RidgeII

February 10, 2019

0.1 Linear Regression: Review

The Boston Housing example.

CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	price	
0.00632	18	2.31	0	0.538	6.575	65.2	4.090	1	296	15.3	396.9		4.98	24.
0.02731	0	7.07	0	0.469	6.421	78.9	4.967	2	242	17.8	396.9		9.14	21.
0.02729	0	7.07	0	0.469	7.185	61.1	4.967	2	242	17.8	392.8		4.03	34.
0.03237	0	2.18	0	0.458	6.998	45.8	6.062	3	222	18.7	394.6		2.94	33.
0.06905	0	2.18	0	0.458	7.147	54.2	6.062	3	222	18.7	396.9		5.33	36.

```
In [1]: %matplotlib inline
        from sklearn.datasets import load_boston
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import seaborn as sns
        from pandas.plotting import scatter_matrix
        from sklearn.linear_model import LinearRegression
        import warnings
        from sklearn.preprocessing import StandardScaler
```

```
In [2]: warnings.filterwarnings('ignore')
```

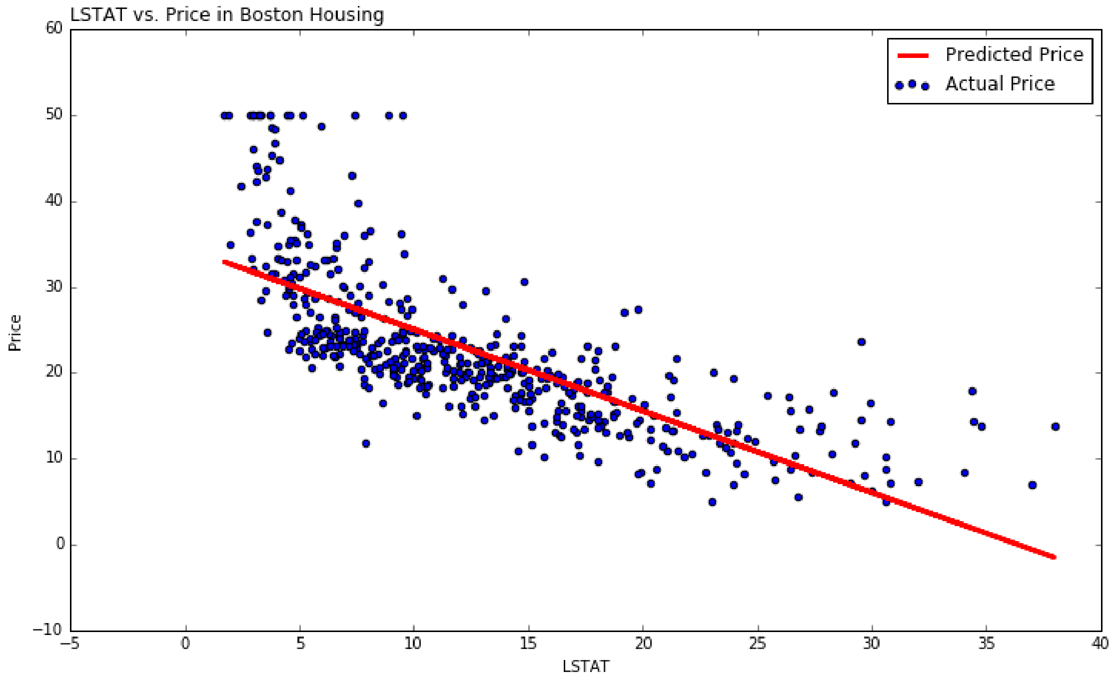
```
In [3]: boston = load_boston()
```

```
In [4]: X = boston.data
        y = boston.target
```

```
In [5]: bdf = pd.DataFrame(X, columns=boston.feature_names)
        bdf['price'] = y
```

```
In [6]: #fitting a scikitlearn model
        lr = LinearRegression()
        lr.fit(bdf[['LSTAT']], y)
        prds= lr.predict(bdf[['LSTAT']])
```

```
In [7]: #making a plot
        plt.figure(figsize = (12, 7))
        plt.scatter(bdf.LSTAT, bdf.price, label = 'Actual Price')
        plt.plot(bdf.LSTAT, prds, color = 'red', linewidth = 3, label = 'Predicted Price')
        plt.title('LSTAT vs. Price in Boston Housing', loc = 'left')
        plt.xlabel('LSTAT')
        plt.ylabel('Price')
        plt.legend(frameon = 'false')
        plt.savefig('presentation/img1.png')
```



0.2 Model Expression

$$w_{LS} = (X^T X)^{-1} X^T y$$

```
In [8]: #calculate product inside parenthesis
        xtx = X.T@X
```

```
In [9]: #find inverse
        inv = np.linalg.inv(xtx)
```

```
In [10]: #compute weights
         wLS = inv@X.T@y
```

0.3 Writing a Function

take in X and y

compute xtx

compute inverse

compute weights

return weights

```
In [11]: def wLS(X, y):
             '''
             This function provides an
             ordinary least squares fit of a dataset X on
             a target variable y.
             ----
             X = input array of feature variables
             y = target array of feature variables
             returns
             array of weights for basic linear regression
             '''
             #check dimensions
             if X.shape[0] < X.shape[1]:
                 X = X.T
             #calculate product inside parenthesis
             xtx = X.T@X
             #find inverse
             inv = np.linalg.inv(xtx)
             #compute weights
             prod = inv@X.T
             wLS = prod@y
             return wLS
```

```
In [12]: wLS(X, y)
```

```
Out[12]: array([-9.16297843e-02,  4.86751203e-02, -3.77930006e-03,  2.85636751e+00,
               -2.88077933e+00,  5.92521432e+00, -7.22447929e-03, -9.67995240e-01,
                1.70443393e-01, -9.38925373e-03, -3.92425680e-01,  1.49832102e-02,
               -4.16972624e-01])
```

0.4 Classes in Python

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

```
In [13]: class MyClass:
             """A simple example class"""
             i = 12345

             def f(self):
                 return 'hello world'
         #create an instance
         class1 = MyClass()
```

```
In [14]: #use our method
         class1.f()
```

```
Out[14]: 'hello world'
```

0.4.1 __init__ and self

From the Python docs:

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named **init**(), like this:

```
def __init__(self):
    self.data = []
```

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart

x = Complex(3.0, -4.5)
x.r, x.i
```

```
In [15]: class Complex:
             '''
             This is a simple class that will
             return a complex number object.
             '''
             def __init__(self, realpart, imagpart):
                 self.r = realpart
                 self.i = imagpart

         x = Complex(3.0, -4.5)
         x.r, x.i
```

Out[15]: (3.0, -4.5)

```
In [16]: #a different class instance
         x2 = Complex(2.3, 5.6)
```

```
In [17]: #another class instance
         x2.r, x2.i
```

Out[17]: (2.3, 5.6)

0.5 Our Regression Class

```
class Regression:
    '''
    This class contains basic linear
    regression capabilities.
    - OLS fit a linear regression model
    - make predictions with the model
    '''
    def __init__(self, coefs_, intercept_):
        self.coefs_ = None
        self.intercept = None

    def OLS(self, X, y)
        ...
        self.coefs_ = wLS
        return wLS

    def predict(self, X):
        return predictions
```

```
In [18]: class Regression:

             def __init__(self, fit_intercept = True):
                 self.coefs_ = None
                 self.intercept_ = None
                 self._fit_intercept = fit_intercept


             def OLS(self, X, y):
                 '''
                 This function provides an
                 ordinary least squares fit of a dataset X on
                 a target variable y.
                 ----
                 X = input array of feature variables
                 y = target array of feature variables
                 returns
                 array of weights for basic linear regression
```

```
            '''
            # Check shapes of input matricies.
            if X.shape[0] < X.shape[1]:
                X = X.T
            if y.shape[0] < y.shape[1]:
                y = y.T

            # Prepend ones to x matrix
            if self._fit_intercept:
                ones = np.ones((len(y), 1), dtype=int)
                X = np.concatenate((ones, X), axis=1)
            else:
                X = X
            # fit the model
            xtx = X.T@X
            inv = np.linalg.inv(xtx)
            w_ls = inv@X.T@y
            # add intercepts and coefs
            self.intercept_ = w_ls[:1]
            self.coefs_ = w_ls[1:]
            return w_ls
```

In [19]:
```
#some test cases
lr = Regression()
```

In [20]:
```
X = np.array([[0, 2], [3, 7], [5, 9], [3.4,6]])
y = np.array([[2.1, 3.2, 4, 5.6]])
```

In [21]:
```
lr.OLS(X, y)
```

Out[21]:
```
array([[ 5.3       ],
       [ 2.41150442],
       [-1.4079646 ]])
```

In [22]:
```
#model without intercept
lr2 = Regression(fit_intercept=False)
```

In [23]:
```
lr2.OLS(X, y)
```

Out[23]:
```
array([[-0.1530117],
       [ 0.6436483]])
```

0.5.1 Predict

$$\hat{y} = \beta_0 + X\beta_i$$

```
def predict(self, X):
    return self.intercept_ + X@self.coef_
```

In [24]:
```
del(Regression)
```

In [25]:
```
class Regression:

    def __init__(self, fit_intercept = True):
        self.coefs_ = None
        self.intercept_ = None
        self._fit_intercept = fit_intercept


    def wLS(self, X, y):
        '''
        This function provides an
        ordinary least squares fit of a dataset X on
        a target variable y.
        ----
        X = input array of feature variables
        y = target array of feature variables
        returns
        array of weights for basic linear regression
        '''
        # Check shapes of input matricies.
        if X.shape[0] < X.shape[1]:
            X = X.T
        try:
            if y.shape[0] < y.shape[1]:
                y = y.T
        except:
            pass
        # Prepend ones to x matrix
        if self._fit_intercept:
            ones = np.ones((len(y), 1), dtype=int)
            X = np.concatenate((ones, X), axis=1)
        else:
            X = X
        # fit the model
        xtx = X.T@X
        inv = np.linalg.inv(xtx)
        w_ls = inv@X.T@y
        # add intercepts and coefs
        self.intercept_ = w_ls[0]
        self.coefs_ = w_ls[1:]
        return w_ls

    def predict(self, X):
        return self.intercept_ + X@self.coefs_
```

In [26]:
```
del(lr)
```

In [27]:
```
lr = Regression()
lr.wLS(X, y)
```

```
Out[27]: array([[ 5.3       ],
                [ 2.41150442],
                [-1.4079646 ]])
```

```
In [28]: lr.coefs_
```

```
Out[28]: array([[ 2.41150442],
                [-1.4079646 ]])
```

```
In [29]: lr.intercept_
```

```
Out[29]: array([5.3])
```

```
In [30]: lr.predict(X)
```

```
Out[30]: array([[2.4840708 ],
                [2.67876106],
                [4.68584071],
                [5.05132743]])
```

```
In [31]: #testing on bigger data
         X = boston.data
         y = boston.target
```

```
In [32]: bos_reg = Regression()
```

```
In [33]: bos_reg.wLS(X, y)
```

```
Out[33]: array([ 3.64911033e+01, -1.07170557e-01,  4.63952195e-02,  2.08602395e-02,
                 2.68856140e+00, -1.77957587e+01,  3.80475246e+00,  7.51061703e-04,
                -1.47575880e+00,  3.05655038e-01, -1.23293463e-02, -9.53463555e-01,
                 9.39251272e-03, -5.25466633e-01])
```

0.5.2 Error

Sum of squared error: $\sum_{i=1}^{n}(\hat{y} - y_i)^2$
Total sum of squared error: $\sum_{i=1}^{n}(\bar{y} - y_i)^2$
$\mathbf{r}^2 = 1 - \frac{sse}{tss}$
Mean Squared Error: $\frac{1}{n}\sum_{i=1}^{n}(\hat{y} - y_i)^2$

```
In [34]: def r2(actual_y, predicted_y):
             sse = np.sum((predicted_y - actual_y)**2)
             tse = np.sum((actual_y - np.mean(actual_y))**2)
             return 1 - sse/tse
```

```
In [35]: def mse(actual_y, predicted_y):
             return np.mean((actual_y - predicted_y)**2)
```

```
In [36]: class Metrics:

             def __init__(self, X, y, model):
                 self.data = X
                 self.target = y
                 self.model = model

             def r2(self):
                 squared_errors = (self.target - self.model.predict(self.data))**2
                 sse = np.sum(squared_errors)
                 tse = np.sum((self.target - np.mean(self.target))**2)
                 return 1 - sse/tse

             def mse(self):
                 return np.mean((self.target - self.model.predict(self.data))**2)

             def rmse(self):
                 return self.mse()**0.5

             def summary_printed(self):
                 print('The r2 score is {:.4}\nThe Mean Squared Error is {:.4}\nand the RMSE is
```

```
In [37]: lr = Regression()
         lr.wLS(X, y)
         performance = Metrics(X, y, lr)
```

```
In [38]: performance.summary_printed()
```

```
The r2 score is 0.7406
The Mean Squared Error is 21.9
and the RMSE is 4.68
```

$$y = mx + b$$

0.5.3 Polynomial Features

$$y = a + bx_i + cx_i^2$$

```
In [39]: from sklearn.preprocessing import PolynomialFeatures
         x = np.array([2, 3, 4])
         poly = PolynomialFeatures(3, include_bias=False)
         poly.fit_transform(x[:, None])
```

```
Out[39]: array([[ 2.,  4.,  8.],
                [ 3.,  9., 27.],
                [ 4., 16., 64.]])
```

0.5.4 sklearn pipelines

In [40]:
```
from sklearn.pipeline import make_pipeline
poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
```

In [41]:
```
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3, include_bias=False)
poly.fit_transform(x[:, None])
```

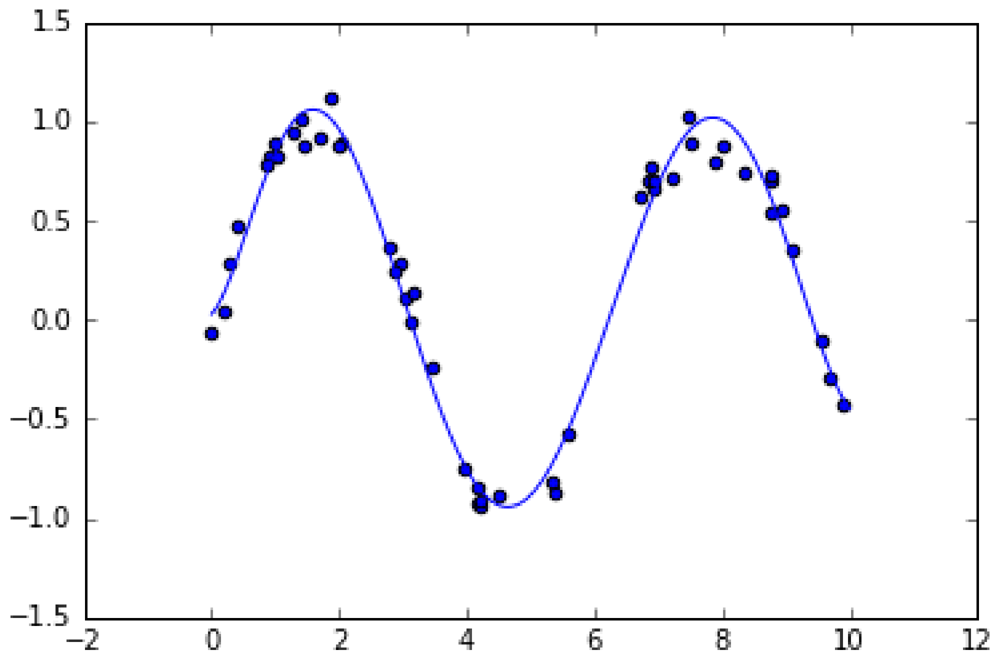
Out[41]:
```
array([[  2.,   4.,   8.],
       [  3.,   9.,  27.],
       [  4.,  16.,  64.]])
```

In [42]:
```
from sklearn.pipeline import make_pipeline
poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
```

In [43]:
```
#fitting a high order polynomial to sin with noise
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
xfit = np.linspace(0, 10, 1000)

poly_model.fit(x[:, np.newaxis], y)
yfit = poly_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit);
```



0.5.5 Any Basis

In [44]:
```
from sklearn.base import BaseEstimator, TransformerMixin

#a class for generating features from gaussian basis
class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                 self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);
```