APPLIED TIMESERIES ANALYSIS PYTHON实现

18级赵呈亮 201800820179

说明：1. 文档图片使用网络URL嵌入

第二章：时间序列转换

分布转换**

时间序列在表现出这些分布特性之前，通常需要进行某种形式的变换， 通常取底 数为 e来进行变换。

为了获得近似正态性，BC变换为一类包含对数的特殊情况的幂变换:

$$f^{(BC)}(x_t,\lambda)=\begin{cases}(x_t^{\lambda}-1)/\lambda & \lambda\neq 0\\ log(x_t) & \lambda=0\end{cases}$$

反双曲正弦(IHS)变换:

$$f^{IHS}(x_t,\lambda)=\frac{sinh^{-1}(\lambda x_t)}{\lambda}=log\frac{\lambda x_t+(\lambda^2x_t^2+1)^{1/2}}{\lambda}\quad \lambda>0$$

转换参数可以估计最大似然(ML)的方法。假设对于一般变换

$$f(x_t,\lambda)=\mu_t+a_t$$

广义幂(GP)变换

$$f^{GP}(x_t,\lambda)=\begin{cases}((x_t+1)^\lambda-1)/\lambda & x_t\geq 0,\lambda\neq 0\\ log(x_t+1) & x_t\geq 0,\lambda=0\\ -((-x_t+1)^{2-\lambda}-1)/2-\lambda & x_t<0,\lambda\neq 2\\ -log(-x_t+1) & x_t<0,\lambda\neq 2\end{cases}$$

其中假设ut是独立的，是正态分布的,并且E（u）=0，方差不变。然后通过最大化集中对数似然函数得到ML估计量:

$$\iota(\lambda)=C_f-(\frac{T}{2})\sum_{t=1}^{T}log\hat{a}^2+D_f(x_t,\lambda)$$

其中

$$\hat{a}_t=f(x_t,\lambda)-\hat{u}_t$$

是模型的ML估计的残差，Cf为常数Df计算公式为

$$D_f(x_t,\lambda)=(\lambda-1)\sum_{t=1}^{T}log|x_t|$$

$$=(\lambda-1)\sum_{t=1}^{T}sgn(x_t)log(|x_t|+1)$$

$$=-\frac{1}{2}\sum_{t=1}^{T}log(1+\lambda^2x_t^2)$$

平稳性诱导转换

一个简单的平稳性变换是取一个级数的连续差分，定义x的一阶差分为

$$\nabla x_t=x_t-x_{t-1}$$

在某些情况下，一阶差分可能不足以引起平稳性，可能需要进一步的差分。

二阶差分定义为一阶差分的一阶差分，即

$$\nabla\nabla x_t=\nabla^2x_t$$

为了给出二阶差分的显式表达式，方便引入滞后值B，定义为

$$B^j x_t \equiv x_{t-j}$$

从而

$$\nabla x_t = x_t - x_{t-1} = x_t - Bx_t = (1-B)x_t$$

进而

$$\nabla^2 x_t = (1-B)^2 x_t = (1-2B+B^2)x_t = x_t - 2x_{t-1} + x_{t-2}$$

在一个变量的变化率与其对数之间有一种有用的关系，即:

$$\frac{x_t - x_{t-1}}{x_{t-1}} = \frac{x_t}{x_{t-1}} \approx log\frac{x_t}{x_{t-1}} = logx_t - logx_{t-1} = \nabla logx_t$$

这个近似是由log(1 + y) 约等于y得到的。因此，如果

$$y_t = (x_t - x_{t-1})/x_{t-1}$$

很小，通货膨胀率可以通过对数的变化来近似，这通常是一种更方便的变换。

分解时间序列和平滑转换

时间序列一般的形式为 数据= 拟合+剩余 ”，更有可能导致平滑的序列，所以最好将其看作“数据=平滑+崎岖”，这是Tukey(1977)借来的术语。Tukey自己喜欢运行或移动中值来做到这一点，但是移动平均(MAs)已经成为到目前为止最流行的平滑时间序列的方法

最简单的MA用它自己、它的前代和后代的平均值代替x，也就是用

$$MA(3)\frac{1}{3}(x_{t-1} + x_t + x_{t+1})$$

####

```
def BC_transform(X,l):
    X_tran=np.array([])
    if l==0:
        X_tran=np.log(X)
    else:
        for t in range(len(X)):
            X_tran.append((X[t]**l)/l)
    return X_tran
```

第三章：平稳时间序列的ARMA模型

4、ARMA模型三种基本形式：自回归模型（AR：Auto-regressive），移动平均模型（MA：Moving-Average）和混合模型（ARMA：Auto-regressive Moving-Average）。

（1）自回归模型AR(p)：如果时间序列y**t满足

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \epsilon_t$$

其中εt是独立同分布的随机变量序列，且满足：

$$E(\epsilon_t) = 0, Var(\epsilon_t) = \sigma_\epsilon^2 > 0$$

则称时间序列y**t服从p阶自回归模型。或者记为$\varphi(B) * y^t = y^t - k$。

平稳条件：滞后算子多项式$phi(B) = 1 - \phi_1 B + \ldots + \phi_p B^p$的根均在单位圆外，即φ(B) = 0的根大于1。

（2）移动平均模型MA(q)：如果时间序列y**t满足：

$$y_t = \epsilon_t - \theta_1 \epsilon_{t-1} - \ldots - \theta_q \epsilon_{t-q}$$

则称时间序列 服从q阶移动平均模型。或者记为y**t = θ(B)ε1。

平稳条件：任何条件下都平稳。

（3）ARMA(p,q)模型：如果时间序列y**t满足

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \epsilon_t - \theta_1 \epsilon_{t-1} - \ldots - \theta_q \epsilon_{t-q}$$

则称时间序列y**t服从(p,q)阶自回归移动平均模型。或者记为φ(B)y**t = θ(B)εt。

特殊情况：q=0,模型即为AR(p)，p=0, 模型即为MA(q)。

获取到完整的数据之后，我们要做的就是先画时序图，检查是否有明显上升或者下降趋势或者有明显周期性的数据是非平稳的。

当时序图不能显然的得出结论的时候就需要通过自相关图和偏自相关图进行判断。如果数据的统计学性质比较好，通过自相关和偏自相关图不但能判断平稳还能直接得出模型的阶数也就是ARMA(P,Q)中的PQ值，进而确定了模型。

在一般的情况下，AIC可以表示为

$$AIC = 2k - 2ln(L)$$

k是参数的数量，L是似然函数。假设条件是模型的误差服从独立正态分布。

自相关系数(ACF)计算公式可以总结为

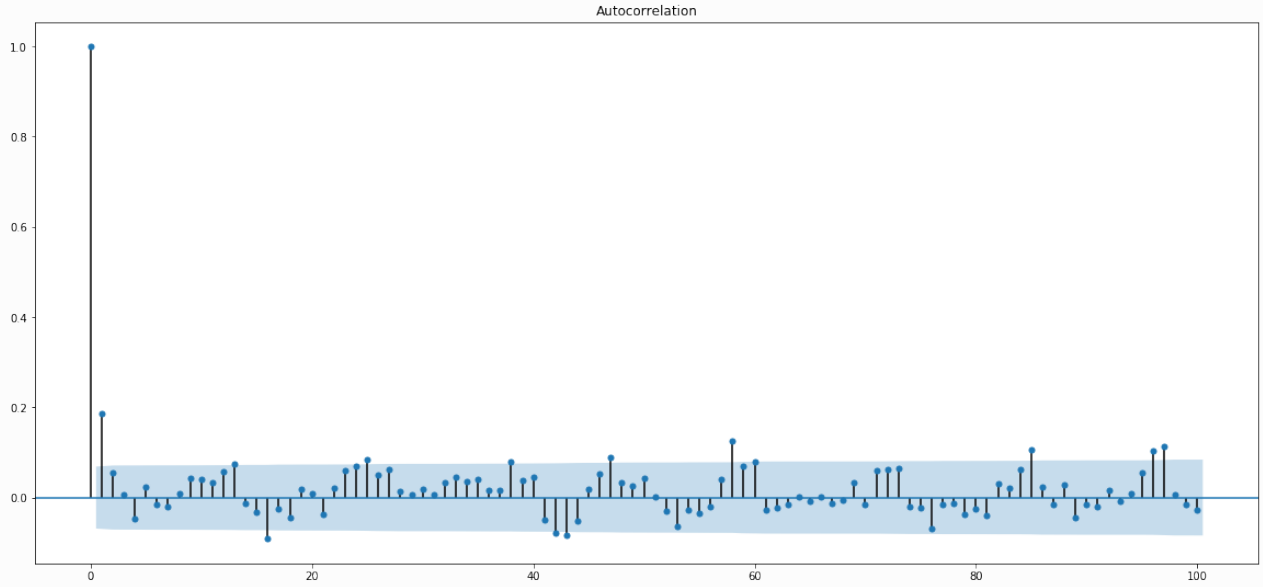
$$acf(k) = \gamma_k = \frac{c_k}{c_0} = \frac{N}{N-k} * \frac{\sum_{t=k+1}^{N}(x_t - \mu)(x_{t-k} - \mu)}{\sum_{t=1}^{N}(x_t - \mu)(x_t - \mu)}$$

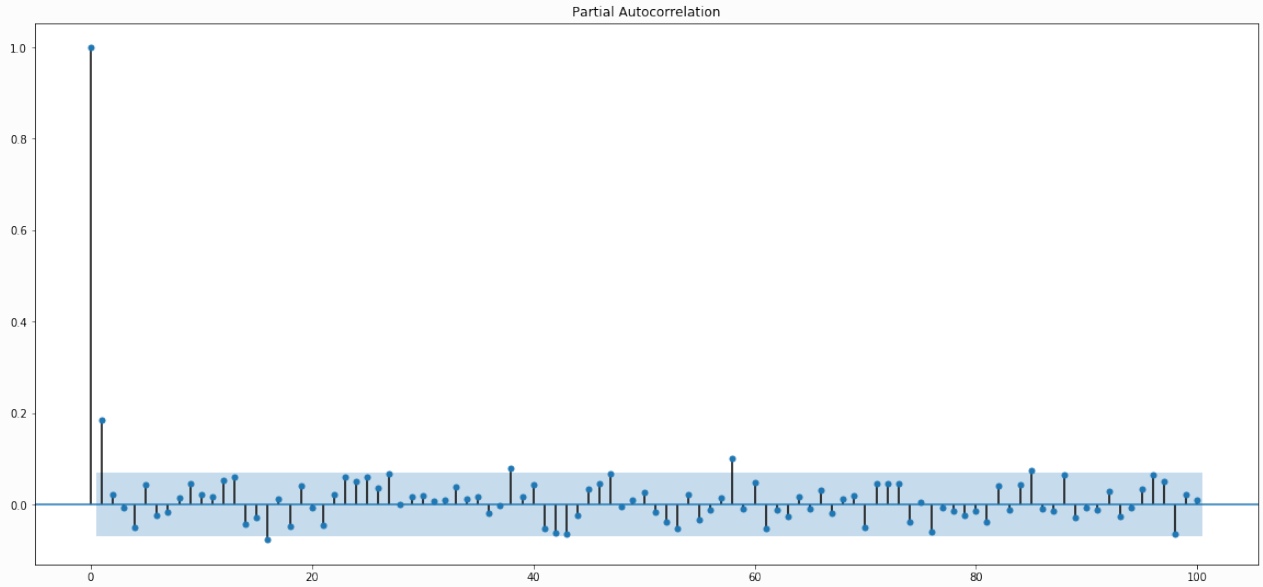
Example 3.1

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima_model import ARMA
from datetime import datetime
from itertools import product
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.stattools import acf,pacf
```

```
#读取的数据
df = pd.read_excel('nao.xlsx')
data1=df.iloc[3:]
data=data1['Unnamed: 1']
data=np.array(data)
# 序列的ACF and PACF
lag_acf = acf(data, nlags=20)
lag_pacf = pacf(data, nlags=20, method='ols')
fig, axes = plt.subplots(2,1, figsize=(20,20))
plot_acf(data, lags=100, ax=axes[0])
plot_pacf(data, lags=100, ax=axes[1])
plt.show()
#这里定阶p和q都是1
#AR(1)模型
tempModel = ARMA(data,(1,0)).fit(ic='aic', method='mle', trend='nc')
print(tempModel.summary())
#R2检验
delta = tempModel.fittedvalues - data # 残差
score = 1 - delta.var()/data.var()
print(score)
#这里编写MA(1)
tempModel2 = ARMA(data,(0,1)).fit(ic='aic', method='mle', trend='nc')
```

```
print(tempModel.summary())
##进行R2检验
delta_2 = tempModel2.fittedvalues - data # 残差
score_2 = 1 - delta_2.var()/data.var()
print(score2)
```






```
                              ARMA Model Results

==============================================================================
=======
Dep. Variable:                          y   No. Observations:
   815
Model:                     ARMA(1, 0)   Log Likelihood
-1151.312
Method:                            mle   S.D. of innovations
0.994
Date:                Thu, 13 Aug 2020   AIC
2306.625
```

```
Time:                        13:23:20   BIC
2316.031
Sample:                             0   HQIC
2310.235


==============================================================================
=======
                 coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
-------
ar.L1.y        0.1860      0.034      5.404      0.000       0.119
  0.253
                                    Roots

==============================================================================
======
                  Real          Imaginary           Modulus
Frequency
------------------------------------------------------------------------------
------
AR.1            5.3771           +0.0000j            5.3771
0.0000
------------------------------------------------------------------------------
------
0.03454268126958937
                              ARMA Model Results

==============================================================================
=======
Dep. Variable:                      y   No. Observations:
   815
Model:                     ARMA(1, 0)   Log Likelihood
-1151.312
Method:                           mle   S.D. of innovations
0.994
Date:                Thu, 13 Aug 2020   AIC
2306.625
Time:                        13:23:20   BIC
2316.031
Sample:                             0   HQIC
2310.235


==============================================================================
=======
                 coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
-------
```

```
ar.L1.y         0.1860       0.034       5.404       0.000        0.119
  0.253
                                    Roots

=============================================================================
======
                  Real          Imaginary           Modulus
Frequency
-----------------------------------------------------------------------------
------
AR.1            5.3771           +0.0000j            5.3771
0.0000
-----------------------------------------------------------------------------
------
0.03222459882982298
```

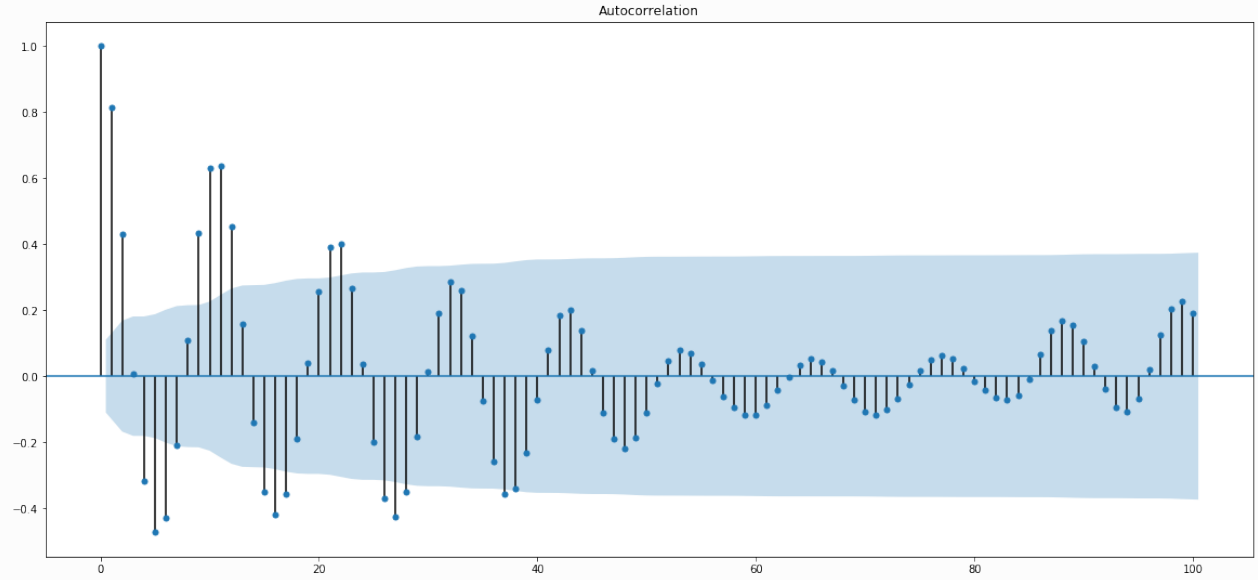
Example 3.2

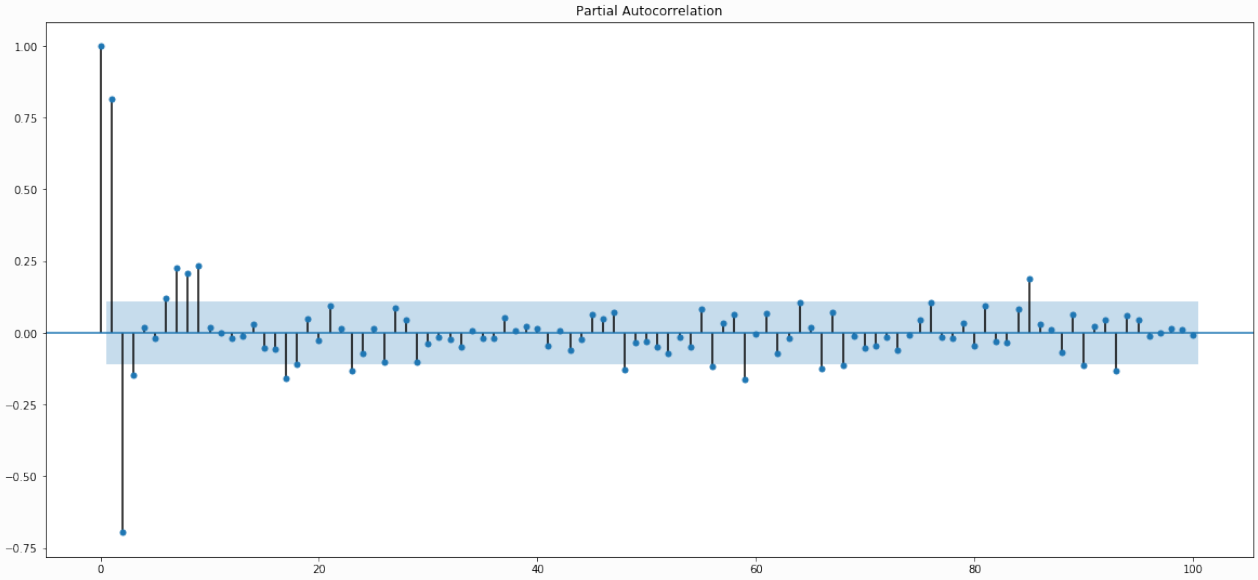
```
df = pd.read_excel('interest_rates.xlsx')
data1=df.iloc[3:788]
data=data1['Unnamed: 1'].tolist()
data=np.array(data)
# 求出序列的ACF and PACF
lag_acf = acf(data, nlags=20)
lag_pacf = pacf(data, nlags=20, method='ols')
fig, axes = plt.subplots(2,1, figsize=(20,20))
plot_acf(data, lags=100, ax=axes[0])
plot_pacf(data, lags=100, ax=axes[1])
plt.show()
##这里定阶p和q都是1
##编写AR(1)模型
order = (1,0)
tempModel = ARMA(data,order).fit(ic='aic', method='mle', trend='nc')
print(tempModel.summary())
##进行R2检验
delta = tempModel.fittedvalues - data # 残差
score = 1 - delta.var()/data.var()
print(score)
```

Example 3.3

```
#读取数据
df = pd.read_excel('sunspots.xlsx')
data1=df.iloc[3:321]
data=data1['Unnamed: 1']
data=np.array(data)

# 求时间序列的ACF and PACF
```

```
acf_lag = acf(data, nlags=20)
pacf_lag = pacf(data, nlags=20, method='ols')
fig, axes = plt.subplots(2,1, figsize=(20,20))
plot_acf(data, lags=100, ax=axes[0])
plot_pacf(data, lags=100, ax=axes[1])
plt.show()
##p,q=1
##AR(9)
tempModel = ARMA(data,(9,0)).fit(ic='aic', method='mle', trend='nc')
print(tempModel.summary())
##进行R2检验
delta = tempModel.fittedvalues - data # 残差
score = 1 - delta.var()/data.var()
print(score)
##编写AR(2)模型
tempModel = ARMA(data,(2,0)).fit(ic='aic', method='mle', trend='nc')
print(tempModel.summary())
##R2检验
delta = tempModel.fittedvalues - data # 残差
score = 1 - delta.var()/data.var()
print(score)
```





ARMA Model Results

Dep. Variable:	y	No. Observations:	317
Model:	ARMA(9, 0)	Log Likelihood	-1456.982
Method:	mle	S.D. of innovations	23.777
Date:	Thu, 13 Aug 2020	AIC	2933.965
Time:	13:25:27	BIC	2971.554
Sample:	0	HQIC	2948.980

```
==============================================================================
=======
                 coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
-------
ar.L1.y        1.1929      0.054     22.016      0.000       1.087
  1.299
ar.L2.y       -0.4067      0.086     -4.724      0.000      -0.575
-0.238
ar.L3.y       -0.1238      0.089     -1.389      0.166      -0.298
  0.051
ar.L4.y        0.1188      0.090      1.320      0.188      -0.058
  0.295
ar.L5.y       -0.0593      0.091     -0.650      0.516      -0.238
  0.119
ar.L6.y        0.0227      0.091      0.250      0.803      -0.156
  0.201
ar.L7.y        0.0340      0.091      0.373      0.709      -0.144
  0.212
ar.L8.y       -0.0536      0.088     -0.609      0.543      -0.226
  0.119
ar.L9.y        0.2550      0.055      4.661      0.000       0.148
  0.362
                                    Roots

==============================================================================
======
                  Real          Imaginary           Modulus
Frequency
------------------------------------------------------------------------------
------
AR.1            1.0079           -0.0000j            1.0079
-0.0000
AR.2            0.8499           -0.5756j            1.0265
-0.0948
AR.3            0.8499           +0.5756j            1.0265
0.0948
AR.4            0.4154           -1.0894j            1.1659
-0.1920
AR.5            0.4154           +1.0894j            1.1659
0.1920
AR.6           -1.1962           -0.4580j            1.2809
-0.4418
AR.7           -1.1962           +0.4580j            1.2809
0.4418
AR.8           -0.4680           -1.1985j            1.2866
-0.3093
AR.9           -0.4680           +1.1985j            1.2866
0.3093
------------------------------------------------------------------------------
------
```

0.8517151228852998

ARMA Model Results

Dep. Variable:	y	No. Observations:	317
Model:	ARMA(2, 0)	Log Likelihood	-1525.319
Method:	mle	S.D. of innovations	29.615
Date:	Thu, 13 Aug 2020	AIC	3056.638
Time:	13:25:28	BIC	3067.915
Sample:	0	HQIC	3061.143

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1.y	1.4793	0.045	32.914	0.000	1.391	1.567
ar.L2.y	-0.5907	0.045	-13.149	0.000	-0.679	-0.503

Roots

	Real	Imaginary	Modulus	Frequency
AR.1	1.2521	-0.3537j	1.3011	-0.0438
AR.2	1.2521	+0.3537j	1.3011	0.0438

0.7910567584643853

第四章：非平稳时间序列的 ARIMA模型

ARMA依赖于弱平稳的时间序列，所以均值和方差是常数，并要求自协方差只依赖于时滞。一种可能性是，均值在时间上演变为 d 阶的(非随机)多项式，而误差则假定为一个 随机的、平稳的、但可能是自相关的、零均值过程。

$$x_t = \mu_t + \epsilon_t = \sum_{j=0}^{d} \beta_j t^j + \Psi(B) a_t$$

```
#1 获取时间序列数据;

import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
data=
[10930,10318,10595,10972,7706,6756,9092,10551,9722,10913,11151,8186,642
2,
6337,11649,11652,10310,12043,7937,6476,9662,9570,9981,9331]
data=pd.Series(dta)
data.index=pd.Index(sm.tsa.datetools.dates_from_range('1996','2019'))
data
```

```
# 2：对数据绘图，检查是否平稳；对于非平稳时间序列要先进行d阶差分运算，化为平稳时间序列;

data.plot(figsize=(10,8))
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(data)
```

```
#3:画出ACF图，由ACF图可以看出，自相关系数长期大于0，说明序列具有很强的长期相关性，所以趋势并不平稳，因此要做差分运算，依次测试。

dif = np.diff(dta)
dif.plot()

#4：**经过第二步处理，已经得到平稳时间序列。要对平稳时间序列分别求得其自相关系数ACF和偏自相关系数PACF ，通过对自相关图和偏自相关图的分析，得到最佳的阶层 p 和阶数 q；

from statsmodels.graphics.tsaplots import plot_acf
plot_acf(dif)
from statsmodels.graphics.tsaplots import plot_pacf
plot_pacf(data)
```

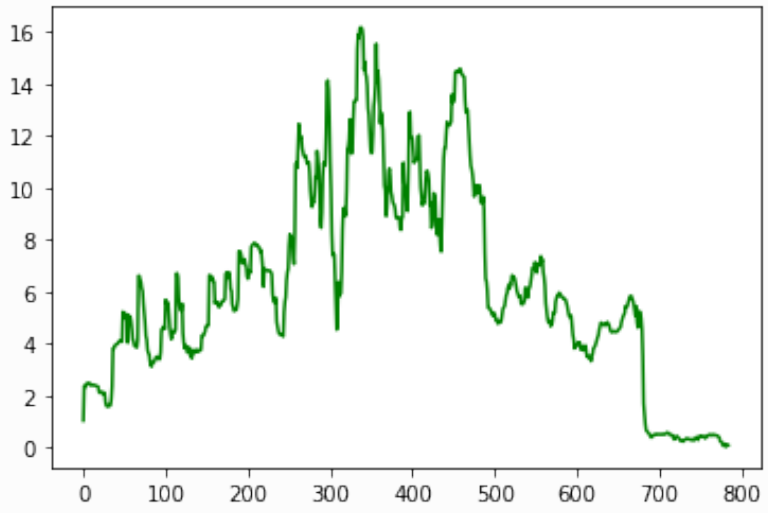
```
#5:拟合ARIMA(0,1,1)
from statsmodels.tsa.arima_model import ARIMA
model = ARIMA(dif, (0,1,1)).fit()
print(model.summary())
```

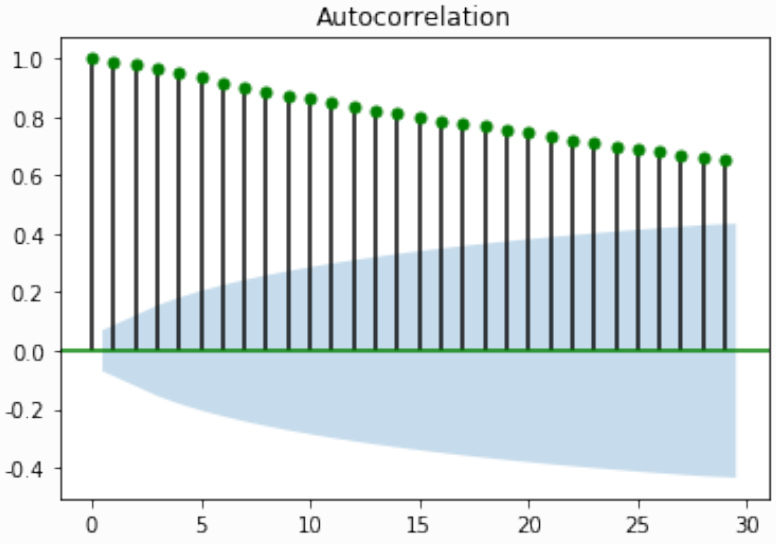
平稳性检验白噪声检验。能够适用ARMA模型进行分析预测的时间序列必须满足的条件是平稳非白噪声序列。对数据的平稳性进行检验是时间序列分析的重要步骤，一般通过时序图和相关图来检验时间序列的平稳性。时序图的特点是直观简单但是误差较大，自相关图即自相关和偏自相关函数图相对复杂但是结果更加准确。本文先用时序图进行直观的判断再利用相关图进行更进一步的检验。对于非平稳时间序列中若存在增长或下降趋势，则需要进行差分处理然后进行平稳性检验直至平稳为止。其中，差分的次数就是模型ARIMA(p,d,q)的阶数，理论上说，差分的次数越多，对时序信息的非平稳确定性信息的提取越充分，但是从理论上说，差分的次数并非越多越好，每一次差分运算，都会造成信息的损失，所以应当避免过分的差分，一般在应用中，差分的阶数不超过2。

Example 4.1

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.arima_model import ARIMA
```

```
##这里开始读取数据
df = pd.read_excel('interest_rates.xlsx')
data_1=df.iloc[3:788]
data=data_1['Unnamed: 1'].tolist()
data=np.array(data)

plt.plot(data ,color="green")
plt.show()
#画出自相关性图
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
plot_acf(data,color="green")
plt.show()
```

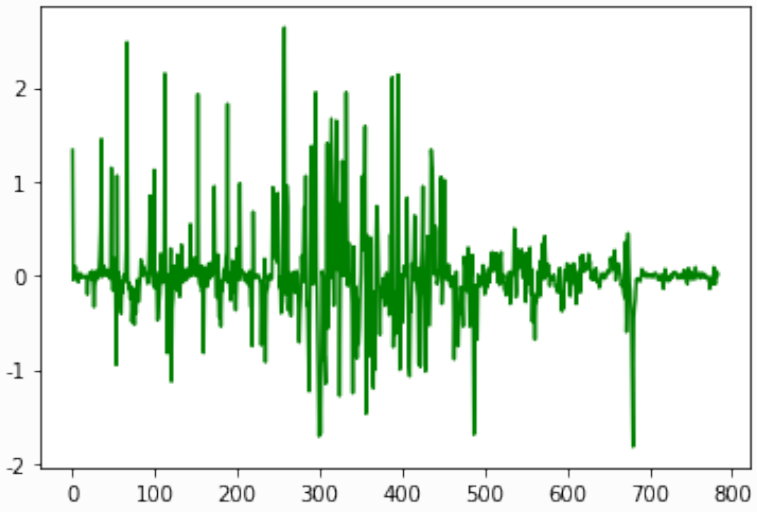


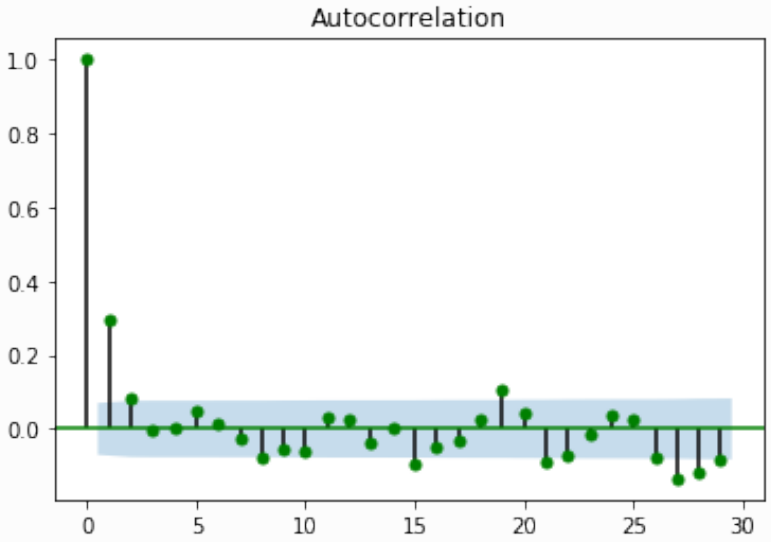


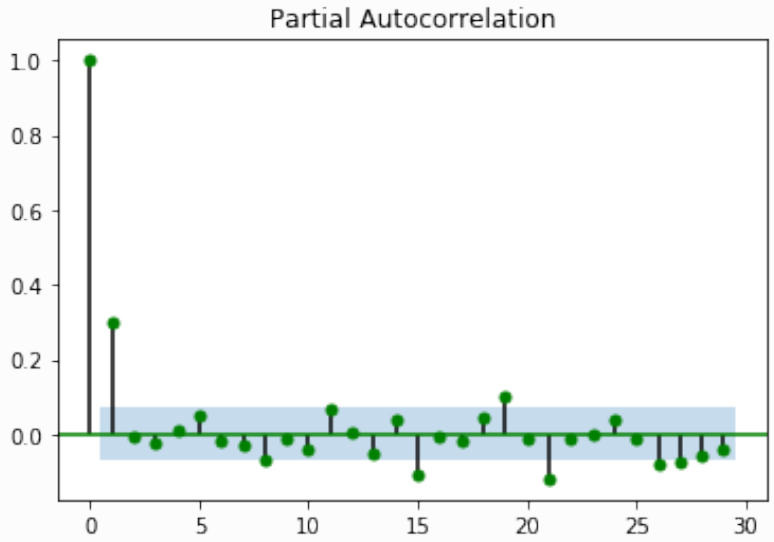

```
#平稳性检测
from statsmodels.tsa.stattools import adfuller
print('原始序列的检验结果为：',adfuller(data))
D_data = np.diff(data)
```

```
plt.plot(D_data,color="green")
plt.show()
plot_acf(D_data,color="green")     #自相关图
plot_pacf(D_data,color="green")    #偏相关图
plt.show()
print('差分序列的ADF 检验结果为： ', adfuller(D_data))   #平稳性检验
from statsmodels.stats.diagnostic import acorr_ljungbox
print('差分序列的白噪声的检验结果：',acorr_ljungbox(D_data, lags= 1)) #返回统计量和 p 值
# p值远小于 0.05
model = ARIMA(data, (0,1,1)).fit()
print(model.summary() )
model.forecast(5)
```

```
原始序列的检验结果为： (-1.753774855211307, 0.4036404438743404, 21, 763,
{'1%': -3.4389495235166416, '5%': -2.8653354363373253, '10%':
-2.56879107669766}, 902.7421499838515)
```







差分序列的ADF 检验结果为： (-6.591050104448266, 7.113180208123565e-09, 20, 763, {'1%': -3.4389495235166416, '5%': -2.8653354363373253, '10%': -2.56879107669766}, 903.5811535071564)
差分序列的白噪声的检验结果： (array([69.8147034]), array([6.51454612e-17]))

ARIMA Model Results

Dep. Variable:	D.y	No. Observations:	784
Model:	ARIMA(0, 1, 1)	Log Likelihood	-462.056
Method:	css-mle	S.D. of innovations	0.436
Date:	Fri, 14 Aug 2020	AIC	930.111
Time:	21:20:10	BIC	944.105
Sample:	1	HQIC	935.492

	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.0007	0.020	-0.036	0.971	-0.040	0.038
ma.L1.D.y	0.2778	0.032	8.705	0.000	0.215	0.340

Roots

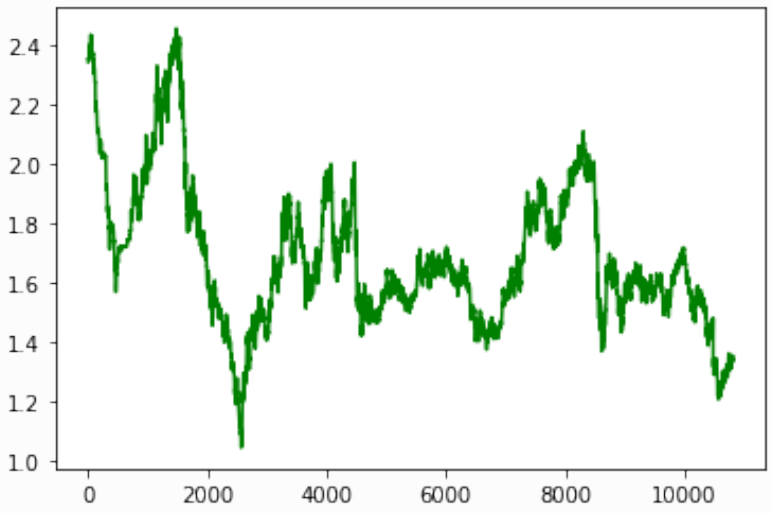
```
=============================================================================
======
                  Real          Imaginary           Modulus
Frequency
-----------------------------------------------------------------------------
------
MA.1           -3.5993           +0.0000j            3.5993
0.5000
-----------------------------------------------------------------------------
------
```

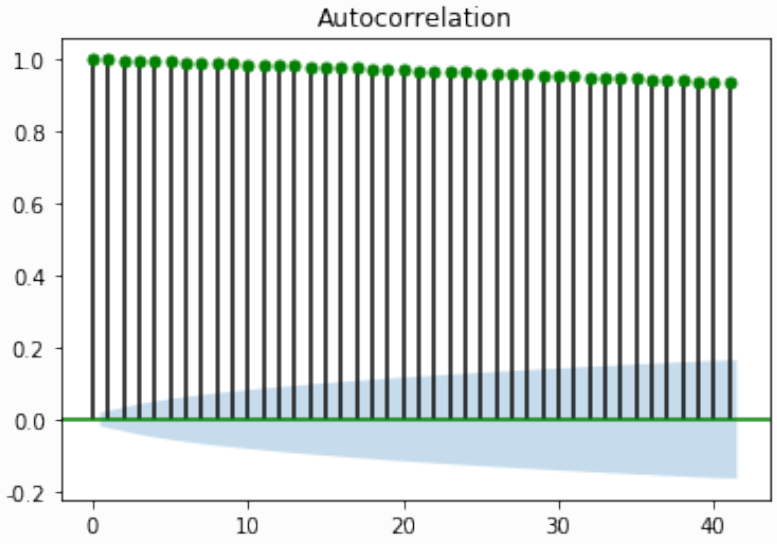
```
(array([0.08727114, 0.08655061, 0.08583007, 0.08510954, 0.08438901]),
 array([0.43620933, 0.70779518, 0.90092699, 1.05941723, 1.19710497]),
 array([[-0.76768344,  0.94222571],
        [-1.30070245,  1.47380366],
        [-1.67995438,  1.85161453],
        [-1.99131008,  2.16152916],
        [-2.26189361,  2.43067164]]))
```

Example 4.2

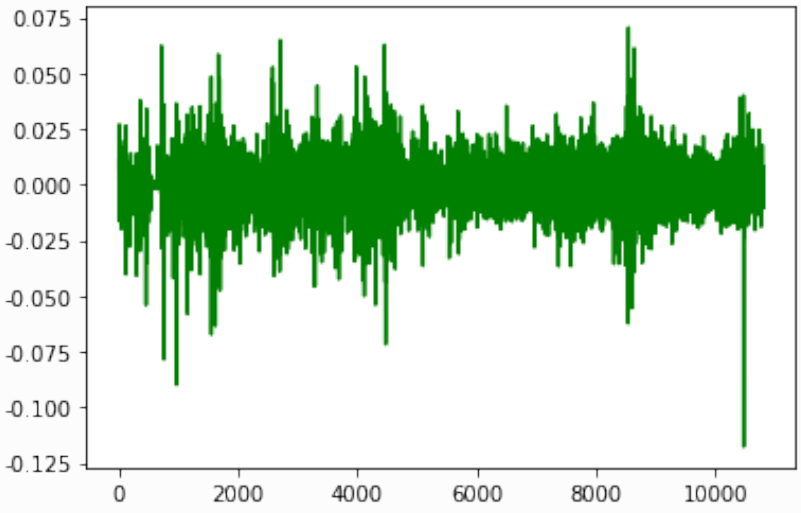
```
from statsmodels.stats.diagnostic import acorr_ljungbox
df = pd.read_excel('dollar.xlsx')
data_new=df.iloc[3:10822]
data=data_new['Unnamed: 1'].tolist()
data=np.array(data)
#时序图
plt.plot(data,color="green")
plt.show()
#自相关性图
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
plot_acf(data,color="green")
plt.show()
#平稳性检测
from statsmodels.tsa.stattools import adfuller
print('原始序列的检验结果为：',adfuller(data))
Dif_data = np.diff(data)
plt.plot(Dif_data,color="green")
plt.show()
plot_acf(Dif_data,color="green")
plot_pacf(Dif_data,color="green")
plt.show()
print('差分序列的ADF 检验结果为：  ', adfuller(D_data))
```

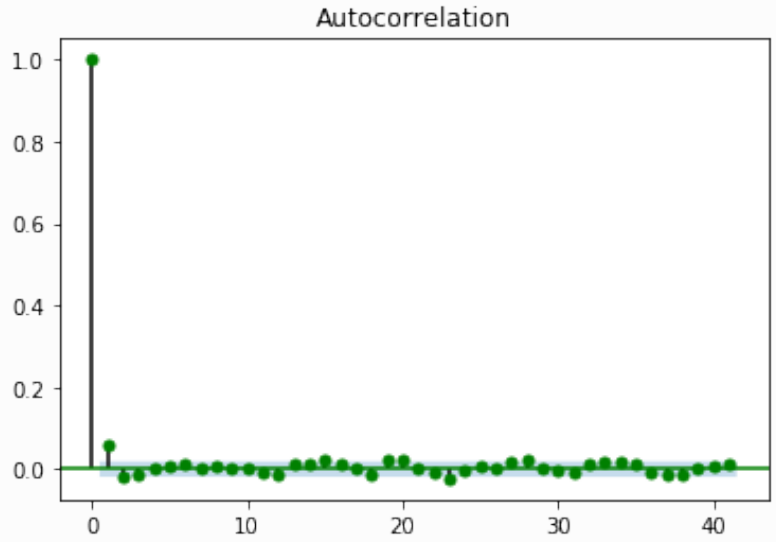
```
print('差分序列的白噪声检验结果：',acorr_ljungbox(D_data, lags= 1)) #返回统计量 p
model = ARIMA(data, (0,1,1)).fit()
print(model.summary() )
model.forecast(5)
##R2检验
delta = model.fittedvalues - np.diff(data) # 残差
score = 1 - delta.var()/np.diff(data).var()
print(score)
```

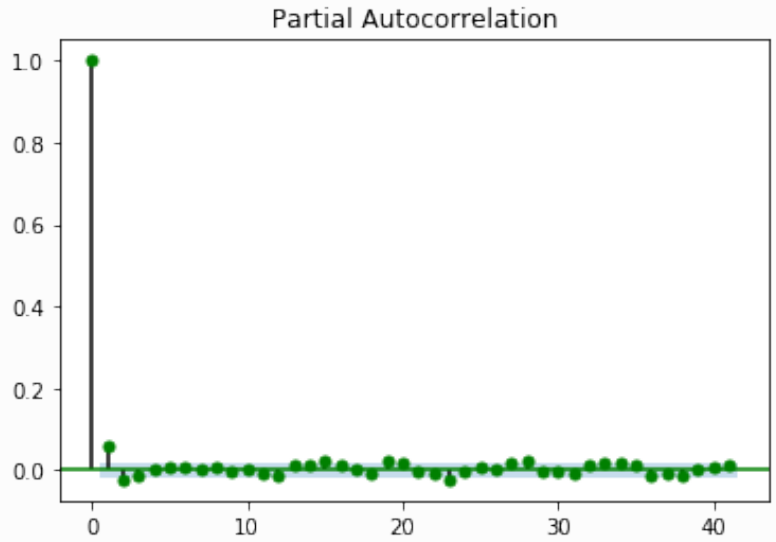





```
原始序列的检验结果为： (-2.955203324315911, 0.0392949390592251, 2, 10815, {'1%': -3.43095479452452, '5%': -2.8618072854216945, '10%': -2.566912270895225}, -68222.84876939279)
```


0.075
0.050
0.025
0.000
-0.025
-0.050
-0.075
-0.100
-0.125
0
2000
4000
6000
8000
10000

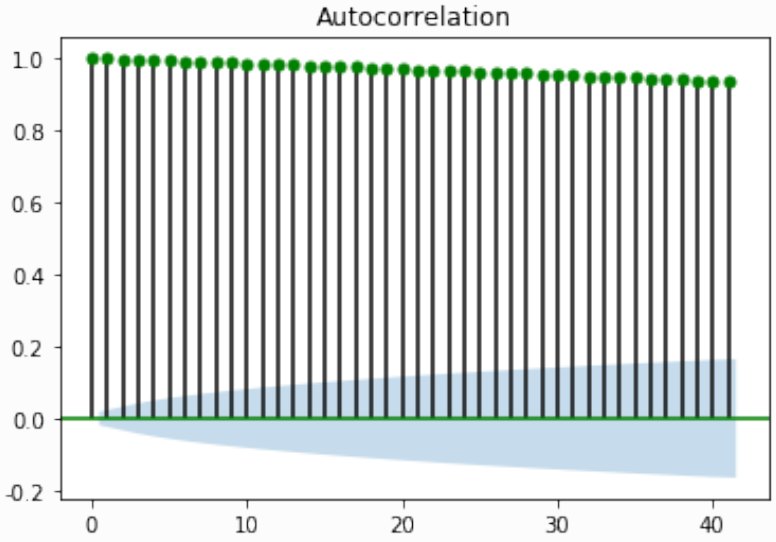

Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40


Partial Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40

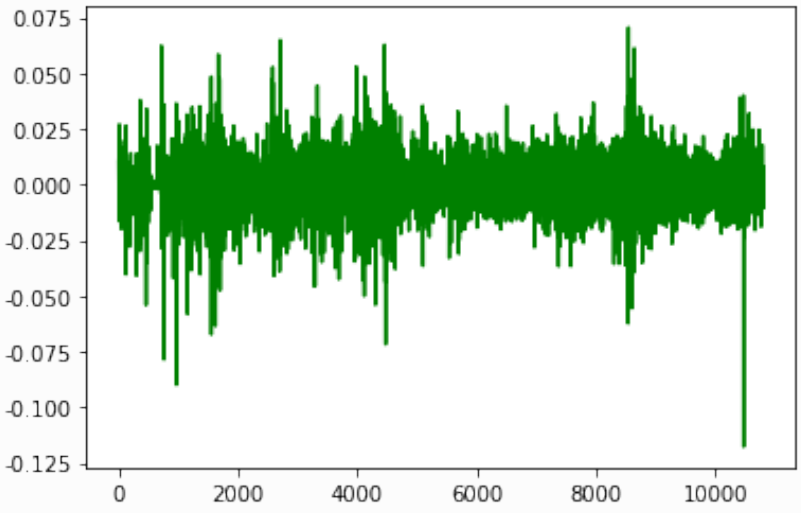
```
差分序列的ADF 检验结果为：  (-6.591050104448266, 7.113180208123565e-09, 20,
763, {'1%': -3.4389495235166416, '5%': -2.8653354363373253, '10%':
-2.56879107669766}, 903.5811535071564)
差分序列的白噪声检验结果：  (array([69.8147034]), array([6.51454612e-17]))
                              ARIMA Model Results

==============================================================================
=======
Dep. Variable:                    D.y   No. Observations:
10817
Model:                 ARIMA(0, 1, 1)   Log Likelihood
34238.095
Method:                       css-mle   S.D. of innovations
0.010
Date:                Fri, 14 Aug 2020   AIC
-68470.189
Time:                        21:21:55   BIC
-68448.323
Sample:                             1   HQIC
-68462.816


==============================================================================
=======
                 coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
-------
const      -9.146e-05      0.000     -0.876      0.381      -0.000
  0.000
ma.L1.D.y      0.0627      0.010      6.425      0.000       0.044
  0.082
                                    Roots

==============================================================================
======
                  Real          Imaginary           Modulus
Frequency
------------------------------------------------------------------------------
------
MA.1          -15.9442           +0.0000j           15.9442
0.5000
------------------------------------------------------------------------------
------
0.003779701178640016
```

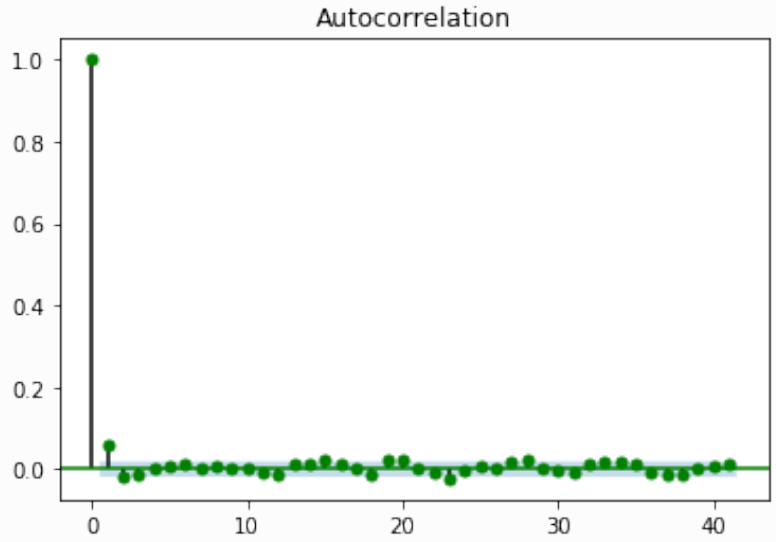
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
plot_acf(data,color="green")
plt.show()
```

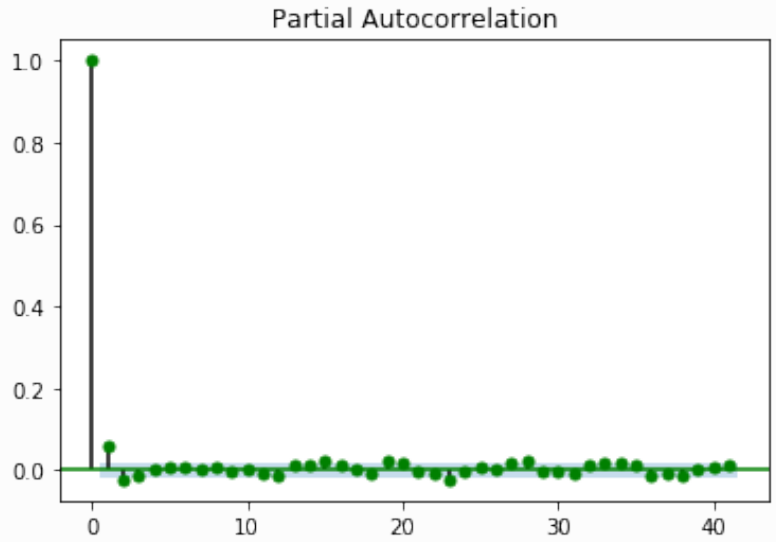
```
#平稳性检测
from statsmodels.tsa.stattools import adfuller
print('原始序列的检验结果为：',adfuller(data))
Dif_data = np.diff(data)
print(Dif_data)
plt.plot(Dif_data,color="green")
plt.show()
plot_acf(Dif_data,color="green")    #画出自相关图
plot_pacf(Dif_data,color="green")   #画出偏相关图
plt.show()
print('差分序列的ADF 检验结果为：  ', adfuller(D_data))   #平稳性检验
from statsmodels.stats.diagnostic import acorr_ljungbox
print('差分序列的白噪声检验结果：',acorr_ljungbox(D_data, lags= 1)) #

model = ARIMA(data, (0,1,3)).fit()
print(model.summary() )
model.forecast(5)
##进行R2检验
delta = model.fittedvalues - np.diff(data) # 残差
score = 1 - delta.var()/np.diff(data).var()
print(score)
```




```
原始序列的检验结果为：  (-2.955203324315911, 0.0392949390592251, 2, 10815,
{'1%': -3.43095479452452, '5%': -2.8618072854216945, '10%':
-2.566912270895225}, -68222.84876939279)
[ 0.0104  0.0034 -0.0067 ...  0.0012  0.0037  0.0074]
```


0.075
0.050
0.025
0.000
-0.025
-0.050
-0.075
-0.100
-0.125
0
2000
4000
6000
8000
10000


Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40


Partial Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40

差分序列的ADF 检验结果为： (-6.591050104448266, 7.113180208123565e-09, 20, 763, {'1%': -3.4389495235166416, '5%': -2.8653354363373253, '10%': -2.56879107669766}, 903.5811535071564)
差分序列的白噪声检验结果： (array([69.8147034]), array([6.51454612e-17]))

ARIMA Model Results

Dep. Variable:	D.y	No. Observations:	10817
Model:	ARIMA(0, 1, 3)	Log Likelihood	34240.883
Method:	css-mle	S.D. of innovations	0.010
Date:	Fri, 14 Aug 2020	AIC	-68471.766
Time:	21:32:59	BIC	-68435.321
Sample:	1	HQIC	-68459.477

	coef	std err	z	P>|z|	[0.025	0.975]
const	-9.153e-05	0.000	-0.906	0.365	-0.000	0.000
ma.L1.D.y	0.0614	0.010	6.386	0.000	0.043	0.080
ma.L2.D.y	-0.0184	0.010	-1.912	0.056	-0.037	0.000
ma.L3.D.y	-0.0140	0.010	-1.475	0.140	-0.033	0.005

Roots

	Real	Imaginary	Modulus	Frequency
MA.1	4.0697	-0.0000j	4.0697	-0.0000
MA.2	-2.6896	-3.2065j	4.1852	-0.3611
MA.3	-2.6896	+3.2065j	4.1852	0.3611

```
0.004293200169928846
```

第五章 单位根、趋势平稳性、以及分数差分

正如我们在yy4.5-4.13中所显示的那样，积分阶数d是时间序列所显示的特性的关键决定因素。如果我们将自己限制在最常见的零值和d的一个值上，因此x t是I(0)或I(1)，那么将这两个过程的性质放在一起是有用的。如果x_t是I(0)，即使以前已经使用这样的符号表示序列的分布特征，有时我们仍将x_t表示为x_t，那么，为了方便起见，假设x_t的均值为零；

- x_t的方差是有限的，不依赖于t；
- a_t对x_t的值只是暂时的影响；
- x_t=0的相交之间的预期时间长度是有限的，因此xt会在其平均值零附近波动；
- 自相关ρk在足够大的k时稳定减小，因此它们的和是有限的。

1、单位根检验

①利用迪基—福勒检验（Dickey-Fuller Test）和菲利普斯—佩荣检验（Philips-Perron Test）,也可以测定时间序列的随机性，这是在计量经济学中非常重要的两种单位根检验方法，与前者不同的事，后一个检验方法主要应用于一阶自回归模型的残差不是白噪声，而且存在自相关的情况。

②随机游动

如果在一个随机过程中，yt的每一次变化均来自于一个均值为零的独立同分布，即随机过程yt满足：

$$y_t = Y_{t-1} + \epsilon_t, t = 1, 2 \dots$$

其中εt独立同分布，并且：

$$E(\epsilon_t) = 0, Var(\epsilon_t) = E(\epsilon_t^2) = \sigma^2 < \infty$$

称这个随机过程是随机游动。它是一个非平稳过程。

③单位根过程

设随机过程yt满足：$y_t = \rho y_{t-1} + \mu_t, t = 1, 2 \dots$,其中ρ = 1，μt为一个平稳过程并且$E(\mu_t) = 0, cov(\mu_t, \mu_{t-s}) = \mu_s < \infty, s = 0, 1, 2 \dots$

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
data = pd.read_csv("/root/experiment01/dataset/nao.csv")
```

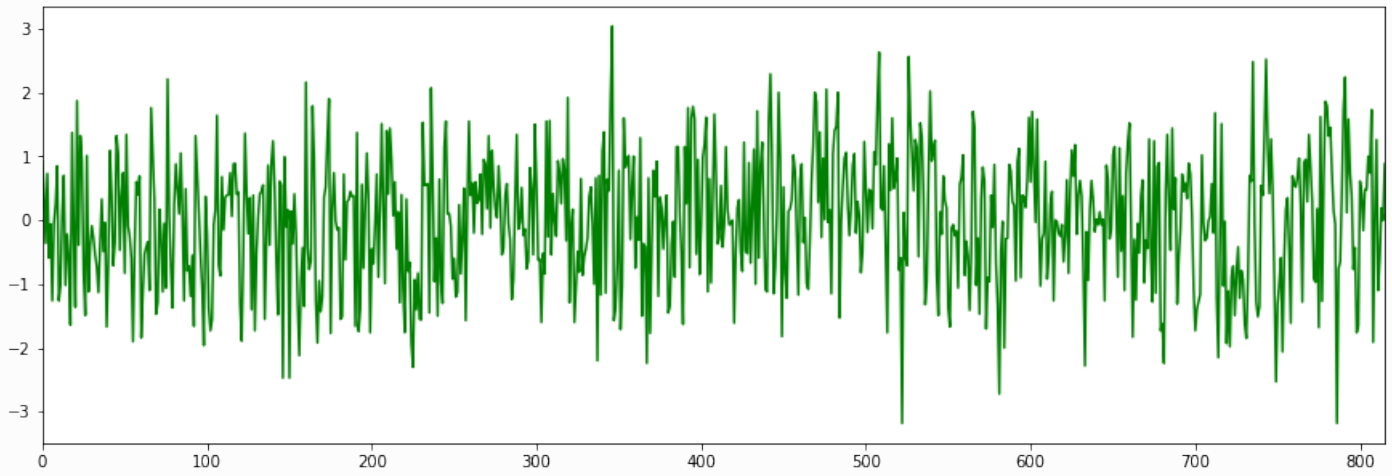
```
data.head()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	time	nao
0	1950M01	0.92
1	1950M02	0.40
2	1950M03	-0.36
3	1950M04	0.73
4	1950M05	-0.59

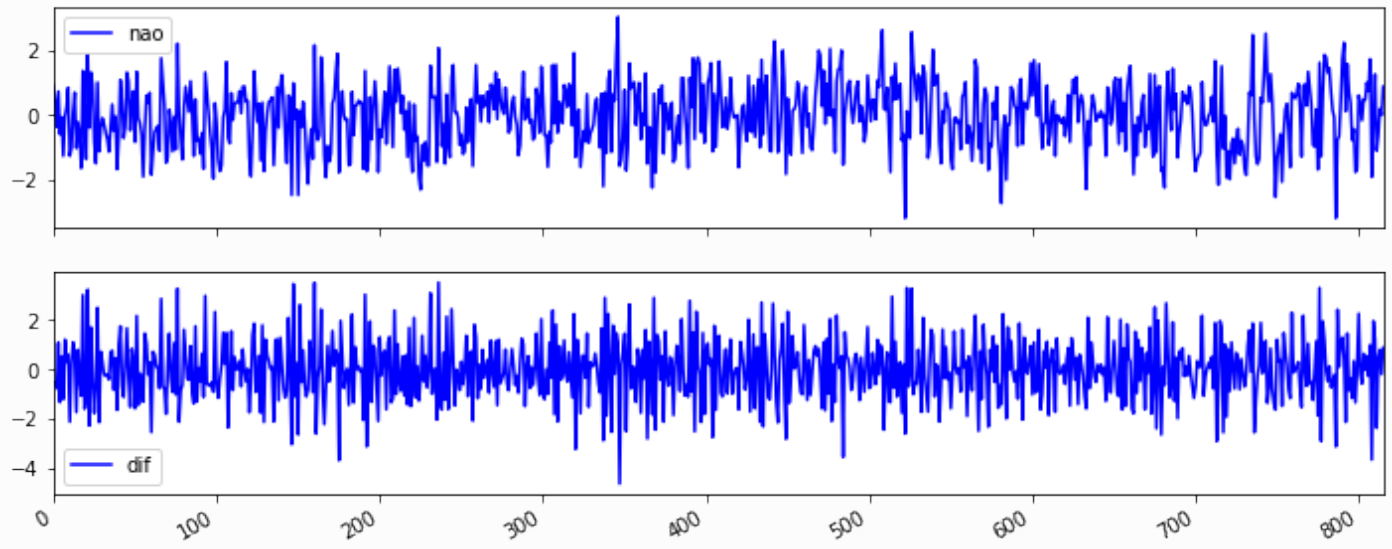
```
plt.figure(figsize=(15,5))
data['nao'].plot(color="green")
```

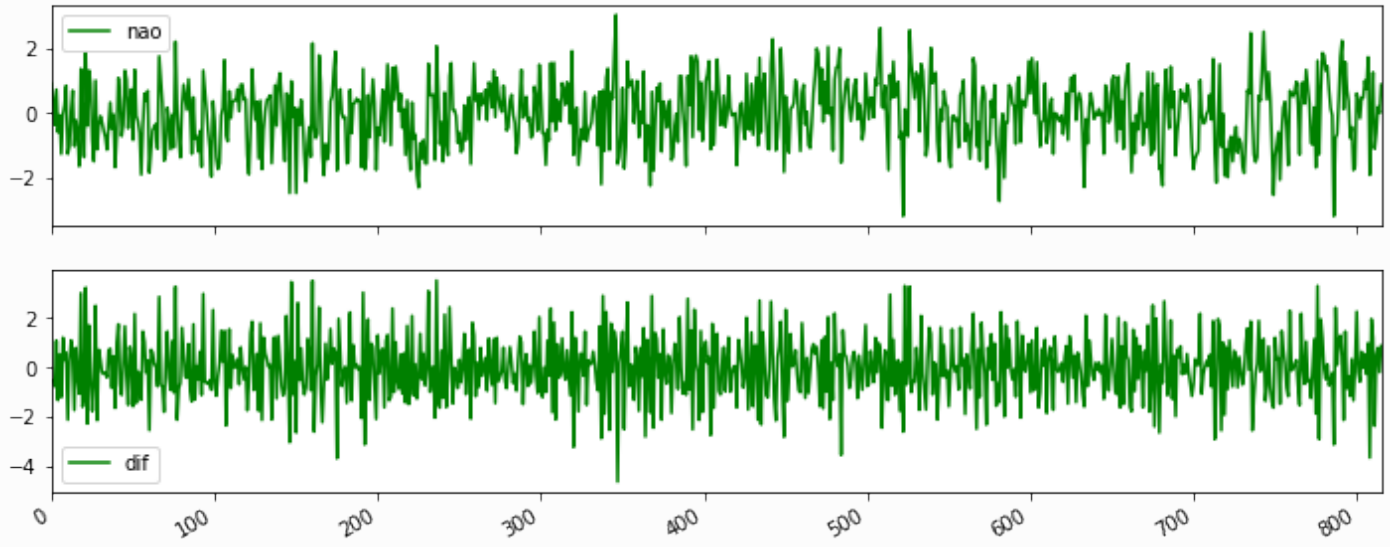
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f03ea1df3d0>
```




```
data['dif'] = data['nao'].diff(1)
data.plot(subplots=True,figsize=(12,5),color="blue")
data.plot(subplots=True,figsize=(12,5),color="green")
```

```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f03ea141150>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f03ea0f1cd0>],
      dtype=object)
```





上图为原始数据与一阶差分

Example 5.5

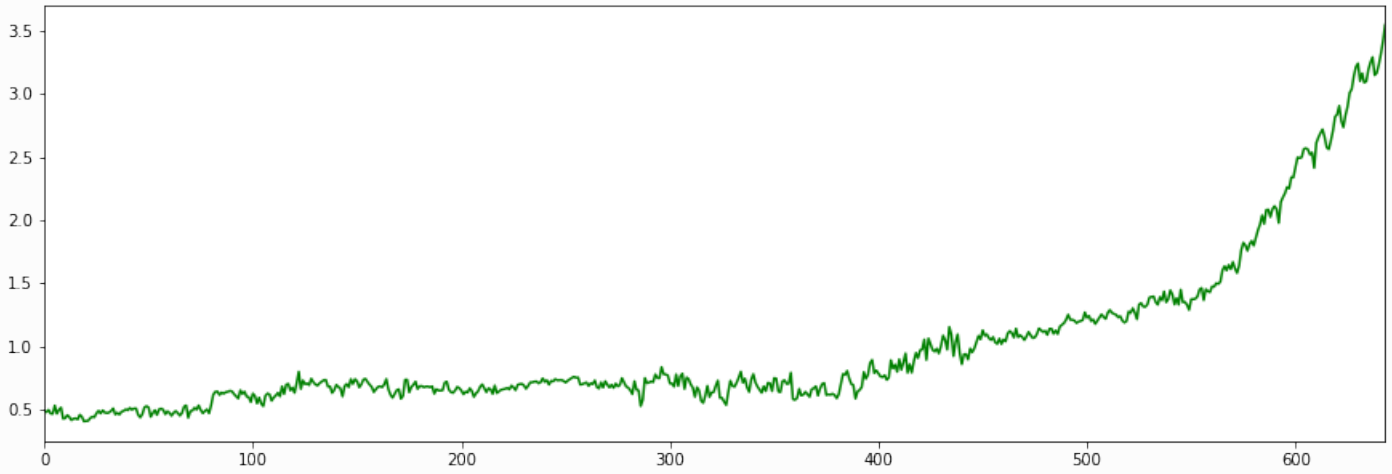
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

```
data = pd.read_csv("/root/experiment01/dataset/gdp.csv")
 # 将字符串索引转换成时间索引
ts = data  # 生成pd.Series对象
# 查看数据格式
ts.head()
```

	year	index
0	1270	0.483038
1	1271	0.477300
2	1272	0.494482
3	1273	0.465640
4	1274	0.463213

```
plt.figure(figsize=(15,5))
data['index'].plot(color="green")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f03e9f47290>
```

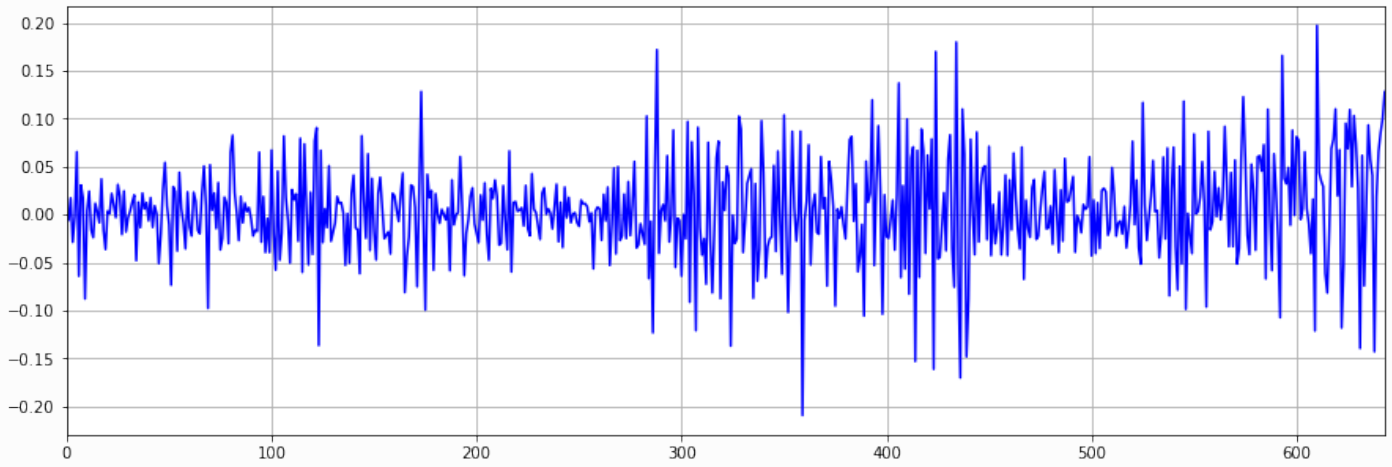



```
#平稳性检测
from statsmodels.tsa.stattools import adfuller as ADF
print(u'原始序列的ADF检验结果为：', ADF(data['index']))
```

```
原始序列的ADF检验结果为： (7.148596453970319, 1.0, 6, 637, {'1%':
-3.44065745275905, '5%': -2.8660879520543534, '10%': -2.5691919933016076},
-2001.7160367999013)
```

```
data['diff_01'] = data['index'].diff(1)

plt.figure(figsize=(15,5))
#绘制差分后的时序图
data['diff_01'].plot(grid=True,color="blue")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f03e9f47d10>
```




```
D_data = data.dropna()
print(u'原始序列的ADF检验结果为：', ADF(D_data['diff_01']))
```

```
原始序列的ADF检验结果为：  (-2.9545365533439276, 0.03936379973573927, 17, 625,
{'1%': -3.440856177517568, '5%': -2.86617548304384, '10%': -2.56923863104},
-1972.9453748537412)
```

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
fig = plt.figure(figsize=(15,5),dpi=300)
ax1 = fig.add_subplot(211)
fig = plot_acf(data['index'],lags=200,ax=ax1,color="green")
fig.tight_layout()

ax2 = fig.add_subplot(212)
fig = plot_pacf(data['index'],lags=200,ax=ax2,color="b")
fig.tight_layout()
```

Autocorrelation

```
#白噪声检测
from statsmodels.stats.diagnostic import acorr_ljungbox
print(u'差分序列的白噪声检验结果为：', acorr_ljungbox(D_data['diff_01'], lags=1)) #返回统计量和p值
```

```
差分序列的白噪声检验结果为：  (array([30.39857711]), array([3.51786103e-08]))
```

```
#aic  bic hqic越小越好
from statsmodels.tsa.arima_model import ARIMA
data = data.dropna()
ts_log = np.log(ts)
from statsmodels.tsa.arima_model import ARMA
model = ARMA(data['diff_01'], order=(2,2)).fit()
print("aic  bic hqic")
print(model.bic,model.aic,model.hqic)
```

```
/root/anaconda3/envs/jupyter_notebook/lib/python3.7/site-packages/ipykernel_launcher.py:4: RuntimeWarning: invalid value encountered in log
  after removing the cwd from sys.path.
/root/anaconda3/envs/jupyter_notebook/lib/python3.7/site-packages/statsmodels/tsa/base/tsa_model.py:215: ValueWarning: An unsupported index was provided and will be ignored when e.g. forecasting.
  ' ignored when e.g. forecasting.', ValueWarning)
```

```
aic  bic hqic
-2003.2450182058167 -2030.0418865512424 -2019.642925814479
```

Example 5.10

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

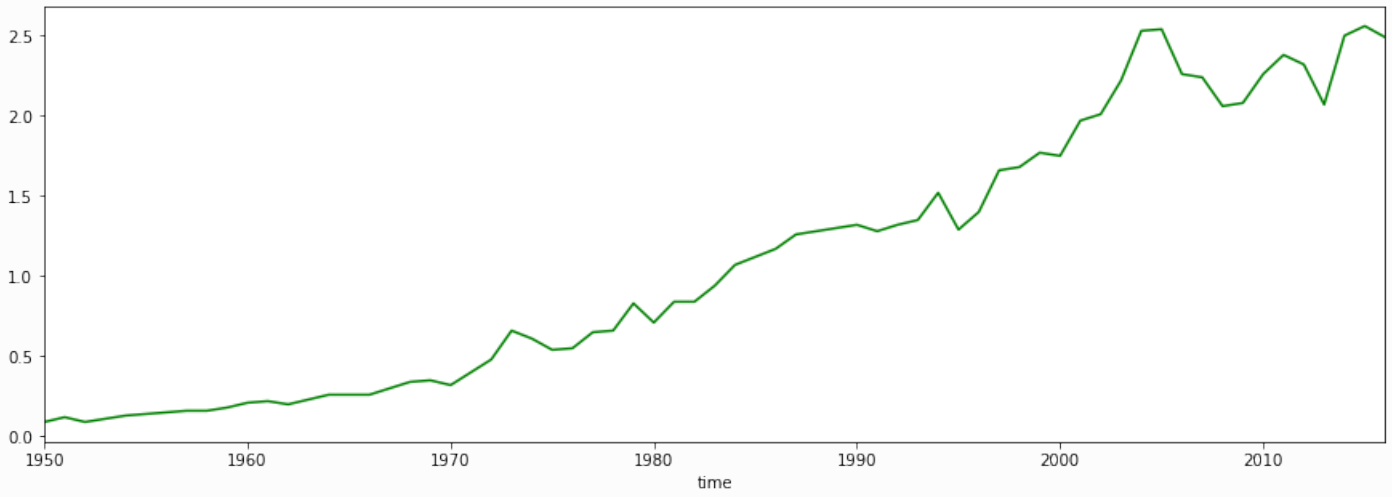
```
data = pd.read_csv("/root/experiment01/dataset/wine_spirits.csv",index_col=0)
data.head()
```

	wine_spirits
time	
1950	0.09

1951	0.12
1952	0.09
1953	0.11
1954	0.13

```
plt.figure(figsize=(15,5))
data['wine_spirits'].plot(color="green")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f03e232ead0>
```



第六章 中断与非线性趋势

5.9-5.10中概述的趋势平稳（TS）与差异平稳（DS）二分法以及相关的测试过程既简单又易于实施，但是否一定现实呢？在某些情况下，“全局”线性趋势的TS替代方案是否过于简单，从而表明可能需要更复杂的趋势函数？通常，趋势的一个更合理的候选者是一个线性函数，该线性函数在一个或多个时间点“中断”。

有几种方式可以终端趋势。为简单起见，假设在已知时间点T_bc_1，T_bc，T处有一个单独的中断，上标“ c“表示“正确的”中断日期，由于这一原因，这一区别将变得很重要。课程。最简单的突破趋势模型是“水平移动”，其中x t的水平在T_{bc}处从μ_0变为μ_1 μ_0 μ。

Example 6.2

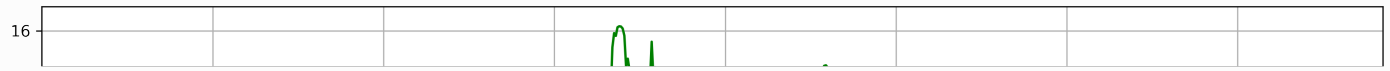
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data = pd.read_csv("/root/experiment01/dataset/interest_rates.csv",index_col=0)
data.head()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	interest_rates
time	
1952-1-1	0.994750
1952-2-1	1.028250
1952-3-1	2.365042
1952-4-1	2.317500
1952-5-1	2.350833

```
plt.figure(figsize=(15,5),dpi=300)
data['interest_rates'].plot(grid=True,color="green")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f03e2317650>
```




```
data['diff_01'] = data['interest_rates'].diff(1)
data['diff_02'] = data['interest_rates'].diff(2)
data.plot(subplots=True,figsize=(15,5))
```

```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f03e2287a10>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f03e2238750>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f03e226add0>],
      dtype=object)
```

Example 6.3

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data = pd.read_csv("/root/experiment01/dataset/dollar.csv",index_col=0)
data.head()
plt.figure(figsize=(15,5))
data['dollar'].plot(grid=True,color="green")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f03e2062910>
```

```
data['diff_01'] = data['dollar'].diff(1)
data.plot(subplots=True,figsize=(12,5))
#1阶差分
```

```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f03e20b8510>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f03e20d57d0>],
      dtype=object)
```

Example 6.3

```
from statsmodels.tsa.seasonal import seasonal_decompose
import statsmodels.api as sm
data = pd.read_csv("/root/experiment01/dataset/interest_rates.csv")
df = pd.DataFrame(data)
df.index = pd.DatetimeIndex(start=df['time']
[0],periods=len(df['time']),freq='MS')  #生成日期索引
ts = df['interest_rates']  # 生成pd.Series对象
```

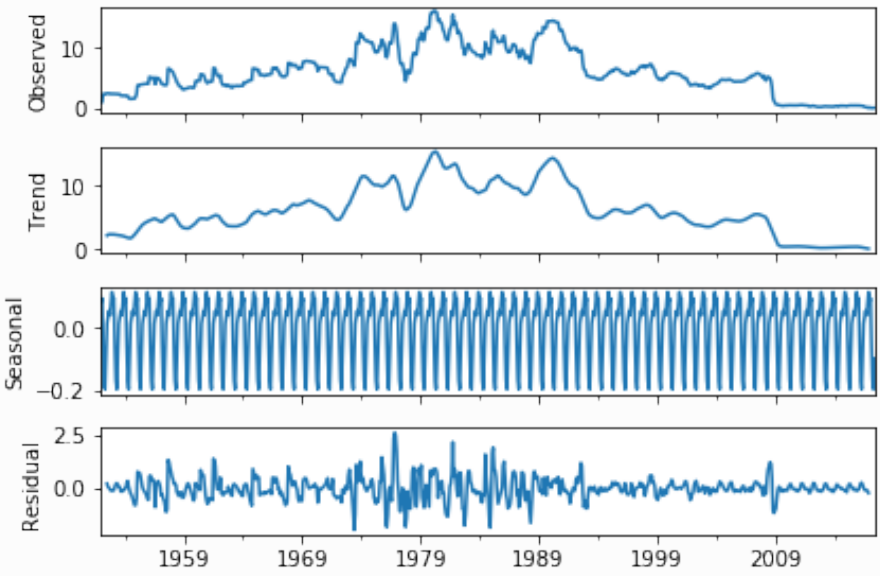
```
/root/anaconda3/envs/jupyter_notebook/lib/python3.7/site-
packages/ipykernel_launcher.py:5: FutureWarning: Creating a DatetimeIndex by
passing range endpoints is deprecated.  Use `pandas.date_range` instead.
  """
```

```
def auto_corr(timeseries, lags):
    fig = plt.figure(figsize=(12, 8))
    ax1 = fig.add_subplot(211)
    sm.graphics.tsa.plot_acf(timeseries, lags=lags, ax=ax1,color="green")
    ax2 = fig.add_subplot(212)
    sm.graphics.tsa.plot_pacf(timeseries, lags=lags, ax=ax2,color="green")
    plt.show()

diff = ts.diff(1)
diff = ts.fillna(0)

auto_corr(diff, 100)
```

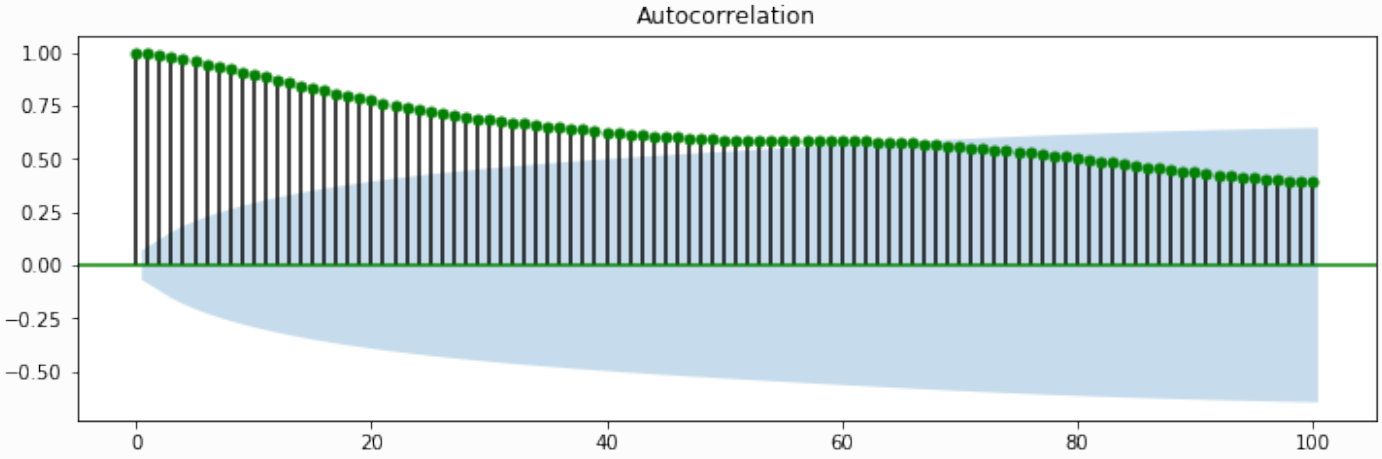
!

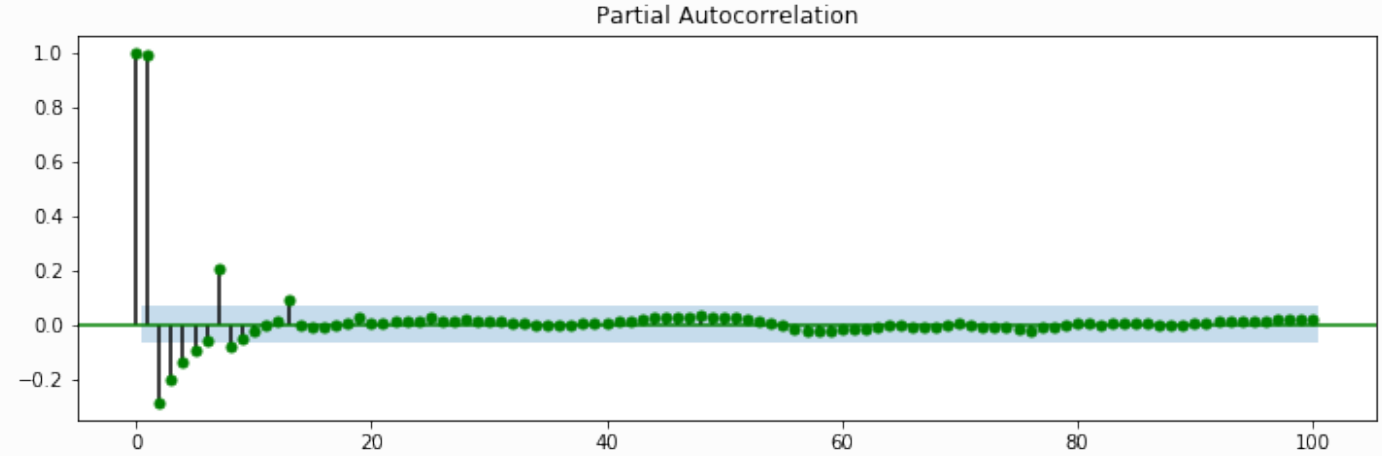
```
decomposition = seasonal_decompose(ts)#趋势项
trend = decomposition.trend#季节性
seasonal = decomposition.seasonal#残差
residual = decomposition.resid
```

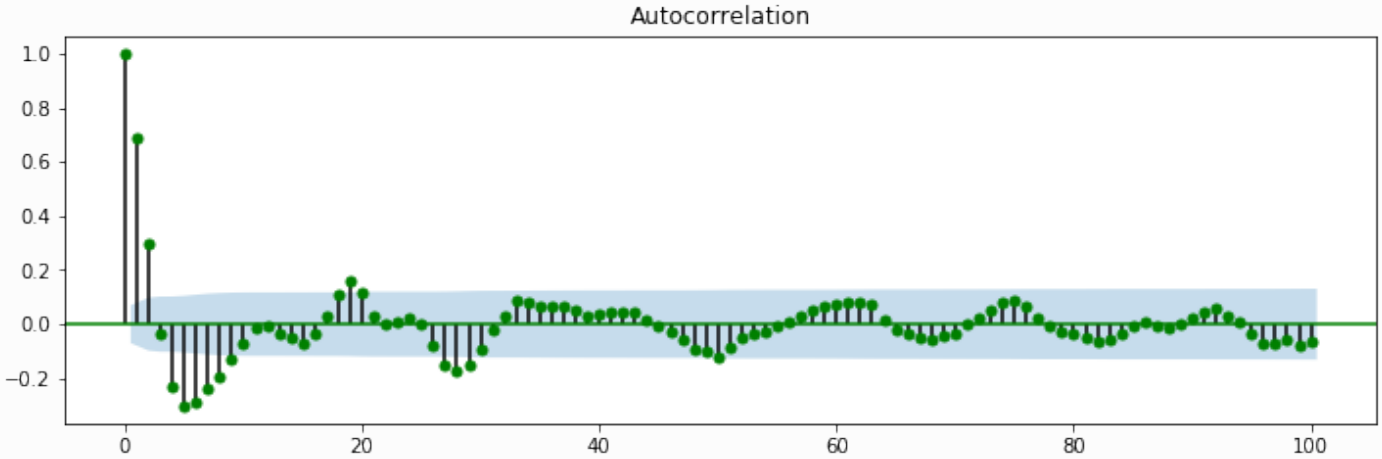
```
fig = decomposition.plot()
```

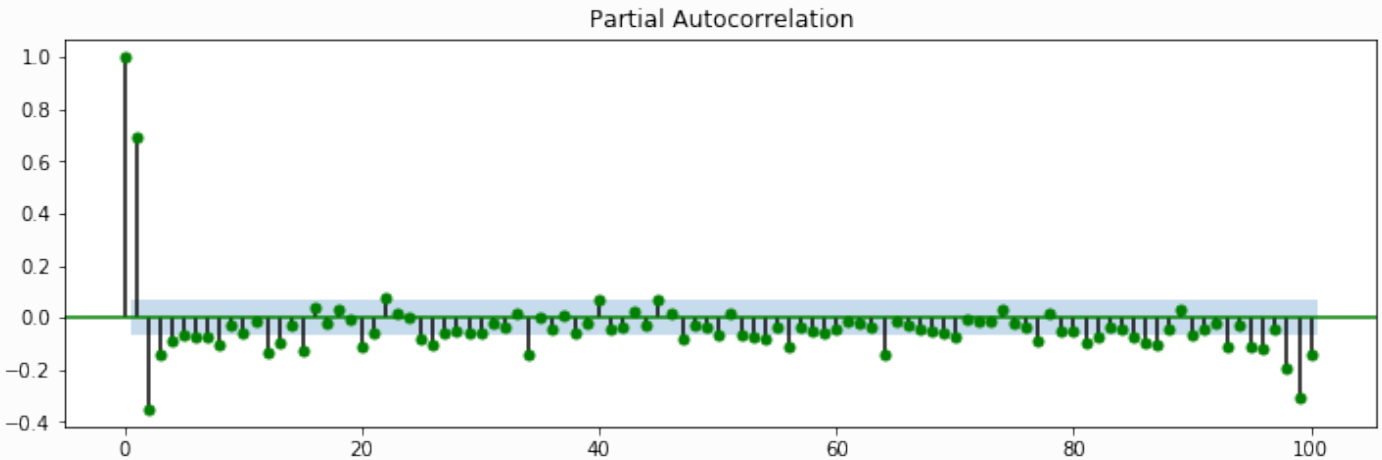



```
trend = trend.fillna(0)
seasonal = seasonal.fillna(0)
residual = residual.fillna(0)
auto_corr(trend, 100)
auto_corr(residual, 100)
```


Autocorrelation
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
0
20
40
60
80
100


Partial Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
20
40
60
80
100


Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
20
40
60
80
100


Partial Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
0
20
40
60
80
100

AIC 准则全称是最小化信息量准则（Akaike Information Criterion）： ，其中 L 表示模型的极大似然函数， K 表示模型参数个数。 AIC 准则存在一定的不足。当样本容量很大时，在 AIC 准则中拟合误差提供的信息就要受到样本容量的放大，而参数个数的惩罚因子却和样本容量没关系（一直是2），因此当样本容量很大时，使用 AIC 准则的模型不收敛于真实模型，它通常比真实模型所含的未知参数个数要多。BIC（Bayesian InformationCriterion）贝叶斯信息准则弥补了 AIC 的不足： ，其中 n 表示样本容量。 显然，这两个评价指标越小越好。我们通过网格搜索，确定 AIC、BIC 最优的模型（p、q）

```
trend = trend.fillna(0)
seasonal = seasonal.fillna(0)
residual = residual.fillna(0)
trend_evaluate = sm.tsa.arma_order_select_ic(trend, ic=['aic', 'bic'],
trend='nc', max_ar=4,
                                     max_ma=4)
print('trend AIC', trend_evaluate.aic_min_order)
print('trend BIC', trend_evaluate.bic_min_order)

residual_evaluate = sm.tsa.arma_order_select_ic(residual, ic=['aic', 'bic'],
trend='nc', max_ar=4,
                                                max_ma=4)
print('residual AIC', residual_evaluate.aic_min_order)
print('residual BIC', residual_evaluate.bic_min_order)
```

trend AIC (4, 3) trend BIC (4, 3) residual AIC (3, 1) residual BIC (3, 1) 从评价准则的结果看（这里采用 AIC 结果）： （1）对趋势序列，p = 4， q = 3 （2）对残差序列，p = 3， q = 1

Example 6.4

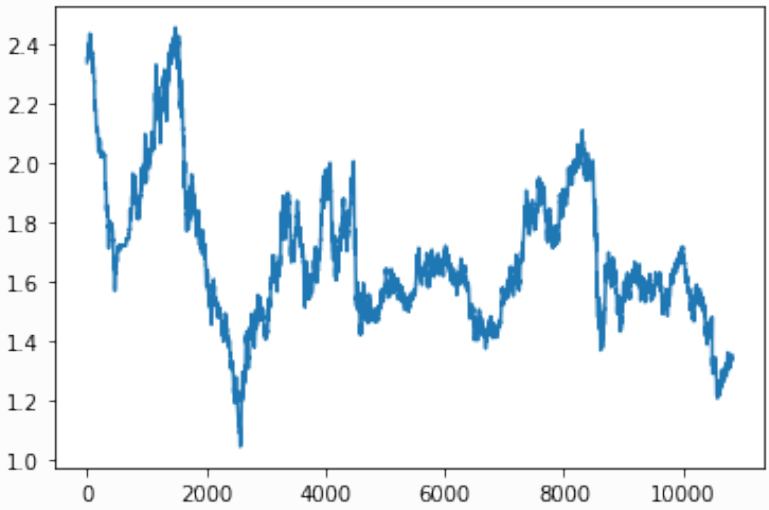
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data = pd.read_csv("/root/experiment01/dataset/dollar.csv")

plt.plot(data['dollar'])
data
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	time	dollar

0	1	2.3359
1	2	2.3414
2	3	2.3518
3	4	2.3552
4	5	2.3485
...	...	...
10814	10815	1.3354
10815	10816	1.3387
10816	10817	1.3399
10817	10818	1.3436
10818	10819	1.3510

10819 rows × 2 columns




```
data['diff_1'] = data['dollar'].diff(1)
from statsmodels.tsa.seasonal import seasonal_decompose
import statsmodels.api as sm
ts = data['dollar']
decomposition = seasonal_decompose(ts)
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid
```

```
train = ts[0:10000]
test = ts[10000:]
#按天采样
train = train.resample('D').mean()
test = test.resample('D').mean()
```

Example 6.5

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
data = pd.read_csv("/root/experiment01/dataset/bj_series_c.csv")
```

```
data['diff'] = data['bi_series'].diff(1)
data.plot(subplots=True,figsize=(15,5))
```

```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7ffbb88dc390>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7ffbb8903c50>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7ffbb785d790>],
      dtype=object)
```

```
from statsmodels.tsa.seasonal import seasonal_decompose
import statsmodels.api as sm
ts = data['bi_series']
decomposition = seasonal_decompose(ts)
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid
def autocorrelation(timeseries, lags):
    fig = plt.figure(figsize=(12, 8))
    ax1 = fig.add_subplot(211)
    sm.graphics.tsa.plot_acf(timeseries, lags=lags, ax=ax1)
    ax2 = fig.add_subplot(212)
    sm.graphics.tsa.plot_pacf(timeseries, lags=lags, ax=ax2)
    plt.show()

diff = ts.diff(1)
diff = ts.fillna(0)
autocorrelation(diff, 100)
```

第七章 单变量模型预测

ARIMA模型全称为自回归移动平均模型(Autoregressive Integrated Moving Average Model,简记ARIMA)，是由博克思(Box)和詹金斯(Jenkins)于70年代初提出的一著名时间序列预测方法，所以又称为box-jenkins模型、博克思-詹金斯法。其中ARIMA（p，d，q）称为差分自回归移动平均模型，AR是自回归, p为自回归项; MA为移动平均，q为移动平均项数，d为时间序列成为平稳时所做的差分次数。

ARIMA步骤

- 根据时间序列的散点图、自相关函数和偏自相关函数图以ADF单位根检验其方差、趋势及其季节性变化规律，对序列的平稳性进行识别。一般来讲，经济运行的时间序列都不是平稳序列。
- 对非平稳序列进行平稳化处理。如果数据序列是非平稳的，并存在一定的增长或下降趋势，则需要对数据进行差分处理，如果数据存在异方差，则需对数据进行技术处理，直到处理后的数据的自相关函数值和偏相关函数值无显著地异于零。
- 根据时间序列模型的识别规则，建立相应的模型。若平稳序列的偏相关函数是截尾的，而自相关函数是拖尾的，可断定序列适合AR模型；若平稳序列的偏相关函数是拖尾的，而自相关函数是截尾的，则可断定序列适合MA模型；若平稳序列的偏相关函数和自相关函数均是拖尾的，则序列适合ARMA模型。
- 进行参数估计，检验是否具有统计意义。
- 进行假设检验，诊断残差序列是否为白噪声。
- 利用已通过检验的模型进行预测分析。

Example 7.1

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
data = pd.read_excel("/root/experiment01/rs1.xlsx",index_col=0)
rs=np.array(data["rs"])
r20=np.array(data["r20"])
spread = r20 - rs
```

```
arma_mod30 = sm.tsa.statespace.SARIMAX(spread, order=(0,1,1),
trend='c').fit(disp=False)
print(arma_mod30.params)
```

```
[-0.00182116  0.20425197  0.15789938]
```

```
print(arma_mod30.aic, arma_mod30.bic, arma_mod30.hqic)
```

```
784.8460037926232 798.9994178776942 790.274572305871
```

```
predict = arma_mod30.predict(start=700,end=710, dynamic=True)
```

Example 7.3

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data = pd.read_csv("/root/experiment01/dataset/kefalonia.csv")
```

```
plt.figure(figsize=(15,5))
data['kefalonia'].plot(grid=True,color="green")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ffbb7f13d10>
```

```
from statsmodels.tsa.seasonal import seasonal_decompose
df = pd.DataFrame(data)
df.index = pd.DatetimeIndex(start=df['date']
[0],periods=len(df['date']),freq='MS')  #生成日期索引
ts = df['kefalonia']  # 生成pd.Series对象
```

```
/root/anaconda3/envs/jupyter_notebook/lib/python3.7/site-
packages/ipykernel_launcher.py:3: FutureWarning: Creating a DatetimeIndex by
passing range endpoints is deprecated.  Use `pandas.date_range` instead.
  This is separate from the ipykernel package so we can avoid doing imports
until
```

```
decomposition = seasonal_decompose(ts)
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid
from statsmodels.tsa.stattools import adfuller   #ADF检验
def stationarity_test(y):
    rolmean = ts.rolling(window=12).mean()
```

```
    rolstd = ts.rolling(window=12).std()
    fig = plt.figure(figsize=(12, 8))
    orig = plt.plot(y, color='blue',label='Original')
    mean = plt.plot(rolmean, color='green', label='Rolling Mean')
    std = plt.plot(rolstd, color='orange', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show()

    print('Results of ADF Test:')
    dftest = adfuller(y, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags
Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print(dfoutput)
stationarity_test(ts)
```

```
Results of ADF Test:
Test Statistic                  -2.585620
p-value                          0.096006
#Lags Used                      11.000000
Number of Observations Used    168.000000
Critical Value (1%)             -3.469886
Critical Value (5%)             -2.878903
Critical Value (10%)            -2.576027
dtype: float64
```

由p值看出并不平稳

```
df['first_difference'] = ts - ts.shift(1)
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(df.first_difference.iloc[13:], lags=100, ax=ax1)
#从13开始是因为差分时window是12
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(df.first_difference.iloc[13:], lags=100, ax=ax2)
```

```
/root/anaconda3/envs/jupyter_notebook/lib/python3.7/site-
packages/statsmodels/regression/linear_model.py:1358: RuntimeWarning: invalid
value encountered in sqrt
  return rho, np.sqrt(sigmasq)
```

```
mod = sm.tsa.statespace.SARIMAX(ts, trend='n', order=(1,1,1), seasonal_order=
(0,1,1,12))
results = mod.fit()
print(results.summary())
```

```
                                 Statespace Model Results

==========================================================================================
==========
Dep. Variable:                            kefalonia   No. Observations:
       180
Model:             SARIMAX(1, 1, 1)x(0, 1, 1, 12)   Log Likelihood
-895.534
Date:                            Sat, 15 Aug 2020   AIC
  1799.068
Time:                                    16:26:25   BIC
  1811.540
Sample:                                01-01-2003   HQIC
1804.130
                                     - 12-01-2017

Covariance Type:                               opg

==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.1083      0.063      1.718      0.086      -0.015       0.232
ma.L1         -0.9495      0.025    -38.650      0.000      -0.998      -0.901
ma.S.L12      -0.6524      0.059    -11.109      0.000      -0.767      -0.537
sigma2      2516.0117    221.363     11.366      0.000    2082.148    2949.876
==============================================================================
===
Ljung-Box (Q):                       33.20   Jarque-Bera (JB):
12.82
Prob(Q):                              0.77   Prob(JB):
0.00
Heteroskedasticity (H):               1.68   Skew:
0.01
Prob(H) (two-sided):                  0.06   Kurtosis:
4.36
==========================================================================================
===

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

```
df['forecast'] = results.predict(start = 100, end= 180, dynamic= True)
df[['kefalonia', 'forecast']].plot(figsize=(12, 8),color=["green","blue"])
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ffbb88943d0>
```

第八章 非观测的组件模型 信号提取和滤波器

非观测的组件模型

差异平稳时间序列可以分解为随机的非平稳趋势，或信号，分量和平稳噪声，或不规则分量：

$x_t = z_t + u_t$

不可观测的组件模型也与HP滤波器有关，该过滤器是通过将噪声分量$u_t = x_t - z_t$最小化而得出的，该最小化问题可以转变为函数最小化的问题：

$$\sum_{t=1}^{t} u_t^2 + \lambda \sum_{t=1}^{T} ((z_{t+1} - z_t) - (z_t - z_{t-1}))^2$$

其中λ是拉格朗日乘子，可以解释为平滑度参数。λ值越高，趋势越平滑，因此在极限情况下，随着λ~N，z_t逐渐趋于线性。 一阶情况下的条件为：

$$0 = -2(x_t - z_t) + 2\lambda((z_t - z_{t-1}) - (z_{t-1} - z_{t-2})) - 4\lambda((z_{t+1} - z_t) - (z_t - z_{t-1})) + 2\lambda((z_{t+2} - z_{t+1}) - (z_{t+1} - z_t))$$

上式可以写为：

$$x_t = (1 + \lambda(1 - B)^2(1 - B^{-1})^2)^{-1} z_t$$

所以H-P滤波器的估计值为：

$$\hat{z}_t(\lambda) = (1 + \lambda(1 - B)^2(1 - B^{-1})^2)^{-1} x_t$$

```
import numpy as np
import pandas as pd
import matplotlib.dates as mdate
import matplotlib.pyplot as plt
```

```
def d_matrix(n):
    d = np.zeros((n-1, n))
    d[:, 1:] = np.eye(n-1)
    d[:, :-1] -= np.eye(n-1)
    return d
def HP_filter(xt, lamb=10):
    n = len(xt)
    d_1 = d_matrix(n)
    d_2 = d_matrix(n-1)
    d = np.dot(d_2,d_1)
    zt = np.linalg.inv((np.eye(n) + lamb * d.T @ d)) @ xt
    return zt
```

Example 8.2

```
data = pd.read_csv('data/global_temps.csv')#读取数据
t = pd.to_datetime(data.iloc[:, 0], format='%Y-%m-%d')
xt = data.iloc[:, 1]
```

```
filter1 = HP_filter(xt, 14400)#使lambda分别取值14400，129600和500000
filter2 = HP_filter(xt, 129600)
filter3 = HP_filter(xt, 500000)
```

将处理好的数据可视化，并进行对比

```
plt.figure(figsize=[10, 7])
plt.subplot(2, 1, 1)
plt.plot(t, xt,color="green")
plt.title('Observed')
plt.subplot(2, 1, 2)
plt.plot(t, filter1, '--b',color="orange")
plt.plot(t, filter2, '--g')
plt.plot(t, filter3, 'r',color="blue")
plt.title('HP_filter')
plt.legend(['lamb=14400', 'lamb=129600', 'lamb=500000'])
plt.show()
```

Example 8.3

```
data = pd.read_csv('data/gdp.csv')
t = np.arange(1270, 1914, 1)
y = data.iloc[:, 1]
x_t = np.log(y)
```

```
#分别对原数据和log处理过的数据进行H-P滤波处理，进而计算年增长率
la = 10000
filter1 = HP_filter(x_t, la)
filter2 = HP_filter(y, la)
annual = pd.DataFrame(filter2).pct_change()
```

将计算出数据画图，得到以下结果

第一张图为经过log处理过后的数据,第二张图为H-P处理过的原数据的年增长率

```
plt.figure(figsize=[12, 8])
plt.subplot(2, 1, 1)
plt.plot(t, x_t,t, filter1, '--')
plt.legend(['Observed', 'hpfilter'])
plt.title('hpfilter')
plt.subplot(2, 1, 2)
plt.plot(t[:], annual,color="green")
plt.axhline(y=0, color='r', linestyle='-')
plt.title('yearly growth rate')
plt.show()
```

第九章 季节性时间序列与指数平滑

Figure 9.1

```
def first_diff(x_t):
    y = np.array([])
    for i in range(len(x_t)-1):
        y = np.append(y, x_t[i+1] - x_t[i])
    return y


def autocorrelation_function(k, x_t):
    tem = 0
    avg = np.mean(x_t)
```

```
    var = np.var(x_t)
    for i in range(k, len(x_t)):
        tem += (x_t[i] - avg)*(x_t[i-k] - avg)
    r = tem/(len(x_t)*var)
    return r
```

```
data = pd.read_csv('data/beer.csv')
t = pd.to_datetime(data.iloc[:, 0], format='%Y-%m-%d')
x_t = data.iloc[:, 1]
```

计算当前项值k从1至24时，对应SACF的值：

```
diff = first_diff(x_t)
r = np.ones(24)
for k in range(1, 25):
    r[k-1] = autocorrelation_function(k, diff)
```

```
plt.figure(figsize=[10, 7])
plt.subplot(2, 1, 1)
plt.plot(t[1:], diff,color="green")
plt.subplot(2, 1, 2)
plt.grid(True)
plt.stem(r, use_line_collection=True)
plt.show()
#第一张图为一次差分后数据，第二幅图为k值与SACF值的关系。
```

Figure 9.2

```
data = pd.read_csv('data/rainfall.csv')
t = pd.to_datetime(data.iloc[:, 0], format='%Y-%m-%d')
x_t = data.iloc[:, 1]
```

计算当前项值k从1至48时，对应SACF的值：

```
x_t = np.sqrt(x_t)
r = np.ones(48)
for k in range(1, 49):
    r[k-1] = autocorrelation_function(k, x_t)
```

将数据以画图的形式展现：

```
plt.figure(figsize=[15, 8])
plt.subplot(2, 1, 1)
plt.plot(t, x_t,color="green")
plt.subplot(2, 1, 2)
plt.grid(True)
plt.stem(r, use_line_collection=True)
plt.show()
#第一张图为平方根后后的数据，第二张图为k值与SACF值的关系。
```

Example 9.1

首先将数据进行次方转化（power transformations），再逐个月份提取数据的季节性特征。

```
#首先运行次方转换函数：
def power_transformations(x_t, lam):
    if lam == 0:
        return np.log(x_t)
    else:
        return (np.power(x_t, lam)-1)/lam
```

```
data = pd.read_csv('data/rainfall.csv')
temp = data.iloc[:, 0]
t = pd.to_datetime(data.iloc[:, 0], format='%Y-%m-%d')
xt = data.iloc[:, 1]
```

```
#将数据进行次方转换，并分别计算12个月的均值
transform_1 = power_transformations(xt, 0.7)
for j in range(12):
    locals()['rain'+str(temp[j][5:7])] = np.array([])
    meanlist = np.ones([1, 12])
for i in range(len(temp)):
    locals()['rain'+str(temp[i][5:7])] = np.append(locals()['rain'+str(temp[i]
[5:7])], transform_1[i])
for j in range(12):
    meanlist[0][j] = np.mean(locals()['rain'+str(temp[j][5:7])])
```

```
meanlist = np.power(meanlist, 2)
print(meanlist)
#这便是12个月降雨量的季节性特征
```

```
[[ 951.3777867   616.10418091  597.47992825  525.84936653  587.14745725
   590.04491129  747.99771629  885.81024871  754.0871994  1072.93158257
  1092.62646744 1073.01702004]]
```

Figure 9.3

本图选取了beer数据，并隔四项进行差分，计算SACF。

```
data = pd.read_csv('data/beer.csv')
t = pd.to_datetime(data.iloc[:, 0], format='%Y-%m-%d')
x_t = np.diff(data.iloc[:, 1])
```

```
#对数据进行4项差分处理：
diff = np.ones(len(x_t)-4)
for i in range(4, len(x_t)):
    diff[i-4] = x_t[i] - x_t[i-4]
```

```
#计算当前项值k从1至24时，对应SACF的值：
r = np.ones(24)
for k in range(1, 25):
    r[k-1] = autocorrelation_function(k, diff)
```

```
plt.figure(figsize=[15, 8])
plt.subplot(2, 1, 1)
plt.plot(t[1:], x_t,color="green")
plt.subplot(2, 1, 2)
plt.grid(True)
plt.stem(r, use_line_collection=True)
plt.show()
```

Example 9.2

首先，可以得到数据的ARIMA模型为：

$$\nabla\nabla_4 x_t = (1-0.694B)(1-0.604B^4)a_t = (1-0.694B-0.604B^4+0.419B^5)a_t \qquad \hat{\sigma} = 271.9$$

根据此模型，我们可以对数据进行季节性差分，并进一步通过模型进行拟合。

```
def MA( phi, sigma,con, n):
    k = len(phi)
    xt = np.zeros(n)
    a = np.random.normal(0, sigma, n)
    for i in range(k):
        xt[i] = a[i]
    for i in range(1, n):
```

```
        for t in range(k):
            if t == 0:
                temp = a[i]
            else:
                temp -= phi[t] * a[i - t]
        xt[i] = temp + con
    return xt
```

```
np.random.seed(1)
data = pd.read_csv('data/beer.csv')
t = pd.to_datetime(data.iloc[:, 0], format='%Y-%m-%d')
y = np.array(data.iloc[:, 1])
xt = np.diff(data.iloc[:, 1])
```

```
ma = MA([0.694, 0, 0, 0.604, -0.419], 271.9, 0, len(xt)+9)
fdiff = np.ones(len(xt)+13)
fxt = np.ones(13)
fdiff[:4] = xt[:4]
for i in range(4, len(fdiff)):
    fdiff[i] = ma[i-4] + fdiff[i-4]
fxt[0] = y[-1]+fdiff[-13]
for j in range(1, 13):
    fxt[j] = fdiff[-13+j] + fxt[j-1]
```

```
t = pd.date_range('2017-10-01', periods=13, freq='3M')
plt.figure(figsize=[10, 5])
plt.plot(t, fxt,color="green")
plt.gca().xaxis.set_major_formatter(mdate.DateFormatter('%Y-%m'))
plt.vlines(x='2017-12-31', ymin=4500, ymax=9500, colors='b', linestyles='--')
plt.vlines(x='2018-12-31', ymin=4500, ymax=9500, colors='b', linestyles='--')
plt.vlines(x='2019-12-31', ymin=4500, ymax=9500, colors='b', linestyles='--')
plt.title('Airline model forecasts of beer sales')
plt.show()
```

Example 9.3

当时间序列无明显的趋势变化，可用**一次指数平滑**预测。此时z_t为：

$$z_t = \alpha x_t + (1-\alpha) z_{t-1}$$

```
def single_exponential_smoothing(xt, alpha):
    zt = np.ones(len(xt))
    zt[0] = xt[0]
    for i in range(1, len(xt)):
        zt[i] = alpha*xt[i]+(1-alpha)*xt[i-1]
    return zt
```

二次指数平滑是指在一次指数平滑的基础上，加上了趋势τ_t，使原式变为$x_t = z_t + u_t + \tau_t$，是一次指数平滑的再平滑。它适用于具线性趋势的时间序列。其预测公式为：

$$z_t = \gamma x_t + (1-\gamma) z_{t-1}$$
$$\tau_t = \gamma(z_t - z_{t-1}) + (1-\gamma)\tau_{t-1}$$

```
def double_exponential_smoothing(xt, gamma):
    zt = np.ones(len(xt))
    tao = np.ones(len(xt))
    zt[0], tao[0] = 0, 0
    zt[1] = gamma*xt[1]+(1-gamma)*zt[0]
    tao[1] = gamma*(zt[1]-zt[0])+(1-gamma)*tao[0]
    for i in range(2, len(xt)):
        zt[i] = gamma*xt[i]+(1-gamma)*zt[i-1]
        tao[i] = gamma * (zt[i] - zt[i-1]) + (1 - gamma) * tao[i-1]
    y = zt + tao
    return y
```

```
data = pd.read_csv('data/global_temps.csv')
t = pd.to_datetime(data.iloc[:, 0], format='%Y-%m-%d')[1:]
xt = np.diff(data.iloc[:, 1])
```

```
# 分别计算一次指数平滑lambda=0.45时，二次指数平滑lambda=0.196时的预测值：
ssmo = single_exponential_smoothing(xt, 0.45)
dsmo = double_exponential_smoothing(xt, 0.196)
```

此处使用RMSE为评价标准

```
#此处使用RMSE
RMSE1 = np.sqrt(np.sum(np.power(xt[:]-ssmo[:], 2))/len(xt))
RMSE2 = np.sqrt(np.sum(np.power(xt[:]-dsmo[:], 2))/len(xt))
print(' single exp RMSE = ', RMSE1, '\n', 'double exp RMSE = ', RMSE2)
```

```
 single exp RMSE =  0.12555221411421877
 double exp RMSE =  0.11876672136164598
```

```
plt.figure(figsize=[15, 8])
plt.subplot(2, 1, 1)
plt.plot(t, xt,color="orange")
plt.plot(t, ssmo,color="green")
plt.legend(['observed', 'ssmo'])
plt.title('single exponential smoothing')
plt.subplot(2, 1, 2)
plt.plot(t, xt, color="orange")
plt.plot(t, dsmo,color="green")
plt.legend(['observed', 'dsmo'])
plt.title('double exponential smoothing')
plt.show()
```

三次指数平滑（Holt-Winters模型）是在二次指数平滑的基础上，增加了季节分量s_t，使原式变为$x_t = z_t + s_t + \tau_t$。

Holt-Winters模型分为加法模型和乘法模型，加法模型的预测公式为：

$$
\begin{aligned}
z_t &= \alpha(x_t - s_{t-m}) + (1-\beta)(z_{t-1} + \tau_{t-1}) \\
\tau_t &= \beta(z_t - z_{t-1}) + (1-\beta)\tau_{t-1} \\
s_t &= \delta(x_t - z_t) + (1-\delta)s_{t-m}
\end{aligned}
$$

乘法模型的预测公式为：

$$
\begin{aligned}
z_t &= \alpha(\frac{x_t}{s_{t-m}}) + (1-\beta)(z_{t-1} + \tau_{t-1}) \\
\tau_t &= \beta(z_t - z_{t-1}) + (1-\beta)\tau_{t-1} \\
s_t &= \delta(\frac{x_t}{z_t}) + (1-\delta)s_{t-m}
\end{aligned}
$$

Example 9.4

```
def returndiff(x0, diff):
    x_t = np.ones(len(diff)+1)
    x_t[0] = x0
    for i in range(1, len(x_t)):
        x_t[i] = diff[i-1]+x_t[i-1]
    return xt


def add_holt(xt, alpha, beta, delta, m, h):
    n = len(xt)
    zt, tao, st = [np.ones(n+h) for _ in range(3)]
    zt[0], tao[0] = xt[0], xt[0]
    st[-m:m] = 0
    for i in range(1, n):
        zt[i] = alpha*(xt[i]-st[i-m])+(1-beta)*(zt[i-1]+tao[i-1])
        tao[i] = beta*(zt[i]-zt[i-1])+(1-beta)*tao[i-1]
        st[i] = delta*(xt[i]-zt[i])+(1-delta)*st[i-m]
    for j in range(h):
```

```
        zt[n+j] = (1-beta)*(zt[j-1]+tao[j-1])
        tao[n+j] = beta*(zt[n+j]-zt[n+j-1])+(1-beta)*tao[n+j-1]
        st[n+j] = (1-delta)*st[n+j-m]
    y = zt+st+tao
    return y


def mul_holt(xt, alpha, beta, delta, m, h):
    n = len(xt)
    zt, tao, st = [np.ones(n+h) for _ in range(3)]
    zt[0], tao[0] = xt[0], 0
    st[-m:m] = delta
    for i in range(1, n):
        zt[i] = alpha*(xt[i]/st[i-m])+(1-beta)*(zt[i-1]+tao[i-1])
        tao[i] = beta*(zt[i]-zt[i-1])+(1-beta)*tao[i-1]
        st[i] = delta*(xt[i]/zt[i])+(1-delta)*st[i-m]
    for j in range(h):
        zt[n+j] = (1-beta)*(zt[j-1]+tao[j-1])
        tao[n+j] = beta*(zt[n+j]-zt[n+j-1])+(1-beta)*tao[n+j-1]
        st[n+j] = (1-delta)*st[n+j-m]
    y = zt*st+tao
    return y
```

```
np.random.seed(1)
data = pd.read_csv('data/beer.csv')
t = pd.to_datetime(data.iloc[:, 0], format='%Y-%m-%d')
y = data.iloc[:, 1]
xt = np.diff(data.iloc[:, 1])
```

```
h = 14
addho = add_holt(xt, 0.1, 0.79, 0.33, 4, h)
mulho = mul_holt(xt, 0.1, 0.81, 0, 4, h)
addho = returndiff(y[0], addho)
mulho = returndiff(y[0], mulho)
```

```
tt = pd.date_range('2017-10-01', periods=13, freq='3M')
plt.figure(figsize=[15, 15])
plt.subplot(3, 1, 1)
plt.plot(t, y,color="green")
plt.legend(['observed'])
plt.title('Obsrved')
plt.subplot(3, 1, 2)
plt.plot(t, addho[:-h] ,color="green")
plt.plot( t, mulho[:-h],color="blue")
plt.legend(['Holt add', 'Holt mul'])
plt.title('Compare')
plt.subplot(3, 1, 3)
plt.plot(tt, addho[-13:],color="green")
plt.plot(tt, mulho[-13:],color="blue")
plt.legend(['Holt add', 'Holt mul'])
```

```
plt.gca().xaxis.set_major_formatter(mdate.DateFormatter('%Y'))
plt.vlines(x='2017-12-31', ymin=5000, ymax=9200, colors='y', linestyles='--')
plt.vlines(x='2018-12-31', ymin=5000, ymax=9200, colors='y', linestyles='--')
plt.vlines(x='2019-12-31', ymin=5000, ymax=9200, colors='y', linestyles='--')
plt.title('Forecast')
plt.show()
```

第十章 自回归条件异方差模型

自回归条件异方差模型(ARCH)

波动率的特征

对于金融时间序列，波动率往往具有以下特征：

（1）存在**波动率聚集**现象。 即波动率在一段时间上高，一段时间上低。

（2）波动率以连续时间变化，很少发生跳跃

（3）波动率不会发散到无穷，波动率往往是**平稳**的

（4）波动率对价格大幅上升和大幅下降的反应是不同的，这个现象为**杠杆效应**

ARCH的原理

在传统计量经济学模型中，干扰项的方差被假设为常数。但是许多经济时间序列呈现出波动的集聚性，在这种情况下假设方差为常数是不恰当的。

ARCH模型将当前一切可利用信息作为条件，并采用某种自回归形式来刻划方差的变异，对于一个时间序列而言，在不同时刻可利用的信息不同，而相应的条件方差也不同，利用ARCH 模型，可以刻划出随时间而变异的条件方差。

ARCH模型

1. 时间序列的扰动 $\{a_t\}$ 是序列不相关的，但是不独立。
2. $\{a_t\}$的不独立性可以用其延迟值的简单二次函数来描述。具体而言，一个ARCH(m)模型为：

$$a_t = \sigma_t \varepsilon_t \sigma_t^2 = \alpha_0 + \alpha_1 a_{t-1}^2 + \cdots + \alpha_m a_{t-m}^2 \alpha_0 > 0; \forall i > 0, \alpha_i \geqslant 0$$

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.tsa.api as smt
import statsmodels.api as sm
```

```
import pandas as pd
import numpy as np

import scipy.stats as scs
from arch import arch_model
#画图

import tushare as ts
import matplotlib as mpl

#正常显示画图时出现的中文和负号
from pylab import mpl
mpl.rcParams['font.sans-serif']=['SimHei']
mpl.rcParams['axes.unicode_minus']=False


# 使用tushare获取沪深300交易数据
token='此处输入个人token接口号'
pro=ts.pro_api(token)
df=pro.index_daily(ts_code='000300.SH')
df.index=pd.to_datetime(df.trade_date)
del df.index.name
df=df.sort_index()
df['ret']=np.log(df.close/df.close.shift(1))
#df.head()
ts_plot(df.ret.dropna(),lags=30,title='沪深300收益率')
```

ARCH模型的建模步骤：

1. 检验收益序列是否平稳，根据自相关性建立合适的均值方程，如ARMA模型，描述收益率如何随时间变化，根据拟合的模型和实际值，得到残差序列。
2. 对拟合的均值方程得到的残差序列进行ARCH效应检验，即检验收益率围绕均值的偏差是否时大时小。检验序列是否具有ARCH效应的方法有两种：Ljung-Box检验和LM检验。
3. 若ARCH效应在统计上显著，则需要再设定一个波动率模型来刻画波动率的动态变化。
4. 对均值方差和波动率方差进行联合估计，即假设实际数据服从前面设定的均值方差和波动率方差后，对均值方差和波动率方差中的参数进行估计，并得到估计的误差。

5. 对拟合的模型进行检验。如果估计结果（残差项）不满足模型本身的假设，则模型的可用性较差。

```
# 模拟ARCH时间序列，对沪深300收益率的ARCH效应进行统计检验
np.random.seed(2)
a0 = 2
a1 = .5
y = w = np.random.normal(size=1000)
Y = np.empty_like(y)
for t in range(1,len(y)):
    Y[t] = w[t] * np.sqrt((a0 + a1*y[t-1]**2))
ts_plot(Y, lags=30,title='模拟ARCH')

def ret_plot(ts, title=''):
    ts1=ts**2
    ts2=np.abs(ts)
    with plt.style.context('ggplot'):
        fig = plt.figure(figsize=(12, 6))
        layout = (2, 1)
        ts1_ax = plt.subplot2grid(layout, (0, 0), colspan=2)
        ts2_ax = plt.subplot2grid(layout, (1, 0))
        ts1.plot(ax=ts1_ax)
        ts1_ax.set_title(title+'日收益率平方')
        ts2.plot(ax=ts2_ax)
        ts2_ax.set_title(title+'日收益率绝对值')
        plt.tight_layout()
    return

ret_plot(df.ret.dropna(), title='沪深300')

# 使用Ljung-Box统计量对收益率平方的自相关性进行统计检验
def whitenoise_test(ts):
    '''计算box pierce 和 box ljung统计量'''
    from statsmodels.stats.diagnostic import acorr_ljungbox
    q,p=acorr_ljungbox(ts)
    with plt.style.context('ggplot'):
        fig = plt.figure(figsize=(10, 4))
        axes = fig.subplots(1,2)
        axes[0].plot(q, label='Q统计量')
        axes[0].set_ylabel('Q')
        axes[1].plot(p, label='p值')
        axes[1].set_ylabel('P')
        axes[0].legend()
        axes[1].legend()
        plt.tight_layout()
    return

ret=df.ret.dropna()
whitenoise_test(ret**2)
```

实际应用中，GARCH(1,1)和GARCH(2,1)一般可以满足对自回归条件异方差的描述。下面使用Python对GARCH(1,1)模型进行模拟和估计

GARCH模型

虽然ARCH模型简单，但为了充分刻画收益率的波动率过程，往往需要很多参数，例如上面用到ARCH(4)模型，有时会有更高的ARCH(m)模型。因此，Bollerslev(1986)年提出了一个推广形式，称为**广义的ARCH模型（GARCH）**。另$a_t = r_t - \mu_t$，为t时刻的信息。若at满足下式：

$$a_t = \sigma_t \varepsilon_t \sigma_t^2 = \alpha_0 + \sum_{i=1}^{m} \alpha_i a_{t-i}^2 + \sum_{j=1}^{s} \beta_j \sigma_{t-j}^2 \alpha_0 > 0; \forall i > 0, \alpha_i \geqslant 0, \beta_i \geqslant 0, (\alpha_i + \beta_i) < 1$$

其中，ε_t为**均值为0，方差为1的独立同分布（iid）随机变量序列**。通常假定其服从**标准正态分布**或**标准化学生-t分布**。σ_t^2为条件异方差。则称a_t服从GARCH(m,s)模型。

GARCH模型建立

与之前的ARCH模型建立过程类似，不过GARCH(m,s)的定阶较难，一般使用低阶模型如GARCH(1,1),GARCH(2,1),GARCH(1,2)等。实际应用中，GARCH(1,1)和GARCH(2,1)一般可以满足对自回归条件异方差的描述。下面使用Python对GARCH(1,1)模型进行模拟和估计

Example 10.2

```
# 模拟GARCH(1, 1) 过程
np.random.seed(1)
a0 = 0.2
a1 = 0.5
b1 = 0.3
n = 10000
w = np.random.normal(size=n)
garch = np.zeros_like(w)
sigsq = np.zeros_like(w)
for i in range(1, n):
    sigsq[i] = a0 + a1*(garch[i-1]**2) + b1*sigsq[i-1]
    garch[i] = w[i] * np.sqrt(sigsq[i])
_ = ts_plot(garch, lags=30,title='模拟GARCH')
# 使用模拟的数据进行 GARCH(1, 1) 模型拟合
#arch_model默认建立GARCH (1,1) 模型
am = arch_model(garch)
res = am.fit(update_freq=0)

print(res.summary())

res.resid.plot(figsize=(12,5))
plt.title('沪深300收益率拟合GARCH(1,1)残差',size=15)
plt.show()
res.conditional_volatility.plot(figsize=(12,5),color='r')
plt.title('沪深300收益率条件方差',size=15)
plt.show()
```

第十一章 非线性随机过程

双线性模型（BILINEAR MODEL）

称随机序列$\{x_i\}$ 服从双线性模型，如果：

$x_i = \sum_{j=1}^{p} \varphi_j x_{i-j} + \sum_{j=0}^{q} \theta_j a_{i-j} + \sum_{k=0}^{Q} \sum_{l=1}^{P} \beta_{kl} x_{i-l} a_{i-k}$

其中a_i为i.i.d.随机序列，$Ea_i = 0, Ea_i^2 = \sigma^2$，当$\beta_{kl} = 0 (k = 0, 1, \ldots, Q, l = 1, \ldots, P)$

则(1)式成为ARMA（p，q）模型，因此双线性模型是线性模型的直接推广。

门限平滑移动自回归模型（THRESHOLD AND SMOOTH TRANSITION AUTOREGRESSIONS）

门限自回归模型(threshold autoregressive model)，又称阈模型，简称TAR模型，它是一种非线性模型。门限自回归模型的模型形式与分段线性模型形式非常相似。门限或阈（Threshold）的概念是指高于或低于门限值（阈值）的自回归过程不同，因此，可以捕捉到一个过程下降和上升模式中的非对称性。

思路及定义

TAR的基本思路：在观测时序$\{x_i\}$ 的取值范围内引入L-1个门限值$r_j = (j = 1, 2, \ldots, L-1)$ ，将该范围分成 L个区间，并根据延迟步数d将$\{x_i\}$按$\{x_{i-d}\}$值的大小分配到不同的门限区间内，再对不同区间内的x_i采用不同的自回归模型来描述，这些自回归模型的总和完成了对时序$\{x_i\}$整个非线性动态系统的描述。其一般形式为：

$x_i = \sum_{i=1}^{n_j} \varphi_i^{(j)} \alpha_i^{(j)}, r_{j-1} < x_{i-d} \le r_j, j = 1, 2, \ldots, L$

其中，$r_j = (j = 1, 2, \ldots, L-1)$为门限值，L为门限期间的个数；d为延迟步数；$\left\{\alpha_t^{(j)}\right\}$对每一固定的j是方差为$\sigma_t^2$的白噪声系列，各$\left\{\alpha_t^{(j)}\right\} (j = 1, 2, \ldots, L-1)$之间是相互独立的；$\varphi^{(j)}$为第j个门限区间自回归系数；$n_j$为第j个门限区间模型的阶数。由于TAR模型实质是分区间的AR模型，建模时沿用AR模型的参数估计方法和模型检验准则，如最小二乘法与AIC准则。其建模过程实质上是一个对d，L，
$r_j (j = 1, 2, \ldots, L-1), n_j (j = 1, 2, \ldots, L-1)$和$\varphi^{(j)} (j = 1, 2, \ldots, L-1)$ 的多维寻优问题。

模型步骤

1. 确实门限变量
2. 确定率定门限数L

3. 确定门限值
4. 确定回归系数的过程

门限平滑移动自回归模型（Threshold and Smooth Transition Autoregressions）

```
from sklearn.linear_model import LinearRegression
import pandas as pd
# 利用pandas读取csv，读取的数据为DataFrame对象
data = pd.read_csv('jl.csv')
data= data.values.copy()

# 计算互相关系数，参数为预报因子序列和滞时k
def get_regre_coef(X,Y,k):
    S_xy=0
    S_xx=0
    S_yy=0
    # 计算预报因子和预报对象的均值
    X_mean = np.mean(X)
    Y_mean = np.mean(Y)
    for i in  range(len(X)-k):
        S_xy += (X[i] - X_mean) * (Y[i+k] - Y_mean)
    for i in range(len(X)):
        S_xx += pow(X[i] - X_mean, 2)
        S_yy += pow(Y[i] - Y_mean, 2)
    return S_xy/pow(S_xx*S_yy,0.5)
#计算相关系数矩阵
def regre_coef_matrix(data):
    row=data.shape[1]#列数
    r_matrix=np.ones((1,row-2))
    # print(row)
    for i in range(1,row-1):
        r_matrix[0,i-1]=get_regre_coef(data[:,i],data[:,row-1],1)#滞时为1
    return r_matrix
r_matrix=regre_coef_matrix(data)
# print(r_matrix)
###输出###
#[[0.048979   0.07829989 0.19005705 0.27501209 0.28604638]]

#对相关系数进行排序找到相关系数最大者作为门限元
def get_menxiannum(r_matrix):
    row=r_matrix.shape[1]#列数
    for i in range(row):
        if r_matrix.max()==r_matrix[0,i]:
            return i+1
    return -1
m=get_menxiannum(r_matrix)
# print(m)
##输出##第五个因子的互相关系数最大
```

```
#根据门限元对因子序列进行排序,m为门限变量的序号
def resort_bymenxian(data,m):
    data=data.tolist()#转化为列表
    data.sort(key=lambda x: x[m])#列表按照m+1列进行排序(升序)
    data=np.array(data)
    return data
data=resort_bymenxian(data,m)#得到排序后的序列数组


def get_var(x):
    return x.std() ** 2 * x.size  # 计算总方差
#统计量F的计算,输入数据为按照门限元排序后的预报对象数据
def get_F(Y):
    col=Y.shape[0]
    FF=np.ones((1,col-1))#存储不同分割点的统计量
    V=get_var(Y)#计算总方差
    for i in range(1,col):#1到col-1
        S=get_var(Y[0:i])+get_var(Y[i:col])#计算两段的组内方差和
        F=(V-S)*(col-2)/S
        FF[0,i-1]=F
    return FF
y=data[:,data.shape[1]-1]
FF=get_F(y)
def get_index(FF,element):#获取element在一维数组FF中第一次出现的索引
    i=-1
    for item in FF.flat:
        i+=1
        if item==element:
            return i
f_index=get_index(FF,np.max(FF))#获取统计量F的最大索引
# print(data[f_index,m-1])#门限元为第五个因子，代入索引得门限值 121

def data_excision(data,f_index):
    f_index=f_index+1
    data1=data[0:f_index,:]
    data2=data[f_index:data.shape[0],:]
    return data1,data2
data1,data2=data_excision(data,f_index)
# 第一段
def get_XY(data):
    # 数组切片对变量进行赋值
    Y = data[:, data.shape[1] - 1]  # 预报对象位于最后一列
    X = data[:, 1:data.shape[1] - 1]#预报因子从第二列到倒数第二列
    return X, Y
X,Y=get_XY(data1)
regs=LinearRegression()
regs.fit(X,Y)

Y1=regs.predict(X)
# print('第二段')
X,Y=get_XY(data2)
```

```
regs.fit(X,Y)
Y2=regs.predict(X)
Y=np.column_stack((data[:,0],np.hstack((Y1,Y2)))).copy()
Y=np.column_stack((Y,data[:,data.shape[1]-1]))
Y=resort_bymenxian(Y,0)


Y=resort_bymenxian(Y,0)

plt.plot(Y[:,0],Y[:,1],'r--',Y[:,0],Y[:,2],'g')
plt.title('predicted and measured',fontsize=10)
plt.xlabel('Years',color='gray')
plt.ylabel('Average traffic in December',color='gray')
plt.legend(['Predicted values','Measured values'])
plt.show()
```

马尔可夫转换模型（MARKOV SWITCHING MODEL）

时间序列建模的最简单方法是线性自回归模型。自回归模型指定输出变量线性地取决于其自身的先前值和随机项，即假定时间序列的均值和方差在所考虑的整个时间段内保持不变，但现实数据往往很难满足这样的条件。由于某些结构上的变化，时间序列可以从一个时期到下一个时期完全改变。区制转移模型（Regime shift models，简称RSM）通过将时间序列分为不同的“状态”，来解决基本时间序列建模中的不足。

基本思路

时间序列存在于两个或多个状态，每个状态都有自己的概率分布，并且一个状态到另一个状态的转换由另一个过程或变量控制。区制转移模型有三种类型：阈值模型（Threshold models）、预测模型（Predictive models）和马尔科夫转换自回归模型（Markov switching autoregressive models）。

阈值模型观察到的变量超过阈值会触发状态转换。马尔科夫转换自回归模型（MSAM），假定状态为“隐藏状态”，并假定潜在状态的的转换遵循同质一阶马尔可夫链，而下一个状态的概率仅取决于当前状态。可以通过最大似然法来估计从一个状态到下一个状态的转移概率，通过使似然函数最大化来估计参数值。

```
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt

%matplotlib inline

df=pd.read_csv("/root/experiment01/dataset/gdp.csv")
#df.index=pd.to_datetime(df.date)
df.head()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	year	index
0	1270	0.483038
1	1271	0.477300
2	1272	0.494482
3	1273	0.465640
4	1274	0.463213

对收益率数据进行建模。此外，日收益率包含的噪声较大，将其转换为周收益率在进行建模

```
#上证综指周收益率
df_ret=df["index"].pct_change().dropna()
df_ret.plot(title='GDP ',figsize=(15,5),color="green")
plt.show()
```

平稳性检验

```
#使用arch包中的单位根检验unitroot导入ADF
from arch.unitroot import ADF
ADF(df_ret)

#模型拟合
mod = sm.tsa.MarkovRegression(df_ret.dropna(),
k_regimes=3, trend='nc', switching_variance=True)

res = mod.fit()
res.summary()
```

Markov Switching Model Results

Dep. Variable:	index	**No. Observations:**	643
Model:	MarkovRegression	**Log Likelihood**	964.142
Date:	Sat, 15 Aug 2020	**AIC**	-1910.285
Time:	10:37:19	**BIC**	-1870.090
Sample:	0	**HQIC**	-1894.686
	- 643		
Covariance Type:	approx		

Regime 0 parameters

	coef	std err	z	P>|z|	(0.025	0.975)
sigma2	0.0009	0.000	8.749	0.000	0.001	0.001

Regime 1 parameters

	coef	std err	z	P>|z|	(0.025	0.975)
sigma2	0.0020	0.000	5.912	0.000	0.001	0.003

Regime 2 parameters

	coef	std err	z	P>|z|	(0.025	0.975)
sigma2	0.0077	0.001	9.380	0.000	0.006	0.009

Regime transition parameters

	coef	std err	z	P>|z|	(0.025	0.975)
p(0->0)	0.9903	0.010	94.567	0.000	0.970	1.011
p(1->0)	0.0142	0.009	1.662	0.096	-0.003	0.031
p(2->0)	2.739e-07	nan	nan	nan	nan	nan
p(0->1)	0.0097	0.010	0.926	0.354	-0.011	0.030
p(1->1)	0.9319	0.030	31.218	0.000	0.873	0.990
p(2->1)	0.0355	0.021	1.673	0.094	-0.006	0.077

Warnings:
(1) Covariance matrix calculated using numerical (complex-step) differentiation.

对上证综指波动性状态的平滑概率进行可视化

```
fig, axes = plt.subplots(3, figsize=(12,8))
ax = axes[0]
ax.plot(res.smoothed_marginal_probabilities[0])
ax.set(title='low smoothing')
ax = axes[1]
ax.plot(res.smoothed_marginal_probabilities[1])
ax.set(title='middle smoothing')
ax = axes[2]
ax.plot(res.smoothed_marginal_probabilities[2])
ax.set(title='high smoothing')
fig.tight_layout()
```

具体情况

```
# 为了分析更多的指数或个股，下面将上述分析过程使用函数表示

df=pd.read_csv("/root/experiment01/dataset/gdp.csv")
df_ret=df[df.columns[1]]
    #模型拟合
mod = sm.tsa.MarkovRegression(df_ret.dropna(), k_regimes=3, trend='nc',
switching_variance=True)
res = mod.fit()
fig, axes = plt.subplots(3, figsize=(12,8))
ax = axes[0]
ax.plot(res.smoothed_marginal_probabilities[0])
ax.set(title='low smoothing')
ax = axes[1]
ax.plot(res.smoothed_marginal_probabilities[1])
ax.set(title='middle smoothing')
ax = axes[2]
ax.plot(res.smoothed_marginal_probabilities[2])
ax.set(title='high smoothing')
fig.tight_layout()
```

非线性检测（BDS Test）

参考Statemodels python开源包代码

```
import numpy as np
from scipy import stats
from statsmodels.tools.validation import array_like

def distance_indicators(x, epsilon=None, distance=1.5):
    x = array_like(x, 'x')
```

```
    if epsilon is not None and epsilon <= 0:
        raise ValueError("Threshold distance must be positive if specified."
                         " Got epsilon of %f" % epsilon)
    if distance <= 0:
        raise ValueError("Threshold distance must be positive."
                         " Got distance multiplier %f" % distance)
    if epsilon is None:
        epsilon = distance * x.std(ddof=1)

    return np.abs(x[:, None] - x) < epsilon


def correlation_sum(indicators, embedding_dim):
    if not indicators.ndim == 2:
        raise ValueError('Indicators must be a matrix')
    if not indicators.shape[0] == indicators.shape[1]:
        raise ValueError('Indicator matrix must be symmetric (square)')

    if embedding_dim == 1:
        indicators_joint = indicators
    else:
        corrsum, indicators = correlation_sum(indicators, embedding_dim - 1)
        indicators_joint = indicators[1:, 1:]*indicators[:-1, :-1]

    nobs = len(indicators_joint)
    corrsum = np.mean(indicators_joint[np.triu_indices(nobs, 1)])
    return corrsum, indicators_joint


def correlation_sums(indicators, max_dim):
    corrsums = np.zeros((1, max_dim))

    corrsums[0, 0], indicators = correlation_sum(indicators, 1)
    for i in range(1, max_dim):
        corrsums[0, i], indicators = correlation_sum(indicators, 2)

    return corrsums

def _var(indicators, max_dim):

    nobs = len(indicators)
    corrsum_1dim, _ = correlation_sum(indicators, 1)
    k = ((indicators.sum(1)**2).sum() - 3*indicators.sum() +
         2*nobs) / (nobs * (nobs - 1) * (nobs - 2))

    variances = np.zeros((1, max_dim - 1))

    for embedding_dim in range(2, max_dim + 1):
        tmp = 0
        for j in range(1, embedding_dim):
            tmp += (k**(embedding_dim - j))*(corrsum_1dim**(2 * j))
```

```
        variances[0, embedding_dim-2] = 4 * (
            k**embedding_dim +
            2 * tmp +
            ((embedding_dim - 1)**2) * (corrsum_1dim**(2 * embedding_dim)) -
            (embedding_dim**2) * k * (corrsum_1dim**(2 * embedding_dim - 2)))

    return variances, k

# 定义bds函数，非线性检验
def bds(x, max_dim=2, epsilon=None, distance=1.5):
    x = array_like(x, 'x', ndim=1)
    nobs_full = len(x)

    if max_dim < 2 or max_dim >= nobs_full:
        raise ValueError("Maximum embedding dimension must be in the range"
                         " [2,len(x)-1]. Got %d." % max_dim)

    indicators = distance_indicators(x, epsilon, distance)
    corrsum_mdims = correlation_sums(indicators, max_dim)

    variances, k = _var(indicators, max_dim)
    stddevs = np.sqrt(variances)

    bds_stats = np.zeros((1, max_dim - 1))
    pvalues = np.zeros((1, max_dim - 1))
    for embedding_dim in range(2, max_dim+1):
        ninitial = (embedding_dim - 1)
        nobs = nobs_full - ninitial

        corrsum_1dim, _ = correlation_sum(indicators[ninitial:, ninitial:], 1)
        corrsum_mdim = corrsum_mdims[0, embedding_dim - 1]

        effect = corrsum_mdim - (corrsum_1dim**embedding_dim)
        sd = stddevs[0, embedding_dim - 2]

        bds_stats[0, embedding_dim - 2] = np.sqrt(nobs) * effect / sd

        pvalue = 2*stats.norm.sf(np.abs(bds_stats[0, embedding_dim - 2]))
        pvalues[0, embedding_dim - 2] = pvalue

    return np.squeeze(bds_stats), np.squeeze(pvalues)
```

十二章 传递函数和滞后自回归模型

单输入传递函数噪声模型

此模型中，内生变量（输出）与单个外生变量（输入）相关：

$$y_t = v(B)x_t + n_t$$

其中，滞后多项式$v(B) = v_0 + v_1B + v_2B^2 + \ldots\ldots$作为传递函数，允许x通过分布滞后影响y。此模型一关键假设为x_t与n_t独立，即过去的x会影响y，但y的变化不能反馈到x。

由于v(B)的阶数无限，我们需要加一些限制使其变得可行。于是把v(B)写为合理的滞后公式$v(B) = \frac{w(B)B^b}{\delta(B)}$，其中$w(B) = w_0 - w_1B - \ldots - w_sB^s$，$\delta(B) = 1 - \delta_1B - \ldots - \delta_rB^r$。若噪声遵循ARMA(p, q)模型:$n_t = \frac{\theta(B)}{\phi(B)}$，则此模型可写为$y_t = \frac{w(B)}{\delta(B)}x_(t-b) + \frac{\theta(B)}{\phi(B)}a_t$。

有多个输入时，模型写为

$$y_t = \sum_{j=1}^{M} v_j(B)x_{j,t} + n_t = \sum_{j=1}^{M} \frac{w_j(B)B^{b_j}}{\delta_j(B)}x_{j,t} + \frac{\theta(B)}{\phi(B)}a_t$$

回归分布滞后模型

在上述模型基础上指定了限制模式$\delta_1(B) = \ldots = \delta_M(B) = \phi(B)$，$\theta(B) = 1$则被称为回归分布滞后模型(ARDL)，他能够把噪声成分降低为白噪声，并用最小二乘法估计。

Example 12.1

查得1952至2017年英国每月的长期利率与短期利率数据如下：

```
     R20        RS
0    4.11  0.994750
1    4.26  1.028250
2    4.33  2.365042
3    4.23  2.317500
4    4.36  2.350833
..    ...       ...
778  1.95  0.140000
779  2.00  0.050000
780  1.99  0.140000
781  1.91  0.110000
782  1.81  0.020000
```

因题目要求的是长期利率变化与短期利率变化的关系，于是先通过微分求得变化率如下：

```
     DR20       DRS
0    0.15  0.033500
1    0.07  1.336792
2   -0.10 -0.047542
3    0.13  0.033333
4    0.21  0.101000
..    ...       ...
778  0.05 -0.090000
779 -0.01  0.090000
780 -0.08 -0.030000
781 -0.10 -0.090000
782 -0.12  0.050000
```

```
```

```
import matplotlib.pyplot as plt
import pandas as pd
#引入数据
data=pd.read_excel("/root/experiment01/data.xlsx")
R20 = data.R20
RS = data.RS
#画图
plt.plot(R20,label="R20",color="green")
plt.plot(RS,label="RS",color="blue")
plt.legend()
plt.show()
```

上图为长期利率、短期利率的时序图

```
DR20 = data.DR20
DRS = data.DRS
#画差分图
plt.plot(DR20,label="DR20",color="green")
plt.plot(DRS,label="DRS",color="blue")
plt.legend()
plt.show()
```

AIC指标

AIC是衡量统计模型拟合优良性的一种标准，又称赤池信息量准则。它建立在熵的概念基础上，可以权衡所估计模型的复杂度和此模型拟合数据的优良性。AIC越大表明模型拟合越优良，但考虑到避免过度拟合的情况，优先考虑AIC值最小的模型。

```
from statsmodels.tsa.arima_model import ARMA
#通过AIC判断模型参数
def slect_model(data_ts, maxLag):
    init_bic = float("inf")
    init_p = 0
    init_q = 0
    init_properModel = None
    for p in np.arange(maxLag):
        for q in np.arange(maxLag):
             model = ARMA(data_ts, order=(p, q))
    try:
    results_ARMA = model.fit(disp=-1, method='css')
    except:
        continue
    bic = results_ARMA.bic
    if bic < init_bic:
    init_p = p
    init_q = q
    init_properModel = results_ARMA
    init_bic = bic
    return init_bic, init_p, init_q, init_properModel

slect_model(trainSeting,40)
```

模型实现

根据此模型形式，我们可以假设为

$$DR20_t = aDR20_{t-1} + bDR20_{t-2} + cDRS_t + dDRS_{t-1} + a_t$$

此模型本质为多元线性回归

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
```

```
data = pd.read_excel("/root/experiment01/data.xlsx",usecols=[3,4,5,6])
target = pd.read_excel("/root/experiment01/data.xlsx",usecols=[7])
X = data.dropna()
y = target.dropna()

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.7,
random_state=1)
#多元回归模型
lr = LinearRegression()
lr.fit(X_train, y_train)

print(lr.coef_)
print(lr.intercept_)     #输出常数项
```

```
[[ 0.16897312 -0.12392131  0.24663085 -0.05758091]]
[-0.00019778]
```

十三章 向量自回归和格兰杰因果关系检验

向量自回归模型（VAR）

VAR模型

以金融价格为例，传统的时间序列模型比如ARIMA,ARIMA-GARCH等，只分析价格自身的变化，模型的形式为：

$$y_t = \beta_1 * y_{t-1} + \beta_2 * y_{t-2} + \dots$$

其中y_{t-1}称为自身的滞后项。

但是VAR模型除了分析自身滞后项的影响外，还分析其他相关因素的滞后项对未来值产生的影响，模型的形式为：

$$y_t = \beta_1 * y_{t-1} + \alpha_1 * x_{t-1} + \beta_2 * y_{t-2} + \alpha_2 * x_{t-2} + \dots$$

其中x_{t-1}就是其他因子的滞后项。

VAR模型

1）对N个因子的原始数据进行平稳性检验，也就是ADF检验

2）对应变量（yt）和影响因子（Xt）做协整检验

3）通过AIC,BIC,以及LR定阶。

4）估计参数，看参数的显著性。

5）对参数进行稳定性检验

6）使用乔里斯基正交化残差进行脉冲响应分析

7）使用乔里斯基正交化残差进行方差分解分析

VAR建模的时候以上面的条件为例，其实模型估计参数时会给出三个3个方程(应变量各自不同）：

方程1：$y_t = \beta_1 * y_{t-1} + \alpha_1 * X1_{t-1} + \Theta_1 * X2_{t-1} + \epsilon_t$

方程2：$X1_t = \beta_1 * X1_{t-1} + \alpha_1 * y_{t-1} + \Theta_1 * X2_{t-1} + \eta_t$

方程3：$X2_t = \beta_1 * X2_{t-1} + \alpha_1 * y_{t-1} + \Theta_1 * X1_{t-1} + w_t$

方程1的残差序列：ϵ_t

方程2的残差序列：η_t

方差3的残差序列：w_t

三个方程的乔里斯基正交化的步骤就是：

正交1：$\frac{\eta_t}{\epsilon_t}$

正交2：$\frac{w_t}{\epsilon_t}$

正交3：$\frac{w_t}{\eta_t}$

正交4：$\frac{\frac{\eta_t}{\varepsilon_t}}{\frac{\omega_t}{\varepsilon_t}}$ 正交5：$\frac{\frac{\eta_t}{\varepsilon_t}}{\frac{\omega_t}{\eta_t}}$ 最后用正交4除正交5，得到的序列就是乔里斯基正交化残差了。

格兰杰因果关系检验

格兰杰因果检验以自回归模型为基础，能够检验一组时间序列是否为另一组时间序列的原因，但并不是指真正意义上的因果关系而是一个变量对另一个变量的依存性。其基本观念是未来的事件不会对目前与过去产生因果影响，而过去的事件才可能对现在及未来产生影响。

格兰杰因果关系检验假设了有关y和x每一变量的预测的信息全部包含在这些变量的时间序列之中。检验要求估计以下的回归：

$$y_t = \sum_{i-1}^{q} \alpha_i x_{i-1} + \sum_{j-1}^{q} \beta_j y_{t-j} + u_{1,t}$$

$$x_t = \sum_{i-1}^{s} \lambda_i x_{t-i} + \sum_{j-1}^{s} \delta_j y_{t-j} + u_{2,t}$$

若在包含了变量x、y的过去信息下，对y的预测效果优于单独由y过去信息对y进行的预测效果，就认为x是引致y的格兰杰关系。

两个变量间存在四种情况：x是引起y变化的原因、y是引起x变化的原因、x与y互为因果关系、x与y独立

Example 13.1

```
import statsmodels.api as sm
import statsmodels.stats.diagnostic
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

```
data = pd.read_excel("/root/experiment01/data.xlsx",usecols=[1,2])
R20 = data.R20
RS = data.RS
fig = plt.figure(figsize=(12,8))
plt.plot(R20,'r',label='R20')
plt.plot(RS,'g',label='RS')
plt.title('Correlation: ')
plt.axis('tight')
plt.legend(loc=0)
plt.ylabel('Price')
plt.show()
```

```
#**ADF单位根**
adfResult = sm.tsa.stattools.adfuller(data,maxlags)
output = pd.DataFrame(index=['Test Statistic Value', "p-value", "Lags Used",
"Number of Observations Used","Critical Value(1%)", "Critical Value(5%)",
"Critical Value(10%)"],
                      columns=['value'])
output['value']['Test Statistic Value'] = adfResult[0]
output['value']['p-value'] = adfResult[1]
output['value']['Lags Used'] = adfResult[2]
output['value']['Number of Observations Used'] = adfResult[3]
output['value']['Critical Value(1%)'] = adfResult[4]['1%']
output['value']['Critical Value(5%)'] = adfResult[4]['5%']
output['value']['Critical Value(10%)'] = adfResult[4]['10%']
```

```
#协整检验
result = sm.tsa.stattools.coint(data1,data2)
lnDataDict = {'lnSHFEDiff':lnSHFEDiff,'lnXAUDiff':lnXAUDiff}
lnDataDictSeries = pd.DataFrame(lnDataDict,index=lnSHFEDiffIndex)
data = lnDataDictSeries[['lnSHFEDiff','lnXAUDiff']]
orgMod = sm.tsa.VARMAX(dataframe,order=(varLagNum,0),trend='nc',exog=None)
res= orgMod.fit(maxiter=1000,disp=False)
print(res.summary())
resid = res.resid
result = {'fitMod':fitMod,'resid':resid}
```

```
Statespace Model Results
==========================================================================================
Dep. Variable:          ['R20', 'RS']   No. Observations:                   783
Model:                         VAR(3)   Log Likelihood                -465.577
                         + intercept   AIC                            965.154
Date:                Wed, 12 Aug 2020   BIC                           1044.428
Time:                        19:55:24   HQIC                           995.638
Sample:                             0
                                  - 783
Covariance Type:                  opg
==========================================================================================
===
Ljung-Box (Q):                70.78, 69.44   Jarque-Bera (JB):      418.35,
1533.78
Prob(Q):                        0.00, 0.00   Prob(JB):                   0.00,
0.00
Heteroskedasticity (H):         1.10, 0.27   Skew:                      -0.17,
1.08
Prob(H) (two-sided):            0.45, 0.00   Kurtosis:                   6.57,
9.51
                            Results for equation R20
==========================================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------------
intercept      0.0221      0.031      0.705      0.481      -0.039       0.084
L1.R20         1.3087      0.030     43.396      0.000       1.250       1.368
L1.RS         -0.0098      0.022     -0.454      0.650      -0.052       0.033
L2.R20        -0.4517      0.046     -9.716      0.000      -0.543      -0.361
L2.RS          0.0337      0.034      0.984      0.325      -0.033       0.101
L3.R20         0.1360      0.029      4.665      0.000       0.079       0.193
L3.RS         -0.0200      0.023     -0.869      0.385      -0.065       0.025
                            Results for equation RS
==========================================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------------
intercept     -0.0145      0.046     -0.313      0.755      -0.106       0.077
L1.R20         0.2979      0.056      5.281      0.000       0.187       0.409
L1.RS          1.1982      0.029     41.048      0.000       1.141       1.255
L2.R20        -0.3507      0.094     -3.750      0.000      -0.534      -0.167
L2.RS         -0.1869      0.053     -3.555      0.000      -0.290      -0.084
```

```
L3.R20          0.0772       0.063       1.226       0.220       -0.046        0.200
L3.RS          -0.0390       0.040      -0.977       0.329       -0.117        0.039
                            Error covariance matrix

==============================================================================
===
                     coef    std err          z      P>|z|      [0.025
 0.975]
------------------------------------------------------------------------------
---
sqrt.var.R20         0.2783      0.004     63.552      0.000       0.270
0.287
sqrt.cov.R20.RS      0.1949      0.012     16.930      0.000       0.172
0.217
sqrt.var.RS          0.3792      0.006     67.425      0.000       0.368
0.390
==============================================================================
===
```

结构向量自回归模型（SVAR）

参考相关资料

结构向量自回归模型（SVAR）可以捕捉模型系统内各个变量之间的即时的结构性关系。而如果仅仅建立一个VAR模型，这样的结构关联性却被转移到了随机扰动向量的方差-协方差矩阵中了。也正是基于这个原因，VAR模型实质上是一个缩减形式，没有明确体现变量间的结构性关系。

一个结构向量自回归模型可以写成为：

$$B_0 y_t = c_0 + B_1 y_1 + B_2 y_{t-2} + \ldots + B_p y_{t-p} + e_t$$

其中：c0是n×1常数向量，Bi是n×n矩阵，et是n×1误差向量。

一个有两个变量的结构VAR(1)可以表示为

$$\begin{pmatrix} 1 & B_{0;1,2} \\ B_{0;2,1} & 1 \end{pmatrix} \begin{pmatrix} y_{1,t} \\ y_{2,t} \end{pmatrix} = \begin{pmatrix} c_{0;1} \\ c_{0;2} \end{pmatrix} + \begin{pmatrix} B_{1;1,1} & B_{1;1,2} \\ B_{1;2,1} & B_{1;2,2} \end{pmatrix} + \begin{pmatrix} e_{1,t} \\ e_{2,t} \end{pmatrix}$$

其中：

$$\sum = E(e_t e_t') = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix}$$

在一定的经济理论基础上的计量经济模型如果已经对各种冲击进行了显性的识别，那么这些模型通常可以变换为VAR或SVAR模型，VAR或SVAR模型是这些模型的简化式。但是有这些模型经过变换得到的VAR模型与一般的VAR模型并不完全相同，变现为两方面：

首先，这些模型经过变换得到的VAR模型是一种带有约束的VAR模型,我们可以通过约束检验和似然函数比例方法进行进一步检验来比较这两种模型。

其次，这些模型经过变换得到的VAR模型比一般的VAR模型有优越性的地方，但也有不足之处。通常这些模型对冲击进行了显性的识别，因而我们不需要进行冲击识别的过程，而一般的VAR模型所包含的冲击更为广泛，只有施加适当的识别条件，才能得到人们感兴趣的冲击，所以二者通常不能完全相互取代。

因此，要使这两种模型都避免Lucas批判(即当经济环境、政策体制、预期等发生变化导致深层次参数发生变化时，可能会导致模型中估计参数的变化及行为方程的不稳定,这将对政策分析和评价造成很大影响)，我们需要对这两种模型进行有关的外生性检验。

第十四章 误差修正与协整关系

时间序列信息生成过程（DGP）：

$$\begin{aligned} y_t &= \phi y_{t-1} + u_t \qquad u_t \sim i.i.d.(0, \sigma_u^2) \\ x_t &= \phi^* x_{t-1} + v_t \qquad v_t \sim i.i.d.(0, \sigma_v^2) \end{aligned}$$

则y_t和x_t是不相关的一阶自回归过程。 由于x_t既不会影响y_t，也不会受到y_t的影响，因此应该希望回归模型中的系数β_1：

$$y_t = \beta_0 + \beta_1 x_t + \epsilon_t$$

会收敛到零，反映出这两个系列之间没有任何关系，正如该回归的R^2统计量一样。

当$\phi = \phi^* = 1$时，两个序列为随机游走序列。这时我们对两序列的统计量t和R^2进行计算。

$$t = \frac{|\hat{\beta}_1|}{se(\hat{\beta}_1)}$$

这就是t-检验统计量

协整检验

考虑到协整在具有集成变量的回归模型中所起的关键作用，测试其存在显然很重要。 检验可以基于协整回归的残差：

$$\hat{e}_t = y_t - \hat{\beta}_0 - \hat{\beta}_1 x_{1,t} - \ldots - \hat{\beta}_M x_{M,t}$$

这种基于残差的过程试图用单位根检验来检验没有协整的零假设。

ARDL模型的条件错误校正（CEC）形式为：

$$\nabla y_t = \alpha_0 + \alpha_1 t - \phi(1) y_{t-1} + \sum_{j=1}^{M} \beta_j(1) x_{j,t-1} + \phi^*(B) \nabla y_{t-1} + \sum_{j=0}^{M} \gamma_j(B) \nabla x_{j,t} + a_t$$

尽管上式是CEC的一般形式，但是有几种方法可以使常数和趋势进入误差校正。 例如，可以使$\alpha_0 = \alpha_1 = 0$，得到以下的化简形式：

$$\nabla y_t = -\phi(1) ec_{t-1} + \phi^*(B) \nabla y_{t-1} + \sum_{j=0}^{M} \gamma_j(B) \nabla x_{j,t} + a_t$$

其中：

$$ec_t = y_t - \sum_{j=1}^{M} \frac{\beta_j(1)}{\phi(1)} x_{j,t}$$

Table 14.1

```
import numpy as np


def random_walk(x0, sigma, const, n):
    a = np.random.normal(0, sigma, n)
    x_t = np.zeros(n)
    x_t[0] = x0
    for i in range(1, n):
        x_t[i] = x_t[i-1] + a[i] + const
    return x_t
```

累计生成1000次，每次生成两个长度为50的随机游走序列，计算统计量区间出现的频次。

```
import pandas as pd
import matplotlib.dates as mdate
import matplotlib.pyplot as plt
import scipy.stats as st
import warnings


warnings.filterwarnings('ignore')
np.random.seed(3)
nth = 1000
n = 50
count1 = count2 = 0
tlist, rlist, tcolumn, rcolumn = [[] for _ in range(4)]
tcal, rcal = [np.zeros(10) for _ in range(2)]
col = np.full(10, '')
```

```
#循环生成1000次，并将每次的结果保存到列表中：
for i in range(nth):
    x = random_walk(0, 1, 0, n)
    y = random_walk(0, 1, 0, n)
    slope, intercept, r_value, p_value, std_err = st.linregress(x, y)
    temp1 = np.sum(np.power(y-slope*x-intercept, 2))/(n-2)
    temp2 = np.sum(np.power(x - np.mean(x), 2))
    se = np.sqrt(temp1/temp2)
    t = np.abs(slope)/se
    tlist.append(int(t/2))
    rlist.append(int(np.power(r_value, 2)*10))
```

```
#分别对每个区间的频数进行统计：
for j in range(9):
    tcal[j] = tlist.count(j)
    rcal[j] = rlist.count(j)
    tcolumn.append(str(j * 2) + '-' + str(2 * (j + 1)))
    rcolumn.append(str(j / 10) + '-' + str((j + 1)/10))
for j in range(len(tlist)):
    if tlist[j] > 8:
        count1 += 1
    if rlist[j] > 8:
        count2 += 1
tcal[9], rcal[9] = count1, count2
tcolumn.append('>18')
rcolumn.append('0.9-1.0')
```

利用pandas将每个区间的结果打印出来：

```
res = np.append(np.append(tcolumn, tcal), np.append(rcolumn, rcal)).reshape([4,
10])
res = pd.DataFrame(df, index=['t-Statistics', '', 'R2 Statistics', ''],
columns=col)
res
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

t-Statistics	0-2	2-4	4-6	6-8	8-10	10-12	12-14	14-16	16-18	>18

	339.0	250.0	180.0	114.0	56.0	28.0	19.0	3.0	7.0	4.0
R2 Statistics	0.0-0.1	0.1-0.2	0.2-0.3	0.3-0.4	0.4-0.5	0.5-0.6	0.6-0.7	0.7-0.8	0.8-0.9	0.9-1.0
	382.0	146.0	125.0	92.0	86.0	64.0	53.0	37.0	14.0	1.0

Figure 14.1

u和v为误差，遵循标准正态分布。

```
def multipl(a, b):
    sumofab = 0.0
    for i in range(len(a)):
        temp = a[i] * b[i]
        sumofab += temp
    return sumofab


def corrcoef(x, y):
    n = len(x)
    sum1 = sum(x)
    sum2 = sum(y)
    sumofxy = multipl(x, y)
    sumofx2 = sum([pow(i, 2) for i in x])
    sumofy2 = sum([pow(j, 2) for j in y])
    num = sumofxy - (float(sum1) * float(sum2) / n)
    den = np.sqrt((sumofx2 - float(sum1 ** 2) / n) * (sumofy2 - float(sum2 ** 2)
/ n))
    return num / den
```

现在将两个序列生成1000次，每个序列的长度为100。统计皮尔逊相关系数的出现频次。

首先生成u、v两个序列：

```
np.random.seed(10)
n = 1000
t = 100
plist = []
for i in range(n):
    u = np.random.normal(0, 1, t)
    v = np.random.normal(0, 1, t)#标准正态分布
    s = corrcoef(u, v)
    plist.append(s)
```

画出对应的频数图

```
plt.figure()
plt.hist(plist, bins=32, facecolor="w", edgecolor="green")
plt.title('Frequency distribution of the correlation coefficient between I(0)
series.')
plt.show()
```

Figure 14.2

```
np.random.seed(1)
n = 1000
t = 100
plist = []
for i in range(n):
    x = random_walk(0, 1, 0, t)
    y = random_walk(0, 1, 0, t)
    s = corrcoef(x, y)
    plist.append(s)
```

画出对应的频数图

```
plt.figure()
plt.hist(plist, bins=40, facecolor="w", edgecolor="green")
plt.title('Frequency distribution of the correlation coefficient between I(1)
series.')
plt.show()
```

Figure 14.3

```
#两个 I(2) 序列的相关系数的频数统计
def auto_reg(phi, sigma, x0, const, n):
    k = len(phi)
    x_t = np.zeros(n)
    a = np.random.normal(0, sigma, n)
    for i in range(k):
        x_t[i] = x0[i]
    for i in range(k, n):
        temp = 0
        for t in range(k):
            temp += phi[t] * x_t[i-t-1]
        x_t[i] = temp + a[i] + const
    return x_t
```

```
#生成序列
np.random.seed(10)
n = 1000
t = 100
plist = []
for i in range(n):
    z = auto_reg([2, -1], 1, [0, 0], 0, t)
    w = auto_reg([2, -1], 1, [0, 0], 0, t)
    s = corrcoef(z, w)
    plist.append(s)
```

画出对应的频数图

```
plt.figure()
plt.hist(plist, bins=40, facecolor="w", edgecolor="green")
plt.title('Frequency distribution of the correlation coefficient between I(2)
series.')
plt.show()
```

Figure 14.3

I(1)和I(2) 序列的相关系数的频数统计

```
np.random.seed(12)
n = 1000
t = 100
plist = []
for i in range(n):
    z = auto_reg([2, -1], 1, [0, 0], 0, t)
    x = random_walk(0, 1, 0, t)
    s = corrcoef(z, x)
    plist.append(s)
```

```
#画出对应的频数图
plt.figure()
plt.hist(plist, bins=40, facecolor="w", edgecolor="green")
plt.title('Frequency distribution of the correlation coefficient between I(1)
and I(2)')
plt.show()
```

Figure 14.3 I(0)和I(1)

序列的相关系数的频数统计

```
np.random.seed(123)
n = 1000
t = 100
plist = []
for i in range(n):
    y = random_walk(0, 1, 0, t)
    v = np.random.normal(0, 1, t)
    s = corrcoef(y, v)
    plist.append(s)
```

```
plt.figure()
plt.hist(plist, bins=32, facecolor="w", edgecolor="green")
plt.title('Frequency distribution of the correlation coefficient between I(0)
and I(1)')
plt.show()
```

Example 14.1

在本题中，选取R20和RS的数据，分别进行回归。其中残差的关系如下：

$$\hat{e}_{1,t} = R20_t - \hat{\beta}_0 - \hat{\beta}_1 RS_t$$
$$\hat{e}_{2,t} = RS_t - \hat{\beta}_0^{'} - \hat{\beta}_1 R20_t$$

```
data = pd.read_csv('data/interest_rates.csv')
temp = data.iloc[:, 0]
t = pd.to_datetime(data.iloc[:, 0], format='%Y-%m-%d')
r20 = np.array(data.iloc[:, 1])
rs = np.array(data.iloc[:, 2])
```

```
#将R20与RS数据进行回归，得到以下结果：
k1 = st.linregress(rs, r20)
k2 = st.linregress(r20, rs)
print(k1)
print(k2)
```

```
LinregressResult(slope=0.7988615802015735, intercept=2.437830758360362,
rvalue=0.8822278137604712, pvalue=1.0694382037998232e-258,
stderr=0.015226154071853432)
LinregressResult(slope=0.9742938384597102, intercept=-1.0051785009305378,
rvalue=0.8822278137604712, pvalue=1.0694382037998232e-258,
stderr=0.01856986049060193)
```

第15章 向量自回归与VEC模型

1.向量自回归与整合变量（INTEGRATED VARIABLES）

在这一章中，我们需要讨论带有I(1)的向量自回归过程。一般VAR模型的过程为 一个n维时间序列Y_t， 而且
$Y = C + \theta_1 Y_{t-1} + \theta_2 Y_{t-2} \ldots + \theta_p Y_{t-p} + \varepsilon_t$

其中$E(\varepsilon) = 0, E(\varepsilon\varepsilon') = \begin{cases} \pi & t = \tau \\ 0 & t \neq \tau \end{cases}$

ε 在不同时刻独立同分布，且服从正态分布，则称为p阶向量自回归模型。

向量自回归模型的一般步骤为

- 对原始数据进行平稳性检验，也就是ADF检验
- 对应变量和自变量做协整检验
- LR（long-run）定阶
- 估计参数

```
import statsmodels.api as sm
import statsmodels.stats.diagnostic
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

```
dta = sm.datasets.webuse('lutkepohl2', 'https://www.stata-press.com/data/r12/')#
获取数据
dta.index = dta.qtr
endog = dta.loc['1960-04-01':'1978-10-01', ['dln_inv', 'dln_inc',
'dln_consump']]
```

```
exog = endog['dln_consump']
mod = sm.tsa.VARMAX(endog[['dln_inv', 'dln_inc']], order=(2,0), trend='n',
exog=exog)
res = mod.fit(maxiter=1000, disp=False)
print(res.summary())
```

```
/root/anaconda3/envs/jupyter_notebook/lib/python3.7/site-
packages/statsmodels/tsa/base/tsa_model.py:165: ValueWarning: No frequency
information was provided, so inferred frequency QS-OCT will be used.
  % freq, ValueWarning)
```

```
                           Statespace Model Results

==========================================================================
==
Dep. Variable:     ['dln_inv', 'dln_inc']   No. Observations:
75
Model:                            VARX(2)   Log Likelihood
361.038
Date:                   Wed, 12 Aug 2020   AIC
-696.077
Time:                           17:59:20   BIC
-665.949
Sample:                       04-01-1960   HQIC
-684.047
                            - 10-01-1978

Covariance Type:                     opg

==========================================================================
===
Ljung-Box (Q):                61.24, 39.25   Jarque-Bera (JB):          11.14,
2.41
Prob(Q):                        0.02, 0.50   Prob(JB):                   0.00,
0.30
```

```
Heteroskedasticity (H):          0.45, 0.40   Skew:                        0.16,
-0.38
Prob(H) (two-sided):             0.05, 0.03   Kurtosis:                    4.86,
3.44
                            Results for equation dln_inv

================================================================================
====
                       coef    std err          z      P>|z|      [0.025
0.975]
--------------------------------------------------------------------------------
----
L1.dln_inv          -0.2388      0.093     -2.564      0.010      -0.421
-0.056
L1.dln_inc           0.2861      0.450      0.636      0.525      -0.595
1.167
L2.dln_inv          -0.1665      0.155     -1.072      0.284      -0.471
0.138
L2.dln_inc           0.0628      0.421      0.149      0.881      -0.762
0.888
beta.dln_consump     0.9750      0.638      1.528      0.127      -0.276
2.226
                            Results for equation dln_inc

================================================================================
====
                       coef    std err          z      P>|z|      [0.025
0.975]
--------------------------------------------------------------------------------
----
L1.dln_inv           0.0633      0.036      1.773      0.076      -0.007
0.133
L1.dln_inc           0.0811      0.107      0.758      0.448      -0.129
0.291
L2.dln_inv           0.0104      0.033      0.315      0.753      -0.054
0.075
L2.dln_inc           0.0350      0.134      0.261      0.794      -0.228
0.298
beta.dln_consump     0.7731      0.112      6.879      0.000       0.553
0.993
                                Error covariance matrix

================================================================================
============
                              coef    std err          z      P>|z|      [0.025
     0.975]
--------------------------------------------------------------------------------
------------
sqrt.var.dln_inv             0.0434      0.004     12.284      0.000       0.036
      0.050
sqrt.cov.dln_inv.dln_inc   5.58e-05      0.002      0.028      0.978      -0.004
      0.004
```

```
sqrt.var.dln_inc             0.0109       0.001     11.222       0.000        0.009
      0.013
==============================================================================
============

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

以上 为VAR（2）的一个估计实例

向量自回归与协整变量（COINTEGRATED VARIABLES）

协整： 如果所考虑的时间序列具有相同的单整阶数，且某种协整向量使得组合时间序列的单整阶数降低，则称这些时间序列之间存在显著的协整关系。

协整即存在共同的随机性趋势。协整检验的目的是决定一组非平稳序列的线性组合是否具有稳定的均衡关系，伪回归的一种特殊情况即是两个时间序列的趋势成分相同，此时可能利用这种共同趋势修正回归使之可靠。正是由于协整传递出了一种长期均衡关系，若是能在看来具有单独随机性趋势的几个变量之间找到一种可靠联系，那么通过引入这种醉汉与狗之间距离的"相对平稳"对模型进行调整，可以排除单位根带来的随机性趋势，即所称的误差修正模型。

在进行时间系列分析时，传统上要求所用的时间系列必须是平稳的，即没有随机趋势或确定趋势，否则会产生"伪回归"问题。但是，在现实经济中的时间系列通常是非平稳的，我们可以对它进行差分把它变平稳，但这样会让我们失去总量的长期信息，而这些信息对分析问题来说又是必要的，所以用协整来解决此问题。

Example15.2

首先用VAR（2）估计rs 与rs20 这两个向量。估计时使用C作为外生变量。

```
df=pd.read_excel("/root/experiment01/rs1.xlsx")
df.head()
df.tail()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	date	rs	r20	c	resid
823	2020-08-01	NaN	NaN	0.0	NaN
824	2020-09-01	NaN	NaN	0.0	NaN
825	2020-10-01	NaN	NaN	0.0	NaN
826	2020-11-01	NaN	NaN	0.0	NaN
827	2020-12-01	NaN	NaN	0.0	NaN

```
endog=df[['rs', 'r20']].dropna()
ex=df["c"][:len(endog)]
endog
ex
```

```
0     -0.012296
1      0.233268
2     -0.081877
3     -0.113973
4      0.007990
         ...
781    0.000000
782    0.000000
783    0.000000
784    0.000000
785    0.000000
Name: c, Length: 786, dtype: float64
```

```
mod = sm.tsa.VARMAX(endog, order=(2,0), trend='n', exog=ex)
res = mod.fit(maxiter=1000, disp=False)
print(res.summary())
```

```
                            Statespace Model Results
==============================================================================
Dep. Variable:          ['rs', 'r20']   No. Observations:                  828
Model:                         VARX(2)   Log Likelihood                -471.867
Date:                Wed, 12 Aug 2020   AIC                            969.733
Time:                        19:22:00   BIC                           1031.080
Sample:                             0   HQIC                           993.262
                                - 828
Covariance Type:                  opg
```

```
===================================================================================
Ljung-Box (Q):                    nan, nan   Jarque-Bera (JB):      1663.06,
188.00
Prob(Q):                          nan, nan   Prob(JB):                   0.00,
0.00
Heteroskedasticity (H):         0.17, 0.60   Skew:                      1.14,
-0.18
Prob(H) (two-sided):            0.00, 0.00   Kurtosis:                  9.75,
5.37
                          Results for equation rs
===================================================================================
                coef    std err          z      P>|z|      [0.025      0.975]
-----------------------------------------------------------------------------------
L1.rs          1.1999      0.027     44.000      0.000       1.146       1.253
L1.r20         0.2807      0.053      5.268      0.000       0.176       0.385
L2.rs         -0.2280      0.027     -8.468      0.000      -0.281      -0.175
L2.r20        -0.2577      0.053     -4.840      0.000      -0.362      -0.153
beta.c        -0.2672      1.567     -0.170      0.865      -3.339       2.804
                          Results for equation r20
===================================================================================
                coef    std err          z      P>|z|      [0.025      0.975]
-----------------------------------------------------------------------------------
L1.rs         -0.0206      0.021     -0.988      0.323      -0.061       0.020
L1.r20         1.2833      0.028     45.356      0.000       1.228       1.339
L2.rs          0.0228      0.021      1.081      0.280      -0.019       0.064
L2.r20        -0.2867      0.028    -10.279      0.000      -0.341      -0.232
beta.c         0.0478      0.664      0.072      0.943      -1.254       1.350
                             Error covariance matrix

===================================================================================
                     coef    std err          z      P>|z|      [0.025
0.975]
-----------------------------------------------------------------------------------
---
sqrt.var.rs          0.4257      0.006     69.935      0.000       0.414
0.438
sqrt.cov.rs.r20      0.1282      0.007     18.027      0.000       0.114
0.142
sqrt.var.r20         0.2492      0.005     53.063      0.000       0.240
0.258
===================================================================================
===

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

```
# 使用VAR(3)估计
mod = sm.tsa.VARMAX(endog, order=(3,0), trend='n', exog=ex)
res = mod.fit(maxiter=1000, disp=False)
print(res.summary())
```

```
                            Statespace Model Results
==============================================================================
Dep. Variable:          ['rs', 'r20']   No. Observations:                  828
Model:                         VARX(3)   Log Likelihood                -465.030
Date:                Wed, 12 Aug 2020   AIC                            964.061
Time:                        19:39:24   BIC                           1044.284
Sample:                             0   HQIC                           994.829
                                - 828
Covariance Type:                  opg
===================================================================================
===
Ljung-Box (Q):                    nan, nan   Jarque-Bera (JB):      1694.47,
168.11
Prob(Q):                          nan, nan   Prob(JB):                 0.00,
0.00
Heteroskedasticity (H):         0.17, 0.57   Skew:                     1.18,
-0.13
Prob(H) (two-sided):            0.00, 0.00   Kurtosis:                 9.80,
5.25
                            Results for equation rs
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
L1.rs          1.1957      0.029     41.645      0.000       1.139       1.252
L1.r20         0.2982      0.056      5.373      0.000       0.189       0.407
L2.rs         -0.1852      0.052     -3.551      0.000      -0.287      -0.083
L2.r20        -0.3507      0.093     -3.772      0.000      -0.533      -0.168
L3.rs         -0.0397      0.040     -0.996      0.319      -0.118       0.038
L3.r20         0.0764      0.063      1.219      0.223      -0.046       0.199
beta.c        -0.2798      1.482     -0.189      0.850      -3.184       2.624
                           Results for equation r20
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
L1.rs         -0.0106      0.022     -0.492      0.623      -0.053       0.032
L1.r20         1.3101      0.030     43.907      0.000       1.252       1.369
L2.rs          0.0335      0.034      0.980      0.327      -0.034       0.101
L2.r20        -0.4525      0.046     -9.785      0.000      -0.543      -0.362
L3.rs         -0.0210      0.023     -0.913      0.361      -0.066       0.024
L3.r20         0.1392      0.029      4.803      0.000       0.082       0.196
beta.c         0.0453      0.708      0.064      0.949      -1.342       1.433
                              Error covariance matrix

===================================================================================
===
```

```
                      coef    std err          z      P>|z|      [0.025
0.975]
------------------------------------------------------------------------------
---
sqrt.var.rs          0.4252      0.006     69.958      0.000       0.413
0.437
sqrt.cov.rs.r20      0.1270      0.007     18.039      0.000       0.113
0.141
sqrt.var.r20         0.2473      0.005     52.501      0.000       0.238
0.257
==============================================================================
===

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

Example 15.3

检验VECM模型在英国利率上的表现

```
from statsmodels.tsa.vector_ar.vecm import VECM
model = VECM(endog = endog, k_ar_diff = 7, coint_rank = 6, deterministic = 'co')
res = model.fit()

X_pred = res.predict(steps=10)
X_pred
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-1-d4a2bef29d17> in <module>
      1 from statsmodels.tsa.vector_ar.vecm import VECM
----> 2 model = VECM(endog = endog, k_ar_diff = 7, coint_rank = 6, deterministic
= 'co')
      3 res = model.fit()
      4
      5 X_pred = res.predict(steps=10)
```

```
NameError: name 'endog' is not defined
```

```
#预测结果
plt.plot(X_pred)
plt.legend(['rs',"r20"])
plt.show()
```

Example15.4

```
import numpy as np
import pandas
import statsmodels.api as sm
from statsmodels.tsa.api import VAR,SVAR
```

```
model = VAR(endog)
results = model.fit(3)
```

```
results.summary()
```

```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Wed, 12, Aug, 2020
Time:                     21:16:26
--------------------------------------------------------------------
No. of Equations:         2.00000    BIC:                   -4.39319
Nobs:                     783.000    HQIC:                  -4.44450
Log likelihood:          -455.483    FPE:                  0.0113724
AIC:                     -4.47656    Det(Omega_mle):       0.0111718
--------------------------------------------------------------------
Results for equation rs
==============================================================================
            coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const         -0.005228         0.036862           -0.142           0.887
L1.rs          1.199313         0.040129           29.886           0.000
L1.r20         0.294782         0.061048            4.829           0.000
L2.rs         -0.187626         0.062012           -3.026           0.002
L2.r20        -0.348557         0.094870           -3.674           0.000
L3.rs         -0.038836         0.039528           -0.982           0.326
L3.r20         0.076997         0.061701            1.248           0.212
==============================================================================

Results for equation r20
```

```
==============================================================================
            coefficient       std. error           t-stat            prob
------------------------------------------------------------------------------
const          0.022874         0.024202            0.945           0.345
L1.rs         -0.009637         0.026347           -0.366           0.715
L1.r20         1.309279         0.040081           32.665           0.000
L2.rs          0.033408         0.040715            0.821           0.412
L2.r20        -0.452051         0.062288           -7.257           0.000
L3.rs         -0.019845         0.025952           -0.765           0.444
L3.r20         0.136011         0.040510            3.357           0.001
==============================================================================

Correlation matrix of residuals
             rs       r20
rs     1.000000  0.459487
r20    0.459487  1.000000
```

```
#results.plot()
```

```
res=results.test_causality('rs', ['r20', 'rs'], kind='wald')
print(res)
```

```
<statsmodels.tsa.vector_ar.hypothesis_test_results.CausalityTestResults object.
H_0: %s do not Granger-cause rs: reject at 5% significance level. Test
statistic: 62868.469, critical value: 12.592>, p-value: 0.000>
```

```
results.to_vecm()
```

```
{'Gamma': array([[ 0.22646186,  0.27155938,  0.0388359 , -0.07699749],
        [-0.01356378,  0.31604068,  0.01984455, -0.13601057]]),
 'Pi': array([[-0.02714869,  0.02322247],
        [ 0.00392675, -0.00676172]])}
```

Example 15.5

```
import numpy as np
import pandas
import statsmodels.api as sm
from statsmodels.tsa.api import VAR
model = VAR(endog)
results = model.fit(4)
res=results.test_causality('rs', ['r20', 'rs'])
print(res)
```

```
<statsmodels.tsa.vector_ar.hypothesis_test_results.CausalityTestResults object.
H_0: %s do not Granger-cause rs: reject at 5% significance level. Test
statistic: 7827.480, critical value: 1.944>, p-value: 0.000>
```

Example 15.6

待解决

```
from statsmodels.tsa.vector_ar.vecm import VECM
model = VECM(endog = endog, k_ar_diff = 2, coint_rank = 1, deterministic = 'co')
res = model.fit()
res.conf_int_det_coef_coint()
```

```
array([], shape=(0, 1), dtype=[('lower', '<f8'), ('upper', '<f8')])
```

Example 15.7

```
data=pd.read_excel("/root/experiment01/trf1.xlsx")

data=data.drop(columns = ['_date_'])
data=data.dropna()
data
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	trf	temp	c	volc	amo	soi
0	0.155966	0.058267	0.309571	0.219079	0.205083	-0.025000
1	0.150523	-0.010733	0.083041	0.229017	0.143583	-0.008333
2	0.173692	0.072267	0.026940	0.231073	0.155917	-0.138333
3	0.217857	0.041267	0.132820	0.222848	0.097083	0.062500
4	0.251133	0.028267	0.188829	0.218736	0.020000	-0.026667
...	...	...	...	...	...	...
145	2.282982	0.725267	0.390467	0.000000	0.101667	0.228333
146	2.342985	0.773267	0.390467	0.000000	0.213333	-0.019167
147	2.374454	0.802267	0.390467	0.000000	0.166500	0.041667
148	2.376194	0.870267	0.390467	0.000000	0.102250	-0.068333
149	2.376060	1.049267	0.390467	0.000000	0.114750	-0.191667

150 rows × 6 columns

```
endog=data[['trf', 'temp']].copy()
exo=data[["volc", "soi", "amo"]].copy()
amo=data["amo"].copy()
```

```
exo["amo(-1)"]=amo.shift(-1)

endog=endog[:-1]
```

```
exo=exo.dropna()
exo
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	volc	soi	amo	amo(-1)
0	0.219079	-0.025000	0.205083	0.143583
1	0.229017	-0.008333	0.143583	0.155917
2	0.231073	-0.138333	0.155917	0.097083
3	0.222848	0.062500	0.097083	0.020000
4	0.218736	-0.026667	0.020000	0.033167
...	...	...	...	...
144	0.000000	0.132500	0.348917	0.101667
145	0.000000	0.228333	0.101667	0.213333
146	0.000000	-0.019167	0.213333	0.166500
147	0.000000	0.041667	0.166500	0.102250
148	0.000000	-0.068333	0.102250	0.114750

149 rows × 4 columns

```
model = statsmodels.tsa.vector_ar.vecm.VECM(endog,exog=exo,k_ar_diff = 4,
coint_rank = 2, deterministic = 'co')
results = model.fit()
print(results.summary(alpha=0.05))
```

```
Det. terms outside the coint. relation & lagged endog. parameters for equation
trf
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0035      0.002     -2.026      0.043      -0.007      -0.000
exog1         -0.0024      0.003     -0.826      0.409      -0.008       0.003
exog2          0.0008      0.008      0.111      0.912      -0.014       0.016
exog3         -0.0113      0.007     -1.650      0.099      -0.025       0.002
exog4          0.0020      0.006      0.308      0.758      -0.011       0.015
```

L1.trf	1.4602	0.071	20.644	0.000	1.322	1.599
L1.temp	0.0180	0.011	1.637	0.102	-0.004	0.039
L2.trf	-1.3739	0.122	-11.220	0.000	-1.614	-1.134
L2.temp	0.0148	0.010	1.499	0.134	-0.005	0.034
L3.trf	0.8562	0.123	6.944	0.000	0.615	1.098
L3.temp	0.0136	0.009	1.555	0.120	-0.004	0.031
L4.trf	-0.6017	0.072	-8.331	0.000	-0.743	-0.460
L4.temp	0.0098	0.008	1.211	0.226	-0.006	0.026

Det. terms outside the coint. relation & lagged endog. parameters for equation temp

	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.0567	0.013	-4.517	0.000	-0.081	-0.032
exog1	0.0748	0.021	3.597	0.000	0.034	0.116
exog2	-0.3637	0.056	-6.531	0.000	-0.473	-0.255
exog3	0.3125	0.050	6.275	0.000	0.215	0.410
exog4	-0.0141	0.047	-0.302	0.762	-0.106	0.078
L1.trf	-0.0225	0.516	-0.044	0.965	-1.034	0.989
L1.temp	-0.1102	0.080	-1.377	0.169	-0.267	0.047
L2.trf	-0.4081	0.893	-0.457	0.648	-2.159	1.343
L2.temp	-0.0968	0.072	-1.348	0.178	-0.238	0.044
L3.trf	0.7501	0.899	0.834	0.404	-1.013	2.513
L3.temp	-0.0974	0.064	-1.524	0.127	-0.223	0.028
L4.trf	-0.6468	0.527	-1.228	0.220	-1.679	0.386
L4.temp	0.0304	0.059	0.512	0.609	-0.086	0.147

Loading coefficients (alpha) for equation trf

	coef	std err	z	P>|z|	[0.025	0.975]
ec1	0.0220	0.005	4.299	0.000	0.012	0.032
ec2	-0.0207	0.011	-1.874	0.061	-0.042	0.001

Loading coefficients (alpha) for equation temp

	coef	std err	z	P>|z|	[0.025	0.975]
ec1	0.2554	0.037	6.856	0.000	0.182	0.328
ec2	-0.6016	0.081	-7.449	0.000	-0.760	-0.443

Cointegration relations for loading-coefficients-column 1

	coef	std err	z	P>|z|	[0.025	0.975]
beta.1	1.0000	0	0	0.000	1.000	1.000
beta.2	0	0	0	0.000	0	0

Cointegration relations for loading-coefficients-column 2

	coef	std err	z	P>|z|	[0.025	0.975]
beta.1	2.776e-17	0	0	0.000	2.78e-17	2.78e-17
beta.2	1.0000	0	0	0.000	1.000	1.000

```
test=results.test_causality('trf', ['trf', 'temp'], kind='wald')
print(test)
data["c"]
```

```
<statsmodels.tsa.vector_ar.hypothesis_test_results.CausalityTestResults object.
H_0: %s do not Granger-cause trf: reject at 5% significance level. Test
statistic: 529094.027, critical value: 18.307>, p-value: 0.000>

0      0.309571
1      0.083041
2      0.026940
3      0.132820
4      0.188829
         ...
145    0.390467
146    0.390467
147    0.390467
148    0.390467
149    0.390467
Name: c, Length: 150, dtype: float64
```

第 16 章 组合与计数时间序列

预测成分时间序列

Example 16.1

在前面的章节中， 我们考虑到时间序列通常没有任何限制，除了它们有一 个自然的下界，这通常是零。然而，有一些级数或级数的组受到进一步的 约束。在对这样的序列建模时，一个 "好的"模型应该不能预测违反已知 约束的值，也就是说，这个模型应该是 " 预测一致的 " 。"本章考虑这类 级数的两个例子:(1)复合时间序列，其中一组级数被定义为一个整体的份额， 因此它们必须是加起来等于一的正分数 ;(2)"计数"时间序列只能取正的、 通常较低的整数值。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
df=pd.read_excel("/root/experiment01/x1.xlsx")
x1=np.array(df["x1"])
x2=np.array(df["x2"])
x3=np.array(df["x3"])
plt.plot(x1)
plt.plot(x2)
plt.plot(x3)
plt.legend(['BMI<25',"BMI 25-30","BMI>30"])
plt.show()
```

!

```
y1 = np.log(x1/x3)
y2 = np.log(x2/x3)
plt.plot(y1)
plt.plot(y2)
plt.legend(["y1","y2"])
plt.show()
y1=pd.Series(y1).dropna()
y2=pd.Series(y2).dropna()
```

```
x1=np.array(range(len(y1)))
X=np.column_stack(((x1), (x1)**2, (x1)**3))

import statsmodels.api as sm
olsmod1 = sm.OLS(y1, X)
olsres1 = olsmod1.fit()
print(olsres1.summary())
olsmod2 = sm.OLS(y2, X)
olsres2 = olsmod2.fit()
print(olsres2.summary())
```

```
                                OLS Regression Results
```

```
==============================================================================
=======
Dep. Variable:                      y   R-squared (uncentered):
  0.720
Model:                            OLS   Adj. R-squared (uncentered):
0.678
Method:                 Least Squares   F-statistic:
17.16
Date:                Thu, 13 Aug 2020   Prob (F-statistic):
9.41e-06
Time:                        18:11:12   Log-Likelihood:
-8.5158
No. Observations:                  23   AIC:
23.03
Df Residuals:                      20   BIC:
26.44
Df Model:                           3

Covariance Type:            nonrobust

==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
x1             0.2832      0.061      4.629      0.000       0.156       0.411
x2            -0.0291      0.008     -3.443      0.003      -0.047      -0.011
x3             0.0008      0.000      2.842      0.010       0.000       0.001
==============================================================================
Omnibus:                       16.712   Durbin-Watson:                   0.154
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               17.328
Skew:                           1.699   Prob(JB):                     0.000173
Kurtosis:                       5.557   Cond. No.                     3.35e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 3.35e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
                                OLS Regression Results

==============================================================================
=======
Dep. Variable:                      y   R-squared (uncentered):
  0.769
Model:                            OLS   Adj. R-squared (uncentered):
0.734
Method:                 Least Squares   F-statistic:
22.19
Date:                Thu, 13 Aug 2020   Prob (F-statistic):
1.42e-06
Time:                        18:11:12   Log-Likelihood:
-3.2967
```

```
No. Observations:                  23   AIC:
12.59
Df Residuals:                      20   BIC:
16.00
Df Model:                           3

Covariance Type:            nonrobust

==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
x1             0.2471      0.049      5.068      0.000       0.145       0.349
x2            -0.0248      0.007     -3.689      0.001      -0.039      -0.011
x3             0.0007      0.000      3.000      0.007       0.000       0.001
==============================================================================
Omnibus:                       17.646   Durbin-Watson:                   0.167
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               19.068
Skew:                           1.754   Prob(JB):                     7.24e-05
Kurtosis:                       5.754   Cond. No.                     3.35e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 3.35e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

```
x2=np.array(range(len(y1)+5))
X2=np.column_stack(((x2), (x2)**2, (x2)**3))
y1_f = olsres1.predict(X2)
y2_f= olsres2.predict(X2)
```

```
x1_f = 100*np.exp(y1_f)/(1 + np.exp(y1_f) + np.exp(y2_f))
x2_f = 100*np.exp(y2_f)/(1 + np.exp(y1_f) + np.exp(y2_f))
x3_f = 100 -x1_f - x2_f
```

第十六章预测计数序列

Example 16.2

```
df=pd.read_excel("/root/experiment01/cons_share.xlsx")
x=np.array(df["other_share"])
c=np.array(df["cons_share"])
g=np.array(df["govt_share"])
i=np.array(df["inv_share"])
```

```
y1 = np.log(c/x)
y2 = np.log(i/x)
y3 = np.log(g/x)
plt.plot(y1)
plt.plot(y2)
plt.plot(y3)
plt.legend(["y1","y2","y3"])
plt.show()
y1=pd.Series(y1).dropna()
y2=pd.Series(y2).dropna()
y3=pd.Series(y3).dropna()
```

```
endog=np.column_stack((y1, y2, y3))
```

```
from statsmodels.tsa.api import VAR
model = VAR(endog)
results = model.fit(3)
print(results.summary())
```

```
  Summary of Regression Results
==================================
Model:                         VAR
Method:                        OLS
Date:           Thu, 13, Aug, 2020
Time:                     18:38:31
--------------------------------------------------------------------
No. of Equations:         3.00000    BIC:                   -19.4086
Nobs:                     247.000    HQIC:                  -19.6632
Log likelihood:           1428.16    FPE:                2.43171e-09
AIC:                     -19.8348    Det(Omega_mle):     2.15875e-09
--------------------------------------------------------------------
Results for equation y1
========================================================================
           coefficient       std. error           t-stat            prob
------------------------------------------------------------------------
const         0.221553         0.113086            1.959           0.050
L1.y1         0.559874         0.575851            0.972           0.331
L1.y2         0.033983         0.261057            0.130           0.896
```

```
L1.y3         0.204065         0.498390            0.409           0.682
L2.y1        -0.144935         0.691576           -0.210           0.834
L2.y2        -0.099647         0.328240           -0.304           0.761
L2.y3         0.415379         0.628964            0.660           0.509
L3.y1         0.380178         0.571015            0.666           0.506
L3.y2         0.044687         0.252893            0.177           0.860
L3.y3        -0.454614         0.495723           -0.917           0.359
================================================================================

Results for equation y2
================================================================================
           coefficient       std. error           t-stat            prob
--------------------------------------------------------------------------------
const         0.189667         0.115814            1.638           0.101
L1.y1        -0.026509         0.589741           -0.045           0.964
L1.y2         0.817267         0.267354            3.057           0.002
L1.y3        -0.015830         0.510412           -0.031           0.975
L2.y1        -0.385279         0.708258           -0.544           0.586
L2.y2         0.063043         0.336158            0.188           0.851
L2.y3         0.513862         0.644135            0.798           0.425
L3.y1         0.228954         0.584789            0.392           0.695
L3.y2         0.060600         0.258993            0.234           0.815
L3.y3        -0.312565         0.507680           -0.616           0.538
================================================================================

Results for equation y3
================================================================================
           coefficient       std. error           t-stat            prob
--------------------------------------------------------------------------------
const         0.202327         0.114515            1.767           0.077
L1.y1        -0.197635         0.583128           -0.339           0.735
L1.y2         0.002252         0.264356            0.009           0.993
L1.y3         1.007289         0.504688            1.996           0.046
L2.y1        -0.414684         0.700315           -0.592           0.554
L2.y2        -0.078056         0.332388           -0.235           0.814
L2.y3         0.653978         0.636912            1.027           0.305
L3.y1         0.423409         0.578231            0.732           0.464
L3.y2         0.057440         0.256089            0.224           0.823
L3.y3        -0.511647         0.501987           -1.019           0.308
================================================================================

Correlation matrix of residuals
            y1        y2        y3
y1    1.000000  0.970225  0.991760
y2    0.970225  1.000000  0.961452
y3    0.991760  0.961452  1.000000
```

Example 16.3

```
df=pd.read_excel("/root/experiment01/storms.xlsx")
df.head()
```

	date	storms	hurricanes
0	NaT	6	3
1	NaT	5	5
2	NaT	8	4
3	NaT	5	3
4	NaT	5	4

```
x=np.array(df["storms"])
r1 = 0.384
r2 = 0.300
phi2 = 0.179
score = np.sqrt(x)*r1
y = x -np.mean(x)
s = np.sum(y)

y2 = pd.Series(x).shift(-2) - np.mean(x)
y1 = y*y2
s1 = np.sum(y1)
qacf = (r2**2)*(s**2)/s1
qpacf = (phi2**2)*(s**2)/s1
```

```
# In-AR
```

```
r1 = 0.384
r2 = 0.300
a = (x*r1 + 1)/(x - 3)
l1 = (1 - a)*np.mean(x)
a1 = r1*(1 - r2)/(1 - r1**2)
a2 = (r2 - r1**2)/(1 - r1**2)
l2 =  (1 - a1 -a2)*np.mean(x)
w1 = x - a*pd.Series(x).shift(-1) - l1
w2 = x - a1*pd.Series(x).shift(-1) - a2*pd.Series(x).shift(-1) - l2
```

Example 16.4

```
from math import factorial
```

```
a = 0.397
lam = 5.78
xt = 17
x_max = 25
x_max_1 = x_max + 1
c=np.zeros(x_max)
b=np.zeros((x_max,x_max))
p=np.zeros((2,x_max_1))

for i in [1,2]:
    h = i
    for j in range(x_max):
        z = j
        c.append(0)
        for k in range(z):
            b[z][k] = (a**(k*h))*((1-(a**h))^(xt-k)*(lam**
(z*k))/(factorial(k)*factorial(z-k)*factorial(abs(xt-k)))
            c[z] = c[z] + b[z][k]

        scalar p[h][z] = factorial(xt)*exp(-lam*(1-(a**h))/(1-a)))*c[z]
    for f in range(x_max_1):
        w = f - 1
        #smpl !5 !5
        p[h] = p[h][w]
```

第17章 SPACE STATE模型与卡尔曼滤波

Space State模型

状态空间模型是动态时域模型，以隐含着的时间为自变量。状态空间模型在经济时间序列分析中的应用正在迅速增加。其中应用较为普遍的状态空间模型是由Akaike提出并由Mehra进一步发展而成的典型相关(canonical correlation)方法。由Aoki等人提出的估计向量值状态空间模型的新方法能得到所谓内部平衡的状态空间模型，只要去掉系统矩阵中的相应元素就可以得到任何低阶近似模型而不必重新估计，而且只要原来的模型是稳定的，则得到的低阶近似模型也是稳定的。

考虑以下状态空间模型: \begin{equation} \label{eq:ssmodel} \begin{split} y_t=z\alpha*{t-1}+e_t\ \alpha{t}=c+w \alpha_{t-1}+u_t \end{split} \end{equation} 我们把 αt−1 + ett 定义为状态变量(未观察到的变量)，把它定义为我们观察到的数据(有些人把它定义为观察方程)。注意，c 是一个常数，t = 1，2，n 是这样的时间，我们假设我们观察到 n 个观察值。假设 et 和 ut 是不相 关的正态分布变量，使得对于任意 j 和 l，etvN(0N(0，σe)和 utvN(0N(0，σu)的 E(et jut l)= 0。因为正态(或 高斯)变量的线性组合保持正常，所以向量 $\begin{bmatrix} y_t \\ \alpha_t \end{bmatrix}$

遵循二元正态分布。现在定义 E(αt−1 + ett)= c+We(αt−1 + ett 1)，我们得到前一个向量是有均值的正规向量 而向量的矩阵/协方差矩阵是$\Sigma = \begin{bmatrix} z^2E(\alpha_{t-1}-E[\alpha_{t-1}])^2+\sigma_e^2 & zwE(\alpha_{t-1}-E[\alpha_{t-1}])^2 \\ zwE(\alpha_{t-1}-E[\alpha_{t-1}])^2 & w^2E(\alpha_{t-1}-E[\alpha_{t-1}])^2+\sigma_u^2 \end{bmatrix}$

```
# 生成Space state模型
n=100
np.random.seed(1123)
from math import sqrt
e=sqrt(.8)*np.random.randn(n)
u=sqrt(.4)*np.random.randn(n)
y=np.zeros(n)
alpha=np.zeros(n)
y[0]=e[0]
alpha[0]=u[0]
for t in range(1,n):
    y[t]=alpha[t-1]+e[t]
    alpha[t]=0.9*alpha[t-1]+u[t]

plt.scatter(range(n),y,color="green")
plt.plot(alpha)
plt.show()
```

```
def StateSpaceGen(sigmae,sigmau,z,w,const):
    n<-100
    e<-sqrt(sigmae)*np.random.randn(n)
    u<-sqrt(sigmau)*np.random.randn(n)
    y=np.zeros(n)
    alpha=np.zeros(n)
    y[0]=e[0]
    alpha[0]=u[0]
    for t in range(1,n):
        y[t]=z*alpha[t-1]+e[t]
        alpha[t]=const+w*alpha[t-1]+u[t]

    return y,alpha
```

Space State模型的优点

- 状态空间模型不仅能反映系统内部状态，而且能揭示系统内部状态与外部的输入和输出变量的联系。
- 状态空间模型将多个变量时间序列处理为向量时间序列，这种从变量到向量的转变更适合解决多输入输出变量情况下的建模问题。
- 状态空间模型能够用现在和过去的最小信息形式描述系统的状态，因此，它不需要大量的历史数据资料，既省时又省力。

卡尔曼滤波

60年代Kalman把状态空间模型引入滤波理论，并导出了一套递推估计算法，后人称之为卡尔曼滤波理论。卡尔曼滤波是以最小均方误差为估计的最佳准则，来寻求一套递推估计的算法，其基本思想是：采用信号与噪声的状态空间模型，利用前一时刻地估计值和现时刻的观测值来更新对状态变量的估计，求出现时刻的估计值。它适合于实时处理和计算机运算。

卡尔曼滤波的实质是由量测值重构系统的状态向量。它以“预测—实测—修正”的顺序递推，根据系统的量测值来消除随机干扰，再现系统的状态，或根据系统的量测值从被污染的系统中恢复系统的本来面目。

```
def KF(sigmae,sigmau,z,w,const,y):

    a=np.zeros(n)
    p=np.zeros(n)
    a[0]=y[0]
    p[0]=10000
    if w<1:
        a[0]=0
        p[0]=sigmau/(1-w**2)
    k=np.zeros(n)
    v=np.zeros(n)
    for t in range(1,n):
        k[t]=(z*w*p[t-1])/(z**2*p[t-1]+sigmae)
        p[t]=w**2*p[t-1]-w*z*k[t]*p[t-1]+sigmau
        v[t]=y[t]-z*a[t-1]
        a[t]=w*a[t-1]+k[t]*v[t]

    return a
```

使用卡尔曼滤波预测AR序列

```
def ForecastARkf(y,h):

    def funcTheta(X):
        w=1-exp(-abs(X[2]))
        n=len(y)
        q=abs(X[0])
        co=abs(X[1])
        z=1
        likelihood=0
        sigmae=0
        a=np.zeros(n)
```

```
        p=np.zeros(n)
        a[0]=y[0]
        p[0]=10000;
        k=np.zeros(n)
        v=np.zeros(n)
        for t in range(1,n):

            k[t]=(z*w*p[t-1])/(z**2*p[t-1]+1)
            p[t]=w**2*p[t-1]-w*z*k[t]*p[t-1]+q
            v[t]=y[t]-z*a[t-1]
            a[t]=co+w*a[t-1]+k[t]*v[t]
            sigmae=sigmae+(v[t]**2/(z**2*p[t-1]+1))
            likelihood=likelihood+.5*np.log(2*pi)+.5+.5*np.log(z**2*p[t-1]+1)

        likelihood=likelihood+.5*n*np.log(sigmae/n)
        return likelihood
    res = minimize(funcTheta, [0.2,1,2])
    q=abs(res.x[0])
    co=abs(res.x[1]);
    z=1
    n=len(y)
    w=1-np.exp(res.x[2])
    sigmae=0
    a=np.zeros(n)
    p=np.zeros(n)
    a[0]=y[0]
    p[0]=100000
    k=np.zeros(n)
    v=np.zeros(n)
    sigmae<-0
    for t in range(1,n):
        k[t]=(z*w*p[t-1])/(z**2*p[t-1]+1)
        p[t]=w**2*p[t-1]-w*z*k[t]*p[t-1]+q
        v[t]=y[t]-z*a[t-1]
        a[t]=co+w*a[t-1]+k[t]*v[t]
        sigmae=sigmae+(v[t]**2/(z**2*p[t-1]+1))



    Forec=np.zeros(h)
    Forec[0]=a[len(y)-1]
    for i in range(1,h):
        Forec[i]=co+w*Forec[i-1];

    return Forec
```

Example 7.3

```
import numpy as np
from scipy import stats
import pandas as pd
import matplotlib.pyplot as plt

import statsmodels.api as sm

df=pd.read_csv("/root/experiment01/dataset/global_temps.csv")
col=df.columns
plt.plot(df[col[1]])
dta=df[col[1]]
```

```
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)
```

```
arima_mod = sm.tsa.statespace.SARIMAX(dta, order=(0,1,3),
trend='c').fit(disp=False)
print(arma_mod20.params)
```

```
intercept    0.000541
ma.L1       -0.504430
ma.L2       -0.090214
ma.L3       -0.116736
sigma2       0.015264
dtype: float64
```