Labs

February 21, 2025

1 Lab: Cross-Validation and the Bootstrap

1.1 Import notebook functions

[1]:
```
from notebookfuncs import *
```

1.2 Import libraries

[2]:
```
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
```

1.3 New libraries needed for Cross-Validation and the Bootstrap

[3]:
```
from functools import partial
from sklearn.model_selection import (cross_validate, KFold , ShuffleSplit)
from sklearn.base import clone
from ISLP.models import sklearn_sm
```

1.4 Validation Set Approach

We explore the use of the validation set approach in order to estimate the test error rates that result from fitting various linear models on the **Auto** data set.

We use the function `train_test_split()` to split the data into training and validation sets. As there are 392 observations, we split into two equal sets of size 196 using the argument `test_size=196`. It is generally a good idea to set a random seed when performing operations like this that contain an element of randomness, so that the results obtained can be reproduced precisely at a later time. We set the random seed of the splitter with the argument `random_state=0`.

[4]:
```
Auto = load_data('Auto')
Auto_train , Auto_valid = train_test_split(Auto , test_size =196, random_state␣
  ↪=0)
```

Now we can fit a linear regression using only the observations corresponding to the training set `Auto_train`.

[5]:

```
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();
```

We now use the `predict()` method of results evaluated on the model matrix for this model created using the validation data set. We also calculate the validation Mean Square Error (MSE) of our model.

[6]:

```
X_valid = hp_mm.transform(Auto_valid)
y_valid = Auto_valid['mpg']
valid_pred = results.predict(X_valid)
mse = np.mean((y_valid - valid_pred)**2)
```

[6]:

```
23.61661706966988
```

[7]:

```
printmd(f"Hence our estimate for the validation MSE of the linear regression fit is **{mse:0.2f}**.")
```

Hence our estimate for the validation MSE of the linear regression fit is **23.62**.

We can also estimate the validation error for higher-degree polynomial regressions. We first provide a function `evalMSE()` that takes a model string as well as a training and test set and returns the MSE on the test set.

[8]:

```
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)
```

Let's use this function to estimate the validation MSE using linear, quadratic and cubic fits. We use the `enumerate()` function here, which gives both the values and indices of objects as one iterates over a for loop.

[9]:

```
MSE = np.zeros(3)
for idx, degree in enumerate(range(1, 4)):
  MSE[idx] = evalMSE([poly('horsepower', degree)], 'mpg', Auto_train, Auto_valid)
MSE
```

```
[9]: array([23.61661707, 18.76303135, 18.79694163])
```

```
[10]: printmd(f"These error rates are **{MSE[0]:0.2f}**, **{MSE[1]:0.2f}**, and␣
       ↪**{MSE[2]:0.2f}**, respectively.")
```

These error rates are **23.62**, **18.76**, and **18.80**, respectively.

If we choose a different training/validation split instead, then we can expect somewhat different errors on the validation set.

```
[11]: 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)
      MSE
```

```
[11]: array([20.75540796, 16.94510676, 16.97437833])
```

```
[12]: printmd(f"The error rates now are **{MSE[0]:0.2f}**, **{MSE[1]:0.2f}**, and␣
       ↪**{MSE[2]:0.2f}**, respectively.")
```

The error rates now are **20.76**, **16.95**, and **16.97**, respectively.

These results are consistent with our previous findings: a model that predicts **mpg** using a quadratic function of **horsepower** performs better than a model that involves only a linear function of **horsepower**, and there is no evidence of an improvement in using a cubic function of **horsepower**.

1.5 Cross-Validation

The simplest way to cross-validate in Python is to use `sklearn`, which has a different interface or API than`statsmodels`.

The `ISLP` package provides a wrapper, `sklearn_sm()`, that enables us to easily use the cross-validation tools of `sklearn` with models fit by ‘statsmodels

The class `sklearn_sm()` has as its first argument a model from `statsmodels`. It can take two additional optional arguments: `model_str` which can be used to specify a formula, and `model_args` which should be a dictionary of additional arguments used when fitting the model. For example, to fit a logistic regression model we have to specify a family argument. This is passed as `model_args` `{'family':sm.families.Binomial()}`.

```
[13]: hp_model = sklearn_sm(sm.OLS, MS(['horsepower']))
      X, Y = Auto.drop(columns =['mpg']), Auto['mpg']
      cv_results = cross_validate(hp_model, X, Y, cv=Auto.shape[0])
      cv_err = np.mean(cv_results['test_score'])
      cv_err
```

```
[13]: 24.23151351792922
```

```
[14]: printmd(f"The arguments to `cross_validate()` are as follows: an object with␣
       ↪the appropriate `fit()`, `predict()`, and `score()` methods, an array of␣
       ↪features X and a response Y. We also included an additional argument `cv` to␣
       ↪`cross_validate()`; specifying an integer K results in K-fold␣
       ↪cross-validation. We have provided a value corresponding to the total number␣
       ↪of observations, which results in leave-one-out cross-validation (LOOCV).␣
       ↪The `cross-validate()` function produces a dictionary with several␣
       ↪components; we simply want the cross-validated test score here (MSE), which␣
       ↪is estimated to be {cv_err:.2f}.");
```

The arguments to `cross_validate()` are as follows: an object with the appropriate `fit()`, `predict()`, and `score()` methods, an array of features X and a response Y. We also included an additional argument `cv` to `cross_validate()`; specifying an integer K results in K-fold cross-validation. We have provided a value corresponding to the total number of observations, which results in leave-one-out cross-validation (LOOCV). The `cross-validate()` function produces a dictionary with several components; we simply want the cross-validated test score here (MSE), which is estimated to be 24.23.

We can repeat this procedure for increasingly complex polynomial fits. To automate the process, we again use a for loop which iteratively fits polynomial regressions of degree 1 to 5, computes the associated crossvalidation error, and stores it in the i_{th} vector `cv_error`. The variable `d` in the for loop corresponds to the degree of the polynomial. We begin by initializing the vector. This command may take a couple of seconds to run.

```
[15]: cv_error = np.zeros (5)
      H = np.array(Auto['horsepower'])
      M = sklearn_sm(sm.OLS)
      for i, d in enumerate(range (1,6)):
        # this sets up the polynomial features for horsepower
        # such as 1, hp, hp**2, hp**3 till hp**5 in the final loop
        X = np.power.outer(H, np.arange(d+1))
        M_CV = cross_validate(M, X, Y, cv=Auto.shape[0])
        cv_error[i] = np.mean(M_CV['test_score'])
      cv_error
```

```
[15]: array([24.23151352, 19.24821312, 19.33498406, 19.42443029, 19.03320648])
```

We see a sharp drop in the estimated test MSE between the linear and quadratic fits, but then no clear improvement from using higher-degree polynomials.

Above we introduced the `outer()` method of the `np.power()` function. The `outer()` method is applied to an operation that has two arguments, such as `add()`, `min()`, or `power()`. It has two arrays as arguments, and then forms a larger array where the operation is applied to each pair of elements of the two arrays.

```
[16]: A = np.array ([3, 5, 9])
      B = np.array ([2, 4])
      np.add.outer(A, B)
```

[16]:

```
array([[ 5,  7],
       [ 7,  9],
       [11, 13]])
```

In the CV example above, we used K = n, but of course we can also use K < n. The code is very similar to the above (and is significantly faster). Here we use `KFold()` to partition the data into K = 10 random groups. We use `random_state` to set a random seed and initialize a vector `cv_error` in which we will store the CV errors corresponding to the polynomial fits of degrees one to five.

[17]:

```
cv_error = np.zeros(5)
# use same splits for each degree
cv = KFold(n_splits=10, shuffle=True, random_state=0)
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'])
cv_error
```

[17]:

```
array([24.20766449, 19.18533142, 19.27626666, 19.47848403, 19.13720065])
```

Notice that the computation time is much shorter than that of LOOCV. (In principle, the computation time for LOOCV for a least squares linear model should be faster than for K-fold CV, due to the availability of the formula

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

$$\text{where } h_i = \frac{1}{n} + \frac{(x_i - \bar{x})^2}{\sum_{l=1}^{n}(x_l - \bar{x})^2}$$

for LOOCV; however, the generic `cross_validate()` function does not make use of this formula.) We still see little evidence that using cubic or higher-degree polynomial terms leads to a lower test error than simply using a quadratic fit.

The `cross_validate()` function is flexible and can take different splitting mechanisms as an argument. For instance, one can use the `ShuffleSplit()` Shuffle function to implement the validation set approach just as easily as K-fold cross-validation.

[18]:

```
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);
results['test_score'][0]
```

[18]:

```
23.61661706966988
```

One can estimate the variability in the test error by running the following:

[19]:

```
validation = ShuffleSplit(n_splits=10, test_size=196, random_state=0)
results = cross_validate(hp_model, Auto.drop (['mpg'], axis=1), Auto['mpg'],␣
 ↪cv=validation)
results['test_score'].mean (), results['test_score'].std()
```

[19]:

```
(23.802232661034168, 1.4218450941091842)
```

This standard deviation is not a valid estimate of the sampling variability of the mean test score or the individual scores, since the randomly-selected training samples overlap and hence introduce correlations. But it does give an idea of the Monte Carlo variation incurred by picking different random folds.

1.6 The Bootstrap

1.6.1 Estimating the Accuracy of a Statistic of Interest

The bootstrap approach can be applied in almost all situations. No complicated mathematical calculations are needed.

To illustrate the bootstrap, we start with a simple example. The Portfolio data set in the ISLP package is described in Section 5.2 The Bootstrap. The goal is to estimate the sampling variance of the parameter α given in formula

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

We will create a function alpha_func(), which takes as input a dataframe D assumed to have columns Xand Y, as well as a vector idx indicating which observations should be used to estimate α. The function then outputs the estimate for α based on the selected observations.

[20]:

```
Portfolio = load_data('Portfolio')
print(Portfolio.head())
print(len(Portfolio))

def alpha_func(D, idx):
  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]))
```

```
          X         Y
0 -0.895251 -0.234924
1 -1.562454 -0.885176
2 -0.417090  0.271888
3  1.044356 -0.734198
4 -0.315568  0.841983
100
```

This function returns an estimate for α based on applying the minimum variance formula (5.7) to the observations indexed by the argument idx. For instance, the following command estimates α using all 100 observations.

[21]:

```
alpha_func(Portfolio , range(100))
```

[21]:

```
0.57583207459283
```

Next we randomly select 100 observations from `range(100)`, with replacement. This is equivalent to constructing a new bootstrap data set and recomputing $\hat{\alpha}$ based on the new data set.

[22]:
```
rng = np.random.default_rng(0)
alpha_func(Portfolio, rng.choice(100, 100, replace=True))
```

[22]:
```
0.6074452469619004
```

This process can be generalized to create a simple function `boot_SE()` for computing the bootstrap standard error for arbitrary functions that take only a data frame as an argument.

[23]:
```
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 i 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)
```

Let's use our function to evaluate the accuracy of our estimate of α using B = 1,000 bootstrap replications.

[24]:
```
alpha_SE = boot_SE(alpha_func, Portfolio, B=1000, seed=0)
alpha_SE
```

[24]:
```
0.09118176521277699
```

[25]:
```
printmd(f"The final output shows that the bootstrap estimate for SE(ˆ) is␣
 ↪{alpha_SE:.4f}.")
```

The final output shows that the bootstrap estimate for SE(ˆ) is 0.0912.

1.6.2 Estimating the Accuracy of a Linear Regression Model

The bootstrap approach can be used to assess the variability of the coefficient estimates and predictions from a statistical learning method. Here we use the bootstrap approach in order to assess the variability of the estimates for β_0 and β_1 ,the intercept and slope terms for the linear regression model that uses horsepower to predict *mpg* in the **Auto** data set. We will compare the estimates obtained using the bootstrap to those obtained using the formulas for $SE(\beta_0)$ and $SE(\beta_1)$ described in Section 3.1.2., Assessing the Accuracy of the Coefficients.

$$
\begin{aligned}
SE(\hat{\beta}_0)^2 &= \sigma^2\left[\frac{1}{n} + \frac{\bar{x}^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2}\right] \\
SE(\hat{\beta}_1)^2 &= \frac{\sigma^2}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \\
&\text{where } \sigma^2 = Var(\epsilon) \\
&\text{and } \hat{\sigma} = RSE = \sqrt{\frac{RSS}{n-2}}
\end{aligned}
$$

To use our `boot_SE()` function, we must write a function (its first argument) that takes a data frame `D` and indices `idx` as its only arguments. But here we want to bootstrap a specific regression model, specified by a model formula and data. We show how to do this in a few simple steps.

We start by writing a generic function `boot_OLS()` for bootstrapping a regression model that takes a formula to define the corresponding regression. We use the `clone()` function to make a copy of the formula that can be refit to the new dataframe. This means that any derived features such as those defined by `poly()` (which we will see shortly), will be re-fit on the resampled data frame.

[26]:
```python
def boot_OLS(model_matrix , response , D, idx):
  D_ = D.loc[idx]
  Y_ = D_[response]
  X_ = clone(model_matrix).fit_transform(D_)
  return sm.OLS(Y_ , X_).fit().params
```

This is not quite what is needed as the first argument to `boot_SE()`. The first two arguments which specify the model will not change in the bootstrap process, and we would like to *freeze* them. The function `partial()` from the `functools` module does precisely this: it takes a function as an argument, and freezes some of its arguments, starting from the left. We use it to freeze the first two model-formula arguments of `boot_OLS()`.

[27]:
```python
hp_func = partial(boot_OLS , MS(['horsepower']), 'mpg')
hp_func?
```

```
Signature:      hp_func(D, idx)
Call signature: hp_func(*args, **kwargs)
Type:           partial
String form:    functools.partial(<function boot_OLS at 0x7cabb5ba7920>,␣
 ↪ModelSpec(terms=['horsepower']), 'mpg')
File:           /usr/lib/python3.12/functools.py
Docstring:
partial(func, *args, **keywords) - new function with partial application
of the given arguments and keywords.
```

Typing `hp_func?` will show that it has two arguments `D` and `idx` — it is a version of `boot_OLS()` with the first two arguments frozen — and hence is ideal as the first argument for `boot_SE()`.

The `hp_func()` function can now be used in order to create bootstrap estimates for the intercept and slope terms by randomly sampling from among the observations with replacement. We first demonstrate its utility on 10 bootstrap samples.

```
[28]: rng = np.random.default_rng(0)
      print(Auto.head())
      # reset index so that index has row index and not model names
      Auto.reset_index(drop=False, inplace=True)
      print(Auto.head())
      np.array([hp_func(Auto, rng.choice(392, 392, replace=True)) for _ in range(10)])
```

```
                            mpg  cylinders  displacement  horsepower  weight  \
name
chevrolet chevelle malibu  18.0          8         307.0         130    3504
buick skylark 320          15.0          8         350.0         165    3693
plymouth satellite         18.0          8         318.0         150    3436
amc rebel sst              16.0          8         304.0         150    3433
ford torino                17.0          8         302.0         140    3449

                           acceleration  year  origin
name
chevrolet chevelle malibu          12.0    70       1
buick skylark 320                  11.5    70       1
plymouth satellite                 11.0    70       1
amc rebel sst                      12.0    70       1
ford torino                        10.5    70       1
                        name   mpg  cylinders  displacement  horsepower  \
0  chevrolet chevelle malibu  18.0          8         307.0         130
1          buick skylark 320  15.0          8         350.0         165
2         plymouth satellite  18.0          8         318.0         150
3              amc rebel sst  16.0          8         304.0         150
4                ford torino  17.0          8         302.0         140

   weight  acceleration  year  origin
0    3504          12.0    70       1
1    3693          11.5    70       1
2    3436          11.0    70       1
3    3433          12.0    70       1
4    3449          10.5    70       1
```

```
[28]: array([[39.88064456, -0.1567849 ],
             [38.73298691, -0.14699495],
             [38.31734657, -0.14442683],
             [39.91446826, -0.15782234],
             [39.43349349, -0.15072702],
             [40.36629857, -0.15912217],
             [39.62334517, -0.15449117],
             [39.0580588 , -0.14952908],
             [38.66688437, -0.14521037],
             [39.64280792, -0.15555698]])
```

Next, we use the `boot_SE()` function to compute the standard errors of 1,000 bootstrap estimates for the intercept and slope terms.

```
[29]: hp_se = boot_SE(hp_func, Auto, B=1000, seed=10)
      hp_se
```

```
[29]: intercept     0.848807
      horsepower    0.007352
      dtype: float64
```

```
[30]: printmd(f"This indicates that the bootstrap estimate for $SE(\\beta_0)$ is␣
        ↪{hp_se.intercept:.2f}, and that the bootstrap estimate for $SE(\\beta_1)$ is␣
        ↪{hp_se.horsepower:.4f}.")
```

This indicates that the bootstrap estimate for $SE(\beta_0)$ is 0.85, and that the bootstrap estimate for $SE(\beta_1)$ is 0.0074.

Standard formulas can be used to compute the standard errors for the regression coefficients in a linear model. These can be obtained using the `summarize()` function from `ISLP.sm`.

```
[31]: hp_model.fit(Auto , Auto['mpg'])
      model_se = summarize(hp_model.results_)['std err']
      model_se
```

```
[31]: intercept     0.717
      horsepower    0.006
      Name: std err, dtype: float64
```

```
[32]: printmd(f"The standard error estimates for $\\beta_0$ and $\\beta_1$ obtained␣
        ↪using the formulas are {model_se.intercept:.3f} for the intercept and␣
        ↪{model_se.horsepower:.3f} for the slope.")
```

The standard error estimates for β_0 and β_1 obtained using the formulas are 0.717 for the intercept and 0.006 for the slope.

Interestingly, these are somewhat different from the estimates obtained using the bootstrap. Does this indicate a problem with the bootstrap? In fact, it suggests the opposite.

The standard formulas rely on certain assumptions. For example, they depend on the unknown parameter σ^2 , the noise variance. We then estimate σ^2 using the RSS. Now although the formula for the standard errors do not rely on the linear model being correct, the estimate for σ^2 does. We see that there is a non-linear relationship in the data, and so the residuals from a linear fit will be inflated, and so will σ^2 . Secondly, the standard formulas assume (somewhat unrealistically) that the x_i are fixed, and all the variability comes from the variation in the errors ϵ_i . The bootstrap approach does not rely on any of these assumptions, and so it is likely giving a more accurate estimate of the standard errors of $\hat{\beta}_0$ and $\hat{\beta}_1$ than the results from `sm.OLS`.

Below we compute the bootstrap standard error estimates and the standard linear regression estimates that result from fitting the quadratic model to the data. Since this model provides a good fit to the data, there is now a better correspondence between the bootstrap estimates and the standard estimates of $SE(\hat{\beta}_0)$, $SE(\hat{\beta}_1)$ and $SE(\hat{\beta}_2)$.

```
[33]: quad_model = MS([poly('horsepower', 2, raw=True)])
      quad_func = partial(boot_OLS, quad_model, 'mpg')
      boot_SE(quad_func, Auto, B=1000)
```

```
[33]: intercept                                 2.067840
      poly(horsepower, degree=2, raw=True)[0]    0.033019
      poly(horsepower, degree=2, raw=True)[1]    0.000120
      dtype: float64
```

We compare the results to the standard errors computed using `sm.OLS()`.

```
[34]: M = sm.OLS(Auto['mpg'], quad_model.fit_transform(Auto))
      summarize(M.fit())['std err']
```

```
[34]: intercept                                 1.800
      poly(horsepower, degree=2, raw=True)[0]    0.031
      poly(horsepower, degree=2, raw=True)[1]    0.000
      Name: std err, dtype: float64
```

```
[35]: allDone();
```

```
<IPython.lib.display.Audio object>
```