08-Vector-AutoRegression-VAR

October 19, 2022

For more information, visit us at www.pieriandata.com

1 VAR(p)

1.1 Vector Autoregression

In our previous SARIMAX example, the forecast variable y_t was influenced by the exogenous predictor variable, but not vice versa. That is, the occurrence of a holiday affected restaurant patronage but not the other way around.

However, there are some cases where variables affect each other. Forecasting: Principles and Practice describes a case where changes in personal consumption expenditures C_t were forecast based on changes in personal disposable income I_t. > However, in this case a bi-directional relationship may be more suitable: an increase in I_t will lead to an increase in C_t and vice versa.An example of such a situation occurred in Australia during the Global Financial Crisis of 2008–2009. The Australian government issued stimulus packages that included cash payments in December 2008, just in time for Christmas spending. As a result, retailers reported strong sales and the economy was stimulated. Consequently, incomes increased.

Aside from investigating multivariate time series, vector autoregression is used for * Impulse Response Analysis which involves the response of one variable to a sudden but temporary change in another variable * Forecast Error Variance Decomposition (FEVD) where the proportion of the forecast variance of one variable is attributed to the effect of other variables * Dynamic Vector Autoregressions used for estimating a moving-window regression for the purposes of making forecasts throughout the data sample

1.1.1 Formulation

We've seen that an autoregression AR(p) model is described by the following:

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + \varepsilon_t$$

where c is a constant, ϕ_1 and ϕ_2 are lag coefficients up to order p, and ε_t is white noise.

A K-dimensional VAR model of order p, denoted VAR(p), considers each variable y_K in the system.

For example, The system of equations for a 2-dimensional VAR(1) model is:

$$y_{1,t} = c_1 + \phi_{11,1}y_{1,t-1} + \phi_{12,1}y_{2,t-1} + \varepsilon_{1,t} \qquad y_{2,t} = c_2 + \phi_{21,1}y_{1,t-1} + \phi_{22,1}y_{2,t-1} + \varepsilon_{2,t}$$

where the coefficient $\phi_{ii,l}$ captures the influence of the lth lag of variable y_i on itself, the coefficient $\phi_{ij,l}$ captures the influence of the lth lag of variable y_j on y_i, and $\varepsilon_{1,t}$ and $\varepsilon_{2,t}$ are white noise processes that may be correlated.

Carrying this further, the system of equations for a 2-dimensional VAR(3) model is:

$$y_{1,t} = c_1 + \phi_{11,1}y_{1,t-1} + \phi_{12,1}y_{2,t-1} + \phi_{11,2}y_{1,t-2} + \phi_{12,2}y_{2,t-2} + \phi_{11,3}y_{1,t-3} + \phi_{12,3}y_{2,t-3} + \varepsilon_{1,t}$$
$$y_{2,t} = c_2 + \phi_{21,1}y_{1,t-1} + \phi_{22,1}y_{2,t-1} + \phi_{21,2}y_{1,t-2} + \phi_{22,2}y_{2,t-2} + \phi_{21,3}y_{1,t-3} + \phi_{22,3}y_{2,t-3} + \varepsilon_{2,t}$$

and the system of equations for a 3-dimensional VAR(2) model is:

$$y_{1,t} = c_1 + \phi_{11,1}y_{1,t-1} + \phi_{12,1}y_{2,t-1} + \phi_{13,1}y_{3,t-1} + \phi_{11,2}y_{1,t-2} + \phi_{12,2}y_{2,t-2} + \phi_{13,2}y_{3,t-2} + \varepsilon_{1,t}$$
$$y_{2,t} = c_2 + \phi_{21,1}y_{1,t-1} + \phi_{22,1}y_{2,t-1} + \phi_{23,1}y_{3,t-1} + \phi_{21,2}y_{1,t-2} + \phi_{22,2}y_{2,t-2} + \phi_{23,2}y_{3,t-2} + \varepsilon_{2,t} \qquad y_{3,t} = c_3 + \phi_{31,1}y_{1,t-1} + \phi_{32,1}y_{2,t-1} + \phi_{33,1}y_{3,t-1} + \phi_{31,2}y_{1,t-2} + \phi_{32,2}y_{2,t-2} + \phi_{33,2}y_{3,t-2} + \varepsilon_{3,t}$$

The general steps involved in building a VAR model are: * Examine the data * Visualize the data * Test for stationarity * If necessary, transform the data to make it stationary * Select the appropriate order p * Instantiate the model and fit it to a training set * If necessary, invert the earlier transformation * Evaluate model predictions against a known test set * Forecast the future

Recall that to fit a SARIMAX model we passed one field of data as our endog variable, and another for exog. With VAR, both fields will be passed in as endog.

Related Functions:

vector_ar.var_model.VAR(endog[, exog, …]) Fit VAR(p) process and do lag order selection vector_ar.var_model.VARResults(endog, …[, …]) Estimate VAR(p) process with fixed number of lags vector_ar.dynamic.DynamicVAR(data[, …]) Estimates time-varying vector autoregression (VAR(p)) using equation-by-equation least squares

For Further Reading:

Statsmodels Tutorial: Vector Autoregressions Forecasting: Principles and Practice: Vector Autoregressions Wikipedia: Vector Autoregression

1.1.2 Perform standard imports and load dataset

For this analysis we'll also compare money to spending. We'll look at the M2 Money Stock which is a measure of U.S. personal assets, and U.S. personal spending. Both datasets are in billions of dollars, monthly, seasonally adjusted. They span the 21 years from January 1995 to December 2015 (252 records). Sources: https://fred.stlouisfed.org/series/M2SL https://fred.stlouisfed.org/series/PCE

[1]:
```
import numpy as np
import pandas as pd
%matplotlib inline

# Load specific forecasting tools
from statsmodels.tsa.api import VAR, DynamicVAR
from statsmodels.tsa.stattools import adfuller
```

```
from statsmodels.tools.eval_measures import rmse

# Ignore harmless warnings
import warnings
warnings.filterwarnings("ignore")

# Load datasets
df = pd.read_csv('../Data/M2SLMoneyStock.csv',index_col=0, parse_dates=True)
df.index.freq = 'MS'

sp = pd.read_csv('../Data/PCEPersonalSpending.csv',index_col=0,␣
 ↪parse_dates=True)
sp.index.freq = 'MS'
```

1.1.3 Inspect the data

[2]:
```
df = df.join(sp)
df.head()
```

[2]:

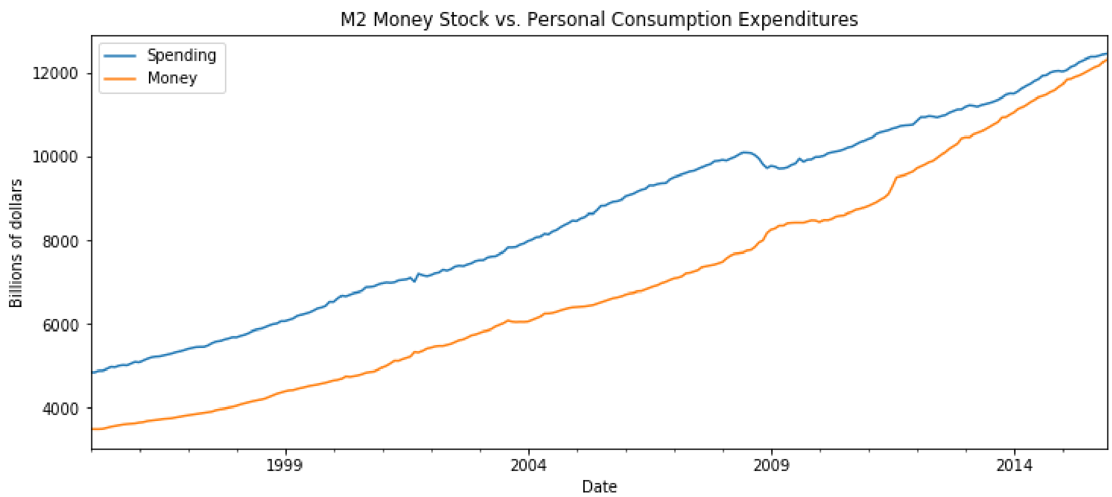
	Money	Spending
Date		
1995-01-01	3492.4	4851.2
1995-02-01	3489.9	4850.8
1995-03-01	3491.1	4885.4
1995-04-01	3499.2	4890.2
1995-05-01	3524.2	4933.1

[3]:
```
df = df.dropna()
df.shape
```

[3]: (252, 2)

1.1.4 Plot the source data

[4]:
```
title = 'M2 Money Stock vs. Personal Consumption Expenditures'
ylabel='Billions of dollars'
xlabel=''

ax = df['Spending'].plot(figsize=(12,5),title=title,legend=True)
ax.autoscale(axis='x',tight=True)
ax.set(xlabel=xlabel, ylabel=ylabel)
df['Money'].plot(legend=True);
```



1.2 Test for stationarity, perform any necessary transformations

[5]:
```python
def adf_test(series,title=''):
    """
    Pass in a time series and an optional title, returns an ADF report
    """
    print(f'Augmented Dickey-Fuller Test: {title}')
    result = adfuller(series.dropna(),autolag='AIC') # .dropna() handles␣
↪differenced data

    labels = ['ADF test statistic','p-value','# lags used','# observations']
    out = pd.Series(result[0:4],index=labels)

    for key,val in result[4].items():
        out[f'critical value ({key})']=val

    print(out.to_string())          # .to_string() removes the line "dtype:␣
↪float64"

    if result[1] <= 0.05:
        print("Strong evidence against the null hypothesis")
        print("Reject the null hypothesis")
        print("Data has no unit root and is stationary")
    else:
        print("Weak evidence against the null hypothesis")
        print("Fail to reject the null hypothesis")
        print("Data has a unit root and is non-stationary")
```

[6]:
```python
adf_test(df['Money'],title='Money')
```

```
Augmented Dickey-Fuller Test: Money
ADF test statistic         4.239022
p-value                    1.000000
# lags used                4.000000
# observations           247.000000
critical value (1%)       -3.457105
critical value (5%)       -2.873314
critical value (10%)      -2.573044
Weak evidence against the null hypothesis
Fail to reject the null hypothesis
Data has a unit root and is non-stationary
```

[7]:
```python
adf_test(df['Spending'], title='Spending')
```

```
Augmented Dickey-Fuller Test: Spending
ADF test statistic         0.149796
p-value                    0.969301
# lags used                3.000000
# observations           248.000000
critical value (1%)       -3.456996
critical value (5%)       -2.873266
critical value (10%)      -2.573019
Weak evidence against the null hypothesis
Fail to reject the null hypothesis
Data has a unit root and is non-stationary
```

Neither variable is stationary, so we'll take a first order difference of the entire DataFrame and re-run the augmented Dickey-Fuller tests. It's advisable to save transformed values in a new DataFrame, as we'll need the original when we later invert the transormations and evaluate the model.

[8]:
```python
df_transformed = df.diff()
```

[9]:
```python
df_transformed = df_transformed.dropna()
adf_test(df_transformed['Money'], title='MoneyFirstDiff')
print()
adf_test(df_transformed['Spending'], title='SpendingFirstDiff')
```

```
Augmented Dickey-Fuller Test: MoneyFirstDiff
ADF test statistic        -2.057404
p-value                    0.261984
# lags used               15.000000
# observations           235.000000
critical value (1%)       -3.458487
critical value (5%)       -2.873919
critical value (10%)      -2.573367
Weak evidence against the null hypothesis
```

```
Fail to reject the null hypothesis
Data has a unit root and is non-stationary

Augmented Dickey-Fuller Test: SpendingFirstDiff
ADF test statistic     -7.226974e+00
p-value                 2.041027e-10
# lags used             2.000000e+00
# observations          2.480000e+02
critical value (1%)    -3.456996e+00
critical value (5%)    -2.873266e+00
critical value (10%)   -2.573019e+00
Strong evidence against the null hypothesis
Reject the null hypothesis
Data has no unit root and is stationary
```

Since Money is not yet stationary, we'll apply second order differencing to both series so they retain the same number of observations

[10]:

```
df_transformed = df_transformed.diff().dropna()
adf_test(df_transformed['Money'], title='MoneySecondDiff')
print()
adf_test(df_transformed['Spending'], title='SpendingSecondDiff')
```

```
Augmented Dickey-Fuller Test: MoneySecondDiff
ADF test statistic     -7.077471e+00
p-value                 4.760675e-10
# lags used             1.400000e+01
# observations          2.350000e+02
critical value (1%)    -3.458487e+00
critical value (5%)    -2.873919e+00
critical value (10%)   -2.573367e+00
Strong evidence against the null hypothesis
Reject the null hypothesis
Data has no unit root and is stationary

Augmented Dickey-Fuller Test: SpendingSecondDiff
ADF test statistic     -8.760145e+00
p-value                 2.687900e-14
# lags used             8.000000e+00
# observations          2.410000e+02
critical value (1%)    -3.457779e+00
critical value (5%)    -2.873609e+00
critical value (10%)   -2.573202e+00
Strong evidence against the null hypothesis
Reject the null hypothesis
Data has no unit root and is stationary
```

[11]:

```
df_transformed.head()
```

[11]:

```
            Money  Spending
Date
1995-03-01    3.7      35.0
1995-04-01    6.9     -29.8
1995-05-01   16.9      38.1
1995-06-01   -0.3       1.5
1995-07-01   -6.2     -51.7
```

[12]:

```
len(df_transformed)
```

[12]:

```
250
```

1.2.1 Train/test split

It will be useful to define a number of observations variable for our test set. For this analysis, let's use 12 months.

[13]:

```
nobs=12
train, test = df_transformed[0:-nobs], df_transformed[-nobs:]
```

[14]:

```
print(train.shape)
print(test.shape)
```

```
(238, 2)
(12, 2)
```

1.3 VAR Model Order Selection

We'll fit a series of models using the first seven p-values, and base our final selection on the model that provides the lowest AIC and BIC scores.

[15]:

```
for i in [1,2,3,4,5,6,7]:
    model = VAR(train)
    results = model.fit(i)
    print('Order =', i)
    print('AIC: ', results.aic)
    print('BIC: ', results.bic)
    print()
```

```
Order = 1
AIC:  14.178610495220896
BIC:  14.266409486135709

Order = 2
AIC:  13.955189367163705
BIC:  14.101961901274958
```

```
Order = 3
AIC:  13.849518291541038
BIC:  14.055621258341116

Order = 4
AIC:  13.827950574458281
BIC:  14.093744506408875

Order = 5
AIC:  13.78730034460964
BIC:  14.113149468980652

Order = 6
AIC:  13.799076756885807
BIC:  14.185349048538066

Order = 7
AIC:  13.797638727913972
BIC:  14.244705963046671
```

[16]:

```
model = VAR(train)
for i in [1,2,3,4,5,6,7]:
    results = model.fit(i)
    print('Order =', i)
    print('AIC: ', results.aic)
    print('BIC: ', results.bic)
    print()
```

```
Order = 1
AIC:  14.178610495220896
BIC:  14.266409486135709

Order = 2
AIC:  13.955189367163705
BIC:  14.101961901274958

Order = 3
AIC:  13.849518291541038
BIC:  14.055621258341116

Order = 4
AIC:  13.827950574458281
BIC:  14.093744506408875

Order = 5
AIC:  13.78730034460964
```

```
BIC:  14.113149468980652

Order = 6
AIC:  13.799076756885807
BIC:  14.185349048538066

Order = 7
AIC:  13.797638727913972
BIC:  14.244705963046671
```

The VAR(5) model seems to return the lowest combined scores. Just to verify that both variables are included in the model we can run .endog_names

[17]:
```
model.endog_names
```

[17]:
```
['Money', 'Spending']
```

1.4 Fit the VAR(5) Model

[18]:
```
results = model.fit(5)
results.summary()
```

[18]:
```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Tue, 02, Apr, 2019
Time:                     18:33:59
--------------------------------------------------------------------
No. of Equations:         2.00000    BIC:                    14.1131
Nobs:                     233.000    HQIC:                   13.9187
Log likelihood:          -2245.45    FPE:                    972321.
AIC:                      13.7873    Det(Omega_mle):         886628.
--------------------------------------------------------------------
Results for equation Money
==============================================================================
                 coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const               0.516683         1.782238            0.290           0.772
L1.Money           -0.646232         0.068177           -9.479           0.000
L1.Spending        -0.107411         0.051388           -2.090           0.037
L2.Money           -0.497482         0.077749           -6.399           0.000
L2.Spending        -0.192202         0.068613           -2.801           0.005
L3.Money           -0.234442         0.081004           -2.894           0.004
L3.Spending        -0.178099         0.074288           -2.397           0.017
L4.Money           -0.295531         0.075294           -3.925           0.000
```

```
L4.Spending        -0.035564       0.069664          -0.511          0.610
L5.Money           -0.162399       0.066700          -2.435          0.015
L5.Spending        -0.058449       0.051357          -1.138          0.255
==============================================================================

Results for equation Spending
==============================================================================
                 coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const               0.203469         2.355446            0.086           0.931
L1.Money            0.188105         0.090104            2.088           0.037
L1.Spending        -0.878970         0.067916          -12.942           0.000
L2.Money            0.053017         0.102755            0.516           0.606
L2.Spending        -0.625313         0.090681           -6.896           0.000
L3.Money           -0.022172         0.107057           -0.207           0.836
L3.Spending        -0.389041         0.098180           -3.963           0.000
L4.Money           -0.170456         0.099510           -1.713           0.087
L4.Spending        -0.245435         0.092069           -2.666           0.008
L5.Money           -0.083165         0.088153           -0.943           0.345
L5.Spending        -0.181699         0.067874           -2.677           0.007
==============================================================================

Correlation matrix of residuals
               Money  Spending
Money       1.000000 -0.267934
Spending   -0.267934  1.000000
```

1.5 Predict the next 12 values

Unlike the VARMAX model we'll use in upcoming sections, the VAR .forecast() function requires that we pass in a lag order number of previous observations as well. Unfortunately this forecast tool doesn't provide a DateTime index - we'll have to do that manually.

[19]:
```python
lag_order = results.k_ar
lag_order
```

[19]:
```
5
```

[20]:
```python
z = results.forecast(y=train.values[-lag_order:], steps=12)
z
```

[20]:
```
array([[-16.99527634,  36.14982003],
       [ -3.17403756, -11.45029844],
       [ -0.377725  ,  -6.68496939],
```

```
       [ -2.60223305,   5.47945777],
       [  4.228557  ,  -2.44336505],
       [  1.55939341,   0.38763902],
       [ -0.99841027,   3.88368011],
       [  0.36451042,  -2.3561014 ],
       [ -1.21062726,  -1.22414652],
       [  0.22587712,   0.786927  ],
       [  1.33893884,   0.18097449],
       [ -0.21858453,   0.21275046]])
```

[21]:
```
test
```

[21]:
```
            Money  Spending
Date
2015-01-01  -15.5     -26.6
2015-02-01   56.1      52.4
2015-03-01 -102.8      39.5
2015-04-01   30.9     -40.4
2015-05-01  -15.8      38.8
2015-06-01   14.0     -34.1
2015-07-01    6.7       6.9
2015-08-01   -0.7      -8.5
2015-09-01    5.5     -39.8
2015-10-01  -23.1      24.5
2015-11-01   55.8      10.7
2015-12-01  -31.2     -15.0
```

[22]:
```
idx = pd.date_range('1/1/2015', periods=12, freq='MS')
df_forecast = pd.DataFrame(z, index=idx, columns=['Money2d','Spending2d'])
df_forecast
```

[22]:
```
              Money2d  Spending2d
2015-01-01 -16.995276   36.149820
2015-02-01  -3.174038  -11.450298
2015-03-01  -0.377725   -6.684969
2015-04-01  -2.602233    5.479458
2015-05-01   4.228557   -2.443365
2015-06-01   1.559393    0.387639
2015-07-01  -0.998410    3.883680
2015-08-01   0.364510   -2.356101
2015-09-01  -1.210627   -1.224147
2015-10-01   0.225877    0.786927
2015-11-01   1.338939    0.180974
2015-12-01  -0.218585    0.212750
```

1.6 Invert the Transformation

Remember that the forecasted values represent second-order differences. To compare them to the original data we have to roll back each difference. To roll back a first-order difference we take the most recent value on the training side of the original series, and add it to a cumulative sum of forecasted values. When working with second-order differences we first must perform this operation on the most recent first-order difference.

Here we'll use the nobs variable we defined during the train/test/split step.

[23]:
```
# Add the most recent first difference from the training side of the original␣
↪dataset to the forecast cumulative sum
df_forecast['Money1d'] = (df['Money'].iloc[-nobs-1]-df['Money'].iloc[-nobs-2])␣
↪+ df_forecast['Money2d'].cumsum()

# Now build the forecast values from the first difference set
df_forecast['MoneyForecast'] = df['Money'].iloc[-nobs-1] +␣
↪df_forecast['Money1d'].cumsum()
```

[24]:
```
# Add the most recent first difference from the training side of the original␣
↪dataset to the forecast cumulative sum
df_forecast['Spending1d'] = (df['Spending'].iloc[-nobs-1]-df['Spending'].
↪iloc[-nobs-2]) + df_forecast['Spending2d'].cumsum()

# Now build the forecast values from the first difference set
df_forecast['SpendingForecast'] = df['Spending'].iloc[-nobs-1] +␣
↪df_forecast['Spending1d'].cumsum()
```

[25]:
```
df_forecast
```

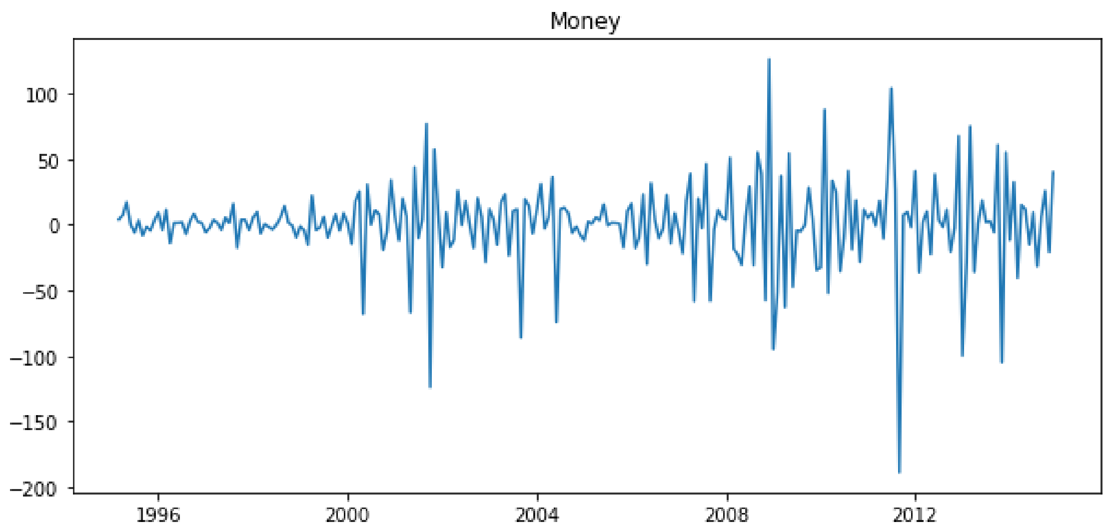
[25]:
```
              Money2d  Spending2d    Money1d  MoneyForecast  Spending1d  \
2015-01-01 -16.995276   36.149820  61.604724   11731.704724   46.749820
2015-02-01  -3.174038  -11.450298  58.430686   11790.135410   35.299522
2015-03-01  -0.377725   -6.684969  58.052961   11848.188371   28.614552
2015-04-01  -2.602233    5.479458  55.450728   11903.639099   34.094010
2015-05-01   4.228557   -2.443365  59.679285   11963.318384   31.650645
2015-06-01   1.559393    0.387639  61.238678   12024.557062   32.038284
2015-07-01  -0.998410    3.883680  60.240268   12084.797331   35.921964
2015-08-01   0.364510   -2.356101  60.604779   12145.402109   33.565863
2015-09-01  -1.210627   -1.224147  59.394151   12204.796261   32.341716
2015-10-01   0.225877    0.786927  59.620028   12264.416289   33.128643
2015-11-01   1.338939    0.180974  60.958967   12325.375256   33.309618
2015-12-01  -0.218585    0.212750  60.740383   12386.115639   33.522368

            SpendingForecast
2015-01-01      12108.749820
2015-02-01      12144.049342
2015-03-01      12172.663894
```

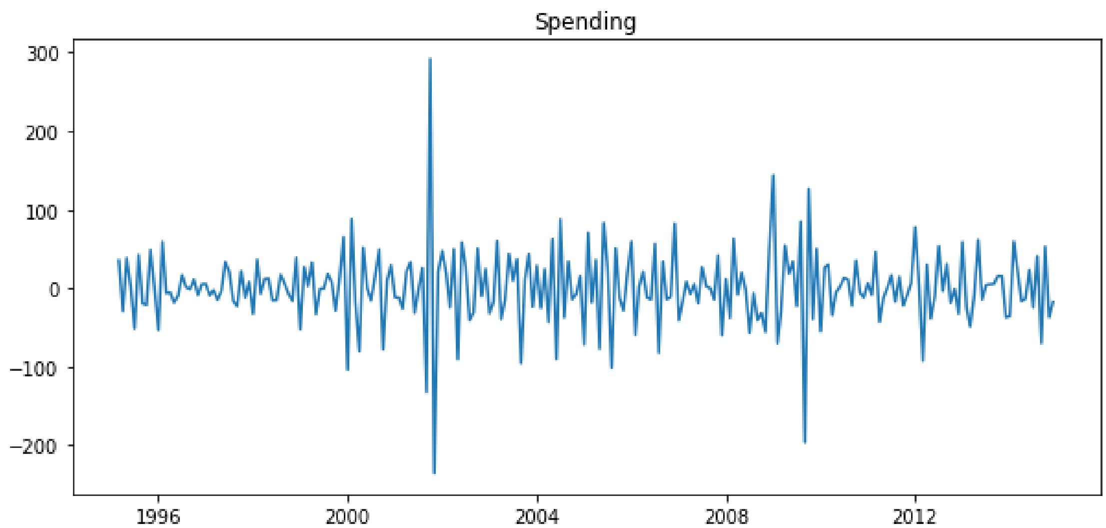
```
2015-04-01      12206.757904
2015-05-01      12238.408549
2015-06-01      12270.446833
2015-07-01      12306.368797
2015-08-01      12339.934659
2015-09-01      12372.276375
2015-10-01      12405.405019
2015-11-01      12438.714636
2015-12-01      12472.237004
```

1.7 Plot the results

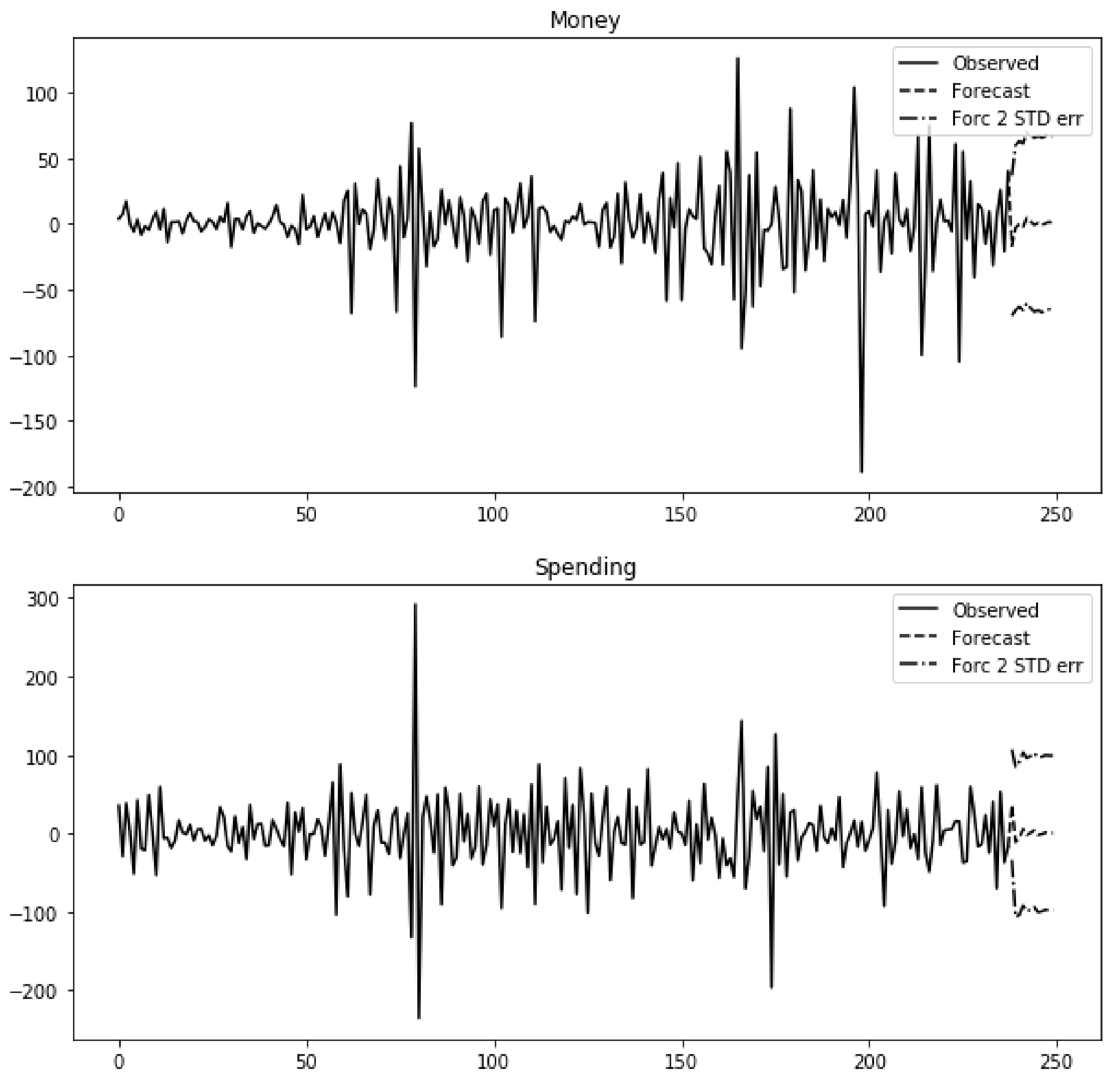
The VARResults object offers a couple of quick plotting tools:

[26]:
```
results.plot();
```



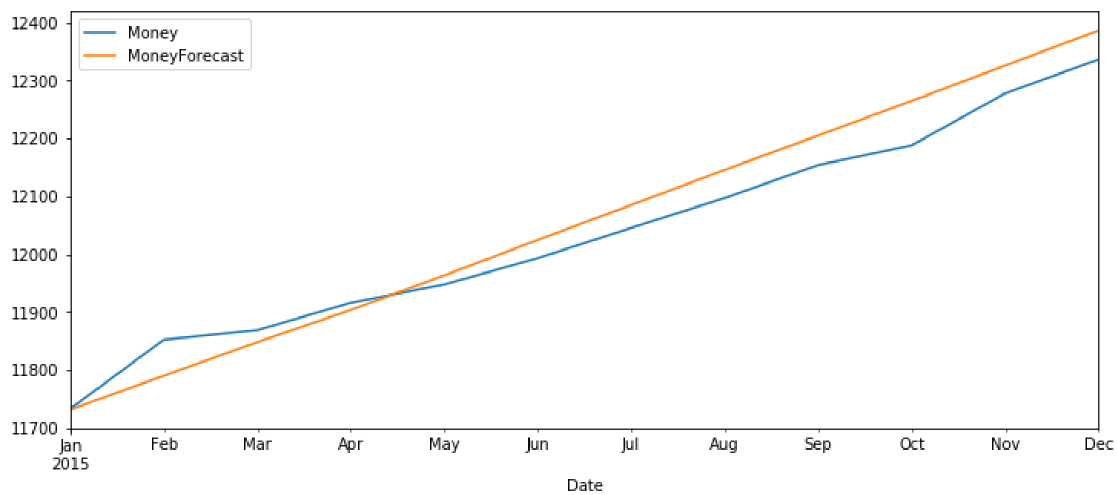


[27]:
```
results.plot_forecast(12);
```

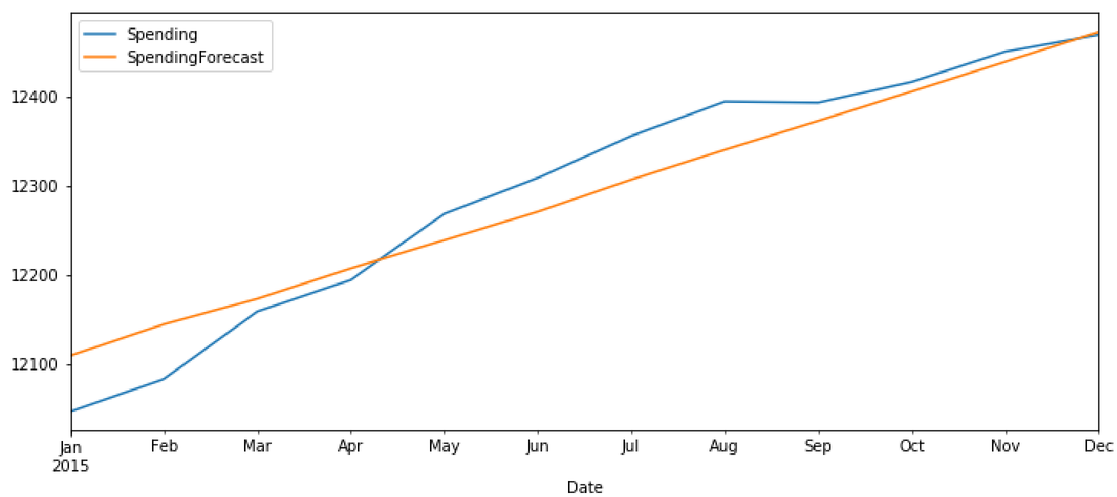


But for our investigation we want to plot predicted values against our test set.

[28]:
```
df['Money'][-nobs:].plot(figsize=(12,5),legend=True).
 ↪autoscale(axis='x',tight=True)
df_forecast['MoneyForecast'].plot(legend=True);
```



[29]:
```
df['Spending'][-nobs:].plot(figsize=(12,5),legend=True).
 ↪autoscale(axis='x',tight=True)
df_forecast['SpendingForecast'].plot(legend=True);
```



1.7.1 Evaluate the model

$$RMSE = \sqrt{\frac{1}{L}\sum_{l=1}^{L}(y_{T+l} - \hat{y}_{T+l})^2} \text{ where } T \text{ is the last observation period and } l \text{ is the lag.}$$

[30]:
```
RMSE1 = rmse(df['Money'][-nobs:], df_forecast['MoneyForecast'])
print(f'Money VAR(5) RMSE: {RMSE1:.3f}')
```

```
Money VAR(5) RMSE: 43.710
```

[31]:
```python
RMSE2 = rmse(df['Spending'][-nobs:], df_forecast['SpendingForecast'])
print(f'Spending VAR(5) RMSE: {RMSE2:.3f}')
```

```
Spending VAR(5) RMSE: 37.001
```

1.8 Let's compare these results to individual AR(5) models

[33]:
```python
from statsmodels.tsa.ar_model import AR,ARResults
```

1.8.1 Money

[34]:
```python
modelM = AR(train['Money'])
AR5fit1 = modelM.fit(maxlag=5,method='mle')
print(f'Lag: {AR5fit1.k_ar}')
print(f'Coefficients:\n{AR5fit1.params}')
```

```
Lag: 5
Coefficients:
const       0.585190
L1.Money   -0.605217
L2.Money   -0.465398
L3.Money   -0.228645
L4.Money   -0.311355
L5.Money   -0.127613
dtype: float64
```

[35]:
```python
start=len(train)
end=len(train)+len(test)-1
z1 = pd.DataFrame(AR5fit1.predict(start=start, end=end,␣
 ↪dynamic=False),columns=['Money'])
```

[36]:
```python
z1
```

[36]:

	Money
2015-01-01	-16.911079
2015-02-01	-11.347188
2015-03-01	9.669315
2015-04-01	-5.699596
2015-05-01	2.353702
2015-06-01	5.293505
2015-07-01	-3.973286
2015-08-01	0.528805
2015-09-01	0.898481

```
2015-10-01   -1.244739
2015-11-01    1.361047
2015-12-01    0.477725
```

1.8.2 Invert the Transformation, Evaluate the Forecast

[37]:
```
# Add the most recent first difference from the training set to the forecast␣
↪cumulative sum
z1['Money1d'] = (df['Money'].iloc[-nobs-1]-df['Money'].iloc[-nobs-2]) +␣
↪z1['Money'].cumsum()

# Now build the forecast values from the first difference set
z1['MoneyForecast'] = df['Money'].iloc[-nobs-1] + z1['Money1d'].cumsum()
```

[38]:
```
z1
```

[38]:

	Money	Money1d	MoneyForecast
2015-01-01	-16.911079	61.688921	11731.788921
2015-02-01	-11.347188	50.341732	11782.130653
2015-03-01	9.669315	60.011047	11842.141700
2015-04-01	-5.699596	54.311452	11896.453152
2015-05-01	2.353702	56.665153	11953.118305
2015-06-01	5.293505	61.958658	12015.076963
2015-07-01	-3.973286	57.985372	12073.062335
2015-08-01	0.528805	58.514177	12131.576512
2015-09-01	0.898481	59.412658	12190.989171
2015-10-01	-1.244739	58.167919	12249.157090
2015-11-01	1.361047	59.528966	12308.686056
2015-12-01	0.477725	60.006691	12368.692747

[39]:
```
RMSE3 = rmse(df['Money'][-nobs:], z1['MoneyForecast'])

print(f'Money VAR(5) RMSE: {RMSE1:.3f}')
print(f'Money  AR(5) RMSE: {RMSE3:.3f}')
```

```
Money VAR(5) RMSE: 43.710
Money  AR(5) RMSE: 36.222
```

1.9 Personal Spending

[40]:
```
modelS = AR(train['Spending'])
AR5fit2 = modelS.fit(maxlag=5,method='mle')
print(f'Lag: {AR5fit2.k_ar}')
print(f'Coefficients:\n{AR5fit2.params}')
```

```
Lag: 5
Coefficients:
const          0.221210
L1.Spending   -0.913123
L2.Spending   -0.677036
L3.Spending   -0.450797
L4.Spending   -0.273218
L5.Spending   -0.159475
dtype: float64
```

[41]:
```
z2 = pd.DataFrame(AR5fit2.predict(start=start, end=end,␣
 ↪dynamic=False),columns=['Spending'])
z2
```

[41]:

	Spending
2015-01-01	30.883394
2015-02-01	-2.227348
2015-03-01	-8.838613
2015-04-01	6.673539
2015-05-01	-4.483675
2015-06-01	-0.535010
2015-07-01	3.507013
2015-08-01	-1.011475
2015-09-01	-0.827619
2015-10-01	0.941987
2015-11-01	-0.495503
2015-12-01	0.126068

1.9.1 Invert the Transformation, Evaluate the Forecast

[42]:
```
# Add the most recent first difference from the training set to the forecast␣
 ↪cumulative sum
z2['Spending1d'] = (df['Spending'].iloc[-nobs-1]-df['Spending'].iloc[-nobs-2])␣
 ↪+ z2['Spending'].cumsum()

# Now build the forecast values from the first difference set
z2['SpendingForecast'] = df['Spending'].iloc[-nobs-1] + z2['Spending1d'].
 ↪cumsum()
```

[43]:
```
z2
```

[43]:

	Spending	Spending1d	SpendingForecast
2015-01-01	30.883394	41.483394	12103.483394
2015-02-01	-2.227348	39.256045	12142.739439
2015-03-01	-8.838613	30.417433	12173.156872
2015-04-01	6.673539	37.090972	12210.247844

```
2015-05-01  -4.483675   32.607297      12242.855140
2015-06-01  -0.535010   32.072286      12274.927426
2015-07-01   3.507013   35.579299      12310.506726
2015-08-01  -1.011475   34.567824      12345.074550
2015-09-01  -0.827619   33.740206      12378.814756
2015-10-01   0.941987   34.682193      12413.496948
2015-11-01  -0.495503   34.186690      12447.683639
2015-12-01   0.126068   34.312758      12481.996397
```

[44]:
```python
RMSE4 = rmse(df['Spending'][-nobs:], z2['SpendingForecast'])

print(f'Spending VAR(5) RMSE: {RMSE2:.3f}')
print(f'Spending  AR(5) RMSE: {RMSE4:.3f}')
```

```
Spending VAR(5) RMSE: 37.001
Spending  AR(5) RMSE: 34.121
```

CONCLUSION: It looks like the VAR(5) model did not do better than the individual AR(5) models. That's ok - we know more than we did before. In the next section we'll look at VARMA and see if the addition of a q parameter helps. Great work!