Python Tutorial 5

March 9, 2020

This tutorial is for Prof. Xin Tong's DSO 530 class at the University of Southern California in spring 2020. It aims to provide supplementary python code for *Lecture 5: Cross-Validation and Bootstrap*.

In this lab, we'll explore the four resampling techniques discussed: the validation set approach, leave-one-out cross-validation (LOOCV), k -fold cross-validation, and the bootstrap. Note that some of the commands in this lab may take a while to run. While in the previous tutorials we used both StatsModels and scikit-learn, in this tutorial we will focus our attention on just scikit-learn, since StatsModels does not come with built-in classes for performing cross-validation.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression
from sklearn import metrics

import statsmodels.api as sm
import statsmodels.formula.api as smf
```

1 The Validation Set Approach

First, we'll explore using the validation set approach to estimate the test error rates that result from fitting various linear models with the *Auto* data set.

The *Auto* data set consists of 392 observations and 9 variables as follows:

mpg: miles per gallon; *cylinders*: Number of cylinders between 4 and 8; *displacement*: Engine displacement (cu. inches); *horsepower*: Engine horsepower; *weight*: Vehicle weight (lbs.); *acceleration*: Time to accelerate from 0 to 60 mph (sec.); *year*: Model year (modulo 100); *origin*: Origin of car (1. American, 2. European, 3. Japanese); *name*: Vehicle name.

The original data contained 408 observations but 16 observations with missing values were removed.

[2]:
```
Auto = pd.read_csv("auto.csv")
```

[3]:
```
Auto.head()
```

[3]:
```
    mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
0  18.0          8         307.0         130    3504          12.0    70
1  15.0          8         350.0         165    3693          11.5    70
2  18.0          8         318.0         150    3436          11.0    70
3  16.0          8         304.0         150    3433          12.0    70
4  17.0          8         302.0         140    3449          10.5    70

   origin                       name
0       1  chevrolet chevelle malibu
1       1          buick skylark 320
2       1         plymouth satellite
3       1              amc rebel sst
4       1                ford torino
```

Here, we will add the square of horsepower and the cube of horsepower to do the quadratic and cubic regression.

[4]:
```
Auto['horsepower_square'] = np.power(Auto['horsepower'], 2)
Auto['horsepower_cube'] = np.power(Auto['horsepower'], 3)

Auto.head()
```

[4]:
```
    mpg  cylinders  displacement  horsepower  weight  acceleration  year  \
0  18.0          8         307.0         130    3504          12.0    70
1  15.0          8         350.0         165    3693          11.5    70
2  18.0          8         318.0         150    3436          11.0    70
3  16.0          8         304.0         150    3433          12.0    70
4  17.0          8         302.0         140    3449          10.5    70

   origin                       name  horsepower_square  horsepower_cube
0       1  chevrolet chevelle malibu              16900          2197000
1       1          buick skylark 320              27225          4492125
2       1         plymouth satellite              22500          3375000
3       1              amc rebel sst              22500          3375000
4       1                ford torino              19600          2744000
```

To start with, we use the *train_test_split()* function from scikit-learn's *model_selection* module to split the set of observations into four sets: X and y validation sets, and X and y test sets. We set $test\_size = 0.5$ and $random\_state = 1$.

[5]:
```
from sklearn.model_selection import train_test_split

X_train, X_validation, y_train, y_validation =␣
↪train_test_split(Auto[["horsepower","horsepower_square",␣
↪"horsepower_cube"]], Auto["mpg"], test_size = 0.5, random_state = 1)
```

We then use *X_train* and *y_train* to fit a linear regression model using only the observations corresponding to those in the training set.

First, we fit a linear model.

[6]:
```
lr1 = LinearRegression()
lr1.fit(X_train["horsepower"].values.reshape(-1, 1), y_train)
```

[6]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

After fitting the linear regression of *mpg* onto *horsepower* using the training set, we use the *predict()* function to estimate the response for the observations in the validation set. Then, we use the *mean_squared_error()* function from scikit-learn's metrics module to compute the mean squared error of the 196 observations in the validation set.

[7]:
```
metrics.mean_squared_error(y_validation, lr1.predict(X_validation["horsepower"].
 ↪values.reshape(-1, 1)))
```

[7]:
```
24.80212062059356
```

As we can see, the estimated test mean squared error for the linear regression fit is 24.80. Now, we will estimate the test error for the quadratic and cubic regressions.

When we create a quadratic model we still use *LinearRegression()*.

[8]:
```
lr2 = LinearRegression()
lr2.fit(X_train[["horsepower", "horsepower_square"]].values, y_train)
metrics.mean_squared_error(y_validation, lr2.
 ↪predict(X_validation[["horsepower", "horsepower_square"]].values))
```

[8]:
```
18.848292603275652
```

For the cubic model, we implement the following code.

[9]:
```
lr3 = LinearRegression()
lr3.fit(X_train[["horsepower", "horsepower_square", "horsepower_cube"]].values,␣
 ↪y_train)
metrics.mean_squared_error(y_validation, lr3.
 ↪predict(X_validation[["horsepower", "horsepower_square", "horsepower_cube"]].
 ↪values))
```

[9]:
```
18.805111358604858
```

Using this training set/validation set split, we get validation set mean squared error values of 24.80, 18.84, and 18.80 for the linear, quadratic, and cubic regression models, respectively.

Then we can conclude that a quadratic model or a cubic model for predicting *mpg* using *horsepower* performs better than a linear model. However, there is little evidence that we should really use a cubic model over a quadratic model.

2 Leave-One-Out Cross-Validation

First, we import *cross_val_score*, *LeaveOneOut*.

[10]:
```
from sklearn.model_selection import cross_val_score, LeaveOneOut
```

Now that we have made the necessary imports, we'll perform leave-one-out cross-validation to estimate the test mean squared error.

Note that score objects follow the convention that higher return values are better than lower return values. In other words, two given two scores $s1 < s2$, then $s2$ is considered to be the better score. Thus, metrics that measure the distance between the model prediction and the data, such as *metrics.mean_squared_error* are called in *cross_val_score()* by the name *neg_mean_squared_error*, to return the negated value of the metric and follow this convention.

[11]:
```
# Using LeaveOneOut cross-validation splitter explicitly
X = Auto["horsepower"].values.reshape(-1, 1)
y = Auto["mpg"]
lr4 = LinearRegression()
loo = LeaveOneOut()
cv_score = abs(cross_val_score(lr4, X, y, scoring = "neg_mean_squared_error",␣
 ↪cv = loo))
# Since cv_scores is an array of scores, need to compute the mean afterward
cv_score.mean()
```

[11]:
```
24.231513517929226
```

We can repeat this procedure for increasingly complex polynomial fits. To automate the process, we use the *for* function to initiate a for loop which iteratively fits polynomial regressions for polynomials of order i =1 to i = 10, computes the associated cross-validation error, and stores it in the ith element of the vector *cv.error*.

[12]:
```
X = Auto["horsepower"].to_frame()
# Pandas Series.to_frame() function is used to convert the given series object␣
 ↪to a dataframe

cv_scores = []
cv_scores.append(abs(cross_val_score(lr4, X.iloc[:,0:1], y, scoring =␣
 ↪"neg_mean_squared_error", cv = loo)).mean())
# we compute for the first order polynomial separately

I = 10
for i in range(2,I+1):
    X["horserpwer^{}".format(i)] = np.power(Auto["horsepower"],i)
    cv_scores.append(abs(cross_val_score(lr4, X.iloc[:,0:i], y, scoring =␣
 ↪"neg_mean_squared_error", cv = loo)).mean())
```

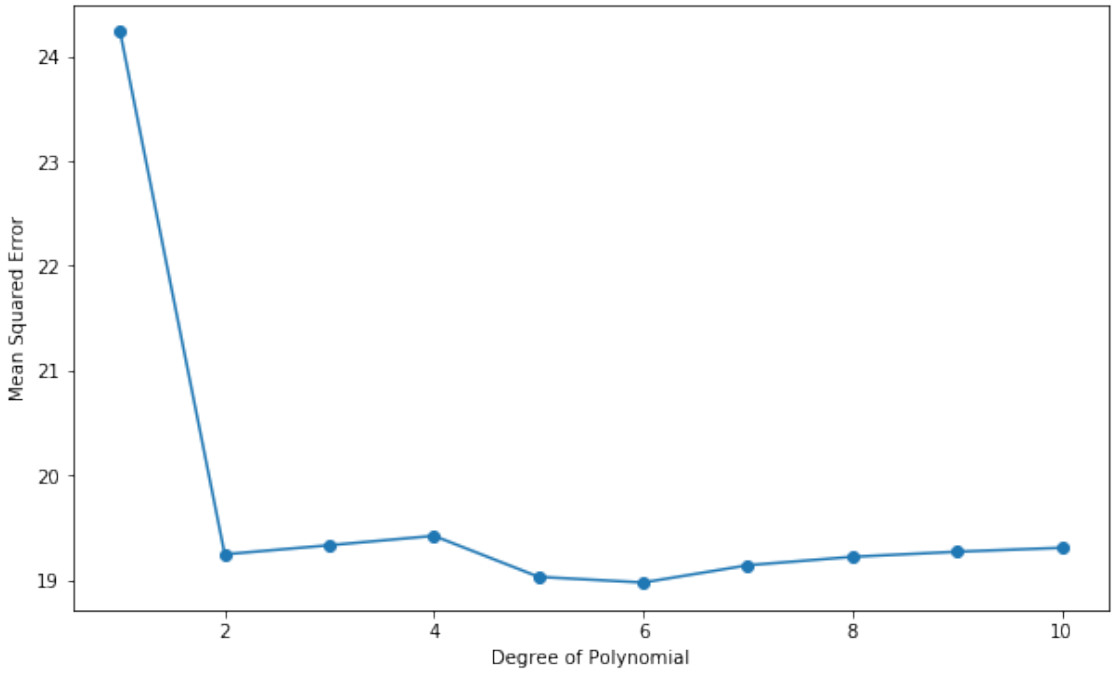
[13]:
```
cv_scores
```

[13]:
```
[24.231513517929226,
 19.24821312448967,
 19.334984064029452,
```

```
 19.4244303103955,
 19.033212804000605,
 18.979170142296876,
 19.14421723872924,
 19.22429533582231,
 19.27309007242442,
 19.310585617597386]
```

[14]:
```
fig, ax = plt.subplots(1,1, figsize=(10,6))
x = range(1,11)
ax.plot(x,cv_scores, '-o')
ax.set_xlabel('Degree of Polynomial')
ax.set_ylabel('Mean Squared Error')
```

[14]:
```
Text(0, 0.5, 'Mean Squared Error')
```



As we can see, there is a sharp drop in the estimated test mean squared error between the linear and quadratic fits, but then no clear improvement from using higher-order polynomials.

3 k -Fold Cross-Validation

[15]:
```
from sklearn.model_selection import KFold
```

We can also use the *cross_val_score()* function to perform k-fold cross validation by supplying a value k to the function call. We'll use k=*10*, a common choice for k, on the *Auto* data set and again

use a *for* loop to iteratively compute the k-fold cross validation errors corresponding to polynomial fits of order i=1,...,10. Note that while k-fold cross validation involves random sampling, it will return consistent results for a given integer value of cv, since passing an integer k for *cv* tells *cross_val_score()* to use *KFold(n_splits = k)* as the cross-validation splitter. The constructor for *KFold* can take an optional argument *shuffle*, which is *False* by default, to specify whether or not the data should be shuffled before computing the folds. If the data is not shuffled, then *KFold* will split it into consistent folds every time.

[16]:
```
# Using 10-fold cross-validation by passing the argument cv = 10 to␣
↪cross_val_score()
X = Auto["horsepower"].values.reshape(-1, 1)
y = Auto["mpg"]
lr5 = LinearRegression()
cv_score = abs(cross_val_score(lr5, X, y, scoring = "neg_mean_squared_error",␣
↪cv = 10))
cv_score.mean()
```

[16]:
```
27.439933652339867
```

We can also repeat this procedure for increasingly complex polynomial fits.

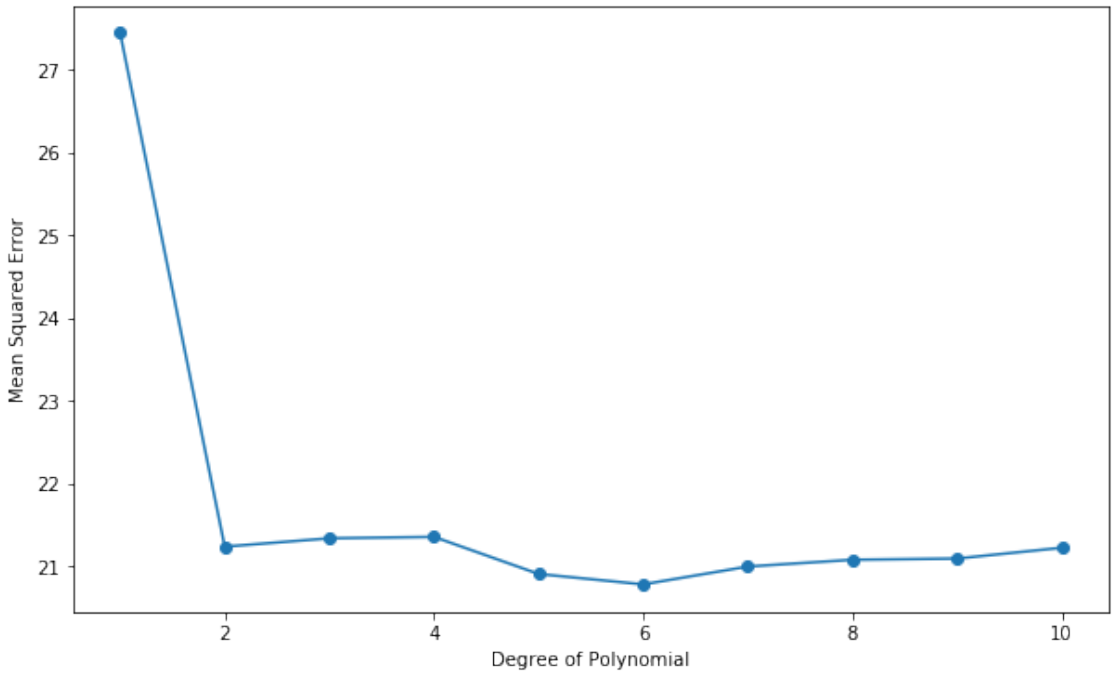
[17]:
```
X = Auto["horsepower"].to_frame()
# Pandas Series.to_frame() function is used to convert the given series object␣
↪to a dataframe

cv_scores = []
cv_scores.append(abs(cross_val_score(lr5, X.iloc[:,0:1], y, scoring =␣
↪"neg_mean_squared_error", cv = 10)).mean())

I = 10
for i in range(2,I+1):
    X["horserpwer^{}".format(i)] = np.power(Auto["horsepower"],i)
    cv_scores.append(abs(cross_val_score(lr5, X.iloc[:,0:i], y, scoring =␣
↪"neg_mean_squared_error", cv = 10)).mean())
```

[18]:
```
cv_scores
```

[18]:
```
[27.439933652339867,
 21.235840055802235,
 21.336606183228007,
 21.353886981505603,
 20.905640766798655,
 20.780115397193125,
 20.995603602567215,
 21.07740048210734,
 21.092260855698022,
 21.22248741379395]
```

[19]:
```
fig, ax = plt.subplots(1,1, figsize=(10,6))
x = range(1,11)
ax.plot(x,cv_scores, '-o')
ax.set_xlabel('Degree of Polynomial')
ax.set_ylabel('Mean Squared Error')
```

[19]:
```
Text(0, 0.5, 'Mean Squared Error')
```



If we do choose to have *KFold* shuffle the data before splitting it into batches, then we should set a random seed to ensure consistent and reproducible results. The two main ways of doing this are using *np.random.seed()* or passing a value to the *random_state* argument in the constructor.

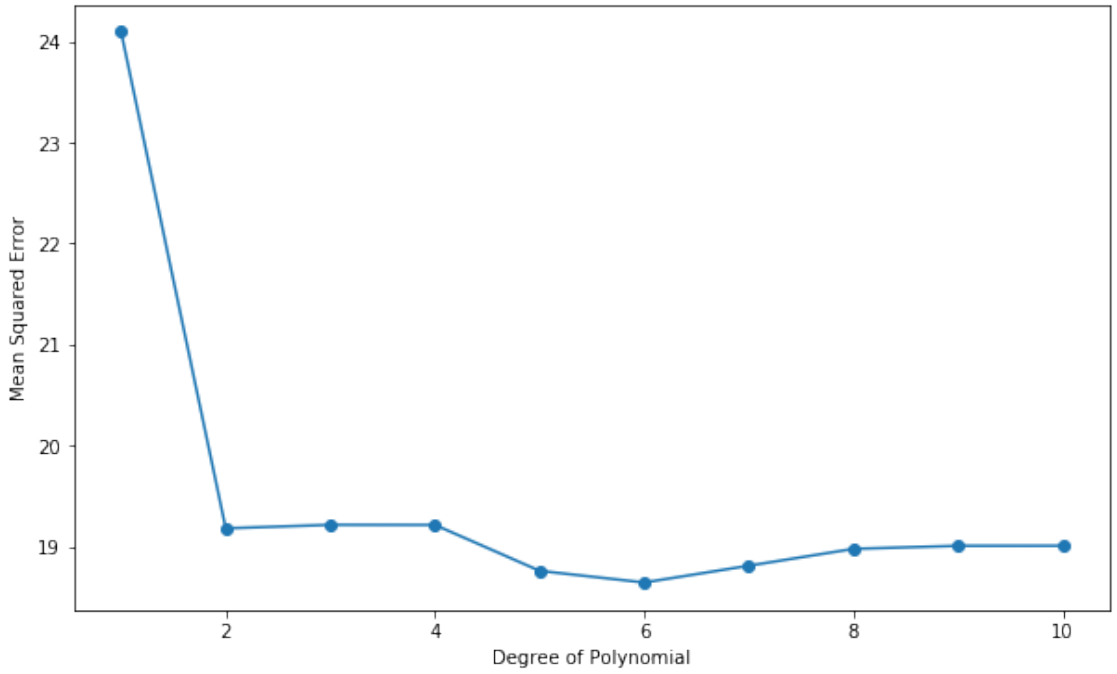
[20]:
```
# Using 10-fold cross-validation by passing an instance of KFold with shuffle =␣
 ↪True
# In this situation the value of random_state matters

kfolds = KFold(n_splits = 10, shuffle = True, random_state = 1) # Here, we set␣
 ↪random_state = 1
cv_scores = []
cv_scores.append(abs(cross_val_score(lr5, X.iloc[:,0:1], y, scoring =␣
 ↪"neg_mean_squared_error", cv = kfolds)).mean())

I = 10

for i in range(2,I+1):
```

```
    X["horserpwer^{}".format(i)] = np.power(Auto["horsepower"],i)
    cv_scores.append(abs(cross_val_score(lr5, X.iloc[:,0:i], y, scoring =␣
↪"neg_mean_squared_error", cv = kfolds)).mean())

cv_scores
```

[20]:
```
[24.097675731883058,
 19.17888986488955,
 19.21385952370887,
 19.212807016427494,
 18.75799176795171,
 18.642292672372356,
 18.80915380822625,
 18.975779235294077,
 19.006234677683487,
 19.00687398212181]
```

[21]:
```
fig, ax = plt.subplots(1,1, figsize=(10,6))
x = range(1,11)
ax.plot(x,cv_scores, '-o')
ax.set_xlabel('Degree of Polynomial')
ax.set_ylabel('Mean Squared Error')
```

[21]:
```
Text(0, 0.5, 'Mean Squared Error')
```



Notice that the computation time is much shorter than that of LOOCV.

We still see little evidence that using cubic or higher-order polynomial terms leads to lower test error than simply using a quadratic fit.

4 The bootstrap

One of the strengths of the bootstrap approach is that it is very widely applicable and does not require complicated mathematical complications; in Python, we only need to take two main steps to perform a bootstrap analysis. The first is creating a function that computes the statistic of interest. Second, we use the *resample()* function from the *sklearn.utils* module to repeatedly sample observations from the data with replacement and compute the bootstrapped statistics of interest. We start by importing the *resample()* function.

resample() has the following crucial parameters:

replace: *boolean*, *True* by default. Implements resampling with replacement. If False, this will implement random permutations.

n_samples: *int*, *None* by default. The sample size of the new sample we create. If left to *None* this is automatically set to the first dimension of the arrays. If *replace* parameter is *False*, it should not be larger than the length of arrays.

random_state: *int*, *RandomState instance* or *None*, *optional* (default=*None*). The seed of the pseudo random number generator to use when shuffling the data. If *int*, *random_state* is the seed used by the random number generator; If *RandomState instance*, random_state is the random number generator; If *None*, the random number generator is the RandomState instance used by *np.random*.

[22]:
```
from sklearn.utils import resample
```

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 of ISLR.

We first create a simple function, *fit_coefs()*, which takes in an array of training X values, an array of training y values, and a scikit-learn estimator class object (e.g. a LinearRegression object for this lab). It then returns a NumPy array consisting of the regression coefficients, with the regression intercept as the last entry.

[23]:
```
def fit_coefs(X, y, estimator):
    reg = estimator.fit(X, y)
    coefs = reg.coef_
    intercept = reg.intercept_
    return np.append(coefs, intercept)
```

First, we'll demonstrate applying this function to the full set of 392 observations in order to compute the estimates of β_0 and β_1 on the entire data set using the usual linear regression coefficient estimate

formulas from Chapter 3 of ISLR.

[24]:
```
X = Auto["horsepower"].values.reshape(-1, 1)
y = Auto["mpg"]
lr6 = LinearRegression()
pd.Series(fit_coefs(X, y, lr6), index = ["horsepower", "intercept"])
```

[24]:
```
horsepower    -0.157845
intercept     39.935861
dtype: float64
```

[25]:
```
sample = resample(Auto, random_state=1)
X = sample["horsepower"].values.reshape(-1, 1)
y = sample["mpg"]
lr7 = LinearRegression()
pd.Series(fit_coefs(X, y, lr7), index = ["horsepower", "intercept"])
```

[25]:
```
horsepower    -0.155898
intercept     39.658479
dtype: float64
```

[26]:
```
sample.shape
```

[26]:
```
(392, 11)
```

[27]:
```
sample = resample(Auto, n_samples = 500, random_state=1)
X = sample["horsepower"].values.reshape(-1, 1)
y = sample["mpg"]
lr8 = LinearRegression()
pd.Series(fit_coefs(X, y, lr8), index = ["horsepower", "intercept"])
```

[27]:
```
horsepower    -0.158142
intercept     39.884421
dtype: float64
```

[28]:
```
sample.shape
```

[28]:
```
(500, 11)
```

Now we use a *for* loop to compute the standard errors of 1,000 bootstrap estimates for the intercept and slope terms.

[29]:
```
lr9 = LinearRegression()
bootstrap_estimates = pd.DataFrame()
for i in range(1000):
    sample = resample(Auto, random_state = i)
    X = sample["horsepower"].values.reshape(-1, 1)
    y = sample["mpg"]
```

```
    coefs = pd.Series(fit_coefs(X, y, lr9), index = ["horsepower",␣
↪"intercept"], name = i)
    bootstrap_estimates = bootstrap_estimates.join(coefs, how = "right")
    # join the new coefs results on the right of bootstrap_estimates (used to␣
↪stores results)

bootstrap_estimates
```

[29]:

	0	1	2	3	4	5	\
horsepower	-0.161562	-0.155898	-0.151630	-0.156031	-0.155020	-0.147481	
intercept	40.480439	39.658479	39.101022	39.792939	39.424444	38.927362	

	6	7	8	9	…	990	\
horsepower	-0.159993	-0.155535	-0.150306	-0.163312	…	-0.142494	
intercept	39.835459	39.291568	39.026775	40.806173	…	37.903170	

	991	992	993	994	995	996	\
horsepower	-0.164200	-0.149787	-0.158422	-0.145411	-0.156443	-0.162847	
intercept	40.665301	38.833686	40.050523	38.738008	39.751692	40.419171	

	997	998	999
horsepower	-0.143950	-0.148367	-0.163341
intercept	37.922457	38.626481	40.908235

[2 rows x 1000 columns]

[30]:
```
# print the mean and standard error of each row of results
pd.DataFrame({"original": bootstrap_estimates.mean(axis = 1), "std. error":␣
↪bootstrap_estimates.std(axis = 1)})
```

[30]:

	original	std. error
horsepower	-0.158133	0.007416
intercept	39.952723	0.862084

We got bootstrap estimates of 0.86 for $SE(\beta_0)$ and 0.0074 for $SE(\beta_1)$. Now we'll compare them with the estimates computed using the formulas described in Section 3.1.2 of *ISLR*, which can be obtained using the *summary()* or *bse()* functions in StatsModels.

[31]:
```
lr_smf = smf.ols('mpg ~ horsepower', data=Auto).fit()
print(lr_smf.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      mpg   R-squared:                       0.606
Model:                              OLS   Adj. R-squared:                  0.605
Method:                   Least Squares   F-statistic:                     599.7
Date:                  Mon, 09 Mar 2020   Prob (F-statistic):           7.03e-81
Time:                          22:50:45   Log-Likelihood:                -1178.7
```

```
No. Observations:                 392   AIC:                             2361.
Df Residuals:                     390   BIC:                             2369.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     39.9359      0.717     55.660      0.000      38.525      41.347
horsepower    -0.1578      0.006    -24.489      0.000      -0.171      -0.145
==============================================================================
Omnibus:                       16.432   Durbin-Watson:                   0.920
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               17.305
Skew:                           0.492   Prob(JB):                     0.000175
Kurtosis:                       3.299   Cond. No.                         322.
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

[32]:
```
lr_smf.bse
```

[32]:
```
Intercept     0.717499
horsepower    0.006446
dtype: float64
```

Using the formulas from Section 3.1.2 of *ISLR*, the standard error estimates are 0.717 for the intercept and 0.0064 for the slope, which are somewhat different from the bootstrap estimates.

Does this indicate a problem with the bootstrap? In fact, it suggests the opposite. The standard formulas given in Equation 3.8 on page 66 rely on certain assumptions. For example, they depend on the unknown parameter σ^2, variance of the random error term. We then estimate σ^2 using the RSE. Now although the formula for the standard errors does not rely on the linear model being correct, the estimate for σ^2 does. We see in Figure 3.8 on page 91 of *ISLR* that there is a non-linear relationship in the data, and so the residuals from a linear fit will be inflated, and so will $\widehat{\sigma}^2$. 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 doesn't rely on any of these assumptions, so it is likely that the bootstrap estimates are more accurate estimates of the standard errors of $\widehat{\beta}_0$ and $\widehat{\beta}_0$ than those from the *summary()* or *bse()* functions in StatsModels.

References:

James, G. , Witten, D. , Hastie, T. , & Tibshirani, R. . (2013). An Introduction to Statistical Learning: With Applications in R.

Müller, Andreas C; Guido, Sarah. (2017). Introduction to Machine Learning with Python.

https://www.kaggle.com/suugaku/islr-lab-4-python

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.LeaveOneOut.html

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

https://scikit-learn.org/stable/modules/generated/sklearn.utils.resample.html

https://matplotlib.org/tutorials/introductory/pyplot.html