# 第六章 线性模型选择和正则化

我们在线性回归中用到的模型：

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_p X_p + \epsilon$$

包含$p$个特征值和一个随机误差项$\epsilon$。当$n >> p$时，最小二乘法对这个模型系数的估计拥有较低的偏差和方差，表现良好

然而当$n$仅仅大于$p$，甚至$n < p$时，最小二乘法的方差会变得非常大，导致模型效果变差。此时，我们必须考虑用其他的方法，以提高模型的**预测精度(prediction accuracy)** 和 **可解释性(interpretability)**

> 其他的一些线性模型，可以实现自动的**特征选择(feature selection)** 或 **变量选择(variable selection)**

本章主要介绍三类重要的模型：

- **子集选择(Subset Selection)** 找到系数的较优子集作为模型的系数估计
- **收缩(Shrinkage)** 从拟合所有系数收缩到部分系数进行模型复杂度的降低，也可以用于变量选择
- **降维(Dimension Reduction)** 将$p$个特征值投影到低维空间，使得$M < p$个投影特征用于估计系数

## 6.1 子集选择

### 6.1.1 最优子集选择

在**最优子集选择(best subset selection)** 策略中，对于$p$个特征值，我们拟合$\sum_{i=1}^{p} C_p^i = 2^p$个模型，并找到最优模型作为我们的参数选择

在子集选择过程中，我们考虑$n = 1, 2, \cdots, p$个特征的模型，并在每一类的模型中，从$p$个参数中选取$k$个来拟合此类模型；对于每一类模型，我们得到该类最优模型；选择完毕后，我们得到$p$个模型，最后从这$p$+1个模型(含null模型)中得到最优模型

> 每类模型的判断标准为$RSS$或$R^2$，$p+1$个模型的判断标准为交叉检验、$C_p, BIC, \text{adjusted } R^2$等

最优子集选择容易导致选择次数过多的问题，从而无法在限定时间内得到结果；而且，过大的子集选择范围，可能导致模型较大的方差

### 6.1.2 逐项选择

**前向选择(Forward Stepwise Selection)** 是一种替代最优子集选择的策略。从null模型出发，每次添加一个最优特征值，直至选完所有特征值，再对$p+1$个模型选择得到最优模型。不难

得出前向选择的选择次数为$1+\frac{p(p+1)}{2}$

虽然实践中前向选择的表现不差，但由于缩小选择子集的缘故，前向选择不能保证获得最优解，因为前向选择可能导致某类模型变量的遗漏

在高维数据中，前项选择也是可以利用的，但只能构筑$M_{n-1}$即特征值为$n-1$个的模型，原因是最小二乘法的唯一解性

**后向选择(Backward stepwise selection)** 也是一种替代最优子集选择的方法。与前向选择相反，该算法从包含$p$个特征值的模型**全模型(full model)** 出发，每次减少一个特征值，并在此类中留下最优模型，直到null模型为止，并选择$p+1$个模型作为最优模型

后向选择不能在$p>n$是使用，只能在$n\geq p$时使用，因为需要拟合全模型

**混合选择(Hybrid Selection)** 在前向选择的同时，可能删除某些变量回退到后向选择，直至获取最优模型。这个算法保留了计算优势，同时紧密地模仿最优子集选择

## 6.1.3 最优模型选择

通过上述方法得到的模型都会包含全模型。全模型的偏差最小，但方差不一定最小，因此不适合用$R^2$来比较不同特征值数量的模型

比较不同特征值数量的模型，常用的方法有：

1. 直接估计测试误差，利用交叉检验等方法
2. 对训练误差进行调整，从而间接估计训练误差

### 6.1.3.1 调整训练误差

对于一个最小二乘法拟合的包括$d$个特征值的模型，$\mathbf{C}_p$估计训练误差：

$$C_p=\frac{1}{n}(RSS+2d\hat{\sigma}^2)$$

其中的$\hat{\sigma}^2$是对回归方程中随机误差$\epsilon$的估计。$C_p$通过添加$wd\hat{\sigma}^2$来放大训练误差，注意到随着特征值的增多，$C_p$也跟着增大

> $C_p$有两个公式，这两者计算的$C_p$正相关

可以证明$C_p$是MSE的一个无偏估计，因此用最小的$C_p$可以判断一个最优的拟合模型

**赤池信息准则(AIC)** 是针对极大似然法估计的一大类模型定义的。在假设$\epsilon$满足高斯分布后，极大似然估计与最小二乘估计结构相同，此时：

$$AIC=\frac{1}{n}(RSS+2d\hat{\sigma}^2)$$

这里省略了一些常数，可知AIC与$C_p$成正比

> AIC有两个公式

**贝叶斯信息准则(BIC)** 是根据贝叶斯公式推导的信息准则，形式接近于$C_p$和AIC。对于$p$个特征值的最小二乘模型：

$$BIC = \frac{1}{n}(RSS + \log(n)d\hat{\sigma}^2)$$

根据公式可知BIC对复杂模型惩罚力度大，因此倾向于选择参数较少的模型(相对于AIC和$C_p$)

**调整后$R^2$(adjusted $R^2$)** 是一种留下的选择模型的准则。根据$R^2$的公式，当模型变量越多时，$R^2$越大。对于$d$个特征值拟合的最小二乘模型：

$$Adjusted\ R^2 = 1 - \frac{RSS/(n-d-1)}{TSS(n-1)}$$

根据公式，当模型中变量增多时，$R^2_{adj}$可能增大也可能减小，取决与变量增多是否能增大模型的拟合效果

> 这一推断取决于$RSS/(n-d-1)$的变化

调整后$R^2$背后的直觉是当所有正确的变量都被包含在模型中，额外添加噪声将导致$R^2_{adj}$下降(少量RSS下降伴随着$n-d-1$增加)。因此$R^2_{adj}$最大的模型理论上将只包含正确变量

需要注意的是：

1. 理论上$R^2_{adj}$不如其他三个准则那么可解释
2. 其他三个准则有复杂的公式推导，基于渐进估计$n \to \infty$
3. AIC和BIC可以应用于更加复杂的模型，有不同的公式形式

### 6.1.3.2 交叉检验方法

交叉检验方法可以替代上面的间接估计对训练误差作出直接估计，且对模型作出的假设更少，能适应更广泛的模型

值得注意的是，当使用交叉验证时，选择算法中的模型$M_k$的序列对于每一个训练折叠都是单独确定的，并且对于每一个模型大小k，验证误差在所有折叠中都是平均的。这意味着，例如在最佳子集回归中，$M_k$，即大小为k的最佳子集，可以在不同的折叠中有所不同。一旦选择了最佳尺寸k，我们在全数据集上找到了该尺寸的最佳模型。

> $k$表示选取的特征数量，当$k < m$，$m$表示特征总数时，一定能选出多种不同的特征集合作为全特征的子集

对于不同的fold，我们得到每个模型的估计训练误差一定是不同的，但可能相差不多。在这种情况下，我们计算交叉检验的最优模型测试误差，并在它的一个标准差内，选择最简单的模型，这个方法叫做 $1SE$**规则(one standard error rule)**，鼓励选择变量少的模型

> 简单理解为选择在最优估计测试误差一个标准差内的最简模型

# 6.2 收缩方法

在6.1节讨论的方法通过选择最小二乘回归你和的子集来选择最优的特征值，下面介绍的**收缩方法(shrinkage methods)** 将通过拟合所有的特征值，再通过收缩去除某些特征值来选择模型

## 6.2.1 岭回归

最小二乘回归试图估计一组$\beta_0, \beta_1, \cdots, \beta_p$，使得：

$$RSS = \sum_{i=1}^{n} \left( y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2$$

取得最小值。**岭回归(Ridge Regression)** 的策略与此类似，它最小化：

$$RSS = \sum_{i=1}^{n} \left( y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 = RSS + \lambda \sum_{j=1}^{p} \beta_j^2$$

来达到选择变量目的，其中$\lambda \geq 0$是一个**调谐变量(tuning parameter)**

注意到岭回归通过引入$\lambda \sum_{j=1}^{p} \beta_j^2$来寻找回归系数。事实上$\lambda \sum_{j=1}^{p} \beta_j^2$是一个**收缩惩罚(shrinkage penalty)**，他在$\beta_0, \beta_1, \cdots, \beta_p$趋近于0时很小，因此岭回归倾向于将回归系数估小

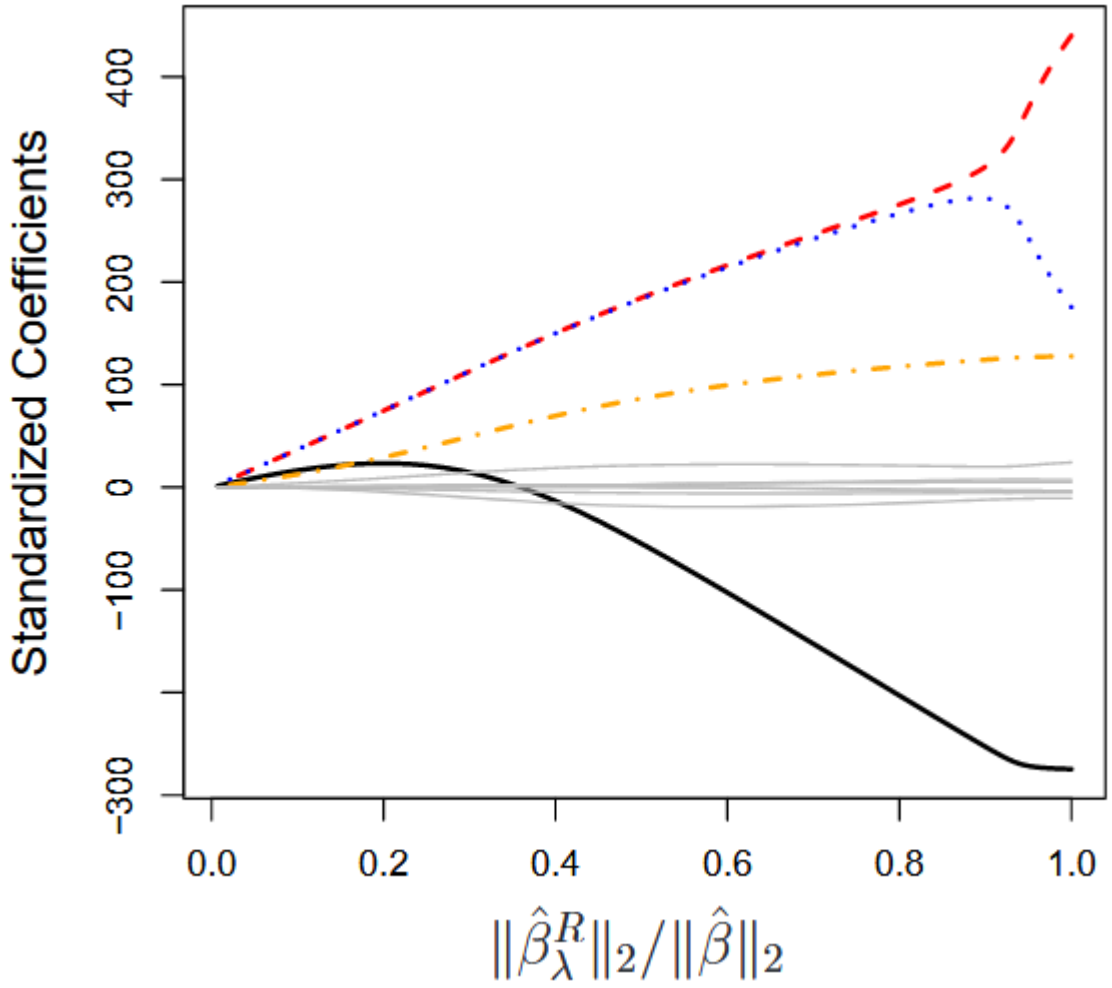
当$\lambda = 0$时，岭回归退化到最小二乘回归，此时惩罚项不起作用；当$\lambda \to \infty$时，惩罚项将回归系数控制到趋近于0。不同于最小二乘回归，岭回归在不同的$\lambda$下都会产生不同的回归系数估计，因此选择一个适当的$\lambda$非常重要

我们不希望岭回归对截距项进行惩罚，因为它是$\mathbf{X} = 0$时均值的估计。当$X$已经中心化为0时，估计的截距为$\hat{\beta}_0 = \overline{y} = \sum_{i=1}^{n} y_i / n$

> $\lambda \to \infty$时对应的实际上是零模型

---

$l_2$**范数(**$l_2$ **norm)** 是用于衡量一个向量与原点距离的值，计算为$||\beta||_2 = \sqrt{\sum_j^p = \beta_j^2}$。通过计算$||\hat{\beta}_\lambda^R||_2 / ||\hat{\beta}||_2$值的变化，我们可以很直观的观察$\lambda$增大对系数估计的影响




与OLS模型不同，岭回归不是**等尺度变化(scale equivariant)** 的。若某个特征值的单位发生变化导致特征值本身数值发生缩放，则岭回归估计的系数将会发生复杂的变化，原因在于惩罚项中$\hat{\beta}$是二次的形式，因此有必要对特征值的观测值进行标准化：

$$x_{ij} = \frac{x_{ij}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_{ij} - \overline{x}_j)^2}}$$

以保证数据在同一尺度

岭回归的计算量相比于最小二乘回归也非常有优势。特别地，对于一个确定的λ，岭回归只得到一个模型(相比最小二乘回归每$p$个特征值需要你和多个模型)

## 6.2.2 拉索回归

岭回归有一个明显的缺点是模型总是包括所有的特征值，这虽然不会显著降低预测精度，但带来了解释上的困难

为了克服这个缺点，我们引入**拉索回归(lasso regression)**，拉索回归试图通过：

$$\sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij}\right)^2 + \lambda\sum_{j=1}^{p}|\beta_j| = RSS + \lambda\sum_{j=1}^{p}|\beta_j|$$

获取回归系数的预测值$\hat{\beta}_{\lambda}^{L}$。与岭回归不同之处在于拉索回归引入的乘法项采用的是$l_1$惩罚,，计算方式为$||\beta||_1 = \sum|\beta_j|$

拉索回归的惩罚项在$n \to \infty$时会将某些特征值控制为0，实际上执行了参数选择的效果

---

拉索回归和岭回归可以用下面的两个式子联系起来：

$$min\left\{\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2\right\}\text{subject to}\sum_{j=1}^{p}|\beta_j| \le s min\left\{\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2\right\}\text{subject}$$

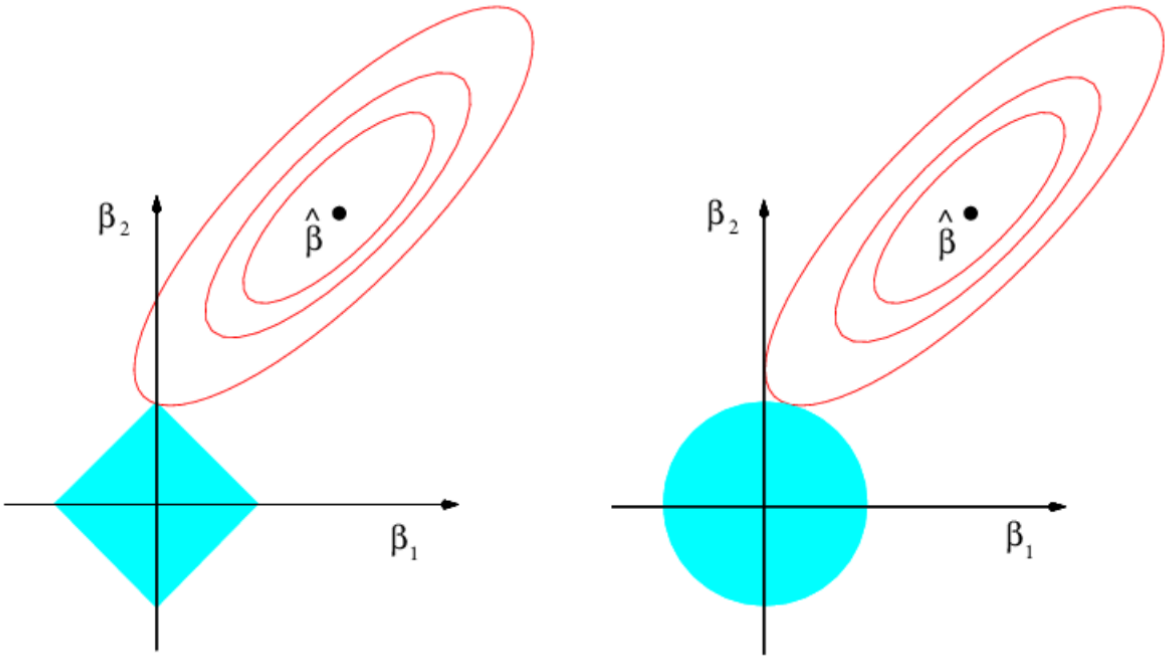
换句话说，给定一个值$s$用以约束最小二乘回归，使得系数的某种组合必须满足约束值$s$

上面两个式子将拉索回归和岭回归联系起来了。同样地，我们考虑下面的式子：

$$min\left\{\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j x_{ij}\right)^2\right\}\text{subject to}\sum_{j=1}^{p}I(\beta_j \ne 0) \le s$$

其中$I$是一个指示变量，表示系数$\beta_j \ne 0$(此时值为1)。这实际上是子集选择的数学表示

当$p$很大时，子集选择很难计算。通过上面的三个式子，我们可以发现岭回归和拉索回归就是子集选择回归的一种代替




对比上面的两个式子，可以发现拉索回归的系数约束区域是一个菱形，而岭回归的系数约束区域是一个圆形，系数的估计区域落在约束区域和估计曲线的交点上。这可以从几何上解释为什么拉索回归可以将系数压缩到0

> 上述的菱形和圆形在$p > 2$时扩展为对应的几何图形

不同的情况下，我们选择不同的模型。一般在预测变量很多且少量的特征拥有大系数时，考虑用拉索回归；在大部分系数都和响应变量有关，且系数相差不大时，我们考虑岭回归。实际应用中我们无法预知这些情况，所以可以考虑交叉检验来实际计算检验误差

> 值得注意的是，拉索回归的可解释性比岭回归好

拉索回归通过**软阈值(soft thresholding)** 算法来执行变量选择的工作。简单的举例，若$|\beta > \lambda$则$\beta$减小到$\beta - \lambda$；若$|\beta| \leq \lambda$则$\beta$被置为0

利用**贝叶斯观点(Bayes Lens)** 研究拉索回归和岭回归得到的结论是：

- 岭回归相当于假设回归系数服从正态分布的先验，最大化后验分布得到的解趋向于平滑且不会将系数压缩为零
- 拉索回归则假设回归系数服从**拉普拉斯分布(Laplace distribution)** 的先验，最大化后验分布得到的解倾向于稀疏化，即一些系数会被压缩为零，从而自动进行特征选择

> 理论分析难度很大，忽略

### 6.2.3 选择调谐参数

使用交叉检验我们可以确定一个合适的$\lambda$参数值以进行岭回归和拉索回归

在回归领域，相关和不相关的特征被分别称为**信号(signal)** 和噪声变量

## 6.3 降维方法

前面使用的两类方法使用的特征向量都是由原始特征得来的。下面介绍的方法将预测变量进行转换，对转换后的预测变量进行最小二乘回归

令$Z_1, Z_2, \cdots, Z_m$表示$M < p$个原始预测变量的**线性组合(linear combination)**：

$$Z_m = \sum_{j=1}^{p} \phi_{jm} X_j$$

其中$\phi_{1m}, \phi_{2m}, \cdots$是常数。我们用这些预测变量进行最小二乘回归：

$$y_i = \theta_0 + \sum_{m=1}^{M} \theta_m z_{im} + \epsilon_i, \quad i = 1, \ldots, n$$

当我们的参数$\varphi_{pm}$选择得当时，拟合上面降维后的线性回归要比直接拟合线性回归得到的结果更好。这时候，问题的维度从$p+1$下降到$M+1$

注意到：

$$\sum_{m=1}^{M} \theta_m z_{im} = \sum_{m=1}^{M} \theta_m \sum_{j=1}^{p} \phi_{jm} x_{ij} = \sum_{j=1}^{p} \sum_{m=1}^{M} \theta_m \phi_{jm} x_{ij} = \sum_{j=1}^{p} \beta_j x_{ij}$$

其中：

$$\beta_j = \sum_{m=1}^{M} \theta_m \phi_{jm}$$

因此降维的线性回归可以看做原线性回归的一个特例，是为了限制原系数$\beta_j$必须满足某个特殊形式(降维后的线性回归)

在$p$相对于$n$很大的情况下，选择$M << p$的一个值可以显著降低预测的方差。当$M = p$且所有$Z_m$线性无关时，执行的实际上是原线性回归

降维方法都分为下面两步：
4. 对预测因子进行转换
5. 对转换后的预测因子进行拟合

然而，如何选组转换系数有多种方法。下面介绍两种方法：主成分分析和偏最小二乘

## 6.3.1 主成分分析

**主成分分析(Principal components analysis)** 是一种常用的降维工具

主成分分析的操作方式是将原数据**投影（project)** 到一条直线上，使原数据在这条新直线上的方差最大

> 方差最大的目的是保留最多的原始信息

我们可以考虑构造多个这样的直线，用以投影原始数据的信息。其中，用于投影数据的直线叫做**主成分轴**，捕获最多变量的主成分是**第一主成分(first principal component)**，依次类推可以得到第二主成分等，原数据投影后得到的数据叫做**主成分得分(principal component scores)**。

> 另一种描述主成分轴的方法是主成分轴刻画了到所有原始数据距离最近的一条直线

主成分之间应当是线性无关的。可以证明，若两个主成分之间线性无关，则主成分轴**垂直(perpendicular)**

对于第$i$主成分，捕获的信息量依次降低。通过选择前$k$个主成分忽视后面的主成分，可以达到降维的效果，同时去除数据的共线性

---

**主成分回归(principal components regression, PCR)** 通过先对原始数据矩阵进行主成分分析PCA，再用主成分得分进行线性回归

> 这里的假设是，$X_1, X_2, \cdots, X_p$变化最大的方向是跟$Y$有关的方向

当主成分回归的假设成立时，回归得到的结果要比用全部特征值进行拟合更好，原因是降低了过拟合的风险

使用模拟的数据表明，主成分回归能在一些数据集中表现出明显优于OLS的效果。主成分回归实际上不能执行变量选择的效果，因为它拟合的是所有变量的线性组合。在这个意义上，主成分回归跟岭回归很类似(事实上岭回归可以看做主成分回归的连续版)

在进行主成分分析是，通常先对数据进行标准化，防止高方差对主成分的干扰；除非数据都是同一尺度下的

### 6.3.2 偏最小二乘回归

上面描述的PCR方法涉及识别最能代表特征向量变化的线性组合，这是以无监督的形式实现的，因为$Y$不用于确定主成分方向。因此PCR的缺点是虽然获得了最能代表特征变化的方向，但无法保证是最能预测$Y$的变化方向

现在介绍**偏最小二乘回归**(Partial Least Squares, PLS) **是一种**监督(supervised)** 形式的PCR代替。它不仅试图找到最能代表特征向量的变化方向，还试图找到最能贴近$Y$变化的方向

PLS的执行方式是，进行数据标准化后，首先将$Y$和$X$投影到低维空间得到一组**潜变量(Latent Variables)**，再将解释不了的部分放回原矩阵，重新迭代，直至选择前$k$个潜变量进行线性回归分析

PLS在实际应用中并不总是比PCR或岭回归等更好，因此需要结合实际应用

## 6.4 高维问题的思考

### 6.4.1 高维数据

现代科技使得数据的维度越来越多，例如：

- 用SNPs(DNA上的核苷酸)来预测患者的血压、脉搏等。此时的$p$可能达到$500,000$的水平
- 将每个用户输入的关键词都当做一个特征，这常常被称为词袋分析(bag-of-words)，这里的$p$将会达到一个特别大的水平

特征值多的问题被称为**高维的(high-dimensional)**。这些高维度的问题通常被认为是$p > n$的问题，我们讨论的问题也实际$n$略大于$p$的问题

### 6.4.2 高维度下的问题

下面以最小二乘回归为例，介绍高维度问题下需要注意的技术

当$p > n$时，不论特征与响应之间是否有关，都会产生一组完美系数使得残差为0。这使得我们可能误认为特征越多的模型预测效果是最好的，因为它的训练误差或$R^2$是最好的

值得注意的是，在高维度状态下，$C_p$、AIC和BIC是不合适的，因为在高维度情况下根据已知公式得出的$\hat{\sigma}^2 = 0$。同样地，调整后$R^2$也不适用，因为可以很快地得到一个$R^2 = 1$的模型

### 6.4.3 高维度回归

拉索回归可以在高维度模型下显示出一定的优越性能，这是由于它λ取值对模型特征值的压缩实现的。一个良好的λ值可以避免过拟合问题。通过对λ选取显示的拟合情况的观察，我们发现：

- 正则化或收缩在高维问题中非常关键
- 选择合适的调谐参数对于发挥模型性能非常重要
- 测试误差往往随着特征维数的增加而增加，除非额外的特征与响应真正相关

上述第三点被称为**维度灾难(curese of dimensionality)**。这是因为噪声增加的问题维度加剧了过拟合问题

### 6.4.4 高维问题下的结果解释

在高维度问题下执行拉索回归等程序，必须严谨地判断结果。在第三章学习的多重共线性告诉我们，回归中的变量可能存在相关性，这在高维问题中表现为某个特征是其他特征的线性组合，导致我们无法得知到底哪些变量才是对回归结果的真正预测(最多将大系数分给这个特征)

在模型特征选择上得到的模型可能有很好的预测效果，但我们必须谨慎地确定是否它们确实是影响模型的特征(而非它们的线性组合)。

需要注意$p > n$时，不能使用$R^2$和MSE等来度量模型有效性(除了测试集上的)

## 6.5 实验：线性模型和正则化方法

本次实验用到的包包括：

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from statsmodels.api import OLS
import sklearn.model_selection as skm
import sklearn.linear_model as skl
from sklearn.preprocessing import StandardScaler
from ISLP import load_data
from ISLP.models import ModelSpec as MS
from functools import partial
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression
from ISLP.models import \
    (Stepwise,
     sklearn_selected,
     sklearn_selection_path)
from l0bnb import fit_path #这个包的首字母是l不是数字1, 需要用pip预先下载
```

### 6.5.1 子集选择方法

用 `isnan()` 方法可以找出dataframe中的某一列的空值；若要去除空值，可以使用对df对象使用 `dropna()` 方法：

```
#Find nan data and deal
Hitters = load_data('Hitters')
print(np.isnan(Hitters['Salary']).sum()) #The number of missing data
Hitters = Hitters.dropna()
print(Hitters.shape) #Show the shape of datas without nan
```

我们的包中没有计算$C_p$的函数，因此我们需要自己写一个。这个$C_p$将会被我们用于后续模型选择的**策略**或**评分标准**：

```
#Cp for calculate
def nCp(sigma2, estimator, X, Y):
    "negative Cp statistic"
    n, p = X.shape
    Yhat = estimator.predict(X)
    RSS = np.sum((Y - Yhat)**2)
    return -(RSS + 2 * p * sigma2) / n
```

> 这里将$C_p$计算为一个负值，是为了满足我们使用模型**最大化评分标准**的要求

下面我们将使用两种方法来选择模型，它们分别是计算训练集的MSE和$C_p$，首先获取训练集的构造矩阵，并拟合模型：

```
design = MS(Hitters.columns.drop('Salary')).fit(Hitters)
Y = np.array(Hitters['Salary'])
X = design.transform(Hitters)
sigma2 = OLS(Y, X).fit().scale #get sigma^2 of model
```

在第五章提到的特殊函数对象 `partial` 可以帮助我们冻结函数参数，从而快速构造新的子函数，我们下面固定$\sigma^2$：

```
neg_Cp = partial(nCp, sigma2) #Freeze sigma^2
```

我们使用 `ISLP.models` 中的 `Stepwise` 类来构造选择策略。 `Stepwise` 被设计用于执行**步进法**，即前向选择和后项选择。其中：

- `first_peak()` 方法用于一个不定步长前向选择，用到的思想是**先上升后平稳**，即模型在添加参数的过程中性能将在上升后趋于平稳。这个策略在模型性能达到峰值或趋于平稳时停止选择
- `fixed_steps()` 方法用于固定步长的特征选择。指定探寻长度后，将会探寻所有步数后停止。这个策略在模型探索完指定步骤后停止选择

下面构造选择策略：

```
#Search best model by stepwise-first
strategy = Stepwise.first_peak(design,
                               direction='forward',
                               max_terms=len(design.terms)) #最多允许添加的特
征项
```

> 在未指定评分标准时，模型使用MSE作为评分标准

来自 `ISLP.models` 的类 `sklearn_selected` 可以返回模型选择后的构造矩阵(用于拟合模型)，要指定用的模型和策略，默认的评分准是MSE：

```
hitters_MSE = sklearn_selected(OLS,
                               strategy) #返回我们策略下的最优模型
hitters_MSE.fit(Hitters, Y) #根据特征矩阵Hitters和我们要预测的目标进行学习(fit)
print(hitters_MSE.selected_state_) #显示模型的参数
```

> 显然，使用MSE作为评分标准得到的模型必然包含所有的参数，因为特征越多对于训练集的拟合程度越好

用$C_p$作为评分标准拟合模型：

```
#Search best model by neg_Cp function
hitters_Cp = sklearn_selected(OLS,
                              strategy,
                              scoring=neg_Cp)
hitters_Cp.fit(Hitters, Y)
print(hitters_Cp.selected_state_)
```

---

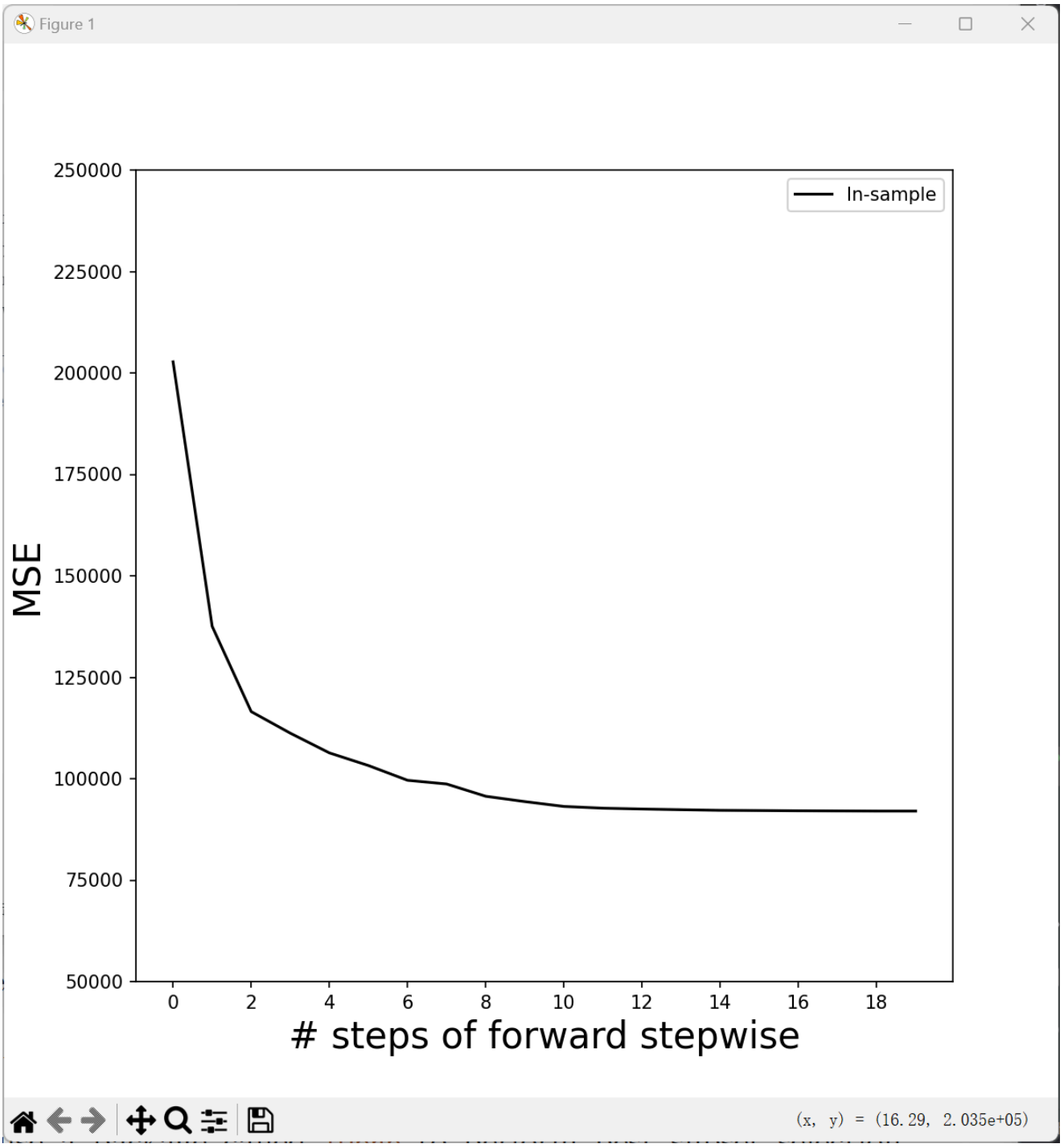
使用交叉检验等方法也可以实现模型选择。为了用验证集方法(validation set approach)选择模型，我们需要存储每次选择模型的路径。这可以通过 `sklearn_selection_path()` 函数实现，这是 `ISLP.models` 中的一个估计器，用于存储选择过程的所有模型路径(即每个步骤选择的特征子集)并为每个模型生成预测。利用我们的存储的结果进行预测，可以得到每个模型的在交叉检验下的MSE。下面获取模型：

```
#Validation set for subsets selection
strategy = Stepwise.fixed_steps(design, #采用固定步长的选择方法
                                len(design.terms), #指定步长为所有特征
                                direction='forward')
full_path = sklearn_selection_path(OLS, strategy) #存储关于不同k个特征值的模型
full_path.fit(Hitters, Y) #为每个模型进行特征学习
Yhat_in = full_path.predict(Hitters) #利用模型(各个特征子集回归模型)预测得到的结果
print(Yhat_in.shape) #查看模型的预测值情况
```

绘图表示MSE和特征值的关系：

```
#Show MSE of each models in full_path
meg_fig, ax = subplots(figsize=(8, 8))
insample_mse = ((Yhat_in - Y[:, None])**2).mean(0) #None匹配维度
n_steps = insample_mse.shape[0] #在模型选择过程中，我们得到的所有模型数量
ax.plot(np.arange(n_steps),
        insample_mse,
        'k', #Color black
        label='In-sample')
ax.set_ylabel('MSE',
              fontsize=20)
ax.set_xlabel('# steps of forward stepwise',
              fontsize=20)
ax.set_xticks(np.arange(n_steps)[::2])
ax.set_ylim([50000, 250000])
ax.legend()
plt.show()
```

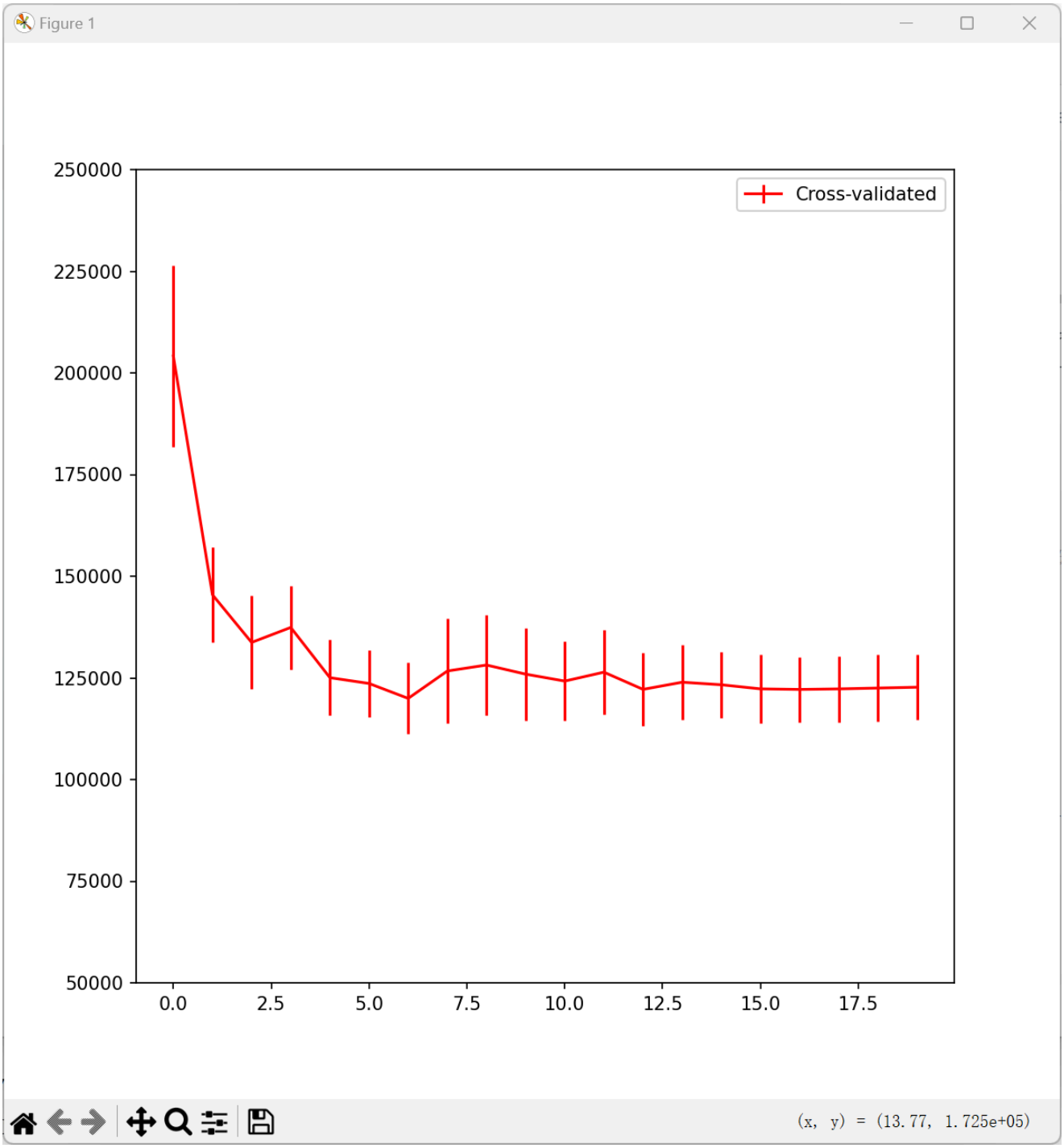
> 上面的None表示空占位，用于两个不同维度的变量进行运算




下面考虑用5折交叉检验选取模型。注意我们的特征选择和模型拟合用到的训练数据都必须是折内的，以保证估计的准确性：

```
#5-folds cross-validation
K = 5
kfold = skm.KFold(K,
                  random_state=0,
                  shuffle=True) #确定fold的选取情况
Yhat_cv = skm.cross_val_predict(full_path,
                                Hitters,
                                Y,
                                cv=kfold) #获取每个折叠模型的预测值
print(Yhat_cv.shape) #显示拟合的情况
cv_mse = []
```

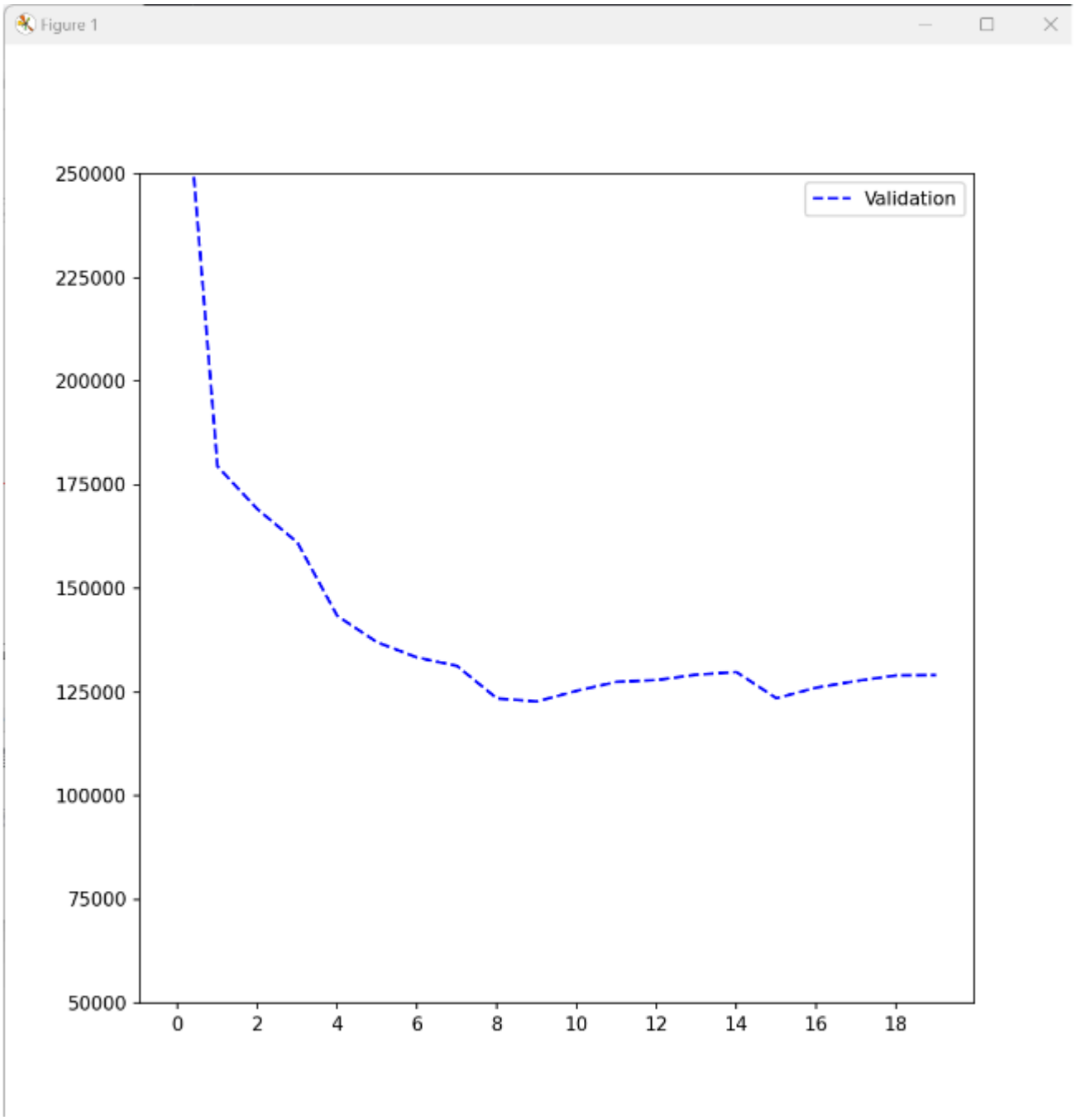
```
for train_idx, test_idx in kfold.split(Y): #计算每个模型对应的MSE
    errors = (Yhat_cv[test_idx] - Y[test_idx, None])**2
    cv_mse.append(errors.mean(0))
cv_mse = np.array(cv_mse).T
print(cv_mse.shape)
meg_fig, ax = subplots(figsize=(8, 8))
ax.errorbar(np.arange(n_steps),
            cv_mse.mean(1),
            cv_mse.std(1) / np.sqrt(K),
            label="Cross-validated",
            c='r')
ax.set_ylim([50000, 250000])
ax.legend()
plt.show()
```




使用20%的检验集的验证集方法，显示MSE和特征值的关系：

```
#考虑用类似的验证集方法作图，选取20%的随机数据作为检验集
mse_fig, ax = subplots(figsize=(8, 8))
validation = skm.ShuffleSplit(n_splits=1,
                              test_size=0.2,
                              random_state=0)
for train_idx, test_idx in validation.split(Y):
    full_path.fit(Hitters.iloc[train_idx],
                  Y[train_idx])
    Yhat_val = full_path.predict(Hitters.iloc[test_idx])
    errors = (Yhat_val - Y[test_idx, None])**2
    validation_mse = errors.mean(0)
ax.plot(np.arange(n_steps),
        validation_mse,
```

```
        'b--',
        label='Validation')
ax.set_xticks(np.arange(n_steps)[::2])
ax.set_ylim([50000, 250000])
ax.legend()
plt.show()
```




`l0bnb` 包可以帮我们快速地获取全局最佳的模型(相比于子集选择的贪心策略)，这是因为它将子集大小作为一个惩罚项，而不是单纯地限制子集的大小。

为了实现灵活的子集大小选取， `l0bnb` 提供了一个解的路径，表示随着惩罚参数 `lambda_0` 的变化， `l0bnb` 是如何为我们选取特征的

下面是一个简单的示例：

```
#最佳子集选择
D = design.fit_transform(Hitters)
D = D.drop('intercept', axis=1) #l0bnb包会自动添加intercept, 因此先舍弃
X = np.asarray(D)
path = fit_path(X,
                Y,
                max_nonzeros=X.shape[1])
print(path[3]) #搜寻路径第4步的结果
```

`l0bnb` 返回结果的解释：

- $B$：拟合的系数向量，其中非零元素表示该特征在模型中被选中，其值表示该特征对目标变量的影响程度。
- $B_0$：截距项，表示当所有特征值为零时，目标变量的预测值。
- $\lambda_0$：当前模型复杂度下的惩罚参数值，用于控制模型的稀疏性。较小的 $\lambda_0$值通常会导致更多的非零系数，即更复杂的模型。
- Time_exceeded：一个布尔值，表示在拟合过程中是否超出了预定的时间限制。这有助于用户了解算法的执行效率和资源消耗情况。

## 6.5.2 岭回归和Lasso回归

我们使用 `sklearn` 中的 `ElasticNet()` 函数来处理岭回归和lasso回归的问题。这是一个混合模型，结合两种正则化方法的优点

`ElasticNet.path()` 方法可用于拟合**正则化路径**，在不同的正则化强度下，它会生成一系列的岭回归和Lasso回归模型，并允许我们在一个$\lambda$下拟合多个模型，通过参数 `l1_ratio` 来调节岭回归和Lasso回归的比例。其中：

- `l1_ratio=0` 表示完全岭回归
- `l1_ratio=1` 表示完全Lasso回归，即只有$l_1$正则化
- `0 < l1_ratio < 1` 表示它们的混合，即ElasticNet

`Elastic` 不会自动对特征进行标准化，因此我们需要先标准化：

```
#加载数据和构造矩阵
Hitters = load_data('Hitters')
Hitters = Hitters.dropna()
design = MS(Hitters.columns.drop('Salary')).fit(Hitters)
D = design.fit_transform(Hitters)
D = D.drop('intercept', axis=1)
X = np.asarray(D)
Y = np.array(Hitters['Salary'])

#标准化特征矩阵
Xs = X - X.mean(0)[None, :]
```

```
X_scale = X.std(0)
Xs = Xs / X_scale[None, :]
```

> 为了获得和原始数据一致的回归系数，我们需要**反标准化回归系数**

我们用一系列的λ值来拟合正则化路径。在默认下，`ElasticNet.path()`会自动选择λ范围，我们现在指定λ的范围(根据$y$的标准差)：

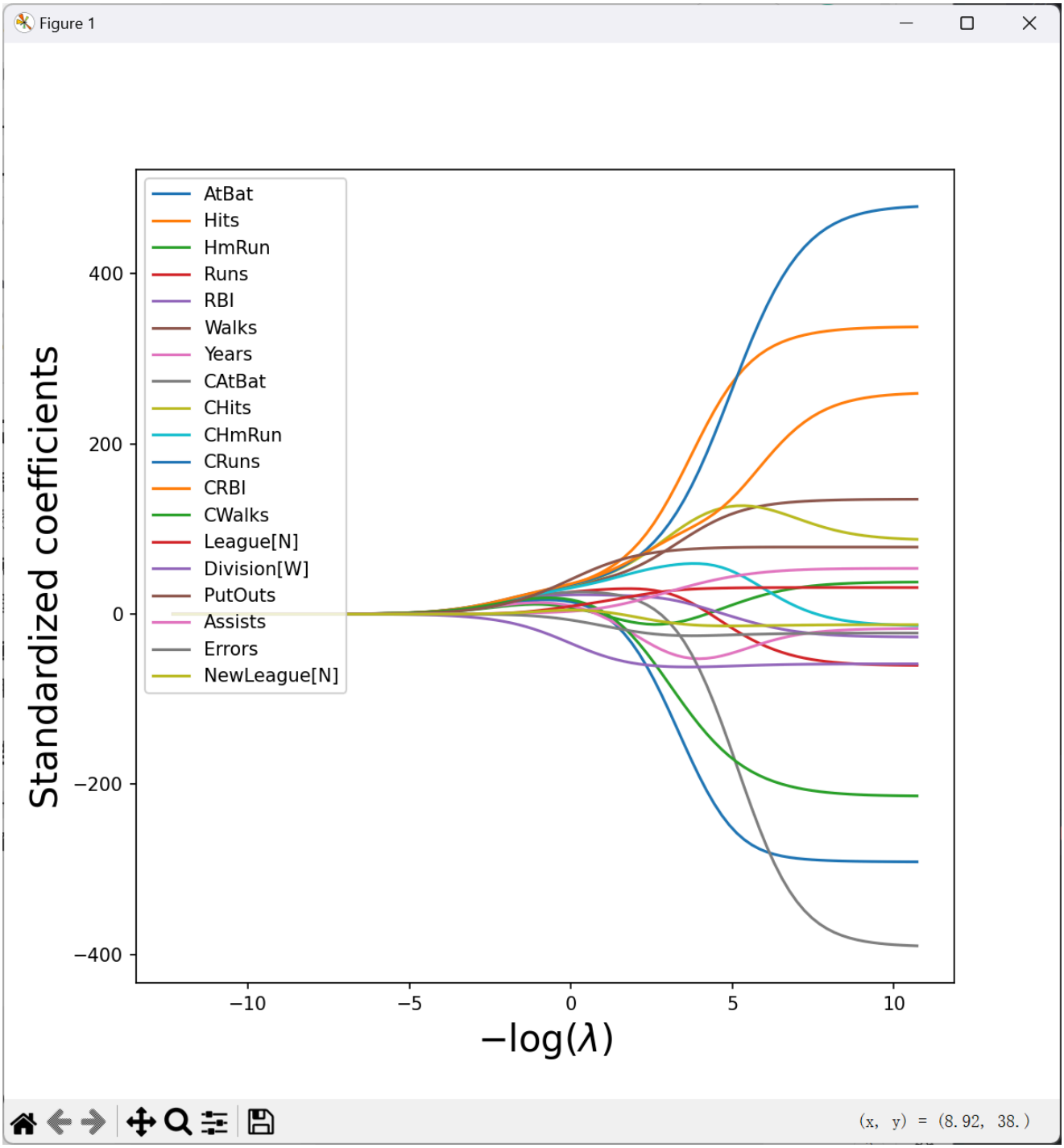
```
lambdas = 10**np.linspace(8, -2, 100) / Y.std() #根据y的标准差对lambda进行缩放，以便后续展示不同lambda的模型变化
soln_array = skl.ElasticNet.path(Xs,
                                 Y,
                                 l1_ratio=0.,
                                 alphas=lambdas)[1]
print("Shape:", soln_array.shape) #输出(19, 100)，表示有19个特征值的100种不同lambda下的模型
```

我们来查看100个模型的系数对应的λ关系：

```
#用岭回归方法拟合100个模型的参数情况
soln_path = pd.DataFrame(soln_array.T,
                         columns=D.columns,
                         index=-np.log(lambdas)) #index指定行的数字情况
soln_path.index.name = 'negative log(lambda)' #指定行的名称
print(soln_path)
```

根据得到的模型路径，可视化各个系数关于λ强度的变化关系：

```
#绘图
path_fig, ax = subplots(figsize=(8, 8))
soln_path.plot(ax=ax, legend=False) #直接用soln_path绘图，即作为对象使用的plot方法
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Standardized coefficients', fontsize=20)
ax.legend(loc='upper left')
plt.show()
```




> 可以观察到，随着λ的变小(从左往右)，各个特征值的取值逐渐变大

我们查看第40步和第60步的特征系数值：

```
#显示40和60步的系数值
beta_hat = soln_path.loc[soln_path.index[39]]
print(lambdas[39], "\n", beta_hat) #输出第40步的lambda值和各个预测特征系数
print(np.linalg.norm(beta_hat)) #输出第40步的特征系数的l2范数
beta_hat = soln_path.loc[soln_path.index[59]]
print(lambdas[59], "\n", np.linalg.norm(beta_hat))
```

`Pipeline` 允许我们将一个模型的训练过程分离开，使得步骤更加清晰，并且能对每个步骤进行命名。下面尝试用 `Pipeline` 将归一化过程和岭回归模型分开：

```
#使用Pipeline显示每个步骤
ridge = skl.ElasticNet(alpha=lambdas[59], #指定第60次的lambda强度
                       l1_ratio=0)
scaler = StandardScaler(with_mean=True, with_std=True)
pipe = Pipeline(steps=[('scaler', scaler), ('ridge', ridge)])
pipe.fit(X, Y)
print(np.linalg.norm(ridge.coef_)) #显示特征系数的l2范数，结果与之前相同
```

为了获得岭回归的检验误差，我们考虑使用验证集方法或者交叉检验方法。之前`Pipeline`划分的步骤可以使我们方便而清晰的进行这个过程

下面考虑使用验证集方法来获得检验误差：

```
#用验证集方法获取检验误差，以lambda=0.01为例
validation = skm.ShuffleSplit(n_splits=1,
                              test_size=0.5,
                              random_state=0)
ridge.alpha = 0.01 #指定lambda强度为0.1
results = skm.cross_validate(ridge,
                             X,
                             Y,
                             scoring='neg_mean_squared_error', #使用负的均方误差作为评估指标，因为cross_validate返回负值分数
                             cv=validation) #使用上面定义的validation作为验证策略
print(-results['test_score'])
```

这里可以观察到$\lambda = 0.01$时的测试误差。我们考虑一个很大的$\lambda$值来确定空模型的测试误差：

```
#获得空模型的检验误差
ridge.alpha = 1e10
results = skm.cross_validate(ridge,
                             X,
                             Y,
                             scoring='neg_mean_squared_error',
                             cv=validation)
print(-results['test_score'])
```

`GridSearchCV`模块实现了对于参数的**穷举搜索**。下面使用随机的$\lambda$值来确定最合适的检验误差和$\lambda$值：

```
#随机选取lambda的值以获得最优参数
param_grid = {'ridge__alpha': lambdas}  # 定义要调整的超参数网格，'ridge__alpha' 是指定的岭回归的正则化参数
grid = skm.GridSearchCV(pipe,  # 使用管道（pipe），它包含了岭回归模型和所有前期处理步骤
```

```
                          param_grid,  # 参数网格（lambdas）中包含了要调节的 λ（alpha）值
                          cv=validation,  # 使用之前定义的验证器（ShuffleSplit 或其他交叉验证方法）
                          scoring='neg_mean_squared_error')  # 使用负均方误差作为评估标准
grid.fit(X, Y)  # 拟合模型，GridSearchCV 会在参数网格上进行穷举搜索
print("Best params: ", grid.best_params_['ridge__alpha'])  # 输出最佳的 λ 参数（alpha）
print("Best estimator: ", grid.best_estimator_)  # 输出最佳的模型（包含最优 λ 参数的岭回归模型）
```

`GridSearchCV` 使用的方法是穷举法，可能导致大量的CPU资源和内存资源消耗，不适用大型数据集。下面是它的部分参数指定：

- 第一个参数 `estimator` 指定需要调优的模型
- 第二个参数 `param_grid` 要搜索的参数字典，表示参数的范围。每个键值对指定参数名和范围
- 第三个参数 `cv` 指定交叉检验的策略
- 第四个参数 `scoring` 制定评估标准，例如 `accuracy` 或 `neg_mean_squared_error`
- 第五个参数 `n_jobs` 指定并行线程数，默认为1，设置为-1表示所有CPU核心

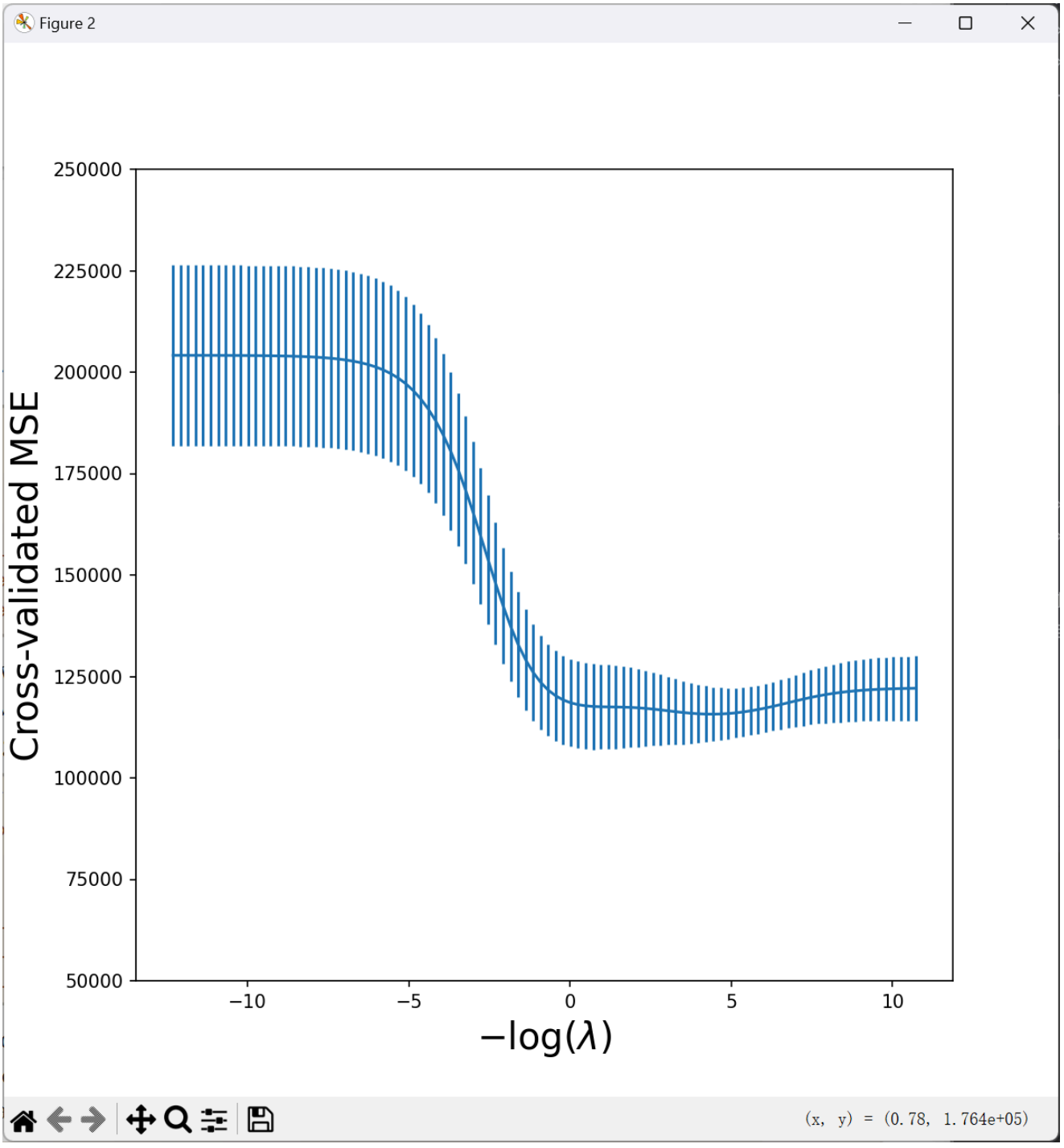
我们可以考虑用交叉检验代替验证集方法来获取检验误差，并绘制出关于λ强度的检验误差图像：

```
#使用交叉检验来获取检验误差
K = 5
kfold = skm.KFold(K,
                  random_state=0,
                  shuffle=True) #确定fold的选取情况
grid = skm.GridSearchCV(pipe,
                        param_grid,
                        cv=kfold,
                        scoring='neg_mean_squared_error')
grid.fit(X, Y)
print("Best params: ", grid.best_params_['ridge__alpha'])
print("Best estimator: ", grid.best_estimator_)

#绘制关于交叉检验的MSE和lambda关系的图像
ridge_fig, ax = subplots(figsize=(8, 8))
ax.errorbar(-np.log(lambdas),
            -grid.cv_results_['mean_test_score'],
            yerr=grid.cv_results_['std_test_score'] / np.sqrt(K)) #指定竖直方向上的误差
ax.set_ylim([50000, 250000])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
```

```
ax.set_ylabel('Cross-validated MSE', fontsize=20)
plt.show()
```

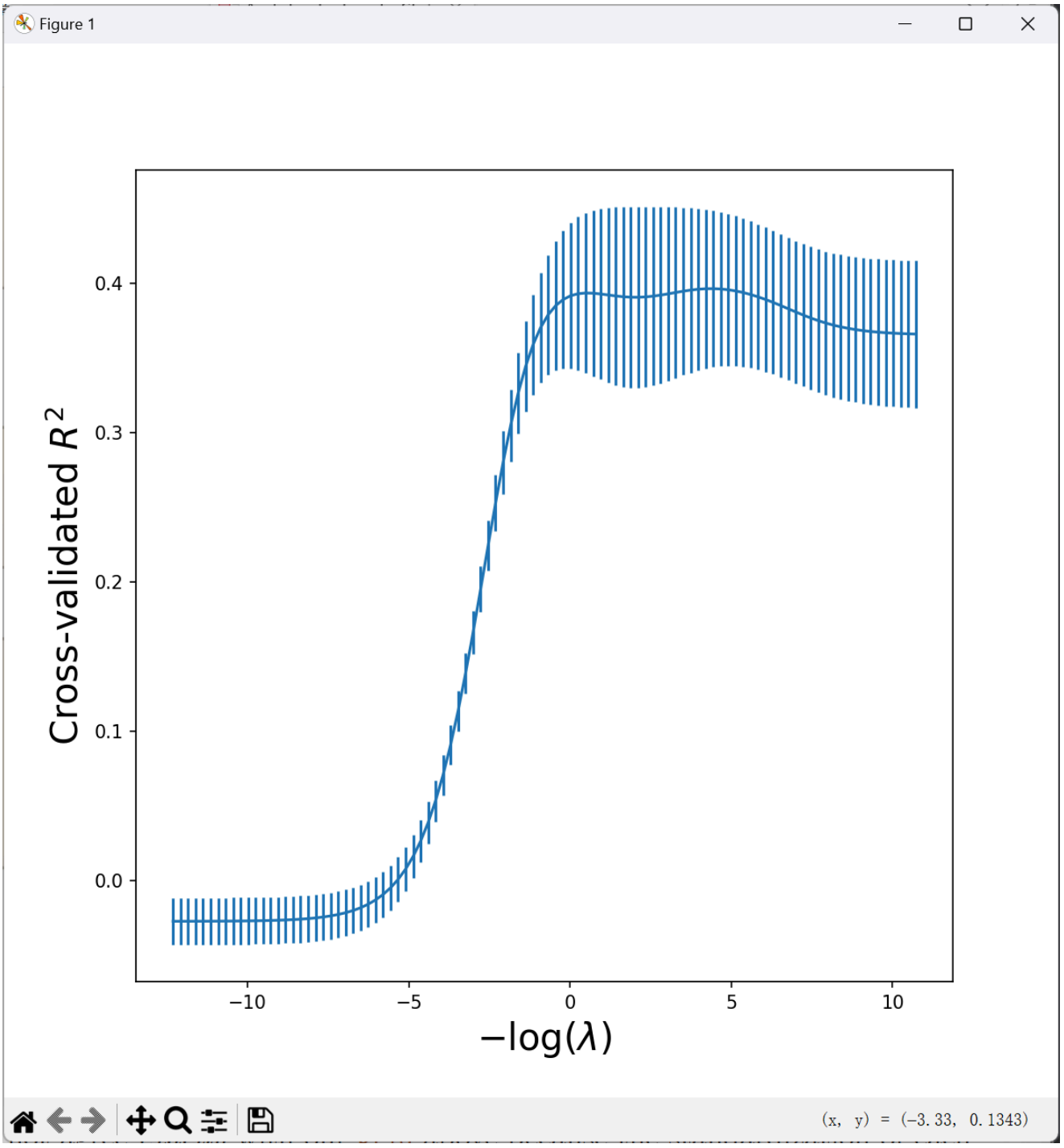
上面的绘图函数用到了`ax.errobar`方法。它可以指定绘图时显示水平、竖直方向上的误差




我们再来查看$R^2$在不同$lambda$强度下的差异：

```
#比较R2在交叉检验中随着选取模型的差异
grid_r2 = skm.GridSearchCV(pipe,
                           param_grid,
                           cv=kfold)
grid_r2.fit(X, Y)
r2_fig, ax = subplots(figsize=(8, 8))
ax.errorbar(-np.log(lambdas),
            grid_r2.cv_results_['mean_test_score'],
```

```
            yerr=grid_r2.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated $R^2$', fontsize=20)
```




对lasso回归、岭回归和弹性网回归在一系列不同λ下拟合得到的模型进行遍历，得到的解是一个**解路径(solution path)**。不同的标准化如每个折单独标准化和整体标准化一次可能导致轻微的结果差异，但整体结果一致。下面考虑对整体进行标准化：

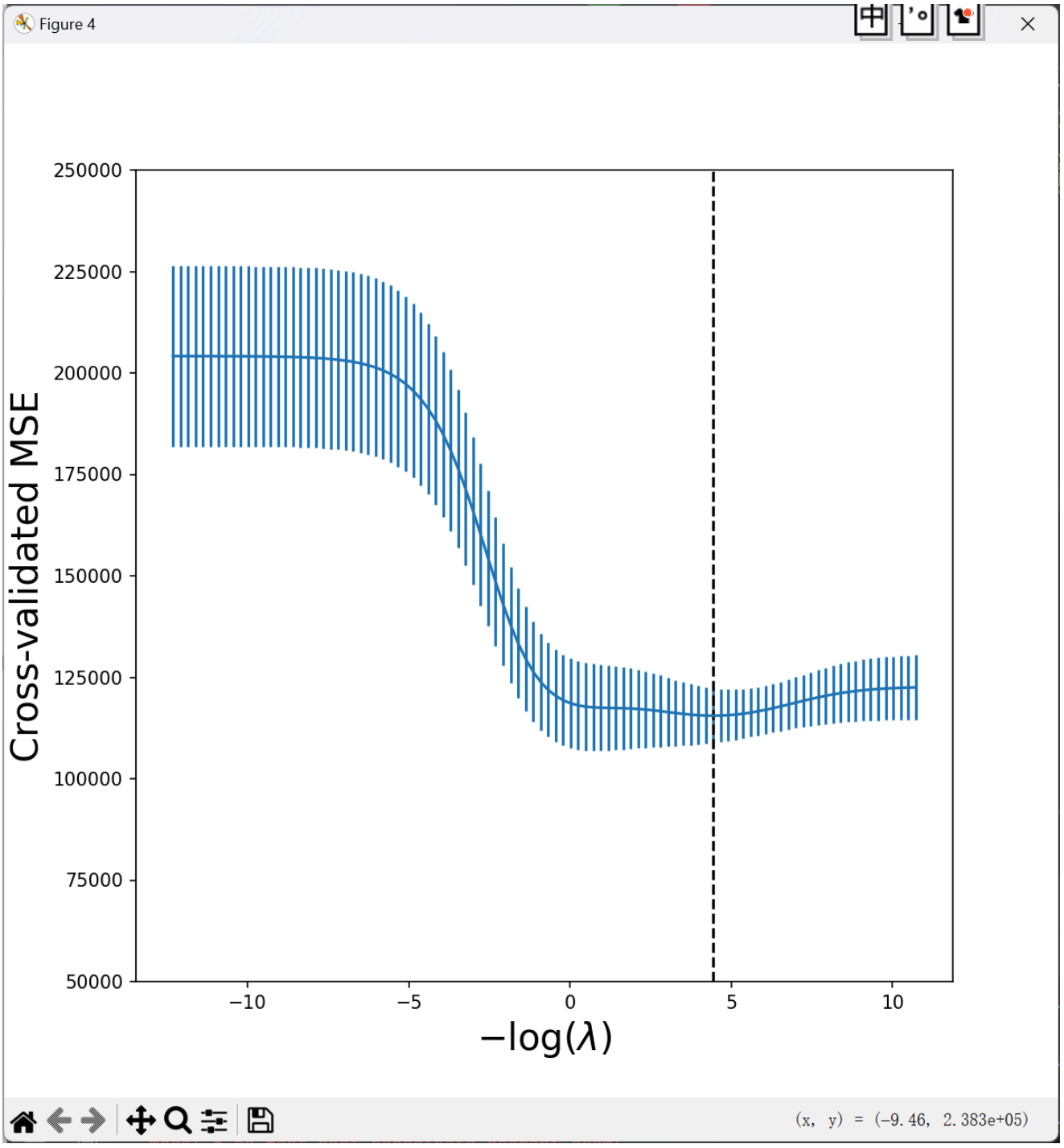
```
ridgCV = skl.ElasticNetCV(alphas=lambdas, #ElasticNetCV是弹性网的交叉检验版本，
自动选择合适的lambda，输入的是候选lambda
                          l1_ratio=0,
```

```
                            cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('ridge', ridgCV)])
pipeCV.fit(X, Y)
tuned_ridge = pipeCV.named_steps['ridge'] #提取ElasticNetCV模型
```

我们寻找图像上的最低测试误差的λ值，注意这里的`tuned_ridge`表示我们提取出的最优模型：

```
ridgCV_fig, ax = subplots(figsize=(8, 8))
ax.errorbar(-np.log(lambdas),
            tuned_ridge.mse_path_.mean(1),
            yerr=tuned_ridge.mse_path_.std(1) / np.sqrt(K))
ax.axvline(-np.log(tuned_ridge.alpha_), c='k', ls='--') #绘制垂直的黑色虚线表示
最优lambda值
ax.set_ylim([50000, 250000])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated MSE', fontsize=20)
plt.show()
print("Min MSE by min lambda: ", np.min(tuned_ridge.mse_path_.mean(1)))
print(tuned_ridge.coef_)
```

这个λ值约为$1.19e-02$，此时的测试误差为$115526.71$，相比于之前得到的$\lambda=4$得到了进一步的改善




进一步地，我们将数据集划分为训练集和测试集，并且分别进行交叉检验。其中训练集的交叉检验用于获取模型的参数，测试集的交叉检验用于评估模型的检验误差：

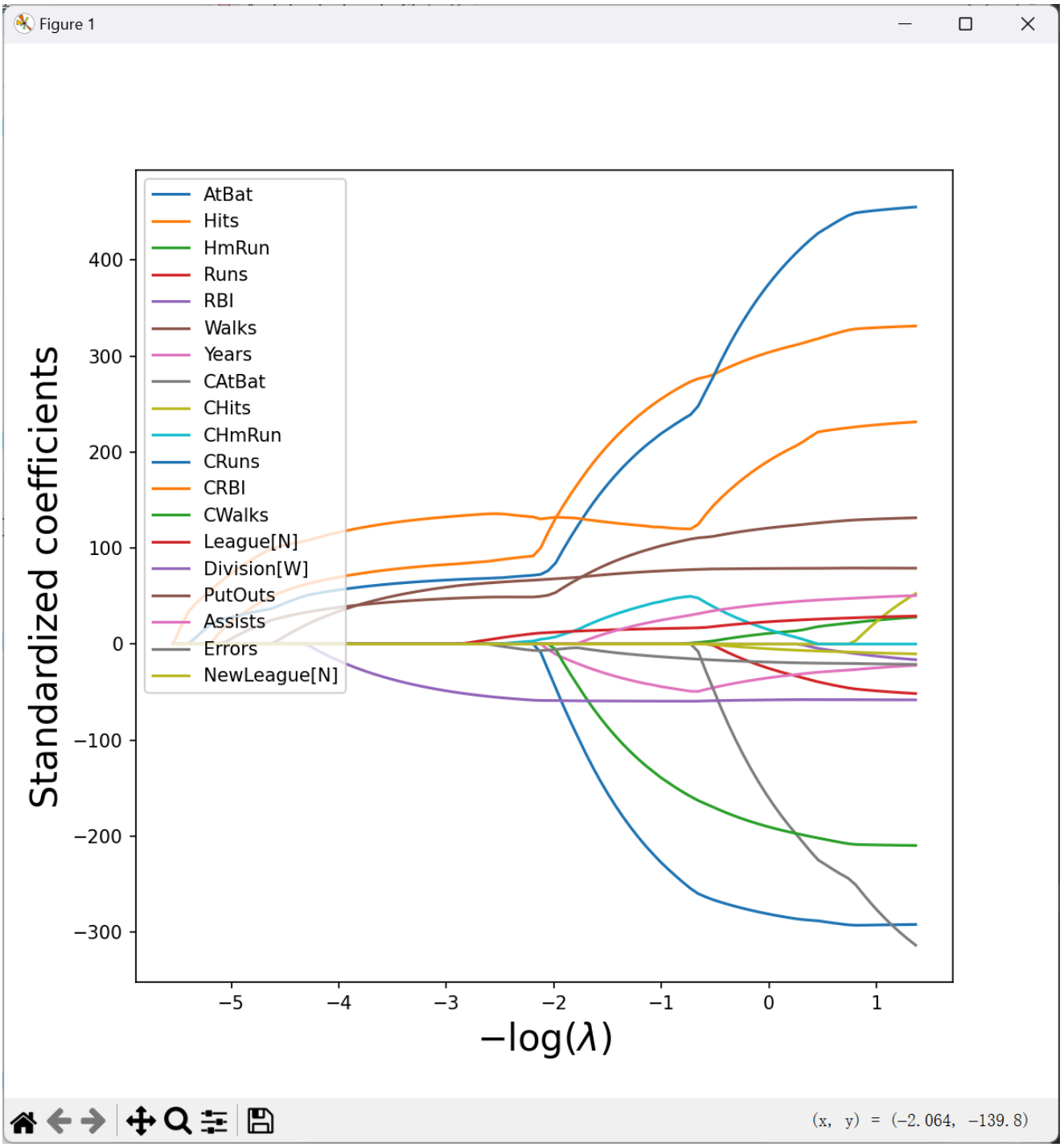
```
#划分数据集的交叉检验模型
outer_valid = skm.ShuffleSplit(n_splits=1, #指定外部的交叉检验模型，划分为测试训练集和验证集
                               test_size=0.25,
                               random_state=1)
inner_cv = skm.KFold(n_splits=5, #指定内部的交叉检验模型，用于拟合参数
                     shuffle=True,
                     random_state=2)
ridgeCV = skl.ElasticNet(alphas=lambdas, #设置弹性网模型，指定内部交叉检验集为inner_cv
                         l1_ratio=0,
```

```
                          cv=inner_cv)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('ridge', ridgeCV)])
results = skm.cross_validate(pipeCV, #用cross_validate评估模型性能，这里是pipe
模型
                             X, #特征矩阵
                             Y, #目标变量
                             cv=outer_valid, #验证集
                             scoring='neg_mean_squared_error') #评分标准
print(-results['test_score']) #得分情况
```

我们考虑使用Lasso回归来拟合模型，试图确定它能否拥有更好的可解释性和检验误差。最后的结果表明，它的检验误差与岭回归差不多：

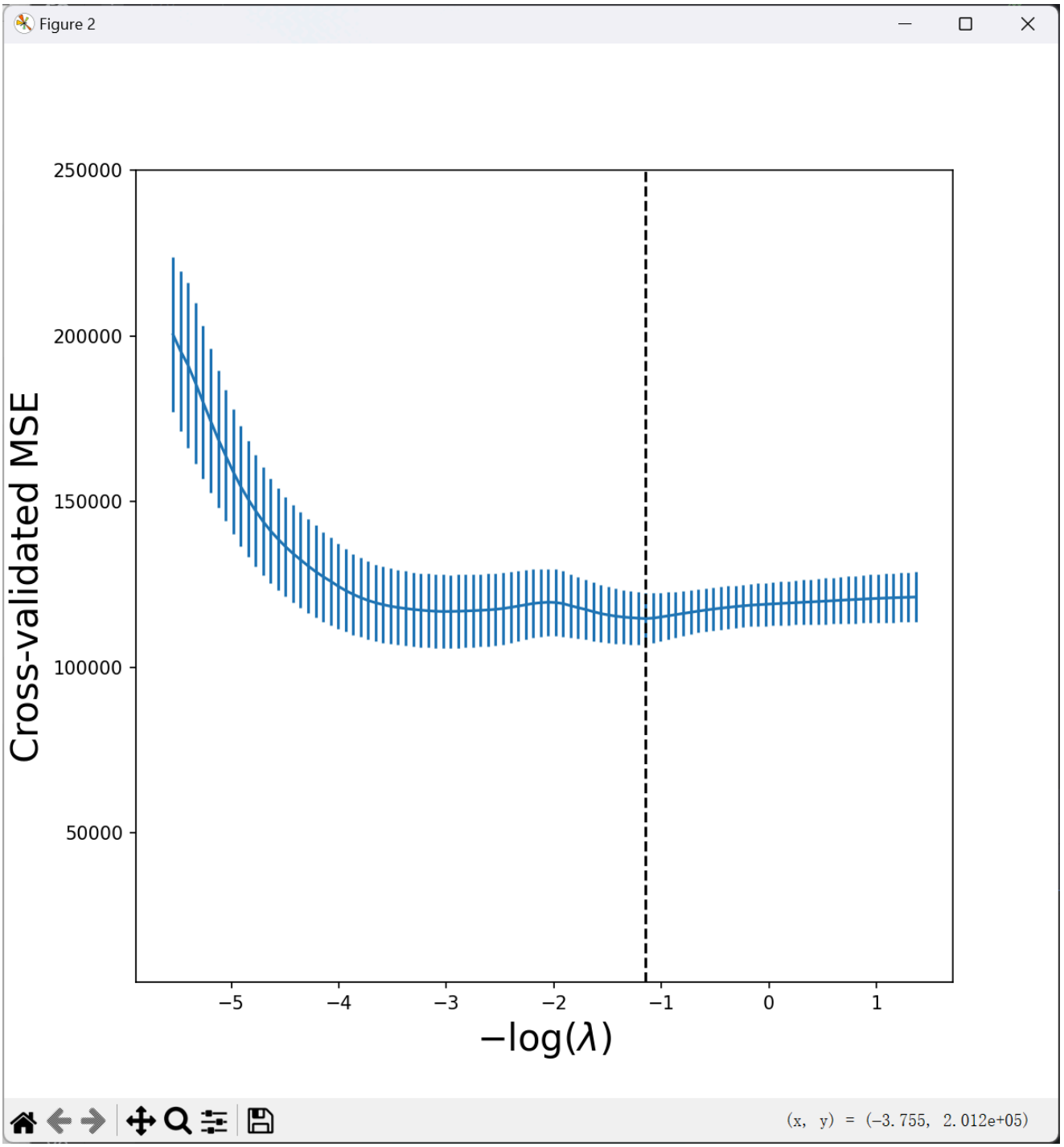
```
"""下面使用lasso模型"""
#lasso模型拟合
lassoCV = skl.ElasticNetCV(n_alphas=100,
                           l1_ratio=1,
                           cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('lasso', lassoCV)])
pipeCV.fit(X, Y)
tuned_lasso = pipeCV.named_steps['lasso']
print("lambda: ", tuned_lasso.alpha_)

#绘制lasso模型的图像
lambdas, soln_array = skl.Lasso.path(Xs,
                                     Y,
                                     l1_ratio=1,
                                     n_alphas=100)[:2]
soln_path = pd.DataFrame(soln_array.T,
                         columns=D.columns,
                         index=-np.log(lambdas))
path_fig, ax = subplots(figsize=(8, 8))
soln_path.plot(ax=ax, legend=False)
ax.legend(loc='upper left')
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Standardized coefficients', fontsize=20)
print(np.min(tuned_lasso.mse_path_.mean(1)))
```




考虑用绘图显示交叉检验法获取最合适的lasso回归λ参数

```
#用交叉检验的方法来选取lasso模型的参数并绘图
lassoCV_fig, ax = subplots(figsize=(8, 8))
ax.errorbar(-np.log(tuned_lasso.alphas_),
            tuned_lasso.mse_path_.mean(1),
            yerr=tuned_lasso.mse_path_.std(1) / np.sqrt(K))
ax.axvline(-np.log(tuned_lasso.alpha_), c='k', ls='--')
ax.set_ylim([5000, 250000])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated MSE', fontsize=20)
print("coef: ", tuned_lasso.coef_)
plt.show()
```




## 6.5.3 PCA和PLS回归

主成分回归的系数取决于数据是否进行标准化，我们用流水线来显示这一过程：

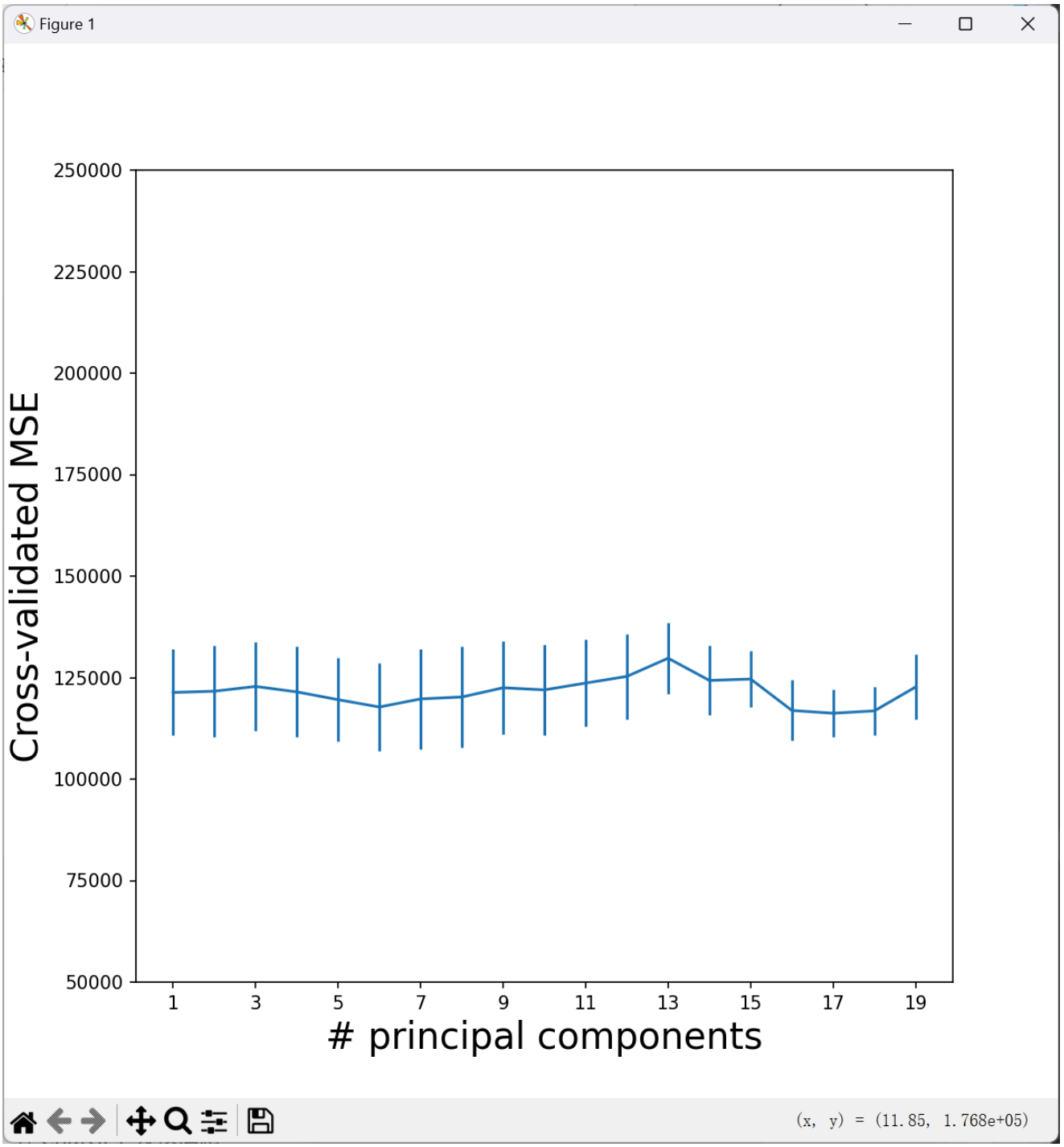
```
#预处理数据
Hitters = load_data('Hitters')
Hitters = Hitters.dropna() #删除空行
design = MS(Hitters.columns.drop('Salary')).fit(Hitters)
D = design.fit_transform(Hitters)
D = D.drop('intercept', axis=1)
X = np.asarray(D)
Y = np.array(Hitters['Salary'])

#执行主成分回归
```

```
pca = PCA(n_components=2) #指定主成分数量为2
linreg = skl.LinearRegression()
scaler = StandardScaler(with_mean=True, with_std=True)
pipe = Pipeline([('scaler', scaler),
                 ('pca', pca),
                 ('linreg', linreg)])
pipe.fit(X, Y)
print(pipe.named_steps['linreg'].coef_) #显示执行主成分回归后的回归系数
```

选择不同数量的主成分存在方差-偏差权衡，我们尝试用交叉检验确定最优的主成分数量。结果显示，当主成分为1或17时，我们有最小的检验误差。考虑到模型的简洁性，我们使用1个主成分：

```
#绘图显示不同的主成分对检验误差的影响
pcr_fig, ax = subplots(figsize=(8, 8))
n_comp = param_grid['pca__n_components']
ax.errorbar(n_comp,
            -grid.cv_results_['mean_test_score'],
            grid.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_ylabel('Cross-validated MSE', fontsize=20)
ax.set_xlabel('# principal components', fontsize=20)
ax.set_xticks(n_comp[::2])
ax.set_ylim([50000, 250000])
```




我们用零模型来作为基准，对比每种主成分模型的解释情况：

```
#拟合一个零模型，用于比较得分
Xn = np.zeros((X.shape[0], 1))
cv_null = skm.cross_validate(linreg,
                             Xn,
                             Y,
                             cv=kfold,
                             scoring='neg_mean_squared_error')
print("null model score: ", -cv_null['test_score'].mean())
```

主成分对象的属性 `explained_variance_ratio_` 可以显示每个主成分的方差，作为衡量每个主成分解释性的情况：

```
print(pipe.named_steps['pca'].explained_variance_ratio_) #显示每个主成分解释的方差比例
```

下面使用PLS回归来拟合模型，并显示最小检验误差下的主成分。结果显示，17个主成分时的检验误差最小，不过和2-3个主成分时相差无几：

```
#构建PLS模型
pls = PLSRegression(n_components=2,
                    scale=True)
pls.fit(X, Y) #拟合好pls模型的参数
param_grid = {'n_components': range(1, 20)}
grid = skm.GridSearchCV(pls, #执行CV下的参数选择过程
                        param_grid,
                        cv=kfold,
                        scoring='neg_mean_squared_error')
grid.fit(X, Y) #选择好最优参数的模型，继续拟合数据
pls_fig, ax = subplots(figsize=(8, 8))
n_comp = param_grid['n_components']
ax.errorbar(n_comp,
            -grid.cv_results_['mean_test_score'],
            grid.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_ylabel('Cross-validated MSE', fontsize=20)
ax.set_xlabel('# principal components', fontsize=20)
ax.set_xticks(n_comp[::2])
ax.set_ylim([50000, 250000])
plt.show()
```

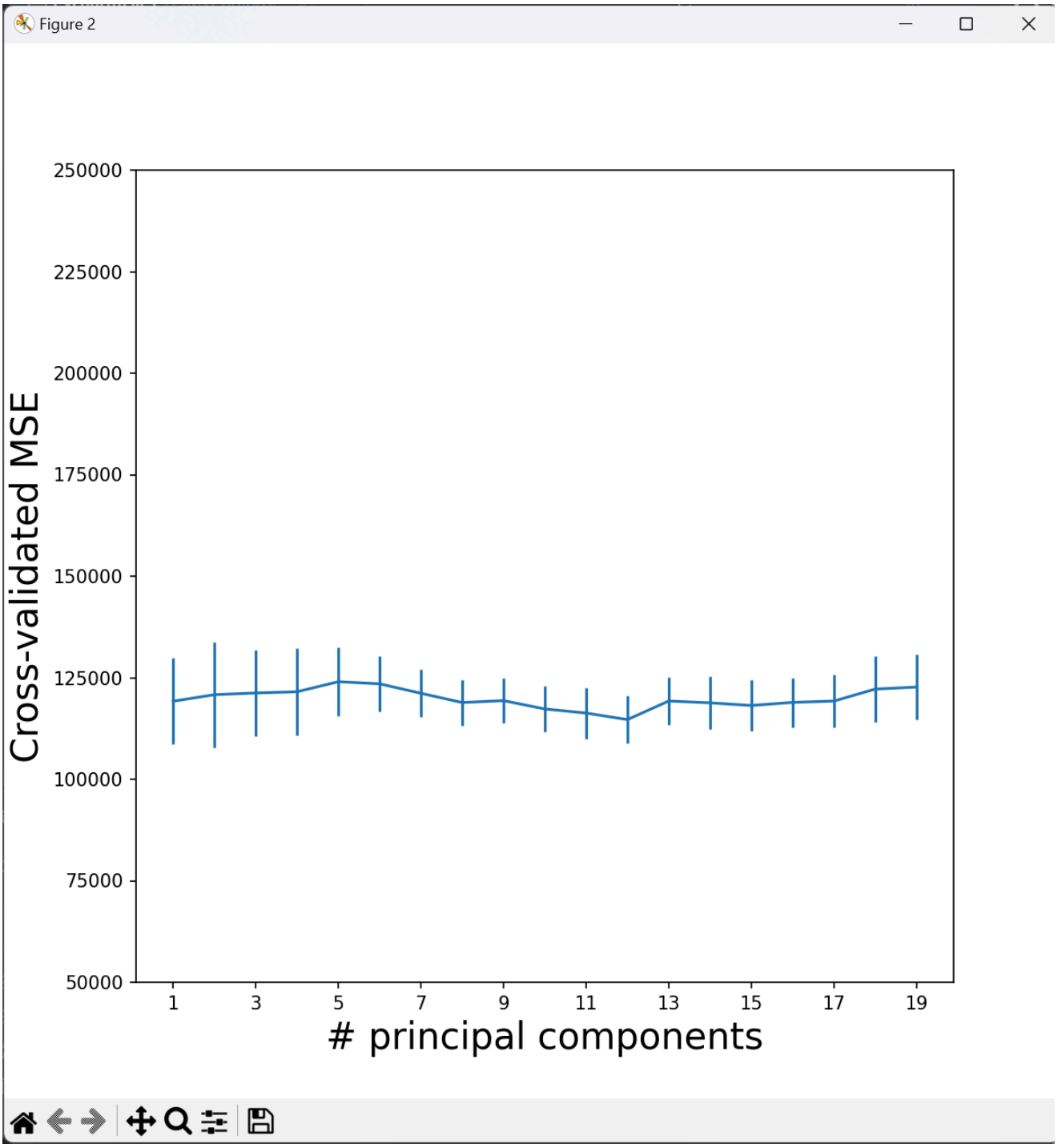


Figure 2
250000
225000
200000
175000
150000
125000
100000
75000
50000
Cross-validated MSE
1
3
5
7
9
11
13
15
17
19
# principal components

#CS
#ML