Regularization

April 5, 2023

Regularization

Goals:

Implement data standardization

Implement variants of regularized regression

Combine data standardization with the train-test split procedure

Implement regularization to prevent overfitting in regression problems

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

# Surpress warnings:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn

np.set_printoptions(precision=3, suppress=True)
```

[2]:
```
# load the data and define some useful plotting functions
np.random.seed(72018)

from sklearn.datasets import load_boston

def to_2d(array):
    return array.reshape(array.shape[0], -1)

def boston_dataframe(description=False):
    boston = load_boston()

    data = boston.data
    target = boston.target
    names = boston.feature_names
```

```
    target = to_2d(target)

    data_all = np.concatenate([data, target], axis=1)
    names_all = np.concatenate([names, np.array(['MEDV'])], axis=0)

    if description:

        return pd.DataFrame(data=data_all, columns=names_all), boston.DESCR

    else:

        return pd.DataFrame(data=data_all, columns=names_all)

def plot_exponential_data():
    data = np.exp(np.random.normal(size=1000))
    plt.hist(data)
    plt.show()
    return data

def plot_square_normal_data():
    data = np.square(np.random.normal(loc=5, size=1000))
    plt.hist(data)
    plt.show()
    return data
```

[3]:
```
# Loading in Boston Data
boston = boston_dataframe(description=True)
boston_data = boston[0]
boston_description = boston[1]
```

Data standardization

Standardizing data refers to transforming each variable so that it more closely follows a **standard** normal distribution, with mean 0 and standard deviation 1.

The `StandardScaler` object in SciKit Learn can do this.

[5]:
```
# Generate X and y
y_col = "MEDV"
X = boston_data.drop(y_col, axis=1)
y = boston_data[y_col]
```

[6]:
```
# Import, fit, and transform using StandardScaler
from sklearn.preprocessing import StandardScaler

s = StandardScaler()
X_ss = s.fit_transform(X)
```

[7]:
```
# Confirm standard scaling
X2 = np.array(X)
man_transform = (X2-X2.mean(axis=0))/X2.std(axis=0)
np.allclose(man_transform, X_ss)
```

[7]:
```
True
```

Coefficients with and without scaling

[10]:
```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()

y_col = "MEDV"
X = boston_data.drop(y_col, axis=1)
y = boston_data[y_col]

lr.fit(X, y)
print(lr.coef_)
print('min. is: ', min(lr.coef_))
```

```
[ -0.108   0.046   0.021   2.687 -17.767   3.81    0.001  -1.476   0.306
  -0.012  -0.953   0.009  -0.525]
min. is:  -17.766611228300444
```

[11]:
```
# Discussion; The coefficients are on widely different scales. Is this "bad"?
from sklearn.preprocessing import StandardScaler

s = StandardScaler()
X_ss = s.fit_transform(X)

lr2 = LinearRegression()
lr2.fit(X_ss, y)
print(lr2.coef_) # coefficients now "on the same scale"
```

```
[-0.928  1.082  0.141  0.682 -2.057  2.674  0.019 -3.104  2.662 -2.077
 -2.061  0.849 -3.744]
```

Exercise

Based on these results, what is the most "impactful" feature (this is intended to be slightly ambiguous)? "In what direction" does it affect "y"?

Hint: "zip up" the names of the features of a DataFrame `df` with a model `model` fitted on that DataFrame using:

```
dict(zip(df.columns.values, model.coef_))
```

[14]:
```
pd.DataFrame(zip(X.columns, lr2.coef_)).sort_values(by=1, key=lambda x: abs(x),␣
↪ascending=False)
```

[14]:

	0	1
12	LSTAT	-3.743627
7	DIS	-3.104044
5	RM	2.674230
8	RAD	2.662218
9	TAX	-2.076782
10	PTRATIO	-2.060607
4	NOX	-2.056718
1	ZN	1.081569
0	CRIM	-0.928146
11	B	0.849268
3	CHAS	0.681740
2	INDUS	0.140900
6	AGE	0.019466

Looking just at the strength of the standardized coefficients LSTAT, DIS, RM and RAD are all the 'most impactful'. Sklearn does not have built in statistical signifigance of each of these variables which would aid in making this claim stronger/weaker

Lasso with and without scaling

1. What is different about Lasso vs. regular Linear Regression?
2. Is standardization more or less important with Lasso vs. Linear Regression? Why?

[15]:
```
from sklearn.linear_model import Lasso
from sklearn.preprocessing import PolynomialFeatures

# Create polynomial features
# include_bias=False since Lasso includes a bias by default
pf = PolynomialFeatures(degree=2, include_bias=False,)
X_pf = pf.fit_transform(X)

X_pf_ss = s.fit_transform(X_pf)
```

[16]:
```
las = Lasso()
las.fit(X_pf_ss, y)
las.coef_
```

[16]:
```
array([-0.   ,  0.   , -0.   ,  0.   , -0.   ,  0.   , -0.   , -0.   ,
       -0.   , -0.   , -0.991,  0.   , -0.   , -0.   ,  0.   , -0.   ,
        0.068, -0.   , -0.   , -0.   , -0.   , -0.   , -0.   , -0.   ,
       -0.   , -0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,
        0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   , -0.   ,  0.   ,
       -0.   , -0.   , -0.   , -0.05 , -0.   , -0.   , -0.   , -0.   ,
       -0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,  0.   ,
        0.   ,  0.   ,  0.   , -0.   , -0.   , -0.   , -0.   , -0.   ,
       -0.   , -0.   ,  0.   , -0.   ,  3.3  , -0.   , -0.   , -0.   ,
       -0.   , -0.   ,  0.42 , -3.498, -0.   , -0.   , -0.   , -0.   ,
       -0.   ,  0.   , -0.   , -0.   , -0.   , -0.146, -0.   , -0.   ,
```

```
-0.   , -0.   , -0.   , -0.   , -0.   , -0.   , -0.   , -0.   ,
-0.   , -0.   , -0.   ,  0.   , -0.   ,  0.   , -0.   , -0.   ])
```

Exercise

Compare

- Sum of magnitudes of the coefficients
- Number of coefficients that are zero

for Lasso with alpha 0.1 vs. 1.

Before doing the exercise, answer the following questions in one sentence each:

- Which do you expect to have greater magnitude?
- Which do you expect to have more zeros?

[17]:
```
las01 = Lasso(alpha = 0.1)
las01.fit(X_pf_ss, y)
print('sum of coefficients:', abs(las01.coef_).sum() )
print('number of coefficients not equal to 0:', (las01.coef_!=0).sum())
```

```
sum of coefficients: 26.138682362877965
number of coefficients not equal to 0: 23
```

[18]:
```
las1 = Lasso(alpha = 1)
las1.fit(X_pf_ss, y)
print('sum of coefficients:',abs(las1.coef_).sum() )
print('number of coefficients not equal to 0:',(las1.coef_!=0).sum())
```

```
sum of coefficients: 8.47240504455307
number of coefficients not equal to 0: 7
```

With more regularization (higher alpha) we will expect the penalty for higher weights to be greater and thus the coefficients to be pushed down. Thus a higher alpha means lower magnitude with more coefficients pushed down to 0.

Exercise R^2

Calculate the R^2 of each model without train/test split.

Recall that we import R^2 using:

```
from sklearn.metrics import r2_score
```

[19]:
```
from sklearn.metrics import r2_score
r2_score(y,las.predict(X_pf_ss))
```

[19]:
```
0.7207000417838496
```

With train/test split

[20]:
```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X_pf, y, test_size=0.3,
                                                    random_state=72018)
```

[21]:
```
X_train_s = s.fit_transform(X_train)
las.fit(X_train_s, y_train)
X_test_s = s.transform(X_test)
y_pred = las.predict(X_test_s)
r2_score(y_pred, y_test)
```

[21]:
```
0.33177406838134416
```

[22]:
```
X_train_s = s.fit_transform(X_train)
las01.fit(X_train_s, y_train)
X_test_s = s.transform(X_test)
y_pred = las01.predict(X_test_s)
r2_score(y_pred, y_test)
```

[22]:
```
0.7102444090776476
```

Exercise

Part 1: Do the same thing with Lasso of:

- `alpha` of 0.001
- Increase `max_iter` to 100000 to ensure convergence.

Calculate the R^2 of the model.

Part 2: Do the same procedure as before, but with Linear Regression.

Calculate the R^2 of this model.

Part 3: Compare the sums of the absolute values of the coefficients for both models, as well as the number of coefficients that are zero. Based on these measures, which model is a "simpler" description of the relationship between the features and the target?

[23]:
```
#----------------- Part 1 -----------------#

# Decreasing regularization and ensuring convergence
las001 = Lasso(alpha = 0.001, max_iter=100000)

# Transforming training set to get standardized units
X_train_s = s.fit_transform(X_train)

# Fitting model to training set
las001.fit(X_train_s, y_train)

# Transforming test set using the parameters defined from training set
X_test_s = s.transform(X_test)
```

```
# Finding prediction on test set
y_pred = las001.predict(X_test_s)

# Calculating r2 score
print("r2 score for alpha = 0.001:", r2_score(y_pred, y_test))


#----------------- Part 2 -----------------#

# Using vanilla Linear Regression
lr = LinearRegression()

# Fitting model to training set
lr.fit(X_train_s, y_train)

# predicting on test set
y_pred_lr = lr.predict(X_test_s)

# Calculating r2 score
print("r2 score for Linear Regression:", r2_score(y_pred_lr, y_test))


#----------------- Part 3 -----------------#

print('Magnitude of Lasso coefficients:', abs(las001.coef_).sum())
print('Number of coeffients not equal to 0 for Lasso:', (las001.coef_!=0).sum())

print('Magnitude of Linear Regression coefficients:', abs(lr.coef_).sum())
print('Number of coeffients not equal to 0 for Linear Regression:', (lr.coef_!
  ↪=0).sum())
```

```
r2 score for alpha = 0.001: 0.8686454101886707
r2 score for Linear Regression: 0.855520209806416
Magnitude of Lasso coefficients: 436.26164263065647
Number of coeffients not equal to 0 for Lasso: 89
Magnitude of Linear Regression coefficients: 1185.2858254467678
Number of coeffients not equal to 0 for Linear Regression: 104
```

L1 vs. L2 Regularization

As mentioned in the deck: `Lasso` and `Ridge` regression have the same syntax in SciKit Learn.

Now we're going to compare the results from Ridge vs. Lasso regression:

0.0.1 Exercise

1. Define a Ridge object `r` with the same `alpha` as `las001`.
2. Fit that object on `X` and `y` and print out the resulting coefficients.

[24]:
```
from sklearn.linear_model import Ridge

# Decreasing regularization and ensuring convergence
r = Ridge(alpha = 0.001)
X_train_s = s.fit_transform(X_train)
r.fit(X_train_s, y_train)
X_test_s = s.transform(X_test)
y_pred_r = r.predict(X_test_s)

# Calculating r2 score
r.coef_
```

[24]:
```
array([  8.727,  10.354, -24.683,   5.299,  -3.13 ,  14.781,  22.301,
       -23.483,  27.356,  -1.512,  16.949,  22.531,  11.444,   1.055,
         0.437,  14.259,   1.83 ,  -8.737,   4.726,  -3.748,  -3.557,
       -16.521,  -6.918,   6.064,  -1.363,   4.604,  -1.303,  -0.057,
        -0.301, -12.828,   2.016,   1.084,  -0.655,  -1.092,   4.458,
        -4.247,   4.156,  -1.282,   8.71 ,  -0.285,  -8.595,  11.595,
         6.544,   1.271,   1.827,   1.007,   1.554,   5.173,  -4.685,
         5.299,  -3.199,  -8.715,   0.983,   1.119,   0.284,  -1.683,
        -2.884,   2.985,  -0.756,  12.297,   0.931,  -7.594,  18.292,
       -22.124,  35.725, -21.894,  -7.789,   1.774,   4.101, -12.263,
        -3.741,  -5.525, -16.111,  -5.706,  -4.666,  -9.884,   0.989,
        -0.092,  17.145, -14.214,  -2.955,  -2.785,  -5.404,  11.825,
         0.085,  -4.352,  -5.616,  -2.843,   0.826, -29.129,  49.636,
       -21.571,  -1.263,  -8.811, -15.822,  11.975,  -1.023,   2.911,
        -0.656,  -4.955,   2.817,  -2.463,  -2.381,   1.686])
```

[25]:
```
las001 # same alpha as Ridge above
```

[25]:
```
Lasso(alpha=0.001, max_iter=100000)
```

[26]:
```
las001.coef_
```

[26]:
```
array([  0.   ,   0.   , -16.945,   2.562,   0.   ,  13.392,   9.907,
       -19.96 ,   9.039,   0.   ,   6.138,  17.628,  11.446,   1.21 ,
         0.226,  11.578,   2.197,  -7.397,   4.117,  -1.589,  -2.157,
        -9.837,   0.   ,   0.   ,  -1.089,   3.793,   0.389,   0.2  ,
        -0.297,  -3.352,   0.396,   0.719,  -0.756,  -0.753,   2.388,
        -0.845,   2.524,  -0.973,   3.882,  -0.983,   6.609,   6.138,
         4.201,   0.858,   1.96 ,   2.661,  -4.148,   2.937,  -4.556,
         2.112,  -1.983,  -7.439,   1.607,   1.668,  -1.262,  -0.   ,
        -0.   ,   2.622,  -0.861,   3.353,  -0.   ,  -2.451,   9.171,
        -6.917,   0.   ,  -5.618,  -5.716,   0.521,   3.354,  -7.528,
        -0.   ,  -6.646, -10.212,  -6.346,  -2.95 ,  -9.559,   0.327,
         0.56 ,  11.006,  -6.666,  -1.158,  -2.25 ,  -3.53 ,   9.035,
        -0.   ,  -2.316,  -2.   ,  -1.124,   0.804, -20.861,  27.166,
```

```
       0.   , -1.51 , -6.745, -5.098, 15.455,   0.   ,   0.   ,
       0.629, -3.959,  2.502, -2.499, -2.23 ,   2.201])
```

[27]:
```
print(np.sum(np.abs(r.coef_)))
print(np.sum(np.abs(las001.coef_)))

print(np.sum(r.coef_ != 0))
print(np.sum(las001.coef_ != 0))
```

```
795.6521694417543
436.26164263065647
104
89
```

Conclusion: Ridge does not make any coefficients 0. In addition, on this particular dataset, Lasso provides stronger overall regularization than Ridge for this value of `alpha` (not necessarily true in general).

[28]:
```
y_pred = r.predict(X_pf_ss)
print(r2_score(y, y_pred))

y_pred = las001.predict(X_pf_ss)
print(r2_score(y, y_pred))
```

```
0.9075278340574183
0.9102933722688235
```

Example: Does it matter when you scale?

[29]:
```
X_train, X_test, y_train, y_test = train_test_split(X_ss, y, test_size=0.3,
                                                    random_state=72018)

lr = LinearRegression()
lr.fit(X_train, y_train)
y_pred = lr.predict(X_test)
r2_score(y_pred, y_test)
```

[29]:
```
0.5832942094971802
```

[30]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=72018)

s = StandardScaler()
lr_s = LinearRegression()
X_train_s = s.fit_transform(X_train)
lr_s.fit(X_train_s, y_train)
X_test_s = s.transform(X_test)
y_pred_s = lr_s.predict(X_test_s)
r2_score(y_pred_s, y_test)
```

[30]: `0.58329420949718`

Conclusion: It doesn't matter whether you scale before or afterwards, in terms of the raw predictions, for Linear Regression. However, it matters for other algorithms. Plus, as we'll see later, we can make scaling part of a `Pipeline`.