Introduction

ARIMA models are, in theory, the most general class of models for forecasting a time series, which can be made to be “stationary” by differencing (if necessary), perhaps in conjunction with nonlinear transformations such as logging or deflating (if necessary). A random variable that is a time series is stationary if its statistical properties are all constant over time. A stationary series has no trend, its variations around its mean have a constant amplitude, and it wiggles in a consistent fashion, i.e., its short-term random time patterns always look the same in a statistical sense. The latter condition means that its autocorrelations (correlations with its prior deviations from the mean) remain constant over time, or equivalently, that its power spectrum remains constant over time. A random variable of this form can be viewed (as usual) as a combination of signal and noise, and the signal (if one is apparent) could be a pattern of fast or slow mean reversion, or sinusoidal oscillation, or rapid alternation in sign, and it could also have a seasonal component. An ARIMA model can be viewed as a “filter” that tries to separate the signal from the noise, and the signal is then extrapolated into the future to obtain forecasts.

The Data

The data we will use is annual sunspot data from 1700 – 2008 recording the number of sunspots per year. The file sunspots.csv and can be downloaded from the line below.

Import Packages

In addition to Statsmodels, we will need to import additional packages, including Numpy, Scipy, Pandas, and Matplotlib.

Also, from Statsmodels we will need to import qqplot.

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

from statsmodels.graphics.api import qqplot
```

In [3]:
```
dta= pd.read_csv("sunspots.csv")
```

In [4]:
```
print(sm.datasets.sunspots.NOTE)
```

```
::

    Number of Observations - 309 (Annual 1700 - 2008)
    Number of Variables - 1
    Variable name definitions::

        SUNACTIVITY - Number of sunspots for each year

    The data file contains a 'YEAR' variable that is not returned by load.
```

Preparing the Data

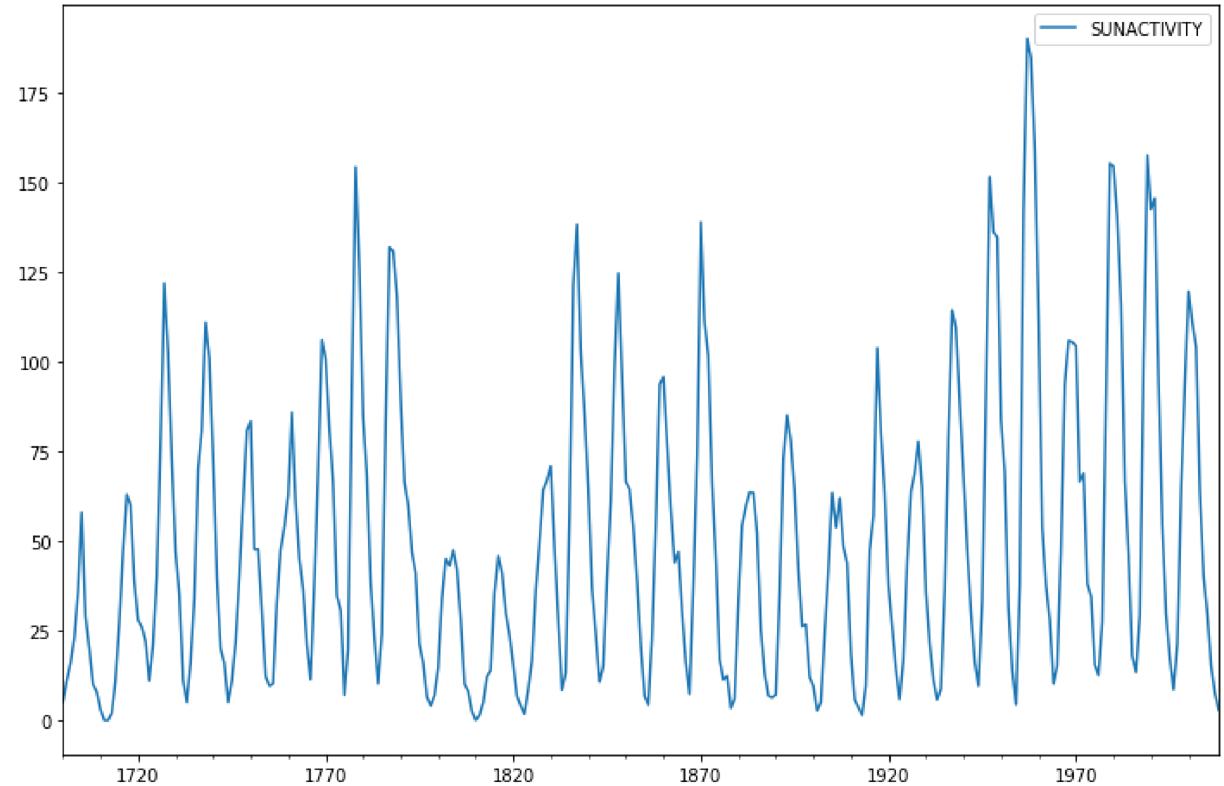
Next we need to do a little dataset preparation. Here, an annual date series must be date-times at the end of the year.

In [5]:
```
dta.index = pd.Index(sm.tsa.datetools.dates_from_range('1700', '2008'))
del dta["YEAR"]
```

Examine the Data

Now we take a look at the data.

In [6]:
```
# show plots in the notebook
%matplotlib inline
dta.plot(figsize=(12,8));
```



Auto-correlations

Before we decide which model to use, we need to look at auto-correlations.

Autocorrelation correlogram.

Seasonal patterns of time series can be examined via correlograms, which display graphically and numerically the autocorrelation function (ACF). Auto-correlation in pandas plotting and statsmodels graphics standardize the data before computing the auto-correlation. These libraries subtract the mean and divide by the standard deviation of the data.

When using standardization, they make an assumption that your data has been generated with a Gaussian law (with a certain mean and standard deviation). This may not be the case in reality.

Correlation is sensitive. Both (matplotlib and pandas plotting) of these functions have their drawbacks. The figure generated by the following code using matplotlib will be identical to figure generated by pandas plotting or statsmodels graphics.

Partial autocorrelations.

Another useful method to examine serial dependencies is to examine the partial autocorrelation function (PACF) – an extension of autocorrelation, where the dependence on the intermediate elements (those within the lag) is removed.

Once we determine the nature of the auto-correlations we use the following rules of thumb.

- Rule 1: If the ACF shows exponential decay, the PACF has a spike at lag 1, and no correlation for other lags, then use one autoregressive (p)parameter
- Rule 2: If the ACF shows a sine-wave shape pattern or a set of exponential decays, the PACF has spikes at lags 1 and 2, and no correlation for other lags, the use two autoregressive (p) parameters
- Rule 3: If the ACF has a spike at lag 1, no correlation for other lags, and the PACF damps out exponentially, then use one moving average (q) parameter.
- Rule 4: If the ACF has spikes at lags 1 and 2, no correlation for other lags, and the PACF has a sine-wave shape pattern or a set of exponential decays, then use two moving average (q) parameter.
- Rule 5: If the ACF shows exponential decay starting at lag 1, and the PACF shows exponential decay starting at lag 1, then use one autoregressive (p) and one moving average (q) parameter.

Removing serial dependency.

Serial dependency for a particular lag can be removed by differencing the series. There are two major reasons for such transformations.

- First, we can identify the hidden nature of seasonal dependencies in the series. Autocorrelations for consecutive lags are interdependent, so removing some of the autocorrelations will change other auto correlations, making other seasonalities more apparent.
- Second, removing serial dependencies will make the series stationary, which is necessary for ARIMA and other techniques.

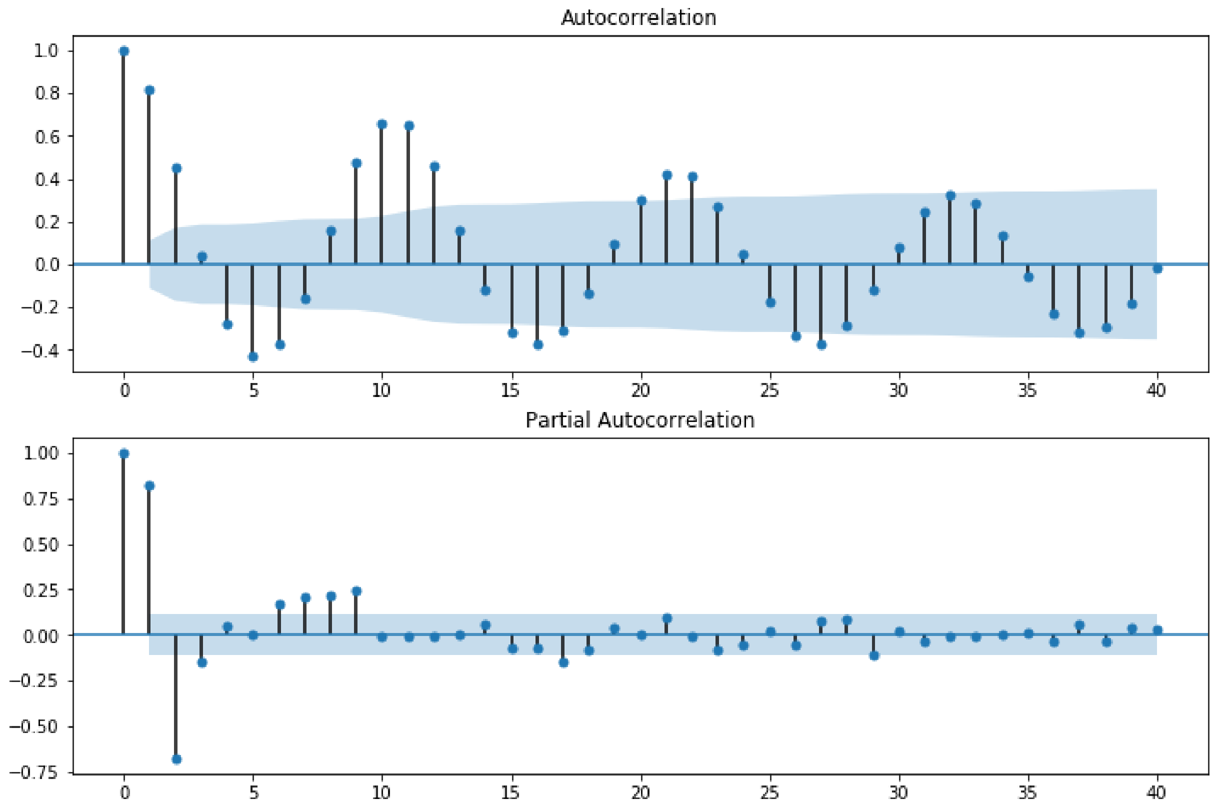
Another popular test for serial correlation is the Durbin-Watson statistic. The DW statistic will lie in the 0-4 range, with a value near two indicating no first-order serial correlation. Positive serial correlation is associated with DW values below 2 and negative serial correlation with DW values above 2.

In [7]:
```
sm.stats.durbin_watson(dta)
```

Out[7]:
```
array([ 0.13952893])
```

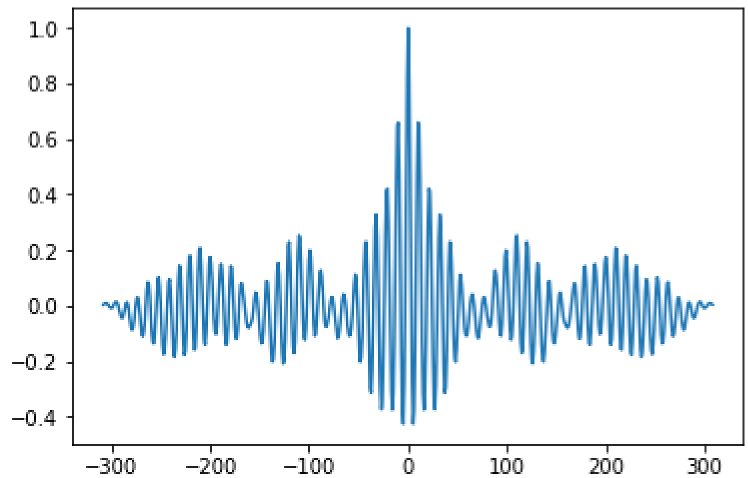
The value of Durbin-Watson statistic is close to 2 if the errors are uncorrelated. In our example, it is 0.1395. That means that there is a strong evidence that the variable open has high autocorrelation.

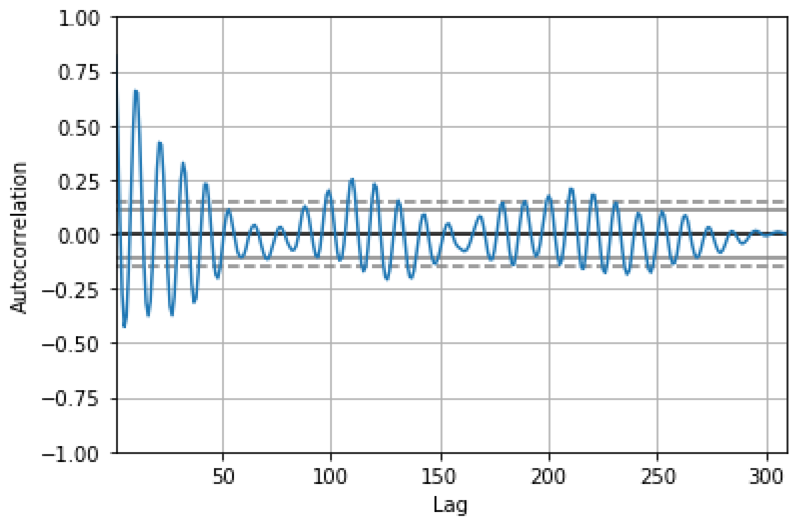
In [8]:
```
# show plots in the notebook
%matplotlib inline
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(dta.values.squeeze(), lags=40, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(dta, lags=40, ax=ax2)
```



The plots also indicate that autocorrelation is present. Another set of plots (shown below) are available using the autocorrelation_plot function from Pandas.

In [11]:
```
from pandas.tools.plotting import autocorrelation_plot
# show plots in the notebook
%matplotlib inline
dta['SUNACTIVITY_2'] = dta['SUNACTIVITY']
dta['SUNACTIVITY_2'] = (dta['SUNACTIVITY_2'] - dta['SUNACTIVITY_2'].mean()) / (dt
plt.acorr(dta['SUNACTIVITY_2'],maxlags = len(dta['SUNACTIVITY_2']) -1, linestyle
plt.show()
autocorrelation_plot(dta['SUNACTIVITY'])
plt.show()
```





For mixed ARMA processes the Autocorrelation function is a mixture of exponentials and damped sine waves after (q-p) lags. The partial autocorrelation function is a mixture of exponentials and dampened sine waves after (p-q) lags.

Times Series Modeling

We will only explore two methods here. An ARMA model is classified as ARMA(p,q), with no differenceing terms. ARMA models can be described by a series of equations. The equations are somewhat simpler if the time series is first reduced to zero-mean by subtracting the sample mean. Therefore, we will work with the mean-adjusted series

```
yt = Yt − Ȳ, t = 1, …N
```

where Yt is the original time series, Ȳ is its sample mean, and yt is the mean-adjusted series. One subset of ARMA models are the so-called autoregressive, or AR models. An AR model expresses a time series as a linear function of its past values. The order of the AR model tells how many lagged past values are included. The simplest AR model is the first-order autoregressive, or AR(1), model

```
yt + a1 yt-1 = et
```

where yt is the mean-adjusted series in year t, yt-1 is the series in the previous year, at is the lag-1 autoregressive coefficient, and et is the noise. The noise also goes by various other names: the error, the random-shock, and the residual. The residuals et are assumed to be random in time (not autocorrelated), and normally distributed. Be rewriting the equation for the AR(1) model as

```
yt = a1 yt-1 + et
```

We see that the AR(1) model has the form of a regression model in which yt is regressed on its previous value. In this form, at is analogous to the negative of the regression coefficient, and et to the regression residuals. The name autoregressive refers to the regression on self (auto).

A nonseasonal ARIMA model is classified as an ARIMA(p,d,q) model, where:

- p is the number of autoregressive terms,
- d is the number of nonseasonal differences needed for stationarity, and
- q is the number of lagged forecast errors in the prediction equation.

The forecasting equation is constructed as follows. First, let y denote the dth difference of Y, which means:

```
If d=0: yt = Yt

If d=1: yt = Yt − Yt-1

If d=2: yt = (Yt − Yt-1) − (Yt-1 − Yt-2) = Yt − 2Yt-1 + Yt-2
```

Note that the second difference of Y (the d=2 case) is not the difference from two periods ago. Rather, it is the first-difference-of-the-first difference, which is the discrete analog of a second derivative, i.e., the local acceleration of the series rather than its local trend.

Modeling the Data

In [13]:
```
arma_mod20 = sm.tsa.ARMA(dta['SUNACTIVITY'], (2,0)).fit()
print(arma_mod20.params)
```

```
const                 49.659374
ar.L1.SUNACTIVITY      1.390656
ar.L2.SUNACTIVITY     -0.688571
dtype: float64
```

We now calculate the Akaike Information Criterion (AIC), Schwarz Bayesian Information Criterion (BIC), and Hannan-Quinn Information Criterion (HQIC). Our goalis to choose a model that minimizes (AIC, BIC, HQIC).

In [15]:
```
print(arma_mod20.aic, arma_mod20.bic, arma_mod20.hqic)
```

```
2622.6363380637513 2637.56970317 2628.60672591
```

Does our model obey the theory? We will use the Durbin-Watson test for autocorrelation. The Durbin-Watson statistic ranges in value from 0 to 4. A value near 2 indicates non-autocorrelation; a value toward 0 indicates positive autocorrelation; a value toward 4 indicates negative autocorrelation.

In [16]:
```
sm.stats.durbin_watson(arma_mod20.resid.values)
```

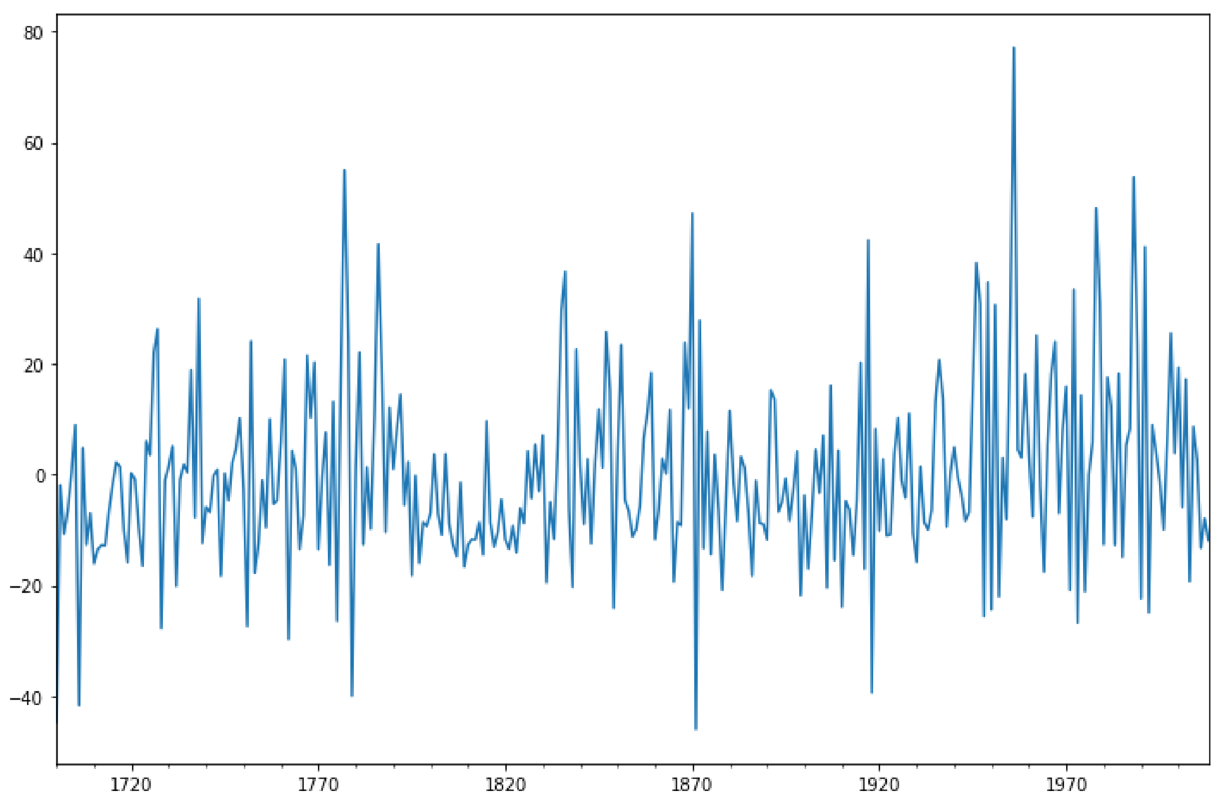
Out[16]:
```
2.1458268282014701
```

The Durbin-Watson test shows no autocorrelation.

Plotting the Data

Next we plot and study the data it represents.

In [17]:
```
# show plots in the notebook
%matplotlib inline
fig = plt.figure(figsize=(12,8))
ax = fig.add_subplot(111)
ax = arma_mod20.resid.plot(ax=ax);
```



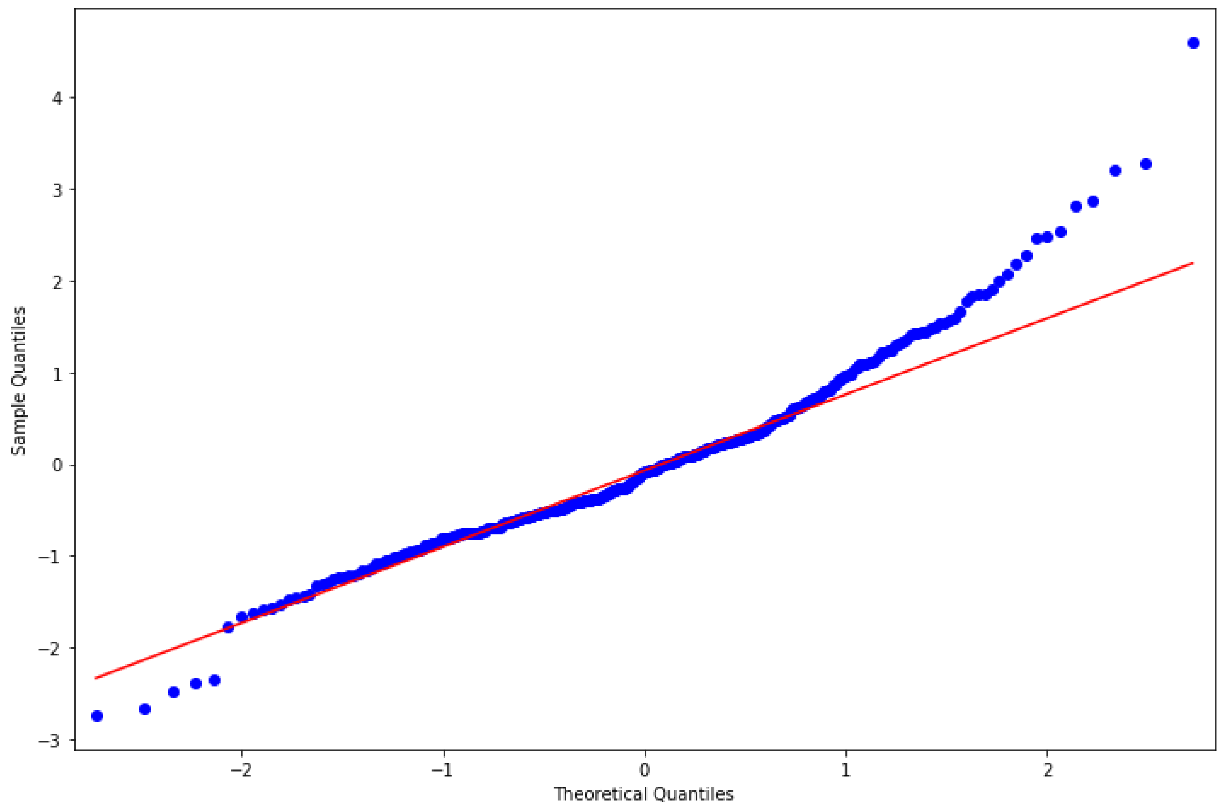
Analyzing the Residuals

In the following steps, we calculate the residuals, tests the null hypothesis that the residuals come from a normal distribution, and construct a qq-plot.

In [21]:
```
resid20 = arma_mod20.resid
stats.normaltest(resid20)
```

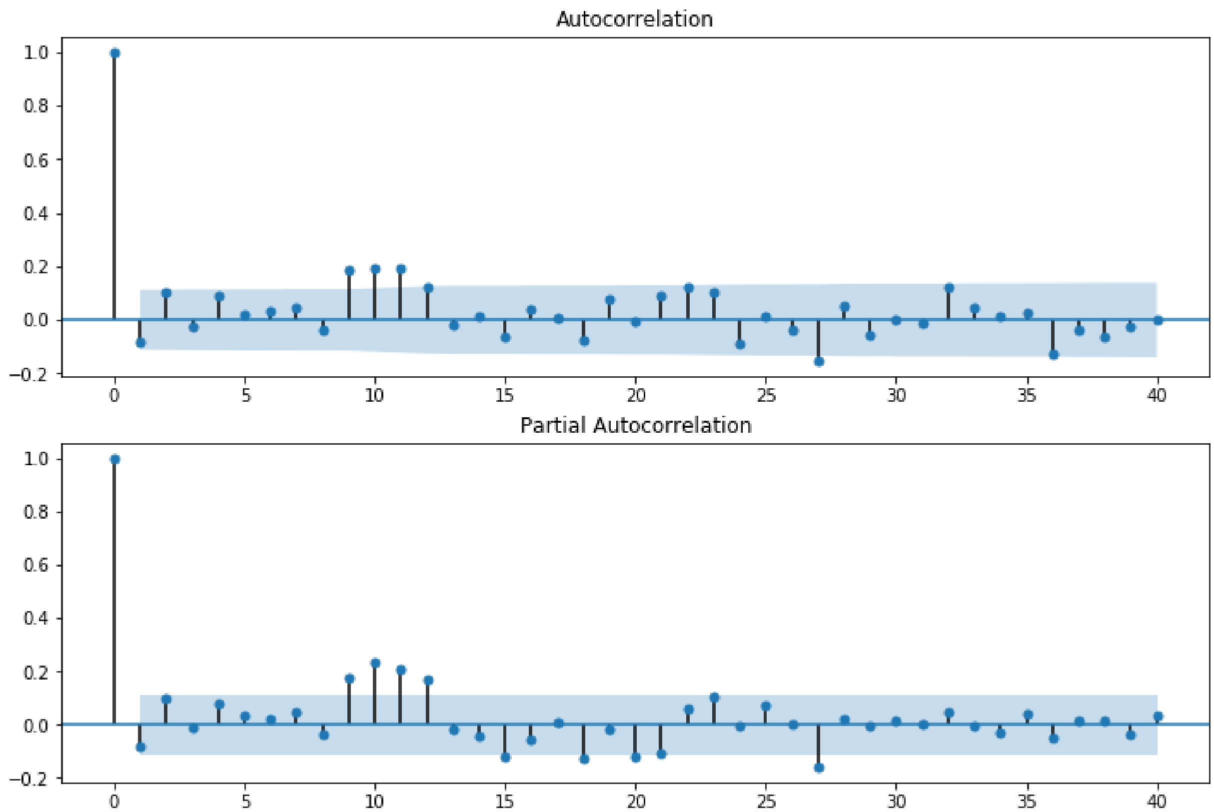
Out[21]:
```
NormaltestResult(statistic=41.736018916111405, pvalue=8.652440949981332e-10)
```

In [22]:
```
# show plots in the notebook
%matplotlib inline
fig = plt.figure(figsize=(12,8))
ax = fig.add_subplot(111)
fig = qqplot(resid20, line='q', ax=ax, fit=True)
```



Model Autocorrelation

In [23]:
```
%matplotlib inline
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(resid20.values.squeeze(), lags=40, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(resid20, lags=40, ax=ax2)
```



Next, we calculate the lag, autocorrelation (AC), Q statistic and Prob>Q. The Ljung–Box Q test (named for Greta M. Ljung and George E. P. Box) is a type of statistical test of whether any of a group of autocorrelations of a time series are different from zero. The null hypothesis is, H0: The data are independently distributed (i.e. the correlations in the population from which the sample is taken are 0, so that any observed correlations in the data result from randomness of the sampling process).

In [25]:
```
r,q,p = sm.tsa.acf(resid20.values.squeeze(), qstat=True)
data = np.c_[range(1,41), r[1:], q, p]
table = pd.DataFrame(data, columns=['lag', "AC", "Q", "Prob(>Q)"])
print(table.set_index('lag'))
```

```
            AC          Q  Prob(>Q)
lag
1.0  -0.085220   2.265960  0.132244
2.0   0.103692   5.631595  0.059857
3.0  -0.027833   5.874879  0.117859
4.0   0.091123   8.491075  0.075158
5.0   0.019010   8.605308  0.125881
6.0   0.031321   8.916433  0.178333
7.0   0.044485   9.546129  0.215785
8.0  -0.034337   9.922560  0.270503
9.0   0.185690  20.967738  0.012794
10.0  0.191608  32.767501  0.000298
11.0  0.190385  44.456250  0.000006
12.0  0.121693  49.247985  0.000002
13.0 -0.016219  49.333387  0.000004
14.0  0.014986  49.406549  0.000008
15.0 -0.063197  50.711997  0.000009
16.0  0.039730  51.229710  0.000015
17.0  0.009577  51.259893  0.000027
18.0 -0.073645  53.050953  0.000026
19.0  0.076469  54.988687  0.000023
20.0 -0.006827  55.004184  0.000041
21.0  0.088818  57.636451  0.000029
22.0  0.120485  62.497164  0.000009
23.0  0.103328  66.084673  0.000005
24.0 -0.085728  68.562789  0.000004
25.0  0.013730  68.626578  0.000006
26.0 -0.036183  69.071149  0.000009
27.0 -0.150156  76.754566  0.000001
28.0  0.049680  77.598636  0.000002
29.0 -0.055467  78.654558  0.000002
30.0  0.003354  78.658432  0.000003
31.0 -0.010905  78.699542  0.000005
32.0  0.120386  83.727466  0.000002
33.0  0.042680  84.361702  0.000002
34.0  0.011107  84.404813  0.000004
35.0  0.024261  84.611254  0.000005
36.0 -0.125046  90.115482  0.000002
37.0 -0.036394  90.583431  0.000002
38.0 -0.060509  91.881754  0.000002
39.0 -0.024440  92.094344  0.000003
40.0  0.000581  92.094465  0.000005
```

Notice that the p-values for the Ljung–Box Q test all are well above .05 for lags 1 through 8, indicating “significance.” This is not a desirable result. However, the p-values for the remaining lags through 40 data values as less than .05. So there is much data not contributing to correlations at high lags.

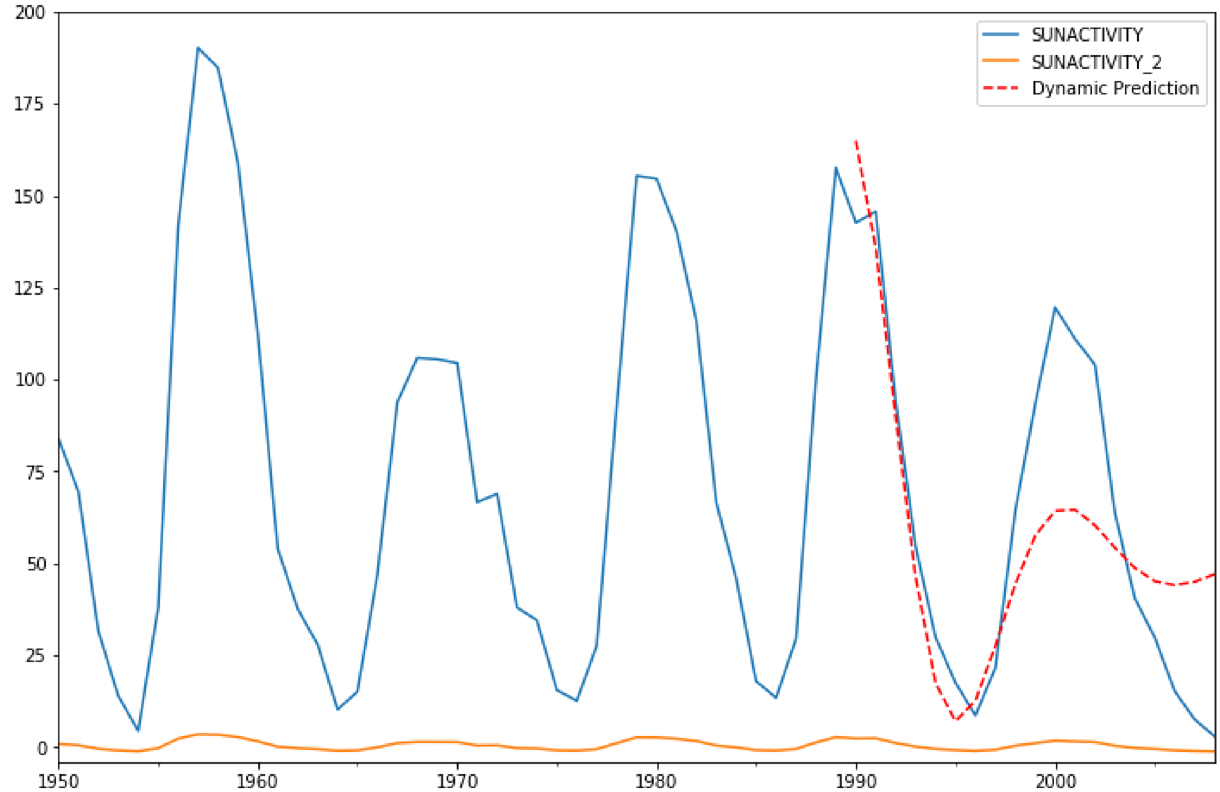
Predictions

Next, we compute the predictions and analyze their fit against actual values.

In [26]:
```
predict_sunspots20 = arma_mod20.predict('1990', '2012', dynamic=True)
print(predict_sunspots20)
```

```
1990-12-31    164.966799
1991-12-31    135.687495
1992-12-31     89.897500
1993-12-31     46.380270
1994-12-31     17.392456
1995-12-31      7.045099
1996-12-31     12.615660
1997-12-31     27.487284
1998-12-31     44.332865
1999-12-31     57.519095
2000-12-31     64.257220
2001-12-31     64.547974
2002-12-31     60.312634
2003-12-31     54.222530
2004-12-31     48.669625
2005-12-31     45.140916
2006-12-31     44.057268
2007-12-31     44.980053
2008-12-31     47.009499
2009-12-31     49.196355
2010-12-31     50.840102
2011-12-31     51.620181
2012-12-31     51.573167
Freq: A-DEC, dtype: float64
```

In [27]:
```
ax = dta.ix['1950':].plot(figsize=(12,8))
ax = predict_sunspots20.plot(ax=ax, style='r--', label='Dynamic Prediction');
ax.legend();
ax.axis((-20.0, 38.0, -4.0, 200.0));
```



The fit looks good up to about 1998 and underfit the data afterwards.

Calculate Forecast Errors

Mean absolute error:

The mean absolute error (MAE) value is computed as the average absolute error value. If this value is 0 (zero), the fit (forecast) is perfect. As compared to the mean squared error value, this measure of fit will “de-emphasize” outliers, that is, unique or rare large error values will affect the MAE less than the MSE value.

Mean Forecast Error (Bias).

The mean forecast error (MFE) is the average error in the observations. A large positive MFE means that the forecast is undershooting the actual observations, and a large negative MFE means the forecast is overshooting the actual observations. A value near zero is ideal.

The MAE is a better indicator of fit than the MFE.

In [28]:
```
def mean_forecast_err(y, yhat):
    return y.sub(yhat).mean()

def mean_absolute_err(y, yhat):
    return np.mean((np.abs(y.sub(yhat).mean()) / yhat)) # or percent error = * 10(
```

In [29]:
```
print("MFE = ", mean_forecast_err(dta.SUNACTIVITY, predict_sunspots20))
print("MAE = ", mean_absolute_err(dta.SUNACTIVITY, predict_sunspots20))
```

```
MFE =  4.73040833622
MAE =  0.134689547232
```

For MFE > 0, models tends to under-forecast. However, as long as the tracking signal is between – 4 and 4, we assume the model is working correctly. The measure of MAE being small would indicate a pretty good fit.

In [ ]: