# Python Tutorial 3

February 12, 2020

This tutorial is for Prof. 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 2b: Multiple Linear Regression* and the code which is corresponding to *Lecture 3a: Classification I* to teach you how to implement logistic regression using python.

## 1 Supplementary Part of Multiple Linear Regression

We still use the same Boston dataset.

[1]:
```
from sklearn.datasets import load_boston
boston_dataset = load_boston()
print(boston_dataset.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value
(attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000
sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0
otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
```

```
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by
town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie
Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
…', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that
address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential
Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In
Proceedings on the Tenth International Conference of Machine Learning, 236-243,
University of Massachusetts, Amherst. Morgan Kaufmann.
```

[2]:
```python
import pandas as pd

boston = pd.DataFrame(boston_dataset.data, columns=boston_dataset.feature_names)
boston['MEDV'] = boston_dataset.target
boston.head()
```

[2]:
```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
```

```
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0

   PTRATIO       B  LSTAT  MEDV
0     15.3  396.90   4.98  24.0
1     17.8  396.90   9.14  21.6
2     17.8  392.83   4.03  34.7
3     18.7  394.63   2.94  33.4
4     18.7  396.90   5.33  36.2
```

### 1.1 Simple Linear Regression

We will start by using the *smf.ols()* function to fit a simple linear regression model, with *MEDV* as the response and *LSTAT* as the predictor. The basic syntax is *smf.ols('y ~ x', data)*,where $y$ is the response, $x$ is the predictor, and data is the *data* set in which these two variables are kept.

P.S. *smf.ols()* function takes in data as *pandas DataFrames* as opposed to *numpy array*.

[3]:
```python
import statsmodels.formula.api as smf

result1 = smf.ols('MEDV ~ LSTAT', data=boston).fit()
```

We use *results.summary()* to output some detailed imformaton about the model.

[4]:
```python
result1.summary()
```

[4]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                   MEDV   R-squared:                       0.544
Model:                            OLS   Adj. R-squared:                  0.543
Method:                 Least Squares   F-statistic:                     601.6
Date:                Mon, 10 Feb 2020   Prob (F-statistic):           5.08e-88
Time:                        19:21:04   Log-Likelihood:                -1641.5
No. Observations:                 506   AIC:                             3287.
Df Residuals:                     504   BIC:                             3295.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     34.5538      0.563     61.415      0.000      33.448      35.659
LSTAT         -0.9500      0.039    -24.528      0.000      -1.026      -0.874
==============================================================================
Omnibus:                      137.043   Durbin-Watson:                   0.892
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              291.373
Skew:                           1.453   Prob(JB):                     5.36e-64
Kurtosis:                       5.319   Cond. No.                         29.7
```

```
==============================================================================

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

### 1.2 Multiple Regression

In order to fit a multiple linear regression model using least squares, we again use the *smf.ols()* function. The syntax *smf.ols('y ~ x1+x2+x3', data)* is used to fit a model with three predictors, *x1*, *x2*,and *x3*. The *summary()* function now outputs the regression coefficients for all the predictors.

[5]:
```
result2 = smf.ols('MEDV ~ LSTAT+AGE', data=boston).fit()
result2.summary()
```

[5]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                   MEDV   R-squared:                       0.551
Model:                            OLS   Adj. R-squared:                  0.549
Method:                 Least Squares   F-statistic:                     309.0
Date:                Mon, 10 Feb 2020   Prob (F-statistic):           2.98e-88
Time:                        19:21:04   Log-Likelihood:                -1637.5
No. Observations:                 506   AIC:                             3281.
Df Residuals:                     503   BIC:                             3294.
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     33.2228      0.731     45.458      0.000      31.787      34.659
LSTAT         -1.0321      0.048    -21.416      0.000      -1.127      -0.937
AGE            0.0345      0.012      2.826      0.005       0.011       0.059
==============================================================================
Omnibus:                      124.288   Durbin-Watson:                   0.945
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              244.026
Skew:                           1.362   Prob(JB):                     1.02e-53
Kurtosis:                       5.038   Cond. No.                         201.
==============================================================================

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

The Boston data set contains 13 variables, and so it would be cumbersome to have to type all of these in order to perform a regression using all of the predictors. Instead, we can use the following short-hand:

[6]:
```
string_cols = ' + '.join(boston.columns[:-1])
result3 = smf.ols('MEDV ~ {}'.format(string_cols), data=boston).fit()
result3.summary()
```

[6]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                   MEDV   R-squared:                       0.741
Model:                            OLS   Adj. R-squared:                  0.734
Method:                 Least Squares   F-statistic:                     108.1
Date:                Mon, 10 Feb 2020   Prob (F-statistic):          6.72e-135
Time:                        19:21:04   Log-Likelihood:                -1498.8
No. Observations:                 506   AIC:                             3026.
Df Residuals:                     492   BIC:                             3085.
Df Model:                          13
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     36.4595      5.103      7.144      0.000      26.432      46.487
CRIM          -0.1080      0.033     -3.287      0.001      -0.173      -0.043
ZN             0.0464      0.014      3.382      0.001       0.019       0.073
INDUS          0.0206      0.061      0.334      0.738      -0.100       0.141
CHAS           2.6867      0.862      3.118      0.002       0.994       4.380
NOX          -17.7666      3.820     -4.651      0.000     -25.272     -10.262
RM             3.8099      0.418      9.116      0.000       2.989       4.631
AGE            0.0007      0.013      0.052      0.958      -0.025       0.027
DIS           -1.4756      0.199     -7.398      0.000      -1.867      -1.084
RAD            0.3060      0.066      4.613      0.000       0.176       0.436
TAX           -0.0123      0.004     -3.280      0.001      -0.020      -0.005
PTRATIO       -0.9527      0.131     -7.283      0.000      -1.210      -0.696
B              0.0093      0.003      3.467      0.001       0.004       0.015
LSTAT         -0.5248      0.051    -10.347      0.000      -0.624      -0.425
==============================================================================
Omnibus:                      178.041   Durbin-Watson:                   1.078
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              783.126
Skew:                           1.521   Prob(JB):                    8.84e-171
Kurtosis:                       8.281   Cond. No.                     1.51e+04
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.51e+04. This might indicate that there are
```

```
strong multicollinearity or other numerical problems.
"""
```

We could see that here we use *string_cols = ' + '.join(boston.columns[:-1])* to get all the variable with correct format which would be used in *smf.ols()* except the target *MEDV*.

*str.format()* is one of the string formatting methods in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting. If you never use that before, you can see more details on the following webpage: https://www.geeksforgeeks.org/python-format-function/

[7]:
```python
print(string_cols)
```

```
CRIM + ZN + INDUS + CHAS + NOX + RM + AGE + DIS + RAD + TAX + PTRATIO + B +
LSTAT
```

What if we would like to perform a regression using all of the variables but one? For example, in the above regression output, *age* has a high p-value. So we may wish to run a regression excluding this predictor. The following syntax results in a regression using all predictors except *age*.

[8]:
```python
string_cols = ' + '.join(boston.columns[:-1].difference(['AGE']))
result4 = smf.ols('MEDV ~ {}'.format(string_cols), data=boston).fit()
result4.summary()
```

[8]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                   MEDV   R-squared:                       0.741
Model:                            OLS   Adj. R-squared:                  0.734
Method:                 Least Squares   F-statistic:                     117.3
Date:                Mon, 10 Feb 2020   Prob (F-statistic):          6.08e-136
Time:                        19:21:04   Log-Likelihood:                -1498.8
No. Observations:                 506   AIC:                             3024.
Df Residuals:                     493   BIC:                             3079.
Df Model:                          12
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     36.4369      5.080      7.172      0.000      26.456      46.418
B              0.0093      0.003      3.481      0.001       0.004       0.015
CHAS           2.6890      0.860      3.128      0.002       1.000       4.378
CRIM          -0.1080      0.033     -3.290      0.001      -0.173      -0.043
DIS           -1.4786      0.191     -7.757      0.000      -1.853      -1.104
INDUS          0.0206      0.061      0.335      0.738      -0.100       0.141
LSTAT         -0.5239      0.048    -10.999      0.000      -0.617      -0.430
NOX          -17.7135      3.679     -4.814      0.000     -24.943     -10.484
PTRATIO       -0.9522      0.130     -7.308      0.000      -1.208      -0.696
```

```
RAD              0.3058      0.066      4.627      0.000       0.176       0.436
RM               3.8144      0.408      9.338      0.000       3.012       4.617
TAX             -0.0123      0.004     -3.283      0.001      -0.020      -0.005
ZN               0.0463      0.014      3.404      0.001       0.020       0.073
==============================================================================
Omnibus:                      178.343   Durbin-Watson:                   1.078
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              786.386
Skew:                           1.523   Prob(JB):                    1.73e-171
Kurtosis:                       8.294   Cond. No.                     1.48e+04
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.48e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

Here, we use *difference* to exclude *AGE*. The function *difference()* returns a set that is the difference between two sets. For example, if $A = \{100, 60\}$ and $B = \{60, 20\}$. Then $A.difference(B) = \{100\}$ and $B.difference(A) = \{20\}$.

[9]:
```
string_cols = ' + '.join(boston.columns[:-1].difference(['AGE']))
print(string_cols)
```

```
B + CHAS + CRIM + DIS + INDUS + LSTAT + NOX + PTRATIO + RAD + RM + TAX + ZN
```

### 1.3 Interaction Terms

It is easy to include interaction terms in a linear model using the *smf.ols()* function. The syntax *x1:x2* tells Python to include an interaction term between *x1* and *x2*.The syntax *LSTAT*AGE* simultaneously includes *LSTAT*, *AGE*, and the interaction term $LSTAT{\times}AGE$ as predictors; it is a shorthand for *LSTAT+AGE+LSTAT:AGE*.

[10]:
```
result4 = smf.ols('MEDV ~ LSTAT * AGE', data=boston).fit() # is same as:
↪result4 = smf.ols('MEDV ~ LSTAT+AGE+LSTAT:AGE', data=boston).fit()
result4.summary()
```

[10]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                   MEDV   R-squared:                       0.556
Model:                            OLS   Adj. R-squared:                  0.553
Method:                 Least Squares   F-statistic:                     209.3
Date:                Mon, 10 Feb 2020   Prob (F-statistic):           4.86e-88
Time:                        19:21:04   Log-Likelihood:                -1635.0
No. Observations:                 506   AIC:                             3278.
```

```
Df Residuals:                     502   BIC:                             3295.
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     36.0885      1.470     24.553      0.000      33.201      38.976
LSTAT         -1.3921      0.167     -8.313      0.000      -1.721      -1.063
AGE           -0.0007      0.020     -0.036      0.971      -0.040       0.038
LSTAT:AGE      0.0042      0.002      2.244      0.025       0.001       0.008
==============================================================================
Omnibus:                      135.601   Durbin-Watson:                   0.965
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              296.955
Skew:                           1.417   Prob(JB):                     3.29e-65
Kurtosis:                       5.461   Cond. No.                     6.88e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 6.88e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

### 1.4 Non-linear Transformations of the Predictors

The *smf.ols()* function can also accommodate non-linear transformations of the predictors. For instance, given a predictor $X$, we can create a predictor $X^2$ using *np.power(LSTAT, 2)*. We now perform a regression of $MEDV$ onto $LSTAT$ and $LSTAT^2$.

[11]:
```python
import numpy as np
result5 = smf.ols('MEDV ~ LSTAT + np.power(LSTAT, 2)', data=boston).fit()
result5.summary()
```

[11]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                   MEDV   R-squared:                       0.641
Model:                            OLS   Adj. R-squared:                  0.639
Method:                 Least Squares   F-statistic:                     448.5
Date:                Mon, 10 Feb 2020   Prob (F-statistic):          1.56e-112
Time:                        19:21:04   Log-Likelihood:                -1581.3
No. Observations:                 506   AIC:                             3169.
Df Residuals:                     503   BIC:                             3181.
Df Model:                           2
Covariance Type:            nonrobust
```

```
==============================================================================
======
                        coef    std err          t      P>|t|      [0.025
0.975]
------------------------------------------------------------------------------
------
Intercept            42.8620      0.872     49.149      0.000      41.149
44.575
LSTAT                -2.3328      0.124    -18.843      0.000      -2.576
-2.090
np.power(LSTAT, 2)    0.0435      0.004     11.628      0.000       0.036
0.051
==============================================================================
Omnibus:                      107.006   Durbin-Watson:                   0.921
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              228.388
Skew:                           1.128   Prob(JB):                     2.55e-50
Kurtosis:                       5.397   Cond. No.                     1.13e+03
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.13e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

## 1.5 Confidence Interval and Prediction Inverval

Here we'd like to talk about how to code to calculate **Confidence Interval** and **Prediction Interval** of the response, which are important concepts mentioned on page 82 of our textbook *ISLR*. The point is that the confidence interval is about an average response and the prediction interval is about a particular response. Note that it is a slight abuse of language to name an interval prediction of the response CI here. But we will resolve the conflict at the end of this section.

We can read off the confidence intervals for the coefficient estimates in *summary()*:

[12]:
```python
result1 = smf.ols('MEDV ~ LSTAT', data=boston).fit()
result1.summary()
```

[12]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                   MEDV   R-squared:                       0.544
Model:                            OLS   Adj. R-squared:                  0.543
Method:                 Least Squares   F-statistic:                     601.6
Date:                Mon, 10 Feb 2020   Prob (F-statistic):           5.08e-88
Time:                        19:21:04   Log-Likelihood:                -1641.5
```

```
No. Observations:                 506   AIC:                             3287.
Df Residuals:                     504   BIC:                             3295.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     34.5538      0.563     61.415      0.000      33.448      35.659
LSTAT         -0.9500      0.039    -24.528      0.000      -1.026      -0.874
==============================================================================
Omnibus:                      137.043   Durbin-Watson:                   0.892
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              291.373
Skew:                           1.453   Prob(JB):                     5.36e-64
Kurtosis:                       5.319   Cond. No.                         29.7
==============================================================================

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

If we want to produce confidence intervals and prediction intervals for the prediction of *MEDV* for a given value of *LSTAT*, we can do as follows:

Assume that we want predict for the following given value of *LSTAT*:

[13]:
```python
test_data = {'LSTAT':[5,10,15]}
test_data_df = pd.DataFrame(test_data)
test_data_df
```

[13]:
```
   LSTAT
0      5
1     10
2     15
```

We can use *get_prediction()* function to produce confidence intervals and prediction intervals for the prediction.

[14]:
```python
prediction1 = result1.get_prediction(test_data_df)
prediction1.summary_frame(alpha=0.05)
```

[14]:
```
        mean   mean_se  mean_ci_lower  mean_ci_upper  obs_ci_lower  \
0  29.803594  0.405247      29.007412      30.599776     17.565675
1  25.053347  0.294814      24.474132      25.632563     12.827626
2  20.303101  0.290893      19.731588      20.874613      8.077742

   obs_ci_upper
0     42.041513
```

```
1      37.279068
2      32.528459
```

For instance, the 95% confidence interval associated with a *LSTAT* value of 10 is (24.474132, 25.632563), and the 95% prediction interval is (12.827626, 37.279068). As expected, the confidence and prediction intervals are centered around the same point (a predicted value of 25.053347 for *MEDV* when *LSTAT* equals 10), but the latter are substantially wider.

P.S. The confidence interval here, in this case, is the confidence interval of $\beta_0 + 10\beta_1$.

## 2 Logistic Regression

We will begin by examining some numerical and graphical summaries of the *Smarket* data, which is part of the *ISLR* library in R and downloaded as a csv file. This data set consists of percentage returns for the S&P 500 stock index over 1, 250 days, from the beginning of 2001 until the end of 2005. For each date, we have recorded the percentage returns for each of the five previous trading days, *Lag1* through *Lag5*. We have also recorded *Volume* (the number of shares traded on the previous day, in billions), *Today* (the percentage return on the date in question) and *Direction* (whether the market was *Up* or *Down* on this date). The *Direction* column can be inferred from the *Today* column.

[15]:
```
smarket = pd.read_csv('smarket.csv')
smarket.head()
```

[15]:
```
   Year   Lag1   Lag2   Lag3   Lag4   Lag5  Volume  Today Direction
0  2001  0.381 -0.192 -2.624 -1.055  5.010  1.1913  0.959        Up
1  2001  0.959  0.381 -0.192 -2.624 -1.055  1.2965  1.032        Up
2  2001  1.032  0.959  0.381 -0.192 -2.624  1.4112 -0.623      Down
3  2001 -0.623  1.032  0.959  0.381 -0.192  1.2760  0.614        Up
4  2001  0.614 -0.623  1.032  0.959  0.381  1.2057  0.213        Up
```

[16]:
```
smarket.describe()
```

[16]:
```
              Year        Lag1        Lag2        Lag3        Lag4  \
count  1250.000000  1250.000000  1250.000000  1250.000000  1250.000000
mean   2003.016000     0.003834     0.003919     0.001716     0.001636
std       1.409018     1.136299     1.136280     1.138703     1.138774
min    2001.000000    -4.922000    -4.922000    -4.922000    -4.922000
25%    2002.000000    -0.639500    -0.639500    -0.640000    -0.640000
50%    2003.000000     0.039000     0.039000     0.038500     0.038500
75%    2004.000000     0.596750     0.596750     0.596750     0.596750
max    2005.000000     5.733000     5.733000     5.733000     5.733000

             Lag5       Volume        Today
count  1250.00000  1250.000000  1250.000000
mean      0.00561     1.478305     0.003138
std       1.14755     0.360357     1.136334
min      -4.92200     0.356070    -4.922000
```

```
25%       -0.64000     1.257400    -0.639500
50%        0.03850     1.422950     0.038500
75%        0.59700     1.641675     0.596750
max        5.73300     3.152470     5.733000
```

[17]:
```
smarket.shape
```

[17]:
```
(1250, 9)
```

The *corr()* function produces a matrix that contains all of the pairwise correlations among the predictors in a data set. It doesn't contain the feature *Direction* because the *Direction* variable is qualitative.

[18]:
```
smarket.corr()
```

[18]:
```
            Year      Lag1      Lag2      Lag3      Lag4      Lag5    Volume  \
Year    1.000000  0.029700  0.030596  0.033195  0.035689  0.029788  0.539006
Lag1    0.029700  1.000000 -0.026294 -0.010803 -0.002986 -0.005675  0.040910
Lag2    0.030596 -0.026294  1.000000 -0.025897 -0.010854 -0.003558 -0.043383
Lag3    0.033195 -0.010803 -0.025897  1.000000 -0.024051 -0.018808 -0.041824
Lag4    0.035689 -0.002986 -0.010854 -0.024051  1.000000 -0.027084 -0.048414
Lag5    0.029788 -0.005675 -0.003558 -0.018808 -0.027084  1.000000 -0.022002
Volume  0.539006  0.040910 -0.043383 -0.041824 -0.048414 -0.022002  1.000000
Today   0.030095 -0.026155 -0.010250 -0.002448 -0.006900 -0.034860  0.014592

           Today
Year    0.030095
Lag1   -0.026155
Lag2   -0.010250
Lag3   -0.002448
Lag4   -0.006900
Lag5   -0.034860
Volume  0.014592
Today   1.000000
```

Next, we will fit a logistic regression model in order to predict Direction using *Lag1* through *Lag5* and *Volume*. The *smf.logit()* function fits logistic models and the syntax of the *smf.logit()* function is similar to that of *smf.ols()*.

The first command below gives an error message because the Direction variable is qualitative.

[19]:
```
result6 = smf.logit('Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 + Volume',␣
 ↪data=smarket).fit()
result6.summary()
```

```
      ␣
↪---------------------------------------------------------------------------
```

```
ValueError                                Traceback (most recent call
↪last)

<ipython-input-19-4c19b8695506> in <module>
----> 1 result6 = smf.logit('Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 +
↪Volume', data=smarket).fit()
      2 result6.summary()


D:\Anaconda\lib\site-packages\statsmodels\base\model.py in
↪from_formula(cls, formula, data, subset, drop_cols, *args, **kwargs)
    175                                  'columns that has shape {0}. This
↪occurs when '
    176                                  'the variable converted to endog is
↪non-numeric'
--> 177                                  ' (e.g., bool or str).'.format(endog.
↪shape))
    178         if drop_cols is not None and len(drop_cols) > 0:
    179             cols = [x for x in exog.columns if x not in drop_cols]


ValueError: endog has evaluated to an array with multiple columns that
↪has shape (1250, 2). This occurs when the variable converted to endog is
↪non-numeric (e.g., bool or str).
```

Therefore, we add a column name *Up* to represent *Direction* and make it numeric.

P.S. *numpy.where(condition, x, y)* return elements chosen from x or y depending on condition. The $==$ here is a logic evaluation, if it is true, value 1 is assigned, and value 0 is assigned to it otherwise.

[20]:
```
smarket['Up'] = np.where(smarket['Direction'] == 'Up', 1, 0)
smarket.head()
```

[20]:

| | Year | Lag1 | Lag2 | Lag3 | Lag4 | Lag5 | Volume | Today | Direction | Up |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2001 | 0.381 | -0.192 | -2.624 | -1.055 | 5.010 | 1.1913 | 0.959 | Up | 1 |
| 1 | 2001 | 0.959 | 0.381 | -0.192 | -2.624 | -1.055 | 1.2965 | 1.032 | Up | 1 |
| 2 | 2001 | 1.032 | 0.959 | 0.381 | -0.192 | -2.624 | 1.4112 | -0.623 | Down | 0 |
| 3 | 2001 | -0.623 | 1.032 | 0.959 | 0.381 | -0.192 | 1.2760 | 0.614 | Up | 1 |
| 4 | 2001 | 0.614 | -0.623 | 1.032 | 0.959 | 0.381 | 1.2057 | 0.213 | Up | 1 |

After that, the *corr()* function produces a matrix that contains all of the pairwise correlations among the predictors in this data set.

[21]:
```
smarket.corr()
```

[21]:
```
            Year      Lag1      Lag2      Lag3      Lag4      Lag5    Volume  \
Year    1.000000  0.029700  0.030596  0.033195  0.035689  0.029788  0.539006
Lag1    0.029700  1.000000 -0.026294 -0.010803 -0.002986 -0.005675  0.040910
Lag2    0.030596 -0.026294  1.000000 -0.025897 -0.010854 -0.003558 -0.043383
Lag3    0.033195 -0.010803 -0.025897  1.000000 -0.024051 -0.018808 -0.041824
Lag4    0.035689 -0.002986 -0.010854 -0.024051  1.000000 -0.027084 -0.048414
Lag5    0.029788 -0.005675 -0.003558 -0.018808 -0.027084  1.000000 -0.022002
Volume  0.539006  0.040910 -0.043383 -0.041824 -0.048414 -0.022002  1.000000
Today   0.030095 -0.026155 -0.010250 -0.002448 -0.006900 -0.034860  0.014592
Up      0.074608 -0.039757 -0.024081  0.006132  0.004215  0.005423  0.022951

           Today        Up
Year    0.030095  0.074608
Lag1   -0.026155 -0.039757
Lag2   -0.010250 -0.024081
Lag3   -0.002448  0.006132
Lag4   -0.006900  0.004215
Lag5   -0.034860  0.005423
Volume  0.014592  0.022951
Today   1.000000  0.730563
Up      0.730563  1.000000
```

[22]:
```
result6 = smf.logit('Up ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 + Volume',␣
 ↪data=smarket).fit()
result6.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.691034
         Iterations 4
```

[22]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                           Logit Regression Results
==============================================================================
Dep. Variable:                     Up   No. Observations:                 1250
Model:                          Logit   Df Residuals:                     1243
Method:                           MLE   Df Model:                            6
Date:                Mon, 10 Feb 2020   Pseudo R-squ.:                0.002074
Time:                        19:21:15   Log-Likelihood:                -863.79
converged:                       True   LL-Null:                       -865.59
Covariance Type:            nonrobust   LLR p-value:                    0.7319
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept     -0.1260      0.241     -0.523      0.601      -0.598       0.346
Lag1          -0.0731      0.050     -1.457      0.145      -0.171       0.025
Lag2          -0.0423      0.050     -0.845      0.398      -0.140       0.056
```

```
Lag3            0.0111      0.050      0.222      0.824      -0.087       0.109
Lag4            0.0094      0.050      0.187      0.851      -0.089       0.107
Lag5            0.0103      0.050      0.208      0.835      -0.087       0.107
Volume          0.1354      0.158      0.855      0.392      -0.175       0.446
==============================================================================
"""
```

The *predict()* function can be used to predict the probability that the market will go up, given values of the predictors. It output probabilities $P(Y = 1|X = x)$. If no data set is supplied to the *predict()* function, then the probabilities are computed for the training data that was used to fit the logistic regression model. Here we have printed only the first ten probabilities. We know that these values correspond to the probability of the market going up, rather than down, because we set *Up = 1* when the *Direction* is *Up*.

[23]:
```
prediction6 = result6.predict()
print(prediction6[0:10])
```

```
[0.50708413 0.48146788 0.48113883 0.51522236 0.51078116 0.50695646
 0.49265087 0.50922916 0.51761353 0.48883778]
```

We can use *pred_table()* fuction to produce pred_table directly in order to determine how many observations were correctly or incorrectly classified. The default threshold 1/2 is used for cutting the $P(Y = 1|X = x)$.

[24]:
```
result6.pred_table()
```

[24]:
```
array([[145., 457.],
       [141., 507.]])
```

It represents the outcome as the following table:

| | Down(result6.pred) | Up(result6.pred) |
|---|---|---|
| Down(Direction) | 145 | 457 |
| Up(Direction) | 141 | 507 |

[25]:
```
(507+145) /1250
```

[25]:
```
0.5216
```

The diagonal elements of the confusion matrix indicate correct predictions, while the off-diagonals represent incorrect predictions. Hence our model correctly predicted that the market would go up on 507 days and that it would go down on 145 days, for a total of $507 + 145 = 652$ correct predictions. In this case, logistic regression correctly predicted the movement of the market 52.2% of the time.

At first glance, it appears that the logistic regression model is working a little better than random guessing. However, this result is misleading because we trained and tested the model on the same set of 1250 observations. In other words, 100− 52.2= 47.8% is the training error rate. As we have seen previously, the training error rate is often overly optimistic—it tends to underestimate the test error rate. In order to better assess the accuracy of the logistic regression model in this setting, we can fit the model using part of the data, and then examine how well it predicts the held out data. This will yield a more realistic error rate, in the sense that in practice we will be interested in our model's performance not on the data that we used to fit the model, but rather on days in the future for which the market's movements are unknown.

To implement this strategy, we will first create a vector corresponding to the observations from 2001 through 2004. We will then use this vector to create a held out data set of observations from 2005 as test dataset. At this point, you should think about why we don't use the same way as in *Python Tutorial 2* to split the training and test sets.

[26]:
```
X = smarket[['Lag1', 'Lag2', 'Lag3', 'Lag4', 'Lag5','Volume']]
y = smarket['Up']

train_bool = smarket['Year'] < 2005

X_test = X[~train_bool]
y_test = y[~train_bool]
```

[27]:
```
print("X_test.shape: ", X_test.shape)
print("y_test.shape: ", y_test.shape)
```

```
X_test.shape:  (252, 6)
y_test.shape:  (252,)
```

We now fit a logistic regression model using only the *subset* of the observations that correspond to dates before 2005, using the subset argument. We then obtain predicted probabilities of the stock market going up for each of the days in our test set—that is, for the days in 2005.

[28]:
```
result7 = smf.logit('Up ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 + Volume',␣
 ↪data=smarket, subset = train_bool).fit()
result7.summary()
```

```
Optimization terminated successfully.
         Current function value: 0.691936
         Iterations 4
```

[28]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                           Logit Regression Results
==============================================================================
Dep. Variable:                     Up   No. Observations:                  998
Model:                          Logit   Df Residuals:                      991
Method:                           MLE   Df Model:                            6
Date:                Mon, 10 Feb 2020   Pseudo R-squ.:                0.001562
Time:                        19:21:15   Log-Likelihood:                -690.55
converged:                       True   LL-Null:                       -691.63
Covariance Type:            nonrobust   LLR p-value:                    0.9044
```

```
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.1912      0.334      0.573      0.567      -0.463       0.845
Lag1          -0.0542      0.052     -1.046      0.295      -0.156       0.047
Lag2          -0.0458      0.052     -0.884      0.377      -0.147       0.056
Lag3           0.0072      0.052      0.139      0.889      -0.094       0.108
Lag4           0.0064      0.052      0.125      0.901      -0.095       0.108
Lag5          -0.0042      0.051     -0.083      0.934      -0.104       0.096
Volume        -0.1163      0.240     -0.485      0.628      -0.586       0.353
==============================================================================
"""
```

Notice that we have trained and tested our model on two completely separate data sets: training was performed using only the dates before 2005, and testing was performed using only the dates in 2005. Finally, we compute the predictions for 2005 and compare them to the actual movements of the market over that time period.

We first use the *predict()* function to compute the probabilities of test data.

[29]:
```
result7_prob = result7.predict(X_test)
result7_prob
```

[29]:
```
998     0.528220
999     0.515669
1000    0.522652
1001    0.513854
1002    0.498334
          …
1245    0.483637
1246    0.506048
1247    0.516658
1248    0.516124
1249    0.508072
Length: 252, dtype: float64
```

Then, we select 0.5 as the threshold. If the probability is larger than 0.5, we label it as *True* or *1* ("Up").

[30]:
```
result7_pred = (result7_prob > 0.5)
result7_pred
```

[30]:
```
998      True
999      True
1000     True
1001     True
1002    False
        …
```

```
1245    False
1246     True
1247     True
1248     True
1249     True
Length: 252, dtype: bool
```

[31]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, result7_pred)
```

[31]:
```
array([[77, 34],
       [97, 44]], dtype=int64)
```

| | Down(result7.pred) | Up(result7.pred) |
|---|---|---|
| Down(y_test) | 77 | 34 |
| Up(y_test) | 97 | 44 |

[32]:
```
np.mean(result7_pred == y_test)
```

[32]:
```
0.4801587301587302
```

[33]:
```
np.mean(result7_pred != y_test)
```

[33]:
```
0.5198412698412699
```

The *!=* notation means *not equal to*, and so the last command computes the test set error rate. The results are rather disappointing: the test error rate is 52 %, which is worse than random guessing! Of course, this result is not all that surprising because stock price prediction is a very hard problem.

We recall that the logistic regression model had very underwhelming p-values associated with all of the predictors, and that the smallest p-value, though not very small, corresponded to *Lag1*. Perhaps by removing the variables that appear not to be helpful in predicting *Direction*, we can obtain a more effective model. Below we have refitted the logistic regression using just *Lag1* and *Lag2*, which seemed to be the most significant in the original logistic regression model.

[34]:
```
result8 = smf.logit('Up ~ Lag1 + Lag2', data=smarket, subset = train_bool).fit()
result8_prob = result8.predict(X_test)
result8_pred = (result8_prob > 0.5)
confusion_matrix(y_test, result8_pred)
```

```
Optimization terminated successfully.
         Current function value: 0.692085
         Iterations 3
```

[34]:
```
array([[ 35,  76],
       [ 35, 106]], dtype=int64)
```

| | Down(result8.pred) | Up(result8.pred) |
|---|---|---|
| Down(y_test) | 35 | 76 |
| Up(y_test) | 35 | 106 |

[35]:
```
np.mean(result8_pred == y_test)
```

[35]: 0.5595238095238095

[36]:
```
(35+106)/(35+76+35+106)
```

[36]: 0.5595238095238095

Now the results appear to be a little better: 56% of the daily movements have been correctly predicted. It is worth noting that in this case, a much simpler strategy of predicting that the market will increase every day will also be correct 56% of the time! Hence, in terms of the overall error rate, the logistic regression method is no better than the naive approach.

References:

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

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

https://github.com/tdpetrou/Machine-Learning-Books-With-Python

https://scikit-learn.org/stable/index.html

https://www.statsmodels.org/dev/index.html

http://www.science.smith.edu/~jcrouser/SDS293/labs/lab4-py.html