
高级量化交易技术

闫涛
科技有限公司
北京
{yt7589}@qq.com

第一篇统计套利

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 Coefficient 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：

$$E(x) = \mu \quad (1)$$

方差定义为：

$$\sigma^2(x) = E[(x - \mu)^2] \quad (2)$$

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：

$$\sigma(x, y) = E[(x - \mu_x)(y - \mu_y)] \quad (3)$$

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：

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

随机变量 x 和 y 的相关系数定义为：

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

采用统计量的表示方法为：

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

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。

1.2 时序序列平稳性

时序信号 x_t 的均值定义为：

$$E(x_t) = \mu(t) \quad (7)$$

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：

$$\bar{x} = \frac{1}{N} \sum_{t=1}^N x_t \quad (8)$$

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：

$$\sigma^2(t) = E[(x_t - \mu)^2] \quad (9)$$

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可以化简为：

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

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：

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

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：

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

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：

$$\rho_k = \frac{C_k}{\sigma^2} \quad (13)$$

由定义可知：

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

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：

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

自相关系数函数 ACF 可以定义为：

$$r_k = \frac{c_k}{c_0} \quad (16)$$

1.5 自相关性举例

下面我们以上证综指时间序列为列，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：

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import matplotlib.dates as mdates
4 from matplotlib.font_manager import FontProperties
5 from statsmodels.tsa import stattools
6 from statsmodels.graphics import tsaplots
7
8 class Chp023(object):
9     def __init__(self):
10         self.name = 'Chp022'
11         # 数据文件格式: 编号 日期 星期几 开盘价 最高价
12         # 最低价 收益价 收益
13         self.data_file = 'data/pqb/chp023_001.txt'
14
15     def startup(self):
16         print('第章: 时间序列基本性质23')
17         data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
18         sh_index = data[data.Indexcd==1]
19         sh_index.index = pd.to_datetime(sh_index.index)
20         sh_return = sh_index.Retindex
21         print('时间序列长为: N={0}'.format(len(sh_return)))
22         acfs = stattools.acf(sh_return)
23         print(acfs)
24         pacfs = stattools.pacf(sh_return)
25         print(pacfs)
26         tsaplots.plot_acf(sh_return, use_vlines=True, lags=30)
27         plt.show()
28         tsaplots.plot_pacf(sh_return, use_vlines=True, lags=30)
29         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 时间序列长度为：N=311						
[1.	0.03527714	-0.01179861	-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	0.04594213	-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	-0.0131316	-0.04551243	-0.07680743		
0.00233848	0.03539094	-0.01313388	-0.02897258	0.16483494	-0.06656327	
0.01423998	0.03690183	-0.01674841	0.03307673	0.05240115	-0.05902877	
0.03637376	0.07184167	0.04734467	0.00131723	0.10809391	0.07778589	
0.02235463	-0.04041105	0.06533274	-0.07473672	0.05527639	-0.1271887	
-0.16407524	0.0362306	0.04578521	0.108878	0.10892202	-0.04912167	
-0.03517882	0.04134873	-0.02148459	-0.03297278	0.05326165	0.04072968	

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：

$$threshold = \frac{1.96}{\sqrt{N}} = \frac{1.96}{\sqrt{311}} = 0.11 \quad (17)$$

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象的表示出来，自相关系数函数图如所示：

Figure 3: 自相关系数函数图

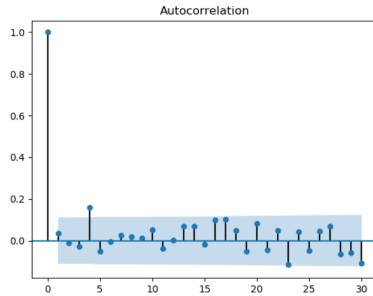
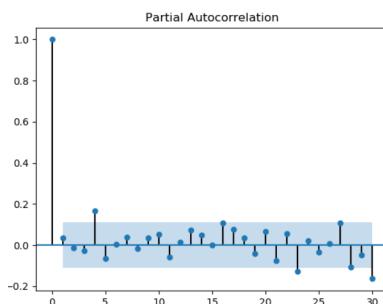


图3中蓝色区域的界限为 $[-\frac{1.96}{\sqrt{N}}, \frac{1.96}{\sqrt{N}}] = [-0.11, 0.11]$ ，除第 1 项外，其他项如果超出蓝色区域则说明此时间序列具有自相关性。

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 $x_t \sim N(0, \sigma^2)$ ，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：

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

我们定义差分运算符为：

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

1.6.3 白噪声定义

对于时间序列 $\{w_t, t = 1, 2, 3, \dots, N\}$ ，满足 $\forall t \quad w_t \sim N(0, \sigma^2)$ 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：

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

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：

$$\begin{aligned} x_t &= x_{t-1} + w_t = Bx_t + w_t \\ x_t &= x_{t-1} + w_t = x_{t-2} + w_{t-1} + w_t \\ &\quad \dots \\ x_t &= w_1 + w_2 + \dots + w_{t-1} + w_t \end{aligned} \quad (21)$$

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：

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

我们再来看随机游走序列的自相关系数函数 ACF：

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

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：

```

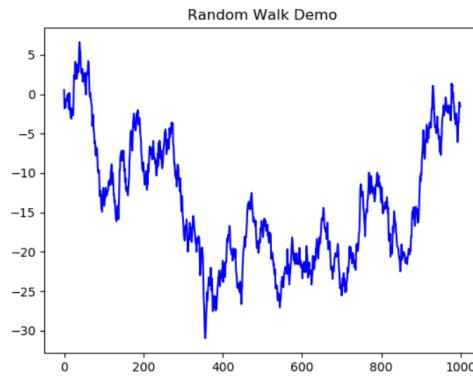
1 def random_wale_demo(self):
2     """
3     随机游走时间序列建模示例
4     """
5     w = np.random.standard_normal(size=1000)
6     x = w
7     for t in range(1, len(w)):
8         x[t] = x[t-1] + w[t]
9     plt.plot(x, c='b')
10    plt.title('Random Walk Demo')
11    plt.show()
12    acfs = stattools.acf(x)
13    print(acfs)
14    tsaplots.plot_acf(x, use_vlines=True, lags=30)
15    plt.show()
16    # 拟合随机游走信号
17    r = []
18    for t in range(1, len(x)):
19        r.append(x[t] - x[t-1])
20    rd = np.array(r)
21    plt.plot(rd, c='r')
22    plt.title('Residue Signal')
23    plt.show()
24    rd_acfs = stattools.acf(rd)
25    print(rd_acfs)
26    tsaplots.plot_acf(rd, use_vlines=True, lags=30)
27    plt.show()

```

Listing 2: 随机游走过程模拟

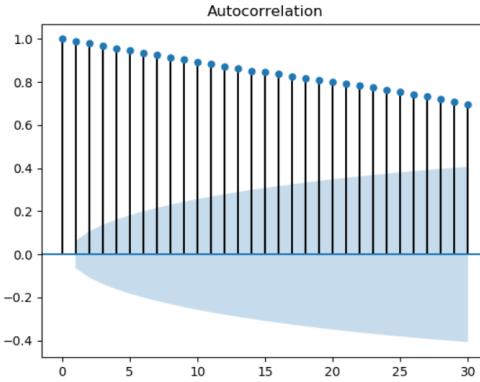
我们首先通过 $x_t = x_{t-1} + w_t$ 生成一个随机游走信号，该信号图形如下所示：

Figure 5: 随机游走时间序列信号



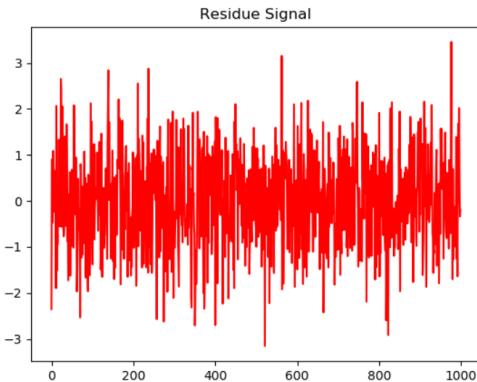
由图5可以看出，其非常像是一个股票收盘价的走势图，这也是为什么有些人说股票走势是随机游走过程了。接着我们求出该时间序列的自相关系数函数 ACF 及其自相关图，如下所示：

Figure 6: 随机游走时间序列信号



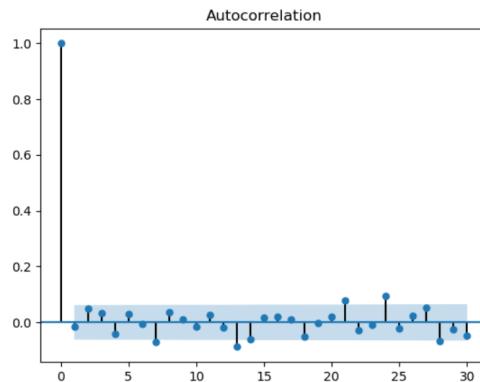
由图可以看出，其具有极强的自相关性，所有 ACF 值均位于蓝色置信区间之外。我们知道 $x_t - x_{t-1} = w_t$ ，而 w_t 是白噪声时间序列信号，这实际上模拟了实际应用过程，我们把 x_t 视为实际的金融信号，而 x_{t-1} 为我们建模的信号，将两个信号相减，得到残差信号，如果残差信号是白噪声信号，就可以认为我们建模是合理的。下面来看我们得到的残差信号：

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 0.88333481
 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  7.92334558e-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]
```

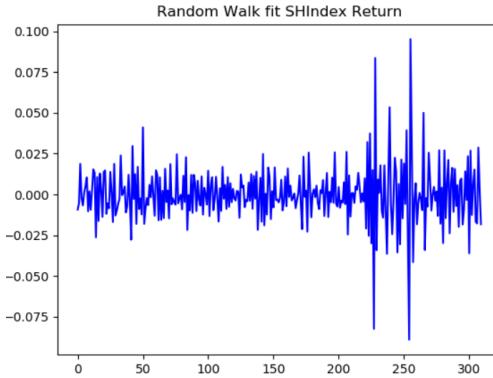
下面我们以上证综指收益率为例，来看随机游走模型是否可以很好的拟合这个时间序列，程序如下所示：

```
1 def random_walk_fit(self):
2     data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
3     sh_index = data[data.Indexcd==1]
4     sh_index.index = pd.to_datetime(sh_index.index)
5     sh_return = sh_index.Retindex
6     print('时间序列长为: N={0}'.format(len(sh_return)))
7     r = []
8     for t in range(1, len(sh_return)):
9         r.append(sh_return[t] - sh_return[t-1])
10    rd = np.array(r)
11    plt.plot(rd, c='b')
12    plt.title('Random Walk fit SHIndex Return')
13    plt.show()
14    rd_acfs = stattools.acf(rd)
15    print(rd_acfs)
16    tsaplots.plot_acf(rd, use_vlines=True, lags=30)
17    plt.show()
```

Listing 3: 随机游走拟合上证综指收益率

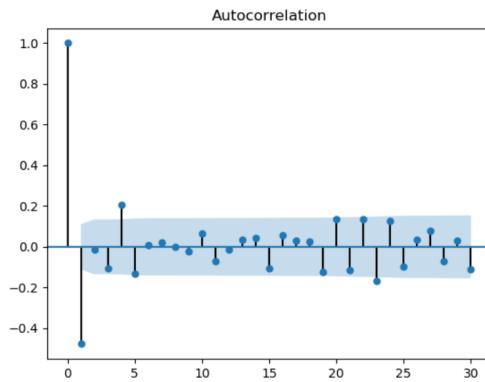
其残差图像为：

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. 0000000e+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。

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。下面我们将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： $x_{t_1}, x_{t_2}, \dots, x_{t_N}$ 和 $x_{t_1+m}, x_{t_2+m}, \dots, x_{t_N+m}$ 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：

$$AIC = -2 \log(L) + 2k \quad (24)$$

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：

$$x_t = x_{t-1} + w_t \quad (25)$$

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：

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

同样我们可以得到其方差：

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

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：

$$\nabla x_t = Bx_t = x_t - x_{t-1} = w_t \quad (28)$$

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：

$$Bx_t = x_{t-1} \quad B^2x_t = x_{t-2} \quad B^3x_t = x_{t-3} \quad \dots \quad B^n x_t = x_{t-n} \quad (29)$$

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：

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

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：

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

其所对应的滞后算子多项式为：

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

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：

$$y_t = \gamma + \rho_1 y_{t-1} + \rho_2 y_{t-2} + \dots + \rho_p y_{t-p} + w_t \quad (33)$$

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：

$$y_t = \gamma + \rho_1 B y_t + \rho_2 B^2 y_t + \dots + \rho_p B^p y_t + w_t \quad (34)$$

将式34右边所有包含 y_t 的项都移到左边，可以得到下式：

$$(1 - \rho_1 B - \rho_2 B^2 - \dots - \rho_p B^p) y_t = \gamma + w_t \quad (35)$$

可以得到其对应的滞后算子多项式方程为：

$$1 - \rho(B) = 1 - \rho_1 B - \rho_2 B^2 - \dots - \rho_p B^p = 0 \quad (36)$$

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：

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \delta_1 \Delta y_{t-1} + \delta_2 \Delta y_{t-2} + \dots + \delta_p \Delta y_{t-p} + w_t \quad (37)$$

其中 α 对应截距， β 对应趋势， $\delta_1 \Delta y_{t-1} + \delta_2 \Delta y_{t-2} + \dots + \delta_p \Delta y_{t-p}$ 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：

$$ADF(y, lags, trend, maxlags, method) \quad (38)$$

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列, 我们首先需要安装 python 的 garch 库:

```
1 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:

```
1 import arch.unitroot as unitroot
2 .....
3 def adf_demo(self):
4     print('检验例程ADF... ')
5     data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
6     sh_index = data[data.Indexcd==1]
7     sh_index.index = pd.to_datetime(sh_index.index)
8     sh_return = sh_index.Retindex
9     sh_return_adf = unitroot.ADF(sh_return)
10    print(sh_return_adf.summary().as_text())
11    print('stat={0:0.4f}; pvalue={0:0.4f}'.format(sh_return_adf.stat,
12        sh_return_adf.pvalue))
13    print('critical_values:{0}'.format(sh_return_adf.critical_values))
14
15    print('1%value={0}'.format(sh_return_adf.critical_values['1%']))
16    if sh_return_adf.stat < sh_return_adf.critical_values['1%']:
17        print('上证综指收益率为平稳时间序列 ^_^')
18    else:
19        print('上证综指收益率为非平稳时间序列 !!!!!!!')
20
21    sh_close = sh_index.Clsindex
22    sh_close_adf = unitroot.ADF(sh_close)
23    if sh_close_adf.stat < sh_close_adf.critical_values['1%']:
24        print('上证综指收盘价为平稳时间序列 ^_^')
25    else:
26        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：

$$x_t = \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \dots + \alpha_p x_{t-p} + w_t = \sum_{i=1}^p \alpha_i x_{t-i} + w_t \quad (39)$$

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：

$$\theta_p(B)x_t = (1 - \alpha_1 B - \alpha_2 B^2 - \dots - \alpha_p B^p)x_t = w_t \quad (40)$$

有了上述模型之后，我们就可以直接拿来作预测，如下所示：

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

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：

$$\theta_p(B) = 0 \quad (42)$$

解这个方程得到的解的绝对值必须大于 1 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:

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import matplotlib.dates as mdates
5 from matplotlib.font_manager import FontProperties
6 from statsmodels.tsa import stattools
7 from statsmodels.graphics import tsaplots
8 from statsmodels.tsa.arima_model import ARIMA
9
10 class Aqt001(object):
11     def __init__(self):
12         self.name = 'Aqt001'
13
14     def startup(self):
15         print('模型ARMA... ')
16         self.simulate_ar2()
17
18     def simulate_ar2(self):
19         print('模拟AR(2)')
20         alpha1 = 0.666
21         alpha2 = -0.333
22         wt = np.random.standard_normal(size=1000)
23         x = wt
24         for t in range(2, len(wt)):
25             x[t] = alpha1 * x[t-1] + alpha2 * x[t-2] + wt[t]
26         plt.plot(x, c='b')
27         plt.show()
28         ar2 = stattools.ARMA(x, (2, 0)).fit(disp=False)
29         print('p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'.format(
30             ar2.k_ar, ar2.arparams, ar2.k_ma, ar2.maparams,
```

```

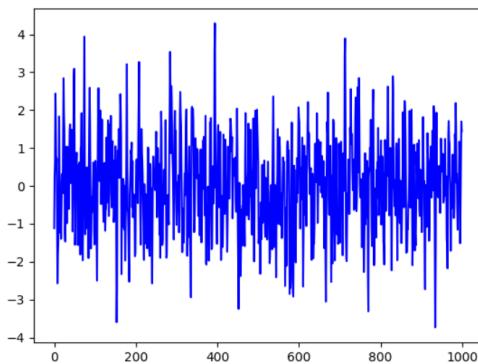
31             ar2.aic, ar2.bic, ar2.hqic)
32     )
33     arima2_0_0 = ARIMA(x, order=(2, 0, 0)).fit(disp=False)
34     print('ARIMA: p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'. \
35           format(arima2_0_0.k_ar, arima2_0_0.arparams,
36                   arima2_0_0.k_ma, arima2_0_0.maparams,
37                   arima2_0_0.aic, arima2_0_0.bic,
38                   arima2_0_0.hqic))
39   )
40   resid = arima2_0_0.resid
41   # 绘制ACF
42   acfs = stattools.acf(resid)
43   print(acfs)
44   tsaplots.plot_acf(resid, use_vlines=True, lags=30)
45   plt.title('ACF figure')
46   plt.show()
47   pacfs = stattools.pacf(resid)
48   print(pacfs)
49   tsaplots.plot_pacf(resid, use_vlines=True, lags=30)
50   plt.title('PACF figure')
51   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，然后计算出模型选择的参数：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



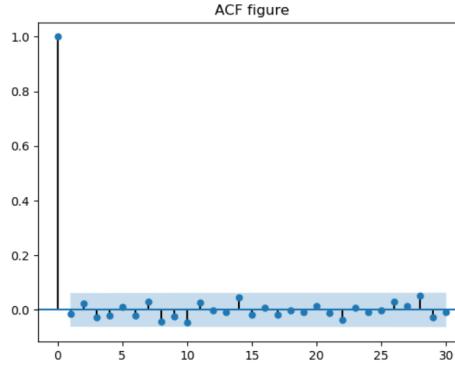
无论是 ARMA 还是 ARIMA 拟合，我们都可以得到较为正确的数据，同时残差的 ACF 和 PACF 也表明其是白噪声序列，运行结果如下所示：

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. 0000000e+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. 0000000e+00 -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]
```

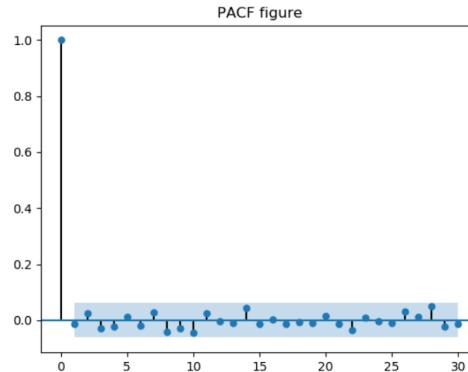
残差的自相关系数函数 ACF 图:

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：

$$\begin{aligned} & \{x_1, x_2, \dots, x_N\} \\ & \{d_1^1, d_2^1, \dots, d_{N-1}^1\} = \{x_2 - x_1, x_3 - x_2, \dots, x_N - x_{N-1}\} \\ & \{d_1^2, d_2^2, \dots, d_{N-1}^2\} = \{d_2^1 - d_1^1, d_3^1 - d_2^1, \dots, d_{N-1}^1 - d_{N-2}^1\} \end{aligned} \quad (44)$$

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：

$$BIC = -2 \log(L) + k \log N \quad (45)$$

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：

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

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：

$$x_t = \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \dots + \alpha_p x_{t-p} + w_t + \beta_1 w_{t-1} + \beta_2 w_{t-2} + \dots + \beta_q w_{t-q} \quad (47)$$

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：

$$\theta_p(B)x_t = \phi_q(B)w_t \quad (48)$$

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：

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

生成模拟数据并利用 ARIMA 拟合的程序如下所示：

```

1 def simulate_arima_p_d_q(self):
2     print('模拟ARIMA(p,d,q)过程')
3     np.random.seed(8)
4     alpha1 = 1.2
5     alpha2 = -0.7
6     beta1 = -0.06
7     beta2 = -0.02
8     w = np.random.standard_normal(size=1000)
9     x = w
10    for t in range(2, len(w)):
11        x[t] = alpha1 * x[t-1] + alpha2*x[t-2] + w[t] + beta1 * w[t-1] + beta2*w[t-2]
12        plt.plot(x, c='b')
13    plt.title('ARIMA(p, d, q) Figure')
14    plt.show()
15    # 查看ACF
16    acfs = stattools.acf(x)
17    print('ARIMA(q,d,q) ACFS:\r\n{}' .format(acfs))
18    tsaplots.plot_acf(x, use_vlines=True, lags=30)
19    plt.title('ARIMA(p,d,q) ACF')
20    plt.show()
21    # 拟合ARIMA
22    min_ABQIC = sys.float_info.max
23    arima_model = None
24    break_loop = False
25    ...
26    for p in range(0, 5):
27        if break_loop:
28            break
29        for q in range(0, 5):
30            print('try {0}, d, {1}...' .format(p, q))
31            try:
32                arima_p_d_q = ARIMA(x, order=(p, 0, q)).fit(disp=False)
33                print('..... fit ok')
34                if arima_p_d_q.aic < min_ABQIC:
35                    print('..... record good model')
36                    min_ABQIC = arima_p_d_q.aic
37                    arima_model = arima_p_d_q

```

```

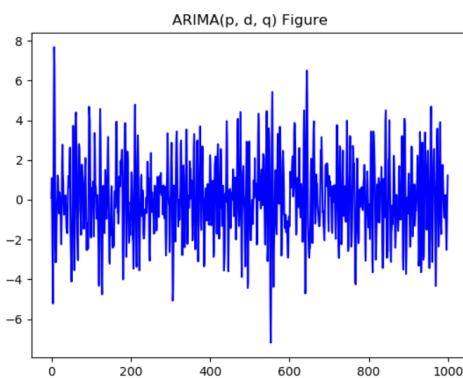
38             #if l==p and 1==q:
39             #    break_loop = True
40         except Exception as ex:
41             print('.....!!!!!! Exception')
42         print('ARIMA: p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'. \
43             format(arima_model.k_ar, arima_model.arparams,
44             arima_model.k_ma, arima_model.maparams,
45             arima_model.aic, arima_model.bic,
46             arima_model.hqic)
47     """
48
49     arima_model = ARIMA(x, order=(2, 0, 2)).fit(disp=False)
50     print('God_View:ARIMA: p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'. \
51             format(arima_model.k_ar, arima_model.arparams,
52             arima_model.k_ma, arima_model.maparams,
53             arima_model.aic, arima_model.bic,
54             arima_model.hqic)
55 )
56     resid = arima_model.resid
57     # 绘制ACF
58     acfs = stattools.acf(resid)
59     print(acfs)
60     tsaplots.plot_acf(resid, use_vlines=True, lags=30)
61     plt.title('ARIMA(p,d,q) ACF figure')
62     plt.show()
63     pacfs = stattools.pacf(resid)
64     print(pacfs)
65     tsaplots.plot_pacf(resid, use_vlines=True, lags=30)
66     plt.title('ARIMA(p,d,q) PACF figure')
67     plt.show()

```

Listing 7: AR 数据模拟和拟合示例

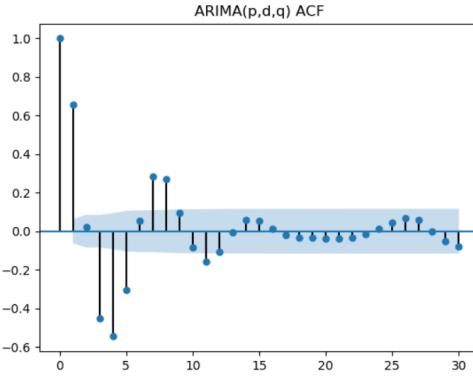
生成的时序信号 x 为:

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。

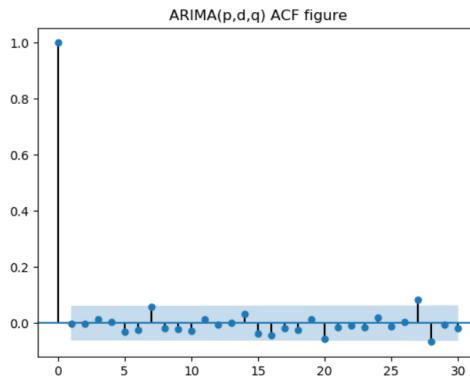
接着我们用 ARIMA 模型来模拟该信号，上面程序注释部分为求最佳 ARIMA 模型的 p 和 q 参数，以 AIC 作为模型选择标准，因为我们知道模型为 ARIMA(2,0,2)，所以我们同时也用 ARIMA(2,0,2) 来进行拟合，程序运行结果如下所示：

Figure 20: 程序运行结果

```
量化投资以python为工具
ARIMA模型
模拟ARIMA(p, d, q) ACFs:
[ 1.0000000e+00  6.56463444e-01  2.20934132e-02 -4.52132579e-01
-5.5049851e-01 -3.05649369e-01  5.20559164e-02  2.84818841e-01
2.72038623e-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]
Gd.View:ARIMA: p=2 *** [ 1.16735279 -0.72626584]; q=2***[-0.05184418 -0.06459538]; 2890.9503824872345 - 2920.3969141611
1274 - 2902.142119294227
[ 1.0000000e+00 -7.2774367e-04 -1.11010108e-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  1.43729502e-02
-4.98312176e-03  2.17342132e-03  3.36256042e-02 -3.60913979e-02
-4.268009658e-02 -1.72822287e-02 -2.50926415e-02  1.28230095e-02
-5.63492734e-02 -1.60329870e-02 -8.58773643e-03 -1.42536089e-02
1.91587444e-02 -1.10064043e-02  4.93949871e-03  8.31719158e-02
-6.380405533e-02 -5.60329946e-03 -1.79923901e-02 -1.18046998e-02
-3.380405533e-02  2.36484039e-02  1.66539078e-02  3.31933370e-02
5.39490149e-02 -6.82510840e-03  2.23837526e-03  3.81197763e-02
3.76186214e-03]
```

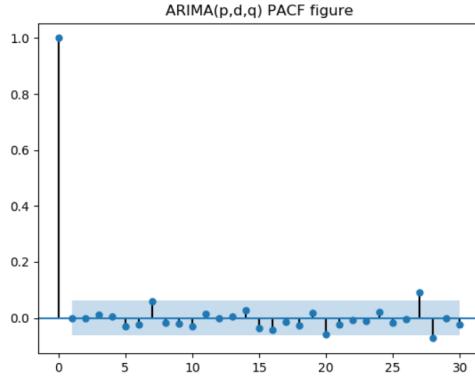
接着我们求出残差序列，残差序列自相关系数函数 ACF 图如下所示：

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：

```

1 def arima_demo(self):
2     register_matplotlib_converters()
3     data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
4     sh_index = data[data.Indexcd==1]
5     sh_index.index = pd.to_datetime(sh_index.index)
6     raw_data = sh_index.Clsindex
7     train_data = raw_data[:-3]
8     close_price = np.log(train_data)
9     plt.plot(close_price)
10    plt.show()
11    print(train_data.head(n=3))
12    # 拟合ARIMA
13    min_ABQIC = sys.float_info.max
14    arima_model = None
15    for p in range(0, 5):
16        for q in range(0, 5):
17            print('try {0}, d, {1}...'.format(p, q))
18            try:
19                arima_p_d_q = ARIMA(close_price, order=(p, 1, q)).fit
20                (disp=False)
21                print('..... fit ok')
22                if arima_p_d_q.aic < min_ABQIC:
23                    print('..... record good model')
24                    min_ABQIC = arima_p_d_q.aic
25                    arima_model = arima_p_d_q
26            except Exception as ex:

```

```

26         print('.....!!!!!! {0}'.format(ex))
27     print('ARIMA: p={0} **** {1}; q={2}***{3}; {4} - {5} - {6}'. \
28           format(arima_model.k_ar, arima_model.arparams,
29                   arima_model.k_ma, arima_model.maparams,
30                   arima_model.aic, arima_model.bic,
31                   arima_model.hqic)
32 )
33     resid = arima_model.resid
34 # 绘制ACF
35     acfs = stattools.acf(resid)
36     print(acfs)
37     tsaplots.plot_acf(resid, use_vlines=True, lags=30)
38     plt.title('ARIMA(p,d,q) ACF figure')
39     plt.show()
40     pacfs = stattools.pacf(resid)
41     print(pacfs)
42     tsaplots.plot_pacf(resid, use_vlines=True, lags=30)
43     plt.title('ARIMA(p,d,q) PACF figure')
44     plt.show()
45 # 检验ADF
46     resid_adf = unitroot.ADF(resid)
47     print('stat={0:0.4f} vs 1%_cv={1:0.4f}'.format(resid_adf.stat,
48             resid_adf.critical_values['1%']))
49     if resid_adf.stat < resid_adf.critical_values['1%']:
50         print('为稳定时间序列resid ^_^')
51     else:
52         print('为非稳定时间序列!!!! resid')
53 # Ljung-检验Box
54     resid_ljung_box = stattools.q_stat(stattools.acf(resid)[1:12],
55                                         len(resid))
56     resid_lbv = resid_ljung_box[1][-1]
57     print('resid_ljung_box_value={0}'.format(resid_lbv))
58     # 为显著性水平0.05
59     if resid_lbv < 0.05:
60         print('为平稳时间序列resid ^_^')
61     else:
62         print('为非平稳时间序列!!!!!! resid')
63 # 预测
64     y = arima_model.forecast(3)[0] #(len(train_data), len(raw_data),
65                               dynamic=True)
66     print('预测值: {0}'.format(np.exp(y)))
67     print('raw_data:{0}'.format(raw_data))
68     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



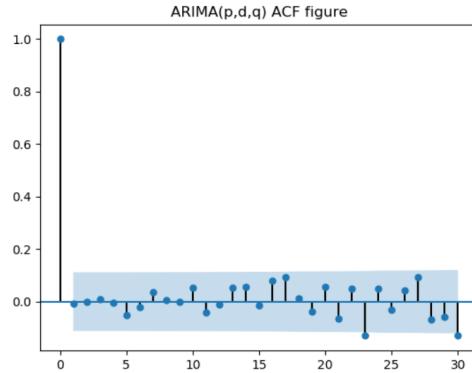
接着我们对该数据经过对数差分后，利用 ARIMA 来进行拟合，得到拟合模型为：

Figure 24: 拟合后的 ARIMA 模型

```
ARIMA: p=0 **** [ ] : q=4***[ 0.0533824 -0.01120743 -0.06658252 0.15689616]; -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 -0.03086033 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 ]
```

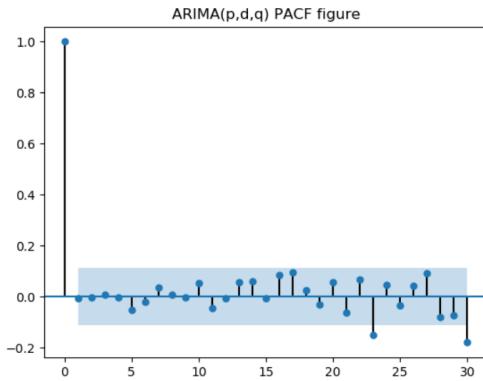
残差序列的自相关系数函数 ACF 图为：

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。

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：

$$\epsilon_t = \sigma_t w_t \quad (50)$$

其中 w_t 为白噪声序列， σ_t 的定义如下所示：

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

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：

$$E(\epsilon_t) = E(\sigma_t w_t) = E(\sigma_t)E(w_t) = 0 \quad (52)$$

我们再来看时间序列 $\{\epsilon_t\}$ 的方差：

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

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：

$$\epsilon_t = \sigma_t w_t \quad (54)$$

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：

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

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 ：

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

其程序如下所示：

```

1 import sys
2 import math
3 import numpy as np
4 import pandas as pd
5 from pandas.plotting import register_matplotlib_converters
6 import matplotlib.pyplot as plt
7 import matplotlib.dates as mdates
8 from matplotlib.font_manager import FontProperties
9 from statsmodels.tsa import stattools
10 from statsmodels.graphics import tsaplots
11 from statsmodels.tsa.arima_model import ARIMA
12 import arch.unitroot as unitroot
13 import arch as arch
14
15 class Aqt002(object):
16     def __init__(self):
17         self.name = 'Aqt001'
18         # 数据文件格式：编号 日期 星期几 开盘价 最高价
19         # 最低价 收益价 收益
20         # Indexcd Trddt Daywk Opnindex Hiindex
21         # Loindex Clsindex Retindex
22         self.data_file = 'data/pqb/aqt002_001.txt'
23
24     def startup(self):
25         print('模型GARCH... ')
26         np.random.seed(1)
27         alpha0 = 0.2
28         alpha1 = 0.5
29         beta1 = 0.3
30         samples = 10000 # 样本数量
31         w = np.random.standard_normal(size=samples)
32         epsilon = np.zeros((samples,), dtype=float)
33         sigma = np.zeros((samples,), dtype=float)

```

```

34     for i in range(2, samples):
35         sigma_2 = alpha0 + alpha1 * math.pow(epsilon[i-1], 2) + \
36                     beta1 * math.pow(sigma[i-1], 2)
37         sigma[i] = math.sqrt(sigma_2)
38         epsilon[i] = sigma[i]*w[i]
39     plt.title('epsilon signal')
40     plt.plot(epsilon)
41     plt.show()
42     # 绘制epsilonACFS
43     acfs = stattools.acf(epsilon)
44     tsaplots.plot_acf(epsilon, use_vlines=True, lags=30)
45     plt.title('epsilon ACF')
46     plt.show()
47     # 绘制epsilon pow2 ACF
48     acfs2 = stattools.acf(np.power(epsilon, 2))
49     tsaplots.plot_acf(np.power(epsilon, 2), use_vlines=True, lags=30)
50     plt.title('pow(epsilon,2) ACF')
51     plt.show()
52     # 拟合GARCH
53     am = arch.arch_model(epsilon, x=None, mean='Constant',
54                           lags=0, vol='Garch', p=1, o=0, q=1,
55                           power=2.0, dist='Normal', hold_back=None)
56     model = am.fit(update_freq=0)
57     # GARCH(1,1)参数
58     print('##### GARCH(1,1)参数 #####')
59     print('model_type:{0}'.format(type(model)))
60     print('mu={0:0.2f}; a0={1:0.2f}; a1={2:0.2f}; b1={3:0.2f}' \
61           .format(model.params['mu'], model.params['omega'],
62                   model.params['alpha[1]'], model.params['beta[1]']))
63     )
64     print('#####')
65     #print(model.summary())
66     # 残差信号
67     plt.title('GARCH(1,1) resid')
68     plt.plot(model.resid)
69     plt.show()
70     # 残差ACF
71     resid_acf = stattools.acf(model.resid)
72     tsaplots.plot_acf(model.resid, use_vlines=True, lags=30)
73     plt.title('GARCH(1,1) resid ACF')
74     plt.show()
75     # 检验ADF
76     resid_adf = unitroot.ADF(model.resid)
77     print('stat={0:0.4f} vs 1%_cv={1:0.4f}'.format(
78         resid_adf.stat, resid_adf.critical_values['1%']))
79     if resid_adf.stat < resid_adf.critical_values['1%']:
80         print('为稳定时间序列resid ^_^')
81     else:
82         print('为非稳定时间序列!!!! resid')
# Ljung-检验Box

```

```

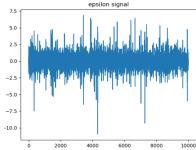
83     resid_ljung_box = stattools.q_stat(stattools.acf( \
84             model.resid)[1:12], len(model.resid))
85     resid_lbv = resid_ljung_box[1][-1]
86     print('resid_ljung_box_value={0}'.format(resid_lbv))
87     # 为显著性水平0.05
88     if resid_lbv < 0.05:
89         print('为平稳时间序列resid ^_^')
90     else:
91         print('为非平稳时间序列!!!!!! resid')
92     # 预测
93     y = model.forecast(horizon=3)
94     print('##### 预测值 #####')
95     print('len={0}; p1={1:0.3f}; p2={2:0.3f}; p3={3:0.3f}'. \
96             format(len(y.mean().iloc[-1]), y.mean().iloc[-1][0],
97                    y.mean().iloc[-1][1], y.mean().iloc[-1][2]))
98     print('#####')

```

Listing 9: GARCH 拟合模拟数据

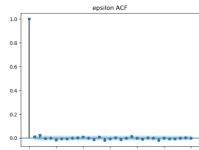
我们欲拟合的时间序列信号为:

Figure 31: 拟拟合的时间序列



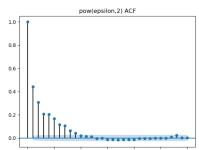
该时间序列的自相关系数函数 ACF 图:

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：

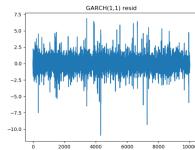
Figure 33: pow(epsilon,2) 的自相关系数函数 ACF 图



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。

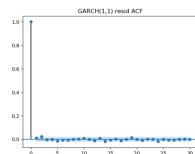
我们采用 GARCH 进行拟合后，得到的残差信号为：

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：

```
1 def garch_finance_demo(self):
2     print('拟合上证综指收盘价 ...')
3     register_matplotlib_converters()
4     data = pd.read_csv(self.data_file, sep='\t', index_col='Trddt')
5     sh_index = data[data.Indexcd==1]
6     sh_index.index = pd.to_datetime(sh_index.index)
7     sh_return = sh_index.Retindex
8     # raw_data = sh_index.Retindex
9     raw_data = sh_index.Clsindex
10    train_data = raw_data[:-3]
11    dcp = np.log(train_data).diff(1)[1:]
12    plt.plot(dcp)
13    plt.show()
14    # 拟合GARCH
15    am = arch.arch_model(dcp, x=None, mean='Constant',
```

```

16         lags=0, vol='Garch', p=3, o=0, q=2,
17             power=2.0, dist='Normal', hold_back=None)
18 model = am.fit(update_freq=0)
19 # GARCH(1,1)参数
20 print('##### GARCH(1,1)参数 #####')
21 ...
22 print('mu={0:0.2f}; a0={1:0.2f}; a1={2:0.2f}; a2={3:0.2f}, b1
23 ={4:0.2f}, b2={5:0.2f}' \
24     .format(model.params['mu'], model.params['omega'],
25             model.params['alpha[1]'], model.params['alpha[2]'],
26             model.params['beta[1]'], model.params['beta[2]'] ))
27 ...
28 resid = model.resid
29 # 绘制ACF
30 acfs = stattools.acf(resid)
31 print(acfs)
32 tsaplots.plot_acf(resid, use_vlines=True, lags=30)
33 plt.title('GARCH(p,q) ACF figure')
34 plt.show()
35 ...
36 pacfs = stattools.pacf(resid)
37 print(pacfs)
38 tsaplots.plot_pacf(resid, use_vlines=True, lags=30)
39 plt.title('ARIMA(p,d,q) PACF figure')
40 plt.show()
41 ...
42 # 检验ADF
43 resid_adf = unitroot.ADF(resid)
44 print('stat={0:0.4f} vs 1%_cv={1:0.4f}'.format(resid_adf.stat,
45     resid_adf.critical_values['1%']))
46 if resid_adf.stat < resid_adf.critical_values['1%']:
47     print('为稳定时间序列resid ^_^')
48 else:
49     print('为非稳定时间序列!!!! resid')
50 # Ljung-检验Box
51 resid_ljung_box = stattools.q_stat(stattools.acf(resid)[1:12],
52 len(resid))
53 resid_lbv = resid_ljung_box[1][-1]
54 print('resid_ljung_box_value={0}'.format(resid_lbv))
55 # 为显著性水平0.05
56 if resid_lbv < 0.05:
57     print('为平稳时间序列resid ^_^')
58 else:
59     print('为非平稳时间序列!!!!!! resid')
60 ...
61 # 预测
62 frst = model.forecast(horizon=3)
63 y = frst.mean.iloc[-1]
64 print('预测值: {0}'.format(y))
65 p1 = math.exp(math.log(train_data[-1]) + y[0])

```

```

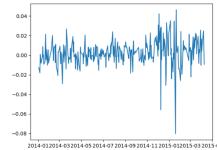
62     p2 = math.exp(math.log(p1) + y[1])
63     p3 = math.exp(math.log(p2) + y[2])
64     print('      预测值    实际值 (3957.534)')
65     print('第一天: {} vs 4034.310'.format(p1))
66     print('第二天: {} vs 4121.715'.format(p2))
67     print('第三天: {} vs 4135.565'.format(p3))

```

Listing 10: GARCH 拟合并预测上证综指收盘价

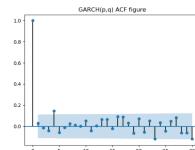
我们要建模的上证综指收盘价对数差分信号如下所示:

Figure 37: 上证综指收盘价对数差分信号



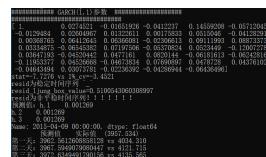
经过 GARCH(3,2) 拟合后的残差信号:

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



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。

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。

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，却不一定符合正态分布，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币市场的应用。

4.1 数据模拟

4.1.1 协整模型构建

根据⁷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：

$$z_t = z_{t-1} + w_t \quad (57)$$

在此基础上我们定义两个非平稳的时间序列信号：

$$x_t = p z_t + w_t = 0.3 z_t + w_t \quad (58)$$

$$y_t = q z_t + w_t = 0.6 z_t + w_t \quad (59)$$

如果我们将 $\{x_t\}$ 和 $\{y_t\}$ 进行线性组合:

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

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:

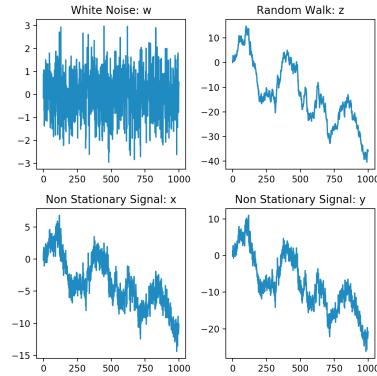
```
1 def simulate_demo(self):
2     """
3     模拟数据生成
4     """
5     # 生成白噪声信号
6     samples = 1000
7     w = np.random.standard_normal(size=samples)
8     # 生成随机游走序列
9     z = np.zeros((samples,))
10    for t in range(1, samples):
11        z[t] = z[t-1] + w[t]
12    # 生成非平稳信号, 即交易对和xy
13    x = np.zeros((samples,))
14    y = np.zeros((samples,))
15    p = 0.3
16    q = 0.6
17    for t in range(samples):
18        x[t] = p*z[t] + w[t]
19        y[t] = q*z[t] + w[t]
20    fig = plt.figure(figsize=(6, 6))
21    w_plt = plt.subplot(2, 2, 1, title='White Noise: w')
22    w_plt.plot(w)
23    z_plt = plt.subplot(2, 2, 2, title='Random Walk: z')
24    z_plt.plot(z)
25    x_plt = plt.subplot(2, 2, 3, title='Non Stationary Signal: x')
26    x_plt.plot(x)
27    y_plt = plt.subplot(2, 2, 4, title='Non Stationary Signal: y')
28    y_plt.plot(y)
29    fig.tight_layout()
30    plt.show()
31    # 生成协整信号
32    a = 2
33    b = -1
34    c = a * x + b * y
35    plt.plot(c)
36    plt.title('Cointegration Model')
37    plt.show()
38    # 采用检验ADF
39    resid_adf = unitroot.ADF(c)
40    print('stat={0:0.4f} vs 1%_cv={1:0.4f}'.format(
41        resid_adf.stat, resid_adf.critical_values['1%']))
42    if resid_adf.stat < resid_adf.critical_values['1%']:
43        print('为稳定时间序列resid ^_^')
44    else:
```

```
45     print('为非稳定时间序列!!!! resid')
```

Listing 11: 线性组合形成平稳时间序列

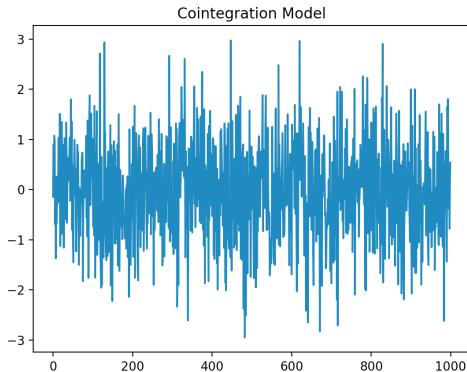
我们首先绘制出自噪声、随机游走和 x 及 y 信号：

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 检验其稳定性：

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：

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import tensorflow as tf
4
5 class QciLinearRegression(object):
6     def __init__(self, train_x=None, train_y=None,
7                  validate_x=None, validate_y=None,
8                  test_x=None, test_y=None):
9         self.name = 'QciLinearRegression'
10        self.loss = 'mean_squared_error'
11        self.learning_rate = 0.01
12        self.optimizer = tf.keras.optimizers.Adam(self.learning_rate)
13        self.epoch = 50000
14        if train_x:
15            self.train_x = train_x
16            self.train_y = train_y
17            self.validate_x = validate_x
18            self.validate_y = validate_y
19            self.test_x = test_x
20            self.test_y = test_y
21        else:
22            self.train_x, self.train_y, self.validate_x, \
23                self.validate_y, self.test_x, \
24                self.test_y = self.generate_dataset()
25
26    def generate_dataset(self):
27        train_x = np.array([-40, -10, 0, 8, 15, 22, 38, 20, 9, 13],\
28                           dtype=float)
29        train_y = np.array([-40, 14, 32, 46, 59, 72, 100, 68, 48.2, \
30                           55.4], dtype=float)
31        validate_x = np.array([], dtype=float)
32        validate_y = np.array([], dtype=float)
33        test_x = np.array([], dtype=float)
34        test_y = np.array([], dtype=float)
35        return train_x, train_y, validate_x, validate_y, test_x, \
36            test_y
37
38    def train(self):
39        model = self.build_model()
40        model.compile(loss=self.loss, optimizer=self.optimizer)
41        class PrintDot(tf.keras.callbacks.Callback):
42            def on_epoch_end(self, epoch, logs):
```

```

40             if epoch % 100 == 0: print(' ')
41             print('epoch:{0}...{1}'.format(epoch, logs))
42         early_stop = tf.keras.callbacks.EarlyStopping(monitor='
43             val_loss', patience=10)
44         history = model.fit(self.train_x, self.train_y,
45             epochs=self.epoch, validation_split=0.1,
46             verbose=False, callbacks=[early_stop, PrintDot()]
47         )
48         plt.title('linear regression training process')
49         plt.xlabel('epochs')
50         plt.ylabel('error')
51         plt.plot(history.history['loss'])
52         plt.show()
53         model.save('./work/aqt003_qiclr')
54         weights = np.array(model.get_weights())
55         print(weights)
56
57     def build_model(self):
58         layer1 = tf.keras.layers.Dense(units=1, input_shape=[1])
59         model = tf.keras.Sequential([layer1])
60         return model
61
62     def predict(self, data):
63         model = tf.keras.models.load_model('./work/aqt003_qiclr')
64         rst = model.predict(data)
65         return rst
66
67 if '__main__' == __name__:
68     lr = QciLinearRegression()
69     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：

$$F = 1.8 \times C + 32 \quad (61)$$

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

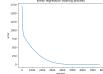
常会很耗时，我们用这个函数打印当前进度情况。接着我们调用 `model.fit` 来开始实际训练过程，在后台日志中我们可以看到，误差会逐渐减少，如图所示：

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：

```
1 data = np.array([[100.0]], dtype=float)
2 rst = qcllr.predict(data)
3 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：

```

1  def qcilr_demo(self):
2      # 生成白噪声信号
3      samples = 1000
4      w = np.random.standard_normal(size=samples)
5      # 生成随机游走序列
6      z = np.zeros((samples,))
7      for t in range(1, samples):
8          z[t] = z[t-1] + w[t]
9      # 生成非平稳信号, 即交易对和xy
10     x = np.zeros((samples,))
11     y = np.zeros((samples,))
12     p = 0.3
13     q = 0.6
14     for t in range(samples):
15         x[t] = p*z[t] + w[t]
16         y[t] = q*z[t] + w[t]
17     fig = plt.figure(figsize=(6, 6))
18     w_plt = plt.subplot(2, 2, 1, title='White Noise: w')
19     w_plt.plot(w)
20     z_plt = plt.subplot(2, 2, 2, title='Random Walk: z')
21     z_plt.plot(z)
22     x_plt = plt.subplot(2, 2, 3, title='Non Stationary Signal: x')
23     x_plt.plot(x)
24     y_plt = plt.subplot(2, 2, 4, title='Non Stationary Signal: y')
25     y_plt.plot(y)
26     fig.tight_layout()
27     plt.show()
28     # 以作为自变量x
29     w1, x_y_p = self.do_linear_regression(x, y)
30     # 将作为自变量y
31     w2, y_x_p = self.do_linear_regression(y, x)
32     print('xToy={0}({1}) vs yToy={2}({3})'.format(x_y_p, w1[0][0],
33     y_x_p, w2[0][0]))
34     if x_y_p < y_x_p:
35         print('##### x 为自变量')
36         c = w1[0][0] * x - y
37     else:
38         print('##### y 为自变量')
39         c = w2[0][0] * y - x
40     plt.title('Final Cointegration Signal')
41     plt.plot(c)
42     plt.show()

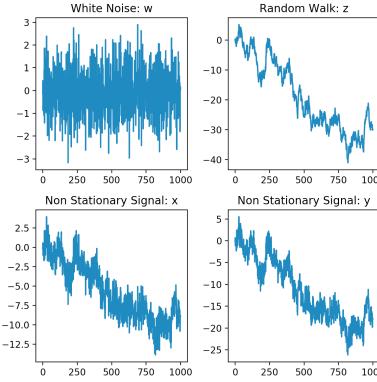
```

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。

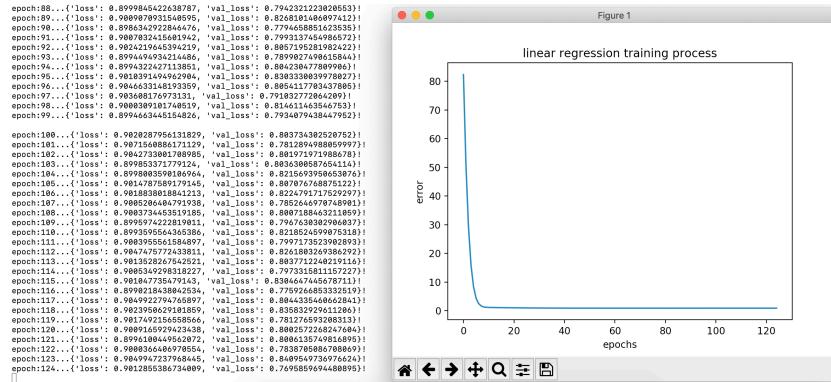
原始信号如下所示：

Figure 47: 原始信号



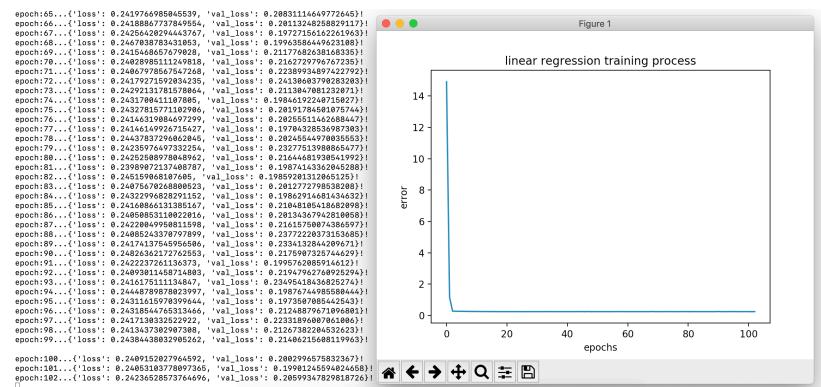
以 x 为自变量时线性回归运行情况:

Figure 48: 以 x 为自变量的线性回归



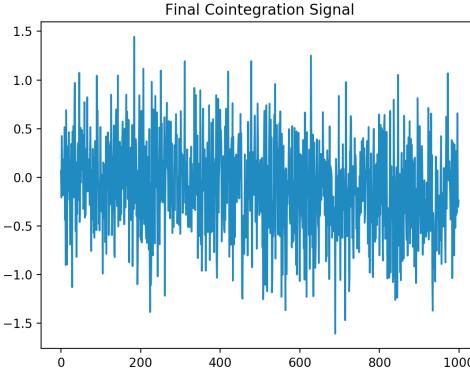
以 y 为自变量时线性回归运行情况:

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]
 [0.10928818583488464]]
weights: [[0.512149453163147]
 [0.10928818583488464]]
stat=-9.9252 vs 1%_cv=-3.4370
resid为稳定时间序列 ^_^
xToy=-4.074526007320772(1.926144003868103) vs yTox=-9.92
##### 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？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：

$$\mathbf{x}_t = \boldsymbol{\mu} + A_1 \mathbf{x}_{t-1} + \dots + A_p \mathbf{x}_{t-p} + \mathbf{w}_t \quad (62)$$

式中 $\boldsymbol{\mu}$ 为序列的均值向量， \mathbf{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)：

$$\Delta \mathbf{x}_t = \boldsymbol{\mu} + A \mathbf{x}_{t-1} + \Gamma_1 \Delta \mathbf{x}_{t-1} + \dots + \Gamma_p \Delta \mathbf{x}_{t-p} + \mathbf{w}_t \quad (63)$$

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

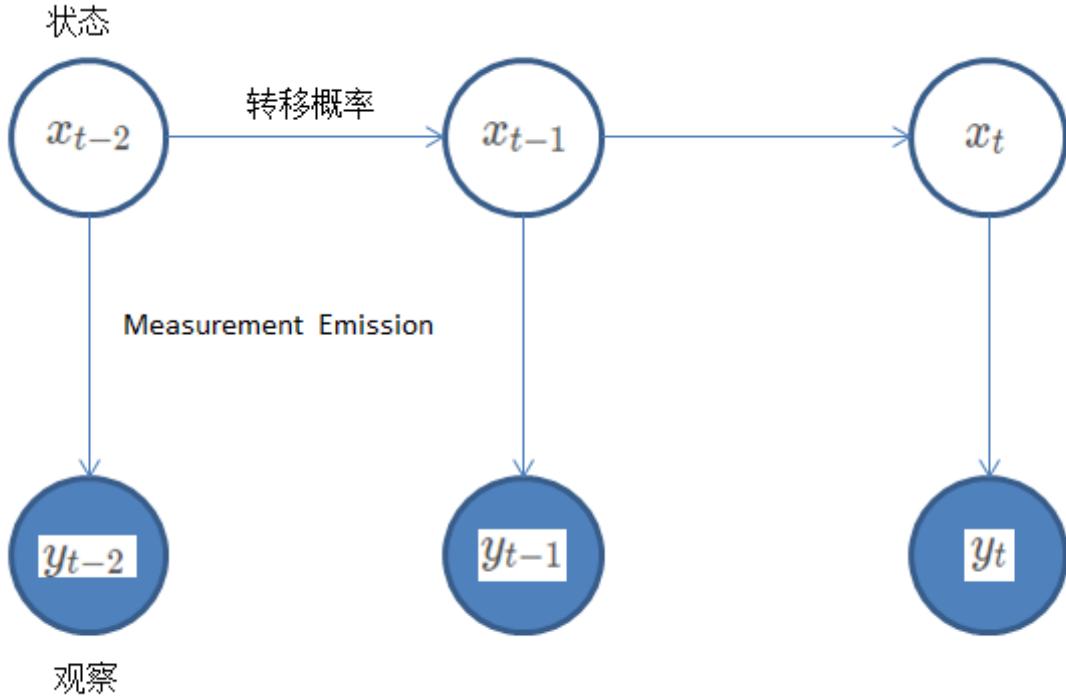
到目前为止，我们所讨论的基于协整模型的交易对策略，有一个重要的假设，即交易对之间的协整系数是不变的。但是在实际应用中，这些协整系数可能会发生缓慢的改变，如果我们用固定值，可能不会取得最佳的应用效果。在本章中，我们将采用 State Space Model 来解决这一问题，具体来讲，就是利用卡尔曼滤波技术，将交易对的对冲比例视为系统不可见的状态，将交易对中标的的收益率作为可观察项，利用卡尔曼滤波器的滤波功能，通过不断增加的新观测数据，利用贝叶斯推理，使我们能够更加精准的估计系统不可见状态，在本例中就是交易对的对冲比例。实际上，交易对的对冲比例，不仅会发生缓慢的变化，有时因为监管、宏观经济、市场事件等原因，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。

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中标的的收益率，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 $\mathbf{x}_t \in R^n$ ，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：

$$\mathbf{x}_t = A\mathbf{x}_{t-1} + \mathbf{b} + \mathbf{u}_t \quad (64)$$

其中 $A \in R^{n \times n}$ 的矩阵， $\mathbf{b} \in R^n$ ， $\mathbf{w}_t \in R^n$ 为高斯白噪声。

系统的观察用 $\mathbf{y}_t \in R^m$ 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：

$$\mathbf{y}_t = H\mathbf{x}_t + \mathbf{c} + \mathbf{v}_r \quad (65)$$

系统初始状态为高斯分布：

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

环境状态在 t 时刻噪声：

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

环境观察在 t 时刻噪声：

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

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？。

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：

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

$$\mathbf{x}_t = A_{t-1}\mathbf{x}_{t-1} + \mathbf{b}_{t-1} + \epsilon_t^1 \quad (70)$$

$$\mathbf{y}_t = C_t\mathbf{x}_t + \mathbf{d}_t + \epsilon_t^2 \quad (71)$$

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

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

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 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：

$$\theta = \{A, \mathbf{b}, C, \mathbf{d}, Q, R, \boldsymbol{\mu}_0, \Sigma_0\} \quad (74)$$

我们的目录是：

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

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。

Table 2: PyKalman 变量意义

表示	模型参数	意义
μ_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
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
d	observation_offsets	观察偏置值
R	observation_covariance	观察协方差矩阵（噪声）

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：

```
1 def hedge_ratio(self):
2     etfs = ['TLT', 'IEI']
3     start_date = "2010-08-01"
4     end_date = "2016-08-01"
5     # 获取调整后的收盘价
6     dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d')
```

```

7     prices = pd.read_csv('./work/aqt005_001.txt', encoding='utf-8',
8     parse_dates=['Date'], date_parser=dateparse, index_col='Date')
9     # 画散点图
10    self.draw_date_coloured_scatterplot(etfs, prices)
11    state_means, state_covs = self.calc_slope_intercept_kalman(etfs,
12     prices)
13    self.draw_slope_intercept_changes(prices, state_means)
14
15    def draw_date_coloured_scatterplot(self, etfs, prices):
16        """
17            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
18            采用黄色代表早期数据而红色代表近期数据
19            @param etfs 标的的字符列表
20            @param prices 价格列表，每行为日期、价格、价格TLTIEI
21        """
22
23        plen = len(prices)
24        colour_map = plt.cm.get_cmap('YlOrRd')
25        colours = np.linspace(0.1, 1, plen)
26        # 生成散点图
27        scatterplot = plt.scatter(
28            prices[etfs[0]], prices[etfs[1]],
29            s=30, c=colours, cmap=colour_map,
30            edgecolor='k', alpha=0.8
31        )
32        # 添加颜色图表
33        colourbar = plt.colorbar(scatterplot)
34        colourbar.ax.set_yticklabels(
35            [str(p.date()) for p in prices[:plen//9].index]
36        )
37        plt.xlabel(prices.columns[0])
38        plt.ylabel(prices.columns[1])
39        plt.show()
40
41    def calc_slope_intercept_kalman(self, etfs, prices):
42        """
43            使用卡尔曼滤波器预测对冲比例
44        """
45        delta = 1e-5
46        mu0 = np.zeros(2)
47        sigma0 = np.ones((2, 2))
48        Q = delta / (1 - delta) * np.eye(2)
49        At = np.eye(2)
50        Ct = np.vstack(
51            [prices[etfs[0]], np.ones(prices[etfs[0]].shape)])
52        Ct.T[:, np.newaxis]
53        R = 1.0
54        kf = KalmanFilter(
55            n_dim_obs=1,
56            n_dim_state=2,
57            initial_state_mean=mu0,

```

```

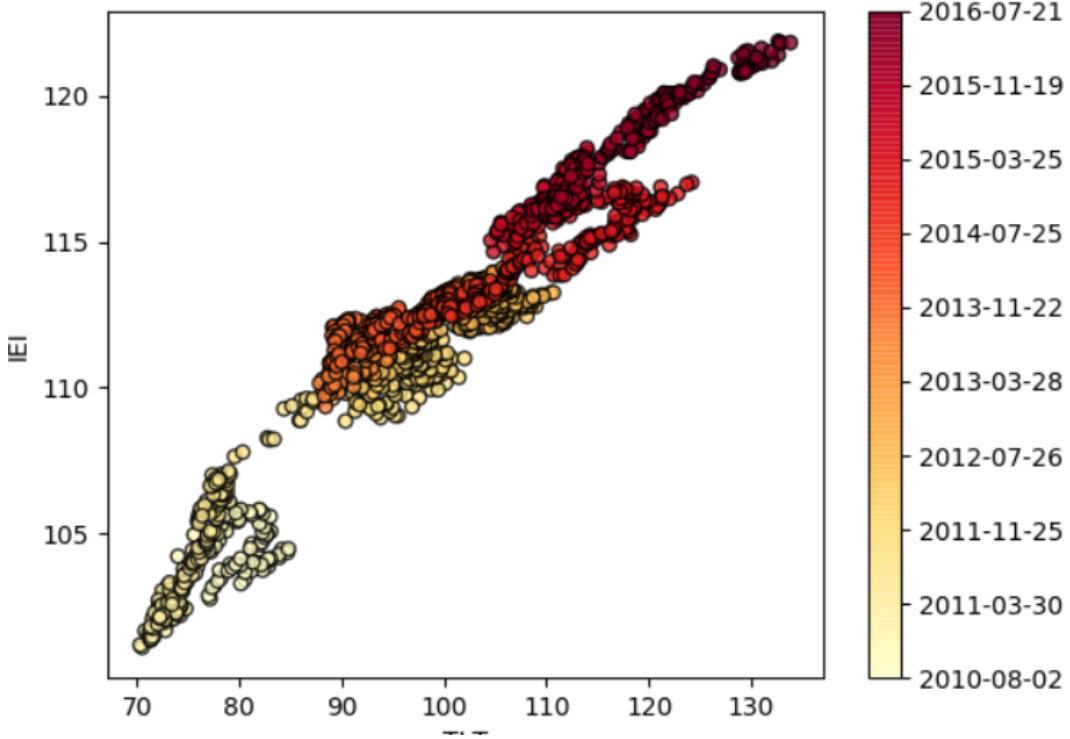
55         initial_state_covariance=sigma0 ,
56         transition_matrices=At,
57         observation_matrices=Ct,
58         observation_covariance=R,
59         transition_covariance=Q
60     )
61
62     yt = prices[etfs[1]].values
63     #state_means , state_covs = kf.em(observations).filter(
64     observations)
65     xt_means , xt_covs = kf.em(yt).filter(yt)
66     return xt_means , xt_covs
67
68 def draw_slope_intercept_changes(self , prices , state_means):
69     """
70     绘制斜率和截距
71     """
72     pd.DataFrame(
73         dict(
74             slope=state_means[:, 0],
75             intercept=state_means[:, 1]
76         ),
77         index=prices.index
78     ).plot(subplots=True)
79     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54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：

$$\mathbf{x}_{t-1} = A_t \mathbf{x}_{t-1} + \mathbf{b}_t + \boldsymbol{\epsilon}_t^1 \quad (76)$$

$$\mathbf{y}_t = C_t \mathbf{x}_t + \mathbf{d}_t + \boldsymbol{\epsilon}_t^2 \quad (77)$$

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

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

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值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，然后利用估计出的参数，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：

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

注意：在上式中，我们将截距项 \mathbf{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

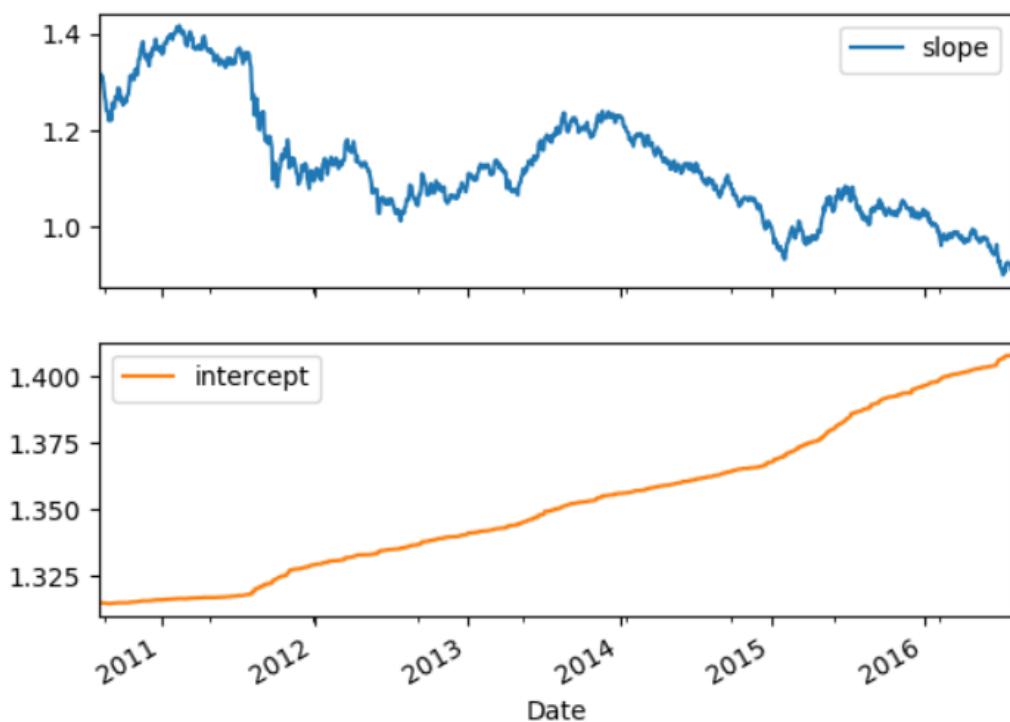


图55中，上面蓝色的曲线为对冲比例变化曲线，可以看出，对冲比例并不是一成不变的，因此如果我们按照动态变化的对冲比例来设计交易策略，我们可以得到更高的收益。

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。

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：

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

Listing 16: 卡尔曼滤波示例

获取股票历史交易记录非常简单，甚至都不用注册用户，例如我们想获取工商银行 2006 到至 2018 年数据，根据 baostock 的规定，工商银行的股票代码为 601398.sh，程序如下所示：

```
1 import baostock as bs
2 import pandas as pd
3
4 class BsCnaDaily(object):
5     def __init__(self):
6         self.name = 'BsADaily'
7
8     def get_history_data(self, stock_code, start_date, end_date):
9         """
10         获取股日线行情历史数据
11         @param stock_code 股票代码，如工商银行为: sh.601398
12         @param start_date 开始日期，格式yyyy-MM-dd
13         @param end_date 结束日期，格式yyyy-MM-dd
14         @return 获取成功返回，否则返回TrueFalse
15         @version v0.0.1 闫涛 2019-04-22
16         """
17         lg = bs.login()
18         if lg.error_code != '0':
19             print('login respond error_msg:' + lg.error_msg)
20             return False, lg.error_msg
21         rs = bs.query_history_k_data_plus(stock_code,
22                                         "date,code,open,high,low,close,preclose,volume,amount",
23                                         adjustflag ,turn ,tradestatus ,pctChg ,isST ,
24                                         start_date=start_date , end_date=end_date ,
```

```

24             frequency="d", adjustflag="3")
25     if rs.error_code != '0':
26         print('query_history_k_data_plus respond error_msg:' + rs.
27             error_msg)
28         return False, rs.error_msg
29     data_list = []
30     while (rs.error_code == '0') & rs.next():
31         data_list.append(rs.get_row_data())
32     result = pd.DataFrame(data_list, columns=rs.fields)
33     result.to_csv('./data/{0}.csv'.format(stock_code), index=False)
34     bs.logout()
35     return True

```

Listing 17: baostock 获取股票历史行情数据

调用方法如下所示:

```

1 import csv
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from core.quotation.bs_cna_daily import BsCnaDaily
5
6 class TpsaDataset(object):
7     def __init__(self):
8         self.name = 'TpsaDataset'
9
10    @staticmethod
11    def get_quotation_data(stocks):
12        """
13            获取交易对行情数据
14            @params stocks np.array 和分别代表交易对 [0][...][1][...]
15            对于交易对: [0]{ stock_code , start_date , end_date , etf_name }
16            其中也是数据文件保存的名字, 共有两条记录, 由etf_name stocks
17            调用者负责保证
18        """
19        print(stocks[0]['stock_code'])
20        bs_cna_daily = BsCnaDaily()
21        # 工商银行(股票代码开头的是上海) 60
22        bs_cna_daily.get_history_data(stocks[0]['stock_code'],
23                                      stocks[0]['start_date'], stocks[0]['end_date'],
24                                      stocks[0]['etf_name'])
25        bs_cna_daily.get_history_data(stocks[1]['stock_code'],
26                                      stocks[1]['start_date'], stocks[1]['end_date'],
27                                      stocks[1]['etf_name'])
28
29    @staticmethod
30    def draw_close_price_curve(stock_files):
31        """
32            绘制股票收益率曲线
33            @stock_files 交易对股票行情文件名, 共有两个
34        """

```

```

35     print('绘制收盘价曲线... ')
36     etf0_prices = TpsaDataset.read_close_prices(stock_files[0])
37     etf1_prices = TpsaDataset.read_close_prices(stock_files[1])
38     plt.title('close price curve')
39     plt.plot(etf0_prices)
40     plt.plot(etf1_prices)
41     plt.show()
42
43     @staticmethod
44     def read_close_prices(stock_file):
45         """
46             读取股票收盘价数据时间序列
47             @param stock_file 股票行情文件
48             @return 股票收盘价的list
49         """
50         close_prices = []
51         with open(stock_file, 'r', newline='') as fd:
52             rows = csv.reader(fd)
53             header = next(rows)
54             print('header:{0}'.format(header))
55             for row in rows:
56                 close_prices.append(float(row[4]))
57         return close_prices
58
59     @staticmethod
60     def read_close_price_pd(stock_file):
61         dateparse = lambda x: pd.datetime.strptime(x, '%Y-%m-%d')
62         prices = pd.read_csv(stock_file, encoding='utf-8',
63                             parse_dates=['Date'], date_parser=dateparse,
64                             index_col='Date')
65         print(prices.values[:, 3])

```

Listing 18: baostock 获取股票历史行情数据示例

6.2 卡尔曼滤波策略

在本例中，我们将工商银行收盘价作为自变量，将建设银行收盘价作为因变量。按照卡尔曼滤波框架，我们将建设银行的收盘价视为可观测状态 y_t ，而建设银行收盘价与工商银行收盘价之比，包括斜率和截距，视为系统验证状态，将工商银行收盘价视为观测矩阵，这样就组成了一个卡尔曼滤波模型。系统在刚开始运行时，我们拿一部分历史数据运行卡尔曼滤波器，通过 EM 算法，估计出卡尔曼滤波器中的参数，之后再根据历史数据运行回测流程。

我们首先运行 `kalman.filter` 方法，求出系统的状态，然后根据工商银行收盘价和 `filter` 方法返回的系统状态中的斜率和截距，求出建设银行的计算值，将建设银行计算值与建设银行收盘价真实值相减得天误差，然后将其与观察向量计算中的误差的标准差相比较，如果小于负的标准差，则买入建设银行股票，并按斜率比例卖出对应数量的工商银行股份；如果大于标准差，则卖出建设银行股票，并按斜率比例买入对应数量的工商银行股票。具体代码如下所示：

```

1 from __future__ import print_function

```

```

2
3 from math import floor
4 import math
5
6 import numpy as np
7
8 from qstrader.price_parser import PriceParser
9 from qstrader.event import (SignalEvent, EventType)
10 from qstrader.strategy.base import AbstractStrategy
11 from pykalman import KalmanFilter
12
13
14 class TpsaStrategy(AbstractStrategy):
15     """
16     Requires:
17     tickers - The list of ticker symbols
18     events_queue - A handle to the system events queue
19     """
20
21     def __init__(self, tickers, events_queue, equity, ts0, ts1):
22         self.tickers = tickers
23         self.events_queue = events_queue
24         self.equity = equity
25         self.time = None
26         self.latest_prices = np.array([-1.0, -1.0])
27         self.invested = None
28
29         self.delta = 1e-4
30         self.wt = self.delta / (1 - self.delta) * np.eye(2)
31         self.vt = 1e-3
32         self.theta = np.zeros(2)
33         self.P = np.zeros((2, 2))
34         self.R = None
35
36         self.days = 0
37         self.qty = 20000
38         self.cur_hedge_qty = self.qty
39
40         self.yt_state = 0
41         self.buy_price = 0.0
42
43         self.ts0 = ts0
44         self.ts1 = ts1
45         self.kalman_mode = 1
46
47         xt_means, xt_covs = self.train_kalman_filter(ts0, ts1)
48         slope = xt_means[-1][0]
49         # 将资金按80%1:比例购买和slopets1ts0
50         price0 = ts0[-1]

```

```

52     price1 = ts1[-1]
53     print('type:{0}; shape:{1}'.format(type(price0), ts0[-1].shape))
54     amount = self.equity * 0.1
55     amount1 = amount / (1 + slope)
56     amount0 = amount * (slope / (1+slope))
57     self.qty1 = int(math.floor(amount1 / price1))
58     self.qty1_0 = self.qty1
59     self.qty0 = int(math.floor(amount0 / price0))
60     self.qty0_0 = self.qty0
61     amt1 = self.qty1 * price1
62     amt0 = self.qty0 * price0
63     self.equity -= (amt0 + amt1)
64     self.equity_0 = self.equity
65     self.events_queue.put(SignalEvent(self.tickers[0], "BOT", self.
66         qty0))
66     print('购买: 数量: ; 价格:
{0}{1}{2}'.format(self.tickers[0], self.qty0, self.ts0[-1]))
67     self.events_queue.put(SignalEvent(self.tickers[1], "BOT", self.
68         qty1))
68     print('购买: 数量: ; 价格:
{0}{1}{2}'.format(self.tickers[1], self.qty1, self.ts1[-1]))
69     print('现金: {0}'.format(self.equity))

70
71     self.ts0 = np.array([])
72     self.ts1 = np.array([])
73     self.deltas = np.array([])

74
75
76
77     def _set_correct_time_and_price(self, event):
78         """
79             Sets the correct price and event time for prices
80             that arrive out of order in the events queue.
81         """
82
83         # Set the first instance of time
84         if self.time is None:
85             self.time = event.time
86
87         # Set the correct latest prices depending upon
88         # order of arrival of market bar event
89         price = event.adj_close_price/float(
90             PriceParser.PRICE_MULTIPLIER
91         )
92         if event.time == self.time:
93             if event.ticker == self.tickers[0]:
94                 self.latest_prices[0] = price
95             else:
96                 self.latest_prices[1] = price
97         else:
98             self.time = event.time

```

```

98         self.days += 1
99         self.latest_prices = np.array([-1.0, -1.0])
100        if event.ticker == self.tickers[0]:
101            self.latest_prices[0] = price
102        else:
103            self.latest_prices[1] = price
104
105    def train_kalman_filter(self, ts0, ts1, mode=1):
106        """
107            Utilise the Kalman Filter from the PyKalman package
108            to calculate the slope and intercept of the regressed
109            ETF prices.
110        """
111        delta = 1e-5
112        mu0 = np.zeros(2)
113        sigma0 = np.ones((2, 2))
114        Q = delta / (1 - delta) * np.eye(2)
115        At = np.eye(2)
116        Ct = np.vstack(
117            [ts0, np.ones(ts0.shape)])
118        ).T[:, np.newaxis]
119        R = 1.0
120        self.kf = KalmanFilter(
121            n_dim_obs=1,
122            n_dim_state=2,
123            initial_state_mean=mu0,
124            initial_state_covariance=sigma0,
125            transition_matrices=At,
126            observation_matrices=Ct,
127            observation_covariance=R,
128            transition_covariance=Q
129        )
130        yt = ts1
131        #state_means, state_covs = kf.em(observations).filter(
132        observations)
133        if mode != 1:
134            xt_means, xt_covs = self.kf.filter(yt)
135        else:
136            xt_means, xt_covs = self.kf.em(yt).filter(yt)
137        return xt_means, xt_covs
138
139    def calculate_signals(self, event):
140        mode = 2
141        if event.type == EventType.BAR:
142            self._set_correct_time_and_price(event)
143            # Only trade if we have both observations
144            if all(self.latest_prices > -1.0):
145                # kalman.filter_update
146                self.ts0 = np.append(self.ts0, self.latest_prices[0])

```

```

147         self.ts1 = np.append(self.ts1, self.latest_prices[1])
148         if self.days < 100:
149             return
150         if self.days % 30 == 0:
151             mode = 1
152             xt_means, x_convs = self.train_kalman_filter(self.ts0,
153             self.ts1, mode=2)
153             slope = xt_means[-1][0]
154             intercept = xt_means[-1][1]
155             yt_hat = self.ts0[-1] * slope + intercept
156             delta = yt_hat - self.ts1[-1]
157             self.deltas = np.append(self.deltas, delta)
158             threshold = np.std(self.deltas)
159             if delta < -threshold:
160                 # 卖掉买入01
161                 qty0 = int(math.floor(self.qty * slope))
162                 self.events_queue.put(SignalEvent(self.tickers[0], "SLD", qty0))
163                 self.qty0 -= qty0
164                 self.equity += qty0 * self.latest_prices[0]
165                 self.events_queue.put(SignalEvent(self.tickers[1], "BOT", self.qty))
166                     self.qty1 += self.qty
167                     self.equity -= self.qty * self.latest_prices[1]
168                     print('    买入: 数量: ; 价格: ; 金额: '
169                         .format(
170                             self.tickers[1], self.qty, self.latest_prices
171                             [1],
172                             self.qty * self.latest_prices[1]))
173                     print('    卖出: 数量: ; 价格: ; 金额: '
174                         .format(
175                             self.tickers[0], qty0, self.latest_prices[0],
176                             qty0 * self.latest_prices[0]))
177                     total = self.equity + self.qty0 * self.latest_prices
178                     [0] + self.qty1 * self.latest_prices[1]
179                     print('#####总资产: '
180                         .format(total, self.equity,
181                             self.qty0 * self.latest_prices[0], self.qty1
182                             * self.latest_prices[1]))
183                     elif delta > threshold:
184                         # 买入卖出01
185                         self.events_queue.put(SignalEvent(self.tickers[1], "SLD", self.qty))
186                         self.qty1 -= self.qty
187                         self.equity += self.qty * self.latest_prices[1]
188                         qty0 = int(math.floor(self.qty * slope))
189                         self.events_queue.put(SignalEvent(self.tickers[0], "BOT", qty0))

```

```

186             self.qty0 += qty0
187             self.equity -= qty0 * self.latest_prices[0]
188             print('    买入: 数量: ; 价格: ; 金额: '
189                 {0}{1}{2}{3}'.format(
190                     self.tickers[0], qty0, self.latest_prices[0],
191                     self.qty * self.latest_prices[0])
192             )
193             print('    卖出: 数量: ; 价格: ; 金额: '
194                 {0}{1}{2}{3}'.format(
195                     self.tickers[1], self.qty, self.latest_prices
196                     [1],
197                     qty0 * self.latest_prices[1]
198                     ))
199             total = self.equity + self.qty0 * self.latest_prices
200             [0] + self.qty1 * self.latest_prices[1]
201             print('#####总资产: '
202                 {0}={1}+{2}+{3}'.format(total, self.equity,
203                     self.qty0 * self.latest_prices[0], self.qty1
204                     * self.latest_prices[1]))

```

Listing 19: 卡尔曼滤波协整模型交易对策略

另外，在程序中需要注意的部分为我们每隔一定的交易日，会重新使用 EM 算法来估计卡尔曼滤波器的参数，这样可以使我们的估计更加准确。

6.3 卡尔曼滤波引擎

我们在策略引擎中，初始化策略，调用平台进行回测，统计回测效果，代码如下所示：

```

1 import calendar
2 import datetime
3 import numpy as np
4 from qstrader import settings
5 from qstrader.strategy.base import AbstractStrategy
6 from qstrader.positionSizer.naive import NaivePositionSizer
7 from qstrader.event import SignalEvent, EventType
8 from qstrader.compat import queue
9 from qstrader.tradingSession import TradingSession
10 from qstrader.priceHandler.bscna_daily_csv_bar import
11     BscnaDailyCsvBarPriceHandler
12
13 import matplotlib.pyplot as plt
14 from app.tpsa.tpsa_dataset import TpsaDataset
15
16
17 from app.tpsa.tpsa_strategy import TpsaStrategy
18
19 class TpsaEngine(object):
20     def __init__(self):
21         self.name = 'QhEngine'

```

```

22
23     def startup(self):
24         testing = False
25         config = settings.load_config()
26         tickers = ['ICBC', 'CBC']
27         # 读取用于估计卡尔曼滤波参数的时间序列
28         ts0 = np.array(TpsaDataset.read_close_prices('./data/{0}_train.csv'.format(tickers[0])))
29         ts1 = np.array(TpsaDataset.read_close_prices('./data/{0}_train.csv'.format(tickers[1])))
30
31         self.title = [
32             '基于卡尔曼滤波器的交易对策略',
33         ]
34         self.initial_equity = 1000000.0
35         self.start_date = datetime.datetime(2017, 1, 1)
36         self.end_date = datetime.datetime(2019, 4, 23)
37         # Use the Monthly Liquidate And Rebalance strategy
38         self.events_queue = queue.Queue()
39         self.strategy = TpsaStrategy(
40             tickers, self.events_queue, self.initial_equity, ts0, ts1
41         )
42         self.run(config, testing, tickers)
43
44     def run(self, config, testing, tickers):
45         # Use the Naive PositionSizer where
46         # suggested quantities are followed
47         positionSizer = NaivePositionSizer()
48         # Set up the backtest
49         backtest = TradingSession(
50             config, self.strategy, tickers,
51             self.initial_equity, self.start_date, self.end_date,
52             self.events_queue, title=self.title,
53             positionSizer=positionSizer,
54             priceHandler=BscnaDailyCsvBarPriceHandler(
55                 config.CSV_DATA_DIR, self.events_queue,
56                 tickers, start_date=self.start_date,
57                 end_date=self.end_date
58             )
59         )
60         results = backtest.start_trading(testing=testing)
61         print('最后金额: {0}'.format(self.strategy.equity))
62
63         total = self.strategy.equity + self.strategy.qty0 * \
64             self.strategy.latest_prices[0] + \
65             self.strategy.qty1 * self.strategy.latest_prices[1]
66         print('##### 总资产: {0}={1}+{2}+{3}'.format(
67             total, self.strategy.equity,
68             self.strategy.qty0 * self.strategy.latest_prices[0],
69             self.strategy.qty1 * self.strategy.latest_prices[1])

```

```

70         ))
71     delta0 = self.strategy.qty0 - self.strategy.qty0_0
72     amt0 = 0.0
73     if delta0 > 0:
74         amt0 = self.strategy.latest_prices[0] * delta0
75     else:
76         amt0 = -self.strategy.latest_prices[0] * delta0
77     delta1 = self.strategy.qty1 - self.strategy.qty1_0
78     amt1 = 0.0
79     if delta1 > 0:
80         amt1 = self.strategy.latest_prices[1] * delta1
81     else:
82         amt1 = -self.strategy.latest_prices[1] * delta1
83
84     print('initial:{0} vs final {1}'.format(self.strategy.equity_0,
85                                              self.strategy.equity+amt0+amt1))
86
87
88
89     def prepare_data(self):
90         stocks = [
91             {
92                 'stock_code': 'sh.601398',
93                 'start_date': '2019-03-01',
94                 'end_date': '2019-04-29',
95                 'etf_name': 'ICBC'
96             },
97             {
98                 'stock_code': 'sh.601939',
99                 'start_date': '2019-03-01',
100                'end_date': '2019-04-29',
101                'etf_name': 'CBC'
102            }
103        ]
104        TpsaDataset.get_quotation_data(stocks)
105        stock_files = [
106            './data/{0}.csv'.format(stocks[0]['etf_name']),
107            './data/{0}.csv'.format(stocks[1]['etf_name'])
108        ]
109        TpsaDataset.draw_close_price_curve(stock_files)

```

Listing 20: 卡尔曼滤波策略引擎

6.4 运行结果

通过如下代码可以运行这个策略的回测:

```

1     tpsaEngine = TpsaEngine()
2     tpsaEngine.startup()

```

Listing 21: 运行卡尔曼滤波策略引擎

运行程序后，得到累积收益率曲线如下所示：

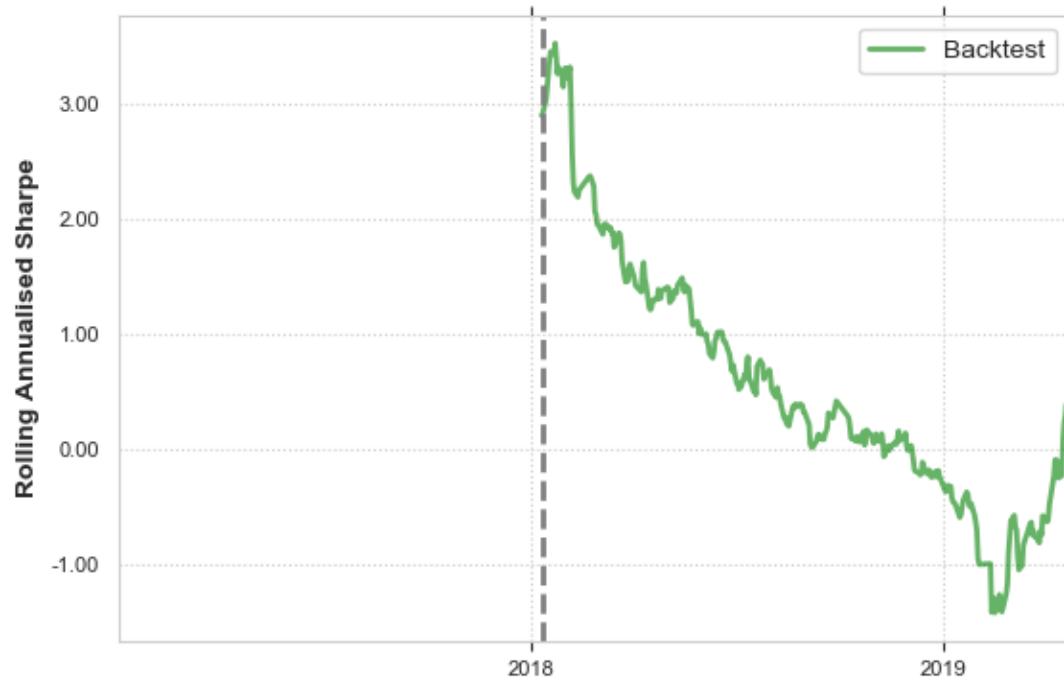
Figure 56: 累积收益率曲线



在图形中有一段收益率非常高，但是在那个时间段，由于我们采用的是统计套利策略，并不会处理暴涨暴跌的情况，因此这段行情并没有把握住。关于这一点我们将在下一节进行讨论。

年化夏普比曲线：

Figure 57: 年化夏普比曲线



策略最大回撤图如下所示：

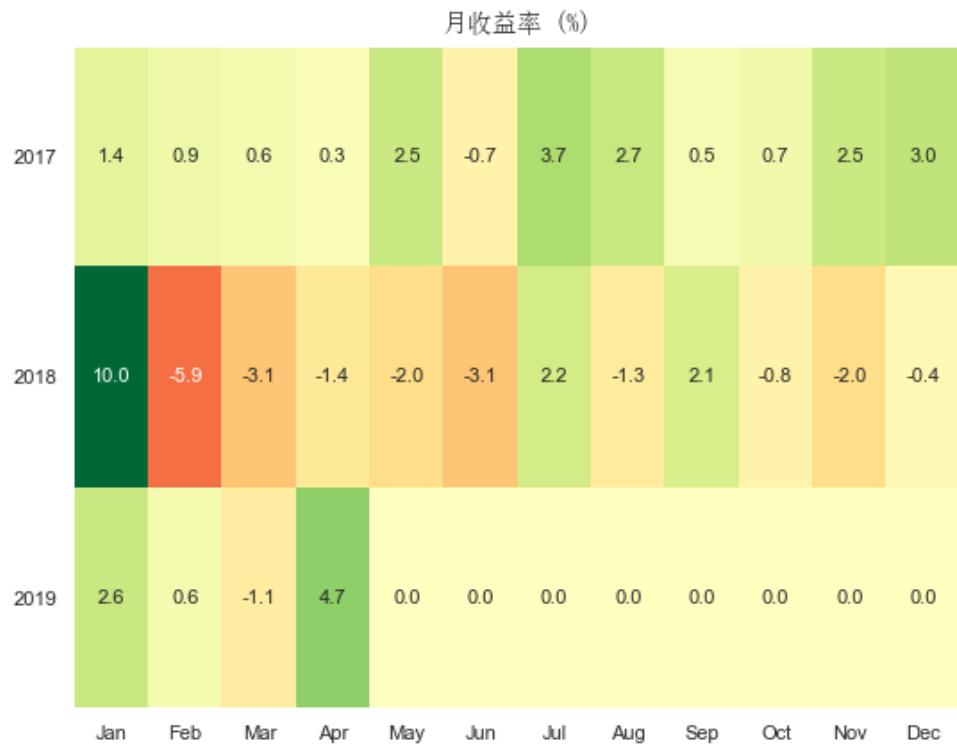
Figure 58: 策略最大回撤



由于我们没在最高点时抛出股票，因此最大回撤图形上显得有些差，但是最后效果上面来看，应该还能说得过去。

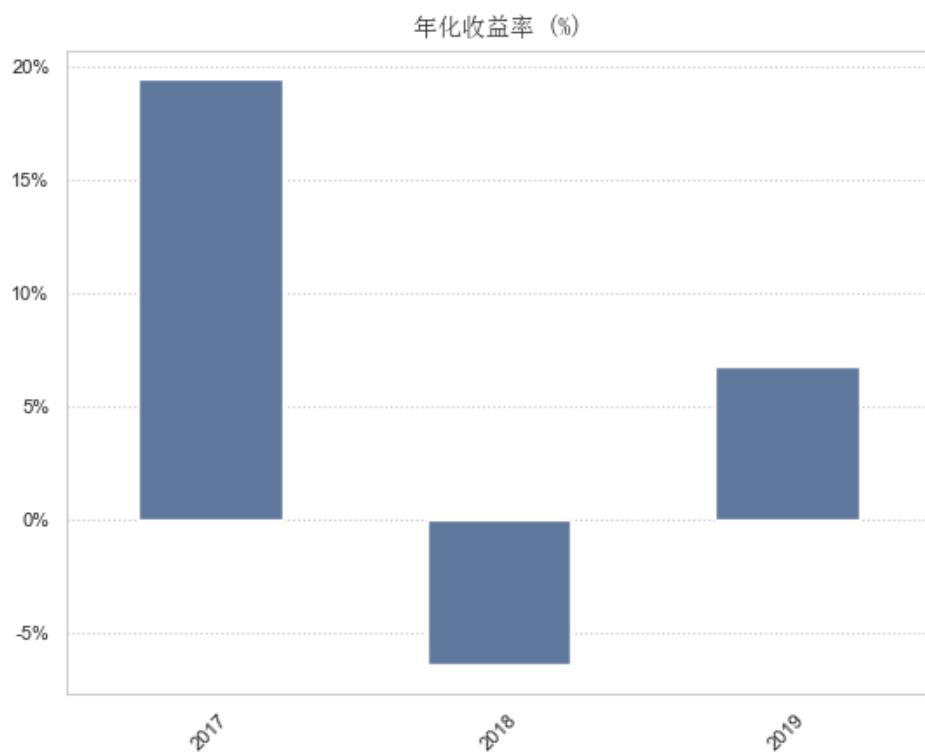
月收益率曲线：

Figure 59: 月收益率曲线



年化收益率：

Figure 60: 年化收益率曲线



总体统计信息：

Figure 61: 总体统计信息

Curve	
总收益	19%
CAGR	8.27%
Sharpe Ratio	0.90
Sortino Ratio	1.29
年化波动	9.32%
R-Squared	0.31
最大日回撤	17.17%
最大回撤期间	291
年交易	0.0

由上图可以看出，夏普比为 0.88，通常较好的交易策略夏普比应该在 1 以上，所以本策略还是一个特别好的策略。策略最终结果如下所示：

Figure 62: 最终结果

```
#####总资产: 1599756.7318739002=869388.4118739001+67068.32+66330
initial:1000000.0 vs final 1388620.0918739
```

在这个策略中，我们假设我们会长期持有 50000 股工商银行和 50000 股建设银行，每次波动对冲时交易 20000 股，初始时我们流动资金是 100 万，不采用资金杠杆，在回测结束时，我们将工商银行和建设银行恢复各 50000 股，然后计算流动资金。由结果可以看出，我们在持股情况未变化的情况下，流动资金变为 138.86 万，实现了 38.86 万的盈利。这证明我们的策略还是基本可行的。策略最终夏普比为 0.9，也处于较为理想的数值区间。

6.5 讨论

虽然我们基于卡尔曼滤波的协整模型，实际表现还算中规中矩，是一个可接受的交易策略，但是毕竟我们希望得到更高的收益率，在这一节中我们就来分析一下，为什么我们的交易策略收益率不是特别高呢？

首先，我们没有采用资金杠杆，因为我们知道，交易对之间的相对波动较小，交易所还会收取交易费和印花税等费用，因此即使判断正确，每一单的盈利也是非常小的。如果我们可以使用资金杠杆，通常统计套利策略资金杠杆可以达到 3~5 倍，这样我们的收益率就会提高到 100% 以上（扣除杠杆利息），这就是一个相当不错的交易策略了。例如，LTCM 专门做德国国债和意大利国债交易对的基金，其资金杠杆高达数万倍。当然，使用资金杠杆，

在放大收益率的同时，也会放大风险，如果判断失误，损失将非常惨重。事实上，红极一时的 LTCM 基金，就是因为一个决策失误，导致资金链断裂，最终破产的。

其次，基于协整模型的交易对策略，虽然是一种市场中性策略，但是在市场剧烈波动时，由于信号具有非平稳性，在此期间应该停止交易，这通常通过交易平台的风控模块来实现，但是我们这个策略中没有风控模块。下图是工商银行和建设银行收盘价走势图：

Figure 63: 工商银行（蓝）和建设银行（红）收盘价曲线



由图63可以看出，二者收盘价走势非常具有一致性，因此作为协整模型的交易对是没有问题的。但是大家请注意，在我们进行回测中间，二者的股价都有一次暴涨和回调，在这种行情下，显然应该采取以跟踪短期趋势为主的程序化交易 CPA 策略，而本章中的策略，基本没有对这一过程进行特别处理，这显然是需要优化的。

第 7 章 隐马可夫模型

Abstract

在本章中我们讲述 State Space Model 中的隐马可夫模型，并且将该模型用于识别市场所处状态。在下一章中，将以此技术为基础，开发程序化 CPA 策略。

7 隐马可夫模型

我们所要研究的金融市场，经常处于变化之中，尤其是受政府政策、宏观调控、业务管制和突发事件等影响，市场状态会发生突然地剧烈的变化，如果我们的策略不作调整，就会使我们遭受巨大的损失。因此，对这种市场机制（Market Regime）进行研究，尤其是识别和预测市场机制的转变，是成功的量化交易所必需的。

我们前几章中对时序信号进行研究过程中，在介绍这些方法时，都在强调这些方法研究的对象是具有平稳性的时序信号。但是当市场机制发生变化时，会使时序信号的均值、方差、自相关性和协方差发生变化，会产生相关性变化、异方差、肥尾等现象，因此研究和预测市场机制变化规律对量化交易而言非常重要。

我们将采用在统计时间序列研究中最重要的隐马可夫模型，来研究市场机制变化规律。这种理论认为，市场机制及其变化，是由内部不可见的过程决定的，其会决定当前的市场机制以及之后市场机制的转换，我们只能通过投资标的的收益率，来间接地观测和研究这一过程。

7.1 马可夫模型

马可夫模型是一种随机的状态空间模型，系统会随机的从一种状态切换到另一种状态，状态之间的转换概率只与前一时刻的状态有关，而与之前的状态无关，我们称之为马可夫性质。例如，我们在前面章节中研究的随机游走时间序列，就是一种典型的具有马可夫性质的过程。

根据系统的自治性和系统状态是否完全可见，可以将马可夫模型分为四大类，如下所示：

Table 3: 马可夫模型分类

	完全可见	部分可见
自治	马可夫链	隐马可夫模型 (HMM)
控制	马可夫决策过程 (MDP)	部分可见马可夫决策过程 (POMDP)

在这里我们首先明确一个概念，就是系统的状态和观测。系统状态是指系统内部的一种状态，我们不一定能完全观测到。系统观测是我们能够看到的系统性质，是系统状态的间接反映，但是不一定与系统状态相同。这样说比较抽象，让我们来看一个例子，如下图所示：

Figure 64: 系统状态



系统状态就是在远处有一只老虎。但是在有些情况下，我们是无法观测到这个系统状态的，例如下图：

Figure 65: 系统状态和系统观测



在上图中，有一辆汽车挡在了老虎的前面，这时我们的观测是系统中有一辆汽车，而我们就无从知道汽车后面还有一只老虎了，因此我们就不能得到系统状态了。在实际应用中，这种状态比较常见，例如在机器人应用中，机器人就观测不到被障碍物遮挡的物体和自己背后的物体，同样在我们的金融应用中，我们也观测不到决定市场机制变化的内部过程。

当系统是自治的，即我们不能改变系统状态，而且系统状态是完全可见的情形，我们称之为马可夫链（MC: Markov Chain）。大家所熟知的马可夫链蒙特卡洛算法，就是在贝叶斯推理计算中一种常用的算法。

当系统还是完体自治的，即我们无法改变系统状态，但是我们只能观测到部分的系统状态，这就是本章将要研究隐马可夫模型。在这种模型中，我们无法直接观测到系统的状态和状态间的转换，我们只能观测系统的某些外在表现，系统内部状态影响系统的外在表现。系统状态具有马可夫属性，即状态间转换只取决于前一状态，与历史状态无关。但是我们对系统的观测，不一定要具有马可夫性质。隐马可夫模型，除了在量化交易领域得到广泛应

用外，在语音识别领域也得到了广泛的应用

如果我们即 Agent 可以通过行动 Action 来影响系统状态，同时我们可以完全观测到系统的状态，这就是马可夫决策过程 (MDP)，又称为强化学习 (RL: Reinforcement Learning)。这是当前人工智能三大方向之一，前两年大火的 Alpha Go/Alpha Zero 等，就是基于这种技术。我们会在最后一章中，详细讲解深度强化学习 (DRL: Deep Reinforcement Learning) 技术在量化交易中的应用。

如果 Agent 可以通过行动 Action 来影响系统状态，但是只能间接观测到系统状态的外在表现，这就是部分可见马可夫决策过程 (POMDP)。这部分是当前最前沿的研究方向，目前还没有特别成熟有效的解决方案。

7.2 马可夫模型数学原理

我们假设有一系列的观察值: $\{X_1, X_2, \dots, X_t, \dots, X_T\}$ ，我们假设我们可以完全观测到系统的状态，因此 X_t 同时也是系统状态。同时，我们假设在任意时刻，我们的观测值 X_t 包含所有的信息，使我们能够预测出未来系统的状态。这一假设可以表示为如下的概率模型:

$$p(X_{1:T}) = p(X_1)p(X_2|X_1)\dots\dots p(X_T|X_{T-1}) = p(X_1) \prod_{t=2}^T p(X_t|X_{t-1}) \quad (81)$$

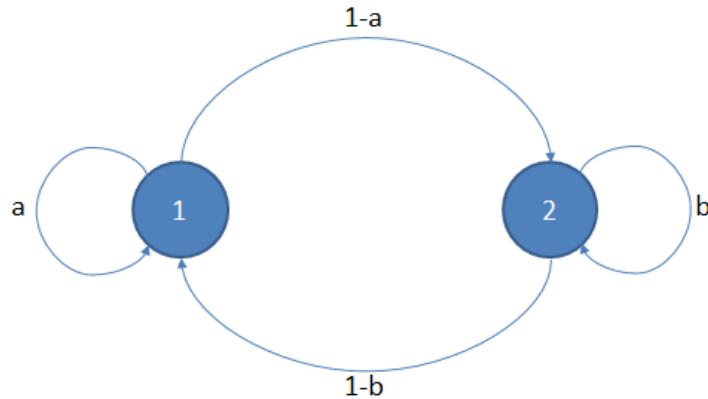
我们将 $p(X_t|X_{t-1})$ 是 $t - 1$ 到 t 时刻的转移函数，同时转移概率与时间无关。

我们假设有 K 个不同的状态，在 $t - 1$ 时刻处于状态 i ，在 t 时刻处于状态 j ，其转移概率为:

$$A_{i,j} = p(X_t = j|X_{t-1} = i) \quad (82)$$

为简化问题，我们假设系统仅有两个状态，分别为 1 和 2，在每个时刻，系统都处于这两个状态中的一个，系统从一个状态转移到另一个状态服从概率分布，并且与时间无关，我们可以通过下图来表示:

Figure 66: 状态转移概率图



如图66所示，当前时刻系统处在状态 1，那么下一时刻还是状态 1 的概率为 a ，下一时刻为状态 2 的概率为 $1-a$ ；如果当前时刻系统处在状态 2，那么下一时刻还在状态 2 的概率为 b ，转移到状态 1 的概率为 $1-b$ ，通常我们可以用状态转移矩阵来表示:

$$A = \begin{bmatrix} 1 - \alpha & \alpha \\ \beta & 1 - \beta \end{bmatrix} \quad (83)$$

7.3 隐马可夫模型数学原理

如果我们不能完全观测到系统的状态，这时我们研究的问题就是隐马可夫模型。以我们本章要研究的问题为例，系统目前的市场机制对我们来说是不可见的，我们只能看到收益率，我们的任务就是根据收益率的变化情况，来推测系统所处的市场机制。

我们假设系统共有 K 个状态，例如本章所研究的问题，我们可以认为 $K=3$ ，分别对应为震荡、上涨、下跌，则系统状态用 $z_k, k \in \{1, \dots, K\}$ 来表示，系统观测用 x_t 来表示，系统状态可以决定系统观测，我们用 $p(x_t|z_t)$ 来表示。在隐马可夫模型中，系统处于某一个特定的状态，然后会突然转换到另一个新的状态，这与金融市场通常处于一种行情，突然发生一个突发事件，系统就从一种行情转换到另一种行情，是非常相似的，因此我们用隐马可夫模型来研究这一问题是非常适合的。从 $t = 1$ 到 $t = T$ 时刻，系统状态出现的概率为：

$$p(z_{1:T}) = p(z_1) \prod_{t=2}^T p(z_t|z_{t-1}) \quad (84)$$

系统观测出来的概率为：

$$p(x_{1:T}|z_{1:T}) = \prod_{t=1}^T p(x_t|z_t) \quad (85)$$

则根据观测值推断系统所处状态的概率表示为：

$$p(z_{1:T}|x_{1:T}) = p(z_{1:T})p(x_{1:T}|z_{1:T}) = \left[p(z_1) \prod_{t=2}^T p(z_t|z_{t-1}) \right] \left[\prod_{t=1}^T p(x_t|z_t) \right] \quad (86)$$

如果将这个模型应用于我们本章研究的问题，细心的读者可能就会有疑问，在上面的隐马可夫模型中，无论观测值还是系统状态，都是离散变量，而我们的所研究的问题中，股票的收益率为连续值，式86中的 $p(x_t|z_t)$ 怎么计算呢？在实际应用中，我们假设在 t 时刻系统所处状态为 k ，即 $z_t = k$ ，此时我们可以假设 x_t 服从如下所示的高斯分布：

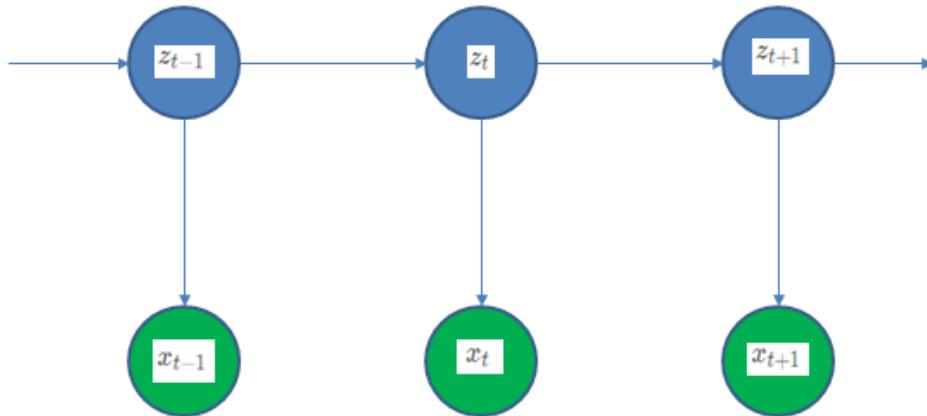
$$x_t \sim \mathcal{N}(\mu_k, \sigma_k^2) \quad (87)$$

则有下式成立：

$$p(x_t|z_t = k; \theta) = \mathcal{N}(x_t|\mu_k, \sigma_k^2) \quad (88)$$

隐马可夫过程可以表示为如下所示形式：

Figure 67: 隐马可夫模型示意图



隐马可夫模型可以实现三大任务：

- 预测：预测接下来的状态值；
- 滤波：从观测序列预测当前状态值；
- 平滑：从观测序列解释过去的状态值；

我们本章研究的任务是通过股票收益率变化曲线，预测市场所处状态，因此属于典型的滤波应用。

在介绍了隐马可夫模型的基本理论之后，我们将以工商银行股票为例，讲解基于隐马可夫过程的程序化交易 CPA 策略。

第二篇深度学习算法

第三篇深度元强化学习

第 301 章 MAML 算法

Abstract

在本章中，我们将讲解 MAML 算法的基本原理，并且以 Omniglot 数据集为例，讲解一个 5-way 1-shot 的算法实现，并且复现论文中的结果。

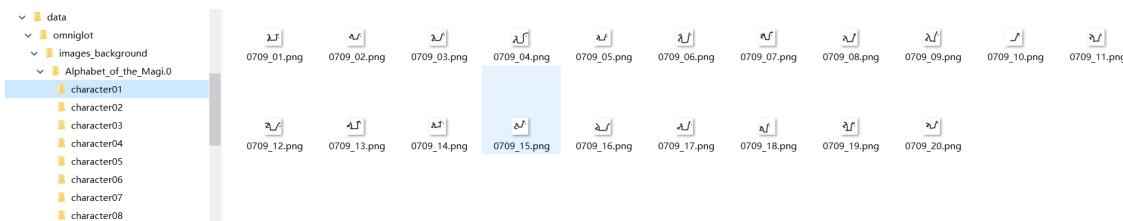
8 MAML 算法

在元学习中，最著名的算法当属 MAML 算法，在本章中，我们将讲解 MAML 算法数学原理和 PyTorch 实现技术。

8.1 Omniglot 数据集

Omniglot 数据集的目录结构如下所示：

Figure 68: Omniglot 数据集目录结构



数据集类代码如下所示：

```
1 class OmniglotDs(Dataset):
2     def __init__(self, data_dir, k_way, q_query):
3         self.file_list = [f for f in glob.glob(data_dir +
4             '**/*', recursive=True)]
5         self.transform = transforms.Compose([transforms.ToTensor()])
6         self.n = k_way + q_query
7
8     def __getitem__(self, idx):
9         sample = np.arange(20)
10        np.random.shuffle(sample)
11        img_path = self.file_list[idx]
12        img_list = [f for f in glob.glob(img_path + "*/*.png",
13            recursive=True)]
14        img_list.sort()
15        imgs = [self.transform(Image.open(img_file))
16                for img_file in img_list]
17        imgs = torch.stack(imgs)[sample[:self.n]]
18        return imgs
19
20    def __len__(self):
21        return len(self.file_list)
```

Listing 22: Omniglot 数据集类

代码解读如下所示：

- 第 3、4 行：取出图片文件最后一层文件目录的列表，如下所示：

```
1 ./data/Omniglot/images_background\Alphabet_of_the_Magi.0\
   character01
2 ./data/Omniglot/images_background\Alphabet_of_the_Magi.0\
   character02
3 ./data/Omniglot/images_background\Alphabet_of_the_Magi.0\
   character03
4 ./data/Omniglot/images_background\Alphabet_of_the_Magi.0\
   character04
5 ./data/Omniglot/images_background\Alphabet_of_the_Magi.0\
   character05
6 ./data/Omniglot/images_background\Alphabet_of_the_Magi.0\
   character06
7 ./data/Omniglot/images_background\Alphabet_of_the_Magi.0\
   character07
8 ./data/Omniglot/images_background\Alphabet_of_the_Magi.0\
   character08
```

Listing 23: self.file_list 内容

- 第 6 行：每个批次中 Support Set 的大小为 k_way，Query Set 的大小为 q_query，该批次的总大小为 self.n；
- 第 11 行：取到某一个包含图片文件的目录；
- 第 12~14 行取出该目录下所有图片文件列表，每个目录下有 20 个文件；
- 第 15、16 行：类型为 list[tensor]，将目录下每个图片内容读出来转为 tensor，形状为 [1, 28, 28]，即黑白图片（通道数），分辨率为 28×28 ，20 个图片文件形成的 tensor 组成一个 list；
- 第 17 行：因为在第 9 行，生成 [0,1,...,19] 的列表，然后将其随机进行排序，运行结果如下所示：

```
1 sample: [ 7  0 18 16 15 17  9 12 11  1  5 14  4  6  8  3  2 10 19
           13];
2 sample[:self.n]: [7  0];
3 [sample[:self.n]]: [array([7,  0])];
4 imgs: torch.Size([2,  1, 28, 28]);
```

Listing 24: 运行原理

由此可见其形成的是一个训练中用到的迷你批次。

下面我们来看模型类 OgmModel，如下所示：

```
1 def ConvBlock(in_ch, out_ch):
2     return nn.Sequential(nn.Conv2d(in_ch, out_ch, 3, padding = 1),
3                         nn.BatchNorm2d(out_ch),
4                         nn.ReLU(),
5                         nn.MaxPool2d(kernel_size = 2, stride = 2))
6
7 def ConvBlockFunction(x, w, b, w_bn, b_bn):
```

```

8   x = F.conv2d(x, w, b, padding = 1)
9   x = F.batch_norm(x, running_mean = None, running_var = None,
10                  weight = w_bn, bias = b_bn, training = True)
11  x = F.relu(x)
12  x = F.max_pool2d(x, kernel_size = 2, stride = 2)
13  return x
14
15 class OgmlModel(nn.Module):
16     def __init__(self, in_ch, n_way):
17         super(OgmlModel, self).__init__()
18         self.conv1 = ConvBlock(in_ch, 64)
19         self.conv2 = ConvBlock(64, 64)
20         self.conv3 = ConvBlock(64, 64)
21         self.conv4 = ConvBlock(64, 64)
22         self.logits = nn.Linear(64, n_way)
23
24     def forward(self, x):
25         x = self.conv1(x)
26         x = self.conv2(x)
27         x = self.conv3(x)
28         x = self.conv4(x)
29         x = nn.Flatten(x)
30         x = self.logits(x)
31         return x
32
33     def functional_forward(self, x, params):
34         """
35         Arguments:
36             x: input images [batch, 1, 28, 28]
37             params: 模型的參數，也就是 convolution 的 weight 跟， bias
38                     以及 batchnormalization 的 weight 跟 bias
39                     這是一個 OrderedDict
40
41         for block in [1, 2, 3, 4]:
42             x = ConvBlockFunction(x, params[f'conv{block}.0.weight'],
43                                   params[f'conv{block}.0.bias'],
44                                   params.get(f'conv{block}.1.weight'),
45                                   params.get(f'conv{block}.1.bias'))
46             x = x.view(x.shape[0], -1)
47             x = F.linear(x, params['logits.weight'], params['logits.bias'])
48         return x

```

Listing 25: OgmlModel 类

OgmlModel 有两种工作模式，一种是正常的 forward 模式，由构造函数、ConvBolock 函数和 forward 方法定义，另一种是 functional_forward 模式，由 ConvBlockFunction 和 functional_forward 方法组成，在程序中我们使用的是 functional_forward 方法，因此我们只讲解这种工作模式。

- 第 18 行：生成第一个卷积层，调用 ConvBlock 方法：

- 第 2 行：调用 `torch.nn.Conv2d` 函数，输入信号通道数 `in_ch` 为 1，输出信号通道数 `out_ch` 为类别数，这里为 5，卷积核为 3×3 ，边缘零填充为 1，即经过卷积后输入信号维度不变；

- 第 3 行：调用 `torch.nn.BatchNorm2d` 函数，进行批归一化，其输入信号格式为 $[N, C, H, W]$ ，其中 N 为批次中序号， C 为通道数， H 为高， W 为宽，归一化公式为：

$$\tilde{x} = \frac{x - E(x)}{\sqrt{Var(x) + \epsilon}} * \gamma + \beta \quad (89)$$

- 第 4 行：经过 `ReLU` 函数；

- 第 5 行：调用 `MaxPool2d` 函数，每 2×2 中取最大值，步长为 2，即将图像尺寸缩小为原来的 $\frac{1}{4}$ ；

- 第 19 行：生成第 2 个卷积层块；
- 第 20 行：生成第 3 个卷积层块；
- 第 21 行：生成第 4 个卷积层块；
- 第 22 行：生成最后的输出层，输出层为类别数，经过第 1 个卷积层块尺寸变为 14×14 ，第 2 个卷积层块后为 7×7 ，经过第 3 个卷积层块为块为 3×3 ，经过第 4 个卷积块为 1×1 ，所以其就只有 64 个神经元；

下面我们来看训练过程，首先来看训练入口方法：

```
1 def train(self):  
2     n_way = 5  
3     k_shot = 1  
4     q_query = 1  
5     inner_train_steps = 1  
6     inner_lr = 0.4  
7     meta_lr = 0.001  
8     meta_batch_size = 32  
9     max_epoch = 4 #40  
10    eval_batches = 20  
11    train_data_path = './data/Omniglot/images_background/'  
12    dataset = OmniglotDs(train_data_path, k_shot, q_query)  
13    train_set, val_set = torch.utils.data.random_split(  
14        dataset, [32000,656])  
15    train_loader = DataLoader(train_set,  
16        batch_size = n_way,  
17        num_workers = 8,  
18        shuffle = True,  
19        drop_last = True)  
20    val_loader = DataLoader(val_set,  
21        batch_size = n_way,  
22        num_workers = 8,  
23        shuffle = True,  
24        drop_last = True)  
25    train_iter = iter(train_loader)  
26    val_iter = iter(val_loader)  
27    #  
28    meta_model = OgmModel(1, n_way).to(self.device)
```

```

29     optimizer = torch.optim.Adam(meta_model.parameters(), lr =
30         meta_lr)
31     loss_fn = nn.CrossEntropyLoss().to(self.device)
32     for epoch in range(max_epoch):
33         print("Epoch %d" %(epoch))
34         train_meta_loss = []
35         train_acc = []
36         for step in tqdm(range(len(train_loader)) //
37             (meta_batch_size)):
37             x, train_iter = self.get_meta_batch(
38                 meta_batch_size, k_shot, q_query,
39                 train_loader, train_iter
40             )
41             print('x: {0} - {1};'.format(type(x), x.shape))
42             sys.exit(0)
43             meta_loss, acc = self.train_batch(
44                 meta_model, optimizer, x, n_way,
45                 k_shot, q_query, loss_fn
46             )
47             train_meta_loss.append(meta_loss.item())
48             train_acc.append(acc)
49             print(" Loss : ", np.mean(train_meta_loss))
50             print(" Accuracy: ", np.mean(train_acc))
51             val_acc = []
52             for eval_step in tqdm(range(len(val_loader)) //
53                 (eval_batches)):
54                 x, val_iter = self.get_meta_batch(
55                     eval_batches, k_shot, q_query,
56                     val_loader, val_iter
57                 )
58                 _, acc = self.train_batch(
59                     meta_model, optimizer, x, n_way,
60                     k_shot, q_query, loss_fn,
61                     inner_train_steps = 3, train = False
62                 )
63                 val_acc.append(acc)
64             print(" Validation accuracy: ", np.mean(val_acc))
65             print('train is OK!')
66             torch.save(meta_model.state_dict(), self.chpt_file)

```

Listing 26: 训练入口类

代码解读如下所示：

- 第 2~10 行：定义所需变量，`n_way` 代表类别数，`k_shot` 是迷你批次中 support set 的样本数，`q_query` 是迷你批次中 query set 的样本数；
- 第 11~26 行：定义训练数据集和验证数据集；
- 第 28 行：创建模型，创建卷积层和最后的全连接层；
- 第 29 行：定义优化器；

- 第 30 行：定义代价函数为交叉熵函数；
- 第 31~61 行：训练指定 epochs 遍：
- 第 33 行：train_meta_loss；
- 第 34 行：train_acc；
- 第 35~48 行：循环处理训练数据集每个迷你批次：
- 第 37~40 行：取出一个迷你批次，其形状为 [32, 10, 1, 28, 28]，迷你批次大小为 32，共有 5 个类别，每个类别有 support set（这里是 1 个样本），query set（这里是 1 个样本），所以 $10 = n_way * (\text{support_set} + \text{query_set})$ ；
- 第 43~46 行：训练一个迷你批次，包括一次前向传播和误差反向传播过程，具体实现将在后面详述，计算出代价函数值和精度；
- 第 47 行：求迷你批次中代价函数之和；
- 第 48 行：求迷你批次中精度之和；
- 第 49 行：显示迷你批次的平均代价函数值；
- 第 50 行：显示迷你批次的平均精度值；
- 第 52、53 行：循环处理验证数据集每个迷你批次：
- 第 54~57 行：取出一个迷你批次；
- 第 58~62 行：求出每个迷你批次上的精度，这里只有前向传播过程，没有误差反向传播；
- 第 63 行：形成迷你批次精度列表；
- 第 64 行：打印迷你批次平均精度；
- 第 66 行：训练完成后保存网络状态；

下面我们来看迷你批次的训练过程，如下所示：

```

1 def train_batch(self, model, optimizer, x, n_way, k_shot,
2                 q_query, loss_fn, inner_train_steps=1,
3                 inner_lr=0.4, train=True):
4     """
5     Args:
6         x is the input omniglot images for a meta_step, shape = [batch_size,
7         n_way * (k_shot + q_query), 1, 28, 28]
8         n_way: 每個分類的 task 要有幾個 class
9         k_shot: 每個類別在 training 的時候會有多少張照片
10        q_query: 在 testing 時, 每個類別會用多少張照片 update
11        """
12        criterion = loss_fn
13        task_loss = []
14        task_acc = []
15        for meta_batch in x:
16            train_set = meta_batch[:n_way*k_shot]
17            val_set = meta_batch[n_way*k_shot:]
18            fast_weights = OrderedDict(model.named_parameters())
            for inner_step in range(inner_train_steps):

```

```

19         train_label = self.create_label(n_way, k_shot).to(self.device)
20     )
21     logits = model.functional_forward(train_set, fast_weights)
22     loss = criterion(logits, train_label)
23     grads = torch.autograd.grad(loss, fast_weights.values(), ,
24     create_graph = True)
25     fast_weights = OrderedDict((name, param - inner_lr * grad)
26                               for (name, param), grad in zip(
27                                 fast_weights.items(), grads))
28     val_label = self.create_label(n_way, q_query).to(self.device)
29     logits = model.functional_forward(val_set, fast_weights)
30     loss = criterion(logits, val_label)
31     task_loss.append(loss)
32     acc = np.asarray([torch.argmax(logits, -1).cpu().numpy() ==
33                       val_label.cpu().numpy()]).mean()
34     task_acc.append(acc)
35   model.train()
36   optimizer.zero_grad()
37   meta_batch_loss = torch.stack(task_loss).mean()
38   if train:
39     meta_batch_loss.backward()
40     optimizer.step()
41   task_acc = np.mean(task_acc)
42   return meta_batch_loss, task_acc

```

Listing 27: 迷你批次训练过程

代码解读如下所示：

- 第 14 行：每个 meta_batch 的形状为 [10, 1, 28, 28];
- 第 15 行：前 n_way*k_shot 个样本为 train_set（或者叫 support set）；
- 第 16 行：其余样本为验证数据集（或者叫 query set）；
- 第 17 行：拷贝一份模型参数，在本例中只有一个任务，所以只拷贝了一份，如果是多个任务，需要拷贝多份；
- 第 18 行：内循环通常为一次，也可能是多次（以下是对一个任务的处理流程）：
- 第 19 行：生成正确答案，对于很多任务，正确答案在数据集中已经定义好了；
- 第 20 行：调用模型前向传播过程，产生网络输出，代码见表25：
 - 第 41~45 行：循环处理四个卷积块，调用 ConvBlockFunction 函数，从命名参数中：conv1.0.weight 和 conv1.0.bias 为卷操作参数，；
 - 第行：；
 - 第行：；
 - 第行：；

9 当前位置

第四篇实盘实践

第 401 章火币网开发实践

Abstract

在本章中，我们将搭建一个火币网交易平台，实验获取行情数据，买入卖出比特币和 USDT，实现自动化交易。

10 火币网开发环境搭建

火币网目前是全球第四大交易所，在国内第一大交易所，除法币与虚拟币交易目前需要采用线下交易外，币币交易非常方便，并且交易量非常大，是一个很好的试验平台。本章中，我们只讨论获取行情数据以及下单买卖，具体交易策略，将在随后章节中逐一详细介绍。

10.1 账号注册

火币网网址为 huobi.com，需要翻墙才能访问，可以通过手机号注册账户。到这个网址申请 API：

```
1 https://www.huobi.com/zh-cn/apikey/
```

Listing 28: 申请火币 API

需要记住网站提供的 AccessKey 和 SecretKey。需要同时进行手机号和邮箱认证，如果不提供公网地址的话，API 有效期为 90 天。API 文档网址：

```
1 https://huobiapi.github.io/docs/spot/v1/cn/#ecae7085ce
```

Listing 29: 火币 API 文档

另外，启动交易引擎：

```
1 cd e:\abiz\aqp
2 conda activate py37
3 python -m unittest tdd.app.fme.t_fme_engine.TFmeEngine.test_startup
```

Listing 30: 启动交易引擎简易方法

10.2 获取行情数据

10.3 币币交易

第 402 章统计套利

Abstract

在本章中，我们将根据 A 股历史数据，完成基于交易对的统计套利交易策略。

11 统计套利

在股市中，同行业的两支股票，由于所处的宏观经济环境和市场环境相同，其股价之间一般会维持一个恒定比例，当出现某些特殊情况时，某一支股票会出现比较大的波动，这时二者之间的股价比例将发生大的波动，但是会在较短的时间内，又回到原来的比特。如果我们同时持有这两支股票 S1 和 S2，当其中 S1 临时出现股价上涨高估时，我们就卖出 S1 的股票，买入 S2 的股票，当 S1 股价回落时，我们再将这次买入的 S2 股票卖掉，重新买入 S1 的股票，保持 S1 和 S2 股票的持有数量和比例不变，但是由于 S1 出现股价波动，我们实际赚到了这次波的利润。S1 出现低估时，需要反向操作，原理相同。

基于交易对的统计套利策略包括三大问题：

- 交易对选择；
- 交易策略制定；
- 交易中的仓位管理；下面我们分别来讨论这三个问题。

11.1 交易对选择

基于交易对的统计套利方法分为形成期和交易期。形成期就是根据历史行情数据，找出适合交易的交易对。交易对选择方法如下所示：

- 行业内匹配法：选择同行业规模和业务模式相近的企业，例如我们在本章例子中所举的，选择银行业相关股票；
- 产业链配对法：选择同一产业链上下游企业配对，例如手机厂商和摄像头厂商配对；
- 财务管理配对法：从基本面分析入手，选择市盈率、负债率、产品种类相似的企业进行配对；

国内沪深两市共有 2780 支股票，如果我们拿任意两种股票进行配对，会有 3862810 种组合，计算量过大。在实际应用中，我们通常选择沪深 500 指数股、上证 50 成份股和创业板指数股作为股票池，从中选择交易对。另外，有一些股票，同时在不同交易所上市，例如中石油就同时在国内 A 股和香港 H 股上市，由于二者表示同一家公司的股票，可以构成交易对。下面我们来看交易对的选择标准。

11.1.1 最小距离法

我们的目标是选择历史价差稳定的交易对，我们首先需要对股票价格数据进行标准化，假设第 i 支股票的价格序列为：

$$P_t^i \quad t \in \{1, 2, 3, \dots, T\} \tag{90}$$

我们定义第 i 支股票在第 t 天的单期收益率为:

$$r_t^i = \frac{P_t^i - P_{t-1}^i}{P_{t-1}^i}, \quad t = 1, 2, 3, \dots, T \quad (91)$$

我们定义第 i 支股票在第 t 天的标准价格为:

$$\hat{p}_t^i = \sum_{\tau=1}^t (1 + r_{\tau}^i) \quad (92)$$

我们定义股票 X、Y 的平方标准化价格差 SSD 为:

$$SSD_{X,Y} = \sum_{t=1}^T (\hat{p}_t^X - \hat{p}_t^Y)^2 \quad (93)$$

下面我们以上证 50 成份股为例, 找出其出潜在的交易对:

```

1  #
2  import numpy as np
3  import pandas as pd
4
5  class TpApp(object):
6      def __init__(self):
7          self.name = 'app_tp.TpApp'
8          self.stock_pool = [
9              '600000', '600010', '600015', '600016', '600018',
10             '600028', '600030', '600036', '600048', '600050',
11             '600104', '600109', '600111', '600150', '600518',
12             '600519', '600585', '600637', '600795', '600837',
13             '600887', '600893', '600999', '601006',
14             '601088', '601166', '601169', '601186',
15             '601318', '601328', '601390',
16             '601398', '601601', '601628', '601668',
17             '601766', '601857',
18             '601988', '601989', '601998']
19          self.tpc = {}
20
21      def startup(self):
22          print('交易对应用')
23          sh = pd.read_csv('./data/sh50p.csv', index_col='Trddt')
24          sh.index = pd.to_datetime(sh.index)
25          form_start = '2014-01-01'
26          form_end = '2015-01-01'
27          sh_form = sh[form_start : form_end]
28          tpc = {}
29          sp_len = len(self.stock_pool)
30          for i in range(sp_len):
31              for j in range(i+1, sp_len):
32                  tpc['{}-{}'.format(self.stock_pool[i], self.stock_pool[j])] = self.trading_pair(sh_form, self.stock_pool[i], self.stock_pool[j])
33          self.tpc = sorted(tpc.items(), key=lambda x: x[1])
34          for itr in self.tpc:

```

```

35     print('{0}: {1}'.format(itr[0], itr[1]))
36
37 def trading_pair(self, sh_form, stock_x, stock_y):
38     # 中国银行股价
39     PAf = sh_form[stock_x]
40     # 取浦发银行股价
41     PBf = sh_form[stock_y]
42     # 求形成期长度
43     pairf = pd.concat([PAf, PBf], axis=1)
44     form_len = len(pairf)
45     return self.calculate_SSD(PAf, PBf)
46
47 def calculate_SSD(self, price_x, price_y):
48     if price_x is None or price_y is None:
49         print('缺少价格序列')
50         return
51     r_x = (price_x - price_x.shift(1)) / price_x.shift(1)[1:]
52     r_y = (price_y - price_y.shift(1)) / price_y.shift(1)[1:]
53     #hat_p_x = (r_x + 1).cumsum()
54     hat_p_x = (r_x + 1).cumprod()
55     #hat_p_y = (r_y + 1).cumsum()
56     hat_p_y = (r_y + 1).cumprod()
57     SSD = np.sum((hat_p_x - hat_p_y)**2)
58     return SSD

```

Listing 31: 基于最小距离法的上证 50 成份股交易对选择

11.1.2 协整模型

协整模型是更常用的一种选择交易对的方法。我们假设股票 X 在第 t 日价格为:

$$P_t^X, \quad t = 1, 2, 3, \dots, T \quad (94)$$

定义股票 X 的对数价格序列:

$$\log(P_t^X), \quad t = 1, 2, 3, \dots, T \quad (95)$$

一般股票 X 的对数价格序列是非平稳时间序列，我们定义股票 X 的一阶差分序列:

$$\log(P_t^X) - \log(P_{t-1}^X) = \log\left(\frac{P_t^X}{P_{t-1}^X}\right), \quad t = 1, 2, 3, \dots, T \quad (96)$$

如果一阶差分序列是平稳时间序列，则称股票 X 的价格序列为一阶单整序列。

股票 X 在第 t 期单期简单收益率为:

$$r_t^X = \frac{P_t^X - P_{t-1}^X}{P_{t-1}^X} = \frac{P_t^X}{P_{t-1}^X} - 1 \quad (97)$$

我们根据式96和式97可得:

$$\frac{P_t^X - P_{t-1}^X}{P_{t-1}^X} = \frac{P_t^X}{P_{t-1}^X} = \log(r_t^X + 1) \approx r_t^X \quad (98)$$

要判断两支股票历史价格是否具有协整关系，首先要检查两支股票的对数价格序列是否为一阶单整序列，根据式98也就是检查两支股票单期收益率 r_t^i 是否为平稳时间序列。

假设股票 X 的对数价格序列 $\log(P_t^X)$ 和股票 Y 的对数价格序列 $\log(P_t^Y)$ 均为一阶单整序列，则可以用最小二乘法构造一阶回归方程：

$$\log(P_t^Y) = \alpha + \beta \log(P_t^X) + \epsilon_t \quad (99)$$

如果我们可以确定回归系数 $\hat{\alpha}$ 和 $\hat{\beta}$ ，则可以得到残差序列：

$$\hat{\epsilon}_t = \log(P_t^Y) - \hat{\alpha} - \hat{\beta} \log(P_t^X) \quad (100)$$

如果我们可以证明 $\hat{\epsilon}_t$ 是时间平稳序列，就可以证明股票 X 和股票 Y 的对数价格序列具有
一阶协整关系。

证明交易对是否具有协整关系的程序如下所示：

```

1 #
2 import numpy as np
3 import pandas as pd
4 from arch.unitroot import ADF
5 import statsmodels.api as sm
6
7 class CitpEngine(object):
8     def __init__(self):
9         self.name = 'apps.tp.CitpEngine'
10
11    def exp(self):
12        print('协整模型试验')
13        stock_x = '601988'
14        stock_y = '600000'
15        sh = pd.read_csv('./data/sh50p.csv', index_col='Trddt')
16        sh.index = pd.to_datetime(sh.index)
17        form_start = '2014-01-01'
18        form_end = '2015-01-01'
19        sh_form = sh[form_start : form_end]
20        # 中国银行股价
21        PAf = sh_form[stock_x]
22        #r = PAf - PAf.shift(1)
23        #adf1 = ADF(r[1:])
24        #print(adf1.summary().as_text())
25        rst = self.calculate_adf(PAf)
26        if not rst:
27            print('中国银行收益率不是时间平稳序列')
28            return
29        else:
30            print('中国银行收益率是时间平稳序列 ^_^')
31        # 取浦发银行股价
32        PBf = sh_form[stock_y]
33        rst = self.calculate_adf(PBf)
34        if not rst:
35            print('浦发银行收益率不是时间平稳序列')
36            return
37        else:
38            print('浦发银行收益率是时间平稳序列 ^_^')
39        log_p_x = np.log(PAf)

```

```

40     log_p_y = np.log(PBf)
41     model = sm.OLS(log_p_y, sm.add_constant(log_p_x))
42     result = model.fit()
43     print(result.summary())
44     alpha = result.params[0]
45     beta = result.params[1]
46     epsilon = log_p_y - alpha - beta * log_p_x
47     # 残差项均值为零, 需要使用trend='nc'
48     adf_epsilon = ADF(epsilon, trend='nc')
49     epsilon_stable = True
50     for val in adf_epsilon.critical_values.values():
51         if adf_epsilon.stat >= val:
52             epsilon_stable = False
53     if not epsilon_stable:
54         print('残差项为非平稳时间序列')
55         return
56     print('可以使用协整模型来创建交易对')
57
58 def calculate_adf(self, p_x):
59     log_p_x = np.log(p_x)
60     ret_x = log_p_x.diff()[1:]
61     adf_ret_x = ADF(ret_x)
62     for val in adf_ret_x.critical_values.values():
63         if adf_ret_x.stat >= val:
64             return False
65     return True

```

Listing 32: 基于协整模型的交易对选择

11.2 强化学习环境

我们为了支持策略回测和与深度元强化学习框架统一，我们采用强化学习环境来做交易回测。我们首先定义强化学习环境，如下所示：

代码解读如下：

- 第 9 行：action 为 [3,3]，第 1 维代表对股票 X 的 0-买入、1-保持、2-卖出，第 2 维表示 0-自有资金、1-5 倍杠杆、2-10 倍杠杆，例如 a[0]=0,a[1]=3，则为买入股票 X，使用 10 倍杠杆，此时需要借钱来买入股票，再例如 a[0]=2；
- 第 10~12 行：股价最低为 0 元，最高为 10000 元，shape 为 (1, 5) 就是之前 5 天的收盘价，为浮点数；
- 第行::；

12 当前

12.1 交易策略

12.1.1 最小距离法

我们在形成期找到了历史价格具有最小距离的交易对，如前例子中的中国银行和浦发银行，假设交易对为股票 X 和 Y ，其标准化价格差为：

$$\hat{p}_t^X - \hat{p}_t^Y \quad (101)$$

我们可以求出其均值 μ 和标准差 σ ，由于我们例子中的银行股比较稳定，我们定义一个较小的阈值 $\lambda = 1.2$ ，我们定义正常的股价标准价差区间为： $\mu \pm \lambda\sigma$ ，当股价标准价差超出该区间时，我们进行如下操作：

- $\hat{p}_t^X - \hat{p}_t^Y < \mu - \lambda\sigma$ 时：我们卖出股票 Y ，全部买入股票 X 。当股价标准价差回归正常区间时，我们卖出刚买入的股票 X ，买入股票 Y ；
- $\hat{p}_t^X - \hat{p}_t^Y > \mu + \lambda\sigma$ 时：我们卖出股票 X ，全部买入股票 Y 。当股价标准价差回归正常区间时，我们卖出刚买入的股票 Y ，买入股票 X ；

12.1.2 协整模型

算法：仓位 position_state 有以下四种情况：

- 0：股票 X 和股票 Y 均为空仓，价格水平为 3 和 -3 时；
- 1：按比例分别持有 X 和 Y，价格水平为 0 时；
- 2：仅持有 X，价格水平为 2 时；
- 3：仅持有 Y，价格水平为 -2 时；

12.2 XGBoost 强化学习

12.2.1 数据集定义

12.2.2 强化学习环境 RxgbEnv

13 附录 X

References