lab5

December 4, 2023

0.1 Lab 5: Resampling Methods

Cross-Validation and the Bootstrap Jack Krebsbach Math 313

Imports

```
[ ]: 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
```

New Imports:

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

The Validation Set Approach We will split the data into two sets

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

Fit a linear regression only using observations in the training set.

```
[ ]: 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()
```

Now we can use the validation set and the train model to see a more accurate metric on how well the model can generalize. We can also calculate an estimate of the MSE of the model.

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

```
[ ]: 23.616617069669882
```

```
[ ]: # Define a function to calculate the MSE of a given model

     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)
```

Now, we can use the previously defined function to evaluate the MSE on the validation set using linear, quadratic, and cubic fits.

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

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

We can expect different MSE values if we use a different train/test split.

```
[ ]: 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
```

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

Overall, while the MSEs are different the findings are the same, the quadratic function of horsepower performs better than the model only using the linear function of horsepower.

Cross-Validation Theoretically we can use cross-validation for any generalized linear model.

The function `cross_validate` takes an object with fit, predict, and score methods, along with an array of features, the response, and how many folds. In this case we are putting in the number of observations so this becomes LOOCV.

```
[ ]: # We can use sklearn to
hp_model = sklearn_sm(sm.OLS, MS(['horsepower']))

# Create model matrix
X, Y = Auto.drop(columns=['mpg']), Auto['mpg']

cv_results = cross_validate(hp_model, X, Y, cv=Auto.shape[0])

# Calculate Error
cv_err = np.mean(cv_results['test_score'])
cv_err
```

```
[ ]: 24.231513517929205
```

```
[ ]: # Now, using cross validation we see what degree of polynomial fits work well␣
  ↪on the data.
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))
    M_CV = cross_validate(M, X, Y, cv=Auto.shape[0])
    cv_error[i] = np.mean(M_CV['test_score'])
cv_error
```

```
[ ]: array([24.23151352, 19.24821312, 19.33498406, 19.42443032, 19.03323815])
```

Here we see that there is a sharp drop in estimated MSE between linear and quadratic fits. However, there is minimal improvement from using higher-degree polynomials.

We can use `np.add.outer()` to add pairs of elements together from two vectors.

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

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

Notes on KFold: - Use K=10 to generate 10 random groups - random_state to set the seed

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

```
[ ]: array([24.20766449, 19.18533142, 19.27626666, 19.47848403, 19.13720287])
```

Here we see that the computational time is much less than LOOCV.

```
[ ]: 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']
```

```
[ ]: array([23.61661707])
```

We can estimate the variability in the test error by running the following. *Note that this is not a valid estimate of the sampling variability of the mean test score or the individual scores because the randomly-selected training samples overlap.

```
[ ]: 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()
```

```
[ ]: (23.802232661034168, 1.4218450941091891)
```

The Bootstrap Boostrap can be applied in almost all situations. We will use it to estimate the accuracy of the linear regression model on the `Auto` data set.

We will estimate the sample variance of the parameter alpha in the following:

```
[ ]: # Load the data
     Portfolio = load_data('Portfolio')

     # Define the function to estimate the variance of the parameter alpha
     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]))
```

```
[ ]: alpha_func(Portfolio, range(100))
```

```
[ ]: 0.57583207459283
```

In the following we randomly select 100 observations with replacement (essentially constructing a new boostrap

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

```
[ ]: 0.6074452469619004
```

Now, we can create a generalized simple function `boot_SE()` to compute the standard error for arbitrary functions.

```
[ ]: 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)
```

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

```
[ ]: 0.09118176521277699
```

Thus, the boostrap estimate for `SE(a)` is 0.0912.

Estimate the Accuracy of a Linear Model We will create a boostrap function that will estimate the accuracy of the linear regression model.

```
[ ]: 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
```

The following will freeze the arguments, now the first two arguments are frozen:

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

```
[ ]: # Check that there are only two arguments
     hp_func?
```

```
[ ]: rng = np.random.default_rng(0)
     np.array([hp_func(Auto,
     rng.choice(392, 392,
replace=True)) for _ in range(10)])
```

```
[ ]: 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]])
```

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

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

This shows that the boostrap estimate for the SE of B_0 is approximately 0.85 and the SE for B_1 (horsepower) is approximately 0.00753.

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

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

While we see that the estimates for the SE of the coefficients are different, it turns out the estimates from our bootstrapped models are more accurate. This is because the bootstrap does not rely on any assumptions such as assuming that all x_i are fixed.

```
[ ]: # Since this model is a good fit the error estimates are closer to what they␣
      ↪actually are.
     quad_model = MS([poly('horsepower', 2, raw=True)])
     quad_func = partial(boot_OLS, quad_model ,
     'mpg')
     boot_SE(quad_func, Auto, B=1000)
```

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

[ ]:
```python
# Estimates for the standard errors using sm.OLS()
M = sm.OLS(Auto['mpg'], quad_model.fit_transform(Auto))
summarize(M.fit())['std err']
```

[ ]:
```
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
```