蘑菇先生学习记

Python实现时间序列分析

2017-03-08 | 统计学 | 阅读量 409

前面花了两章篇幅介绍了时间序列模型的数学基础。ARIMA时间序列模型(一)和ARIMA时间序列模型(二)。本文重点介绍使用python开源库进行时间序列模型实践。

基本概念

回顾一下自回归移动平均模型ARMA，它主要由两部分组成：AR代表p阶自回归过程，MA代表q阶移动平均过程，形式如下：

$$Z_t = \theta_0 + \phi_1 Z_{t-1} + \phi_2 Z_{t-2} + \ldots + \phi_p Z_{t-p} \\ + a_t - \theta_1 a_{t-1} - \theta_2 a_{t-2} - \ldots - \theta_q a_{t-q}$$

为了方便，我们重写以上等式为：

$$\phi(B) Z_t = \theta_0 + \theta(B) a_t$$

其中，$\phi(x)$和$\theta(x)$分别是AR模型和MA模型的的特征多项式

$$\phi(x) = 1 - \phi_1 x - \phi_2 x^2 - \ldots - \phi_p x^p$$

$$\theta(x) = 1 - \theta_1 x - \theta_2 x^2 - \ldots - \theta_p x^q$$

根据前两篇的分析，我们总结ARMA模型的性质如下：

	AR(p)	MA(q)	ARMA(p,q)
模型方程	$\varphi(B)=a_t$	$z_t=\theta(B)a_t$	$\varphi(B)z_t=\theta(B)\ a_t$
平稳性条件	$\varphi(B)=0$的根在单位圆外	无	$\varphi(B)=0$的根在单位圆外
可逆性条件	无	$\theta(B)=0$的根在单位圆外	$\theta(B)=0$的根在单位圆外
自相关函数	拖尾	Q步截尾	拖尾
偏自相关函数	P步截尾	拖尾	拖尾

p值检验

在开始之前，我们首先回顾一下p值检验。

一般地，用X表示检验的统计量，当H0为真时，可由样本数据计算出该统计量的值C，根据检验统计量X的具体分布，可求出P值。具体地说：

- 左侧检验的P值为检验统计量X小于样本统计值C的概率，即：P = P{ X < C}
- 右侧检验的P值为检验统计量X大于样本统计值C的概率：P = P{ X > C}
- 双侧检验的P值为检验统计量X落在样本统计值C为端点的尾部区域内的概率的2倍：P = 2P{ X > C} (当C位于分布曲线的右端时) 或P = 2P{ X< C}(当C位于分布曲线的左端时) 。若X服从正态分布和t分布，其分布曲线是关于纵轴对称的，故其P值可表示为P=P{|X|>C}。

  计算出P值后，将给定的显著性水平α与P值比较，就可作出检验的结论：

  如果$p < \alpha$值，则在显著性水平α下拒绝原假设。

  如果$P \geq \alpha$值，则在显著性水平α下接受原假设。

pandas数据操作

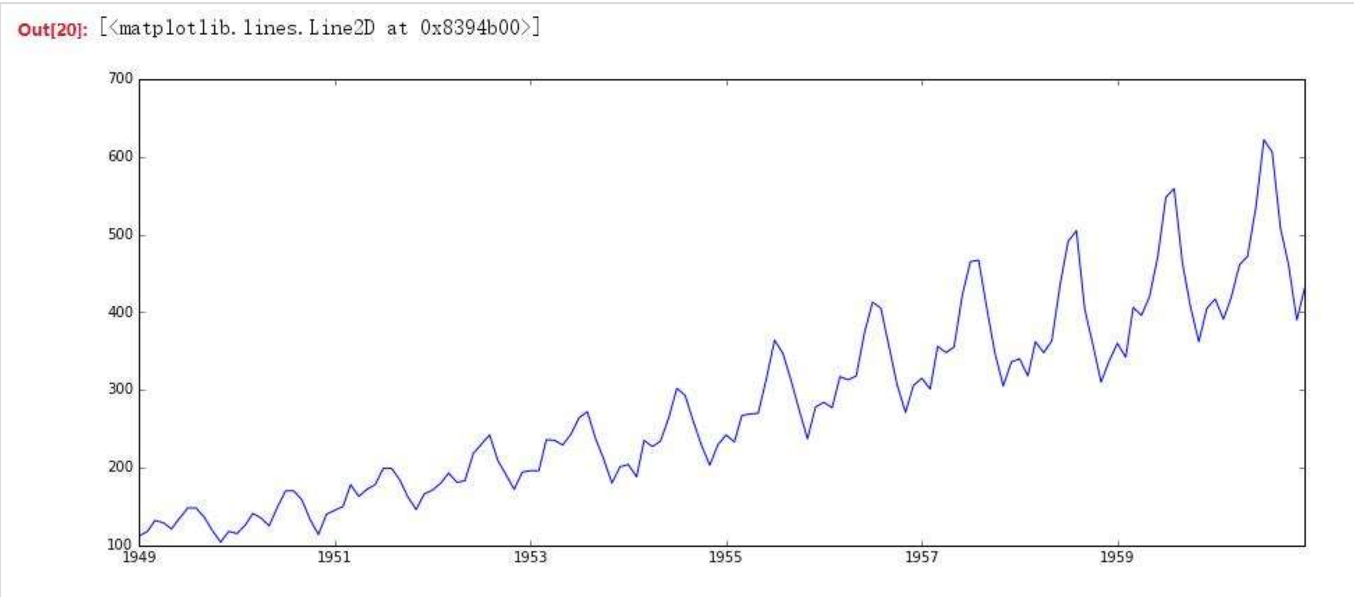
使用pandas来加载数据，并对数据索引进行转换，使用日期作为索引。

```
dateparse = lambda dates:pd.datetime.strptime(dates,'%Y-%m')
data=pd.read_csv('AirPassengers.csv',parse_dates='Month',index_col='Month',date_parser=
print data.head()
# 数据如下所示：
```

```
Month

1949-01-01          112

1949-02-01          118

1949-03-01          132

1949-04-01          129

1949-05-01          121
```

接着绘制数据：

```
ts = data['#Passengers']
plt.plot(ts)
```



非常清晰的看到，随着季节性的变动，飞机乘客的数量总体上是在不断增长的。但是，不是经常都可以获得这样清晰的视觉体验。我们可以通过下面的方法测试稳定性。

稳定性检测

- **绘制滚动统计**：我们可以绘制移动平均数和移动方差，观察它是否随着时间变化。
- **ADF检验：**这是一种检查数据稳定性的统计测试。无效假设：时间序列是不稳定的。测试结果由测试统计量和一些置信区间的临界值组成。如果“测试统计量”少于“临界

值”，我们可以拒绝无效假设，并认为序列是稳定的。或者根据前面提高的p值检验，如果p值小于显著性水平，我们可以拒绝无效假设，认为序列稳定。

滚动统计

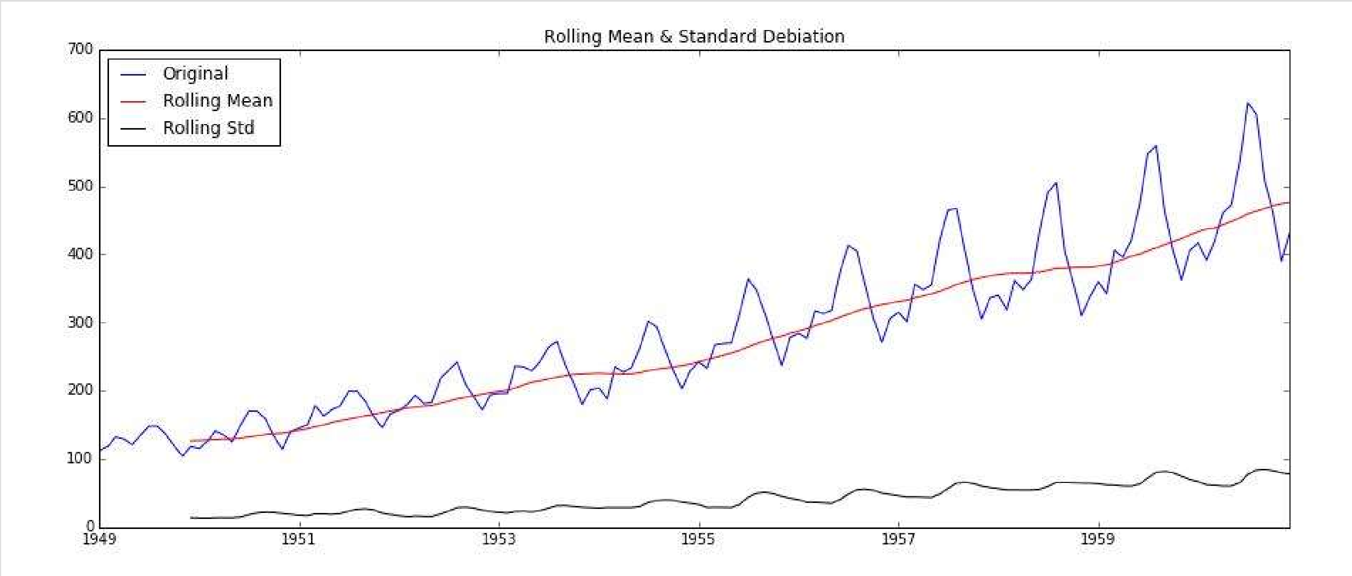
```
def rolling_statistics(timeseries):
    #Determing rolling statistics
    rolmean = pd.rolling_mean(timeseries, window=12)
    rolstd = pd.rolling_std(timeseries, window=12)

    #Plot rolling statistics:
    orig = plt.plot(timeseries, color='blue',label='Original')
    mean = plt.plot(rolmean, color='red', label='Rolling Mean')
    std = plt.plot(rolstd, color='black', label = 'Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)
```

pd.rolling_mean有两个参数，第一个是输入数据，第二个是窗口大小。假设有个序列是，1 2 3 3 5 8 6 9，如果窗口大小为3，那么移动平均数计算过程如下： 第一步: (1+2+3)/3 =2; 第二步:往右移动一个数据，(2+3+3)/3=2.667; 第三步, (3+3+5)/3=3.667; 第四步: (3+5+8)/3=5.333; 第四步: (5+8+6)/3=6.333; 第五步;(8+6+9)/3=7.667; 因此移动平均数序列为： NA NA 2 2.667 3.667 5.3333 6.333 7.667. 共用n-windows+1个数。

0	NaN
1	NaN
2	2.000000
3	2.666667
4	3.666667
5	5.333333
6	6.333333
7	7.666667

移动标准差类似，只不过把求平均变成了求标准差。

绘图如下：可以看出移动平均数仍然是上升趋势，而移动标准差相对比较平稳。



ADF检验

```
from statsmodels.tsa.stattools import adfuller
def adf_test(timeseries):
    rolling_statistics(timeseries)#绘图
    print 'Results of Augment Dickey-Fuller Test:'
    dftest = adfuller(timeseries, autolag='AIC')
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags Used','Nu
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    print dfoutput
```

```
Results of Fickey-Fuller Test:
Test Statistic                 0.815369
p-value                        0.991880
#Lags Used                    13.000000
Number of Observations Used  130.000000
Critical Value (5%)           -2.884042
Critical Value (1%)           -3.481682
Critical Value (10%)          -2.578770
dtype: float64
```

上述输出如何解读?

- Test statistic：代表检验统计量
- p-value：代表p值检验的概率
- Lags used：使用的滞后k，autolag=AIC时会自动选择滞后
- Number of Observations Used：样本数量
- Critical Value(5%) : 显著性水平为5%的临界值。

ADF检验

- 假设是存在单位根，即不平稳；
- 显著性水平，1%：严格拒绝原假设；5%：拒绝原假设，10%类推。
- 看P值和显著性水平a的大小，p值越小，小于显著性水平的话，就拒绝原假设，认为序列是平稳的；大于的话，不能拒绝，认为是不平稳的
- 看检验统计量和临界值，检验统计量小于临界值的话，就拒绝原假设，认为序列是平稳的；大于的话，不能拒绝，认为是不平稳的

根据上文提到的p值检验以及上面的结果，我们可以发现p=0.99>10%>5%>1%, 并且检验统计量0.815>>-2.58>-2.88>-3.48，因此可以认定原序列不平稳。

先让我们弄明白是什么导致时间序列不稳定。两个主要原因。

- **趋势-随着时间产生不同的平均值。**举例：在飞机乘客这个案例中，我们看到总体上，飞机乘客的数量是在不断增长的。
- **季节性-特定时间框架内的变化。**举例：在特定的月份购买汽车的人数会有增加的趋势，因为车价上涨或者节假日到来。

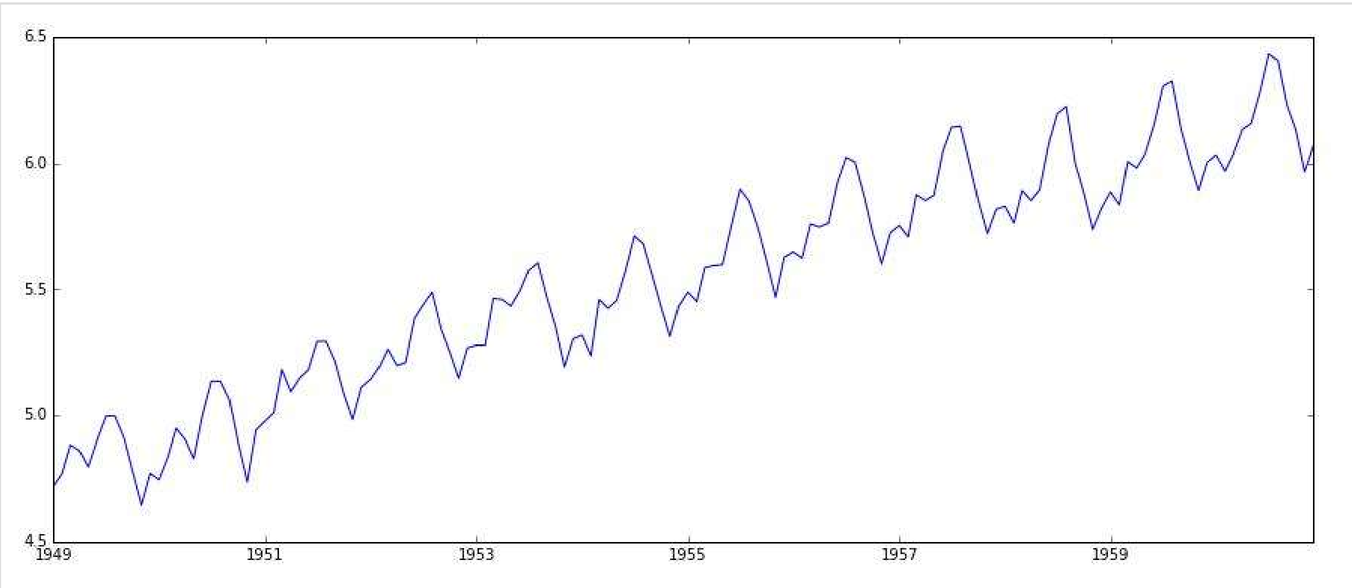
我们的基本原理是，通过建模并估计趋势和季节性这些因素，并从时间序列中移除，来获得一个稳定的时间序列，然后再使用统计预测技术来处理时间序列，最后将预测得到的数据，通过加入趋势和季节性等约束，来回退到原始时间序列数据。

平稳性处理

消除趋势的第一个方法是转换。例如,在本例中,我们可以清楚地看到该时间序列有显著趋势。所以我们可以通过变换，惩罚较高值而不是较小值。这可以采用对数, 平方根,立方跟等等。

对数变换

```
ts_log = np.log(ts)
plt.plot(ts_log)
```

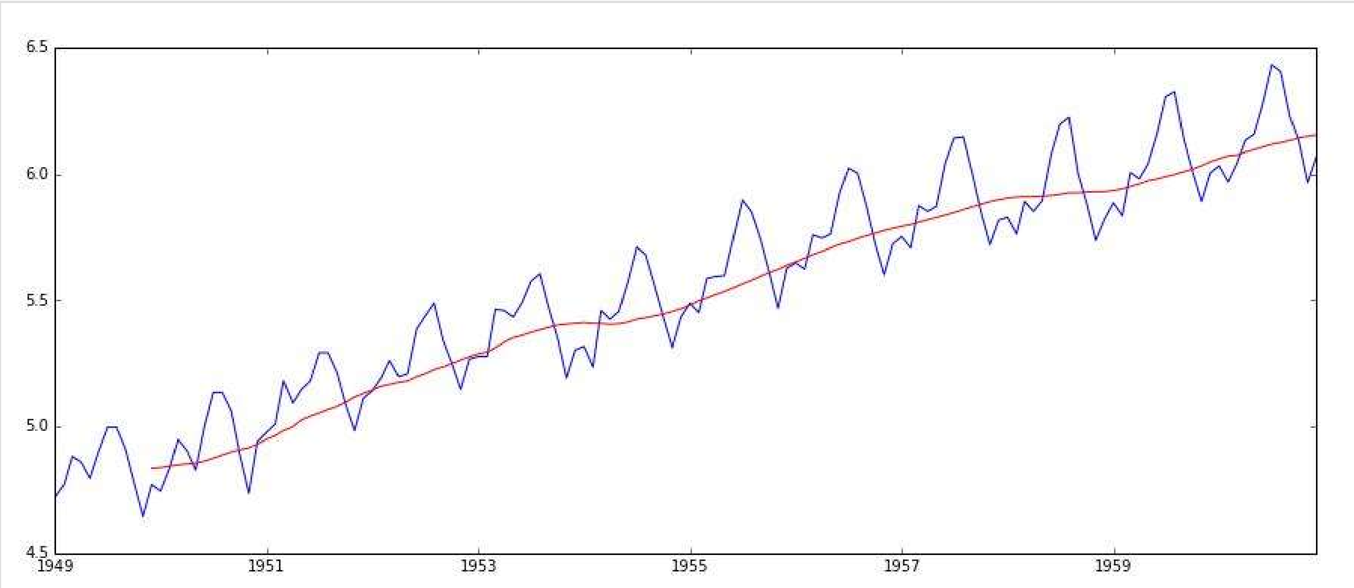


在这个例子中,很容易看到一个向前的趋势。但是它表现的不是很直观。我们可以使用一些技术来对这个趋势建模,然后将它从序列中删除。最常用的方法有:

- **平滑-取滚动平均数**
- **差分**
- **分解**

移动平均数

```
moving_avg = pd.rolling_mean(ts_log,12)
plt.plot(ts_log)
plt.plot(moving_avg,color='red')
```

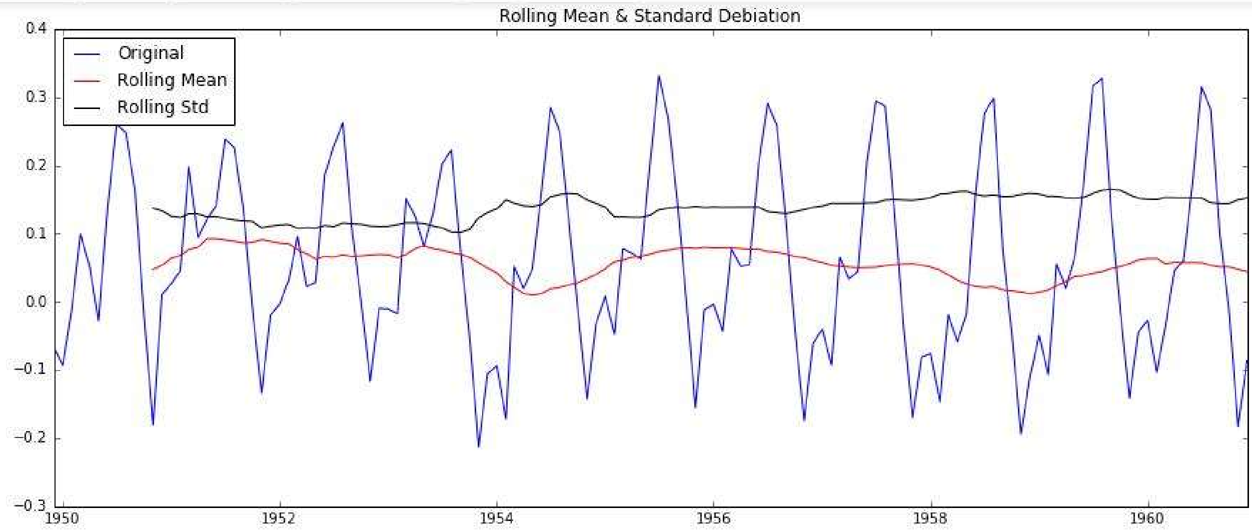



```
#做差
ts_log_moving_avg_diff = ts_log - moving_avg
ts_log_moving_avg_diff.head(12)
```

```
Month
1949-01-01          NaN
1949-02-01          NaN
1949-03-01          NaN
1949-04-01          NaN
1949-05-01          NaN
1949-06-01          NaN
1949-07-01          NaN
1949-08-01          NaN
1949-09-01          NaN
1949-10-01          NaN
1949-11-01          NaN
1949-12-01    -0.065494
1950-01-01    -0.093449
Name: #Passengers, dtype: float64
```

前11个数是NA

```
adf_test(ts_log_moving_avg_diff)
```

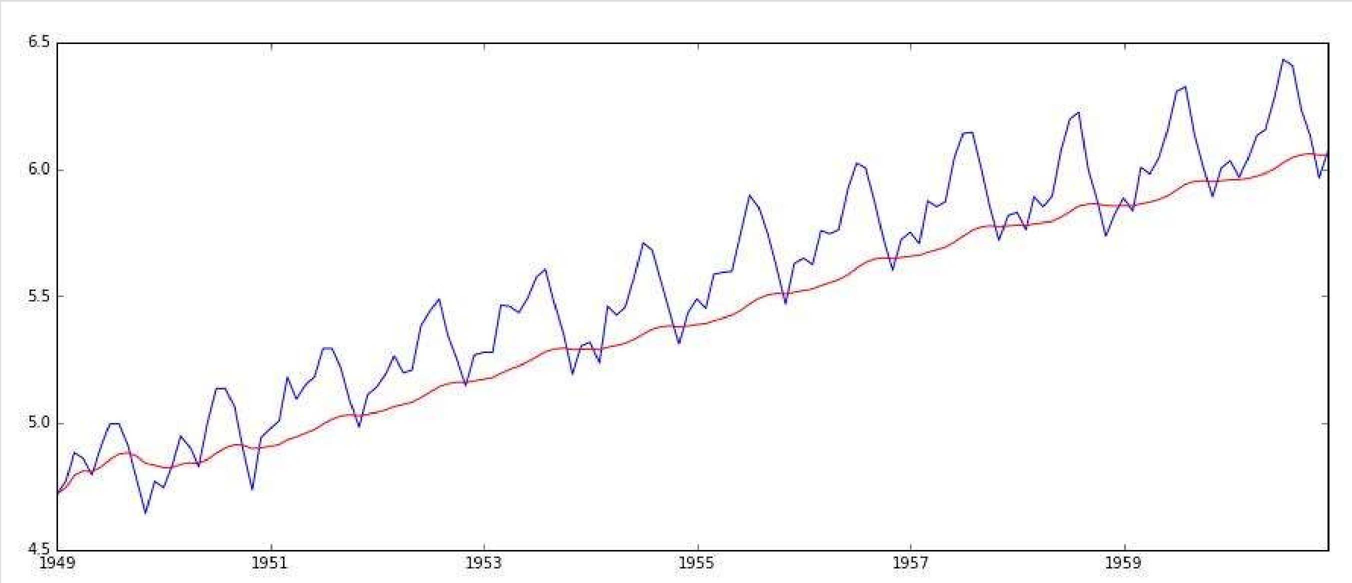



```
Results of Fickey-Fuller Test:
Test Statistic                  -3.162908
p-value                          0.022235
#Lags Used                      13.000000
Number of Observations Used    119.000000
Critical Value (5%)             -2.886151
Critical Value (1%)             -3.486535
Critical Value (10%)            -2.579896
dtype: float64
```

可以发现通过了5%和10%的显著性检验，即在该水平下，拒绝原假设，认为序列是平稳的，但是没有通过1%的检验。

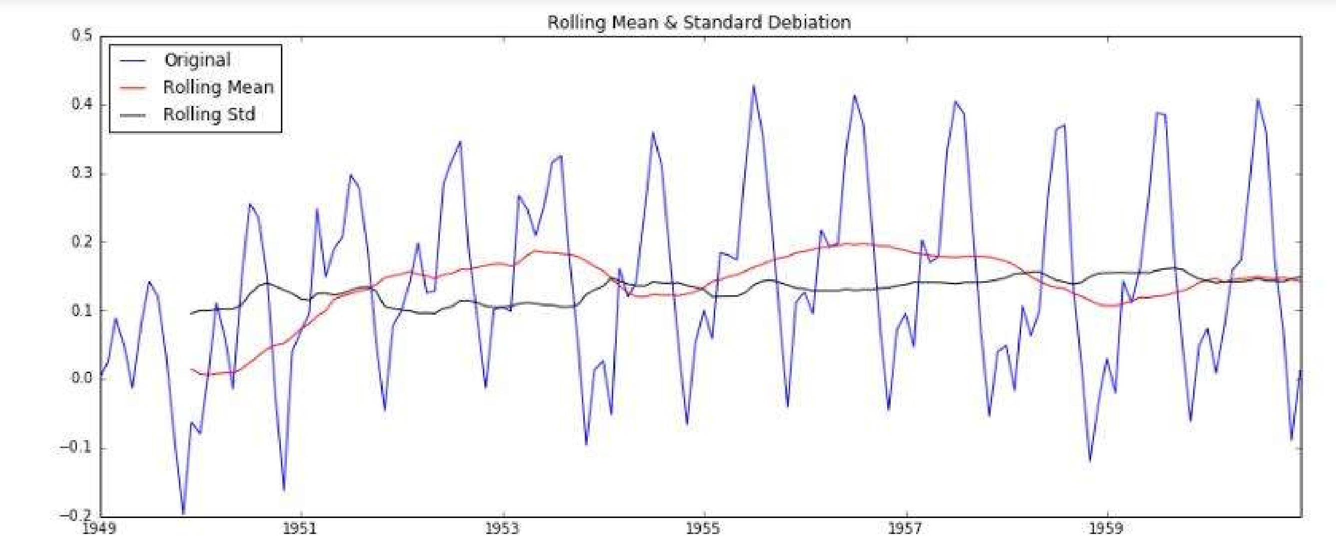
指数加权移动平均

```
expwighted_avg=pd.ewma(ts_log,halflife=12)
plt.plot(ts_log)
plt.plot(expwighted_avg, color='red')
```



前面移动平均数需要指定window,并且对所有的数一视同仁；这里采用指数加权移动平均方法，会对当前的数据加大权重，对过去的数据减小权重。halflife半衰期，用来定义衰减量。其他参数,如跨度span和质心com也可以用来定义衰减。

```
#做差
ts_log_ewma_diff = ts_log - expwighted_avg
adf_test(ts_log_ewma_diff)
```



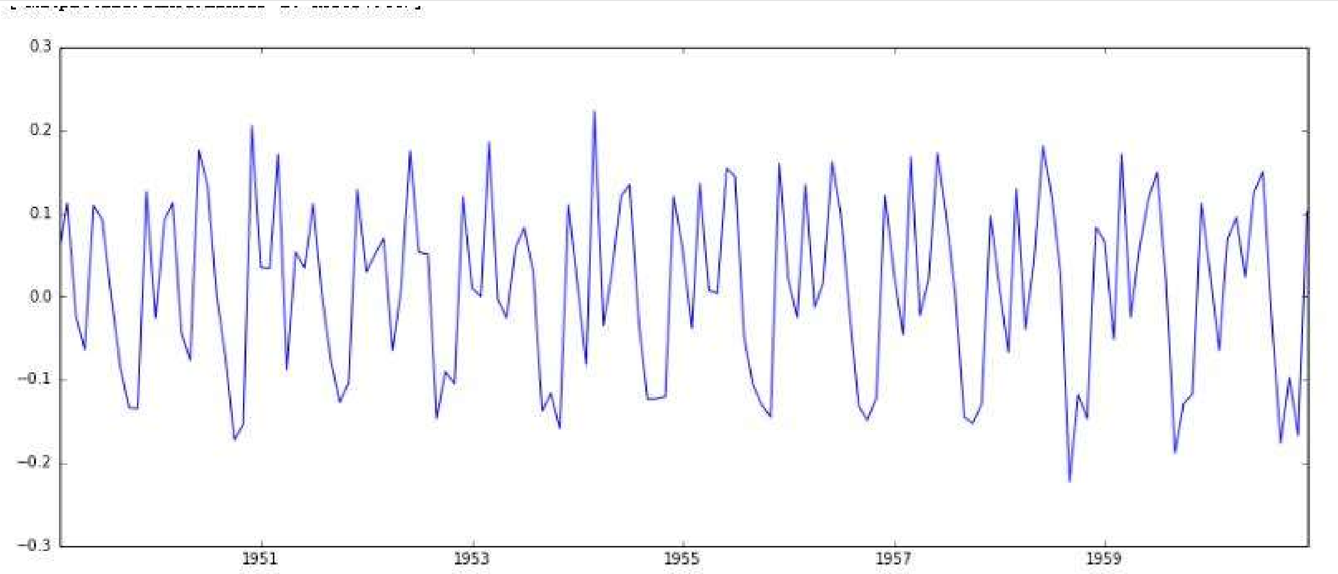

```
Results of Fickey-Fuller Test:
Test Statistic                  -3.601262
p-value                          0.005737
#Lags Used                      13.000000
Number of Observations Used    130.000000
Critical Value (5%)             -2.884042
Critical Value (1%)             -3.481682
Critical Value (10%)            -2.578770
dtype: float64
```

可以发现，经过指数移动平均后，再做差的结果，已经能够通过1%显著性水平检验了。

差分

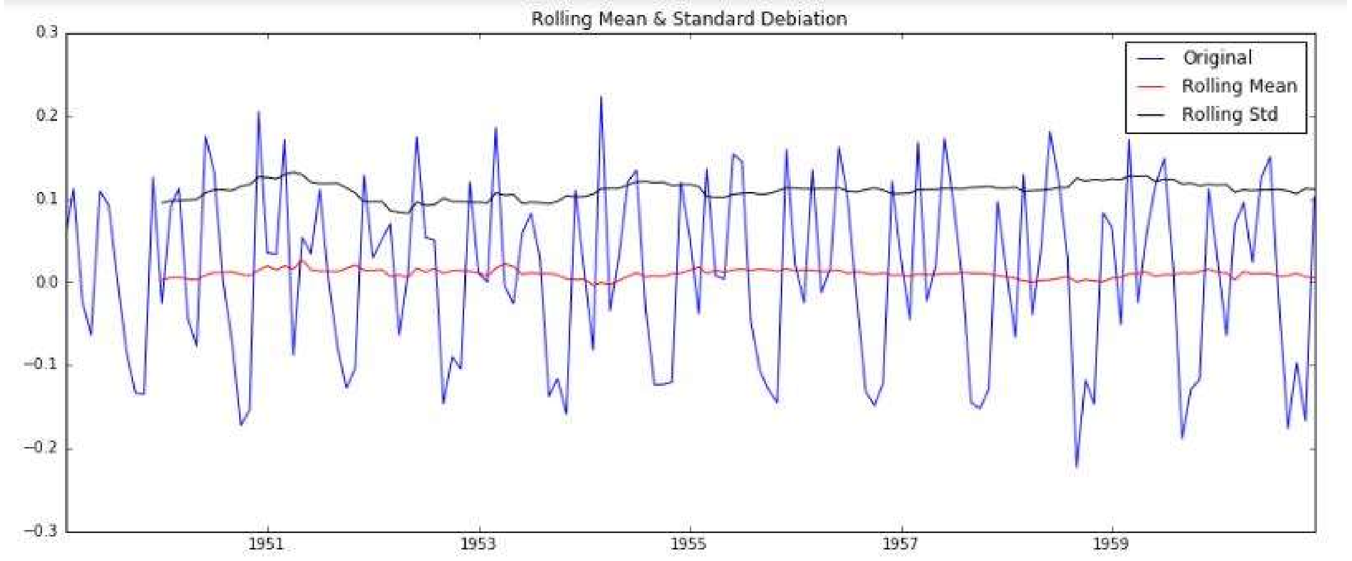
```
#步长为1的一阶差分
ts_log_diff = ts_log - ts_log.shift(periods=1)
plt.plot(ts_log_diff)
```

我们首先使用步长为1的一阶差分，得到如下图：



接着进行adf检验，

```
#只通过了10%的检验
ts_log_diff.dropna(inplace=True)
test_stationarity(ts_log_diff)
```



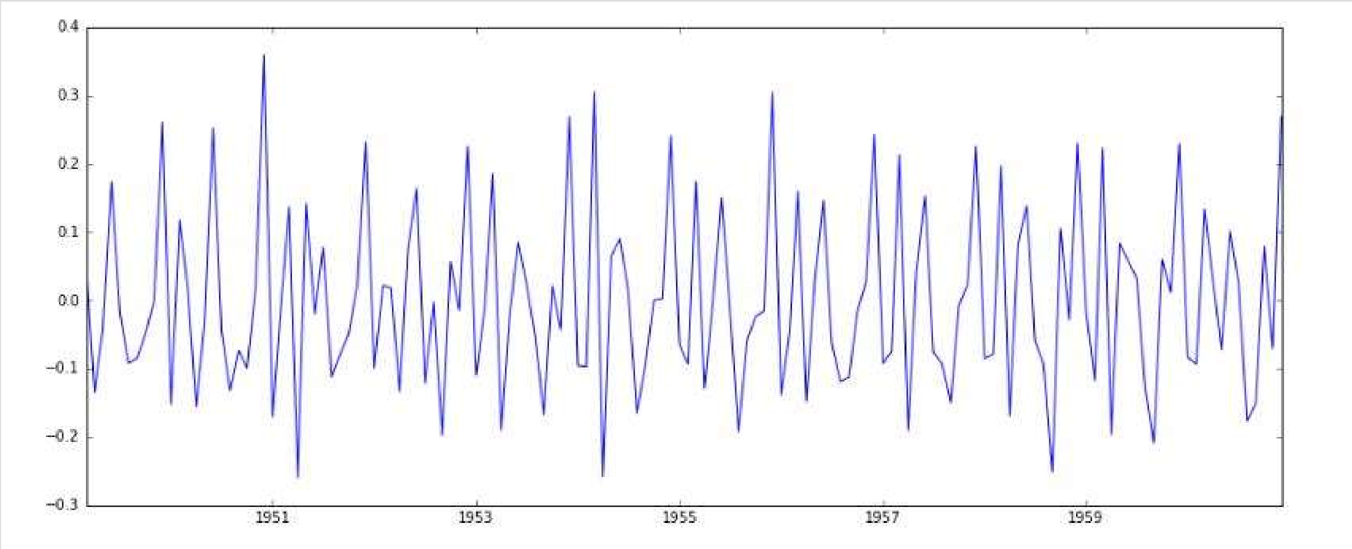

```
Results of Fickey-Fuller Test:
Test Statistic                 -2.717131
p-value                         0.071121
#Lags Used                     14.000000
Number of Observations Used   128.000000
Critical Value (5%)            -2.884398
Critical Value (1%)            -3.482501
Critical Value (10%)           -2.578960
dtype: float64
```

可以发现只通过了10%的显著性水平检验。

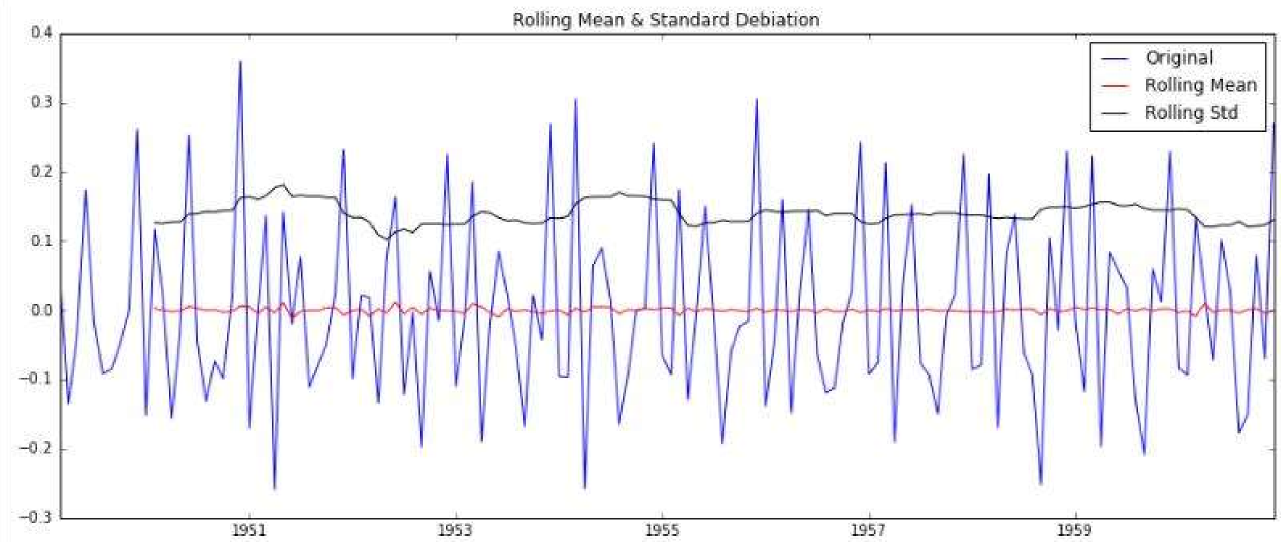
二阶差分

我们继续进行二阶差分

```
#一阶差分：Y(k)=X(k+1)-X(k)
#二阶差分：Y(k)的一阶差分Z(k)=Y(k+1)-Y(k)=X(k+2)-2*X(k+1)+X(k)为此函数的二阶差分
ts_log_diff = ts_log - ts_log.shift(periods=1)
ts_log_diff2 = ts_log_diff - ts_log_diff.shift(periods=1)
plt.plot(ts_log_diff2)
```



```
#二阶差分检验
#可以看到，二阶差分，p值非常小，小于1%，检验统计量也明显小于%1的临界值。因此认定为很平稳
ts_log_diff2.dropna(inplace=True)
adf_test(ts_log_diff2)
```



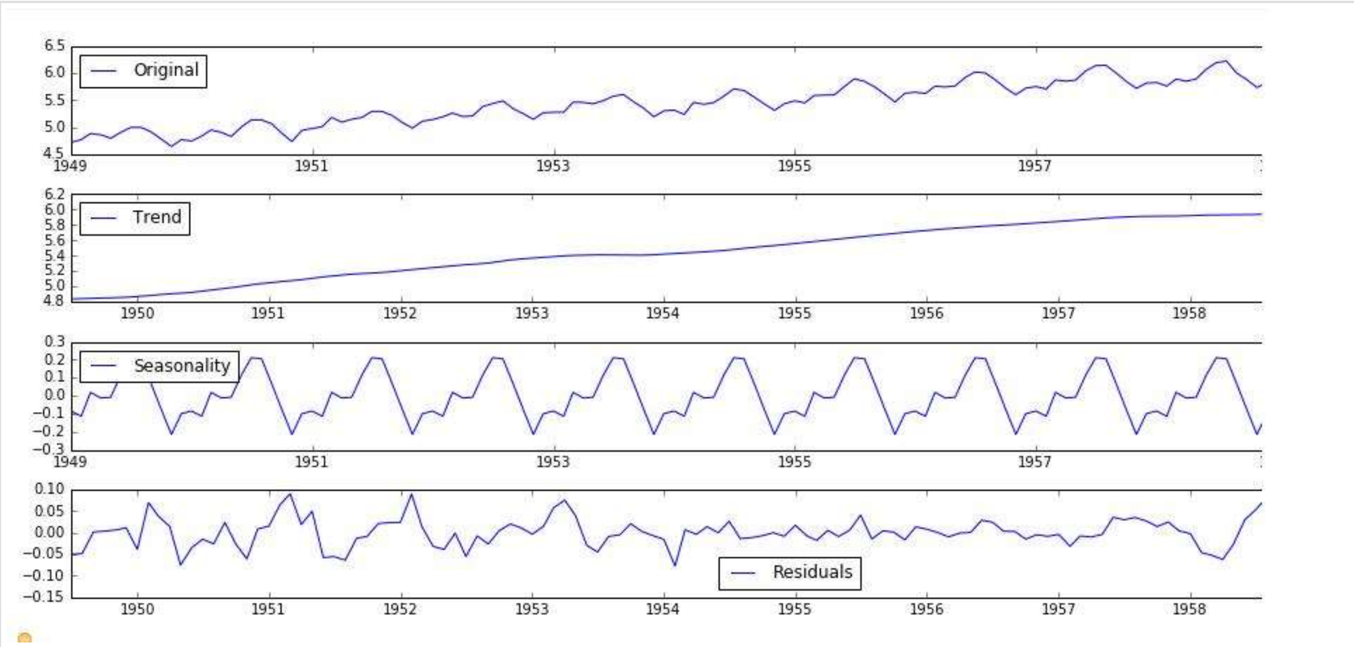

```
Results of Fickey-Fuller Test:
Test Statistic                 -8.196629e+00
p-value                         7.419305e-13
#Lags Used                      1.300000e+01
Number of Observations Used     1.280000e+02
Critical Value (5%)            -2.884398e+00
Critical Value (1%)            -3.482501e+00
Critical Value (10%)           -2.578960e+00
dtype: float64
```

对二阶差分进行adf检验,可以看到，二阶差分，p值非常小，小于1%，检验统计量也明显小于%1的临界值。因此认定为很平稳.

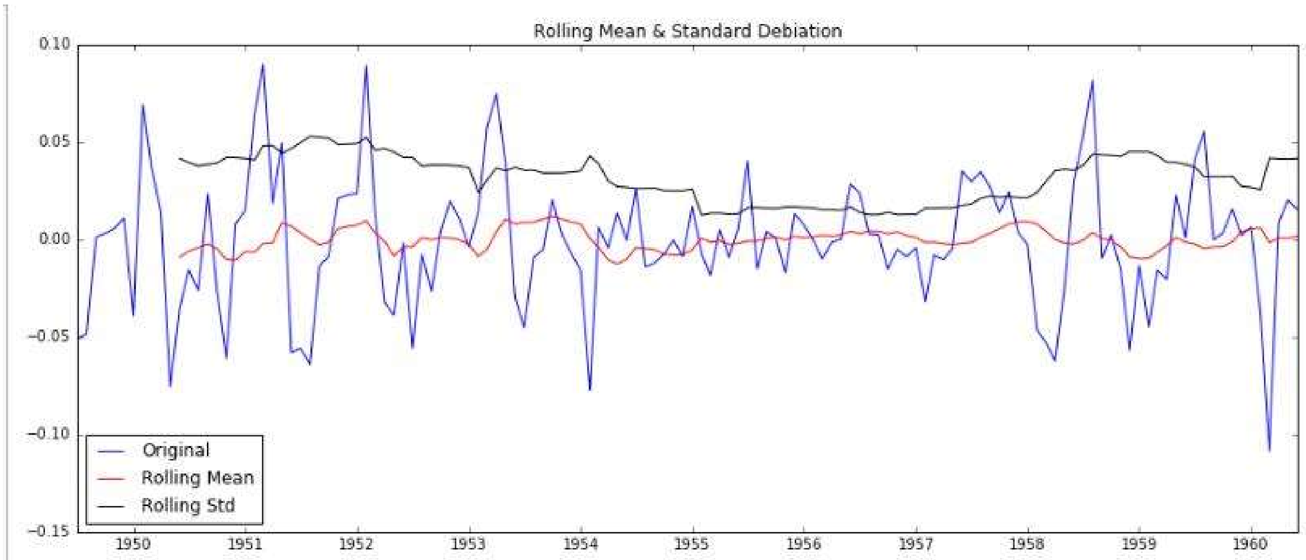
分解

建立有关趋势和季节性的模型，并从模型中删除它们。

```
#时间序列分解
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(ts_log)
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

plt.subplot(411)
plt.plot(ts_log,label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413);
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```




```
#对残差进行ADF检验
#可以发现序列非常平稳
ts_log_decompose = residual
ts_log_decompose.dropna(inplace=True)
adf_test(ts_log_decompose)
```




```
Results of Fickey-Fuller Test:
Test Statistic                -6.332387e+00
p-value                        2.885059e-08
#Lags Used                     9.000000e+00
Number of Observations Used    1.220000e+02
Critical Value (5%)           -2.885538e+00
Critical Value (1%)           -3.485122e+00
Critical Value (10%)          -2.579569e+00
dtype: float64
```

对残差进行ADF检验，可以发现序列非常平稳。

时间序列建模

平稳性检验

平稳性检验的目的是为了判断序列是否平稳，如果不平稳，需要采取一定的措施进行平稳性处理，常见的方法是差分，我们需要选择合适的差分阶数。只要能够通过1%显著性检测，差分阶数就是合理的，我们希望阶数越小越好。

ADF检验

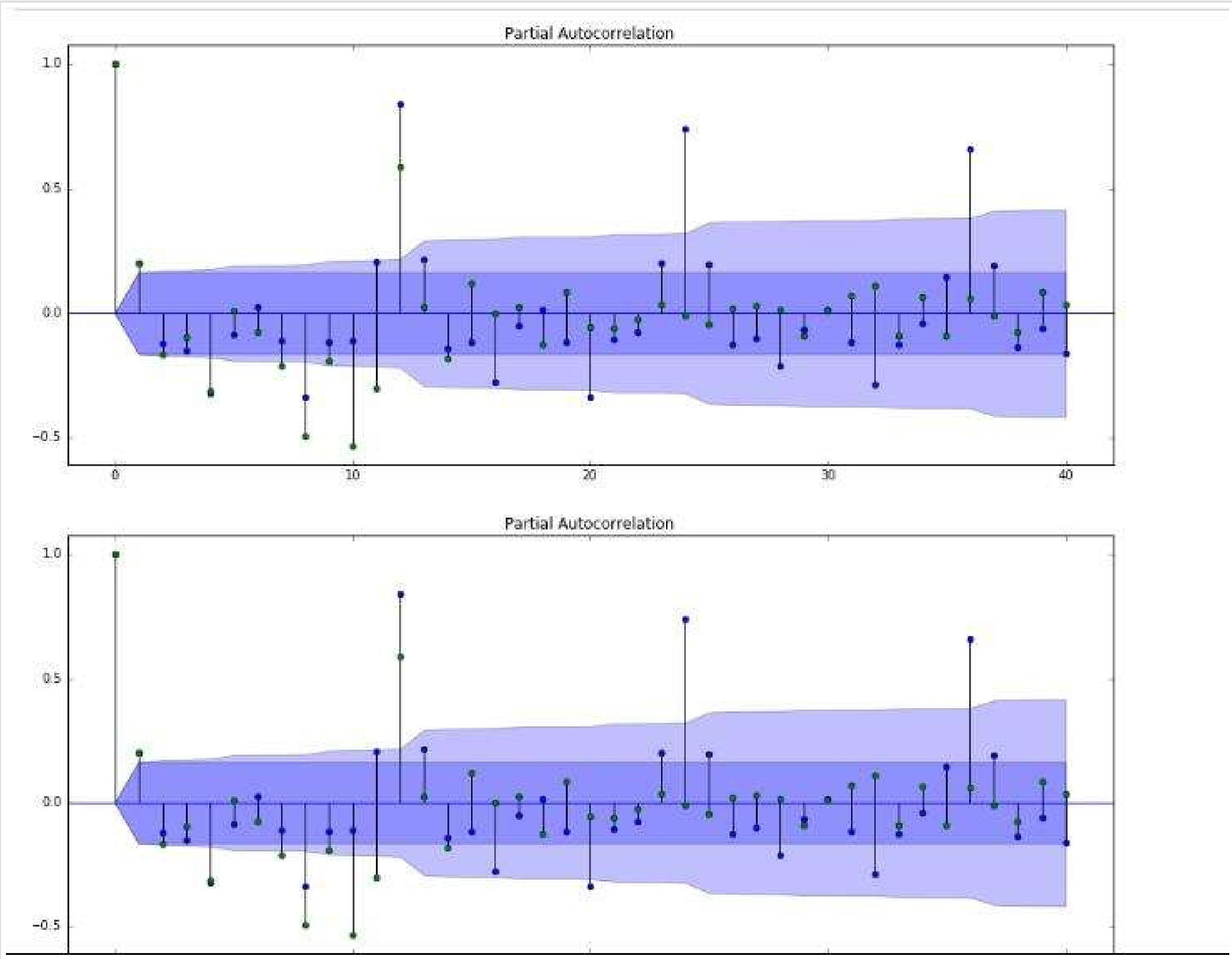
ADF检验前文已经说过，用于判断序列是否平稳。

自相关图和偏自相关图

前面我们对数据进行ADF检验，判断序列是否平稳，这里我们使用自相关图和偏自相关图对数据平稳性再次进行验证，一阶差分如下图：

```
import statsmodels.api as sm
def acf_pacf_plot(ts_log_diff):
    sm.graphics.tsa.plot_acf(ts_log_diff,lags=40) #ARIMA,q
```

```
    sm.graphics.tsa.plot_pacf(ts_log_diff,lags=40) #ARIMA,p
acf_pacf_plot(ts_log_diff) #调用一阶差分
```



可以看出，一阶差分自相关和偏相系数拖尾特点明显。p=1,q=1

参数选择

差分阶数选择

我们发现，ARIMA该开源库，不支持3阶以上的差分。我们唯一的办法是先数据差分好，再传入模型进行建模。但是这样也带来了回退数据到原始序列数据的难度。

```
D:\Anaconda2\lib\site-packages\statsmodels\tsa\arima_model.pyc in __init__(self, endog, order, exog, dates, freq, missin
g)
    972             #NOTE: to make more general, need to address the d == 2 stuff
    973             # in the predict method
--> 974             raise ValueError("d > 2 is not supported")
    975         super(ARIMA, self).__init__(endog, (p, q), exog, dates, freq, missing)
    976         self.k_diff = d

ValueError: d > 2 is not supported
```

这里开发了差分和回退的方法如下:

```
# 差分操作,d代表差分序列，比如[1,1,1]可以代表3阶差分。  [12,1]可以代表第一次差分偏移量是12，第
def diff_ts(ts, d):
    global shift_ts_list
    #  动态预测第二日的值时所需要的差分序列
    global last_data_shift_list #这个序列在恢复过程中需要用到
    shift_ts_list = []
    last_data_shift_list = []
    tmp_ts = ts
    for i in d:
        last_data_shift_list.append(tmp_ts[-i])
        print last_data_shift_list
        shift_ts = tmp_ts.shift(i)
        shift_ts_list.append(shift_ts)
        tmp_ts = tmp_ts - shift_ts
    tmp_ts.dropna(inplace=True)
    return tmp_ts

# 还原操作
def predict_diff_recover(predict_value, d):
    if isinstance(predict_value, float):
        tmp_data = predict_value
        for i in range(len(d)):
            tmp_data = tmp_data + last_data_shift_list[-i-1]
    elif isinstance(predict_value, np.ndarray):
        tmp_data = predict_value[0]
        for i in range(len(d)):
            tmp_data = tmp_data + last_data_shift_list[-i-1]
    else:
        tmp_data = predict_value
        for i in range(len(d)):
            try:
                tmp_data = tmp_data.add(shift_ts_list[-i-1])
            except:
                raise ValueError('What you input is not pd.Series type!')
        tmp_data.dropna(inplace=True)
    return tmp_data # return np.exp(tmp_data)也可以return到最原始，tmp_data是对原始数据取对
```

使用的时候，必须先调用diff_ts进行差分处理，然后进行建模，将预测数据传入predict_diff_recover方法进行还原。

```
d=[1, 1] # 定义差分序列
ts_log = np.log(ts)
diffed_ts = diff_ts(ts_log, d)
```

```
# model = arima_model(diffed_ts)构建模型
predict_ts = model.properModel.predict() #预测，这是对训练数据的预测
diff_recover_ts = predict_diff_recover(predict_ts, d)
log_recover = np.exp(diff_recover_ts) #恢复对数前数据，该数据可以和原始数据ts进行作图对比
```

差分阶数的选择通常越小越好，只要能够使得序列稳定就行。我们可以通过选择不同的阶数，然后进行平稳性检测，选择平稳性表现良好的阶数就行，一般一阶和二阶用的比较多。

p和q选择

差分阶数确定后，我们需要确定p和q. 对于个数不多的时序数据，我们可以通过观察自相关图和偏相关图来进行模型识别，倘若我们要分析的时序数据量较多，例如要预测每只股票的走势，我们就不可能逐个去调参了。这时我们可以依据BIC准则识别模型的p, q值，通常认为BIC值越小的模型相对更优。这里我简单介绍一下BIC准则，它综合考虑了残差大小和自变量的个数，残差越小BIC值越小，自变量个数越多BIC值越大。个人觉得BIC准则就是对模型过拟合设定了一个标准。当然，我们也可以使用AIC指标。

```
#注意这里面使用的ts_log_diff是经过合适阶数的差分之后的数据，上文中提到ARIMA该开源库，不支持3阶
import sys
from statsmodels.tsa.arima_model import ARMA
def _proper_model(ts_log_diff, maxLag):
    best_p = 0
    best_q = 0
    best_bic = sys.maxint
    best_model=None
    for p in np.arange(maxLag):
        for q in np.arange(maxLag):
            model = ARMA(ts_log_diff, order=(p, q))
            try:
                results_ARMA = model.fit(disp=-1)
            except:
                continue
            bic = results_ARMA.bic
            print bic, best_bic
            if bic < best_bic:
                best_p = p
                best_q = q
                best_bic = bic
                best_model = results_ARMA
    return best_p,best_q,best_model
_proper_model(ts_log_diff, 10) #对一阶差分求最优p和q
```

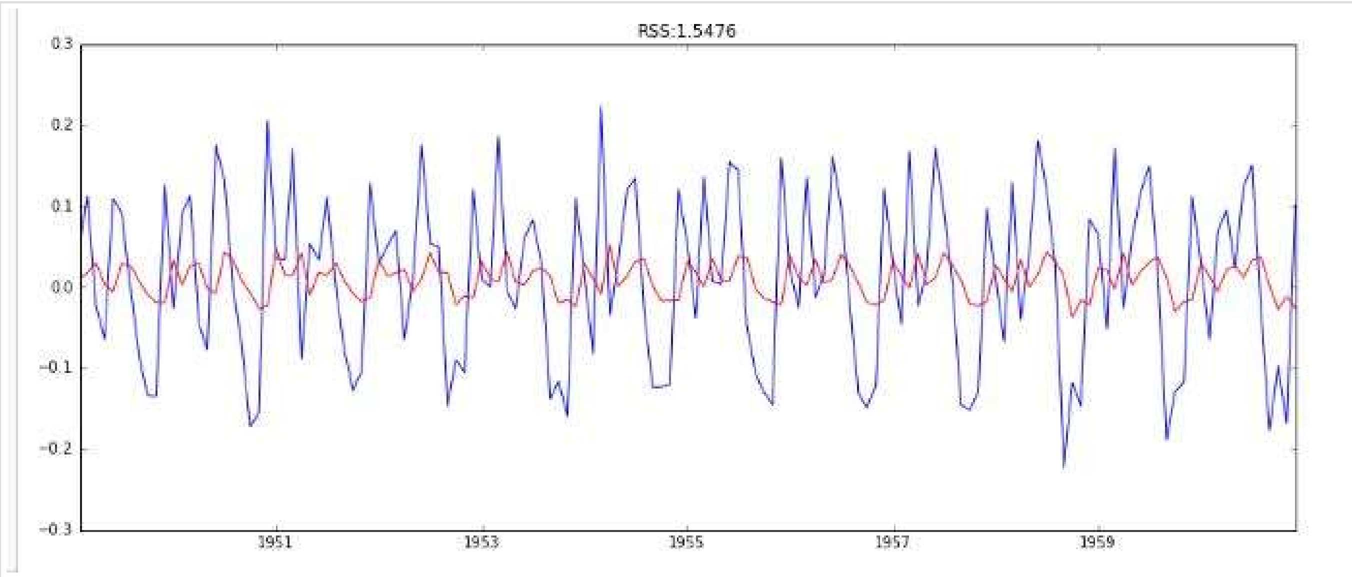
通过上述方法可以得到最优的p和q。

模型

我们使用一阶差分进行构建。

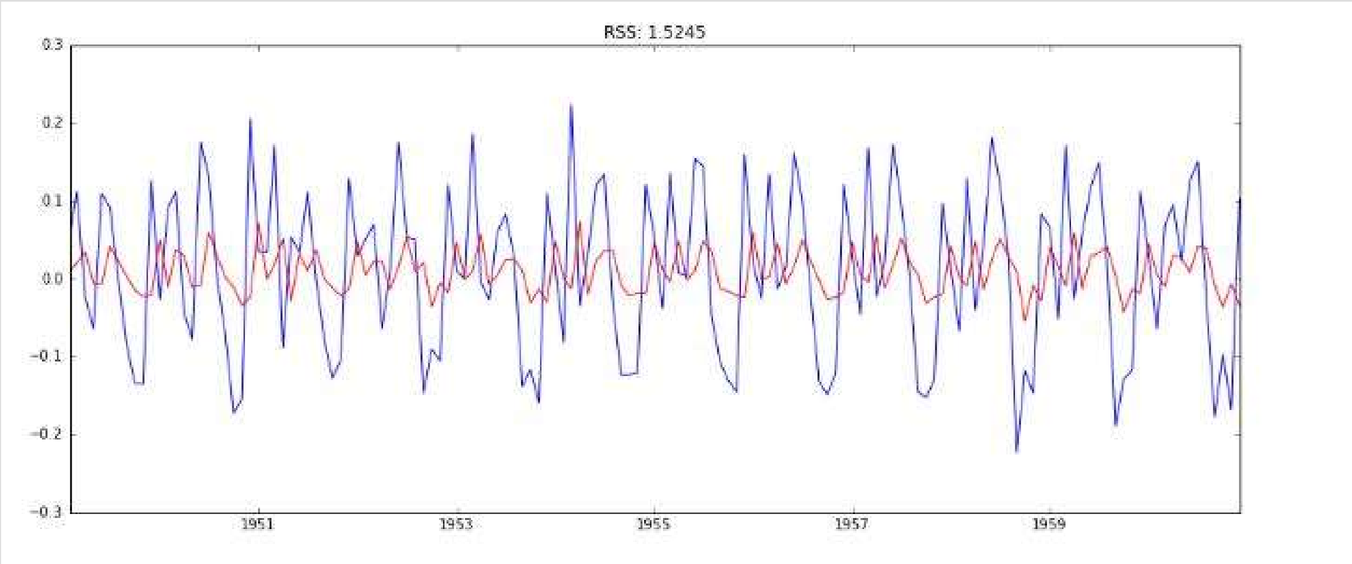
AR(p)模型

```
# AR模型，q=0
#RSS是残差平方和
# disp为-1代表不输出收敛过程的信息，True代表输出
model = ARIMA(ts_log,order=(1,1,0)) #第二个参数代表使用了一阶差分
results_AR = model.fit(disp=-1)
plt.plot(ts_log_diff)
plt.plot(results_AR.fittedvalues, color='red') #红色线代表预测值
plt.title('RSS:%.4f' % sum((results_AR.fittedvalues-ts_log_diff)**2))#残差平方和
```



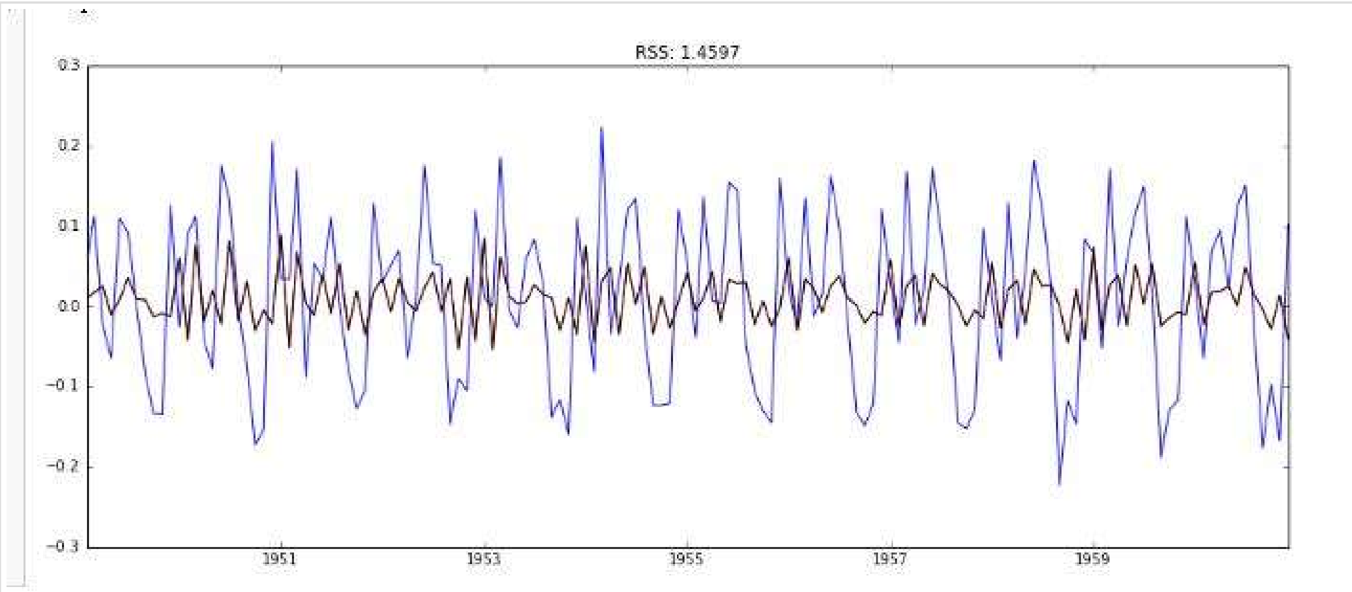
MA(q)模型

```
#MA模型 p=0
model = ARIMA(ts_log,order=(0,1,1))
results_MA = model.fit(disp=-1)
plt.plot(ts_log_diff)
plt.plot(results_MA.fittedvalues, color='red')
plt.title('RSS: %.4f'% sum((results_MA.fittedvalues-ts_log_diff)**2))
```



ARIMA(p,q)模型

```
#ARIMA
model = ARIMA(ts_log, order=(1, 1, 1))
results_ARIMA = model.fit(disp=-1)  #不展示信息
plt.plot(ts_log_diff)
plt.plot(results_ARIMA.fittedvalues, color='red')#和下面这句结果一样
plt.plot(results_ARIMA.predict(), color='black')#predict得到的就是fittedvalues，只是差分的
plt.title('RSS: %.4f'% sum((results_ARIMA.fittedvalues-ts_log_diff)**2))
```

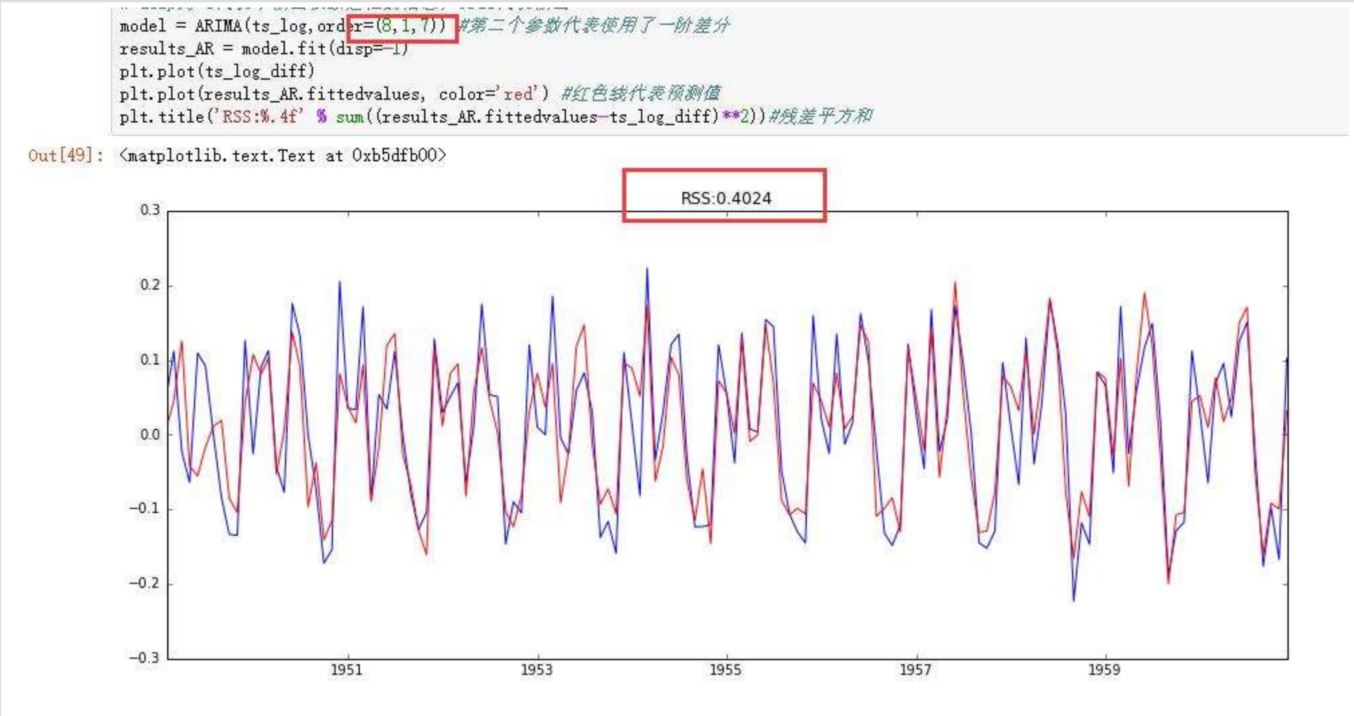


可以发现，ARIMA在AR和MA基础上，RSS有所减少，故模型有所提高。

我们使用上文中提高的p和q选择方法，对一阶差分结果进行p和q选择。

```
proper_model(ts_log_diff, 9)
# 输出最优结果如下:
(8, 7, <statsmodels.tsa.arima_model.ARMAResultsWrapper at 0xb4e2898>)
```

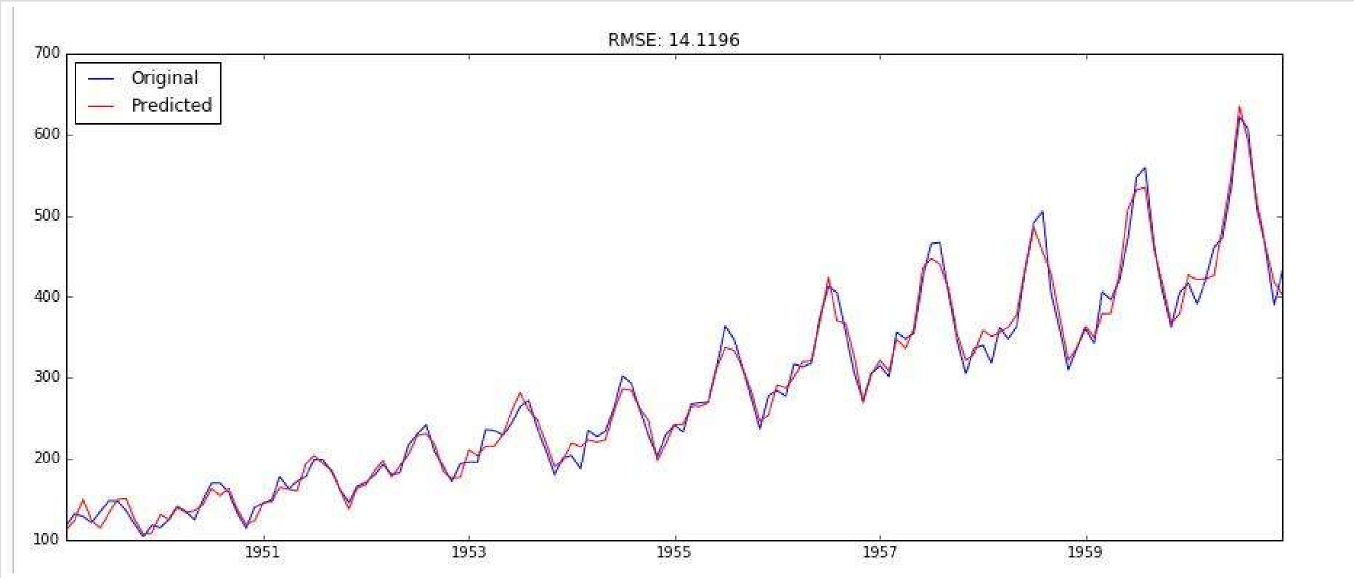
故可以使用p=8,q=7再次进行测试。得到如下结果:



可以发现，残差平方和RSS已经优化到0.40了。

数据还原

```
ts_log_diff = diff_ts(ts_log, d=[1])#调用差分方法，方便后续还原
model = ARIMA(ts_log, order=(8, 1, 7))  #建模
results_ARIMA = model.fit(disp=-1)  #fit
predict_ts = model.predict() #对训练数据进行预测

#还原
diff_recover_ts = predict_diff_recover(predict_ts, d=[1])#恢复数据
log_recover = np.exp(diff_recover_ts)#还原对数前数据

#绘图
#ts = ts[log_recover.index]#排除空的数据
plt.plot(ts,color="blue",label='Original')
plt.plot(log_recover,color='red',label='Predicted')
plt.legend(loc='best')
plt.title('RMSE: %.4f'% np.sqrt(sum((log_recover-ts)**2)/len(ts)))#RMSE,残差平方和开根号,
```



预测未来走势

使用forecast进行预测，参数为预测值个数。这个得到的就是进行自动差分还原后的数据，因为我们建立模型的时候ARIMA(p,1,q),第二个参数就是差分阶数，forecast会将结果恢复回差分前的数据，因此我们直接将结果通过np.exp来恢复到最原始数据即可。但是ARIMA只支持最多2阶差分，因此我们可以使用ARMA模型，将我们手动差分完的数据传入。最后预测的时候，使用我们自定义的差分还原方法，对预测得到的值进行差分还原。

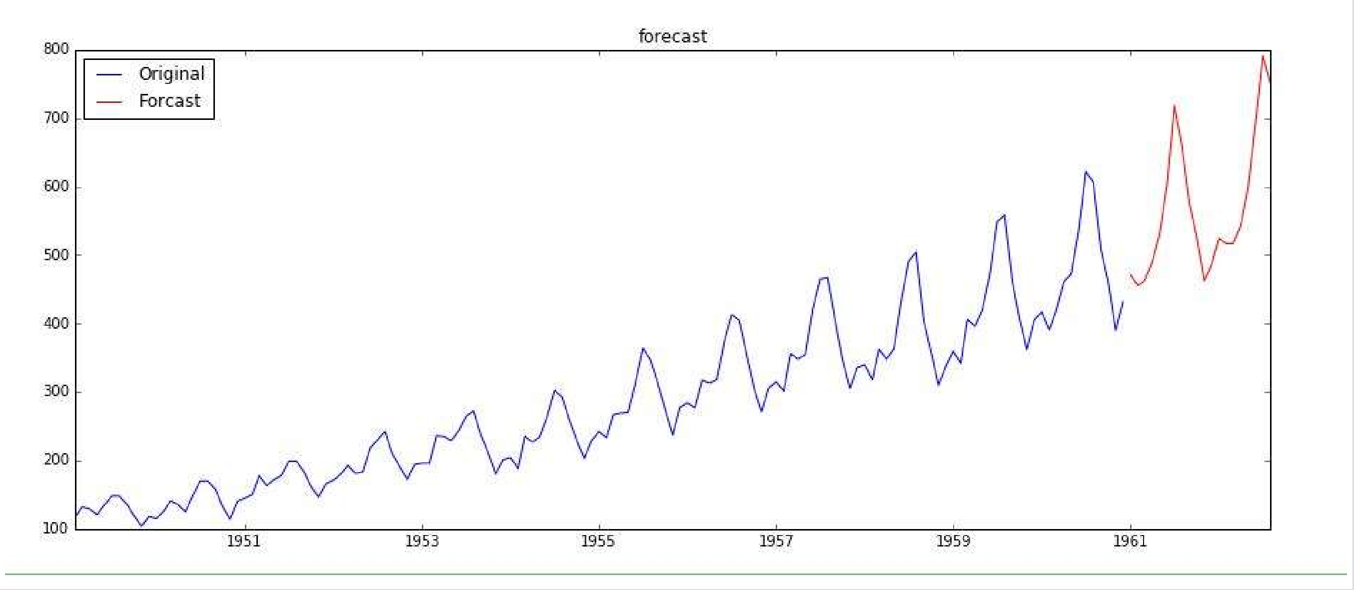
```
# forecast方法会自动进行差分还原，当然仅限于支持的1阶和2阶差分
forecast_n = 12 #预测未来12个月走势
forecast_ARIMA_log = results_ARIMA.forecast(forecast_n)
forecast_ARIMA_log = forecast_ARIMA_log[0]
print forecast_ARIMA_log

##如下是差分还原后的数据：
[6.15487901  6.12150398  6.13788758  6.19511156  6.27419885  6.40259838
  6.57706431  6.49128697  6.35429917  6.2679321   6.13597822  6.18507789
  6.26245365  6.24740859  6.24775066  6.29778253  6.3935587   6.54015482
  6.67409705  6.62124844]
```

我们希望能够将预测的数据和原来的数据绘制在一起，为了实现这一目的，我们需要增加数据索引，使用开源库arrow:

```
#定义获取连续时间，start是起始时间，limit是连续的天数,level可以是day,month,year
import arrow
def get_date_range(start, limit, level='month',format='YYYY-MM-DD'):
    start = arrow.get(start, format)
```

```
    result=(list(map(lambda dt: dt.format(format) , arrow.Arrow.range(level, start,
    dateparse2 = lambda dates:pd.datetime.strptime(dates,'%Y-%m-%d')
    return map(dateparse2, result)
```

```
# 预测从1961-01-01开始，也就是我们训练数据最后一个数据的后一个日期
new_index = get_date_range('1961-01-01', forecast_n)
forecast_ARIMA_log = pd.Series(forecast_ARIMA_log, copy=True, index=new_index)
print forecast_ARIMA_log.head()

# 直接取指数，即可恢复至原数据
forecast_ARIMA = np.exp(forecast_ARIMA_log)
print forecast_ARIMA
plt.plot(ts,label='Original',color='blue')
plt.plot(forecast_ARIMA, label='Forcast',color='red')
plt.legend(loc='best')
plt.title('forecast')
```



遗留问题：

如果直接将差分处理的结果传入ARMA模型，再进行forecast预测，如何对预测的结果进行还原至原始序列？

参考

[Complete guide to create a Time Series Forecast (with Codes in Python)]

[时间序列分析]

坚持原创技术分享，您的支持将鼓励我继续创作！

赏

统计学 # 时间序列 # 人工智能 # ARIMA

ARIMA时间序列模型(二)

生成算法