时间序列分析

这篇属于补充的内容，最近在参加天池口碑的客流量预测比赛，从很多人的结果看出，对于时间序列的预测问题，一般的机器学习算法并没有很占优势，RNN类的算法训练成本又太高，往往不可取，倒是传统的时间序列模型的效果较好，这里对此，进行一定的学习。内容来自互联网。

一、

顾名思义，时间序列是时间间隔不变的情况下收集的时间点集合。这些集合被分析用来了解长期发展趋势，为了预测未来或者表现分析的其他形式。但是是什么令时间序列与常见的回归问题的不同？

有两个原因：

1、时间序列是跟时间有关的。所以基于线性回归模型的假设：观察结果是独立的在这种情况下是不成立的。

2、随着上升或者下降的趋势，更多的时间序列出现季节性趋势的形式，如：特定时间框架的具体变化。即：如果你看到羊毛夹克的销售上升，你就一定会在冬季做更多销售。

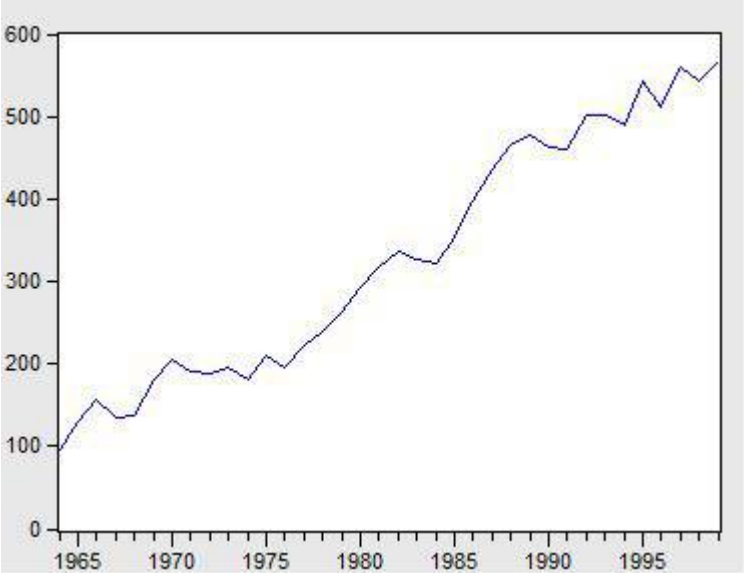
常用的时间序列模型有AR模型、MA模型、ARMA模型和ARIMA模型等。

一、时间序列的预处理

拿到一个观察值序列之后，首先要对它的平稳性和纯随机性进行检验，这两个重要的检验称为序列的预处理。根据检验的结果可以将序列分为不同的类型，对不同类型的序列我们会采用不同的分析方法。

先说下什么是平稳，平稳就是围绕着一个常数上下波动且波动范围有限，即有常数均值和常数方差。如果有明显的趋势或周期性，那它通常不是平稳序列。序列平稳不平稳，一般采用三种方法检验：

（1）时序图检验



看看上面这个图，很明显的增长趋势，不平稳。

（2）自相关系数和偏相关系数

还以上面的序列为例：用SPSS得到自相关和偏相关图。

@马春杰杰你好： 想问下，你更改了源码没？可以输出每一类的recall,AP,以及mAP了吗？我也在做这步。...

--flysnow_88

5. Re:CV：object detection(YOLO)

@马春杰杰recall和mAP都是分类任务的指标，只是需要针对多标签任务进行一些修改，具体的，百度即可知道...

--AI-ML-DL

阅读排行榜

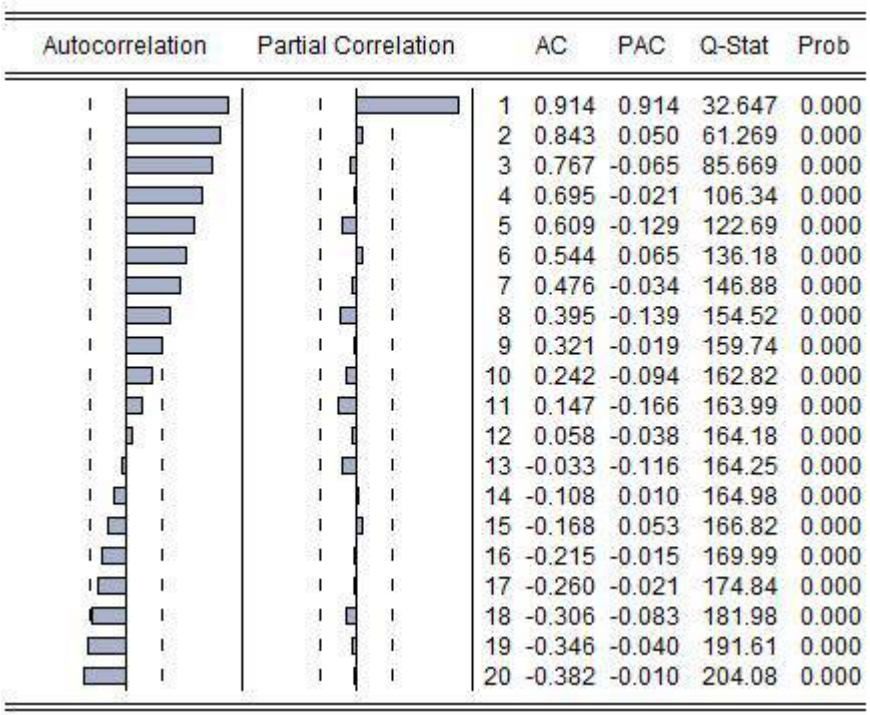
1. LSTM与GRU结构(8609)
2. 聚类算法（clustering）(3630)
3. CV：object recognition(ZFNet)(3615)
4. 生成对抗式网络(2711)
5. CV：image caption(Show, Attend and Tell: Neural Image Caption Generation with Visual Attention)(1701)

评论排行榜

1. CV：object detection(YOLO)(5)
2. 时间序列分析(1)
3. 聚类算法（clustering）(1)
4. 生成对抗式网络(1)

推荐排行榜

1. 时间序列分析(2)
2. CV：object recognition(ZFNet)(1)
3. LSTM与GRU结构(1)
4. 聚类算法（clustering）(1)



	AC	PAC	Q-Stat	Prob
1	0.914	0.914	32.647	0.000
2	0.843	0.050	61.269	0.000
3	0.767	-0.065	85.669	0.000
4	0.695	-0.021	106.34	0.000
5	0.609	-0.129	122.69	0.000
6	0.544	0.065	136.18	0.000
7	0.476	-0.034	146.88	0.000
8	0.395	-0.139	154.52	0.000
9	0.321	-0.019	159.74	0.000
10	0.242	-0.094	162.82	0.000
11	0.147	-0.166	163.99	0.000
12	0.058	-0.038	164.18	0.000
13	-0.033	-0.116	164.25	0.000
14	-0.108	0.010	164.98	0.000
15	-0.168	0.053	166.82	0.000
16	-0.215	-0.015	169.99	0.000
17	-0.260	-0.021	174.84	0.000
18	-0.306	-0.083	181.98	0.000
19	-0.346	-0.040	191.61	0.000
20	-0.382	-0.010	204.08	0.000

分析：左边第一个为自相关图（Autocorrelation），第二个偏相关图(Partial Correlation)。

平稳的序列的自相关图和偏相关图要么拖尾，要么是截尾。截尾就是在某阶之后，系数都为 0 ，怎么理解呢，看上面偏相关的图，当阶数为 1 的时候，系数值还是很大， 0.914. 二阶长的时候突然就变成了 0.050. 后面的值都很小，认为是趋于 0 ，这种状况就是截尾。什么是拖尾，拖尾就是有一个缓慢衰减的趋势，但是不都为 0 。

自相关图既不是拖尾也不是截尾。以上的图的自相关是一个三角对称的形式，这种趋势是单调趋势的典型图形，说明这个序列不是平稳序列。

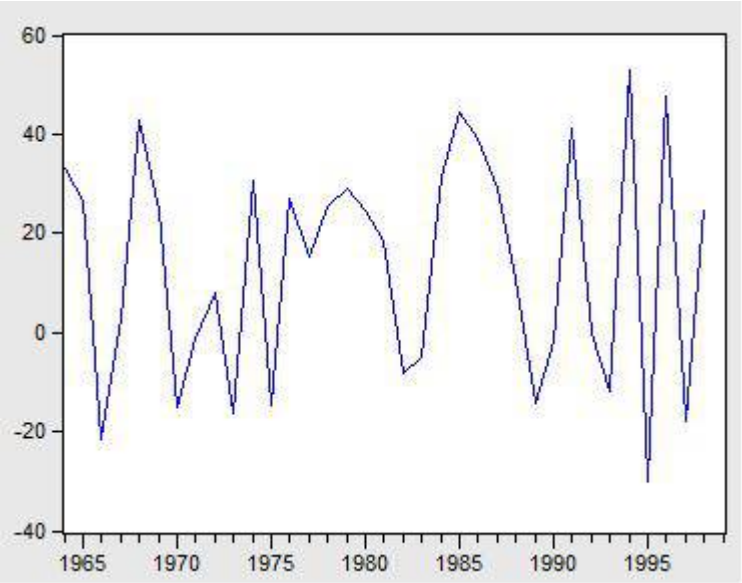
（3）单位根检验

单位根检验是指检验序列中是否存在单位根，如果存在单位根就是非平稳时间序列。

不平稳，怎么办？

答案是差分，转换为平稳序列。什么是差分？一阶差分指原序列值相距一期的两个序列值之间的减法运算；k阶差分就是相距k期的两个序列值之间相减。如果一个时间序列经过差分运算后具有平稳性，则该序列为差分平稳序列，可以使用ARIMA模型进行分析。

还是上面那个序列，两种方法都证明他是不靠谱的，不平稳的。确定不平稳后，依次进行1阶、2阶、3阶...差分，直到平稳为止。先来个一阶差分，上图:



从图上看，一阶差分的效果不错，看着是平稳的。

平稳性检验过后，下一步是纯随机性检验。

对于纯随机序列，又称白噪声序列，序列的各项数值之间没有任何相关关系，序列在进行完全无序的随机波动，可以终止对该序列的分析。白噪声序列是没有信息可提取的平稳序列。

对于平稳非白噪声序列，它的均值和方差是常数。通常是建立一个线性模型来拟合该序列的发展，借此提取该序列的有用信息。ARMA模型是最常用的平稳序列拟合模型。

二、平稳时间序列建模

某个时间序列经过预处理，被判定为平稳非白噪声序列，就可以进行时间序列建模。

建模步骤：

（1）计算出该序列的自相关系数（ACF）和偏相关系数（PACF）；

（2）模型识别，也称模型定阶。根据系数情况从AR(p)模型、MA(q)模型、ARMA(p，q)模型、ARIMA（p，d，q）模型中选择合适模型，其中p为自回归项，d为差分阶数，q为移动平均项数。

下面是平稳序列的模型选择：

自相关系数（ACF）	偏相关系数（PACF）	选择模型
拖尾	p阶截尾	AR(p)
q阶截尾	拖尾	MA(q)
p阶拖尾	q阶拖尾	ARMA(p，q)

ARIMA 是 ARMA 算法的扩展版，用法类似 。

（3）估计模型中的未知参数的值并对参数进行检验；

（4）模型检验；

（5）模型优化；

（6）模型应用：进行短期预测。

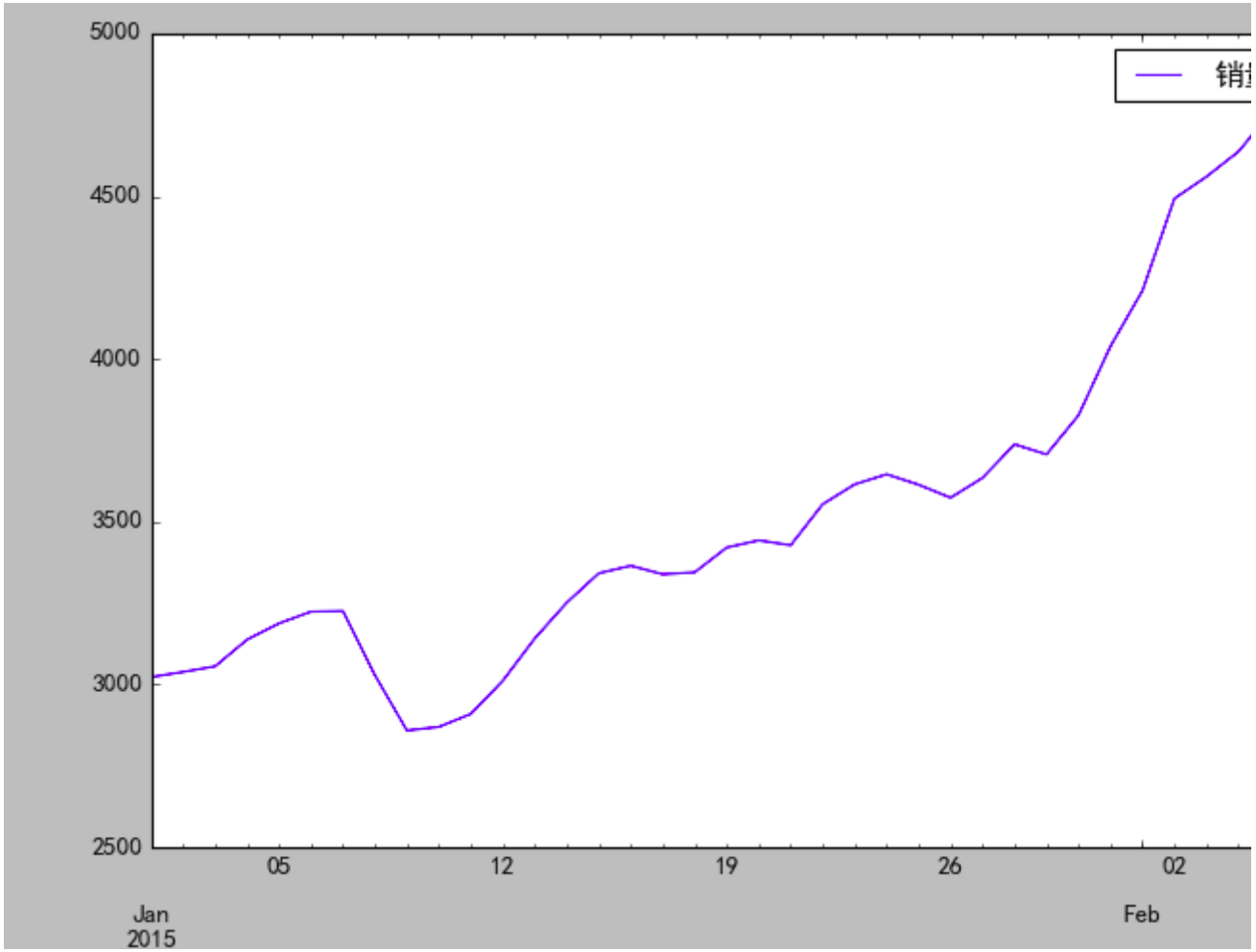
三、python实例操作

以下为某店铺2015/1/1~2015/2/6的销售数据,以此建模预测2015/2/7~2015/2/11的销售数据。

	A	B
1	日期	销量
2	2015/1/1	3023
3	2015/1/2	3039
4	2015/1/3	3056
5	2015/1/4	3138
6	2015/1/5	3188
7	2015/1/6	3224
8	2015/1/7	3226
9	2015/1/8	3029
10	2015/1/9	2859
11	2015/1/10	2870
12	2015/1/11	2910
13	2015/1/12	3012
14	2015/1/13	3142
15	2015/1/14	3252
16	2015/1/15	3342
17	2015/1/16	3365
18	2015/1/17	3339
19	2015/1/18	3345
20	2015/1/19	3421
21	2015/1/20	3443
22	2015/1/21	3428
23	2015/1/22	3554

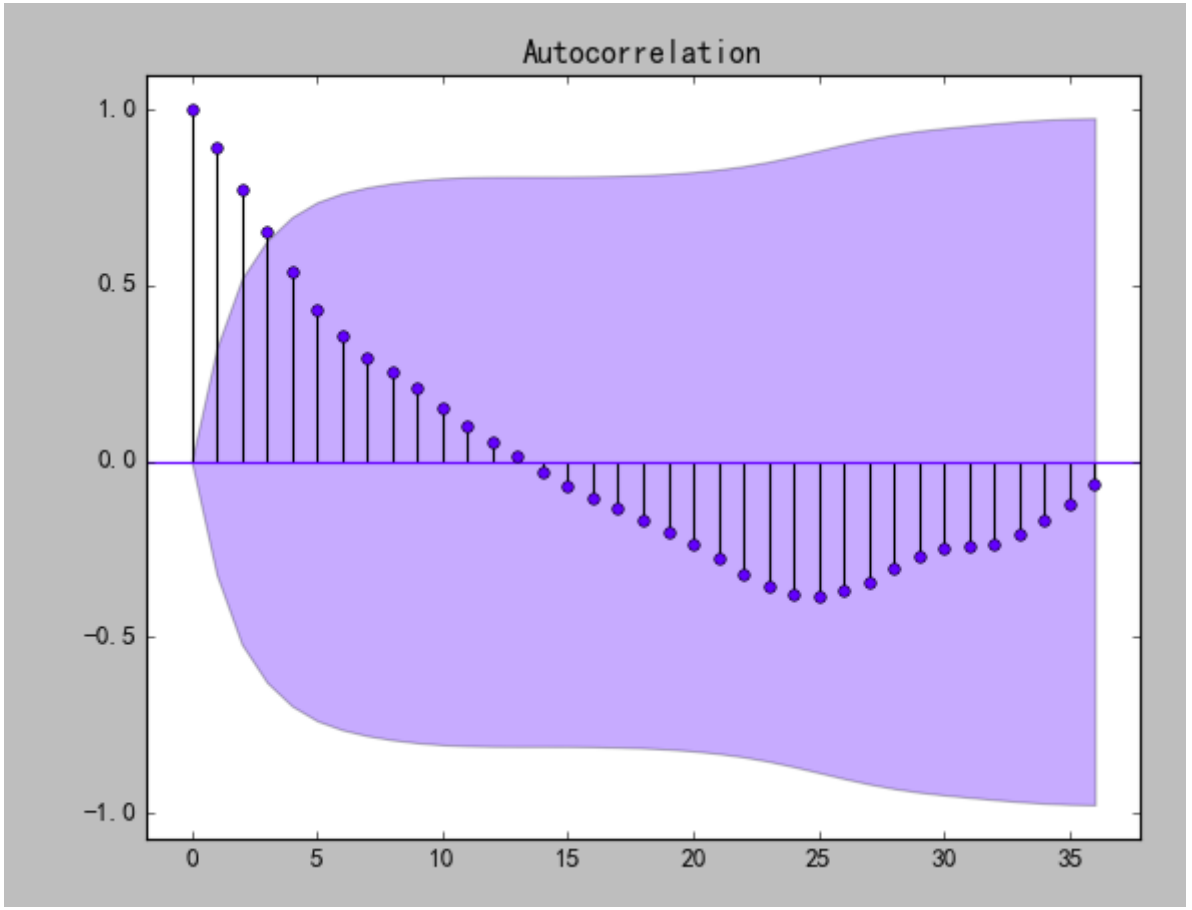
```
#-*- coding: utf-8 -*-
#arima时序模型
```

```
import pandas as pd

#参数初始化
discfile = 'E:/destop/text/arima_data.xls'
forecastnum = 5

#读取数据，指定日期列为指标，Pandas自动将“日期”列识别为Datetime格式
data = pd.read_excel(discfile, index_col = u'日期')

#时序图
import matplotlib.pyplot as plt

#用来正常显示中文标签

plt.rcParams['font.sans-serif'] = ['SimHei']


#用来正常显示负号

plt.rcParams['axes.unicode_minus'] = False
data.plot()
plt.show()
```



#自相关图

```
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(data).show()
```



#平稳性检测

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

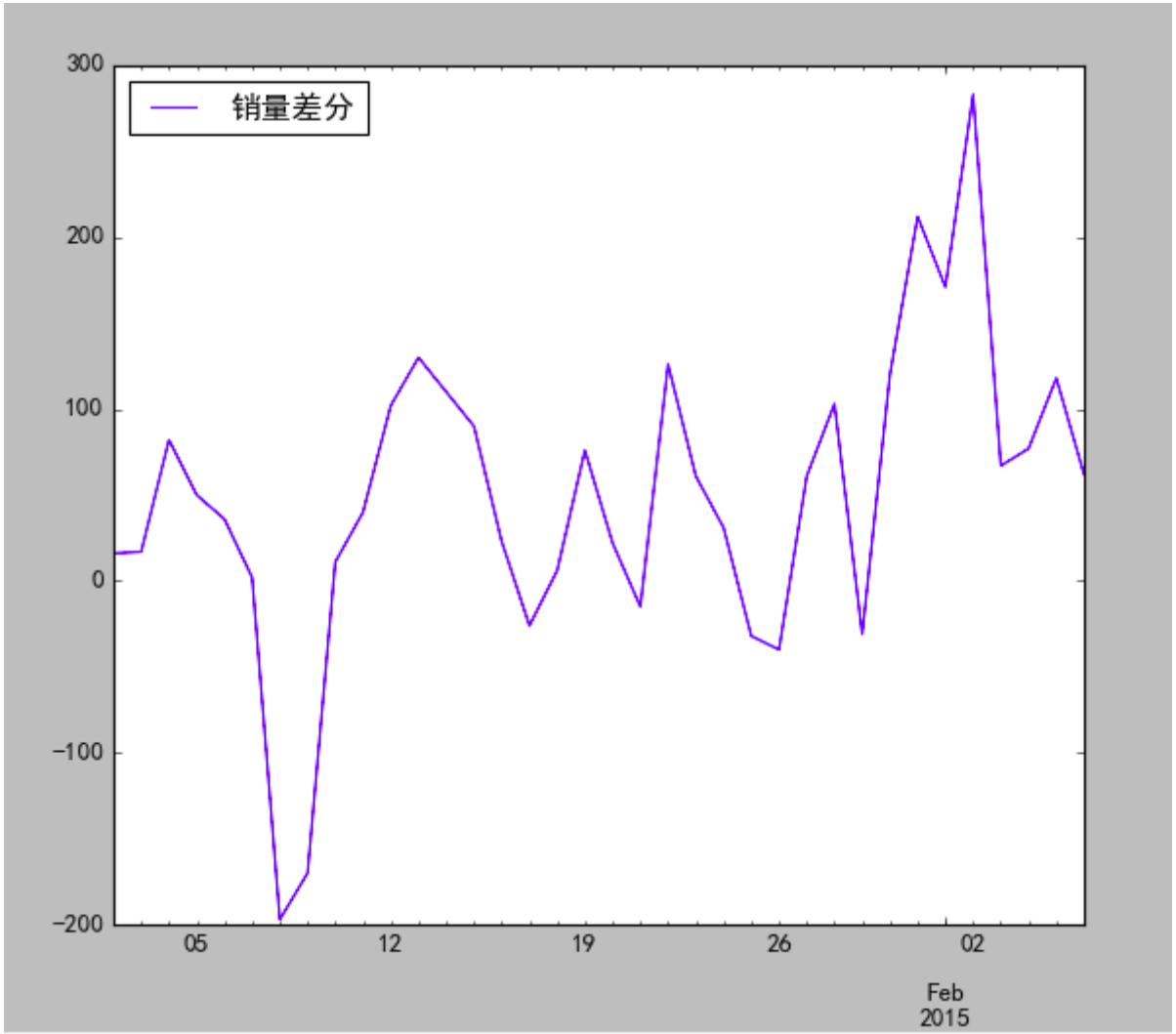
#返回值依次为adf、pvalue、usedlag、nobs、critical values、icbest、regresults、resstore

原始序列的单位根（adf）检验				
adf	cValue			p值

	1%	5%	10%	
1.81	-3.7112	-2.9812	-2.6301	0.9984

Pdf值大于三个水平值，p值显著大于0.05，该序列为非平稳序列。

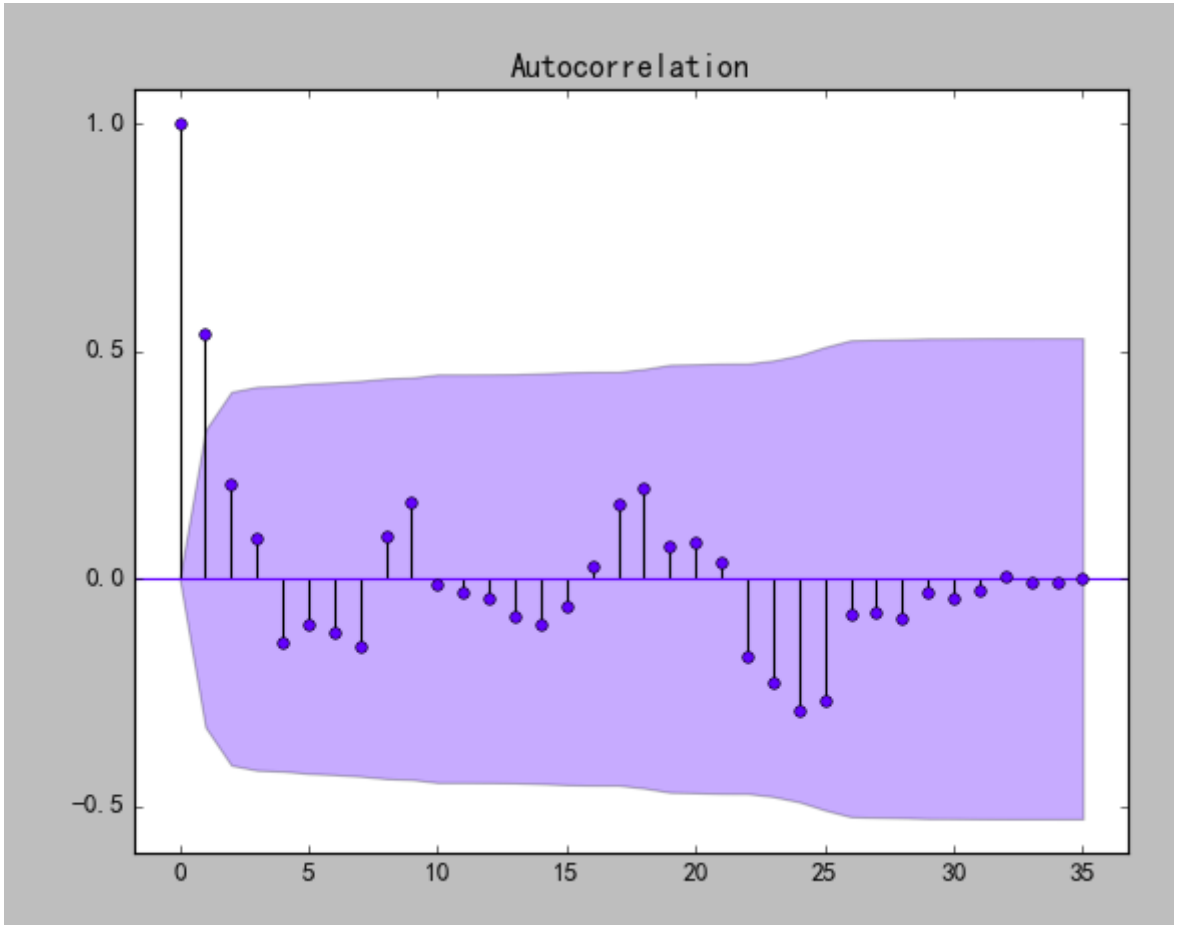
```
#差分后的结果
D_data = data.diff().dropna()
D_data.columns = [u'销量差分']

#时序图

D_data.plot()
plt.show()
```



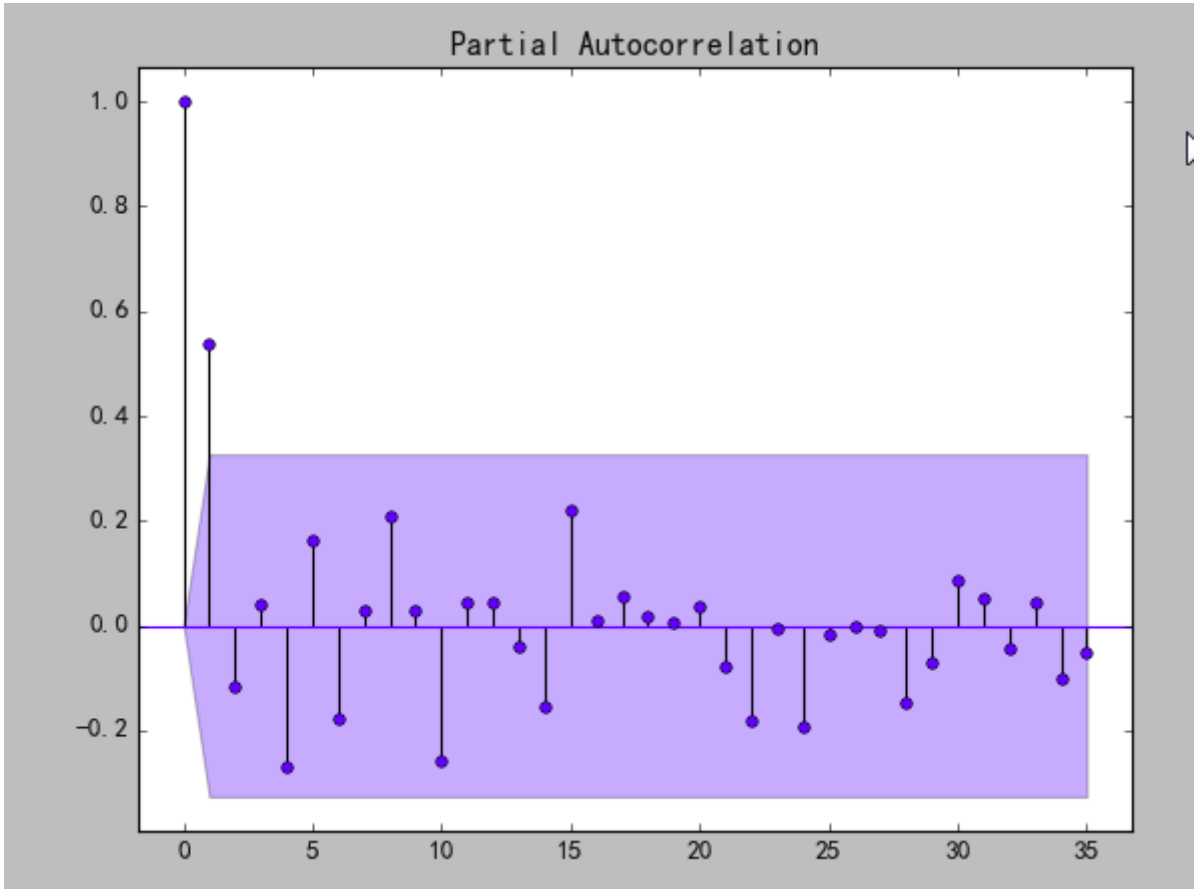
#自相关图

plot_acf(D_data).show()

plt.show()




```
from statsmodels.graphics.tsaplots import plot_pacf

#偏自相关图
plot_pacf(D_data).show()
```



#平稳性检测

print(u'差分序列的ADF检验结果为：', ADF(D_data[u'销量差分']))

一阶差分后序列的单位根（adf）检验				
adf	cValue			p值
	1%	5%	10%	

-3.15	-3.6327	-2.9485	-2.6130	0.0227

Pdf值小于两个水平值，p值显著小于0.05，一阶差分后序列为平稳序列。

```
#白噪声检验
from statsmodels.stats.diagnostic import acorr_ljungbox

#返回统计量和p值

print(u'差分序列的白噪声检验结果为：', acorr_ljungbox(D_data, lags=1))
```

一阶差分后序列的白噪声检验	
stat	P值
11.304	0.007734

P值小于0.05，所以一阶差分后的序列为平稳非白噪声序列。

```
from statsmodels.tsa.arima_model import ARIMA
#定阶

#一般阶数不超过length/10

pmax = int(len(D_data)/10)
```

```
#一般阶数不超过length/10
qmax = int(len(D_data)/10)

#bic矩阵
bic_matrix = []
for p in range(pmax+1):
  tmp = []
  for q in range(qmax+1):

#存在部分报错，所以用try来跳过报错。
    try:
      tmp.append(ARIMA(data, (p,1,q)).fit().bic)
    except:
      tmp.append(None)
  bic_matrix.append(tmp)

#从中可以找出最小值
bic_matrix = pd.DataFrame(bic_matrix)

#先用stack展平，然后用idxmin找出最小值位置。
p,q = bic_matrix.stack().idxmin()

print(u'BIC最小的p值和q值为：%s、%s' %(p,q))
```

取BIC信息量达到最小的模型阶数，结果p为0，q为1，定阶完成。

```
 #建立ARIMA(0, 1, 1)模型
model = ARIMA(data, (p,1,q)).fit()
#给出一份模型报告
```

```
model.summary2()
```

#作为期5天的预测，返回预测结果、标准误差、置信区间。

```
model.forecast(5)
```

最终模型预测值如下:

2015/2/7	2015/2/8	2015/2/9	2015/2/10	2015/2/11
4874.0	4923.9	4973.9	5023.8	5073.8

利用模型向前预测的时间越长，预测的误差将会越大，这是时间预测的典型特点。

参数检验如下：

	Coef.	Std.Err.	t	P值
const	49.956	20.139	2.4806	0.0182
ma.L1.D.销量	0.671	0.1648	4.0712	0.0003

从检验结果p值来看，建立的模型效果良好。

二、

什么是时间序列

时间序列简单的说就是各时间点上形成的数值序列，时间序列分析就是通过观察历史数据预测未来的值。在这里需要强调一点的是，时间序列分析并不是关于时间的回归，它主要是研究自身的变化规律的（这里不考虑含外生变量的时间序列）。

为什么用python

用两个字总结“情怀”，爱屋及乌，个人比较喜欢python，就用python撸了。能做时间序列的软件很多，SAS、R、SPSS、Eviews甚至matlab等等，实际工作中应用得比较多的应该还是SAS和R，前者推荐王燕写的《应用时间序列分析》，后者推荐“基于R语言的时间序列建模完整教程”这篇博文（翻译版）。python作为科学计算的利器，当然也有相关分析的包:statsmodels中tsa模块，当然这个包和SAS、R是比不了，但是python有另一个神器：pandas！pandas在时间序列上的应用，能简化我们很多的工作。

环境配置

python推荐直接装Anaconda，它集成了许多科学计算包，有一些包自己手动去装还是挺费劲的。statsmodels需要自己去安装，这里我推荐使用0.6的稳定版，0.7及其以上的版本能在github上找到，该版本在安装时会用C编译好，所以修改底层的一些代码将不会起作用。

时间序列分析

1.基本模型

自回归移动平均模型(ARMA(p，q))是时间序列中最为重要的模型之一，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，其公式如下：

$$z_t = \varphi_1 z_{t-1} + \varphi_2 z_{t-2} + \ldots + \varphi_p z_{t-p} + a_t - \theta_1 a_{t-1} - \ldots - \theta_q a_{t-q}$$

模型的简化形式为：$\varphi_p(B) z_t = \theta_q(B) a_t$

其中：$\varphi_p(B) = 1 - \varphi_1 B - \varphi_2 B^2 - \ldots - \varphi_p B^p$

$\theta_q(B) = 1 - \theta_1 B - \theta_2 B^2 - \ldots - \theta_q B^q$

依据模型的形式、特性及自相关和偏自相关函数的特征，总结如下：

	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	φ(B)=0的根在单位圆外	无	φ(B)=0的根在单位圆外
可逆性条件	无	θ(B)=0的根在单位圆外	θ(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

在时间序列中，ARIMA模型是在ARMA模型的基础上多了差分的操作。

2.pandas时间序列操作

大熊猫真的很可爱，这里简单介绍一下它在时间序列上的可爱之处。和许多时间序列分析一样，本文同样使用航空乘客数据（AirPassengers.csv）作为样例。

数据读取：

```
# -*- coding:utf-8 -*-
import numpy as np
import pandas as pd
from datetime import datetime
import matplotlib.pylab as plt
# 读取数据，pd.read_csv默认生成DataFrame对象，需将其转换成Series对象
df = pd.read_csv('AirPassengers.csv', encoding='utf-8', index_col='date')
df.index = pd.to_datetime(df.index)  # 将字符串索引转换成时间索引
ts = df['x']  # 生成pd.Series对象
# 查看数据格式
ts.head()
ts.head().index
```

```
date
1949-01-01  112
1949-02-01  118
1949-03-01  132
1949-04-01  129
1949-05-01  121
DatetimeIndex(['1949-01-01', '1949-02-01', '1949-03-01', '1949-04-01'
               '1949-05-01'],
              dtype='datetime64[ns]', name=u'date', freq=None)
```

查看某日的值既可以使用字符串作为索引，又可以直接使用时间对象作为索引

```
ts['1949-01-01']
ts[datetime(1949,1,1)]
```

两者的返回值都是第一个序列值：112

如果要查看某一年的数据，pandas也能非常方便的实现

```
ts['1949']
```

```
date
1949-01-01    112
1949-02-01    118
1949-03-01    132
1949-04-01    129
1949-05-01    121
1949-06-01    135
1949-07-01    148
1949-08-01    148
1949-09-01    136
1949-10-01    119
1949-11-01    104
1949-12-01    118
Name: x, dtype: int64
```

切片操作：

```
ts['1949-1' : '1949-6']
```

```
date
1949-01-01    112
1949-02-01    118
1949-03-01    132
1949-04-01    129
1949-05-01    121
1949-06-01    135
Name: x, dtype: int64
```

注意时间索引的切片操作起点和尾部都是包含的，这点与数值索引有所不同

pandas还有很多方便的时间序列函数，在后面的实际应用中在进行说明。

3. 平稳性检验

我们知道序列平稳性是进行时间序列分析的前提条件，很多人都会有疑问，为什么要满足平稳性的要求呢？在大数定理和中心定理中要求样本同分布（这里同分布等价于时间序列中的平稳性），而我们的建模过程中有很多都是建立在大数定理和中心极限定理的前提条件下的，如果它不满足，得到的许多结论都是不可靠的。以虚假回归为例，当响应变量和输入变量都平稳时，我们用t统计量检验标准化系数的显著性。而当响应变量和输入变量不平稳时，其标准化系数不在满足t分布，这时再用t检验来进行显著性分析，导致拒绝原假设的概率增加，即容易犯第一类错误，从而得出错误的结论。

平稳时间序列有两种定义：严平稳和宽平稳

严平稳顾名思义，是一种条件非常苛刻的平稳性，它要求序列随着时间的推移，其统计性质保持不变。对于任意的τ，其联合概率密度函数满足：

$$F_{t1,t2,\cdots tm}(x_1,x_2,\cdots,x_m)=F_{t1+\tau,t2+\tau,\cdots tm+\tau}(x_1,x_2,\cdots,x_m)$$

严平稳的条件只是理论上的存在，现实中用得比较多的是宽平稳的条件。

宽平稳也叫弱平稳或者二阶平稳（均值和方差平稳），它应满足：

- 常数均值
- 常数方差
- 常数自协方差

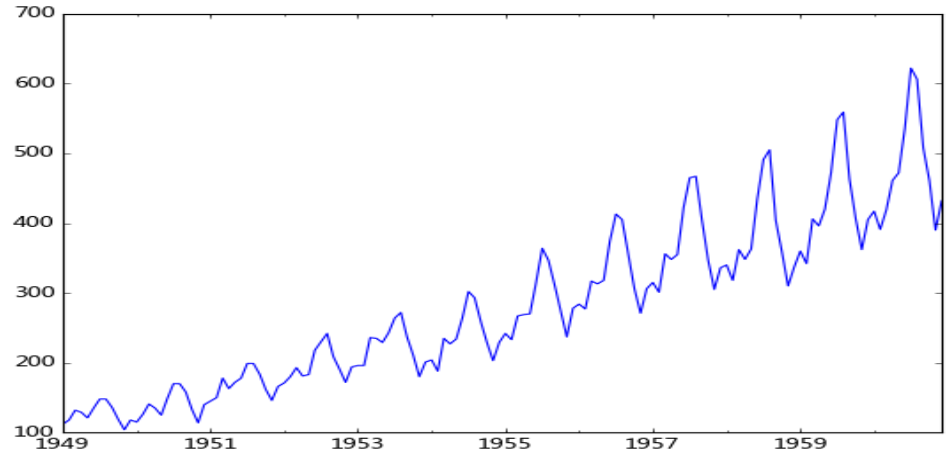
平稳性检验：观察法和单位根检验法

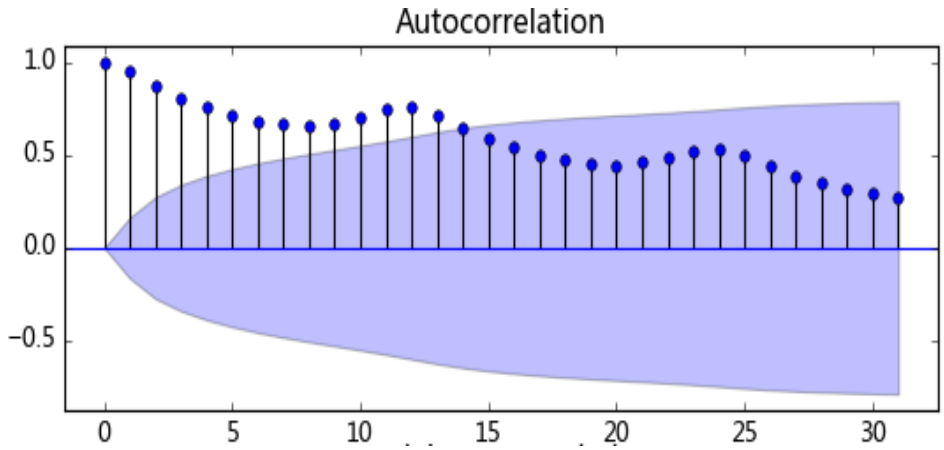
基于此，我写了一个名为test_stationarity的统计性检验模块，以便将某些统计检验结果更加直观的展现出来。

```
# -*- coding:utf-8 -*-
from statsmodels.tsa.stattools import adfuller
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# 移动平均图
def draw_trend(timeSeries, size):
    f = plt.figure(facecolor='white')
    # 对size个数据进行移动平均
    rol_mean = timeSeries.rolling(window=size).mean()
    # 对size个数据进行加权移动平均
    rol_weighted_mean = pd.ewma(timeSeries, span=size)

    timeSeries.plot(color='blue', label='Original')
    rolmean.plot(color='red', label='Rolling Mean')
    rol_weighted_mean.plot(color='black', label='Weighted Rolling Mean')
    plt.legend(loc='best')
    plt.title('Rolling Mean')
    plt.show()

def draw_ts(timeSeries):
    f = plt.figure(facecolor='white')
    timeSeries.plot(color='blue')
    plt.show()

'''
    Unit Root Test
   The null hypothesis of the Augmented Dickey-Fuller is that there is a unit
   root, with the alternative that there is no unit root. That is to say the
   bigger the p-value the more reason we assert that there is a unit root
```

```
'''
def testStationarity(ts):
    dftest = adfuller(ts)
    # 对上述函数求得的值进行语义描述
    dfoutput = pd.Series(dftest[0:4], index=['Test Statistic','p-value','#Lags
Used','Number of Observations Used'])
    for key,value in dftest[4].items():
        dfoutput['Critical Value (%s)'%key] = value
    return dfoutput

# 自相关和偏相关图，默认阶数为31阶
def draw_acf_pacf(ts, lags=31):
    f = plt.figure(facecolor='white')
    ax1 = f.add_subplot(211)
    plot_acf(ts, lags=31, ax=ax1)
    ax2 = f.add_subplot(212)
    plot_pacf(ts, lags=31, ax=ax2)
    plt.show()
```

观察法，通俗的说就是通过观察序列的趋势图与相关图是否随着时间的变化呈现出某种规律。所谓的规律就是时间序列经常提到的周期性因素，现实中遇到得比较多的是线性周期成分，这类周期成分可以采用差分或者移动平均来解决，而对于非线性周期成分的处理相对比较复杂，需要采用某些分解的方法。下图为航空数据的线性图，可以明显的看出它具有年周期成分和长期趋势成分。平稳序列的自相关系数会快速衰减，下面的自相关图并不能体现出该特征，所以我们有理由相信该序列是不平稳的。





单位根检验：ADF是一种常用的单位根检验方法，他的原假设为序列具有单位根，即非平稳，对于一个平稳的时序数据，就需要在给定的置信水平上显著，拒绝原假设。ADF只是单位根检验的方法之一，如果想采用其他检验方法，可以安装第三方包arch，里面提供了更加全面的单位根检验方法，个人还是比较钟情ADF检验。以下为检验结果，其p值大于0.99，说明并不能拒绝原假设。

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

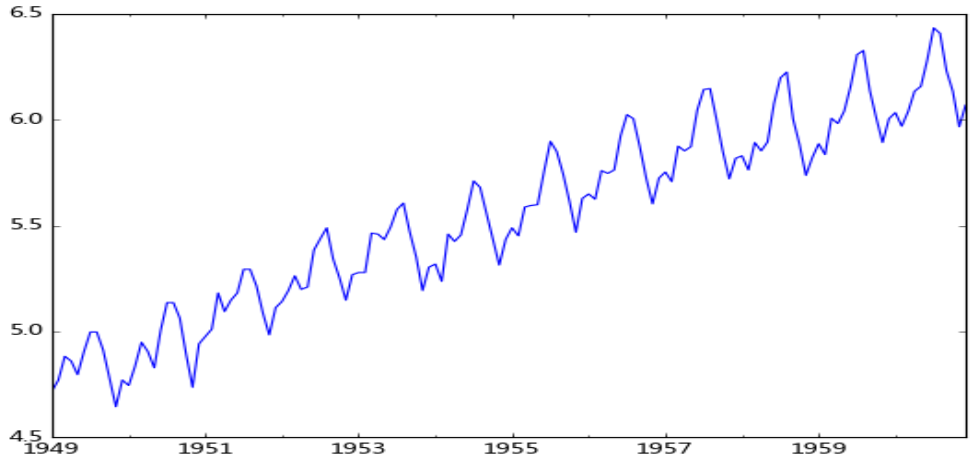
3. 平稳性处理

由前面的分析可知，该序列是不平稳的，然而平稳性是时间序列分析的前提条件，故我们需要对不平稳的序列进行处理将其转换成平稳的序列。

a. 对数变换

对数变换主要是为了减小数据的振动幅度，使其线性规律更加明显（我是这么理解的时间序列模型大部分都是线性的，为了尽量降低非线性的因素，需要对其进行预处理，也许我理解的不对）。对数变换相当于增加了一个惩罚机制，数据越大其惩罚越大，数据越小惩罚越小。这里强调一下，变换的序列需要满足大于0，小于0的数据不存在对数变换。

```
ts_log = np.log(ts)
test_stationarity.draw_ts(ts_log)
```

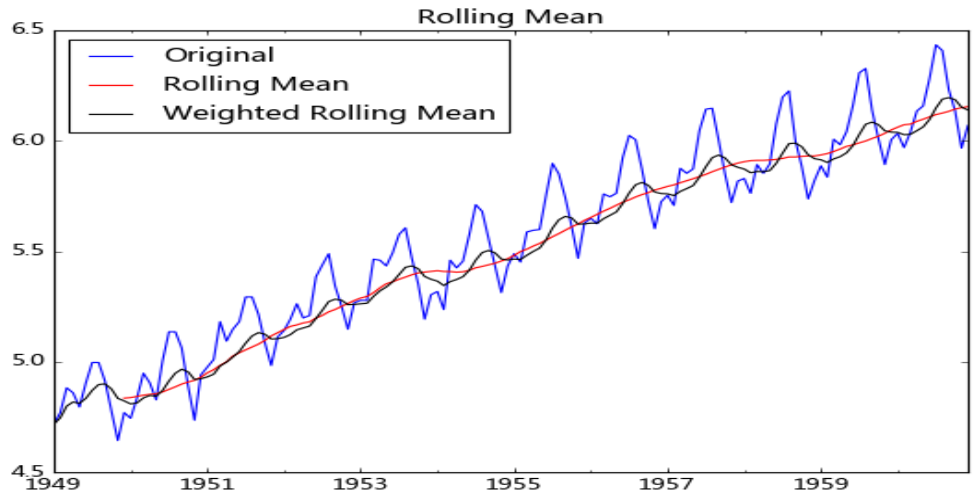


b. 平滑法

根据平滑技术的不同，平滑法具体分为移动平均法和指数平均法。

移动平均即利用一定时间间隔内的平均值作为某一期的估计值，而指数平均则是用变权的方法来计算均值

```
test_stationarity.draw_trend(ts_log, 12)
```



从上图可以发现窗口为12的移动平均能较好的剔除年周期性因素，而指数平均法是对周期内的数据进行了加权，能在一定程度上减小年周期因素，但并不能完全剔除，如要完全剔除可以进一步进行差分操作。

c. 差分

时间序列最常用来剔除周期性因素的方法当属差分了，它主要是对等周期间隔的数据进行线性求减。前面我们说过，ARIMA模型相对ARMA模型，仅多了差分操作，ARIMA模型几乎是所有时间序列软件都支持的，差分的实现与还原都非常方便。而statsmodel中，对差分的支持不是很好，它不支持高阶和多阶差分，为什么不支持，这里引用作者的说法：

```
if d > 2:
    #NOTE: to make more general, need to address the d == 2 stuff
    # in the predict method
    raise ValueError("d > 2 is not supported")
```

作者大概的意思是说预测方法中并没有解决高于2阶的差分，有没有感觉很牵强，不过没关系，我们有pandas。我们可以先用pandas将序列差分好，然后在对差分好的序列进行ARIMA拟合，只不过这样后面会多了一步人工还原的工作。

```
diff_12 = ts_log.diff(12)
diff_12.dropna(inplace=True)
diff_12_1 = diff_12.diff(1)
```

```
diff_12_1.dropna(inplace=True)
test_stationarity.testStationarity(diff_12_1)
```

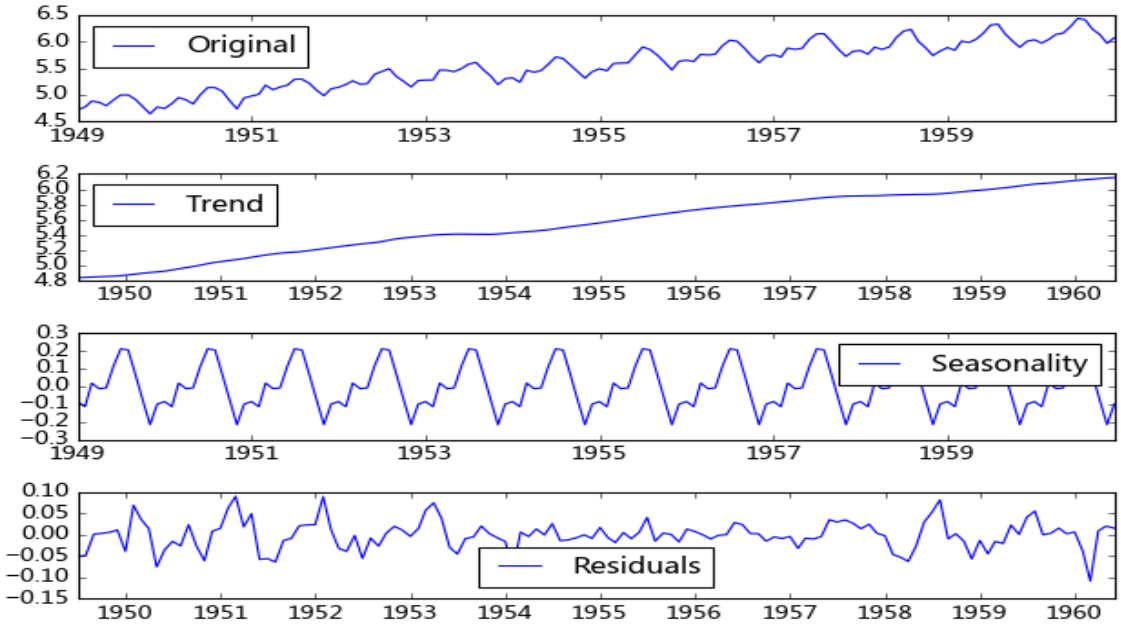
```
Test Statistic                -4.443325
p-value                        0.000249
#Lags Used                    12.000000
Number of Observations Used   118.000000
Critical Value (5%)           -2.886363
Critical Value (1%)           -3.487022
Critical Value (10%)          -2.580009
dtype: float64
```

从上面的统计检验结果可以看出，经过12阶差分和1阶差分后，该序列满足平稳性的要求了。

d. 分解

所谓分解就是将时序数据分离成不同的成分。statsmodels使用的X-11分解过程，它主要将时序数据分离成长期趋势、季节趋势和随机成分。与其它统计软件一样，statsmodels也支持两类分解模型，加法模型和乘法模型，这里我只实现加法，乘法只需将model的参数设置为"multiplicative"即可。

```
from statsmodels.tsa.seasonal import seasonal_decompose
decomposition = seasonal_decompose(ts_log, model="additive")

trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid
```



得到不同的分解成分后，就可以使用时间序列模型对各个成分进行拟合，当然也可以选择其他预测方法。我曾经用过小波对时序数据进行过分解，然后分别采用时间序列拟合，效果还不错。由于我对小波的理解不是很好，只能简单的调用接口，如果有谁对小波、傅里叶、卡尔曼理解得比较透，可以将时序数据进行更加准确的分解，由于分解后的时序数据避免了他们在建模时的交叉影响，所以我相信它将有助于预测准确性的提高。

4. 模型识别

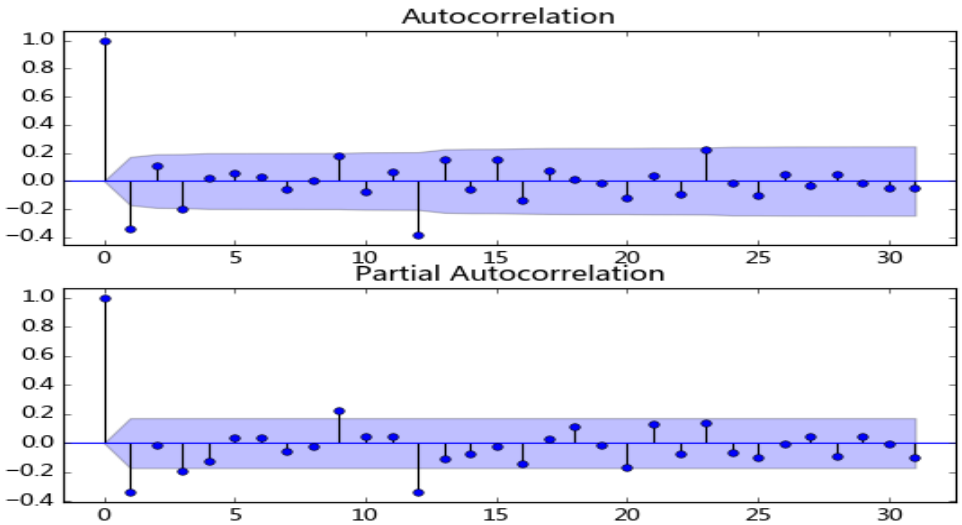
在前面的分析可知，该序列具有明显的年周期与长期成分。对于年周期成分我们使用窗口为12的移动平进行处理，对于长期趋势成分我们采用1阶差分来进行处理。

```
rol_mean = ts_log.rolling(window=12).mean()
rol_mean.dropna(inplace=True)
ts_diff_1 = rol_mean.diff(1)
ts_diff_1.dropna(inplace=True)
test_stationarity.testStationarity(ts_diff_1)
```

```
Test Statistic                  -2.709577
p-value                          0.072396
#Lags Used                      12.000000
Number of Observations Used    119.000000
Critical Value (5%)             -2.886151
Critical Value (1%)             -3.486535
Critical Value (10%)            -2.579896
dtype: float64
```

观察其统计量发现该序列在置信水平为95%的区间下并不显著，我们对其进行再次一阶差分。再次差分后的序列其自相关具有快速衰减的特点，t统计量在99%的置信水平下是显著的，这里我不再做详细说明。

```
ts_diff_2 = ts_diff_1.diff(1)
ts_diff_2.dropna(inplace=True)
```



数据平稳后，需要对模型定阶，即确定p、q的阶数。观察上图，发现自相关和偏相系数都存在拖尾的特点，并且他们都具有明显的一阶相关性，所以我们设定p=1, q=1。下面就可以使用ARMA模型进行数据拟合了。这里我不使用ARIMA(ts_diff_1, order=(1, 1, 1))进行拟合，是因为含有差分操作时，预测结果还原老出问题，至今还没弄明白。

```
from statsmodels.tsa.arima_model import ARMA
model = ARMA(ts_diff_2, order=(1, 1))
result_arma = model.fit( disp=-1, method='css')
```

5. 样本拟合

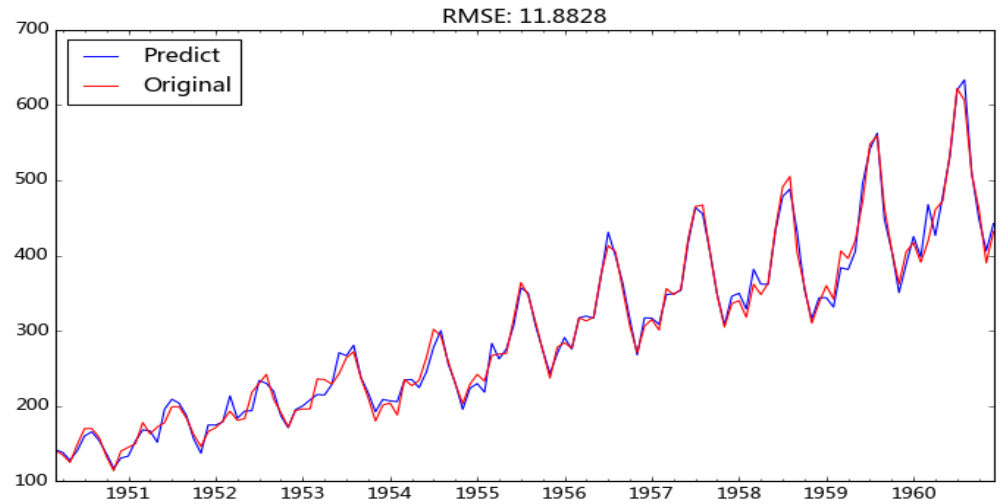
模型拟合完后，我们就可以对其进行预测了。由于ARMA拟合的是经过相关预处理后的数据，故其预测值需要通过相关逆变换进行还原。

```
predict_ts = result_arma.predict()
# 一阶差分还原
diff_shift_ts = ts_diff_1.shift(1)
diff_recover_1 = predict_ts.add(diff_shift_ts)
# 再次一阶差分还原
rol_shift_ts = rol_mean.shift(1)
diff_recover = diff_recover_1.add(rol_shift_ts)
# 移动平均还原
rol_sum = ts_log.rolling(window=11).sum()
rol_recover = diff_recover*12 - rol_sum.shift(1)
# 对数还原
log_recover = np.exp(rol_recover)
log_recover.dropna(inplace=True)
```

我们使用均方根误差（RMSE）来评估模型样本内拟合的好坏。利用该准则进行判别时，需要剔除“非预测”数据的影响。

```
ts = ts[log_recover.index]  # 过滤没有预测的记录
plt.figure(facecolor='white')
log_recover.plot(color='blue', label='Predict')
ts.plot(color='red', label='Original')
plt.legend(loc='best')
```

```
plt.title('RMSE: %.4f'% np.sqrt(sum((log_recover-ts)**2)/ts.size))
plt.show()
```



观察上图的拟合效果，均方根误差为11.8828，感觉还过得去。

6. 完善ARIMA模型

前面提到statsmodels里面的ARIMA模块不支持高阶差分，我们的做法是将差分分离出来，但是这样会多了一步人工还原的操作。基于上述问题，我将差分过程进行了封装，使序列能按照指定的差分列表依次进行差分，并相应的构造了一个还原的方法，实现差分序列的自动还原。

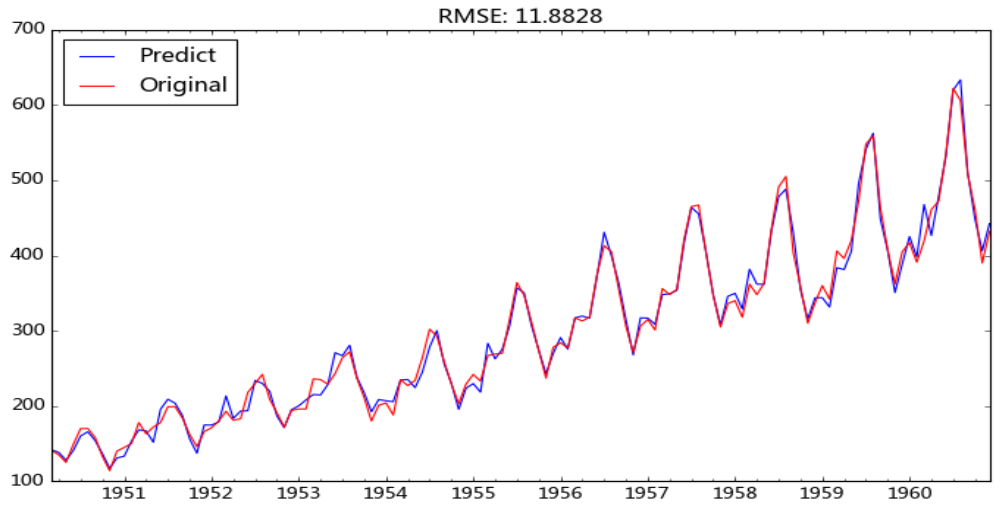
```
# 差分操作
def diff_ts(ts, d):
    global shift_ts_list
    #  动态预测第二日的值时所需要的差分序列
    global last_data_shift_list
    shift_ts_list = []
    last_data_shift_list = []
```

```
    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
```

现在我们直接使用差分的方法进行数据处理，并以同样的过程进行数据预测与还原。

```
diffed_ts = diff_ts(ts_log, d=[12, 1])
model = arima_model(diffed_ts)
```

```
model.certain_model(1, 1)
predict_ts = model.properModel.predict()
diff_recover_ts = predict_diff_recover(predict_ts, d=[12, 1])
log_recover = np.exp(diff_recover_ts)
```



是不是发现这里的预测结果和上一篇的使用12阶移动平均的预测结果一模一样。这是因为12阶移动平均加上一阶差分与直接12阶差分是等价的关系，后者是前者数值的12倍，这个应该不难推导。

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（过拟合这东西应该以辩证的眼光看待）。

```
def proper_model(data_ts, maxLag):
    init_bic = sys.maxint
    init_p = 0
    init_q = 0
    init_properModel = None
    for p in np.arange(maxLag):
        for q in np.arange(maxLag):
```

```
            model = ARMA(data_ts, order=(p, q))
            try:
                results_ARMA = model.fit(disp=-1, method='css')
            except:
                continue
            bic = results_ARMA.bic
            if bic < init_bic:
                init_p = p
                init_q = q
                init_properModel = results_ARMA
                init_bic = bic
    return init_bic, init_p, init_q, init_properModel
```

相对最优参数识别结果：BIC: -1090.44209358 p: 0 q: 1 ， RMSE:11.8817198331。我们发现模型自动识别的参数要比我手动选取的参数更优。

7.滚动预测

所谓滚动预测是指通过添加最新的数据预测第二天的值。对于一个稳定的预测模型，不需要每天都去拟合，我们可以给他设定一个阀值，例如每周拟合一次，该期间只需通过添加最新的数据实现滚动预测即可。基于此我编写了一个名为arima_model的类，主要包含模型自动识别方法，滚动预测的功能，详细代码可以查看附录。数据的动态添加：

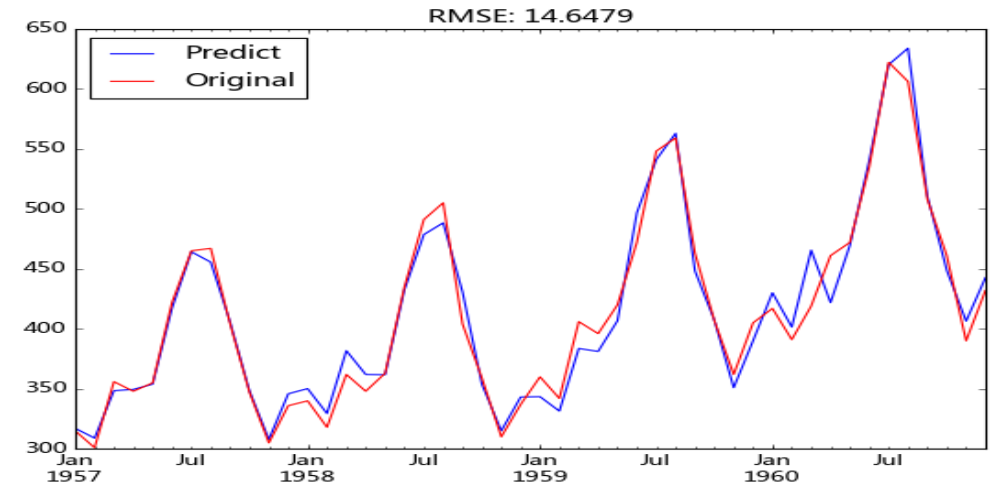
```
from dateutil.relativedelta import relativedelta
def _add_new_data(ts, dat, type='day'):
if type == 'day':
        new_index = ts.index[-1] + relativedelta(days=1)
    elif type == 'month':
        new_index = ts.index[-1] + relativedelta(months=1)
    ts[new_index] = dat

def add_today_data(model, ts,  data, d, type='day'):
```

```
    _add_new_data(ts, data, type)  # 为原始序列添加数据
    # 为滞后序列添加新值
    d_ts = diff_ts(ts, d)
    model.add_today_data(d_ts[-1], type)

def forecast_next_day_data(model, type='day'):
    if model == None:
        raise ValueError('No model fit before')
    fc = model.forecast_next_day_value(type)
    return predict_diff_recover(fc, [12, 1])
```

现在我们就可以使用滚动预测的方法向外预测了，取1957年之前的数据作为训练数据，其后的数据作为测试，并设定模型每第七天就会重新拟合一次。这里的diffed_ts对象会随着add_today_data方法自动添加数据，这是由于它与add_today_data方法中的d_ts指向的同一对象，该对象会动态的添加数据。

```
ts_train = ts_log[:'1956-12']
ts_test = ts_log['1957-1':]

diffed_ts = diff_ts(ts_train, [12, 1])
forecast_list = []

for i, dta in enumerate(ts_test):
    if i%7 == 0:
        model = arima_model(diffed_ts)
        model.certain_model(1, 1)
    forecast_data = forecast_next_day_data(model, type='month')
    forecast_list.append(forecast_data)
    add_today_data(model, ts_train, dta, [12, 1], type='month')

predict_ts = pd.Series(data=forecast_list, index=ts['1957-1':].index)
```

```
log_recover = np.exp(predict_ts)
original_ts = ts['1957-1':]
```



动态预测的均方根误差为：14.6479，与前面样本内拟合的均方根误差相差不大，说明模型并没有过拟合，并且整体预测效果都较好。

8. 模型序列化

在进行动态预测时，我们不希望将整个模型一直在内存中运行，而是希望有新的数据到来时才启动该模型。这时我们就应该把整个模型从内存导出到硬盘中，而序列化正好能满足该要求。序列化最常用的就是使用json模块了，但是它是时间对象支持得不是很好，有人对json模块进行了拓展以使得支持时间对象，这里我们不采用该方法，我们使用pickle模块，它和json的接口基本相同，有兴趣的可以去查看一下。我在实际应用中是采用的面向对象的编程，它的序列化主要是将类的属性序列化即可，而在面向过程的编程中，模型序列化需要将需要序列化的对象公有化，这样会使得对前面函数的参数改动较大，我不在详细阐述该过程。

总结

与其它统计语言相比，python在统计分析这块还显得不那么“专业”。随着numpy、pandas、scipy、sklearn、gensim、statsmodels等包的推动，我相信也祝愿python在数据分析这块越来越好。与SAS和R相比，python的时间序列模块还不是很成熟，我这里仅起到抛砖引玉的作用，希望各位能人志士能贡献自己的力量，使其更加完善。实际应用中我全是面向过程来编写的，为了阐述方便，我用面向过程重新罗列了一遍，实在感觉很不方便。原本打算分三篇来写的，还有一部分实际应用的部分，不打算再写了，还请大家原谅。实际应用主要是具体问题具体分析，这当中第一步就是要查询问题，这步花的时间往往会比较多，然后再是解决问题。以我前面项目遇到

的问题为例，当时遇到了以下几个典型的问题：1.周期长度不恒定的周期成分，例如每月的1号具有周期性，但每月1号与1号之间的时间间隔是不相等的；2.含有缺失值以及含有记录为0的情况无法进行对数变换；3.节假日的影响等等。

代码：

```
# -*-coding:utf-8-*-
import pandas as pd
import numpy as np
from statsmodels.tsa.arima_model import ARMA
import sys
from dateutil.relativedelta import relativedelta
from copy import deepcopy
import matplotlib.pyplot as plt

class arima_model:

    def __init__(self, ts, maxLag=9):
        self.data_ts = ts
        self.resid_ts = None
        self.predict_ts = None
        self.maxLag = maxLag
        self.p = maxLag
        self.q = maxLag
        self.properModel = None
        self.bic = sys.maxint

    # 计算最优ARIMA模型，将相关结果赋给相应属性
    def get_proper_model(self):
        self._proper_model()
        self.predict_ts = deepcopy(self.properModel.predict())
        self.resid_ts = deepcopy(self.properModel.resid)

    # 对于给定范围内的p,q计算拟合得最好的arima模型，这里是对差分好的数据进行拟合，故差分恒为0
    def _proper_model(self):
        for p in np.arange(self.maxLag):
```

```
            for q in np.arange(self.maxLag):
                # print p,q,self.bic
                model = ARMA(self.data_ts, order=(p, q))
                try:
                    results_ARMA = model.fit(disp=-1, method='css')
                except:
                    continue
                bic = results_ARMA.bic
                # print 'bic:',bic,'self.bic:',self.bic
                if bic < self.bic:
                    self.p = p
                    self.q = q
                    self.properModel = results_ARMA
                    self.bic = bic
                    self.resid_ts = deepcopy(self.properModel.resid)
                    self.predict_ts = self.properModel.predict()

    # 参数确定模型
    def certain_model(self, p, q):
            model = ARMA(self.data_ts, order=(p, q))
            try:
                self.properModel = model.fit( disp=-1, method='css')
                self.p = p
                self.q = q
                self.bic = self.properModel.bic
                self.predict_ts = self.properModel.predict()
                self.resid_ts = deepcopy(self.properModel.resid)
            except:
                print 'You can not fit the model with this parameter p,q, ' \
                      'please use the get_proper_model method to get the best model'

    # 预测第二日的值
    def forecast_next_day_value(self, type='day'):
        # 我修改了statsmodels包中arima_model的源代码，添加了constant属性，需要先运行forecast方
法，为constant赋值
        self.properModel.forecast()
        if self.data_ts.index[-1] != self.resid_ts.index[-1]:
```

```
                raise ValueError('''The index is different in data_ts and resid_ts, please
add new data to data_ts.
                If you just want to forecast the next day data without add the real next day
data to data_ts,
                please run the predict method which arima_model included itself''')
            if not self.properModel:
                raise ValueError('The arima model have not computed, please run the
proper_model method before')
            para = self.properModel.params

            # print self.properModel.params
            if self.p == 0:   # It will get all the value series with setting self.data_ts[-
self.p:] when p is zero
                ma_value = self.resid_ts[-self.q:]
                values = ma_value.reindex(index=ma_value.index[::-1])
            elif self.q == 0:
                ar_value = self.data_ts[-self.p:]
                values = ar_value.reindex(index=ar_value.index[::-1])
            else:
                ar_value = self.data_ts[-self.p:]
                ar_value = ar_value.reindex(index=ar_value.index[::-1])
                ma_value = self.resid_ts[-self.q:]
                ma_value = ma_value.reindex(index=ma_value.index[::-1])
                values = ar_value.append(ma_value)

            predict_value = np.dot(para[1:], values) + self.properModel.constant[0]
            self._add_new_data(self.predict_ts, predict_value, type)
            return predict_value

        # 动态添加数据函数，针对索引是月份和日分别进行处理
        def _add_new_data(self, ts, dat, type='day'):
            if type == 'day':
                new_index = ts.index[-1] + relativedelta(days=1)
            elif type == 'month':
                new_index = ts.index[-1] + relativedelta(months=1)
            ts[new_index] = dat
```

```
    def add_today_data(self, dat, type='day'):
        self._add_new_data(self.data_ts, dat, type)
        if self.data_ts.index[-1] != self.predict_ts.index[-1]:
            raise ValueError('You must use the forecast_next_day_value method forecast
the value of today before')
        self._add_new_data(self.resid_ts, self.data_ts[-1] - self.predict_ts[-1], type)

if __name__ == '__main__':
    df = pd.read_csv('AirPassengers.csv', encoding='utf-8', index_col='date')
    df.index = pd.to_datetime(df.index)
    ts = df['x']

    # 数据预处理
    ts_log = np.log(ts)
    rol_mean = ts_log.rolling(window=12).mean()
    rol_mean.dropna(inplace=True)
    ts_diff_1 = rol_mean.diff(1)
    ts_diff_1.dropna(inplace=True)
    ts_diff_2 = ts_diff_1.diff(1)
    ts_diff_2.dropna(inplace=True)

    # 模型拟合
    model = arima_model(ts_diff_2)
    #  这里使用模型参数自动识别
    model.get_proper_model()
    print 'bic:', model.bic, 'p:', model.p, 'q:', model.q
    print model.properModel.forecast()[0]
    print model.forecast_next_day_value(type='month')

    # 预测结果还原
    predict_ts = model.properModel.predict()
    diff_shift_ts = ts_diff_1.shift(1)
    diff_recover_1 = predict_ts.add(diff_shift_ts)
    rol_shift_ts = rol_mean.shift(1)
    diff_recover = diff_recover_1.add(rol_shift_ts)
    rol_sum = ts_log.rolling(window=11).sum()
    rol_recover = diff_recover*12 - rol_sum.shift(1)
```

```
log_recover = np.exp(rol_recover)
log_recover.dropna(inplace=True)

# 预测结果作图
ts = ts[log_recover.index]
plt.figure(facecolor='white')
log_recover.plot(color='blue', label='Predict')
ts.plot(color='red', label='Original')
plt.legend(loc='best')
plt.title('RMSE: %.4f'% np.sqrt(sum((log_recover-ts)**2)/ts.size))
plt.show()
```

三、LSTM

关于技术理论部分，可以参考这两篇文章（RNN、LSTM），本文主要从数据、代码角度，利用LSTM进行时间序列预测。

> 时间序列（或称动态数列）是指将同一统计指标的数值按其发生的时间先后顺序排列而成的数列。时间序列分析的主要目的是根据已有的历史数据对未来进行预测。

时间序列构成要素：长期趋势，季节变动，循环变动，不规则变动

- 长期趋势（ T ）现象在较长时期内受某种根本性因素作用而形成的总的变动趋势
- 季节变动（ S ）现象在一年内随着季节的变化而发生的有规律的周期性变动
- 循环变动（ C ）现象以若干年为周期所呈现出的波浪起伏形态的有规律的变动
- 不规则变动（I ）是一种无规律可循的变动，包括严格的随机变动和不规则的突发性影响很大的变动两种类型

（1）原始时间序列数据（只列出了18行）

```
1455.219971
1399.420044
1402.109985
1403.449951
1441.469971
1457.599976
1438.560059
1432.25
1449.680054
1465.150024
1455.140015
1455.900024
1445.569946
1441.359985
1401.530029
1410.030029
1404.089966
1398.560059
```

（2）处理数据使之符合LSTM的要求

为了更加直观的了解数据格式，代码中加入了一些打印（print），并且后面加了注释，就是输出值

```
def load_data(filename, seq_len):
    f = open(filename, 'rb').read()
    data = f.split('\n')

    print('data len:',len(data))       #4172
    print('sequence len:',seq_len)     #50

    sequence_length = seq_len + 1
    result = []
    for index in range(len(data) - sequence_length):
        result.append(data[index: index + sequence_length])  #得到长度为seq_len+1的向量，
最后一个作为label
```

```
    print('result len:',len(result))   #4121
    print('result shape:',np.array(result).shape)  #(4121,51)

    result = np.array(result)

    #划分train、test
    row = round(0.9 * result.shape[0])
    train = result[:row, :]
    np.random.shuffle(train)
    x_train = train[:, :-1]
    y_train = train[:, -1]
    x_test = result[row:, :-1]
    y_test = result[row:, -1]

    x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
    x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))

    print('X_train shape:',X_train.shape)  #(3709L, 50L, 1L)
    print('y_train shape:',y_train.shape)  #(3709L,)
    print('X_test shape:',X_test.shape)    #(412L, 50L, 1L)
    print('y_test shape:',y_test.shape)    #(412L,)

    return [x_train, y_train, x_test, y_test]
```

（3）LSTM模型

本文使用的是keras深度学习框架，读者可能用的是其他的，诸如theano、tensorflow等，大同小异。

Keras LSTM官方文档

LSTM的结构可以自己定制，Stack LSTM or Bidirectional LSTM

```
def build_model(layers):  #layers [1,50,100,1]
    model = Sequential()

    #Stack LSTM
    model.add(LSTM(input_dim=layers[0],output_dim=layers[1],return_sequences=True))
    model.add(Dropout(0.2))

    model.add(LSTM(layers[2],return_sequences=False))
    model.add(Dropout(0.2))

    model.add(Dense(output_dim=layers[3]))
    model.add(Activation("linear"))

    start = time.time()
    model.compile(loss="mse", optimizer="rmsprop")
    print("Compilation Time : ", time.time() - start)
    return model
```

（4）LSTM训练预测

1.直接预测

```
def predict_point_by_point(model, data):
    predicted = model.predict(data)
    print('predicted shape:',np.array(predicted).shape)  #(412L,1L)
    predicted = np.reshape(predicted, (predicted.size,))
    return predicted
```

2.滚动预测

```
def predict_sequence_full(model, data, window_size):  #data X_test
    curr_frame = data[0]  #(50L,1L)
    predicted = []
    for i in xrange(len(data)):
        #x = np.array([[[1],[2],[3]], [[4],[5],[6]]])  x.shape (2, 3, 1) x[0,0] =
array([1])  x[:,np.newaxis,:,:].shape  (2, 1, 3, 1)
        predicted.append(model.predict(curr_frame[newaxis,:,:])[0,0])
#np.array(curr_frame[newaxis,:,:]).shape (1L,50L,1L)
        curr_frame = curr_frame[1:]
        curr_frame = np.insert(curr_frame, [window_size-1], predicted[-1], axis=0)
#numpy.insert(arr, obj, values, axis=None)
    return predicted
```

2.滑动窗口+滚动预测

```
def predict_sequences_multiple(model, data, window_size, prediction_len):
#window_size = seq_len
    prediction_seqs = []
    for i in xrange(len(data)/prediction_len):
        curr_frame = data[i*prediction_len]
        predicted = []
        for j in xrange(prediction_len):
            predicted.append(model.predict(curr_frame[newaxis,:,:])[0,0])
            curr_frame = curr_frame[1:]
            curr_frame = np.insert(curr_frame, [window_size-1], predicted[-1], axis=0)
        prediction_seqs.append(predicted)
    return prediction_seqs
```

完整代码：

```
# -*- coding: utf-8 -*-

from __future__ import print_function
```

```
import time
import warnings
import numpy as np
import time
import matplotlib.pyplot as plt
from numpy import newaxis
from keras.layers.core import Dense, Activation, Dropout
from keras.layers.recurrent import LSTM
from keras.models import Sequential

warnings.filterwarnings("ignore")

def load_data(filename, seq_len, normalise_window):
    f = open(filename, 'rb').read()
    data = f.split('\n')

    print('data len:',len(data))
    print('sequence len:',seq_len)

    sequence_length = seq_len + 1
    result = []
    for index in range(len(data) - sequence_length):
        result.append(data[index: index + sequence_length])  #得到长度为seq_len+1的向量，最
后一个作为label

    print('result len:',len(result))
    print('result shape:',np.array(result).shape)
    print(result[:1])

    if normalise_window:
        result = normalise_windows(result)

    print(result[:1])
    print('normalise_windows result shape:',np.array(result).shape)

    result = np.array(result)
```

```
    #划分train、test
    row = round(0.9 * result.shape[0])
    train = result[:row, :]
    np.random.shuffle(train)
    x_train = train[:, :-1]
    y_train = train[:, -1]
    x_test = result[row:, :-1]
    y_test = result[row:, -1]

    x_train = np.reshape(x_train, (x_train.shape[0], x_train.shape[1], 1))
    x_test = np.reshape(x_test, (x_test.shape[0], x_test.shape[1], 1))

    return [x_train, y_train, x_test, y_test]

def normalise_windows(window_data):
    normalised_data = []
    for window in window_data:   #window shape (sequence_length L ,)  即(51L,)
        normalised_window = [((float(p) / float(window[0])) - 1) for p in window]
        normalised_data.append(normalised_window)
    return normalised_data

def build_model(layers):  #layers [1,50,100,1]
    model = Sequential()

    model.add(LSTM(input_dim=layers[0],output_dim=layers[1],return_sequences=True))
    model.add(Dropout(0.2))

    model.add(LSTM(layers[2],return_sequences=False))
    model.add(Dropout(0.2))

    model.add(Dense(output_dim=layers[3]))
    model.add(Activation("linear"))

    start = time.time()
    model.compile(loss="mse", optimizer="rmsprop")
    print("Compilation Time : ", time.time() - start)
```

```
    return model

#直接全部预测
def predict_point_by_point(model, data):
    predicted = model.predict(data)
    print('predicted shape:',np.array(predicted).shape)  #(412L,1L)
    predicted = np.reshape(predicted, (predicted.size,))
    return predicted

#滚动预测
def predict_sequence_full(model, data, window_size):  #data X_test
    curr_frame = data[0]  #(50L,1L)
    predicted = []
    for i in xrange(len(data)):
        #x = np.array([[[1],[2],[3]], [[4],[5],[6]]])  x.shape (2, 3, 1) x[0,0] =
array([1])  x[:,np.newaxis,:,:].shape  (2, 1, 3, 1)
        predicted.append(model.predict(curr_frame[newaxis,:,:])[0,0])
#np.array(curr_frame[newaxis,:,:]).shape (1L,50L,1L)
        curr_frame = curr_frame[1:]
        curr_frame = np.insert(curr_frame, [window_size-1], predicted[-1], axis=0)
#numpy.insert(arr, obj, values, axis=None)
    return predicted

def predict_sequences_multiple(model, data, window_size, prediction_len):  #window_size
= seq_len
    prediction_seqs = []
    for i in xrange(len(data)/prediction_len):
        curr_frame = data[i*prediction_len]
        predicted = []
        for j in xrange(prediction_len):
            predicted.append(model.predict(curr_frame[newaxis,:,:])[0,0])
            curr_frame = curr_frame[1:]
            curr_frame = np.insert(curr_frame, [window_size-1], predicted[-1], axis=0)
        prediction_seqs.append(predicted)
    return prediction_seqs

def plot_results(predicted_data, true_data, filename):
```

```
    fig = plt.figure(facecolor='white')
    ax = fig.add_subplot(111)
    ax.plot(true_data, label='True Data')
    plt.plot(predicted_data, label='Prediction')
    plt.legend()
    plt.show()
    plt.savefig(filename+'.png')

def plot_results_multiple(predicted_data, true_data, prediction_len):
    fig = plt.figure(facecolor='white')
    ax = fig.add_subplot(111)
    ax.plot(true_data, label='True Data')
    #Pad the list of predictions to shift it in the graph to it's correct start
    for i, data in enumerate(predicted_data):
        padding = [None for p in xrange(i * prediction_len)]
        plt.plot(padding + data, label='Prediction')
        plt.legend()
    plt.show()
    plt.savefig('plot_results_multiple.png')

if __name__=='__main__':
    global_start_time = time.time()
    epochs  = 1
    seq_len = 50

    print('> Loading data... ')

    X_train, y_train, X_test, y_test = lstm.load_data('sp500.csv', seq_len, True)

    print('X_train shape:',X_train.shape)  #(3709L, 50L, 1L)
    print('y_train shape:',y_train.shape)  #(3709L,)
    print('X_test shape:',X_test.shape)    #(412L, 50L, 1L)
    print('y_test shape:',y_test.shape)    #(412L,)

    print('> Data Loaded. Compiling...')

    model = lstm.build_model([1, 50, 100, 1])
```

```
    model.fit(X_train,y_train,batch_size=512,nb_epoch=epochs,validation_split=0.05)

    multiple_predictions = lstm.predict_sequences_multiple(model, X_test, seq_len,
prediction_len=50)
    print('multiple_predictions shape:',np.array(multiple_predictions).shape)   #
(8L,50L)

    full_predictions = lstm.predict_sequence_full(model, X_test, seq_len)
    print('full_predictions shape:',np.array(full_predictions).shape)    #(412L,)

    point_by_point_predictions = lstm.predict_point_by_point(model, X_test)
    print('point_by_point_predictions
shape:',np.array(point_by_point_predictions).shape)  #(412L)

    print('Training duration (s) : ', time.time() - global_start_time)

    plot_results_multiple(multiple_predictions, y_test, 50)
    plot_results(full_predictions,y_test,'full_predictions')
    plot_results(point_by_point_predictions,y_test,'point_by_point_predictions')
```

相关博客：

http://www.cnblogs.com/foley/p/5582358.html

http://www.10tiao.com/html/284/201608/2652390079/1.html

http://blog.csdn.net/a819825294/article/details/54376781

http://www.cnblogs.com/arkenstone/p/5794063.html

分类: ML

AI-ML-DL
关注 - 0
粉丝 - 15
+加关注

2 0

« 上一篇：CV：object detection(SS)
» 下一篇：CV：object detection(RCNN)

posted @ 2017-02-15 12:10 AI-ML-DL 阅读(706) 评论(1) 编辑 收藏

评论列表

#1楼 2017-03-24 18:14 bubbleStar

总结的超棒，赞

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。