Multiple Linear Regression

Regularization

Topics

- Why gradient descent?
- Types of gradient descent
- Bias vs Variance
- Regularization
    - Lasso (L1)
    - Ridge (L2)

Notation (1)

$$\hat{y} = h_\theta(X) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots$$

Where,

h_θ is our hypothesis

θ_0 is the bias or the *y*-intercept

$\theta_1, \theta_2, \ldots$ are the parameters

$x_1, x_2, \ldots$ are the features

$x_0 = 1$ for mathematical simplicity

$\hat{y}$ is the variable you're predicting

Notation (2)

House Price Prediction Dataset:

Size (feet^2) x_1	Number of Bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price (\$1000) y
2104	5	1	45	460
1420	4	1	35	240
…	…	…	…	…

n = number of features = 4

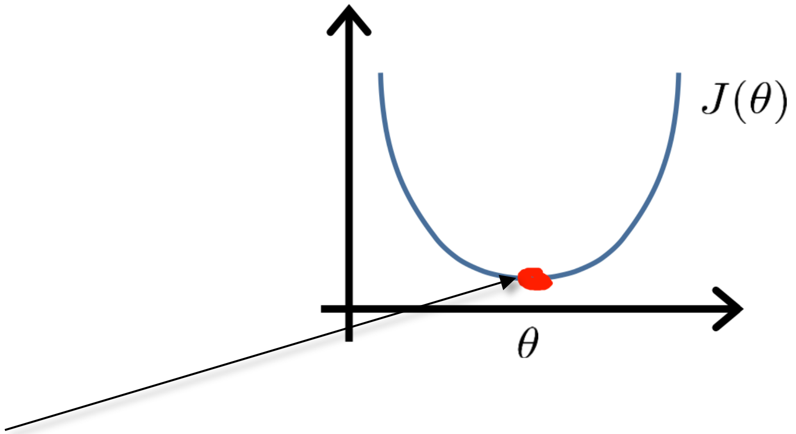
x_i= *i*-th feature

y = output or the Dependent variable

Why Gradient Descent? (1)

Normal Equations: Method to solve for θ analytically

Intuition:



Find the point where $J(\theta)$ is the lowest.

Which is when $\frac{dJ(\theta)}{dx} = 0$

Why Gradient Descent? (2)

Size x_1	**No. of Bedrooms** x_2	**No. of floors** x_3	**Age of home** x_4	**Price** y
2104	5	1	45	460
1420	4	1	35	240

Let's say,

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ x_3^{(i)} \\ x_4^{(i)} \end{bmatrix} \Rightarrow x^{(1)} = \begin{bmatrix} 1 \\ 2104 \\ 5 \\ 1 \\ 45 \end{bmatrix} \Rightarrow \left(x^{(1)}\right)^T = [1 \quad 2104 \quad 5 \quad 1 \quad 45]$$

Now,

$$X = \begin{bmatrix} - & \left(x^{(1)}\right)^T & - \\ - & \left(x^{(2)}\right)^T & - \end{bmatrix} = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1420 & 4 & 1 & 35 \end{bmatrix} \text{ and } y = \begin{bmatrix} 460 \\ 240 \end{bmatrix}$$

Why Gradient Descent? (3)

Using Normal Equations method,

$$\theta = (X^T X)^{-1} X^T y$$

Where θ is a column vector

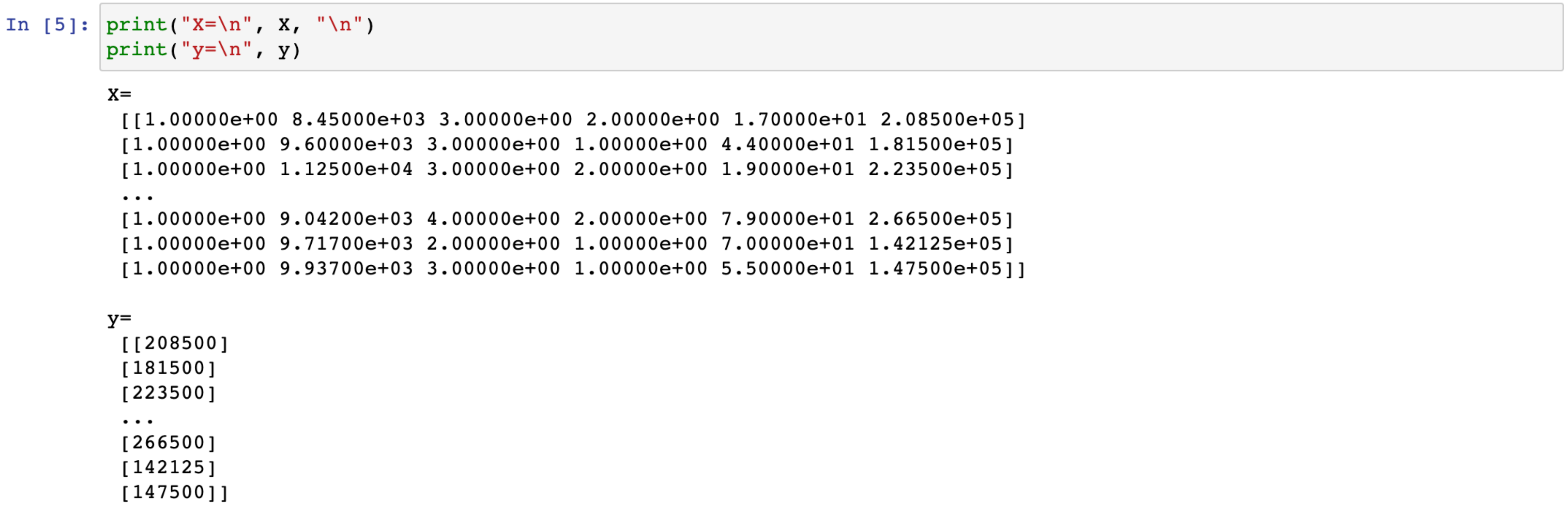
Derivation:

[1] https://medium.com/swlh/understanding-mathematics-behind-normal-equation-in-linear-regression-aa20dc5a0961

[2] https://www.geeksforgeeks.org/ml-normal-equation-in-linear-regression/

[3] https://ayearofai.com/rohan-3-deriving-the-normal-equation-using-matrix-calculus-1a1b16f65dda

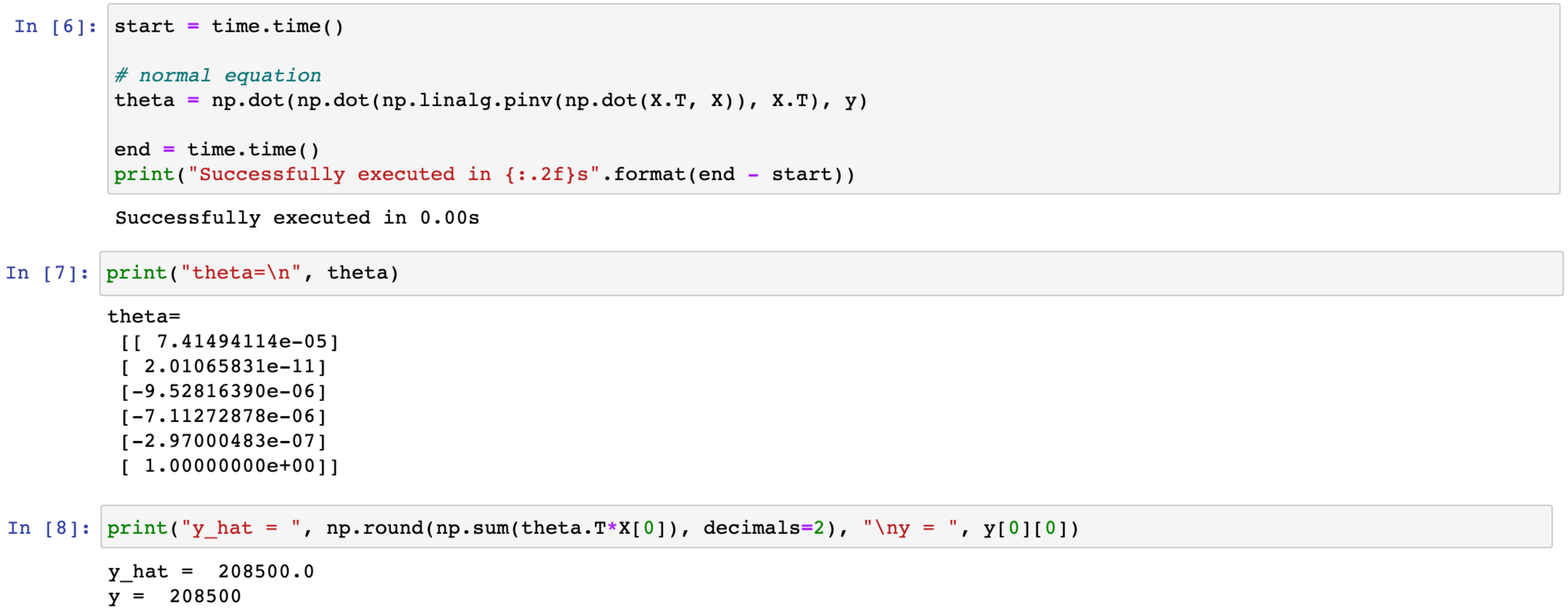
Why Gradient Descent? (4)



```
In [5]: print("X=\n", X, "\n")
        print("y=\n", y)
```

```
X=
 [[1.00000e+00 8.45000e+03 3.00000e+00 2.00000e+00 1.70000e+01 2.08500e+05]
 [1.00000e+00 9.60000e+03 3.00000e+00 1.00000e+00 4.40000e+01 1.81500e+05]
 [1.00000e+00 1.12500e+04 3.00000e+00 2.00000e+00 1.90000e+01 2.23500e+05]
 ...
 [1.00000e+00 9.04200e+03 4.00000e+00 2.00000e+00 7.90000e+01 2.66500e+05]
 [1.00000e+00 9.71700e+03 2.00000e+00 1.00000e+00 7.00000e+01 1.42125e+05]
 [1.00000e+00 9.93700e+03 3.00000e+00 1.00000e+00 5.50000e+01 1.47500e+05]]

y=
 [[208500]
 [181500]
 [223500]
 ...
 [266500]
 [142125]
 [147500]]
```

Why Gradient Descent? (5)



In [6]:
```
start = time.time()

# normal equation
theta = np.dot(np.dot(np.linalg.pinv(np.dot(X.T, X)), X.T), y)

end = time.time()
print("Successfully executed in {:.2f}s".format(end - start))
```

```
Successfully executed in 0.00s
```

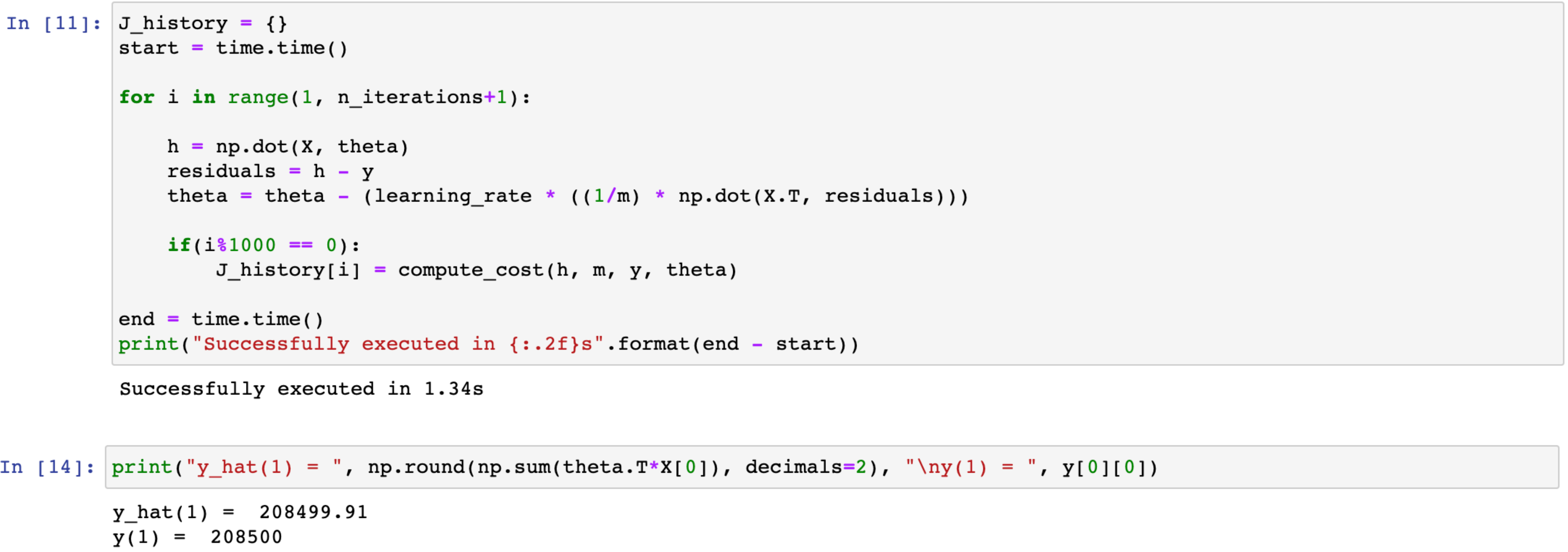
In [7]:
```
print("theta=\n", theta)
```

```
theta=
 [[ 7.41494114e-05]
 [ 2.01065831e-11]
 [-9.52816390e-06]
 [-7.11272878e-06]
 [-2.97000483e-07]
 [ 1.00000000e+00]]
```

In [8]:
```
print("y_hat = ", np.round(np.sum(theta.T*X[0]), decimals=2), "\ny = ", y[0][0])
```

```
y_hat =  208500.0
y =  208500
```

Why Gradient Descent? (6)



```
In [11]: J_history = {}
         start = time.time()

         for i in range(1, n_iterations+1):

             h = np.dot(X, theta)
             residuals = h - y
             theta = theta - (learning_rate * ((1/m) * np.dot(X.T, residuals)))

             if(i%1000 == 0):
                 J_history[i] = compute_cost(h, m, y, theta)

         end = time.time()
         print("Successfully executed in {:.2f}s".format(end - start))
```

```
Successfully executed in 1.34s
```

```
In [14]: print("y_hat(1) = ", np.round(np.sum(theta.T*X[0]), decimals=2), "\ny(1) = ", y[0][0])
```

```
y_hat(1) =  208499.91
y(1) =  208500
```

Why Gradient Descent? (7)

Gradient Descent

✗ Need to choose α (learning rate)
✗ Needs many iterations

✓ Works well even with large n
✓ $O(kn^2)$

Normal Equations Method

✓ No need to choose α
✓ Don’t need to iterate

✗ Doesn’t work well even with large n
✗ $O(n^3)$

Types of Gradient Descent (1)

Intention:

- To allow for trivial parallelization
- Keep only a piece of data in memory

Types:

1. Batch Gradient Descent: Use all examples in each iteration

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) \cdot x^{(i)}$$

} for n iterations

Types of Gradient Descent (2)

2. Stochastic Gradient Descent: Use 1 example in each iteration

```
Randomly shuffle the data
Repeat {
        for $i = 1, \dots, m${
                $\theta_j := \theta_j - \alpha\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x^{(i)}$
                }
} for $n$ iterations
```

Types of Gradient Descent (3)

3. Mini-batch Gradient Descent: Use b examples in each iteration

```
Randomly shuffle and split the data into batches of size $b$
Repeat {
        for each batch $b${
                $\theta_j := \theta_j - \alpha \frac{1}{m} \underbrace{\sum_{i=b\_start}^{b\_end} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) \cdot x^{(i)}}$
                }
} for $n$ iterations
```

Compute this for b_0 $i = 0 \dots 99$	Compute this for b_1 $i = 100 \dots 199$	Compute this for b_2 $i = 200 \dots 299$

Bias vs Variance (1) - Bias

- Assumptions made by a model to make the target function easier to learn
- Generally, linear algorithms have a high bias:
  - **Fast to learn** and **easier to understand**
  - But **less flexible**
  - Thus, **low predictive performance**
- **Low Bias**: Suggests **fewer assumptions** about the form of the target function.
- **High Bias**: Suggests **more assumptions** about the form of the target function.
- E.g. Linear Regression assumes:
  - Normality of residuals around 0
  - No multicollinearity
  - No autocorrelation of residuals
  - Homoscedasticity of residuals

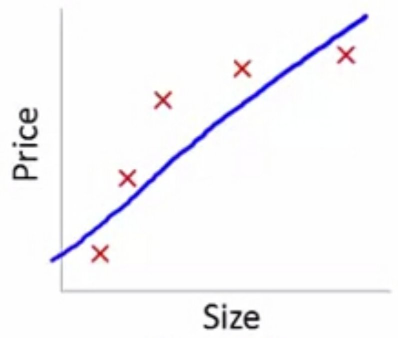
Bias vs Variance (2) - Variance

- Amount that the estimate of the target function will change if different training data was used.
- **specifics of the training influences** the number and types of parameters used to characterize the mapping function
- **Low Variance**: Suggests **small changes** to the estimate of the target function with changes to the training dataset.
- **High Variance**: Suggests **large changes** to the estimate of the target function with changes to the training dataset.
- Generally, **nonlinear** machine learning algorithms that have a **lot of flexibility** have a high variance.
- E.g., decision trees have a high variance, that is even higher if the trees are not pruned before use.

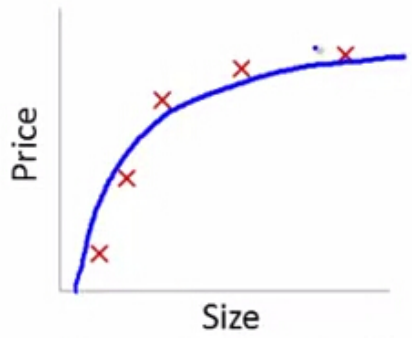
Bias vs Variance (3)

- There is no escaping the relationship between bias and variance in machine learning.
    - Increasing the bias will decrease the variance.
    - Increasing the variance will decrease the bias.

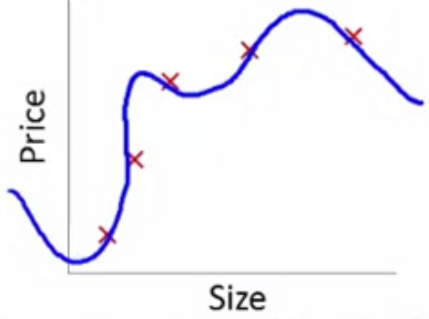
Bias vs Variance (4)



$\theta_0 + \theta_1 x$



$\theta_0 + \theta_1 x + \theta_2 x^2$



$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

High bias
(underfit)
Cannot capture underlying trend

“Just right”

High variance
(overfit)
Captures the noise of the data

Bias vs Variance (4)

- Ways to treat high bias:
  - DO NOT get more data
  - Add more features
  - Add polynomial features
  - Decreasing λ
  - Use a non-linear algorithm e.g. decision trees

- Ways to treat high Variance:
  - Get more data
  - Reduce number of features
  - Increasing λ

Regularization (1)

- Helps solve overfitting problem
- **Intuition:**
  - Large weights tend to cause overfitting
  - Large weights are more sensitive to small noises
  - in the feature space, only directions along which the parameters contribute significantly to reducing the objective function are preserved
  - E.g. if number of bedrooms is not contributing to price, no point in assigning a large weight to it
- adding a penalty term to the objective function

Regularization (2)

L^P Norm:

$$\|x\|_p = \left(\sum_i |x_i|^p\right)^{1/p}$$

E.g. L1 norm

$$\|x\|_1 = \sum_i |x_i|$$

E.g. L2 norm in linear regression

$$\|x\|_2 = \sum_i x_i^2$$

By not introducing the square root, the gradient has a more elegant form (computationally simple)

Regularization (3)

- Lasso regression or L1 Regularization minimizes:

$$\mathrm{OLS} + \lambda \sum_{j} |\theta_j|$$

- Ridge regression or L2 Regularization minimizes:

$$\mathrm{OLS} + \lambda \sum_{j} {\theta_j}^2$$

Where,
λ is the tuning parameter, greater this is, the higher the penalty is
OLS are the ordinary least square errors or the cost we used previously

Regularization (4)

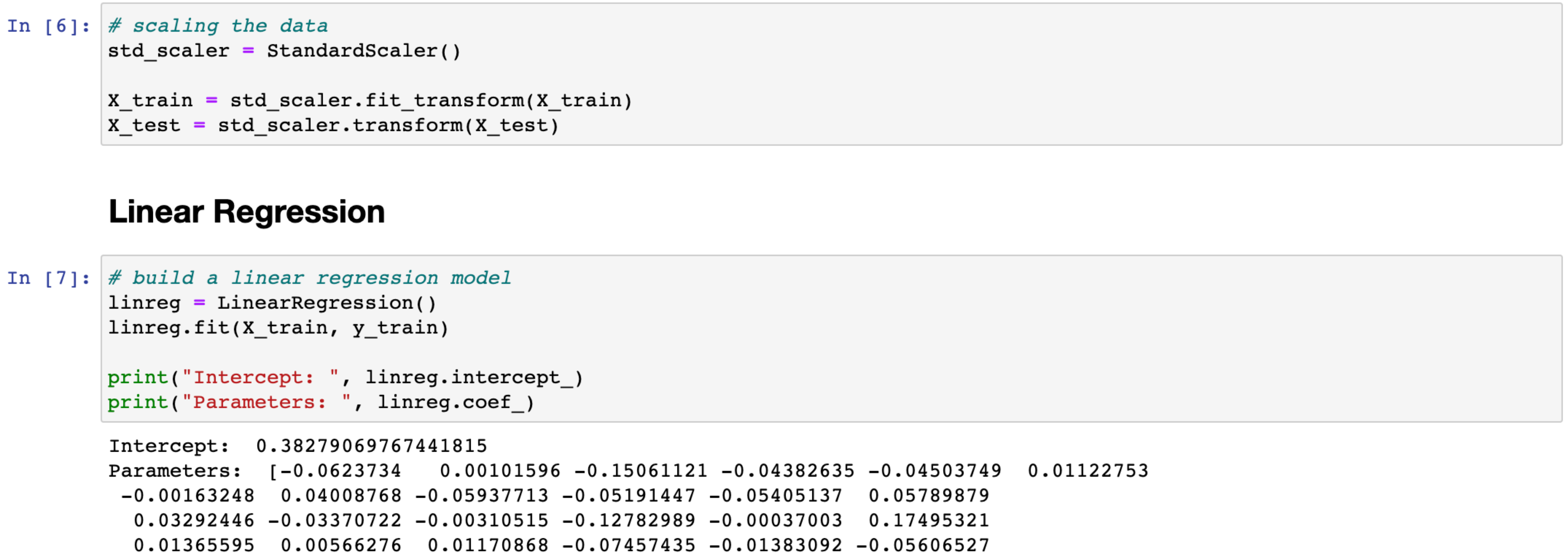
- Derive gradient descent equations with new regularization parameter

- Always standardized, because otherwise, features would be penalized simply because of their scale.
- Standardizing features will also help gradient descent converge faster
- Standardizing features/Feature scaling:

Shifting the features to a mean 0 and standard deviation of 1 (or some equivalent form):

$$x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}$$

Regularization (5) – Linear Regression



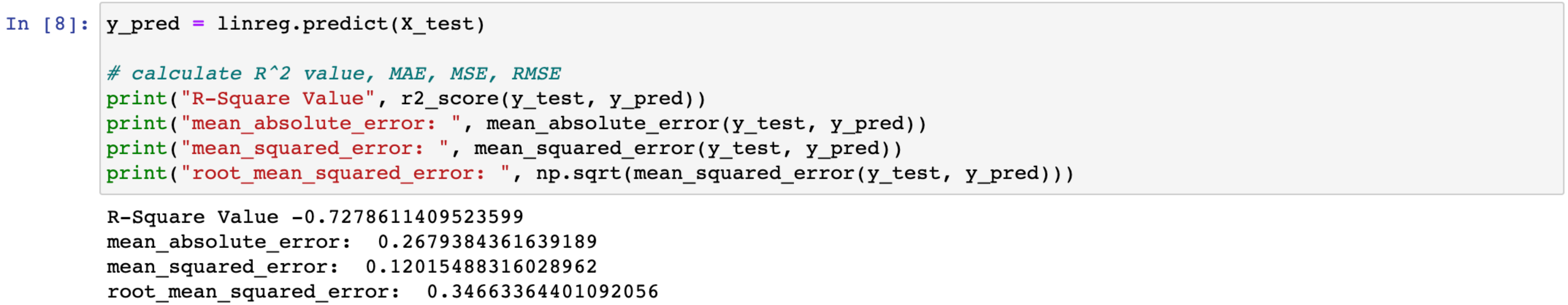
```
In [6]: # scaling the data
        std_scaler = StandardScaler()

        X_train = std_scaler.fit_transform(X_train)
        X_test = std_scaler.transform(X_test)
```

Linear Regression

```
In [7]: # build a linear regression model
        linreg = LinearRegression()
        linreg.fit(X_train, y_train)

        print("Intercept: ", linreg.intercept_)
        print("Parameters: ", linreg.coef_)
```

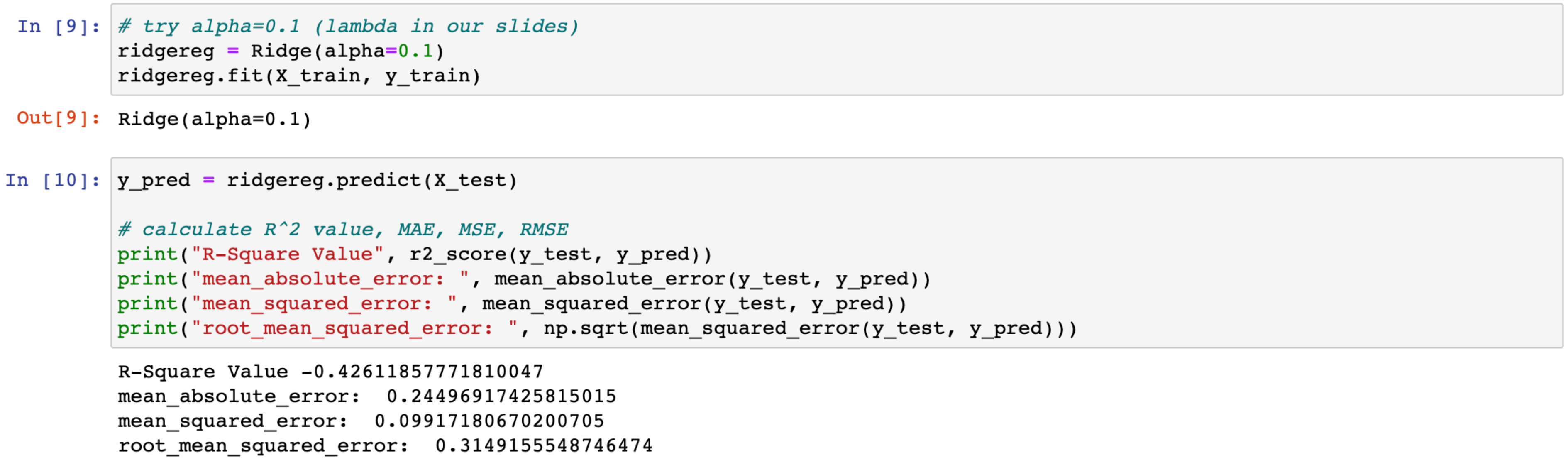
```
Intercept:  0.38279069767441815
Parameters:  [-0.0623734   0.00101596 -0.15061121 -0.04382635 -0.04503749  0.01122753
 -0.00163248  0.04008768 -0.05937713 -0.05191447 -0.05405137  0.05789879
  0.03292446 -0.03370722 -0.00310515 -0.12782989 -0.00037003  0.17495321
  0.01365595  0.00566276  0.01170868 -0.07457435 -0.01383092 -0.05606527
```

Regularization (6) – Linear Regression



```
In [8]: y_pred = linreg.predict(X_test)

        # calculate R^2 value, MAE, MSE, RMSE
        print("R-Square Value", r2_score(y_test, y_pred))
        print("mean_absolute_error: ", mean_absolute_error(y_test, y_pred))
        print("mean_squared_error: ", mean_squared_error(y_test, y_pred))
        print("root_mean_squared_error: ", np.sqrt(mean_squared_error(y_test, y_pred)))

        R-Square Value -0.7278611409523599
        mean_absolute_error:  0.2679384361639189
        mean_squared_error:  0.12015488316028962
        root_mean_squared_error:  0.34663364401092056
```

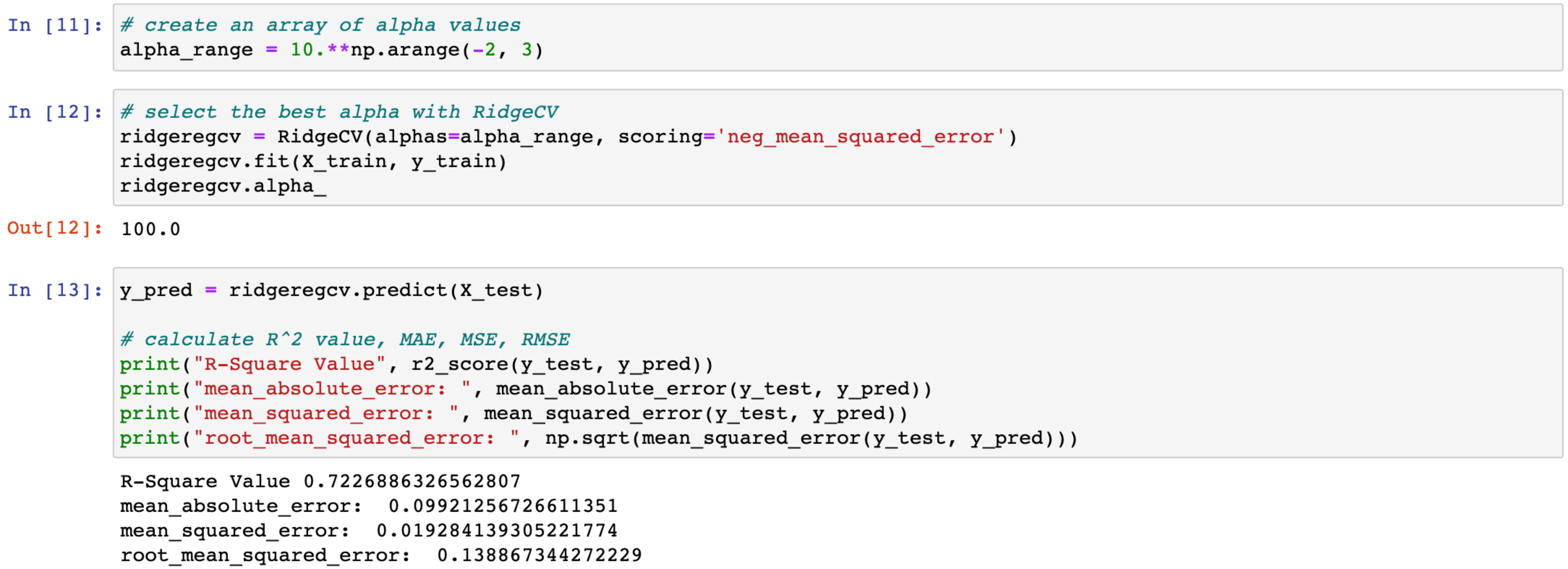
Regularization (7) – Ridge Regression



```
In [9]:  # try alpha=0.1 (lambda in our slides)
         ridgereg = Ridge(alpha=0.1)
         ridgereg.fit(X_train, y_train)

Out[9]:  Ridge(alpha=0.1)
```

```
In [10]: y_pred = ridgereg.predict(X_test)

         # calculate R^2 value, MAE, MSE, RMSE
         print("R-Square Value", r2_score(y_test, y_pred))
         print("mean_absolute_error: ", mean_absolute_error(y_test, y_pred))
         print("mean_squared_error: ", mean_squared_error(y_test, y_pred))
         print("root_mean_squared_error: ", np.sqrt(mean_squared_error(y_test, y_pred)))

         R-Square Value -0.42611857771810047
         mean_absolute_error:  0.24496917425815015
         mean_squared_error:  0.09917180670200705
         root_mean_squared_error:  0.3149155548746474
```

Regularization (7) – RidgeCV Regression

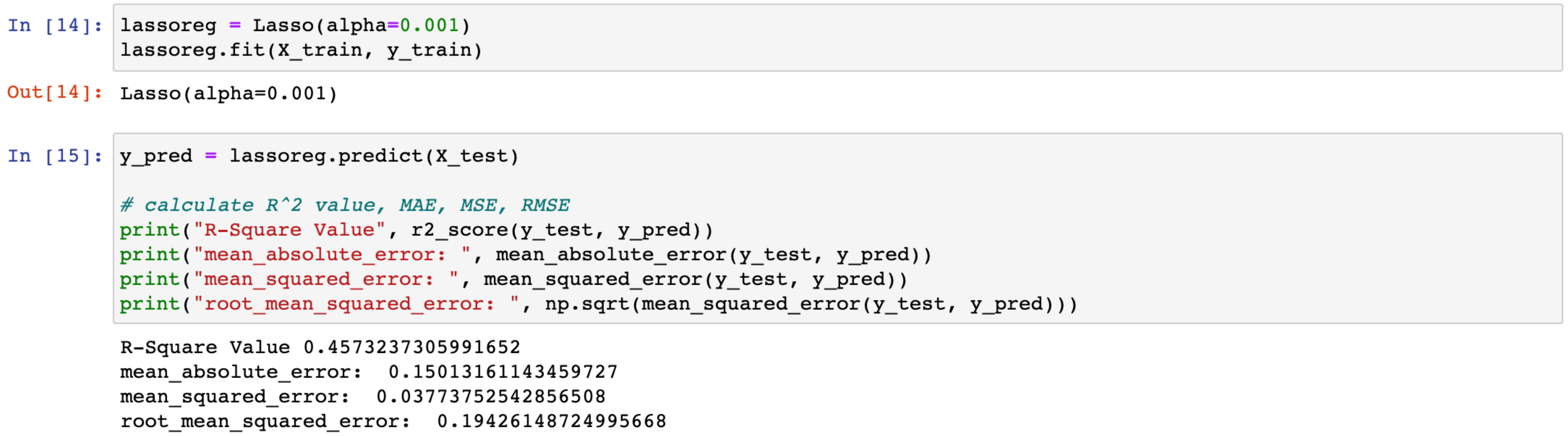


```
In [11]: # create an array of alpha values
         alpha_range = 10.**np.arange(-2, 3)
```

```
In [12]: # select the best alpha with RidgeCV
         ridgeregcv = RidgeCV(alphas=alpha_range, scoring='neg_mean_squared_error')
         ridgeregcv.fit(X_train, y_train)
         ridgeregcv.alpha_

Out[12]: 100.0
```

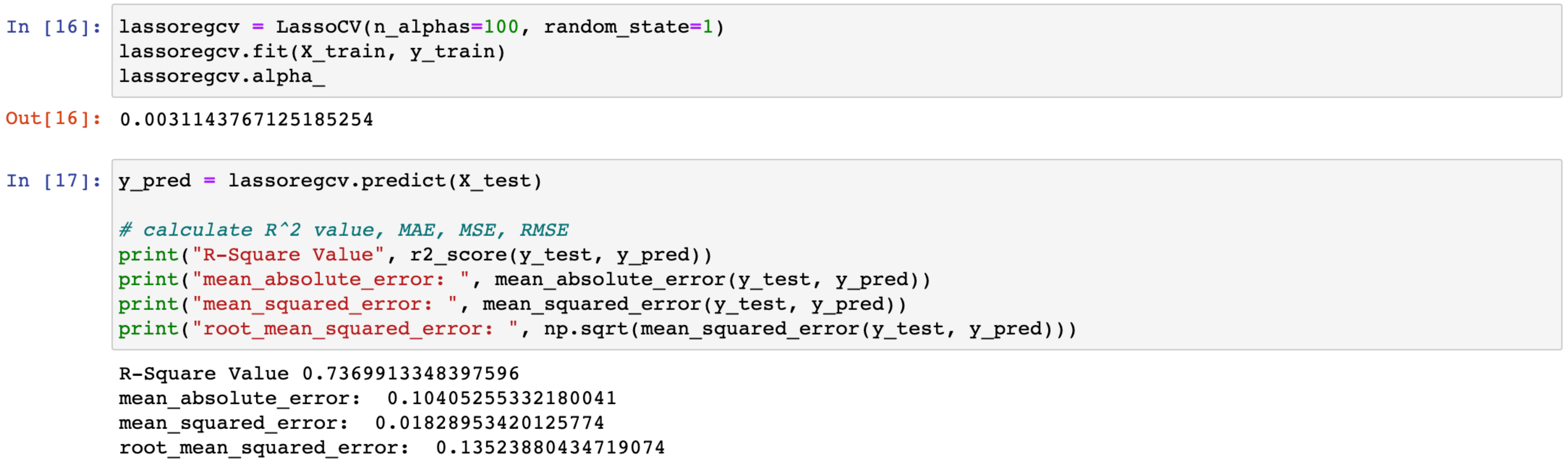
```
In [13]: y_pred = ridgeregcv.predict(X_test)

         # calculate R^2 value, MAE, MSE, RMSE
         print("R-Square Value", r2_score(y_test, y_pred))
         print("mean_absolute_error: ", mean_absolute_error(y_test, y_pred))
         print("mean_squared_error: ", mean_squared_error(y_test, y_pred))
         print("root_mean_squared_error: ", np.sqrt(mean_squared_error(y_test, y_pred)))

         R-Square Value 0.7226886326562807
         mean_absolute_error:  0.09921256726611351
         mean_squared_error:  0.019284139305221774
         root_mean_squared_error:  0.138867344272229
```

Regularization (8) – Lasso Regression



```
In [14]: lassoreg = Lasso(alpha=0.001)
         lassoreg.fit(X_train, y_train)

Out[14]: Lasso(alpha=0.001)

In [15]: y_pred = lassoreg.predict(X_test)

         # calculate R^2 value, MAE, MSE, RMSE
         print("R-Square Value", r2_score(y_test, y_pred))
         print("mean_absolute_error: ", mean_absolute_error(y_test, y_pred))
         print("mean_squared_error: ", mean_squared_error(y_test, y_pred))
         print("root_mean_squared_error: ", np.sqrt(mean_squared_error(y_test, y_pred)))

         R-Square Value 0.4573237305991652
         mean_absolute_error:  0.15013161143459727
         mean_squared_error:  0.03773752542856508
         root_mean_squared_error:  0.19426148724995668
```

Regularization (9) – LassoCV Regression



```
In [16]: lassoregcv = LassoCV(n_alphas=100, random_state=1)
         lassoregcv.fit(X_train, y_train)
         lassoregcv.alpha_

Out[16]: 0.0031143767125185254
```

```
In [17]: y_pred = lassoregcv.predict(X_test)

         # calculate R^2 value, MAE, MSE, RMSE
         print("R-Square Value", r2_score(y_test, y_pred))
         print("mean_absolute_error: ", mean_absolute_error(y_test, y_pred))
         print("mean_squared_error: ", mean_squared_error(y_test, y_pred))
         print("root_mean_squared_error: ", np.sqrt(mean_squared_error(y_test, y_pred)))

         R-Square Value 0.7369913348397596
         mean_absolute_error:  0.10405255332180041
         mean_squared_error:  0.01828953420125774
         root_mean_squared_error:  0.13523880434719074
```

Regularization (10) - Comparison

	Linear Regression	**Ridge Regression**	**RidgeCV Regression**	**Lasso Regression**	**LassoCV Regression**
MAE	0.268	0.245	0.099	0.037	0.0031
RMSE	0.3466	0.3149	0.1389	0.1942	0.1352

Conclusion

- Always check to ensure conditions of MLR are met – normality of residuals, no autocorrelation of residuals, etc.
- Try out a series of models
- Grid search/Randomly search/Bayesian search for your hyperparameters – lambda and alpha
- Cross-validate your models
- Pick one with lowest error

References

[1] https://machinelearningmastery.com/gentle-introduction-to-the-bias-variance-trade-off-in-machine-learning/

[2] http://r-statistics.co/Assumptions-of-Linear-Regression.html

[3] https://datascience.stackexchange.com/questions/23287/why-large-weights-are-prohibited-in-neural-networks

[4] https://stats.stackexchange.com/questions/449748/why-does-l-2-norm-regularization-not-have-a-square-root

[5] https://medium.com/@harishreddyp98/regularization-in-python-699cfbad8622