Python Tutorial 10

April 3, 2022

This tutorial is for Dr. Xin Tong's DSO 530 class at the University of Southern California in spring 2022. It aims to give you some supplementary code to implement *Shrinkage Methods* using Python.

1 Shrinkage Methods

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

First we prepare the Hitters data as we did in the previous tutorial.

[2]:
```
df = pd.read_csv('Hitters.csv')
df.head()
```

[2]:
```
             Player  AtBat  Hits  HmRun  Runs  RBI  Walks  Years  CAtBat  \
0    -Andy Allanson    293    66      1    30   29     14      1     293
1       -Alan Ashby    315    81      7    24   38     39     14    3449
2      -Alvin Davis    479   130     18    66   72     76      3    1624
3     -Andre Dawson    496   141     20    65   78     37     11    5628
4 -Andres Galarraga    321    87     10    39   42     30      2     396

   CHits  …  CRuns  CRBI  CWalks  League Division PutOuts  Assists  Errors  \
0     66  …     30    29      14       A        E     446       33      20
1    835  …    321   414     375       N        W     632       43      10
2    457  …    224   266     263       A        W     880       82      14
3   1575  …    828   838     354       N        E     200       11       3
4    101  …     48    46      33       N        E     805       40       4

   Salary  NewLeague
0     NaN          A
1   475.0          N
2   480.0          A
3   500.0          N
4    91.5          N

[5 rows x 21 columns]
```

[3]:
```
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:

[4]:
```
# 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
```

[5]:
```
df[['League', 'Division', 'NewLeague']].head()
```

[5]:

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

[6]:
```
dummies = pd.get_dummies(df[['League', 'Division', 'NewLeague']])
```

[7]:
```
dummies.head()
```

[7]:

	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.

[8]:
```
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)

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

1.1 Ridge Regression

[9]:
```
from sklearn.linear_model import Ridge, RidgeCV, Lasso, LassoCV
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
```

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:

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

[10]:
```
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:

[11]:
```
np.linspace(10,1,19)
```

[11]:
```
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. ])
```

[12]:
```
# 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)
```

[13]:
```
## standardize the data

from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

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).

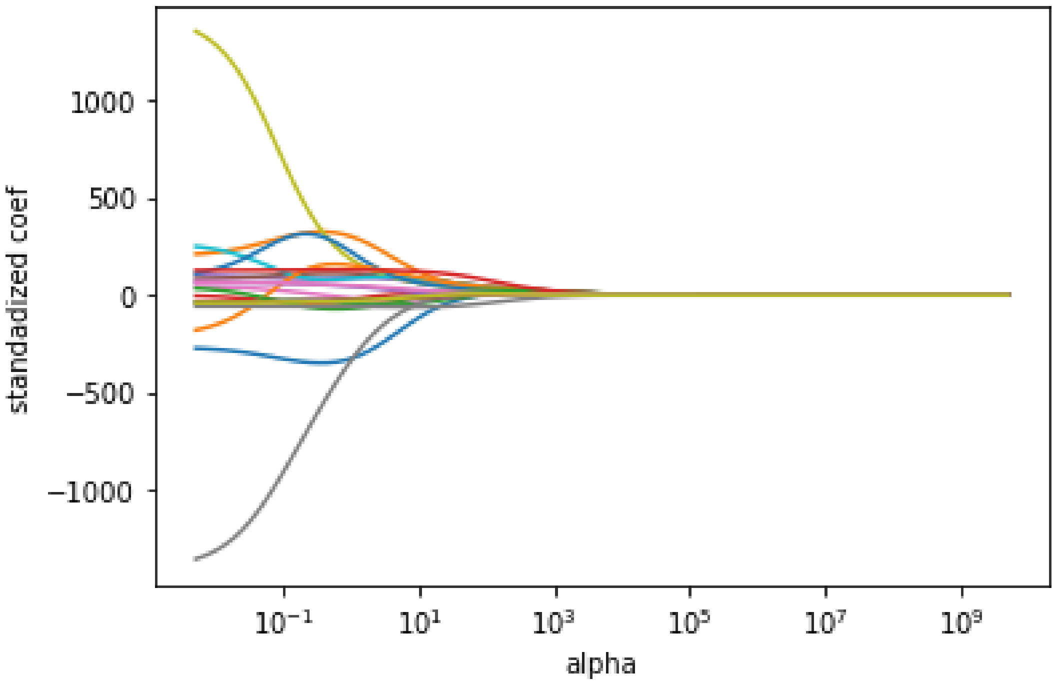
[14]:
```
ridge = Ridge()
coefs = []

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

np.shape(coefs)
```

[14]:
```
(100, 19)
```

[15]:
```
%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('standadized coef')
```

[15]: Text(0, 0.5, 'standadized coef')



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

[16]:
```
ridge2 = Ridge(alpha=4)
ridge2.fit(X_train_std, y_train)  # Fit a ridge regression on the training data
pred2 = ridge2.predict(X_test_std) # 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
print("\nout-of-sample R squared: ",ridge2.score(X_test_std, y_test)) #␣
↪Calculate out-of-sample R squared
```

```
AtBat         -210.807322
Hits           193.076932
HmRun          -51.437368
Runs             1.624070
RBI             81.458917
Walks           94.582539
Years          -28.029476
CAtBat        -117.924364
```

```
CHits            91.159632
CHmRun           91.762010
CRuns           101.115450
CRBI            117.598963
CWalks          -38.434685
PutOuts         125.556071
Assists          25.559188
Errors          -18.518904
League_N         35.792050
Division_W      -60.245442
NewLeague_N     -18.650623
dtype: float64

mean_squared_error:  102144.52395076491

out-of-sample R squared:  0.4089582182719276
```

People can look at either MSE or R squared in practice. Here we look at R squared. The out-of-sample R squared when alpha = 4 is about 0.40896. Now let's see what will happen if we use a huge value of alpha, say 10^{10}:

[17]:
```
ridge3 = Ridge(alpha=10**10)
ridge3.fit(X_train_std, y_train) # Fit a ridge regression on the training data
pred3 = ridge3.predict(X_test_std) # Use this model to predict the test data
print(pd.Series(ridge3.coef_, index=X.columns)) # Print coefficients
print("\nout-of-sample R squared: ",ridge3.score(X_test_std, y_test)) #␣
↪Calculate out-of-sample R squared
```

```
AtBat           2.526133e-06
Hits            2.826091e-06
HmRun           2.174902e-06
Runs            2.632138e-06
RBI             3.183658e-06
Walks           2.941154e-06
Years           2.478179e-06
CAtBat          3.213788e-06
CHits           3.432349e-06
CHmRun          3.435819e-06
CRuns           3.522298e-06
CRBI            3.602930e-06
CWalks          3.216811e-06
PutOuts         2.851209e-06
Assists        -4.856786e-08
Errors          1.736624e-07
League_N       -1.637144e-07
Division_W     -1.015188e-06
NewLeague_N    -1.325154e-07
dtype: float64
```

```
out-of-sample R squared:  -0.00023761334171323867
```

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 larger out-of-sample R squared 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.

[18]:
```
ridge4 = Ridge(alpha=0)
ridge4.fit(X_train_std, y_train) # Fit a ridge regression on the training data
pred = ridge4.predict(X_test_std) # Use this model to predict the test data
print(pd.Series(ridge4.coef_, index=X.columns)) # Print coefficients
print("\nout-of-sample R squared: ",ridge4.score(X_test_std, y_test)) #␣
↪Calculate the out-of-sample R squared
```

```
AtBat          -266.553048
Hits            197.706218
HmRun           -38.103182
Runs             -1.007996
RBI             103.119845
Walks            79.750209
Years            45.357697
CAtBat        -1399.811384
CHits          1426.954812
CHmRun          264.037977
CRuns            86.858781
CRBI           -211.142393
CWalks           42.533597
PutOuts         126.075563
Assists          65.816094
Errors          -38.313885
League_N         66.822855
Division_W      -56.870280
NewLeague_N     -40.962688
dtype: float64

out-of-sample R squared:  0.3247906027195909
```

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. Note that a better practice is to use `KFold()` to shutle the data first. Here we are being a bit lazy as this is not the focus on this tutorial.

[19]:
```
ridgecv = RidgeCV(alphas=alphas, scoring='r2', cv=10)
ridgecv.fit(X_train_std, y_train)
```

```
ridgecv.alpha_
```

[19]: 201.85086292982749

Hence, the value of alpha that results in the smallest cross-validation error is 201.85. Let's see the out-of-sample R squared is associated with this alpha

[20]:
```
ridge5 = Ridge(alpha=ridgecv.alpha_)
ridge5.fit(X_train_std, y_train)
ridge5.score(X_test_std,y_test)
```

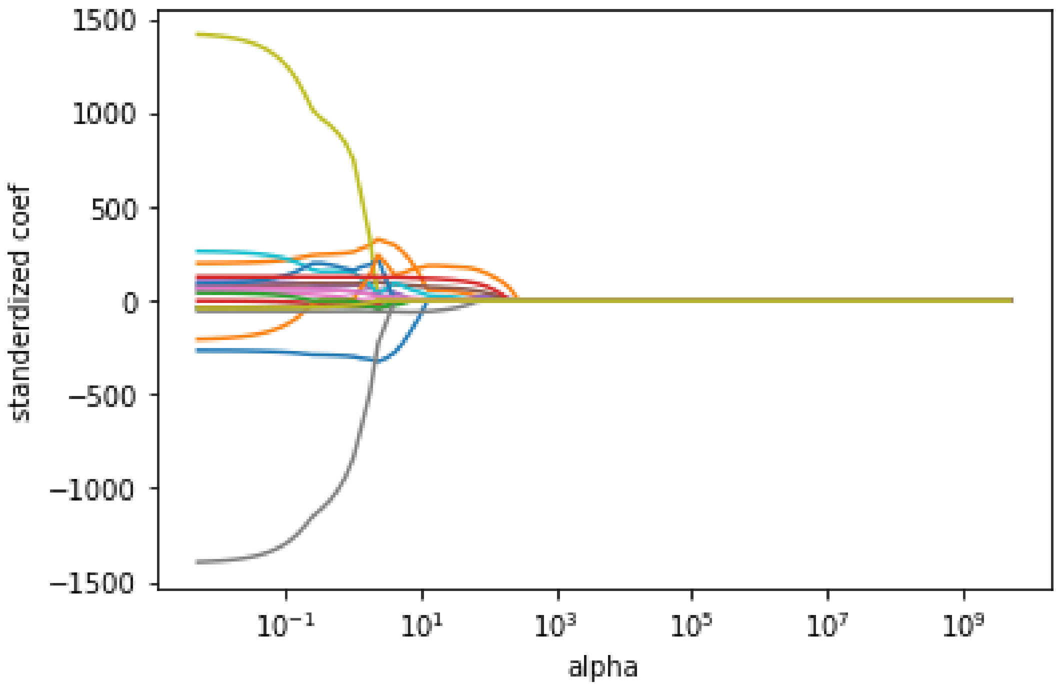
[20]: 0.4206079388142959

This represents a further improvement over the out-out-sample R squared that we got using alpha=4.

1.2 The Lasso

[21]:
```
lasso = Lasso(max_iter=10000)  # default max_iter is 1000, but here we need␣
 ↪more iterations to converge
coefs = []

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

%matplotlib inline
ax = plt.gca() # Get the current Axes instance
ax.plot(alphas, coefs)
ax.set_xscale('log')
plt.xlabel('alpha')
plt.ylabel('standerdized coef')
```

[21]: Text(0, 0.5, 'standerdized coef')



[22]:
```
lassocv = LassoCV(alphas=alphas, cv=10, max_iter=10000)
lassocv.fit(X_train_std, y_train)
print(lassocv.score(X_test_std, y_test))


## alternatively, one can implement the follwoing
lasso1 = Lasso(max_iter=10000)
lasso1.set_params(alpha=lassocv.alpha_)
lasso1.fit(X_train_std, y_train)
print(lasso1.score(X_test_std, y_test))
```

```
0.392244522565934
0.392244522565934
```

Note that in `LassoCV()`, one cannot set `scoring =`, as current version does not allow this option yet. I expect that in future versions, this will be added. The current `LassoCV()` only allows MSE as the CV criterion. If we want to do CV by R squared for LASSO, we will have to write some code to do it.

However, even if `LassoCV()` did not use R squared as the criterion, its resulting out-of-sample R squared is substantially larger than the out-of-sample R squared of the null model and of least squares. It is a little worse than out-of-sample R squared of the ridge regression with alpha chosen by cross-validation according to `scoring = r2`.

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 zero:

[23]:
```
# Some of the coefficients are now reduced to exactly zero.
pd.Series(lassocv.coef_, index=X.columns)
```

[23]:
```
AtBat             0.000000
Hits             49.848096
HmRun             0.000000
Runs              0.000000
RBI               0.000000
Walks            66.380815
Years             0.000000
CAtBat            0.000000
CHits             0.000000
CHmRun           19.017234
CRuns             0.000000
CRBI            180.885189
CWalks            0.000000
PutOuts         109.766693
Assists          -0.000000
Errors           -0.000000
League_N          0.000000
Division_W      -43.461215
NewLeague_N       0.000000
dtype: float64
```

1.3 Logistic Regression with Penalty

Similary to the regression, we can add a penalty (thrinkage) term when we do 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.

[24]:
```
import pandas as pd
import numpy as np
caravan = pd.read_csv('caravan.csv')
caravan.head()
```

[24]:
```
   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]
```

[25]:
```python
caravan.shape
```

[25]: (5822, 86)

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

[26]:
```python
X = caravan.drop(['Purchase'], axis=1)
y = caravan['Purchase']
```

[27]:
```python
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)
```

[28]:
```python
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`.

[29]:
```python
fit1 = LogisticRegression(random_state=1, penalty='none', max_iter = 1000).
↪fit(X_train, y_train)
fit1.score(X_test, y_test)
```

```
/Users/xintong/opt/anaconda3/lib/python3.8/site-
packages/sklearn/linear_model/_logistic.py:762: 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(
```

[29]: 0.9357609069048437

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.

[30]:
```python
fit1 = LogisticRegression(random_state=1, penalty='none', max_iter=5000).
 ↪fit(X_train, y_train)
fit1.score(X_test, y_test)
```

[30]: 0.9350738577808313

Then we do the logistic regression with the penalty and use cross-validation to pick up the best penalty level. We use the `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. 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`. But note that if you want to shuttle the data (recommended), you need to call `StratifiedKFold`.

Note that like the Lasso and ridge for linear regression, we will also standardize the features for penalized logistic regression.

[31]:
```python
stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

[32]:
```python
## For logistic regression WIHOUT penalty, whether or not to do standardization␣
 ↪does not make a difference.

fit2 = LogisticRegression(random_state=1, penalty='none', max_iter = 10000).
 ↪fit(X_train_std, y_train)
fit2.score(X_test_std, y_test)
```

[32]: 0.9350738577808313

The following implements Logistic Regression with L1 penalty. Note that the default `scoring` is `accuracy`.

[33]:
```python
fit3 = LogisticRegressionCV(Cs=30,random_state=1,␣
 ↪penalty='l1',solver='saga',cv=5,max_iter = 5000).fit(X_train_std, y_train)
fit3.score(X_test_std, y_test)
```

[33]: 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=30`. We can check that `LogisticRegressionCV` generates a grid of 30 values are chosen in a logarithmic scale between 1e-4 and 1e4.

[34]:
```
fit3.Cs_
```

[34]:
```
array([1.00000000e-04, 1.88739182e-04, 3.56224789e-04, 6.72335754e-04,
       1.26896100e-03, 2.39502662e-03, 4.52035366e-03, 8.53167852e-03,
       1.61026203e-02, 3.03919538e-02, 5.73615251e-02, 1.08263673e-01,
       2.04335972e-01, 3.85662042e-01, 7.27895384e-01, 1.37382380e+00,
       2.59294380e+00, 4.89390092e+00, 9.23670857e+00, 1.74332882e+01,
       3.29034456e+01, 6.21016942e+01, 1.17210230e+02, 2.21221629e+02,
       4.17531894e+02, 7.88046282e+02, 1.48735211e+03, 2.80721620e+03,
       5.29831691e+03, 1.00000000e+04])
```

Logistic Regression with L2 penalty:

[35]:
```
fit4 = LogisticRegressionCV(Cs=30,random_state=1,␣
 ↪penalty='l2',cv=5,max_iter=10000).fit(X_train_std, y_train)
fit4.score(X_test_std, y_test)
```

[35]: 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. Note that for an imbalanced dataset, `accuracy` might not be the best measure of success of classifier. We use it here only for demonstration purpose.

References:

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

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