第五章 重采样方法

重采样方法(Resampling methods) 是现代统计学中不可缺少的方法。为了获取更多的样本信息，我们对样本进行重复采样，得到更多的数据集来训练或评估模型。这一章介绍的重采样方法为交叉检验和自助检验

5.1 交叉检验

在第二章中介绍的测试误差和训练误差分别是使用测试数据集和训练数据集得出的。当我们没有足够大的数据集以划分训练集和测试集时，此时就无法直接用测试集来计算测试误差，只能通过训练集的训练误差来估计。然而，训练误差常常低估测试误差。因此我们必须考虑用别的方法来估计测试误差

5.1.1 验证集方法

验证集方法(the validation set approach) 是一种用来估计测试误差的方法。 通过随机划分测试集和**验证集(Validation Set)** 来分别训练和评估模型

验证集方法的优点在于简单易用，缺点是随机划分的测试集和验证集会导致每次得到的测试误差估计都不一样，且有许多数据未被用于训练模型(训练数据越多，模型越准确)造成浪费，容易高估测试误差，在数据集小的时候不实用

> 验证集方法可以用于大数据的模型。此时，需要将数据集划分为训练集、验证集和测试集三个集合，用训练集拟合模型，再用验证集评估参数选择方案，最后用测试集评估选好参数的模型效果

5.1.2 留一交叉检验

留一交叉检验(Leave-one-out Cross Validation, LOOCV) 的思想近似于验证集方法，但能克服其缺点

留一交叉检验的步骤如下：

- 划分检验集与训练集。特别地，检验集仅包含一个观测值，训练集包含$n-1$个观测值
- 用训练集拟合模型，并用来估计$MSE_1=(y_1-\hat{y}_1)^2$，这个估计近似于MSE_1的无偏估计

  > 注意不同MSE值不同

- 重复步骤1和2，不同的是更新检验集，直到所有的观测数据都被使用过一次
- 计算估计的总体MSE

$$CV_{(n)}=\frac{1}{n}\sum_{i=1}^{n}MSE_i$$

留一交叉检验相比于检验集方法的优点在于：

- 训练模型的偏差小，因为训练集大
- 不会因为数据集的随机分割而导致不同的结果，重复试验结果相同

留一交叉检验在n大的时候计算量很大，因为留一交叉检验需要计算n次，然而用下面的等式可以用一次计算得出$CV_(n)$

在最小二乘回归或多项式回归中，下面的等式成立：

$$CV_{(n)} = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{1-h_i}\right)^2$$

其中h_i是之前定义的杠杆值，计算公式为：

$$h_i = \frac{1}{n} + \frac{(x_i - \overline{x})^2}{\sum_{i'=1}^{n}(x_{i'} - \overline{x})^2}$$

> 这个公式用h_i杠杆来影响残差，使得等式成立

LOOCV是一种非常通用的方法，可以与任何一种预测模型结合使用，例如逻辑回归和LDA。需要注意的是，在这些模型中，上面的杠杆值估计公式不成立，必须重新拟合n次模型

5.1.3 K-fold 交叉检验

一种替代LOOCV的检验方式是**K-fold交叉检验(K-fold Cross Validation)** 。这种方法非常类似LOOCV，不同的是他将数据集划分为k个**折叠(fold)**，将其中的一个折叠用于检验集，另外$k-1$个用于训练模型，重复k次后有：

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} MSE_i$$

不难发现，LOOCV是K-fold交叉检验的一种特例。K-fold交叉检验的优势不仅体现在计算次数减少上，还有可能带来非计算的方差-偏差权衡

K-fold交叉检验得到的测试误差估计仍然是变动的，不过相较于LOOCV，这种变动在合适的k上变化不大

用已知分布的模拟数据检验LOOCV和$K=10$的K-fold交叉检验可发现，二者的区别并不太大

有时我们的目的并不是观察不同参数模型拟合过程中测试MSE的变动，而是寻找拟合模型的最小MSE点，这是也可以借助交叉检验，因为上面的验证中，交叉检验预测的最低MSE点与实际差距不大

5.1.4 K-fold交叉检验的偏差-方差权衡

抛开计算问题，一个K-fold交叉检验的潜在优势是偏差-方差权衡

用LOOCV训练的模型虽然偏差低于K-fold交叉检验，但是方差往往是前者高于后者。这是因为LOOCV虽然拟合了多个模型，但是这些模型的相关性非常高。数学上的证明指出，这样得到的测试MSE往往具有更大的方差

为了偏差-方差权衡，我们一般使用$k=5$或$k=10$的K-fold交叉检验

5.1.5 分类器中的交叉检验

在分类器中，我们关心的一个类似的指标是误分率。例如，此时LOOCV计算的是：

$$CV_{(n)}=\frac{1}{n}\sum_{i=1}^{n}\mathrm{ERR}_i$$

其中，$\mathrm{Err}_i=I(y_i\neq\hat{y}_i)$。K-fold交叉检验的定义与之类似

很容易发现随着模型灵活度的增加，训练误差的趋势是一直减小的，这是因为模型会尽可能地拟合每一个使用的数据。然而，用交叉检验估计的测试误差则呈一个U型曲线，表示出过拟合的现象。进一步的模拟表明，这个U型曲线是对真实测试误差的很好的估计

> 值得注意的是，交叉检验对测试误差的最小值估计也非常接近

5.2 自助法

自助法(bootstrap) 是一种广泛而常用的重采样方法，常用于统计量分布的估计和参数置信区间的计算，也可以用于评估模型性能

下面举例说明自助法的应用。假设需要投资X和Y两类商品(X,Y表示收益率)，我们将部分资金α投资于X，另一部分$1-\alpha$投资于Y。现在，我们希望确定一个组合来最小化风险$Var(\alpha X+(1-\alpha)Y$，也即是最小化：

$$\alpha=\frac{\sigma_Y^2-\sigma_{XY}}{\sigma_X^2+\sigma_Y^2-2\sigma_{XY}}$$

实际中，σ是未知的，我们通过过去的收益率来估计$\hat{\sigma}$，得到：

$$\hat{\alpha}=\frac{\hat{\sigma}_Y^2-\hat{\sigma}_{XY}}{\hat{\sigma}_X^2+\hat{\sigma}_Y^2-2\hat{\sigma}_{XY}}$$

为了估计$\hat{\sigma}$，我们考虑生成用数据集生成100个X,Y数据对，估计一次$\hat{\alpha}$，将这个过程重复1000次，从而得到1000个$\hat{\alpha}_i$的估计，用他们的均值来估计总体α：

$$\overline{\alpha}=\frac{1}{n}\sum_{i=1}^{n}\hat{\alpha}_i$$

我们可以发现这个模拟值与真实的α非常接近，标准差也非常小

当我们只有一个数据集时，使用自助法可以生成许多用于估计总体参数的样本，而避免了继续向总体抽取样本。自助法通过有放回地向样本抽样得到多个新样本，估计参数的均值完成对总体估计。实践中，这个估计非常的好

> 这里作的假设是样本分布可以代表总体分布
> 一般来说，抽取新样本的次数为1000或2000次

5.3 实验：交叉检验和自助法

本次实验用到的包有：

```
import numpy as np
import statsmodels.api as sm
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
                         summarize ,
                         poly)
from sklearn.model_selection import train_test_split, KFold, ShuffleSplit
from functools import partial
from sklearn.model_selection import \
    (cross_validate,
    KFold,
    ShuffleSplit)
from sklearn.base import clone
from ISLP.models import sklearn_sm
```

5.3.1 验证集方法

使用 train_test_split 函数来划分训练集和验证集，一般的，取验证集和训练集的数量相等(注意是随机选取等量而非连续的区间)：

```
#加载数据集并划分Validation
Auto = load_data('Auto')
Auto_train, Auto_valid = train_test_split(Auto,
                                          test_size=196,
                                          random_state=0) #设置随机数种子便于复现结果

#设置测试集和检验集
hp_mm = MS(['horsepower'])
X_train = hp_mm.fit_transform(Auto_train)
y_train = Auto_train['mpg']
model = sm.OLS(y_train, X_train)
results = model.fit()
X_valid = hp_mm.transform(Auto_valid)
y_valid = Auto_valid['mpg']
valid_pred = results.predict(X_valid)
print(np.mean((y_valid - valid_pred)**2)) #显示检验集的均方误差
```

考虑到多次计算的计算量，我们利用函数来计算多项式线性回归的检验均方误差：

```
#定义估计高次多项式回归验证误差的函数
def evalMSE(terms,
            response,
            train,
            test):
    mm = MS(terms)
    X_train = mm.fit_transform(train)
    y_train = train[response]
    X_test = mm.transform(test)
    y_test = test[response]

    results = sm.OLS(y_train, X_train).fit()
    test_pred = results.predict(X_test)
    return np.mean((y_test-test_pred)**2)

#使用函数来计算对应多项式的均方误差估计
MSE = np.zeros(3)
for idx, degree in enumerate(range(1, 4)):
    MSE[idx] = evalMSE([poly('horsepower', degree)],
                       'mpg',
                       Auto_train,
                       Auto_valid)
print(MSE) #显示不同多项式回归的检验均方误差
```

实际上，不同的检验集划分会导致不同的检验均方误差结果：

```
#更换检验集
Auto_train, Auto_valid = train_test_split(Auto, test_size=196,
random_state=3)
MSE = np.zeros(3)
for idx, degree in enumerate(range(1, 4)):
    MSE[idx] = evalMSE([poly('horsepower', degree)],
                       'mpg',
                       Auto_train,
                       Auto_valid)
print(MSE)
```

虽然数值上结果不一致，但我们发现他们对模型评估效果是类似的

5.3.2 交叉检验

在数据处理的过程中，我们经常需要对数据集用不同的函数处理，将其中的结果互相使用以达到目的。为此，我们需要用到**包装(wrapper)** 的概念

`ISLP` 包中的 `sklearn_sm()` 函数为我们提供了这样一个包装，使得我们很容易对用 `statsmodels` 拟合的模型用 `sklearn` 进行交叉检验

> 这个函数用于广义线性回归中的模型，来自 `statemodels` 。这个包不兼容 `scikit-learn` ，因此需要进行封装使得 `cross_validate` 函数能正常运行

尝试使用 `ISLP` 的包装：

```
#用封装好的函数来拟合模型
Auto = load_data('Auto')
hp_model = sklearn_sm(sm.OLS,
                      MS(['horsepower'])) #两个参数分别是sm模型和特征矩阵
X, Y = Auto.drop(columns=['mpg']), Auto['mpg']
cv_results = cross_validate(hp_model, #满足scikit-learn API 的模型
                            X,
                            Y,
                            cv=Auto.shape[0]) #指定LOOCV方法，即行数
cv_err = np.mean(cv_results['test_score']) #返回一个字典，我们关心的是其中一个键值对'test_socre'
print(cv_err)
```

> 这里使用包装的目的是后续的 `cross_validate` 函数需要用到

`cross_validate` 函数接受参数为：

```
cross_validate(model, X, y, cv=?)
```

其中的 `model` 必须满足 `scikit-learn` 定义的API。使用 `scikit-learn` 的原生模型可以满足API，否则需要用 `wrapper` 进行包装。对于 `statesmodel` 中的模型，下面给出一种包装实例：

```
from sklearn.base import BaseEstimator, ClassifierMixin

class MyCustomModel(BaseEstimator, ClassifierMixin):
    def __init__(self, param1=1.0):
        self.param1 = param1

    def fit(self, X, y):
        # 训练代码
        self.model = my_model_train(X, y, self.param1)  # 自定义的训练函数
        return self

    def predict(self, X):
        # 预测代码
        return my_model_predict(X, self.model)  # 自定义的预测函数
```

其中 `sklearn.base` 中的类 `BaseEstimator, ClassifierMixin` 能方便我们自定义wrapper

> 为了避免使用wrapper，尽可能调用 `scikit-learn` 的原生模型

这个包装直接帮助我们用自定义的特征向量来拟合一个 `OLS` 模型，同时用 `cross_validate` 函数来进行交叉检验。这个函数接受模型、特征向量、预测值和可选参数 `cv` (用于指定折叠数)来进行交叉检验

拟合多项式回归并评估：

```
#用for循环拟合1-5次多项式的回归
cv_error = np.zeros(5)
H = np.array(Auto['horsepower'])
M = sklearn_sm(sm.OLS)
for i, d in enumerate(range(1, 6)):
    X = np.power.outer(H, np.arange(d+1)) #生成多项式特征矩阵，用于下面的交叉检验
    M_CV = cross_validate(M, X, Y, cv=Auto.shape[0]) #LOOCV模型
    cv_error[i] = np.mean(M_CV['test_score'])
print(cv_error)
```

其中，我们使用 `np.power.outer()` 方法来生成多项式的特征矩阵， `outer()` 表示求外积。例如当$d=3$时，矩阵X会有4列H^0, H^1, H^2, H^3

> 举例说明多项式特征矩阵：$X_1+X_2+X_1^2+X_2^2$，需要四项的值

通过改变 `cv` 参数的值，我们可以得到K-fold交叉检验对测试误差的估计：

```
#K-fold交叉检验，K=10
cv_error = np.zeros(5)
cv = KFold(n_splits=10, shuffle=True, random_state=0) #Kfold按顺序分割数据集，
指定shuffle后改为随机分割
for i, d in enumerate(range(1, 6)):
    X = np.power.outer(H, np.arange(d+1))
    M_CV = cross_validate(
        M,
        X,
        Y,
    cv=cv)
    cv_error[i] = np.mean(M_CV['test_score'])
print(cv_error)
```

> 时间序列不应指定 `shuffle=True`

由于 `cross_validate` 函数的灵活性，我们可以用它来模拟验证集方法：

```
#用ShuffleSplit来实现验证集方法
validation = ShuffleSplit(n_splits=1,
                          test_size=196,
                            random_state=0)
results = cross_validate(hp_model,
                         Auto.drop(['mpg'], axis=1),
```

```
                         Auto['mpg'],
                         cv=validation)
print(results['test_score'])
```

`ShuffleSplit`类可以让我们指定划分次数，以实现多次验证集方法实现。这里可以用作验证集方法方差的估计：

```
#估计检验误差的方差
validation = ShuffleSplit(n_splits=10,
                          test_size=196,
                          random_state=0) #这里会划分十次不同的validation，因此validation变量有10维
results = cross_validate(hp_model,
                         Auto.drop(['mpg'], axis=1),
                         Auto['mpg'],
                         cv=validation) #这里会计算10次
print(results['test_score'].mean(), results['test_score'].std())
```

> 事实上，这里计算的`std()`不是验证集方法方差的有效估计，因为样本的相关性。然而，这个方法是一种有效的对**Monte Carlo variation**的估计

5.3.3 自助法

5.3.3.1 用自助法估计统计量的精确度

这里用到了5.2部分中的例子，使用的数据集是`Portfolio`

利用5.2中的公式，估计标准差：

```
#加载数据集
Portfolio = load_data('Portfolio')

#定义函数，计算重抽样下的标准差
def alpha_func(D, idx): #根据计算公式计算统计量
    cov_ = np.cov(D[['X', 'Y']].loc[idx], rowvar=False)
    return ((cov_[1,1] - cov_[0,1]) /
            (cov_[0, 0]+cov_[1, 1]-2*cov_[0,1]))

#计算100个抽样观测对下的标准差
print(alpha_func(Portfolio, range(100)))

#从100个观测对里挑选变量，重新构建自助集
rng = np.random.default_rng(0)
print(alpha_func(Portfolio,
                 rng.choice(100,
                            100,
                            replace=True))) #表示索引的随机抽取
```

`np.random.default_rng()` 方法用于生成随机数种子，`rng.choice()` 则表示用这个种子的随机数进行随机抽取(上面的例子是100个数字抽取100个)，可选参数 `replace` 表示有放回地

> 这里的100是个magic number，实际上是Dataframe的行数

构造一个仅以数据集为参数的(其他为可选参数)的函数来计算每个数据集的自助法标准差：

```
#构造一个函数，计算仅以df为参数的自助法标准差
def boot_SE(func,
            D,
            n=None,
            B=1000,
            seed=0):
    rng = np.random.default_rng(seed)
    first_, second_ = 0, 0
    n = n or D.shape[0]
    for  _ in range(B):
        idx = rng.choice(D.index,
                         n,
                         replace=True)
        value = func(D, idx)
        first_ += value
        second_ += value**2
    return np.sqrt(second_ / B - (first_ / B)**2)

#计算标准误差
alpha_SE = boot_SE(alpha_func,
                   Portfolio,
                   B=1000,
                   seed=0)
print(alpha_SE)
```

5.3.3.2 用自助法估计系数的标准差

自助法可用于估计系数的标准差。虽然我们的模型可以用公式计算，但是自助法的估计也能给我们提供一些有用的信息

构建函数计算自助法系数标准差：

```
#构建一个自助法估计系数标准差函数，试图实现通用计算
def boot_SE(func,
            D,
            n=None,
            B=1000,
            seed=0):
   rng = np.random.default_rng(seed)
   first_, second_ = 0, 0
   n = n or D.shape[0]
```

```
    for  _ in range(B):
        idx = rng.choice(D.index,
                         n,
                         replace=True)
        value = func(D, idx)
        first_ += value
        second_ += value**2
    return np.sqrt(second_ / B - (first_ / B)**2)

#加载数据集
Auto = load_data('Auto')

#定义函数
def boot_OLS(model_matrix, response, D, idx):
    D_ = D.loc[idx]
    Y_ = D_[response]
    X_ = clone(model_matrix).fit_transform(D_) #clone拷贝一个模型，免去了重新拟合
的计算
    return sm.OLS(Y_, X_).fit().params

#冻结参数
hp_func = partial(boot_OLS, MS(['horsepower']), 'mpg')

#创建实例
rng = np.random.default_rng(0)
SEs = np.array([hp_func(Auto,
                  rng.choice(Auto.index,
                             392,
                             replace=True)) for _ in range(10)])
print(SEs)
hp_se = boot_SE(hp_func,
                Auto,
                B=1000,
                seed=10)
print(hp_se)

#设计模型来检验估计值是否接近
hp_model = sklearn_sm(sm.OLS,
                      MS(['horsepower']))
hp_model.fit(Auto, Auto['mpg'])
model_se = summarize(hp_model.results_)['std err']
print(model_se)
```

`partial` 类用于固定函数的参数为定值，从而生成新的子函数。注意到二者计算的标准差略有不同，这是因为公式法计算的标准差需要满足随机误差ϵ不变(线性假设)的假设，而自助法没有假设。在这个意义上，自助法对参数标注差的估计可能更好

用自助法估计二次模型：

```
#计算二次模型的自助法模型标准差
quad_model = MS([poly('horsepower', 2, raw=True)])
quad_func = partial(boot_OLS,
                    quad_model,
                    'mpg')
print(boot_SE(quad_func, Auto, B=1000))

#计算回归模型用公式计算的标准差用于比较
M = sm.OLS(Auto['mpg'],
           quad_model.fit_transform(Auto))
print(summarize(M.fit())['std err'])
```

#CS
#ML