Predict Sales with Time-series Models

Predicting total sales for every product and store in the next month for Russian Software company.

In [1]:
```
# check out the files
!ls ../input/*
```

```
../input/item_categories.csv  ../input/sample_submission.csv
../input/items.csv            ../input/shops.csv
../input/sales_train.csv      ../input/test.csv
```

In [2]:
```
import numpy as np
import pandas as pd
import random as rd
import datetime

import matplotlib.pyplot as plt
import seaborn as sns


# TIME SERIES
from statsmodels.tsa.arima_model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from pandas.plotting import autocorrelation_plot
from statsmodels.tsa.stattools import adfuller, acf, pacf,arma_order_select_ic
import statsmodels.formula.api as smf
import statsmodels.tsa.api as smt
import statsmodels.api as sm
import scipy.stats as scs


# settings
import warnings
warnings.filterwarnings("ignore")
```

```
/opt/conda/lib/python3.6/site-packages/statsmodels/compat/pandas.py:56: FutureWarning: The pandas.core.dateto
ols module is deprecated and will be removed in a future version. Please use the pandas.tseries module instea
d.
  from pandas.core import datetools
```

In [3]:
```
# Import all of them
sales=pd.read_csv("../input/sales_train.csv")

# settings
import warnings
warnings.filterwarnings("ignore")

item_cat=pd.read_csv("../input/item_categories.csv")
item=pd.read_csv("../input/items.csv")

shops=pd.read_csv("../input/shops.csv")
test=pd.read_csv("../input/test.csv")
```

In [4]:
```
#formatting the date column correctly
sales.date=sales.date.apply(lambda x:datetime.datetime.strptime(x, '%d.%m.%Y'))
# check
print(sales.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2935849 entries, 0 to 2935848
Data columns (total 6 columns):
date              datetime64[ns]
date_block_num    int64
shop_id           int64
item_id           int64
item_price        float64
item_cnt_day      float64
dtypes: datetime64[ns](1), float64(2), int64(3)
memory usage: 134.4 MB
None
```

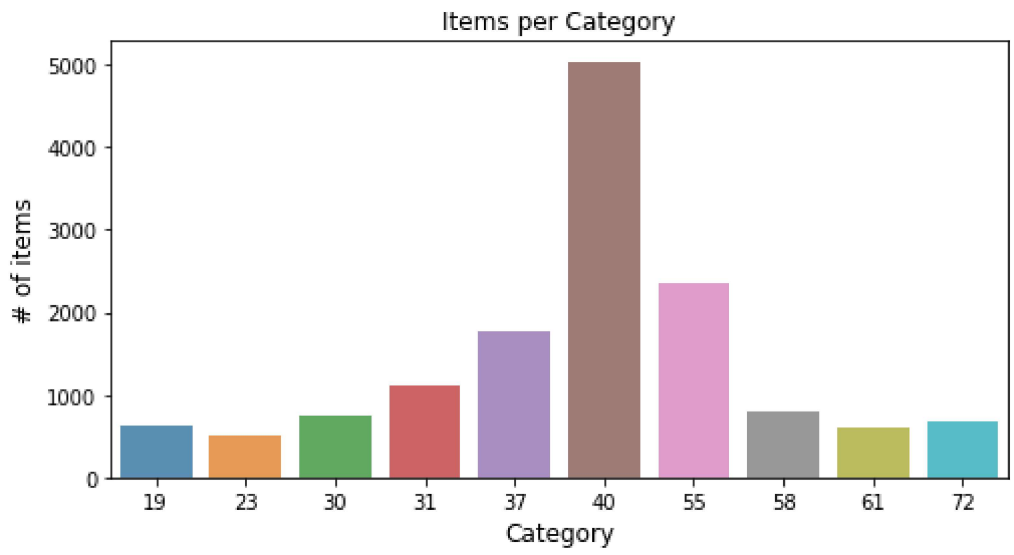
In [5]:
```
# Aggregate to monthly level the required metrics

monthly_sales=sales.groupby(["date_block_num","shop_id","item_id"])[
    "date","item_price","item_cnt_day"].agg({"date":["min",'max'],"item_price":"mean","item_cnt_day":"sum"})

## Lets break down the line of code here:
# aggregate by date-block(month),shop_id and item_id
# select the columns date,item_price and item_cnt(sales)
# Provide a dictionary which says what aggregation to perform on which column
# min and max on the date
# average of the item_price
# sum of the sales
```

In [6]:
```
monthly_sales.head(20)
```

Out[6]:

			date		item_price	item_cnt_day
			min	max	mean	sum
date_block_num	shop_id	item_id				
0	0	32	2013-01-03	2013-01-31	221.0	6.0
		33	2013-01-03	2013-01-28	347.0	3.0
		35	2013-01-31	2013-01-31	247.0	1.0
		43	2013-01-31	2013-01-31	221.0	1.0
		51	2013-01-13	2013-01-31	128.5	2.0
		61	2013-01-10	2013-01-10	195.0	1.0
		75	2013-01-17	2013-01-17	76.0	1.0
		88	2013-01-16	2013-01-16	76.0	1.0
		95	2013-01-06	2013-01-06	193.0	1.0
		96	2013-01-10	2013-01-10	70.0	1.0
		98	2013-01-04	2013-01-31	268.0	25.0
		111	2013-01-17	2013-01-17	89.0	1.0
		149	2013-01-11	2013-01-28	99.0	3.0
		151	2013-01-16	2013-01-16	75.0	1.0
		153	2013-01-09	2013-01-09	258.0	1.0
		198	2013-01-10	2013-01-10	112.0	1.0
		210	2013-01-05	2013-01-25	118.0	2.0
		282	2013-01-04	2013-01-04	109.0	1.0
		306	2013-01-22	2013-01-22	59.0	1.0
		351	2013-01-21	2013-01-21	89.0	1.0

In [7]:
```
# number of items per category
x=item.groupby(['item_category_id']).count()
x=x.sort_values(by='item_id',ascending=False)
x=x.iloc[0:10].reset_index()
x
# #plot
plt.figure(figsize=(8,4))
ax= sns.barplot(x.item_category_id, x.item_id, alpha=0.8)
plt.title("Items per Category")
plt.ylabel('# of items', fontsize=12)
plt.xlabel('Category', fontsize=12)
plt.show()
```



Single series:

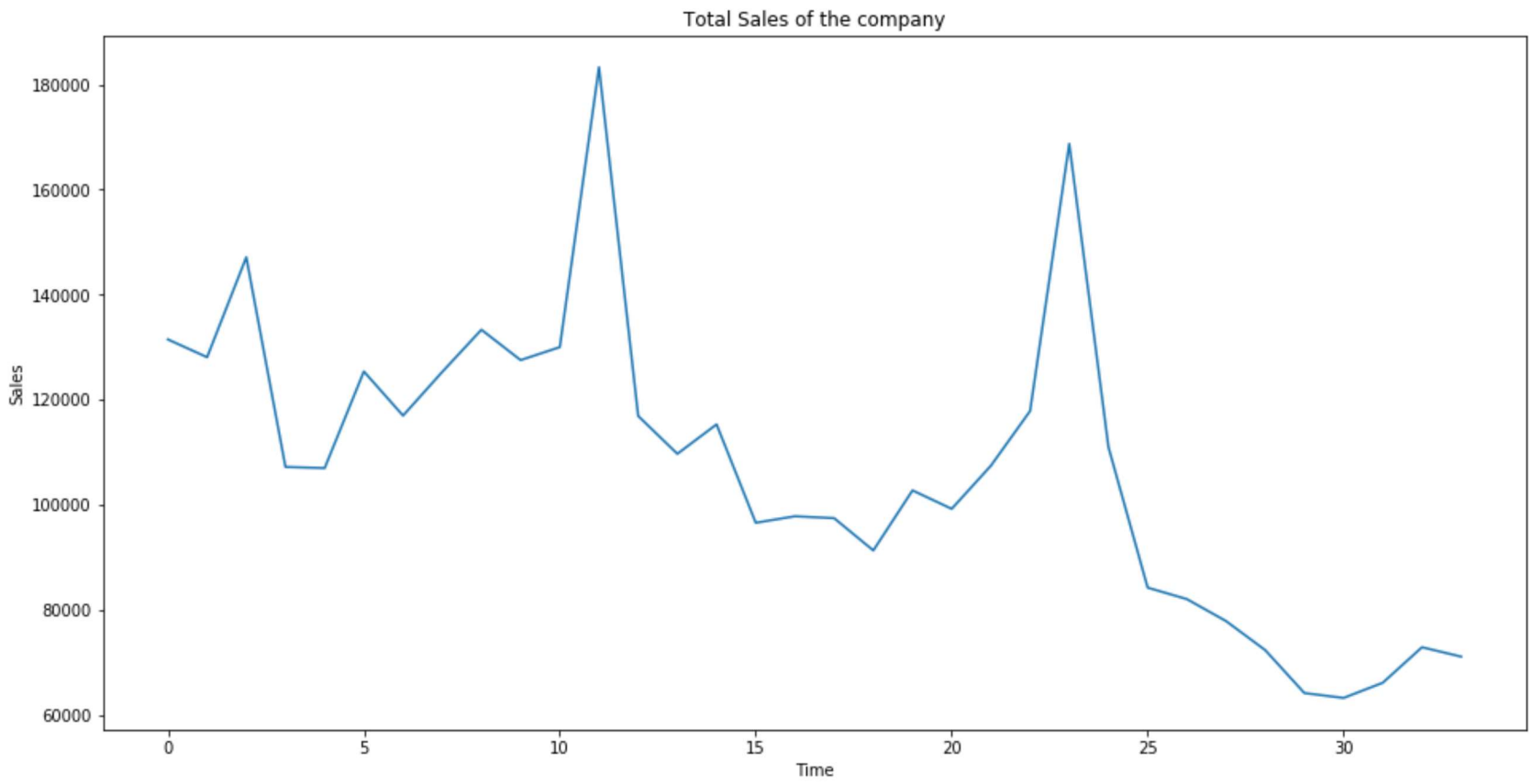
The objective requires us to predict sales for the next month at a store-item combination.

Sales over time of each store-item is a time-series in itself. Before we dive into all the combinations, first let's understand how to forecast for a single series.

I've chosen to predict for the total sales per month for the entire company.

First let's compute the total sales per month and plot that data.

In [8]:
```
ts=sales.groupby(["date_block_num"])["item_cnt_day"].sum()
ts.astype('float')
plt.figure(figsize=(16,8))
plt.title('Total Sales of the company')
plt.xlabel('Time')
plt.ylabel('Sales')
plt.plot(ts);
```

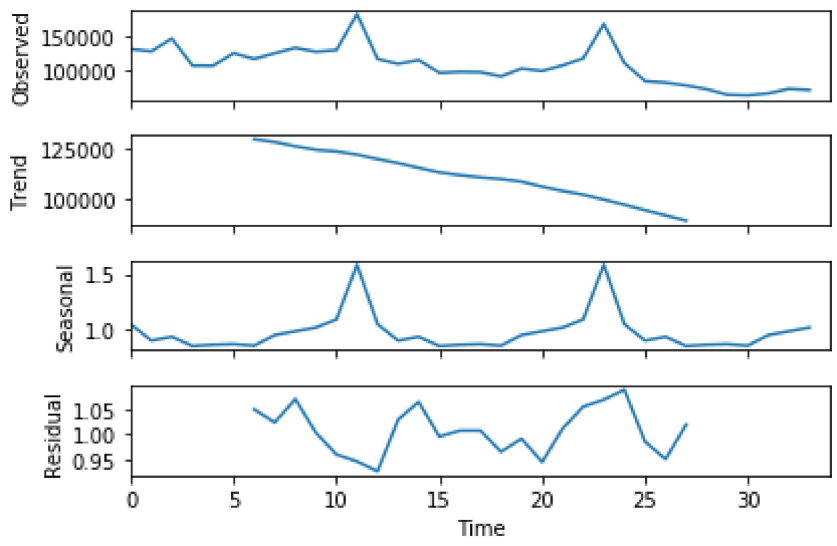


In [9]:
```
plt.figure(figsize=(16,6))
plt.plot(ts.rolling(window=12,center=False).mean(),label='Rolling Mean');
plt.plot(ts.rolling(window=12,center=False).std(),label='Rolling sd');
plt.legend();
```

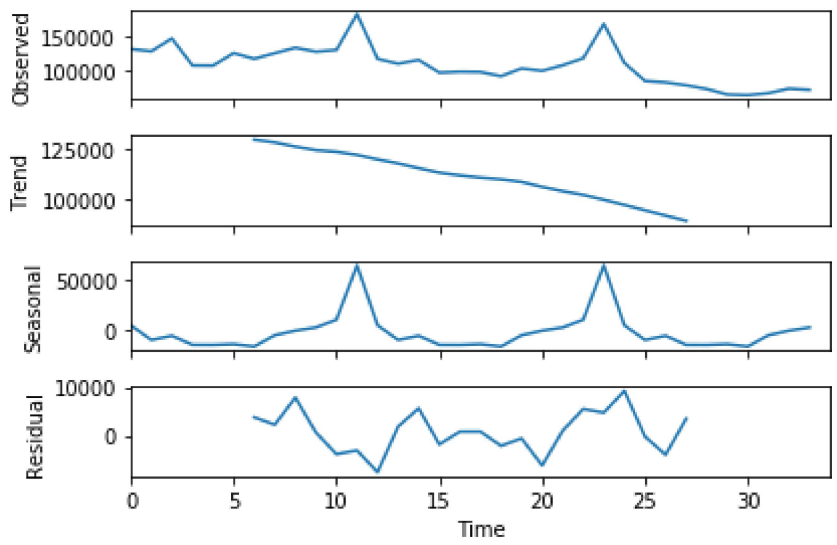

Quick observations: There is an obvious "seasonality" (Eg: peak sales around a time of year) and a decreasing "Trend".

Let's check that with a quick decomposition into Trend, seasonality and residuals.

In [10]:
```
import statsmodels.api as sm
# multiplicative
res = sm.tsa.seasonal_decompose(ts.values,freq=12,model="multiplicative")
#plt.figure(figsize=(16,12))
fig = res.plot()
#fig.show()
```



In [11]:
```
# Additive model
res = sm.tsa.seasonal_decompose(ts.values,freq=12,model="additive")
#plt.figure(figsize=(16,12))
fig = res.plot()
#fig.show()
```



we assume an additive model, then we can write

> yt=St+Tt+Et

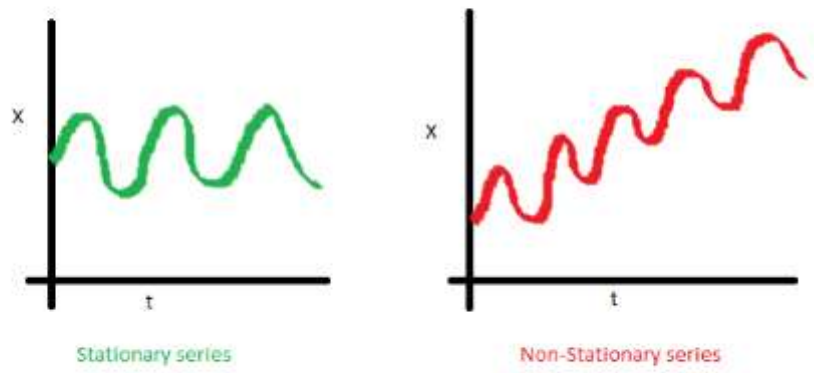
where yt is the data at period t, St is the seasonal component at period t, Tt is the trend-cycle component at period tt and Et is the remainder (or irregular or error) component at period t Similarly for Multiplicative model,

> yt=St x Tt x Et

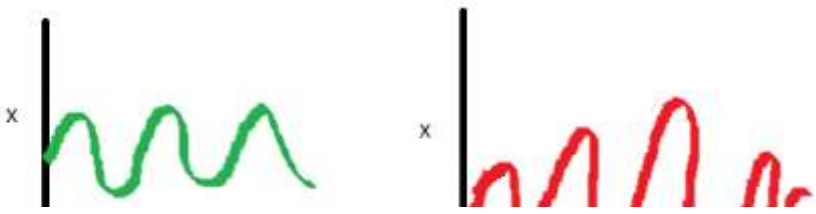
Stationarity:

What does it mean for data to be stationary?

1. The mean of the series should not be a function of time. The red graph below is not stationary because the mean increases over time.



2. The variance of the series should not be a function of time. This property is known as homoscedasticity. Notice in the red graph the varying spread of data over time.



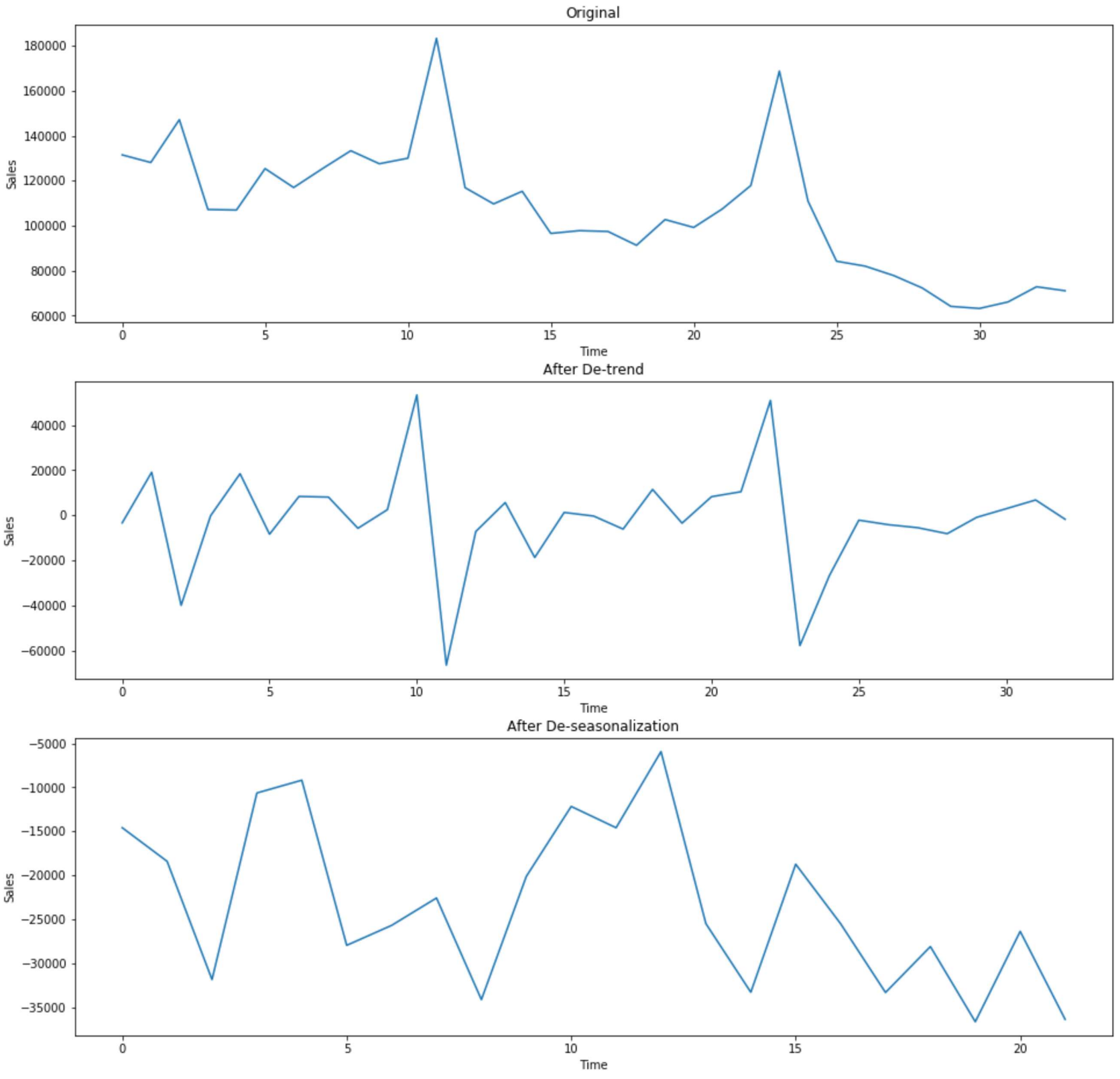
In [13]:
```
# Stationarity tests
def test_stationarity(timeseries):

    #Perform Dickey-Fuller test:
    print('Results of Dickey-Fuller Test:')
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Number of Observations U
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print (dfoutput)

test_stationarity(ts)
```

```
Results of Dickey-Fuller Test:
Test Statistic                -2.395704
p-value                        0.142953
#Lags Used                     0.000000
Number of Observations Used   33.000000
Critical Value (1%)           -3.646135
Critical Value (5%)           -2.954127
Critical Value (10%)          -2.615968
dtype: float64
```

In [14]:
```
# to remove trend
from pandas import Series as Series
# create a differenced series
def difference(dataset, interval=1):
    diff = list()
    for i in range(interval, len(dataset)):
        value = dataset[i] - dataset[i - interval]
        diff.append(value)
    return Series(diff)

# invert differenced forecast
def inverse_difference(last_ob, value):
    return value + last_ob
```

In [15]:
```
ts=sales.groupby(["date_block_num"])["item_cnt_day"].sum()
ts.astype('float')
plt.figure(figsize=(16,16))
plt.subplot(311)
plt.title('Original')
plt.xlabel('Time')
plt.ylabel('Sales')
plt.plot(ts)
plt.subplot(312)
plt.title('After De-trend')
plt.xlabel('Time')
plt.ylabel('Sales')
new_ts=difference(ts)
plt.plot(new_ts)
plt.plot()

plt.subplot(313)
plt.title('After De-seasonalization')
plt.xlabel('Time')
plt.ylabel('Sales')
new_ts=difference(ts,12)       # assuming the seasonality is 12 months long
plt.plot(new_ts)
plt.plot()
```

Out[15]: []



In [16]:
```
# now testing the stationarity again after de-seasonality
test_stationarity(new_ts)
```

```
Results of Dickey-Fuller Test:
Test Statistic                -3.270101
p-value                        0.016269
#Lags Used                     0.000000
Number of Observations Used   21.000000
Critical Value (1%)           -3.788386
Critical Value (5%)           -3.013098
Critical Value (10%)          -2.646397
dtype: float64
```

Now after the transformations, our p-value for the DF test is well within 5 %. Hence we can

assume Stationarity of the series

We can easily get back the original series using the inverse transform function that we have defined above.

AR, MA and ARMA models:

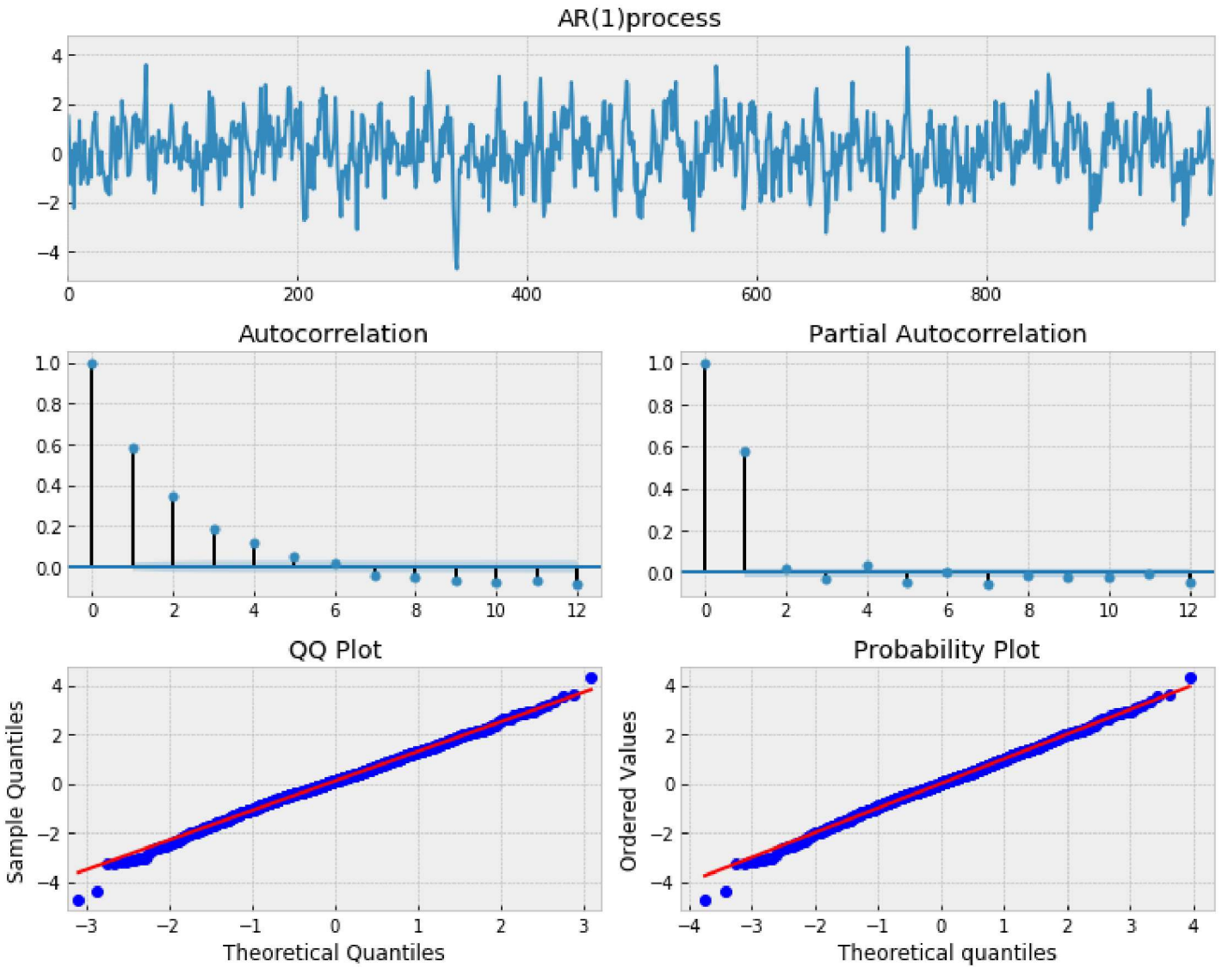
TL: DR version of the models:

MA - Next value in the series is a function of the average of the previous n number of values AR - The errors(difference in mean) of the next value is a function of the errors in the previous n number of values ARMA - a mixture of both.

Now, How do we find out, if our time-series in AR process or MA process?

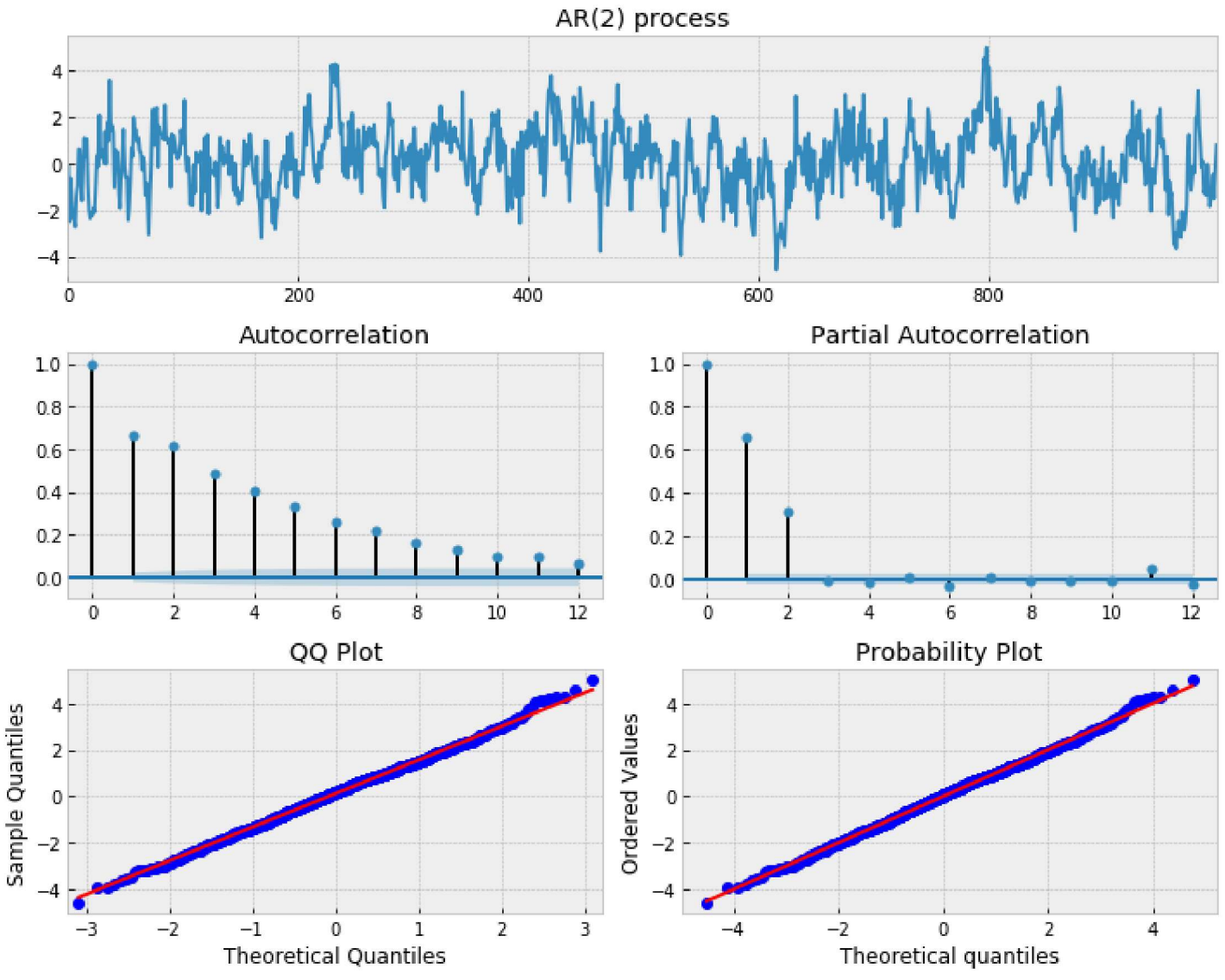
In [17]:
```
def tsplot(y, lags=None, figsize=(10, 8), style='bmh',title=''):
    if not isinstance(y, pd.Series):
        y = pd.Series(y)
    with plt.style.context(style):
        fig = plt.figure(figsize=figsize)
        #mpl.rcParams['font.family'] = 'Ubuntu Mono'
        layout = (3, 2)
        ts_ax = plt.subplot2grid(layout, (0, 0), colspan=2)
        acf_ax = plt.subplot2grid(layout, (1, 0))
        pacf_ax = plt.subplot2grid(layout, (1, 1))
        qq_ax = plt.subplot2grid(layout, (2, 0))
        pp_ax = plt.subplot2grid(layout, (2, 1))

        y.plot(ax=ts_ax)
        ts_ax.set_title(title)
        smt.graphics.plot_acf(y, lags=lags, ax=acf_ax, alpha=0.5)
        smt.graphics.plot_pacf(y, lags=lags, ax=pacf_ax, alpha=0.5)
        sm.qqplot(y, line='s', ax=qq_ax)
        qq_ax.set_title('QQ Plot')
        scs.probplot(y, sparams=(y.mean(), y.std()), plot=pp_ax)

        plt.tight_layout()
    return
```

In [18]:
```
# Simulate an AR(1) process with alpha = 0.6
np.random.seed(1)
n_samples = int(1000)
a = 0.6
x = w = np.random.normal(size=n_samples)

for t in range(n_samples):
    x[t] = a*x[t-1] + w[t]
limit=12
_ = tsplot(x, lags=limit,title="AR(1)process")
```



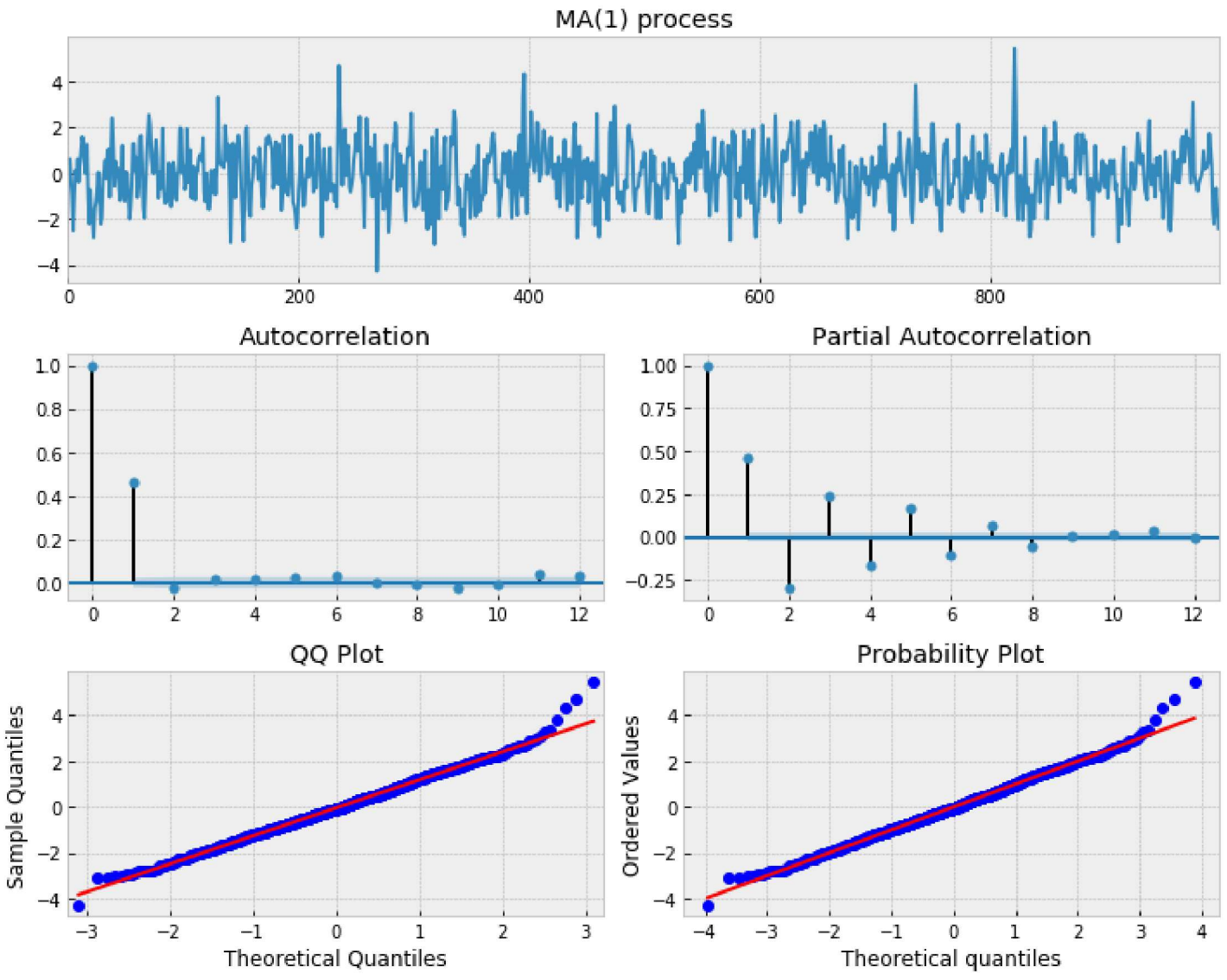
AR(1) process -- has ACF tailing out and PACF cutting off at lag=1

In [19]:
```
# Simulate an AR(2) process

n = int(1000)
alphas = np.array([.444, .333])
betas = np.array([0.])

# Python requires us to specify the zero-lag value which is 1
# Also note that the alphas for the AR model must be negated
# We also set the betas for the MA equal to 0 for an AR(p) model
# For more information see the examples at statsmodels.org
ar = np.r_[1, -alphas]
ma = np.r_[1, betas]

ar2 = smt.arma_generate_sample(ar=ar, ma=ma, nsample=n)
_ = tsplot(ar2, lags=12,title="AR(2) process")
```



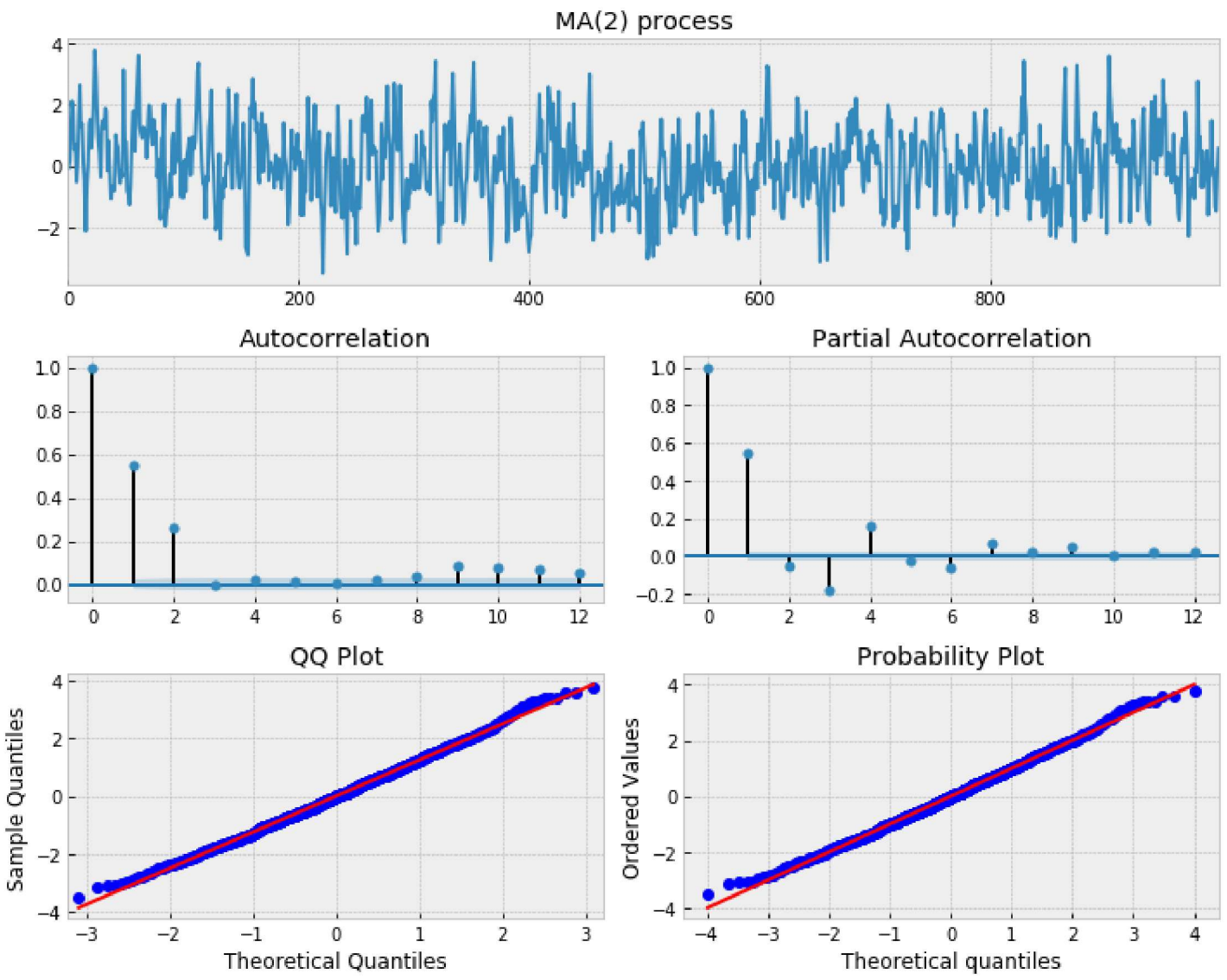
AR(2) process -- has ACF tailing out and PACF cutting off at lag=2

In [20]:
```
# Simulate an MA(1) process
n = int(1000)
# set the AR(p) alphas equal to 0
alphas = np.array([0.])
betas = np.array([0.8])
# add zero-lag and negate alphas
ar = np.r_[1, -alphas]
ma = np.r_[1, betas]
ma1 = smt.arma_generate_sample(ar=ar, ma=ma, nsample=n)
limit=12
_ = tsplot(ma1, lags=limit,title="MA(1) process")
```



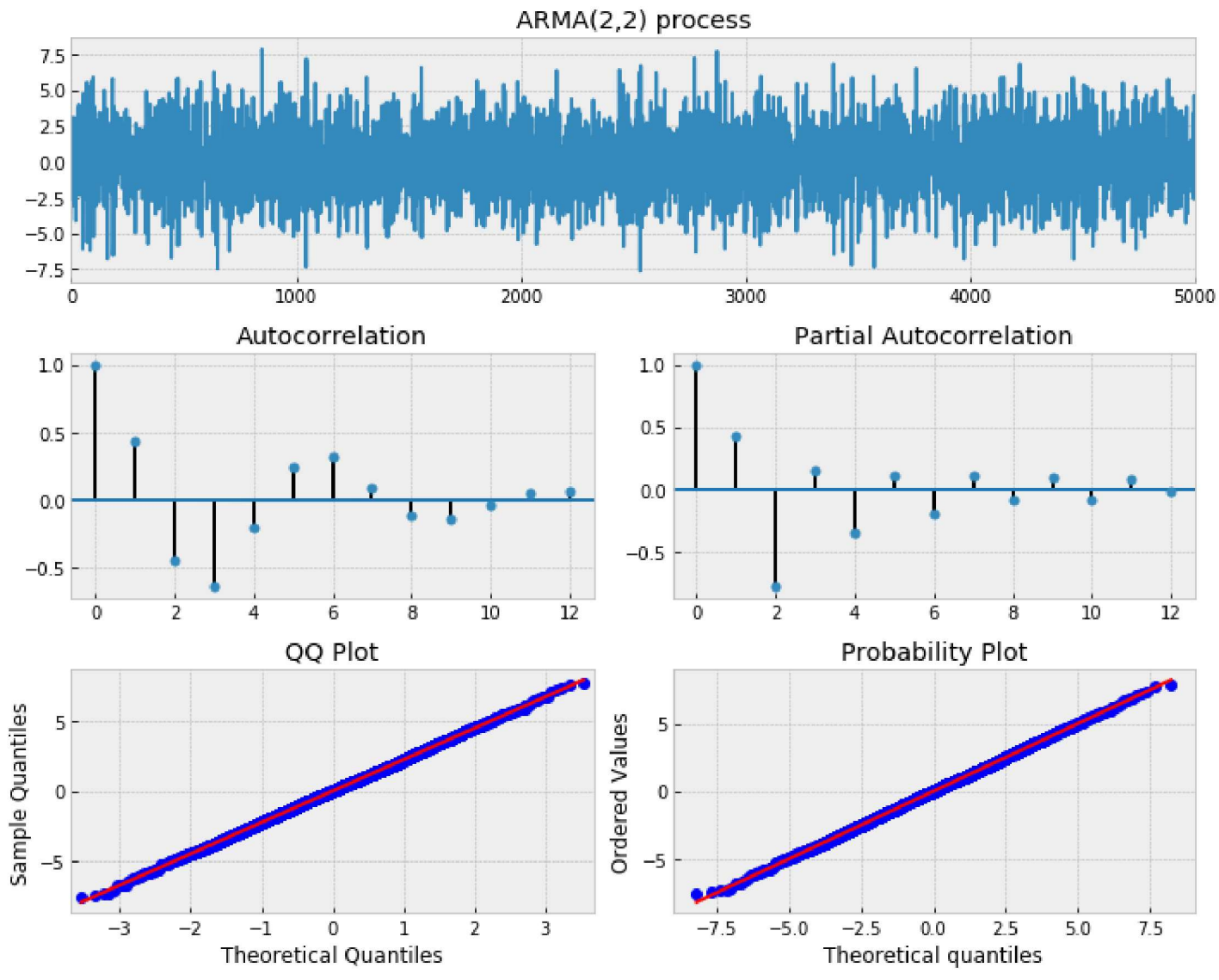
MA(1) process -- has ACF cut off at lag=1

In [21]:
```
# Simulate MA(2) process with betas 0.6, 0.4
n = int(1000)
alphas = np.array([0.])
betas = np.array([0.6, 0.4])
ar = np.r_[1, -alphas]
ma = np.r_[1, betas]

ma3 = smt.arma_generate_sample(ar=ar, ma=ma, nsample=n)
_ = tsplot(ma3, lags=12,title="MA(2) process")
```



MA(2) process -- has ACF cut off at lag=2

In [22]:
```
# Simulate an ARMA(2, 2) model with alphas=[0.5,-0.25] and betas=[0.5,-0.3]
max_lag = 12

n = int(5000) # lots of samples to help estimates
burn = int(n/10) # number of samples to discard before fit

alphas = np.array([0.8, -0.65])
betas = np.array([0.5, -0.7])
ar = np.r_[1, -alphas]
ma = np.r_[1, betas]

arma22 = smt.arma_generate_sample(ar=ar, ma=ma, nsample=n, burnin=burn)
_ = tsplot(arma22, lags=max_lag,title="ARMA(2,2) process")
```



Now things get a little hazy. Its not very clear/straight-forward.

A nifty summary of the above plots:

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.
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

Let's use a systematic approach to find the order of AR and MA processes.

In [23]:
```
# pick best order by aic
# smallest aic value wins
best_aic = np.inf
best_order = None
best_mdl = None

rng = range(5)
for i in rng:
    for j in rng:
        try:
            tmp_mdl = smt.ARMA(arma22, order=(i, j)).fit(method='mle', trend='nc')
            tmp_aic = tmp_mdl.aic
            if tmp_aic < best_aic:
                best_aic = tmp_aic
                best_order = (i, j)
                best_mdl = tmp_mdl
        except: continue


print('aic: {:6.5f} | order: {}'.format(best_aic, best_order))
```

```
aic: 15326.68109 | order: (2, 2)
```

We've correctly identified the order of the simulated process as ARMA(2,2).

Lets use it for the sales time-series.

In [24]:
```
#
# pick best order by aic
# smallest aic value wins
best_aic = np.inf
best_order = None
best_mdl = None

rng = range(5)
for i in rng:
    for j in rng:
        try:
            tmp_mdl = smt.ARMA(new_ts.values, order=(i, j)).fit(method='mle', trend='nc')
            tmp_aic = tmp_mdl.aic
            if tmp_aic < best_aic:
                best_aic = tmp_aic
                best_order = (i, j)
                best_mdl = tmp_mdl
        except: continue


print('aic: {:6.5f} | order: {}'.format(best_aic, best_order))
```

```
aic: 472.99703 | order: (1, 1)
```

In [25]:
```
# Simply use best_mdl.predict() to predict the next values
```

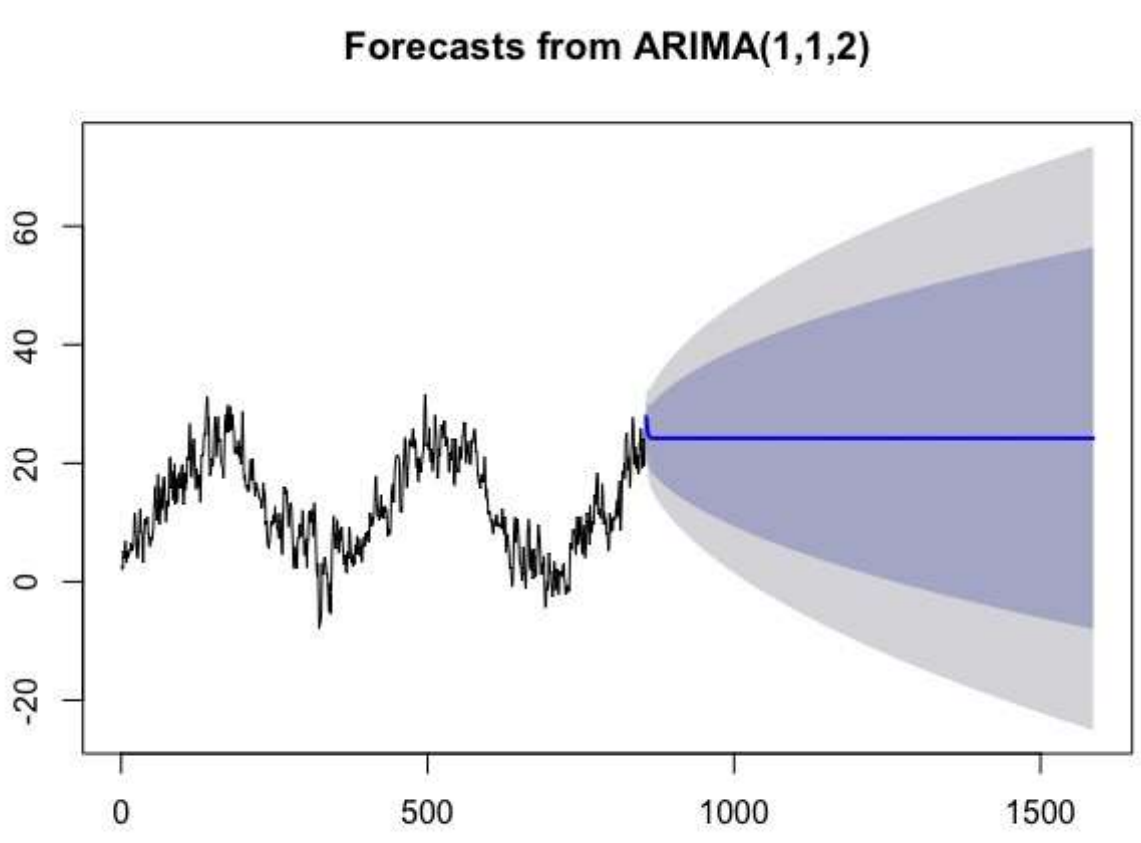
In [26]:
```
# adding the dates to the Time-series as index
ts=sales.groupby(["date_block_num"])["item_cnt_day"].sum()
ts.index=pd.date_range(start = '2013-01-01',end='2015-10-01', freq = 'MS')
ts=ts.reset_index()
ts.head()
```

Out[26]:

	index	item_cnt_day
0	2013-01-01	131479.0
1	2013-02-01	128090.0
2	2013-03-01	147142.0
3	2013-04-01	107190.0
4	2013-05-01	106970.0

Prophet:

Recently open-sourced by Facebook research. It's a very promising tool, that is often a very handy and quick solution to the frustrating **flatline** :P



Sure, one could argue that with proper pre-processing and carefully tuning the parameters the above graph would not happen.

But the truth is that most of us don't either have the patience or the expertise to make it happen.

Also, there is the fact that in most practical scenarios- there is often a lot of time-series that needs to be predicted. Eg: This competition. It requires us to predict the next month sales for the **Store - item level combinations** which could be in the thousands.(ie) predict 1000s of parameters!

Another neat functionality is that it follows the typical **sklearn** syntax.

At its core, the Prophet procedure is an additive regression model with four main components:

- A piecewise linear or logistic growth curve trend. Prophet automatically detects changes in trends by selecting changepoints from

In [27]:
```
from fbprophet import Prophet
#prophet reqiures a pandas df at the below config
# ( date column named as DS and the value column as Y)
ts.columns=['ds','y']
model = Prophet( yearly_seasonality=True) #instantiate Prophet with only yearly seasonality as our data is mon
model.fit(ts) #fit the model with your dataframe
```

```
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
```

Out[27]:
```
<fbprophet.forecaster.Prophet at 0x7fa643651dd8>
```

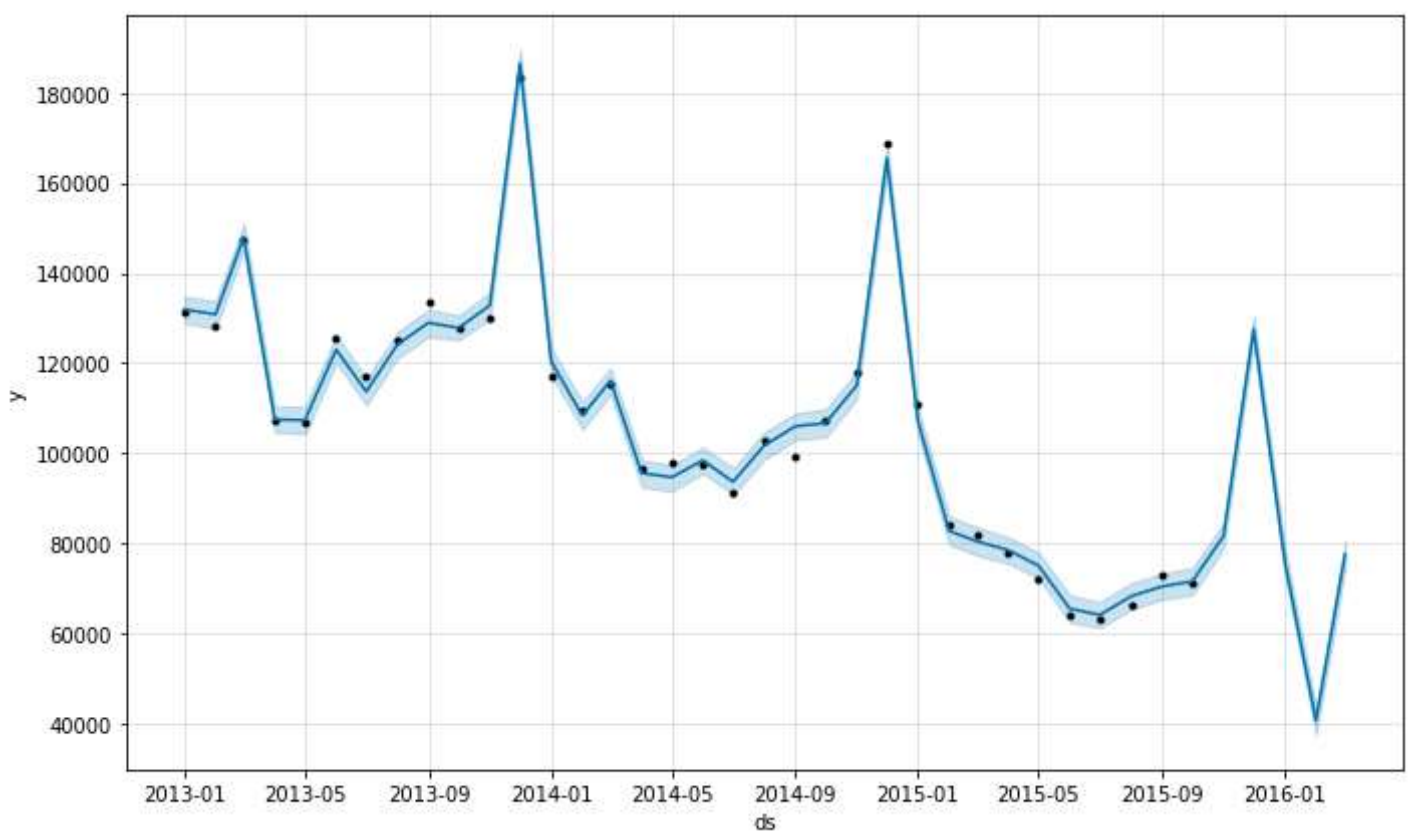
In [28]:
```
# predict for five months in the furure and MS - month start is the frequency
future = model.make_future_dataframe(periods = 5, freq = 'MS')
# now lets make the forecasts
forecast = model.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()
```

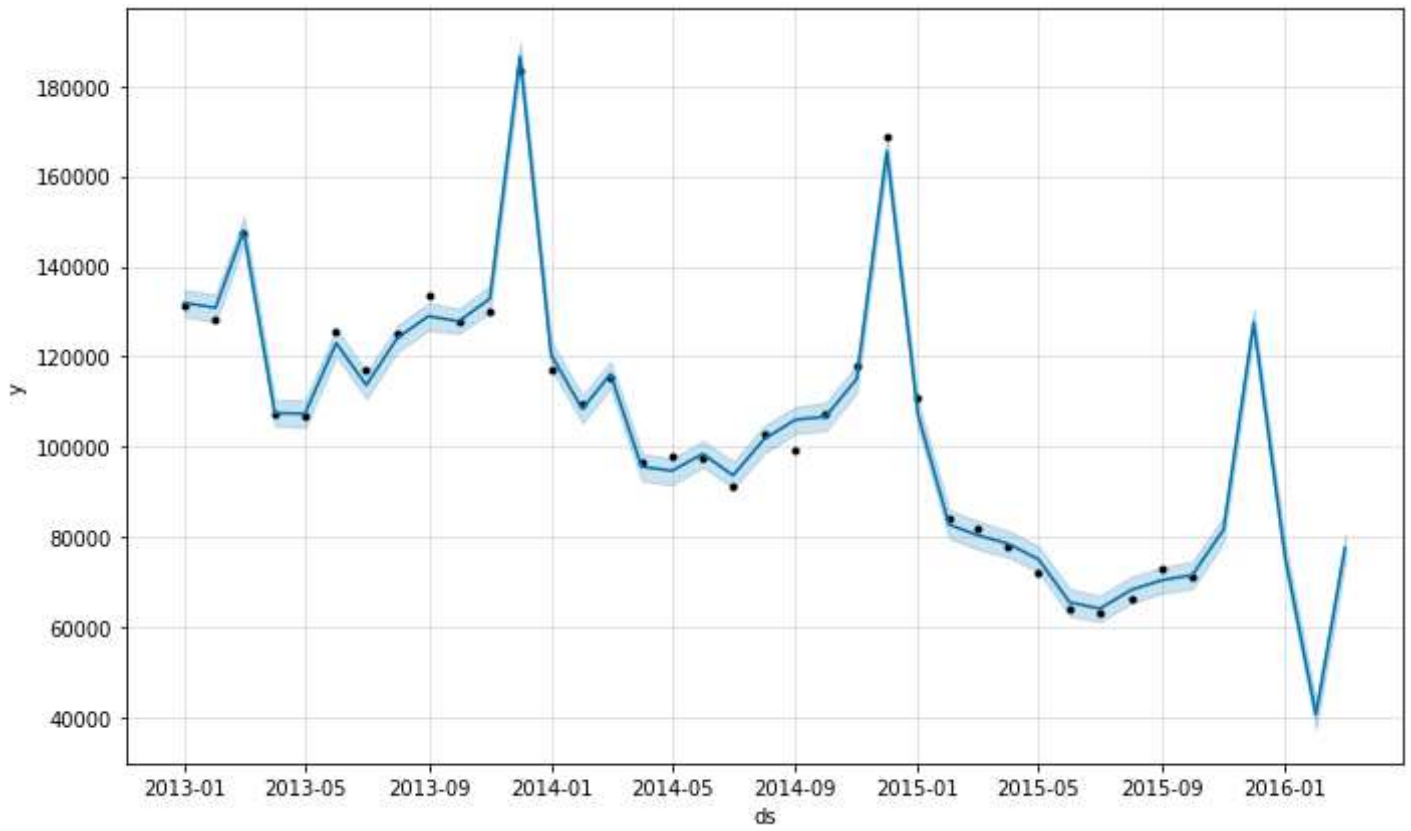
Out[28]:

	ds	yhat	yhat_lower	yhat_upper
34	2015-11-01	81617.472975	78879.464522	84529.678691
35	2015-12-01	127581.716957	124735.908617	130549.252947
36	2016-01-01	76029.414691	72973.723357	79154.966091
37	2016-02-01	40622.118526	37454.347148	43493.266733
38	2016-03-01	77502.906081	74449.018690	80662.314829

In [29]:
```
model.plot(forecast)
```

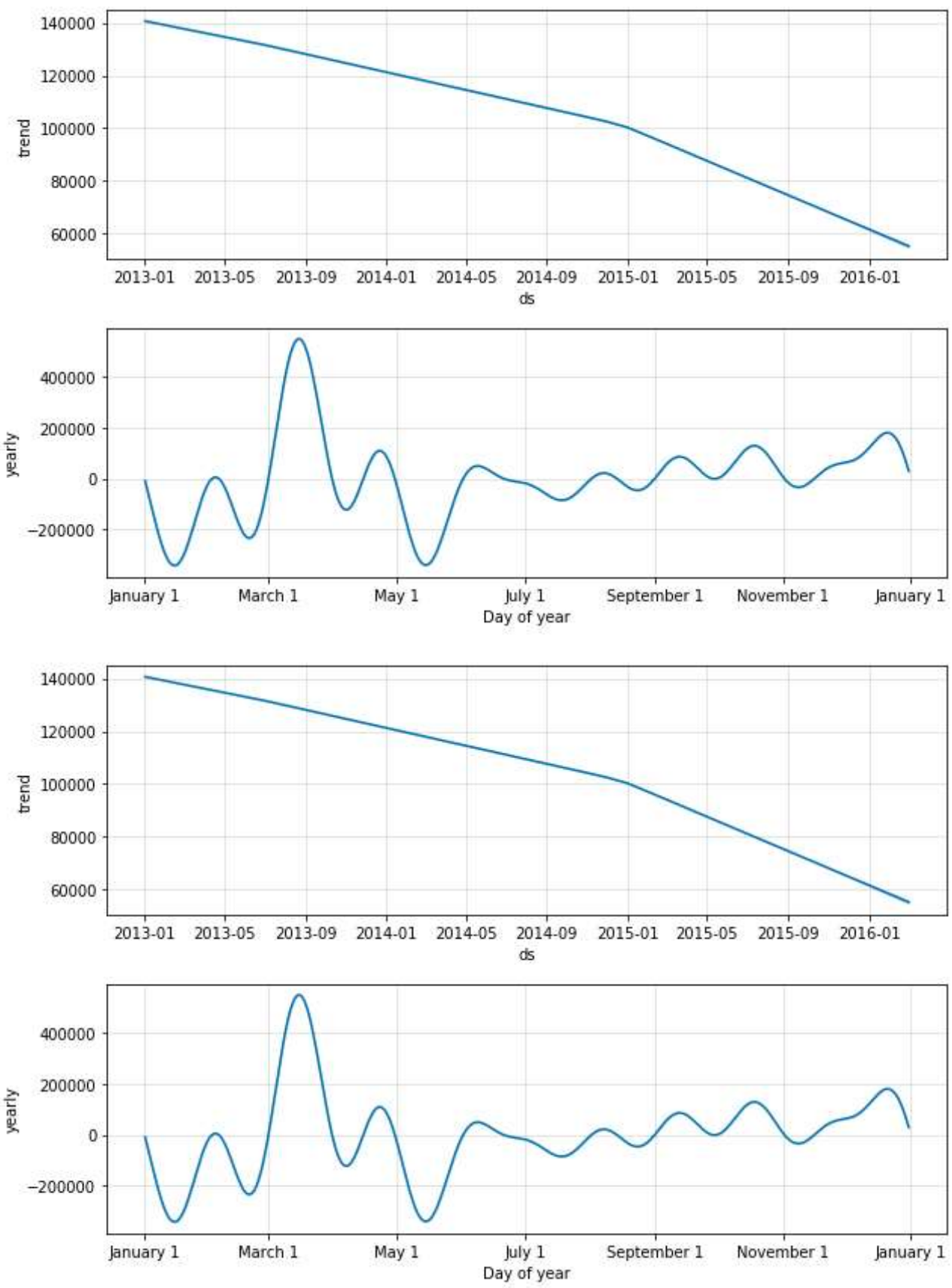
Out[29]:





In [30]:
```
model.plot_components(forecast)
```

Out[30]:



The trend and seasonality from Prophet look similar to the ones that we had earlier using the traditional methods.

Middle out:

Let's predict for the store level

In [34]:
```
monthly_shop_sales=sales.groupby(["date_block_num","shop_id"])["item_cnt_day"].sum()
# get the shops to the columns
monthly_shop_sales=monthly_shop_sales.unstack(level=1)
monthly_shop_sales=monthly_shop_sales.fillna(0)
monthly_shop_sales.index=dates
monthly_shop_sales=monthly_shop_sales.reset_index()
monthly_shop_sales.head()
```

Out[34]:

shop_id	index	0	1	2	3	4	5	6	7	8	...	50	51	52	53	54	55
0	2013-01-01	5578.0	2947.0	1146.0	767.0	2114.0	0.0	3686.0	2495.0	1463.0	...	3406.0	2748.0	1812.0	2055.0	9386.0	0.0
1	2013-02-01	6127.0	3364.0	488.0	798.0	2025.0	877.0	4007.0	2513.0	1156.0	...	3054.0	2284.0	1737.0	1906.0	8075.0	0.0
2	2013-03-01	0.0	0.0	753.0	959.0	2060.0	1355.0	4519.0	2460.0	977.0	...	3610.0	2605.0	2034.0	2365.0	9488.0	0.0
3	2013-04-01	0.0	0.0	583.0	647.0	285.0	1008.0	3168.0	1540.0	-1.0	...	2740.0	1945.0	1446.0	1515.0	6726.0	0.0
4	2013-05-01	0.0	0.0	553.0	710.0	1211.0	1110.0	3022.0	1647.0	0.0	...	2839.0	2243.0	1482.0	1767.0	7006.0	852.0

5 rows × 61 columns

In [35]:
```
start_time=time.time()

# Calculating the base forecasts using prophet
forecastsDict = {}
for node in range(len(monthly_shop_sales)):
    # take the date-column and the col to be forecasted
    nodeToForecast = pd.concat([monthly_shop_sales.iloc[:,0], monthly_shop_sales.iloc[:, node+1]], axis = 1)
#     print(nodeToForecast.head())  # just to check
# rename for prophet compatability
    nodeToForecast = nodeToForecast.rename(columns = {nodeToForecast.columns[0] : 'ds'})
    nodeToForecast = nodeToForecast.rename(columns = {nodeToForecast.columns[1] : 'y'})
    growth = 'linear'
    m = Prophet(growth, yearly_seasonality=True)
    m.fit(nodeToForecast)
    future = m.make_future_dataframe(periods = 1, freq = 'MS')
    forecastsDict[node] = m.predict(future)
```

```
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override 
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
```

```
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
INFO:fbprophet.forecaster:Disabling weekly seasonality. Run prophet with weekly_seasonality=True to override
this.
INFO:fbprophet.forecaster:Disabling daily seasonality. Run prophet with daily_seasonality=True to override th
is.
```

In [36]:

```python
#predictions = np.zeros([len(forecastsDict[0].yhat),1])
nCols = len(list(forecastsDict.keys()))+1
for key in range(0, nCols-1):
    f1 = np.array(forecastsDict[key].yhat)
    f2 = f1[:, np.newaxis]
    if key==0:
        predictions=f2.copy()
       # print(predictions.shape)
    else:
       predictions = np.concatenate((predictions, f2), axis = 1)
```

In [37]:

```python
predictions_unknown=predictions[-1]
predictions_unknown
```

Out[37]:

```
array([ -924.43882842,  -630.78081916,  1119.39115324,   861.79658395,
         991.66923161,  1396.67516198,  2244.39881487,  1828.02079534,
        -259.79364248,  -271.98490203,   574.1868259 ,    23.68915328,
        4444.7867686 ,  -326.457982  ,  1302.22012026,  1709.45700264,
        1024.25030693,   971.47186231,   866.03150203,  1735.97759069,
        -681.39293253,  2131.2297677 ,  1551.66886659,  -677.33325093,
        1699.92851736,  6739.29814657,  1382.62318184,  3614.84692615,
        4196.93296455,   731.69633225,   392.42796896,  8160.61897373,
        -714.56141747,   455.88823494])
```