Isha Jain

Section 5.4

Conceptual

2. Bootstrap is obtained from a set of n observations

(a) $P\begin{pmatrix}\text{Selecting } j^{th} \text{ observation from} \\ \text{sample in the first bootstrap selection}\end{pmatrix} = \frac{1}{n}$

$\therefore P\begin{pmatrix}\text{Not selecting } j^{th} \text{ observation from} \\ \text{sample in the first bootstrap selection}\end{pmatrix} = 1 - \frac{1}{n} = \frac{n-1}{n}$

(b) There is no change in the sample where the ~~bootstrap~~ is obtained since the sampling occurs with replacement

$\therefore P\begin{pmatrix}\text{Selecting the } j^{th} \text{ observation from} \\ \text{sample in the } 2^{nd} \text{ bootstrap sample}\end{pmatrix} = \frac{1}{n}$

$\therefore P\begin{pmatrix}\text{Not selecting the } j^{th} \text{ observation from} \\ \text{sample in the } 2^{nd} \text{ bootstrap sample}\end{pmatrix} = 1 - \frac{1}{n} = \frac{n-1}{n}$

(c) ~~Fro~~ On extrapolating the above

The probability of not selecting the j^{th} observation in each bootstrap sample is: $1 - \frac{1}{n}$

g. Assuming n draws for the bootstrap

$P(A \cap B) = P(A) \times P(B/A)$

Event: A : n^{th} draw does not contain the j^{th} observation

B : The previous $(n-1)$ draws did not contain j^{th} observation

$\therefore$ Required probability $= \left(1 - \frac{1}{n}\right) P(B/A)$

Ily we can infer $P(B/A) = \left(1 - \frac{1}{n}\right)^{n-1}$

$\therefore$ Required probability $= \left(1 - \frac{1}{n}\right)^{n}$

Page No.
Date

(d) Using the result for (c) and substituting $n=5$
(Assuming 5 samples in bootstrap)
~~Requ~~ Required probability $= 1 - \left(1-\frac{1}{5}\right)^5$

∴ required probability $= 1 - \left(\frac{4}{5}\right)^5 = 0.672$

(e) For $n = 100$:
Required probability $= 1-\left(1-\frac{1}{100}\right)^{100} = 1-\left(\frac{99}{100}\right)^{100}$

(Assuming 100 bootstrap samples)
∴ Required probability $= 0.634$

(f) For $n = 10{,}000$
Required probability $= 1-\left(1-\frac{1}{10000}\right)^{10000} = 1-\left(\frac{9999}{10000}\right)^{1000}$

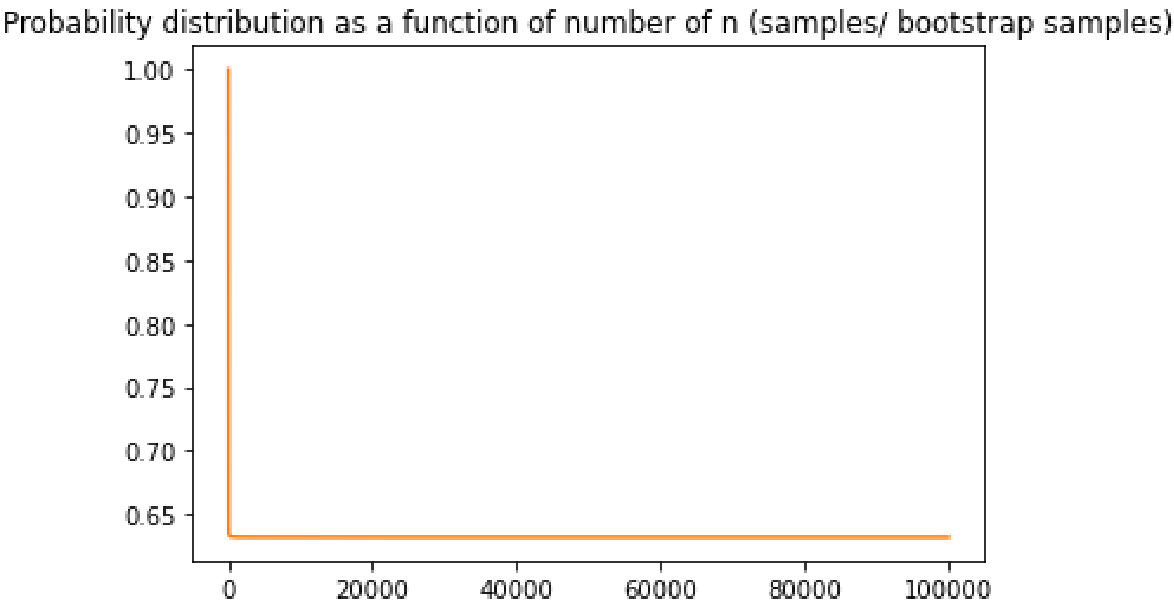
∴ Required probability $= 0.632$.

(g)

Section 5.4

2 (g)

In [7]:
```
import matplotlib.pyplot as plt
import numpy as np

# generating x (integers in the given range)
x=np.arange(1, 100001)


# corresponding probability derived in 2(c)
y = 1 - (1 - 1/x)**x


# plot the data
plt.plot(x, y, color ='tab:orange')

plt.title('Probability distribution as a function of number of n (samples/ bootstrap samples)')

# display the plot
plt.show()
```



From the above plot, we can observe that the probability of the j^{th} observation being in the bootstrap sample (where n is the number of samples and bootstrap draws), falls sharply as n increases and is almost constant between 0.7 and 0.6 as n crosses 3.

(h)

In [4]:
```
rng = np.random.default_rng(10)
store = np.empty(10000)
for i in range(10000):
    store[i] = np.sum(rng.choice(100, 100, replace=True) == 4)>0

np.mean(store)
```
Out[4]: 0.6362

The above corresponds to simulating subpart (e)of this question where we calculate the proba bility of the j^{th} sample (4^{th} in this case- value of j is not consequential to the result) being in in the bootstrap sample of size 100. We can see that the simulated results (giving a probability o f 0.6362 corresponding to the 4^{th} observation being considered in the bootstrap) is quite close to that predicted in part(e) (0.6340 rounded to 2 decimal places) corresponding to the probabi lity of the j^{th} sample being in in the bootstrap sample.

Applied

5.

5

In [8]:
```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
import statsmodels.api as sm
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
                         summarize)
```

In [9]:
```
from ISLP import confusion_table
from ISLP.models import contrast
from sklearn.discriminant_analysis import \
     (LinearDiscriminantAnalysis as LDA,
      QuadraticDiscriminantAnalysis as QDA)
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

In [11]:
```
Default = load_data('Default')
Default.head(5)
```

Out[11]:

	default	student	balance	income
0	No	No	729.526495	44361.625074
1	No	Yes	817.180407	12106.134700
2	No	No	1073.549164	31767.138947
3	No	No	529.250605	35704.493935
4	No	No	785.655883	38463.495879

(a)

(a)

In [12]:
```
#Feature engineering to convert the label to discrete numerical values with defa
Default['default'] = np.where((Default.default =='Yes'), 1, Default.default)
Default['default'] = np.where((Default.default =='No'), 0, Default.default)
print(Default)
```

```
      default student      balance        income
0           0      No   729.526495  44361.625074
1           0     Yes   817.180407  12106.134700
2           0      No  1073.549164  31767.138947
3           0      No   529.250605  35704.493935
4           0      No   785.655883  38463.495879
...       ...     ...          ...           ...
9995        0      No   711.555020  52992.378914
9996        0      No   757.962918  19660.721768
9997        0      No   845.411989  58636.156984
9998        0      No  1569.009053  36669.112365
9999        0     Yes   200.922183  16862.952321

[10000 rows x 4 columns]
```

In [13]:
```
from sklearn import preprocessing
#splitting dataset into features and labels

X = Default[['income', 'balance']]
y = Default['default']
lab_enc = preprocessing.LabelEncoder()
encoded_Y = lab_enc.fit_transform(y)
print(encoded_Y)
```

```
[0 0 0 ... 0 0 0]
```

In [14]:
```
#splitting dataset into features and labels

model = LogisticRegression(random_state=0)
model.fit(X, encoded_Y)
```

Out[14]:
```
LogisticRegression(random_state=0)
```

(b)

(b)

i.

In [15]:
```
#transforming 'default' column to have discrete numeric values
#Default['default'] = np.where(Default['default'] =='Yes', 1,0)
Default['default'] = np.where((Default.default =='Yes'), 1, Default.default)
Default['default'] = np.where((Default.default =='No'), 0, Default.default)
print(Default)
```

```
     default student      balance        income
0          0      No   729.526495  44361.625074
1          0     Yes   817.180407  12106.134700
2          0      No  1073.549164  31767.138947
3          0      No   529.250605  35704.493935
4          0      No   785.655883  38463.495879
...      ...     ...          ...           ...
9995       0      No   711.555020  52992.378914
9996       0      No   757.962918  19660.721768
9997       0      No   845.411989  58636.156984
9998       0      No  1569.009053  36669.112365
9999       0     Yes   200.922183  16862.952321

[10000 rows x 4 columns]
```

In [16]:
```
#creating a validation set with 30 percent of the dataset
from sklearn.model_selection import train_test_split

# split the dataset
X_train, X_validation, y_train, y_validation = train_test_split(
    X, encoded_Y, test_size=0.3, random_state=42)
```

In [17]:
```
X_train
```

Out[17]:

	income	balance
9069	41239.020510	0.000000
2603	37073.192381	961.999353
7738	19039.168273	655.611221
1579	27690.113535	864.047198
5058	57561.411261	1306.832034

(ii)

In [46]:
```
#Logistic regression model using training samples

model = LogisticRegression(random_state=0)
model.fit(X_train, y_train)
```

```
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:460: ConvergenceWarning: lbfgs failed to converge
(status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[46]:
```
LogisticRegression
LogisticRegression(random_state=0)
```

(iii)

In [20]:
```
#Obtain a prediction of default status
decision_threshold=0.5
y_pred = np.where(model.predict_proba(X_validation)[:,1] > decision_threshold, 1, 0)
print(y_pred)
#y_pred = model.predict(X_validation)
```

```
[0 0 0 ... 0 0 0]
```

(iv)

In [21]:
```
# Computing the validation set error
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score

cm = confusion_matrix(y_validation, y_pred)
print ("Confusion Matrix : \n", cm)
print ("Accuracy : ", accuracy_score(y_validation, y_pred))
```

In [22]:
```
#evaluating model after standardization to observe effect of standardization
from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
xtrainss = ss.fit_transform(X_train)
xvalidationss = ss.transform(X_validation)

model2 = LogisticRegression(random_state=0)
model2.fit(xtrainss, y_train)
decision_threshold=0.5
y_pred = np.where(model2.predict_proba(xvalidationss)[:,1] > decision_threshold, 1, 0)
cm = confusion_matrix(y_validation, y_pred)
print ("Confusion Matrix : \n", cm)
print ("Accuracy : ", accuracy_score(y_validation, y_pred))
```

```
Confusion Matrix :
 [[2896   10]
 [  70   24]]
Accuracy :  0.9733333333333334
```

Test error = [100*(1-0.9733)]%=2.67%

(c)

(c)

In [23]:
```
# splitting the dataset with different parameters (changing random state and fraction of data in validation set)
X_train, X_validation, y_train, y_validation = train_test_split(
    X, encoded_Y, test_size=0.2, random_state=2)

ss = StandardScaler()
xtrainss = ss.fit_transform(X_train)
xvalidationss = ss.transform(X_validation)

model3 = LogisticRegression(random_state=0)
model3.fit(xtrainss, y_train)
decision_threshold=0.5
y_pred = np.where(model2.predict_proba(xvalidationss)[:,1] > decision_threshold, 1, 0)
cm = confusion_matrix(y_validation, y_pred)
print ("Confusion Matrix : \n", cm)
print ("Accuracy : ", accuracy_score(y_validation, y_pred))
```

```
Confusion Matrix :
 [[1928   15]
 [  42   15]]
Accuracy :  0.9715
```

In [24]:
```
X_train, X_validation, y_train, y_validation = train_test_split(
    X, encoded_Y, test_size=0.25, random_state=23)

ss = StandardScaler()
xtrainss = ss.fit_transform(X_train)
xvalidationss = ss.transform(X_validation)

model4= LogisticRegression(random_state=0)
model4.fit(xtrainss, y_train)
decision_threshold=0.5
y_pred = np.where(model2.predict_proba(xvalidationss)[:,1] > decision_threshold, 1, 0)
cm = confusion_matrix(y_validation, y_pred)
print ("Confusion Matrix : \n", cm)
print ("Accuracy : ", accuracy_score(y_validation, y_pred))
```

```
Confusion Matrix :
 [[2404   12]
 [  57   27]]
Accuracy :  0.9724
```

In [25]:
```
X_train, X_validation, y_train, y_validation = train_test_split(
    X, encoded_Y, test_size=0.35, random_state=38)

ss = StandardScaler()
xtrainss = ss.fit_transform(X_train)
xvalidationss = ss.transform(X_validation)

model5 = LogisticRegression(random_state=0)
model5.fit(xtrainss, y_train)
decision_threshold=0.5
y_pred = np.where(model2.predict_proba(xvalidationss)[:,1] > decision_threshold, 1, 0)
cm = confusion_matrix(y_validation, y_pred)
print ("Confusion Matrix : \n", cm)
print ("Accuracy : ", accuracy_score(y_validation, y_pred))
```

```
Confusion Matrix :
 [[3363   19]
 [  79   39]]
Accuracy :  0.972
```

We have experimented by varying the size of validation and training set along with different random states, i.e., different splits of data. We obtain the following test error rate:

1^{st} model- 100*(1-0.9715) %=2.85%

2^{nd} model- 100*(1-0.9724) % =2.76%

3^{rd} model- 100*(1-0.972) % =2.8%

Due to the different splits, we observe the test error changes slightly, however the change is relatively small suggesting the model did not just get ‘lucky’ on the original validation set and the error on applying the model to ‘real world data’ would be similar assuming the ‘real world data’ to come from the same distribution as the train and validation data.

(d)

(d)

```
In [40]: lab_enc = preprocessing.LabelEncoder()
         encoded_student = lab_enc.fit_transform(Default['student'])
         X['student']=encoded_student
         print(X)

                    income      balance  student
         0     44361.625074   729.526495        0
         1     12106.134700   817.180407        1
         2     31767.138947  1073.549164        0
         3     35704.493935   529.250605        0
         4     38463.495879   785.655883        0
         ...            ...          ...      ...
         9995  52992.378914   711.555020        0
         9996  19660.721768   757.962918        0
         9997  58636.156984   845.411989        0
         9998  36669.112365  1569.009053        0
         9999  16862.952321   200.922183        1

         [10000 rows x 3 columns]

         C:\Users\ishaj\AppData\Local\Temp/ipykernel_44624/1955496617.py:3: SettingWithCopyWarning:
         A value is trying to be set on a copy of a slice from a DataFrame.
         Try using .loc[row_indexer,col_indexer] = value instead

         See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
         rsus-a-copy
           X['student']=encoded_student
```

```
In [42]: X_train, X_validation, y_train, y_validation = train_test_split(
             X, encoded_Y, test_size=0.2, random_state=2)

         ss = StandardScaler()
         xtrainss = ss.fit_transform(X_train)
         xvalidationss = ss.transform(X_validation)

         model6 = LogisticRegression(random_state=0)
         model6.fit(xtrainss, y_train)
         decision_threshold=0.5
         y_pred = np.where(model6.predict_proba(xvalidationss)[:,1] > decision_threshold, 1, 0)
         cm = confusion_matrix(y_validation, y_pred)
         print ("Confusion Matrix : \n", cm)
         print ("Accuracy : ", accuracy_score(y_validation, y_pred))
```

```
Confusion Matrix :
 [[1928   15]
 [  42   15]]
Accuracy :  0.9715
```

The test error on including ‘student’ along with ‘income’ and ‘balance’ to predict ‘default’ using the validation set approach is 100*(1-0.9715) % = 2.85 % which is similar to that on using only ‘income’ and ‘balance’ to predict ‘default’ i.e., including ‘student’ as a model predictor does not lead to a significant reduction in the test error rate.

6.

6

(a)

In [28]:
```
model = MS(['income', 'balance']).fit(Default)
X = model.transform(Default)

ss = StandardScaler()
xss = ss.fit_transform(X)

y=Default['default']
glm_train = sm.GLM(y.astype(float),
                   xss.astype(float),
                   family=sm.families.Binomial())
results = glm_train.fit()
print(results.summary())
```

```
                 Generalized Linear Model Regression Results
==============================================================================
Dep. Variable:                default   No. Observations:                10000
Model:                            GLM   Df Residuals:                     9998
Model Family:                Binomial   Df Model:                            1
Link Function:                  Logit   Scale:                          1.0000
Method:                          IRLS   Log-Likelihood:                -6851.2
Date:                Wed, 25 Oct 2023   Deviance:                       13702.
Time:                        04:54:57   Pearson chi2:                 1.00e+04
No. Iterations:                     4   Pseudo R-squ. (CS):             -1.939
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const               0          0        nan        nan           0           0
x1             0.0249      0.020      1.223      0.221      -0.015       0.065
x2             0.2587      0.021     12.514      0.000       0.218       0.299
==============================================================================
```

(b)

(b)

In [32]:
```
print(model)
```
```
ModelSpec(terms=['income', 'balance'])
```

In [33]:
```
import functools
def boot_fn(Default,idx):
    model_matrix=model
    response='default'
    D_ = Default.loc[idx]
    Y_ = Default[response]
    X_ = model_matrix.fit_transform(Default)
    return sm.OLS(Y_.astype(float), X_.astype(float)).fit().params
#df_func = functools.partial(boot_fn, MS(['income','balance']), 'default')
```

In [34]:
```
rng = np.random.default_rng(0)
np.array([boot_fn(Default,
rng.choice(392,
392,
replace=True)) for _ in range(1)])
```
Out[34]:
```
array([[-9.22396837e-02,  4.60456800e-07,  1.31804970e-04]])
```

In [35]:
```
rng = np.random.default_rng(0)
np.array([boot_fn(Default,
rng.choice(8,
84,
replace=True)) for _ in range(1)])
```
Out[35]:
```
array([[-9.22396837e-02,  4.60456800e-07,  1.31804970e-04]])
```

In [170]:
```
hp_func = partial(boot_fn, MS(['income','balance']), 'default')
```

In [174]:
```
hp_se = boot_fn(hp_func,
Default,
idx)
hp_se
idx=[2,8,10,23]
```
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_1844/2000488034.py in <module>
      1 hp_se = boot_fn(hp_func,
      2 Default,
----> 3 idx)
      4 hp_se
      5 idx=[2,8,10,23]
```

The method in the textbook was giving an error I was not able to resolve, hence I incorporated a simpler function not considering idx.

Simpler implementation:

In [166]:
```
from sklearn.utils import resample
from statsmodels.discrete.discrete_model import Logit


def boot_fn(df):
    return resample(df)

train_samples = boot_fn(Default)
X = Default[['income', 'balance']]
X = sm.add_constant(X, prepend=True)
y = Default['default']

model = Logit(y.astype(float), X.astype(float))
result = model.fit()
print(result.summary())
```
```
Optimization terminated successfully.
         Current function value: 0.078948
         Iterations 10
                           Logit Regression Results
==============================================================================
Dep. Variable:                default   No. Observations:                10000
Model:                          Logit   Df Residuals:                     9997
Method:                           MLE   Df Model:                            2
Date:                Sat, 28 Oct 2023   Pseudo R-squ.:                  0.4594
Time:                        06:48:57   Log-Likelihood:                -789.48
converged:                       True   LL-Null:                       -1460.3
Covariance Type:            nonrobust   LLR p-value:                 4.541e-292
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const        -11.5405      0.435    -26.544      0.000     -12.393     -10.688
income      2.081e-05   4.99e-06      4.174      0.000     1.1e-05    3.06e-05
balance        0.0056      0.000     24.835      0.000       0.005       0.006
==============================================================================

Possibly complete quasi-separation: A fraction 0.14 of observations can be
perfectly predicted. This might indicate that there is complete
quasi-separation. In this case some parameters will not be identified.
```

(c)

(c)

```
In [175]: BSamples = 1000

intercept = []
income = []
balance = []

for i in range(BSamples):
    train = boot_fn(Default)
    X = Default[['income', 'balance']]
    X = sm.add_constant(X, prepend=True)
    y = Default['default']

    model = Logit(y.astype(float), X.astype(float))
    result = model.fit(disp=False)
    intercept.append(result.params.const)
    income.append(result.params.income)
    balance.append(result.params.balance)

print("Standard error for intercept: " +str(np.std(intercept, ddof=1)))
print("Standard error for income: " +str(np.std(income, ddof=1)))
print("Standard error for balance: " +str(np.std(balance, ddof=1)))
```

```
Standard error for intercept: 1.777245684509363e-15
Standard error for income: 6.779654253041699e-21
Standard error for balance: 0.0
```

(d) The error for the intercept/ constant is zero for both methods. However, the error using bootstrap leads to a significant reduction in the standard error for income and balance, proving it could be an effective technique to get more value extracted from a given dataset.

Section 6.6

Conceptual

1. (a) The model obtained using best subset selection is likely to have smallest training RSS. Forward stepwise selection would incrementally add one predictor at a time starting with a null model having no predictors, whereas backward stepwise selection gets rid of one predictor at a time starting with a model having all predictors, based on the effect of that predictor on training RSS (adjusted). For forward stepwise selection a total of p+1+p+…+p-k+1 models need to be evaluated with the selection for getting k best predictors needing comparison among p-k+1 models in the k^{th} iteration. For backward selection, a total of 1(corresponding to p+1 predictors)+2+…+k+1 models need to be evaluated with the selection for getting k best predictors needing comparison among k+1 models in the k^{th} iteration. However, for a particular k we are likely to get the best result with best subset selection as it has considered all models with k predictors among p possible predictor choices ${}^{p+1}C_k$ models.

(b) Since test data is unseen data, it is not possible to say for sure which model would perform the best on it. However it is likely that the model chosen using best subset selection (for k predictors) would give the best result.

(c) i. True. The predictors in the k-variable model identified by forward stepwise are a subset of the predictors in the (k+1)-variable model identified by forward stepwise selection. The (k+1) variable model would have one more predictor added to those identified by the k-variable model making the k predictors a subset.

ii. True. The predictors in the k-variable model identified by backward stepwise are a subset of the predictors in the (k + 1)- variable model identified by backward stepwise selection as it would have all k predictors in the (k+1) variable model barring 1 making it a subset.

iii. False. We cannot establish a relationship between the predictors arrived at using forward subset selection and backward subset selection.

iv. False. We cannot assume/establish a relationship between the predictors arrived at using forward subset selection and backward subset selection.

v. False, since best subset selection for a particular k takes place by evaluating $^{p+1}C_{k}$ possible models without incremental addition or reduction of predictors.

8.

(a)

In [47]:
```
np.random.seed(46)
X = np.random.normal(0, 1, 100)
noise_vector = np.random.normal(0, 1, 100)
print(X)
print(noise_vector)
```

```
[ 0.58487584  1.23119574  0.82190026 -0.79922836  0.41205323 -0.17615661
 -0.07317197 -0.56566639 -0.09346524  0.85730108 -0.86222329  0.0164811
  1.56511109 -0.46912008 -0.39230073  0.816667    0.07637529 -0.10009311
  1.62375712 -1.33654165 -0.13513225 -0.47834068 -1.59497503 -0.86895932
 -0.03272285 -1.52743151 -0.12459807 -0.26194916  0.99535121  0.31754335
 -0.03826044 -0.06819798 -0.44227583 -0.47929677  0.05151458 -0.97491329
 -1.43318077 -0.35901965  0.45429619 -0.805498   -2.69420458  0.50108854
 -0.00463496 -0.39293844 -0.15526017  0.4497048   1.18759353 -0.33459034
 -0.11523609  1.83660093  0.72858767 -1.13211109 -1.41819363  2.16117541
 -1.75606623  0.71127492 -0.69144395  0.25705654  1.04501157 -1.92214712
 -0.14661499  1.36325084  0.95771094  2.06416345 -0.85902444 -0.39719568
```

(b)

In [48]:
```
β0=6
β1=4
β2=2
β3=12
y = β0 + β1*X + β2*(X**2) + β3*(X**3) + noise_vector
print(y)
```

```
[  11.21683919   35.79932435   17.59508604   -4.01793709    9.90428841
    4.27394399    5.11250602    1.75837689    5.42753084   18.05517605
   -3.48615483    5.70989305   65.23553271    3.09541324    4.29687452
   15.16241291    6.27227851    4.77910948   71.12081528  -24.23460295
    6.29694318    4.51415132  -41.9450757    -5.79332842    6.27585372
  -37.75803925    6.61525131    4.42554255   23.45174852    7.91814683
    6.51984004    6.48326662    1.7583945     2.71964104    6.30781542
   -7.51152982  -29.62688168    4.83747611    8.85369108   -2.6220777
 -224.15285592   11.44439463    6.02613695    3.15551188    6.14665961
    7.34713482   32.90339284    5.51241574    4.78233493   95.27775823
```

(c)

In [41]:
```
import sklearn.model_selection as skm
import sklearn.linear_model as skl
from ISLP.models import \
    (Stepwise,
    sklearn_selected,
    sklearn_selection_path)
```

In [42]:
```
!pip install l0bnb
```

```
Requirement already satisfied: l0bnb in c:\users\ishaj\anaconda3\lib\site-packa
ges (1.0.0)
Requirement already satisfied: numba>=0.53.1 in c:\users\ishaj\anaconda3\lib\si
te-packages (from l0bnb) (0.54.1)
Requirement already satisfied: numpy>=1.18.1 in c:\users\ishaj\anaconda3\lib\si
te-packages (from l0bnb) (1.20.3)
Requirement already satisfied: scipy>=1.4.1 in c:\users\ishaj\anaconda3\lib\sit
e-packages (from l0bnb) (1.7.1)
Requirement already satisfied: llvmlite<0.38,>=0.37.0rc1 in c:\users\ishaj\anac
onda3\lib\site-packages (from numba>=0.53.1->l0bnb) (0.37.0)
Requirement already satisfied: setuptools in c:\users\ishaj\anaconda3\lib\site-
packages (from numba>=0.53.1->l0bnb) (58.0.4)
```

In [43]:
```
from functools import partial
import sklearn
```

In [44]:
```
c1=X
c2=X**2
c3=X**3
c4=X**4
c5=X**5
c6=X**6
c7=X**7
c8=X**8
c9=X**9
c10=X**10
```

```
In [46]: X_df=pd.DataFrame()
         X_df['c1']=pd.DataFrame(c1)
         print(X_df)
```

```
          c1
0   0.584876
1   1.231196
2   0.821900
3  -0.799228
4   0.412053
..       ...
95 -0.618316
96 -0.305315
97  0.326829
98  0.181195
99 -0.955610

[100 rows x 1 columns]
```

```
In [47]: X_df['c2']=pd.DataFrame(c2)
         print(X_df)
```

```
          c1        c2
0   0.584876  0.342080
1   1.231196  1.515843
2   0.821900  0.675520
3  -0.799228  0.638766
4   0.412053  0.169788
..       ...       ...
95 -0.618316  0.382314
96 -0.305315  0.093217
97  0.326829  0.106817
98  0.181195  0.032832
99 -0.955610  0.913190

[100 rows x 2 columns]
```

```
In [48]: X_df['c3']=pd.DataFrame(c3)
         X_df['c4']=pd.DataFrame(c4)
         X_df['c5']=pd.DataFrame(c5)
         X_df['c6']=pd.DataFrame(c6)
         X_df['c7']=pd.DataFrame(c7)
         X_df['c8']=pd.DataFrame(c8)
         X_df['c9']=pd.DataFrame(c9)
         X_df['c10']=pd.DataFrame(c10)
         print(X_df)
```

```
          c1        c2        c3        c4        c5        c6        c7  \
0   0.584876  0.342080  0.200074  0.117019  0.068441  0.040030  0.023412
1   1.231196  1.515843  1.866299  2.297780  2.829017  3.483073  4.288345
2   0.821900  0.675520  0.555210  0.456327  0.375056  0.308258  0.253358
3  -0.799228  0.638766 -0.510520  0.408022 -0.326103  0.260631 -0.208303
4   0.412053  0.169788  0.069962  0.028828  0.011879  0.004895  0.002017
..       ...       ...       ...       ...       ...       ...       ...
95 -0.618316  0.382314 -0.236391  0.146164 -0.090376  0.055881 -0.034552
96 -0.305315  0.093217 -0.028461  0.008689 -0.002653  0.000810 -0.000247
97  0.326829  0.106817  0.034911  0.011410  0.003729  0.001219  0.000398
98  0.181195  0.032832  0.005949  0.001078  0.000195  0.000035  0.000006
99 -0.955610  0.913190 -0.872653  0.833916 -0.796898  0.761524 -0.727719

          c8            c9           c10
0   0.013693  8.008905e-03  4.684215e-03
1   5.279792  6.500458e+00  8.003336e+00
2   0.208235  1.711481e-01  1.406667e-01
3   0.166482 -1.330571e-01  1.063430e-01
4   0.000831  3.424364e-04  1.411020e-04
..       ...           ...           ...
95  0.021364 -1.320968e-02  8.167754e-03
96  0.000076 -2.305357e-05  7.038611e-06
97  0.000130  4.254854e-05  1.390609e-05
98  0.000001  2.105355e-07  3.814807e-08
99  0.695416 -6.645460e-01  6.350466e-01

[100 rows x 10 columns]
```

In [49]:
```
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
```

In [50]:
```
from statsmodels.api import OLS
design = MS(X_df.columns).fit(X_df)
Y = y
X = design.transform(X_df)
sigma2 = OLS(Y,X).fit().scale
```

In [51]:
```
neg_Cp = partial(nCp, sigma2)
```

In [52]:
```
strategy = Stepwise.first_peak(design,
  direction='forward',
  max_terms=len(design.terms))
```

In [53]:
```
#Using all predictors
X_df_MSE = sklearn_selected(OLS,
          strategy)
X_df_MSE.fit(X_df, Y)
X_df_MSE.selected_state_
```

Out[53]:
```
('c1', 'c10', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9')
```

In [54]:
```
#forward stepwise selection
X_df_Cp = sklearn_selected(OLS,
        strategy,
        scoring=neg_Cp)
X_df_Cp.fit(X_df, Y)
X_df_Cp.selected_state_
```

Out[54]:
```
('c1', 'c2', 'c3')
```

The model created on the basis of c_p has c1, c3 and c3 (X, X^2 and X^3) as the predictors which aligns with the simulated data.

To get the coefficients for the same, we build a model using OLS (model chosen and passed to evaluate c_p). The details for the same are below.

```
In [58]: #Estimating coefficients
         x=X_df[['c1','c2','c3']]
         x = sm.add_constant(x)
         model = sm.OLS(y, x).fit()
         model.summary()
```

Out[58]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.992
Model:	OLS	**Adj. R-squared:**	0.991
Method:	Least Squares	**F-statistic:**	3781.
Date:	Sat, 28 Oct 2023	**Prob (F-statistic):**	1.74e-99
Time:	01:48:57	**Log-Likelihood:**	-143.32
No. Observations:	100	**AIC:**	294.6
Df Residuals:	96	**BIC:**	305.1
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	11.9643	0.129	92.983	0.000	11.709	12.220
c1	3.8956	0.200	19.457	0.000	3.498	4.293
c2	6.1987	0.088	70.194	0.000	6.023	6.374
c3	2.0437	0.063	32.638	0.000	1.919	2.168

Omnibus:	0.386	**Durbin-Watson:**	2.100
Prob(Omnibus):	0.824	**Jarque-Bera (JB):**	0.542
Skew:	-0.113	**Prob(JB):**	0.763
Kurtosis:	2.720	**Cond. No.**	6.41

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

(d)

In [55]:
```
strategy = Stepwise.first_peak(design,
  direction='backwards',
  max_terms=len(design.terms))
```

In [56]:
```
#Using all predictors
X_df_MSE = sklearn_selected(OLS,
          strategy)
X_df_MSE.fit(X_df, Y)
X_df_MSE.selected_state_
```

Out[56]:
```
('c1', 'c10', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9')
```

In [57]:
```
#backward stepwise selection
X_df_Cp = sklearn_selected(OLS,
        strategy,
        scoring=neg_Cp)
X_df_Cp.fit(X_df, Y)
X_df_Cp.selected_state_
```

Out[57]:
```
('c1', 'c2', 'c3')
```

In [58]:
```
#Estimating coefficients
x=X_df[['c1','c2','c3']]
x = sm.add_constant(x)
model = sm.OLS(y, x).fit()
model.summary()
```

Backward stepwise selection includes the same predictors as forward stepwise selection and is in accordance with the simulated data. Hence we end up with same model and coefficients.

In [58]:
```
#Estimating coefficients
x=X_df[['c1','c2','c3']]
x = sm.add_constant(x)
model = sm.OLS(y, x).fit()
model.summary()
```

Out[58]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.992
Model:	OLS	**Adj. R-squared:**	0.991
Method:	Least Squares	**F-statistic:**	3781.
Date:	Sat, 28 Oct 2023	**Prob (F-statistic):**	1.74e-99
Time:	01:48:57	**Log-Likelihood:**	-143.32
No. Observations:	100	**AIC:**	294.6
Df Residuals:	96	**BIC:**	305.1
Df Model:	3		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	11.9643	0.129	92.983	0.000	11.709	12.220
c1	3.8956	0.200	19.457	0.000	3.498	4.293
c2	6.1987	0.088	70.194	0.000	6.023	6.374
c3	2.0437	0.063	32.638	0.000	1.919	2.168

Omnibus:	0.386	**Durbin-Watson:**	2.100
Prob(Omnibus):	0.824	**Jarque-Bera (JB):**	0.542
Skew:	-0.113	**Prob(JB):**	0.763
Kurtosis:	2.720	**Cond. No.**	6.41

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

(e)

In [59]:
```
full_path = sklearn_selection_path(OLS, strategy)
```

In [60]:
```
K=5
kfold = skm.KFold(K,
          random_state=0,
          shuffle=True)
Yhat_cv = skm.cross_val_predict(full_path,
          X_df,
          Y,
          cv=kfold)
Yhat_cv.shape
```

Out[60]: (100, 12)

In [61]:
```
X=X_df
Xs = X - X.mean(0)[None,:]
X_scale = X.std(0)
Xs = Xs / X_scale[None,:]
lambdas = 10**np.linspace(8, -2, 100) / Y.std()
soln_array = skl.ElasticNet.path(Xs,
            Y,
            l1_ratio=1.,
            alphas=lambdas)[1]
soln_array.shape
```

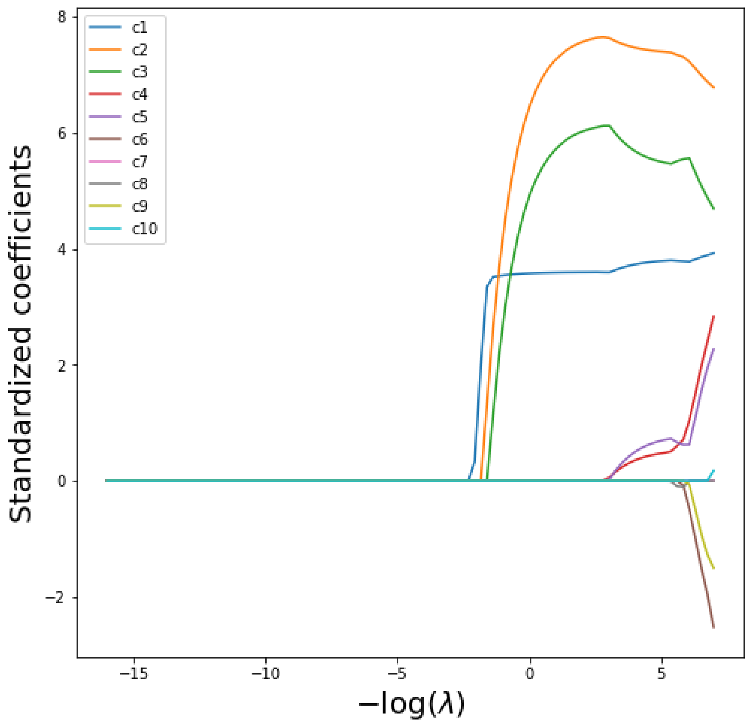
In [62]:
```
soln_path = pd.DataFrame(soln_array.T,
            columns=X_df.columns,
            index=-np.log(lambdas))
soln_path.index.name ='negative log(lambda)'
soln_path
```

Out[62]:

negative log(lambda)	c1	c2	c3	c4	c5	c6	c7	c8	c9	
-16.016163	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.00
-15.783579	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.00
-15.550995	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.00
-15.318410	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.00
-15.085826	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.000000	0.00
...	...	...	...	...	...	...	...	...	...	
6.079350	3.779575	7.217595	5.564099	1.047526	0.621828	-0.485962	-0.0	-0.0	-0.062701	0.00
6.311934	3.817243	7.109070	5.328511	1.482148	1.074609	-0.965380	-0.0	-0.0	-0.473306	0.00
6.544519	3.856826	6.984260	5.083798	1.968306	1.544372	-1.491779	-0.0	-0.0	-0.904324	0.00
6.777103	3.890130	6.873742	4.876664	2.396546	1.943720	-1.954646	-0.0	-0.0	-1.273198	0.00
7.009687	3.923948	6.776972	4.687856	2.828237	2.269246	-2.518595	-0.0	-0.0	-1.503776	0.17

100 rows × 10 columns

In [63]:
```
path_fig, ax = subplots(figsize=(8,8))
soln_path.plot(ax=ax, legend=False)
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Standardized coefficients', fontsize=20)
ax.legend(loc='upper left');
```



In accordance with the simulated data, the coefficients for c1, c2 and c3 seem to have the largest values.

In [64]:
```
beta_hat = soln_path.loc[soln_path.index[-5]]
lambdas[-5], beta_hat
```

Out[64]:
```
(0.0022896643024310913,
 c1     3.779575
 c2     7.217595
 c3     5.564099
 c4     1.047526
 c5     0.621828
 c6    -0.485962
 c7    -0.000000
 c8    -0.000000
 c9    -0.062701
 c10    0.000000
 Name: 6.079350064986618, dtype: float64)
```

In [65]:
```
np.linalg.norm(beta_hat)
```

Out[65]:
```
9.952996067489783
```

In [66]:
```
beta_hat = soln_path.loc[soln_path.index[88]]
lambdas[88], np.linalg.norm(beta_hat)
```

Out[66]:
```
(0.011663865964182216, 10.05320649591814)
```

In [67]:
```
scaler = StandardScaler(with_mean=True, with_std=True)
```

In [68]:
```
y.shape
```

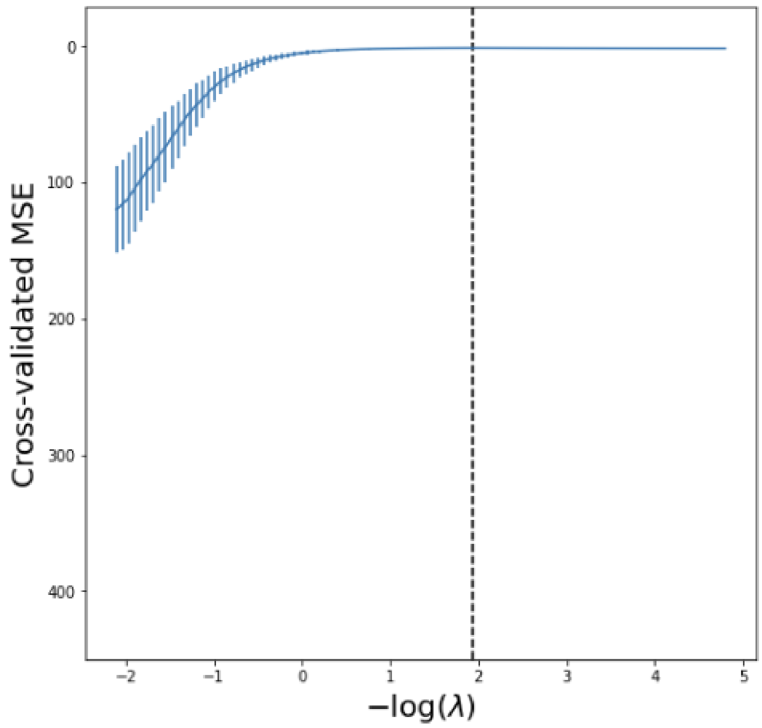
Out[68]:
```
(100,)
```

In [69]:
```
from sklearn.pipeline import Pipeline
lassoCV = skl.ElasticNetCV(n_alphas=100,
        l1_ratio=1,
        cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler),
        ('lasso', lassoCV)])
pipeCV.fit(X_df, y)
tuned_lasso = pipeCV.named_steps['lasso']
tuned_lasso.alpha_
```

Out[69]:
```
0.14372763056044713
```

In [72]:
```
np.min(tuned_lasso.mse_path_.mean(1))
```
Out[72]:
```
1.2964279916775396
```

In [73]:
```
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([450,-29])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated MSE', fontsize=20);
```



In [74]:
```
tuned_lasso.coef_
```
Out[74]:
```
array([3.5779212 , 7.51304844, 5.97585017, 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ])
```

The following plot allows us to conclude that the minimum MSE of around 1.3v is obtained with lambda around $e^{-1.9}$.

The tuned lasso coefficients seem to be in accordance with simulated data having coefficients 0 for all predictors besides c1, c2 and c3.

(f) Forward stepwise selection:

(f)

In [75]:
```
β7=9
β0=12
X_df.head()
```

Out[75]:

	c1	c2	c3	c4	c5	c6	c7	c8	c9	
0	0.584876	0.342080	0.200074	0.117019	0.068441	0.040030	0.023412	0.013693	0.008009	0.0
1	1.231196	1.515843	1.866299	2.297780	2.829017	3.483073	4.288345	5.279792	6.500458	8.0
2	0.821900	0.675520	0.555210	0.456327	0.375056	0.308258	0.253358	0.208235	0.171148	0.1
3	-0.799228	0.638766	-0.510520	0.408022	-0.326103	0.260631	-0.208303	0.166482	-0.133057	0.1
4	0.412053	0.169788	0.069962	0.028828	0.011879	0.004895	0.002017	0.000831	0.000342	0.0

In [76]:
```
np.random.seed(44)
X = np.random.normal(0, 1, 100)
noise_vector = np.random.normal(0, 1, 100)
```

In [77]:
```
y=β0+β7*(X**7)+noise_vector
```

In [107]:
```
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
```

In [108]:
```
from statsmodels.api import OLS
design = MS(X_df.columns).fit(X_df)
Y = y
X = design.transform(X_df)
sigma2 = OLS(Y,X).fit().scale
```

In [109]:
```
neg_Cp = partial(nCp, sigma2)
```

In [110]:
```
strategy = Stepwise.first_peak(design,
  direction='forward',
  max_terms=len(design.terms))
```

In [111]:
```
#Using all predictors
X_df_MSE = sklearn_selected(OLS,
          strategy)
X_df_MSE.fit(X_df, Y)
X_df_MSE.selected_state_
```

Out[111]: ('c1', 'c10', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9')

In [112]:
```
#forward stepwise selection
X_df_Cp = sklearn_selected(OLS,
        strategy,
        scoring=neg_Cp)
X_df_Cp.fit(X_df, Y)
X_df_Cp.selected_state_
```

Out[112]: ()

Lasso:

In [78]:
```
K=5
Y=y
kfold = skm.KFold(K,
          random_state=0,
          shuffle=True)
Yhat_cv = skm.cross_val_predict(full_path,
          X_df,
          Y,
          cv=kfold)
Yhat_cv.shape
```

Out[78]:
```
(100, 12)
```

In [80]:
```
X=X_df
Xs = X - X.mean(0)[None,:]
X_scale = X.std(0)
Xs = Xs / X_scale[None,:]
lambdas = 10**np.linspace(8, -2, 100) / Y.std()
soln_array = skl.ElasticNet.path(Xs,
            Y,
            l1_ratio=1.,
            alphas=lambdas)[1]
soln_array.shape
```

```
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:614: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations. Duality gap: 262110515.83100846, tolerance: 55505.76739653672
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:614: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations. Duality gap: 263982848.4162998, tolerance: 55505.76739653672
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:614: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations. Duality gap: 265610622.53577352, tolerance: 55505.76739653672
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:614: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations. Duality gap: 266925246.14245558, tolerance: 55505.76739653672
```

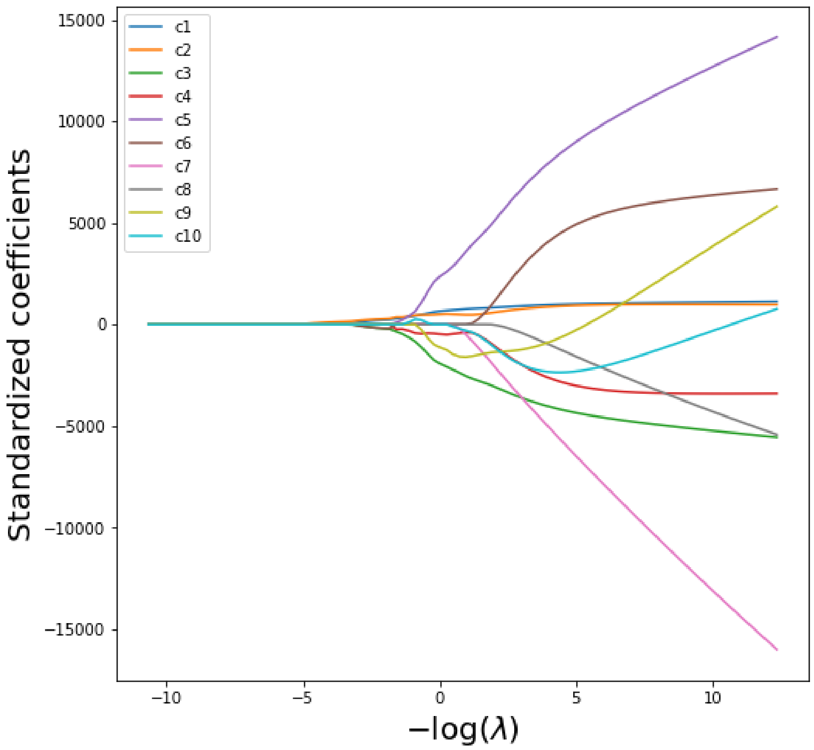
In [81]:
```python
soln_path = pd.DataFrame(soln_array.T,
             columns=X_df.columns,
             index=-np.log(lambdas))
soln_path.index.name ='negative log(lambda)'
soln_path
```

Out[81]:

negative log(lambda)	c1	c2	c3	c4	c5	c6	
-10.662147	0.000000	0.000000	0.000000	0.000000	-0.000000	0.000000	-
-10.429563	0.000000	0.000000	0.000000	0.000000	-0.000000	0.000000	-
-10.196978	0.000000	0.000000	0.000000	0.000000	-0.000000	0.000000	-
-9.964394	0.000000	0.000000	0.000000	0.000000	-0.000000	0.000000	-
-9.731810	0.000000	0.000000	0.000000	0.000000	-0.000000	0.000000	-
...	...	...	...	...	...	...	
11.433366	1107.452222	975.638856	-5436.774682	-3411.253282	13585.869240	6555.266305	-1488
11.665951	1110.191593	974.654454	-5467.870550	-3410.209082	13730.056342	6584.786762	-1517
11.898535	1112.911489	973.652242	-5498.779233	-3409.028459	13873.476314	6613.842527	-1545
12.131119	1115.613002	972.635307	-5529.507799	-3407.728755	14016.145268	6642.464693	-1573
12.363704	1118.297087	971.606229	-5560.062445	-3406.324554	14158.077525	6670.679690	-1601

100 rows × 10 columns

In [82]:
```python
path_fig, ax = subplots(figsize=(8,8))
soln_path.plot(ax=ax, legend=False)
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Standardized coefficients', fontsize=20)
ax.legend(loc='upper left');
```



In [97]:
```
beta_hat = soln_path.loc[soln_path.index[88]]
lambdas[88], beta_hat
```

Out[97]:
```
(0.35477811071336224,
 c1      778.034200
 c2      461.886456
 c3    -2701.522286
 c4     -374.267625
 c5     4011.712341
 c6        0.000000
 c7     -945.067346
 c8       -0.000000
 c9    -1496.307063
 c10    -358.066704
 Name: 1.0362627251698588, dtype: float64)
```

In [98]:
```
beta_hat = soln_path.loc[soln_path.index[88]]
lambdas[88], np.linalg.norm(beta_hat)
```

Out[98]:
```
(0.35477811071336224, 5254.6274861788725)
```

In [99]:
```
scaler = StandardScaler(with_mean=True, with_std=True)
```

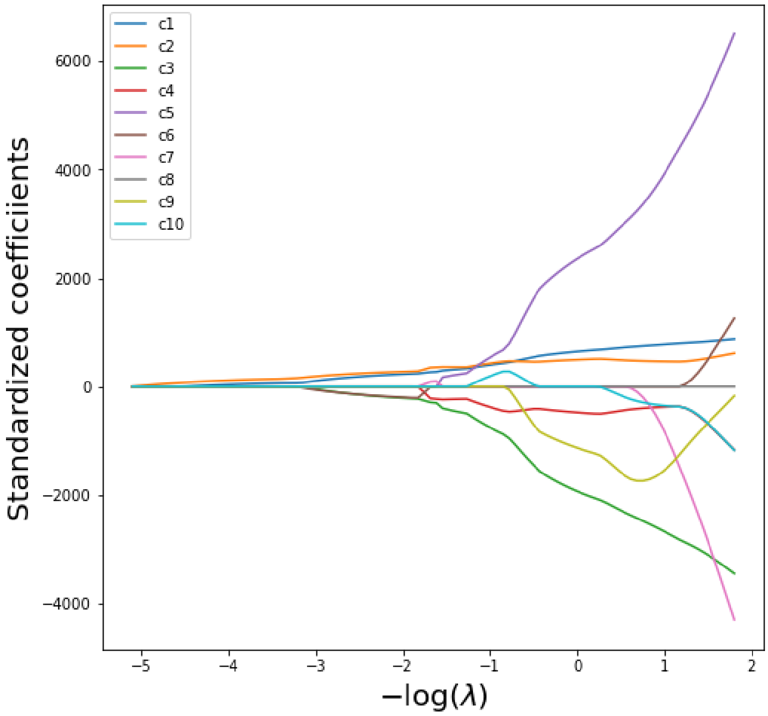
In [100]:
```
from sklearn.pipeline import Pipeline
lassoCV = skl.ElasticNetCV(n_alphas=100,
        l1_ratio=1,
        cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler),
        ('lasso', lassoCV)])
pipeCV.fit(X_df, y)
tuned_lasso = pipeCV.named_steps['lasso']
tuned_lasso.alpha_
```

```
erance: 54049.88087435726
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 681199.0972127914, tol
erance: 54049.88087435726
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 1026439.9174004793, to
lerance: 54049.88087435726
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 1192254.7242325544, to
lerance: 54049.88087435726
```

In [101]:
```
lambdas , soln_array = skl.Lasso.path(Xs,
            Y,
            l1_ratio=1,
            n_alphas=100)[:2]
soln_path = pd.DataFrame(soln_array.T,
            columns=X_df.columns,
            index=-np.log(lambdas))
```

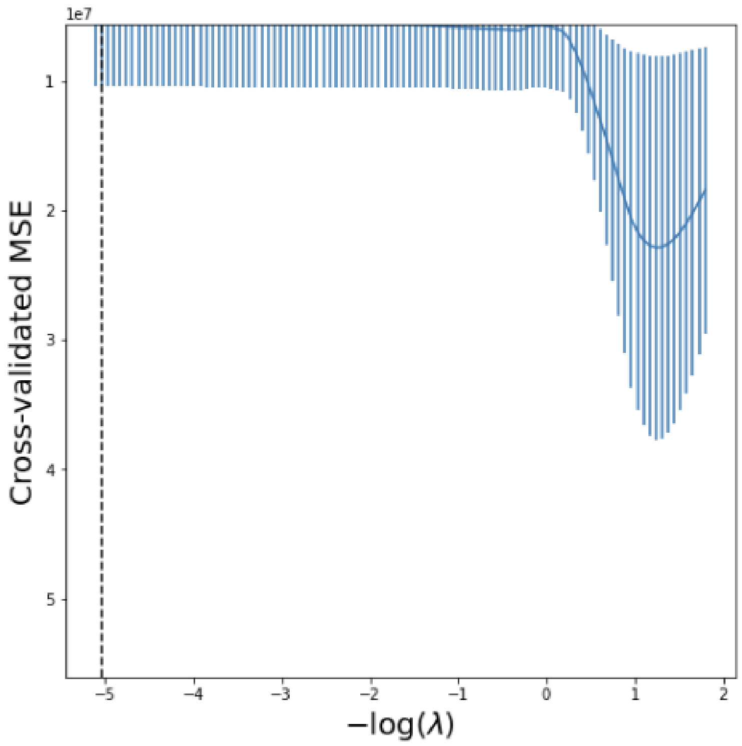
```
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 97982.79800987244, tol
erance: 55505.76739653672
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 67098.16203403473, tol
erance: 55505.76739653672
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 316633.50047945976, to
lerance: 55505.76739653672
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 465166.39819544554, to
lerance: 55505.76739653672
```

In [102]:
```
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);
```



In [103]:
```
np.min(tuned_lasso.mse_path_.mean(1))
```
Out[103]:
```
5606162.20587495
```

In [105]:
```
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([5606162000,5606162])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('Cross-validated MSE', fontsize=20);
```



In [106]:
```
tuned_lasso.coef_
```
Out[106]:
```
array([ 0.        , 11.15434868,  0.        ,  0.        , -0.        ,
        0.        , -0.        ,  0.        , -0.        ,  0.        ])
```

In [163]:
```python
from sklearn.linear_model import LassoCV

# 5 fold cross-validation
model = LassoCV(alphas=alphas, fit_intercept=True, cv=5,random_state=8)
model.fit(X_df, y)

predictions = model.predict(X_df)
print("Test Error: " +str(mean_squared_error(y, predictions)))
print("Number of Non-zero coefficients: " + str(len(model.coef_)))
```

```
nt to increase the number of iterations. Duality gap: 262772340.79266232, to
lerance: 54082.490853037845
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 262767909.55811313, to
lerance: 54082.490853037845
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 262763513.71105042, to
lerance: 54082.490853037845
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:628: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations, check the scale of the features or
consider increasing regularisation. Duality gap: 1.739e+08, tolerance: 5.482
e+04
  model = cd_fast.enet_coordinate_descent(
```

In [164]:
```python
print("Test Error: " +str(mean_squared_error(y, predictions)))
print("Number of Non-zero coefficients: " + str(len(model.coef_)))
```

```
Test Error: 5416860.879167331
Number of Non-zero coefficients: 10
```

Both the methods seem to be inaccurate since they do not seem to identify c7 as a predictor of interest.

9.

9

In [55]:
```
College = load_data('college')
College.head()
```

Out[55]:

	Private	Apps	Accept	Enroll	Top10perc	Top25perc	F.Undergrad	P.Undergrad	Outstate	Room.Board	Books	Personal	PhD	Terminal	S.F.Ratio	perc.alu
0	Yes	1660	1232	721	23	52	2885	537	7440	3300	450	2200	70	78	18.1	
1	Yes	2186	1924	512	16	29	2683	1227	12280	6450	750	1500	29	30	12.2	
2	Yes	1428	1097	336	22	50	1036	99	11250	3750	400	1165	53	66	12.9	
3	Yes	417	349	137	60	89	510	63	12960	5450	450	875	92	97	7.7	
4	Yes	193	146	55	16	44	249	869	7560	4120	800	1500	76	72	11.9	

In [56]:
```
lab_enc = preprocessing.LabelEncoder()
encoded_college = lab_enc.fit_transform(College['Private'])
College['Private']=encoded_college
print(College)
```

```
     Private   Apps  Accept  Enroll  Top10perc  Top25perc  F.Undergrad  \
0          1   1660    1232     721         23         52         2885   
1          1   2186    1924     512         16         29         2683   
2          1   1428    1097     336         22         50         1036   
3          1    417     349     137         60         89          510   
4          1    193     146      55         16         44          249   
..       ...    ...     ...     ...        ...        ...          ...   
772        0   2197    1515     543          4         26         3089   
773        1   1959    1805     695         24         47         2849   
774        1   2097    1915     695         34         61         2793   
775        1  10705    2453    1317         95         99         5217   
776        1   2989    1855     691         28         63         2988   

     P.Undergrad  Outstate  Room.Board  Books  Personal  PhD  Terminal  \
0            537      7440        3300    450      2200   70        78   
1           1227     12280        6450    750      1500   29        30   
2             99     11250        3750    400      1165   53        66   
3             63     12960        5450    450       875   92        97   
4            869      7560        4120    800      1500   76        72   
..           ...       ...         ...    ...       ...  ...       ...   
772         2029      6797        3900    500      1200   60        60   
```

(a)

In [57]:
```
# split the dataset

X=College.drop(['Apps'], axis=1)
X = sm.add_constant(X)
#X.head()
y=College['Apps']
#y.head()
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=44)
```

(b)

In [63]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
model = LinearRegression(fit_intercept=True)
model.fit(X_train,y_train)
y_predicted = model.predict(X_test)

print("Test Error: " +str(mean_squared_error(y_test, y_predicted)))
```

```
Test Error: 1635283.8950918918
```

(c)

In [155]:
```
from sklearn.linear_model import RidgeCV
#Trying 200 values of lambda
n_alphas = 200
alphas = np.logspace(-10, 2, n_alphas)
#5 fold cross-validation
model = RidgeCV(alphas=alphas, fit_intercept=True, cv=5)
model.fit(X_train, y_train)

predictions = model.predict(X_test)
print("Test Error: " +str(mean_squared_error(y_test, predictions)))
```

```
be accurate.
  return linalg.solve(A, Xy, assume_a="pos", overwrite_a=True).T
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_ridge.py:21
1: LinAlgWarning: Ill-conditioned matrix (rcond=9.03572e-17): result may not
be accurate.
  return linalg.solve(A, Xy, assume_a="pos", overwrite_a=True).T
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_ridge.py:21
1: LinAlgWarning: Ill-conditioned matrix (rcond=9.45719e-17): result may not
be accurate.
  return linalg.solve(A, Xy, assume_a="pos", overwrite_a=True).T
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_ridge.py:21
1: LinAlgWarning: Ill-conditioned matrix (rcond=1.03816e-16): result may not
be accurate.
  return linalg.solve(A, Xy, assume_a="pos", overwrite_a=True).T
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_ridge.py:21
1: LinAlgWarning: Ill-conditioned matrix (rcond=1.08658e-16): result may not
be accurate.
  return linalg.solve(A, Xy, assume_a="pos", overwrite_a=True).T
```

```
Test Error: 1624684.52400796
```

(d)

In [157]:
```python
from sklearn.linear_model import LassoCV

# 5 fold cross-validation
model = LassoCV(alphas=alphas, fit_intercept=True, cv=5,random_state=8)
model.fit(X_train, y_train)

predictions = model.predict(X_test)
print("Test Error: " +str(mean_squared_error(y_test, predictions)))
print("Number of Non-zero coefficients: " + str(len(model.coef_)))
```

```
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 161708181.11003068, to
lerance: 543928.5227117242
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 178689288.56169817, to
lerance: 543928.5227117242
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 162131029.25556096, to
lerance: 543928.5227117242
  model = cd_fast.enet_coordinate_descent_gram(
C:\Users\ishaj\anaconda3\lib\site-packages\sklearn\linear_model\_coordinate_
descent.py:614: ConvergenceWarning: Objective did not converge. You might wa
nt to increase the number of iterations. Duality gap: 181776686.38645488, to
lerance: 543928.5227117242
  model = cd_fast.enet_coordinate_descent_gram(
```

In [158]:
```python
print("Test Error: " +str(mean_squared_error(y_test, predictions)))
print("Number of Non-zero coefficients: " + str(len(model.coef_)))
```

```
Test Error: 1635283.8950919267
Number of Non-zero coefficients: 18
```

(g) Both Lasso and ridge models seem to give similar error and the lasso method seems to suggest all predictors should be considered.