

Linear regression

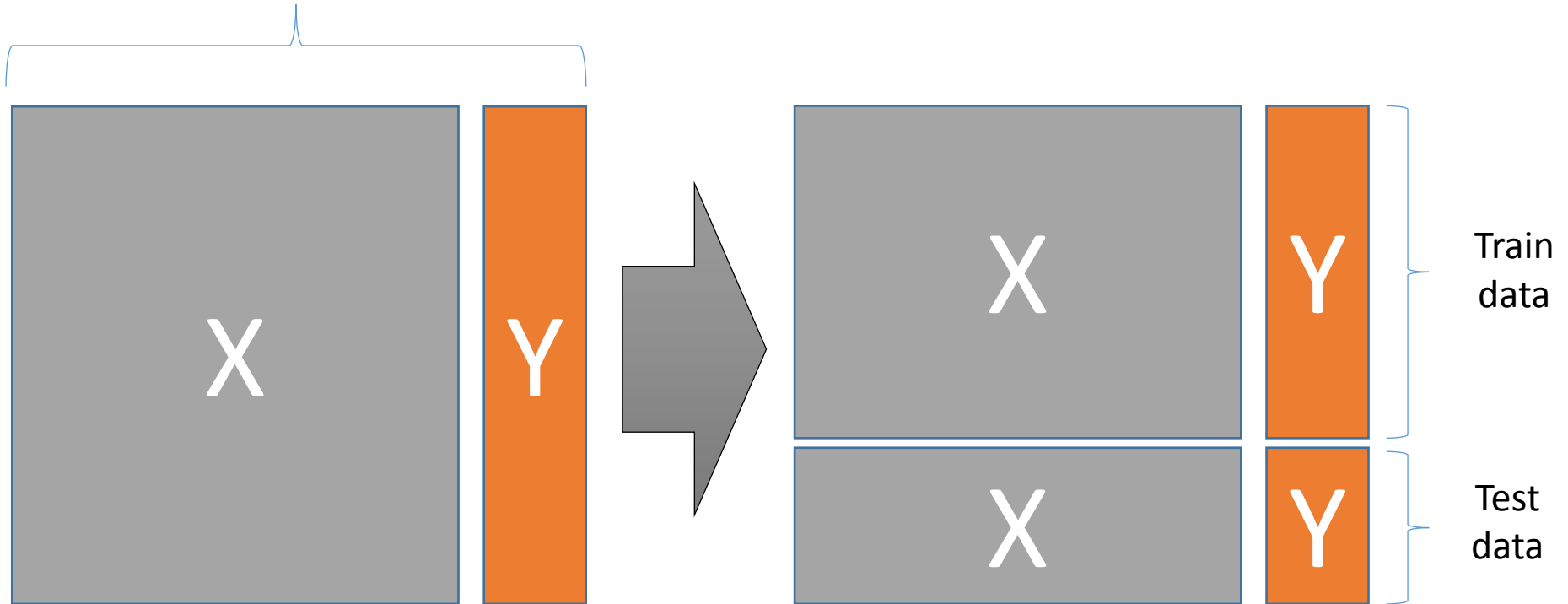
Taehoon Ko (thoon.koh@gmail.com)

Predictive modeling

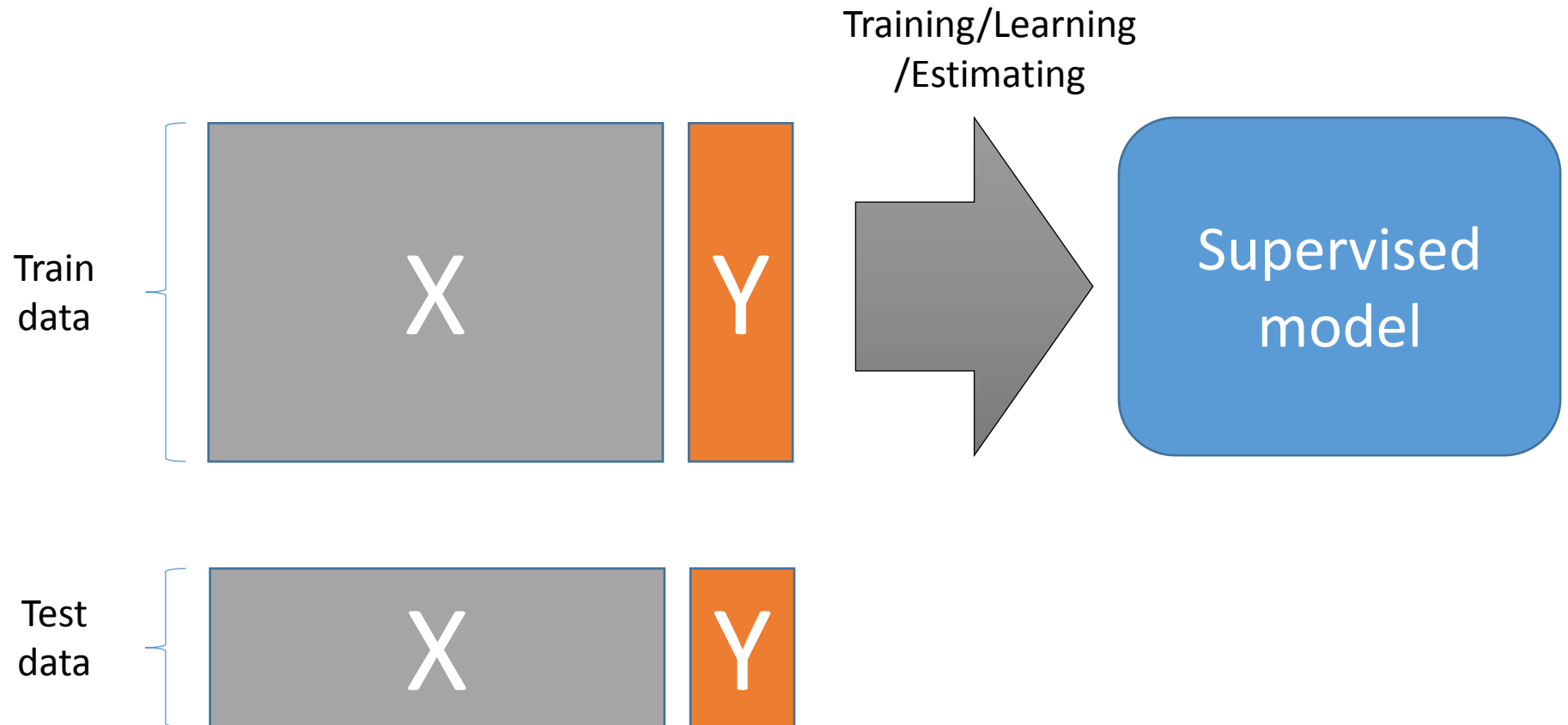
- Split the data into *train set* and *test set*
- Fit the model to *train set*
 - ➔ Test the model to *train set* and *test set*

Predictive modeling

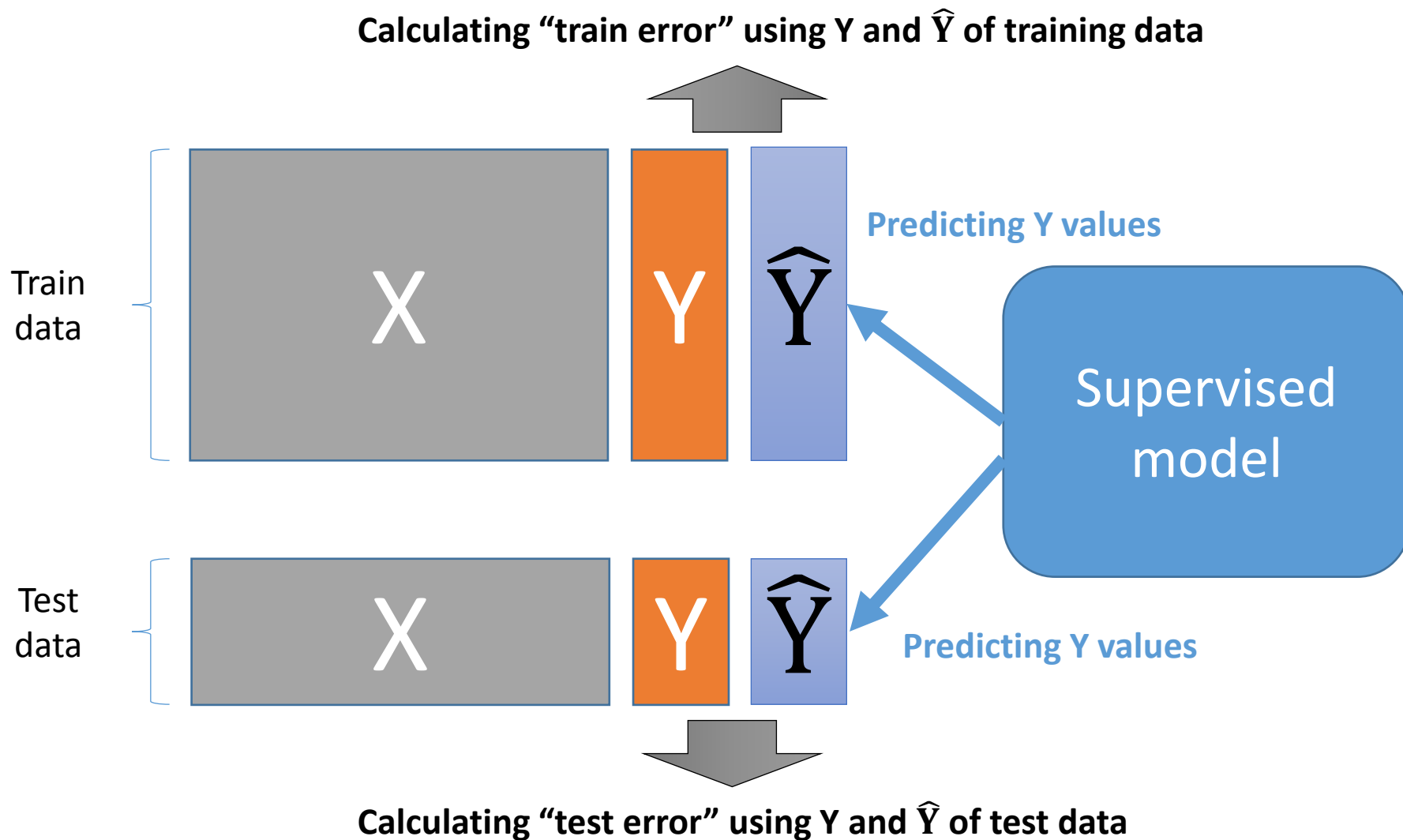
Given data



Predictive modeling



Predictive modeling



Predictive modeling: train_test_split

C. Splitting the data and Evaluating the model

- 앞에서는 데이터를 학습/테스트 셋으로 나누지 않고 선형회귀모델을 학습
- 여기에서는 데이터를 학습/테스트 셋으로 나누고 선형회귀모델의 **예측 성능**을 평가

```
In [34]: # from sklearn.cross_validation import train_test_split
from sklearn.model_selection import train_test_split
```

```
In [35]: # Training set: 70%, Test set: 30%
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=123)

print(X.shape)
print(Y.shape)
print(X_train.shape)
print(Y_train.shape)
print(X_test.shape)
print(Y_test.shape)

(10886, 7)
(10886,)
(7620, 7)
(7620,)
(3266, 7)
(3266,)
```

Predictive modeling: Calculate train_error and test_error

```
In [36]: linreg = LinearRegression()

# Fit the model using training set
linreg.fit(X_train, Y_train)

# Calculate predicted Y using X of test set
Y_train_pred = linreg.predict(X_train)
Y_test_pred = linreg.predict(X_test)

# Calculate RMSE in training/test sets
training_RMSE = np.sqrt(metrics.mean_squared_error(Y_train,
                                                    Y_train_pred))
test_RMSE = np.sqrt(metrics.mean_squared_error(Y_test,
                                                Y_test_pred))
```

```
In [37]: print('Training_RMSE: ', training_RMSE)
print('Test_RMSE: ', test_RMSE)
```

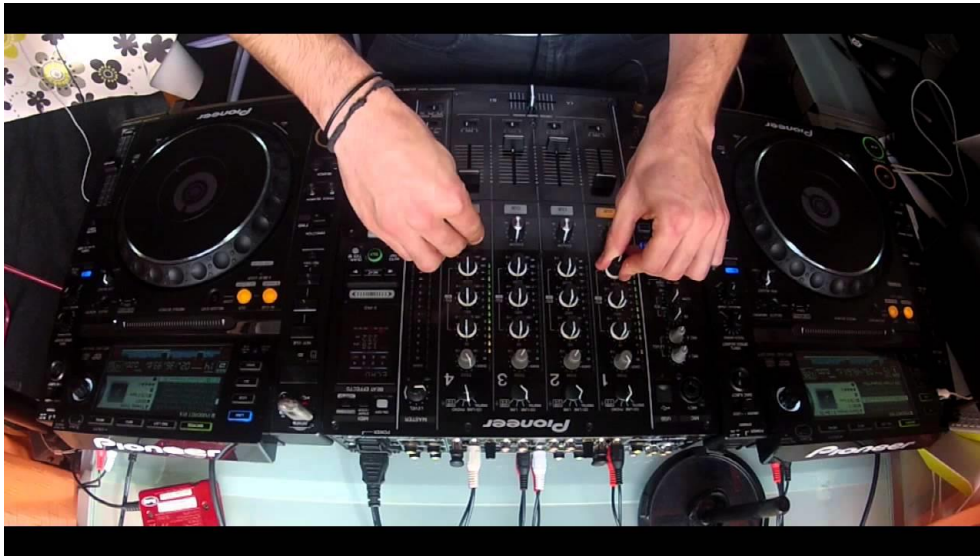
```
Training_RMSE: 129.439086786
Test_RMSE: 130.658969658
```

Parameter control

- All estimators (or models) have parameters.
- If you want to get good estimators, you have to control parameters!

Control parameters of the estimator

- **Club DJ, pilot and data scientist**
 - DJ set, airplane → machine learning model
 - Buttons, dials → parameters for model



Control parameters of the estimator

- Example: Ridge

`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.

Initial parameters

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]`).

Control parameters of the estimator

- **Example: Ridge**

- You can see parameters and responding values

'solver' parameter for *Ridge*

`solver : {'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag'}`

Solver to use in the computational routines:

- 'auto' chooses the solver automatically based on the type of data.
- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. More stable for singular matrices than 'cholesky'.
- 'cholesky' uses the standard `scipy.linalg.solve` function to obtain a closed-form solution.
- 'sparse_cg' uses the conjugate gradient solver as found in `scipy.sparse.linalg.cg`. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set `tol` and `max_iter`).
- 'lsqr' uses the dedicated regularized least-squares routine `scipy.sparse.linalg.lsqr`. It is the fastest but may not be available in old scipy versions. It also uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent. It also uses an iterative procedure, and is often faster than other solvers when both `n_samples` and `n_features` are large. Note that 'sag' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from `sklearn.preprocessing`.

All last four solvers support both dense and sparse data. However, only 'sag' supports sparse input when `fit_intercept` is `True`.

New in version 0.17: Stochastic Average Gradient descent solver.

Ridge and Lasso: Parameters

Simple linear regression vs. Ridge vs. Lasso

- **Objective functions (or cost functions)**

- Linear regression (f)

$$\min \sum_{i=1}^n \{y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip})\}^2$$

- Ridge (f + L2-norm regularization)

$$\min \sum_{i=1}^n \{y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip})\}^2 + \alpha \sum_{j=1}^p \beta_j^2$$

- Lasso (f + L1-norm regularization)

$$\min \sum_{i=1}^n \{y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip})\}^2 + \alpha \sum_{j=1}^p |\beta_j|$$

Simple linear regression vs. Ridge vs. Lasso

- **Objective functions (or cost functions)**

- Linear regression (f)

$$\min \sum_{i=1}^n \{y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip})\}^2$$

α (alpha): A parameter to control effectiveness of regularization

- Ridge (f + L2-norm regularization)

$$\min \sum_{i=1}^n \{y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip})\}^2 + \alpha \sum_{j=1}^p \beta_j^2$$

- Lasso (f + L1-norm regularization)

$$\min \sum_{i=1}^n \{y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip})\}^2 + \alpha \sum_{j=1}^p |\beta_j|$$

Parameter: α (alpha)

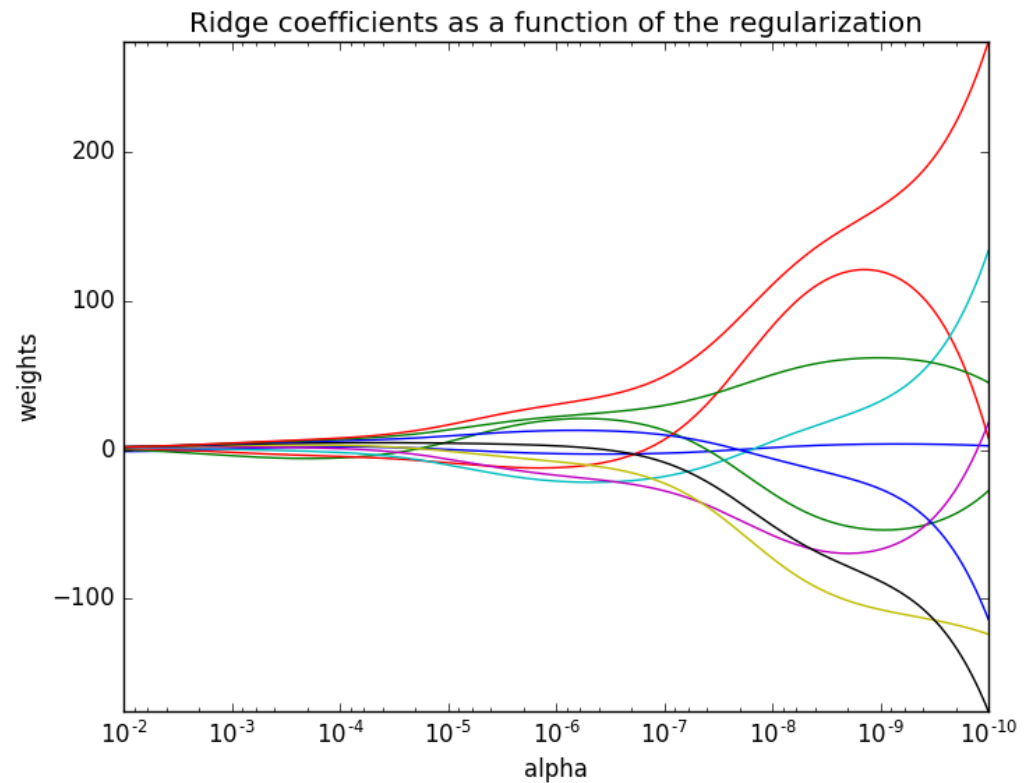
- If $\alpha = 0$,
 - Objective functions of Ridge and Lasso regressions are equal to that of linear regression.
→ *No regularization!*
- If α becomes larger,
 - Ridge and Lasso try to minimize L2 penalty term ($\alpha \sum_{j=1}^p \beta_j^2$) and L1 penalty term ($\alpha \sum_{j=1}^p |\beta_j|$) when they are trained.
→ $\sum_{j=1}^p \beta_j^2$ and $\sum_{j=1}^p |\beta_j|$ becomes smaller, if α becomes larger.
→ *Strong regularization!*

Small α → Regularization strength is weak.
Large α → Regularization strength is strong.

Parameter: α (alpha)

- In Ridge,

- The larger α , the greater the amount of shrinkage.
 - The absolute value of all the coefficients becomes small.
- We expect that the coefficients become more robust to collinearity.



Parameter: α (alpha)

- In Lasso,

- The larger α , the fewer the non-zero coefficients.
 - ***Sparse*** modeling
- It is useful in some contexts due to its tendency to prefer solutions with fewer parameter values, effectively reducing the number of variables upon which the given solution is dependent.

Parameter: α (alpha)

- You can control this parameter when using Ridge and Lasso in scikit-learn.

```
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)
```

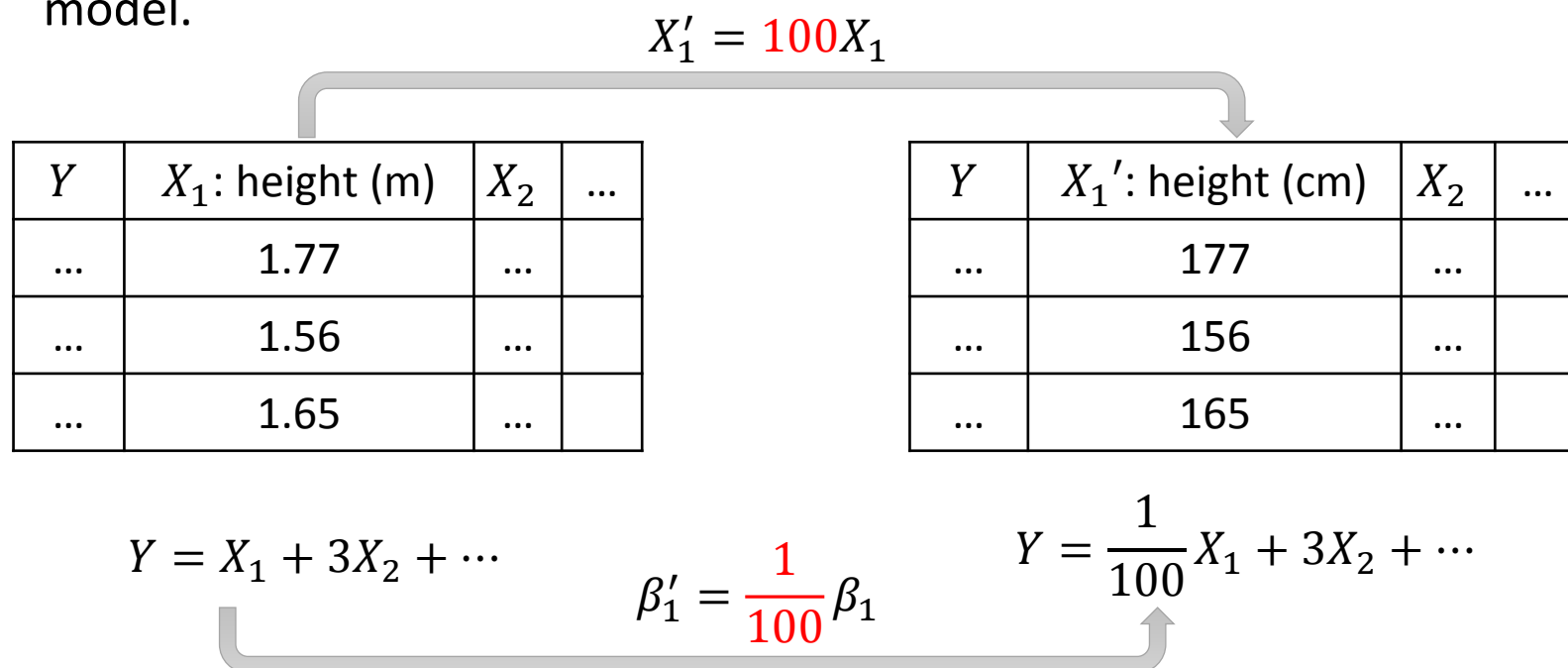
[source]

```
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]

Parameter: *normalize*

- Is feature scaling (such as normalization or standardization) important in the OLS regression model?
 - In terms of interpretability, it can be useful.
 - However, it does not affect the predictive performance of the OLS regression model.



Parameter: *normalize*

- In Ridge and Lasso, feature scaling affects the process of optimizing objective (or cost) function.

- Cost function of OLS regression

$$\min \sum_{i=1}^n \{y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip})\}^2$$



Absolute values of coefficients affect penalty terms.

- Cost function of Ridge

$$\min \sum_{i=1}^n \{y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip})\}^2 + \alpha \sum_{j=1}^p \beta_j^2$$

- Cost function of Lasso

$$\min \sum_{i=1}^n \{y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_p x_{ip})\}^2 + \alpha \sum_{j=1}^p |\beta_j|$$

If input features does not have the same scale, they have different contribution to the penalty term.

Parameter: *normalize*

- You can control this parameter when using Ridge and Lasso in scikit-learn.

```
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)
```

[\[source\]](#)

```
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') ¶
```

[\[source\]](#)

[***normalize=True***]

In fact, each input feature is centered to have mean zero. And then the function `sklearn.preprocessing.normalize(norm='l2')` is applied to centered data.

If you want this logic, please see these pages (scikit-learn source [Link1](#), [Link2](#)) or following side notes.

Parameter: *normalize*

- If you wish to standardize, please use *preprocessing.StandardScaler* before calling fit on an estimator (Ridge or Lasso) with *normalize=False*.
- However, as you see, using *normalize=True* is recommended in the manual.

normalize : boolean, optional, default False

If `True`, the regressors X will be normalized before regression. This parameter is ignored when `fit_intercept` is set to `False`. When the regressors are normalized, note that this makes the hyperparameters learnt more robust and almost independent of the number of samples. The same property is not valid for standardized data. However, if you wish to standardize, please use `preprocessing.StandardScaler` before calling `fit` on an estimator with `normalize=False`.

*Side note: *normalize=True* in Ridge, Lasso

```
X = np.asarray([[1, 2, 3], [2, 4, 7], [3, 6, 9]], dtype=np.float32)
```

```
print(X)
```

```
[[ 1.  2.  3.]
 [ 2.  4.  7.]
 [ 3.  6.  9.]]
```

- Zero-centered and l2-norm normalization for each input feature

```
# mean vector for centering data
X_offset = np.average(X, axis=0)
print(X_offset)
```

```
[ 2.          4.          6.33333349]
```

```
# Centered data
X_centered = X - X_offset
print(X_centered)
```

```
[[-1.          -2.          -3.33333349]
 [ 0.           0.           0.66666651]
 [ 1.           2.           2.66666651]]
```

```
X_normalized, X_centered_norm = normalize(X_centered, axis=0,
                                           copy=False, return_norm=True)
```

```
print(X_normalized)
print("=" * 60)
print(X_centered_norm)
```

```
[[-0.70710677 -0.70710677 -0.77151674]
 [ 0.          0.          0.1543033 ]
 [ 0.70710677  0.70710677  0.61721331]]
```

```
=====
[ 1.41421354  2.82842708  4.32049417]
```

- l2-norm for 1st feature in centered data
: $\sqrt{(-1)^2 + 0^2 + 1^2} = 1.41421354$
- l2-norm for 2nd feature in centered data
: $\sqrt{(-2)^2 + 0^2 + 2^2} = 2.82842708$
- l2-norm for 3rd feature in centered data
: $\sqrt{(-3.33)^2 + 0.66^2 + 2.66^2} = 4.32049417$

- normalized 1st feature
: $\left[\frac{-1}{1.4142}, 0, \frac{1}{1.4142} \right] = [-0.7071, 0, 0.7071]$
- ...

*Side note: *Standardization*

- Standardization: Calculating means and standard deviations

```
# mean vector
X_offset = np.average(X, axis=0)

# standard deviations
X_std = np.std(X, axis=0)

print(X_offset)
print(X_std)
```

```
[ 2.          4.          6.33333349]
[ 0.81649661  1.63299322  2.49443841]
```

```
X_standardized_v2 = (X - X_offset) / X_std
print(X_standardized_v2)
```

```
[[-1.2247448  -1.2247448  -1.33630621]
 [ 0.          0.          0.26726115]
 [ 1.2247448   1.2247448   1.06904483]]
```


- Standardization: Using `sklearn.preprocessing.StandardScaler`

```
scaler = StandardScaler()
X_standardized = scaler.fit_transform(X)
print(X_standardized)
```

```
[[-1.2247448  -1.2247448  -1.33630621]
 [ 0.          0.          0.26726115]
 [ 1.2247448   1.2247448   1.06904483]]
```



*Side note: *normalize=True* vs. *Standardization*

- **Standardization** also center the data by subtracting mean values like previous method.


$$\sqrt{\sum_{i=1}^n x_{centered,i}^2}$$

- **Difference**

- Previous method calculates l2-norm values using zero-centered data, and centered data is divided by l2-norm values.
- Standardization calculates standard deviations of original data, and centered data is divided by standard deviations.


$$\sqrt{variance} = \sqrt{\frac{1}{n} \sum_{i=1}^n \left(x_i - \frac{1}{n} \sum_{i=1}^n x_i \right)^2}$$

*Side note: L^p norm

- Norm

- In a narrow sense, a **norm** can be considered as the size of a vector.
 - A **norm** is a function that assigns a *strictly positive length* or *size* to each vector in a vector space (in Wikipedia).

- Given a vector $\mathbf{x} = (x_1, x_2, \dots, x_n)$, L^p -norm of \mathbf{x} is defined by

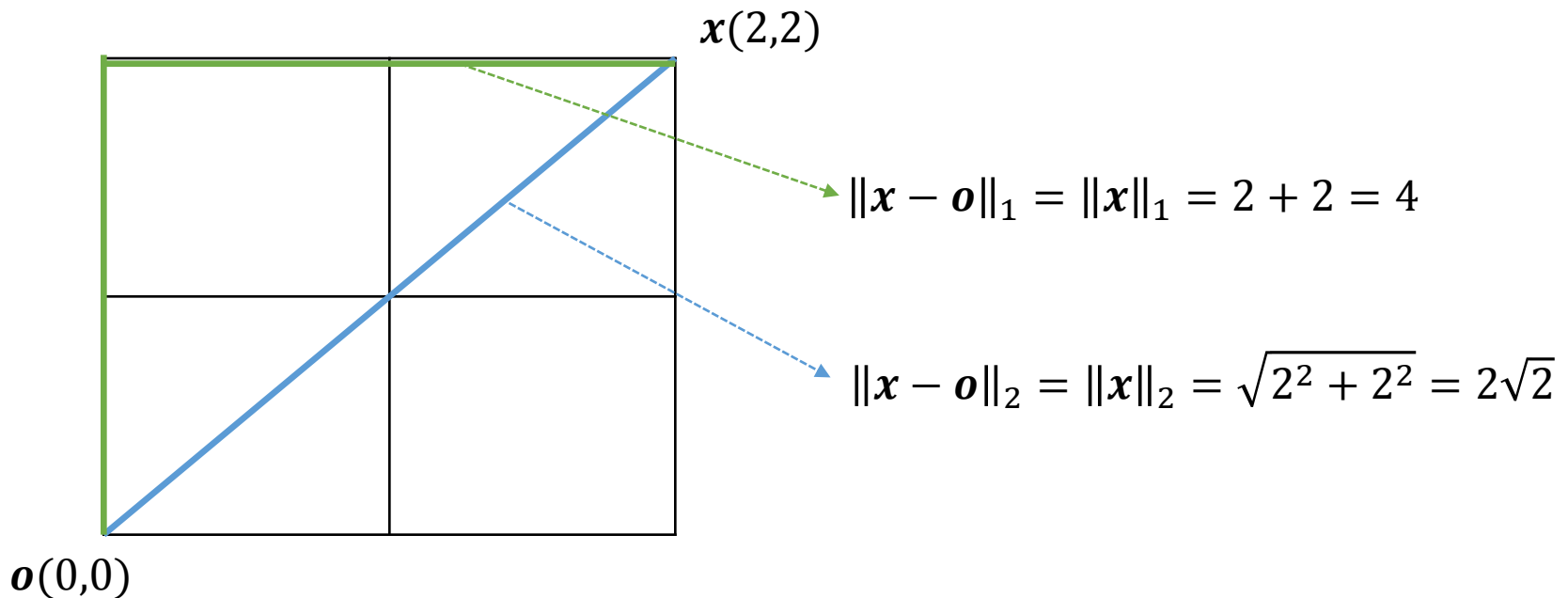
$$\|\mathbf{x}\|_p = (|x_1|^p + |x_2|^p + \dots + |x_n|^p)^{\frac{1}{p}}.$$

- $l1$ -norm: $\|\mathbf{x}\|_1 = |x_1| + |x_2| + \dots + |x_n|$

- $l2$ -norm: $\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2}$

*Side note: L^p norm

- $l1$ -norm: $\|\mathbf{x}\|_1 = |x_1| + |x_2| + \cdots + |x_n|$
 - taxicab distance or Manhattan distance between \mathbf{x} and the origin.
- $l2$ -norm: $\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$
 - Euclidean distance between \mathbf{x} and the origin.



Parameter: *normalize*

- **In practice,**

- Feature scaling do an important role when input features have significantly different scales.
- However, feature scaling (such as *setting the parameter 'normalize' as True or standardizing the data*) does not always guarantee good predictive performance in Ridge and Lasso.
 - Try and test them.

Feature engineering: Polynomial features

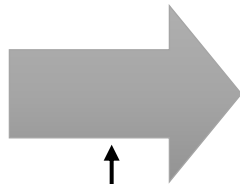
Polynomial regression

- **Linear regression vs. polynomial regression**

- Unlike linear regression models, polynomial regression models can fit a non-linear relationship between X_i and Y .
- By '*PolynomialFeatures*' class in *scikit-learn*, we can make polynomial features easily.

X_1	X_2	Y

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2$$



X_1	X_2	X_1^2	$X_1 X_2$	X_2^2	Y

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1^2 + \beta_4 X_1 X_2 + \beta_5 X_2^2$$

`PolynomialFeatures(degree=2, interaction_only=False, include_bias=False)`

PolynomialFeatures

`sklearn.preprocessing.PolynomialFeatures`

```
class sklearn.preprocessing. PolynomialFeatures (degree=2, interaction_only=False, include_bias=True)
```

[\[source\]](#)

Generate polynomial and interaction features.

Generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form $[a, b]$, the degree-2 polynomial features are $[1, a, b, a^2, ab, b^2]$.

Parameters: **degree** : integer

The degree of the polynomial features. Default = 2.

interaction_only : boolean, default = False

If true, only interaction features are produced: features that are products of at most `degree` *distinct* input features (so not `x[1] ** 2`, `x[0] * x[2] ** 3`, etc.).

include_bias : boolean

If True (default), then include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model).

PolynomialFeatures

- **In practice,**

- If your simple linear regression model fits poorly to the train set, polynomial regression can be a good alternative model.
 - After fitting and transform your input data X using '*PolynomialFeatures*' class, train the regression models.
- Caution!
 - '*PolynomialFeatures*' class generates many features.
 - You can encounter the 'curse of dimensionality' problem.
 - If you want to avoid this problem, you should try to
 - use regularized models such as Ridge, Lasso, ElasticNet, etc.
 - reduce the input dimension by selecting or extracting features (will be explained later).