连续变量预测：线性回归

回归（**regression**）是统计学中最常用的**拟合**（**fitting**）和**预测**（**predicting**）方法。针对不同的数据通常有不同的回归方法，比如针对连续变量预测的线性回归、分位数回归，以及针对离散变量的Logistic回归等。

通常回归方法对特征的函数假定非常强烈，因而在以上**参数**（**parametric**）回归基础上，还有对函数形式假设更加宽松的**非参数**（**nonparametric**）回归。

这里我们首先介绍参数回归中最简单的一个类型：针对连续的预测目标（被解释变量）的线性回归。

线性回归

针对连续的预测目标，最常用、最简单的方法是使用普通线性回归。

我们不不妨先从总体的角度考虑预测问题。如果我们有连续变量y以及一些特征x，我们的目的是使用特征x对y进行预测，意味着我们需要使用特征x的一个函数：$h\left(x\right)$对y进行预测，那么预测误差为：

$$y-h\left(x\right)$$

误差必然越‘小’越好，但是有两个困难阻碍了我们定义‘小’：

- 误差可正可负
- 误差是随机的

为了克服以上第一个困难，我们可以把误差求平方，从而误差的平方都为正数，且只有当误差为0时，误差的平方也为0，达到了完美预测，因而预测误差的平方应该是越小越好。

为了克服第二个困难，我们可以对误差平方求期望。如此，就诞生了**均方误差**（**mean squared error，MSE**）：

$$\mathbb{E}\left[\left(y-h\left(x\right)\right)^{2}\right]$$

均方误差越小说明预测误差越小，当均方误差为0时，模型达到了完全拟合。当然完全拟合或者完美预测在现实中是不可能达到的，所以均方误差总是大于0的。

如果一个函数$h^{*}\left(x\right)$能够使得以上均方误差最小：

$$h^{*}\left(x\right)=\arg\min_{h\in\mathbb{H}}\left\{\mathbb{E}\left[\left(y-h\left(x\right)\right)^{2}\right]\right\}$$

那么我们称$h^{*}\left(x\right)$为y给定x的**条件期望**（**conditional expectation**），并记为：$\mathbb{E}\left(y|x\right)$。

然而$h\left(x\right)$的可能性太多，为了简化问题，我们必须对$h\left(x\right)$做一定的假设。其中最强的一个假设是假设$h\left(x\right)$为x的线性函数，即：

$$h(x) = \beta_1 \times x_1 + \beta_2 \times x_2 + \ldots + \beta_K \times x_K = x'\beta$$

其中

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_K \end{bmatrix}, \beta = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_K \end{bmatrix}$$

不是一般性，我们通常会假设第一个变量$x_1 = 1$，从而β_1为**常数项**（**constant term**）。如此，我们将寻找函数$h^*(x)$的问题转换为了寻找参数β_0的问题：

$$\mathbb{E}(y|x) = x'\beta_0 = \arg\min_{\beta} \left\{ \mathbb{E}\left[\left(y - x'\beta \right)^2 \right] \right\}$$

只要β_0确定了，那么$h^*(x)$就确定了。

以上是总体层面的讨论。如果考虑样本层面，我们需要使用数据对β_0进行估计，那么我们可以将以上的期望符号换位平均符号，即：

$$\hat{\beta} = \arg\min_{\beta} \frac{1}{N} \sum_{i=1}^{N} \left(y_i - x_i'\beta \right)^2$$

以上即经典的**普通最小二乘法**（**ordinary least squares**）。如果我们记：

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}, X = \left[x_1, x_2, \cdots x_N \right]' = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1K} \\ x_{21} & x_{22} & \cdots & x_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NK} \end{bmatrix}$$

其中：

$$x_i = \begin{bmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{iK} \end{bmatrix}$$

那么最小二乘法的解可以写为：

$$\hat{b} = \left(X'X \right)^{-1} X'Y = \left[\sum_{i=1}^{N} \left(x_i x_i' \right) \right]^{-1} \left[\sum_{i=1}^{N} \left(x_i y_i \right) \right]$$

In [1]:
```
import pandas as pd

raw_data = pd.read_csv('csv/cfps_adult.csv')
raw_data.head()
```

Out[1]:

	pid	fid14	provcd14	cfps_birthy	cfps_minzu	cfps_party	cfps_gender
0	100051501	100051	11	1969	-8	-8	0
1	100051502	100051	11	1966	-8	-8	1
2	100453431	100453	43	1981	0	0	1
3	101129501	101129	13	1990	-8	-8	0
4	103671501	103671	21	1988	-8	-8	0

5 rows × 90 columns

In [2]:
```
## 挑选被解释变量和解释变量
subdata = raw_data.loc[:, ['qp102', 'qp101']]
subdata = subdata.dropna()
subdata.head()
```

Out[2]:

	qp102	qp101
0	190.0	164.0
1	140.0	169.0
2	130.0	180.0
3	89.0	155.0
4	100.0	160.0

In [3]:
```
subdata.describe()
```

Out[3]:

	qp102	qp101
count	33476.000000	33476.000000
mean	121.267911	160.219590
std	24.987053	26.447926
min	-8.000000	-9.000000
25%	106.000000	158.000000
50%	120.000000	164.000000
75%	136.000000	170.000000
max	260.000000	216.000000

In [4]:
```
subdata = subdata[(subdata['qp101'] >= 130) & (subdata['qp102'] > 0)]
subdata.describe()
```

Out[4]:

	qp102	qp101
count	32491.000000	32491.000000
mean	122.582764	164.221846
std	22.481906	8.054319
min	50.000000	130.000000
25%	108.000000	160.000000
50%	120.000000	165.000000
75%	137.000000	170.000000
max	260.000000	216.000000

In [5]:
```
y = subdata.iloc[:, 0]
X = subdata.iloc[:, 1:]
## 创建模型对象
from sklearn import linear_model

reg = linear_model.LinearRegression()
## 进行回归
reg.fit(X, y)  ## 训练模型
## 查看系数
print("系数=", reg.coef_)
print('截距项=', reg.intercept_)
```

```
系数= [1.58369352]
截距项= -137.49430839521733
```

接下来可以进行预测：

$$\hat{y} = x'\hat{\beta}$$

，比如如果我们知道某个人身高为175cm，而不知道其具体身高，那么对其身高的最优预测为：

$$\hat{y}_{175} = -137.4943 + 1.58369 \times 175 = 139.652$$

使用scikit-learn做预测：

In [6]:
```
import numpy as np

reg.predict(np.array([[170], [175], [180]]))
```

```
/opt/Anaconda/lib/python3.11/site-packages/sklearn/base.py:493: UserWarning: X do
es not have valid feature names, but LinearRegression was fitted with feature nam
es
  warnings.warn(
```

Out[6]:
```
array([131.73358941, 139.65205699, 147.57052457])
```

以及对所有样本进行样本内预测：

In [7]:
```
yhat = reg.predict(X)
yhat
```

Out[7]:
```
array([122.23142831, 130.14989589, 147.57052457, ..., 126.98250886,
       147.57052457, 122.23142831])
```

当然我们可以包括更多的解释变量，比如：

In [8]:
```
subdata = raw_data.loc[:, ['qp102', 'qp101', 'cfps_gender']]
subdata = subdata.dropna()
y = subdata.iloc[:, 0]
X = subdata.iloc[:, 1:]
## 创建模型对象
reg = linear_model.LinearRegression()
## 进行回归
reg.fit(X, y)  ## 训练模型
## 查看系数
print("系数=", reg.coef_)
print('截距项=', reg.intercept_)
```

```
系数= [ 0.23227872 16.08926018]
截距项= 76.06295134743743
```

如果需要预测，同样可以使用predict()方法：

In [9]:
```
reg.predict(np.array([[170, 0], [175, 1], [180, 1]]))
```

```
/opt/Anaconda/lib/python3.11/site-packages/sklearn/base.py:493: UserWarning: X do
es not have valid feature names, but LinearRegression was fitted with feature nam
es
  warnings.warn(
```

Out[9]:
```
array([115.55033371, 132.80098749, 133.96238109])
```

以及包含平方项、交叉项等等：

In [10]:
```
subdata = raw_data.loc[:, ['qp102', 'qp101', 'cfps_gender']]
subdata = subdata.dropna()
subdata['qp101_2'] = subdata['qp101']**2
subdata['qp101_gender'] = subdata['qp101'] * subdata['cfps_gender']
y = subdata.iloc[:, 0]
X = subdata.iloc[:, 1:]
## 创建模型对象
reg = linear_model.LinearRegression()
## 进行回归
reg.fit(X, y)  ## 训练模型
## 查看系数
print("系数=", reg.coef_)
print('截距项=', reg.intercept_)
```

```
系数= [-1.03361111 -0.93091655  0.00736492  0.03281556]
截距项= 90.39354698294906
```

以及样本内预测：

In [11]:
```
reg.predict(X)
```

Out[11]:
```
array([118.9682252 , 130.67767589, 147.94285447, ..., 127.73004044,
       147.94285447, 118.9682252 ])
```

函数形式的讨论

虽然线性回归方法非常简单，但是线性函数的假设太强，很多时候我们需要详细讨论函数形式的问题。在有的情况下，函数形式可能会非常有问题，在使用时需要仔细讨论。比如以下例子：

- （支撑集问题）支撑集（support）即一个随机变量的取值范围。如果被解释变量为家庭的储蓄率（saving_rate），我们知道储蓄率的取值范围应该在0到1之间。此时，如果我们选取家庭资产规模（wealth）作为解释变量：

  $$saving\_rate_i = \beta_0 + \beta_1 \cdot wealth_i + u_i$$

  由于家庭资产规模的取值范围应为$wealth_i \in [0, \infty)$，因而不管回归得到的系数$\hat{\beta}_1$是正或者负，对于一个资产规模足够大的家庭，总会使得预测的储蓄率超过1（或者低于0）。
- （理论模型的函数形式）在国际贸易理论中（Head and Mayer, 2014），双边贸易与两个国家的GDP之间存在着被称为"引力模型"的关系，即：

  $$X_{ni} = GY_i^a Y_n^b \phi_{ni}$$

  其中下标i代表国家，而n代表出口目的地国，X_{ni}为两国之间的贸易额，G为常数，Y为国家的GDP，ϕ_{ni}则是两国之间贸易成本的函数。双边贸易额与GDP之间的关系并非简单的线性关系。

解决函数形式的一些方法：

解决方案：

- 使用非参数回归（kernel, sieve）
- 引入多项式（平方项、交叉项）
- 对数据进行变换：
  - 对数变换：将正数变换到$(-\infty, \infty)$上——具有弹性解释
  - Box-Cox变换：

    $$y(\lambda) = \begin{cases} \frac{y^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \ln y & \lambda = 0 \end{cases}$$

  - Logistic逆变换：使用

    $$f(x) = \ln \frac{x}{1-x}$$

    将$(0, 1)$区间上的实数映射到$(-\infty, \infty)$上

In [12]:
```
log_subdata = raw_data.loc[:, ['qp102', 'qp101', 'cfps_gender']]
log_subdata['qp102'] = np.log(log_subdata['qp102'])
log_subdata['qp101'] = np.log(log_subdata['qp101'])
log_subdata = log_subdata.dropna()
y = log_subdata.iloc[:, 0]
X = log_subdata.iloc[:, 1:]
reg.fit(X, y)  ## 训练模型
## 查看系数
print("系数=", reg.coef_)
print('截距项=', reg.intercept_)
```

```
系数= [1.74333395 0.04237759]
截距项= -4.119634539552865
```

```
/opt/Anaconda/lib/python3.11/site-packages/pandas/core/arraylike.py:399: RuntimeW
arning: invalid value encountered in log
  result = getattr(ufunc, method)(*inputs, **kwargs)
```

In [13]:
```
reg.predict(X)
```

Out[13]:
```
array([4.77113575, 4.86586964, 4.97580103, ..., 4.84511542, 4.97580103,
       4.77113575])
```

模型评价方法

通常选择不同的模型是在「大模型」和「小模型」之间进行选择。小模型更容易欠拟合，而大模型更容易过拟合，评价模型主要是为了使模型能够恰好拟合，达到最优的预测效果。

在线性回归中，我们通常会使用R^2作为拟合程度的一个度量，即

$$R^2 = 1 - \frac{\hat{e}'\hat{e}}{Y'M_0Y} = 1 - \frac{RSS}{TSS}$$

其中RSS为残差平方和，TSS为总平方和。然而R^2一般不能作为模型选择的标准，过高的R^2可能是过拟合的体现。为此针对线性回归，至少有这些标准可以使用：

- 调整的R^2，即：

$$\bar{R}^2 = 1 - \frac{\hat{e}'\hat{e}/(N-K)}{Y'M_0Y/(N-1)} = 1 - \frac{N-1}{N-K}(1-R^2)$$

- 赤池信息准则（Akaike information creterion,AIC）：

$$AIC = -2Log\_Likelihood + 2K$$

- 贝叶斯信息准则（Bayesian information creterion,BIC，又称施瓦茨信息准则，Schwarz information creterion,SIC）：

$$BIC = -2Log\_Likelihood + \ln(N)K$$

其中AIC、BIC适用于所有使用极大似然估计的统计模型。

以上的准则都是在拟合程度的度量基础上，对变量个数进行惩罚，避免选出比较「大」的模型。

比如我们使用如下代码计算以上指标，首先计算回归：

In [14]:
```
subdata = raw_data.loc[:, ['qp102', 'qp101', 'cfps_gender']]
subdata = subdata.dropna()
subdata['qp101_2'] = subdata['qp101']**2
subdata['qp101_gender'] = subdata['qp101'] * subdata['cfps_gender']
y = subdata.iloc[:, 0]
X1 = subdata.iloc[:, 1:2]
X2 = subdata.iloc[:, 1:3]
X3 = subdata.iloc[:, 1:4]
```

```
X4 = subdata.iloc[:, 1:]
## 回归
reg1 = linear_model.LinearRegression().fit(X1, y)
reg2 = linear_model.LinearRegression().fit(X2, y)
reg3 = linear_model.LinearRegression().fit(X3, y)
reg4 = linear_model.LinearRegression().fit(X4, y)
## 查看系数
print("系数=", reg1.coef_, '截距项=', reg1.intercept_)
print("系数=", reg2.coef_, '截距项=', reg2.intercept_)
print("系数=", reg3.coef_, '截距项=', reg3.intercept_)
print("系数=", reg4.coef_, '截距项=', reg4.intercept_)
```

```
系数= [0.31582403] 截距项= 70.66671440178149
系数= [ 0.23227872 16.08926018] 截距项= 76.06295134743743
系数= [-1.04245345  4.27682516  0.00746794] 截距项= 89.23758538186534
系数= [-1.03361111 -0.93091655  0.00736492  0.03281556] 截距项= 90.39354698294906
```

接下来使用score函数计算R^2和$\bar{R}^2$以及AIC、BIC：

In [15]:
```
def r2_ic(reg, x, y):
    N = x.shape[0]
    K = x.shape[1] + 1
    result = dict()
    result['r2'] = reg.score(x, y)
    result['r2_adjusted'] = 1 - (N - 1) / (N - K) * (1 - result['r2'])
    ## 计算似然函数
    e_hat = y - reg.predict(x)
    sigma2_hat = np.sum(e_hat**2) / N
    ll = -0.5 * N * np.log(sigma2_hat)
    result['aic'] = -2 * ll + 2 * K
    result['bic'] = -2 * ll + np.log(N) * K
    return result


print(r2_ic(reg1, X1, y))
print(r2_ic(reg2, X2, y))
print(r2_ic(reg3, X3, y))
print(r2_ic(reg4, X4, y))
```

```
{'r2': 0.11174896601143425, 'r2_adjusted': 0.11172243046043973, 'aic': 211511.556
1574294, 'bic': 211528.3933255157}
{'r2': 0.20758038959695013, 'r2_adjusted': 0.20753304280339102, 'aic': 207691.824
3099361, 'bic': 207717.08006206556}
{'r2': 0.29857574006882537, 'r2_adjusted': 0.2985128734107293, 'aic': 203610.4841
738876, 'bic': 203644.15851006022}
{'r2': 0.2987764278913496, 'r2_adjusted': 0.2986926271597182, 'aic': 203602.90482
625779, 'bic': 203644.99774647356}
```

为了展示使用以上标准进行模型选择的过程，我们利用以上函数使用一个模拟的数据，对多项式的阶数进行选择。

在以下程序中，我们首先产生了一个伪数据集：

$$y = e^x + u$$

其中$x \sim U(0,3)$，$u \sim N(0,4)$。接着，我们从x的一次方开始，逐渐向回归中添加$x^2, x^3, \ldots, x^{10}$对模型进行拟合，首先生成数据：

In [16]:
```
N = 100  #样本量
np.random.seed(1900)
x = pd.Series(np.random.random(N)) * 3
X = pd.DataFrame()
X['y'] = np.exp(x) + np.random.normal(0, 2, N)
for i in range(10):
    var_name = 'x' + str(i + 1)
    X[var_name] = x**(i + 1)

X.head()
```

Out[16]:

	y	x1	x2	x3	x4	x5	x6	
0	5.188967	0.996738	0.993487	0.990246	0.987016	0.983797	0.980588	
1	7.257260	2.142828	4.591711	9.839245	21.083807	45.178965	96.810739	20
2	3.187709	1.135042	1.288321	1.462300	1.659772	1.883912	2.138320	
3	0.669471	0.747247	0.558378	0.417247	0.311786	0.232982	0.174095	
4	12.051389	2.507959	6.289858	15.774706	39.562317	99.220667	248.841364	62

接下来逐步加入多项式进行回归，并报告统计量：

In [17]:
```
print(" i  | R^2   |  R^2_a |  AIC   |  BIC ")
for i in range(10):
    reg = linear_model.LinearRegression().fit(X.iloc[:, 1:i + 2], X['y'])
    result = r2_ic(reg, X.iloc[:, 1:i + 2], X['y'])
    print("%2d | %1.4f | %1.4f | %3.2f | %3.2f" %
          (i + 1, result['r2'], result['r2_adjusted'], result['aic'],
           result['bic']))
```

```
i  | R^2   |  R^2_a |  AIC   |  BIC
 1 | 0.6895 | 0.6863 | 200.26 | 205.47
 2 | 0.8194 | 0.8157 | 148.07 | 155.89
 3 | 0.8266 | 0.8212 | 146.00 | 156.42
 4 | 0.8282 | 0.8210 | 147.08 | 160.11
 5 | 0.8284 | 0.8193 | 148.94 | 164.57
 6 | 0.8294 | 0.8184 | 150.35 | 168.58
 7 | 0.8295 | 0.8166 | 152.29 | 173.13
 8 | 0.8302 | 0.8152 | 153.93 | 177.38
 9 | 0.8308 | 0.8139 | 155.54 | 181.59
10 | 0.8308 | 0.8118 | 157.54 | 186.19
```

从以上结果看出，以$\bar{R}^2$为标准，似乎3阶多项式是比较好的；以AIC为标准，3阶多项式比较好；而以BIC为标准，似乎2阶多项式比较好。一般而言，BIC会挑选出比AIC更少的多项式阶数，或者更小的模型，不过在这里区别不大。

交叉验证

然而以上标准要么只适用于线性回归（调整的R^2），要么只适用于极大似然估计，通用性比较差，特别是很多机器学习方法并不依赖于统计方法，这些模型评价的指标通常不能直接用于机器学习中。

注意到，不管是欠拟合还是过拟合，都会导致**样本外**预测的误差变大，因而我们可以只使用一部分样本进行估计，而在另外一部分样本中检验模型的预测能力。我们可以将样本分为两部分：**训练集（training set）**用于估计模型、**验证集（validation set）**用于评价模型的样本外预测能力。

比如在以上多项式的例子中，我们可以首先随机将样本分为两部分，一部分训练模型一部分测试：

In [18]:
```
## 产生一个随机顺序，并排序，从而顺序是随机的
X['random_order'] = np.random.random(N)
X = X.sort_values(['random_order'])
X = X.drop('random_order', axis=1)
## 区分训练集和测试集
X_train = X.iloc[0:80, :]
X_test = X.iloc[80:, :]
## 使用训练集回归
print(" i | R^2    |  MSE")
for i in range(10):
    # 先使用训练集训练模型
    reg = linear_model.LinearRegression().fit(X_train.iloc[:, 1:i + 2],
                                              X_train['y'])
    # 再使用测试集查看在测试集中预测的R2
    r2 = reg.score(X_test.iloc[:, 1:i + 2], X_test['y'])
    # 也可以计算均方误差
    errors = X_test['y'] - reg.predict(X_test.iloc[:, 1:i + 2])
    mse = np.mean(errors**2)
    print("%2d | %1.4f | %1.4f" % (i + 1, r2, mse))
```

```
i | R^2    |  MSE
1 | 0.6965 | 7.9009
2 | 0.8054 | 5.0663
3 | 0.7973 | 5.2770
4 | 0.7978 | 5.2640
5 | 0.7983 | 5.2493
6 | 0.7950 | 5.3352
7 | 0.7823 | 5.6670
8 | 0.7811 | 5.6976
9 | 0.7820 | 5.6759
10 | 0.7721 | 5.9315
```

看起来似乎使用2阶多项式比较好。

或者我们可以使用Scikit-Learn中的train_test_split函数自动帮我们区分训练集和测试集：

In [19]:
```
from sklearn.model_selection import train_test_split
## 获取随机的训练集和测试集，选取20%的数据作为最终的测试集
X_train, X_test, y_train, y_test = train_test_split(X.iloc[:, 1:],
                                                    X['y'],
                                                    test_size=0.2)
## 使用训练集回归
print(" i | R^2    |  MSE")
for i in range(10):
    # 先使用训练集训练模型
    reg = linear_model.LinearRegression().fit(X_train.iloc[:, 0:i + 1],
                                              y_train)
    # 再使用测试集查看在测试集中预测的R2
```

```
    r2 = reg.score(X_test.iloc[:, 0:i + 1], y_test)
    # 计算均方误差
    errors = y_test - reg.predict(X_test.iloc[:, 0:i + 1])
    mse = np.mean(errors**2)
    print("%2d | %1.4f | %1.4f" % (i + 1, r2, mse))
```

```
 i | R^2    |  MSE
 1 | 0.7210 | 5.5723
 2 | 0.8445 | 3.1064
 3 | 0.8534 | 2.9268
 4 | 0.8557 | 2.8814
 5 | 0.8518 | 2.9595
 6 | 0.8526 | 2.9447
 7 | 0.8527 | 2.9411
 8 | 0.8543 | 2.9089
 9 | 0.8555 | 2.8868
10 | 0.8505 | 2.9858
```

不过这样区分训练集和测试集具有随机性，为了克服这一问题，将以上方法进行推广，就得到了所谓的**交叉验证**（**cross validation**）。交叉验证常见的用法如：

- **S折交叉验证**（**S-fold cross validation**）：将N个样本随机的分为大小相同的S组，然后利用$S-1$组的数据对数据进行拟合，并使用该模型对剩下的一组计算目标函数值（在这里即预测误差的度量，如误差平方）。将这一过程对S种组合重复进行，最终得到了N个目标函数值的加总（如均方误差）。对于不同的模型，选择使得验证集目标函数最优的那个，或者预测误差最小的那个模型。
- **留一验证**（**leave-one-out cross validation**）：即$S=N$的S折交叉验证的特殊情形，每一次都用$N-1$个样本训练模型，对剩下的一个样本进行预测。

可见，交叉验证的主要缺点是运算时间：对于S折交叉验证，必须重复估计模型S次，而对于留一验证，必须对模型估计N次。但是由于交叉验证度量了样本外预测的效果，因而可以有效避免欠拟合、过拟合。

Scikit-learn里面包含了一些帮助我们完成交叉验证的方法，详情可参照：https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation

比如我们可以使用如下代码进行一个交叉验证的抽样：

In [20]:
```
from sklearn.model_selection import KFold

kf = KFold(n_splits=5)
for train, test in kf.split(X):
    print('---\n', train, "\n", test)
```

```
---
 [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 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
 92 93 94 95 96 97 98 99]
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
---
 [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 40 41 42 43
 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
 92 93 94 95 96 97 98 99]
 [20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39]
---
 [ 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 60 61 62 63 64 65 66 67
 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
 92 93 94 95 96 97 98 99]
 [40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59]
---
 [ 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 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 80 81 82 83 84 85 86 87 88 89 90 91
 92 93 94 95 96 97 98 99]
 [60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79]
---
 [ 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 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71
 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99]
```

接下来对每个训练集和测试集进行训练和验证（一个好的习惯是先随机排序），比如我们进行一个10折的交叉验证：

In [21]:
```
## 产生一个随机顺序，并排序，从而顺序是随机的
X['random_order'] = np.random.random(N)
X = X.sort_values(['random_order'])
X = X.drop('random_order', axis=1)
## 区分训练集和测试集
kf = KFold(n_splits=10)
## 使用训练集回归
CV_error = np.array([])
for train, test in kf.split(X):
    X_train = X.iloc[train, :]
    X_test = X.iloc[test, :]
    # 先使用训练集训练模型
    reg = linear_model.LinearRegression().fit(X_train.iloc[:, 1:],
                                              X_train['y'])
    # 接下来在验证集上进行预测，得到预测误差
    pre_error = X_test['y'] - reg.predict(X_test.iloc[:, 1:])
    CV_error = np.concatenate([CV_error, pre_error])
print("预测误差=\n", CV_error)
print("均方误差MSE=", np.mean(CV_error**2))
```

```
预测误差=
 [-0.97941084  0.32874278  1.477193   -3.14933293  0.46618479  1.40394522
 -0.21679122  1.21180279  3.4972513  -1.56271975  1.84773268  0.51310592
 -2.08866055  0.13658632 -0.17915758  2.4379034  -0.34805687 -0.44027518
  1.59031476 -0.80566504  0.64889446 -0.44540489  2.8423408  -0.8616925
  0.12735213 -2.04989713 -2.87733855 -1.39231671  0.97359429  0.90986541
  2.70436355 -0.57201358 -1.78529588  2.32833123  1.66283294 -1.83501106
 -1.11321433 -3.37964145  0.61982464 -0.03179838 -0.63886976 -0.28310296
  1.35407805 -1.50679739 -1.79239822  3.7199329   1.39225659  2.30569115
 -0.28060709  0.22218465 -5.3584453   0.31956103  0.38424195  3.03171169
 -1.33503162  2.73426759 -0.4662434   0.560205   -2.63945096 -0.50945458
 -1.38566038  0.803114    1.3793609  -4.34613695  1.70817211 -0.13837416
  4.21096444  0.88537031  0.04219259 -4.95117825  0.7421255  -0.26115026
 -0.41656705 -4.67789521 -0.59473336 -0.01690212  1.82349422  5.7589625
 -1.39339433 -4.34252865  0.85450942  1.03111986 -1.1397553  -0.65685293
 -3.24076097  2.18782303  5.2745927  -0.24446157 -4.81707257 -0.58814495
  4.53148109  0.78275333  3.74828052  0.59474008  3.08925621 -0.13007291
 -0.76838144 -2.94042547 -2.4511454  -0.18668522]
均方误差MSE= 4.746266884535841
```

为了评价所有阶数模型的交叉验证结果，可以嵌套一个循环：

In [22]:
```
## 产生一个随机顺序，并排序，从而顺序是随机的
X['random_order'] = np.random.random(N)
X = X.sort_values(['random_order'])
X = X.drop('random_order', axis=1)
## 区分训练集和测试集
kf = KFold(n_splits=10)  ##设定为N则变成了留一验证
print(" i | MSE")
for i in range(10):
    CV_error = np.array([])
    for train, test in kf.split(X):
        X_train = X.iloc[train, :]
        X_test = X.iloc[test, :]
        # 先使用训练集训练模型
        reg = linear_model.LinearRegression().fit(X_train.iloc[:, 1:i + 2],
                                                  X_train['y'])
        # 接下来在验证集上进行预测，得到预测误差
        pre_error = X_test['y'] - reg.predict(X_test.iloc[:, 1:i + 2])
        CV_error = np.concatenate([CV_error, pre_error])
    print("%2d | %1.4f" % (i + 1, np.mean(CV_error**2)))
```

```
 i | MSE
 1 | 7.3474
 2 | 4.3243
 3 | 4.2543
 4 | 4.2508
 5 | 4.3852
 6 | 4.4638
 7 | 4.5507
 8 | 4.6245
 9 | 4.9105
10 | 5.0725
```

可以见从交叉验证的角度来看，3阶多项式是预测效果比较理想的。

模型选择的常用方法：正则化

虽然交叉验证提供了一个评价模型的很好的标准，但是我们潜在的有非常多模型可以选择，而交叉验证本身就是非常耗时的一个计算，因而大多数情况我们并不会直接使用交叉验证去评价每一个潜在的模型。而对于大量的基于参数的模型（比如各种回归模型、神经网络模型等等），有一个比较通用的限制模型不能太多「大」的方法，即**正则化**（**regularization**）。

正则化可以看做是**收缩估计量**（**shrinkage estimator**）的一种推广，其基本原理是统计学中经典的**偏差-方差权衡**（**bias-variance tradeof**）。

对于预测问题，一个通常的性能度量是所谓的「**均方误差**」（**mean squared error, MSE**）：

$$MSE = E[y - f(x)]^2$$

而均方误差可以被分解为偏差的平方加上方差：

$$MSE = Bias^2 + Variance$$

。问题是，bias和variance通常不太可能同时降低，如果要做到无偏，那么variance就会变大；如果想要降低variance，bias也会变大，就产生了矛盾。

原始的统计学中，对于无偏性（或者渐进无偏性）是非常执着的，通常要求预测无偏，然而很快统计学家们发现无偏并不代表MSE最小，如果放弃无偏能够导致方差大幅度减小，反而会改进MSE。

这一点可以从以下例子中看到。比如如果我们需要评估10个学生的真实计算机水平（图中黑点），但是我们只进行了一次考试，考试成绩为图中的红点。传统的OLS或者MLE告诉我们，红点是黑点的无偏估计。然而Stein(1956)指出红点任何时候都不是最好的预测，并进一步在James and Stein(1961)中提出了收缩估计量，即James-Stein估计量，如下图中的白点。

可以看到白色的点虽然不是无偏的，但是预测效果却好很多，即通过放弃无偏性，降低方差，从而使得总的MSE降低。

更令人震惊的是，James-Stein估计量并不要求向任何一个具体的点收缩，只要按照他们的方法进行收缩，那么就可以使得MSE降低。也就是说，**重要的是收缩本身，而不是往哪里收缩。**

Stein的发现导致了统计学的一场重大变革，从此经验贝叶斯以及收缩估计量蓬勃发展起来。

比如，扩展到线性回归上，常见的模型有**岭回归**（**ridge regression**）和**LASSO回归**（**Least absolute shrinkage and selection operator**）。

其中岭回归的最优化问题为：

$$\hat{\beta}^r(\lambda) = \arg\min_{\beta} \sum_{i=1}^{N} \left(y_i - x_i'\beta\right)^2 + \lambda\|\beta\|_2^2 = \arg\min_{\beta} (Y - X\beta)'(Y - X\beta) + \lambda\beta'$$

而Lasso回归的优化问题为：

$$\hat{\beta}^{lasso}(\lambda)=\arg\min_{\beta}\sum_{i=1}^{N}\left(y_i-x_i'\beta\right)^2+\lambda\|\beta\|_1=\arg\min_{\beta}\sum_{i=1}^{N}\left(y_i-x_i'\beta\right)^2+\lambda\sum_{k=1}^{K}$$

两者的区别在于使用的「惩罚项」或者「正则化项」不同，岭回归使用了L2范数，而Lasso回归使用了L1范数，注意点不同导致了其估计量有非常不同的性质：

可以看到，虽然岭回归都会使得系数向0收缩，但是很难完全收缩到0；而Lasso回归则会使得部分系数直接变为0。因而**Lasso回归在进行收缩的同时，完成了变量选择**，或者说挑出了一个比较小的模型。

正是由于这些特点，在高维问题上（$K\approx N$或者$K\geq N$）的情况下，岭回归和Lasso回归仍然可以使用。当然，由于Lasso还额外可以做变量选择，因而用的更多。

注意其中涉及到一个参数λ的选取，由于只是一个一维参数，从而可以很方便的使用交叉验证法进行选取。

此外，需要提示的是，由于上述方法都是基于参数的大小的，因而对数据进行标准化（通常减均值除以标准差）就非常重要了！！否则，方差小的变量系数会更大，更容易被筛选出来。因而做岭回归、Lasso回归之前，**一定要对数据进行标准化。**

在Lasso的基础上后续还发展了很多扩展的方法，比如adaptive Lasso、rLasso等等，我们在此不再赘述。

在机器学习领域，其实多数参数模型包括Logistic回归、神经网络等等，都可以在目标函数后面加入L1范数进行特征（节点）的选取。

最后，由于使用L1范数的正则化方法相当于在原始的最优化问题上加了一个约束，而这个约束的可行集是一个凸集，在很多算法里面，结合L1范数正则化的优化问题是一个凸优化（convex optimization）问题，在优化理论和计算机算法中有成熟的理论和算法解决这类优化问题，因而在算法层面也是完全可行的。

一个预测实例

线性回归是非常经典的对连续因变量预测的方法，当然在Scikit-Learn中自然也有非常成熟的应用。为了展示Python中线性回归的用法，我们不妨以计量经济学中的HCW方法（Hsiao, Ching and Wan, 2012）为例。

比如，为了评价香港回归祖国对于香港经济的影响，我们可以利用其它国家和地区的经济增长率，使用1997年之前的数据进行回归，最后预测1997年之后香港的GDP增速，并与现实的香港GDP增速相比较。下面给出了具体过程，具体使用方法可以查询文档（https://scikit-learn.org/stable/modules/linear_model.html#lasso ）。

首先，我们先准备数据：

In [23]:
```
import pandas as pd

raw_data = pd.read_csv('csv/hcw.csv')
raw_data.head()
```

Out[23]:

	time	HongKong	Australia	Austria	Canada	Denmark	Finland	Fra
0	1993q1	0.062	0.040489	-0.013084	0.010064	-0.012292	-0.028357	-0.015
1	1993q2	0.059	0.037857	-0.007581	0.021264	-0.003093	-0.023397	-0.014
2	1993q3	0.058	0.022509	0.000543	0.018919	-0.007764	-0.006018	-0.016
3	1993q4	0.062	0.028747	0.001181	0.025317	-0.004049	-0.004774	-0.007
4	1994q1	0.079	0.033990	0.025511	0.043567	0.031094	0.012886	0.003

5 rows × 26 columns

In [24]:
```
data = raw_data.set_index(pd.to_datetime(raw_data['time']))
data.index = data.index.to_period('Q')
data
```

```
/tmp/ipykernel_54132/426571931.py:1: UserWarning: Could not infer format, so each element will be parsed individually, falling back to `dateutil`. To ensure parsing is consistent and as-expected, please specify a format.
  data = raw_data.set_index(pd.to_datetime(raw_data['time']))
```

Out[24]:

	time	HongKong	Australia	Austria	Canada	Denmark	Finland
time							
1993Q1	1993q1	0.062	0.040489	-0.013084	0.010064	-0.012292	-0.028357
1993Q2	1993q2	0.059	0.037857	-0.007581	0.021264	-0.003093	-0.023397
1993Q3	1993q3	0.058	0.022509	0.000543	0.018919	-0.007764	-0.006018
1993Q4	1993q4	0.062	0.028747	0.001181	0.025317	-0.004049	-0.004774
1994Q1	1994q1	0.079	0.033990	0.025511	0.043567	0.031094	0.012886
...	...	...	...	...	...	...	...
2007Q1	2007q1	0.055	0.058013	0.036198	0.030712	0.033134	0.047340
2007Q2	2007q2	0.062	0.059519	0.032570	0.039827	-0.007169	0.046808
2007Q3	2007q3	0.068	0.056649	0.031558	0.034742	0.013517	0.045647
2007Q4	2007q4	0.069	0.045825	0.019095	0.038128	0.023794	0.044177
2008Q1	2008q1	0.073	0.027523	0.017431	0.029217	-0.005200	0.023732

61 rows × 26 columns

In [25]:
```
pre1997 = data[data.index <= '1997Q2']
pre1997
```

Out[25]:

	time	HongKong	Australia	Austria	Canada	Denmark	Finland
time							
1993Q1	1993q1	0.062	0.040489	-0.013084	0.010064	-0.012292	-0.028357
1993Q2	1993q2	0.059	0.037857	-0.007581	0.021264	-0.003093	-0.023397
1993Q3	1993q3	0.058	0.022509	0.000543	0.018919	-0.007764	-0.006018
1993Q4	1993q4	0.062	0.028747	0.001181	0.025317	-0.004049	-0.004774
1994Q1	1994q1	0.079	0.033990	0.025511	0.043567	0.031094	0.012886
1994Q2	1994q2	0.068	0.037919	0.019941	0.050225	0.064280	0.035090
1994Q3	1994q3	0.046	0.052289	0.017088	0.065122	0.045955	0.035247
1994Q4	1994q4	0.052	0.031071	0.023035	0.067331	0.055166	0.057251
1995Q1	1995q1	0.037	0.008696	0.025293	0.050921	0.048057	0.068382
1995Q2	1995q2	0.029	0.006774	0.021850	0.031525	0.011954	0.079265
1995Q3	1995q3	0.012	0.003028	0.018319	0.018180	0.020810	0.085122
1995Q4	1995q4	0.015	0.010982	0.013457	0.015166	0.008304	0.085290
1996Q1	1996q1	0.025	0.038182	0.015387	0.007821	0.010102	0.049422
1996Q2	1996q2	0.036	0.034520	0.017336	0.011510	0.030859	0.029334
1996Q3	1996q3	0.047	0.036673	0.013595	0.021660	0.040113	0.017545
1996Q4	1996q4	0.059	0.038987	0.004195	0.024704	0.025694	0.020857
1997Q1	1997q1	0.058	0.025748	-0.001535	0.035775	0.029462	0.054267
1997Q2	1997q2	0.072	0.051863	-0.002021	0.038688	0.039856	0.055036

18 rows × 26 columns

In [26]:

```
y = pre1997['HongKong']
indep_var = [
    'Australia', 'France', 'Taiwan', 'Singapore', 'Denmark', 'Japan',
    'Indonesia', 'Philippines'
]
X = pre1997[indep_var]
X
```

Out[26]:

time	Australia	France	Taiwan	Singapore	Denmark	Japan	Indonesia
1993Q1	0.040489	-0.015177	0.064902	0.087145	-0.012292	0.012683	0.064024
1993Q2	0.037857	-0.014549	0.065123	0.118075	-0.003093	-0.005571	0.066068
1993Q3	0.022509	-0.016704	0.067379	0.111130	-0.007764	-0.017558	0.057959
1993Q4	0.028747	-0.007476	0.069164	0.125324	-0.004049	-0.010101	0.062365
1994Q1	0.033990	0.003748	0.069451	0.130709	0.031094	-0.022503	0.049743
1994Q2	0.037919	0.016165	0.070135	0.100987	0.064280	-0.005157	0.071988
1994Q3	0.052289	0.023915	0.069298	0.115984	0.045955	0.014087	0.069357
1994Q4	0.031071	0.029711	0.076049	0.092047	0.055166	0.005427	0.070347
1995Q1	0.008696	0.027446	0.071104	0.066013	0.048057	0.003919	0.103778
1995Q2	0.006774	0.021708	0.069214	0.079020	0.011954	0.015349	0.083252
1995Q3	0.003028	0.014152	0.063417	0.087519	0.020810	0.015939	0.070873
1995Q4	0.010982	0.007562	0.048752	0.079930	0.008304	0.026233	0.076905
1996Q1	0.038182	0.008412	0.054664	0.111706	0.010102	0.026878	0.038864
1996Q2	0.034520	0.003447	0.059415	0.087513	0.030859	0.022471	0.066380
1996Q3	0.036673	0.008331	0.059393	0.043669	0.040113	0.012733	0.088341
1996Q4	0.038987	0.006623	0.070151	0.060471	0.025694	0.021495	0.129308
1997Q1	0.025748	0.004803	0.063613	0.055836	0.029462	0.025330	0.130339
1997Q2	0.051863	0.018441	0.060897	0.089873	0.039856	0.006164	0.100466

下面开始回归：

In [27]:
```
from sklearn import linear_model

## 创建模型对象
reg = linear_model.LinearRegression()
## 进行回归
reg.fit(X, y)  ## 训练模型
## 查看系数
print("系数=", reg.coef_)
print('截距项=', reg.intercept_)
```

```
系数= [ 0.73937958 -0.28910318  0.29587194 -0.02260917  0.11208418 -0.81544603
  0.30000394 -0.09297764]
截距项= -0.003890434533065368
```

如果需要做预测，可以使用：

In [28]:
```
predicted = reg.predict(data[indep_var])
data['predict_HongKong'] = predicted
data
```

Out[28]:

	time	HongKong	Australia	Austria	Canada	Denmark	Finland
time							
1993Q1	1993q1	0.062	0.040489	-0.013084	0.010064	-0.012292	-0.028357
1993Q2	1993q2	0.059	0.037857	-0.007581	0.021264	-0.003093	-0.023397
1993Q3	1993q3	0.058	0.022509	0.000543	0.018919	-0.007764	-0.006018
1993Q4	1993q4	0.062	0.028747	0.001181	0.025317	-0.004049	-0.004774
1994Q1	1994q1	0.079	0.033990	0.025511	0.043567	0.031094	0.012886
...	...	...	...	...	...	...	...
2007Q1	2007q1	0.055	0.058013	0.036198	0.030712	0.033134	0.047340
2007Q2	2007q2	0.062	0.059519	0.032570	0.039827	-0.007169	0.046808
2007Q3	2007q3	0.068	0.056649	0.031558	0.034742	0.013517	0.045647
2007Q4	2007q4	0.069	0.045825	0.019095	0.038128	0.023794	0.044177
2008Q1	2008q1	0.073	0.027523	0.017431	0.029217	-0.005200	0.023732

61 rows × 27 columns

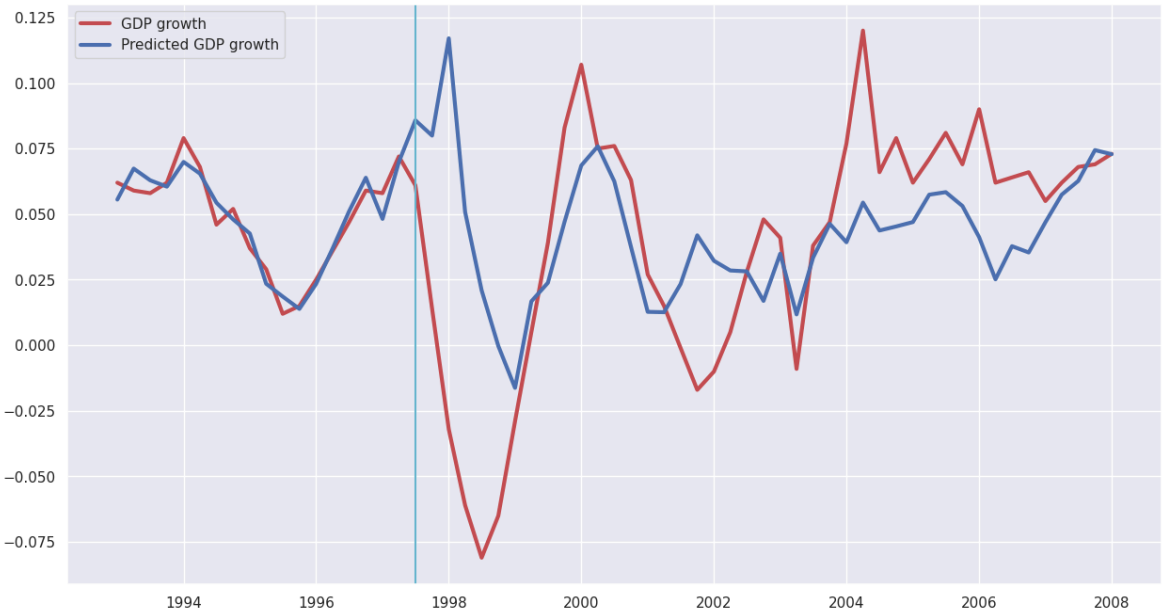
注意上面我们为了预测，需要首先将原来的数据框变成与训练用的数据集相同的维度（和变量顺序），因为Scikit-Learn实际上把数据框当矩阵看待，所以不会自动匹配列标题。这点尤其需要注意。

继而可以画图：

In [29]:
```
import matplotlib.pyplot as plt
import seaborn as sb

sb.set()
%matplotlib inline
plt.rcParams['figure.figsize'] = (15.0, 8.0)
data['year'] = pd.to_datetime(data['time'])
plt.plot(data['year'], data['HongKong'], c='r', lw=3, label='GDP growth')
plt.plot(data['year'],
         data['predict_HongKong'],
         c='b',
         lw=3,
         label='Predicted GDP growth')
plt.legend(loc='upper left', frameon=True)
plt.axvline(pd.to_datetime('1997Q3'), c='c')
plt.show()
```

```
/tmp/ipykernel_54132/496109544.py:7: UserWarning: Could not infer format, so each
element will be parsed individually, falling back to `dateutil`. To ensure parsin
g is consistent and as-expected, please specify a format.
  data['year'] = pd.to_datetime(data['time'])
```



以上我们选取了一些国家进行回归，问题是我们并不能保证如此处理是预测效果最好的（没有过拟合、欠拟合）。接下来我们展示在线性回归的基础上进行L1正则化，即Lasso回归的做法：

In [30]:
```
data = pd.read_csv('csv/hcw.csv')
data['quarter'] = pd.to_datetime(data['time'])
data = data.set_index('quarter')
data.index = data.index.to_period('Q')
pre1997 = data[data.index <= '1997Q2']
## 准备X,y
y = pre1997['HongKong']
X = pre1997.drop(['HongKong', 'China', 'time'], axis=1)
std_X = (X - X.mean()
         ) / X.std()  # 标准化X，如果下面的linear.model.Lasso不设置normalize=Tru
std_X.describe()
```

```
/tmp/ipykernel_54132/2270252052.py:2: UserWarning: Could not infer format, so eac
h element will be parsed individually, falling back to `dateutil`. To ensure pars
ing is consistent and as-expected, please specify a format.
  data['quarter'] = pd.to_datetime(data['time'])
```

Out[30]:

	Australia	Austria	Canada	Denmark	Finlanc
count	1.800000e+01	1.800000e+01	1.800000e+01	1.800000e+01	1.800000e+01
mean	-1.480297e-16	8.635068e-17	5.859510e-17	4.317534e-17	-1.788693e-16
std	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00
min	-1.855402e+00	-1.991950e+00	-1.269459e+00	-1.606998e+00	-1.777956e+00
25%	-4.605165e-01	-8.371103e-01	-6.916538e-01	-6.786943e-01	-5.799627e-01
50%	2.912820e-01	3.180241e-01	-3.274983e-01	1.516707e-01	1.661822e-02
75%	5.567160e-01	7.405953e-01	6.225495e-01	7.017792e-01	6.247855e-01
max	1.531036e+00	1.241132e+00	1.991602e+00	1.770627e+00	1.432527e+00

8 rows × 23 columns

In [31]:
```
## 回归
lasso = linear_model.Lasso(alpha=0.0005)
lasso.fit(std_X, y)  ## 训练模型
lasso.coef_
```

Out[31]:
```
array([ 0.00384836, -0.        ,  0.00106811,  0.        ,  0.        ,
        0.        ,  0.        , -0.        , -0.00839452, -0.00936549,
        0.00102252,  0.        , -0.        , -0.        ,  0.        ,
        0.        ,  0.00012505,  0.        , -0.00051634,  0.        ,
       -0.00216037, -0.        ,  0.00594295])
```

选项中的alpha即我们上面提到的λ，而normalize选项设定为需要进行标准化。可以看到上面很多系数的取值都为0，达到了变量选择的目的。

然而上面的alpha是我们设定的，一个更好的办法是通过交叉验证选取合适的alpha。可以使用LassoCV来做：

In [32]:
```
lasso_cv = linear_model.LassoCV(eps=1e-5, cv=10,
                                 max_iter=1000)  #10折交叉验证，最大迭代次数1000次
lasso_cv.fit(std_X, y)
print("选择的alpha=", lasso_cv.alpha_)
lasso_cv.coef_
```

```
/opt/Anaconda/lib/python3.11/site-packages/sklearn/linear_model/_coordinate_descent.py:697: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing regularisation. Duality gap: 8.715e-07, tolerance: 6.232e-07
  model = cd_fast.enet_coordinate_descent(
/opt/Anaconda/lib/python3.11/site-packages/sklearn/linear_model/_coordinate_descent.py:697: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing regularisation. Duality gap: 7.393e-07, tolerance: 6.232e-07
  model = cd_fast.enet_coordinate_descent(
选择的alpha= 0.0006731855207581835
```

Out[32]:
```
array([ 0.0038306 , -0.        ,  0.00030148,  0.        , -0.        ,
        0.        ,  0.        , -0.        , -0.00826843, -0.00891092,
        0.00145376, -0.        , -0.        , -0.        ,  0.        ,
       -0.        ,  0.        ,  0.        , -0.        ,  0.        ,
       -0.00204314, -0.        ,  0.00594418])
```

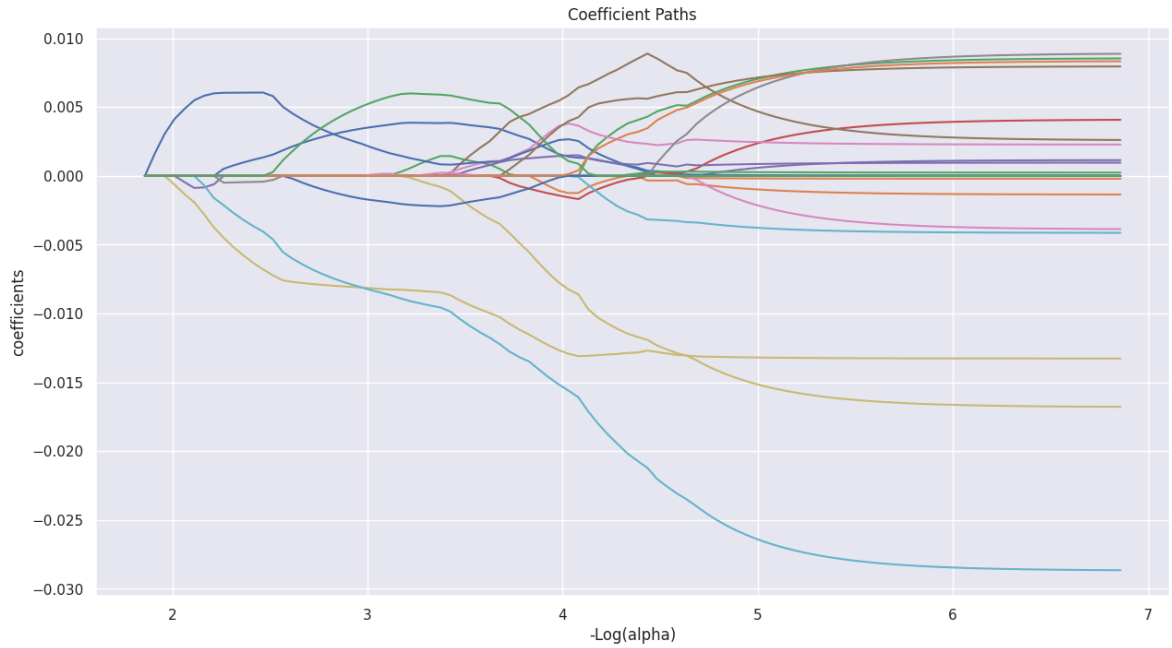
可以使用路径图来查看随着alpha的变动，系数的变动情况：

In [33]:
```
from sklearn.linear_model import lasso_path

alphas_lasso, coefs_lasso, _ = lasso_path(std_X, y, eps=1e-5)
neg_log_alphas_lasso = -np.log10(alphas_lasso)
for coef_l in coefs_lasso:
    l1 = plt.plot(neg_log_alphas_lasso, coef_l)

plt.xlabel('-Log(alpha)')
plt.ylabel('coefficients')
plt.title('Coefficient Paths')
```

Out[33]:
```
Text(0.5, 1.0, 'Coefficient Paths')
```

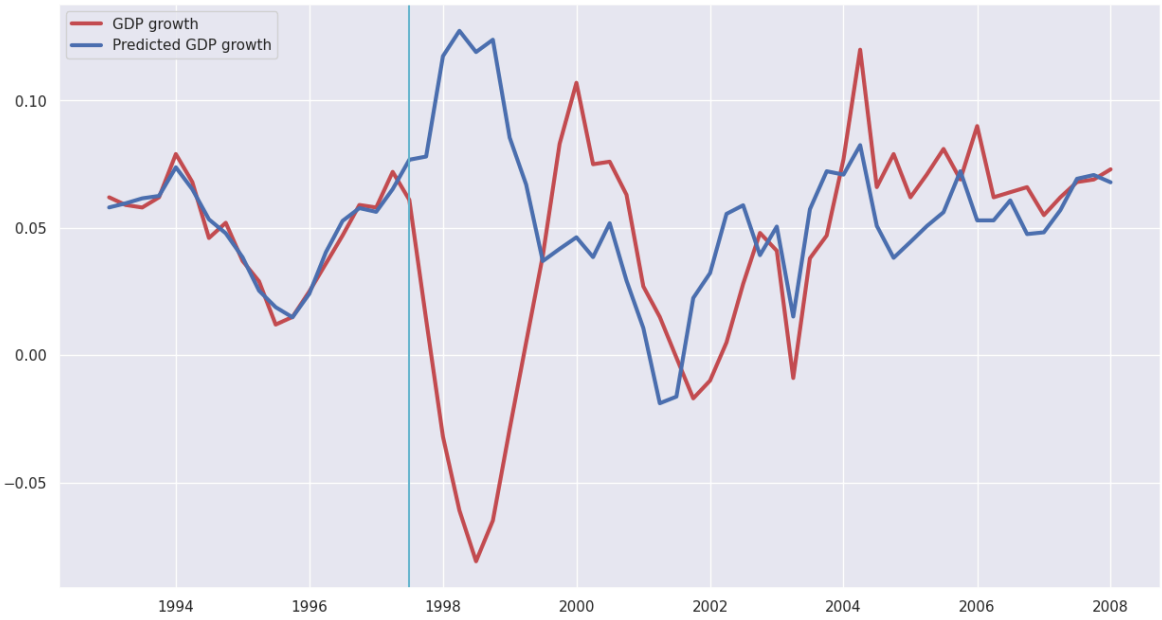


进一步做预测：

In [34]:
```
pred_set = data.drop(['HongKong', 'China', 'time'], axis=1)
pred_set = (pred_set -
            X.mean()) / X.std()  ## 因为Lasso命令之前的步骤中对X进行了标准化，所以
predicted = lasso_cv.predict(pred_set)
data['predict_HongKong'] = predicted
import matplotlib.pyplot as plt
import seaborn as sb

sb.set()
%matplotlib inline
plt.rcParams['figure.figsize'] = (15.0, 8.0)
data['year'] = pd.to_datetime(data['time'])
plt.plot(data['year'], data['HongKong'], c='r', lw=3, label='GDP growth')
plt.plot(data['year'],
         data['predict_HongKong'],
         c='b',
         lw=3,
         label='Predicted GDP growth')
plt.legend(loc='upper left', frameon=True)
plt.axvline(pd.to_datetime('1997Q3'), c='c')
plt.show()
```

```
/tmp/ipykernel_54132/347982631.py:12: UserWarning: Could not infer format, so eac
h element will be parsed individually, falling back to `dateutil`. To ensure pars
ing is consistent and as-expected, please specify a format.
  data['year'] = pd.to_datetime(data['time'])
```



此外，还有一个比较常见的做法是单纯使用Lasso作为变量选择工具，使用Lasso选取变量后，继续使用OLS进行估计和预测，这杯成为『Post-estimation』，我们可以如此进行此操作：

In [35]:
```
selected_vars = X.columns[lasso_cv.coef_ != 0]
print("选取的国家地区：\n", selected_vars)
data_ols = data.loc[:, selected_vars]
data_ols['HongKong'] = data["HongKong"]
data_ols.head()
```

```
选取的国家地区：
 Index(['Australia', 'Canada', 'Japan', 'Korea', 'Mexico', 'Malaysia',
       'Taiwan'],
      dtype='object')
```

Out[35]:

quarter	Australia	Canada	Japan	Korea	Mexico	Malaysia	Taiwan	H
1993Q1	0.040489	0.010064	0.012683	0.058586	0.043746	0.085938	0.064902	
1993Q2	0.037857	0.021264	-0.005571	0.069521	0.012290	0.131189	0.065123	
1993Q3	0.022509	0.018919	-0.017558	0.081646	0.004462	0.109666	0.067379	
1993Q4	0.028747	0.025317	-0.010101	0.085533	0.015493	0.075801	0.069164	
1994Q1	0.033990	0.043567	-0.022503	0.085922	0.034448	0.049147	0.069451	

In [36]:
```
pre1997 = data_ols[data.index <= '1997Q2']
pre1997
```

Out[36]:

quarter	Australia	Canada	Japan	Korea	Mexico	Malaysia	Taiwan	H
1993Q1	0.040489	0.010064	0.012683	0.058586	0.043746	0.085938	0.064902	
1993Q2	0.037857	0.021264	-0.005571	0.069521	0.012290	0.131189	0.065123	
1993Q3	0.022509	0.018919	-0.017558	0.081646	0.004462	0.109666	0.067379	
1993Q4	0.028747	0.025317	-0.010101	0.085533	0.015493	0.075801	0.069164	
1994Q1	0.033990	0.043567	-0.022503	0.085922	0.034448	0.049147	0.069451	
1994Q2	0.037919	0.050225	-0.005157	0.088415	0.063911	0.061173	0.070135	
1994Q3	0.052289	0.065122	0.014087	0.092796	0.064322	0.101110	0.069298	
1994Q4	0.031071	0.067331	0.005427	0.117461	0.066942	0.134978	0.076049	
1995Q1	0.008696	0.050921	0.003919	0.115388	0.043418	0.130363	0.071104	
1995Q2	0.006774	0.031525	0.015349	0.121889	-0.059943	0.119140	0.069214	
1995Q3	0.003028	0.018180	0.015939	0.132274	-0.079015	0.065085	0.063417	
1995Q4	0.010982	0.015166	0.026233	0.092351	-0.067619	0.074479	0.048752	
1996Q1	0.038182	0.007821	0.026878	0.094772	-0.055922	0.103842	0.054664	
1996Q2	0.034520	0.011510	0.022471	0.072358	0.022377	0.092732	0.059415	
1996Q3	0.036673	0.021660	0.012733	0.055408	0.054119	0.096904	0.059393	
1996Q4	0.038987	0.024704	0.021495	0.057294	0.055560	0.095856	0.070151	
1997Q1	0.025748	0.035775	0.025330	0.031708	0.028028	0.068248	0.063613	
1997Q2	0.051863	0.038688	0.006164	0.053364	0.056595	0.065035	0.060897	

In [37]:

```
y = pre1997['HongKong']
X = pre1997.drop("HongKong", axis=1)
reg = linear_model.LinearRegression().fit(X, y)  ## 训练模型
predicted = reg.predict(data_ols.drop("HongKong", axis=1))
data_ols['predict_HongKong'] = predicted
data_ols
```

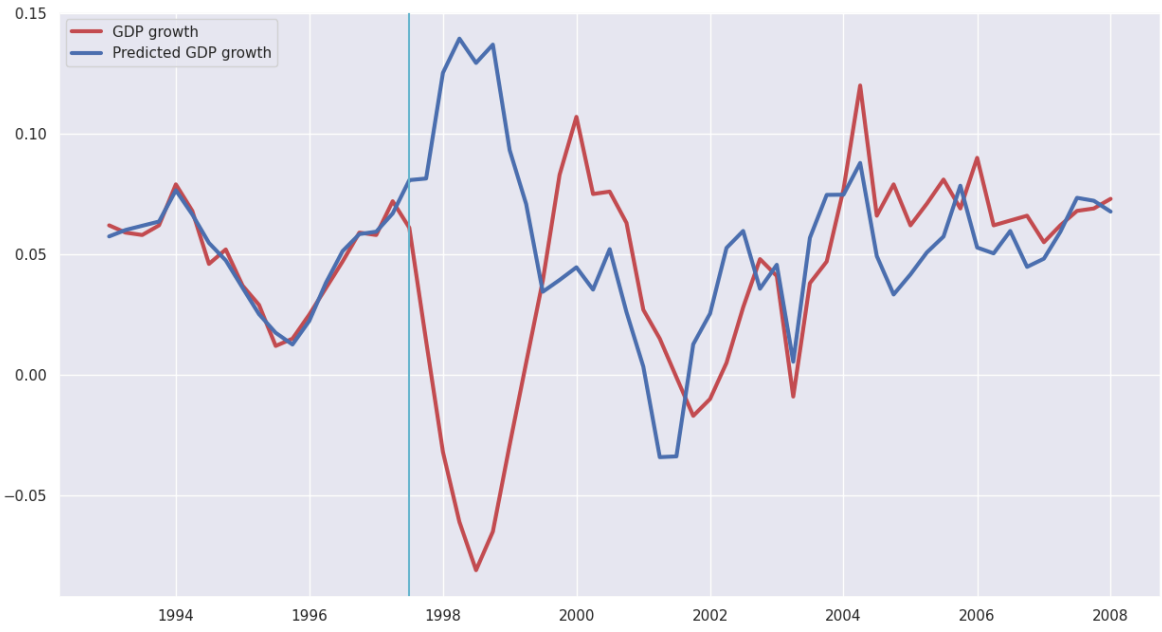
Out[37]:

quarter	Australia	Canada	Japan	Korea	Mexico	Malaysia	Taiwan	H
1993Q1	0.040489	0.010064	0.012683	0.058586	0.043746	0.085938	0.064902	
1993Q2	0.037857	0.021264	-0.005571	0.069521	0.012290	0.131189	0.065123	
1993Q3	0.022509	0.018919	-0.017558	0.081646	0.004462	0.109666	0.067379	
1993Q4	0.028747	0.025317	-0.010101	0.085533	0.015493	0.075801	0.069164	
1994Q1	0.033990	0.043567	-0.022503	0.085922	0.034448	0.049147	0.069451	
...	...	...	...	...	...	...	...	...
2007Q1	0.058013	0.030712	0.023959	0.030068	0.038222	0.039884	0.041019	
2007Q2	0.059519	0.039827	0.014193	0.037898	0.022226	0.080276	0.051073	
2007Q3	0.056649	0.034742	0.012907	0.039380	0.031162	0.093361	0.066369	
2007Q4	0.045825	0.038128	-0.004768	0.035359	0.062714	0.151739	0.062929	
2008Q1	0.027523	0.029217	-0.013647	0.028462	0.056493	0.167034	0.058841	

61 rows × 9 columns

In [38]:
```
%matplotlib inline
plt.rcParams['figure.figsize'] = (15.0, 8.0)
data_ols['year'] = pd.to_datetime(data['time'])
plt.plot(data_ols['year'],
         data_ols['HongKong'],
         c='r',
         lw=3,
         label='GDP growth')
plt.plot(data_ols['year'],
         data_ols['predict_HongKong'],
         c='b',
         lw=3,
         label='Predicted GDP growth')
plt.legend(loc='upper left', frameon=True)
plt.axvline(pd.to_datetime('1997Q3'), c='c')
plt.show()
```

```
/tmp/ipykernel_54132/2598912699.py:3: UserWarning: Could not infer format, so eac
h element will be parsed individually, falling back to `dateutil`. To ensure pars
ing is consistent and as-expected, please specify a format.
  data_ols['year'] = pd.to_datetime(data['time'])
```


GDP growth
Predicted GDP growth
0.15
0.10
0.05
0.00
−0.05
1994
1996
1998
2000
2002
2004
2006
2008