基于高斯过程回归和 SIR 模型的 COVID-19 预测

1710018 崔嘉珩 统计学
1710066 沈石禹 统计学
1710070 苏昊宇 统计学

1 高斯过程回归：

1.1 背景：

在概率论和统计学中，高斯过程(Gaussian Process，简称 GP)是随机过程的一种，其随机变量组成的每个有限集合都具有多元正态分布，即它们的任意有限线性组合都是正态分布的.高斯过程的分布是所有这些（无限多个）随机变量的联合分布.[1]

高斯过程在统计建模中用处很大，得益于其正态分布的性质.例如高斯过程可以显式导出一些量的分布，包括某个时间范围内的过程平均值.我们还可以用一小段时间内的样本值估算其误差.

涉及高斯过程的机器学习算法，如高斯过程回归，需要使用惰性学习(lazy learning)和核函数(kernel function)，核函数是一种点之间相似度的度量.我们可以利用训练集对高斯过程回归模型进行训练，最终的模型能对给定点进行函数值预测，并认为预测值是从一个一维正态分布得到的样本.[2]

1.2 高斯过程回归的理论推理：

1.2.1 一个简单的模型：

我们先从一个简单的情形入手：如图 1.1 所示，我们已知来自某一总体的$X_1(x_1,y_1),\dots,X_5(x_5,y_5)$共五个数据点，其中$x_1 \le \cdots \le x_5$.第六个数据点$X_6$也来自于同一个总体，其横坐标$x_6$已知，且知道$x_6 \in [x_1,x_5]$，如图中虚线所示.现在想要利用高斯过程回归对$X_6$的纵坐标$y_6$进行预测.

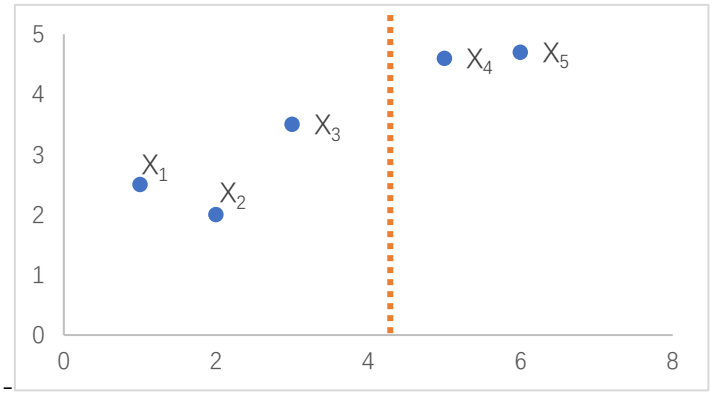

图 1.1 一个简单的模型

若要利用高斯过程回归，我们需要进行如下分析与假设：

我们发现，x_6在x_3和x_4之间，由于所有的点都来自于同一个总体，y_3和y_4会限制y_6的取值，我们假设限制的方式为y_3和y_4让y_6服从一个正态分布.

记$y=(y_1,\dots,y_6)'$.我们可以引入隐变量f，使得$f(x_i) \overset{\text{def}}{=\!=} y_i$，则有$f \overset{\text{def}}{=\!=} (f(x_1),\dots,f(x_6))'$.为了方便推导$y_6$，也为了能利用所有的已知样本，不妨假设 f 服从联

合正态分布$N(m(y),\ k(y,\ y'))$，其中y'为y的转置，$m(y)=E[f(y)]$为期望，$k(y,y')=E[(f(y)-m(y))(f(y)-m(y))']$为协方差矩阵.

1.2.2 高斯过程回归模型：

我们利用贝叶斯统计的观点来解释高斯回归模型:

记已知的数据向量为f，所需要预测的的集合为X^*，对应的预测值向量为f^*.由假设我们知f和f^*都服从多元正态分布，记$f\sim N(\mu,K)$，$f^*\sim N(\mu^*,K^*)$.直接用期望和协方差阵的定义计算f^*的分布，比较麻烦；如果以贝叶斯统计的观点来看，假设我们有先验信息，那么f^*的分布可以由先验分布与通过样本进行估计的分布推出.[3]

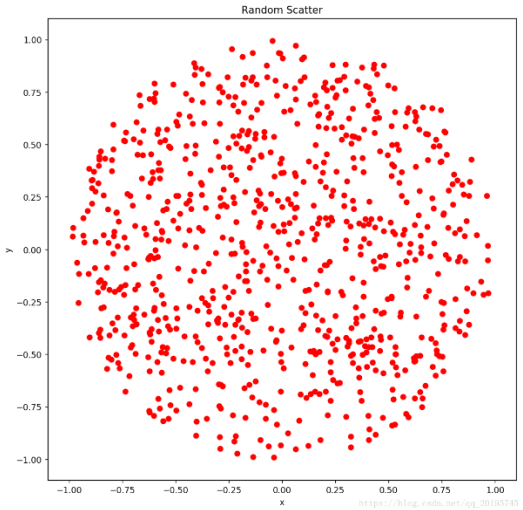

图 1.2：y与x相关性很小的数据[3]

若y的大小与自变量x的取值相关性很小，如图 1.2 所示.对于这样的数据，我们可以如下给出先验分布:

$$f(x)\sim N(0,\begin{bmatrix}1 & & \\ & \ddots & \\ & & 1\end{bmatrix})$$

若y的大小与x的取值有一定的相关性，如图 1.1 所示，我们需要用到核函数来给出先验分布.由于数据可以进行中心化，我们不妨假设先验分布的均值为0，再使用 RBF kernel[4]作为协方差阵.其中 RBF kernel 可以表示为:

$$K(\theta,x,x')=\theta^2\exp\left(-\frac{1}{2\sigma^2}(x-x')^2\right)$$

其中θ为超参数，σ为需要通过学习进行确定的参数.我们可以使用梯度下降法进行学习，目标是在$f(x)\sim N(0,\ K(\theta,x,x'))$的情况下使$p(y^*|X^*,f(x))$最大.因此我们只需要使$\log p(y^*|X^*,f(x))$最大即可，可以证明如下式子[1]:

$$\frac{\partial\log p(y^*|X^*)}{\partial\theta}=\frac{1}{2}y'K_y^{-1}\frac{\partial K_y}{\partial\theta}K_y^{-1}y-\frac{1}{2}tr(K_y^{-1}\frac{\partial K_y}{\partial\theta})$$

其中K_y为后验分布的协方差阵，我们可以利用该式求$p(y^*|X^*,f(x))$的最大值点.

在进行学习后，我们可以得到$K\overset{\text{def}}{=}K(\theta,x,x')$，$K^*\overset{\text{def}}{=}K(\theta,x^*,x)$，$K^{**}\overset{\text{def}}{=}K(\theta,x^*,x^*)$.其中$K^*$的意义为新数据点$x^*$与训练集所有样本点间的协方差，$K^{**}$的意义为新数据点$x^*$的自协方差.可以证明[1]，$f^*\sim N(A,B)$，$A$和$B$有如下形式:

$$A = K^* K^{-1} f$$
$$B = K^{**} - K^* K^{-1} K^{*\prime}$$

有了A和B，我们可以用A来对y^*进行估计，并用B对估计值建立置信区间.

1.3 高斯过程回归适用的情况及优劣

回到图 1.1 表示的模型中，值得注意到：我们要进行预测的点X_6满足横坐标夹在已知点横坐标之间，因此我们能说其他点的纵坐标对y_6进行“限制”.如果X_6没有满足横坐标夹在已知点横坐标之间，如下图所示，其他点对y_6的“限制”便无法用正态性来进行描述：

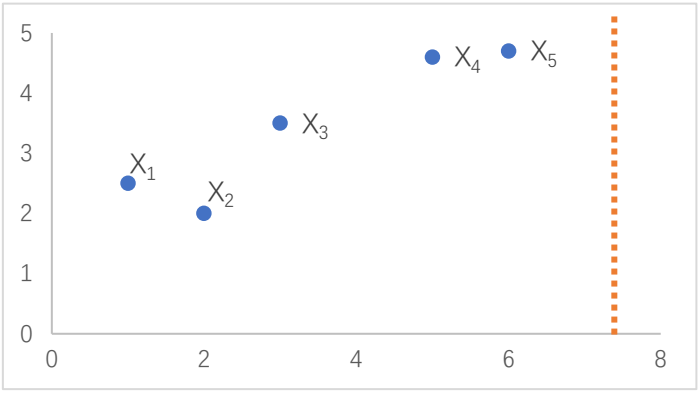

图 1.3 x_6超出了已知范围

此时如强行对y_6进行高斯过程回归，则很容易出现 Runge 现象，误差较大.

我们可以总结出高斯过程回归的优点和缺点[5]：

优点：

① 能够量化预测的不确定性，我们可以根据置信区间在一些感兴趣区域再进行拟合预测

② 可以指定不同的核函数

③ 处理小数据特别合适，由于高斯过程回归是非参数模型，计算复杂度为$O(n^3)$

缺点：

① 需要用完整的样本信息来进行预测，即非稀疏的

② 对无法被“限制”的数据点进行预测时，会出现 Runge 现象

③ 不适于处理大数据

1.4 高斯过程回归的 Python 实现

我们可以用 Python 的 scikit-learn 包中的 gaussian_process 库[6]来进行高斯过程求解，并利用 GaussianProcessRegressor 来进行高斯过程回归的求解.

与其他黑箱模型一样，我们只需要利用 GaussianProcessRegressor(kernel).fit(X，y)来进行学习，其中 kernel 为使用的核函数，X 为数据阵，y 为其隐变量函数值；另外用 GaussianProcessRegressor(kernel).predict(T)来进行预测，其中 T 为测试集.模型的全部参数可见参考文献[6]，即官方帮助文档.

2 SIR 模型：

2.1 背景：

SIR 模型是一种广泛使用于模拟传染病的发展过程的经典传染病模型.此模型可以模拟

传染病从产生、发展到最终消失的过程，是传染病模型中最基本的一种模型，为传染病动力学的研究做出了奠基性的贡献.

2.2 SIR 模型的理论[7] [8]：

2.2.1 基础定义：

SIR 模型将所有的人群分为三类：S 表示易感人群(Susceptible)，在接触患者人群后可能会受到传染；I 表示当前患者人群(Infected)，可以将疾病传染给易感人群；R 表示总恢复人群(Recovered)，治愈与死亡的人都包括在内.

SIR 模型将上述三个人群都看作时间t的函数，即$S = S(t)$， $I = I(t)$， $R = R(t)$.

2.2.2 SIR 模型的基本假设 ：

SIR 模型基于三个基本假设：

① 群体总人数不变，即假设系统不考虑人口流动的影响，上述三个人群数量之和在任何时刻都视为常数.

② t时刻的单位时间内，单个病人的传染能力与$S(t)$成正比，比例系数为β.则可认为在t时刻，每个病人使其他人感染的平均人数为$\beta S(t)$，患病人群使其他人感染的总人数为$\beta S(t)I(t)$.

③ t时刻的单位时间内，总恢复人数与总患病人数成正比，比例系数为γ.故t时刻单位时间内，总恢复人数为$\gamma I(t)$.且被治愈或死亡的患者不会再次被感染.

用微分方程，可将上述过程表示为：

$$\frac{dS(t)}{dt} = -\beta S(t)I(t)$$

$$\frac{dR(t)}{dt} = \gamma I(t)$$

$$\frac{dI(t)}{dt} = \beta S(t)I(t) - \gamma I(t)$$

2.3 基于 SIR 模型的数据处理步骤：

① 数据预处理：

由 SIR 模型假设，死亡人数与被治愈人数的和为恢复人数；由于恢复人群不再被感染，故累积恢复人数即为当前时刻现存治愈人数.现存患病人数应为累积患病人数与恢复人数之差.

② 估计恢复能力系数γ：

通常情况下，我们能获取累积感染人数和死亡人数、恢复人数$R(t)$，并借此计算当前感染人数$I(t)$，但无法判断易感人群人数$S(t)$.t时刻单位时间内新增恢复人数即为$R(t+1) - R(t)$，由每个时刻的$R(t)$与$I(t)$数据均可得到一个γ的估计：

$$\gamma = \frac{R(t+1) - R(t)}{I(t)}$$

对所有的γ求均值，记为$\bar{\gamma}$，即为对γ的估计.在之后的分析中认为γ为一常数.

③ 估计初始易感人群数量$S(0)$与传染能力系数β：

记初始易感人群数量为$S(0)$，将$S(0)$与β视为二元自变量，患病人数$I(t)$视为因变量，每给出一对$(S(0), \beta)$，通过 SIR 微分方程和高斯过程回归，即可得到$I(t)$的一个模拟分布.我们可以现在一个大区域D中取数个$(S(0), \beta)$，如D: $[0, 100000000] \times [0,1]$中按均匀分布找出

10000个点，并得到对应的模拟分布$i(t)$.选取适当的损失函数$loss(S(0),\beta)$，如均方误差来比较$I(t)$和$i(t)$，逐渐缩小D，最终得到几个较优的$(S(0),\beta)$，使均方误差最小的$(S(0),\beta)$即作为对$(S(0),\beta)$的估计，即

$$\underset{S(0),\beta}{\operatorname{argmin}} loss\big(S(0)\big) = \underset{S(0),\beta}{\operatorname{argmin}} \| I - i \|_2$$

④ 预测：

得到γ，β，$S(0)$的估计值后，给出当前患病人数，根据微分方程便能对任何给定的t算出t时刻的患病人数与恢复人数，并可画出相应的图像.

2.4 高斯过程回归与 SIR 模型的联系：

我们在估计初始易感人群数量$S(0)$与传染能力系数β的时候用到了高斯过程回归，接下来我们解释使用高斯过程回归的理由与优势:

① 其他的回归模型，如最小二乘法，要求待回归的函数是带未知参数的简单函数（即有显式表达式的函数），再对$loss$求导来求出最小值点，而 SIR 模型的微分方程没有显式解.

② 强行对$loss$求导时间复杂度极大，所用时间随着样本量的增加呈指数级增长.

③ 数据本身是带噪声的，严格地计算$loss$具体取值的最小值会极大地受到噪声的影响.

3 分析实例：

我们基于高斯过程回归和 SIR 模型，对日本的 COVID-19 进行了预测，使用的语言为 Python，具体代码请参见附录.

数据来源：https://github.com/CSSEGISandData/COVID-19 ，选取日本 2020.1.22 至 2020.6.5 的数据.

在对$(S(0),\beta)$进行估计时，分别取D为$[0,100000]\times[0,10^{-5}]$，$[3\times10^{-6},1.3\times10^{-5}]\times[15000,25000]$，$[7.4\times10^{-6},8.4\times10^{-6}]\times[18500,19500]$，能得到如下损失函数的图像:

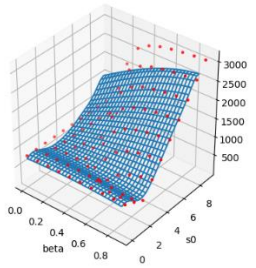

图 3.1 D为$[0,100000]\times[0,10^{-5}]$时$loss$的图像

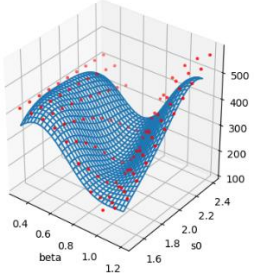

图 3.2 D为$[3\times10^{-6},1.3\times10^{-5}]\times[15000,25000]$时$loss$的图像

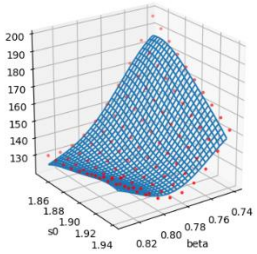

图 3.3 D为$[7.4\times10^{-6},8.4\times10^{-6}]\times[18500,19500]$时$loss$的图像

在得到所有的估计值后，我们便可以根据已有的数据进行预测，并画出图 3.4：其中将 2020.1.22 记为时间轴零点，步长为 1 天，当$I(t)<=I(0)*0.1$时停止画图；绿色代表易感人群S，蓝色代表预测出的感染人群i，橘色代表恢复人群R，红色代表真正的感染人群I.

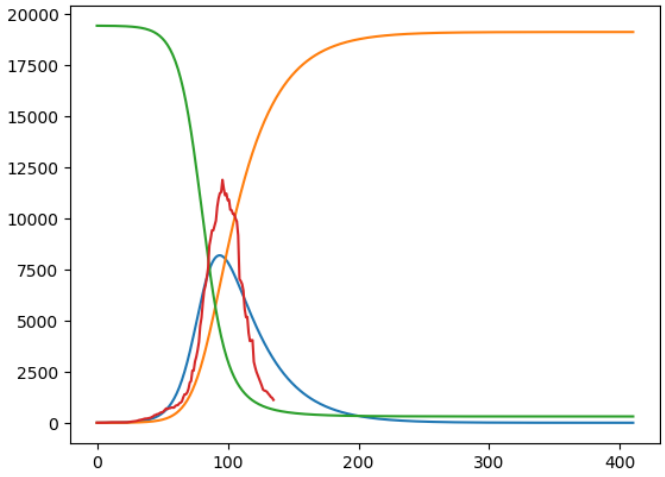

图 3.4 对日本疫情进行预测的图像

可以看到，预测感染人数和实际的最大值处差异大，是因为 SIR 模型在过了拐点后尾巴会变得平缓（参见蓝线），而实际情况不符合，在取最小的$loss$时会变成类似蓝线所示. 而疫情拐点时间较为贴合实际，借用该算法，可以在疫情早期便靠蓝色线来预测达到拐点的时间.

参考文献：

[1] Wikipedia-Gaussian process, https://en.wikipedia.org/wiki/Gaussian_process#cite_note-gpml-9

[2] Rasmussen, C.E. and Williams, C.K.I. . Gaussian processes in machine learning：MIT Press, 2006：Chapter 2, 4-5, 7-8, pp.7-30, 79-128, 151-185.

[3] 浅析高斯过程回归, https://blog.csdn.net/qq_20195745/article/details/82721666

[4] Jean-Philippe Vert, Koji Tsuda, and Bernhard Schölkopf (2004). "A primer on kernel methods". Kernel Methods in Computational Biology.

[5] Neal, Radford M. Bayesian learning for neural networks. Vol. 118. Springer Science & Business Media, 2012.

[6] sklearn.gaussian_process.GaussianProcessRegressor()官方文档, https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.GaussianProcessRegressor.html

[7] 沈朋裕-流行病的 SIR 模型, http://mathcenter.ck.tp.edu.tw/Resources/Ctrl/ePaper/eArticleDetail.aspx?id=23095cab-826c-4b28-a2f3-bdc8187afb3b

[8] 基于 SIR 模型的疫情预测, https://blog.csdn.net/chen_chen_chen_/article/details/105317181

[9] 基于 SIR 模型对新型冠状病毒疫情趋势的简单分析, https://zhuanlan.zhihu.com/p/104379096

附录：

我们给出第 3 节中的 Python 源代码：

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process.kernels import RBF, ConstantKernel as C
from sklearn.gaussian_process import GaussianProcessRegressor
from mpl_toolkits.mplot3d import Axes3D

i = [ 2, 2, 2, 2, 4, 4, 7, 7, 11, 15, 20, 20, 20, 22, 22, 22, 25, 25, 26, 26, 26
      , 28, 28, 29, 43, 59, 66, 74, 84, 94, 105, 122, 147, 159, 170, 189, 214, 228, 241
      , 256, 274, 293, 331, 360, 420, 461, 502, 511, 581, 639, 639, 701, 773, 839, 839
      , 878, 889, 924, 963, 1007, 1101, 1128, 1193, 1307, 1387, 1468, 1693, 1866, 1866, 1953
      , 2178, 2495, 2617, 3139, 3139, 3654, 3906, 4257, 4667, 5530, 6005, 6748, 7370, 7645
      , 8100, 8626, 9787, 10296, 10797, 10797, 11135, 11512, 12368, 12829, 13231, 13441, 14153
      , 13736, 13895, 14088, 14305, 14571, 14877, 15078, 15253, 15253, 15477, 15575, 15663, 15777
      , 15847, 15968, 16049, 16120, 16203, 16237, 16285, 16305, 16367, 16367, 16424, 16513, 16536
      , 16550, 16581, 16623, 16651, 16598, 16673, 16716, 16751, 16787, 16837, 16867, 16911
      , 16958 ]
d = [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1
      , 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 4, 4, 5, 6, 6, 6, 6, 6, 6, 6
      , 6, 10, 10, 15, 16, 19, 22, 22, 27, 29, 29, 29, 33, 35, 41, 42, 43, 45, 47
      , 49, 52, 54, 54, 56, 57, 62, 63, 77, 77, 85, 92, 93, 94, 99, 99, 108, 123, 143
      , 146, 178, 190, 222, 236, 236, 263, 281, 328, 345, 360, 372, 385, 394, 413, 430
      , 455, 474, 487, 536, 556, 556, 577, 590, 607, 624, 633, 657, 678, 697, 713, 725
      , 744, 749, 768, 768, 777, 796, 808, 820, 830, 846, 858, 881, 887, 894, 898, 899
      , 902, 905, 911, 916]
r = [ 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 4, 9, 9, 9
      , 9, 12, 12, 12, 13, 18, 18, 22, 22, 22, 22, 22, 22, 22, 22, 32, 32, 32, 43
      , 43, 43, 46, 76, 76, 76, 101, 118, 118, 118, 118, 118, 144, 144, 144, 150, 191
      , 232, 235, 235, 285, 310, 359, 372, 404, 424, 424, 424, 472, 472, 514, 514, 514
      , 575, 592, 622, 632, 685, 762, 762, 784, 799, 853, 901, 935, 1069, 1159, 1159, 1239
      , 1356, 1494, 1530, 1656, 1809, 1899, 1899, 2368, 2460, 2975, 3205, 3981, 4156, 4496
      , 4496, 4918, 5146, 5906, 8127, 8293, 8531, 8920, 9868, 10338, 10338, 11153, 11564, 11564
      , 11564, 12672, 13005, 13244, 13413, 13612, 13810, 13973, 14096, 14213, 14267, 14342, 14463
      , 14585, 14702, 14785, 14925 ]
for a in range(0,136):
    r[a]=d[a]+r[a]
for a in range(0,136):
    i[a]=i[a]-r[a]
date=xobs = np.array([[0],[1],[2],[3],[4],[5],[6],[7],[8],[9],[10], [11], [12],[13],[14],[15],[16],[17],[18],[19],[20], [21],
[22],[23],[24],[25],[26],[27],[28],[29],[30],[31],[32],[33],[34],[35],[36],[37],[38],[39],[40],[41],[42],[43],[44],[45],])
#数据用 japan 1.22-6.5
```

```
gama=[]

for a in range(0,135):
    gama.append((r[a+1]-r[a])/i[a])
gama_true = sum(gama)/136
print(gama_true)

#接下来将 S0 和 beta 视作二元自变量，因变量为 I[t],t=1-136，与真值的均方误差
#这样就可以用高斯过程回归，模拟出 s0 与 beta 的分布，估计 s0 与 beta 的真值
#第一次 0.00000-0.00001,000000-100000
#第二次 0.000008+-0.000005,20000+-5000
#第三次 0.0000079+-0.0000005，19000+-500
beta_min=0.0000079-0.0000005
beta_max=0.0000079+0.0000005
s0_min=19000-500
s0_max=19000+500

def y(beta0,s0):
    s1=[]
    i1=[]
    s1.append(s0)
    i1.append(i[0])
    res = 0
    for a in range(0,135):
        s1.append(s1[a]-beta0*i1[a]*s1[a])
        i1.append(i1[a]*(1-gama_true)+beta0*i1[a]*s1[a])
    for a in range(1,136):
        res+=((i[a]-i1[a])**2)
    return(res**0.5/136)

beta=[]
s=[]

for a in range(0,10):
    beta.append(beta_min+a*(beta_max-beta_min)/10)
    s.append(s0_min+a*(s0_max-s0_min)/10)
beta_s=np.zeros((100,2))

for a in range(0,10):
    for b in range(0,10):
        beta_s[10*a+b]=[beta[a],s[b]]

#我不知道为什么，直接用上面的 beta_s 做高斯回归，模型是分块片状的，下面将 beta 和 s 变换成相同尺
```

```
度的坐标
beta_s_for_train = np.zeros((100,2))
for a in range(0,100):
    beta_s_for_train[a,0]=beta_s[a,0]*100000
    beta_s_for_train[a,1]=beta_s[a,1]/10000
    #全部变化到 1 位数

l=[]

for a in range(0,100):
    l.append(y(beta_s[a,0],beta_s[a,1]))



kernel = C(0.01, (0.0001, 0.01)) * RBF(0.05, (1e-7, 10))
gp2 = GaussianProcessRegressor(kernel=kernel, n_restarts_optimizer=50, alpha=0.001)
gp2.fit(beta_s_for_train,l)

x_min, x_max = beta_s_for_train[:,0].min(), beta_s_for_train[:,0].max()
y_min, y_max = beta_s_for_train[:,1].min(), beta_s_for_train[:,1].max()
xset, yset = np.meshgrid(np.arange(x_min, x_max, (x_max-x_min)/30), np.arange(y_min, y_max,(y_max-
y_min)/30))
output, err = gp2.predict(np.c_[xset.ravel(), yset.ravel()], return_std=True)
output, err = output.reshape(xset.shape), err.reshape(xset.shape)
sigma = np.sum(gp2.predict(beta_s, return_std=True)[1])
up, down = output * (1 + 1.96 * err), output * (1 - 1.96 * err)

fig = plt.figure(figsize=(10.5, 5))
ax1 = fig.add_subplot(121, projection='3d')
surf = ax1.plot_wireframe(xset, yset, output, rstride=1, cstride=1, antialiased=True)

ax1.scatter(beta_s_for_train[:,0], beta_s_for_train[:,1], l, c='red',marker='.')
#ax1.scatter(beta_s[:,0], beta_s[:,1], l[:], c='red')

ax1.set_xlabel('beta')
ax1.set_ylabel('s0')

plt.show()

beta_true=7.90e-06
s0_true=1.94e+04

s_pred=[]
i_pred=[]
s_pred.append(s0_true)
```

```
i_pred.append(i[0])
for a in range(0,136):
    s_pred.append(s_pred[a]-beta_true*i_pred[a]*s_pred[a])
    i_pred.append(i_pred[a]*(1-gama_true)+beta_true*i_pred[a]*s_pred[a])

plt.plot(i_pred)
plt.plot(i)

#完整的预测
s_pred=[]
i_pred=[]
r_pred=[]
s_pred.append(s0_true)
i_pred.append(i[0])
r_pred.append(0)
a=0
while i_pred[a]>=i_pred[0]*0.1:
    if s_pred[a]-beta_true*i_pred[a]*s_pred[a]>0:
        s_pred.append(s_pred[a]-beta_true*i_pred[a]*s_pred[a])
    else:
        s_pred.append(0)
    i_pred.append(i_pred[a]*(1-gama_true)+beta_true*i_pred[a]*s_pred[a])
    r_pred.append(r_pred[a]+gama_true*i_pred[a])
    a+=1

plt.plot(i_pred)
plt.plot(r_pred)
plt.plot(s_pred)
plt.plot(i)
```