Python Tutorial 7

April 11, 2020

This tutorial is for Dr. Xin Tong's DSO 530 class at the University of Southern California in spring 2020. It aims to give you some supplementary code of *Lecture 6* on how to implement *Subset Selection* and *Shrinkage Methods* using Python.

1 Linear Model Subset Selection

[1]:
```
%matplotlib inline
import pandas as pd
import numpy as np
import itertools
import time
import statsmodels.api as sm
import matplotlib.pyplot as plt
```

1.1 Best Subset Selection

Here we apply the best subset selection approach to the Hitters data. We wish to predict a baseball player's Salary on the basis of various statistics associated with performance in the previous year. Let's take a quick look:

[14]:
```
df = pd.read_csv('Hitters.csv')
df.head()
```

[14]:
```
   Player  AtBat  Hits  HmRun  Runs  RBI  Walks  Years  CAtBat  CHits  …  \
0  #NAME?    293    66      1    30   29     14      1     293     66  …
1  #NAME?    315    81      7    24   38     39     14    3449    835  …
2  #NAME?    479   130     18    66   72     76      3    1624    457  …
3  #NAME?    496   141     20    65   78     37     11    5628   1575  …
4  #NAME?    321    87     10    39   42     30      2     396    101  …

   CRuns  CRBI  CWalks  League Division  PutOuts  Assists  Errors  Salary  \
0     30    29      14       A        E      446       33      20     NaN
1    321   414     375       N        W      632       43      10   475.0
2    224   266     263       A        W      880       82      14   480.0
3    828   838     354       N        E      200       11       3   500.0
4     48    46      33       N        E      805       40       4    91.5
```

```
  NewLeague
0         A
1         N
2         A
3         N
4         N

[5 rows x 21 columns]
```

First, we note that the *Salary* variable is missing for some of the players. The *isnull()* function can be used to identify the missing observations. It returns a vector of the same length as the input vector, with a *TRUE* value for any missing element, and a *FALSE* value for a non-missing element. The *sum()* function can then be used to count the missing elements:

[15]:
```
print(df["Salary"].isnull().sum())
```

```
59
```

We see that *Salary* is missing for 59 players. The *dropna()* function removes all of the rows that have missing values in any variable:

[16]:
```
# Drop the player names as they are not a reasonable potential predictor
df = df.drop('Player', axis=1)

# Print the dimensions of the original Hitters data (322 rows x 20␣
↪columns)(Players' names not included)
print("before dropna():",df.shape)

# Drop any rows the contain missing values. Note that this is not necessarily␣
↪the recommended practice for a given problem.
df = df.dropna()

# Print the dimensions of the modified Hitters data (263 rows x 20 columns)
print("after dropna():",df.shape)

# One last check: should return 0
print("check the number of missing salary after dropna():",df["Salary"].
↪isnull().sum())
```

```
before dropna(): (322, 20)
after dropna(): (263, 20)
check the number of missing salary after dropna(): 0
```

Here, we use *pd.get_dummies* function to transform the original categorical variable *League*, *Division* and *NewLeague* into the usable "1/0" format.

[17]:
```
df[['League', 'Division', 'NewLeague']].head()
```

[17]:

	League	Division	NewLeague
1	N	W	N
2	A	W	A
3	N	E	N
4	N	E	N
5	A	W	A

[18]:
```
dummies = pd.get_dummies(df[['League', 'Division', 'NewLeague']])
```

[19]:
```
dummies.head()
```

[19]:

	League_A	League_N	Division_E	Division_W	NewLeague_A	NewLeague_N
1	0	1	0	1	0	1
2	1	0	0	1	1	0
3	0	1	1	0	0	1
4	0	1	1	0	0	1
5	1	0	0	1	1	0

Note that for every categorical variable with K categories, we only need K-1 dummies to represent it.

[20]:
```
y = df.Salary

# Drop the column with the dependent variable (Salary), and columns for which␣
↪we created dummy variables
X_ = df.drop(['Salary', 'League', 'Division', 'NewLeague'], axis=1).
↪astype('float64')

# Define the feature set X.
X = pd.concat([X_, dummies[['League_N', 'Division_W', 'NewLeague_N']]], axis=1)
```

We can perform *best subset selection* by identifying the best model that contains a given number of predictors, where **best** is quantified as having the smallest RSS. We'll define a helper function to output the best set of variables for each model size:

[21]:
```
def processSubset(feature_set):
    # Fit model on feature_set and calculate RSS
    model = sm.OLS(y,X[list(feature_set)])
    regr = model.fit()
    RSS = ((regr.predict(X[list(feature_set)]) - y) ** 2).sum()
    return {"model":regr, "RSS":RSS}
```

Here, we calculate the RSS along with the process of model building.

[22]:
```
def getBest(k):

    tic = time.time()
```

```
    results = []

    for combo in itertools.combinations(X.columns, k):
        results.append(processSubset(combo))

    # Wrap everything up in a nice dataframe
    models = pd.DataFrame(results)

    # Choose the model with the smallest RSS
    best_model = models.loc[models['RSS'].idxmin]
    # idxmin() function returns index of first occurrence of minimum.

    toc = time.time()
    print("Processed ", models.shape[0], "models on", k, "predictors in",␣
↪(toc-tic), "seconds.")

    # Return the best model, along with some other useful information about the␣
↪model
    return best_model
```

Note that getting the smallest RSS is the same as getting the highest R^2.

This returns a *DataFrame* containing the best model that we generated, along with the RSS.

What function *itertools.combinations(iterable, r)* does is to return r length subsequences of elements from the input *iterable*. For example:

[23]:
```
print(list(itertools.combinations('12345',2)))
```

```
[('1', '2'), ('1', '3'), ('1', '4'), ('1', '5'), ('2', '3'), ('2', '4'), ('2',
'5'), ('3', '4'), ('3', '5'), ('4', '5')]
```

[24]:
```
print(list(itertools.combinations('12345',3)))
```

```
[('1', '2', '3'), ('1', '2', '4'), ('1', '2', '5'), ('1', '3', '4'), ('1', '3',
'5'), ('1', '4', '5'), ('2', '3', '4'), ('2', '3', '5'), ('2', '4', '5'), ('3',
'4', '5')]
```

Now we want to call that function for each number of predictors k:

[25]:
```
# Could take quite awhile to complete...

models = pd.DataFrame(columns=["RSS", "model"])

tic = time.time()
for i in range(1,8):
    models.loc[i] = getBest(i)

toc = time.time()
```

```
print("Total elapsed time:", (toc-tic), "seconds.")
```

```
Processed  19 models on 1 predictors in 0.1361401081085205 seconds.
Processed  171 models on 2 predictors in 0.3293931484222412 seconds.
Processed  969 models on 3 predictors in 1.890481948852539 seconds.
Processed  3876 models on 4 predictors in 7.917340040206909 seconds.
Processed  11628 models on 5 predictors in 24.284637212753296 seconds.
Processed  27132 models on 6 predictors in 59.575676918029785 seconds.
Processed  50388 models on 7 predictors in 113.07359886169434 seconds.
Total elapsed time: 208.29628586769104 seconds.
```

Now we have one big *DataFrame* that contains the best models of sizes 1 to 7. Note that to save computation time, we did not look at models of sizes 8 to 19.

[26]:
```
models
```

[26]:

	RSS	model
1	4.321393e+07	<statsmodels.regression.linear_model.Regressio…
2	3.073305e+07	<statsmodels.regression.linear_model.Regressio…
3	2.941071e+07	<statsmodels.regression.linear_model.Regressio…
4	2.797678e+07	<statsmodels.regression.linear_model.Regressio…
5	2.718780e+07	<statsmodels.regression.linear_model.Regressio…
6	2.639772e+07	<statsmodels.regression.linear_model.Regressio…
7	2.606413e+07	<statsmodels.regression.linear_model.Regressio…

Theoretically, we need to build the null model as the first step. You can do that separately.

If we want to access the details of each model, no problem! We can get a full rundown of a single model using the *summary()* function:

[27]:
```
print(models.loc[2, "model"].summary())
```

```
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                 Salary   R-squared (uncentered):
0.761
Model:                            OLS   Adj. R-squared (uncentered):
0.760
Method:                 Least Squares   F-statistic:
416.7
Date:                Sat, 11 Apr 2020   Prob (F-statistic):
5.80e-82
Time:                        15:35:45   Log-Likelihood:
-1907.6
No. Observations:                 263   AIC:
3819.
Df Residuals:                     261   BIC:
3826.
```

```
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Hits           2.9538      0.261     11.335      0.000       2.441       3.467
CRBI           0.6788      0.066     10.295      0.000       0.549       0.809
==============================================================================
Omnibus:                      117.551   Durbin-Watson:                   1.933
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              654.612
Skew:                           1.729   Prob(JB):                    7.12e-143
Kurtosis:                       9.912   Cond. No.                         5.88
==============================================================================

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

This output indicates that the best two-variable model contains only *Hits* and *CRBI*.

Here, as a digression, we recall a key difference between *df.loc()* and *df.iloc()*:

loc() gets rows (or columns) with particular *labels* from the index.

iloc() gets rows (or columns) at particular *positions* in the index (so it only takes integers).

For example:

[28]:
```python
data = pd.Series(np.arange(10), index=[20, 21, 22, 23, 24, 1, 2, 3, 4, 5])
data
```

[28]:
```
20    0
21    1
22    2
23    3
24    4
1     5
2     6
3     7
4     8
5     9
dtype: int64
```

[29]:
```python
data.loc[:3]
```

[29]:
```
20    0
21    1
22    2
23    3
24    4
```

```
1    5
2    6
3    7
dtype: int64
```

[30]:
```
data.iloc[:3]
```

[30]:
```
20    0
21    1
22    2
dtype: int64
```

You can use the functions we defined above to explore as many variables as are desired.

[31]:
```
print(getBest(19)["model"].summary())
```

```
Processed  1 models on 19 predictors in 0.011976957321166992 seconds.
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                 Salary   R-squared (uncentered):
0.810
Model:                            OLS   Adj. R-squared (uncentered):
0.795
Method:                 Least Squares   F-statistic:
54.64
Date:                Sat, 11 Apr 2020   Prob (F-statistic):
1.31e-76
Time:                        15:41:34   Log-Likelihood:
-1877.9
No. Observations:                 263   AIC:
3794.
Df Residuals:                     244   BIC:
3862.
Df Model:                          19
Covariance Type:            nonrobust
===============================================================================
                  coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------
AtBat          -1.5975      0.600     -2.663      0.008      -2.779      -0.416
Hits            7.0330      2.374      2.963      0.003       2.357      11.709
HmRun           4.1210      6.229      0.662      0.509      -8.148      16.390
Runs           -2.3776      2.994     -0.794      0.428      -8.276       3.520
RBI            -1.0873      2.613     -0.416      0.678      -6.234       4.059
Walks           6.1560      1.836      3.352      0.001       2.539       9.773
Years           9.5196     10.128      0.940      0.348     -10.429      29.468
CAtBat         -0.2018      0.135     -1.497      0.136      -0.467       0.064
CHits           0.1380      0.678      0.204      0.839      -1.197       1.473
```

```
CHmRun        -0.1669      1.625     -0.103      0.918      -3.367       3.033
CRuns          1.5070      0.753      2.001      0.047       0.023       2.991
CRBI           0.7742      0.696      1.113      0.267      -0.596       2.144
CWalks        -0.7851      0.329     -2.384      0.018      -1.434      -0.137
PutOuts        0.2856      0.078      3.673      0.000       0.132       0.439
Assists        0.3137      0.220      1.427      0.155      -0.119       0.747
Errors        -2.0463      4.350     -0.470      0.638     -10.615       6.522
League_N      86.8139     78.463      1.106      0.270     -67.737     241.365
Division_W   -97.5160     39.084     -2.495      0.013    -174.500     -20.532
NewLeague_N  -23.9133     79.361     -0.301      0.763    -180.234     132.407
==============================================================================
Omnibus:                       97.217   Durbin-Watson:                   2.024
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              626.205
Skew:                           1.320   Prob(JB):                    1.05e-136
Kurtosis:                      10.083   Cond. No.                     2.06e+04
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 2.06e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In addition to using the *summary* function to print to the screen, we can access just the parts we need using the model's attributes. For example, if we want the R^2 value:

[32]:
```python
models.loc[2, "model"].rsquared
```

[32]:
```
0.7614950002332872
```

In addition to the verbose output, we get when we print the summary to the screen, fitting the OLM also produced many other useful statistics such as $adjusted R^2$, AIC, and BIC. We can examine these to try to select the best overall model across different model sizes. Let's start by looking at R^2 across all our models:

[33]:
```python
# Gets the second element from each row ('model') and pulls out its R rsquared
↪attribute
models.apply(lambda row: row[1].rsquared, axis=1)
```

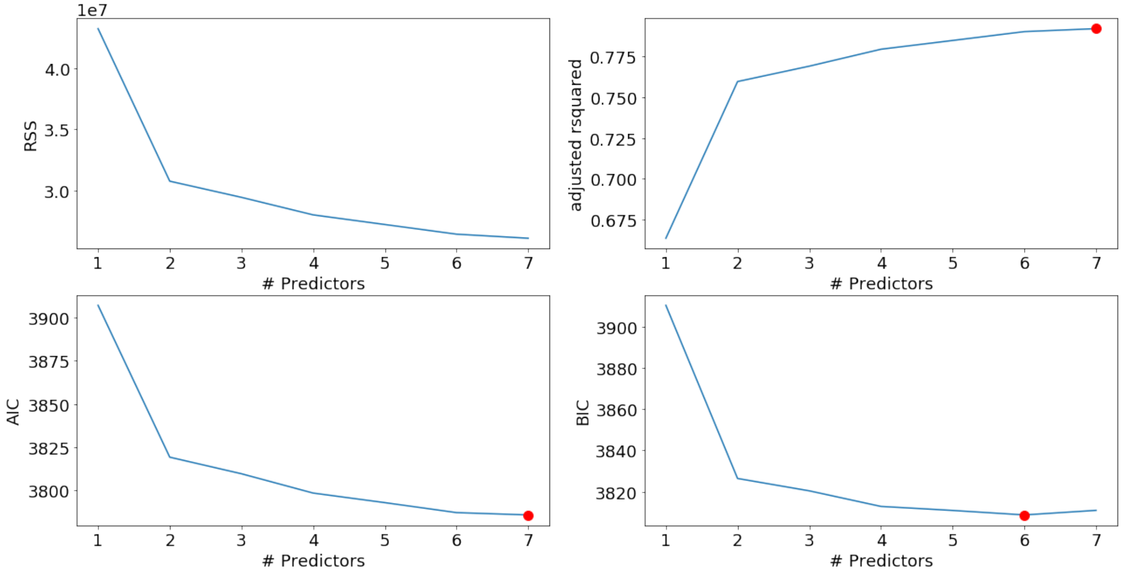
[33]:
```
1    0.664637
2    0.761495
3    0.771757
4    0.782885
5    0.789008
6    0.795140
7    0.797728
dtype: float64
```

As expected, the R^2 statistic increases monotonically as more variables are included.

Plotting RSS, $adjustedR^2$, AIC, and BIC for all of the models at once will help us decide which model to select.:

[34]:
```
plt.figure(figsize=(20,10))
plt.rcParams.update({'font.size': 18, 'lines.markersize': 10})

# Set up a 2x2 grid so we can look at 4 plots at once
plt.subplot(2, 2, 1)

# We will now plot a curve to show the relationship between the number of␣
↪predictors and the RSS
plt.plot(models["RSS"])
plt.xlabel('# Predictors')
plt.ylabel('RSS')

# We will now plot a red dot to indicate the model with the largest adjusted␣
↪R^2 statistic.
# The idxmax() function can be used to identify the location of the maximum␣
↪point of a vector

rsquared_adj = models.apply(lambda row: row[1].rsquared_adj, axis=1)

plt.subplot(2, 2, 2)
plt.plot(rsquared_adj)
plt.plot(rsquared_adj.idxmax(), rsquared_adj.max(), "or")
plt.xlabel('# Predictors')
plt.ylabel('adjusted rsquared')

# We'll do the same for AIC and BIC, this time looking for the models with the␣
↪SMALLEST statistic
aic = models.apply(lambda row: row[1].aic, axis=1)

plt.subplot(2, 2, 3)
plt.plot(aic)
plt.plot(aic.idxmin(), aic.min(), "or")
plt.xlabel('# Predictors')
plt.ylabel('AIC')

bic = models.apply(lambda row: row[1].bic, axis=1)

plt.subplot(2, 2, 4)
plt.plot(bic)
plt.plot(bic.idxmin(), bic.min(), "or")
plt.xlabel('# Predictors')
plt.ylabel('BIC')
```

[34]:
```
Text(0, 0.5, 'BIC')
```



Recall that in the second step of our selection process, we narrowed the field down to just one model of each size $k(\leq p)$. According to BIC, the best performer is the model with 6 variables. According to AIC and $adjustedR^2$ something a bit more complex might be better. Again, no one measure is going to give us an entirely accurate picture, but they all agree that a model with 5 or fewer predictors seems insufficient.

1.2 Forward Stepwise Selection

We can also use a similar approach to perform forward stepwise or backward stepwise selection, using a slight modification of the functions we defined above:

[36]:
```
def forward(predictors):

    # Pull out predictors we still need to process
    remaining_predictors = [p for p in X.columns if p not in predictors]

    tic = time.time()

    results = []

    for p in remaining_predictors:
        results.append(processSubset(predictors+[p]))

    # Wrap everything up in a nice dataframe
    models = pd.DataFrame(results)

    # Choose the model with the highest RSS
    best_model = models.loc[models['RSS'].idxmin]
```

```
    toc = time.time()
    print("Processed ", models.shape[0], "models on", len(predictors)+1,␣
↪"predictors in", (toc-tic), "seconds.")

    # Return the best model, along with some other useful information about the␣
↪model
    return best_model
```

Also, you need to build the null model as the first step but we will skip it in this tutorial. Then you can implement the steps below.

[37]:
```
models2 = pd.DataFrame(columns=["RSS", "model"])

tic = time.time()
predictors = []

for i in range(1,len(X.columns)+1):
    models2.loc[i] = forward(predictors)
    predictors = models2.loc[i]["model"].model.exog_names

toc = time.time()
print("Total elapsed time:", (toc-tic), "seconds.")
```

```
Processed  19 models on 1 predictors in 0.05067086219787598 seconds.
Processed  18 models on 2 predictors in 0.03463411331176758 seconds.
Processed  17 models on 3 predictors in 0.03197193145751953 seconds.
Processed  16 models on 4 predictors in 0.03381085395812988 seconds.
Processed  15 models on 5 predictors in 0.032254934310913086 seconds.
Processed  14 models on 6 predictors in 0.031058788299560547 seconds.
Processed  13 models on 7 predictors in 0.028916120529174805 seconds.
Processed  12 models on 8 predictors in 0.0296628475189209 seconds.
Processed  11 models on 9 predictors in 0.024724960327148438 seconds.
Processed  10 models on 10 predictors in 0.022943258285522246 seconds.
Processed  9 models on 11 predictors in 0.021386146545410156 seconds.
Processed  8 models on 12 predictors in 0.019852638244628906 seconds.
Processed  7 models on 13 predictors in 0.01714777946472168 seconds.
Processed  6 models on 14 predictors in 0.015273809432983398 seconds.
Processed  5 models on 15 predictors in 0.013061285018920898 seconds.
Processed  4 models on 16 predictors in 0.011316061019897461 seconds.
Processed  3 models on 17 predictors in 0.008790969848632812 seconds.
Processed  2 models on 18 predictors in 0.0065479278564453125 seconds.
Processed  1 models on 19 predictors in 0.004437923431396484 seconds.
Total elapsed time: 0.48656487464904785 seconds.
```

Clearly, forward stepwise selection runs faster than best subset selection.

Let's see how the forward stepwise selection models stack up against best subset selection for models with 5 predictors:

[38]:
```python
print("Best Subset Selection:\n", models.loc[5, "model"].summary())
print("\n\nForward Stepwise Selection:\n", models2.loc[5, "model"].summary())
```

```
Best Subset Selection:
                                 OLS Regression Results
================================================================================
=======
Dep. Variable:                 Salary   R-squared (uncentered):
0.789
Model:                            OLS   Adj. R-squared (uncentered):
0.785
Method:                 Least Squares   F-statistic:
193.0
Date:                Sat, 11 Apr 2020   Prob (F-statistic):
5.32e-85
Time:                        16:01:24   Log-Likelihood:
-1891.5
No. Observations:                 263   AIC:
3793.
Df Residuals:                     258   BIC:
3811.
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
AtBat         -1.9281      0.463     -4.166      0.000      -2.839      -1.017
Hits           7.9757      1.596      4.998      0.000       4.833      11.118
Walks          3.9129      1.226      3.191      0.002       1.498       6.328
CRBI           0.6453      0.065      9.961      0.000       0.518       0.773
PutOuts        0.2664      0.076      3.517      0.001       0.117       0.416
==============================================================================
Omnibus:                      108.395   Durbin-Watson:                   2.018
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              754.154
Skew:                           1.480   Prob(JB):                    1.73e-164
Kurtosis:                      10.750   Cond. No.                         56.2
==============================================================================

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


Forward Stepwise Selection:
                                 OLS Regression Results
================================================================================
=======
```

```
Dep. Variable:                 Salary   R-squared (uncentered):
0.788
Model:                            OLS   Adj. R-squared (uncentered):
0.783
Method:                 Least Squares   F-statistic:
191.2
Date:                Sat, 11 Apr 2020   Prob (F-statistic):
1.33e-84
Time:                        16:01:24   Log-Likelihood:
-1892.4
No. Observations:                 263   AIC:
3795.
Df Residuals:                     258   BIC:
3813.
Df Model:                           5
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Hits           6.5426      1.632      4.009      0.000       3.329       9.756
CRBI           0.7011      0.063     11.081      0.000       0.577       0.826
Division_W  -110.0525     38.238     -2.878      0.004    -185.351     -34.754
PutOuts        0.2973      0.076      3.938      0.000       0.149       0.446
AtBat         -1.0915      0.461     -2.368      0.019      -1.999      -0.184
==============================================================================
Omnibus:                      104.548   Durbin-Watson:                   1.984
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              685.581
Skew:                           1.437   Prob(JB):                    1.34e-149
Kurtosis:                      10.369   Cond. No.                     1.28e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.28e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

The results above indicate that the best five-variable models are different between the best subset selection and forward stepwise selection.

1.3 Backward Stepwise Selection

We only need a minor change to implement backward stepwise selection: loop through the predictors in reverse.

[39]:
```python
def backward(predictors):

    tic = time.time()
```

```
    results = []

    for combo in itertools.combinations(predictors, len(predictors)-1):
        results.append(processSubset(combo))

    # Wrap everything up in a nice dataframe
    models = pd.DataFrame(results)

    # Choose the model with the highest RSS
    best_model = models.loc[models['RSS'].idxmin]

    toc = time.time()
    print("Processed ", models.shape[0], "models on", len(predictors)-1,␣
 ↪"predictors in", (toc-tic), "seconds.")

    # Return the best model, along with some other useful information about the␣
 ↪model
    return best_model
```

[40]:
```
models3 = pd.DataFrame(columns=["RSS", "model"], index = range(1,len(X.
 ↪columns)))

tic = time.time()
predictors = X.columns

models3.loc[len(predictors)] = getBest(len(predictors))

while(len(predictors) > 1):
    models3.loc[len(predictors)-1] = backward(predictors)
    predictors = models3.loc[len(predictors)-1]["model"].model.exog_names

toc = time.time()
print("Total elapsed time:", (toc-tic), "seconds.")
```

```
Processed  1 models on 19 predictors in 0.0055119991302490234 seconds.
Processed  19 models on 18 predictors in 0.057672977447509766 seconds.
Processed  18 models on 17 predictors in 0.044020891189575195 seconds.
Processed  17 models on 16 predictors in 0.04086709022521973 seconds.
Processed  16 models on 15 predictors in 0.03754997253417969 seconds.
Processed  15 models on 14 predictors in 0.03711581230163574 seconds.
Processed  14 models on 13 predictors in 0.03434324264526367 seconds.
Processed  13 models on 12 predictors in 0.028679847717285156 seconds.
Processed  12 models on 11 predictors in 0.02761697769165039 seconds.
Processed  11 models on 10 predictors in 0.024317026138305664 seconds.
Processed  10 models on 9 predictors in 0.02207326889038086 seconds.
Processed  9 models on 8 predictors in 0.019916057586669922 seconds.
```

```
Processed  8 models on 7 predictors in 0.017557859420776367 seconds.
Processed  7 models on 6 predictors in 0.015281915664672852 seconds.
Processed  6 models on 5 predictors in 0.01420283317565918 seconds.
Processed  5 models on 4 predictors in 0.010006904602050781 seconds.
Processed  4 models on 3 predictors in 0.009287834167480469 seconds.
Processed  3 models on 2 predictors in 0.00684809684753418 seconds.
Processed  2 models on 1 predictors in 0.004202842712402344 seconds.
Total elapsed time: 0.48042988777160645 seconds.
```

For this data, the best five-variable models identified by *forward stepwise selection*, *backward stepwise selection*, and *best subset selection* are different.

[42]:
```
print("Best Subset Selection:\n",models.loc[5, "model"].params)
print("\nForward Stepwise Selection:\n",models2.loc[5, "model"].params)
print("\nBackward Stepwise Selection:\n",models3.loc[5, "model"].params)
```

```
Best Subset Selection:
 AtBat     -1.928099
Hits       7.975733
Walks      3.912872
CRBI       0.645349
PutOuts    0.266424
dtype: float64

Forward Stepwise Selection:
 Hits             6.542622
CRBI             0.701100
Division_W    -110.052466
PutOuts          0.297317
AtBat           -1.091528
dtype: float64

Backward Stepwise Selection:
 AtBat     -1.899448
Hits       7.754626
Walks      3.687280
CRuns      0.624463
PutOuts    0.301334
dtype: float64
```

2 Shrinkage Methods

[43]:
```
from sklearn.linear_model import Ridge, RidgeCV, Lasso, LassoCV
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
```

2.1 Ridge Regression

The *Ridge()* function has an *alpha* argument (same as λ in the lecture slides, but with a different name!) that is used to tune the model. We'll generate an array of alpha values ranging from very large to very small, essentially covering the full range of scenarios from (close to) the null model containing only the intercept, to the *least squares* fit:

[45]:
```
alphas = 10**np.linspace(10,-2,100)*0.5
alphas
```

[45]:
```
array([5.00000000e+09, 3.78231664e+09, 2.86118383e+09, 2.16438064e+09,
       1.63727458e+09, 1.23853818e+09, 9.36908711e+08, 7.08737081e+08,
       5.36133611e+08, 4.05565415e+08, 3.06795364e+08, 2.32079442e+08,
       1.75559587e+08, 1.32804389e+08, 1.00461650e+08, 7.59955541e+07,
       5.74878498e+07, 4.34874501e+07, 3.28966612e+07, 2.48851178e+07,
       1.88246790e+07, 1.42401793e+07, 1.07721735e+07, 8.14875417e+06,
       6.16423370e+06, 4.66301673e+06, 3.52740116e+06, 2.66834962e+06,
       2.01850863e+06, 1.52692775e+06, 1.15506485e+06, 8.73764200e+05,
       6.60970574e+05, 5.00000000e+05, 3.78231664e+05, 2.86118383e+05,
       2.16438064e+05, 1.63727458e+05, 1.23853818e+05, 9.36908711e+04,
       7.08737081e+04, 5.36133611e+04, 4.05565415e+04, 3.06795364e+04,
       2.32079442e+04, 1.75559587e+04, 1.32804389e+04, 1.00461650e+04,
       7.59955541e+03, 5.74878498e+03, 4.34874501e+03, 3.28966612e+03,
       2.48851178e+03, 1.88246790e+03, 1.42401793e+03, 1.07721735e+03,
       8.14875417e+02, 6.16423370e+02, 4.66301673e+02, 3.52740116e+02,
       2.66834962e+02, 2.01850863e+02, 1.52692775e+02, 1.15506485e+02,
       8.73764200e+01, 6.60970574e+01, 5.00000000e+01, 3.78231664e+01,
       2.86118383e+01, 2.16438064e+01, 1.63727458e+01, 1.23853818e+01,
       9.36908711e+00, 7.08737081e+00, 5.36133611e+00, 4.05565415e+00,
       3.06795364e+00, 2.32079442e+00, 1.75559587e+00, 1.32804389e+00,
       1.00461650e+00, 7.59955541e-01, 5.74878498e-01, 4.34874501e-01,
       3.28966612e-01, 2.48851178e-01, 1.88246790e-01, 1.42401793e-01,
       1.07721735e-01, 8.14875417e-02, 6.16423370e-02, 4.66301673e-02,
       3.52740116e-02, 2.66834962e-02, 2.01850863e-02, 1.52692775e-02,
       1.15506485e-02, 8.73764200e-03, 6.60970574e-03, 5.00000000e-03])
```

Function *numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0)* returns *num* **evenly** spaced numbers, calculated over the interval *[start, stop]*. For example:

[32]:
```
np.linspace(10,1,19)
```

[32]:
```
array([10. ,  9.5,  9. ,  8.5,  8. ,  7.5,  7. ,  6.5,  6. ,  5.5,  5. ,
        4.5,  4. ,  3.5,  3. ,  2.5,  2. ,  1.5,  1. ])
```

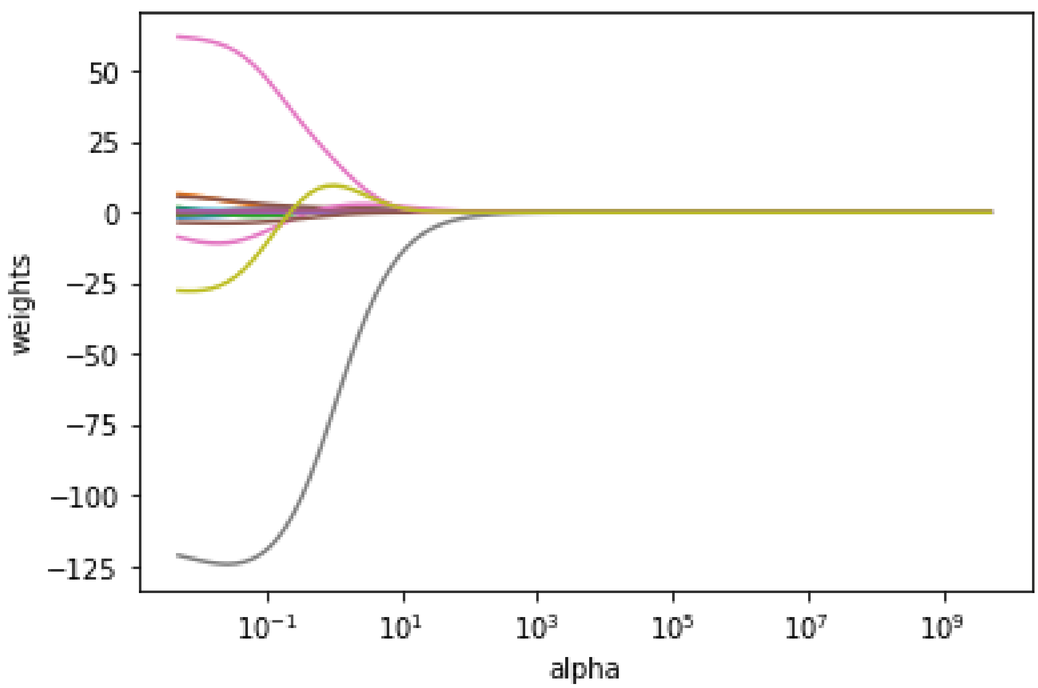
Associated with each alpha value is a vector of ridge regression coefficients, which we'll store in a matrix *coefs*. In this case, it is a 100×19 matrix, with 19 columns (one for each predictor) and 100 rows (one for each value of alpha). Remember that we'll want to standardize the predictors for Ridge regression so that they are on the same scale. To do this, we can use the *normalize=True* parameter:

[47]:
```
ridge = Ridge(normalize=True)
coefs = []

for a in alphas:
    ridge.set_params(alpha=a)
    ridge.fit(X, y)
    coefs.append(ridge.coef_)

np.shape(coefs)
```

[47]: (100, 19)

[51]:
```
%matplotlib inline
ax = plt.gca() # Get the current Axes instance
ax.plot(alphas, coefs)
ax.set_xscale('log')  ## you can try removing this line and see what it looks␣
↪like
plt.xlabel('alpha')
plt.ylabel('weights')
```

[51]: Text(0, 0.5, 'weights')



We now split the data into a training set and a test set in order to estimate the test error of ridge regression.

[53]:
```
# Use the train_test_split function to split data into training and test sets
X_train, X_test , y_train, y_test = train_test_split(X, y, test_size=0.5,␣
↪random_state=1)
```

Next, we fit a ridge regression model on the training set, and evaluate its MSE on the test set, using λ (i.e., alpha)$=4$:

[54]:
```
ridge2 = Ridge(alpha=4, normalize=True)
ridge2.fit(X_train, y_train)  # Fit a ridge regression on the training data
pred2 = ridge2.predict(X_test) # Use trained model to predict on the test data
print(pd.Series(ridge2.coef_, index=X.columns)) # Print coefficients
print("\nmean_squared_error: ",mean_squared_error(y_test, pred2)) # Calculate␣
↪the test MSE
```

```
AtBat           0.098658
Hits            0.446094
HmRun           1.412107
Runs            0.660773
RBI             0.843403
Walks           1.008473
Years           2.779882
CAtBat          0.008244
CHits           0.034149
CHmRun          0.268634
CRuns           0.070407
CRBI            0.070060
CWalks          0.082795
PutOuts         0.104747
Assists        -0.003739
Errors          0.268363
League_N        4.241051
Division_W    -30.768885
NewLeague_N     4.123474
dtype: float64

mean_squared_error:  106216.52238005563
```

The test MSE when alpha $=4$ is about 106216. Now let's see what will happen if we use a huge value of alpha, say 10^{10}:

[37]:
```
ridge3 = Ridge(alpha=10**10, normalize=True)
ridge3.fit(X_train, y_train) # Fit a ridge regression on the training data
pred3 = ridge3.predict(X_test) # Use this model to predict the test data
print(pd.Series(ridge3.coef_, index=X.columns)) # Print coefficients
print("\nmean_squared_error: ",mean_squared_error(y_test, pred3)) # Calculate␣
↪the test MSE
```

```
AtBat           1.317464e-10
Hits            4.647486e-10
HmRun           2.079865e-09
Runs            7.726175e-10
RBI             9.390640e-10
Walks           9.769219e-10
Years           3.961442e-09
CAtBat          1.060533e-11
CHits           3.993605e-11
CHmRun          2.959428e-10
CRuns           8.245247e-11
CRBI            7.795451e-11
CWalks          9.894387e-11
PutOuts         7.268991e-11
Assists        -2.615885e-12
Errors          2.084514e-10
League_N       -2.501281e-09
Division_W     -1.549951e-08
NewLeague_N    -2.023196e-09
dtype: float64

mean_squared_error:  172862.23580379886
```

This huge penalty shrinks the coefficients by a large degree, essentially reducing to a model containing just the intercept.

Fitting a ridge regression model with alpha = 4 leads to a much lower test MSE than fitting a model with just an intercept. We now check whether there is any benefit to performing ridge regression with alpha = 4 instead of just performing least-squares regression. Recall that *least squares* is simply ridge regression with alpha = 0.

[38]:
```
ridge2 = Ridge(alpha=0, normalize=True)
ridge2.fit(X_train, y_train) # Fit a ridge regression on the training data
pred = ridge2.predict(X_test) # Use this model to predict the test data
print(pd.Series(ridge2.coef_, index=X.columns)) # Print coefficients
print("\nmean_squared_error: ",mean_squared_error(y_test, pred)) # Calculate␣
 ↪the test MSE
```

```
AtBat          -1.821115
Hits            4.259156
HmRun          -4.773401
Runs           -0.038760
RBI             3.984578
Walks           3.470126
Years           9.498236
CAtBat         -0.605129
CHits           2.174979
CHmRun          2.979306
CRuns           0.266356
```

```
CRBI             -0.598456
CWalks            0.171383
PutOuts           0.421063
Assists           0.464379
Errors           -6.024576
League_N        133.743163
Division_W     -113.743875
NewLeague_N     -81.927763
dtype: float64

mean_squared_error:  116690.4685666044
```

It looks like we are indeed improving over *least squares*.

Instead of arbitrarily choosing alpha = 4, it is better to use cross-validation to choose the tuning parameter alpha. We can do this using the cross-validated ridge regression function, *RidgeCV()*. We set *cv = 10* to perform 10-fold cross-validation.

[55]:
```
ridgecv = RidgeCV(alphas=alphas, scoring='neg_mean_squared_error',␣
 ↪normalize=True, cv=10)
ridgecv.fit(X_train, y_train)
ridgecv.alpha_
```

[55]:
```
0.5748784976988678
```

Hence, the value of alpha that results in the smallest cross-validation error is 0.5749. Let's see the test MSE associated with this value of alpha

[56]:
```
ridge4 = Ridge(alpha=ridgecv.alpha_, normalize=True)
ridge4.fit(X_train, y_train)
mean_squared_error(y_test, ridge4.predict(X_test))
```

[56]:
```
99825.64896292728
```

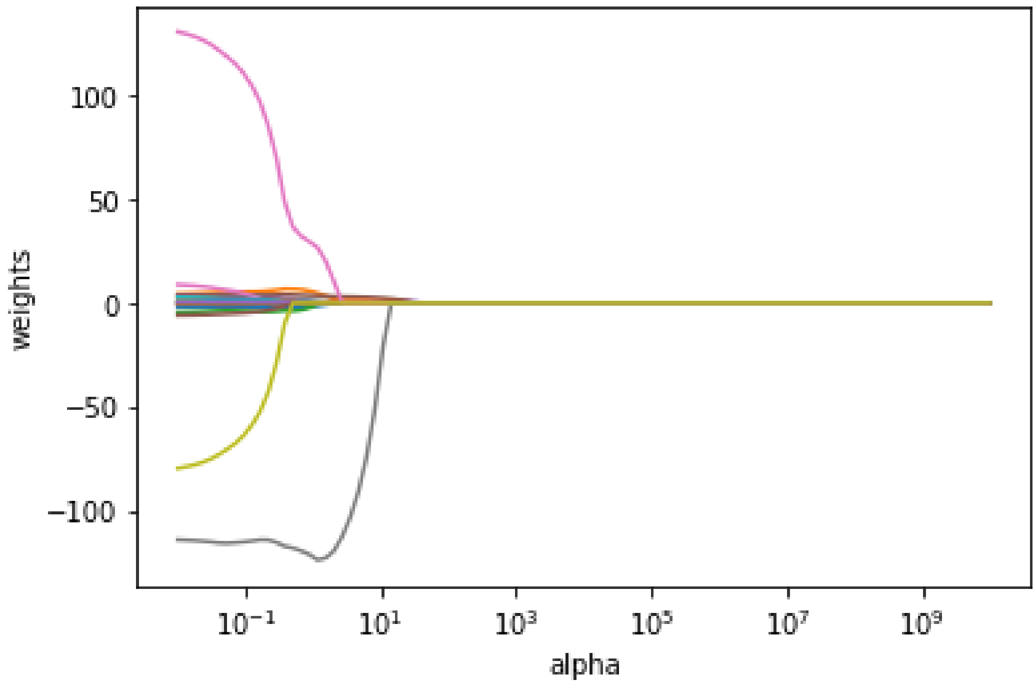
This represents a further improvement over the test MSE that we got using alpha=4.

2.2 The Lasso

[58]:
```
lasso = Lasso(max_iter=10000, normalize=True)
coefs = []

for a in alphas:
    lasso.set_params(alpha=a)
    lasso.fit(X_train, y_train)
    coefs.append(lasso.coef_)

%matplotlib inline
ax = plt.gca() # Get the current Axes instance
ax.plot(alphas*2, coefs)
```

```
ax.set_xscale('log')
plt.xlabel('alpha')
plt.ylabel('weights')
```

[58]: Text(0, 0.5, 'weights')



[59]:
```
lassocv = LassoCV(alphas=alphas, cv=10, max_iter=3000, normalize=True)
lassocv.fit(X_train, y_train)

lasso.set_params(alpha=lassocv.alpha_)
lasso.fit(X_train, y_train)
mean_squared_error(y_test, lasso.predict(X_test))
```

[59]: 104904.36377748138

This is substantially lower than the test set MSE of the null model and of least squares, and only a little worse than the test MSE of ridge regression with alpha chosen by cross-validation.

However, from a model interpretation point of view, the lasso has a substantial advantage over ridge regression in that the resulting coefficient estimates are sparse. In this example, 13 of the 19 lasso coefficient estimates are exactly zero:

[61]:
```
# Some of the coefficients are now reduced to exactly zero.
pd.Series(lasso.coef_, index=X.columns)
```

[61]:
```
AtBat            0.000000
Hits             1.089755
HmRun            0.000000
Runs             0.000000
RBI              0.000000
Walks            2.921569
Years            0.000000
CAtBat           0.000000
CHits            0.000000
CHmRun           0.223579
CRuns            0.000000
CRBI             0.515025
CWalks           0.000000
PutOuts          0.369934
Assists         -0.000000
Errors          -0.000000
League_N         0.000000
Division_W     -90.878889
NewLeague_N      0.000000
dtype: float64
```

2.3 Logistic Regression with Penalty

Similary to the regression, we can add the penalty (thrinkage) term when we use logistic regression.

Here, we use the dataset *caravan* for demonstration. This data set includes 85 predictors that measure demographic characteristics for 5,822 individuals. The response variable is Purchase, which indicates whether or not a given individual purchases a caravan insurance policy. In this data set, only 6% of people purchased caravan insurance.

[62]:
```python
import pandas as pd
import numpy as np
caravan = pd.read_csv('caravan.csv')
caravan.head()
```

[62]:
```
   MOSTYPE  MAANTHUI  MGEMOMV  MGEMLEEF  MOSHOOFD  MGODRK  MGODPR  MGODOV  \
0       33         1        3         2         8       0       5       1
1       37         1        2         2         8       1       4       1
2       37         1        2         2         8       0       4       2
3        9         1        3         3         3       2       3       2
4       40         1        4         2        10       1       4       1

   MGODGE  MRELGE  …  APERSONG  AGEZONG  AWAOREG  ABRAND  AZEILPL  APLEZIER  \
0       3       7  …         0        0        0       1        0         0
1       4       6  …         0        0        0       1        0         0
2       4       3  …         0        0        0       1        0         0
3       4       5  …         0        0        0       1        0         0
4       4       7  …         0        0        0       1        0         0
```

```
   AFIETS  AINBOED  ABYSTAND  Purchase
0       0        0         0        No
1       0        0         0        No
2       0        0         0        No
3       0        0         0        No
4       0        0         0        No

[5 rows x 86 columns]
```

[63]:
```
caravan.shape
```

[63]:
```
(5822, 86)
```

We use *Purchase* as the response variable and the others as the predictor variables.

[64]:
```
X = caravan.drop(['Purchase'], axis=1).astype('float64')
y = caravan['Purchase']
```

[65]:
```
from sklearn.model_selection import train_test_split
X_train, X_test , y_train, y_test = train_test_split(X, y, test_size=0.5,␣
 ↪random_state=1)
```

[66]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV
```

First, we use *LogisticRegression* and set *penalty='none'* to the logistic regression without penalty.

Note that the default values of *penalty* parameter is *"l2"* in *LogisticRegression* of *sklearn*.

[67]:
```
fit1 = LogisticRegression(random_state=1, penalty='none').fit(X_train, y_train)
fit1.score(X_test, y_test)
```

```
/Users/xintong/anaconda3/lib/python3.7/site-
packages/sklearn/linear_model/_logistic.py:940: 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
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

[67]:
```
0.9374785297148747
```

It reports that "lbfgs failed to converge" and "TOTAL NO. of ITERATIONS REACHED LIMIT"; thus we modify the *max_iter* to *5000* to check whether the algorithm coverges.

[68]:
```
fit1 = LogisticRegression(random_state=1, penalty='none', max_iter=5000).
↪fit(X_train, y_train)
fit1.score(X_test, y_test)
```

[68]: 0.9350738577808313

Then we do the logistic regression with the penalty and use cross-validation to pick up the best *alpha*. We use *LogisticRegressionCV* function from *sklearn* to choose the thrinkage level by cross-validation and fit penalized logistic regression with the chosen penalty (i.e., thrinkage) level.

The *LogisticRegressionCV* function has a parameter named *Cs*. Each of the values in *Cs* describes the inverse of regularization strength which is similar to s we talked about in lectures. If *Cs* is an int, then a grid of Cs values are automatically generated in a logarithmic scale between 1e-4 (i.e., 10^{-4}) and 1e4 (i.e., 10^4).

We also need to set the *cv* parameter. If we set *cv* to *5*, it means that we will do 5-fold cross-validation to pick up the best C in *Cs*.

Logistic Regression with L1 penalty:

[72]:
```
fit2 = LogisticRegressionCV(Cs=20,random_state=1,␣
↪penalty='l1',solver='liblinear',cv=5).fit(X_train, y_train)
fit2.score(X_test, y_test)
```

[72]: 0.9395396770869117

Note that according to the description of the function, the default *solver* of *LogisticRegressionCV* is *'lbfgs'* which supports only 'l2' or 'none' penalties. If we want to use 'l1' penalty, we have to set *solver* to *'liblinear'* or *'saga'*.

In the above, we set *Cs=20*. We can check that *LogisticRegressionCV* generates a grid of 20 values are chosen in a logarithmic scale between 1e-4 and 1e4.

[73]:
```
fit2.Cs_
```

[73]:
```
array([1.00000000e-04, 2.63665090e-04, 6.95192796e-04, 1.83298071e-03,
       4.83293024e-03, 1.27427499e-02, 3.35981829e-02, 8.85866790e-02,
       2.33572147e-01, 6.15848211e-01, 1.62377674e+00, 4.28133240e+00,
       1.12883789e+01, 2.97635144e+01, 7.84759970e+01, 2.06913808e+02,
       5.45559478e+02, 1.43844989e+03, 3.79269019e+03, 1.00000000e+04])
```

And the model uses 5-fold cross-validation to select the best C which gives us the highest score in *Cs*.

[74]:
```
fit2.C_
```

[74]: array([0.0001])

Logistic Regression with L2 penalty:

[75]:

```
fit3 = LogisticRegressionCV(Cs=20,random_state=1,␣
 ↪penalty='l2',cv=5,max_iter=10000).fit(X_train, y_train)
fit3.score(X_test, y_test)
```

[75]: 0.9395396770869117

As for the *caravan* dataset, we can draw a conclusion that in terms of prediction accuracy (i.e., 1-classification error), the logistic models with L1 or L2 penalty only improve a little bit compared with the logistic model without penalty.

References:

https://github.com/jcrouser/islr-python

https://docs.python.org/2/library/itertools.html#itertools.combinations

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html

https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model