高级量化交易技术

闫涛
科技有限公司
北京
{yt7589}@qq.com

第 1 章　时间序列基本特性

Abstract

在本章中我们将讨论时间序列的基本特性，包括自相关性和平稳性。

1　时间序列基本特性

时间序列的自相关性是指时间序列过去与未来存在某种关系，是我们时间序列预测的基础。主要用自协方差函数（Autocovariance Function, AF）、自相关系数函数（Autocorrelation Coefficent Function, ACF）和偏自相关系数函数（Partial Autocorrelation Coefficient Function, PACF）来描述。

1.1　随机变量统计量

随机变量 X 其取值为 x 的均值定义为：

$$E(x) = \mu \tag{1}$$

方差定义为：

$$\sigma^2(x) = E\left[\left(x-\mu\right)^2\right] \tag{2}$$

其中 $\sigma(x)$ 为标准差。对于两个随机变量 x 和 y，其协方差可以定义为：

$$\sigma(x,y) = E\left[\left(x-\mu_x\right)\left(y-\mu_y\right)\right] \tag{3}$$

在实际应用中，我们不可能知道真实的均值，只能使用统计量，因此协方差可以定义为：

$$Cov(x,y) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y}) \tag{4}$$

随机变量 x 和 y 的相关系数定义为：

$$\rho(x,y) = \frac{E[(x-\mu_x)(y-\mu_y)]}{\sigma_x\sigma_y} = \frac{\sigma(x,y)}{\sigma_x\sigma_y} \tag{5}$$

采用统计量的表示方法为：

$$Cor(x,y) = \frac{Cov(x,y)}{std(x)\times std(y)} \tag{6}$$

以上我们讨论的都是不同随机变量之间的关系，对于时间序列来说，我们可以把从不同时间点开始的子时间序列，视为不同的随机变量，那么我们就可以定义自协方差、自相关系数函数和偏自相关系数函数了。

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

1.2 时序序列平稳性

时序信号 x_t 的均值定义为:

$$E(x_t) = \mu(t) \tag{7}$$

时间序列的均值与所考虑的时间点有关。我们可以把时间序列上每个时间点都视为一个独立的时间变量，但是对于时间序列而言，每个时间点的随机变量只有一个观测值，怎么求出均值呢？在实际应用中，我们会将时间序列中的趋势信号（上涨或下跌）、季节性信号等从时间序列中去除掉，对于剩下的残差序列，我们可以视为其各个时间点上的随机变量的均值是不变的，于是就可以使用下面的公式来计算均值:

$$\bar{x} = \frac{1}{N}\sum_{t=1}^{N} x_t \tag{8}$$

由此我们引入平稳时间序列的概念，对于平稳时间序列，其各个时间点上对应的随机变量的均值相等。时间序列的方差可以定义为:

$$\sigma^2(t) = E[(x_t - \mu_t)^2] \tag{9}$$

根据上面平稳时间序列的定义，各个时间点对应的随机变量的均值不变，则式9可以化间为:

$$\sigma^2(t) = E[(x_t - \mu)^2] \tag{10}$$

我们同时规定，平稳时间序列各个时间点对应的随机变量的方差也不变，则10可进一步化简为:

$$Var(x_t) = \frac{1}{N-1}\sum_{t=1} N(x_t - \bar{x})^2 \tag{11}$$

1.3 自协方差

在讨论自协方差之前，我们首先要定义二阶平稳性。根据上一节定义，平稳时间序列是指各个时间点对应的随机变量的均值和方差相同。二阶平稳性是指在这一基础上，不同时间点对应的随机变量的相关系数只与时间相隔（lag）相关。注意：以下我们讨论的各种性质，均以此为前提。对于 lag=k 的自协方差定义为:

$$C_k = E[(x_t - \mu)(x_{t+k} - \mu)] \tag{12}$$

1.4 自相关系数函数 ACF

由于自协方差的大小与随机变量的大小有关，无法准确衡量其间的关系，因此我们引入自相关系数函数 ACF:

$$\rho_k = \frac{C_k}{\sigma^2} \tag{13}$$

由定义可知:

$$\begin{aligned}\rho_0 = \frac{C_0}{\sigma^2} &= \frac{E[(x_t - \mu)(x_t - \mu)]}{\sigma^2} \\ &= \frac{E[(x_t - \mu)^2]}{\sigma^2} = \frac{\sigma^2}{\sigma^2} = 1\end{aligned} \tag{14}$$

在实际应用中，我们都是处理的离散数据点，则自协方差可以定义为:

$$c_k = \frac{1}{N}\sum_{t=1}^{N}(x_t - \bar{x})(x_{t+k} - \bar{x}) \tag{15}$$

自相关系数函数 ACF 可以定义为:

$$r_k = \frac{c_k}{c_0} \tag{16}$$

1.5 自相关性举例

下面我们以上证综指时间序列为例，来看自相关系数函数 ACF 和偏自相关系数函数 PACF 的求法和作图，程序代码如下所示：

```python
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from matplotlib.font_manager import FontProperties
from statsmodels.tsa import stattools
from statsmodels.graphics import tsaplots

class Chp023(object):
    def __init__(self):
        self.name = 'Chp022'
        # 数据文件格式：编号 日期 星期几 开盘价 最高价
        # 最低价 收益价 收益
        self.data_file = 'data/pqb/chp023_001.txt'

    def startup(self):
        print('第23章：时间序列基本性质')
        data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
        sh_index = data[data.Indexcd==1]
        sh_index.index = pd.to_datetime(sh_index.index)
        sh_return = sh_index.Retindex
        print('时间序列长为：N={0}'.format(len(sh_return)))
        acfs = stattools.acf(sh_return)
        print(acfs)
        pacfs = stattools.pacf(sh_return)
        print(pacfs)
        tsaplots.plot_acf(sh_return, use_vlines=True, lags=30)
        plt.show()
        tsaplots.plot_pacf(sh_return, use_vlines=True, lags=30)
        plt.show()
```

Listing 1: 时间序列基本性质

数据文件格式为：

Figure 1: 数据文件格式

Indexcd	Trddt	Daywk	Opnindex	Hiindex	Loindex	Clsindex	Retindex
1	2014/1/2	4	2112.126	2113.11	2101.016	2109.387	-0.003115
1	2014/1/3	5	2101.542	2102.167	2075.899	2083.136	-0.012445
1	2014/1/6	1	2078.684	2078.684	2034.006	2045.709	-0.017967
1	2014/1/7	2	2034.224	2052.279	2029.246	2047.317	0.000786
1	2014/1/8	3	2047.256	2062.952	2037.11	2044.34	-0.001454
1	2014/1/9	4	2041.773	2057.196	2026.446	2027.622	-0.008178
1	2014/1/10	5	2023.535	2029.297	2008.007	2013.298	-0.007064
1	2014/1/13	1	2014.978	2027.181	2000.404	2009.564	-0.001855

其运行结果为：

Figure 2: 运行结果

```
量化投资以python为工具
第23章：时间序列基本性质
时间序列长为：N=311
[ 1.          0.03527714 -0.01178861 -0.02953388  0.16043181 -0.0506902
 -0.00557277  0.02556123  0.01763209  0.01170585  0.05137502 -0.03961812
  0.00219185  0.06976089  0.07020657 -0.0165844   0.09777829  0.10084446
  0.04706095 -0.05291647  0.08159786 -0.04505366  0.04894213 -0.11532665
  0.04273513 -0.04828588  0.04385656  0.06786466 -0.0642814  -0.05893641
 -0.107162    0.05179026 -0.04171157  0.08340151  0.05368795  0.04874785
 -0.04272709  0.03709712 -0.01313166 -0.04551243 -0.07680743]
[ 1.          0.03539094 -0.01313388 -0.02897258  0.16483494 -0.06656327
  0.00233848  0.03690183 -0.01674841  0.03307673  0.05240115 -0.05902877
  0.01423995  0.07184167  0.04734467  0.00131723  0.10809391  0.07778589
  0.03637376 -0.04041105  0.06533274 -0.07473672  0.05527639 -0.1271887
  0.02235463 -0.03279396  0.00769879  0.10885562 -0.10892202 -0.04912167
 -0.16407526  0.0362806  -0.04578521  0.106876    0.05526165  0.04072968
 -0.03517882  0.04134873 -0.02148459 -0.03297278 -0.08411357]
```

在图2中，第一个数据为自相关系数函数 ACF 各期的值，而第二个数组为偏自相关系数函数 PACF 各期的值。判断时间序列是否具有自相关性，可以看除 ACF 和 PACF 中，除第一个元素外，有没有显著超过阈值的元素，阈值定义为：

$$threshold = \frac{1.96}{\sqrt{N}} = \frac{1.96}{\sqrt{311}} = 0.11 \tag{17}$$

式17中的 N 为时间序列样本数，在本例中，共有 311 条记录，故 N=311，所以其阈值为 0.11 左右。由于 $acf[4] = 0.16 > 0.11$ 所以可以推断其具有自相关性，同时 $pacf[4] = 0.165 > 0.11$ 也可以推断其具有自相关性。我们还可以通过图形的方式形像的表示出来，自相关系数函数图如所示：

Figure 3: 自相关系数函数图

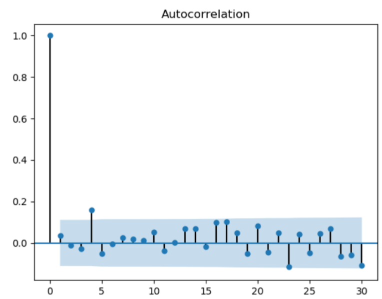
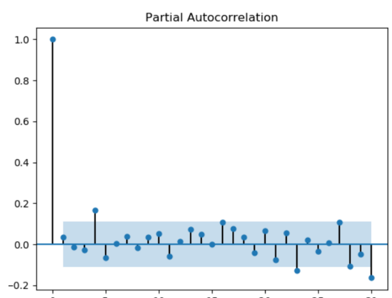


图3中蓝色区域的界限为 $[-\frac{1.96}{\sqrt{N}}, \frac{1.96}{\sqrt{N}}] = [-0.11, 0.11]$，除第 1 项外，其他项如果超出蓝色区域则说明此时间序列具有自相关性。
偏自相关系数函数图为：

Figure 4: 偏自相关系数函数图



1.6 白噪声和随机游走

1.6.1 残差序列定义

我们要对任意时间序列 y_t 进行建模，我们的模型为 $\hat{y}_t$，残差序列 x_t 可以定义为：$x_t = y_t - \hat{y}_t$，我们的任务就是使残差时间序列中每一个时间点对应的随机变量互相独立，没有自相关性，即满足独立同分布（Independent and Identical Distribution,I.I.D）条件。如果各个随机变量 $x_t \sim \mathbb{N}(0, \sigma^2)$，则称其为高斯白噪声。

1.6.2 差分运算符

为了后续讨论问题方便，我们首先定义 BSO 运算符：

$$Bx_t = x_{t-1} \quad B^n x_t = x_{t-n} \tag{18}$$

我们定义差分运算符为：

$$\begin{aligned} \nabla x_t &= x_t - x_{t-1} = (1-B)x_t \\ \nabla x_t^n &= \left(x_t - x_{t-n}\right)^n = (1-B)^n x_t \end{aligned} \tag{19}$$

1.6.3 白噪声定义

对于时间序列 $\{w_t, t=1,2,3,...,N\}$，满足 $\forall t \quad w_t \sim \mathcal{N}(0,\sigma^2)$ 且 $\forall i \neq j \quad Cor(w_i,w_j)=0$，则其为白噪声时间序列。
下面我们来看白噪声的二阶属性：

$$\mu = E(w_t) = 0$$
$$\gamma_k = Cor(w_t, w_{t+k}) = \begin{cases} 1 & if \quad k=0 \\ 0 & if \quad k \neq 0 \end{cases} \tag{20}$$

1.6.4 随机游走

随机游走（Random Walk）时间序列是指 x_t 可以定义为：$x_t = x_{t-1} + w_t$，其中 w_t 为白噪声时间序列。随机游走时间序列可以表示为：

$$\begin{aligned} x_t &= x_{t-1} + w_t = Bx_t + w_t \\ x_t &= x_{t-1} + w_t = x_{t-2} + w_{t-1} + w_t \\ &...... \\ x_t &= w_1 + w_2 + ... + w_{t-1} + w_t \end{aligned} \tag{21}$$

所以随机游走时间序列可以看作是多个白噪声时间序列的叠加。下面我们来看随机游走时间序列的均值和协方差：

$$\begin{aligned} \mu &= 0 \\ \gamma_k(t) &= Cov(x_t, x_{t+k}) = t\sigma^2 \end{aligned} \tag{22}$$

我们再来看随机游走序列的自相关系数函数 ACF：

$$\begin{aligned} \rho_k(t) &= \frac{Cov(x_t, x_{t+k})}{\sqrt{Var(x_t) \cdot Var(x_{t+k})}} \\ &= \frac{t\sigma^2}{\sqrt{t\sigma^2(t+k)\sigma^2}} = \frac{1}{\sqrt{1+\frac{k}{t}}} \end{aligned} \tag{23}$$

在通常情况下，t 比 k 要大得多，因此 ρ_k 会比较接近于 1。
下面我们来模拟一个随机游走时间序列信号，程序如下所示：

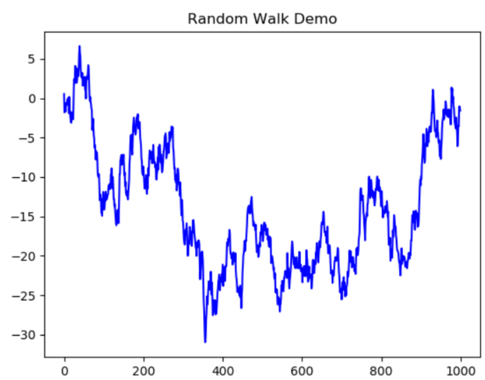
```python
    def random_wale_demo(self):
        '''
        随机游走时间序列建模示例
        '''
        w = np.random.standard_normal(size=1000)
        x = w
        for t in range(1, len(w)):
            x[t] = x[t-1] + w[t]
        plt.plot(x, c='b')
        plt.title('Random Walk Demo')
        plt.show()
        acfs = stattools.acf(x)
        print(acfs)
        tsaplots.plot_acf(x, use_vlines=True, lags=30)
        plt.show()
        # 拟合随机游走信号
        r = []
        for t in range(1, len(x)):
            r.append(x[t] - x[t-1])
        rd = np.array(r)
        plt.plot(rd, c='r')
        plt.title('Residue Signal')
        plt.show()
        rd_acfs = stattools.acf(rd)
        print(rd_acfs)
        tsaplots.plot_acf(rd, use_vlines=True, lags=30)
```

```
p l t . show ( )
```

Listing 2: 随机游走过程模拟

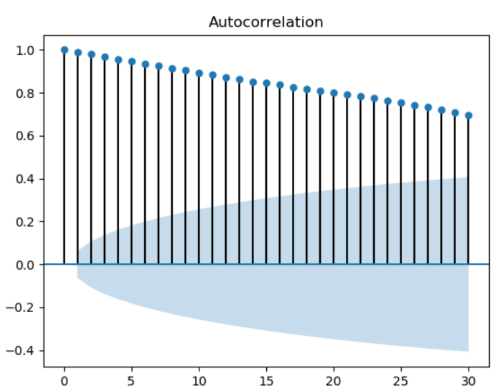
我们首先通过 $x_t = x_{t-1} + w_t$ 生成一个随机游走信号，该信号图形如下所示：

Figure 5: 随机游走时间序列信号



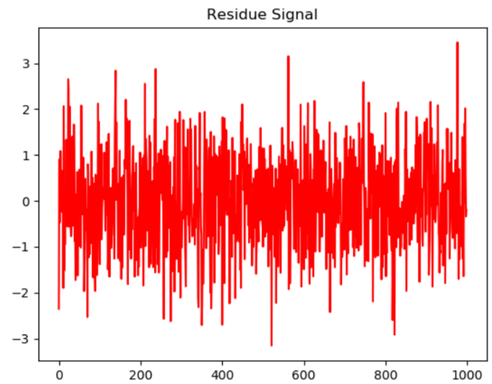
由图5可以看出，其非常像是一个股票收盘价的走势图，这也是为什么有些人说股票走势是随机游走过程了。接着我们求出该时间序列的自相关系数函数 ACF 及其自相关图，如下所示：

Figure 6: 随机游走时间序列信号



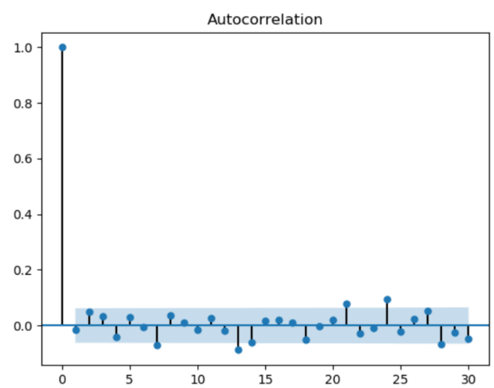
由图可以看出，其具有极强的自相关性，所有 ACF 值均位于蓝色置信区间之外。
我们知道 $x_t - x_{t-1} = w_t$，而 w_t 是白噪声时间序列信号，这实际上模拟了实际应用过程，我们把 x_t 视为实际的金融信号，而 x_{t-1} 为我们建模的信号，将两个信号相减，得到残差信号，如果残差信号是白噪声信号，就可以认为我们建模是合理的。下面来看我们得到的残差信号：

Figure 7: 残差时间序列信号



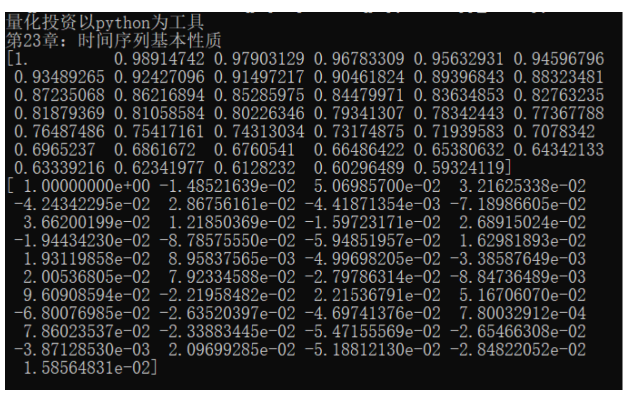
计算并绘制 ACF 如下所示：

Figure 8: 残差自相关系数函数



程序的运行结果如下所示：

Figure 9: 程序运行结果



```
量化投资以python为工具
第23章：时间序列基本性质
[1.         0.98914742 0.97903129 0.96783309 0.95632931 0.94596796
 0.93489265 0.92427096 0.91497217 0.90461824 0.89396843 0.88323481
 0.87235068 0.86216894 0.85285975 0.84479971 0.83634853 0.82763235
 0.81879369 0.81058584 0.80226346 0.79341307 0.78342443 0.77367788
 0.76487486 0.75417161 0.74313034 0.73174875 0.71939583 0.7078342
 0.6965237  0.6861672  0.6760541  0.66486422 0.65380632 0.64342133
 0.63339216 0.62341977 0.6128232  0.60296489 0.59324119]
[ 1.00000000e+00 -1.48521639e-02  5.06985700e-02  3.21625338e-02
 -4.24342295e-02  2.86756161e-02 -4.41871354e-03 -7.18986605e-02
  3.66200199e-02  1.21850369e-02 -1.59723171e-02  2.68915024e-02
 -1.94434230e-02 -8.78575550e-02 -5.94851957e-02  1.62981893e-02
  1.93119858e-02  8.95837565e-03 -4.99698205e-02 -3.38587649e-03
  2.00536805e-02  7.92334588e-02 -2.79786314e-02 -8.84736489e-03
  9.60908594e-02 -2.21958482e-02  2.21536791e-02  5.16706070e-02
 -6.80076985e-02 -2.63520397e-02 -4.69741376e-02  7.80032912e-04
  7.86023537e-02 -2.33883445e-02 -5.47155569e-02 -2.65466308e-02
 -3.87128530e-03  2.09699285e-02 -5.18812130e-02 -2.84822052e-02
  1.58564831e-02]
```

下面我们以上证综指收益率为例，来看随机游走模型是否可以很好的拟合这个时间序列，程序如下所示：

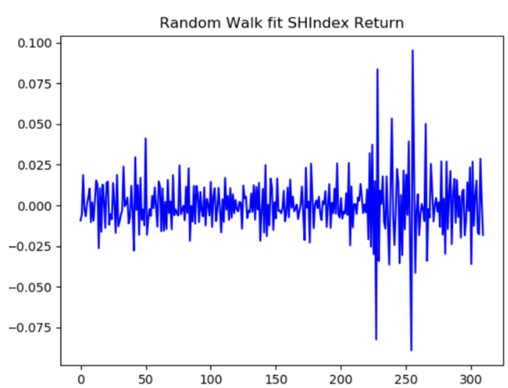
```python
    def random_walk_fit(self):
        data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
        sh_index = data[data.Indexcd==1]
        sh_index.index = pd.to_datetime(sh_index.index)
```

```python
        sh_return = sh_index.Retindex
        print('时间序列长为: N={0}'.format(len(sh_return)))
        r = []
        for t in range(1, len(sh_return)):
            r.append(sh_return[t] - sh_return[t-1])
        rd = np.array(r)
        plt.plot(rd, c='b')
        plt.title('Random Walk fit SHIndex Return')
        plt.show()
        rd_acfs = stattools.acf(rd)
        print(rd_acfs)
        tsaplots.plot_acf(rd, use_vlines=True, lags=30)
        plt.show()
```

Listing 3: 随机游走拟合上证综指收益率

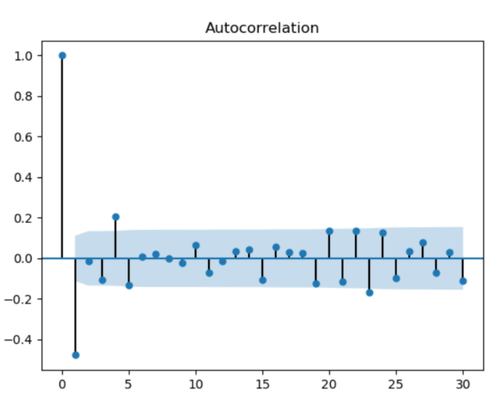
其残差图像为:

Figure 10: 残差图形



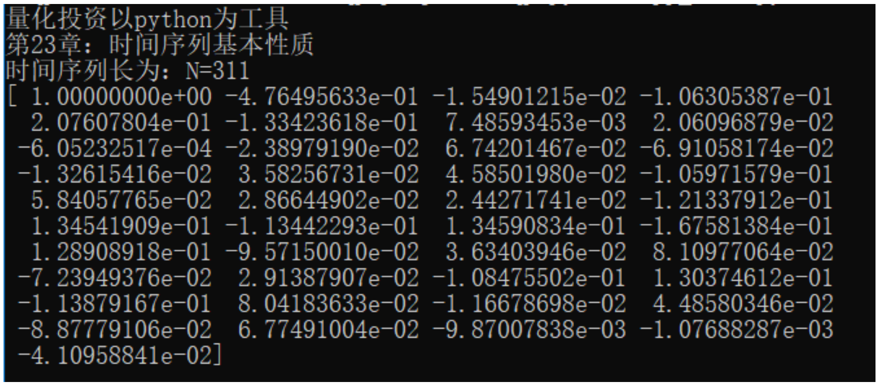
自相关系数函数 ACF 图形:

Figure 11: 自相关系数函数 ACF



由图11所示，在 1、4 时间点，明显超出置信范围，因此随机游走过程不能很好的拟合上证综指收益率时间序列信号。程序的运行结果如下所示:

Figure 12: 程序运行结果



```
量化投资以python为工具
第23章：时间序列基本性质
时间序列长为：N=311
[ 1.00000000e+00 -4.76495633e-01 -1.54901215e-02 -1.06305387e-01
  2.07607804e-01 -1.33423618e-01  7.48593453e-03  2.06096879e-02
 -6.05232517e-04 -2.38979190e-02  6.74201467e-02 -6.91058174e-02
 -1.32615416e-02  3.58256731e-02  4.58501980e-02 -1.05971579e-01
  5.84057765e-02  2.86644902e-02  2.44271741e-02 -1.21337912e-01
  1.34541909e-01 -1.13442293e-01  1.34590834e-01 -1.67581384e-01
  1.28908918e-01 -9.57150010e-02  3.63403946e-02  8.10977064e-02
 -7.23949376e-02  2.91387907e-02 -1.08475502e-01  1.30374612e-01
 -1.13879167e-01  8.04183633e-02 -1.16678698e-02  4.48580346e-02
 -8.87779106e-02  6.77491004e-02 -9.87007838e-03 -1.07688287e-03
 -4.10958841e-02]
```

第 2 章 ARIMA 模型

Abstract

在本章中我们将首先讲述自回归模型 AR(p)，接着讲述移动平均 MA(q)，最后讲解 ARMA(p,q)，然后将其泛化为 ARIMA(p,d,q)，分别将这些模型用于实际金融时间序列数据拟合。aqt001.py

2 ARIMA 模型

2.1 稳定性和模型选择标准

2.1.1 强稳定性

在我们以前的讨论中，我们说如果一个时间序列各个时间点所对应的随机变量，只要均值和方差不变，就是平稳时间序列。下面我们对强平稳性进行定义。
对于一个时间序列 $\{x_t\}$，如果对于 $\forall t_i, m$，两个序列：$x_{t_1}, x_{t_2}, ..., x_{t_N}$ 和 $x_{t_1+m}, x_{t_2+m}, ..., x_{t_N+m}$ 的统计特性完全相同，则说明该时间序列为强平稳特性。

2.1.2 模型选择标准

我们将用 AIC 来进行模型选择，AIC 的全称为：Akaike Information Criterion，我们通常会选择 AIC 值较小的模型。在实际应用中，还可以使用 BIC 来进行模型选择，BIC 的全称为 Bayes Information Criterion。在本章中我们只用 AIC 来进行模型选择。
假设统计模型的似然函数有 k 参数，最大似然值为 L，则 AIC 定义为：

$$AIC = -2\log(L) + 2k \tag{24}$$

由式24可知，最大似然值越大或者参数越少，AIC 的值越小，模型就越是好模型。

2.1.3 ADF 检验

在前面所讨论的问题中，我们通常根据自相关系数函数 ACF 和偏自相关系数函数 PACF 来判断稳定性，但是主观性比较强，我们需要一个客观的标准。
我们首先来定义时间序列的阶数，对于下面的非平稳时间序列：

$$x_t = x_{t-1} + w_t \tag{25}$$

其中 $w_t \sim \mathcal{N}(0, \sigma^2)$ 为白噪声信号，且 $x_0 = 0$, 我们可以得到其均值为：

$$E(x_t) = E(x_{t-1} + w_t) = E(x_{t-1}) + E(w_t) = E(x_{t-1}) = ... = E(x_0) = 0 \tag{26}$$

同样我们可以得到其方差：

$$Var(x_t) = Var(x_{t-1} + w_t) = Var(x_{t-1}) + Var(w_t) = Var(x_{t-1}) + \sigma^2 = ... = t\sigma^2 \tag{27}$$

x_t 由于其各时间点对应的随机变量的方差随时间变化，因此不是平稳时间序列。
我们定义 1 阶差分算子：

$$\nabla x_t = B x_t = x_t - x_{t-1} = w_t \tag{28}$$

对于 Bx_t 为白噪声信号，其显然是平稳时间序列，所以我们称 x_t 为 I(1) 的非平稳时间序列。
我们可以将其定义扩展到 n 阶：

$$Bx_t = x_{t-1} \quad B^2 x_t = x_{t-2} \quad B^3 x_t = x_{t-3} \quad ... \quad B^n x_t = x_{t-n} \tag{29}$$

我们还以上面的时间序列 $x_t = x_{t-1} + w_t$ 为例，我们可以将其写为：

$$x_t - x_{t-1} = x_t - Bx_t = (1-B)x_t = w_t \tag{30}$$

式30中 $1-B$ 为滞后算子多项式，我们令 $1-B=0$ 得出的解为 $B=1$，其为单位根，所以其为非平稳时间序列。这一结论可以推广到更一般的情况，对于如下所示的时间序列：

$$y_t = (1+\rho)y_{t-1} - \rho y_{t-2} + w_t \tag{31}$$

其所对应的滞后算子多项式为：

$$y_t - (1-\rho)By_t + \rho B^2 y_t = w_t \tag{32}$$

令式32左边为 0，得到的解为：$B=1$ 和 $B=\frac{1}{\rho}$，因为其存在单位根，所以其不是平稳时间序列。
对于任意如下所示时间序列：

$$y_t = \gamma + \rho_1 y_{t-1} + \rho_2 y_{t-2} + ... + \rho_p y_{t-p} + w_t \tag{33}$$

其中 $w_t \sim \mathcal{N}(0, \sigma^2)$ 为独立同分布（i.i.d）噪声信号。可以将式33改写为如下形式：

$$y_t = \gamma + \rho_1 B y_t + \rho_2 B^2 y_t + ... + \rho_p B^p y_t + w_t \tag{34}$$

将式34右边所有包含 y_t 的项都移到左边，可以得到下式：

$$(1 - \rho_1 B - \rho_2 B^2 - .. - \rho_p B^p) y_t = \gamma + w_t \tag{35}$$

可以得到其对应的滞后算子多项式方程为：

$$1 - \rho(B) = 1 - \rho_1 B - \rho_2 B^2 - .. - \rho_p B^p = 0 \tag{36}$$

解这个方程，如果所有解的绝对值均大于 1，则该时间序列为平稳时间序列，如果存在单位根或绝对值小于 1 的根，则其为非平稳时间序列。
以上我们讲解的判断时间序列平稳性的原理，在实际应用中，我们通常采用 ADF 来判断时间序列的平稳性，ADF 模型如下所示：

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \delta_1 \Delta y_{t-1} + \delta_2 \Delta y_{t-2} + ... + \delta_p \Delta y_{t-p} + w_t \tag{37}$$

其中 α 对应截距，β 对应趋势，$\delta_1 \Delta y_{t-1} + \delta_2 \Delta y_{t-2} + ... + \delta_p \Delta y_{t-p}$ 为 ADF 的增广项，p 为增广项的期数，其值由 AIC 或 BIC 算法来决定。原假设 H_0 为该序列有单位根是非平稳时间序列：$\gamma = 0$；备择假设 H_1 为该序列为平稳时间序列：$\gamma < 0$。
这部分原理比较复杂，我们在实际应用中，通常使用 arch 包中的 ADF 函数来完成检验工作，其函数定义为：

$$ADF(y, lags, trend, max_lags, method) \tag{38}$$

其中：

- y：待判断的时间序列；
- lags：滞后期数
- trend：用来控制检验模型的类型
  - 'nc'：不含截距项；
  - 'c'：含截距项；
  - 'ct'：包含截距项和线性趋势项；
  - 'ctt'：包含截距项和线性趋势项以及二次趋势项；
- max_lags：最大期数
- method：常用方法为：'aic'、'bic'、't_stat'

下面我们通过一个例子来看怎样使用 ADF 方法，我们以上证综指收益率和收盘价这两个序列为例，我们首先需要安装 python 的 garch 库：

```
pip install arch
```

Listing 4: 随机游走拟合上证综指收益率

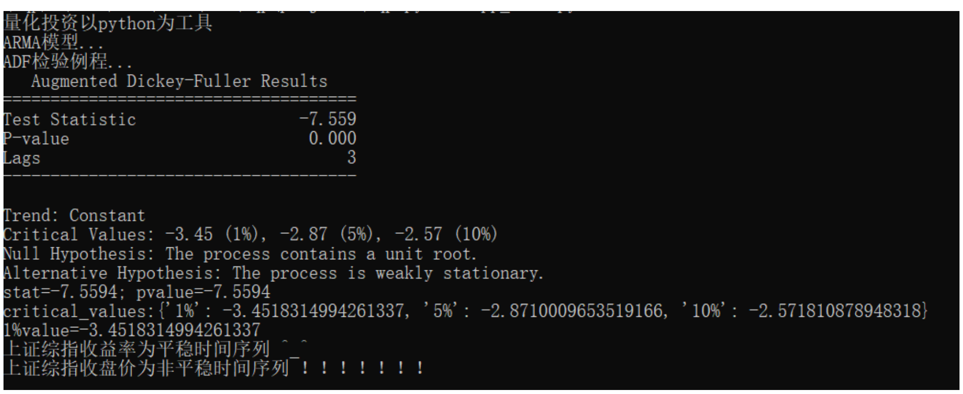
程序代码如下所示：

```python
import arch.unitroot as unitroot
......
    def adf_demo(self):
        print('ADF检验例程...')
        data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
        sh_index = data[data.Indexcd==1]
        sh_index.index = pd.to_datetime(sh_index.index)
        sh_return = sh_index.Retindex
        sh_return_adf = unitroot.ADF(sh_return)
        print(sh_return_adf.summary().as_text())
        print('stat={0:0.4f}; pvalue={0:0.4f}'.format(sh_return_adf.stat,
     sh_return_adf.pvalue))
        print('critical_values:{0}'.format(sh_return_adf.critical_values)
    )
        print('1%value={0}'.format(sh_return_adf.critical_values['1%']))
        if sh_return_adf.stat < sh_return_adf.critical_values['1%']:
            print('上证综指收益率为平稳时间序列 ^_^')
        else:
            print('上证综指收益率为平稳时间序列  !!!!!!!!!')
        sh_close = sh_index.Clsindex
        sh_close_adf = unitroot.ADF(sh_close)
        if sh_close_adf.stat < sh_close_adf.critical_values['1%']:
            print('上证综指收盘价为平稳时间序列 ^_^')
        else:
            print('上证综指收盘价为非平稳时间序列 ！！！！！！！')
```

Listing 5: ADF 检验

运行结果如下所示：

Figure 13: 自相关系数函数 ACF


```
量化投资以python为工具
ARMA模型...
ADF检验例程...
   Augmented Dickey-Fuller Results
=====================================
Test Statistic                 -7.559
P-value                         0.000
Lags                                3
-------------------------------------

Trend: Constant
Critical Values: -3.45 (1%), -2.87 (5%), -2.57 (10%)
Null Hypothesis: The process contains a unit root.
Alternative Hypothesis: The process is weakly stationary.
stat=-7.5594; pvalue=-7.5594
critical_values:{'1%': -3.4518314994261337, '5%': -2.8710009653519166, '10%': -2.571810878948318}
1%value=-3.4518314994261337
上证综指收益率为平稳时间序列 ^_^
上证综指收盘价为非平稳时间序列！！！！！！！
```

由上面的结果可以看出，上证综指收益率是稳定的时间序列，收盘价却是不稳定的时间序列。

2.2 自回归模型

2.2.1 背景

我们可以扩展随机游走模型，使当前时间点数据不仅依赖前一时间点的值，同时还依赖前 p 个时间点的值，是这 p 个值的线性组合，这就得到了自回归模型。

2.2.2 模型定义

自回归模型 AR(p) 是随机游走模型的扩展，p 阶自回归模型定义为：

$$x_t = \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + ... + \alpha_p x_{t-p} + w_t = \sum_{i=1}^{p} \alpha_i x_{t-i} + w_t \tag{39}$$

其中 $\{w_t\}$ 为白噪声，$\alpha_i \in R$ 且 $\alpha_p \neq 0$。
当 $p=1$ 且 $\alpha_1 = 1$ 时，自回归模型就退化为随机游走模型。
为后续讨论方便，我们定义如下运算符：

$$\theta_p(B)x_t = (1 - \alpha_1 B - \alpha_2 B^2 - ... - \alpha_p B^p)x_t = w_t \tag{40}$$

有了上述模型之后，我们就可以直接拿来作预测，如下所示：

$$\begin{aligned} \hat{x}_t &= \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + ... + \alpha_p x_{t-p} \\ \hat{x}_{t+1} &= \alpha_1 \hat{x}_t + \alpha_2 x_{t-1} + ... + \alpha_p x_{t-p+1} \end{aligned} \tag{41}$$

然后依此类推，可以求出其后 n 个时间点的预测值。

2.2.3 二阶特性

我们首先定义特性方程：

$$\theta_p(B) = 0 \tag{42}$$

解这个方程得到的解的绝对值必须大于才是平稳序列。我们可以举几个实例，首先是随机游走序列：

随机游走 根据定义 $x_t = x_{t-1} + w_t$ 我们可以得到 $\alpha_1 = 1$，其特性方程为 $\theta = 1 - B = 0$，其解为 $B = 1$，因为其解的绝对值不大于 1，所以其不是平稳模型。

1 阶自回归 我们假设自回归模型为 $x_t = \frac{1}{4}x_{t-1} + w_t$，其中 $\alpha_1 = \frac{1}{4}$，其特性方程为 $\theta = 1 - \frac{1}{4} = 0$，其解为 $B = 4$，该解绝对值大于 1，所以其是平稳模型。

2 阶自回归模型 我们假设自回归模型为 $x_t = \frac{1}{2}x_{t-1} + \frac{1}{2}x_{t-2} + w_t$，其特性方程为 $\theta_2(B) = \frac{1}{2}(1-B)(B+2) = 0$，则其解为 $B = 1, -2$，其中一个解为单位根，其绝对值不大于 1，因此本模型不是平稳模型。虽然本模型不是平稳模型，但是其他 2 阶自回归模型是完全有可能是平稳模型的。

二阶特性 均值、自协方差、自相关系数定义如下所示：

$$\begin{aligned} \mu_x &= E(x_t) = 0 \\ \gamma_k &= \sum_{i=1}^{p} \alpha_i \gamma_{k-i}, \quad k > 0 \\ \rho_k &= \sum_{i=1}^{p} \alpha_i \rho_{k-i}, \quad k > 0 \end{aligned} \tag{43}$$

2.2.4 模拟数据

下面我们来模拟一个 AR(2) 的时间序列，我们的数据生成和拟合代码如下所示：

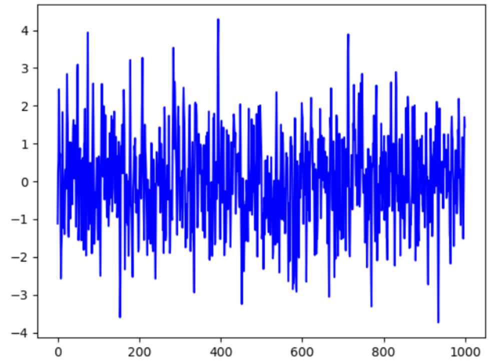
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from matplotlib.font_manager import FontProperties
from statsmodels.tsa import stattools
from statsmodels.graphics import tsaplots
from statsmodels.tsa.arima_model import ARIMA

```

```python
class Aqt001(object):
    def __init__(self):
        self.name = 'Aqt001'

    def startup(self):
        print('ARMA模型...')
        self.simulate_ar2()

    def simulate_ar2(self):
        print('模拟AR(2)')
        alpha1 = 0.666
        alpha2 = -0.333
        wt = np.random.standard_normal(size=1000)
        x = wt
        for t in range(2, len(wt)):
            x[t] = alpha1 * x[t-1] + alpha2 * x[t-2] + wt[t]
        plt.plot(x, c='b')
        plt.show()
        ar2 = stattools.ARMA(x, (2, 0)).fit(disp=False)
        print('p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'.format(
                    ar2.k_ar, ar2.arparams, ar2.k_ma, ar2.maparams,
                    ar2.aic, ar2.bic, ar2.hqic)
        )
        arima2_0_0 = ARIMA(x, order=(2, 0, 0)).fit(disp=False)
        print('ARIMA: p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'. \
                    format(arima2_0_0.k_ar, arima2_0_0.arparams,
                    arima2_0_0.k_ma, arima2_0_0.maparams,
                    arima2_0_0.aic, arima2_0_0.bic,
                    arima2_0_0.hqic)
        )
        resid = arima2_0_0.resid
        # 绘制ACF
        acfs = stattools.acf(resid)
        print(acfs)
        tsaplots.plot_acf(resid, use_vlines=True, lags=30)
        plt.title('ACF figure')
        plt.show()
        pacfs = stattools.pacf(resid)
        print(pacfs)
        tsaplots.plot_pacf(resid, use_vlines=True, lags=30)
        plt.title('PACF figure')
        plt.show()
```

Listing 6: AR 数据模拟和拟合示例

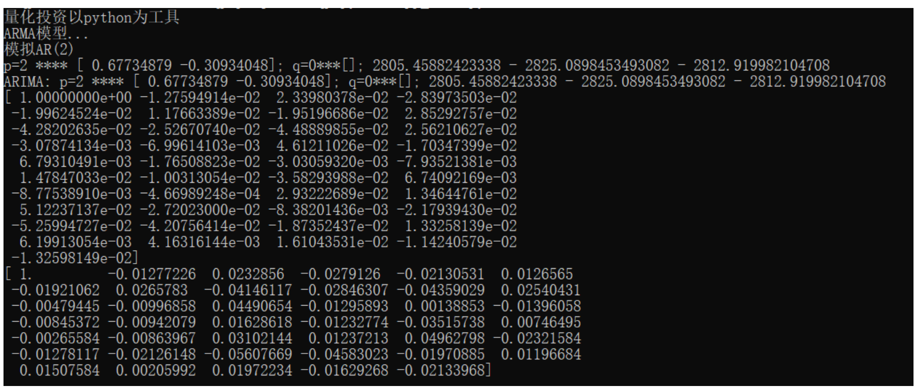
我们首先生成一个 1000 个数据点的均值为 0 方差为 1 的白噪声数据，然后根据 $x_t = \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + w_t = 0.666 \times x_{t-1} - 0.333 \times x_{t-2} + w_t$ 公式，生成拟合数据。我们绘制该模拟数据图像，接着我们分别用 ARMA 和 ARIMA 进行拟合，求出系数，然后计算出模拟选择的参数：AIC、BIC、HQIC 的值，最后我们求出模型拟合的残差，并绘制出残差自相关系数函数 ACF 和偏自相关系数函数 PACF 的图像。模拟数据图像为：

Figure 14: AR2 模拟生成数据图



无论是 ARMA 还是 ARIMA 拟合，我们都可以得到较为正确的数据，同时残差的 ACF 和 PACF 也表明其是白噪声序列，运行结果如下所示：

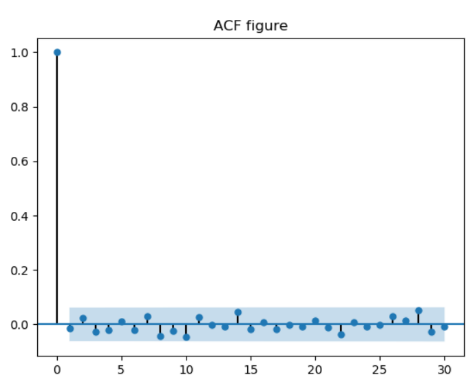
Figure 15: 程序运行结果



```
量化投资以python为工具
ARMA模型...
模拟AR(2)
p=2 **** [ 0.67734879 -0.30934048]; q=0***[]; 2805.45882423338 - 2825.0898453493082 - 2812.919982104708
ARIMA: p=2 **** [ 0.67734879 -0.30934048]; q=0***[]; 2805.45882423338 - 2825.0898453493082 - 2812.919982104708
[ 1.00000000e+00 -1.27594914e-02  2.33980378e-02 -2.83973503e-02
 -1.99624524e-02  1.17663389e-02 -1.95196686e-02  2.85292757e-02
 -4.28202635e-02 -2.52670740e-02 -4.48889855e-02  2.56210627e-02
 -3.07874134e-03 -6.99614103e-03  4.61211026e-02 -1.70347399e-02
  6.79310491e-03 -1.76508823e-02 -3.03059320e-03 -7.93521381e-03
  1.47847033e-02 -1.00313054e-02 -3.58293988e-02  6.74092169e-03
 -8.77538910e-03 -4.66989248e-04  2.93222689e-02  1.34644761e-02
  5.12237137e-02 -2.72023000e-02 -8.38201436e-03 -2.17939430e-02
 -5.25994727e-02 -4.20756414e-02 -1.87352437e-02  1.33258139e-02
  6.19913054e-03  4.16316144e-03  1.61043531e-02 -1.14240579e-02
 -1.32598149e-02]
[ 1.         -0.01277226  0.0232856  -0.0279126  -0.02130531  0.0126565
 -0.01921062  0.0265783  -0.04146117 -0.02846307 -0.04359029  0.02540431
 -0.00479445 -0.00996858  0.04490654 -0.01295893  0.00138853 -0.01396058
 -0.00845372 -0.00942079  0.01628618 -0.01232774 -0.03515738  0.00746495
 -0.00265584 -0.00863967  0.03102144  0.01237213  0.04962798 -0.02321584
 -0.01278117 -0.02126148 -0.05607669 -0.04583023 -0.01970885  0.01196684
  0.01507584  0.00205992  0.01972234 -0.01629268 -0.02133968]
```

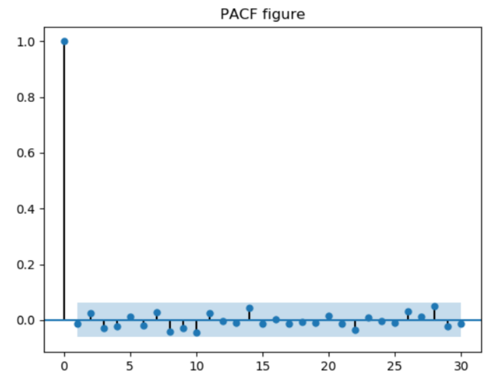
残差的自相关系数函数 ACF 图：

Figure 16: 残差的自相关系数函数 ACF 图



残差的偏自相关系数函数 PACF 图：

Figure 17: 残差的偏自相关系数函数 PACF 图



由 ACF 和 PACF 图可以看出，我们残差是比较典型的随机白噪声序列，由此可见我们拟合还是很好的。

2.2.5 展望

在理解了基本理论之后，我们将引入最终的 ARIMA 模型，并用 ARIMA 模型来拟合真实上证综指收盘价时间序列，并用我们的拟合模型来预测最后 5 日的收盘价，在这个实际例子中，看我们模型的表现如何。

2.3 ARIMA 模型

2.3.1 背景

我们不仅可以对当前时间点数值对之前时间点数值进行建模，我们也可以对前面时间点随机噪声对当前时间点的影响进行建模，同时由于原始信号可能非常不平稳，但是我们求出其差值序列后，可能就变为平稳序列了，这就是 ARIMA 模型要解决的问题。为了讨论 ARIMA 模型，我们首先介绍差分的概念：

$$\begin{aligned} \{x_1, x_2, ..., x_N\} \\ \{d_1^1, d_2^1, ..., d_{N-1}^1\} = \{x_2 - x_1, x_3 - x_2, ..., x_N - x_{N-1}\} \\ \{d_1^2, d_2^2, ..., d_{N-1}^2\} = \{d_2^1 - d_1^1, d_3^1 - d_2^1, ..., d_{N-1}^1 - d_{N-1}^1\} \end{aligned} \tag{44}$$

上式中分别为原始时序信号，然后是一阶差分和二阶差分。

2.3.2 模型选择标准

BIC 定义 我们已经介绍过一个模型选择标准 AIC（Akaike Information Criterion），其会惩罚参数多的模型，因为这些模型容易产生过拟合（Overfitting）。接下来我们要介绍另一个模型选择参数 BIC（Bayes Information Criterion），与 AIC 相比，其会更倾向于惩罚参数多的模型，同样是值越小越好，BIC 定义如下所示：

$$BIC = -2\log(L) + k\log N \tag{45}$$

其中 L 为似然函数的最大值，k 为模型的参数，N 为数据点个数。

Ljung-Box 检测 我们的缺省假设 H_0 为：对于一个拟合的时序信号，对所有滞后时点 lags，都是独立同分布（i.i.d）的，即不存在相关性。
备择假设 H_a 为：这些信号不是独立同分布（i.i.d）的，具有相关性。
我们定义统计量 Q：

$$Q = n(n+2)\sum_{k=1}^{h} \frac{\hat{\rho}_k^2}{n-k} \tag{46}$$

式46中 n 为时间序列长度，h 为最大滞后期数，ρ_k 第 k 自相关系数。Ljung-Box 检测原理比较复杂，但是在 python 语言中，经过运算可以求出 Q 值，以及大于 Q 值的概率，实际上我们看 1~12 滞后期的 Q 值和大于 Q 值的概率，如果该概率小于显著水平如 0.05 时，就拒绝缺省假设（不存在相关性），选择备择假设，否则反之。

2.3.3 定义

同时考虑之前时间点的信号和噪声值，我们就可以得到如下 ARMA 模型：

$$x_t = \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + ... + \alpha_p x_{t-p} + w_t + \beta_1 w_{t-1} + \beta_2 w_{t-2} + ... + +\beta_q w_{t-q} \tag{47}$$

如果我们对欲研究的信号求出一阶或二阶差分，然后再利用用式47的模型，就是 ARIMA 模型了。其中 $\{w_t\}$ 为白噪声，其均值为 0，方差为 σ^2。
其特性方程可以表示为：

$$\theta_p(B)x_t = \phi_q(B)w_t \tag{48}$$

由上面的讨论可以看出，AR(p) 和 MA(q) 都是 ARIMA 模型的特殊情况，在同样精度的条件下，ARIMA 模型所需参数最小。

2.3.4 数据仿真

在理解了 ARIMA 模型定义之后，我们来模拟一下 ARIMA 过程。假设我们要模拟的 ARIMA 模型为：

$$\begin{aligned} x_t &= \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + w_t + \beta_1 w_{t-1} + \beta_2 w_{t-2} \\ &= 1.2 \times x_{t-1} - 0.7 \times x_{t-2} + w_t - 0.06 \times w_{t-1} - 0.02 \times w_{t-2} \end{aligned} \tag{49}$$

生成模拟数据并利用 ARIMA 拟合的程序如下所示：

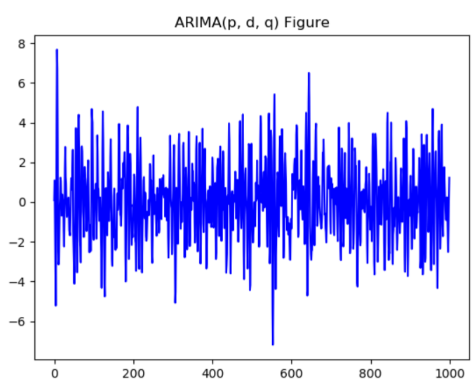
```
    def simulate_arima_p_d_q(self):
        print('模拟ARIMA(p,d,q)过程')
        np.random.seed(8)
        alpha1 = 1.2
        alpha2 = -0.7
        beta1 = -0.06
        beta2 = -0.02
        w = np.random.standard_normal(size=1000)
        x = w
        for t in range(2, len(w)):
            x[t] = alpha1 * x[t-1] + alpha2*x[t-2] + w[t] + beta1 * w[t
    -1] + beta2*w[t-2]
        plt.plot(x, c='b')
        plt.title('ARIMA(p, d, q) Figure')
        plt.show()
        # 查看ACF
        acfs = stattools.acf(x)
        print('ARIMA(q,d,q) ACFS:\r\n{0}'.format(acfs))
        tsaplots.plot_acf(x, use_vlines=True, lags=30)
        plt.title('ARIMA(p,d,q) ACF')
        plt.show()
        # ARIMA拟合
        min_ABQIC = sys.float_info.max
        arima_model = None
        break_loop = False
        '''
        for p in range(0, 5):
            if break_loop:
                break
            for q in range(0, 5):
                print('try {0}, d, {1}...'.format(p, q))
                try:
                    arima_p_d_q = ARIMA(x, order=(p, 0, q)).fit(disp=
    False)
                    print('..... fit ok')
                    if arima_p_d_q.aic < min_ABQIC:
                        print('..... record good model')
                        min_ABQIC = arima_p_d_q.aic
                        arima_model = arima_p_d_q
                        #if 1==p and 1==q:
                        #    break_loop = True
                except Exception as ex:
                    print('.....!!!!!! Exception')
```

```
        print('ARIMA: p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'. \
                     format(arima_model.k_ar, arima_model.arparams,
                     arima_model.k_ma, arima_model.maparams,
                     arima_model.aic, arima_model.bic,
                     arima_model.hqic)
        )
        '''
        arima_model = ARIMA(x, order=(2, 0, 2)).fit(disp=False)
        print('God_View:ARIMA: p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'. \
                     format(arima_model.k_ar, arima_model.arparams,
                     arima_model.k_ma, arima_model.maparams,
                     arima_model.aic, arima_model.bic,
                     arima_model.hqic)
        )
        resid = arima_model.resid
        # 绘制ACF
        acfs = stattools.acf(resid)
        print(acfs)
        tsaplots.plot_acf(resid, use_vlines=True, lags=30)
        plt.title('ARIMA(p,d,q) ACF figure')
        plt.show()
        pacfs = stattools.pacf(resid)
        print(pacfs)
        tsaplots.plot_pacf(resid, use_vlines=True, lags=30)
        plt.title('ARIMA(p,d,q) PACF figure')
        plt.show()
```

Listing 7: AR 数据模拟和拟合示例

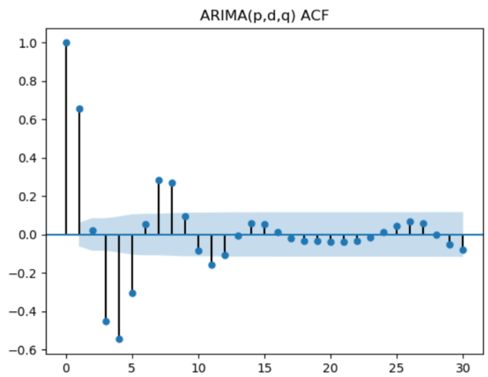
生成的时序信号 x 为:

Figure 18: 原始模拟信号图



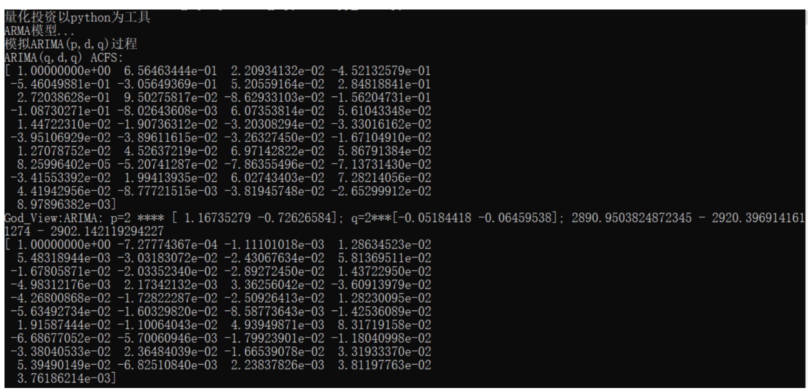
该信号的自相关系数函数 ACF 图为:

Figure 19: 原始模拟信号 ACF 图



由图中可以看出，该信号具有非常强的自相关性。
接着我们用 ARIMA 模型来模拟该信号，上面程序注释部分为求最佳 ARIMA 模型的 p 和 q 参数，以 AIC 作为模型选择标准，因为我们知道模型为 ARIMA(2,0,2)，所以我们同时也用 ARIMA(2,0,2) 来进行拟合，程序运行结果如下所示：

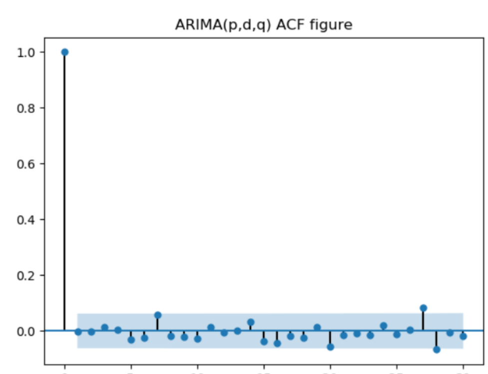
Figure 20: 程序运行结果



```
量化投资以python为工具
ARMA模型...
模拟ARIMA(p,d,q)过程
ARIMA(q,d,q) ACFS:
[ 1.00000000e+00  6.56463444e-01  2.20934132e-02 -4.52132579e-01
 -5.46049881e-01 -3.05649369e-01  5.20559164e-02  2.84818841e-01
  2.72038628e-01  9.50275817e-02 -8.62933103e-02 -1.56204731e-01
 -1.08730271e-01 -8.02643608e-03  6.07353814e-02  5.61043348e-02
  1.44722310e-02 -1.90736312e-02 -3.20308294e-02 -3.33016162e-02
 -3.95106929e-02 -3.89611615e-02 -3.26327450e-02 -1.67104910e-02
  1.27078752e-02  4.52637219e-02  6.97142822e-02  5.86791384e-02
  8.25996402e-05 -5.20741287e-02 -7.86355496e-02 -7.13731430e-02
 -3.41553392e-02  1.99413935e-02  6.02743403e-02  7.28214056e-02
  4.41942956e-02 -8.77721515e-03 -3.81945748e-02 -2.65299912e-02
  8.97896382e-03]
God_View:ARIMA: p=2 **** [ 1.16735279 -0.72626584]; q=2***[-0.05184418 -0.06459538]; 2890.9503824872345 - 2920.396914161
1274 - 2902.142119294227
[ 1.00000000e+00 -7.27774367e-04 -1.11101018e-03  1.28634523e-02
  5.48318944e-03 -3.03183072e-02 -2.43067634e-02  5.81369511e-02
 -1.67805871e-02 -2.03352340e-02 -2.89272450e-02  1.43722950e-02
 -4.98312176e-03  2.17342132e-03  3.36256042e-02 -3.60913979e-02
 -4.26800868e-02 -1.72822287e-02 -2.50926413e-02  1.28230095e-02
 -5.63492734e-02 -1.60329820e-02 -8.58773643e-03 -1.42536089e-02
  1.91587444e-02 -1.10064043e-02  4.93949871e-03  8.31719158e-02
 -6.68677052e-02 -5.70060946e-03 -1.79923901e-02 -1.18040998e-02
 -3.38040533e-02  2.36484039e-02 -1.66539078e-02  3.31933370e-02
  5.39490149e-02 -6.82510840e-03  2.23837826e-03  3.81197763e-02
  3.76186214e-03]
```

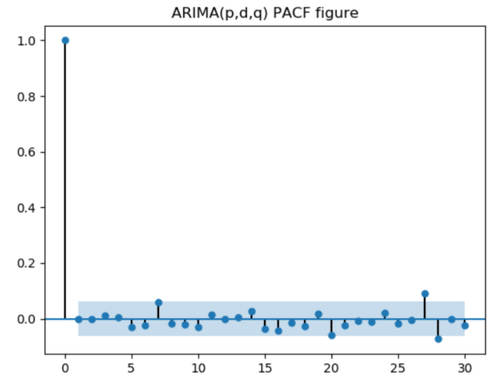
接着我们求出残差序列，残差序列自相关系数函数 ACF 图如下所示：

Figure 21: 残差自相关系数函数 ACF 图



残差序列偏自相关系数函数图 PACF 如下所示：

Figure 22: 残差偏自相关系数函数 ACF 图



由图中可以看出，残差序列基本上是白噪声信号。

2.3.5 金融数据拟合及预测

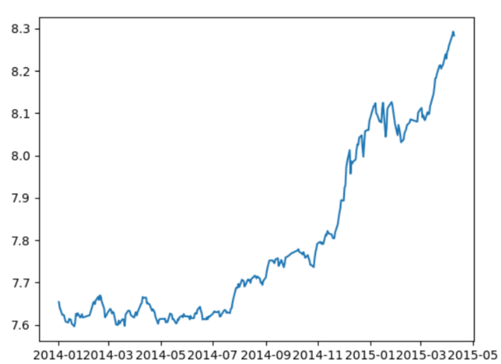
接下来我们用 ARIMA 模型，来拟合上证综指收盘价，我们利用除最后 3 天的数据来得出 ARIMA 模型，然后利用拟合出的 ARIMA 模型来预测后 3 天的收盘价，来看我们模型的性能。根据经验，对于股票的收盘数据来说，采用取对数后再求 1 阶差分的形式，可以取得更好的效果，因此我们会先对数据进行预处理，然后再来用 ARIMA 模型来拟合数据。
程序如下所示：

```
    def arima_demo(self):
        register_matplotlib_converters()
        data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
        sh_index = data[data.Indexcd==1]
        sh_index.index = pd.to_datetime(sh_index.index)
        raw_data = sh_index.Clsindex
        train_data = raw_data[:-3]
        close_price = np.log(train_data)
        plt.plot(close_price)
        plt.show()
        print(train_data.head(n=3))
        # ARIMA拟合
        min_ABQIC = sys.float_info.max
        arima_model = None
        for p in range(0, 5):
            for q in range(0, 5):
                print('try {0}, d, {1}...'.format(p, q))
                try:
                    arima_p_d_q = ARIMA(close_price, order=(p, 1, q)).fit
    (disp=False)
                    print('..... fit ok')
                    if arima_p_d_q.aic < min_ABQIC:
                        print('..... record good model')
                        min_ABQIC = arima_p_d_q.aic
                        arima_model = arima_p_d_q
                except Exception as ex:
                    print('.....!!!!!! {0}'.format(ex))
        print('ARIMA: p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'. \
                    format(arima_model.k_ar, arima_model.arparams,
                    arima_model.k_ma, arima_model.maparams,
                    arima_model.aic, arima_model.bic,
                    arima_model.hqic)
        )
        resid = arima_model.resid
        # 绘制ACF
        acfs = stattools.acf(resid)
        print(acfs)
```

```
        tsaplots.plot_acf(resid, use_vlines=True, lags=30)
        plt.title('ARIMA(p,d,q) ACF figure')
        plt.show()
        pacfs = stattools.pacf(resid)
        print(pacfs)
        tsaplots.plot_pacf(resid, use_vlines=True, lags=30)
        plt.title('ARIMA(p,d,q) PACF figure')
        plt.show()
        # ADF检验
        resid_adf = unitroot.ADF(resid)
        print('stat={0:0.4f} vs 1%_cv={1:0.4f}'.format(resid_adf.stat, resid_adf.critical_values['1%']))
        if resid_adf.stat < resid_adf.critical_values['1%']:
            print('resid为稳定时间序列 ^_^')
        else:
            print('resid为非稳定时间序列！！！！！')
        # Ljung-Box检验
        resid_ljung_box = stattools.q_stat(stattools.acf(resid)[1:12], len(resid))
        resid_lbv = resid_ljung_box[1][-1]
        print('resid_ljung_box_value={0}'.format(resid_lbv))
        # 0.05为显著性水平
        if resid_lbv < 0.05:
            print('resid为平稳时间序列 ^_^')
        else:
            print('resid为非平稳时间序列！！！！！！')
        # 预测
        y = arima_model.forecast(3)[0] #(len(train_data), len(raw_data), dynamic=True)
        print('预测值: {0}'.format(np.exp(y)))
        print('row_data:{0}'.format(raw_data))
        print('train_data:{0}'.format(train_data))
```
Listing 8: ARIMA 数据拟合上证综指收盘价

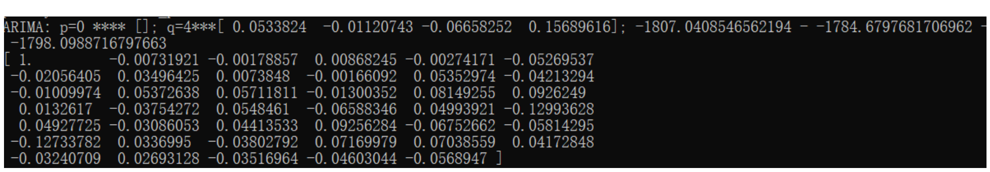
我们首先绘制出上证综指收盘价曲线:

Figure 23: 上证综指收盘价



接着我们对该数据经过对数差分后，利用 ARIMA 来进行拟合，得到拟合模型为:

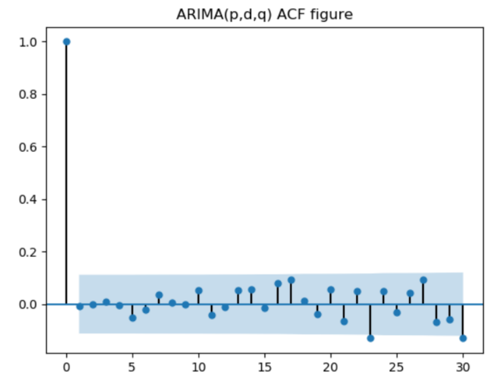
Figure 24: 拟合后的 ARIMA 模型



```
ARIMA: p=0 **** []; q=4***[ 0.0533824  -0.01120743 -0.06658252  0.15689616]; -1807.0408546562194 - -1784.6797681706962 -
 -1798.0988716797663
[ 1.         -0.00731921 -0.00178857  0.00868245 -0.00274171 -0.05269537
 -0.02056405  0.03496425  0.0073848  -0.00166092  0.05352974 -0.04213294
 -0.01009974  0.05372638  0.05711811 -0.01300352  0.08149255  0.0926249
  0.0132617  -0.03754272  0.0548461  -0.06588346  0.04993921 -0.12993628
  0.04927725 -0.03086053  0.04413533  0.09256284 -0.06752662 -0.05814295
 -0.12733782  0.0336995  -0.03802792  0.07169979  0.07038559  0.04172848
 -0.03240709  0.02693128 -0.03516964 -0.04603044 -0.0568947 ]
```

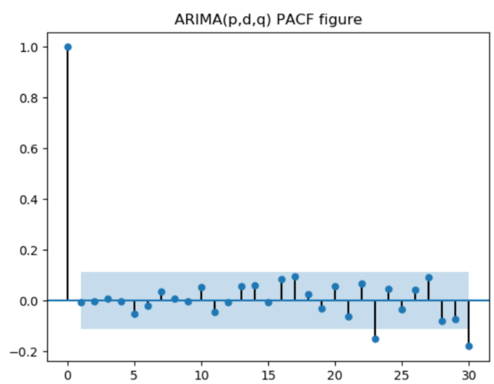
残差序列的自相关系数函数 ACF 图为：

Figure 25: 残差序列的自相关系数函数 ACF 图



残差序列的偏自相关系数函数 PACF 图为：

Figure 26: 残差序列的偏自相关系数函数 PACF 图



进行 ADF 检验的结果为：

Figure 27: ADF 检验的结果

```
stat=-17.5787 vs 1%_cv=-3.4519
resid为稳定时间序列 ^_^
```

进行 Ljung-Box 检验结果为：

Figure 28: Ljung-Box 检验结果

```
resid_ljung_box_value=0.9915215214530787
resid为非平稳时间序列！！！！！！！
```

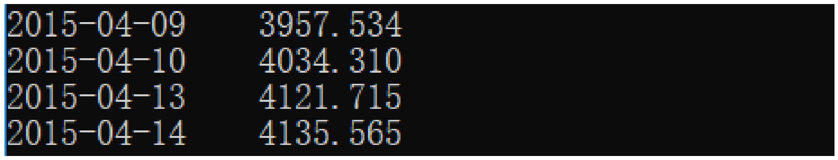
最后我们拿我们的模型进行预测，后三天的预测结果为：

Figure 29: ARIMA 模型预测后三天结果

```
预测值：[3958.4030514  3982.14296761 3990.28084529]
```

实际值为:

Figure 30: 实际收盘价



2015-04-09	3957.534
2015-04-10	4034.310
2015-04-13	4121.715
2015-04-14	4135.565

我们看到，我们的模型基本预测出了后三天的连涨行情，只不过上涨的幅度有一些小。

第 3 章 GARCH 模型

Abstract

在本章中我们将首先讲述条件异方差模型 GARCH（Generalized AutoRegressive Conditional Heteroskedastic），并将 GARCH 模型用于实际金融时间序列数据拟合。aqt002.py

3 GARCH 模型

我们之前讨论的时序信号，都是假定其为平稳的。但是有很多时序信号，如某些商品的需求或价格，会随着季节的变化而发生变化，股票的价格为出现长时间的上涨或下跌趋势，在这些情况下，使用 ARIMA 模型的效果就不太好。对于我们要研究的股票数据，由于市场上很多交易是由大机构的算法来进行交易的，当出现价格明显示上涨或下跌时，会自动触发这些算法进行交易，从而放大了这种上涨或下跌趋势。这种情况我们称之为异方差，研究这种现象的方法我们称之为通用自回归条件异方差 GARCH（Generalized AutoRegression Conditional Heteroskedasticity）模型。

3.1 定义

3.1.1 ARCH 模型

我们先来看较为简单的自回归条件异方差 ARCH 模型，我们假设时序信号如下所示:

$$\epsilon_t = \sigma_t w_t \tag{50}$$

其中 w_t 为白噪声序列，σ_t 的定义如下所示:

$$\sigma_t^2 = \alpha_0 + \alpha_1 \epsilon_{t-1}^2 \tag{51}$$

这个模型我们称之为 ARCH(1) 模型。我们以这个简单的模型为例，来说明 ARCH(1) 模型是对时序信号方差的变化进行建模。我们首先来看时间序列 $\{\epsilon_t\}$ 的均值:

$$E(\epsilon_t) = E(\sigma_t w_t) = E(\sigma_t)E(w_t) = 0 \tag{52}$$

我们再来看时间序列 $\{\epsilon_t\}$ 的方差:

$$\begin{aligned} Var(\epsilon_t) &= E\big(\epsilon_t - E(\epsilon_t)\big)^2 = E\big(\epsilon_t^2 - 2\epsilon_t E(\epsilon_t) + (E(\epsilon_t))^2\big) \\ &= E(\epsilon_t^2) - 2E(\epsilon_t)E(\epsilon_t) + (E(\epsilon_t))^2 = E(\epsilon_t^2) - (E(\epsilon_t))^2 \\ &= E(\epsilon_t^2) = E(\sigma_t^2 w_t^2) = E(\sigma_t^2)E(w_t^2) = E(\alpha_0 + \alpha_1 \epsilon_{t-1}^2) \\ &= \aleph_0 + \alpha_1 E(\epsilon_{t-1}^2) = \alpha_0 + \alpha_1 Var(\epsilon_{t-1}) \end{aligned} \tag{53}$$

在上面的公式推导中，我们用到了 $\{w_t\}$ 为白噪声信号，其均值为 0，方差为 1。
了解了基本 ARCH 模型之后，我们可以将 ARCH(1) 模型扩展到 ARCH(p) 模型，这里我们就不再展开了，将在下一节 GARCH 模型中进行详细介绍。

3.1.2 GARCH 模型

对于时间序列信号 $\{\epsilon_t\}$ 其表达式为:

$$\epsilon_t = \sigma_t w_t \tag{54}$$

其中 $\{w_t\}$ 为白噪声信号，其均值为 0 方差为 1。其 σ_t^2 的定义为:

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{p} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{q} \beta_j \sigma_{t-j}^2 \tag{55}$$

其中 α_0、α_i、β_j 为参数。

3.2 数据模拟

为了对问题进行简化，我们在这里只模拟 GARCH(1,1) 模型，具体欲模拟的时间序列信号如下所示，在 arch 包的 GARCH 模型中，我们研究的信号为 r_t，并且多出一个参数 μ:

$$\begin{aligned} r_t &= \epsilon_t + \mu \\ \epsilon_t &= \sigma_t w_t \\ \sigma_t^2 = \omega + \alpha_1 \epsilon_{t-1}^2 + \beta_1 \sigma_{t-1}^2 &= 0.2 + 0.5\epsilon_{t-1}^2 + 0.3\sigma_{t-1}^2 \end{aligned} \tag{56}$$

其程序如下所示:

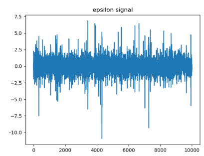
```python
import sys
import math
import numpy as np
import pandas as pd
from pandas.plotting import register_matplotlib_converters
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
from matplotlib.font_manager import FontProperties
from statsmodels.tsa import stattools
from statsmodels.graphics import tsaplots
from statsmodels.tsa.arima_model import ARIMA
import arch.unitroot as unitroot
import arch as arch

class Aqt002(object):
    def __init__(self):
        self.name = 'Aqt001'
        # 数据文件格式: 编号 日期 星期几 开盘价 最高价
        # 最低价 收益价 收益
        # Indexcd   Trddt   Daywk   Opnindex   Hiindex
        # Loindex   Clsindex   Retindex
        self.data_file = 'data/pqb/aqt002_001.txt'

    def startup(self):
        print('GARCH模型...')
        np.random.seed(1)
        alpha0 = 0.2
        alpha1 = 0.5
        beta1 = 0.3
        samples = 10000 # 样本数量
        w = np.random.standard_normal(size=samples)
        epsilon = np.zeros((samples,), dtype=float)
        sigma = np.zeros((samples,), dtype=float)
        for i in range(2, samples):
            sigma_2 = alpha0 + alpha1 * math.pow(epsilon[i-1], 2) + \
                        beta1 * math.pow(sigma[i-1], 2)
            sigma[i] = math.sqrt(sigma_2)
            epsilon[i] = sigma[i]*w[i]
        plt.title('epsilon signal')
        plt.plot(epsilon)
```

```python
        plt.show()
        # 绘制epsilonACFS
        acfs = stattools.acf(epsilon)
        tsaplots.plot_acf(epsilon, use_vlines=True, lags=30)
        plt.title('epsilon ACF')
        plt.show()
        # 绘制epsilon pow2 ACF
        acfs2 = stattools.acf(np.power(epsilon, 2))
        tsaplots.plot_acf(np.power(epsilon, 2), use_vlines=True, lags=30)
        plt.title('pow(epsilon,2) ACF')
        plt.show()
        # GARCH拟合
        am = arch.arch_model(epsilon, x=None, mean='Constant',
                    lags=0, vol='Garch', p=1, o=0, q=1,
                    power=2.0, dist='Normal', hold_back=None)
        model = am.fit(update_freq=0)
        # GARCH(1,1)参数
        print('############ GARCH(1,1)参数   ###################')
        print('model_type:{0}'.format(type(model)))
        print('mu={0:0.2f}; a0={1:0.2f}; a1={2:0.2f}; b1={3:0.2f}' \
                    .format(model.params['mu'], model.params['omega'],
                    model.params['alpha[1]'], model.params['beta[1]']    )
        print('################################')
        #print(model.summary())
        # 残差信号
        plt.title('GARCH(1,1) resid')
        plt.plot(model.resid)
        plt.show()
        # 残差ACF
        resid_acf = stattools.acf(model.resid)
        tsaplots.plot_acf(model.resid, use_vlines=True, lags=30)
        plt.title('GARCH(1,1) resid ACF')
        plt.show()
        # ADF检验
        resid_adf = unitroot.ADF(model.resid)
        print('stat={0:0.4f} vs 1%_cv={1:0.4f}'.format( \
                    resid_adf.stat, resid_adf.critical_values['1%']))
        if resid_adf.stat < resid_adf.critical_values['1%']:
            print('resid为稳定时间序列 ^_^')
        else:
            print('resid为非稳定时间序列！！！！！')
        # Ljung-Box检验
        resid_ljung_box = stattools.q_stat(stattools.acf( \
                    model.resid)[1:12], len(model.resid))
        resid_lbv = resid_ljung_box[1][-1]
        print('resid_ljung_box_value={0}'.format(resid_lbv))
        # 0.05为显著性水平
        if resid_lbv < 0.05:
            print('resid为平稳时间序列 ^_^')
        else:
            print('resid为非平稳时间序列！！！！！！！')
        # 预测
        y = model.forecast(horizon=3)
        print('########### 预测值  ###############')
        print('len={0}; p1={1:0.3f}; p2={2:0.3f}; p3={3:0.3f}'. \
                    format(len(y.mean.iloc[-1]), y.mean.iloc[-1][0],
                    y.mean.iloc[-1][1], y.mean.iloc[-1][2]))
        print('##########################')
```

Listing 9: GARCH 拟合模拟数据

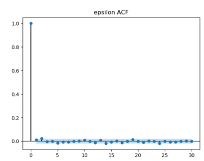
我们欲拟合的时间序列信号为:

Figure 31: 拟拟合的时间序列



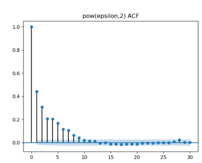
该时间序列的自相关系数函数 ACF 图:

Figure 32: epsilon 的自相关系数函数 ACF 图



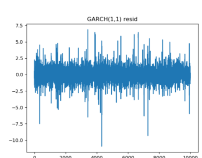
由上图可以看出，其基本是一个平稳时间序列。我们取其平方，其自相关系数函数 ACF 图如下所示:

Figure 33: pow(epsilon,2) 的自相关系数函数 ACF 图



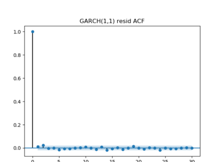
由上图可以看出，将其平方后，其就是不平稳的时间序列了。GARCH 模型就是来处理这种情况的。
我们采用 GARCH 进行拟合后，得到的残差信号为:

Figure 34: GARCH(1,1) 残差序列



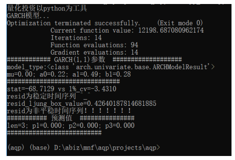
残差信号的自相关系数函数 ACF 图为:

Figure 35: GARCH(1,1) 残差序列自相关系数函数 ACF 图



程序的运行结果如下所示:

Figure 36: 程序运行结果



由上图可知，我们对 GARCH(1,1) 模型的参数预测还是相当准确的，说明我们方法是正确的。

3.3 金融数据拟合

下面我们以上证综指的收盘价为例，来讨论 GARCH 拟合预测后三天股价问题。对于收盘价这样的信号，我们仍然采用对数一阶差分形式来作为建模的时间序列，我们用 GARCH(3,2) 来进行拟合，程序如下所示：

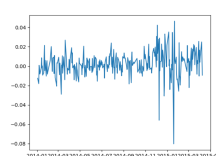
```python
    def garch_finance_demo(self):
        print('拟合上证综指收盘价...')
        register_matplotlib_converters()
        data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
        sh_index = data[data.Indexcd==1]
        sh_index.index = pd.to_datetime(sh_index.index)
        sh_return = sh_index.Retindex
        # raw_data = sh_index.Retindex
        raw_data = sh_index.Clsindex
        train_data = raw_data[:-3]
        dcp = np.log(train_data).diff(1)[1:]
        plt.plot(dcp)
        plt.show()
        # GARCH拟合
        am = arch.arch_model(dcp, x=None, mean='Constant',
                    lags=0, vol='Garch', p=3, o=0, q=2,
                    power=2.0, dist='Normal', hold_back=None)
        model = am.fit(update_freq=0)
        # GARCH(1,1)参数
        print('############ GARCH(1,1)参数   ###################')
        '''
        print('mu={0:0.2f}; a0={1:0.2f}; a1={2:0.2f}; a2={3:0.2f}, b1
    ={4:0.2f}, b2={5:0.2f}' \
                    .format(model.params['mu'], model.params['omega'],
                    model.params['alpha[1]'], model.params['alpha[2]'],
    model.params['beta[1]'], model.params['beta[2]']    ))
        '''
        print('################################')
        resid = model.resid
        # 绘制ACF
        acfs = stattools.acf(resid)
        print(acfs)
        tsaplots.plot_acf(resid, use_vlines=True, lags=30)
        plt.title('GARCH(p,q) ACF figure')
        plt.show()
        '''
        pacfs = stattools.pacf(resid)
        print(pacfs)
        tsaplots.plot_pacf(resid, use_vlines=True, lags=30)
        plt.title('ARIMA(p,d,q) PACF figure')
        plt.show()
        '''
        # ADF检验
        resid_adf = unitroot.ADF(resid)
        print('stat={0:0.4f} vs 1%_cv={1:0.4f}'.format(resid_adf.stat,
    resid_adf.critical_values['1%']))
```

```python
        if resid_adf.stat < resid_adf.critical_values['1%']:
            print('resid为稳定时间序列 ^_^')
        else:
            print('resid为非稳定时间序列！！！！！')
        # Ljung-Box检验
        resid_ljung_box = stattools.q_stat(stattools.acf(resid)[1:12],
    len(resid))
        resid_lbv = resid_ljung_box[1][-1]
        print('resid_ljung_box_value={0}'.format(resid_lbv))
        # 0.05为显著性水平
        if resid_lbv < 0.05:
            print('resid为平稳时间序列 ^_^')
        else:
            print('resid为非平稳时间序列！！！！！！！')
        # 预测
        frst = model.forecast(horizon=3)
        y = frst.mean.iloc[-1]
        print('预测值：{0}'.format(y))
        p1 = math.exp(math.log(train_data[-1]) + y[0])
        p2 = math.exp(math.log(p1) + y[1])
        p3 = math.exp(math.log(p2) + y[2])
        print('        预测值      实际值   (3957.534)')
        print('第一天：{0} vs 4034.310'.format(p1))
        print('第二天：{0} vs 4121.715'.format(p2))
        print('第三天：{0} vs 4135.565'.format(p3))
```

Listing 10: GARCH 拟合并预测上证综指收盘价

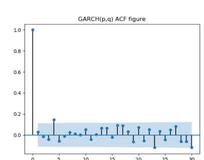
我们要建模的上证综指收盘价对数差分信号如下所示：

Figure 37: 上证综指收盘价对数差分信号



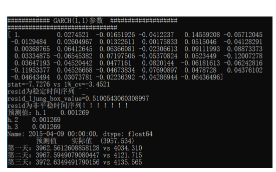
经过 GARCH(3,2) 拟合后的残差信号：

Figure 38: 上证综指收盘价对数差分信号



由图可见，其基本是一个平稳时间序列，可以用来进行预测。但是我们 ADF 检验表明其是平稳时间序列，但是 Ljung-Box 检验表明其不是时间平稳时间序列，因此我们还需要仔细分析其中的原因。程序运行结果如下所示：

Figure 39: 程序运行结果



```
############ GARCH(1,1)参数  ###################
################################
[ 1.          0.0274521  -0.01651926 -0.0412237   0.14559208 -0.05712045
 -0.0129484   0.02604967  0.01322611  0.00175833  0.0515046  -0.04128291
  0.00368765  0.06412645  0.06366081 -0.02306613  0.09111993  0.08873373
  0.03334875 -0.06545382  0.07197506 -0.05370824  0.0523449  -0.12007278
  0.03647193 -0.04520442  0.0477161   0.0820144  -0.06181613 -0.06242816
 -0.11953377  0.04526668 -0.04673834  0.07690897  0.0478728   0.04376102
 -0.04643494  0.03073781 -0.02236392 -0.04286944 -0.06436496]
stat=-7.7276 vs 1%_cv=-3.4521
resid为稳定时间序列 ^_^
resid_ljung_box_value=0.5100543060308997
resid为非平稳时间序列！！！！！！！
预测值：h.1    0.001269
h.2    0.001269
h.3    0.001269
Name: 2015-04-09 00:00:00, dtype: float64
        预测值    实际值  (3957.534)
第一天：3962.5612608858128 vs 4034.310
第二天：3967.5949079080447 vs 4121.715
第三天：3972.6349491790156 vs 4135.565
```

由上图可以看出，预测结果也基本捕捉到了后三天连涨趋势，只是涨幅估计偏低，这证明我们采用 GARCH(p,q) 模型来预测是可行的。

python 的 ARIMA：https://www.colabug.com/3933896.html Ln125 Advanced Algorithmic Trading

第 4 章协整模型

Abstract

在本章中我们将首先讲述交易对的概念，并讲述如何利用交易对的均值回归特性，通过对交易对的多空操作，实现稳定的盈利。在本章中，我们主要讲解交易对的数学原理，怎样确定对冲比例。对于怎样利用交易对进行统计套利策略研发，将在后续章节中介绍。aqt003.py

4 协整模型

本章讲述的内容是统计套利策略，这是一种市场中性策略，无论是市场是涨是跌，均有可能盈利。将统计套利用到极致的例子，应该属于上世纪 90 年代的 LTCM 基金。LTCM 基金成立于 1994 年，成立时基金规模为 12.5 亿美金，但是到 1997 年，基金规模就达到了 48 亿美金，是当时世界上四大基金之一。LTCM 基金的创始人是金牌交易员，合伙人是顶级统计学家，他们发现德国国债和意大利国债有一个稳定的比例关系，大概率波动的机率会非常小，因此当德国国债上涨时，德国国债与意大利国债之比将增大，大幅度偏离均值，他们就卖出德国国债，买入意大利国债，一段时间之后，德国国债会下跌，德国国债和意大利国债之比就会回归到均值，此时他们卖出之前买入的意大利国债，重新买入之前卖出的德国国债，这样他们持有德国国债和意大利国债没有发生变化，但是却赚到了这次波动差价所产生的利润。利用这种方式，LTCM 基金获得了巨大的成功。然而 LTCM 基金的神话终结于 1997 年，主要有以下几个原因：首先 LTCM 认为德国国债和意大利国债之比符合正态分布，但是虽然自然界大部分复杂的现象都是正态分布，但是人类社会现象，却不一定是正态分布，黑天鹅事件也许是长态，在 1997 年，东南亚发生了金融危机，俄罗斯发生了国债违约，正是这两个黑天鹅事件，使德国国债与意大利国债之比不再符合正态分布，而 LTCM 基金由于对这些事件缺乏预见性，而遭受了巨大的损失；同时，在 LTCM 基金破产之后，人们看 LTCM 基金的财务数据发现，其资金杠杆高达数万倍，因此一次失误的决策，就足以使整个公司破产。由这个实例我们可以看出，基于交易对均值回归的统计回归策略，具有巨大的盈利潜力，同时我们也应看到，杠杆是魔鬼，任何策略中最重要的都是资金和仓位管理。
交易对不仅在传统金融市场，如股市、期货、外汇、债市上存在，在新兴的加密货币市场，也是一种非常重要的策略，前一两年兴起的比特币搬砖模式，就是交易对策略在加密货市场的应用。

4.1 数据模拟

4.1.1 协整模型构建

根据Halls-Moore [2017] 的内容，我们可以通过模拟数据来加深我们对基本概念的理解。假设 $\{w_t\}$ 为白噪声，在此基础上定义随机游走信号：

$$z_t = z_{t-1} + w_t \tag{57}$$

在此基础上我们定义两个非平稳的时间序列信号：

$$x_t = pz_t + w_t = 0.3z_t + w_t \tag{58}$$

$$y_t = qz_t + w_t = 0.6z_t + w_t \tag{59}$$

如果我们将 $\{x_t\}$ 和 $\{y_t\}$ 进行线性组合：

$$ax_t + by_t = (ap + bq)z_t + w_t = (0.3a + 0.6b)z_t + w_t \tag{60}$$

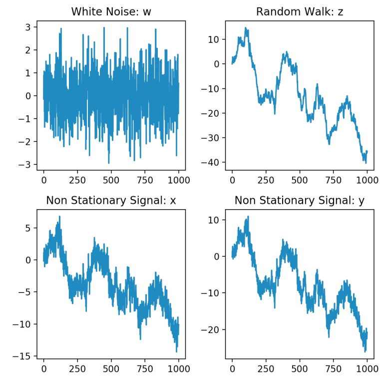
如果 $0.3a + 0.6b = 0$ 的话，那么 $\{x_t\}$ 和 $\{y_t\}$ 的这个线性组合就只剩下白噪声信号，因此其是平稳时间序列信号，因此当 $a = 2, b = -1$ 时，这个序列就是平稳时间序列信号。
根据这一思想，程序如下所示：

```python
    def simulate_demo(self):
        '''
        模拟数据生成
        '''
        # 生成白噪声信号
        samples = 1000
        w = np.random.standard_normal(size=samples)
        # 生成随机游走序列
        z = np.zeros((samples,))
        for t in range(1, samples):
            z[t] = z[t-1] + w[t]
        # 生成非平稳信号，即交易对x和y
        x = np.zeros((samples,))
        y = np.zeros((samples,))
        p = 0.3
        q = 0.6
        for t in range(samples):
            x[t] = p*z[t] + w[t]
            y[t] = q*z[t] + w[t]
        fig = plt.figure(figsize=(6, 6))
        w_plt = plt.subplot(2, 2, 1, title='White Noise: w')
        w_plt.plot(w)
        z_plt = plt.subplot(2, 2, 2, title='Random Walk: z')
        z_plt.plot(z)
        x_plt = plt.subplot(2, 2, 3, title='Non Stationary Signal: x')
        x_plt.plot(x)
        y_plt = plt.subplot(2, 2, 4, title='Non Stationary Signal: y')
        y_plt.plot(y)
        fig.tight_layout()
        plt.show()
        # 生成协整信号
        a = 2
        b = -1
        c = a * x + b * y
        plt.plot(c)
        plt.title('Cointegration Model')
        plt.show()
        # 采用ADF检验
        resid_adf = unitroot.ADF(c)
        print('stat={0:0.4f} vs 1%_cv={1:0.4f}'.format( \
                    resid_adf.stat, resid_adf.critical_values['1%']))
        if resid_adf.stat < resid_adf.critical_values['1%']:
            print('resid为稳定时间序列 ^_^')
        else:
            print('resid为非稳定时间序列！！！！！')
```

Listing 11: 线性组合形成平稳时间序列

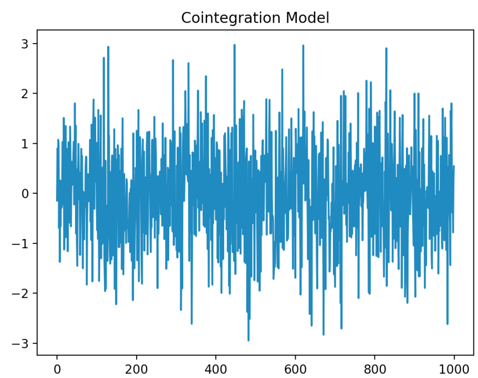
我们首先绘制出白噪声、随机游走和 x 及 y 信号:

Figure 40: 研究序列



接着我们绘制出 x 和 y 的线性组合：

Figure 41: x 和 y 的线性组合



我们利用 ADF 检验其稳写性：

Figure 42: ADF 检验

```
量化投资以python为工具
交易对协整模型...
stat=-31.9595 vs 1%_cv=-3.4369
resid为稳定时间序列 ^_^
```

由上面的 ADF 检验可以看出，x 和 y 的线性组合与理论分析一致，是平稳时间序列。

4.1.2 对冲比例确定

在上面的例子中，我们验证了我们的假设，即对两个非平稳的时间序列进行线性组合，通过选择合适的系数，可以得到平稳时间序列信号。现在的问题就变为怎样求出这些系数，使线性组合后的时间序列具有平稳性。在这一节中，我们将向大家演示采用线性回归技术，来求出线性组合的系数。虽然线性回归算法非常简单，尤其是我们这里只有一维，就更简单了。但是这里我们将采用 Google 在今年 3 月新推出的 TensorFlow 2.0 alpha，采用高级 API keras 来完成这一过程。

线性回归模型 在讲解确定线性组合系数之前，我们先来讲解一下为达到这一目的所开发的基于 TensorFlow 2.0 的线性回归模型。代码如下所示：

```python
import numpy as np
import matplotlib.pyplot as plt
```

```python
import tensorflow as tf

class QciLinearRegression(object):
    def __init__(self, train_x=None, train_y=None,
                validate_x=None, validate_y=None,
                test_x=None, test_y=None):
        self.name = 'QciLinearRegression'
        self.loss = 'mean_squared_error'
        self.learning_rate = 0.01
        self.optimizer = tf.keras.optimizers.Adam(self.learning_rate)
        self.epoch = 50000
        if train_x:
            self.train_x = train_x
            self.train_y = train_y
            self.validate_x = validate_x
            self.validate_y = validate_y
            self.test_x = test_x
            self.test_y = test_y
        else:
            self.train_x, self.train_y, self.validate_x, \
                        self.validate_y, self.test_x, \
                        self.test_y = self.generate_dataset()

    def generate_dataset(self):
        train_x = np.array([-40, -10, 0, 8, 15, 22, 38, 20, 9, 13], dtype=float)
        train_y = np.array([-40, 14, 32, 46, 59, 72, 100, 68, 48.2, 55.4], dtype=float)
        validate_x = np.array([], dtype=float)
        validate_y = np.array([], dtype=float)
        test_x = np.array([], dtype=float)
        test_y = np.array([], dtype=float)
        return train_x, train_y, validate_x, validate_y, test_x, test_y

    def train(self):
        model = self.build_model()
        model.compile(loss=self.loss, optimizer=self.optimizer)
        class PrintDot(tf.keras.callbacks.Callback):
            def on_epoch_end(self, epoch, logs):
                if epoch % 100 == 0: print('')
                print('epoch:{0}...{1}!'.format(epoch, logs))
        early_stop = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)
        history = model.fit(self.train_x, self.train_y,
                    epochs=self.epoch, validation_split = 0.1,
                    verbose=False, callbacks=[early_stop, PrintDot()])
        plt.title('linear regression training process')
        plt.xlabel('epochs')
        plt.ylabel('error')
        plt.plot(history.history['loss'])
        plt.show()
        model.save('./work/aqt003_qiclr')
        weights = np.array(model.get_weights())
        print(weights)

    def build_model(self):
        layer1 = tf.keras.layers.Dense(units=1, input_shape=[1])
        model = tf.keras.Sequential([layer1])
        return model

    def predict(self, data):
        model = tf.keras.models.load_model('./work/aqt003_qiclr')
        rst = model.predict(data)
```

```python
            return rst

    if '__main__' == __name__:
        lr = QciLinearRegression()
        lr.train()
```

Listing 12: 基于 TensorFlow 2.0 的线性回归模型

初始化函数的参数为训练样本集、验证样本集和测试样本集，缺省值为空，当不传递这些值时，我们采用类内定义的 generate_dataset 方法来为这些属性赋值。同时，在构造函数中设定了代价函数为最小平方误差函数，学习率为 0.01，优化算法为最常用的 Adam，训练遍数为 5 万。
在 generate_dataset 函数中，我们输入特征为摄氏度的温度，而输出值为华氏度的温度，二者的关系为：

$$F = 1.8 \times C + 32 \tag{61}$$

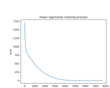
我们任务就是学出 1.8 和 32 这两个参数。
外部程序通过调用 train 方法作为入口，程序首先调用 build_model 方法生成模型，我们的这个网络由两层组成，第 1 层为输入层只有一个节点，第 2 层为输出层，其也只有一个节点，这个网络的参数为：第 1 层节点到第 2 层节点的连接权值和第 2 层神经元的偏置值。创建完模型之后，我们通过 model.compile 来设置模型的代价函数和优化算法；为了提高模型的泛化能力避免 overfitting，我们采用 early stopping 算法，当验证集上的精度在 10 个循环中没有明显改进时，停止训练过程（在后面的日志中，我们可以看到虽然我们定义训练 5 万次，但是实际训练不到 1 万次就停止了）。接着我们定义一个临时类，因为神经网络训练通常会很耗时，我们用这个函数打印当前进度情况。接着我们调用 model.fit 来开始实际训练过程，在后台日志中我们可以看到，误差会逐渐减少，如图所示：

Figure 43: 程序运行日志

```
epoch:7661...{'loss': 0.050047121942043304, 'val_loss': 0.003036373993381858}!
epoch:7662...{'loss': 0.05004706233739853, 'val_loss': 0.003035953501239419}!
epoch:7663...{'loss': 0.05004725232720375, 'val_loss': 0.003035953501239419}!
epoch:7664...{'loss': 0.05004772171378136, 'val_loss': 0.003035953501239419}!
epoch:7665...{'loss': 0.050047118216753006, 'val_loss': 0.003035953501239419}!
epoch:7666...{'loss': 0.050047118216753006, 'val_loss': 0.003035953501239419}!
epoch:7667...{'loss': 0.050047118216753006, 'val_loss': 0.003035953501239419}!
epoch:7668...{'loss': 0.0500471256673336, 'val_loss': 0.003035953501239419}!
epoch:7669...{'loss': 0.050047121942043304, 'val_loss': 0.003035953501239419}!
epoch:7670...{'loss': 0.050047118216753006, 'val_loss': 0.003035953501239419}!
epoch:7671...{'loss': 0.050047118216753006, 'val_loss': 0.003035953501239419}!
epoch:7672...{'loss': 0.050047118216753006, 'val_loss': 0.003035953501239419}!
```

我们可以看到最后一列的验证集上的误差在 10 次没有明显变化时，由于 early stopping 算法，训练过程就自动结束了。接下来我们可以绘制出误差随训练时间的变化过程，如下所示：

Figure 44: 误差变化曲线



最终学到的参数值为：

Figure 45: 误差变化曲线

```
[[1.7982784509658813]
 [31.967283248901367]]
```

我们看到其与实际值就非常接近了，证明我们线性回归模型是正确的。虽然在我们协整模型中用不到，但是我们还是介绍一下当网络训练完成之后，下一步就给出数据，让其给我们算出真实的结果，这在 predict 方法中实现。在 predict 方法中，首先从模型文件中恢复出网络结构，然后运行模型的 predict 方法生成并返回预测值，调用代码如下所示：

```python
    data = np.array([[100.0]], dtype=float)
    rst = qcilr.predict(data)
    print(rst)
```

Listing 13: 利用线性回归模型进行预测

运行结果如下所示：

Figure 46: 预测值

```
交易对协整模型...
2019-04-13 17:43:04.951507: I tensorflow/core/platform/cpu_feature_guard.cc:142]
[[211.79514]]
```

对冲比例确定 在有了线性回归模型之后，下面我们就用线性回归模型来确定对冲比例。程序如下所示：

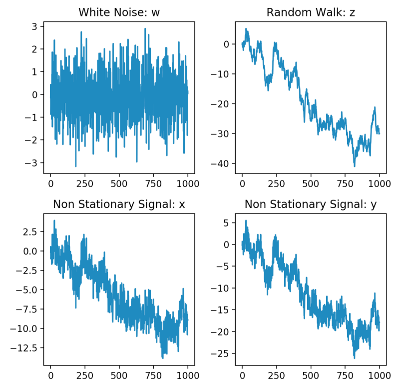
```python
    def qcilr_demo(self):
        # 生成白噪声信号
        samples = 1000
        w = np.random.standard_normal(size=samples)
        # 生成随机游走序列
        z = np.zeros((samples,))
        for t in range(1, samples):
            z[t] = z[t-1] + w[t]
        # 生成非平稳信号，即交易对x和y
        x = np.zeros((samples,))
        y = np.zeros((samples,))
        p = 0.3
        q = 0.6
        for t in range(samples):
            x[t] = p*z[t] + w[t]
            y[t] = q*z[t] + w[t]
        fig = plt.figure(figsize=(6, 6))
        w_plt = plt.subplot(2, 2, 1, title='White Noise: w')
        w_plt.plot(w)
        z_plt = plt.subplot(2, 2, 2, title='Random Walk: z')
        z_plt.plot(z)
        x_plt = plt.subplot(2, 2, 3, title='Non Stationary Signal: x')
        x_plt.plot(x)
        y_plt = plt.subplot(2, 2, 4, title='Non Stationary Signal: y')
        y_plt.plot(y)
        fig.tight_layout()
        plt.show()
        # 以x作为自变量
        w1, x_y_p = self.do_linear_regression(x, y)
        # 将y作为自变量
        w2, y_x_p = self.do_linear_regression(y, x)
        print('xToy={0}({1}) vs yTox={2}({3})'.format(x_y_p, w1[0][0],
    y_x_p, w2[0][0]))
        if x_y_p < y_x_p:
            print('######## x  为自变量')
            c = w1[0][0] * x - y
        else:
            print('######### y  为自变量')
            c = w2[0][0] * y - x
        plt.title('Final Cointegration Signal')
        plt.plot(c)
        plt.show()
```

Listing 14: 线性回归模型确定对冲比例

在上面的程序中，我们首先像上一节一样，先做出 x 和 y 两个非平稳的时间序列信号，然后我们先以 x 为自变量做一次线性回归，求出 ADF 检验的 p 值和对冲比例，再以 y 作为自变

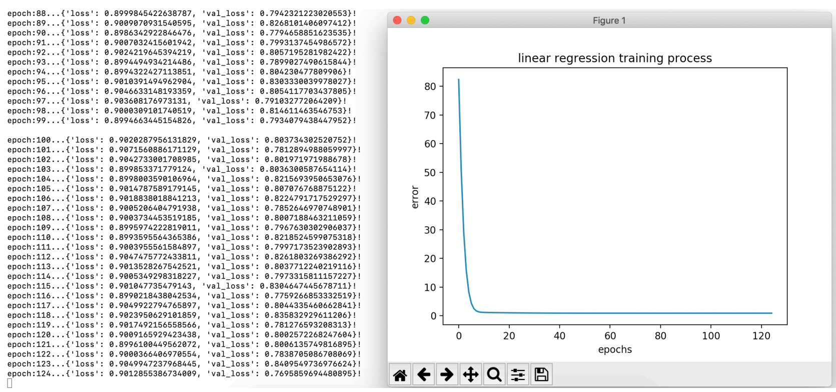
量做一次线性回归，求出 ADF 检验的值 p 和对冲比例，我们取 ADF 检验的 p 值较小的一个作为最终结果，求出线性组合的信号 c，最后绘制出其信号典线。
原始信号如下所示：

Figure 47: 原始信号



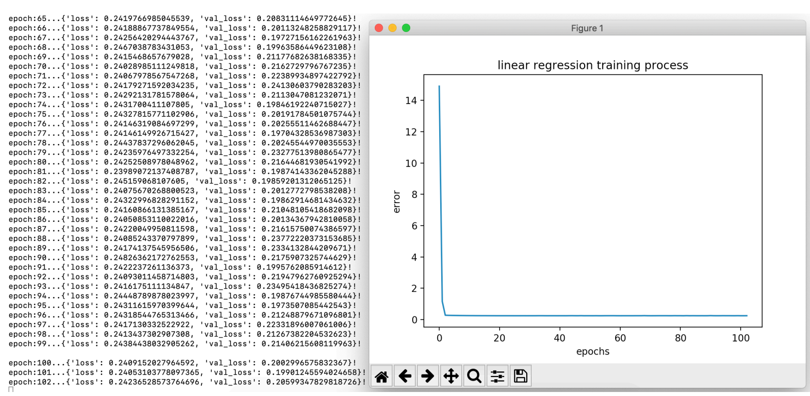
以 x 为自变量时线性回归运行情况：

Figure 48: 以 x 为自变量的线性回归



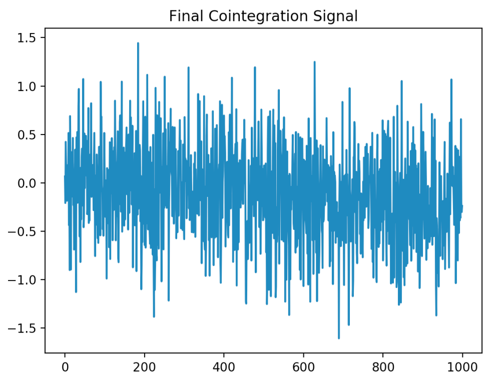
以 y 为自变量时线性回归运行情况：

Figure 49: 以 y 为自变量时线性回归



最后的协整信号结果：

Figure 50: 协整信号



程序的运行结果如下所示：

Figure 51: 程序运行结果

```
[[0.512149453163147]
 [0.1092881858348846]]
weights:[[0.512149453163147]
 [0.1092881858348846]]
stat=-9.9252 vs 1%_cv=-3.4370
resid为稳定时间序列 ^_^
xToy=-4.074526007320772(1.926144003868103) vs yTox=-9.925
######### y  为自变量
```

由于以 y 为自变量时 ADF 检验的 p 值较小，所以取以 y 为自变量的情况，计算出来的对冲比例为 0.51，与实际值 0.5 非常接近，这证明我们的算法是正确的。

4.2 Johansen 检验

我们已经讲述了协整模型理论，可以比较好的处理两个交易对的对冲比例和均值回归的问题。但是这里面有两个问题，首先在协整模型中，我们假设交易对中的两个标的是线性关系，其次我们只能处理有两个标的的交易对。在本节中，我们将根据tushare and 匠芯量化 [2019] 投资组合理论，处理标的之间不是线性关系，以及多个交易标的的均值回归策略研发问题。在实际应用中，我们经常用到的模型是 Johansen Test 模型，有时也叫 VECM 模型。在这一节中，我们将来讲解这个模型。Johansen 模型是基于向量自回归 VAR 模型的。我们以前所讲的自回归模型，都是标量的自回归模型，向量自回归模型定义为：

$$\boldsymbol{x}_t = \boldsymbol{\mu} + A_1 \boldsymbol{x}_{t-1} + ... + A_p \boldsymbol{x}_{t-p} + \boldsymbol{w}_t \tag{62}$$

式中 为序列的均值向量，$\boldsymbol{w}_t$ 为白噪声向量。接下来我们定义向量误差修正模块 VECM（Vector Error Correction Model）：

$$\Delta \boldsymbol{x}_t = \boldsymbol{\mu} + A \boldsymbol{x}_{t-1} + \Gamma_1 \Delta \boldsymbol{x}_{t-1} + ... + \Gamma_p \Delta \boldsymbol{x}_{t-p} + \boldsymbol{w}_t \tag{63}$$

其中 A 为系数矩阵，Γ_i 是各滞后期的系数矩阵。我们对矩阵 A 进行特征值分解，阶数为 r，分为以下两种情况：

- $r = 0$：表明没有线性协整模型；
- $r > 0$：至少有 $r + 1$ 个序列存在协整关系；

在实际应用中，我们取出最大的特征值对应的特征向量，将其元素作为对应序列的协整系数。

第 5 章卡尔曼滤波

Abstract

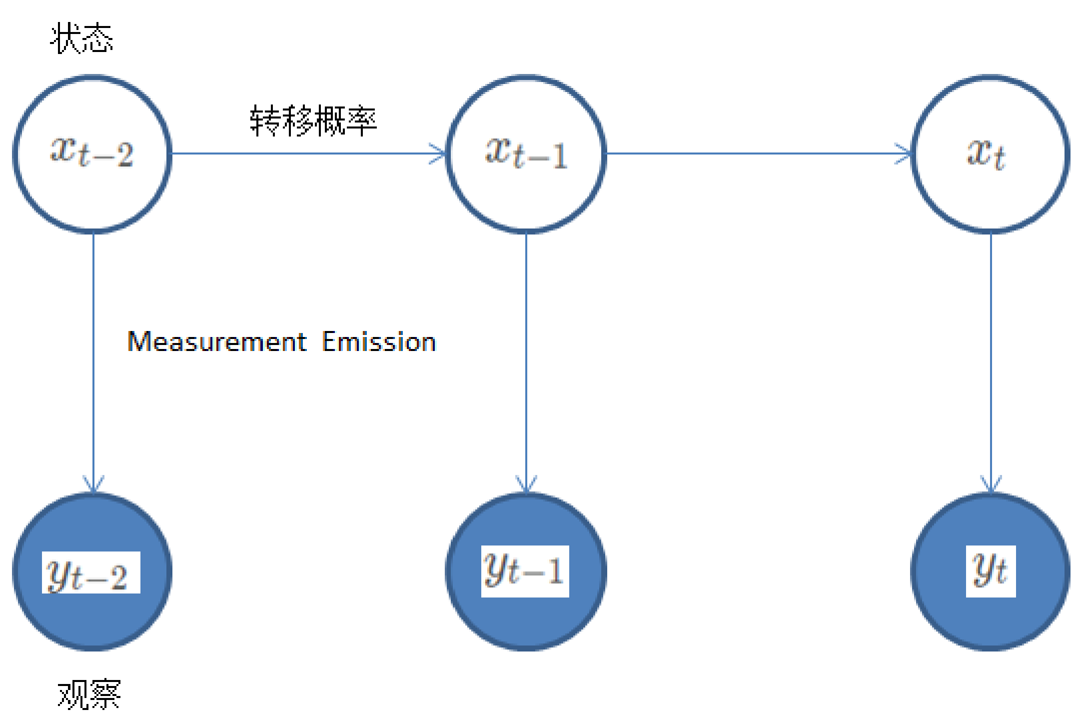
到目前为止，我们所讨论的基于协整模型的交易对策略，有一个重要的假设，即交易对之间的协整系数是不变的。但是在实际应用中，这些协整系数可能会发生缓慢的改变，如果我们用固定值，可能不会取得最佳的应用效果。在本章中，我们将采用 State Space Model 来解决这一问题，具体业讲，就是利用卡尔曼滤波技术，将交易对的对冲比例视为系统不可见的状态，将交易对中标的的收益率作为可观察项，利用卡尔曼滤波器的滤波功能，通过不断增加的新观测数据，利用贝叶斯推理，使我们能够更加精准的估计系统不可见状态，在本例中就是交易对的对冲比例。实际上，交易对的对冲比例，不仅会发生缓慢的变化，有时因为监管、宏观经济、市场事件等原因，交易对的对冲比例还可能发生剧烈的变化，这就需要我们识别市场所处状态，识别出市场状态的改变，从而做出更加科学的决策，这部分内容将在下一章隐马可夫模型章节中介绍。aqt005.py

5 卡尔曼滤波

5.1 State Space Model 概述

在 State Space Model 中，我们要研究的对象是环境，这里就是市场。环境会处于某种状态，而且环境所处的状态随着时间而改变。但是我们不能直接观测到环境，我们只能得到一些观察，我们希望通过观察来推测出系统年处的状态。以我们当前的任务为例，我们的环境就是市场上的交易对，而状态就是交易对的对冲比例，而由于各种原因，如市场微观结构等，我们只能得到交易对标的的收益率，其中具有很大的噪声，信噪比很低，我们的任务就是通过观察这些收益率数据，得到交易对的对冲比例。State Space Model 可以用如下图形来表示：

Figure 52: 通用 State Space Model



图中上面白色的圆圈代表环境的隐藏状态 x_t，蓝色的圆圈代表环境的观测值 y_t，系统会从一个状态转移到另一个状态，可以用一个概率 $P(x_t)x_{t-1}$ 来表示，环境所处状态决定观察值 $P(y_t|x_t)$，环境初始时的概率为 $P(x_0)$，根据这些变量的不同特性，我们可以把 State Space Model 分为四种类型：

Table 1: State Space Model 类型

名称	$P(x_t\|x_{t-1})$	$P(y_t\|x_t)$	$P(x_0)$
离散模型（隐马可夫模型）	A_{x_{t-1},x_t}	Any	π
线性高斯（卡尔曼滤波器）	$\mathcal{N}(Ax_{t-1}+B,\theta)$	$\mathcal{N}(Hx_t+C,R)$	$\mathcal{N}(\mu_0,\epsilon_0)$
非线性非高斯（粒子滤波器）	$f(x_t)$	$g(y_t)$	$f_0(x_0)$

利用 State Space Model 可以做三件事情：

1. 预测：预测下一个到几个时刻的状态值；
2. 滤波：根据当前的观察值，预测系统状态；
3. 平滑：利用过去的观察值，解释过去状态变化情况；

在这一章，我们所研究是线性高斯模型，即卡尔曼滤波器，我们同时会利用贝叶斯推断，通过不断增加的观察值，来逐步精确预测环境状态，在这里就是交易对的对冲比例。

5.2 数学原理

在这一部分，我们将简单讲述卡尔曼滤波的数学原理，公式和符号会比较多，但是重点是理解其背后的数据直觉，不必纠结与具体推导过程。
环境状态我们用一个向量来表示 $\boldsymbol{x}_t \in R^n$，由于其会随时间变化，因此我们为其添加了下标 t，因为卡尔曼滤波是线性高斯模型，当前时刻的状态，是由前一时刻的状态经线性组合再加上高斯噪声所组成，如下所示：

$$\boldsymbol{x}_t = A\boldsymbol{x}_{t-1} + \boldsymbol{b} + \boldsymbol{u}_t \tag{64}$$

其中 $A \in Rn \times n$ 的矩阵，$\boldsymbol{b} \in R^n$，$\boldsymbol{w}_t \in R^n$ 为高斯白噪声。
系统的观察用 $\boldsymbol{y}_t \in R^m$ 表示，其由当前时刻环境状态的线性组合加高斯白噪声组成，如下所示：

$$\boldsymbol{y}_t = H\boldsymbol{x}_t + \boldsymbol{c} + \boldsymbol{v}_r \tag{65}$$

系统初始状态为高斯分布：

$$\boldsymbol{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}_0, \Sigma_0) \tag{66}$$

环境状态在 t 时刻噪声：

$$\boldsymbol{u}_t \sim \mathcal{N}(0, \Sigma_t^u) \tag{67}$$

环境观察在 t 时刻噪声：

$$\boldsymbol{v}_t \sim \mathcal{N}(0, \Sigma_t^v) \tag{68}$$

5.3 PyKalman 库介绍

在这一节中，我们将讲述利用卡尔曼滤波来确定交易对的动态对冲比例。我们要研究的交易对为 TLT 和 IEI，其面向美国债券市场，具有相同的市场因素，因此应该具有稳定的协整比例。在本节中我们将使用 pykalman 库，所以我们先来讲解一下在 pykalman 库中，卡尔曼滤波的表示方式，主要参考自PyKalman [2019]。
我们同样用 x_t 来代表环境的隐藏状态，用 y_t 来代表对环境的观察，则卡尔曼滤波可以表示为：

$$\boldsymbol{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}_0, \Sigma_0) \tag{69}$$

$$\boldsymbol{x}_t = A_{t-1}\boldsymbol{x}_{t-1} + \boldsymbol{b}_{t-1} + \boldsymbol{\epsilon_t^1} \tag{70}$$

$$\boldsymbol{y}_t = C_t\boldsymbol{x}_t + \boldsymbol{d}_t + \boldsymbol{\epsilon_t^2} \tag{71}$$

$$\epsilon_t^1 \sim \mathcal{N}(0, Q) \tag{72}$$

$$\epsilon_t^2 \sim \mathcal{N}(0, R) \tag{73}$$

式中所用到符号如下所示：卡尔曼滤波中参数的估计是一个比较大的问题，我们可以通过

Table 2: PyKalman 变量意义

表示	模型参数	意义
$\boldsymbol{\mu}_0$	initial_state_mean	初始值均值向量
Σ_0	initial_state_covariance	初始状态协方差矩阵
A	transition_matrices	转移矩阵
$\boldsymbol{b}$	transition_offsets	转移偏置值
Q	transition_covariance	转移协方差矩阵（噪声）
C	observation_matrices	观察矩阵
$\boldsymbol{d}$	observation_offsets	观察偏置值
R	observation_covariance	观察协方差矩阵（噪声）

EM（）算法来根据现有的观察和环境状态数据，对参数进行估计。卡尔曼滤波的参数为：

$$\theta = \{A, \boldsymbol{b}, C, \boldsymbol{d}, Q, R, \boldsymbol{\mu}_0, \Sigma_0\} \tag{74}$$

我们的目录是：

$$\arg\max_{\theta} P(\boldsymbol{y}_{0:T-1}; \theta) \tag{75}$$

具体算法比较复杂，作为应用开发者，我们只需要知道这可以通过 PyKalman.em(observations) 函数来实现就可以了。

5.4 卡尔曼滤波应用

我们取到两个 ETF 的价格数据，并将其保存为文本文件，格式如下所示：

Figure 53: 数据文件格式

```
Date,TLT,IEI
2010-08-02,77.1137466430664,102.78514862060547
2010-08-03,77.55883026123047,103.22837829589844
2010-08-04,76.96540069580078,102.89151000976562
2010-08-05,77.32463836669922,103.16630554199219
2010-08-06,78.16798400878906,103.48545837402344
2010-08-09,77.87902069091797,103.387939453125
2010-08-10,78.04302215576172,103.76029205322266
2010-08-11,79.08940124511719,103.96420288085938
2010-08-12,78.89420318603516,103.73368072509766
2010-08-13,79.87818145751953,103.89326477050781
2010-08-16,81.87725830078125,104.22130584716797
2010-08-17,81.40867614746094,104.03512573242188
2010-08-18,81.6273422241211,103.94645690917969
2010-08-19,82.90021514892578,104.13267517089844
2010-08-20,82.8064956665039,103.89326477050781
2010-08-23,82.82213592529297,104.07054901123047
2010-08-24,84.14970397949219,104.46954345703125
2010-08-25,83.87635040283203,104.19473266601562
2010-08-26,84.66507720947266,104.36309814453125
2010-08-27,82.26773071289062,103.76029205322266
2010-08-30,83.8450698852539,104.30110931396484
2010-08-31,84.77436828613281,104.48726654052734
2010-09-01,83.02153015136719,104.154296875
```

卡尔曼滤波程序如下所示:

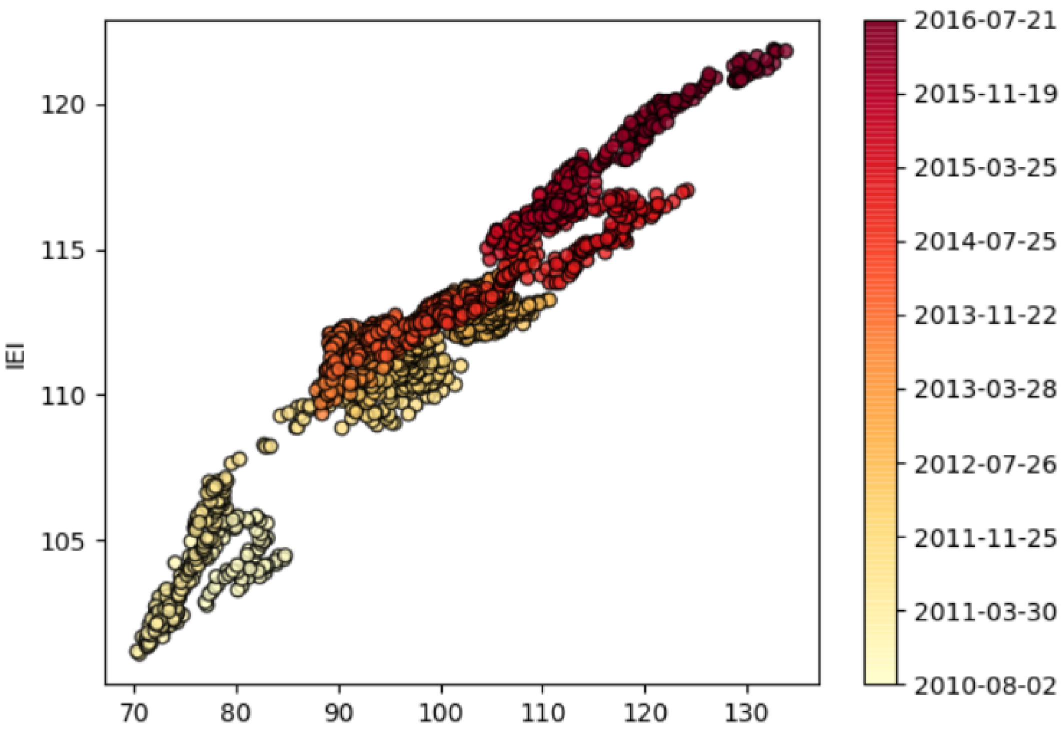
```
    def hedge_ratio(self):
        etfs = ['TLT', 'IEI']
        start_date = "2010-8-01"
        end_date = "2016-08-01"
        # 获取调整后的收盘价
        dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d')
        prices = pd.read_csv('./work/aqt005_001.txt', encoding='utf-8',
parse_dates=['Date'], date_parser=dateparse, index_col='Date')
        # 画散点图
        self.draw_date_coloured_scatterplot(etfs, prices)
        state_means, state_covs = self.calc_slope_intercept_kalman(etfs,
prices)
        self.draw_slope_intercept_changes(prices, state_means)

    def draw_date_coloured_scatterplot(self, etfs, prices):
        """
        生成散点图，以交易对中两个标的为坐标轴，可以直观观察两个标的间的
关系，并且
        采用黄色代表早期数据而红色代表近期数据
        @param etfs 标的字符列表
        @param prices 价格列表，每行为日期、TLT价格、IEI价格
        """
        plen = len(prices)
        colour_map = plt.cm.get_cmap('YlOrRd')
        colours = np.linspace(0.1, 1, plen)
        # 生成散点图
        scatterplot = plt.scatter(
        prices[etfs[0]], prices[etfs[1]],
        s=30, c=colours, cmap=colour_map,
        edgecolor='k', alpha=0.8
        )
        # 添加颜色图表
        colourbar = plt.colorbar(scatterplot)
        colourbar.ax.set_yticklabels(
        [str(p.date()) for p in prices[::plen//9].index]
        )
        plt.xlabel(prices.columns[0])
        plt.ylabel(prices.columns[1])
        plt.show()

    def calc_slope_intercept_kalman(self, etfs, prices):
        """
        使用卡尔曼滤波器预测对冲比例
        """
        delta = 1e-5
        mu0 = np.zeros(2)
        sigma0 = np.ones((2, 2))
        Q = delta / (1 - delta) * np.eye(2)
        At = np.eye(2)
        Ct = np.vstack(
            [prices[etfs[0]], np.ones(prices[etfs[0]].shape)]
        ).T[:, np.newaxis]
        R = 1.0
        kf = KalmanFilter(
            n_dim_obs=1,
            n_dim_state=2,
            initial_state_mean=mu0,
            initial_state_covariance=sigma0,
            transition_matrices=At,
            observation_matrices=Ct,
            observation_covariance=R,
            transition_covariance=Q
```

```python
        )

        yt = prices[etfs[1]].values
        #state_means, state_covs = kf.em(observations).filter(
    observations)
        xt_means, xt_covs = kf.em(yt).filter(yt)
        return xt_means, xt_covs

    def draw_slope_intercept_changes(self, prices, state_means):
        """
        绘制斜率和截距
        """
        pd.DataFrame(
            dict(
                slope=state_means[:, 0],
                intercept=state_means[:, 1]
            ),
            index=prices.index
        ).plot(subplots=True)
        plt.show()
```

Listing 15: 卡尔曼滤波示例

我们首先从数据文件中读出调整后的收盘价格数据，然后以 TFT 为横轴，IEI 为纵轴，绘制出这两个标的间的对应关系。如下所示：

Figure 54: 散点图



由图**??**可以看出，二者之间有一个相对稳定的线性关系，这是我们应用交易对交易策略的基础。
我们接着利用卡尔曼滤波来预测对冲比例，为了讲述方便，我们把 PyKalman 库中的公式重新列在这里：

$$\boldsymbol{x}_{t-1} = A_t \boldsymbol{x}_{t-1} + \boldsymbol{b}_t + \boldsymbol{\epsilon}_t^1 \tag{76}$$

$$\boldsymbol{y}_t = C_t \boldsymbol{x}_t + \boldsymbol{d}_t + \boldsymbol{\epsilon}_t^2 \tag{77}$$

$$\boldsymbol{\epsilon}_t^1 \in \mathcal{N}(0, Q) \tag{78}$$

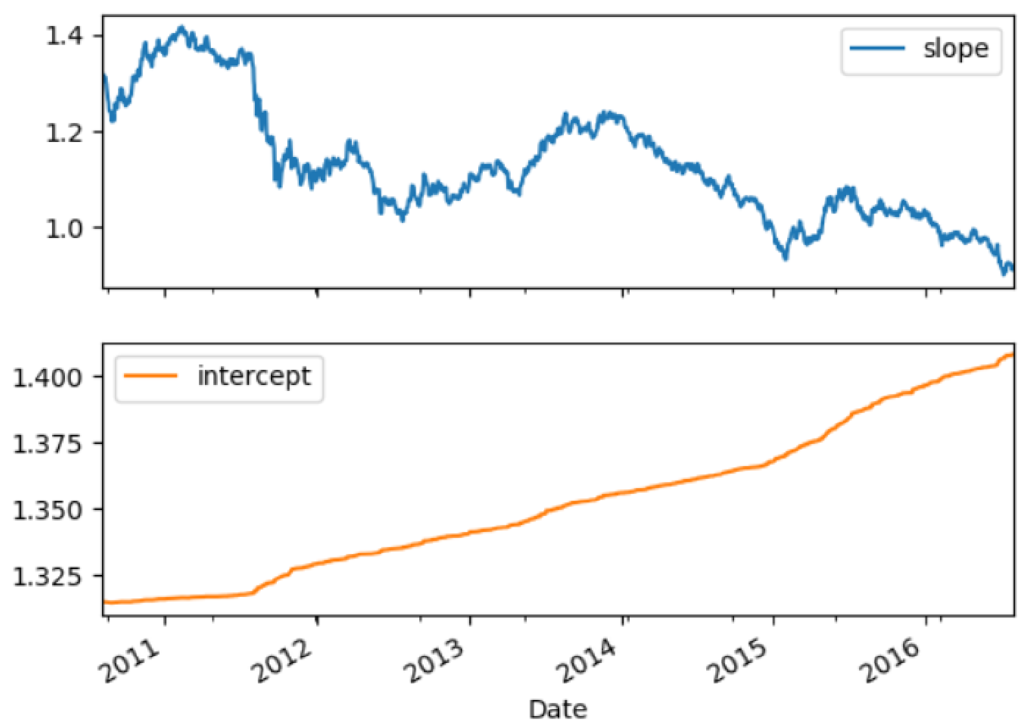
$$\boldsymbol{\epsilon}_t^2 \in \mathcal{N}(0, R) \tag{79}$$

我们将 TLT 的价格，后面加上 1 之后形成向量，将当前所有时间点的向量组合到一起，形成的矩阵作为观察矩阵，即公式中的 C_t。观察 y_t 在这里为 1 维，因此初始时其协方差矩阵为 1。我们按照给定的初始初始化一个卡尔曼滤波器对象，首先调用 em() 方法对参数进行估计，然后利估计出的参数，运行滤波功能，得到本时刻系统状态即对冲比例。对于每个时间点，其公式为：

$$y_t = \begin{bmatrix} c_t & 1 \end{bmatrix} \begin{bmatrix} x_{t,1} \\ x_{t,2} \end{bmatrix} \quad => \quad IEI.price = \begin{bmatrix} TLT.price & 1 \end{bmatrix} \begin{bmatrix} slope \\ intercept \end{bmatrix} \tag{80}$$

注意：在上式中，我们将截距项 $\boldsymbol{d}_t$ 作为向量中的一维来表示，因此式中就没有这一项了。运行上面的程序，就可以得到各个时间点的斜率和截距，我们接着绘制斜率和截距随时变化的曲线，如下所示：

Figure 55: 对冲比例变化曲线



图**??**中，上面蓝色的曲线为对冲比例变化曲线，可以看出，对冲比例并不是一成不变的，因此如果我们按照动态变化的对冲比例来设计交易策略，我们可以得到更高的收益。

第 6 章统计套利策略

Abstract

在本章中，我们将利用卡尔曼滤波技术，设计统计套利策略，利用我们的量化交易研究平台进行回测，初步验证策略的正确性。并将 GARCH 模型用于实际金融时间序列数据拟合。aqt002.py

6 统计套利策略

统计套利策略是一种市场中性策略，在实际中有大量应用。

6.1 数据获取

我们要研究交易策略，首先要获取历史交易数据，在本节中，我们将讲述如何通过 baostock 的接口，来获取历史交易数据。我们之所以选择 baostock 接口，因为 baostock 可以提供免费的 6 分钏的日内交易数据，这在免费数据中是很少见的。不过我们这一篇中，我们只研究日线交易数据。baostock 接口网站：

```
http://baostock.com/baostock/index.php
```

Listing 16: 卡尔曼滤波示例

7 汇总

f000022 c000010 e000070

第 6 章隐马可夫模型

Abstract

在本章中我们将首先讲述条件异方差模型 GARCH（Generalized AutoRegressive Conditional Heteroskedastic），并将 GARCH 模型用于实际金融时间序列数据拟合。aqt002.py

8 隐马可夫模型

第 8 章机器学习

Abstract

在本章中我们将首先讲述条件异方差模型 GARCH（Generalized AutoRegressive Conditional Heteroskedastic），并将 GARCH 模型用于实际金融时间序列数据拟合。aqt002.py

9 机器学习

第 9 章高频交易

Abstract

在本章中我们将首先讲述条件异方差模型 GARCH（Generalized AutoRegressive Conditional Heteroskedastic），并将 GARCH 模型用于实际金融时间序列数据拟合。aqt002.py

10 高频日内交易

第 10 章程序化 CTA

Abstract

在本章中我们将首先讲述条件异方差模型 GARCH（Generalized AutoRegressive Conditional Heteroskedastic），并将 GARCH 模型用于实际金融时间序列数据拟合。aqt002.py

11 程序化 CTA

第 100 章 Transformer 量化平台

Abstract

在本章中我们将首先讲述去年年末在自然语言处理 NLP 中最流行的架构 Transformer，然后介绍 Transformer 在股价预测方面的应用，接着利用 TensorFlow Serving 搭建策略服务，我们将该策略服务放到回测平台上进行测试，最后在实盘模拟平台上进行模拟交易。app/tqp

12 Transformer 量化平台

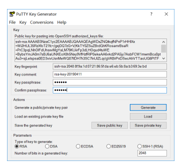
12.1 Transformer 框架

12.2 Transformer 量化应用

12.2.1 搭建 colab 开发环境

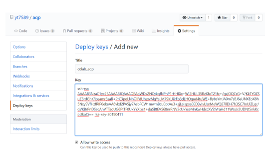
配置 Github 首先到 *https : //www.puttygen.com/* 网站下载 puttygen 的 windows 版本，用于生成访问 github 项目的公钥和私钥。以管理员身份运行 puttygen.exe，点击界面中的"Generate"：

Figure 56: 生成公钥和私钥对



在图中的密码为"123456"，点击保存私钥，将其保存到 F:/docs/aqp_github.ppk 文件中。
登录 https://github.com/yt7589/aqp，在 settings=>deploy keys 中添加新的 Key，将图54中 Public Key 框中内容拷贝到下面的文本框中，如下所示：

Figure 57: 将公钥添加到 github 中



接下来在 Google colab 中创建 aqp.ipynb 项目，添加代码单元格：

```
key = \
'''
PuTTY-User-Key-File-2: ssh-rsa
Encryption: aes256-cbc
Comment: rsa-key-20190411
Public-Lines: 6
AAAAB3NzaC1yc2EAAAABJQAAAQEAgWDxZNQtkqfNPnP1rHHIXx+Wi2HUL35RzXfo
T21fc+/gxjOQ7zG+V/KkTY0ZSuZBrdGhKRosamrBsaR+PrC3pqLNhOlFdUhsxvMg
YaLM79KUiirFp3dLHOqud4tuWE+BybsYmJA0m7sBJ6aUNXEoX8h5Nxy9VfHzfRIP
0ekeAA6vkd2PA5jy7AsbFCW1mwmBcu0ptAu3+sjLeIspsa0ED3vivUsnMeWQ87l1
DH7h3SC7InUlZLqr/gVKBrPnDSecAhVTTaoUG6PtTF0UkYYXixs1+daSBtEV568n
rRNV2cUkYseMnKwHdccXV2iVraHd11WsxJr2UDNt5nkKcpUkoQ==
Private-Lines: 14
```

```
qBX9VNU5GZoUAPvntTt8IRI7YqXfVgmKNVWDeyRFRGBlBsMVsU/o1AEm2pO9plHz
6vAuIje1fg70kDXcBuRCBiEJZ8IDdkNR5nqcgSFEnn2e/5UpisrA4FmGD99sSmWl
XJQAUEw5d7rxpQukiF6H19cJI+9e5vj9WIIKrCCXDawJyKaMQw0nn1zMJHsv6o21
MFlZqD8UUP4eoYrzRk2ga99O99b1MdYIwo2dUR40RQUGshKqIKdLHyiSBuy3gQm1
bmHNyrdMqCAN/88wadQRZvuyRudVXQAGbgYvDgZZV00EbiSOKPpmneK6DvgPaXea
J/1F+wAoZY8LsXZWlP2o44DjeJNmtc9eVPoR4b/hrG5fyMt9OmnCmL6Te6yue59L
pjH1gyvIMuoo3La71y8mlLKH8Hnb+bM5z6Ew7f/oMXzP9fNEdIsDN11L4WY9Gju1
GY0TUzi4h4li4K4Khzx1lOPDyaJahArx5iobD5xbemb4kj4OMRdsuQDqD7g32zVo
e8vVMgv8j3bzIpCAYmAmVhxslcpiFEUwjRtR6eyz7OL7g9IpkeG5fezGYMOLXxko
A1SWbIwqfODQhPJA1Pj3vnys8XVeQU04hplqyn955bwxl4abhVR3Q05E7DMYfdrR
f+a2VI4tAoWi7LCgskDnZ5754P3mUFkxSbILLt6tW+1Y4jbmPj6tP8GUI3Dm3zK6
qQAGkp3U9QeYlwsbO8lkTuWcW+NEyTfekjxnLtt1KLZufsKRDD1Dk4vQiXjKRCOU
UgjZ2Uui24d6N6ug/uiVNuxrDuZbpWpp2SYZEiuQCNhu3br+Q/+ZEZsmgslLewnr
ocK3qv7FUKq/Jg1QrjOmLq/RF1hVr+XcXzubiDcEdmZpFzsn7/VKHk/Jvv4f6IOd
Private-MAC: e16aabf8a23afa92f89fdeb3f5b2d36c20afb0ff

'''
! mkdir -p /root/.ssh
with open(r'/root/.ssh/id_rsa', 'w', encoding='utf8') as fh:
    fh.write(key)
! chmod 600 /root/.ssh/id_rsa
! ssh-keyscan gitlab.com >> /root/.ssh/known_hosts
```

Listing 17: 初始化 github 私钥

在上面的代码中，先取出私钥文件中的内容，然后创建/root/.ssh 目录，将私钥内容保存到/root/.ssh/id_rsa 文件中。运行上面的代码单元格，如果不报错就证明可以运行成功了。

下载源码文件 可以通过如下命令获取 github 上的源码：

```
!git clone https://github.com/yt7589/aqp.git
```

Listing 18: 初始化 github 私钥

配置 Google Drive 文件系统 我们需要读取保存在 Google Drive 中的数据集文件，因此我们需要建立一些特别工具来完成下面的内容。我们首先初始化环境：

```
from google.colab import drive
drive.mount('/content/drive')
```

Listing 19: 初始化环境

这时需要访问提示中的网站，同意授权，将页面中授权码拷贝，将授权码粘贴到编辑框中并回车，然后我们就可像使用正常的目录一样来读写文件了。

```
with open('/content/drive/My Drive/fbm/README.md', 'r') as fd:
  readme = fd.read()
  print(readme)
```

Listing 20: 初始化环境

其中 fbm/是我们在 Google Drive 中创建的目录，使用这种方法，也不会出现中文乱码的问题。具体用 Transformer 来预测股票价格的代码，colab 的地址为：

```
https://colab.research.google.com/drive/1PVwlbHfrndFhnNIjSLWaD1bsr-5olV1d#scrollTo=7s5HTXF9xall。
参考：
\lstset{language=BASH}
\begin{lstlisting}
https://github.com/llSourcell/Make_Money_with_Tensorflow_2.0/issues
```

Listing 21: 初始化环境

12.3 tqp 平台

12.4 策略回测

12.5 实盘模拟

参考文献：**?—?—?**

References

Michael L. Halls-Moore. *Advanced Algorithmic Trading*. His Publisher, Publisher, 2017.

PyKalman. Pykalman 0.9.2 文档. *http://pykalman.github.io*, blog, 2019. URL http://pykalman.github.io/#pykalman.KalmanFilter.sample.

tushare and 匠芯量化. 机器量化分析（二）——模型评估与仓位管理. *tushare.org*, 平台介绍, 2019. URL https://tushare.pro/document/1?doc_id=68.