p-value                  1.291867e-08
Lags Used                2.100000e+01
Observations Used        7.700000e+02
Critical Value (1%)     -3.438871e+00
Critical Value (5%)     -2.865301e+00
Critical Value (10%)    -2.568773e+00
dtype: float64
```



In [19]:
```
# run DFT on annual
dftest(annual_temp.temp)
```

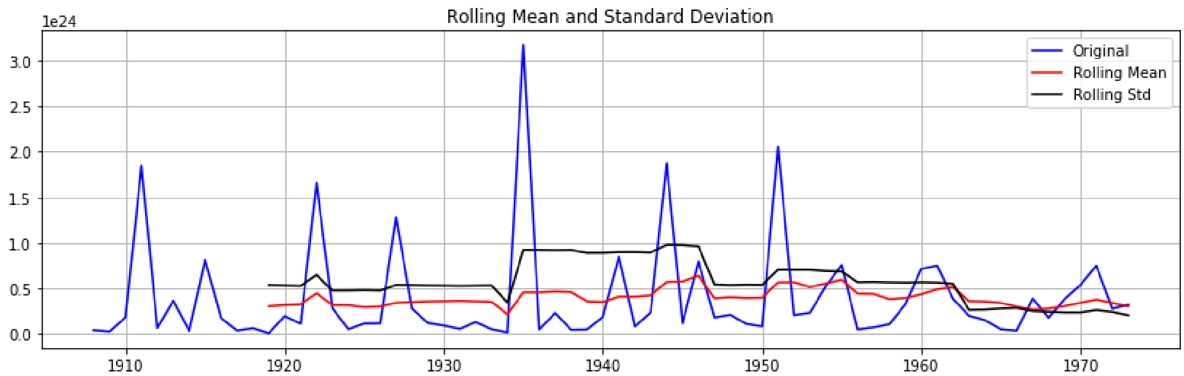
```
Test Statistic         -7.878242e+00
p-value                 4.779473e-12
Lags Used               0.000000e+00
Observations Used       6.500000e+01
Critical Value (1%)    -3.535217e+00
Critical Value (5%)    -2.907154e+00
Critical Value (10%)   -2.591103e+00
dtype: float64
```



The p-value allows us to *reject* a unit root (i.e. the data is stationary).

In [20]:
```
# here's an example of non-stationary with DFT results
dftest(np.exp(annual_temp.temp))
```

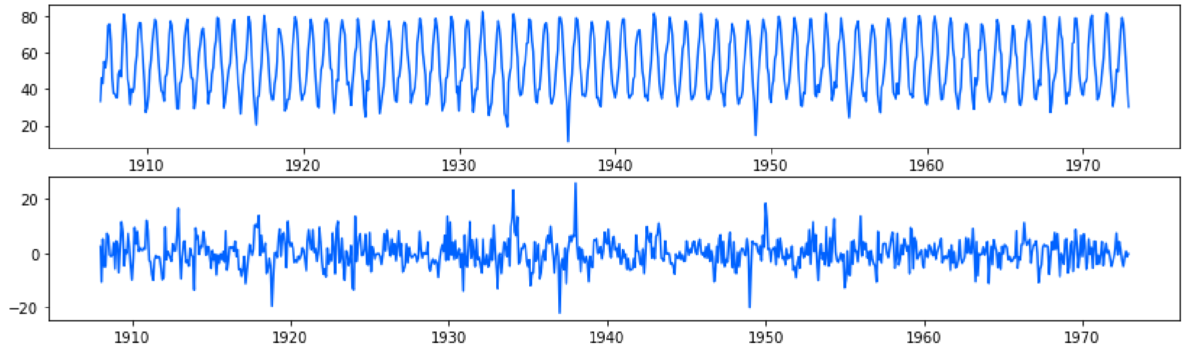
```
Test Statistic         -0.449458
p-value                 0.901508
Lags Used              10.000000
Observations Used      55.000000
Critical Value (1%)    -3.555273
Critical Value (5%)    -2.915731
Critical Value (10%)   -2.595670
dtype: float64
```



In [21]:
```
# Important to note that values have strong seasonality and adf test as well as ro
# That is why it is always important to pay attention to run sequence plot
monthly_temp['lag_12'] = monthly_temp.shift(12)
monthly_temp['seasonal_diff'] = monthly_temp.temp - monthly_temp['lag_12']

fig,axes = plt.subplots(2,1)
axes[0].plot(monthly_temp.temp,label ='original')
axes[1].plot(monthly_temp.seasonal_diff,label = 'seasonal diff')
```

```
INFO:numexpr.utils:NumExpr defaulting to 8 threads.
```

Out[21]:
```
[<matplotlib.lines.Line2D at 0x7fc8150e5850>]
```



Section 2: SARIMA with Statsmodels

We went through getting stationary data and differencing as that is the last piece of the puzzle that we are missing for understanding ARIMA models. The I stands for "Integrated" which just refers to the amount of differcing done on the data.

When we are determining our ARIMA model we will come across the following standard inputs:

- order(p,d,q):
  - p is number of AR terms
  - d is number of times that we would difference our data
  - q is number of MA terms

When we work with SARIMA models 'S' refers to 'seasonal' and we have the additional standard inputs:

- seasonal order(p,d,q):
  - p is number of AR terms in regards to seasonal lag
  - d is number of times that we would difference our seasonal lag (as seen above)
  - q is number of MA terms in regards to seasonal lag
  - s is number of periods in a season

Reminder of some good resources:

- ARIMA in R
- Duke ARIMA Guide
- Great explanation on MA in practice

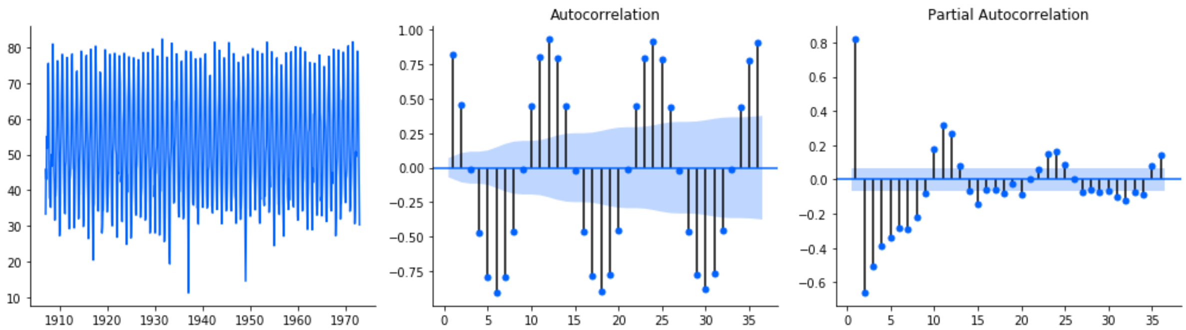
Some rules to highlight from the Duke ARIMA Guide:

1. If the series has positive autocorrelations out to a high number of lags, then it probably needs a higher order of differencing
2. If the lag-1 autocorrelation is zero or negative, or the autocorrelations are all small and patternless, then the series does not need a higher order of differencing. If the lag-1 autocorrelation is -0.5 or more negative, the series may be overdifferenced. BEWARE OF OVERDIFFERENCING!!
3. A model with no orders of differencing assumes that the original series is stationary (mean-reverting). A model with one order of differencing assumes that the original series has a constant average trend (e.g. a random walk or SES-type model, with or without growth). A model with two orders of total differencing assumes that the original series has a time-varying trend (e.g. a random trend or LES-type model)

Create Helper Functions

In [22]:
```
# define helper plot function for visualization
def plots(data, lags=None):
    layout = (1, 3)
    raw  = plt.subplot2grid(layout, (0, 0))
    acf  = plt.subplot2grid(layout, (0, 1))
    pacf = plt.subplot2grid(layout, (0, 2))

    raw.plot(data)
    sm.tsa.graphics.plot_acf(data, lags=lags, ax=acf, zero=False)
    sm.tsa.graphics.plot_pacf(data, lags=lags, ax=pacf, zero = False)
    sns.despine()
    plt.tight_layout()
```

In [23]:
```
# helper plot for monthly temps
plots(monthly_temp.temp, lags=36);
# open Duke guide for visual
# we note a 12-period cycle (yearly) with suspension bridge design, so must use SAI
```



Box-Jenkins Method

ACF Shape	Indicated Model
Exponential, decaying to zero	Autoregressive model. Use the partial autocorrelation plot to identify the order of the autoregressive model.
Alternating positive and negative, decaying to zero	Autoregressive model. Use the partial autocorrelation plot to help identify the order.
One or more spikes, rest are essentially zero	Moving average model, order identified by where plot becomes zero.
Decay, starting after a few lags	Mixed autoregressive and moving average (ARMA) model.

ACF Shape	Indicated Model
All zero or close to zero	Data are essentially random.
High values at fixed intervals	Include seasonal autoregressive term.
No decay to zero	Series is not stationary.

In [24]:
```
# we might need to install dev version for statespace functionality
#!pip install git+https://github.com/statsmodels/statsmodels.git

# fit SARIMA monthly based on helper plots
sar = sm.tsa.statespace.SARIMAX(monthly_temp.temp,
                                order=(1,0,0),
                                seasonal_order=(0,1,1,12),
                                trend='c').fit()
sar.summary()
```

Out[24]:

SARIMAX Results

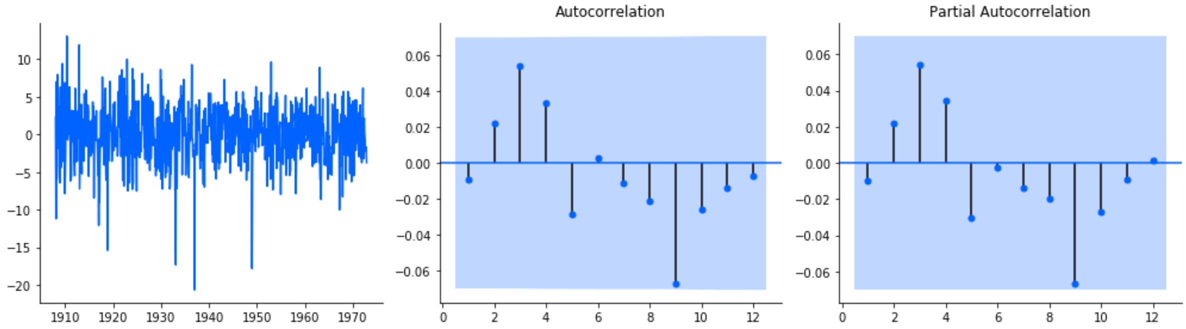
Dep. Variable:	temp	**No. Observations:**	792
Model:	SARIMAX(1, 0, 0)x(0, 1, [1], 12)	**Log Likelihood**	-2128.873
Date:	Tue, 16 Jun 2020	**AIC**	4265.746
Time:	11:28:04	**BIC**	4284.383
Sample:	01-01-1907	**HQIC**	4272.914
	- 12-01-1972		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	0.0127	0.007	1.697	0.090	-0.002	0.027
ar.L1	0.1791	0.035	5.105	0.000	0.110	0.248
ma.S.L12	-0.9996	1.368	-0.731	0.465	-3.681	1.681
sigma2	12.8928	17.515	0.736	0.462	-21.437	47.222

Ljung-Box (Q):	27.91	**Jarque-Bera (JB):**	252.77
Prob(Q):	0.93	**Prob(JB):**	0.00
Heteroskedasticity (H):	0.71	**Skew:**	-0.56
Prob(H) (two-sided):	0.01	**Kurtosis:**	5.55

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

In [25]:
```
# plot resids
plots(sar.resid[sar.loglikelihood_burn:], lags=12);
```



Thought process:

010010 is probably overdifferenced as we can see by negative ACF at lag 1

000010 is a big underdiff at seasonal lag, but with better AIC

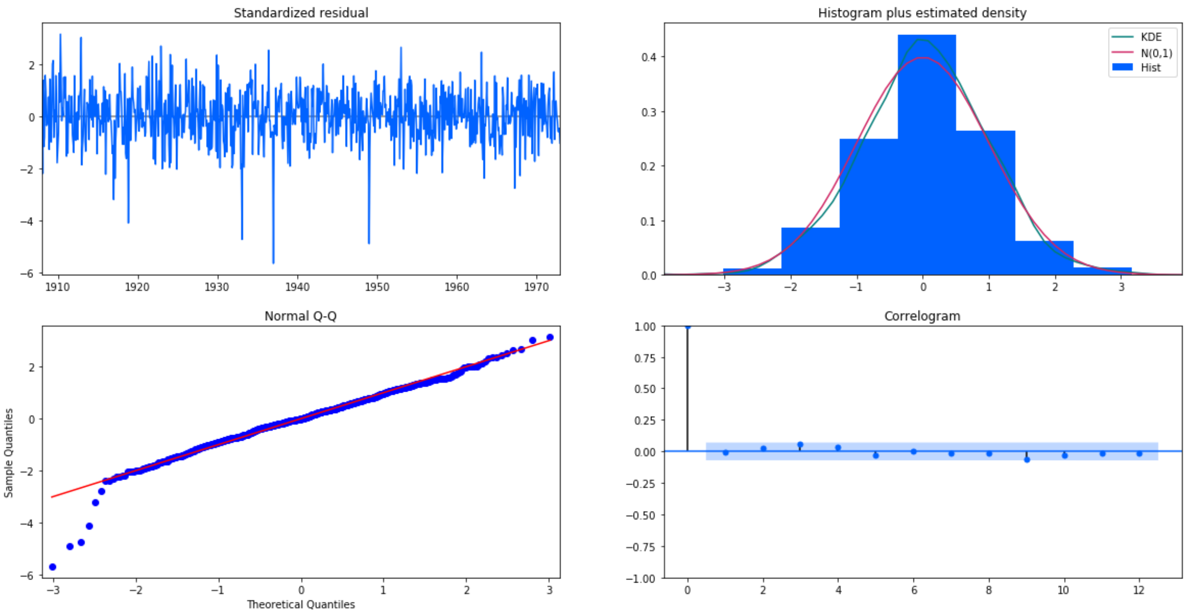
Looks like 000010,12 and Trend='c' per rule

Looking back at seasonal we notice negative ACR spike at 12: we will thus add a SMA term and we see a big drop in AIC to 4289

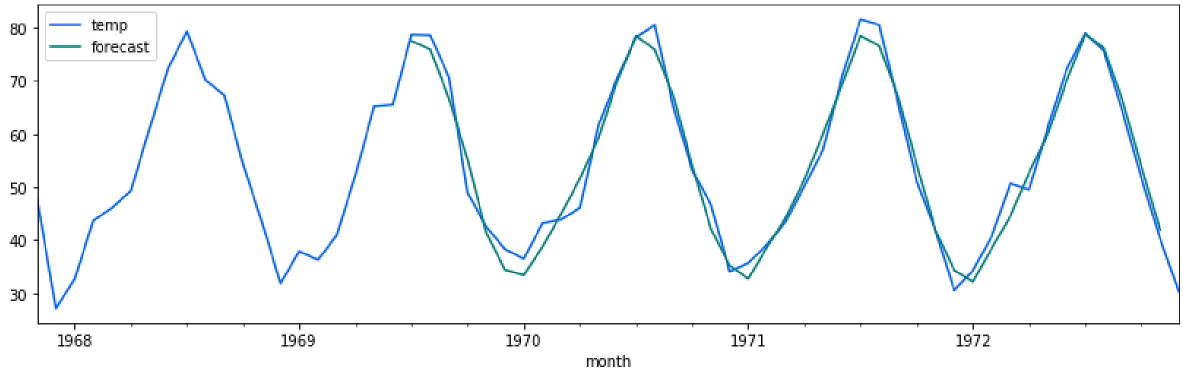
looks like ACF looks good at seasonal lags, so we move back to ARIMA portion.

ACF shows we can use AR terms. AR=1,2 or 3 have similar AIC

In [26]:
```
# plot residual diagnostics
sar.plot_diagnostics(lags=12,figsize = (20,10),);
```

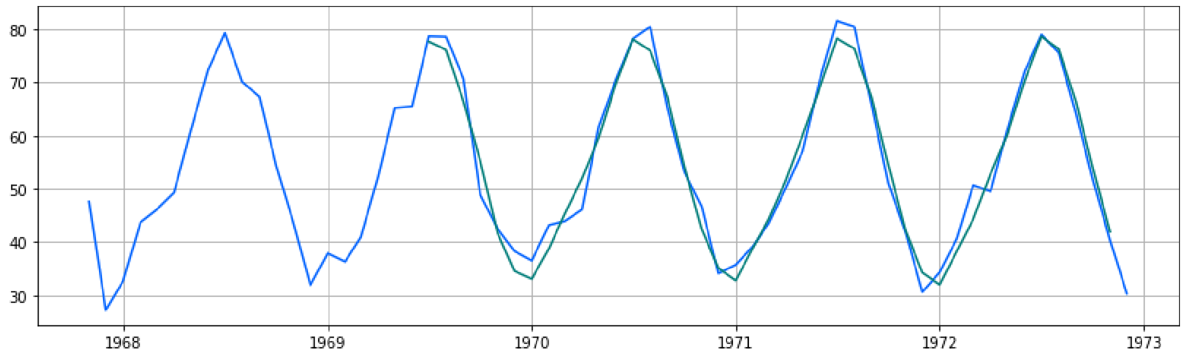


In [27]:
```
# plot predictions
pd.plotting.register_matplotlib_converters()
#use model.predict() start and end in relation to series
monthly_temp['forecast'] = sar.predict(start = 750, end= 790)
monthly_temp[730:][['temp', 'forecast']].plot();
```

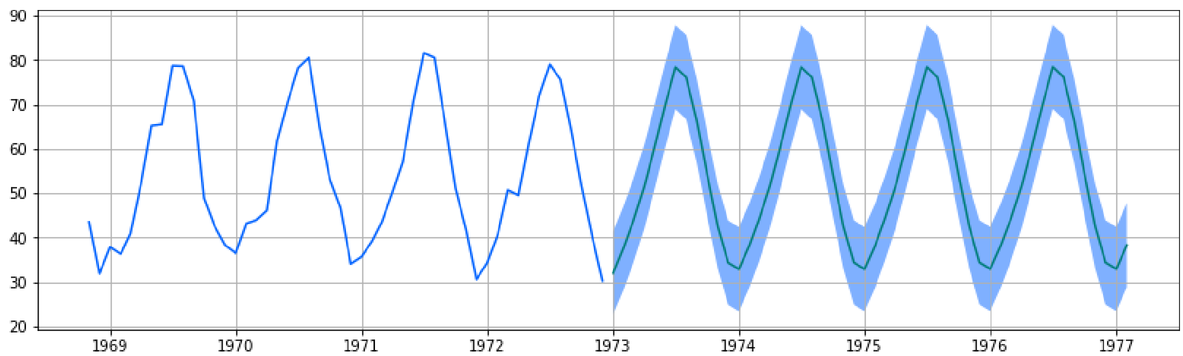


In [28]:
```
#Introducing another model
sar2 = sm.tsa.statespace.SARIMAX(monthly_temp.temp,
                                order=(3,0,0),
                                seasonal_order=(0,1,1,12),
                                trend='c').fit()
```

In [29]:
```
# plot predictions
monthly_temp['forecast'] = sar2.predict(start = 750, end= 790, dynamic=False)
plt.plot(monthly_temp[730:][['temp', 'forecast']])
plt.grid();
```



In [30]:
```
# can use get forecast to create a forecast object
future_fcst = sar2.get_forecast(50)
# That will have a method to pull in confidence interval
confidence_int = future_fcst.conf_int(alpha = 0.01)
# Has an attribute to pull in predicted mean
fcst = future_fcst.predicted_mean
# Plot predictions and confidence intervals
plt.plot(monthly_temp.temp[-50:])
plt.plot(fcst)
plt.fill_between(confidence_int.index,confidence_int['lower temp'],confidence_int[
plt.grid()
```



Section 3: Statistical Tests

- Normality (Jarque-Bera)
    - Null hypothesis is normally distributed residuals (good, plays well with RMSE and similar error metrics)
- Serial correlation (Ljung-Box)
    - Null hypothesis is no serial correlation in residuals (independent of each other)
- Heteroskedasticity
    - Tests for change in variance between residuals.
    - The null hypothesis is no heteroskedasticity. That means different things depending on which alternative is selected:
        - Increasing: Null hypothesis is that the variance is not increasing throughout the sample; that the sum-of-squares in the later subsample is not greater than the sum-of-squares in the earlier subsample.
        - Decreasing: Null hypothesis is that the variance is not decreasing throughout the sample; that the sum-of-squares in the earlier subsample is not greater than the sum-of-squares in the later subsample.
        - Two-sided (default): Null hypothesis is that the variance is not changing throughout the sample. Both that the sum-of-squares in the earlier subsample is not greater than the sum-of-squares in the later subsample and that the sum-of-squares in the later subsample is not greater than the sum-of-squares in the earlier subsample.
- Durbin Watson
    - Tests autocorrelation of residuals: we want between 1-3, 2 is ideal (no serial correlation)

In [31]:
```
sar.test_normality?
```

```
Signature: sar.test_normality(method)
Docstring:
Test for normality of standardized residuals.

Null hypothesis is normality.

Parameters
----------
method : {'jarquebera', None}
    The statistical test for normality. Must be 'jarquebera' for
    Jarque-Bera normality test. If None, an attempt is made to select
    an appropriate test.

Notes
-----
Let `d` = max(loglikelihood_burn, nobs_diffuse); this test is
calculated ignoring the first `d` residuals.

In the case of missing data, the maintained hypothesis is that the
data are missing completely at random. This test is then run on the
standardized residuals excluding those corresponding to missing
observations.

See Also
--------
statsmodels.stats.stattools.jarque_bera
    The Jarque-Bera test of normality.
File:      /Applications/anaconda3/lib/python3.7/site-packages/statsmodels/tsa/sta
tespace/mlemodel.py
Type:      method
```

In [32]:

```
# create and run statistical tests on model
norm_val, norm_p, skew, kurtosis = sar.test_normality('jarquebera')[0]
lb_val, lb_p = sar.test_serial_correlation(method='ljungbox',)[0]
het_val, het_p = sar.test_heteroskedasticity('breakvar')[0]


# we want to look at largest lag for Ljung-Box, so take largest number in series
# there's intelligence in the method to determine how many lags back to calculate
lb_val = lb_val[-1]
lb_p = lb_p[-1]
durbin_watson = sm.stats.stattools.durbin_watson(
    sar.filter_results.standardized_forecasts_error[0, sar.loglikelihood_burn:])

print('Normality: val={:.3f}, p={:.3f}'.format(norm_val, norm_p));
print('Ljung-Box: val={:.3f}, p={:.3f}'.format(lb_val, lb_p));
print('Heteroskedasticity: val={:.3f}, p={:.3f}'.format(het_val, het_p));
print('Durbin-Watson: d={:.2f}'.format(durbin_watson))
```

```
Normality: val=252.770, p=0.000
Ljung-Box: val=27.915, p=0.925
Heteroskedasticity: val=0.708, p=0.006
Durbin-Watson: d=2.01
```

Note on autofit methods

R has an autoARIMA function (and other automagic methods) that gridsearches/optimizes our model hyperparameters for us. Over time, more of these goodies are porting to Python (e.g. pmdarima). While there's nothing wrong with utilizing these resources, the *human makes the final determination!* Don't become over-reliant on these methods, especially early on when you are grasping the underlying mechanics and theory!

In [33]:
```
#from pyramid.arima import auto_arima
stepwise_model = pm.auto_arima(monthly_temp.temp, start_p=1, start_q=1,
                           max_p=3, max_q=3, m=12,
                           start_P=0, seasonal=True,
                           d=0, D=1, trace=True,
                           error_action='ignore',
                           suppress_warnings=True,
                           stepwise=True)
print(stepwise_model.aic())
```

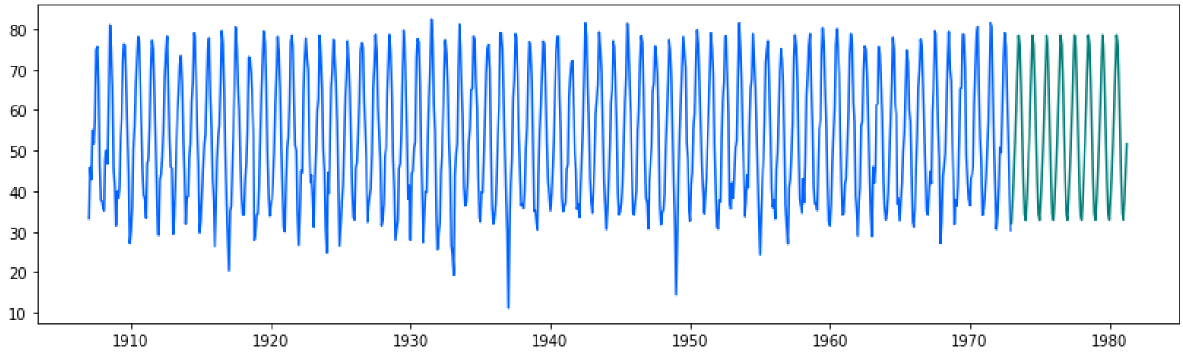
```
Fit ARIMA: order=(1, 0, 1) seasonal_order=(0, 1, 1, 12); AIC=4265.781, BIC=4289.07
7, Fit time=2.590 seconds
Fit ARIMA: order=(0, 0, 0) seasonal_order=(0, 1, 0, 12); AIC=4796.839, BIC=4806.15
8, Fit time=0.040 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(1, 1, 0, 12); AIC=4546.179, BIC=4564.81
6, Fit time=0.664 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 1, 1, 12); AIC=4267.567, BIC=4286.20
4, Fit time=1.673 seconds
Fit ARIMA: order=(1, 0, 1) seasonal_order=(1, 1, 1, 12); AIC=4267.267, BIC=4295.22
3, Fit time=3.281 seconds
Fit ARIMA: order=(1, 0, 1) seasonal_order=(0, 1, 0, 12); AIC=4767.543, BIC=4786.18
0, Fit time=0.271 seconds
Fit ARIMA: order=(1, 0, 1) seasonal_order=(0, 1, 2, 12); AIC=4267.235, BIC=4295.19
0, Fit time=8.678 seconds
Fit ARIMA: order=(1, 0, 1) seasonal_order=(1, 1, 2, 12); AIC=4269.045, BIC=4301.66
0, Fit time=7.571 seconds
Fit ARIMA: order=(2, 0, 1) seasonal_order=(0, 1, 1, 12); AIC=4267.467, BIC=4295.42
3, Fit time=1.955 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(0, 1, 1, 12); AIC=4265.746, BIC=4284.38
3, Fit time=1.690 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(1, 1, 1, 12); AIC=4267.159, BIC=4290.45
5, Fit time=1.953 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(0, 1, 0, 12); AIC=4769.737, BIC=4783.71
5, Fit time=0.112 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(0, 1, 2, 12); AIC=4267.125, BIC=4290.42
2, Fit time=5.787 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(1, 1, 2, 12); AIC=4269.204, BIC=4297.16
0, Fit time=5.260 seconds
Fit ARIMA: order=(0, 0, 0) seasonal_order=(0, 1, 1, 12); AIC=4289.161, BIC=4303.13
9, Fit time=1.121 seconds
Fit ARIMA: order=(2, 0, 0) seasonal_order=(0, 1, 1, 12); AIC=4266.874, BIC=4290.17
0, Fit time=2.168 seconds
Total fit time: 44.818 seconds
4265.745875119801
```

In [34]:
```
from dateutil.relativedelta import relativedelta
def future_preds_df(model,series,num_months):
    pred_first = series.index.max()+relativedelta(months=1)
    pred_last = series.index.max()+relativedelta(months=num_months)
    date_range_index = pd.date_range(pred_first,pred_last,freq = 'MS')
    vals = model.predict(n_periods = num_months)
    return pd.DataFrame(vals,index = date_range_index)
```

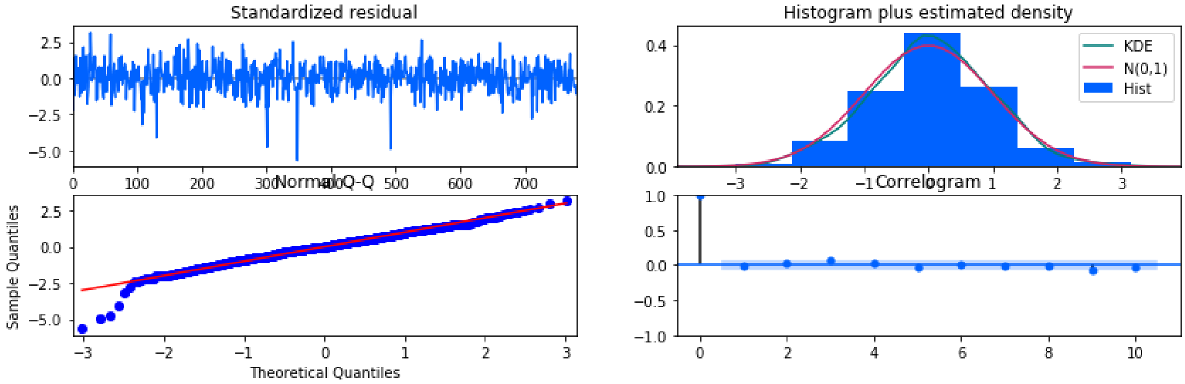
In [35]:
```
preds = future_preds_df(stepwise_model,monthly_temp.temp,100)
```

In [36]:
```
plt.plot(monthly_temp.temp)
plt.plot(preds)
```

Out[36]:
```
[<matplotlib.lines.Line2D at 0x7fc814e341d0>]
```



In [37]:
```
stepwise_model.plot_diagnostics();
```



In [38]:
```
print('auto-fit order: :', stepwise_model.order)
print('auto-fit seasonal_order: :', stepwise_model.seasonal_order)
```

```
auto-fit order: : (1, 0, 0)
auto-fit seasonal_order: : (0, 1, 1, 12)
```

When deciding on a model, often what truly matters is how well we would be able to produce out of sample predictions. Here we create a function that looks at multiple out of sample predictions to see which model had lowest out of sample error

In [39]:
```
def cross_validate(series,horizon,start,step_size,order = (1,0,0),seasonal_order =
    '''
    Function to determine in and out of sample testing of arima model

    arguments
    ---------
    series (seris): time series input
    horizon (int): how far in advance forecast is needed
    start (int): starting location in series
    step_size (int): how often to recalculate forecast
    order (tuple): (p,d,q) order of the model
    seasonal_order (tuple): (P,D,Q,s) seasonal order of model

    Returns
    -------
    DataFrame: gives fcst and actuals with date of prediction
    '''
    fcst = []
    actual = []
    date = []
    for i in range(start,len(series)-horizon,step_size):
        model = sm.tsa.statespace.SARIMAX(series[:i+1], #only using data through to
                                        order=order,
                                        seasonal_order=seasonal_order,
                                        trend=trend).fit()
        fcst.append(model.forecast(steps = horizon)[-1]) #forecasting horizon steps
```

```
        actual.append(series[i+horizon]) # comparing that to actual value at that p
        date.append(series.index[i+horizon]) # saving date of that value
    return pd.DataFrame({'fcst':fcst,'actual':actual},index=date)
```

In [40]:
```
warnings.filterwarnings("ignore")
series = monthly_temp.temp
horizon = 12
start = 700
step_size = 3
order = (1,0,0)
seasonal_order = (0,1,1,12)

cv1 = cross_validate(monthly_temp.temp,12,700,3,
                     order = order,
                     seasonal_order = seasonal_order)
```

In [41]:
```
#example to see underpinning of cv


model = sm.tsa.statespace.SARIMAX(series[:701], #only using data through start of 1
                                  order=order,
                                  seasonal_order=seasonal_order,
                                  trend=None).fit()
```

In [42]:
```
#end of input
series[:701].tail()
```

Out[42]:
```
month
1965-01-01    32.8
1965-02-01    37.8
1965-03-01    42.0
1965-04-01    49.8
1965-05-01    54.5
Name: temp, dtype: float64
```

In [43]:
```
#value to predict horizon steps into the future
series[712:713]
```

Out[43]:
```
month
1966-05-01    65.6
Name: temp, dtype: float64
```

In [44]:
```
# what model predicted for that date
model.forecast(12)[-1:]
```

Out[44]:
```
1966-05-01    59.580988
Freq: MS, dtype: float64
```

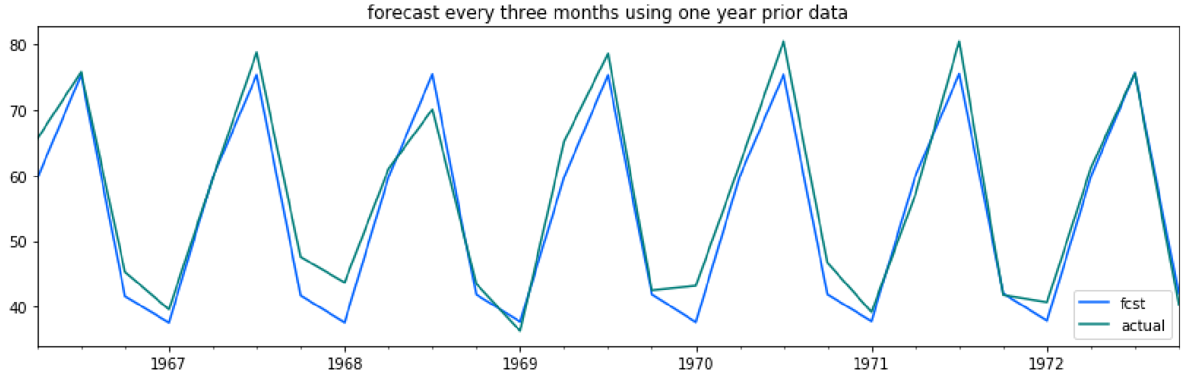
In [45]:
```
cv1.head()
```

Out[45]:

	fcst	actual
1966-05-01	59.580988	65.6
1966-08-01	75.351411	75.8
1966-11-01	41.607102	45.3
1967-02-01	37.584439	39.6
1967-05-01	59.753915	59.6

In [46]:
```
cv1.plot(title = 'forecast every three months using one year prior data')
```

Out[46]: <matplotlib.axes._subplots.AxesSubplot at 0x7fc81aa859d0>



In [47]:
```
#Defining an error metric to see out of sample accuracy
def mape(df_cv):
    return abs(df_cv.actual - df_cv.fcst).sum() / df_cv.actual.sum()
```

In [48]:
```
mape(cv1)
```

Out[48]: 0.05214814790939277

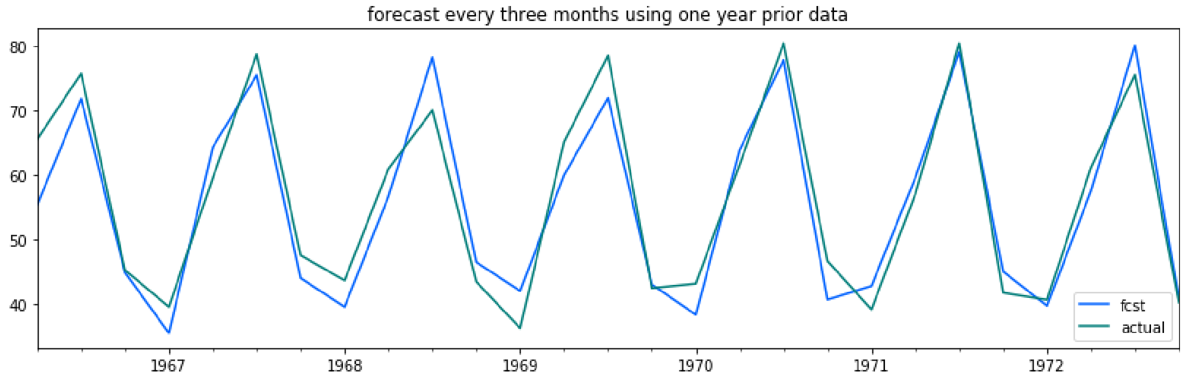
In [49]:
```
warnings.filterwarnings("ignore")
series = monthly_temp.temp
horizon = 12
start = 700
step_size = 3
order = (1,1,0)
seasonal_order = (0,1,1,12)

cv2 = cross_validate(monthly_temp.temp,12,700,3,
                    order = order,
                    seasonal_order = seasonal_order)
```

In [50]:
```
mape(cv2)
```

Out[50]: 0.06835834478161248

In [51]:
```
cv2.plot(title = 'forecast every three months using one year prior data')
```

Out[51]: <matplotlib.axes._subplots.AxesSubplot at 0x7fc8152a1910>



In [52]:
```
mape(cv2)
```

Out[52]: 0.06835834478161248

In [53]:
```
len(monthly_temp.temp)
```

Out[53]: 792

In [54]:
```
def grid_search_ARIMA(series,horizon,start,step_size,orders = [(1,0,0)],seasonal_o
    best_mape = np.inf
    best_order = None
    best_seasonal_order = None
    best_trend = None
    for order_ in orders:
        for seasonal_order_ in seasonal_orders:
            for trend_ in trends:

                cv = cross_validate(series,
                                    horizon,
                                    start,
                                    step_size,
                                    order = order_,
                                    seasonal_order = seasonal_order_,
                                    trend=trend_)
                if mape(cv)<best_mape:
                    best_mape = mape(cv)
                    best_order = order_
                    best_seasonal_order = seasonal_order_
                    best_trend = trend_
    return (best_order,best_seasonal_order, best_trend, best_mape)
```

In [55]:
```
series = monthly_temp.temp
horizon = 12
start = 760
step_size = 3
orders = [(1,1,0),(1,0,0)]
seasonal_orders = [(0,1,1,12)]
trends = [None,'c']

grid_search_ARIMA(series = series,
                  horizon = horizon,
                  start = start,
                  step_size = step_size,
                  orders = orders,
                  seasonal_orders = seasonal_orders,
                  trends=trends)
```

Out[55]: ((1, 0, 0), (0, 1, 1, 12), None, 0.03513163490408245)

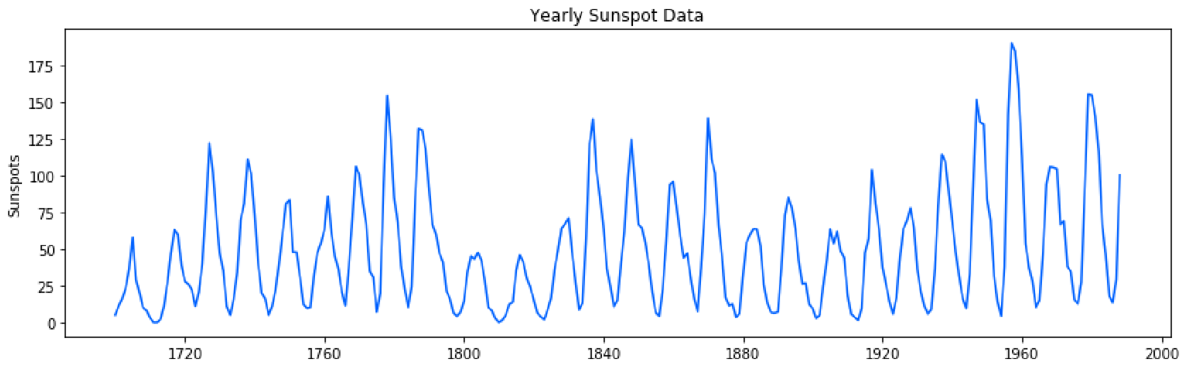
Forecasting Exercise:

1. Find appropriate model to forecast sunspots data
2. Run diagnostics to check if fit makes sense
3. Run cross-validation on a few different choices to see which one has best out of sample error

Note: This data is annual, but we still need to find the seasonality The variance does not seem to stay constant throughout, think of transformations we discussed earlier in course to take care of this

In [56]:
```
# read and plot data
data_path = 'https://vincentarelbundock.github.io/Rdatasets/csv/datasets/sunspot.ye
data = pd.read_csv(data_path,usecols = ['time','value'],index_col = 'time',parse_da
plt.figure()
```

```
plt.plot(data.index,data['value'])
plt.ylabel('Sunspots')
plt.title('Yearly Sunspot Data');
```

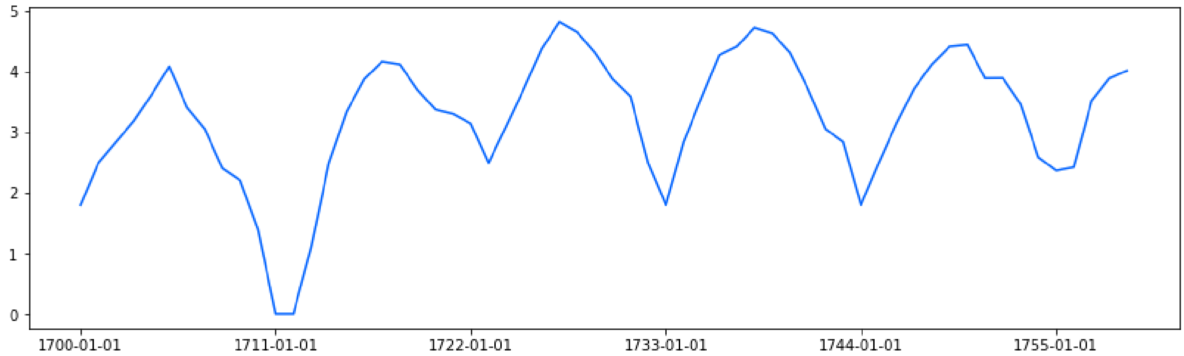


In [57]:

```
#given the difference in variance we should probably take log of data
data['log_ss'] = np.log1p(data['value'])
# going to zoom in on last 60 values to get a better idea of frequency of seasonal
plt.plot(data['log_ss'][:60])
plt.xticks(ticks = data.iloc[0:60:11].index)
```

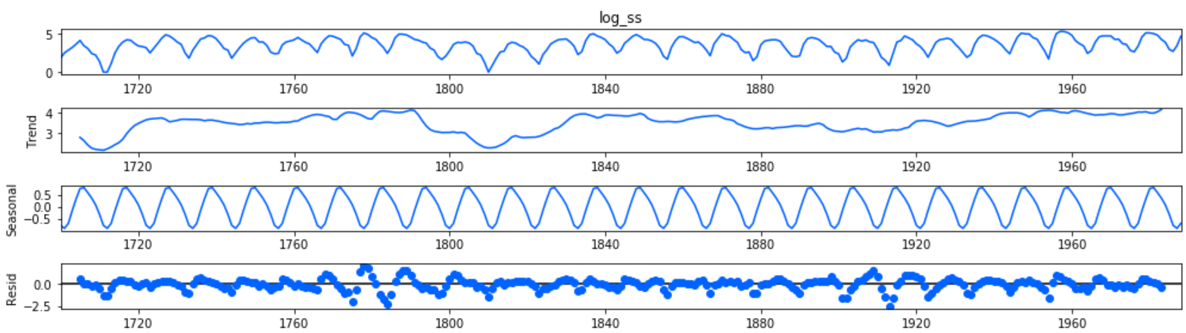
Out[57]:

```
([<matplotlib.axis.XTick at 0x7fc7c4996b10>,
  <matplotlib.axis.XTick at 0x7fc7c4996490>,
  <matplotlib.axis.XTick at 0x7fc7c49b6b50>,
  <matplotlib.axis.XTick at 0x7fc7c49c5650>,
  <matplotlib.axis.XTick at 0x7fc7c49c5b10>,
  <matplotlib.axis.XTick at 0x7fc7c49c5d50>],
 <a list of 6 Text xticklabel objects>)
```



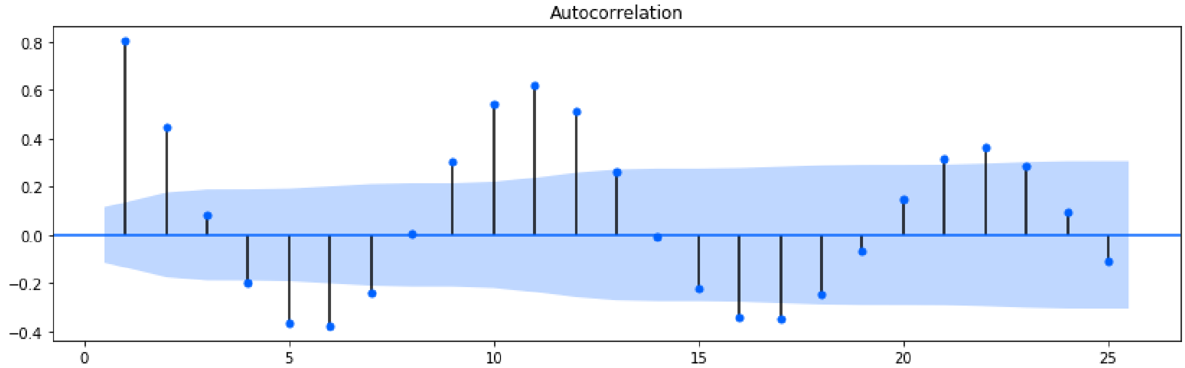
In [58]:

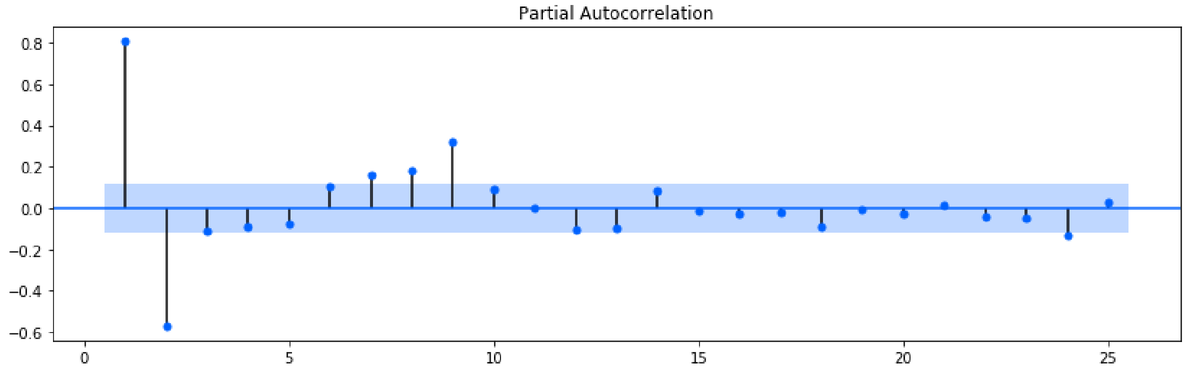
```
# plot decomposition with frequency 11
# Seems to do decent job of capturing seasonality
from statsmodels.tsa.seasonal import seasonal_decompose
seasonal_decompose(data.log_ss,freq=11).plot();
```



In [59]:

```
sm.tsa.graphics.plot_acf(data.log_ss,zero=False)
sm.tsa.graphics.plot_pacf(data.log_ss,zero = False);
```

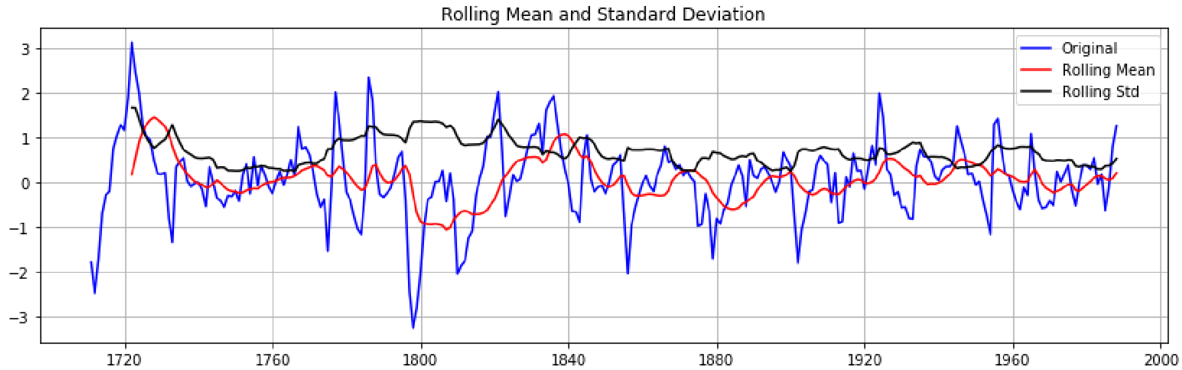




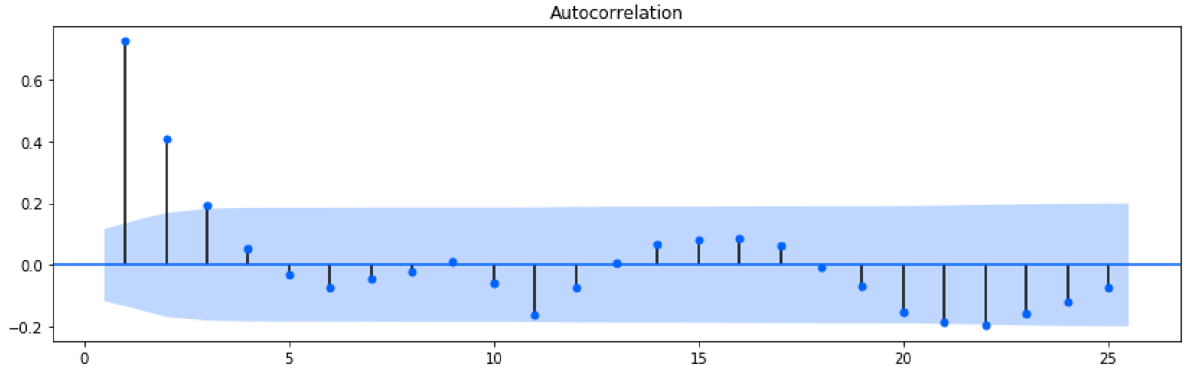
In [60]:
```
data['lag_11'] = data.log_ss.shift(11)
data['seasonal_diff'] = data.log_ss - data['lag_11']
```

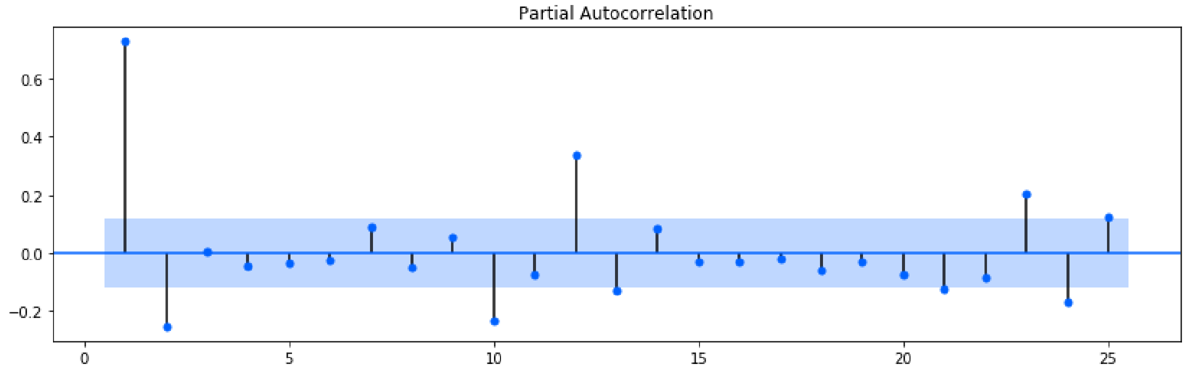
In [61]:
```
# mean moves, not perfect. p-value shows we can reject the null of non stationarity
dftest(data['seasonal_diff'].dropna())
```

```
Test Statistic            -4.127070
p-value                    0.000873
Lags Used                 12.000000
Observations Used        265.000000
Critical Value (1%)       -3.455270
Critical Value (5%)       -2.872509
Critical Value (10%)      -2.572615
dtype: float64
```



In [62]:
```
sm.tsa.graphics.plot_acf(data['seasonal_diff'].dropna(),zero=False)
sm.tsa.graphics.plot_pacf(data['seasonal_diff'].dropna(),zero=False);
```

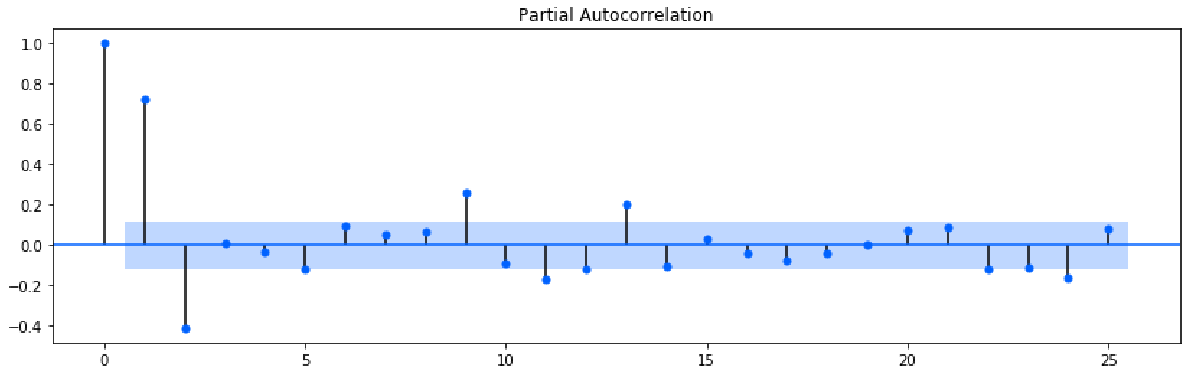




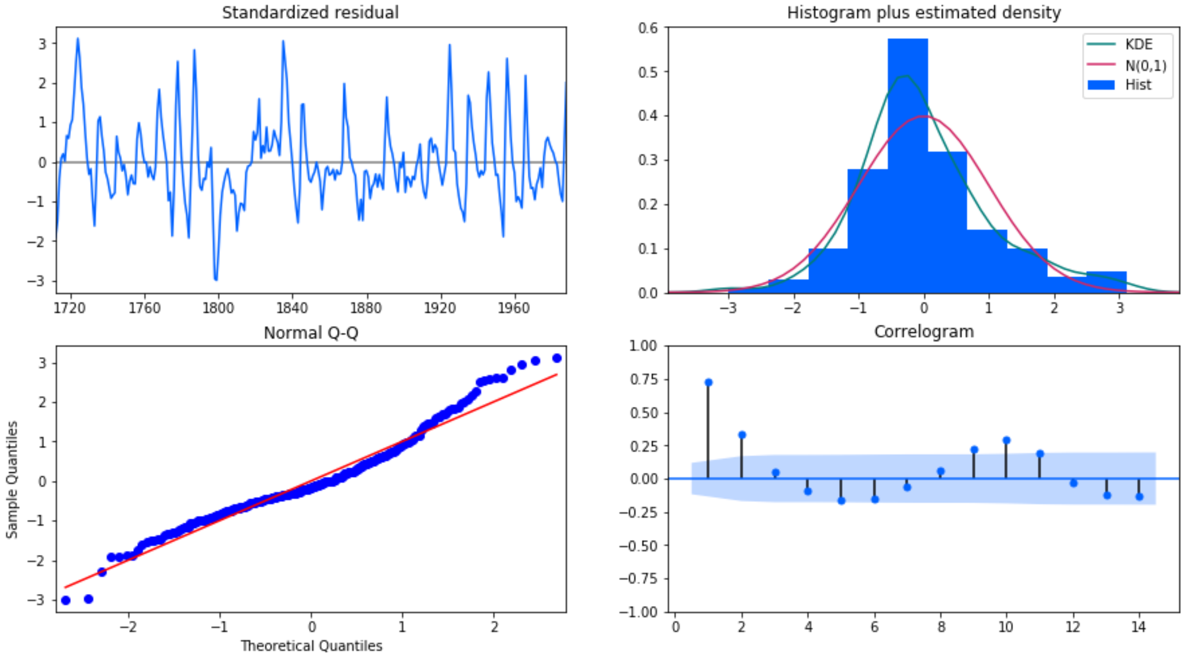
looks like an ar2 model with seasonal differencing

In [63]:
```
sar3 = sm.tsa.statespace.SARIMAX(data.log_ss,
                                order=(0,0,0),
                                seasonal_order=(0,1,0,12),
                                trend='c').fit()
```

In [64]:
```
sm.tsa.graphics.plot_pacf(sar3.resid[sar3.loglikelihood_burn:]);
```



In [65]:
```
sar3.plot_diagnostics(figsize = (15,8),lags = range(1,15));
```



In [66]:
```
auto_model = pm.auto_arima(data.log_ss, start_p=0, start_q=0,
                           max_p=3, max_q=3, m=11,
                           start_P=0, seasonal=True,
                           d=0, D=1, trace=True,
                           error_action='ignore',
                           suppress_warnings=True,
                           stepwise=True)
```

```
Fit ARIMA: order=(0, 0, 0) seasonal_order=(0, 1, 1, 11); AIC=694.412, BIC=705.295,
Fit time=0.229 seconds
Fit ARIMA: order=(0, 0, 0) seasonal_order=(0, 1, 0, 11); AIC=708.047, BIC=715.302,
Fit time=0.120 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(1, 1, 0, 11); AIC=464.732, BIC=479.243,
Fit time=0.289 seconds
Fit ARIMA: order=(0, 0, 1) seasonal_order=(0, 1, 1, 11); AIC=501.605, BIC=516.115,
Fit time=0.425 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(0, 1, 0, 11); AIC=495.304, BIC=506.187,
Fit time=0.110 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(2, 1, 0, 11); AIC=450.988, BIC=469.126,
Fit time=1.037 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(2, 1, 1, 11); AIC=424.445, BIC=446.210,
Fit time=4.993 seconds
Fit ARIMA: order=(0, 0, 0) seasonal_order=(2, 1, 1, 11); AIC=657.075, BIC=675.213,
Fit time=3.799 seconds
Fit ARIMA: order=(2, 0, 0) seasonal_order=(2, 1, 1, 11); AIC=391.800, BIC=417.194,
Fit time=5.188 seconds
Fit ARIMA: order=(2, 0, 1) seasonal_order=(2, 1, 1, 11); AIC=393.588, BIC=422.609,
Fit time=5.591 seconds
Fit ARIMA: order=(3, 0, 1) seasonal_order=(2, 1, 1, 11); AIC=395.801, BIC=428.450,
Fit time=6.498 seconds
Fit ARIMA: order=(2, 0, 0) seasonal_order=(1, 1, 1, 11); AIC=390.035, BIC=411.800,
Fit time=1.738 seconds
Fit ARIMA: order=(2, 0, 0) seasonal_order=(1, 1, 0, 11); AIC=443.928, BIC=462.066,
Fit time=0.369 seconds
Fit ARIMA: order=(2, 0, 0) seasonal_order=(1, 1, 2, 11); AIC=391.735, BIC=417.129,
Fit time=5.501 seconds
Fit ARIMA: order=(2, 0, 0) seasonal_order=(0, 1, 0, 11); AIC=480.371, BIC=494.882,
Fit time=0.153 seconds
Fit ARIMA: order=(2, 0, 0) seasonal_order=(2, 1, 2, 11); AIC=392.902, BIC=421.923,
Fit time=4.964 seconds
Fit ARIMA: order=(1, 0, 0) seasonal_order=(1, 1, 1, 11); AIC=423.456, BIC=441.594,
Fit time=1.071 seconds
Fit ARIMA: order=(3, 0, 0) seasonal_order=(1, 1, 1, 11); AIC=391.862, BIC=417.256,
Fit time=1.822 seconds
Fit ARIMA: order=(2, 0, 1) seasonal_order=(1, 1, 1, 11); AIC=391.797, BIC=417.191,
Fit time=1.711 seconds
Fit ARIMA: order=(3, 0, 1) seasonal_order=(1, 1, 1, 11); AIC=393.800, BIC=422.821,
Fit time=2.616 seconds
Fit ARIMA: order=(2, 0, 0) seasonal_order=(0, 1, 1, 11); AIC=402.386, BIC=420.524,
Fit time=0.629 seconds
Total fit time: 48.858 seconds
```

In [67]:

```
print('order: ',auto_model.order)
print('seasonal order: ',auto_model.seasonal_order)
```

```
order:  (2, 0, 0)
seasonal order:  (1, 1, 1, 11)
```

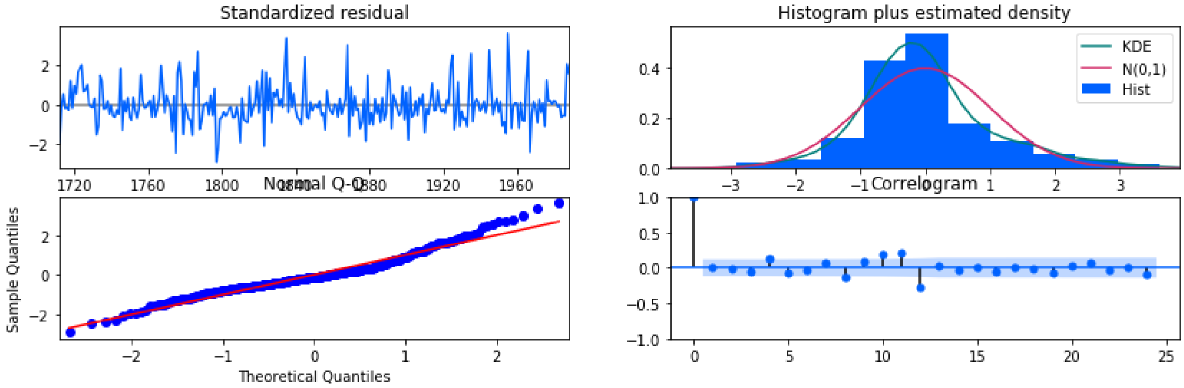
In [68]:

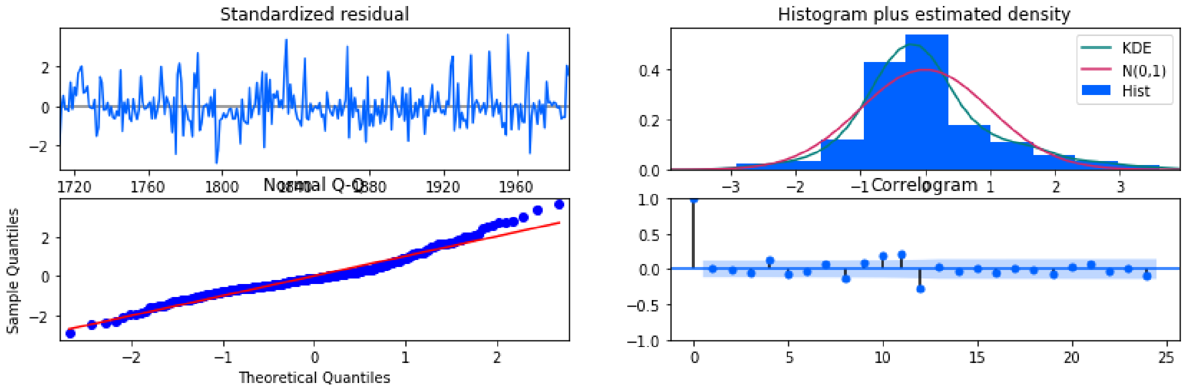
```
sar4 = sm.tsa.statespace.SARIMAX(data.log_ss,
                                order=(2,0,0),
                                seasonal_order=(0,1,0,12),
                                trend='c').fit()
```

In [69]:

```
sar4.plot_diagnostics(lags=24)
```

Out[69]:



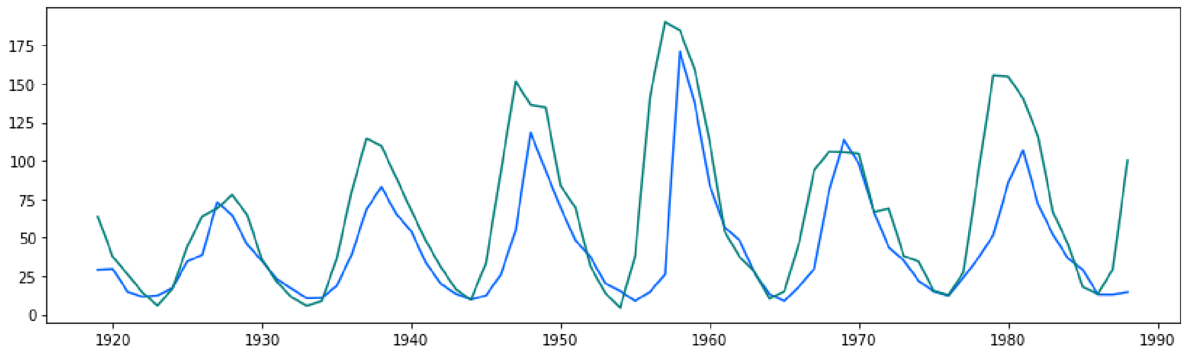


In [70]:
```
warnings.filterwarnings("ignore")
series = data['log_ss']
horizon = 3
start = int(len(data.value)*.75)
step_size = 1
order = auto_model.order
seasonal_order = auto_model.seasonal_order

log_cv1 = cross_validate(series,horizon,start,step_size,
                    order = order,
                    seasonal_order = seasonal_order)
```

In [71]:
```
log_cv1 = np.expm1(log_cv1)
```

In [72]:
```
plt.plot(log_cv1)
```

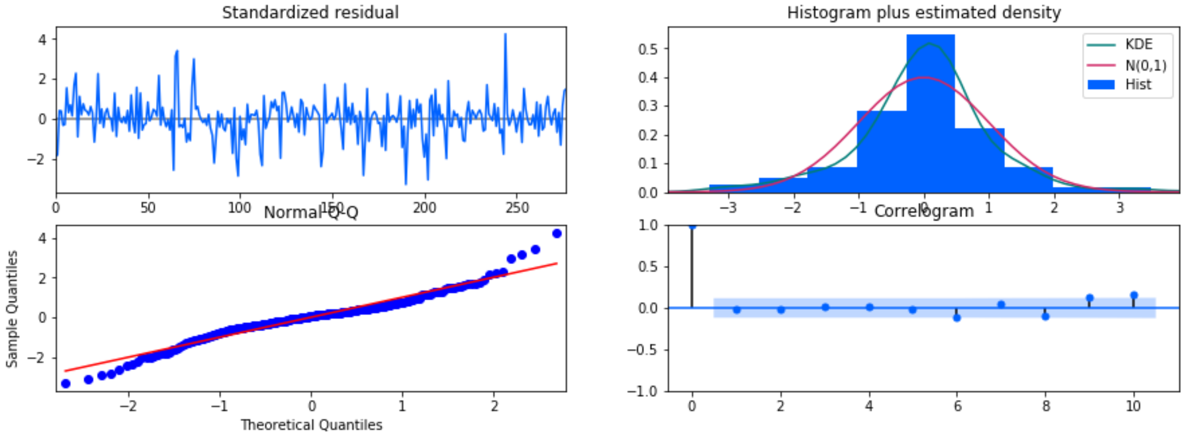
Out[72]:
```
[<matplotlib.lines.Line2D at 0x7fc7e9972e10>,
 <matplotlib.lines.Line2D at 0x7fc7e9aee990>]
```



In [73]:
```
mape(log_cv1)
```

Out[73]:
```
0.36972802846359965
```

In [74]:
```
auto_model.plot_diagnostics(figsize = (15,5));
```



In [75]:

```
warnings.filterwarnings("ignore")
series = data['log_ss']
horizon = 3
start = int(len(data.value)*.75)
step_size = 1
order = (2,0,0)
seasonal_order = (1,1,0,11)

log_cv2 = cross_validate(series,horizon,start,step_size,
                    order = order,
                    seasonal_order = seasonal_order)
```

In [76]:

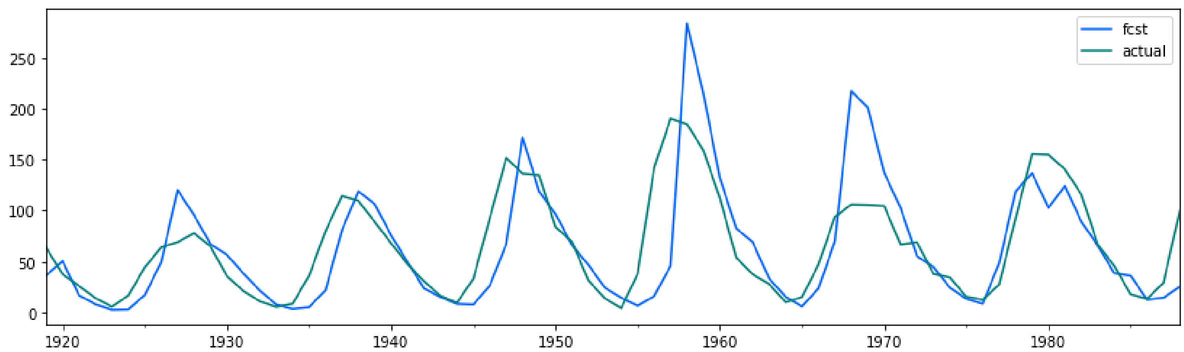
```
log_cv2 = np.expm1(log_cv2)
```

In [77]:

```
log_cv2.plot()
```

Out[77]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc81ad3d2d0>
```



In [78]:

```
mape(log_cv2)
```

Out[78]:

```
0.4137612967220888
```

Section 6: Predicting with Facebook Prophet

From site:

> Today Facebook is open sourcing Prophet, a forecasting tool available in Python and R. The idea is that producing high quality forecasts is not an easy problem for either machines or for most analysts. The models revolves

> around two main observations in the practice of creating a variety of business forecasts:
>
> - Completely automatic forecasting techniques can be brittle and they are often too inflexible to incorporate useful assumptions or heuristics.
> - Analysts who can produce high quality forecasts are quite rare because forecasting is a specialized data science skill requiring substantial experience.

Prophet is an general additive model that includes a number of highly advanced, intelligent forecasting methods, including changepoint analysis:

y = g(t) + s(t) + h(t) + ϵ_t

Here g(t) is the trend function which models non-periodic changes in the value of the time series, s(t) represents periodic changes (e.g., weekly and yearly seasonality), and h(t) represents the effects of holidays which occur on potentially irregular schedules over one or more days

- For trend, a piecewise linear or logistic growth curve trend is used.
    - Prophet automatically detects changes in trends by selecting changepoints from the data.
- For seasonalities, different seasonality components are modeled using Fourier series.
- One can either use fb provided list or incorporate their own holidays into model.

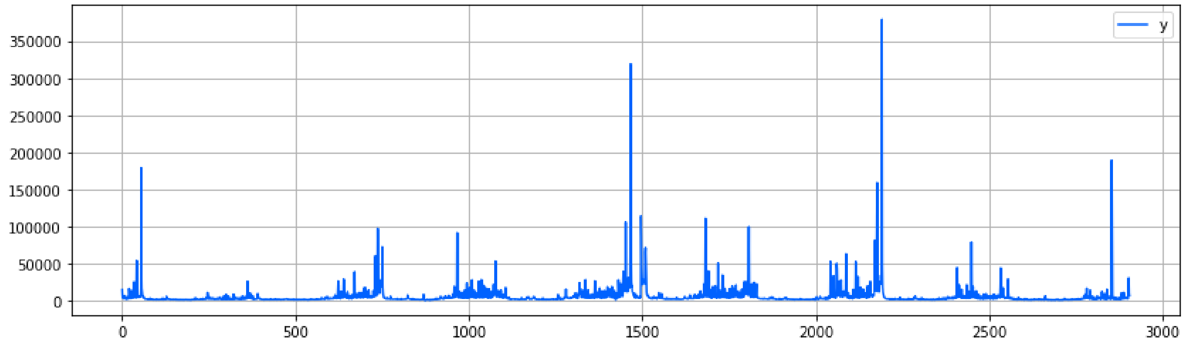
Prophet was originally optimized with the business forecast tasks encountered at Facebook in mind, which typically have any of the following characteristics:

- Hourly, daily, or weekly observations with at least a few months (preferably a year) of history
- Strong multiple “human-scale” seasonalities: day of week and time of year
- Important holidays that occur at irregular intervals that are known in advance
- A reasonable number of missing observations or large outliers
- Historical trend changes, for instance due to product launches or logging changes
- Trends that are non-linear growth curves, where a trend hits a natural limit or saturates

Technical details behind prophet: built around a generalized additive model (GAM)

In [79]:
```
# read daily page views for the Wikipedia page for Peyton Manning; scraped into hos
# conda install -c conda-forge fbprophet (to install)
from fbprophet import Prophet
plt.rcParams['figure.figsize'] = [14, 4]

data_path = 'https://raw.githubusercontent.com/PinkWink/DataScience/master/data/07
peyton = pd.read_csv(data_path)
```

In [80]:
```
# plot data
peyton.plot()
plt.grid();
```

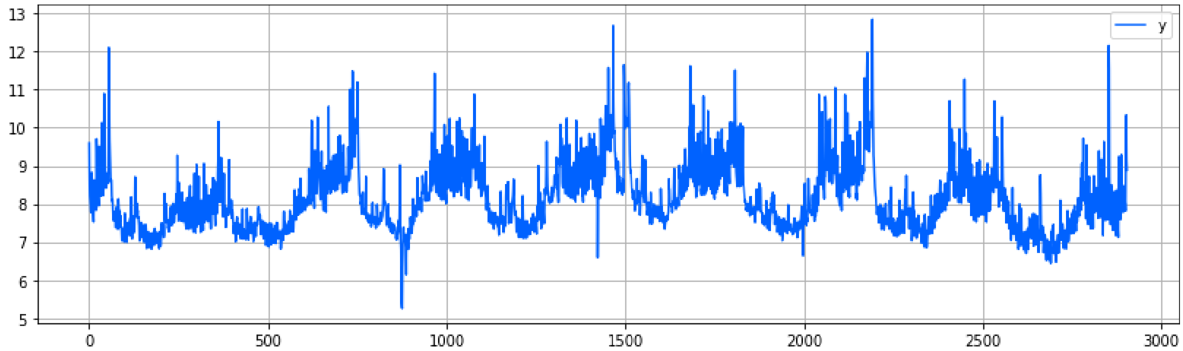


In [81]:
```
# log data due to spikes
# dataframe must have ds column with type datetime and y column which is time serie
peyton['y'] = np.log(peyton['y'])
peyton.head()
```

Out[81]:

	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

In [82]:
```
# plot log
peyton.plot()
plt.grid();
```



In [83]:
```
# fit model
m = Prophet()
m.fit(peyton)
```

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=Tru
e to override this.
```

Out[83]:
```
<fbprophet.forecaster.Prophet at 0x7fc82acfa490>
```

In [84]:
```
peyton.head()
```

Out[84]:

	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

In [85]:

```
# forecast 365 days into future
# prophet requires a blank dataframe to input predictions
# will also provide blank set for dates within dataset to allow for fit
future = m.make_future_dataframe(periods=365)
print(future.head())
print(future.tail())
```

```
          ds
0 2007-12-10
1 2007-12-11
2 2007-12-12
3 2007-12-13
4 2007-12-14
             ds
3265 2017-01-15
3266 2017-01-16
3267 2017-01-17
3268 2017-01-18
3269 2017-01-19
```

In [86]:

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

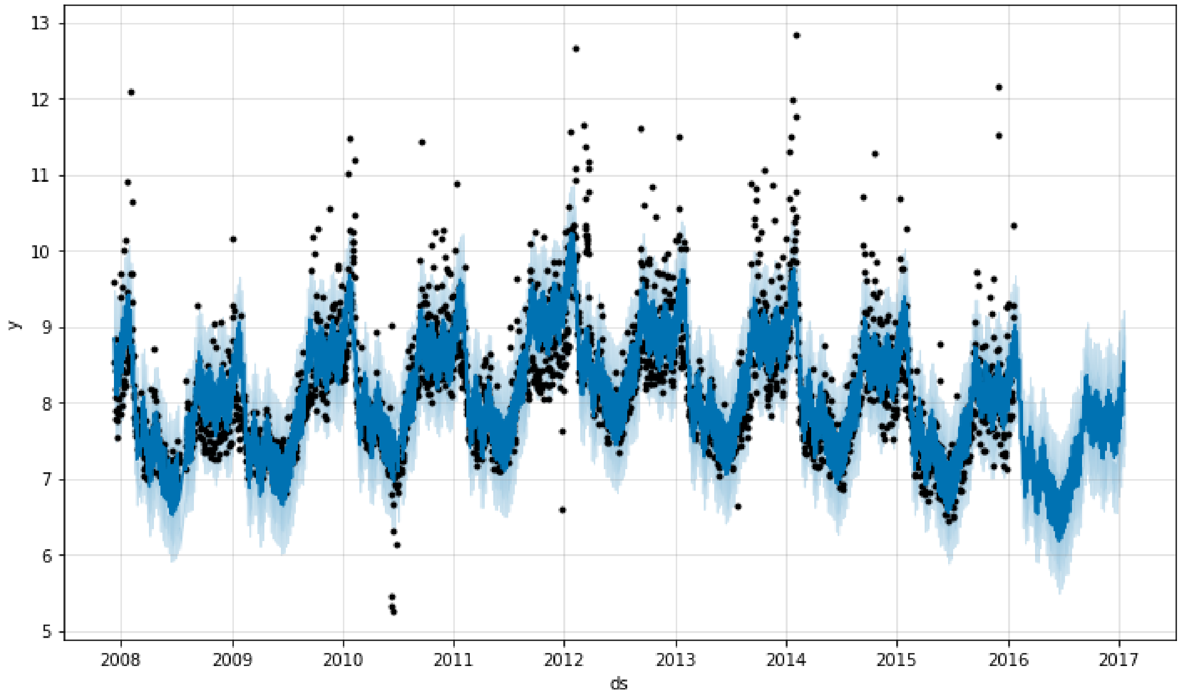
```
Index(['ds', 'trend', 'yhat_lower', 'yhat_upper', 'trend_lower', 'trend_upper',
       'additive_terms', 'additive_terms_lower', 'additive_terms_upper',
       'weekly', 'weekly_lower', 'weekly_upper', 'yearly', 'yearly_lower',
       'yearly_upper', 'multiplicative_terms', 'multiplicative_terms_lower',
       'multiplicative_terms_upper', 'yhat'],
      dtype='object')
```

Out[86]:

	ds	yhat	yhat_lower	yhat_upper
3265	2017-01-15	8.208171	7.503198	8.922039
3266	2017-01-16	8.533159	7.753492	9.214067
3267	2017-01-17	8.320549	7.631910	9.061524
3268	2017-01-18	8.153184	7.425827	8.886352
3269	2017-01-19	8.165106	7.441329	8.879306

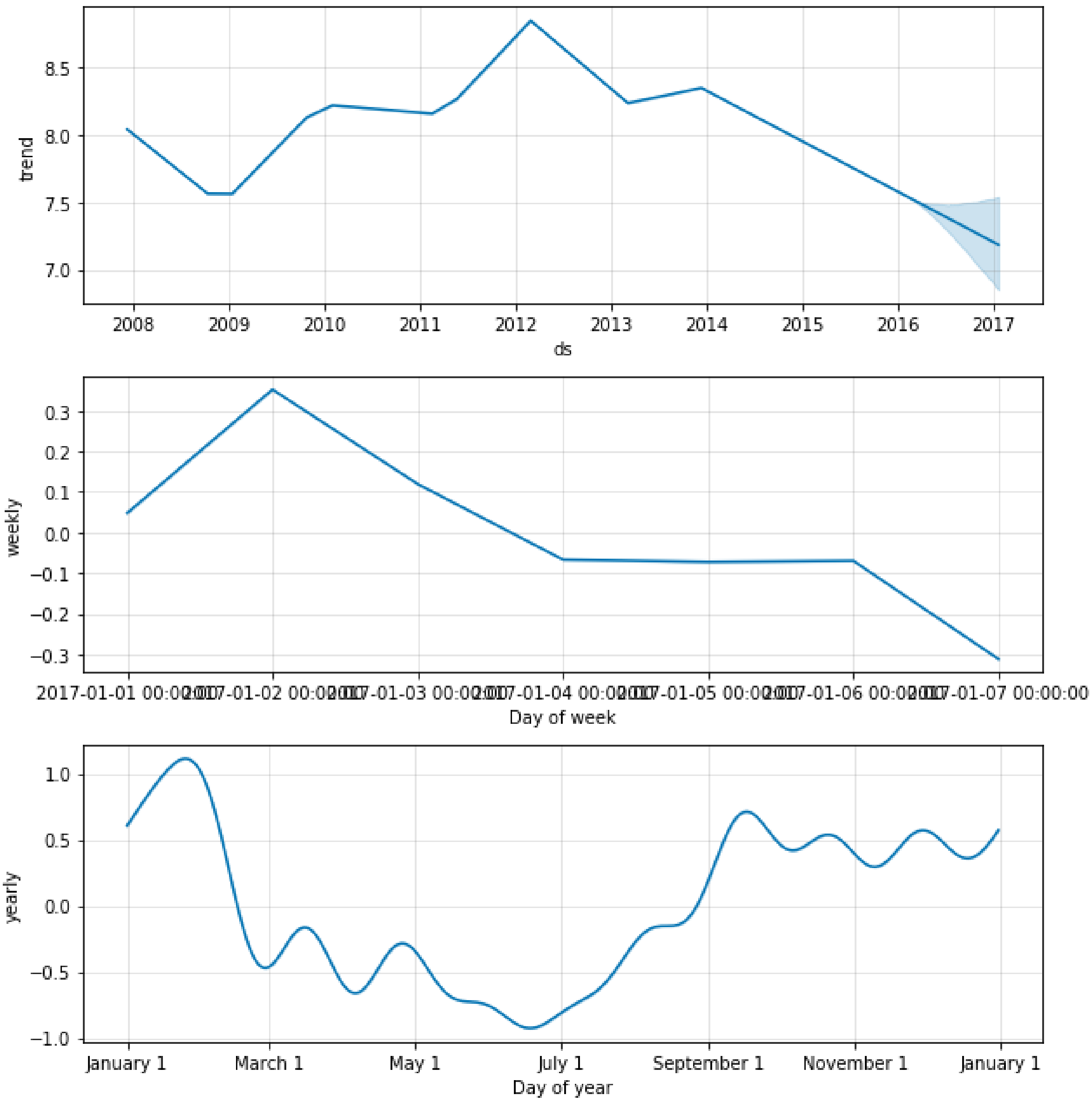
In [87]:

```
# plot forecast
m.plot(forecast);
```



In [88]:

```
# plot individual components of forecast: trend, weekly/yearly seasonality,
m.plot_components(forecast);
```



> We can also add holiday and Superbowl date information to Peyton's forecast, since we hypothesize people will visit his site more often on those dates.

In [89]:
```
# add holidays
playoffs = pd.DataFrame({
  'holiday': 'playoff',
  'ds': pd.to_datetime(['2008-01-13', '2009-01-03', '2010-01-16',
                        '2010-01-24', '2010-02-07', '2011-01-08',
                        '2013-01-12', '2014-01-12', '2014-01-19',
                        '2014-02-02', '2015-01-11', '2016-01-17',
                        '2016-01-24', '2016-02-07']),
  'lower_window': 0, # these help us specify spillover into previous and future day
  'upper_window': 1,
})

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

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

In [90]:
```
holidays.head()
```

Out[90]:

	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

In [91]:
```
# fit and predict
m = Prophet(holidays=holidays)
forecast = m.fit(peyton).predict(future)
```

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=Tru
e to override this.
```

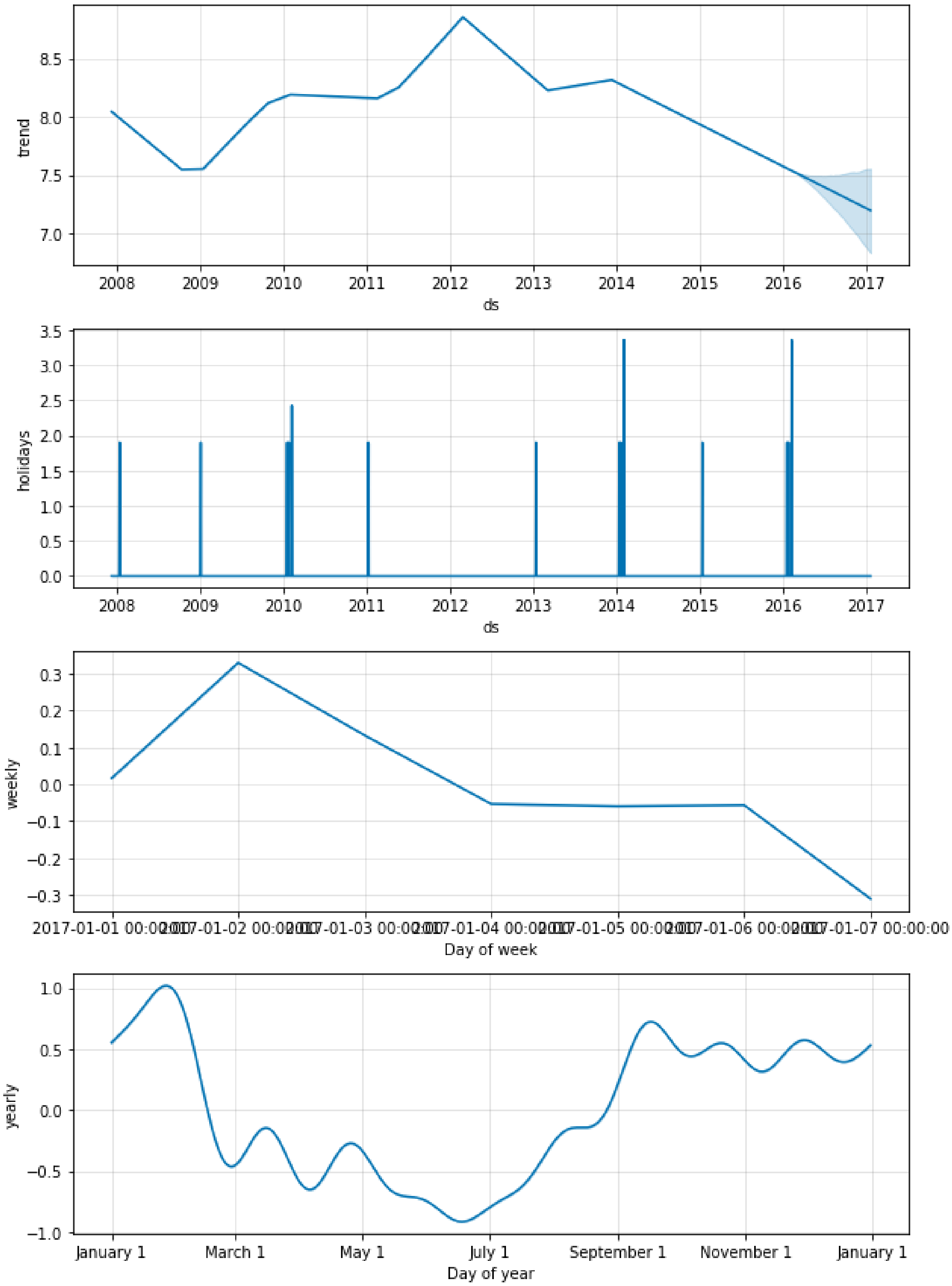
In [92]:
```
# we can see the effects of various 'holidays' on site visits
forecast[(forecast['playoff'] + forecast['superbowl']).abs() > 0][
        ['ds', 'playoff', 'superbowl']][-10:]
```

Out[92]:

	ds	playoff	superbowl
2190	2014-02-02	1.224221	1.203104
2191	2014-02-03	1.898940	1.459895
2532	2015-01-11	1.224221	0.000000
2533	2015-01-12	1.898940	0.000000
2901	2016-01-17	1.224221	0.000000
2902	2016-01-18	1.898940	0.000000
2908	2016-01-24	1.224221	0.000000
2909	2016-01-25	1.898940	0.000000
2922	2016-02-07	1.224221	1.203104
2923	2016-02-08	1.898940	1.459895

In [93]:

```
# check the impacts visually
m.plot_components(forecast);
```



> Peyton won Superbowls XLI (41, 2007) and 50 (2016), while losing XLIV (44, 2010) and XLVIII(48, 2014). We can see these spikes in the holidays chart.

In [94]:

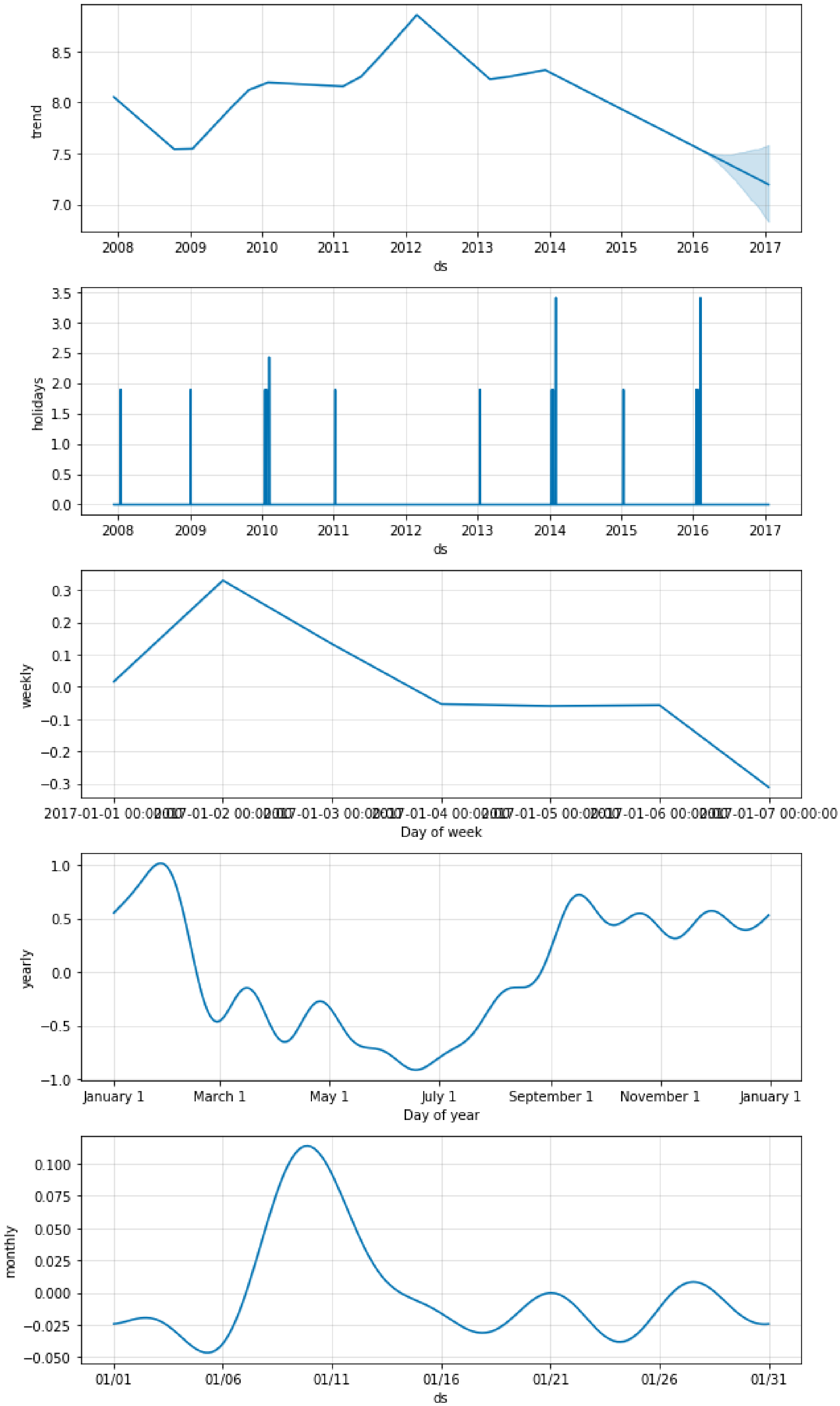
```
# Add in another seasonality besides yearly, weekly, daily
# fit model
m = Prophet(holidays=holidays,)
m.add_seasonality(name='monthly', period=30.5, fourier_order=5)
m.fit(peyton)
```

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
```

Out[94]: <fbprophet.forecaster.Prophet at 0x7fc7c620f390>

In [95]:
```
fcst_month = m.predict(future)
```

In [96]:
```
m.plot_components(fcst_month)
plt.show()
```



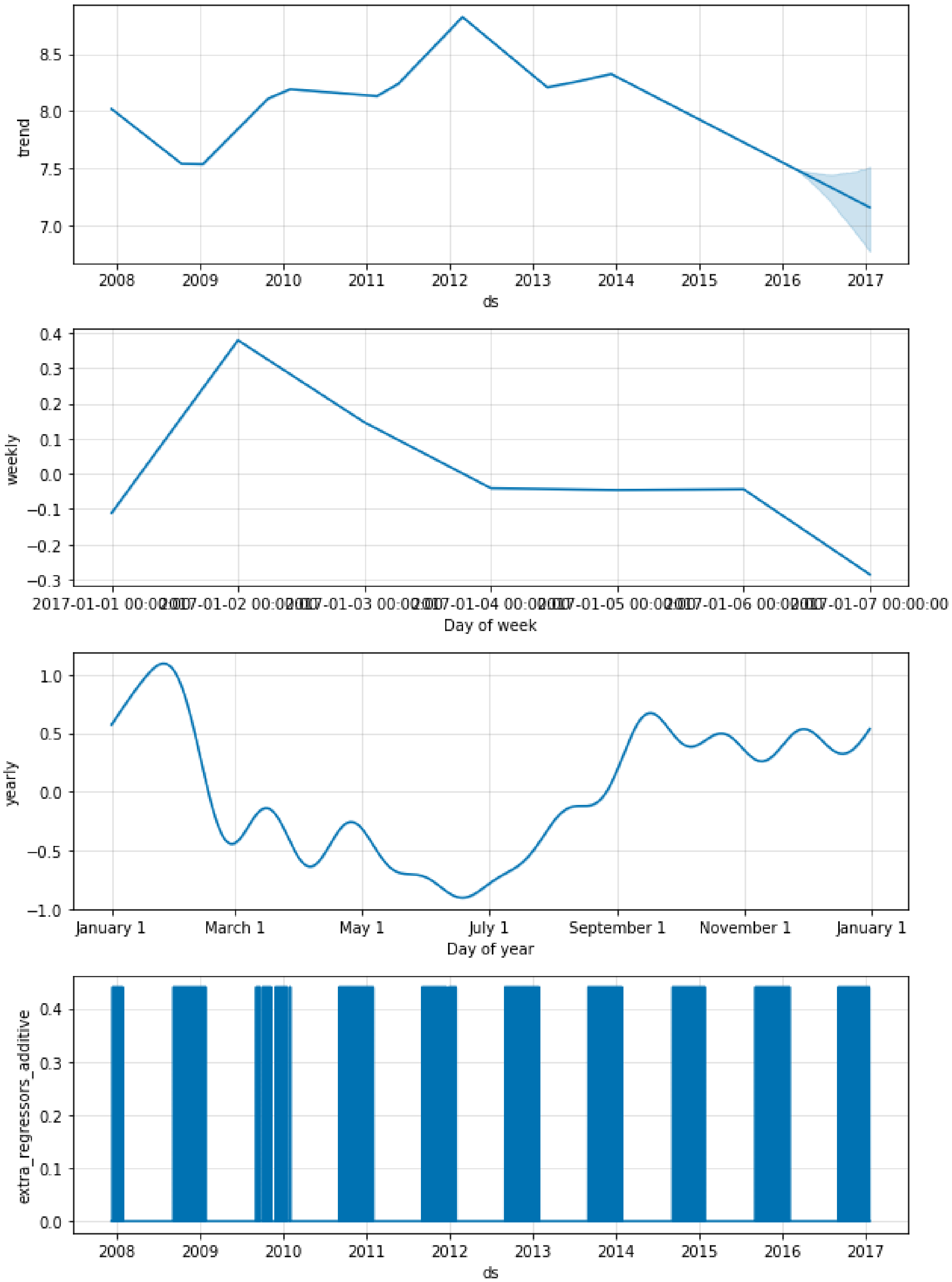
trend
8.5
8.0
7.5
7.0
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
ds
holidays
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
ds
weekly
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
2017-01-01 00:00:00
2017-01-02 00:00:00
2017-01-03 00:00:00
2017-01-04 00:00:00
2017-01-05 00:00:00
2017-01-06 00:00:00
2017-01-07 00:00:00
Day of week
yearly
1.0
0.5
0.0
−0.5
−1.0
January 1
March 1
May 1
July 1
September 1
November 1
January 1
Day of year
monthly
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050
01/01
01/06
01/11
01/16
01/21
01/26
01/31
ds

Adding a regressor

In [97]:

```
# creating indicator variable for nfl sundays
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

#adding that to our df
peyton['nfl_sunday'] = peyton['ds'].apply(nfl_sunday)
print(peyton)

m = Prophet()

# must be in the fit df
m.add_regressor('nfl_sunday')
m.fit(peyton)

# regressor must also be available in future df
future['nfl_sunday'] = future['ds'].apply(nfl_sunday)

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

```
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=Tru
e to override this.
              ds          y  nfl_sunday
0     2007-12-10   9.590761           0
1     2007-12-11   8.519590           0
2     2007-12-12   8.183677           0
3     2007-12-13   8.072467           0
4     2007-12-14   7.893572           0
...          ...        ...         ...
2900  2016-01-16   7.817223           0
2901  2016-01-17   9.273878           1
2902  2016-01-18  10.333775           0
2903  2016-01-19   9.125871           0
2904  2016-01-20   8.891374           0

[2905 rows x 3 columns]
```



In [98]:

```
# These are points where trend has changed
print('originally: ',m.changepoints[:5])
# you can specify changepoints if you want trend to only be allowed at certain poi
m_c = Prophet(changepoints=['2014-01-01'])

print('\nnow: ',m_c.changepoints[:5])
```

```
originally:  93    2008-03-17
186   2008-06-20
279   2008-10-11
372   2009-01-14
465   2009-04-17
Name: ds, dtype: datetime64[ns]

now:  DatetimeIndex(['2014-01-01'], dtype='datetime64[ns]', freq=None)
```

Cross validation with fbprophet

In [99]:

```
from fbprophet.diagnostics import cross_validation
#Starting from 730 days in, making a prediction every 180 days, 365 days into the j
df_cv = cross_validation(m, initial='730 days', period='180 days', horizon = '365 d
df_cv.head()
```

```
INFO:fbprophet:Making 11 forecasts with cutoffs between 2010-02-15 00:00:00 and 20
15-01-20 00:00:00
```

Out[99]:

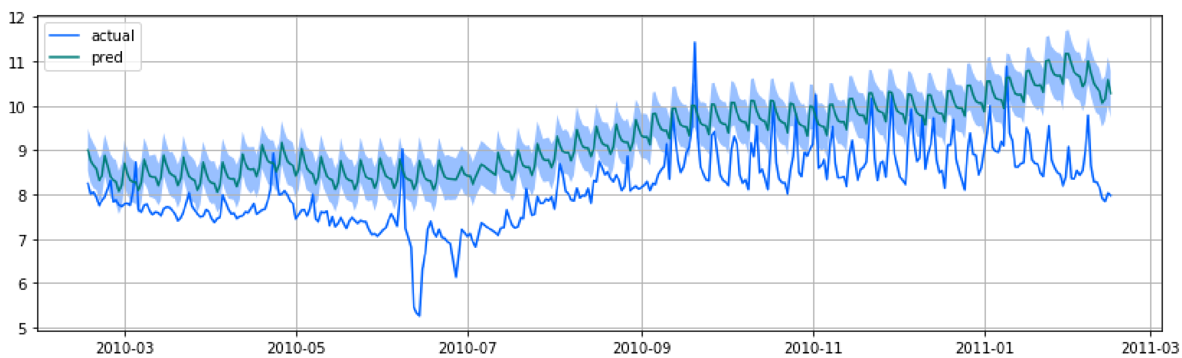
	ds	yhat	yhat_lower	yhat_upper	y	cutoff
0	2010-02-16	8.998933	8.532879	9.482757	8.242493	2010-02-15
1	2010-02-17	8.764575	8.258785	9.284304	8.008033	2010-02-15
2	2010-02-18	8.649843	8.143984	9.148289	8.045268	2010-02-15
3	2010-02-19	8.570880	8.091072	9.030213	7.928766	2010-02-15
4	2010-02-20	8.313209	7.839553	8.817683	7.745003	2010-02-15

In [100…

```
# Just looking at data from first cutoff
first_cut = df_cv[df_cv.cutoff == datetime(2010,2,15)]
plt.plot(first_cut.ds,first_cut.y,label='actual')
plt.plot(first_cut.ds,first_cut.yhat,label = 'pred')
plt.fill_between(first_cut.ds,first_cut.yhat_lower,first_cut.yhat_upper,alpha=0.4)
plt.grid()
plt.legend()
```

Out[100]:

```
<matplotlib.legend.Legend at 0x7fc7e9b98550>
```



In [101…

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

Out[101]:

	horizon	mse	rmse	mae	mape	coverage
0	37 days	0.494728	0.703369	0.506860	0.058717	0.666514
1	38 days	0.499801	0.706966	0.510718	0.059151	0.665372
2	39 days	0.522379	0.722758	0.517344	0.059815	0.662631
3	40 days	0.529516	0.727679	0.519364	0.060029	0.661261
4	41 days	0.534146	0.730853	0.518587	0.059902	0.668799

Predicting CO2

In [102…

```
# load data
co2 = pd.read_csv('./co2-ppm-mauna-loa-19651980.csv',
                  header = 0,
                  names = ['idx', 'co2'],
                  skipfooter = 2)
co2 = co2.drop('idx', 1)

# recast co2 col to float
co2['co2'] = pd.to_numeric(co2['co2'])
co2.drop(labels=0, inplace=True)

# set index
index = pd.date_range('1/1/1965', periods=191, freq='M')
co2.index = index
```

In [103…

```
# load co2 data, rename headers, and check
# data = sm.datasets.co2.load_pandas()
# co2 = data.data

co2['ds'] = co2.index
co2.rename(columns={'co2': 'y'}, inplace=True)

co2.tail()
```

Out[103]:

	y	ds
1980-07-31	337.19	1980-07-31
1980-08-31	335.49	1980-08-31
1980-09-30	336.63	1980-09-30
1980-10-31	337.74	1980-10-31
1980-11-30	338.36	1980-11-30

In [104…

```
# fit model
model = Prophet()
model.fit(co2);
```

```
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonality=T
rue to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=Tru
e to override this.
```

In [105…

```
# forecast 15 years into future
future = model.make_future_dataframe(periods=120, freq='M', include_history=True)
future.tail()
```

```
#future = model.make_future_dataframe(periods=365*15)
#future.tail()
```

Out[105]:

	ds
306	1990-07-31
307	1990-08-31
308	1990-09-30
309	1990-10-31
310	1990-11-30

In [106…

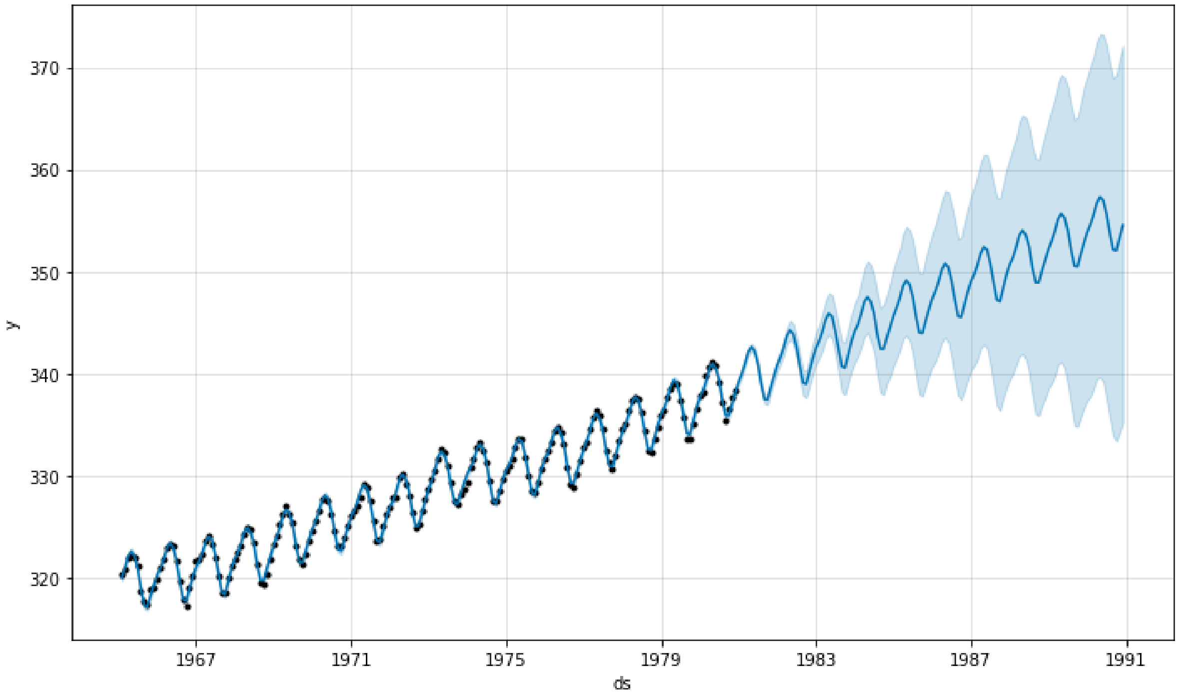
```
# populate forecast
forecast = model.predict(future)
forecast.tail()
```

Out[106]:

	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	additive_terms	additiv
306	1990-07-31	354.924971	335.607648	370.380483	336.769191	371.471783	-1.117225	
307	1990-08-31	355.063263	333.919260	368.933589	336.610682	371.921462	-2.908550	
308	1990-09-30	355.197093	333.431875	369.218141	336.460935	372.322623	-3.146657	
309	1990-10-31	355.335385	334.254718	370.809356	336.313187	372.623899	-2.000587	
310	1990-11-30	355.469215	335.230407	372.129528	336.179776	372.967186	-0.940856	

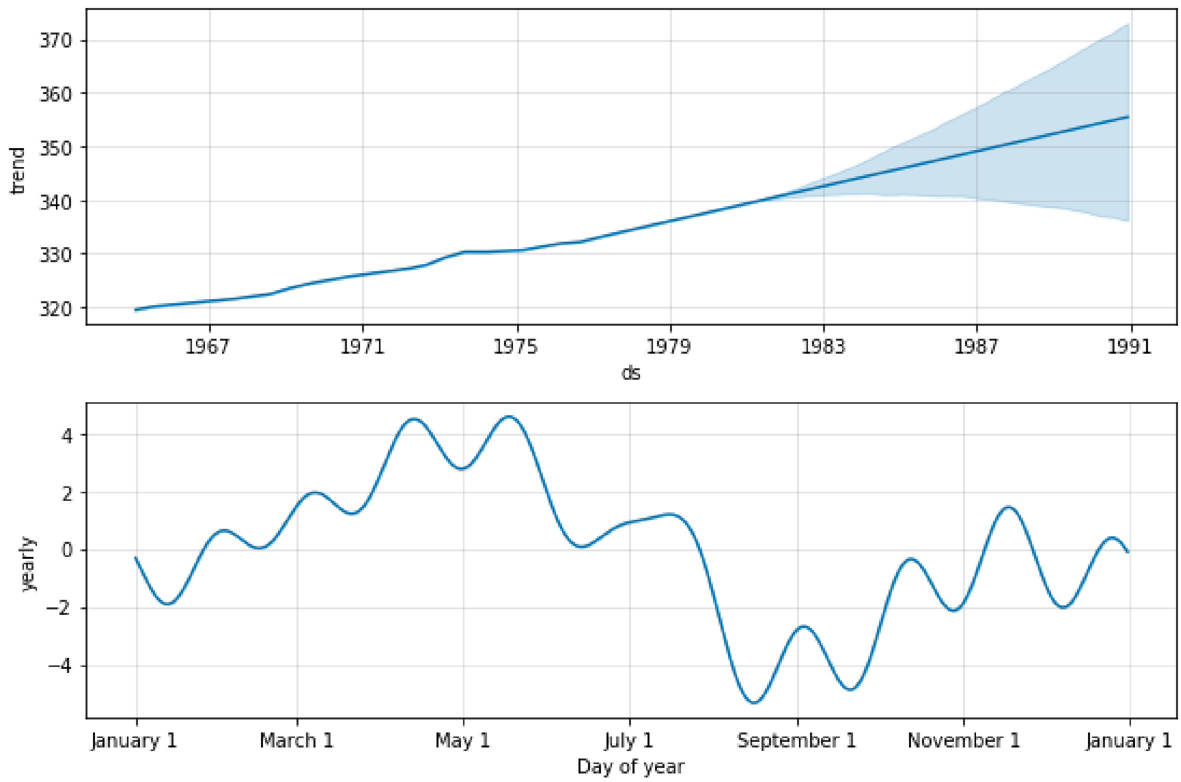
In [107…

```
model.plot(forecast);
```



In [108…

```
# plot individual components of forecast: trend, weekly/yearly seasonality,
model.plot_components(forecast);
```



Things to look into adjusting

In [109…

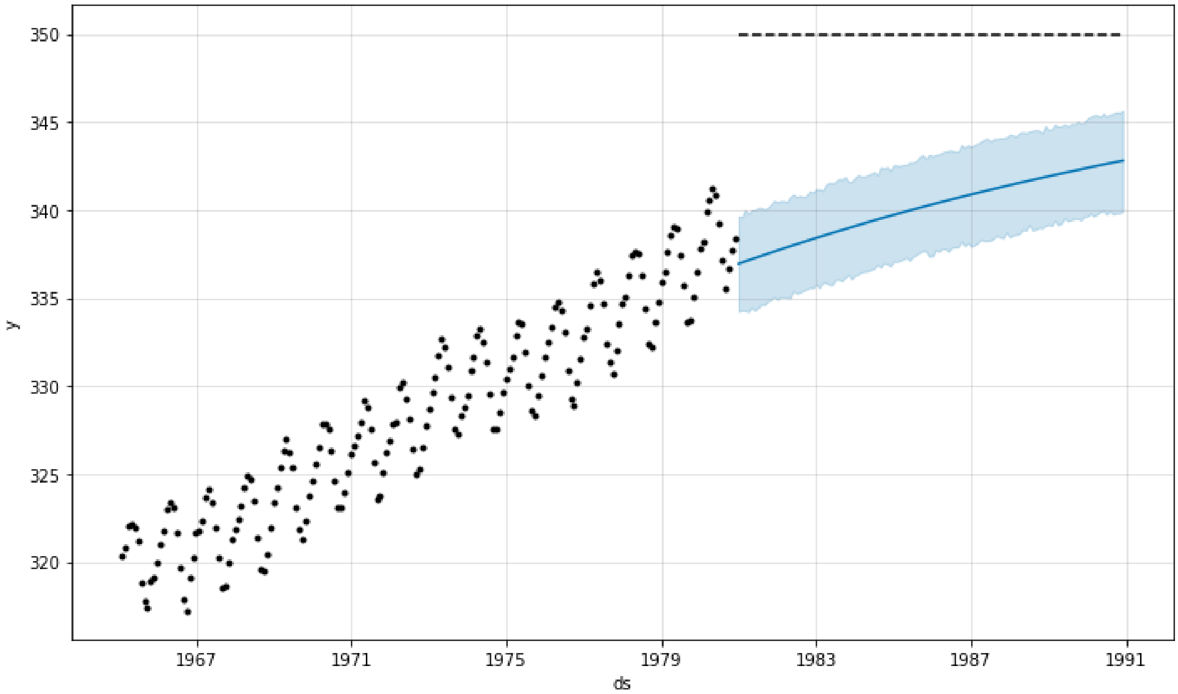
```
# Decreasted drastically from defaults
changepoint_prior_scale = 0.05
seasonality_prior_scale = 0.00001

growth='logistic'
```

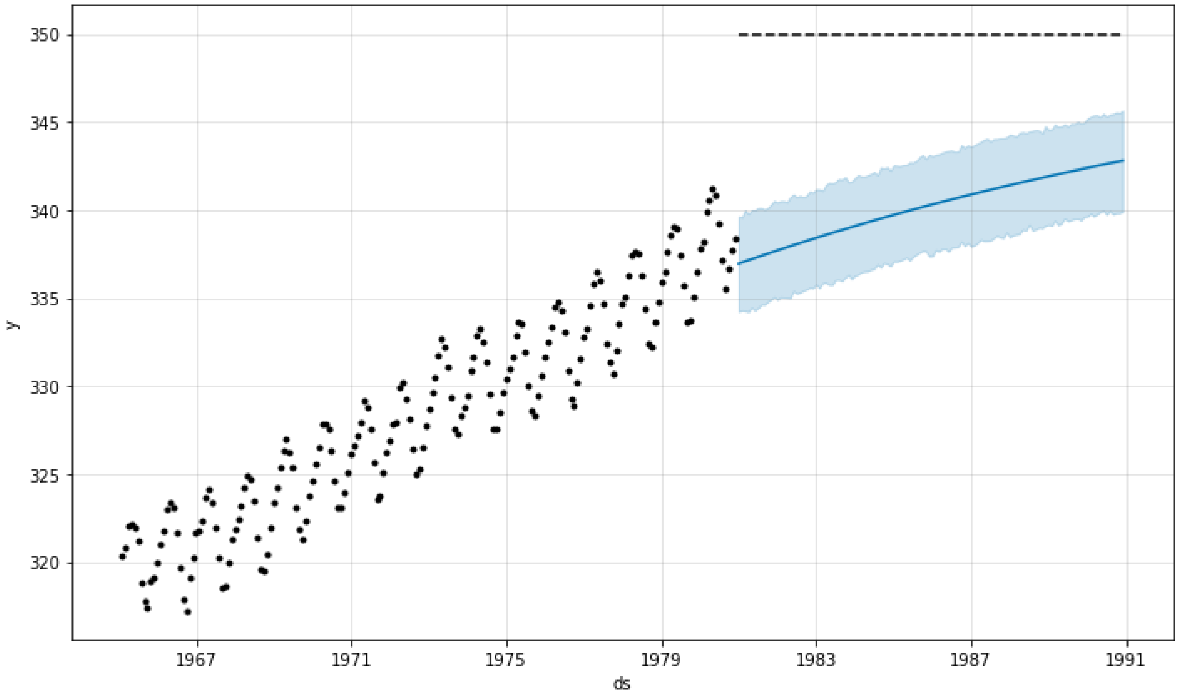
In [110…

```
# we can add a cap to limit our theoretical growth if we are using logistic growth
co2['cap'] = 350
m = Prophet(growth=growth, #weekly_seasonality=10000,

            seasonality_prior_scale=seasonality_prior_scale,
            changepoint_prior_scale=changepoint_prior_scale)
m.fit(co2);

# forecast 15 years into future with cap of 380
future = m.make_future_dataframe(periods=120,freq='M', include_history=False)
future['cap'] = 350

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

```
INFO:fbprophet:Disabling weekly seasonality. Run prophet with weekly_seasonality=T
rue to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=Tru
e to override this.
```

Out[110]:





Exercise

- Create prediction using hourly data of PM_Dongsi for last day half of december of 2015
  - Note you will have to use what we first learned to convert the year, month, day, hour columns to datetime object
- Plot daily and weekly seasonality of forecast

In [111…

```
df_Beijing = pd.read_csv('./FiveCitiesPM/Beijing.csv')
df_Beijing.head()
```

Out[111]:

	No	year	month	day	hour	season	PM_Dongsi	PM_Dongsihuan	PM_Nongzhanguan	PM_U Pos
0	1	2010	1	1	0	4	NaN	NaN	NaN	Na
1	2	2010	1	1	1	4	NaN	NaN	NaN	Na
2	3	2010	1	1	2	4	NaN	NaN	NaN	Na
3	4	2010	1	1	3	4	NaN	NaN	NaN	Na
4	5	2010	1	1	4	4	NaN	NaN	NaN	Na

In [112…

```
# Instructor solution
# Create a new column that is a datetime object
def make_date(row):
    return datetime(year = row['year'], month = row['month'], day = row['day'], hou

df_Beijing['date'] = df_Beijing.apply(make_date,axis=1)
# Make index for easy indexing of time values

df_Beijing.set_index('date',inplace=True)
df_Beijing['ds'] = df_Beijing.index

# Only take required fields
df = df_Beijing[["ds",'PM_Dongsi']]
df.rename(columns = {'PM_Dongsi':'y'},inplace=True)

# create a training set and a test set. We are only going to use last month's data
df_train = df['2015-11']
df_test = df['2015-12':'2015-12-15']
print(df_train.tail())
print(df_test.tail())
```

```
                                     ds      y
date
2015-11-30 19:00:00 2015-11-30 19:00:00  685.0
2015-11-30 20:00:00 2015-11-30 20:00:00  685.0
2015-11-30 21:00:00 2015-11-30 21:00:00  638.0
2015-11-30 22:00:00 2015-11-30 22:00:00  548.0
2015-11-30 23:00:00 2015-11-30 23:00:00  490.0
                                     ds    y
date
2015-12-15 19:00:00 2015-12-15 19:00:00  9.0
2015-12-15 20:00:00 2015-12-15 20:00:00  7.0
2015-12-15 21:00:00 2015-12-15 21:00:00  6.0
2015-12-15 22:00:00 2015-12-15 22:00:00  4.0
2015-12-15 23:00:00 2015-12-15 23:00:00  6.0
```

In [113…

```
# fit model
m = Prophet()
m.fit(df_train)
```

```
INFO:fbprophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=T
rue to override this.
```

Out[113]:

```
<fbprophet.forecaster.Prophet at 0x7fc81a731850>
```

In [114…

```
future = m.make_future_dataframe(periods = 15*24,freq = 'h') # could also leave de
print(future.head())
future.tail()
```

```
                   ds
0 2015-11-01 00:00:00
1 2015-11-01 01:00:00
2 2015-11-01 02:00:00
3 2015-11-01 03:00:00
4 2015-11-01 04:00:00
```

Out[114]:

	ds
1075	2015-12-15 19:00:00
1076	2015-12-15 20:00:00
1077	2015-12-15 21:00:00
1078	2015-12-15 22:00:00
1079	2015-12-15 23:00:00

In [115…

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

```
Index(['ds', 'trend', 'yhat_lower', 'yhat_upper', 'trend_lower', 'trend_upper',
       'additive_terms', 'additive_terms_lower', 'additive_terms_upper',
       'daily', 'daily_lower', 'daily_upper', 'weekly', 'weekly_lower',
       'weekly_upper', 'multiplicative_terms', 'multiplicative_terms_lower',
       'multiplicative_terms_upper', 'yhat'],
      dtype='object')
```

Out[115]:

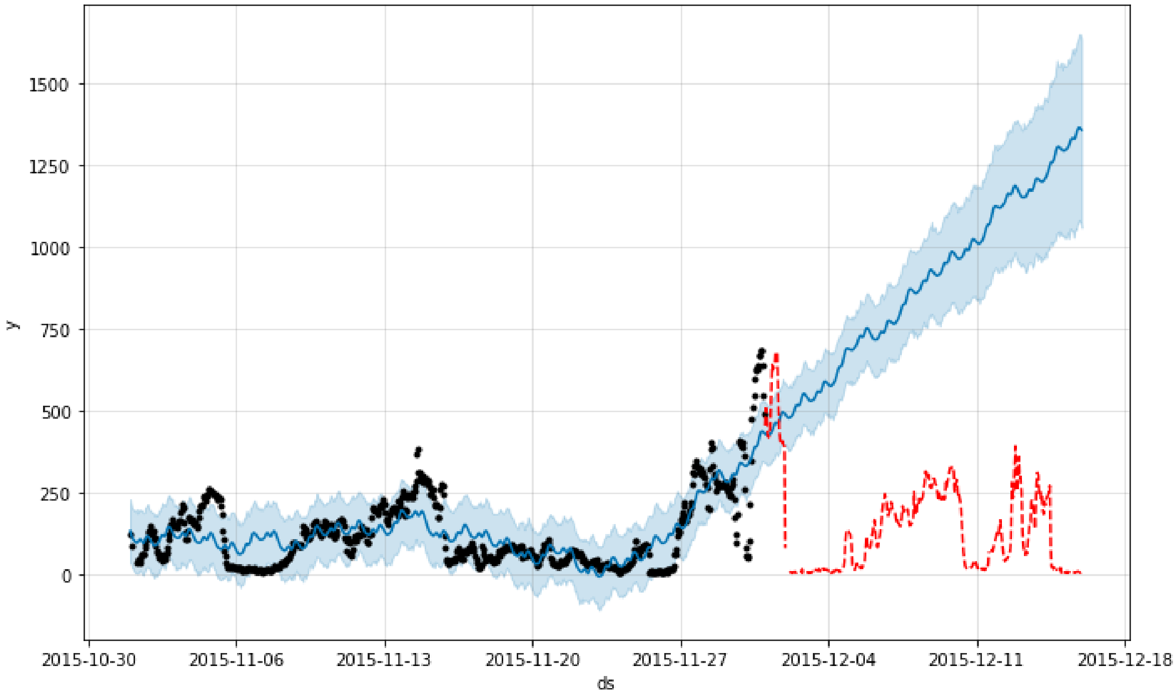
	ds	yhat	yhat_lower	yhat_upper
1075	2015-12-15 19:00:00	1361.302132	1073.065506	1634.896592
1076	2015-12-15 20:00:00	1363.598920	1081.769522	1648.047760
1077	2015-12-15 21:00:00	1362.396483	1073.093923	1648.953237
1078	2015-12-15 22:00:00	1359.504008	1076.761574	1646.262377
1079	2015-12-15 23:00:00	1356.114245	1057.776719	1632.299111

In [116…

```
m.plot(forecast)
plt.plot(df_test.y,'r--')
```

Out[116]:

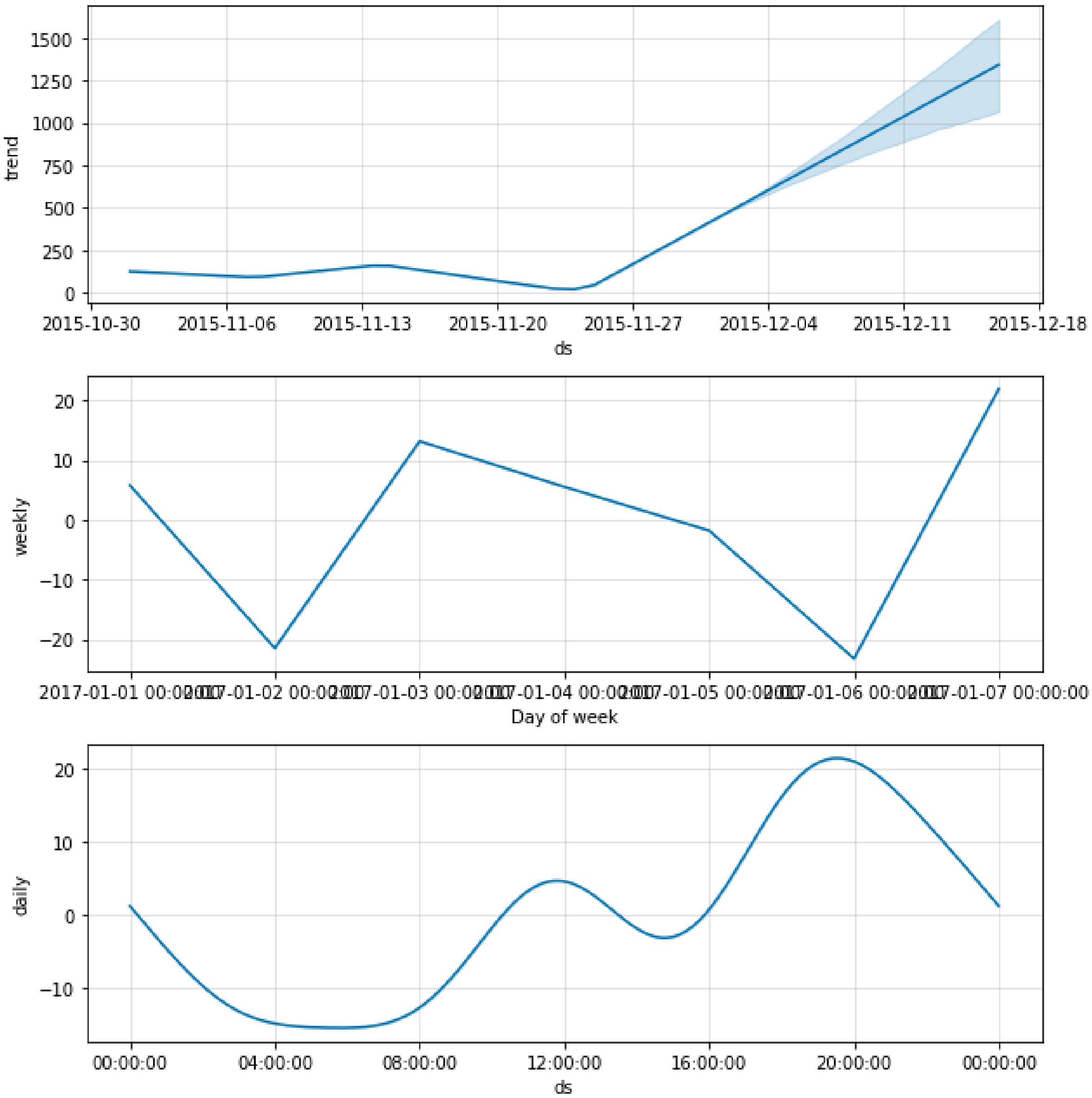
```
[<matplotlib.lines.Line2D at 0x7fc81acc7b50>]
```

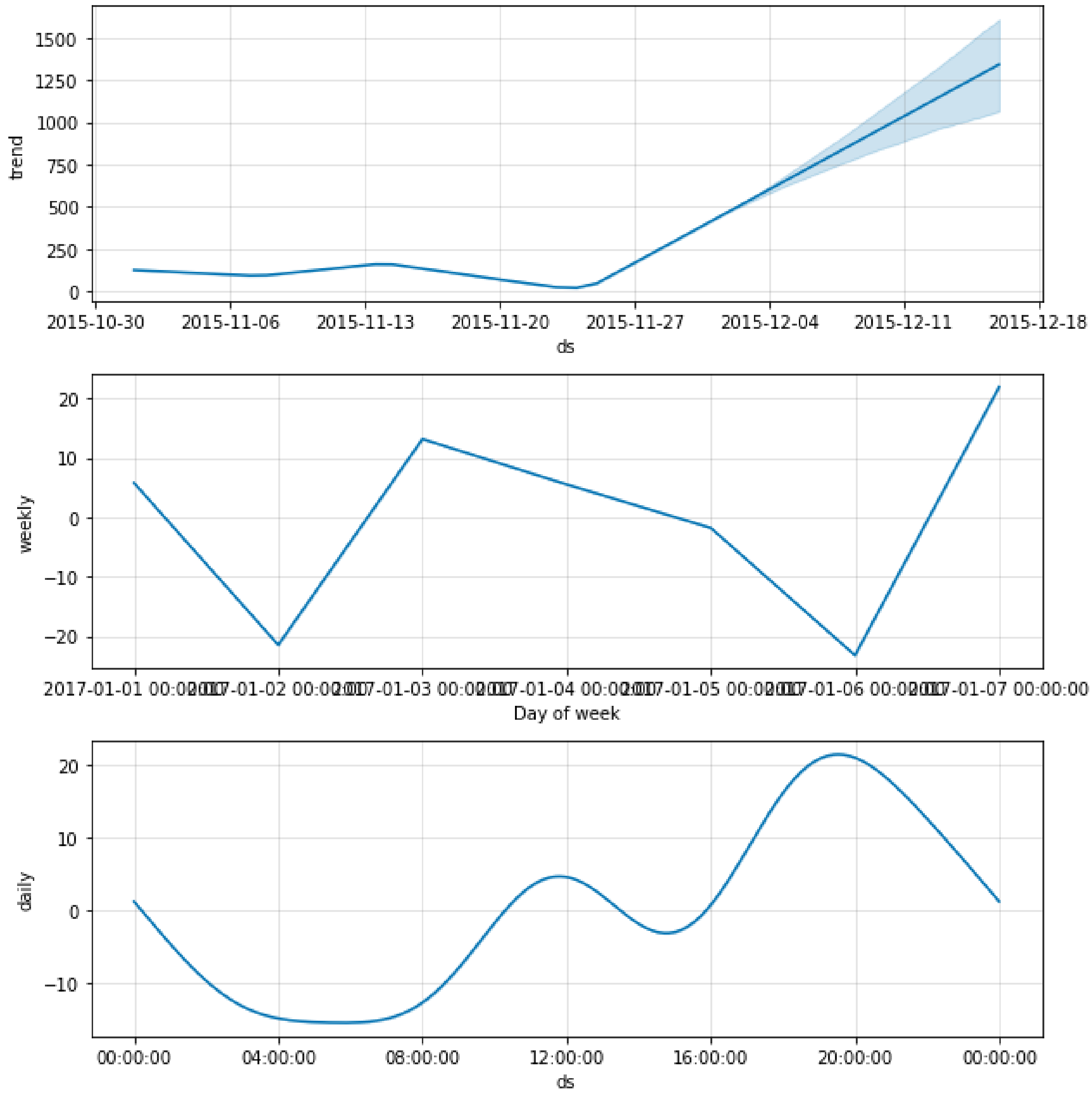


In [117…

```
m.plot_components(forecast)
```

Out[117]:





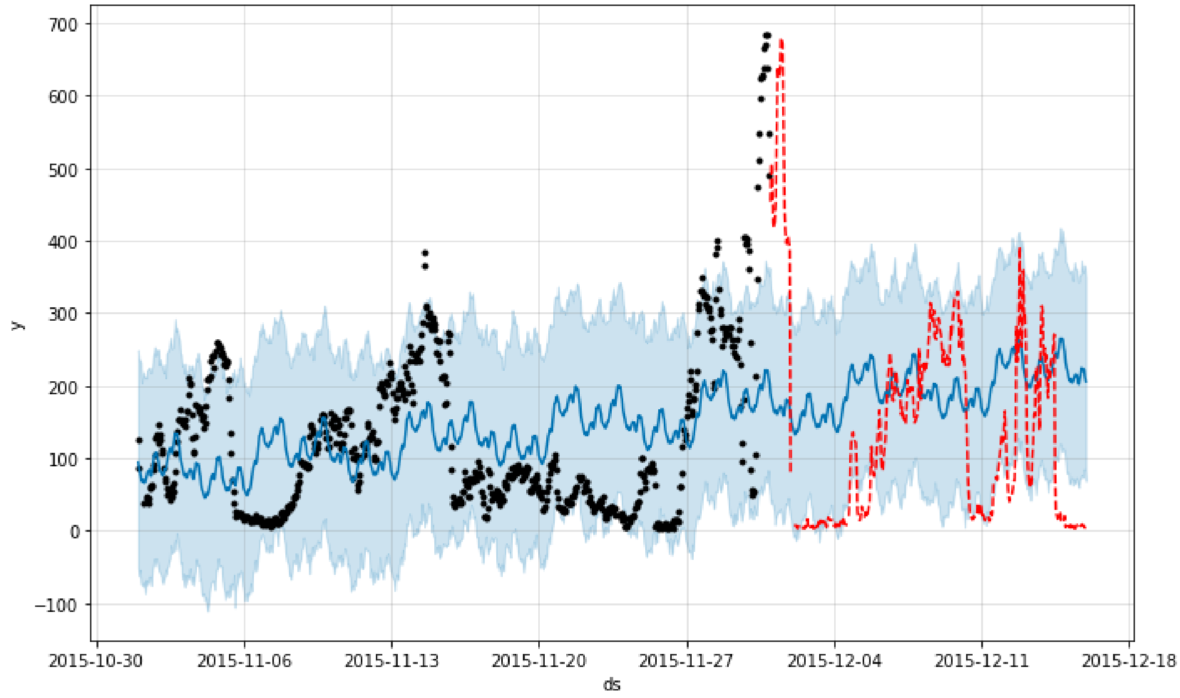
In [118…

```
# Decreasted drastically from defaults
changepoint_prior_scale = 0.0005
seasonality_prior_scale = 10


m = Prophet(daily_seasonality=10,

            seasonality_prior_scale=seasonality_prior_scale,
            changepoint_prior_scale=changepoint_prior_scale)
m.fit(df_train);


forecast = m.predict(future)
m.plot(forecast)
plt.plot(df_test.y,'r--')
```

```
INFO:fbprophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=T
rue to override this.
WARNING:fbprophet:Optimization terminated abnormally. Falling back to Newton.
```

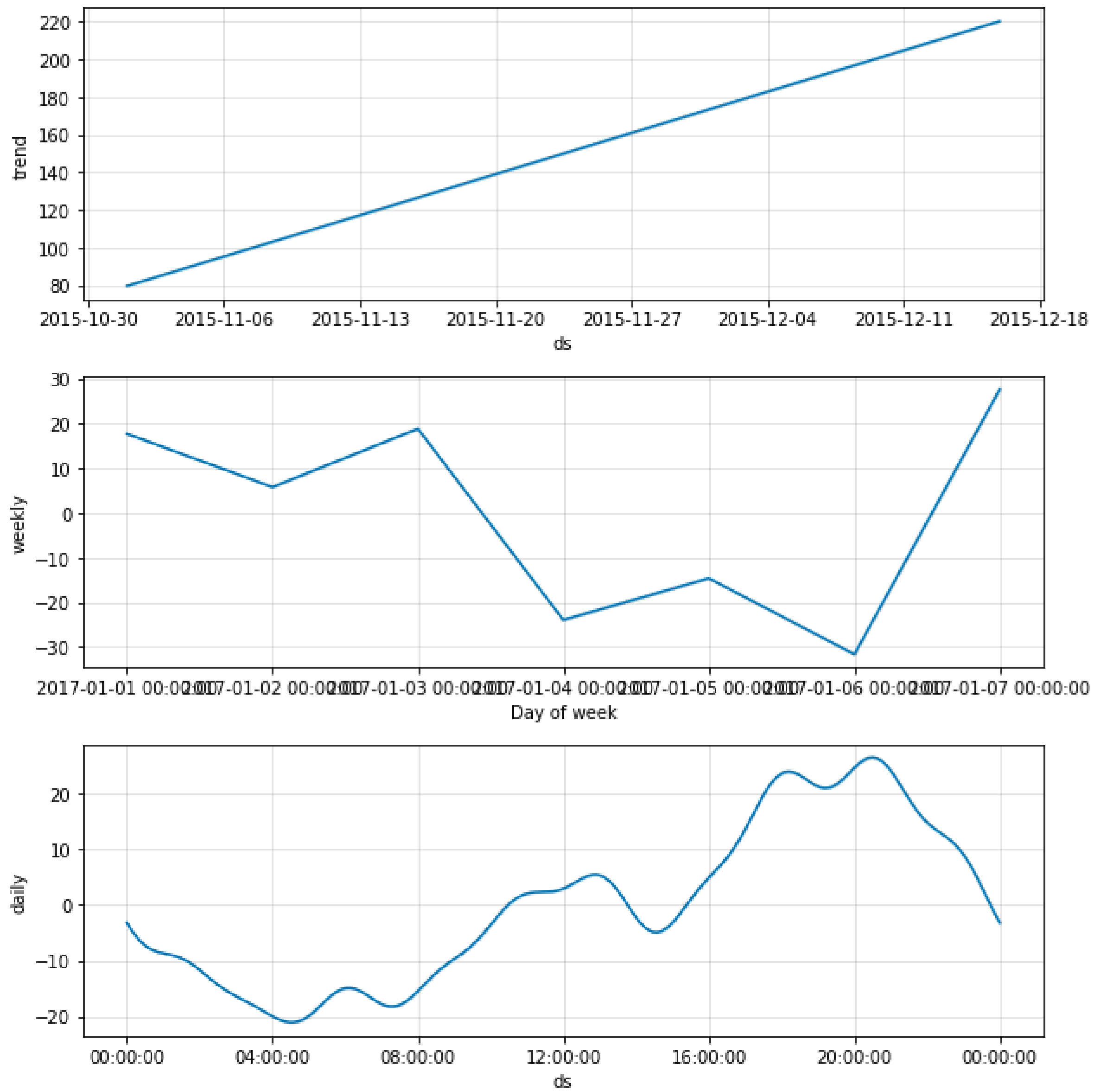
Out[118]:

```
[<matplotlib.lines.Line2D at 0x7fc819b5bf50>]
```



In [119…

```
m.plot_components(forecast);
```



Summary

In this notebook, we have covered:

1. A practical understanding of Autoregressive Moving Average (ARMA) models.
2. A basic understanding of the Autocorrelation Function (ACF).
3. Insight into choosing the order q of MA models.
4. A practical understanding of Autoregressive (AR) models.
5. A basic understanding of the Partial Autocorrelation Function (PACF).
6. Insight into choosing the order p of AR models.
