Supervised Learning – Part 3

ESM3081 Programming for Data Science

Seokho Kang

성균관대학교
SUNG KYUN KWAN UNIVERSITY(SKKU)

Learning algorithms covered in this course

- **Supervised Learning** (Classification/Regression)
  - K-Nearest Neighbors
  - **Linear Models (Logistic/Linear Regression)**
  - Decision Trees
  - Random Forests
  - Support Vector Machines
  - Neural Networks

** Many algorithms have a classification and a regression variant, and we will describe both.*

** We will review the most popular machine learning algorithms, explain how they learn from data and how they make predictions, and examine the strengths and weaknesses of each algorithm.*

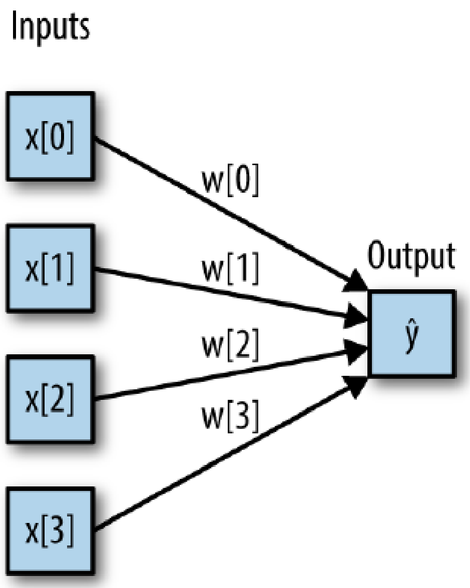
Linear Models

Linear Models

- **Linear models make a prediction using a *linear function* of the input features**

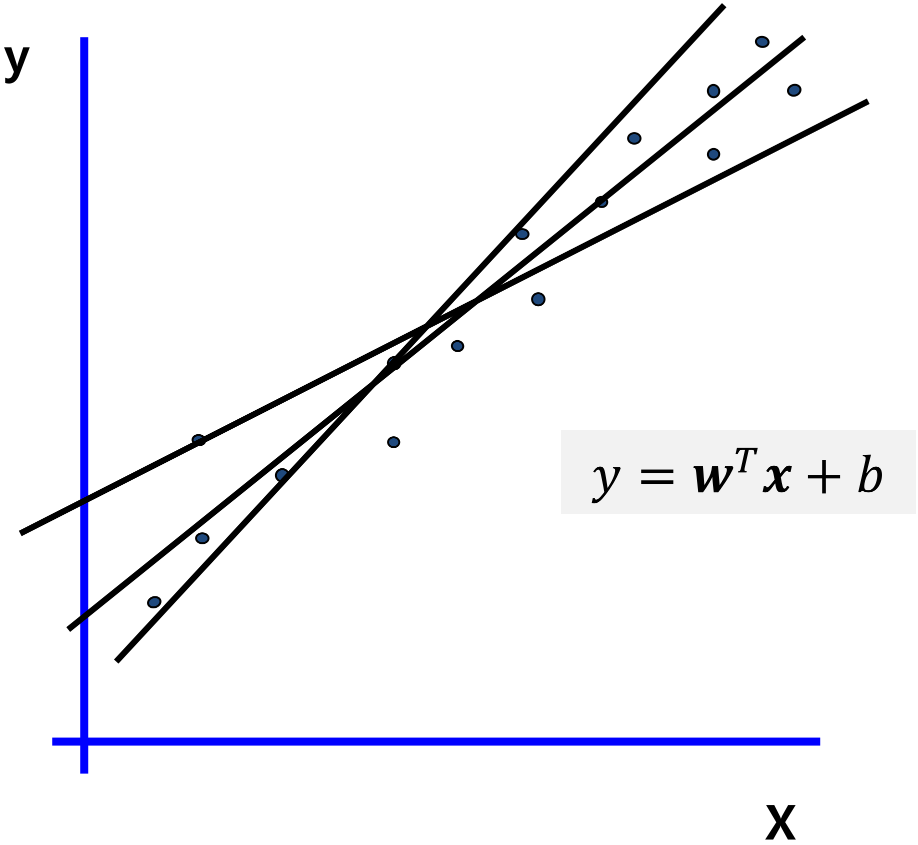
- **Learning algorithms for regression**
  - Linear Regression
  - Ridge Regression
  - Lasso Regression
  - Elastic Net Regression
  - Principal Component Regression
  - Partial Least Squares Regression
  - (Linear) Support Vector Regression
  - …

- **Learning algorithms for binary classification**
  - Logistic Regression
  - (Linear) Support Vector Machine
  - Linear Discriminant Analysis
  - …



Linear Regression

- For linear models for regression, the prediction $\hat{y}$ is a linear function of input features.



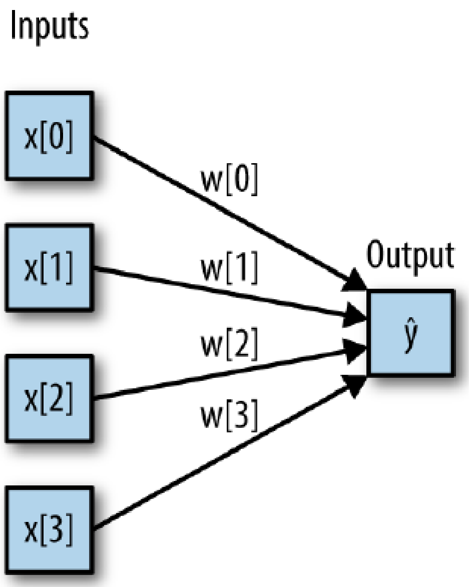
$$y = \boldsymbol{w}^T \boldsymbol{x} + b$$

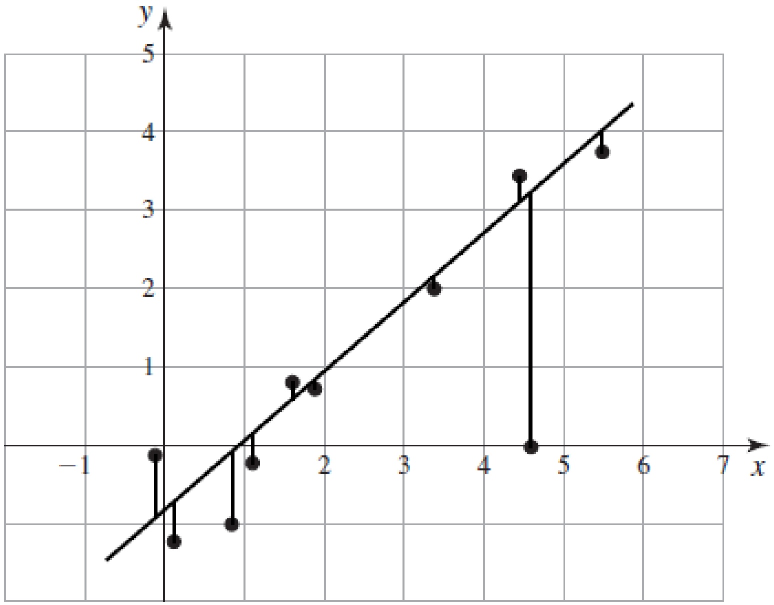
Linear Regression

- **Linear Regression (ordinary least squares (OLS))**
  - Linear regression finds the parameters $\mathbf{w}$ and b that minimize the *mean squared error* between predictions and the true regression targets on the training set.

$$\hat{y} = \mathbf{w}^T \boldsymbol{x} + b = w_1 x_1 + \cdots + w_d x_d + b$$
$$\boldsymbol{x} = (x_1, \dots, x_d) \in \mathbb{R}^d, \qquad y, \hat{y} \in \mathbb{R}$$

  - Linear regression has no hyperparameters, thus has no way to control model complexity.





Linear Regression

- Given a (training) dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (1, x_{i1}, \dots, x_{id}) \in \mathbb{R}^{d+1}$ is the *i*-th input vector of *d* features and $y_i \in \mathbb{R}$ is the corresponding target label.

- the first entry is always set to "1"

- The output of model f (prediction of y)
: $\hat{y} = f(\boldsymbol{x}) = \mathbf{w}^T\boldsymbol{x}$, where $\mathbf{w} = (w_0, w_1, \dots, w_d)$ is a vector of parameters.

- $w_1, \dots, w_d$ are called "coefficients" or "weights"
- w_0 is called "intercept" or "bias"

- **Training**: To find the optimal parameter $\mathbf{w}^*$ that minimizes the training error (cost function)

Here we use "squared error" loss $L(y, \hat{y}) = (\hat{y} - y)^2$, then $J(\mathbf{w}) = \mathrm{MSE}_{\mathrm{train}}$

$$J(\mathbf{w}) = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} L(y_i, \hat{y}_i) = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} (\hat{y}_i - y_i)^2 = \frac{1}{n} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2$$

- $\mathbf{X}$, $\mathbf{y}$ are matrix representation of D

Linear Regression

- **Training:** To find the optimal parameter $\mathbf{w}^*$ that minimizes the training error
  → **an optimization problem**

$$\mathrm{MSE}_{\mathrm{train}} = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} (\hat{y}_i - y_i)^2 = \frac{1}{n} \|\mathbf{Xw} - \mathbf{y}\|^{\mathbf{2}}$$

▶ how? set the gradient to 0 → a closed-form solution (normal equation)

$$\nabla_{\mathbf{w}} \mathrm{MSE}_{\mathrm{train}} = \frac{1}{n} \nabla_{\mathbf{w}} \|\mathbf{Xw} - \mathbf{y}\|^2 = 0$$

$$\ldots$$

$$\ldots$$

$$\ldots$$

$$\mathbf{w}^* = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1} \mathbf{X}^{\mathrm{T}} \mathbf{y}$$

- The trained model $f(\boldsymbol{x}) = \mathbf{w}^{*\mathrm{T}} \boldsymbol{x}$

Probabilistic Interpretation of Linear Regression

- **Probabilistic Interpretation of Linear Regression**
  - Assume $y \sim \mathcal{N}(\hat{y}, \sigma^2), \hat{y} = \mathbf{w}^T \boldsymbol{x}$

$$p(y|\boldsymbol{x}; \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - \mathbf{w}^T \boldsymbol{x})^2}{2\sigma^2}\right)$$

p.d.f. of $N(\hat{y},\ \sigma^2)$

- **Maximum Likelihood Estimation** (with respect to **w**)

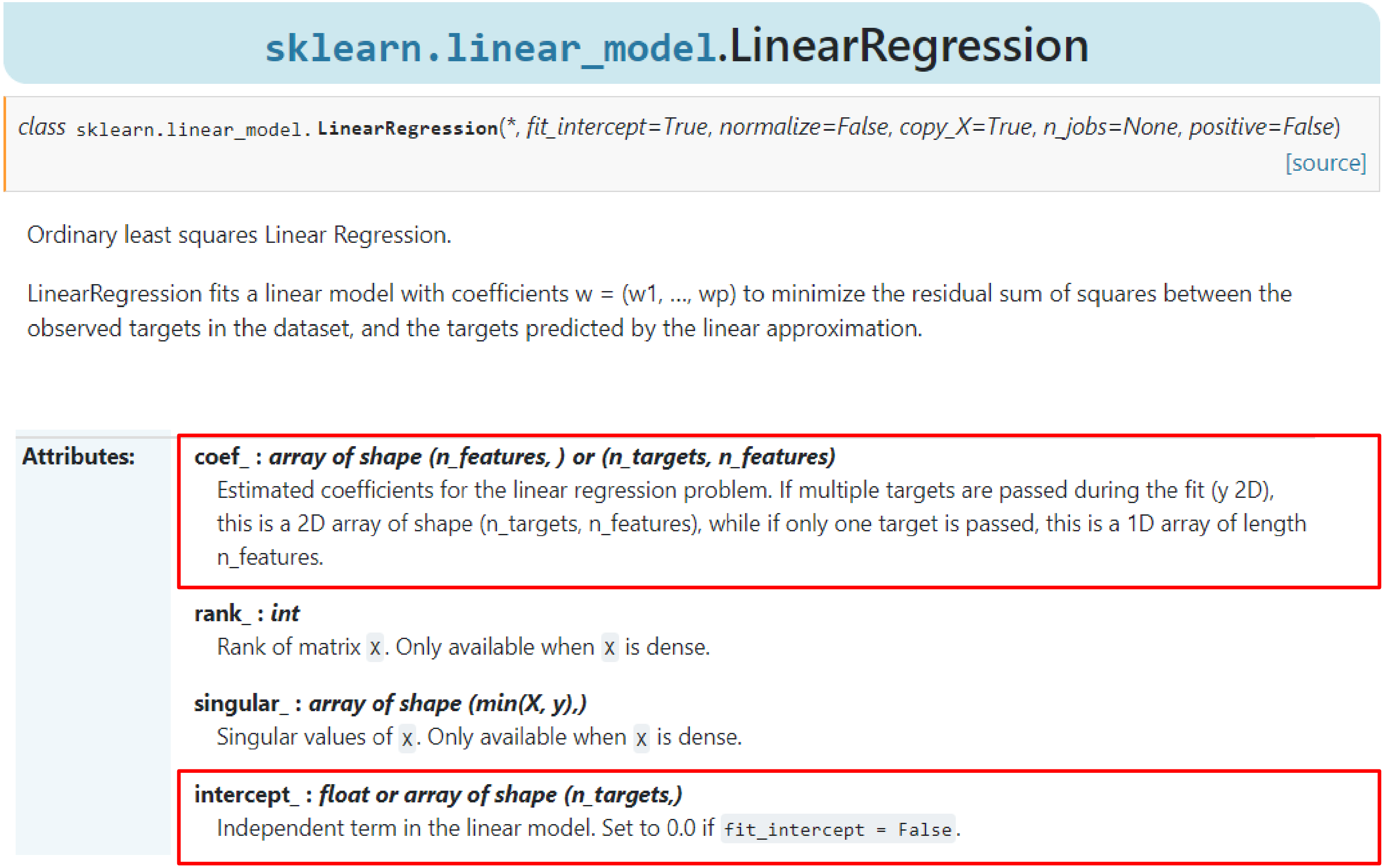
log-likelihood

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmax}} \prod_{(\boldsymbol{x}_i, y_i) \in D} p(y_i | \boldsymbol{x}_i; \mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{(\boldsymbol{x}_i, y_i) \in D} \log p(y_i | \boldsymbol{x}_i; \mathbf{w})$$

$$= \underset{\mathbf{w}}{\operatorname{argmax}} \left[-\frac{n}{2} \log 2\pi\sigma^2 - \frac{1}{2\sigma^2} \boxed{\sum_{(\boldsymbol{x}_i, y_i) \in D} (y_i - \mathbf{w}^T \boldsymbol{x}_i)^2} \right]$$

scikit-learn Practice: *LinearRegression*

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html



`sklearn.linear_model`.LinearRegression

class `sklearn.linear_model.`**`LinearRegression`**(**, fit_intercept=True, normalize=False, copy_X=True, n_jobs=None, positive=False*) [source]

Ordinary least squares Linear Regression.

LinearRegression fits a linear model with coefficients w = (w1, …, wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

Attributes:

coef_ : *array of shape (n_features,) or (n_targets, n_features)*
Estimated coefficients for the linear regression problem. If multiple targets are passed during the fit (y 2D), this is a 2D array of shape (n_targets, n_features), while if only one target is passed, this is a 1D array of length n_features.

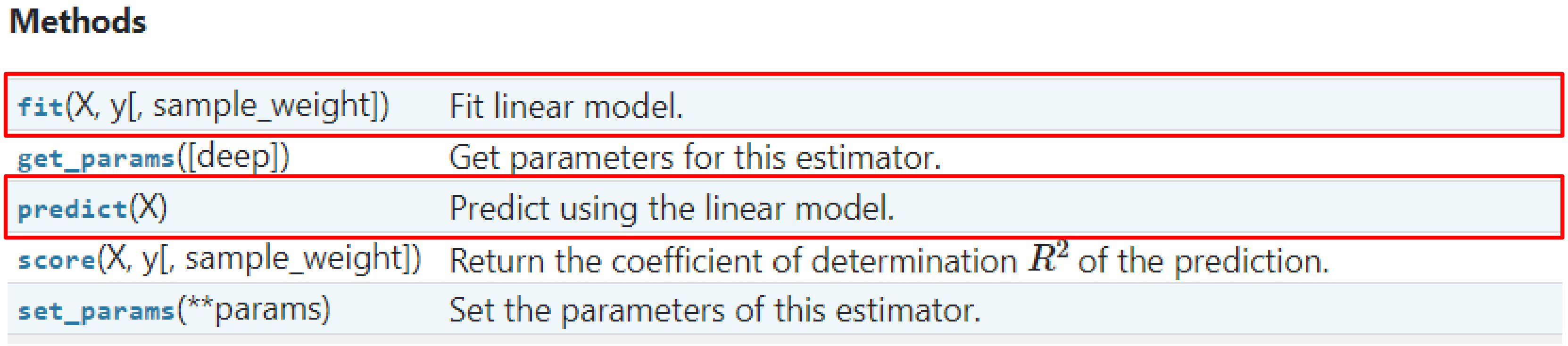
rank_ : *int*
Rank of matrix `X`. Only available when `X` is dense.

singular_ : *array of shape (min(X, y),)*
Singular values of `X`. Only available when `X` is dense.

intercept_ : *float or array of shape (n_targets,)*
Independent term in the linear model. Set to 0.0 if `fit_intercept = False`.

scikit-learn Practice: *LinearRegression*

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

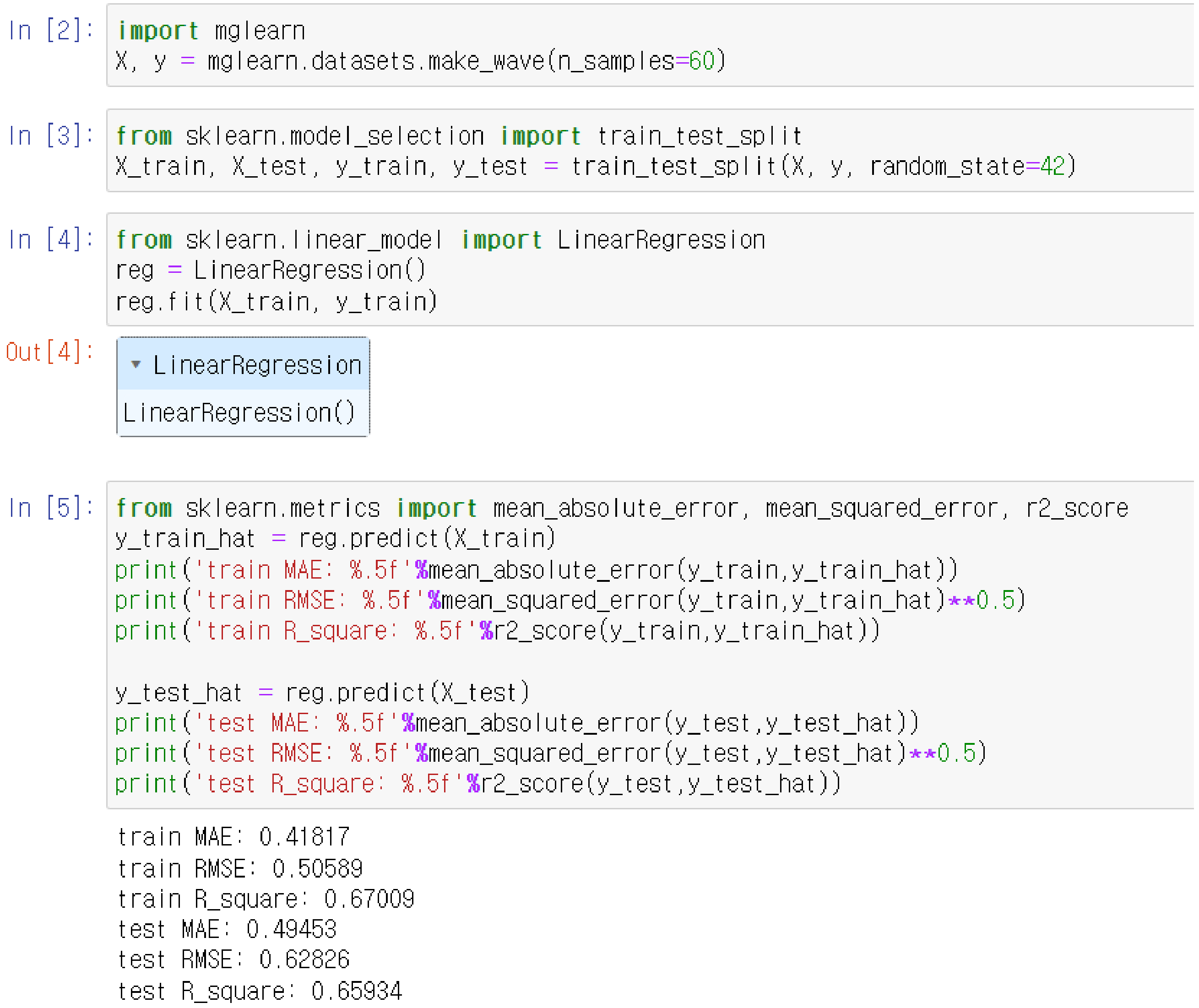


Methods

`fit`(X, y[, sample_weight])	Fit linear model.
`get_params`([deep])	Get parameters for this estimator.
`predict`(X)	Predict using the linear model.
`score`(X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
`set_params`(**params)	Set the parameters of this estimator.

scikit-learn Practice: *LinearRegression*

- **Example (*wave* dataset)**



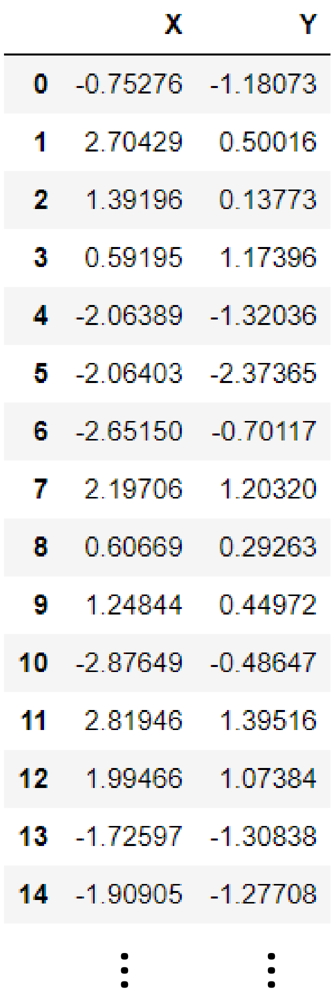
```
In [2]: import mglearn
        X, y = mglearn.datasets.make_wave(n_samples=60)
```

```
In [3]: from sklearn.model_selection import train_test_split
        X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```

```
In [4]: from sklearn.linear_model import LinearRegression
        reg = LinearRegression()
        reg.fit(X_train, y_train)

Out[4]: ▾ LinearRegression
        LinearRegression()
```

```
In [5]: from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
        y_train_hat = reg.predict(X_train)
        print('train MAE: %.5f'%mean_absolute_error(y_train,y_train_hat))
        print('train RMSE: %.5f'%mean_squared_error(y_train,y_train_hat)**0.5)
        print('train R_square: %.5f'%r2_score(y_train,y_train_hat))

        y_test_hat = reg.predict(X_test)
        print('test MAE: %.5f'%mean_absolute_error(y_test,y_test_hat))
        print('test RMSE: %.5f'%mean_squared_error(y_test,y_test_hat)**0.5)
        print('test R_square: %.5f'%r2_score(y_test,y_test_hat))
```

```
train MAE: 0.41817
train RMSE: 0.50589
train R_square: 0.67009
test MAE: 0.49453
test RMSE: 0.62826
test R_square: 0.65934
```

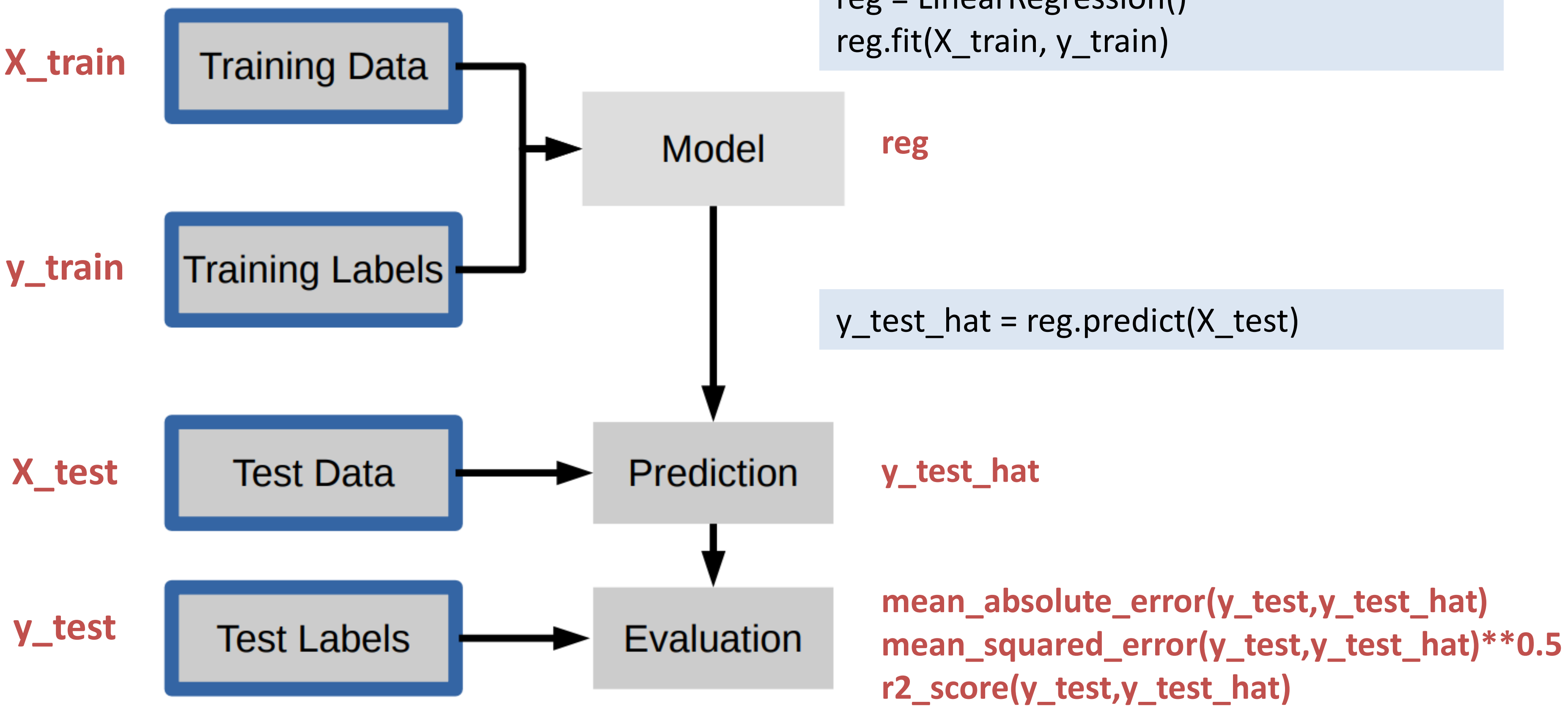


	X	Y
0	-0.75276	-1.18073
1	2.70429	0.50016
2	1.39196	0.13773
3	0.59195	1.17396
4	-2.06389	-1.32036
5	-2.06403	-2.37365
6	-2.65150	-0.70117
7	2.19706	1.20320
8	0.60669	0.29263
9	1.24844	0.44972
10	-2.87649	-0.48647
11	2.81946	1.39516
12	1.99466	1.07384
13	-1.72597	-1.30838
14	-1.90905	-1.27708
	⋮	⋮

scikit-learn Practice: *LinearRegression*

- **Example (*wave* dataset)**

```
X, y = mglearn.datasets.make_wave(n_samples=60)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
```



```
reg = LinearRegression()
reg.fit(X_train, y_train)
```

X_train — Training Data

y_train — Training Labels

Model — reg

```
y_test_hat = reg.predict(X_test)
```

X_test — Test Data

Prediction — y_test_hat

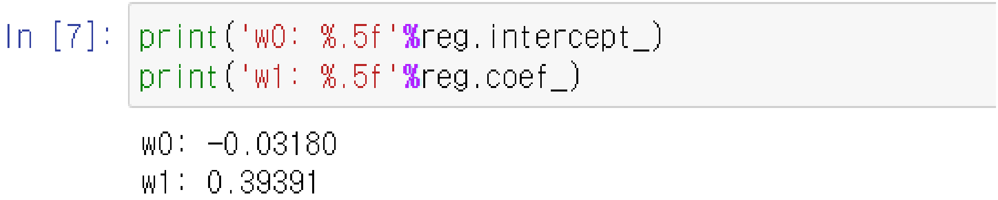
y_test — Test Labels

Evaluation

mean_absolute_error(y_test,y_test_hat)
mean_squared_error(y_test,y_test_hat)**0.5
r2_score(y_test,y_test_hat)

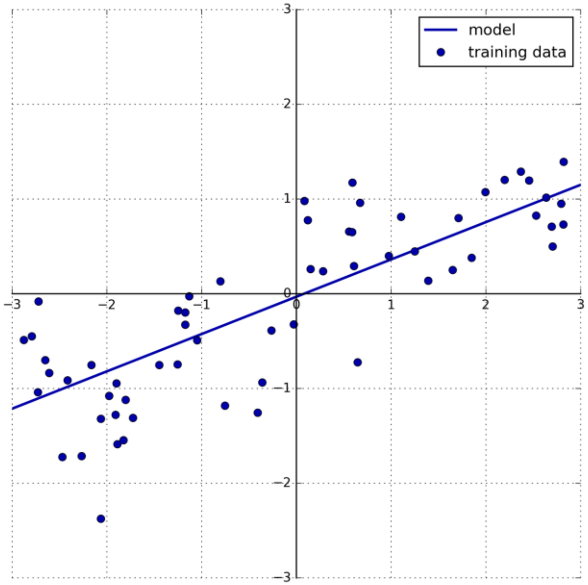
scikit-learn Practice: *LinearRegression*

- **Example (*wave* dataset)**



```
In [7]: print('w0: %.5f'%reg.intercept_)
        print('w1: %.5f'%reg.coef_)

        w0: -0.03180
        w1: 0.39391
```

$$\hat{y} = -0.03180 + 0.39391x$$

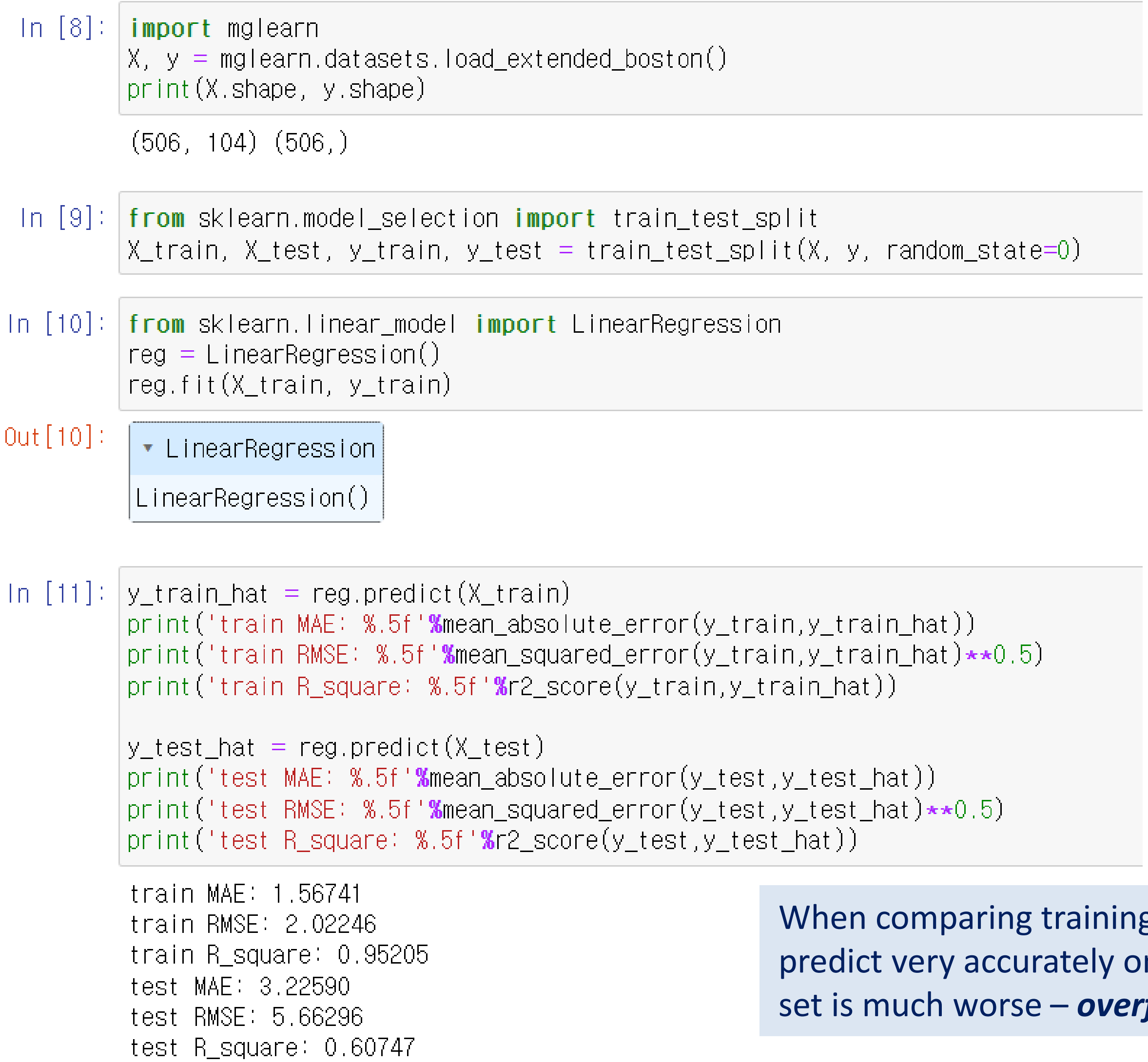


scikit-learn Practice: *LinearRegression*

- **Example with the *extended_boston* dataset**
  - The dataset consists of 506 data points described by 104 features
  - The 104 features are the 13 original features together with the 91 possible combinations of two features within those 13 (all products between original features).
  - The regression task associated with this dataset is to predict the median value of homes in several Boston neighborhoods in the 1970s, using information such as crime rate, proximity to the Charles River, highway accessibility, and so on.

scikit-learn Practice: *LinearRegression*

- **Example (*extended_boston* dataset)**



```
In [8]:  import mglearn
         X, y = mglearn.datasets.load_extended_boston()
         print(X.shape, y.shape)

         (506, 104) (506,)

In [9]:  from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

In [10]: from sklearn.linear_model import LinearRegression
         reg = LinearRegression()
         reg.fit(X_train, y_train)

Out[10]: ▾ LinearRegression
         LinearRegression()

In [11]: y_train_hat = reg.predict(X_train)
         print('train MAE: %.5f'%mean_absolute_error(y_train,y_train_hat))
         print('train RMSE: %.5f'%mean_squared_error(y_train,y_train_hat)**0.5)
         print('train R_square: %.5f'%r2_score(y_train,y_train_hat))

         y_test_hat = reg.predict(X_test)
         print('test MAE: %.5f'%mean_absolute_error(y_test,y_test_hat))
         print('test RMSE: %.5f'%mean_squared_error(y_test,y_test_hat)**0.5)
         print('test R_square: %.5f'%r2_score(y_test,y_test_hat))

         train MAE: 1.56741
         train RMSE: 2.02246
         train R_square: 0.95205
         test MAE: 3.22590
         test RMSE: 5.66296
         test R_square: 0.60747
```

When comparing training set and test set scores, we find that we predict very accurately on the training set, but the R^2 on the test set is much worse – ***overfitting***

Regularized Linear Regression

- **Linear Regression:** $\hat{y} = \mathbf{w}^{\boldsymbol{T}}\boldsymbol{x}$

  Find the optimal parameter $\mathbf{w}^*$ that minimizes the training error (cost function)

$$J(\mathbf{w}) = \mathrm{MSE}_{\mathrm{train}} = \frac{1}{n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$$

- **Ridge Regression:** $\hat{y} = \mathbf{w}^{\boldsymbol{T}}\boldsymbol{x}$**, L2 regularization** for linear regression

  Add an L2 regularization term $\alpha\|\mathbf{w}\|_2^2$ *to the cost function*

$$\tilde{J}(\mathbf{w}) = \mathrm{MSE}_{\mathrm{train}} + \alpha\|\mathbf{w}\|_2^2 = \frac{1}{n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 + \alpha\|\mathbf{w}\|_2^2$$

- **Lasso Regression:** $\hat{y} = \mathbf{w}^{\boldsymbol{T}}\boldsymbol{x}$**, L1 regularization** for linear regression

  Add an L1 regularization term $\alpha\|\mathbf{w}\|_1$ *to the cost function*

$$\tilde{J}(\mathbf{w}) = \mathrm{MSE}_{\mathrm{train}} + \alpha\|\mathbf{w}\|_1 = \frac{1}{n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2 + \alpha\|\mathbf{w}\|_1$$

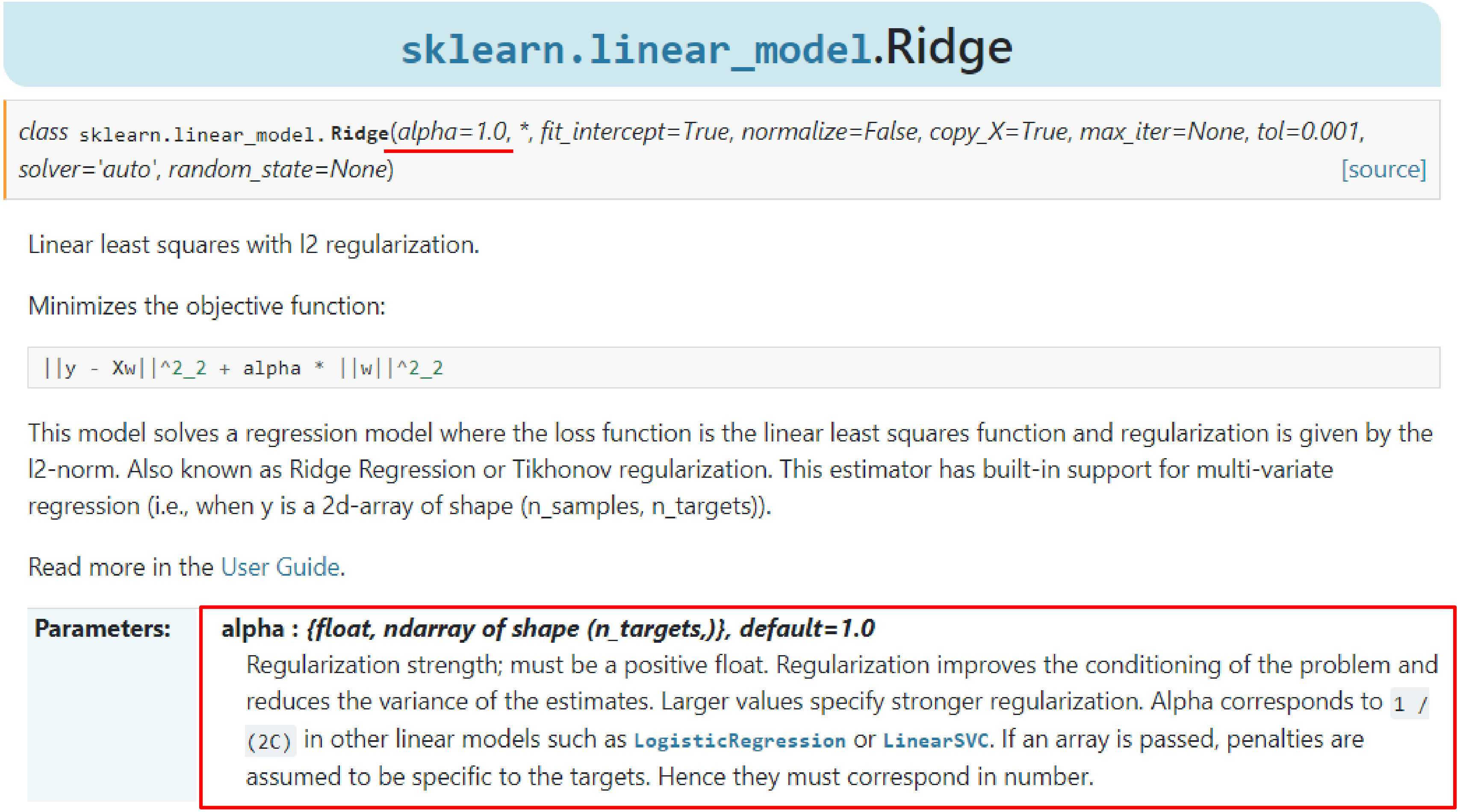
Regularized Linear Regression

- **Adding regularization (explicitly restricting a model to avoid *overfitting*) forces the learning algorithm to not only fit the data but also keep the magnitude of the model parameters as small as possible.**

- **The hyperparameter α controls how much you want to regularize the model.**
  - If α = 0 then Regularized Linear Regression (Ridge and Lasso) is just Linear Regression.
  - If α is very large, then all parameters end up very close to zero and the result is a flat line going through the mean of the labels in the training set.

scikit-learn Practice: *Ridge*

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html



sklearn.linear_model.Ridge

class `sklearn.linear_model.`**`Ridge`**(*alpha=1.0*, ***, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*, *random_state=None*) [source]

Linear least squares with l2 regularization.

Minimizes the objective function:

```
||y - Xw||^2_2 + alpha * ||w||^2_2
```

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape (n_samples, n_targets)).

Read more in the User Guide.

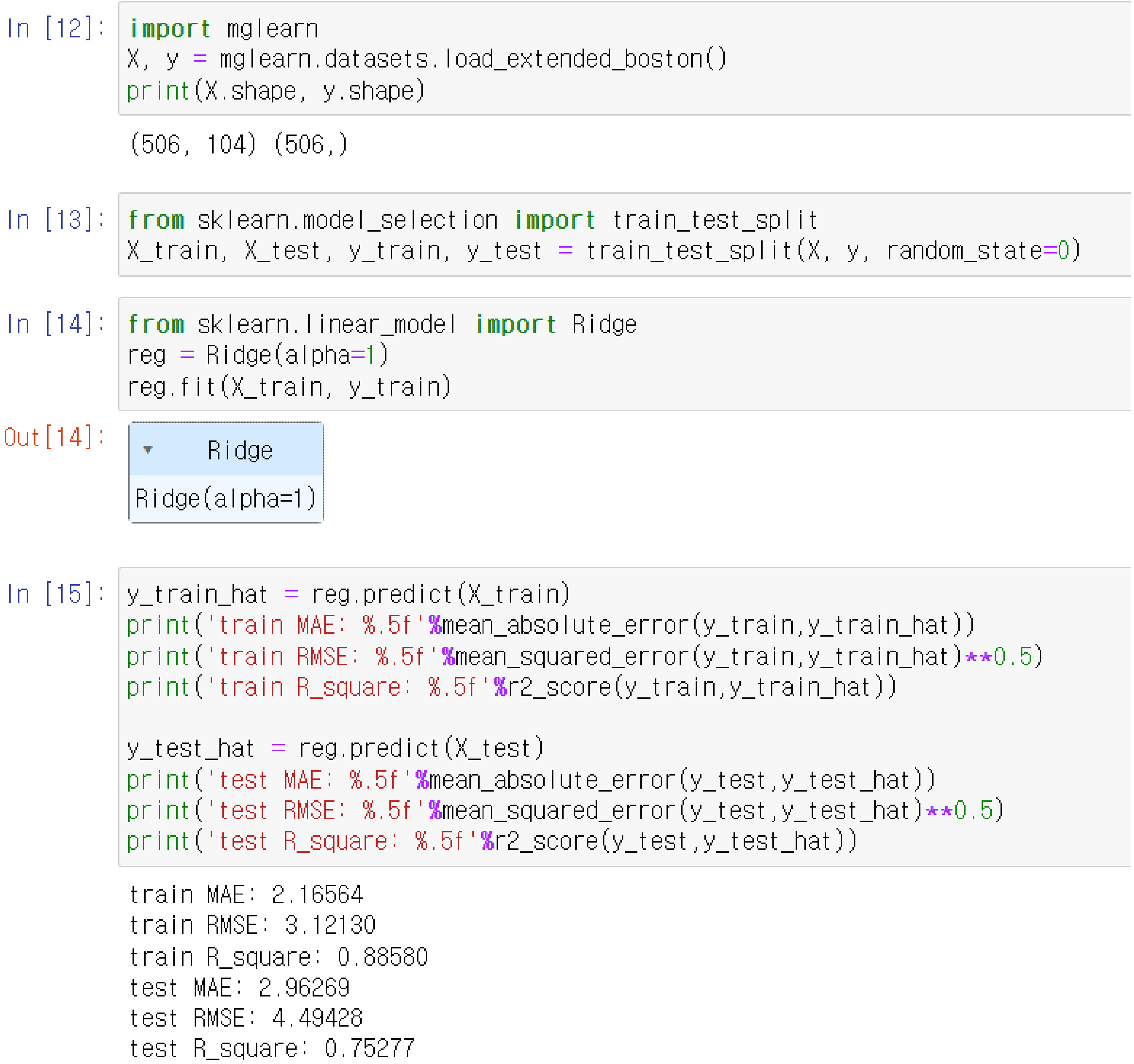
Parameters:

alpha : *{float, ndarray of shape (n_targets,)}, default=1.0*

Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `1 / (2C)` in other linear models such as `LogisticRegression` or `LinearSVC`. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

scikit-learn Practice: *Ridge*

- **Example (*extended_boston* dataset)**



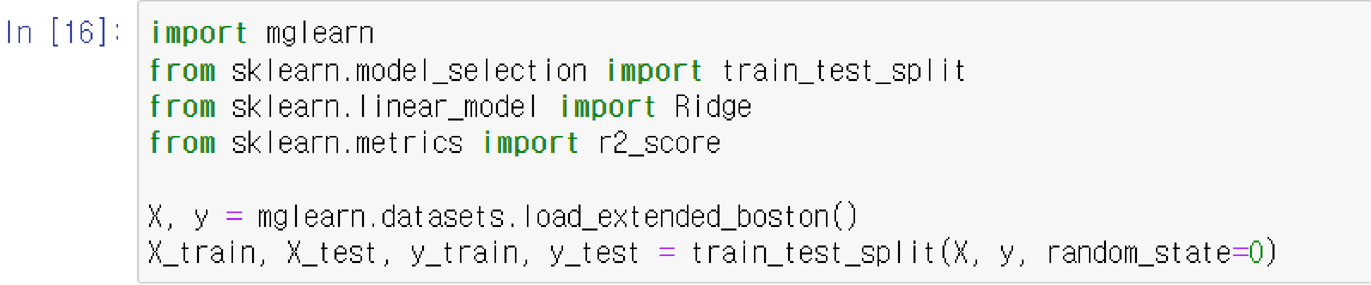
```
In [12]: import mglearn
         X, y = mglearn.datasets.load_extended_boston()
         print(X.shape, y.shape)

         (506, 104) (506,)

In [13]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

In [14]: from sklearn.linear_model import Ridge
         reg = Ridge(alpha=1)
         reg.fit(X_train, y_train)

Out[14]: ▾     Ridge
         Ridge(alpha=1)

In [15]: y_train_hat = reg.predict(X_train)
         print('train MAE: %.5f'%mean_absolute_error(y_train,y_train_hat))
         print('train RMSE: %.5f'%mean_squared_error(y_train,y_train_hat)**0.5)
         print('train R_square: %.5f'%r2_score(y_train,y_train_hat))

         y_test_hat = reg.predict(X_test)
         print('test MAE: %.5f'%mean_absolute_error(y_test,y_test_hat))
         print('test RMSE: %.5f'%mean_squared_error(y_test,y_test_hat)**0.5)
         print('test R_square: %.5f'%r2_score(y_test,y_test_hat))

         train MAE: 2.16564
         train RMSE: 3.12130
         train R_square: 0.88580
         test MAE: 2.96269
         test RMSE: 4.49428
         test R_square: 0.75277
```

LinearRegression Results

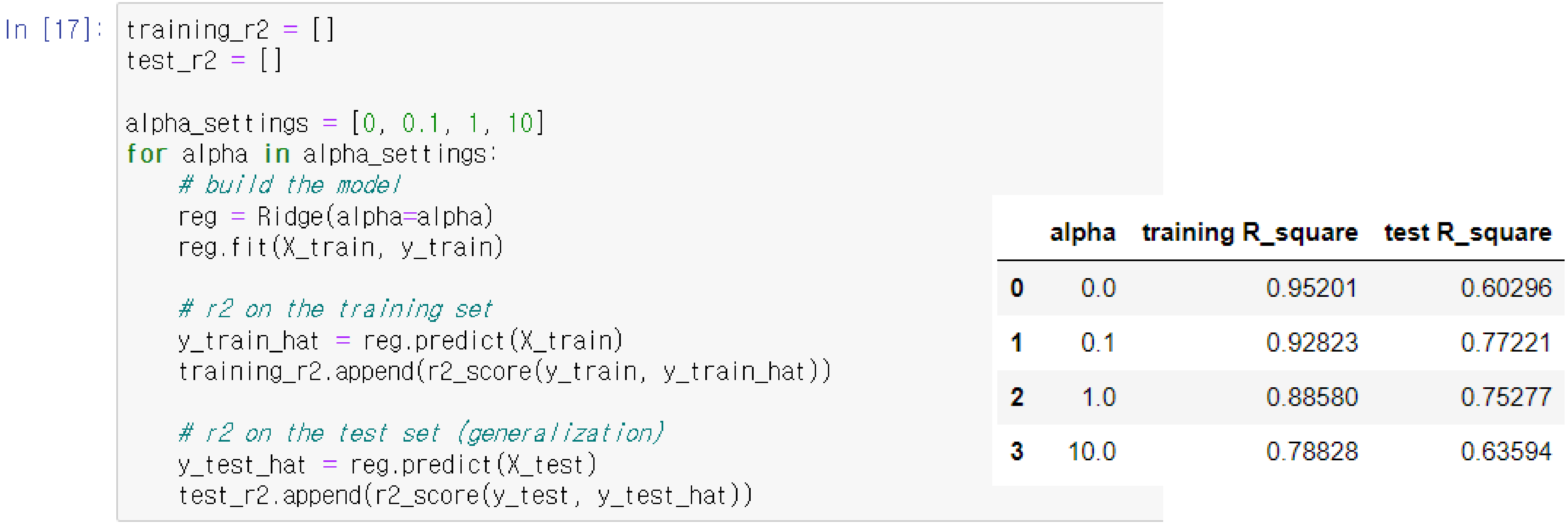
```
train MAE: 1.56741
train RMSE: 2.02246
train R_square: 0.95205
test MAE: 3.22590
test RMSE: 5.66296
test R_square: 0.60747
```

scikit-learn Practice: *Ridge*

- **Example (*extended_boston* dataset): varying the hyperparameter α**



```
In [16]: import mglearn
         from sklearn.model_selection import train_test_split
         from sklearn.linear_model import Ridge
         from sklearn.metrics import r2_score

         X, y = mglearn.datasets.load_extended_boston()
         X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

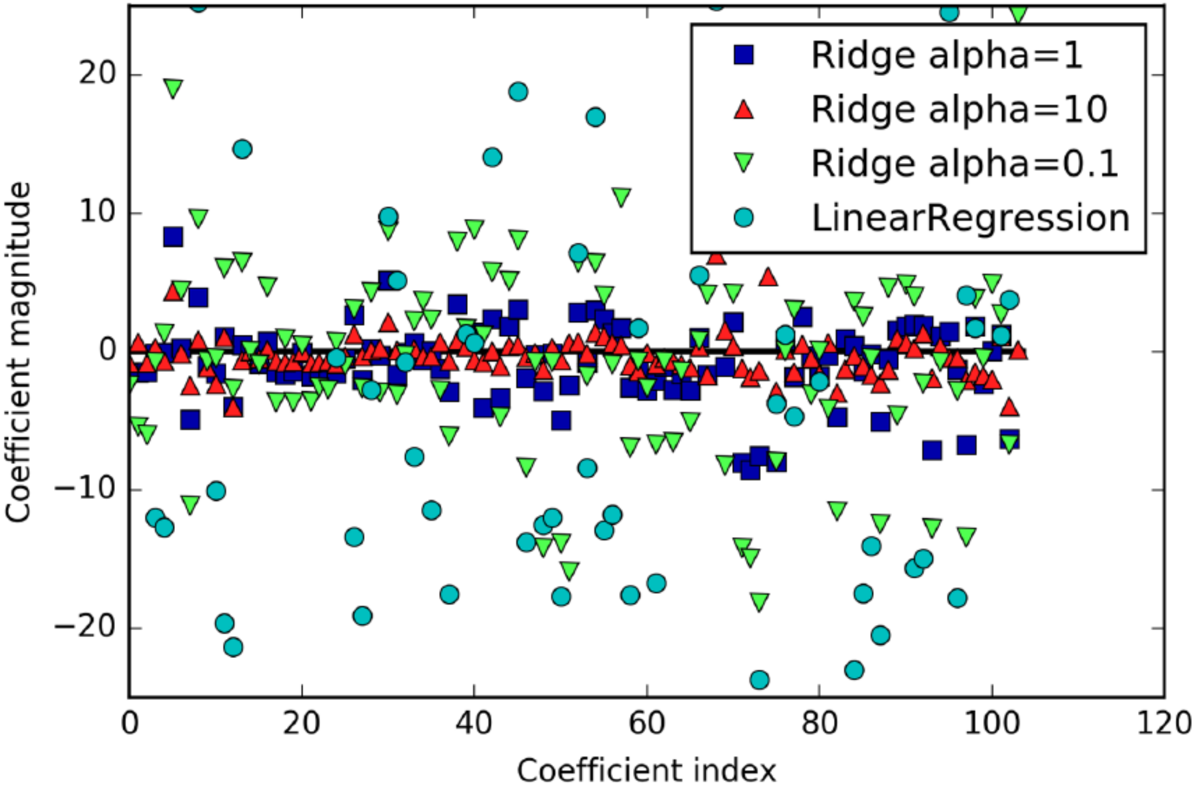


```
In [17]: training_r2 = []
         test_r2 = []

         alpha_settings = [0, 0.1, 1, 10]
         for alpha in alpha_settings:
             # build the model
             reg = Ridge(alpha=alpha)
             reg.fit(X_train, y_train)

             # r2 on the training set
             y_train_hat = reg.predict(X_train)
             training_r2.append(r2_score(y_train, y_train_hat))

             # r2 on the test set (generalization)
             y_test_hat = reg.predict(X_test)
             test_r2.append(r2_score(y_test, y_test_hat))
```

	alpha	training R_square	test R_square
0	0.0	0.95201	0.60296
1	0.1	0.92823	0.77221
2	1.0	0.88580	0.75277
3	10.0	0.78828	0.63594

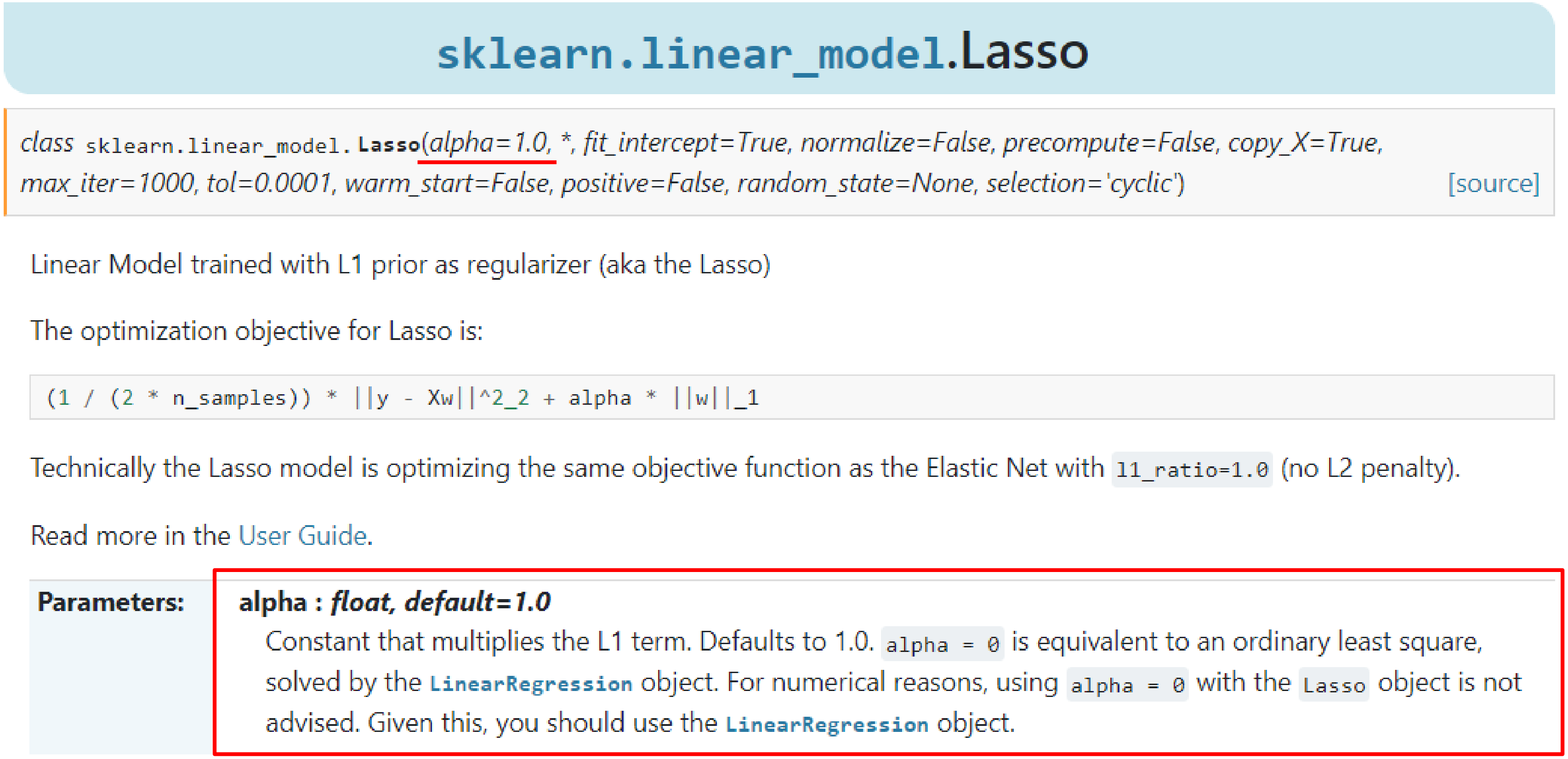
scikit-learn Practice: *Ridge*

- **The effect of the hyperparameter α**
  - Comparing coefficient magnitudes for ridge regression
    - When α=10, the coefficients are mostly between around –3 and 3
    - When α=0 (Linear Regression), the coefficients have larger magnitude



scikit-learn Practice: *Lasso*

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html



sklearn.linear_model.Lasso

class `sklearn.linear_model.`**`Lasso`**(*alpha=1.0, *, fit_intercept=True, normalize=False, precompute=False, copy_X=True, max_iter=1000, tol=0.0001, warm_start=False, positive=False, random_state=None, selection='cyclic'*) [source]

Linear Model trained with L1 prior as regularizer (aka the Lasso)

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Technically the Lasso model is optimizing the same objective function as the Elastic Net with `l1_ratio=1.0` (no L2 penalty).

Read more in the User Guide.

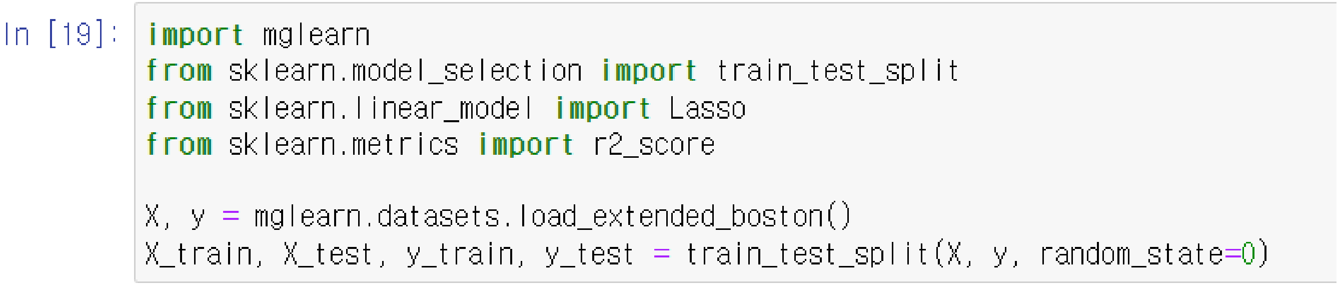
Parameters:

alpha : *float, default=1.0*

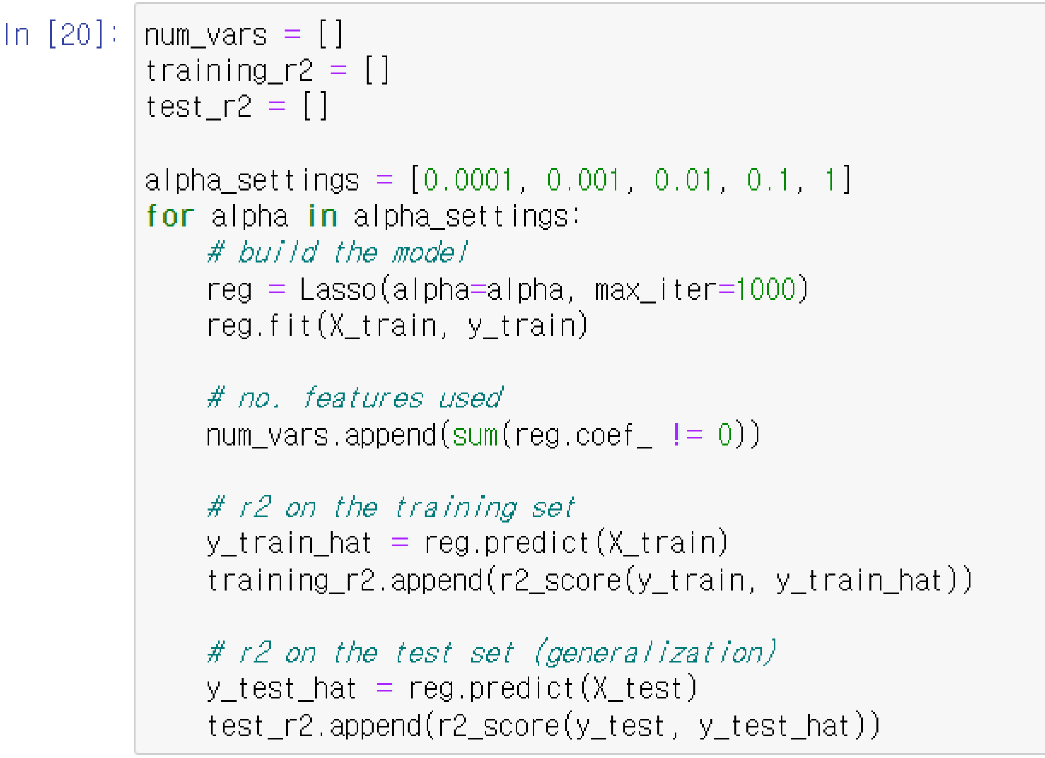
Constant that multiplies the L1 term. Defaults to 1.0. `alpha = 0` is equivalent to an ordinary least square, solved by the `LinearRegression` object. For numerical reasons, using `alpha = 0` with the `Lasso` object is not advised. Given this, you should use the `LinearRegression` object.

scikit-learn Practice: *Lasso*

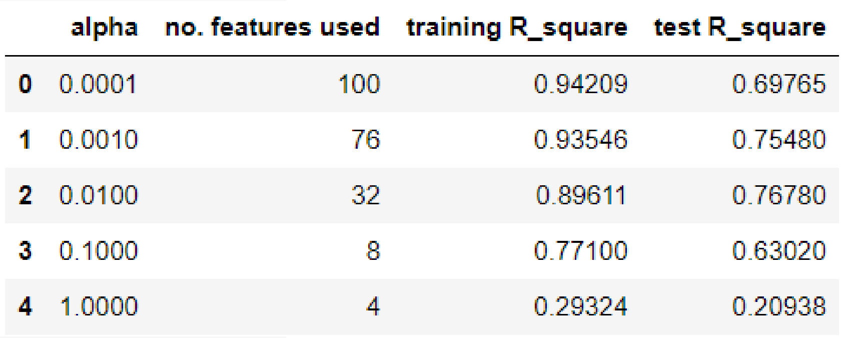
- **Example (*extended_boston* dataset): varying the hyperparameter α**



```
In [19]: import mglearn
         from sklearn.model_selection import train_test_split
         from sklearn.linear_model import Lasso
         from sklearn.metrics import r2_score

         X, y = mglearn.datasets.load_extended_boston()
         X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```



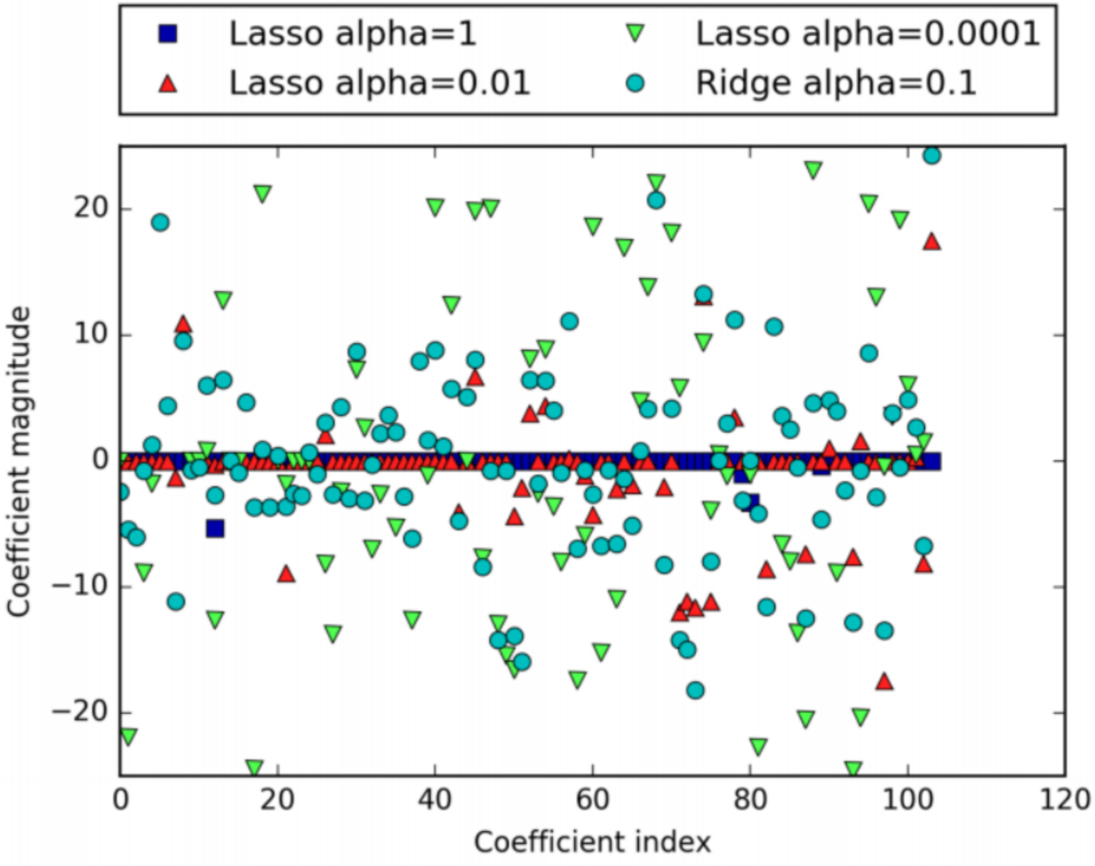
```
In [20]: num_vars = []
         training_r2 = []
         test_r2 = []

         alpha_settings = [0.0001, 0.001, 0.01, 0.1, 1]
         for alpha in alpha_settings:
             # build the model
             reg = Lasso(alpha=alpha, max_iter=1000)
             reg.fit(X_train, y_train)

             # no. features used
             num_vars.append(sum(reg.coef_ != 0))

             # r2 on the training set
             y_train_hat = reg.predict(X_train)
             training_r2.append(r2_score(y_train, y_train_hat))

             # r2 on the test set (generalization)
             y_test_hat = reg.predict(X_test)
             test_r2.append(r2_score(y_test, y_test_hat))
```



	alpha	no. features used	training R_square	test R_square
0	0.0001	100	0.94209	0.69765
1	0.0010	76	0.93546	0.75480
2	0.0100	32	0.89611	0.76780
3	0.1000	8	0.77100	0.63020
4	1.0000	4	0.29324	0.20938

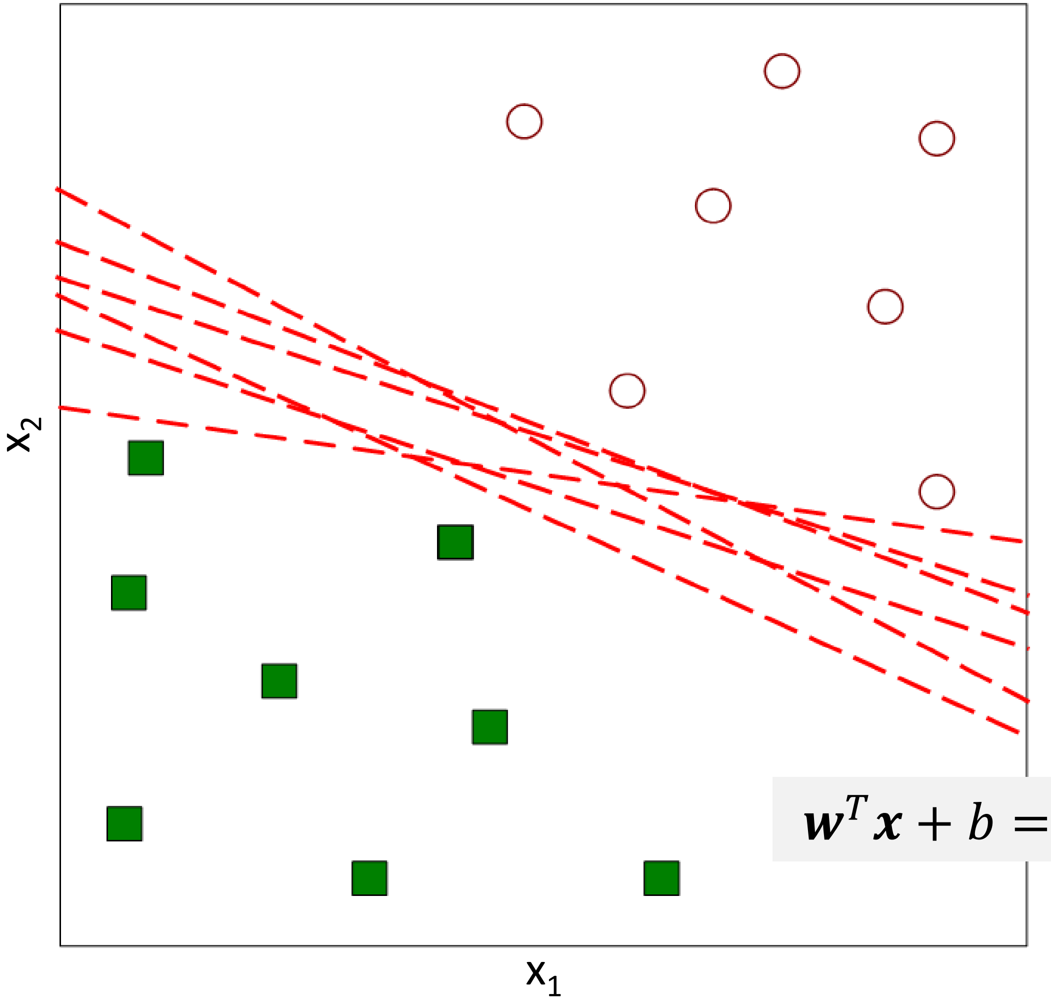
scikit-learn Practice: *Lasso*

- **The effect of the hyperparameter α**
  - Comparing coefficient magnitudes for lasso regression
    - Some coefficients are *exactly zero*, meaning that some features are entirely ignored by the model – feature selection



Logistic Regression

- For linear models for binary classification, the ***decision boundary (hyperplane)*** that separates two classes is a **linear function** of input features.



$$\boldsymbol{w}^T\boldsymbol{x} + b = 0$$

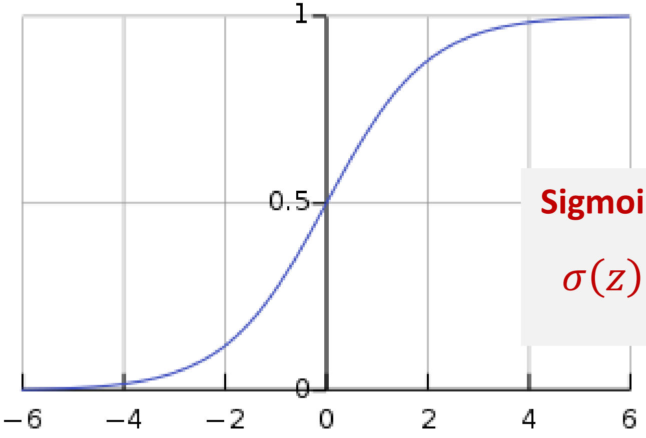
Logistic Regression

- **Logistic Regression**
  - Extends the idea of linear regression to situation where the target label is binary (y = 0 or 1)

$$\hat{y} = \sigma(\mathbf{w}^T \boldsymbol{x} + b) = \frac{1}{1 + \exp(-\mathbf{w}^T \boldsymbol{x} - b)}$$

$$\boldsymbol{x} = (x_1, \dots, x_d) \in \mathbb{R}^d, \qquad y \in \{0,1\}, \qquad \hat{y} \in [0,1]$$

  - If $\hat{y}$ > 0.5, classify as "1", If $\hat{y}$ < 0.5, classify as "0"



Sigmoid function

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Logistic Regression

- Given a (training) dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (1, x_{i1}, \dots, x_{id}) \in \mathbb{R}^{d+1}$ is the *i*-th input vector of *d* features and $y_i \in \{0,1\}$ is the corresponding target label.

- the first entry is always set to "1"

- The output of model f (prediction of y)

$$: \hat{y} = f(\boldsymbol{x}) = \sigma(\mathbf{w}^T\boldsymbol{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\boldsymbol{x})}, \hat{y} \in [0,1]$$

- **Training**: To find the optimal parameter $\mathbf{w}^*$ that minimizes the training error (cost function) → here we use "binary cross-entropy" loss

$$J(\mathbf{w}) = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} L(y_i, \hat{y}_i) = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} [-y_i \log \hat{y}_i - (1 - y_i)\log(1 - \hat{y}_i)]$$

Logistic Regression

- **Training**: To find the optimal parameter $\mathbf{w}^*$ that minimizes the training error (cost function)

$$J(\mathbf{w}) = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} L(y_i, \hat{y}_i) = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} [-y_i \log \hat{y}_i - (1 - y_i) \log(1 - \hat{y}_i)]$$

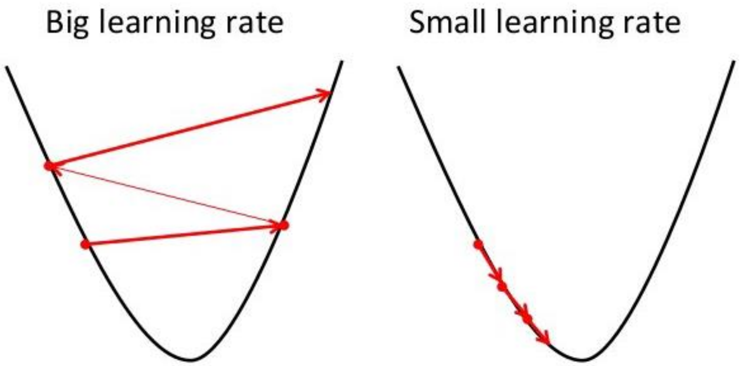
▶ how? gradient descent! (no closed-from solution)

Repeat the following until convergence

$$\mathbf{w} \coloneqq \mathbf{w} - \epsilon \nabla_{\mathbf{w}} J(\mathbf{w})$$

$$\rightarrow w_j \coloneqq w_j - \epsilon \frac{\partial}{\partial w_j} J(\boldsymbol{w}), \forall w_j \in \boldsymbol{w}$$

ϵ is the learning rate



- The trained model $f(\boldsymbol{x}) = \sigma(\mathbf{w}^{*\mathrm{T}} \boldsymbol{x})$

Logistic Regression

- $\boldsymbol{\theta} := \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$? Where does it come from?

- **Let’s recall “Taylor series” of calculus**
  - Taylor expansion of a function of $\boldsymbol{\theta}$

$$J(\boldsymbol{\theta}) = J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \cdots$$

  - First-order approximation (assume that $\boldsymbol{\theta}$ is very close to $\boldsymbol{\theta}_0$)

$$J(\boldsymbol{\theta}) \simeq J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

  - We want to find a direction $\boldsymbol{\theta}_0 \rightarrow \boldsymbol{\theta}$ to make $J(\boldsymbol{\theta}) < J(\boldsymbol{\theta}_0)$

$$J(\boldsymbol{\theta}) - J(\boldsymbol{\theta}_0) \simeq (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^T \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) < 0$$

linear function w.r.t. $\boldsymbol{\theta}$

  - The best direction

$$(\boldsymbol{\theta} - \boldsymbol{\theta}_0) \propto -\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

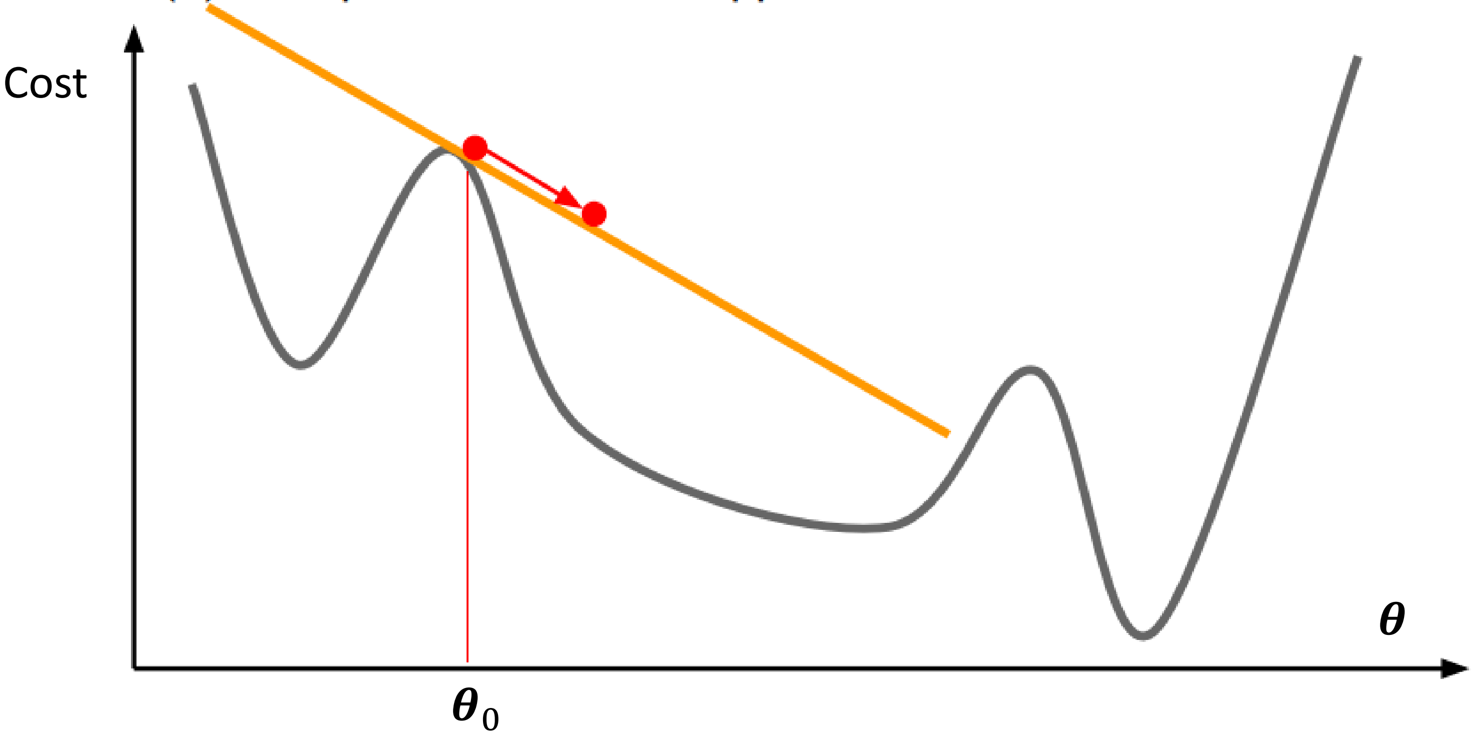
$$(\boldsymbol{\theta} - \boldsymbol{\theta}_0) = -\epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \epsilon > 0$$

$$\boldsymbol{\theta} = \boldsymbol{\theta}_0 - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0), \epsilon > 0$$

why?

Logistic Regression

- **Illustrative Example of Optimization based on First-Order Approximation**



(1) Use gradient form linear approximation

(2) Step to minimize the approximation

Logistic Regression

$$L(y, \hat{y}) = -y \log \sigma(z) - (1 - y) \log\big(1 - \sigma(z)\big),$$

where $\hat{y} = \sigma(z), z = \mathbf{w}^T \boldsymbol{x} = w_0 + w_1 x_1 + \cdots + w_d x_d$

$$\frac{\partial L(\boldsymbol{w})}{\partial w_j} = \frac{\partial L(\boldsymbol{w})}{\partial z} \frac{\partial z}{\partial w_j} = (\hat{y} - y) x_j$$

$$\begin{aligned}\frac{\partial L(\boldsymbol{w})}{\partial z} &= -y \frac{\partial \log \sigma(z)}{\partial z} - (1 - \mathrm{y}) \frac{\partial \log\big(1 - \sigma(z)\big)}{\partial z} \\ &= -y \frac{1}{\sigma(z)} \frac{\partial \sigma(z)}{\partial z} - (1 - \mathrm{y}) \frac{-1}{1 - \sigma(z)} \frac{\partial \sigma(z)}{\partial z} \\ &= -y \frac{1}{\sigma(z)} \sigma(z)\big(1 - \sigma(z)\big) - (1 - \mathrm{y}) \frac{-1}{1 - \sigma(z)} \sigma(z)\big(1 - \sigma(z)\big) \\ &= -y + \sigma(z) = \hat{y} - y\end{aligned}$$

$$\frac{\partial z}{\partial w_j} = \frac{\partial (w_0 + w_1 x_1 + \cdots + w_d x_d)}{\partial w_j} = x_j$$

Probabilistic Interpretation of Logistic Regression

- **Probabilistic Interpretation of Logistic Regression**
  - Assume $y \sim \text{Bernoulli}(\hat{y})$, $\hat{y} = \sigma(\mathbf{w}^T \boldsymbol{x})$

$$p(y = 1 | x; \mathbf{w}) = \sigma(\mathbf{w}^T \boldsymbol{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T \boldsymbol{x})}$$

$$p(y = 0 | x; \mathbf{w}) = 1 - \sigma(\mathbf{w}^T \boldsymbol{x}) = \frac{\exp(-\mathbf{w}^T \boldsymbol{x})}{1 + \exp(-\mathbf{w}^T \boldsymbol{x})}$$

▼

$$p(y | x; \mathbf{w}) = \left(\sigma(\mathbf{w}^T \boldsymbol{x})\right)^y \left(1 - \sigma(\mathbf{w}^T \boldsymbol{x})\right)^{1-y}$$

p.f. of Bernoulli($\hat{y}$)

- **Maximum Likelihood Estimation** (with respect to **w**)

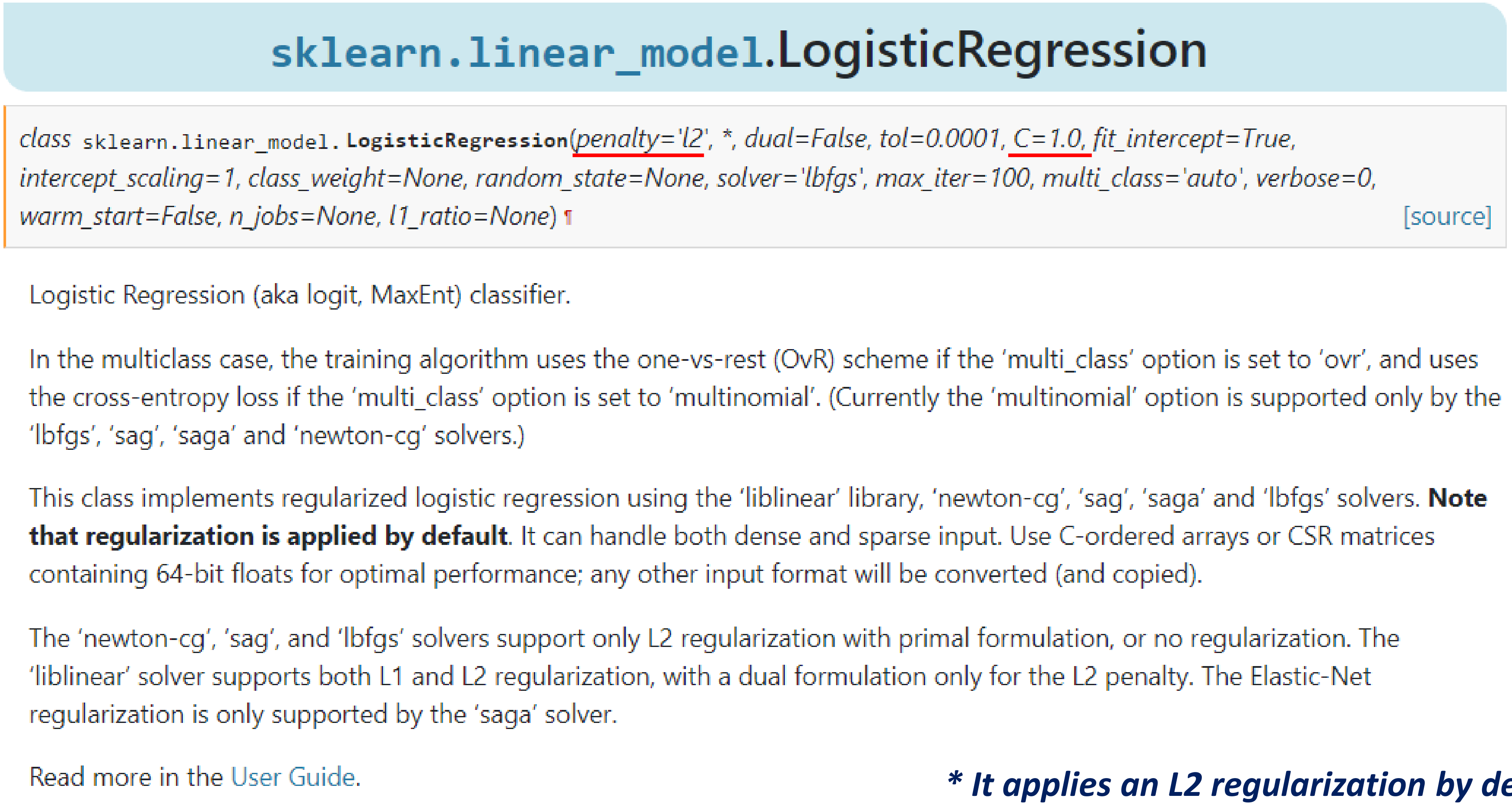
log-likelihood

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmax}} \prod_{(\boldsymbol{x}_i, y_i) \in D} p(y_i | \boldsymbol{x}_i; \mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmax}} \sum_{(\boldsymbol{x}_i, y_i) \in D} \log p(y_i | \boldsymbol{x}_i; \mathbf{w})$$

$$= \underset{\mathbf{w}}{\operatorname{argmax}} \boxed{\sum_{(\boldsymbol{x}_i, y_i) \in D} [y_i \log \sigma(\mathbf{w}^T \boldsymbol{x}_i) + (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \boldsymbol{x}_i))]}$$

scikit-learn Practice: *LogisticRegression*

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html



sklearn.linear_model.LogisticRegression

class `sklearn.linear_model.LogisticRegression`(*penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None*) ¶ [source]

Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses the one-vs-rest (OvR) scheme if the 'multi_class' option is set to 'ovr', and uses the cross-entropy loss if the 'multi_class' option is set to 'multinomial'. (Currently the 'multinomial' option is supported only by the 'lbfgs', 'sag', 'saga' and 'newton-cg' solvers.)

This class implements regularized logistic regression using the 'liblinear' library, 'newton-cg', 'sag', 'saga' and 'lbfgs' solvers. **Note that regularization is applied by default**. It can handle both dense and sparse input. Use C-ordered arrays or CSR matrices containing 64-bit floats for optimal performance; any other input format will be converted (and copied).

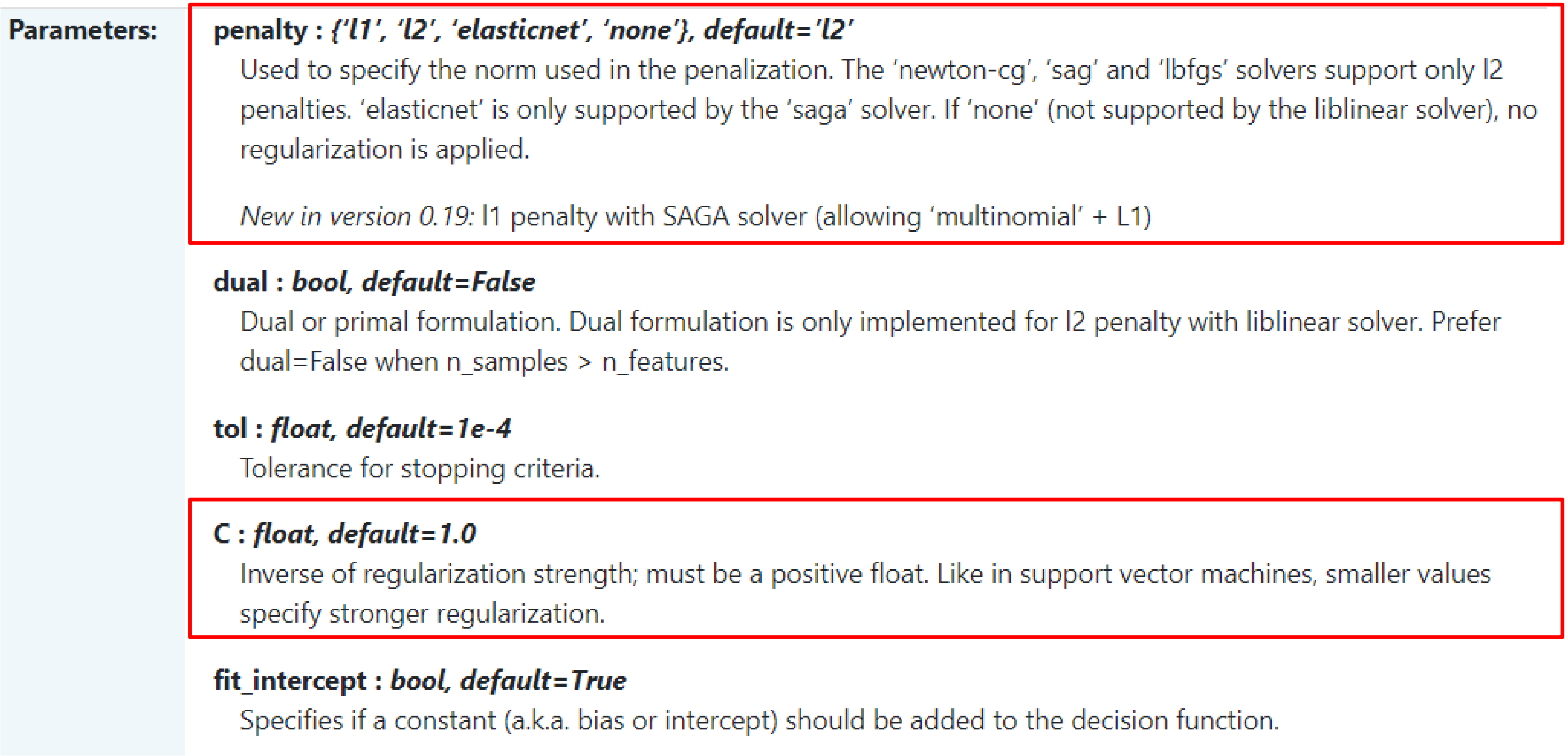
The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularization with primal formulation, or no regularization. The 'liblinear' solver supports both L1 and L2 regularization, with a dual formulation only for the L2 penalty. The Elastic-Net regularization is only supported by the 'saga' solver.

Read more in the User Guide.

**** It applies an L2 regularization by default***

scikit-learn Practice: *LogisticRegression*

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html



Parameters:

penalty : *{'l1', 'l2', 'elasticnet', 'none'}, default='l2'*

Used to specify the norm used in the penalization. The 'newton-cg', 'sag' and 'lbfgs' solvers support only l2 penalties. 'elasticnet' is only supported by the 'saga' solver. If 'none' (not supported by the liblinear solver), no regularization is applied.

New in version 0.19: l1 penalty with SAGA solver (allowing 'multinomial' + L1)

dual : *bool, default=False*

Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

tol : *float, default=1e-4*

Tolerance for stopping criteria.

C : *float, default=1.0*

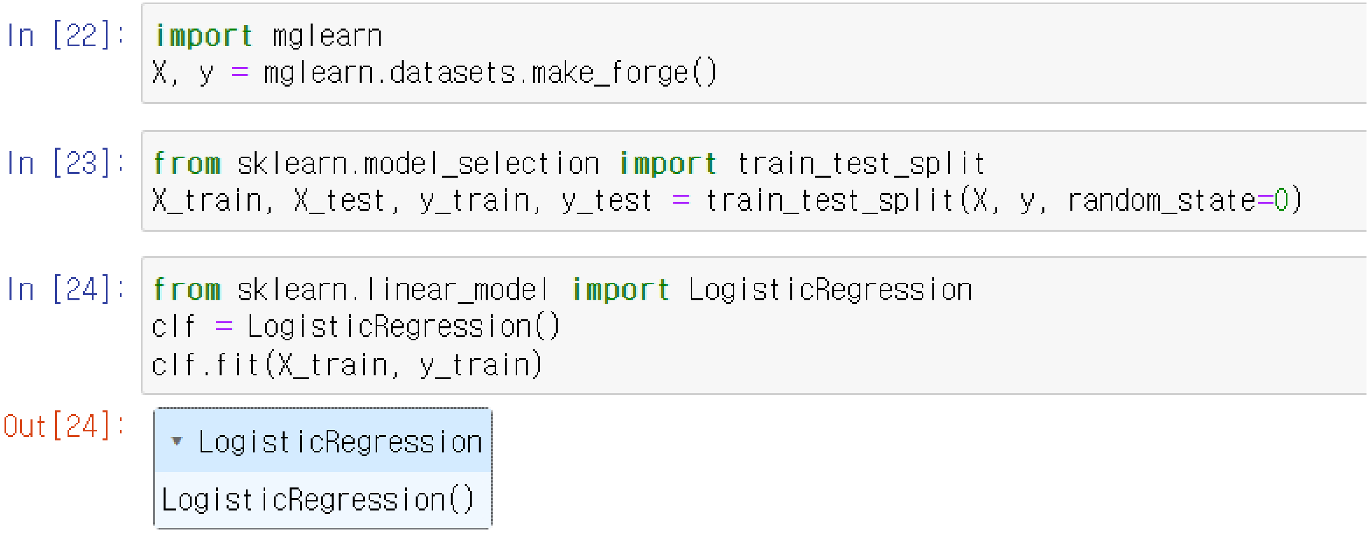
Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

fit_intercept : *bool, default=True*

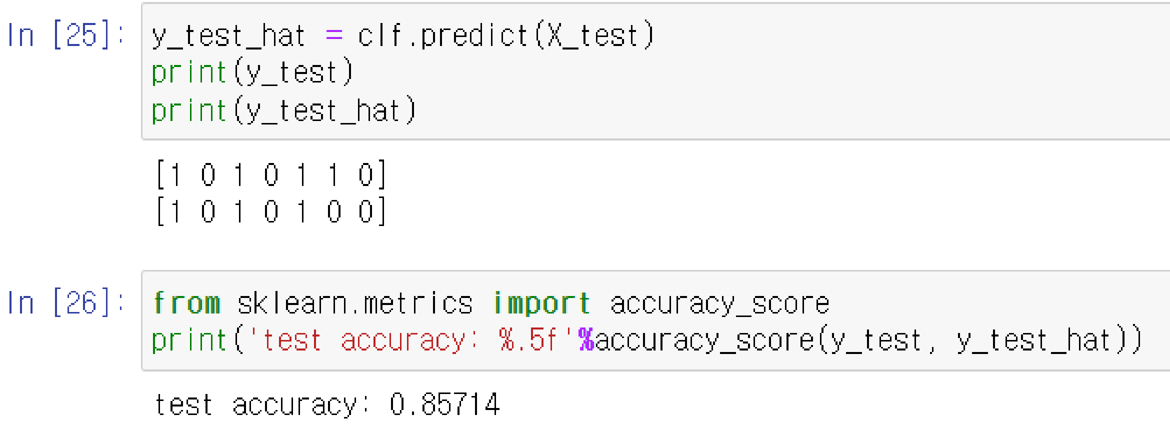
Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

scikit-learn Practice: *LogisticRegression*

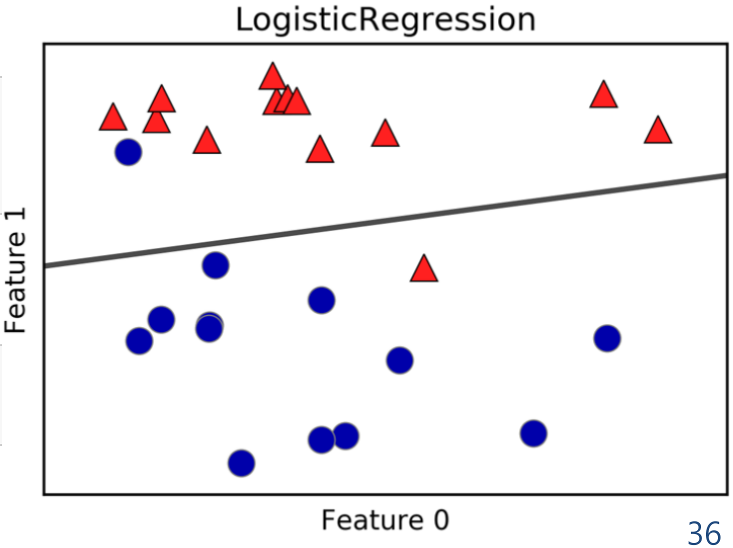
- **Example (*forge* dataset)**



```
In [22]: import mglearn
         X, y = mglearn.datasets.make_forge()

In [23]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

In [24]: from sklearn.linear_model import LogisticRegression
         clf = LogisticRegression()
         clf.fit(X_train, y_train)

Out[24]: ▾ LogisticRegression
         LogisticRegression()
```

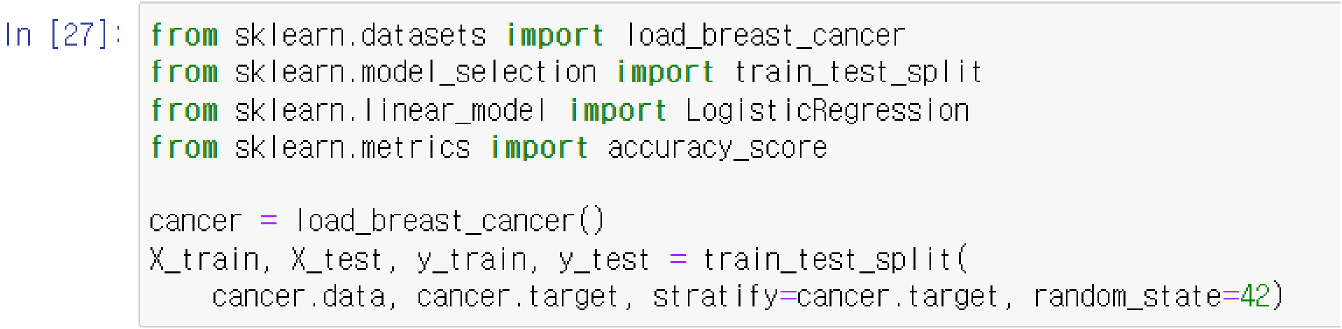


```
In [25]: y_test_hat = clf.predict(X_test)
         print(y_test)
         print(y_test_hat)

         [1 0 1 0 1 1 0]
         [1 0 1 0 1 0 0]

In [26]: from sklearn.metrics import accuracy_score
         print('test accuracy: %.5f'%accuracy_score(y_test, y_test_hat))

         test accuracy: 0.85714
```

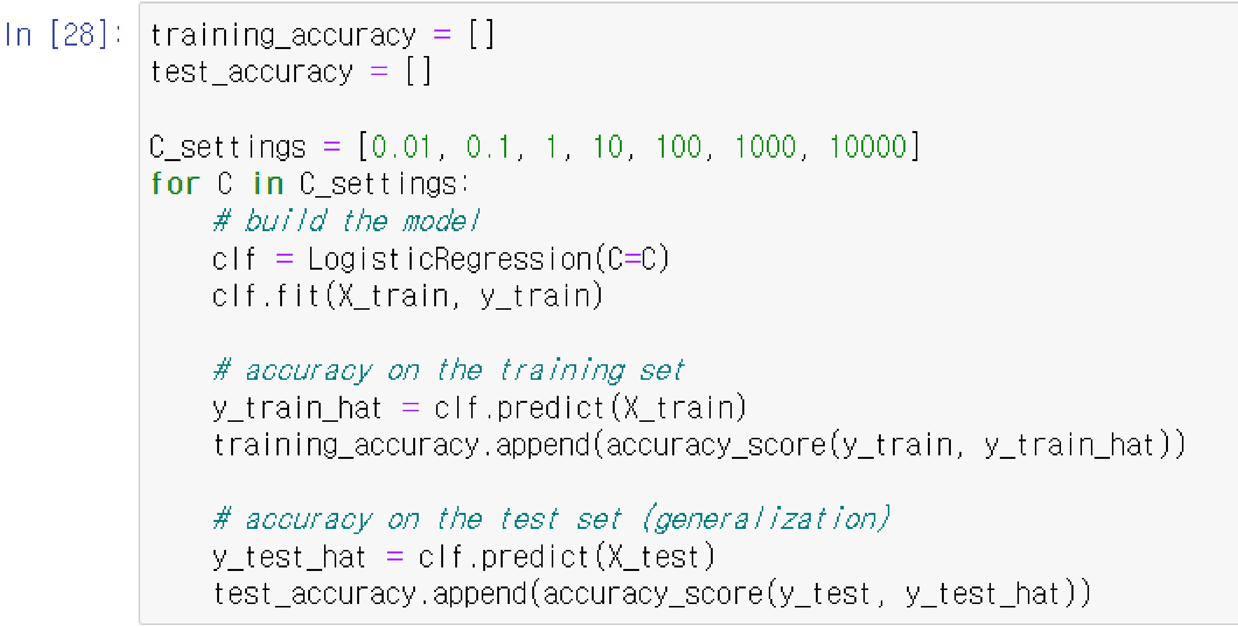


scikit-learn Practice: *LogisticRegression*

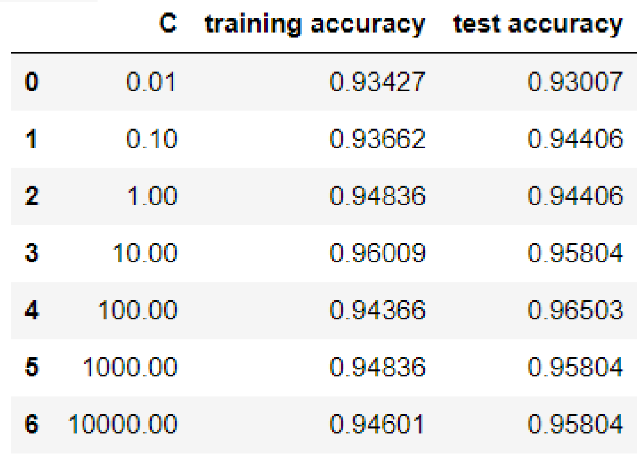
- **Example (*breast_cancer* dataset): varying the hyperparameter *C***



```
In [27]: from sklearn.datasets import load_breast_cancer
         from sklearn.model_selection import train_test_split
         from sklearn.linear_model import LogisticRegression
         from sklearn.metrics import accuracy_score

         cancer = load_breast_cancer()
         X_train, X_test, y_train, y_test = train_test_split(
             cancer.data, cancer.target, stratify=cancer.target, random_state=42)
```



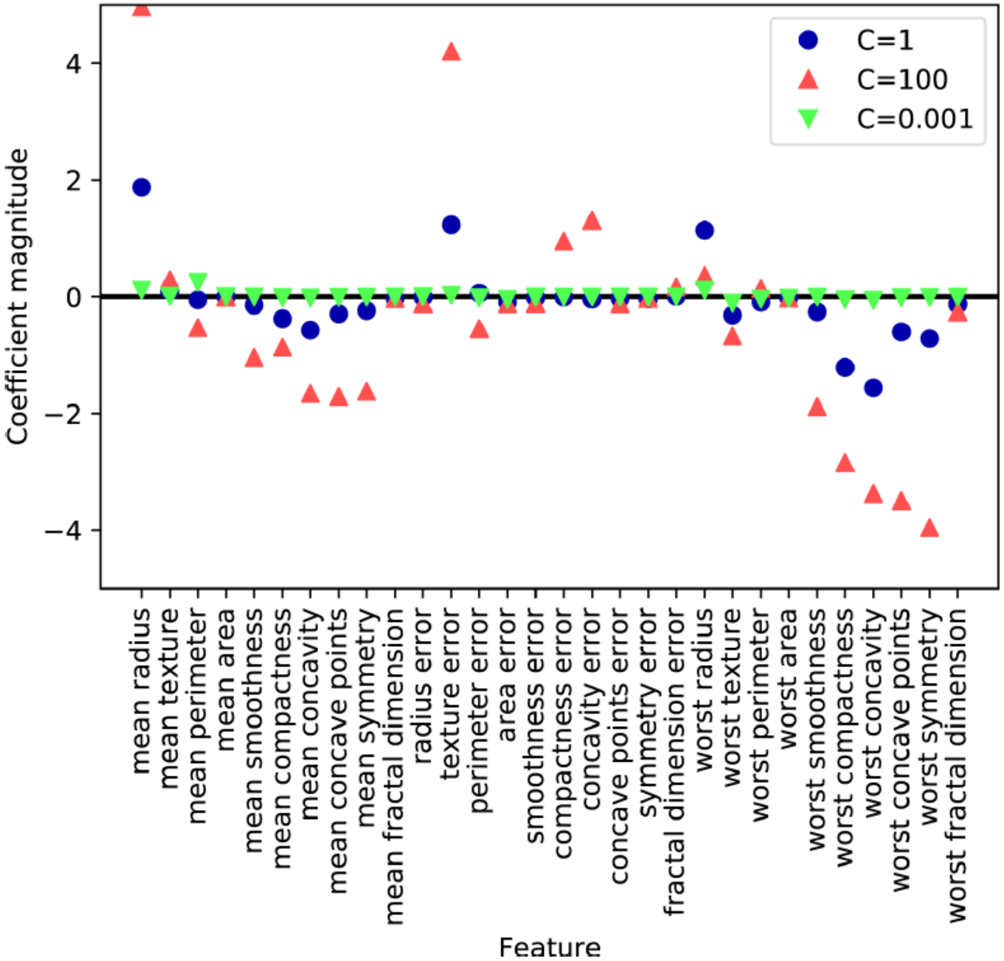
```
In [28]: training_accuracy = []
         test_accuracy = []

         C_settings = [0.01, 0.1, 1, 10, 100, 1000, 10000]
         for C in C_settings:
             # build the model
             clf = LogisticRegression(C=C)
             clf.fit(X_train, y_train)

             # accuracy on the training set
             y_train_hat = clf.predict(X_train)
             training_accuracy.append(accuracy_score(y_train, y_train_hat))

             # accuracy on the test set (generalization)
             y_test_hat = clf.predict(X_test)
             test_accuracy.append(accuracy_score(y_test, y_test_hat))
```



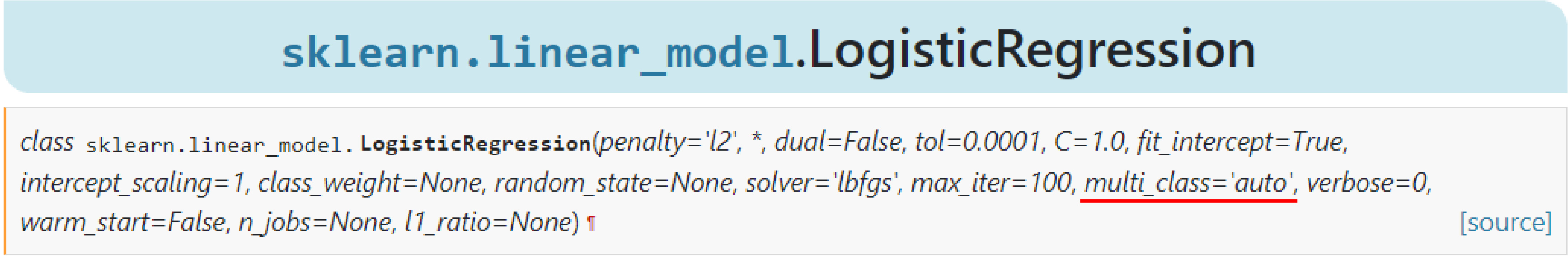
	C	training accuracy	test accuracy
0	0.01	0.93427	0.93007
1	0.10	0.93662	0.94406
2	1.00	0.94836	0.94406
3	10.00	0.96009	0.95804
4	100.00	0.94366	0.96503
5	1000.00	0.94836	0.95804
6	10000.00	0.94601	0.95804

scikit-learn Practice: *LogisticRegression*

- **The effect of the hyperparameter C**
  - Coefficients learned by logistic regression for different values of C
    - Decreasing C results in more regularized model

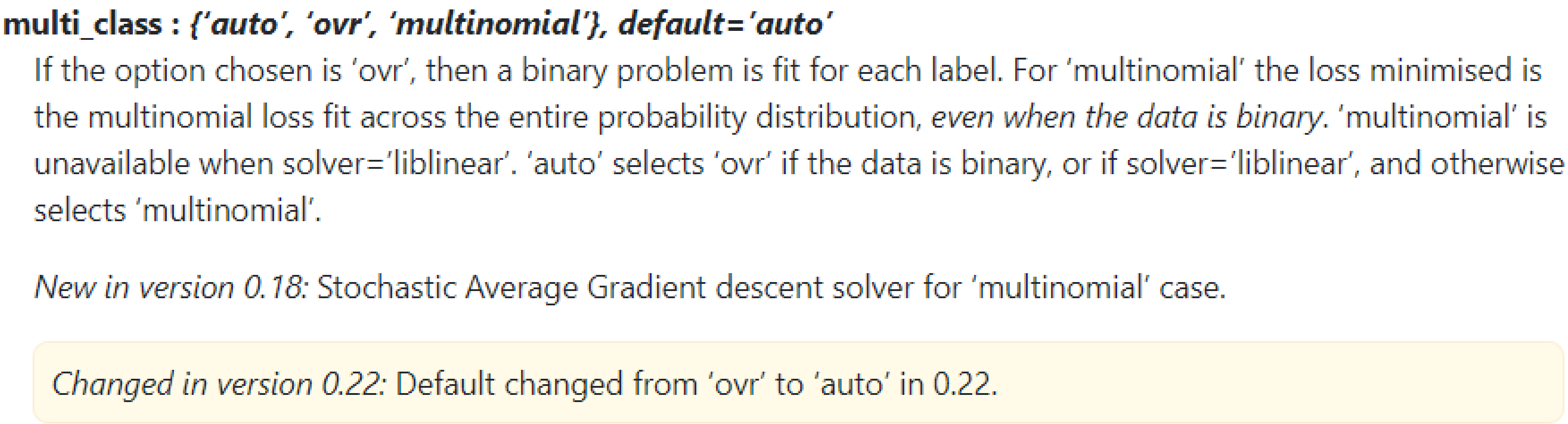


Linear Models for Multi-class Classification



sklearn.linear_model.LogisticRegression

class `sklearn.linear_model.`**`LogisticRegression`**(*penalty='l2', *, dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='lbfgs', max_iter=100, multi_class='auto', verbose=0, warm_start=False, n_jobs=None, l1_ratio=None*) ¶ [source]



multi_class : *{'auto', 'ovr', 'multinomial'}, default='auto'*

If the option chosen is 'ovr', then a binary problem is fit for each label. For 'multinomial' the loss minimised is the multinomial loss fit across the entire probability distribution, *even when the data is binary*. 'multinomial' is unavailable when solver='liblinear'. 'auto' selects 'ovr' if the data is binary, or if solver='liblinear', and otherwise selects 'multinomial'.

New in version 0.18: Stochastic Average Gradient descent solver for 'multinomial' case.

Changed in version 0.22: Default changed from 'ovr' to 'auto' in 0.22.

Linear Models for Multi-class Classification

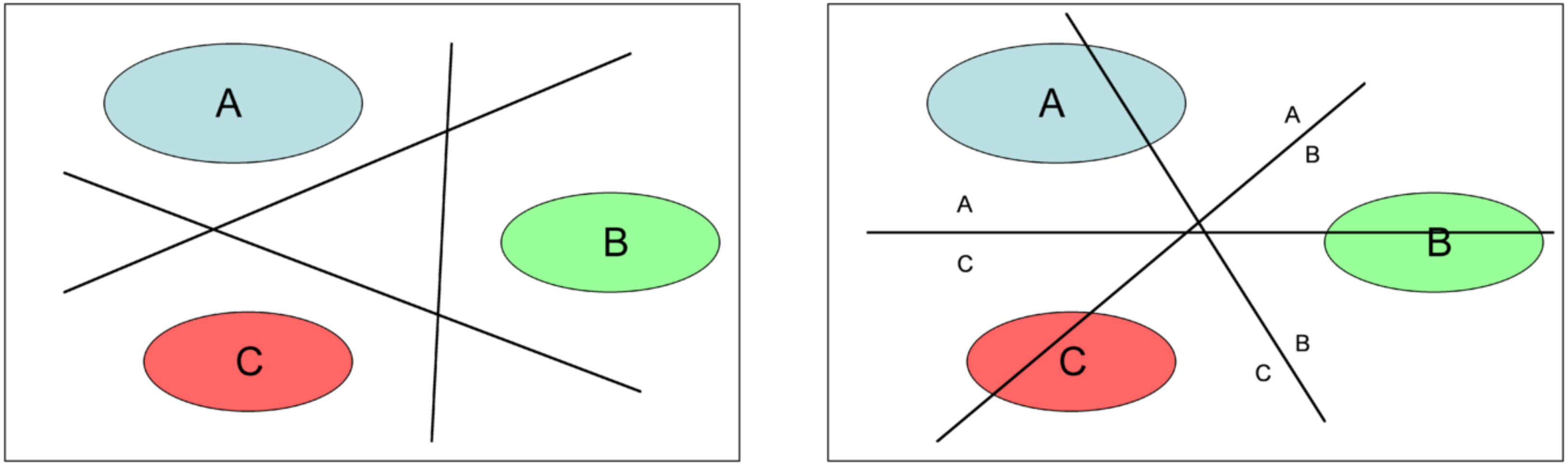
- Naturally, linear models are for binary classification only. Common techniques to extend a binary classification algorithm to a multi-class (*c* classes) classification algorithm are the *one-vs.-rest* approach and *one-vs.-one* approach.

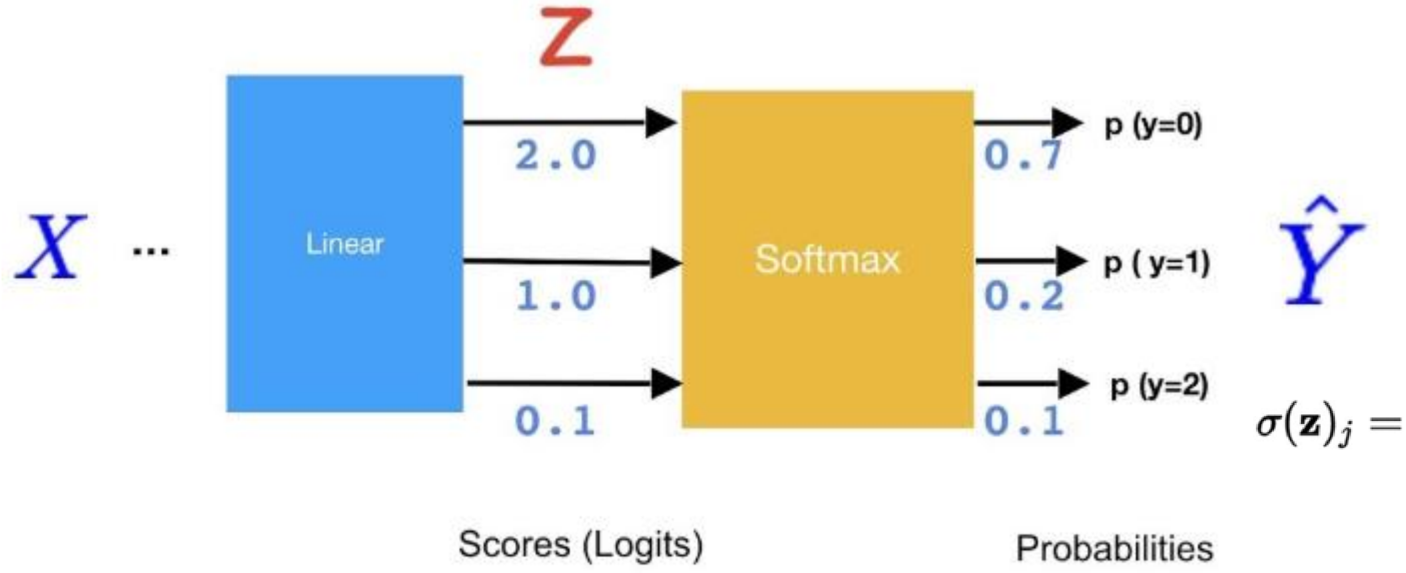
- ***One-vs.-Rest (OVR)* Approach**
  - A model is trained for each class that tries to separate that class from all of the other classes.
    → *c* models
  - To make a prediction, all models are run on a test point. The model that has the highest score on its single class "wins," and this class label is returned as the prediction.

- ***One-vs.-One (OVO) Approach***
  - A model is trained for each class pair → *c(c-1)/2* models
  - To make a prediction, the class label of a test data point is predicted based on majority voting by all models.

- **Multinomial logistic regression (*a.k.a.*, Softmax regression)**

Linear Models for Multi-class Classification

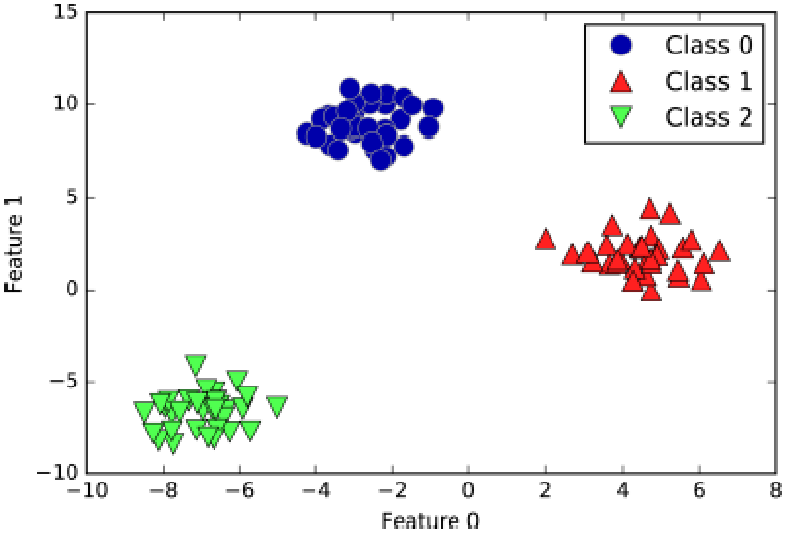


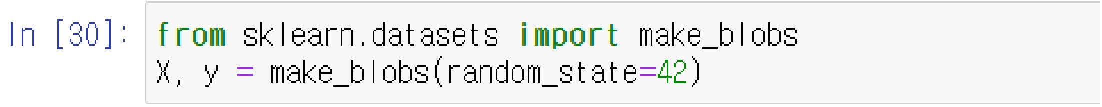


$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

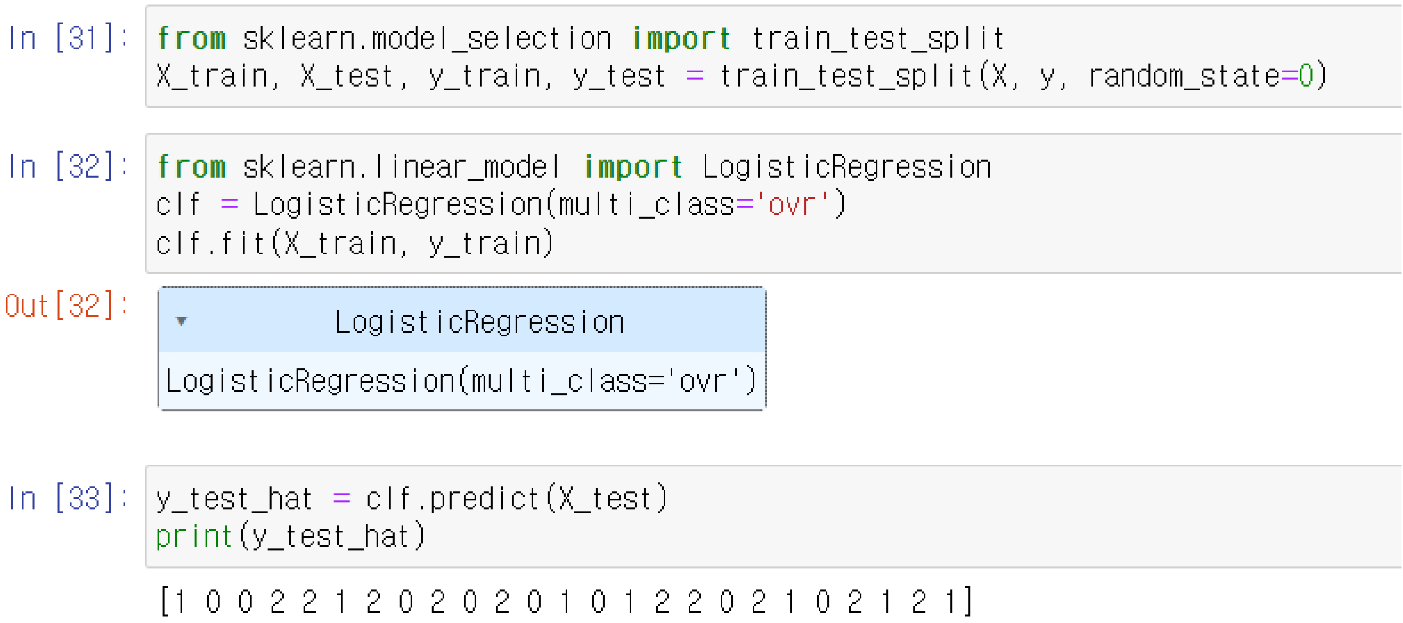
scikit-learn Practice: *LogisticRegression*

- **Example (*blobs* dataset)**





```
In [30]: from sklearn.datasets import make_blobs
         X, y = make_blobs(random_state=42)
```



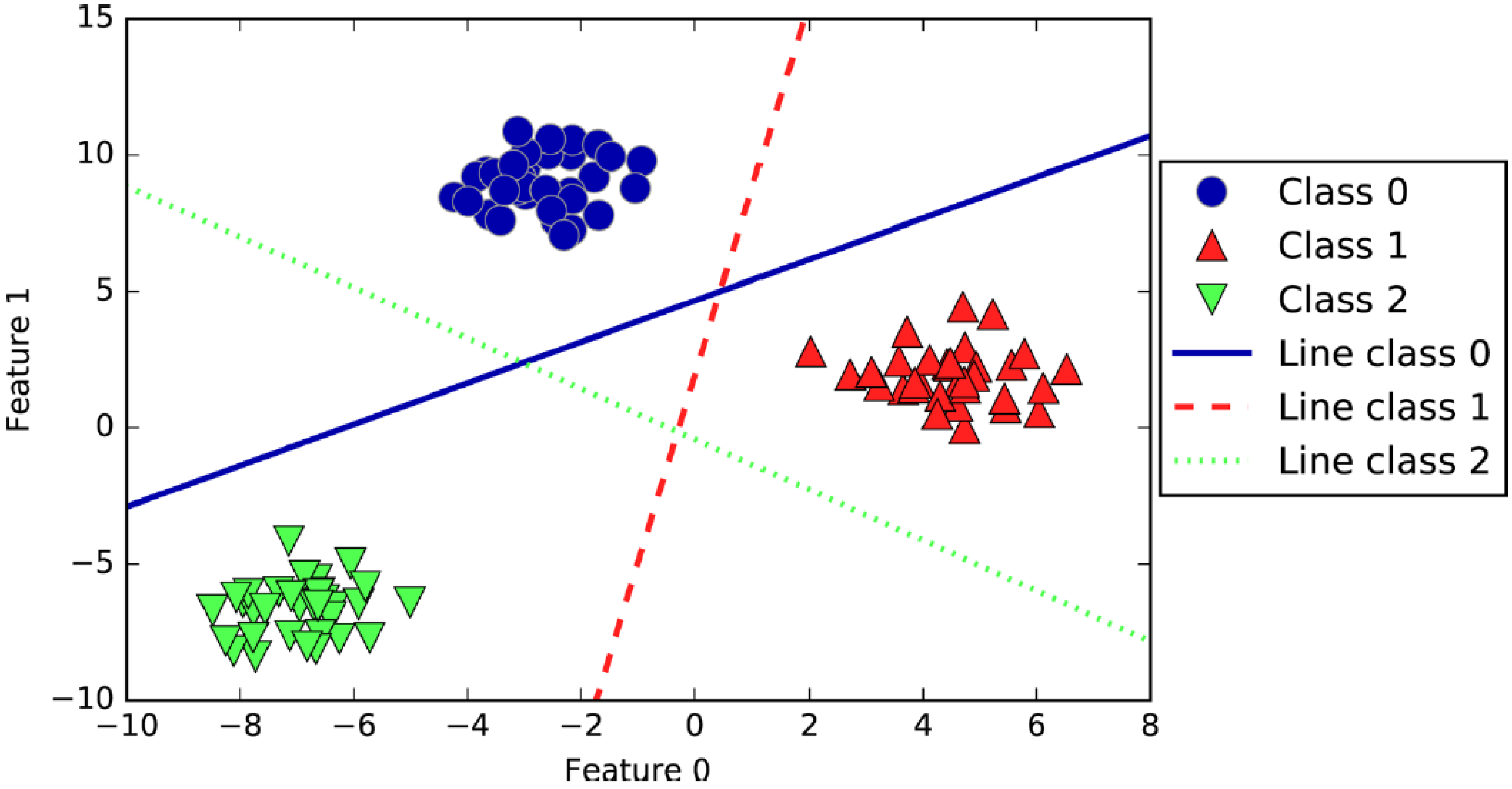
```
In [31]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

```
In [32]: from sklearn.linear_model import LogisticRegression
         clf = LogisticRegression(multi_class='ovr')
         clf.fit(X_train, y_train)

Out[32]: ▾ LogisticRegression
         LogisticRegression(multi_class='ovr')
```

```
In [33]: y_test_hat = clf.predict(X_test)
         print(y_test_hat)

[1 0 0 2 2 1 2 0 2 0 2 0 1 0 1 2 2 0 2 1 0 2 1 2 1]
```

scikit-learn Practice: *LogisticRegression*

- **Example (*blobs* dataset)**
  - Decision boundaries learned by the three models based on **the one-vs.-rest approach**



Discussion

- **The main hyperparameters of linear models**
  - The type of regularization (L1 vs L2)
  - The regularization strength hyperparameter α (or C)

  ** Typically chosen to have the highest performance in validation data*

  * It's important to preprocess your data (including *data scaling* and *one-hot encoding*)
- **Strengths**
  - Linear models are very fast to train, and also fast to predict.
  - They scale to very large datasets and work well with sparse data.
  - They make it relatively easy to understand how a prediction is made.
- **Weaknesses**
  - If your dataset has highly correlated features, it is often not entirely clear why coefficients are the way they are. (It is important to remove redundant features – feature selection)
  - They would perform worse if the relationship between features and target in your dataset is non-linear.

Discussion

- **Whether to use L1 regularization or L2 regularization**
  - Use L1 if
    - You have a large amount of features and assume that only a few of them are actually important.
    - You would like to have a model that is easy to interpret.
  - Use L2 otherwise
    - L2 regularization is usually the default choice