

現場で使える 機械学習・データ分析基礎講座

第 2 章：教師あり学習の基礎

ノートブック解説

- 各章のノートブックについて、セルのスクリーンショットやセルの出力を見ながらコードの意味を解説
 - NumPy, Pandas, Matplotlib, Seaborn の基本操作は習得済みとする
- ノートブック演習において、分からぬいコードが出てきたときに活用しよう
 - 自身でも、適宜公式ドキュメントを参照すること（メソッドの引数など）

```
# 入力データx = [0, 1, 2, ..., 99]
x = np.arange(100)

# 教師データy
# 平均0, 標準偏差10の正規分布からサンプリングした乱数（ノイズ）をxに加えたもの
np.random.seed(seed=1234)
y = np.random.normal(loc=x, scale=10)

# 理想的なモデルの出力
# ノイズなしのデータ、つまりxそのもの
y_true = x.copy()
```

■ 線形回帰モデル

- 2-1_linear_regression_psedo_data.ipynb
- 2-2_linear_regression_real_data.ipynb (_trainee あり)
- 2-3_linear_regression_multi_psedo_data.ipynb
- 2-4_linear_regression_multi_real_data.ipynb (_trainee あり)

■ ロジスティック回帰モデル

- 2-5_logistic_regression_psedo_data.ipynb
- 2-6_logistic_regression_real_data.ipynb (_trainee あり)

線形回帰モデル

2-1_linear_regression_psedo_data

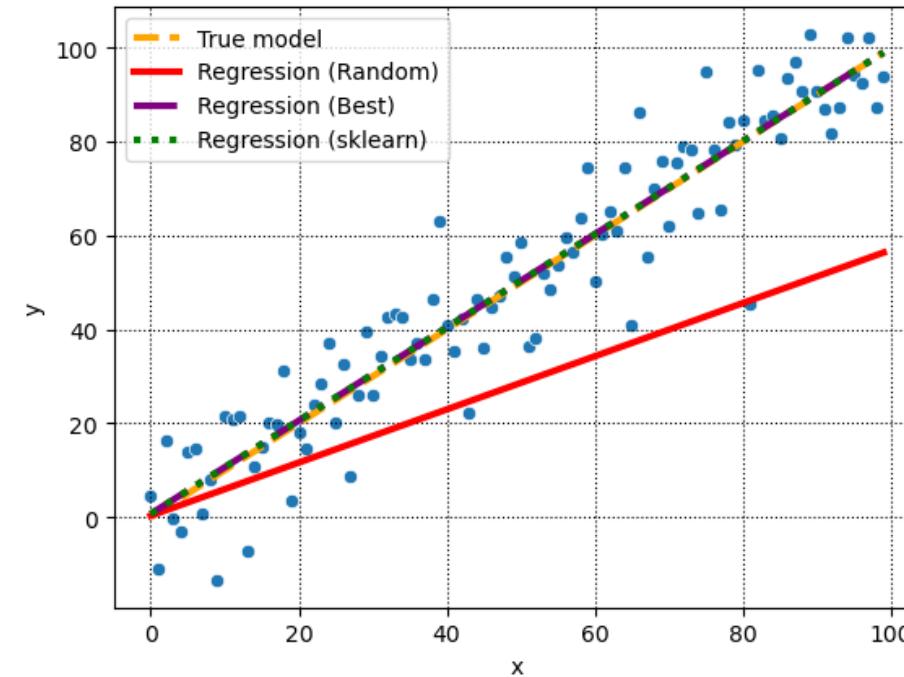
2-1_linear_regression_psedo_data

■ 内容

- 疑似データを用いて線形回帰モデルを作成
- ランダムに決めたパラメータを用いるよりも、
最小二乗法によって求めたパラメータを用いる方が二乗誤差が小さくなることを確認

■ 手順

1. ライブラリの読み込み
2. 疑似データの作成
3. パラメータをランダムに設定
4. 最適なパラメータを計算
5. scikit-learn を用いたモデルの学習



2-1 | 1. ライブラリの読み込み

```
# グラフをセルの出力部分に表示
%matplotlib inline

# 基本的なライブラリの読み込み
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# 線形回帰モデル
from sklearn.linear_model import LinearRegression
# MSE (平均二乗誤差)
from sklearn.metrics import mean_squared_error
```

2-1 | 2. 疑似データの作成

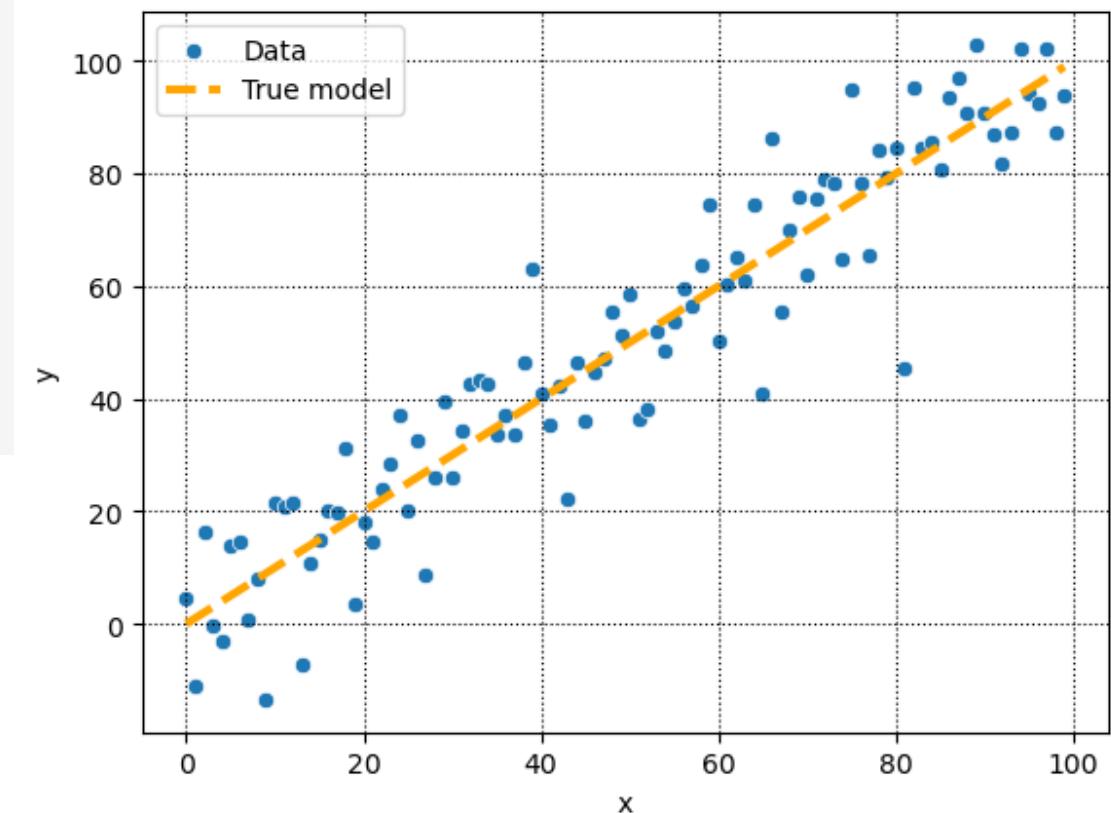
```
# 入力データx = [0, 1, 2, ... , 99]
x = np.arange(100)

# 教師データy
# 平均0, 標準偏差10の正規分布からサンプリングした乱数（ノイズ）をxに加えたもの
np.random.seed(seed=1234)
y = np.random.normal(loc=x, scale=10)

# 理想的なモデルの出力
# ノイズなしのデータ、つまりxそのもの
y_true = x.copy()
```

2-1 | 2. 疑似データの作成

```
# データの散布図
sns.scatterplot(x=x, y=y, label='Data', marker='o')
# 理想的なモデル（点線）
sns.lineplot(x=x, y=y_true, linestyle='--', linewidth=3, color='orange', label='True model')
# グリッド線の表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')
# 軸ラベルの設定
plt.xlabel('x')
plt.ylabel('y')
# グラフの表示
plt.show()
```



2-1 | 3. パラメータをランダムに設定

```
# パラメータwの値を、まずはデタラメに決めてみる
# 実行するたびに結果が変わる
w = np.random.normal(loc=0, scale=1, size=2)
y_est = w[0] + w[1] * x
print('w0 = {:.3f}, w1 = {:.3f}'.format(w[0], w[1]))
```

w0 = 0.291, w1 = 0.567

2-1 | 3. パラメータをランダムに設定

```
# データの散布図
sns.scatterplot(x=x, y=y, marker='o')

# オレンジ色の点線：理想的なモデル
sns.lineplot(x=x, y=y_true, linestyle='--', linewidth=3, color='orange', label='True model')

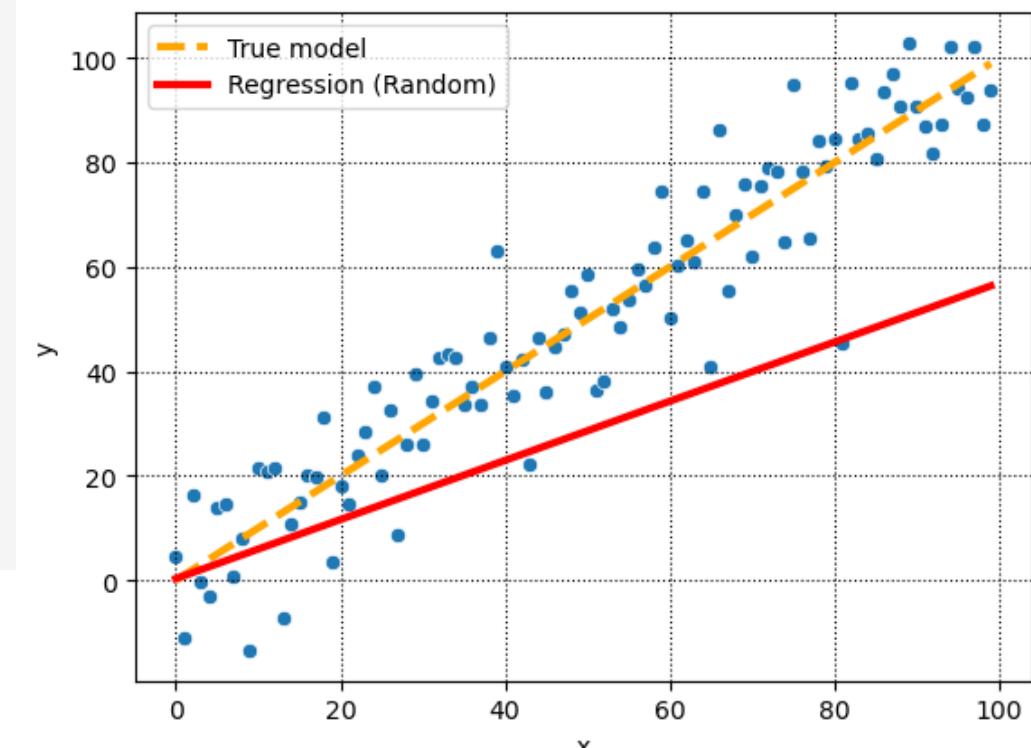
# 赤色の実線：パラメータをデタラメに決めた場合のモデル
sns.lineplot(x=x, y=y_est, linestyle='-', linewidth=3, color='red', label='Regression (Random)')

# グリッド線の表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')

# 軸ラベルの設定
plt.ylabel('y')
plt.xlabel('x')

# 凡例の表示位置を調整
plt.legend(loc='best')

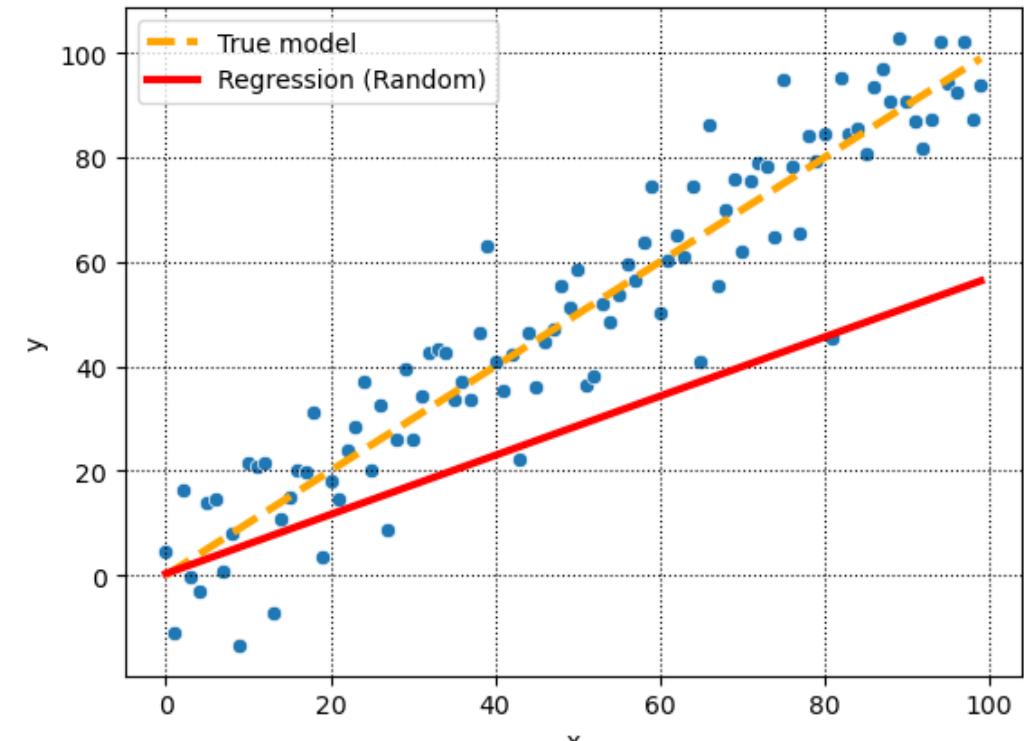
plt.show()
```



2-1 | 3. パラメータをランダムに設定

```
# モデルの評価値を求める
# モデルの出力と教師データとの二乗誤差を計算
squared_error = mean_squared_error(y, y_est)
print('w0 = {:.3f}, w1 = {:.3f}, 二乗誤差 = {:.3f}'.format(w[0], w[1], squared_error))
```

w0 = 0.291, w1 = 0.567, 二乗誤差 = 713.633



2-1 | 4. 最適なパラメータを計算

```
# 最小二乗法を用いて最適なパラメータを計算
phi = np.ones((len(x), 2))
phi[:, 1] = x

# 注：行列積は * ではなく @ で計算する
w_best = np.linalg.inv(phi.T @ phi) @ phi.T @ y
# またはnp.dot(A, B)で計算
# w_best = np.dot(np.dot(np.linalg.inv(np.dot(phi.T, phi))), phi.T), y)

# 求めた最適なパラメータを使ってモデルの出力を計算
y_est_best = w_best[0] + w_best[1] * x

squared_error = mean_squared_error(y, y_est_best)
print('w0 = {:.3f}, w1 = {:.3f}, 二乗誤差 = {:.3f}'.format(w_best[0], w_best[1], squared_error))
```

w0 = 0.696, w1 = 0.993, 二乗誤差 = 99.098

最適なパラメータ

$$\mathbf{w}^* = (\boldsymbol{\Phi}^T \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^T \mathbf{y}$$

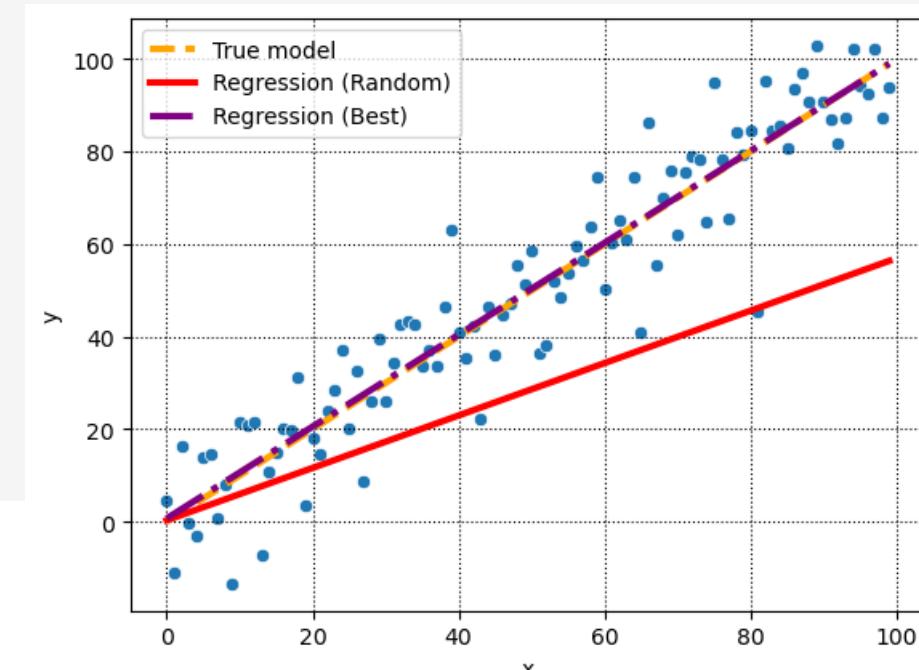
計画行列

$$\boldsymbol{\Phi} = \begin{pmatrix} \mathbf{x}^{(1)^T} \\ \vdots \\ \mathbf{x}^{(N)^T} \end{pmatrix}$$

2-1 | 4. 最適なパラメータを計算

```
# データの散布図
sns.scatterplot(x=x, y=y, marker='o')
# オレンジ色の点線：理想的なモデル
sns.lineplot(x=x, y=y_true, linestyle='--', linewidth=3, color='orange', label='True model')
# 赤色の実線：パラメータをデタラメに決めた場合のモデル
sns.lineplot(x=x, y=y_est, linestyle='-', linewidth=3, color='red', label='Regression (Random)')
# 紫色の点線：最適なパラメータを最小二乗法で計算した場合のモデル
sns.lineplot(x=x, y=y_est_best, linestyle='-.', linewidth=3, color='purple', label='Regression (Best)')

# グリッド線の表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')
# 軸ラベルの設定
plt.ylabel('y')
plt.xlabel('x')
# 凡例の表示位置を調整
plt.legend(loc='best')
plt.show()
```



2-1 | 5. scikit-learn を用いたモデルの学習

```
# 入力データの形状を変換
# scikit-learnでは、入力Xは必ず2次元の配列にする必要あり
X = x.reshape(-1,1)
```

```
# 線形回帰モデルをインスタンス化
# fit_intercept=False に設定すると、切片を計算しない
# 目的変数が原点を必ず通る性質のデータを扱うときに利用(デフォルト値: True)
regr = LinearRegression(fit_intercept=True)

# モデルを学習させる
regr.fit(X, y)

# モデルから予測を得る
y_est_sklearn = regr.predict(X)

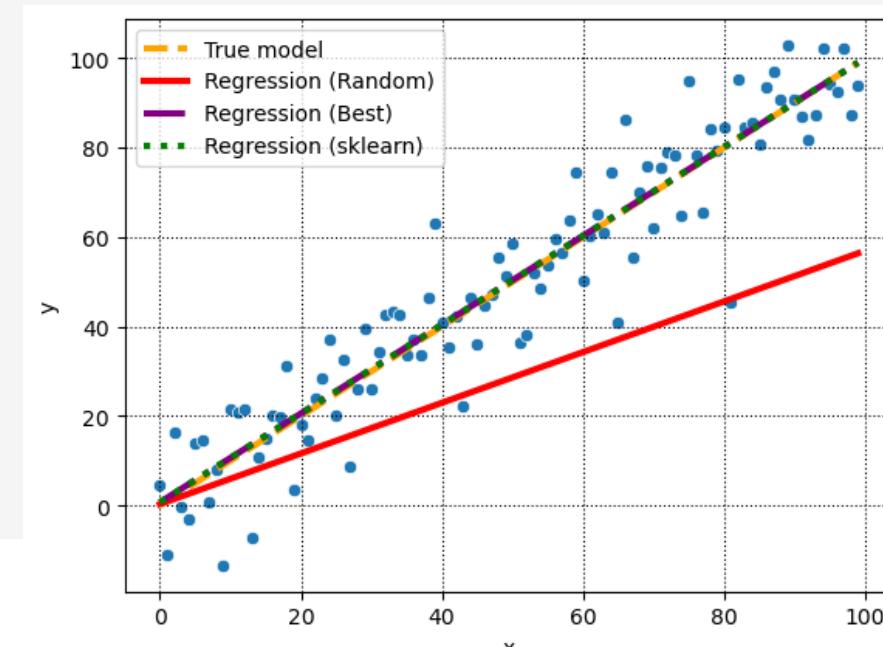
# モデルの評価値を計算
squared_error = mean_squared_error(y, y_est_sklearn)
print("w0 = {:.3f}, w1 = {:.3f}, 二乗誤差 = {:.3f}".format(regr.intercept_, regr.coef_[0], squared_error))
```

w0 = 0.696, w1 = 0.993, 二乗誤差 = 99.098

2-1 | 5. scikit-learn を用いたモデルの学習

```
# データの散布図
sns.scatterplot(x=x, y=y, marker='o')
# オレンジ色の点線：理想的なモデル
sns.lineplot(x=x, y=y_true, linestyle='--', linewidth=3, color='orange', label='True model')
# 赤色の実線：パラメータをデタラメに決めた場合のモデル
sns.lineplot(x=x, y=y_est, linestyle='-', linewidth=3, color='red', label='Regression (Random)')
# 紫色の点線：最適なパラメータを最小二乗法で計算した場合のモデル
sns.lineplot(x=x, y=y_est_best, linestyle='-.', linewidth=3, color='purple', label='Regression (Best)')
# 緑色の点線：scikit-learnを用いて学習させた場合のモデル
sns.lineplot(x=x, y=y_est_sklearn, linestyle=':', linewidth=3, color='green', label='Regression (sklearn)')

# グリッド線の表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')
# 軸ラベルの設定
plt.ylabel('y')
plt.xlabel('x')
# 凡例の表示位置を調整
plt.legend(loc='best')
plt.show()
```



2-2_linear_regression_real_data

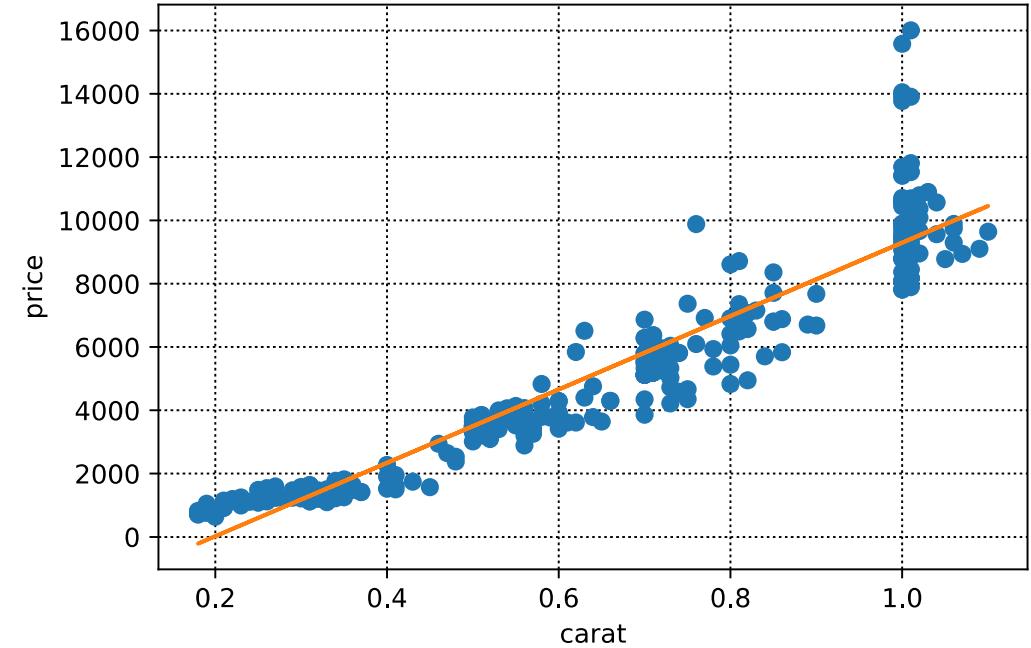
2-2_linear_regression_real_data

■ 内容

- ・ 実際のデータで線形回帰を実行
- ・ ダイヤモンドのカラット数からその価格を予測するモデルを構築

■ 手順

1. ライブラリの読み込み
2. データの読み込み
3. [演習] モデルの学習



2-2 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
```

2-2 | 2. データの読み込み

```
# CSVファイルの読み込み
df_diamond = pd.read_csv("../../../1_data/ch2/diamond_data.csv")

# ダイヤモンドの重さの単位であるカラットとその価格に関する実際のデータ
# 『回帰分析入門』より引用
display(df_diamond.head())
df_diamond.describe()
```

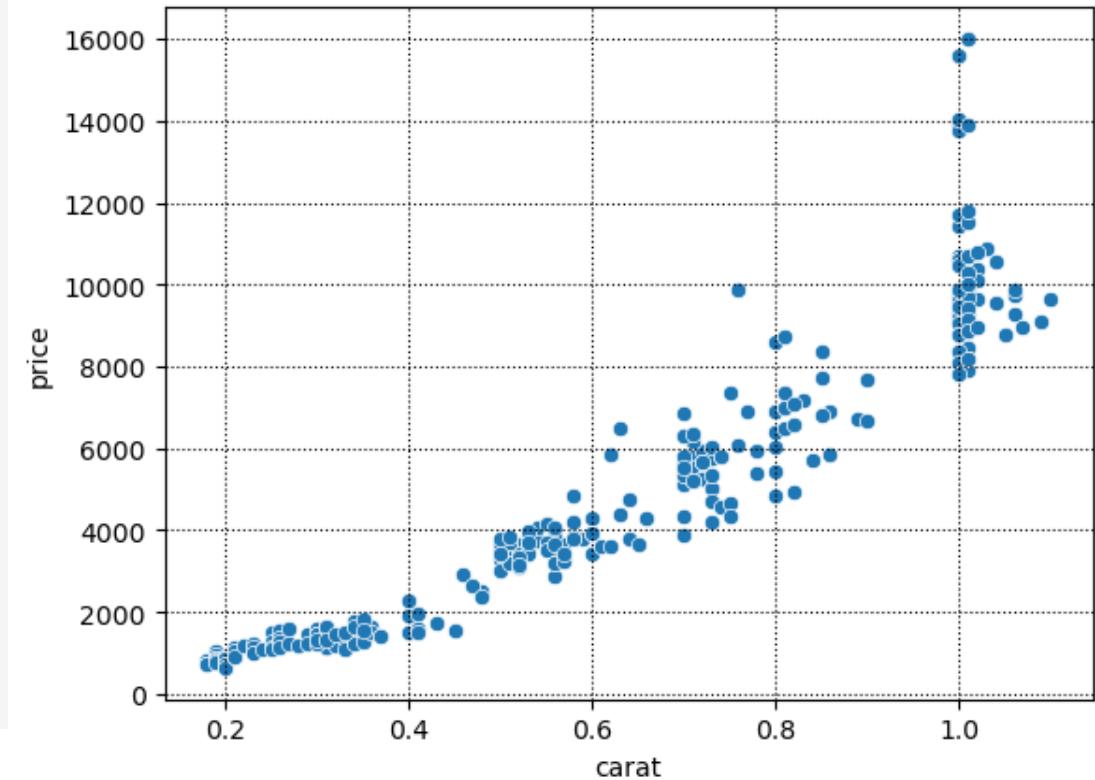
	carat	price
0	0.30	1302
1	0.30	1510
2	0.30	1510
3	0.30	1260
4	0.31	1641

	carat	price
count	308.000000	308.000000
mean	0.630909	5019.483766
std	0.277183	3403.115715
min	0.180000	638.000000
25%	0.350000	1625.000000
50%	0.620000	4215.000000
75%	0.850000	7446.000000
max	1.100000	16008.000000

2-2 | 2. データの読み込み

```
# 学習用データ作成
x = df_diamond["carat"].values
# 正解データ作成
y = df_diamond["price"].values

# データの散布図
sns.scatterplot(x=x, y=y, marker='o')
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')
plt.ylabel('price')
plt.xlabel('carat')
plt.show()
```



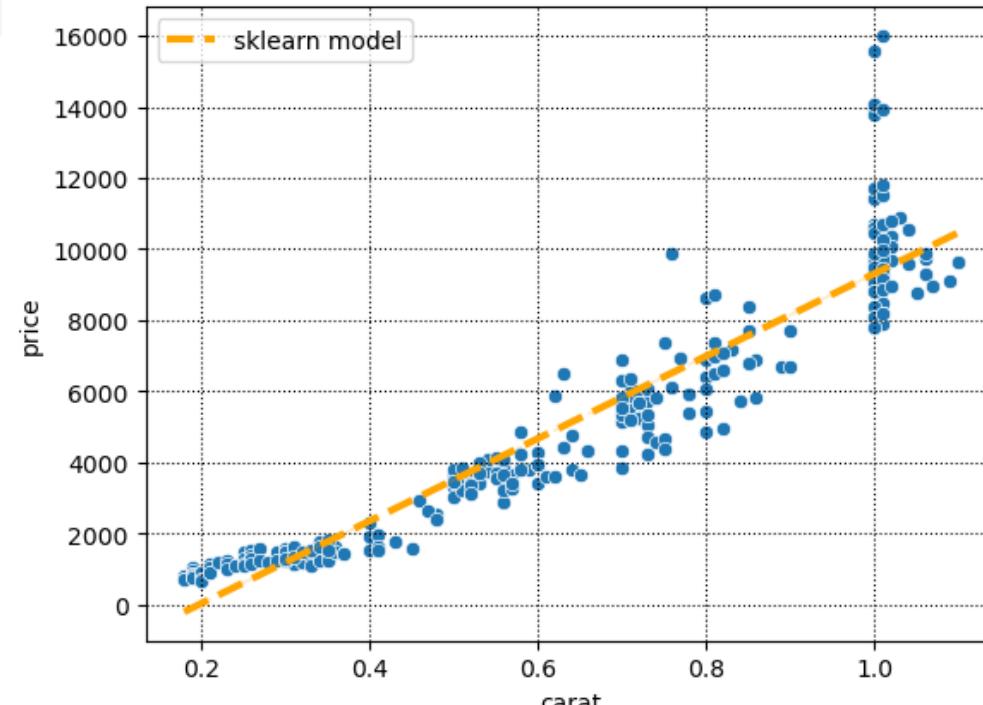
2-2 | 3. [演習] モデルの学習

```
# scikit learn で使用できる2次元の配列へ変換
X = x.reshape(-1,1)
# 線形回帰モデルをインスタンス化
regr = LinearRegression(fit_intercept=True)
# モデルを学習させる
regr.fit(X, y)
# モデルから予測を得る
y_est_sklearn = regr.predict(X)
squared_error = mean_squared_error(y, y_est_sklearn)
print("w0 = {:.3f}, w1 = {:.3f}, 二乗誤差 = {:.3f}".format(regr.intercept_, regr.coef_[0], squared_error))
```

w0 = -2298.358, w1 = 11598.884, 二乗誤差 = 1240839.690

2-2 | 3. [演習] モデルの学習

```
# グラフに重ねて表示する
sns.scatterplot(x=x, y=y, marker='o')
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')
plt.ylabel('price')
plt.xlabel('carat')
sns.lineplot(x=x, y=y_est_sklearn, linestyle='--', linewidth=3, color='orange', label='sklearn model')
plt.show()
```



2-3_linear_regression_multi_psedo_data

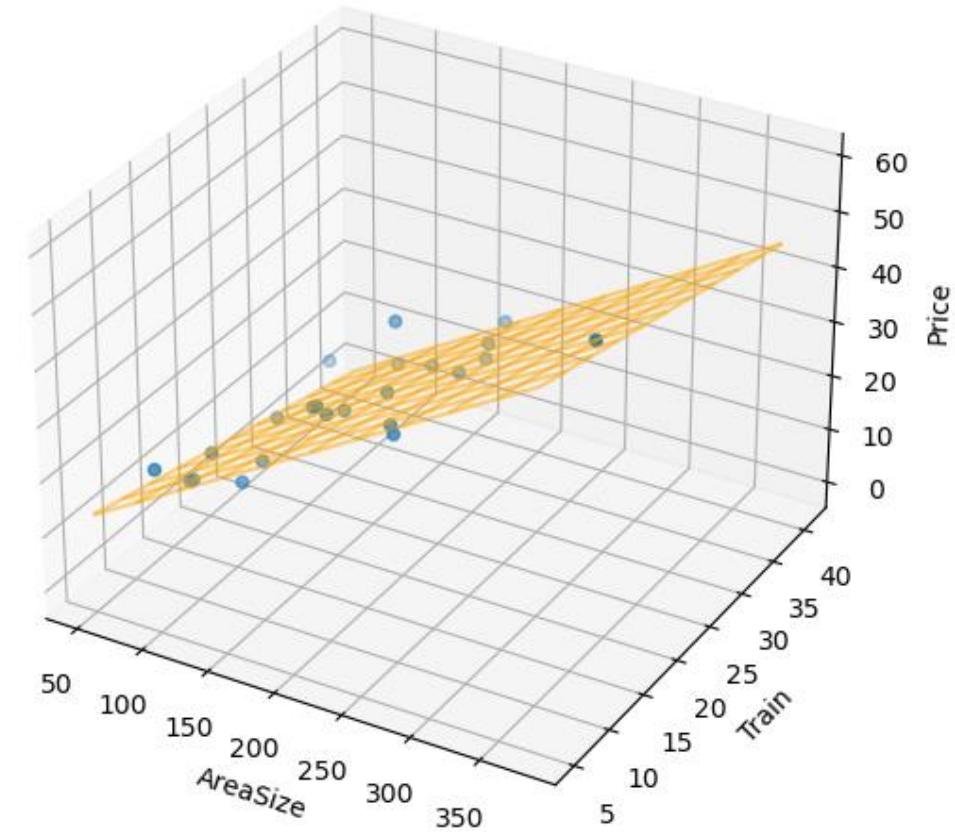
2-3_linear_regression_multi_psedo_data

■ 内容

- 説明変数の多い、架空の住宅価格データセットに線形回帰を適用
- 統計量や散布図行列、相関行列などを確認

■ 手順

- ライブラリの読み込み
- 疑似データの作成
- データの可視化
- モデルの学習
- モデルの性能を確認
- 予測結果の可視化



2-3 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import seaborn as sns
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D #3D散布図の描画

# warning を非表示にする
import warnings
warnings.simplefilter('ignore')
```

2-3 | 2. 疑似データの作成

```
df_house = pd.DataFrame({  
    "Price": [24.8, 59.5, 7, 7.5, 9.8, 13.5, 14.9, 27, 27, 28, 28.5, 23, 12.9, 18, 23.7, 29.8, 17.8, 5.5, 8.7, 10.3, 14.5, 17.6, 16.8],  
    "AreaSize": [98.4, 379.8, 58.6, 61.5, 99.6, 76.2, 115.7, 165.2, 215.2, 157.8, 212.9, 137.8, 87.2, 139.6, 172.6, 151.9, 179.5, 50, 105, 132, 174, 176, 168.7],  
    "HouseSize": [74.2, 163.7, 50.5, 58, 66.4, 66.2, 59.6, 98.6, 87.4, 116.9, 96.9, 82.8, 75.1, 77.9, 125, 85.6, 70.1, 48.7, 66.5, 51.9, 82.3, 86.1, 80.8],  
    "PassedYear": [4.8, 9.3, 13, 12.8, 14, 6, 14.7, 13.6, 13.3, 6.7, 3.1, 10.3, 11.6, 10.5, 3.8, 5.4, 4.5, 14.6, 13.7, 13, 10.3, 4.4, 12.8],  
    "Train": [5, 12, 16, 16, 16, 16, 16, 16, 16, 19, 23, 23, 23, 28, 32, 37, 37, 37, 37, 37, 41],  
    "Walk": [6, 12, 2, 1, 5, 1, 4, 2, 7, 6, 5, 20, 8, 3, 5, 4, 2, 3, 11, 6, 18, 10, 2]  
})  
df_house.index.name = "id"  
df_house.head()
```

	Price	AreaSize	HouseSize	PassedYear	Train	Walk
id						
0	24.8	98.4	74.2	4.8	5	6
1	59.5	379.8	163.7	9.3	12	12
2	7.0	58.6	50.5	13.0	16	2
3	7.5	61.5	58.0	12.8	16	1
4	9.8	99.6	66.4	14.0	16	5

今回使用する中古住宅のデータ

- Price : 値段(百万円)
- AreaSize : 土地面積(m²)
- HouseSize : 家面積(m²)
- PassedYear : 経過年数(年)
- Train : 電車での最寄り駅から主要駅までの所要時間(分)
- Walk : 徒歩での最寄り駅から家までの所要時間(分)

2-3 | 2. 疑似データの作成

```
# 基本的な統計量の確認  
df_house.describe()
```

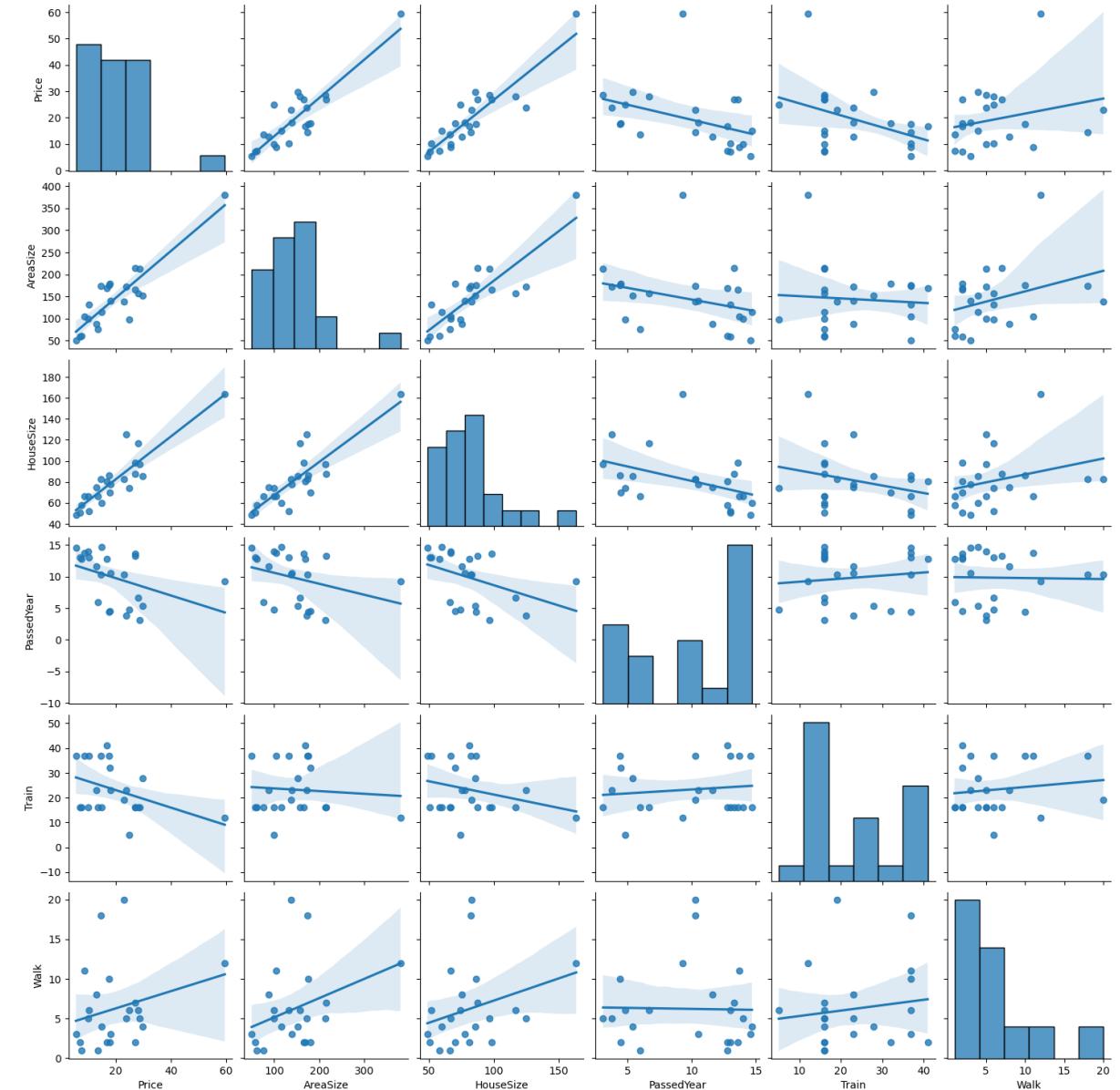
	Price	AreaSize	HouseSize	PassedYear	Train	Walk
count	23.000000	23.000000	23.000000	23.000000	23.000000	23.000000
mean	19.395652	144.139130	81.356522	9.834783	23.260870	6.217391
std	11.605151	70.086095	26.436955	4.023071	10.247915	5.062846
min	5.500000	50.000000	48.700000	3.100000	5.000000	1.000000
25%	11.600000	99.000000	66.300000	5.700000	16.000000	2.500000
50%	17.600000	139.600000	77.900000	10.500000	19.000000	5.000000
75%	25.900000	173.300000	86.750000	13.150000	34.500000	7.500000
max	59.500000	379.800000	163.700000	14.700000	41.000000	20.000000

今回使用する中古住宅のデータ

- Price : 値段(百万円)
- AreaSize : 土地面積(m²)
- HouseSize : 家面積(m²)
- PassedYear : 経過年数(年)
- Train : 電車での最寄り駅から主要駅までの所要時間(分)
- Walk : 徒歩での最寄り駅から家までの所要時間(分)

2-3 | 3. データの可視化

```
# 散布図行列を表示  
# kind="reg"を指定すると回帰直線も表示できる  
sns.pairplot(data=df_house, kind="reg")  
plt.show()
```



2-3 | 3. データの可視化

```
# 相関係数の行列を表示  
df_house.corr()
```

今回使用する中古住宅のデータ

- Price : 値段(百万円)
- AreaSize : 土地面積(m^2)
- HouseSize : 家面積(m^2)
- PassedYear : 経過年数(年)
- Train : 電車での最寄り駅から主要駅までの所要時間(分)
- Walk : 徒歩での最寄り駅から家までの所要時間(分)

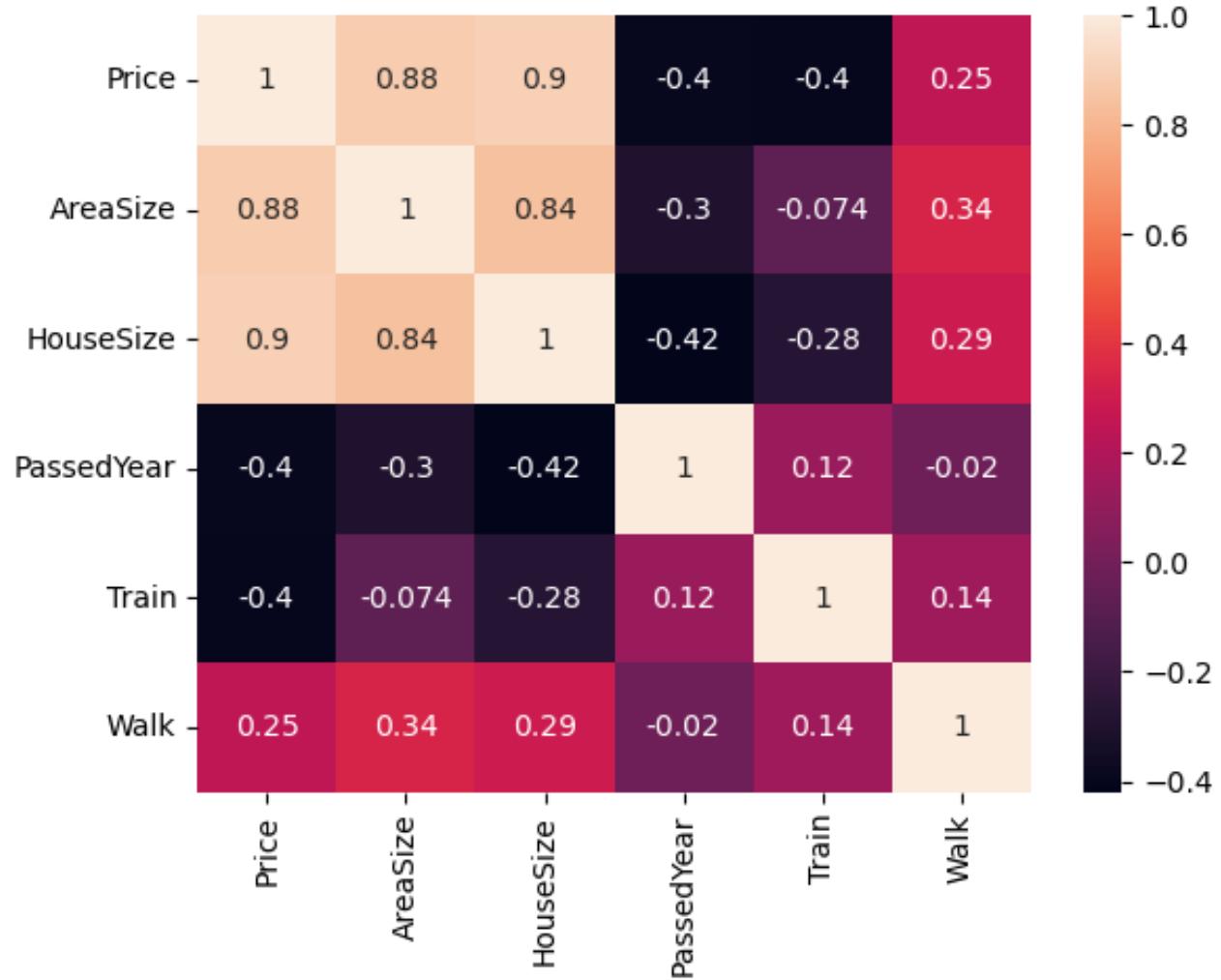
	Price	AreaSize	HouseSize	PassedYear	Train	Walk
Price	1.000000	0.878357	0.896135	-0.395842	-0.400421	0.248661
AreaSize	0.878357	1.000000	0.843471	-0.303278	-0.074319	0.336687
HouseSize	0.896135	0.843471	1.000000	-0.420226	-0.276636	0.291113
PassedYear	-0.395842	-0.303278	-0.420226	1.000000	0.124133	-0.020027
Train	-0.400421	-0.074319	-0.276636	0.124133	1.000000	0.138155
Walk	0.248661	0.336687	0.291113	-0.020027	0.138155	1.000000

2-3 | 3. データの可視化

```
# 相関行列をヒートマップに変換  
# annot=Trueをつけることでマスの中に値を表示  
sns.heatmap(data=df_house.corr(), annot=True)  
plt.show()
```

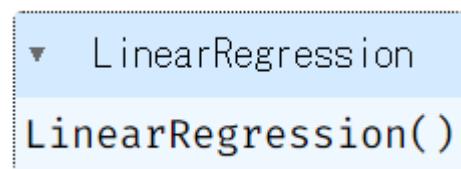
今回使用する中古住宅のデータ

- Price : 値段(百万円)
- AreaSize : 土地面積(m^2)
- HouseSize : 家面積(m^2)
- PassedYear : 経過年数(年)
- Train : 電車での最寄り駅から主要駅までの所要時間(分)
- Walk : 徒歩での最寄り駅から家までの所要時間(分)



```
# 正解データ
y = df_house["Price"].values
# 学習用データ
X = df_house[["AreaSize", "Train"]].values

# 線形回帰モデルの作成
regr = LinearRegression(fit_intercept=True)
# モデルを学習させる
regr.fit(X, y)
```



2-3 | 5. モデルの性能を確認

```
# モデルのパラメータを取り出す
w0 = regr.intercept_
w1 = regr.coef_[0]
w2 = regr.coef_[1]

# モデルから予測を得る
y_est_sklearn = regr.predict(X)

# 二乗誤差を計算
squared_error = mean_squared_error(y, y_est_sklearn)
print('w0 = {:.3f}, w1 = {:.3f}, w2 = {:.3f}, 二乗誤差 = {:.3f}'.format(w0, w1, w2, squared_error))
```

w0 = 7.907, w1 = 0.141, w2 = -0.382, 二乗誤差 = 14.885

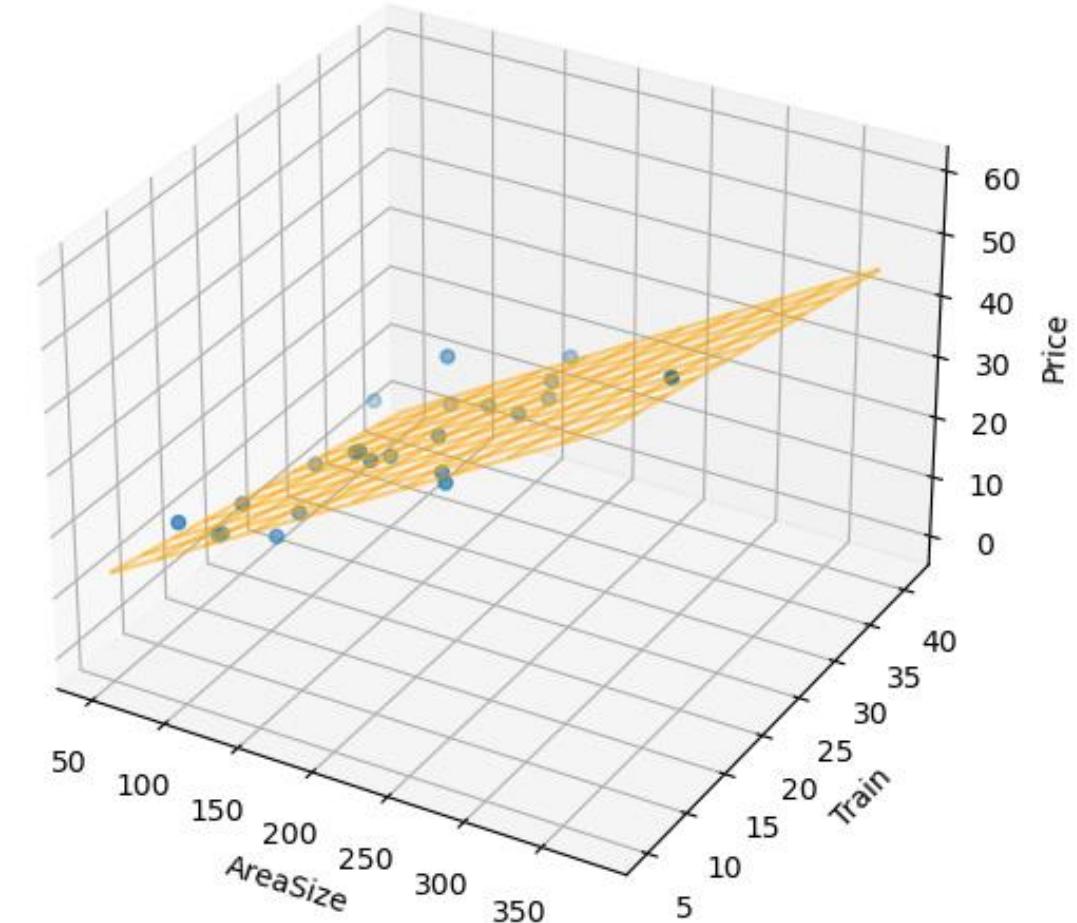
2-3 | 6. 予測結果の可視化

```
# 変数の設定
x1 = df_house["AreaSize"]
x2 = df_house["Train"]
y = df_house["Price"]

# 表示領域の設定
fig = plt.figure()
ax = Axes3D(fig)
# 軸の設定
ax.set_xlabel("AreaSize")
ax.set_ylabel("Train")
ax.set_zlabel("Price")

# 3D散布図の表示
ax.scatter3D(x1, x2, y)
# メッシュの作成
x1 = np.arange(min(x1), max(x1), (max(x1) - min(x1)) / 100)
x2 = np.arange(min(x2), max(x2), (max(x2) - min(x2)) / 100)
x1, x2 = np.meshgrid(x1, x2)
# メッシュ上の点に対する予測値を計算
y_est = w0 + w1 * x1 + w2 * x2

# メッシュをワイヤーフレームとして表示
# alphaを用いて透明度を調整
# rstrideとcstrideを用いて密度を調節
ax.plot_wireframe(x1, x2, y_est, color="orange", alpha=0.5, rstride=10, cstride=10)
plt.show()
```



2-4_linear_regression_multi_real_data

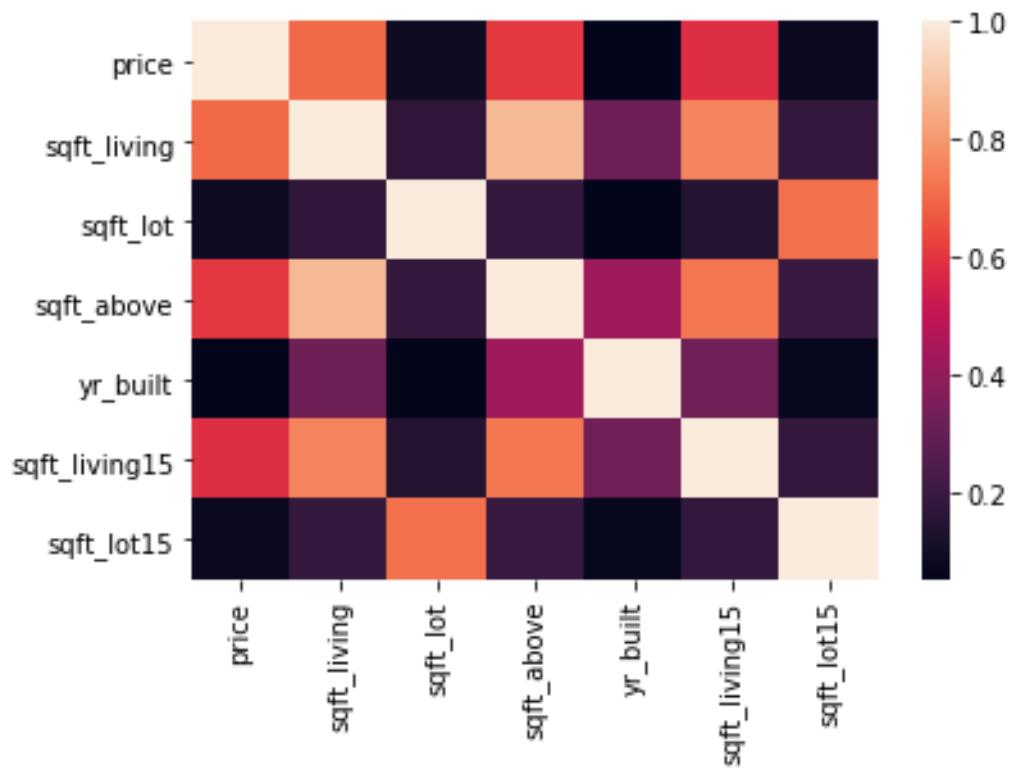
2-4_linear_regression_multi_real_data

■ 内容

- サイズの大きい住宅価格データセットに線形回帰を適用
- 統計量や散布図行列、相関係数などを確認
- 6つの説明変数を用いて学習

■ 手順

1. ライブラリの読み込み
2. データの読み込み
3. [演習] データの可視化
4. [演習] モデルの学習
5. [演習] モデルの性能を確認



2-4 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import seaborn as sns
import matplotlib.pyplot as plt
```

2-4 | 2. データの読み込み

```
df_house = pd.read_csv("../..\\1_data\\ch2\\kc_house_data.csv")[
    ['price', 'sqft_living', 'sqft_lot', 'sqft_above', 'yr_built', 'sqft_living15', 'sqft_lot15']
]

# 先程と似た中古住宅のデータ
df_house.head()
```

	価格	延床面積	敷地面積	地下室を除いた面積	建築年	2015 年の延床面積	2015 年の敷地面積
	price	sqft_living	sqft_lot	sqft_above	yr_built	sqft_living15	sqft_lot15
0	221900.0	1180	5650	1180	1955	1340	5650
1	538000.0	2570	7242	2170	1951	1690	7639
2	180000.0	770	10000	770	1933	2720	8062
3	604000.0	1960	5000	1050	1965	1360	5000
4	510000.0	1680	8080	1680	1987	1800	7503

出典：<https://www.kaggle.com/code/hnnhytc/data-analysis-with-python>

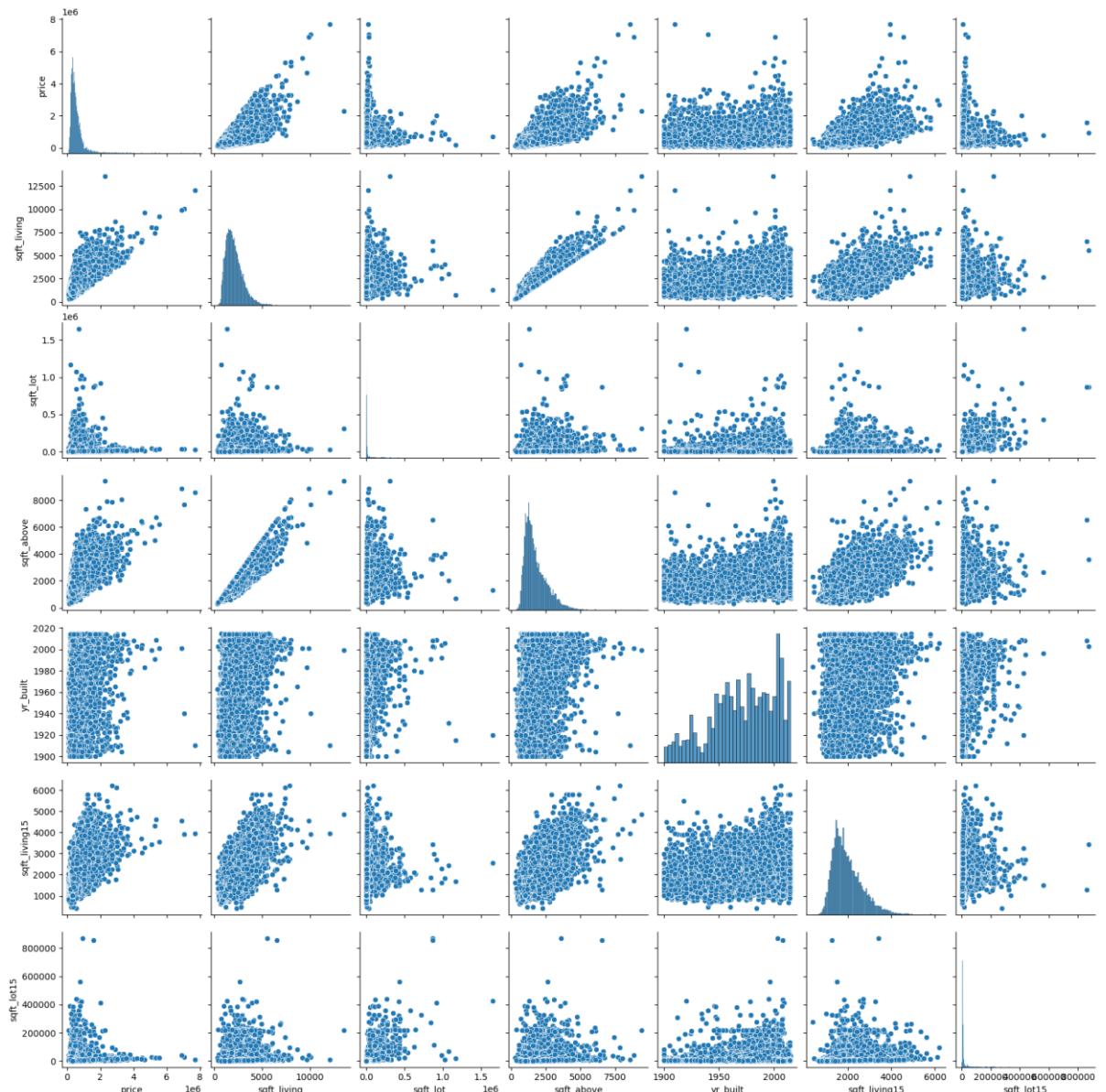
2-4 | 2. データの読み込み

```
df_house.describe()
```

	価格	延床面積	敷地面積	地下室を除いた面積	建築年	2015 年の延床面積	2015 年の敷地面積
	price	sqft_living	sqft_lot	sqft_above	yr_built	sqft_living15	sqft_lot15
count	2.161300e+04	21613.000000	2.161300e+04	21613.000000	21613.000000	21613.000000	21613.000000
mean	5.400881e+05	2079.899736	1.510697e+04	1788.390691	1971.005136	1986.552492	12768.455652
std	3.671272e+05	918.440897	4.142051e+04	828.090978	29.373411	685.391304	27304.179631
min	7.500000e+04	290.000000	5.200000e+02	290.000000	1900.000000	399.000000	651.000000
25%	3.219500e+05	1427.000000	5.040000e+03	1190.000000	1951.000000	1490.000000	5100.000000
50%	4.500000e+05	1910.000000	7.618000e+03	1560.000000	1975.000000	1840.000000	7620.000000
75%	6.450000e+05	2550.000000	1.068800e+04	2210.000000	1997.000000	2360.000000	10083.000000
max	7.700000e+06	13540.000000	1.651359e+06	9410.000000	2015.000000	6210.000000	871200.000000

2-4 | 3. [演習] データの可視化

```
# 散布図行列を表示  
# kind="reg"を指定すると回帰直線も表示できる  
# 項目数・データ数によっては時間がかかる  
sns.pairplot(data=df_house)  
plt.show()
```



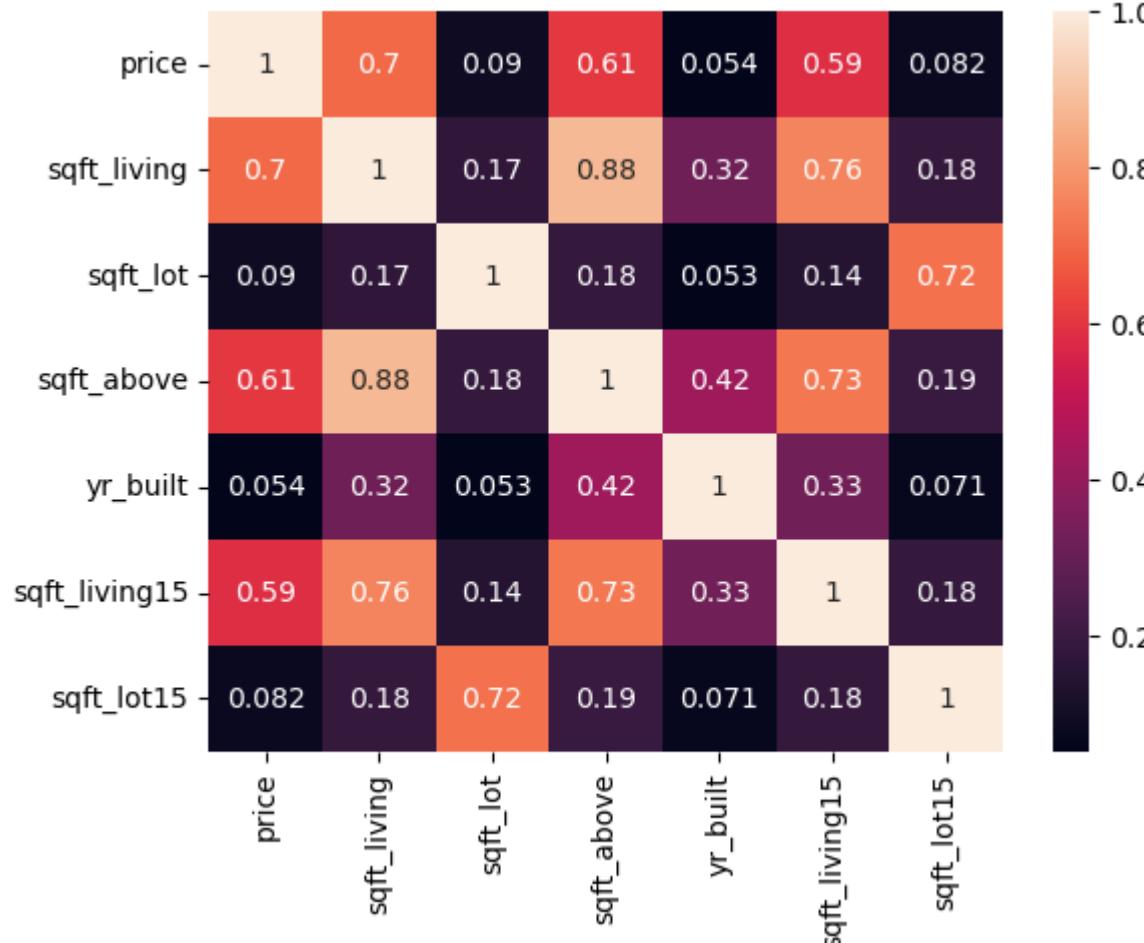
2-4 | 3. [演習] データの可視化

```
# 相関係数の行列を確認  
df_house.corr()
```

	price	sqft_living	sqft_lot	sqft_above	yr_built	sqft_living15	sqft_lot15
price	1.000000	0.702035	0.089661	0.605567	0.054012	0.585379	0.082447
sqft_living	0.702035	1.000000	0.172826	0.876597	0.318049	0.756420	0.183286
sqft_lot	0.089661	0.172826	1.000000	0.183512	0.053080	0.144608	0.718557
sqft_above	0.605567	0.876597	0.183512	1.000000	0.423898	0.731870	0.194050
yr_built	0.054012	0.318049	0.053080	0.423898	1.000000	0.326229	0.070958
sqft_living15	0.585379	0.756420	0.144608	0.731870	0.326229	1.000000	0.183192
sqft_lot15	0.082447	0.183286	0.718557	0.194050	0.070958	0.183192	1.000000

2-4 | 3. [演習] データの可視化

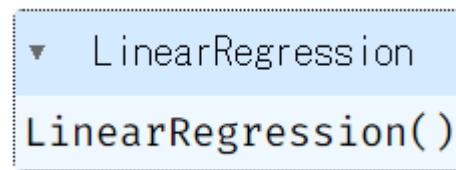
```
# 相関行列をヒートマップに変換  
sns.heatmap(df_house.corr(), annot=True)  
plt.show()
```



2-4 | 4. [演習] モデルの学習

```
# 正解データ
y = df_house["price"].values
# 学習用データ
X = df_house[['sqft_living', 'sqft_lot', 'sqft_above', 'yr_built', 'sqft_living15', 'sqft_lot15']].values

# 線形回帰モデルを作成
regr = LinearRegression(fit_intercept=True)
# モデルを学習させる
regr.fit(X, y)
```



2-4 | 5. [演習] モデルの性能を確認

```
# モデルのパラメータを取り出す
w0 = regr.intercept_
w1 = regr.coef_[0]
w2 = regr.coef_[1]
w3 = regr.coef_[2]
w4 = regr.coef_[3]
w5 = regr.coef_[4]
w6 = regr.coef_[5]

# モデルから予測を得る
y_est_sklearn = regr.predict(X)

# 二乗誤差を計算
squared_error = mean_squared_error(y, y_est_sklearn)
print('w0 = {:.3f}, w1 = {:.3f}, w2 = {:.3f}, w3 = {:.3f}, w4 = {:.3f}, w5 = {:.3f}, w6 = {:.3f}'.format(w0, w1, w2, w3, w4, w5, w6))
print('二乗誤差 = {:.3f}'.format(squared_error))
```

w0 = 5028516.266, w1 = 244.020, w2 = 0.027, w3 = 21.221, w4 = -2639.639, w5 = 89.670, w6 = -0.761

二乗誤差 = 62030861147.038

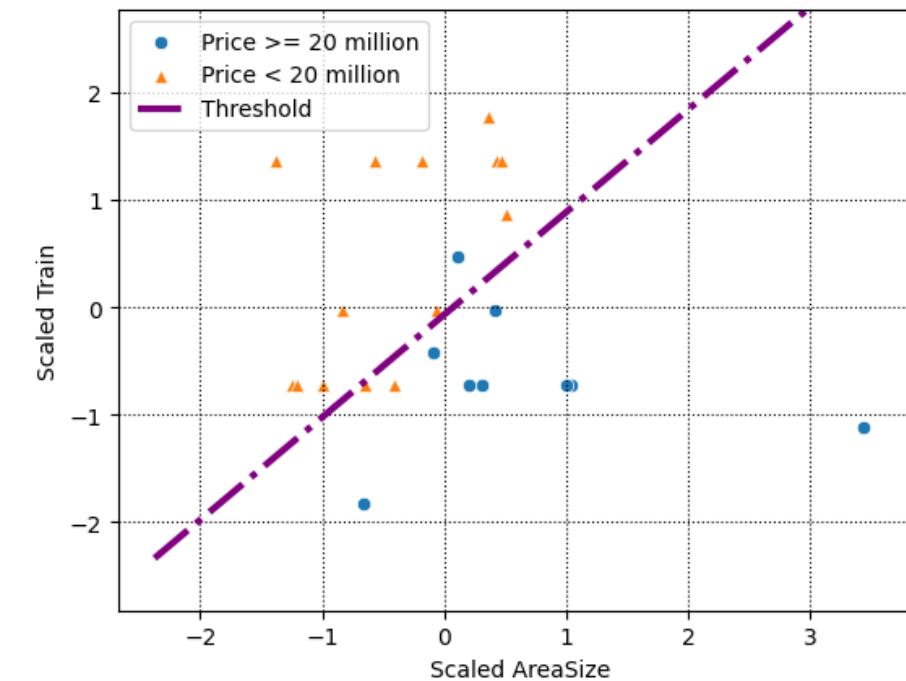
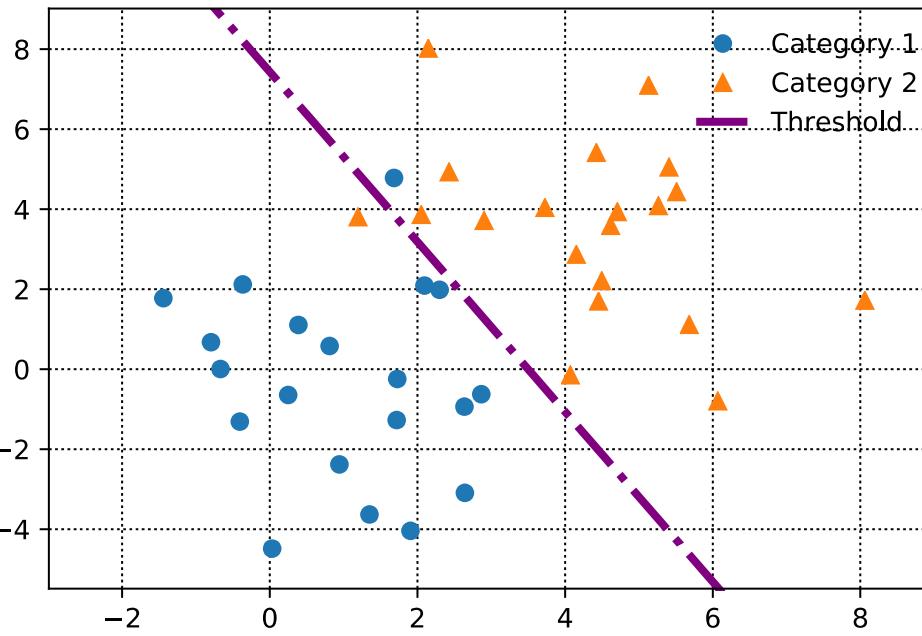
ロジスティック回帰モデル

2-5_logistic_regression_psedo_data

2-5_logistic_regression_psedo_data

■ 内容

- 2種類の疑似データを用いて、ロジスティック回帰モデルを構築
- 単純な2値分類の場合 / 連続変数をカテゴリ変数に変換する場合



■ 手順

1. ライブラリの読み込み
2. 疑似データの分類（その①）
 1. 疑似データの作成
 2. データの可視化
 3. モデルの学習
 4. 分類境界の表示
 5. モデルの性能を確認
3. 疑似データの分類（その②）
 1. 疑似データの作成
 2. データの前処理
 3. モデルの学習
 4. 分類境界の表示
 5. モデルの性能を確認

2-5 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# 勾配降下法（SGD）を使う分類モデル（ロジスティック回帰など）
from sklearn.linear_model import SGDClassifier
# 分類モデルの評価指標
from sklearn.metrics import log_loss, accuracy_score, confusion_matrix
# 標準化を行うためのクラス
from sklearn.preprocessing import StandardScaler
```

2-5 | 2-1. 疑似データの作成

- `np.random.multivariate_normal(mean, cov, size)`:多変量正規分布に従う乱数を生成
 - mean: 分布の平均
 - cov: 分散共分散行列
 - size: サンプルサイズ

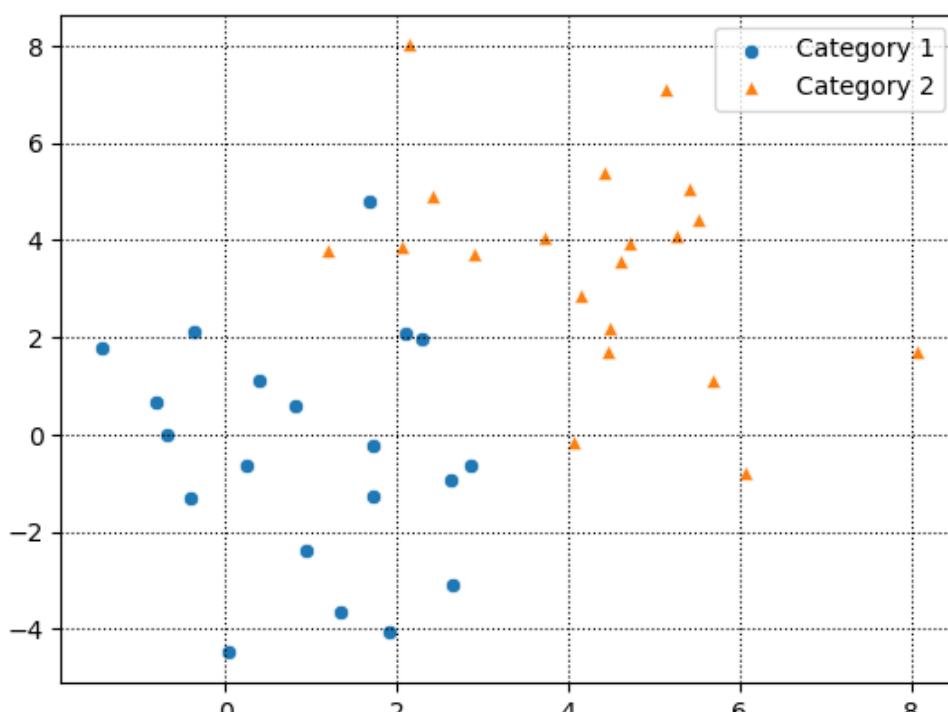
```
# 亂数シードの固定
np.random.seed(1234)

# 1つ目のカテゴリに属する2次元データを20個生成
# 両次元とも平均0、分散4
data1 = np.random.multivariate_normal((0, 0), [[4, 0], [0, 4]], 20)
label1 = np.zeros(len(data1)) # ラベルは0とする

# 2つ目のカテゴリに属する2次元データを20個生成
# 両次元とも平均4、分散4
data2 = np.random.multivariate_normal((4, 4), [[4, 0], [0, 4]], 20)
label2 = np.ones(len(data2)) # ラベルは1とする
```

2-5 | 2-2. データの可視化

```
# グリッドの表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')
# 散布図の表示
sns.scatterplot(x=data1[:, 0], y=data1[:, 1], marker='o', color='C0', label='Category 1')
sns.scatterplot(x=data2[:, 0], y=data2[:, 1], marker='^', color='C1', label='Category 2')
# 凡例の位置を調整
plt.legend(loc='best')
plt.show()
```



2-5 | 2-3. モデルの学習

```
# scikit-learnに渡すため、2つのデータを結合する
X = np.concatenate([data1, data2])
y = np.concatenate([label1, label2])
# ロジスティック回帰モデルをインスタンス化
clf = SGDClassifier(loss='log_loss', max_iter=10000,
                     fit_intercept=True, random_state=1234, tol=1e-3)
# モデルを学習させる
clf.fit(X, y)

# 重みを取得して表示
w0 = clf.intercept_[0]
w1 = clf.coef_[0, 0]
w2 = clf.coef_[0, 1]
print("w0 = {:.3f}, w1 = {:.3f}, w2 = {:.3f}".format(w0, w1, w2))
```

- SGDClassifier: ロジスティック回帰など、勾配降下法を用いる分類モデルを実装
 - loss: 使用する損失関数を指定
 - penalty: 正則化の種類を指定
 - max_iter: 最大のイテレーション回数（エポック数）を指定
 - fit_intercept: 切片を計算するかどうかを指定
 - random_state: 再現性を確保するための乱数シードを指定
 - tol: 収束判定のための許容誤差を指定

w0 = -183.316, w1 = 57.480, w2 = 22.085

2-5 | 2-4. 分類境界の表示

```
# グリッドの表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')

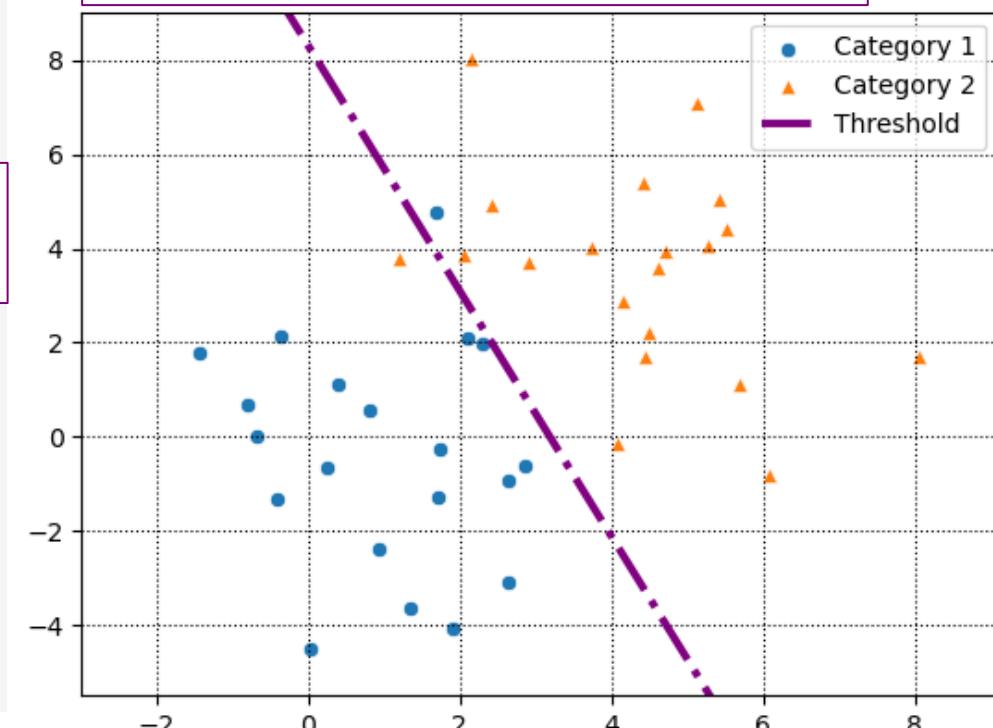
# データの散布図を表示
sns.scatterplot(x=data1[:, 0], y=data1[:, 1], marker='o', color='C0', label='Category 1')
sns.scatterplot(x=data2[:, 0], y=data2[:, 1], marker='^', color='C1', label='Category 2')

# 説明変数をx1とx2に分離
x1, x2 = X[:, 0], X[:, 1]
# 分類境界上にある点のx1座標（横軸の値）を計算
line_x = np.arange(np.min(x1) - 1, np.max(x1) + 1)
# 分類境界上にある点のx2座標（縦軸の値）を計算
line_y = - line_x * w1 / w2 - w0 / w2
# 分類境界を直線として表示（紫色の点線）
sns.lineplot(x=line_x, y=line_y, linestyle='-.',
             linewidth=3, color='purple', label='Threshold')

# グラフの表示範囲を調整
plt.ylim([np.min(x2) - 1, np.max(x2) + 1])
# 凡例の表示位置を調整
plt.legend(loc='best')
plt.show()
```

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{w_0}{w_2}$$

分類境界 $w_0 + w_1x_1 + w_2x_2 = 0$



2-5 | 2-5. モデルの性能を確認（予測値）

```
# ラベルを予測
y_est = clf.predict(X)
print('---予測ラベル---')
print(y_est) # (データ数, ) の1次元配列

# 確率値を得る
y_est_proba = clf.predict_proba(X)
print('---確率値---')
print(y_est_proba) # (データ数, クラス数)の二次元配列.
# 1列目はラベル0である確率, 2列目はラベル1である確率を意味する
```

2-5 | 2-5. モデルの性能を確認（対数尤度）

```
# 対数尤度は-∞から0の値を返す (0に近いほど当てはまりがよい)
# log_lossは「負の」対数尤度を計算する関数であるため、- をつけて戻している
# normalizeという引数をTrueのままにしておくと、合計の対数尤度ではなく平均の対数尤度が計算されるので注意
# 対数尤度の計算には確率値を使う
print('対数尤度 = {:.3f}'.format(-log_loss(y, y_est_proba, normalize=False)))

# 自分で対数尤度を計算
# epsilonの使い方が異なるので、微妙に結果が変わる
l = 0
epsilon = 1e-15 # アンダーフローを防ぐための定数
for i in range(len(y)):
    l += y[i] * np.log(y_est_proba[i,1] + epsilon) + (1 - y[i]) * np.log(y_est_proba[i,0] + epsilon)
print('対数尤度 = {:.3f}'.format(l))
```

対数尤度 = -49.737
対数尤度 = -49.716

$$\ln P(\mathbf{w}) = \sum_{n=1}^N \left\{ y^{(n)} \ln \hat{y}^{(n)} + (1 - y^{(n)}) \ln (1 - \hat{y}^{(n)}) \right\}$$

2-5 | 2-5. モデルの性能を確認（正解率・混同行列）

```
# 正解率(Accuracy)を表示
print('正解率 = {}%'.format(100 * accuracy_score(y, y_est)))

# 混同行列を表示
conf_mat = pd.DataFrame(
    confusion_matrix(y, y_est),
    index=['正解 = 0', '正解 = 1'],
    columns=['予測値 = 0', '予測値 = 1'])
conf_mat
```

正解率 = 95.0%

	予測値 = 0	予測値 = 1
正解 = 0	19	1
正解 = 1	1	19

2-5 | 3-1. 疑似データの作成

```
df_house = pd.DataFrame({  
    "Price": [24.8, 59.5, 7, 7.5, 9.8, 13.5, 14.9, 27, 27, 28, 28.5, 23, 12.9, 18, 23.7, 29.8, 17.8, 5.5, 8.7, 10.3, 14.5, 17.6, 16.8],  
    "AreaSize": [98.4, 379.8, 58.6, 61.5, 99.6, 76.2, 115.7, 165.2, 215.2, 157.8, 212.9, 137.8, 87.2, 139.6, 172.6, 151.9, 179.5, 50, 105, 132, 174, 176, 168.7],  
    "HouseSize": [74.2, 163.7, 50.5, 58, 66.4, 66.2, 59.6, 98.6, 87.4, 116.9, 96.9, 82.8, 75.1, 77.9, 125, 85.6, 70.1, 48.7, 66.5, 51.9, 82.3, 86.1, 80.8],  
    "PassedYear": [4.8, 9.3, 13, 12.8, 14, 6, 14.7, 13.6, 13.3, 6.7, 3.1, 10.3, 11.6, 10.5, 3.8, 5.4, 4.5, 14.6, 13.7, 13, 10.3, 4.4, 12.8],  
    "Train": [5, 12, 16, 16, 16, 16, 16, 16, 16, 19, 23, 23, 23, 28, 32, 37, 37, 37, 37, 37, 41],  
    "Walk": [6, 12, 2, 1, 5, 1, 4, 2, 7, 6, 5, 20, 8, 3, 5, 4, 2, 3, 11, 6, 18, 10, 2]  
})  
df_house.index.name="id"  
  
# 目的変数をカテゴリに変換  
# Priceの値を2000万以上なら1, そうでなければ0に変更  
df_house['Price'] = df_house['Price'] >= 20  
df_house.head()
```

	Price	AreaSize	HouseSize	PassedYear	Train	Walk
id						
0	True	98.4	74.2	4.8	5	6
1	True	379.8	163.7	9.3	12	12
2	False	58.6	50.5	13.0	16	2
3	False	61.5	58.0	12.8	16	1
4	False	99.6	66.4	14.0	16	5

- 中古住宅の疑似データを再利用する
 - Price : 値段(百万円)
 - AreaSize : 土地面積(m²)
 - HouseSize : 家面積(m²)
 - PassedYear : 経過年数(年)
 - Train : 電車での最寄り駅から主要駅までの所要時間(分)
 - Walk : 徒歩での最寄り駅から家までの所要時間(分)

2-5 | 3-1. 疑似データの作成

```
df_house.describe()
```

	AreaSize	HouseSize	PassedYear	Train	Walk
count	23.000000	23.000000	23.000000	23.000000	23.000000
mean	144.139130	81.356522	9.834783	23.260870	6.217391
std	70.086095	26.436955	4.023071	10.247915	5.062846
min	50.000000	48.700000	3.100000	5.000000	1.000000
25%	99.000000	66.300000	5.700000	16.000000	2.500000
50%	139.600000	77.900000	10.500000	19.000000	5.000000
75%	173.300000	86.750000	13.150000	34.500000	7.500000
max	379.800000	163.700000	14.700000	41.000000	20.000000

- 中古住宅の疑似データを再利用する
 - Price : 値段(百万円)
 - AreaSize : 土地面積(m^2)
 - HouseSize : 家面積(m^2)
 - PassedYear : 経過年数(年)
 - Train : 電車での最寄り駅から主要駅までの所要時間(分)
 - Walk : 徒歩での最寄り駅から家までの所要時間(分)

2-5 | 3-2. データの前処理

```
# 正解データ作成
y = df_house["Price"].values
# 学習用データ作成（説明変数を2つに絞る）
X = df_house[["AreaSize", "Train"]].values

# 各特徴量を平均0, 分散1となるように変換（標準化）するためのクラス
# 今回のデータセットは特徴量の平均や分散がそれぞれ大きく異なるため、
# そのままでは確率値がうまく算出されない
std_scaler = StandardScaler()

# 標準化した特徴量X_scaledを得る
# 以降はX_scaledをモデルの学習等に使う
X_scaled = std_scaler.fit_transform(X)
```

2-5 | 3-3. モデルの学習

```
# ロジスティック回帰モデルをインスタンス化
clf = SGDClassifier(loss='log_loss', max_iter=10000,
                     fit_intercept=True, random_state=1234, tol=1e-3)
# モデルを学習させる
clf.fit(X_scaled, y)

# 重みを取得して表示
w0 = clf.intercept_[0]
w1 = clf.coef_[0, 0]
w2 = clf.coef_[0, 1]
print("w0 = {:.3f}, w1 = {:.3f}, w2 = {:.3f}".format(w0, w1, w2))
```

```
w0 = -1.688, w1 = 23.292, w2 = -24.392
```

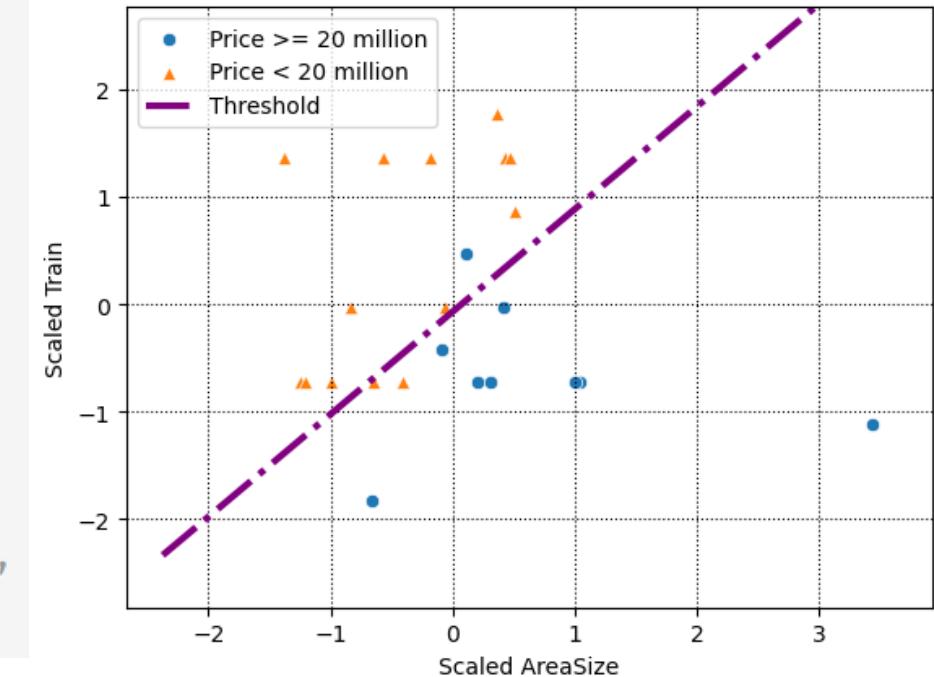
2-5 | 3-4. 分類境界の表示

```
# 説明変数をx1とx2に分離
x1 = X_scaled[:,0] # 標準化されたAreaSize
x2 = X_scaled[:,1] # 標準化されたTrain

# グリッドの表示
plt.grid(which='major',color='black',linestyle=':')
plt.grid(which='minor',color='black',linestyle=':')

# 正解ラベル（配列y）の値が1であるデータの散布図
sns.scatterplot(x=x1[y], y=x2[y], marker='o', color='C0',
                  label='Price >= 20 million')

# 正解ラベル（配列y）の値が0であるデータの散布図
sns.scatterplot(x=x1[~y], y=x2[~y], marker='^', color='C1',
                  label='Price < 20 million')
```



2-5 | 3-4. 分類境界の表示

```
# 分類境界上にある点のx1座標（横軸の値）を計算
```

```
line_x = np.arange(np.min(x1) - 1, np.max(x1) + 1)
```

```
# 分類境界上にある点のx2座標（縦軸の値）を計算
```

```
line_y = - line_x * w1 / w2 - w0 / w2
```

```
# 分類境界を直線として表示（紫色の点線）
```

```
plt.plot(line_x, line_y, linestyle='-.',  
         linewidth=3, color='purple', label='Threshold')
```

```
# グラフの表示範囲を調整
```

```
plt.ylim([np.min(x2) - 1, np.max(x2) + 1])
```

```
# 凡例の表示位置を調整
```

```
plt.legend(loc='best')
```

```
# 軸ラベルの設定
```

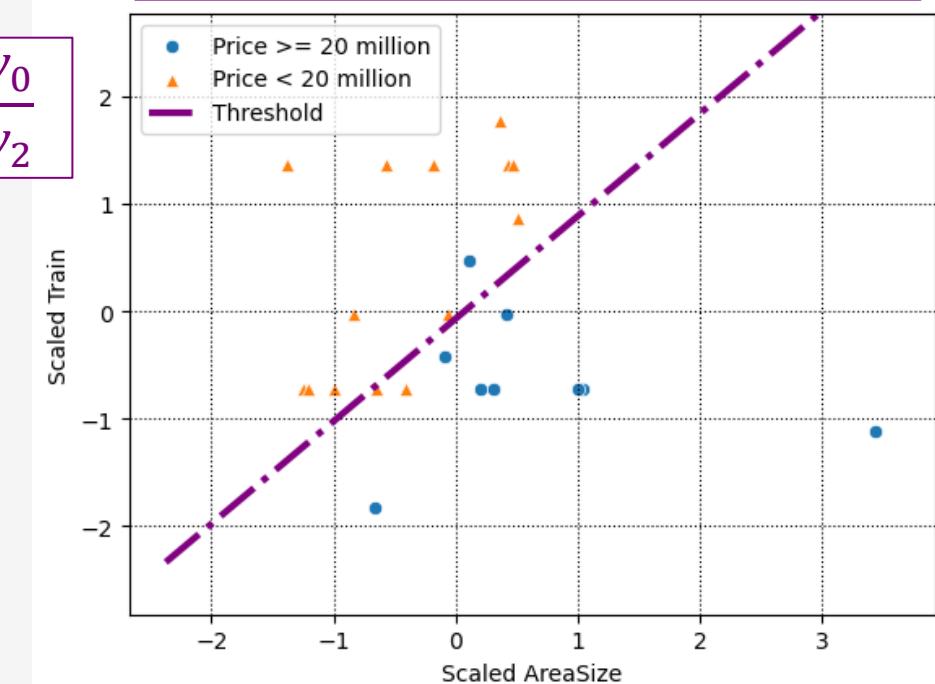
```
plt.xlabel("Scaled AreaSize")
```

```
plt.ylabel("Scaled Train")
```

```
plt.show()
```

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{w_0}{w_2}$$

分類境界 $w_0 + w_1x_1 + w_2x_2 = 0$



2-5 | 3-5. モデルの性能を確認（予測値）

```
# ラベルを予測  
y_est = clf.predict(X_scaled)  
  
# 確率値を得る  
y_est_proba = clf.predict_proba(X)
```

2-5 | 3-5. モデルの性能を確認（対数尤度）

```
# log_lossは「負の」対数尤度を計算する関数であるため、- をつけて戻している
# normalizeという引数をTrueのままにしておくと、合計の対数尤度ではなく平均の対数尤度が計算されるので注意
print('対数尤度 = {:.3f}'.format(- log_loss(y, y_est_proba, normalize=False)))

# 自分で対数尤度を計算
# epsilonの使い方が異なるので、微妙に結果が変わる
l = 0
epsilon = 1e-15 # アンダーフローを防ぐための定数
for i in range(len(y)):
    l += y[i] * np.log(y_est_proba[i,1] + epsilon) + (1 - y[i]) * np.log(y_est_proba[i,0] + epsilon)
print('対数尤度 = {:.3f}'.format(l))
```

対数尤度 = -504.611

対数尤度 = -483.543

$$\ln P(\mathbf{w}) = \sum_{n=1}^N \left\{ y^{(n)} \ln \hat{y}^{(n)} + (1 - y^{(n)}) \ln (1 - \hat{y}^{(n)}) \right\}$$

2-5 | 3-5. モデルの性能を確認（正解率・混同行列）

```
# 正解率を表示
print('正解率 = {:.3f}%'.format(100 * accuracy_score(y, y_est)))

# 混同行列の表示
conf_mat = pd.DataFrame(confusion_matrix(y, y_est),
                         index=['正解 = 2000万未満', '正解 = 2000万以上'],
                         columns=['予測 = 2000万未満', '予測 = 2000万以上'])
conf_mat
```

正解率 = 86.957%

	予測 = 2000万未満	予測 = 2000万以上
正解 = 2000万未満	12	2
正解 = 2000万以上	1	8

2-6_logistic_regression_real_data

2-6_logistic_regression_real_data

■ 内容

- 説明変数の多いデータセットにロジスティック回帰を適用
- 住宅がリノベーション（改装）されているかどうかを予測

■ 手順

- ライブラリの読み込み
- データの読み込み
- データの前処理
- [演習] モデルの学習
- [演習] モデルの性能確認

	yr_renovated	sqft_living	sqft_lot	sqft_above	yr_built	sqft_living15	sqft_lot15
0	False	1180	5650	1180	1955	1340	5650
1	True	2570	7242	2170	1951	1690	7639
2	False	770	10000	770	1933	2720	8062
3	False	1960	5000	1050	1965	1360	5000
4	False	1680	8080	1680	1987	1800	7503

	予測 = リノベーションなし	予測 = リノベーション済み
正解 = リノベーションなし	20624	75
正解 = リノベーション済み	882	32

2-6 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# 勾配降下法（SGD）を使う分類モデル（ロジスティック回帰など）
from sklearn.linear_model import SGDClassifier
# 分類モデルの評価指標
from sklearn.metrics import log_loss, accuracy_score, confusion_matrix
# 標準化を行うためのクラス
from sklearn.preprocessing import StandardScaler
```

2-6 | 2. データの読み込み

```
# CSVファイルの読み込み
df_house = pd.read_csv("../..\\1_data\\ch2\\kc_house_data.csv")[
    ['yr_renovated', 'sqft_living', 'sqft_lot', 'sqft_above', 'yr_built', 'sqft_living15', 'sqft_lot15']
]
# yr_renovatedを「改装済み」を表すカテゴリ変数に変換
df_house['yr_renovated'] = df_house['yr_renovated'] > 0

# 先程と似た中古住宅のデータ
df_house.head()
```

	yr_renovated	sqft_living	sqft_lot	sqft_above	yr_built	sqft_living15	sqft_lot15
0	False	1180	5650	1180	1955	1340	5650
1	True	2570	7242	2170	1951	1690	7639
2	False	770	10000	770	1933	2720	8062
3	False	1960	5000	1050	1965	1360	5000
4	False	1680	8080	1680	1987	1800	7503

2-6 | 2. データの読み込み

```
df_house.describe()
```

	sqft_living	sqft_lot	sqft_above	yr_built	sqft_living15	sqft_lot15
count	21613.000000	2.161300e+04	21613.000000	21613.000000	21613.000000	21613.000000
mean	2079.899736	1.510697e+04	1788.390691	1971.005136	1986.552492	12768.455652
std	918.440897	4.142051e+04	828.090978	29.373411	685.391304	27304.179631
min	290.000000	5.200000e+02	290.000000	1900.000000	399.000000	651.000000
25%	1427.000000	5.040000e+03	1190.000000	1951.000000	1490.000000	5100.000000
50%	1910.000000	7.618000e+03	1560.000000	1975.000000	1840.000000	7620.000000
75%	2550.000000	1.068800e+04	2210.000000	1997.000000	2360.000000	10083.000000
max	13540.000000	1.651359e+06	9410.000000	2015.000000	6210.000000	871200.000000

```
# 正解データ作成
y = df_house["yr_renovated"].values
# 学習用データ作成
X = df_house.drop('yr_renovated', axis=1).values

# 各特徴量を平均0, 分散1となるように変換（標準化）するためのクラス
# 今回のデータセットは特徴量の平均や分散がそれぞれで大きく異なるため、
# そのままでは確率値がうまく算出されない
std_scaler = StandardScaler()

# 標準化した特徴量X_scaledを得る
# 以降はX_scaledをモデルの学習等に使う
X_scaled = std_scaler.fit_transform(X)
```

2-6 | 4. [演習] モデルの実装

```
##### 以降は自分で実装しよう #####
# ロジスティック回帰モデルをインスタンス化
clf = SGDClassifier(loss='log_loss', max_iter=10000, fit_intercept=True, random_state=1234, tol=1e-3, )
# モデルを学習させる
clf.fit(X_scaled, y)

# 重みを取得して表示
w0 = clf.intercept_[0]
w1 = clf.coef_[0, 0]
w2 = clf.coef_[0, 1]
w3 = clf.coef_[0, 2]
w4 = clf.coef_[0, 3]
w5 = clf.coef_[0, 4]
w6 = clf.coef_[0, 5]
print('w0 = {:.3f}, w1 = {:.3f}, w2 = {:.3f}, w3 = {:.3f}, w4 = {:.3f}, w5 = {:.3f}, w6 = {:.3f}'.format(w0, w1, w2, w3, w4, w5, w6))
```

```
w0 = -3.817, w1 = 0.432, w2 = -0.041, w3 = 0.101, w4 = -1.181, w5 = 0.009, w6 = 0.012
```

2-6 | 5. [演習] モデルの性能を確認（予測値）

```
# ラベルを予測  
y_est = clf.predict(X_scaled)  
  
# 確率値を得る  
y_est_proba = clf.predict_proba(X_scaled)
```

2-6 | 5. [演習] モデルの性能を確認（対数尤度）

```
# log_lossという関数は、負の対数尤度を返す
# normalizeという引数をTrueにすると、合計の対数尤度ではなく平均の対数尤度が計算される
print('対数尤度 = {:.3f}'.format(- log_loss(y, y_est_proba, normalize=False)))

# 自分で対数尤度を計算
# ここでは、epsilonがあっても結果は同じ
l = 0
epsilon = 1e-15 # アンダーフローを防ぐための定数
for i in range(len(y)):
    l += y[i] * np.log(y_est_proba[i,1] + epsilon) + (1 - y[i]) * np.log(y_est_proba[i,0] + epsilon)
print('対数尤度 = {:.3f}'.format(l))
```

対数尤度 = -3125.727

対数尤度 = -3125.727

2-6 | 5. [演習] モデルの性能を確認（正解率・混同行列）

```
# 正解率を表示
print('正解率 = {:.3f}%'.format(100 * accuracy_score(y, y_est)))

# 混同行列の表示
conf_mat = pd.DataFrame(confusion_matrix(y, y_est),
                         index=['正解 = リノベーションなし', '正解 = リノベーション済み'],
                         columns=['予測 = リノベーションなし', '予測 = リノベーション済み'])
conf_mat
```

正解率 = 95.725%

	予測 = リノベーションなし	予測 = リノベーション済み
正解 = リノベーションなし	20681	18
正解 = リノベーション済み	906	8

現場で使える 機械学習・データ分析基礎講座

第3章：モデルの評価指標

ノートブック解説

■ モデルの評価指標

- 3-1_model_evaluation.ipynb
- 3-2_model_evaluation_roc_curve.ipynb
- 3-3_micro_macro_average.ipynb

3-1_model_evaluation

■ 内容

- 回帰問題、分類問題それぞれにおける評価指標を計算
- 住宅価格の疑似データを用いる

■ 手順

- ライブラリの読み込み
- 回帰問題の評価指標
 - 疑似データの作成
 - モデルの構築・学習
 - モデルの評価
- 分類問題の評価指標
 - 疑似データの作成
 - モデルの構築・学習
 - モデルの評価

```
MAE = 3.057
MSE = 14.885
RMSE = 3.858
MAPE = 20.8 %
r2 = 0.884
```

```
正解率 (Accuracy) = 91.304%
適合率 (Precision) = 88.889%
再現率 (Recall) = 88.889%
F1値 (F1-score) = 88.889%
```

3-1 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np

# モデルを表すクラス
from sklearn.linear_model import LinearRegression, SGDClassifier

# 回帰問題における評価指標を計算する関数
from sklearn.metrics import mean_squared_error, mean_absolute_error, mean_absolute_percentage_error, r2_score

# 分類問題における評価指標を計算する関数
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confusion_matrix
```

■ 回帰問題で用いる指標は、主に以下の3つ

- MSE : 平均二乗誤差
- RMSE : 平方根平均二乗誤差
- MAE : 平均絶対値誤差

■ 他にも以下の指標がある

- MAPE : 平均絶対パーセント誤差
- R^2 : 決定係数

3-1 | 2-1. 疑似データの作成

```
# 疑似データの作成
df_house = pd.DataFrame({
    "Price": [24.8, 59.5, 7, 7.5, 9.8, 13.5, 14.9, 27, 27, 28, 28.5, 23, 12.9, 18, 23.7, 29.8, 17.8, 5.5, 8.7, 10.3, 14.5, 17.6, 16.8],
    "AreaSize": [98.4, 379.8, 58.6, 61.5, 99.6, 76.2, 115.7, 165.2, 215.2, 157.8, 212.9, 137.8, 87.2, 139.6, 172.6, 151.9, 179.5, 50, 105, 132, 174, 176, 168.7],
    "HouseSize": [74.2, 163.7, 50.5, 58, 66.4, 66.2, 59.6, 98.6, 87.4, 116.9, 96.9, 82.8, 75.1, 77.9, 125, 85.6, 70.1, 48.7, 66.5, 51.9, 82.3, 86.1, 80.8],
    "PassedYear": [4.8, 9.3, 13, 12.8, 14, 6, 14.7, 13.6, 13.3, 6.7, 3.1, 10.3, 11.6, 10.5, 3.8, 5.4, 4.5, 14.6, 13.7, 13, 10.3, 4.4, 12.8],
    "Train": [5, 12, 16, 16, 16, 16, 16, 16, 16, 19, 23, 23, 23, 28, 32, 37, 37, 37, 37, 37, 41],
    "Walk": [6, 12, 2, 1, 5, 1, 4, 2, 7, 6, 5, 20, 8, 3, 5, 4, 2, 3, 11, 6, 18, 10, 2]
})
df_house.index.name="id"
df_house.head()
```

	Price	AreaSize	HouseSize	PassedYear	Train	Walk
id						
0	24.8	98.4	74.2	4.8	5	6
1	59.5	379.8	163.7	9.3	12	12
2	7.0	58.6	50.5	13.0	16	2
3	7.5	61.5	58.0	12.8	16	1
4	9.8	99.6	66.4	14.0	16	5

- Price : 値段(百万円)
- AreaSize : 土地面積(m^2)
- HouseSize : 家面積(m^2)
- PassedYear : 経過年数(年)
- Train : 電車での最寄り駅から主要駅までの所要時間(分)
- Walk : 徒歩での最寄り駅から家までの所要時間(分)

```
# 目的変数と説明変数を分割
y = df_house["Price"].values
X = df_house[["AreaSize", "Train"]].values

# 線形回帰モデルの構築
regr = LinearRegression(fit_intercept=True)
# モデルの学習
regr.fit(X, y)
```

▼ LinearRegression
LinearRegression()

```
# 値を予測
y_pred = regr.predict(X)

# MAEを計算
mae = mean_absolute_error(y, y_pred)
print("MAE = %s"%round(mae,3) )

# MSEを計算
mse = mean_squared_error(y, y_pred)
print("MSE = %s"%round(mse,3) )

# RMSEを計算
rmse = np.sqrt(mse)
print("RMSE = %s"%round(rmse, 3) )

# MAPEを計算
mape = mean_absolute_percentage_error(y, y_pred)
print("MAPE = %s"%round(mape*100,3), "%")

# 決定係数を計算
r2 = r2_score(y, y_pred)
print("r2 = %s"%round(r2,3) )
```

- $MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i|$
- $MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$
- $RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$
- $MAPE = \frac{100}{n} \sum_{i=1}^n \frac{|y_i - \hat{y}_i|}{|y_i|}$
- $R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$

MAE = 3.057

MSE = 14.885

RMSE = 3.858

MAPE = 20.8 %

r2 = 0.884

■ 分類問題で用いる指標は、主に以下の4つ

- Accuracy
- Precision
- Recall
- F1 (f1-score)

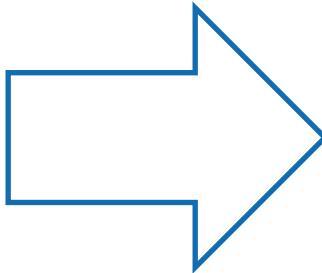
■ これらを計算するためには、混同行列を用意する

- TP (True Positive)
- FP (False Positive)
- FN (False Negative)
- TN (True Negative)

3-1 | 3-1. 疑似データの作成

```
# Priceの値を2000万以上ならTrue, そうでなければFalseに変更  
df_house['Price'] = df_house['Price'] >= 20  
df_house.head()
```

	Price
id	
0	24.8
1	59.5
2	7.0
3	7.5
4	9.8



	Price	AreaSize	HouseSize	PassedYear	Train	Walk
id						
0	True	98.4	74.2	4.8	5	6
1	True	379.8	163.7	9.3	12	12
2	False	58.6	50.5	13.0	16	2
3	False	61.5	58.0	12.8	16	1
4	False	99.6	66.4	14.0	16	5

```
# 目的変数と説明変数を分割
y = df_house["Price"].values
X = df_house[["AreaSize", "Train"]].values

# ロジスティック回帰モデルの構築
clf = SGDClassifier(loss='log_loss', max_iter=10000, fit_intercept=True,
                     random_state=1234, tol=1e-3)

# モデルの学習
clf.fit(X, y)
```

▼

SGDClassifier

```
SGDClassifier(loss='log_loss', max_iter=10000, random_state=1234)
```

■ scikit-learn の混同行列

- 縦軸（列）が予測ラベル、横軸（行）が正解ラベル
- 左側・上側から 0, 1 の順に並んでいる

	予測 : 0	予測 : 1
正解 : 0	TN	FP
正解 : 1	FN	TP

3-1 | 3-3. モデルの評価 2/3

```
# ラベルを予測
y_pred = clf.predict(X)

# 予測値と正解のクロス集計（混同行列）
conf_mat = confusion_matrix(y, y_pred)

conf_mat = pd.DataFrame(conf_mat,
                        index=['正解 = 2000万未満', '正解 = 2000万以上'],
                        columns=['予測 = 2000万未満', '予測 = 2000万以上'])

conf_mat
```

	予測 = 2000万未満	予測 = 2000万以上
正解 = 2000万未満	13	1
正解 = 2000万以上	1	8

```
# 正解率を計算
accuracy = accuracy_score(y, y_pred)
print('正解率 (Accuracy) = {:.3f}%'.format(100 * accuracy))

# Precision, Recall, F1-scoreを計算
precision, recall, f1_score, _ = precision_recall_fscore_support(y, y_pred)

# カテゴリ「2000万以上」に関するPrecision, Recall, F1-scoreを表示
print('適合率 (Precision) = {:.3f}%'.format(100 * precision[1]))
print('再現率 (Recall) = {:.3f}%'.format(100 * recall[1]))
print('F1値 (F1-score) = {:.3f}%'.format(100 * f1_score[1]))
```

正解率 (Accuracy) = 91.304%
 適合率 (Precision) = 88.889%
 再現率 (Recall) = 88.889%
 F1値 (F1-score) = 88.889%

- Accuracy = $\frac{TP+TN}{TP+TN+FN+FP}$
- Recall = $\frac{TP}{TP+FN}$
- Precision = $\frac{TP}{TP+FP}$
- F1 = $\frac{2 \times Recall \times Precision}{Recall + Precision}$

3-2_model_evaluation_roc_curve

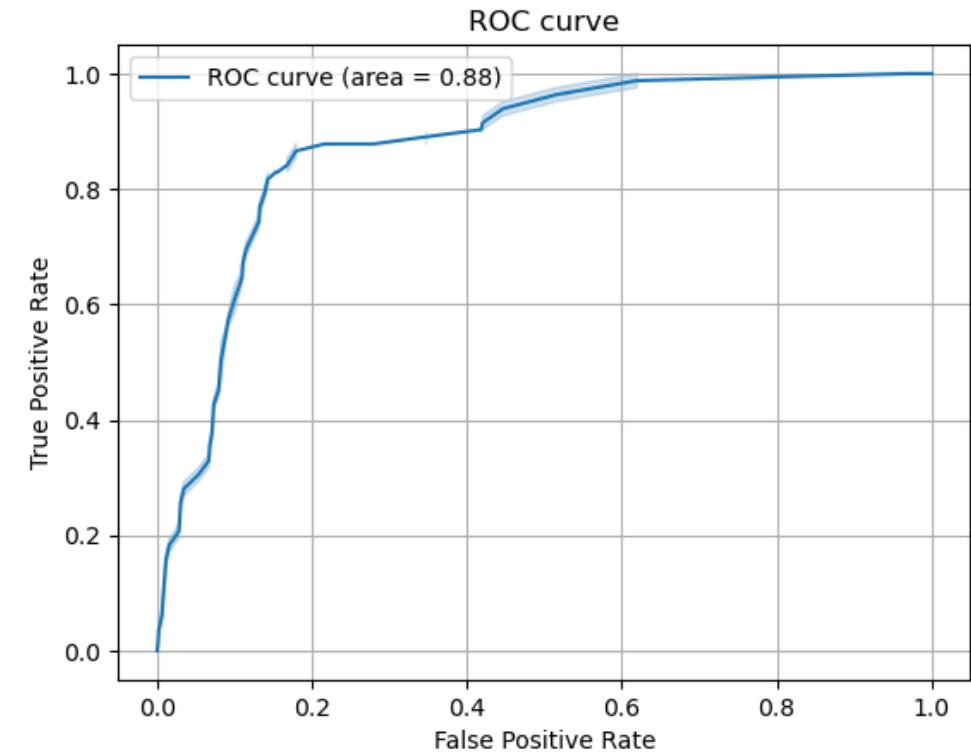
3-2_model_evaluation_roc_curve

■ 内容

- ROC 曲線を描き、AUC を算出
- 住宅価格の実データ ([House Sales in King County, USA](#)) を用いる

■ 手順

1. ライブラリの読み込み
2. データの読み込み
3. データの前処理
4. モデルの構築・学習
5. モデルの評価



3-2 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import SGDClassifier

# ROC曲線とAUCを算出するための関数
from sklearn.metrics import roc_curve, auc

# 標準化を行うためのクラス
from sklearn.preprocessing import StandardScaler
```

3-2 | 2. データの読み込み

```
# CSVファイルの読み込み
df_house = pd.read_csv("../..../1_data/ch3/kc_house_data.csv")[
    ['yr_renovated', 'sqft_living', 'sqft_lot', 'sqft_above', 'yr_built', 'sqft_living15', 'sqft_lot15']
]
# yr_renovatedを「改装済み」を表すカテゴリ変数に変換
df_house['yr_renovated'] = df_house['yr_renovated'] > 0

# データの確認(5件/全体21613件)
df_house.head()
```

	yr_renovated	sqft_living	sqft_lot	sqft_above	yr_built	sqft_living15	sqft_lot15
0	False	1180	5650	1180	1955	1340	5650
1	True	2570	7242	2170	1951	1690	7639
2	False	770	10000	770	1933	2720	8062
3	False	1960	5000	1050	1965	1360	5000
4	False	1680	8080	1680	1987	1800	7503

- sqft_living: 延床面積 [平方フィート]
- sqft_lot: 敷地面積 [平方フィート]
- sqft_above: 地下室を除いた延床面積 [平方フィート]
- yr_built: 建築年
- yr_renovated: 改装された年
- sqft_living15: 2015年における延床面積
- sqft_lot15: 2015年における敷地面積

```
# 目的変数と説明変数を分割
y = df_house["yr_renovated"].values
X = df_house.drop('yr_renovated', axis=1).values

# 各特徴量を平均0, 分散1となるように変換（標準化）するためのクラス
# 今回のデータセットは特徴量の平均や分散がそれぞれで大きく異なるため、
# そのままでは確率値がうまく算出されない
std_scaler = StandardScaler()

# 標準化した特徴量X_scaledを得る
# 以降はX_scaledをモデルの学習等に使う
X_scaled = std_scaler.fit_transform(X)
```

3-2 | 4. モデルの構築・学習

```
# ロジスティック回帰モデルの構築
clf = SGDClassifier(loss='log_loss', max_iter=10000, fit_intercept=True,
                     random_state=1234, tol=1e-3, )

# モデルの学習
clf.fit(X_scaled, y)
```

```
▼ SGDClassifier
SGDClassifier(loss='log_loss', max_iter=10000, random_state=1234)
```

3-2 | 5. モデルの評価 1/3

```
# 入力Xに対する各クラスの確率値を取得
y_pred = clf.predict_proba(X_scaled)

# 各入力に対するクラス確率が [クラス0の確率, クラス1の確率]
# という順番で配列に格納されている
y_pred[:5]
```

```
array([[0.97522111, 0.02477889],
       [0.93902818, 0.06097182],
       [0.95333629, 0.04666371],
       [0.97641698, 0.02358302],
       [0.9906112 , 0.0093888 ]])
```

3-2 | 5. モデルの評価 2/3

```
# y_pred[:, 1] でクラス1の確率のみを取得
y_score = y_pred[:, 1]

# ROC曲線の計算
# 偽陽性率 (FPR)、真陽性率 (TPR)、閾値を計算
# 21613件全てに対応する点の表示には時間がかかるため、1000件に絞る
num_test = 1000
fpr, tpr, thresholds = roc_curve(y[:num_test], y_score[:num_test])

# AUC を計算
auc_value = auc(fpr, tpr)
auc_value
```

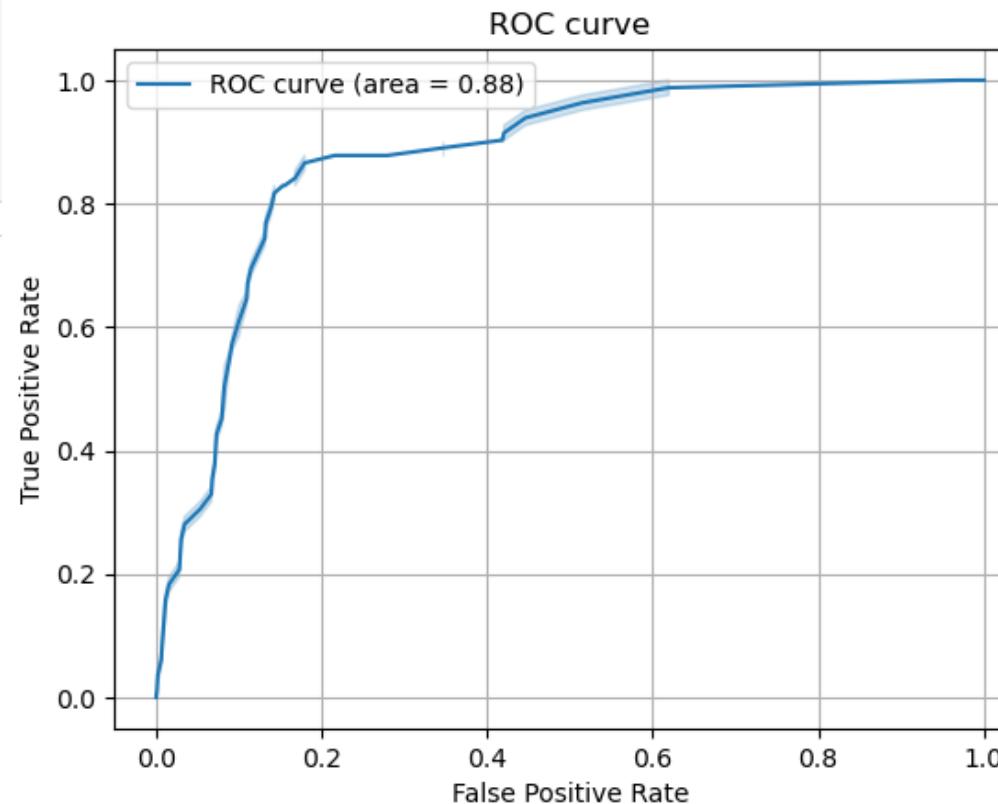
0.878226811465195

3-2 | 5. モデルの評価 3/3

```
# 閾値の数 (ROC 曲線上の点の数)
print(thresholds.shape[0], "points")

# ROC 曲線をプロット
sns.lineplot(x=fpr, y=tpr, label='ROC curve (area = %.2f)'%auc_value)
plt.title('ROC curve')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.grid(True)
```

89 points



3-3_micro_macro_average

■ 内容

- 多クラス分類における評価指標を計算
- アヤメの品種分類のデータセット ([The Iris Dataset](#)) を使用

■ 手順

- ライブラリの読み込み
- データの読み込み
- モデルの構築・学習
- 混同行列の確認
- Micro 平均
- Macro 平均
- Macro 平均（手計算で行う場合）

	precision	recall	f1-score	support
setosa	0.55	1.00	0.71	50
versicolor	1.00	0.08	0.15	50
virginica	0.91	1.00	0.95	50
accuracy			0.69	150
macro avg	0.82	0.69	0.60	150
weighted avg	0.82	0.69	0.60	150

3-3 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np

# データセットを読み込む関数
from sklearn.datasets import load_iris

# モデルを表すクラス
from sklearn.linear_model import SGDClassifier

# 分類問題における評価指標の関数
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confusion_matrix
from sklearn.metrics import classification_report, balanced_accuracy_score
```

3-3 | 2. データの読み込み

```
# アヤメの品種 (Setosa, Versicolor, Virginica) を分類する3クラス分類問題
iris = load_iris()

# 目的変数
y = iris.target
# 説明変数 (萼片の長さ、萼片の幅、花弁の長さ、花弁の幅)
X = iris.data

# ラベルに対応する文字列のリスト
label_txt = iris.target_names
```

3-3 | 3. モデルの構築・学習

```
# ロジスティック回帰モデルの構築
clf = SGDClassifier(loss='log_loss', max_iter=10000, fit_intercept=True,
                     random_state=1234, tol=1e-3)

# 学習を実行
clf.fit(X, y)
```

```
▼ SGDClassifier
SGDClassifier(loss='log_loss', max_iter=10000, random_state=1234)
```

3-3 | 4. 混同行列の確認

```
# ラベルを予測  
y_pred = clf.predict(X)  
  
# 予測値と正解のクロス集計（混同行列）  
conf_mat = confusion_matrix(y, y_pred)  
conf_mat = pd.DataFrame(conf_mat,  
                        index=label_txt,  
                        columns=label_txt)  
  
conf_mat
```

	setosa	versicolor	virginica
setosa	50	0	0
versicolor	41	4	5
virginica	0	0	50

■ 全クラスのデータをまとめて扱い、評価指標を求める

- Precision · Recall · F1 は、Accuracy と同じ値になるため、計算する意味はない

```
# 正解率を計算
accuracy = accuracy_score(y, y_pred)
print('正解率 (Accuracy) = {:.3f}%'.format(100 * accuracy))
```

正解率 (Accuracy) = 69.333%

3-3 | 6. Macro 平均

```
# Precision, Recall, F1, Accuracyをまとめて表示
scores = classification_report(y, y_pred, target_names=label_txt)
print(scores)
```

	precision	recall	f1-score	support
setosa	0.55	1.00	0.71	50
versicolor	1.00	0.08	0.15	50
virginica	0.91	1.00	0.95	50
accuracy			0.69	150
macro avg	0.82	0.69	0.60	150
weighted avg	0.82	0.69	0.60	150

3-3 | 7. Macro 平均（手計算で行う場合）

```
# macro-Accuracyの計算
macro_accuracy = balanced_accuracy_score(y, y_pred)
print('macro-Accuracy = {:.3f}%'.format(100 * macro_accuracy))

# 各クラスのPrecision, Recall, F1を計算
precision, recall, f1_score, _ = precision_recall_fscore_support(y, y_pred)

# macro-Precisionの計算
macro_precision = np.mean(precision)
print('macro-Precision = {:.3f}%'.format(100 * macro_precision))

# macro-Recallの計算
macro_recall = np.mean(recall)
print('macro-Recall = {:.3f}%'.format(100 * macro_recall))

# macro-f1の計算①(各クラスのF1を平均する)
macro_f1 = np.mean(f1_score)
print('macro-F1① = {:.3f}%'.format(100 * macro_f1))

# macro-f1の計算②(macro-Recallとmacro-Precisionを用いて計算)
macro_f1 = 2*macro_precision*macro_recall/(macro_precision+macro_recall)
print('macro-F1② = {:.3f}%'.format(100 * macro_f1))
```

```
macro-Accuracy = 69.333%
macro-Precision = 81.951%
macro-Recall = 69.333%
macro-F1① = 60.325%
macro-F1② = 75.116%
```

現場で使える 機械学習・データ分析基礎講座

第4章：モデルの検証と正則化

ノートブック解説

■ モデルの検証と正則化

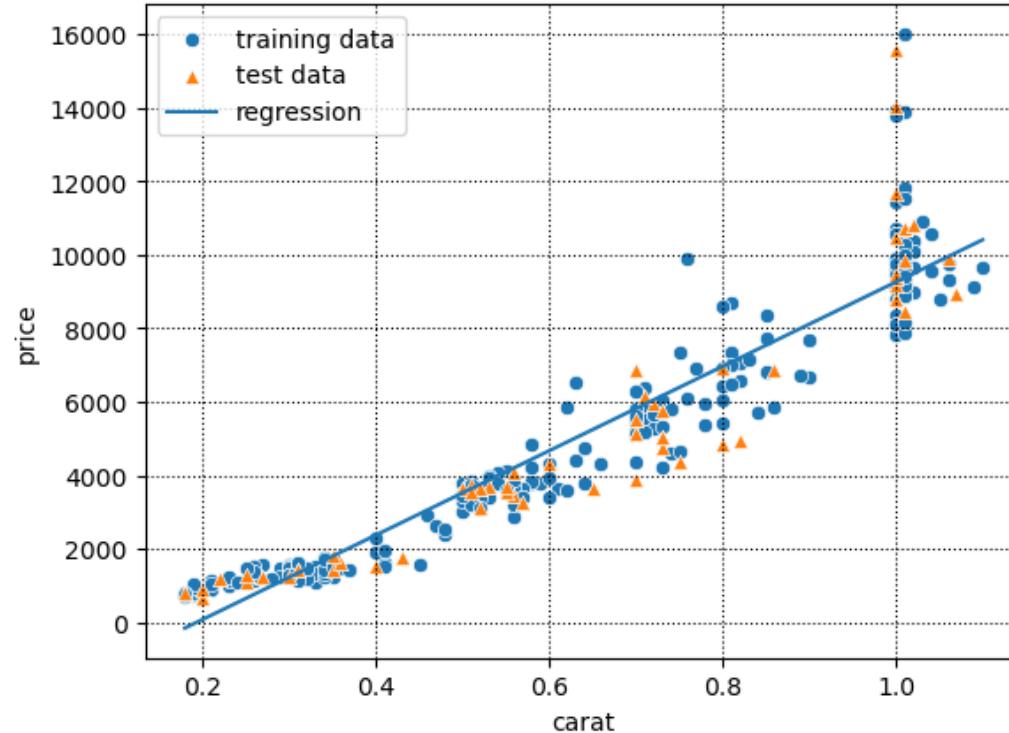
- 4-1_how_to_validation.ipynb
- 4-2_regularization.ipynb
- 4-3_bias_variance_noise.ipynb

4-1_how_to_validation

4-1_how_to_validation

■ 内容

- ホールドアウト法と交差検証法によるモデルの評価
- 線形回帰モデルを用いる



Fold 1	MAE = 855.624
Fold 2	MAE = 663.313
Fold 3	MAE = 555.675
Fold 4	MAE = 803.69
Fold 5	MAE = 874.072
Cross Validation MAE = 750.475	

■ 手順

1. ライブラリの読み込み
2. データの読み込み
3. データの可視化
4. ホールドアウト法
 1. データの分割
 2. モデルの構築・学習
 3. 汎化誤差の評価
 4. 予測結果の可視化
5. 交差検証法
 1. KFold() のみを用いる場合
 2. cross_val_score() を用いる場合
 3. cross_validate() を用いる場合

4-1 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression

# MAEを計算するための関数
from sklearn.metrics import mean_absolute_error
# ホールドアウト法を実行するための関数
from sklearn.model_selection import train_test_split
# 交差検証法を実行するための関数
from sklearn.model_selection import KFold, cross_val_score, cross_validate

# dict型の変数を綺麗にprintするための関数
from pprint import pprint
```

4-1 | 2. データの読み込み

```
# CSVファイルの読み込み  
df_diamond = pd.read_csv("../.. /1_data/ch4/diamond_data.csv")
```

```
# ダイヤモンドの重さの単位であるカラットとその価格に関する実際のデータ  
# 『回帰分析入門』より引用  
display(df_diamond.head())  
df_diamond.describe()
```

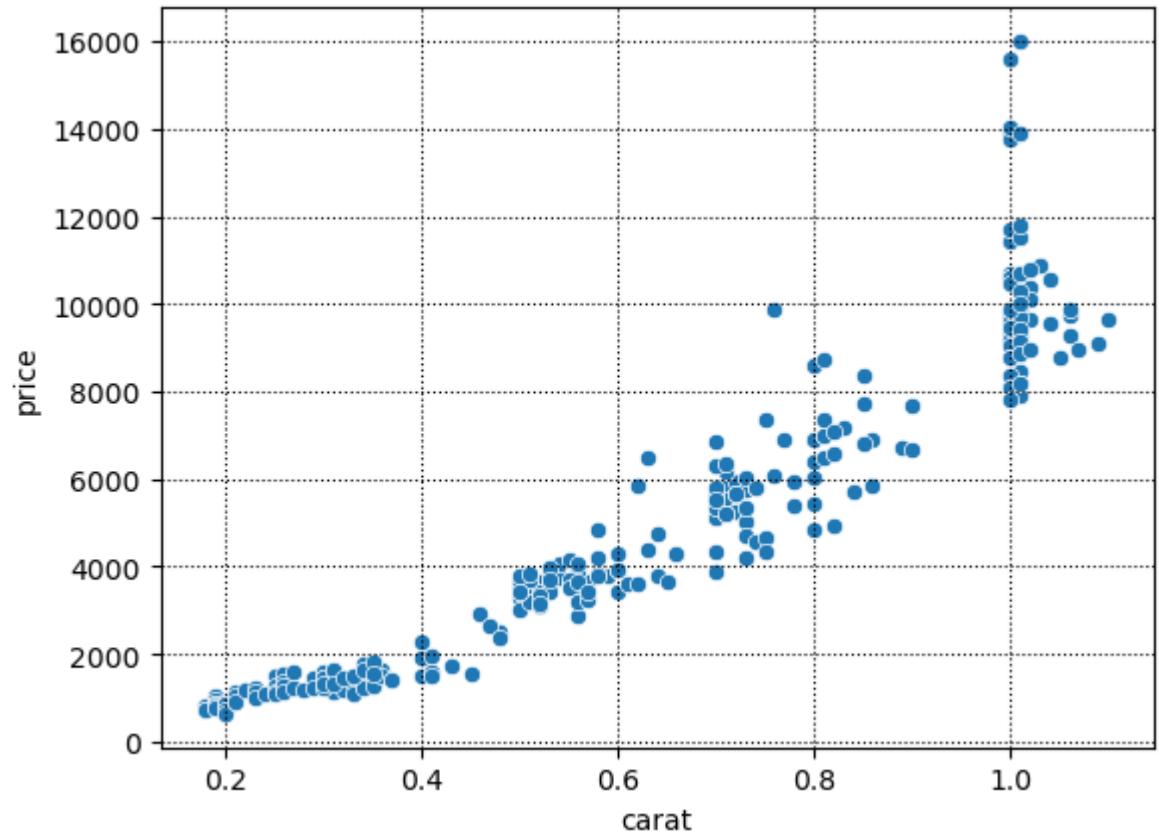
	carat	price
0	0.30	1302
1	0.30	1510
2	0.30	1510
3	0.30	1260
4	0.31	1641

	carat	price
count	308.000000	308.000000
mean	0.630909	5019.483766
std	0.277183	3403.115715
min	0.180000	638.000000
25%	0.350000	1625.000000
50%	0.620000	4215.000000
75%	0.850000	7446.000000
max	1.100000	16008.000000

4-1 | 3. データの可視化

```
# データのプロット
# 正解データ作成
x = df_diamond["carat"].values
# 学習用データ作成
y = df_diamond["price"].values

sns.scatterplot(x=x, y=y, marker='o')
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')
plt.ylabel('price')
plt.xlabel('carat')
plt.show()
```



■ 手順

- データを事前に学習用とテスト用に分割
 - それぞれ訓練用・検証用と呼ぶ場合もある
 - ここでは簡単化のため学習用・テスト用と呼ぶ
- テスト用データで学習済みモデルの汎化誤差を評価する

```
# scikit-learnに入力するために整形
X = x.reshape(-1,1)

# 全データのうち、何%をテストデータにするか（今回は20%に設定）
test_size = 0.2

# 学習データとテストデータの分割
# テストデータはランダムに選択される
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state=1234)
```

- `X_train` : 学習用データの説明変数
- `X_test` : テスト用データの説明変数
- `y_train` : 学習用データの目的変数
- `y_test` : テスト用データの目的変数

4-1 | 4-2. モデルの構築・学習

```
# 線形回帰モデルの構築
regr = LinearRegression(fit_intercept=True)
# モデルを学習させる
regr.fit(X_train, y_train)

# 学習用データに対する予測を実行
y_pred_train = regr.predict(X_train)

# 学習データに対するMAEを計算（訓練誤差の評価）
mae = mean_absolute_error(y_train, y_pred_train)
print("MAE = %s"%round(mae,3))
```

MAE = 707.119

4-1 | 4-3. 汎化誤差の評価

```
# テストデータに対する予測を実行  
y_pred_test = regr.predict(X_test)  
  
# テストデータに対するMAEを計算（汎化誤差の評価）  
mae = mean_absolute_error(y_test, y_pred_test)  
print("MAE = %s"%round(mae,3))
```

MAE = 855.624

4-1 | 4-4. 予測結果の可視化

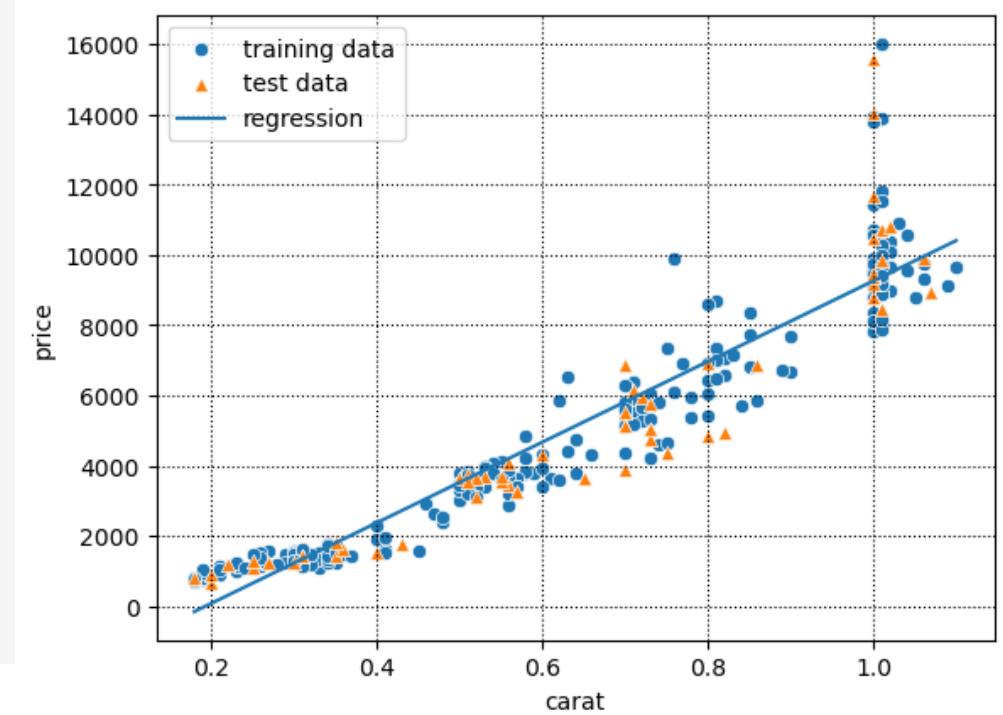
```
# データの散布図を表示
# X_train, X_test (は2次元配列なので、1次元の配列へ変換
sns.scatterplot(x=np.ravel(X_train), y=y_train, marker='o', label='training data')
sns.scatterplot(x=np.ravel(X_test), y=y_test, marker='^', label='test data')

# 軸ラベルの設定
plt.ylabel("price")
plt.xlabel("carat")

# グリッド線の表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')

# 回帰直線の表示
sns.lineplot(x=np.ravel(X_train), y=y_pred_train, label='regression')

# 凡例の表示位置を調整
plt.legend(loc='best')
plt.show()
```



■ 手順

- データを複数のグループに分ける
- 1つのグループをテスト用、それ以外を学習用とする
- テスト用と学習用を交代させながら、汎化誤差を評価する

■ 実装方法

- KFold() のみを用いる
- KFold() と cross_val_score() を用いる
- KFold() と cross_validate() を用いる

4-1 | 5-1. KFold() のみを用いる場合

```
X = x.reshape(-1,1) # scikit-learnに入力するために整形
n_split = 5 # グループ数を設定（今回は5分割）

cross_valid_mae = 0
split_num = 1

# K-Foldクロスバリデーションの定義
kf = KFold(n_splits=n_split, shuffle=True, random_state=1234)
```

4-1 | 5-1. KFold() のみを用いる場合

```
# テスト役を交代させながら学習と評価を繰り返す
# KFoldオブジェクトから取り出せる値は、インデックスの集まり
for train_idx, test_idx in kf.split(X, y):
    X_train, y_train = X[train_idx], y[train_idx] #学習用データ
    X_test, y_test = X[test_idx], y[test_idx]      #テスト用データ

    # 学習用データを使って線形回帰モデルを学習
    regr = LinearRegression(fit_intercept=True)
    # モデルを学習させる
    regr.fit(X_train, y_train)

    # テストデータに対する予測を実行
    y_pred_test = regr.predict(X_test)

    # テストデータに対するMAEを計算
    mae = mean_absolute_error(y_test, y_pred_test)
    print("Fold %s"%split_num)
    print("MAE = %s"%round(mae, 3), end="\n\n")

    cross_valid_mae += mae #後で平均を取るためにMAEを加算
    split_num += 1

# MAEの平均値を計算し、最終的な汎化誤差の値とする
final_mae = cross_valid_mae / n_split
print("Cross Validation MAE = %s"%round(final_mae, 3))
```

```
Fold 1
MAE = 855.624

Fold 2
MAE = 663.313

Fold 3
MAE = 555.675

Fold 4
MAE = 803.69

Fold 5
MAE = 874.072

Cross Validation MAE = 750.475
```

4-1 | 5-2. cross_val_score() を用いる場合

```
from sklearn.model_selection import cross_val_score

# モデルを定義
model = LinearRegression()

# 5-fold cross-validationを実行し、MAEを計算
# scoringで二乗誤差を選択
scores = cross_val_score(model, X, y, cv=kf, scoring='neg_mean_absolute_error')

# スコアは符号反転させたMAEなので、-1を掛けて通常のMAEに戻す
mae_scores = -scores

print('Cross-validation MAE: {}'.format(mae_scores))
print('Average cross-validation MAE: {}'.format(mae_scores.mean()))
```

```
Cross-validation MAE: [855.62377818 663.31306948 555.67508545 803.68954335 874.07185185]
Average cross-validation MAE: 750.4746656623304
```

4-1 | 5-3. cross_validate() を用いる場合

```
# scoringにMAEとRMSEを設定
scoring = {"mae": "neg_mean_absolute_error",
           "rmse": "neg_root_mean_squared_error"}

# 5-fold cross-validationを実行し、評価指標を計算
scores = cross_validate(regr, X, y, cv=kf, scoring=scoring,
                        return_estimator=True)

# scoresはdict型の変数になっている
pprint(scores)

# スコアは符号反転させたものなので、-1を掛けて通常の指標に戻す
mae_array = scores['test_mae']*-1
print("MAE = %s"%np.round(mae_array, 3))
rmse_array = scores['test_rmse']*-1
print("RMSE = %s"%np.round(rmse_array, 3))

# 評価指標の平均値を計算し、最終的な汎化誤差の値とする
final_mae_ = mae_array.mean()
print("Cross Validation MAE = %s"%round(final_mae_, 3))
final_rmse_ = rmse_array.mean()
print("Cross Validation RMSE = %s"%round(final_rmse_, 3))
```

```
{'estimator': [LinearRegression(),
                LinearRegression(),
                LinearRegression(),
                LinearRegression(),
                LinearRegression()],
 'fit_time': array([0.00200105, 0.00100088, 0.00100064, 0.00099897, 0.00099874]),
 'score_time': array([0.00099945, 0.00199771, 0.00100064, 0.0010016 , 0.00100136]),
 'test_mae': array([-855.62377818, -663.31306948, -555.67508545, -803.68954335,
                    -874.07185185]),
 'test_rmse': array([-1358.85819781, -814.03785238, -718.08808711, -1303.92266334,
                    -1286.04938875])}
MAE = [855.624 663.313 555.675 803.69 874.072]
RMSE = [1358.858 814.038 718.088 1303.923 1286.049]
Cross Validation MAE = 750.475
Cross Validation RMSE = 1096.191
```

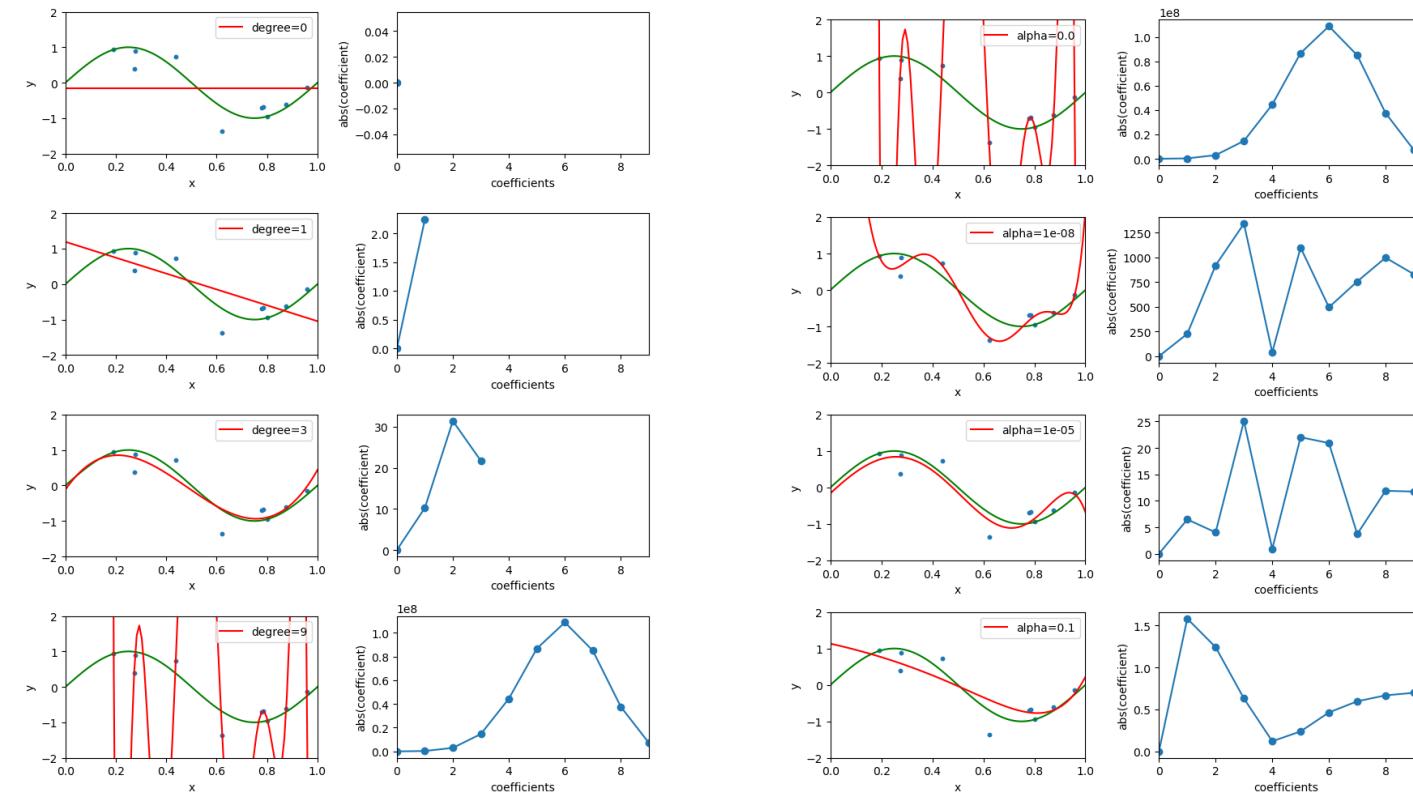
4-2 regularization

■ 内容

- 多項式特徴量を用いたモデルで、過学習や未学習を確認
- 正則化の強さを変更することで、どのような変化が起こるか確認

■ 手順

- ライブラリの読み込み
- 疑似データの生成
- 正則化なしで学習
- 正則化ありで学習
 - Ridge
 - Lasso
 - ElasticNet



4-2 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# 線形回帰モデル
from sklearn.linear_model import LinearRegression
# 正則化つき線形回帰モデル
from sklearn.linear_model import Ridge,Lasso,ElasticNet
# 多項式特徴量を計算する前処理用の関数
from sklearn.preprocessing import PolynomialFeatures
# 機械学習パイプラインを作成するための関数
from sklearn.pipeline import make_pipeline
```

4-2 | 2. 疑似データの生成

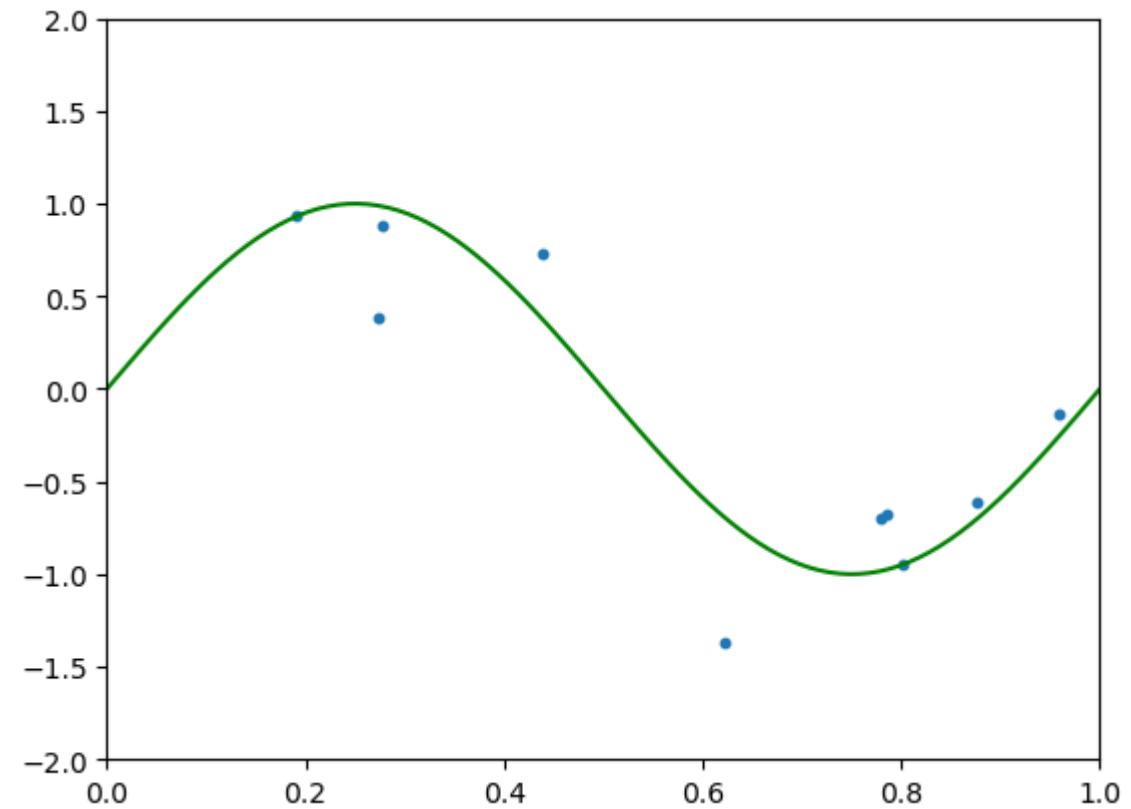
```
## ノイズのないデータの生成
# sin(2πx)を関数として定義
def f(x):
    return np.sin(2 * np.pi * x)

# x軸の値を生成
x_plot = np.linspace(0, 1, 100)
# きれいなsin関数を表示
plt.plot(x_plot, f(x_plot), color='green')

## ノイズのあるデータの生成
# サンプル数を設定
n_samples = 10
# 乱数シードを固定
np.random.seed(1234)

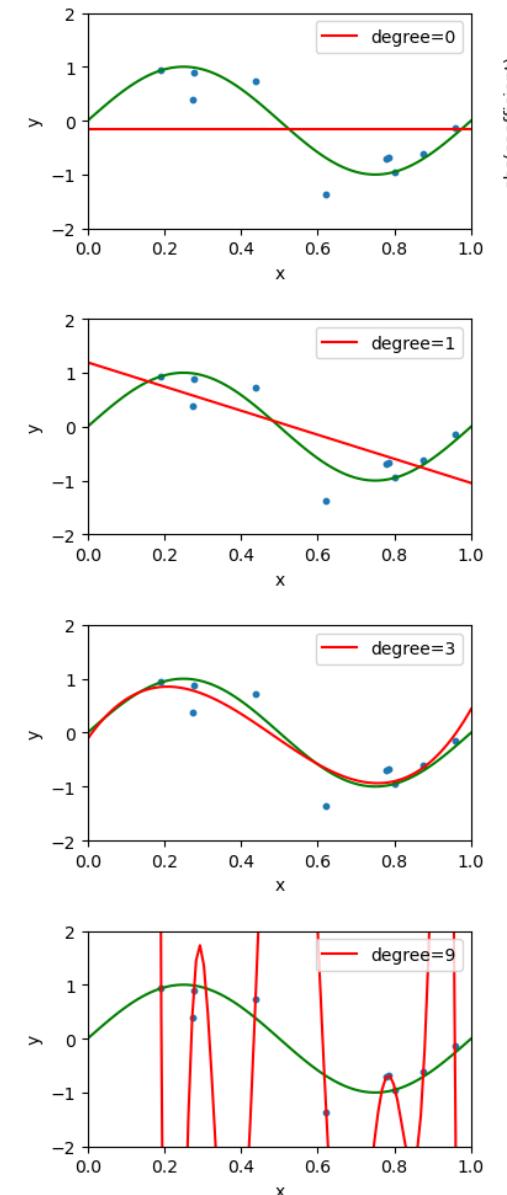
# x軸の値を生成
X = np.random.uniform(0, 1, size=n_samples)[:, np.newaxis]
# sin関数の値にノイズを加えて、ばらつきを表現
y = f(X) + np.random.normal(scale=0.3, size=n_samples)[:, np.newaxis]
# ノイズのあるデータを表示
plt.scatter(X, y, s=10)

# 表示範囲の調整
plt.ylim((-2, 2))
plt.xlim((0, 1))
```



4-2 | 3. 正則化なしで学習 1/3

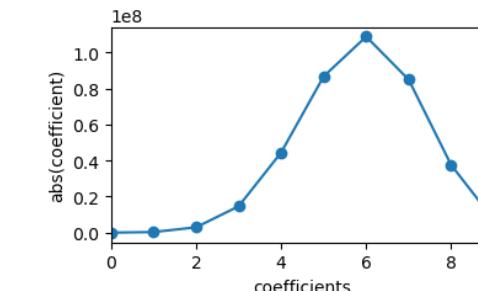
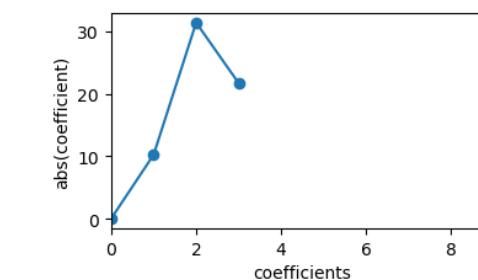
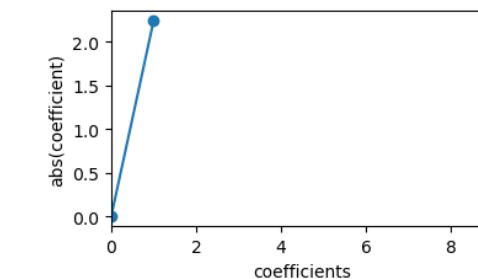
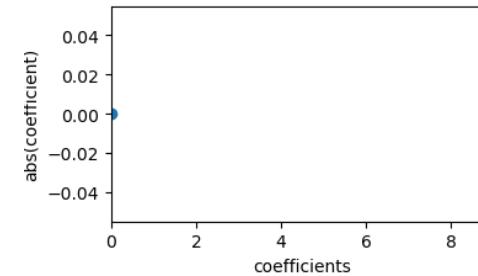
```
# 左側のグラフをまとめて描画する関数
def plot_approximation(est, ax, label=None):
    # 理想的なモデルを描画（緑色の線）
    ax.plot(x_plot, f(x_plot), color='green')
    # 学習用データを描画（青色の点）
    ax.scatter(X, y, s=10)
    # モデルの予測結果を描画（赤色の線）
    ax.plot(x_plot, est.predict(x_plot[:, np.newaxis]), color='red', label=label)
    # 表示範囲の調整
    ax.set_xlim((0, 1))
    ax.set_ylim((-2, 2))
    # 軸ラベルの設定
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    # 凡例の表示位置を調整
    ax.legend(loc='upper right') #, fontsize='small')
```



4-2 | 3. 正則化なしで学習 2/3

```
# 右側のグラフをまとめて描画する関数
def plot_coefficients(est, ax, label=None, yscale='log'):
    # 最後のステップ（線形回帰）の係数を取得
    coef = est.steps[-1][1].coef_.ravel()
    if yscale == 'log':
        # logスケールで描画する場合、係数の絶対値をとる
        coef = np.abs(coef) + 1e-1 # log(0)を防ぐため
        # y軸を対数スケールに変更
        ax.semilogy(coef, marker='o', label=label)
        ax.set_yscale('log')
        ax.set_ylabel('abs(coefficient) in log scale')
    else:
        ax.plot(np.abs(coef), marker='o', label=label)
        ax.set_ylabel('abs(coefficient)')

    ax.set_xlabel('coefficients')
    ax.set_xlim((0, 9))
```



4-2 | 3. 正則化なしで学習 3/3

```
# 4×2の描画エリアを作成
fig, ax_rows = plt.subplots(4, 2, figsize=(8, 10))

# 各行で異なる次数の多項式特徴量を作成
degrees = [0, 1, 3, 9]
# グラフを表示する場所 (ax) を指定
for ax_row, degree in zip(ax_rows, degrees):
    ax_left, ax_right = ax_row
    # 多項式特徴量と線形回帰モデルを組み合わせたパイプラインを作成
    est = make_pipeline(
        PolynomialFeatures(degree),
        LinearRegression()
    )
    # モデルを学習
    est.fit(X, y)
    # 予測結果と係数を描画
    plot_approximation(est, ax_left, label='degree=%d' % degree)
    plot_coefficients(est, ax_right, yscale=None)

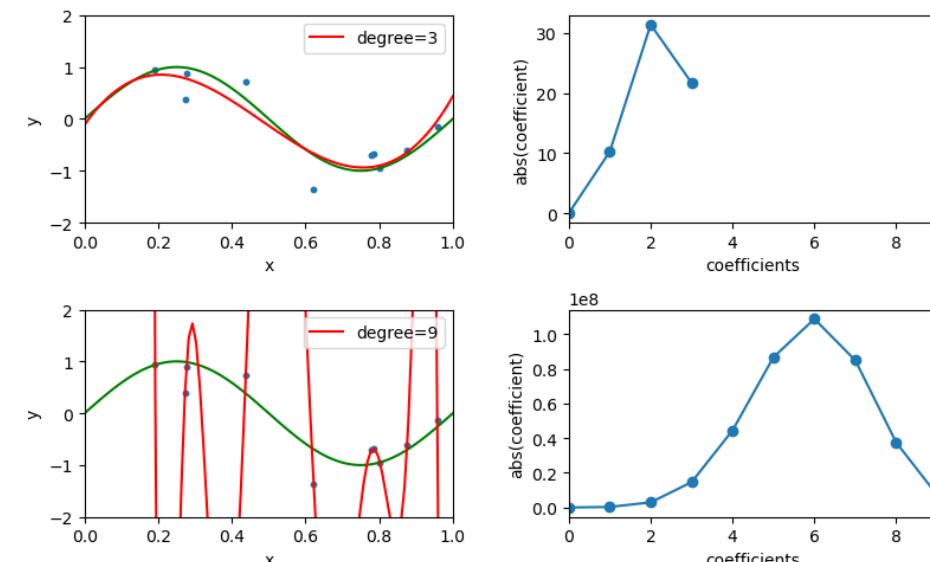
# 各グラフの表示位置を調整
plt.tight_layout()
```

PolynomialFeatures

- データから多項式特徴量を生成するメソッド
- 引数degreeで次数を設定
- 主にpipeline()の中に組み込んで、モデルの入力として使用

degreeの値による特徴量の違い

- degree=0: $y = a_0$
- degree=1: $y = a_0 + a_1 x$
- degree=3: $y = a_0 + a_1 x + a_2 x^2 + a_3 x^3$
- degree=9: $y = a_0 + \sum_{i=1}^9 a_i x^i$

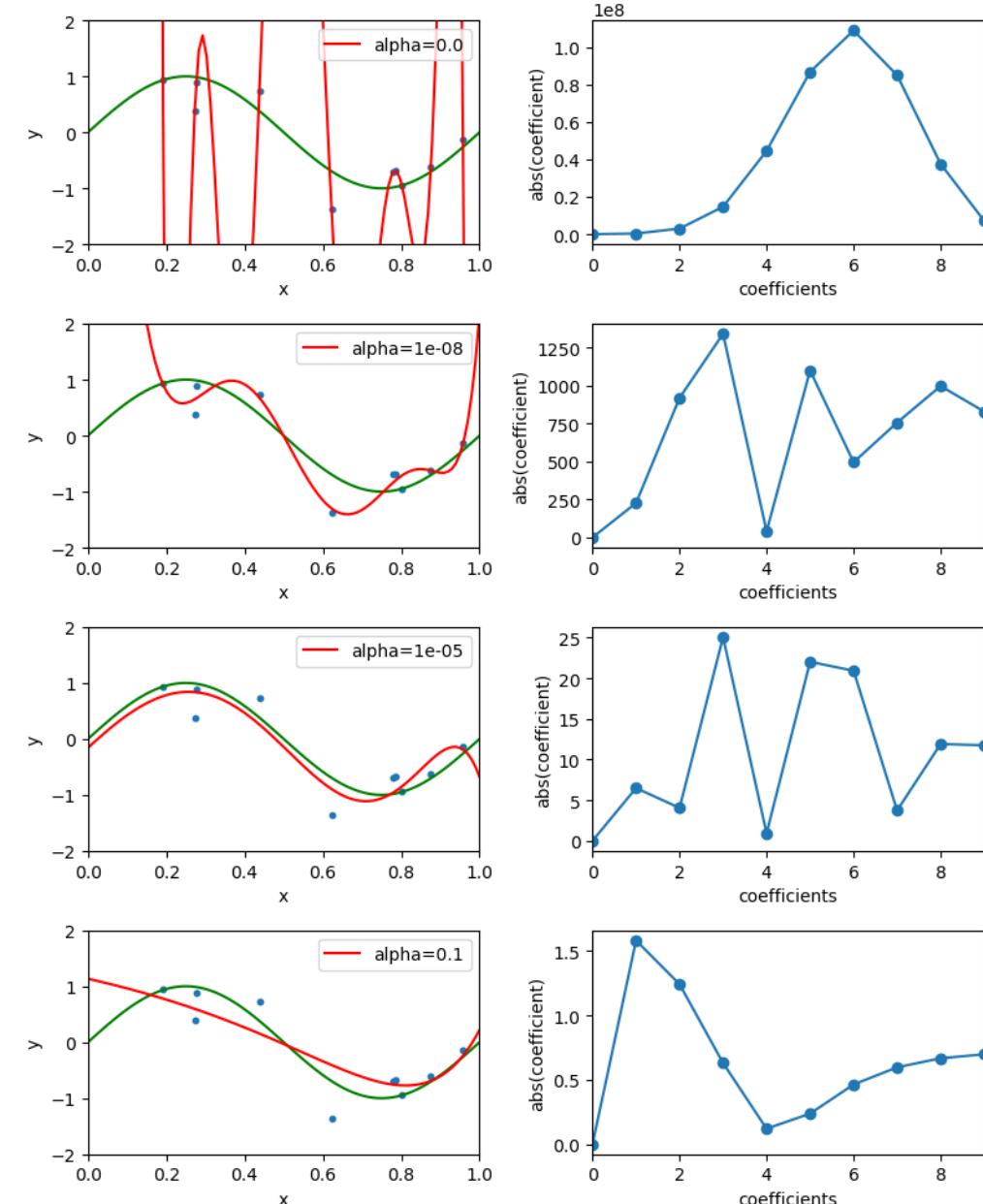


4-2 | 4-1. Ridge

```
# 4×2の描画エリアを作成
fig, ax_rows = plt.subplots(4, 2, figsize=(8, 10))
# 多項式特徴量の次数を設定(9に固定)
degree = 9
# alpha(数式ではλ)の値を4つ指定する
alphas = [0.0, 1e-8, 1e-5, 1e-1]

# alphaを変えながら先程と同様のグラフを表示
for alpha, ax_row in zip(alphas, ax_rows):
    ax_left, ax_right = ax_row
    # 多項式特徴量とRidgeを組み合わせたパイプラインを作成
    # ConvergenceWarning(学習が収束しない)は、無視してよい
    est = make_pipeline(
        PolynomialFeatures(degree),
        Ridge(alpha=alpha, random_state=1234)
    )
    # モデルを学習
    est.fit(X, y)
    # 予測結果と係数を描画
    plot_approximation(est, ax_left, label='alpha=%r' % alpha)
    plot_coefficients(est, ax_right, yscale=None)

# 各グラフの表示位置を調整
plt.tight_layout()
```

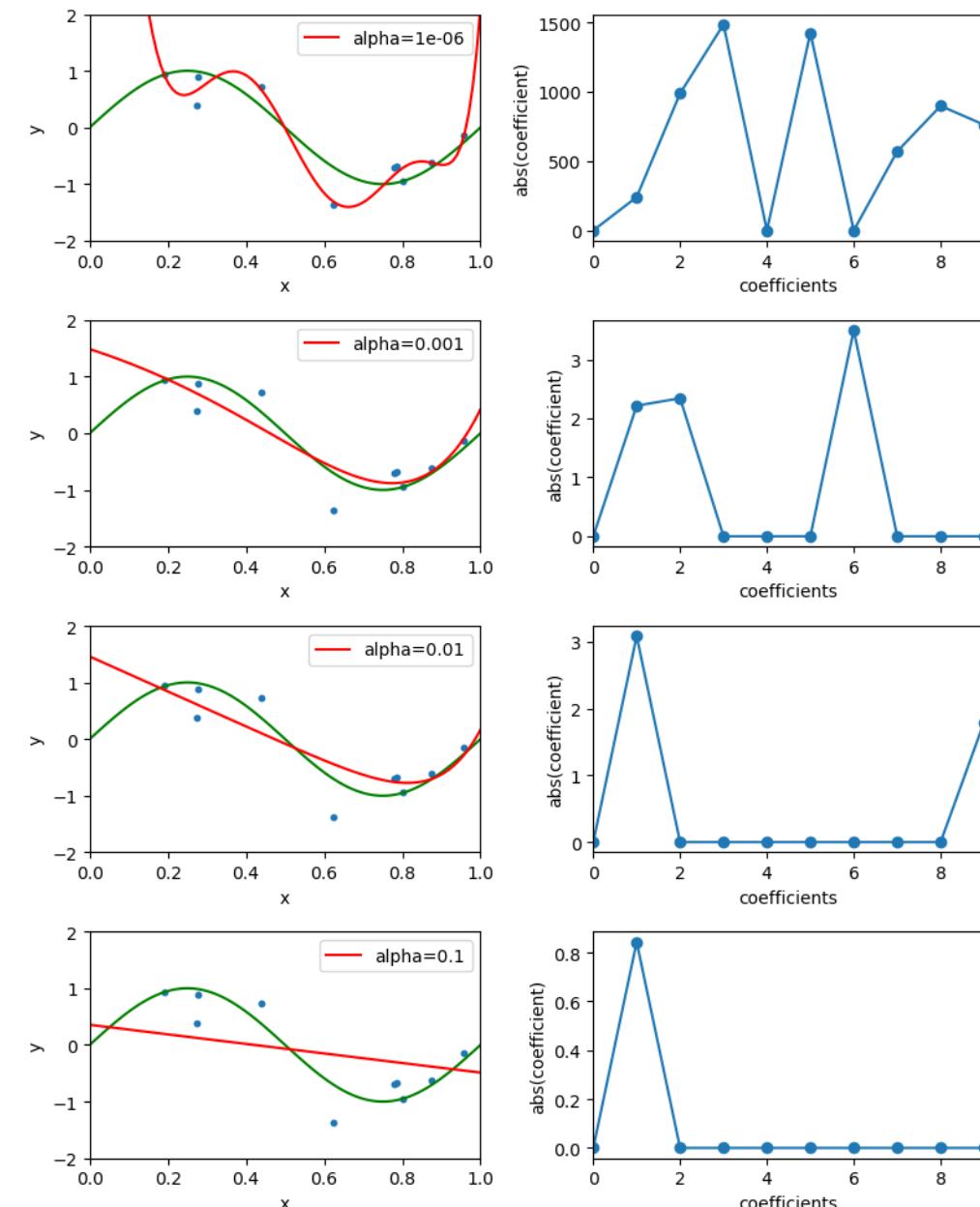


4-2 | 4-2. Lasso

```
# 4×2の描画エリアを作成
fig, ax_rows = plt.subplots(4, 2, figsize=(8,10))
# 多項式特徴量の次数を設定（9に固定）
degree = 9
# alpha(数式ではλ)の値を4つ指定する
alphas = [1e-6, 1e-3, 1e-2, 1e-1]

# alphaを変えながら先程と同様のグラフを表示
for alpha, ax_row in zip(alphas, ax_rows):
    ax_left, ax_right = ax_row
    # 多項式特徴量とLassoを組み合わせたバイオラインを作成
    # ConvergenceWarning (学習が収束しない) は、無視してよい
    est = make_pipeline(
        PolynomialFeatures(degree),
        Lasso(alpha=alpha, max_iter=int(1e8), random_state=1234)
    )
    # モデルを学習
    est.fit(X, y)
    # 予測結果と係数を描画
    plot_approximation(est, ax_left, label='alpha=%r' % alpha)
    plot_coefficients(est, ax_right, yscale=None)

# 各グラフの表示位置を調整
plt.tight_layout()
plt.show()
```

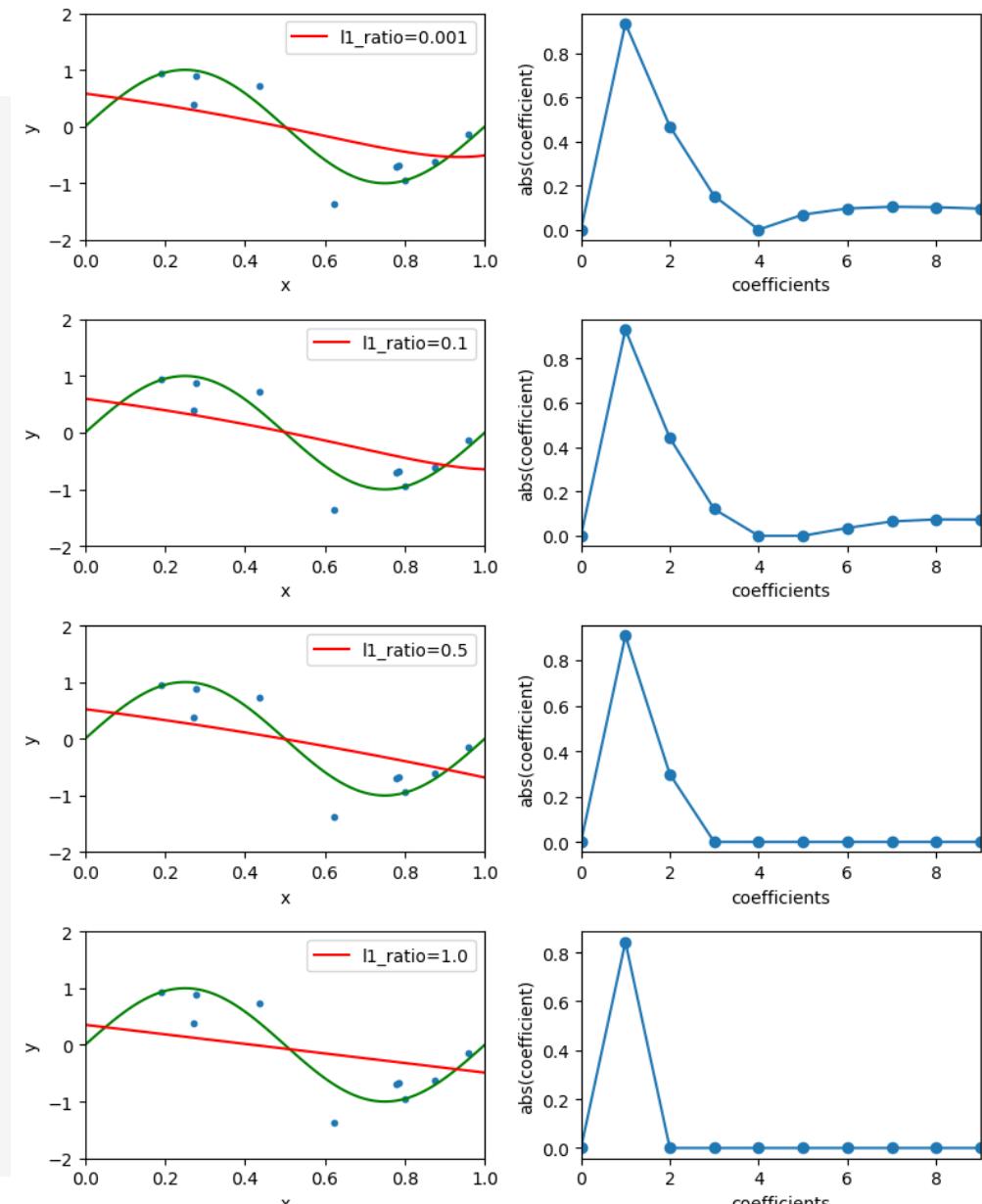


4-2 | 4-3. ElasticNet

```
# 4×2の描画エリアを作成
fig, ax_rows = plt.subplots(4, 2, figsize=(8,10))
# 多項式特徴量の次数を設定（9に固定）
degree = 9
# 正則化全体の強さを指定
alpha = 1e-1
# L1正則化の強さを4つ指定する
# L2正則化の強さは $1 - l1\_ratio$ で自動的に設定される
l1_ratios = [0.001, 0.1, 0.5, 1.0]

# l1_ratioを変えながら先程と同様のグラフを表示
for l1_ratio, ax_row in zip(l1_ratios, ax_rows):
    ax_left, ax_right = ax_row
    # 多項式特徴量とElasticNetを組み合わせたバイオラインを作成
    # ConvergenceWarning (学習が収束しない) は、無視してよい
    est = make_pipeline(
        PolynomialFeatures(degree),
        ElasticNet(alpha=alpha, l1_ratio=l1_ratio, max_iter=int(1e8), tol=1e7)
    )
    # モデルを学習
    est.fit(X, y)
    # 予測結果と係数を描画
    plot_approximation(est, ax_left, label='l1_ratio=%r' % l1_ratio)
    plot_coefficients(est, ax_right, yscale=None)

# 各グラフの表示位置を調整
plt.tight_layout()
```



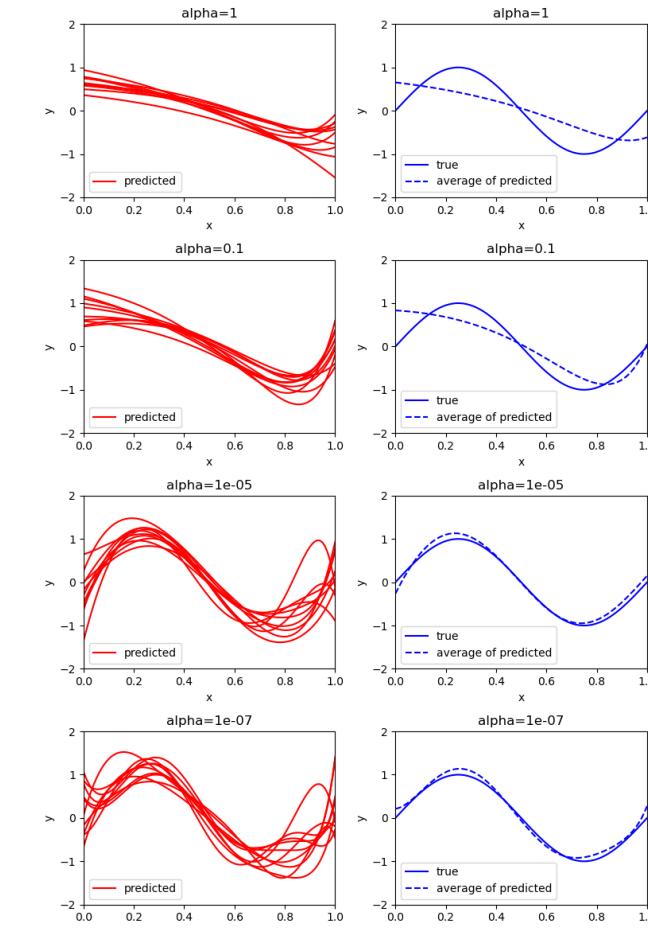
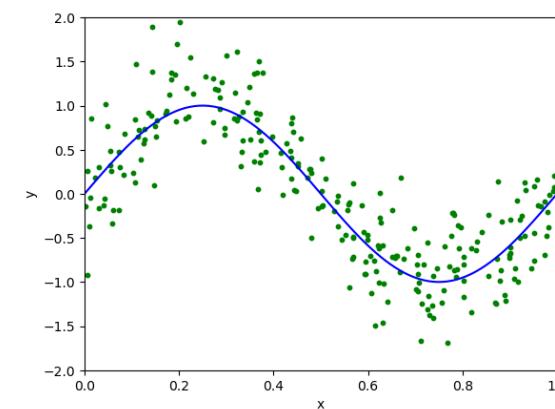
4-3_bias_variance_noise

■ 内容

- バイアスの大きいモデルと、バリアンスの大きいモデルの性質を確認
- 多項式特徴量を用いたモデルを例にとる

■ 手順

- ライブラリの読み込み
- 真の関数（理想的なモデル）の作成
- ノイズを含むデータの生成
- モデルの構築・学習



4-3 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

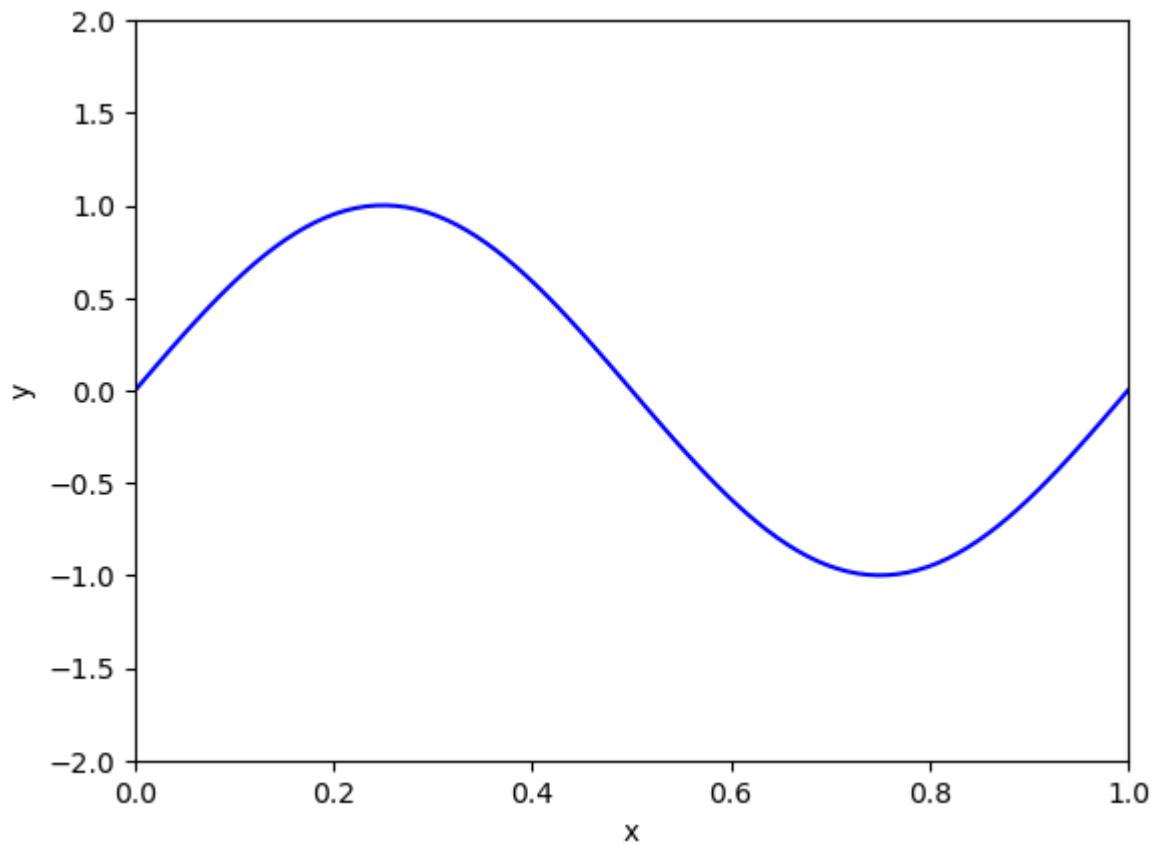
# L2正則化つき線形回帰モデル
from sklearn.linear_model import Ridge
# 多項式特徴量を計算する前処理用の関数
from sklearn.preprocessing import PolynomialFeatures
# 機械学習バイブルインを作成するための関数
from sklearn.pipeline import make_pipeline
```

4-3 | 2. 真の関数（理想的なモデル）の作成

```
def f(x):
    return np.sin(2 * np.pi * x)

# x座標の値を計算
x_plot = np.linspace(0, 1, 100)
# y座標の値を計算
y = f(x_plot)
# 理想的なモデルの曲線を表示
plt.plot(x_plot, y, color='b')

plt.ylim((-2, 2))
plt.xlim((0, 1))
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



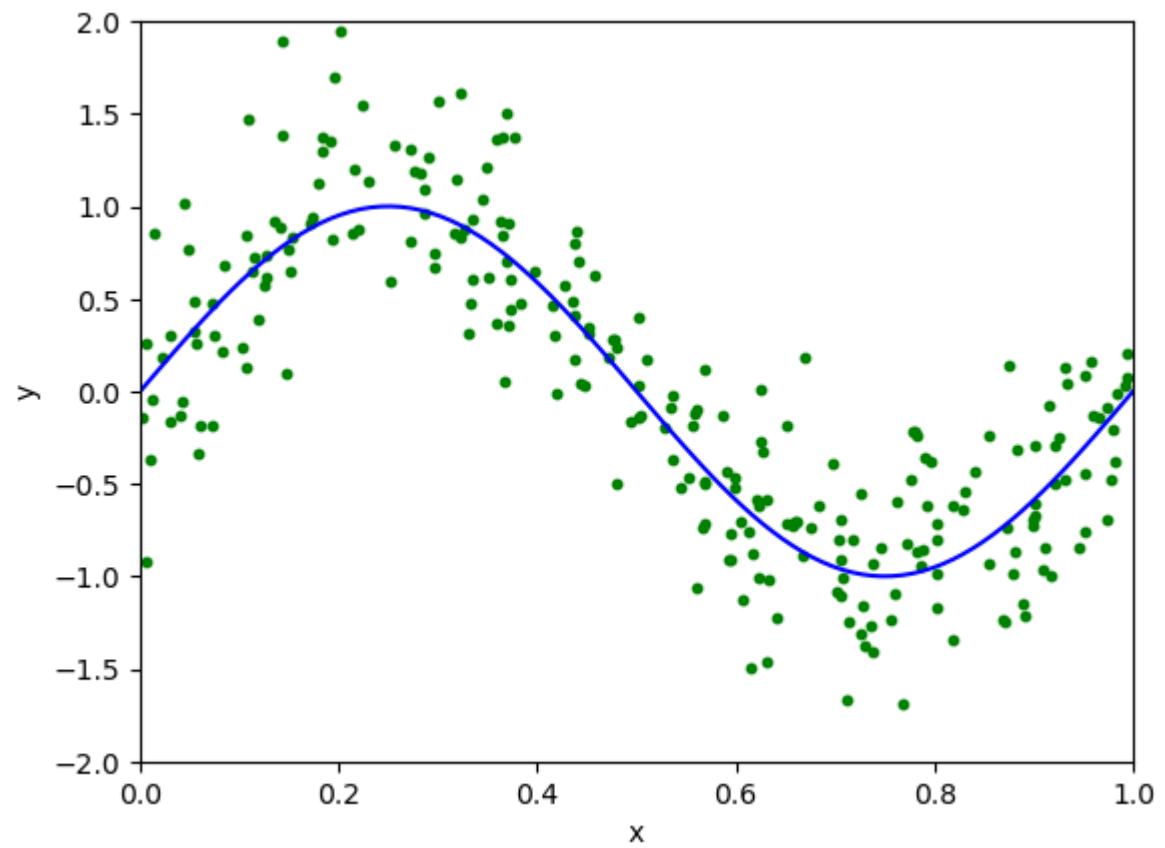
4-3 | 3. ノイズを含むデータの生成

```
# sin(2πx)を関数として定義
def f(x):
    return np.sin(2 * np.pi * x)

# 亂数シードの固定
np.random.seed(1234)

L = 10 # データを生成する回数
N = 25 # データ生成1回あたりのサイズ
n_total = N*L # データサイズの総計

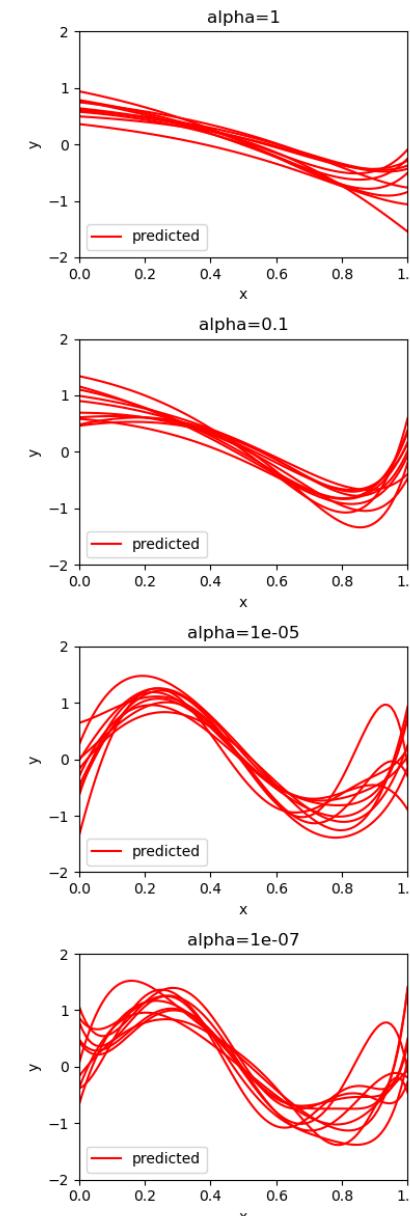
# x座標をランダムに決める
x = np.random.uniform(0, 1, size=n_total).reshape(N, L)
# ノイズ込みでデータを生成
y = f(x) + np.random.normal(scale=0.4, size=n_total).reshape(N, L)
# 理想的なモデル（青色の曲線）
plt.plot(x_plot, f(x_plot), color='b')
# ノイズを含むデータ（緑色の点）
plt.scatter(x, y, s=10, color="g")
# 表示範囲などの調整
plt.ylim((-2, 2))
plt.xlim((0, 1))
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



4-3 | 4. モデルの構築・学習 1/3

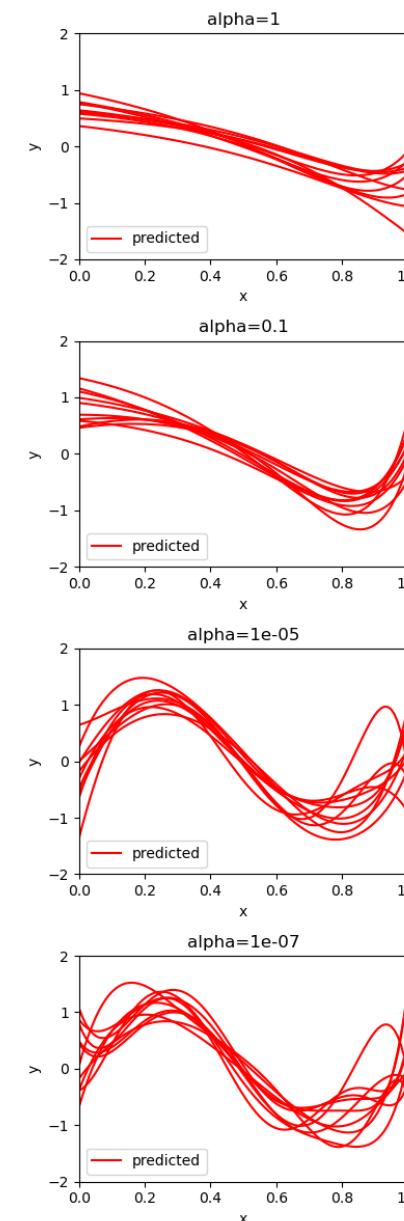
```
# 正則化係数を4つ指定
alphas = [1, 1e-1, 1e-5, 1e-7]
# 多項式回帰の次数を設定(9に固定)
degree = 9
# 4×2の描画エリアを作成
fig, ax_rows = plt.subplots(4, 2, figsize=(8, 12))

# グラフを表示する場所(ax)を指定
for alpha, ax_row in zip(alphas, ax_rows):
    # N行目のグラフ(N個目のalphaを適用)
    ax_left, ax_right = ax_row
    # 左列のグラフ
    # 表示範囲などの設定
    ax_left.set_ylim((-2, 2))
    ax_left.set_xlim((0, 1))
    ax_left.set_ylabel('y')
    ax_left.set_xlabel('x')
    ax_left.set_title("alpha=%s"%alpha)
    # 予測値を保存するための2次元配列
    # (データサイズ,データを生成する回数)
    y_pred = np.zeros((x_plot.shape[0], L))
```



4-3 | 4. モデルの構築・学習 2/3

```
# グラフを表示する場所 (ax) を指定
for alpha, ax_row in zip(alphas, ax_rows):
    for i in range(L):
        # データを1回分取り出す
        x_ = x[:, i].reshape(-1, 1)
        y_ = y[:, i]
        # バイブラインの構築
        model = make_pipeline(
            PolynomialFeatures(degree),
            Ridge(alpha=alpha, random_state=1234)
        )
        # モデルの学習
        model.fit(x_, y_)
        # 予測値を算出
        y_p = model.predict(x_plot.reshape(-1, 1))
        # 予測値を配列に追加
        y_pred[:, i] = y_p
        # 予測値のグラフ (赤色の曲線)
        if i==0:
            ax_left.plot(x_plot, y_p, color='r', label="predicted")
            ax_left.legend(loc="lower left")
        else:
            # 同じalphaの場合は、同じaxに重ねて描画
            ax_left.plot(x_plot, y_p, color='r')
```

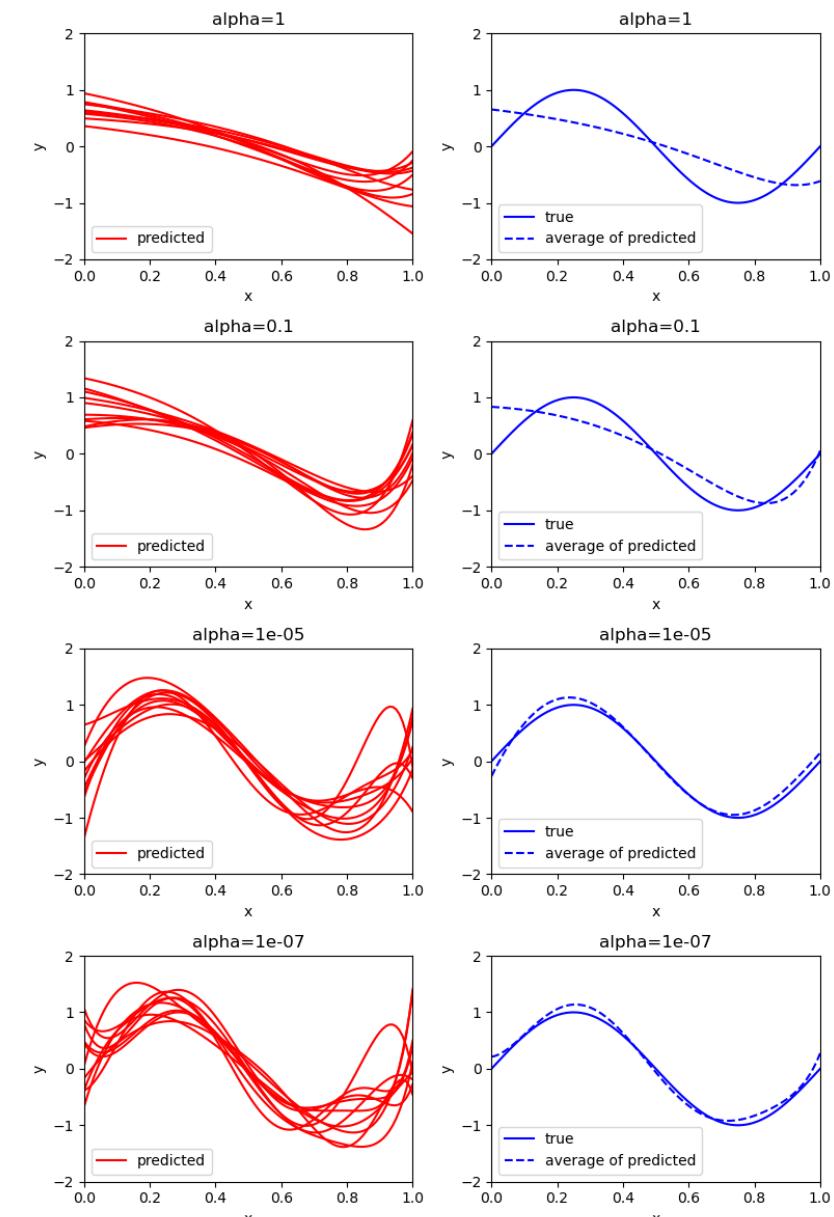


4-3 | 4. モデルの構築・学習 3/3

```
# グラフを表示する場所 (ax) を指定
for alpha, ax_row in zip(alphas, ax_rows):
    # 右列のグラフ
    # 理想的なモデル（青色の曲線）
    ax_right.plot(x_plot, f(x_plot), color="b")
    # L個のモデルから得た予測値の平均（青色の点線）
    ax_right.plot(x_plot, y_pred.mean(axis=1), color="b", ls="--")
    # 表示範囲などの設定
    ax_right.set_ylim((-2, 2))
    ax_right.set_xlim((0, 1))
    ax_right.set_ylabel('y')
    ax_right.set_xlabel('x')
    ax_right.set_title("alpha=%s" % alpha)
    ax_right.legend(["true", "average of predicted"], loc="lower left")

# 各グラフの表示位置を調整
plt.tight_layout()
plt.show()
```

- 予測（グラフではpredicted）がばらついているほど、バリアンスが大きい
- 真の関数(グラフではtrue)と予測の平均値（グラフではaverage of predicted）が重なっているほど、バイアスが小さい
- バリアンスを小さくしようとすると、バイアスが大きくなる
 - これをバイアスとバリアンスのトレードオフという



現場で使える 機械学習・データ分析基礎講座

第 6 章：代表的な前処理

ノートブック解説

■ 代表的な前処理

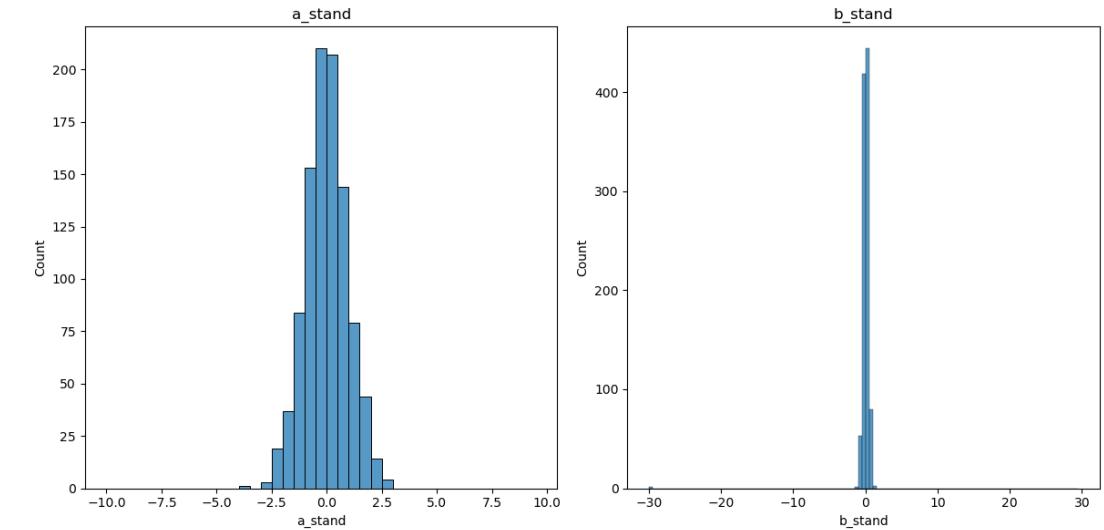
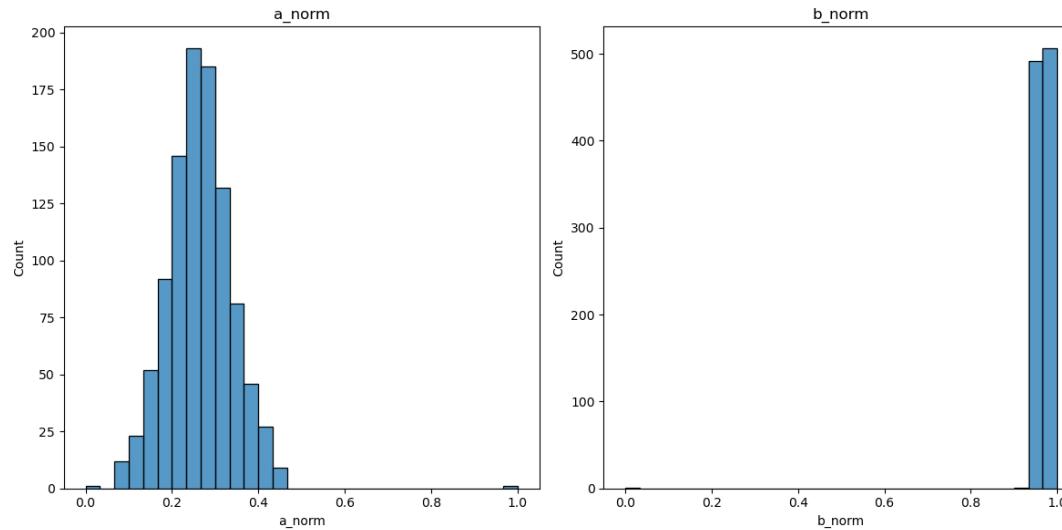
- 6-1_normalization_and_standarization.ipynb (_trainee あり)
- 6-2_decorrelation_and_whitening.ipynb

6-1_normalization_and_standarization

6-1_normalization_and_standarization

■ 内容

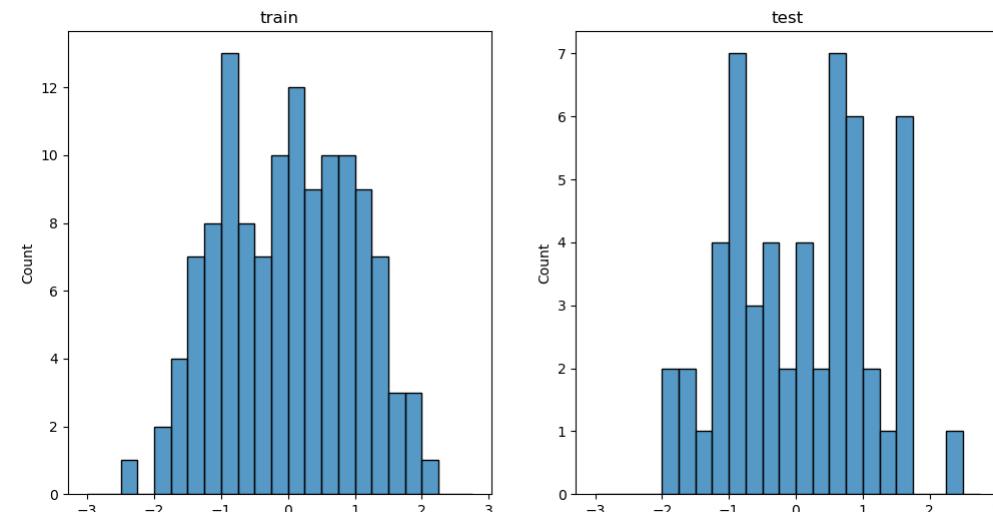
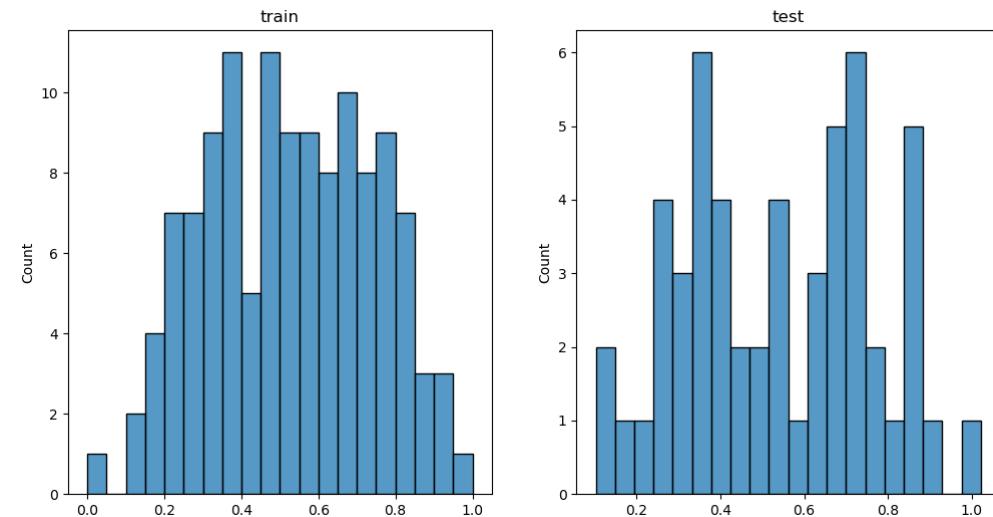
- 正規化および標準化の処理手順を確認
- 処理前後のヒストグラムを表示して、その性質を確認
- 疑似データと、ワインのデータを用いる



6-1_normalization_and_standarization

■ 手順

1. ライブラリの読み込み
2. 正規化・標準化の手順
 1. 手計算で行う場合
 2. scikit-learn を用いる場合
3. 疑似データに対する処理
 1. 疑似データの生成
 2. 異常値の付加
 3. 正規化・標準化の実行
4. ワインデータセットに対する処理
 1. データの読み込み
 2. データの可視化
 3. [演習] 正規化・標準化の実行



6-1 | 1. ライブラリの読み込み

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split

# セルの途中でhead()などを使うための関数
from IPython.display import display

# 正規化を行うためのクラスと、標準化を行うためのクラス
from sklearn.preprocessing import MinMaxScaler, StandardScaler
```

6-1 | 2-1. 手計算で行う場合

```
# データの生成
df = pd.DataFrame({"input": [0, 1, 2, 3, 4, 5]}, dtype=float)

# 正規化の計算
df["normalized"] = (df["input"] - df["input"].min()) / (df["input"].max() - df["input"].min())

# 標準化の計算
df["standardized"] = (df["input"] - df["input"].mean()) / df["input"].std(ddof=0)
# ddof; Delta Degrees of Freedom
# std(ddof=1) を実行すると、不偏分散の平方根が求まる

# 処理前後のデータを表示
display(df)
```

	input	normalized	standardized
0	0.0	0.0	-1.46385
1	1.0	0.2	-0.87831
2	2.0	0.4	-0.29277
3	3.0	0.6	0.29277
4	4.0	0.8	0.87831
5	5.0	1.0	1.46385

6-1 | 2-2. scikit-learn を用いる場合

```
# データの生成
df = pd.DataFrame({"input": [0, 1, 2, 3, 4, 5]}, dtype=float)

# 正規化を行うためのクラス
mms = MinMaxScaler()
# 正規化の実行
mms.fit_transform(df[["input"]].values)
```

```
array([[0. ],
       [0.2],
       [0.4],
       [0.6],
       [0.8],
       [1. ]])
```

```
# データの生成
df = pd.DataFrame({"input": [0, 1, 2, 3, 4, 5]}, dtype=float)

# 標準化を行うためのクラス
stdsc = StandardScaler()
# 標準化の実行
stdsc.fit_transform(df[["input"]].values)
```

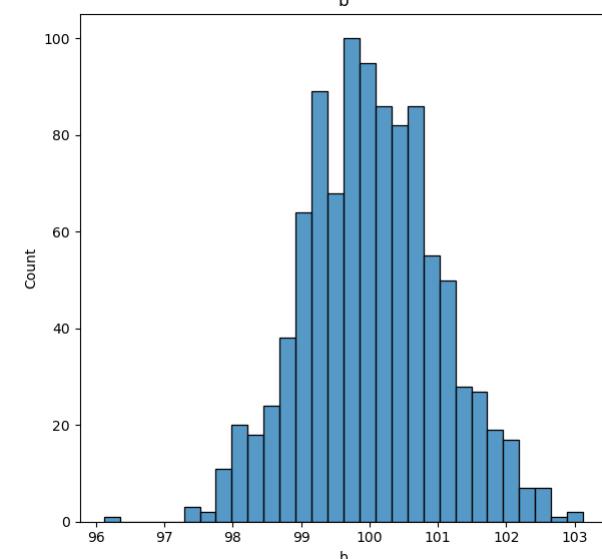
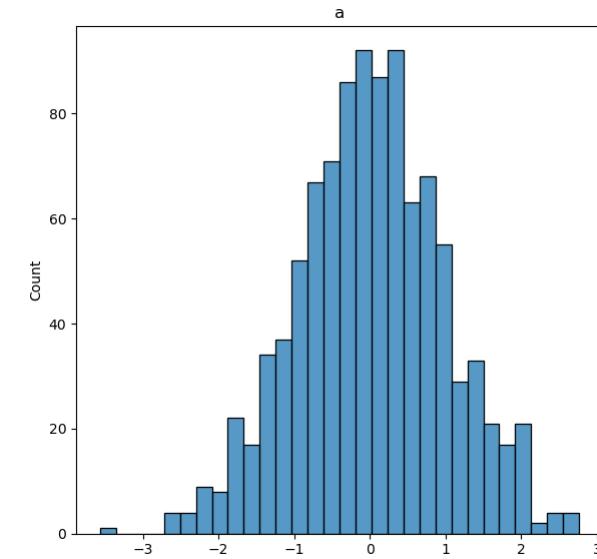
```
array([[-1.46385011],
       [-0.87831007],
       [-0.29277002],
       [ 0.29277002],
       [ 0.87831007],
       [ 1.46385011]])
```

6-1 | 3-1. 疑似データの生成

```
# 亂数シードの固定
np.random.seed(1234)
# 正規分布からデータをサンプリング
df = pd.DataFrame({
    "a": np.random.normal(loc=0, scale=1, size=1000),
    "b": np.random.normal(loc=100, scale=1, size=1000)
})
# データフレームの最初の10行を表示
display(df.head(10))

# 描画エリアの作成
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
# ヒストグラムを描画
sns.histplot(df['a'], kde=False, bins=30, ax=axes[0])
sns.histplot(df['b'], kde=False, bins=30, ax=axes[1])
# グラフタイトルの設定
axes[0].set_title("a")
axes[1].set_title("b")
# 表示位置の調整
plt.tight_layout()
plt.show()
```

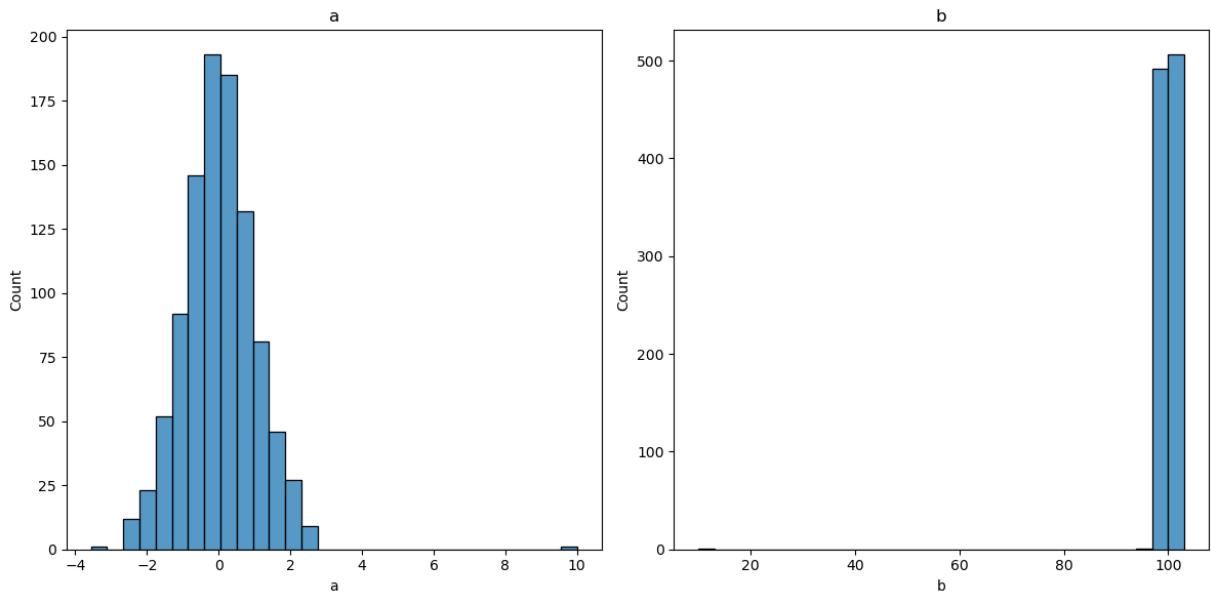
	a	b
0	0.471435	98.417792
1	-1.190976	98.379810
2	1.432707	100.046562
3	-0.312652	98.320171
4	-0.720589	101.395892
5	0.887163	99.155029
6	0.859588	100.814007
7	-0.636524	99.950258
8	0.015696	100.534247
9	-2.242685	99.192991



6-1 | 3-2. 異常値の付加

```
# 異常値を混ぜる
df.loc[0,"a"] = 10
df.loc[0,"b"] = 10

# 描画エリアの作成
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
# ヒストグラムを描画
sns.histplot(df['a'], kde=False, bins=30, ax=axes[0])
sns.histplot(df['b'], kde=False, bins=30, ax=axes[1])
# グラフタイトルの設定
axes[0].set_title("a")
axes[1].set_title("b")
# 表示位置の調整
plt.tight_layout()
plt.show()
```



6-1 | 3-3. 正規化・標準化の実行

```
# 正規化
mms = MinMaxScaler()
df["a_norm"] = mms.fit_transform(df[["a"]].values)
df["b_norm"] = mms.fit_transform(df[["b"]].values)

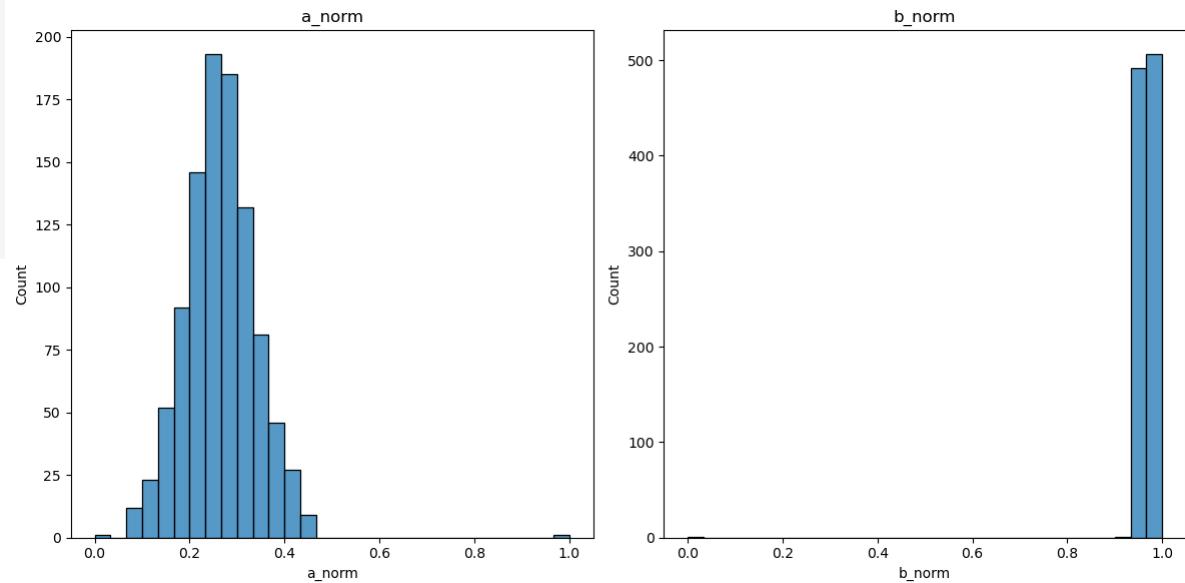
# 標準化
stdsc = StandardScaler()
df["a_stand"] = stdsc.fit_transform(df[["a"]].values)
df["b_stand"] = stdsc.fit_transform(df[["b"]].values)

# 処理前後のデータを表示
display(df.head())
```

	a	b	a_norm	b_norm	a_stand	b_stand
0	10.000000	10.000000	1.000000	0.000000	9.751794	-29.853237
1	-1.190976	98.379810	0.174921	0.949038	-1.189062	-0.522216
2	1.432707	100.046562	0.368358	0.966936	1.375981	0.030937
3	-0.312652	98.320171	0.239677	0.948398	-0.330368	-0.542009
4	-0.720589	101.395892	0.209601	0.981426	-0.729188	0.478745

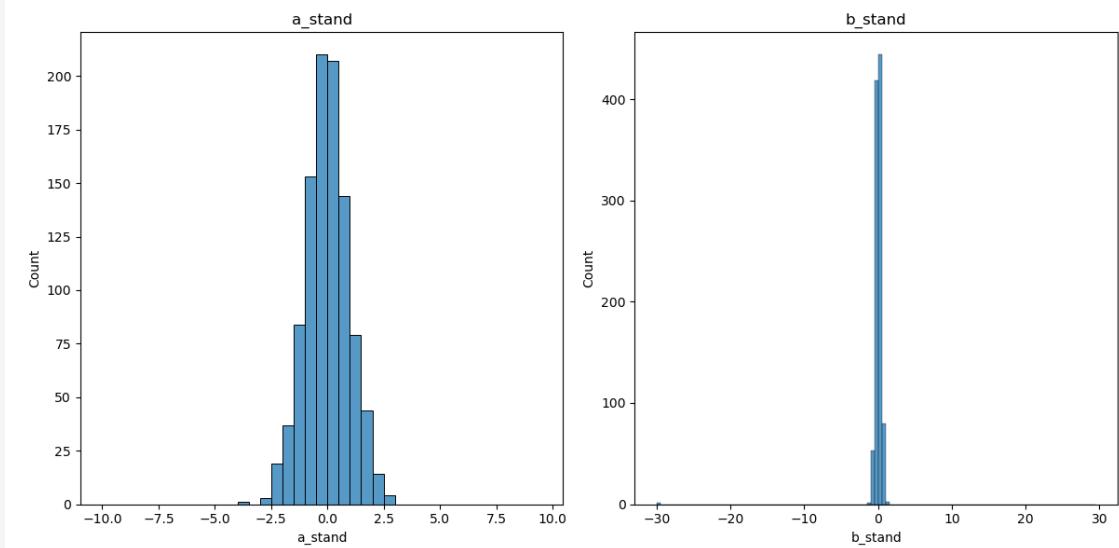
6-1 | 3-3. 正規化・標準化の実行

```
# 正規化後のヒストグラム
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
sns.histplot(df['a_norm'], kde=False, bins=30, ax=axes[0])
sns.histplot(df['b_norm'], kde=False, bins=30, ax=axes[1])
axes[0].set_title("a_norm")
axes[1].set_title("b_norm")
print("正規化の結果")
plt.tight_layout()
plt.show()
```



6-1 | 3-3. 正規化・標準化の実行

```
# 標準化後のヒストグラム
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
# 最も大きい絶対値を取り出す
max_abs_a = np.ceil(np.abs(df["a_stand"]).max())
max_abs_b = np.ceil(np.abs(df["b_stand"]).max())
# 0が中心となるようにbinを設定
bins_a = np.arange(-max_abs_a, max_abs_a, 0.5)
bins_b = np.arange(-max_abs_b, max_abs_b, 0.5)
# ヒストグラムの表示
sns.histplot(df['a_stand'], kde=False, bins=bins_a, ax=axes[0])
sns.histplot(df['b_stand'], kde=False, bins=bins_b, ax=axes[1])
axes[0].set_title("a_stand")
axes[1].set_title("b_stand")
print("標準化の結果")
plt.tight_layout()
plt.show()
```



6-1 | 4-1. データの読み込み

```
# CSVファイルの読み込み
df_wine = pd.read_csv("../..\\1_data\\ch6\\wine.csv", index_col=[0])
print("")
print("データセットの頭出し")
display(df_wine.head())
```

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

6-1 | 4-1. データの読み込み

```
print("")  
print("目的変数（クラスラベル）の内訳")  
display(df_wine.groupby(["Class label"])["Class label"].count())  
  
print("")  
print("説明変数の要約")  
display(df_wine.iloc[:,1:].describe())
```

目的変数（クラスラベル）の内訳

Class label

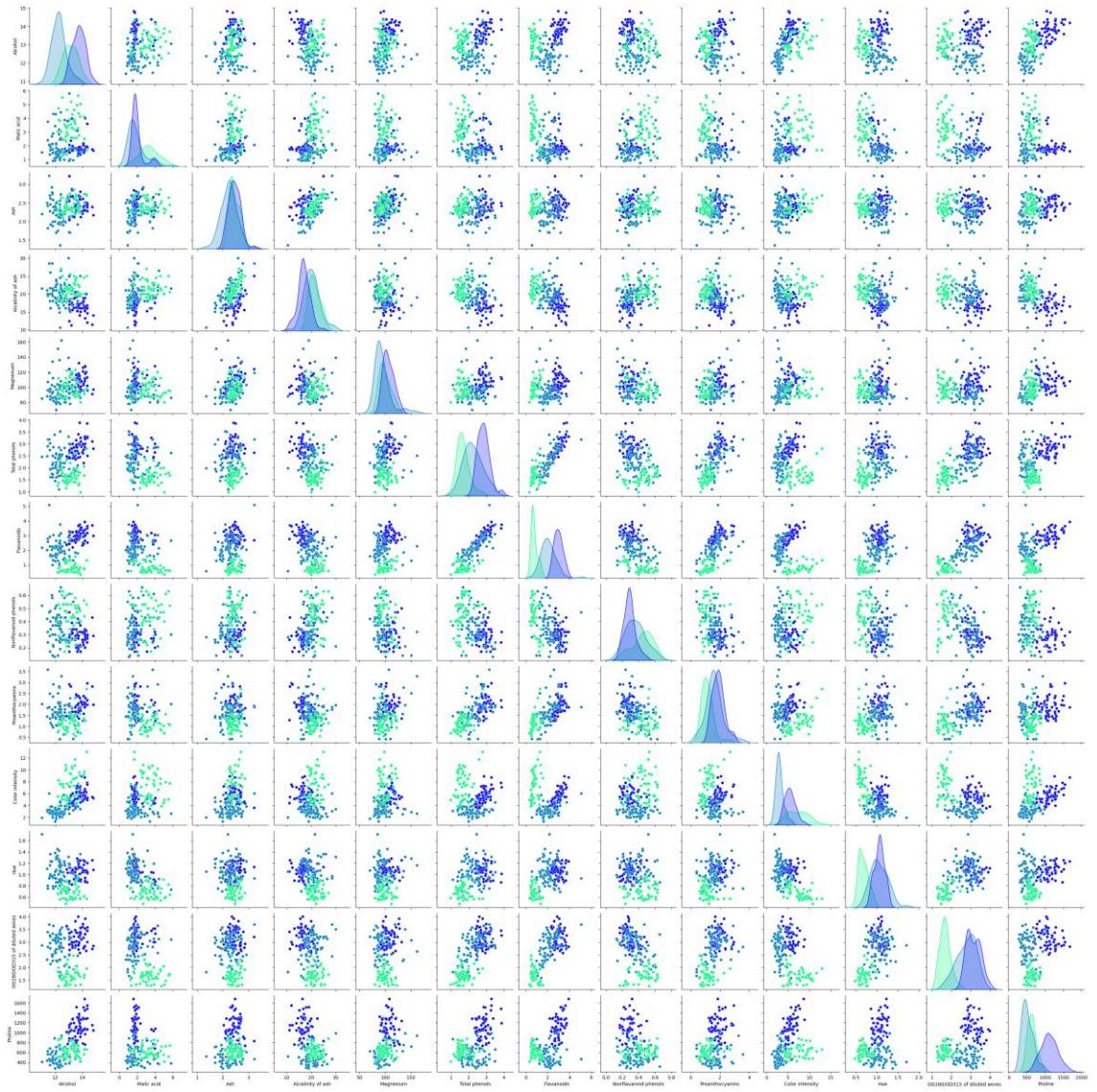
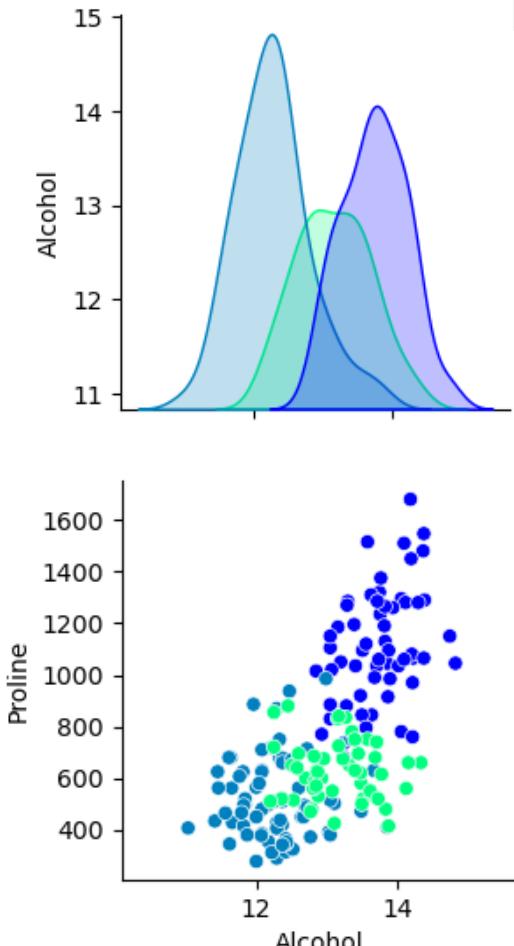
1	59
2	71
3	48

Name: Class label, dtype: int64

	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
count	178.000000	178.000000	178.000000	178.000000	178.000000	178.000000	178.000000	178.000000	178.000000	178.000000	178.000000	178.000000	178.000000
mean	13.000618	2.336348	2.366517	19.494944	99.741573	2.295112	2.029270	0.361854	1.590899	5.058090	0.957449	2.611685	746.893258
std	0.811827	1.117146	0.274344	3.339564	14.282484	0.625851	0.998859	0.124453	0.572359	2.318286	0.228572	0.709990	314.907474
min	11.030000	0.740000	1.360000	10.600000	70.000000	0.980000	0.340000	0.130000	0.410000	1.280000	0.480000	1.270000	278.000000
25%	12.362500	1.602500	2.210000	17.200000	88.000000	1.742500	1.205000	0.270000	1.250000	3.220000	0.782500	1.937500	500.500000
50%	13.050000	1.865000	2.360000	19.500000	98.000000	2.355000	2.135000	0.340000	1.555000	4.690000	0.965000	2.780000	673.500000
75%	13.677500	3.082500	2.557500	21.500000	107.000000	2.800000	2.875000	0.437500	1.950000	6.200000	1.120000	3.170000	985.000000
max	14.830000	5.800000	3.230000	30.000000	162.000000	3.880000	5.080000	0.660000	3.580000	13.000000	1.710000	4.000000	1680.000000

6-1 | 4-2-1. 散布図行列の表示

```
# 変数の数が多いため、かなり時間がかかる  
# クラスラベルで色分け  
sns.pairplot(df_wine, hue="Class label", palette="winter")  
plt.show()
```



6-1 | 4-2-2. データの分割

```
# 説明変数
X = df_wine.iloc[:,1: ].values
# 目的変数
y = df_wine["Class label"].values

# 学習用とテスト用に分割
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

print("X_train:", X_train.shape)
print("y_train:", y_train.shape)
print("X_test:", X_test.shape)
print("y_test:", y_test.shape)
```

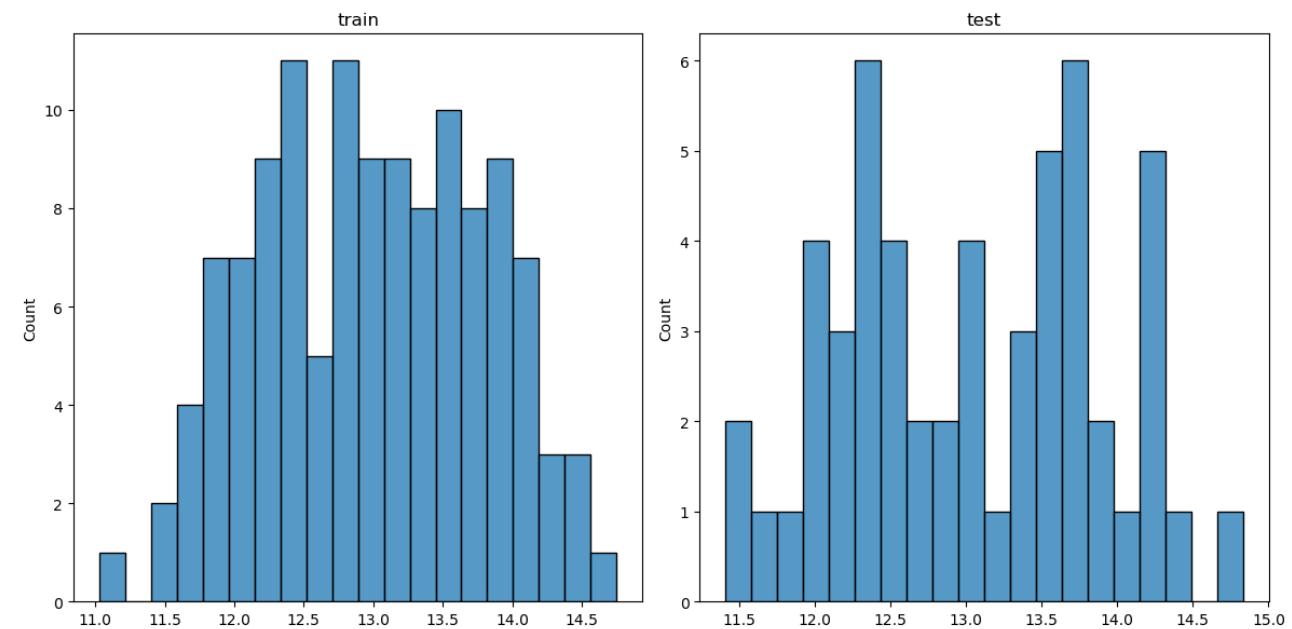
```
X_train: (124, 13)
y_train: (124,)
X_test: (54, 13)
y_test: (54,)
```

6-1 | 4-2-3. ヒストグラムの表示

```
# 描画エリアの作成
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# 学習用データのヒストグラム
sns.histplot(X_train[:, 0], bins=20, ax=axes[0])
# テスト用データのヒストグラム
sns.histplot(X_test[:, 0], bins=20, ax=axes[1])

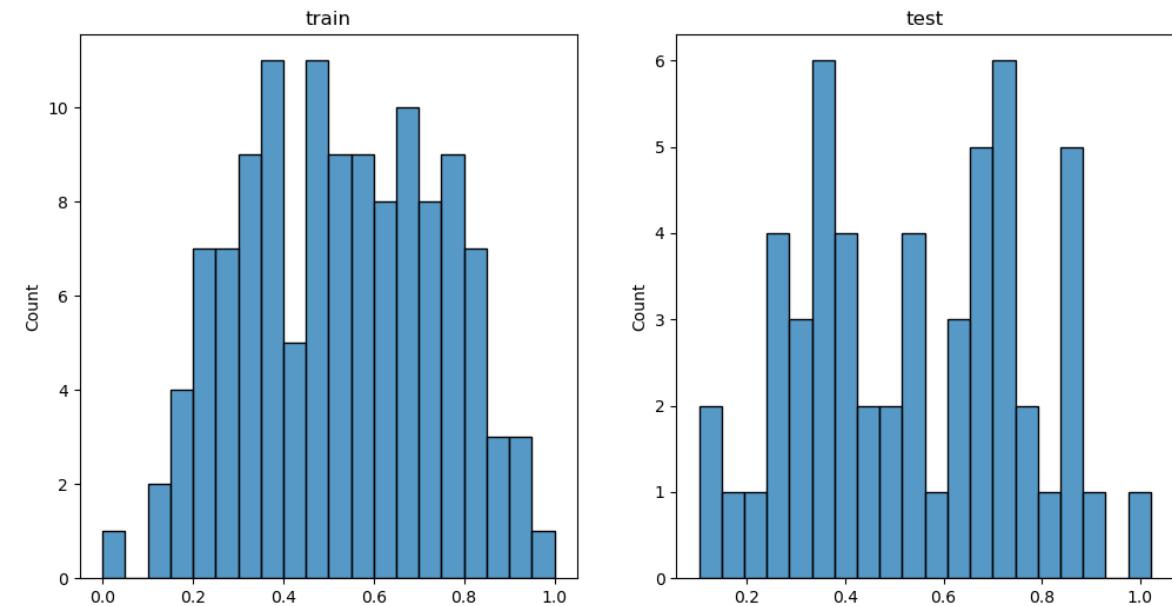
# グラフタイトルの設定
axes[0].set_title("train")
axes[1].set_title("test")
print("加工前のデータ")
plt.tight_layout()
plt.show()
```



6-1 | 4-3. [演習] 正規化・標準化の実行

```
# 学習用データの正規化
normsc = MinMaxScaler()
X_train_norm = normsc.fit_transform(X_train)

# テスト用データの正規化
X_test_norm = normsc.transform(X_test)
# 描画エリアの作成
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
# 1つ目の特徴に関するヒストグラム
sns.histplot(X_train_norm[:, 0], bins=20, ax=axes[0])
sns.histplot(X_test_norm[:, 0], bins=20, ax=axes[1])
print("正規化後のデータ")
# グラフタイトルの設定
axes[0].set_title("train")
axes[1].set_title("test")
plt.show()
```

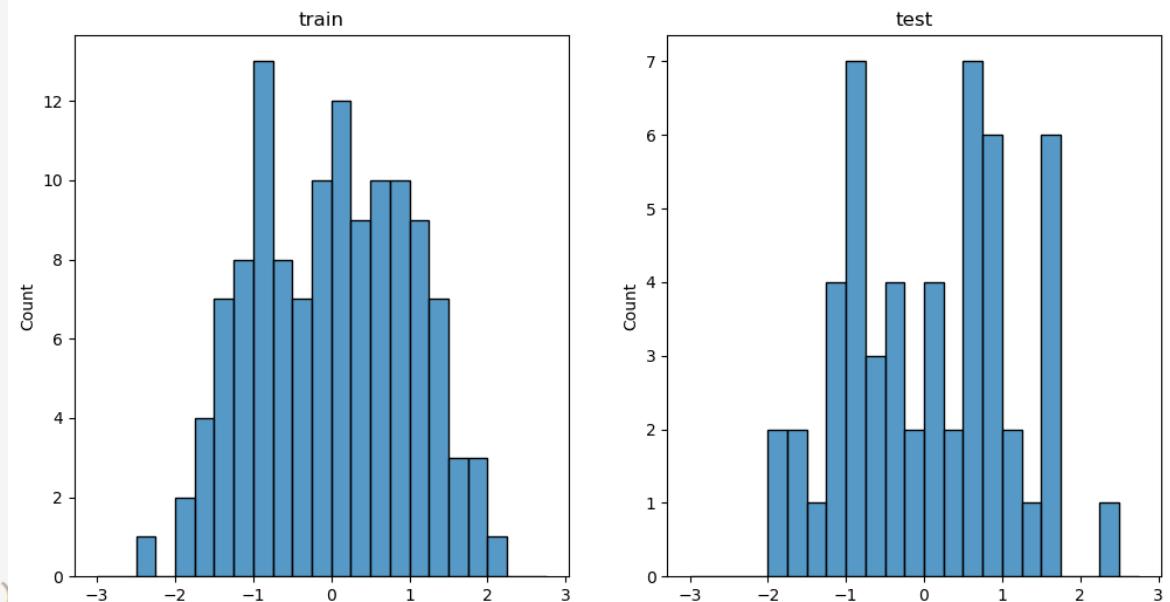


6-1 | 4-3. [演習] 正規化・標準化の実行

```
# 学習用データの標準化
stdsc = StandardScaler()
X_train_stand = stdsc.fit_transform(X_train)

# テスト用データの標準化
X_test_stand = stdsc.transform(X_test)

# 描画エリアの作成
fig, axes = plt.subplots(1, 2, figsize=(12, 6))
# 最も大きい絶対値を取り出す
max_abs_train = np.ceil(np.abs(X_train_stand[:, 0]).max())
max_abs_test = np.ceil(np.abs(X_test_stand[:, 0]).max())
# 0が中心となるようにbinを設定
bins_train = np.arange(-max_abs_train, max_abs_train, 0.25)
bins_test = np.arange(-max_abs_test, max_abs_test, 0.25)
# 1つ目の特徴に関するヒストグラム
sns.histplot(X_train_stand[:, 0], bins=bins_train, ax=axes[0])
sns.histplot(X_test_stand[:, 0], bins=bins_test, ax=axes[1])
print("標準化後のデータ")
# グラフタイトルの設定
axes[0].set_title("train")
axes[1].set_title("test")
plt.show()
```



6-2_decorrelation_and_whitening

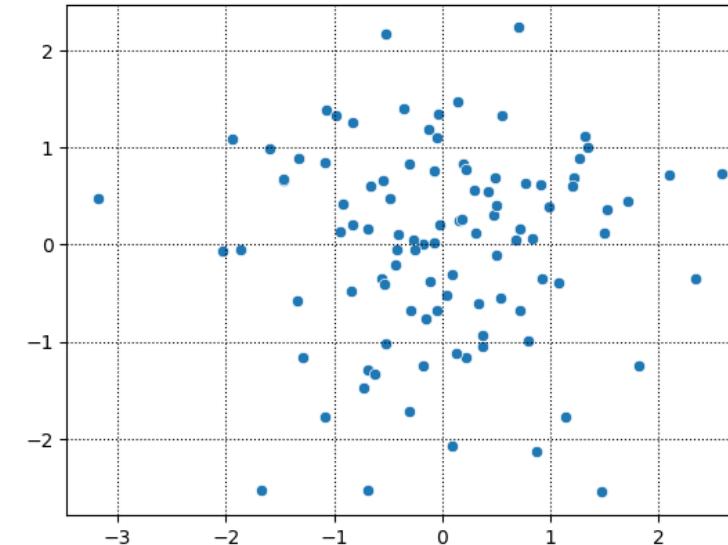
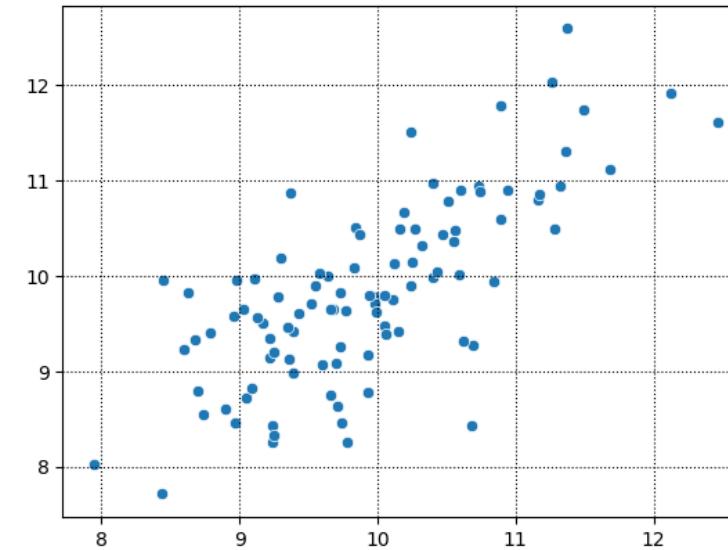
6-2_decorrelation_and_whitening

■ 内容

- 無相関化および白色化の処理手順を確認

■ 手順

- ライブラリの読み込み
- 無相関化
 - 疑似データの生成
 - PCA() を用いる場合
 - 手計算で行う場合
 - 処理結果の表示
- 白色化



6-2 | 1. ライブラリの読み込み

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# 標準化を行うためのクラス
from sklearn.preprocessing import StandardScaler

# 主成分分析を行うためのクラス
from sklearn.decomposition import PCA
```

6-2 | 2-1. 疑似データの生成

```
# 亂数シードの固定
np.random.seed(1234)

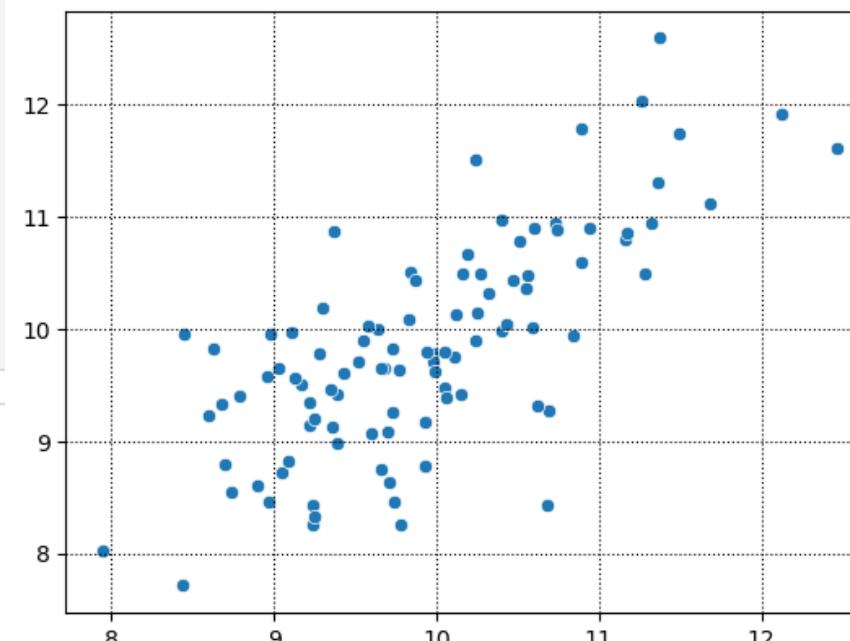
# 多変量正規分布からサンプリング
# 平均、分散共分散行列、データサイズを指定
data1 = np.random.multivariate_normal((10, 10), [[1, 0.8], [0.8, 1]], 100)

# 相関係数を計算
print('相関係数: {:.3f}'.format(np.corrcoef(data1[:, 0], data1[:, 1])[0,1]))

# グリッド線の表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')

# 散布図の表示
sns.scatterplot(x=data1[:, 0], y=data1[:, 1], marker='o')
plt.show()
```

相関係数: 0.728



6-2 | 2-2. PCA() を用いる場合

```
# 主成分分析のクラスをインスタンス化
# 主成分の数を、データ次元数と同じに設定
pca = PCA(n_components=data1.shape[1])

# 主成分分析を実行し、データを無相関化
data1_decorr = pca.fit_transform(data1)
```

6-2 | 2-3. 手計算で行う場合

```
# 分散共分散行列を求める
cov = np.cov(data1, rowvar=0)

# 分散共分散行列を固有値分解
# 固有ベクトルを取り出す
_, S = np.linalg.eig(cov)

# 固有ベクトルとデータの積をとる（無相関化）
data1_decorr = np.dot(S.T, data1.T).T
```

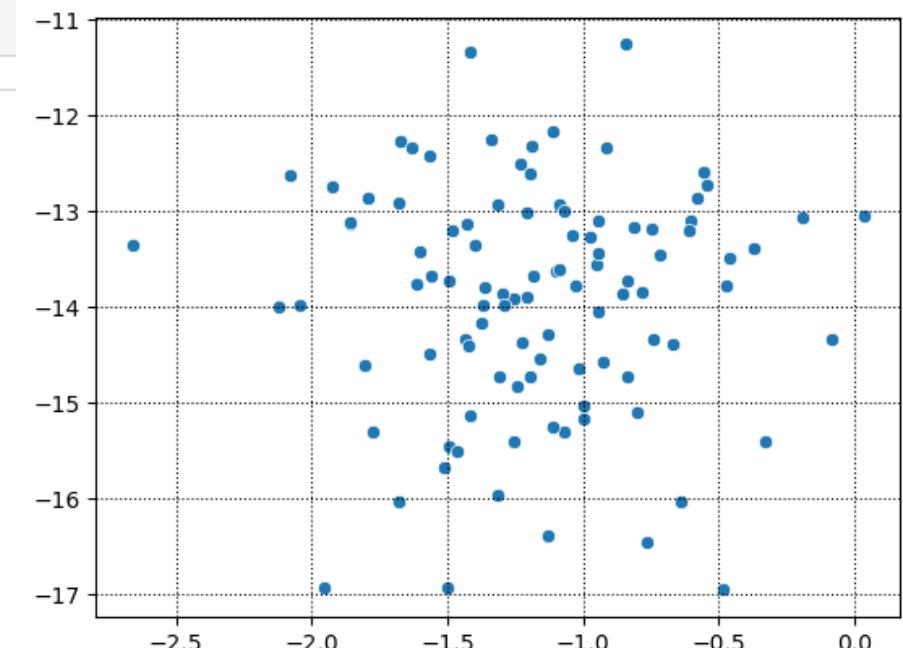
6-2 | 2-4. 処理結果の表示

```
# 相関係数を計算
print('相関係数: {:.3f}'.format(np.corrcoef(data1_decorr[:, 0], data1_decorr[:, 1])[0,1]))

# グリッド線の表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')

# 散布図の表示
sns.scatterplot(x=data1_decorr[:, 0], y=data1_decorr[:, 1], marker='o')
plt.show()
```

相関係数: -0.000



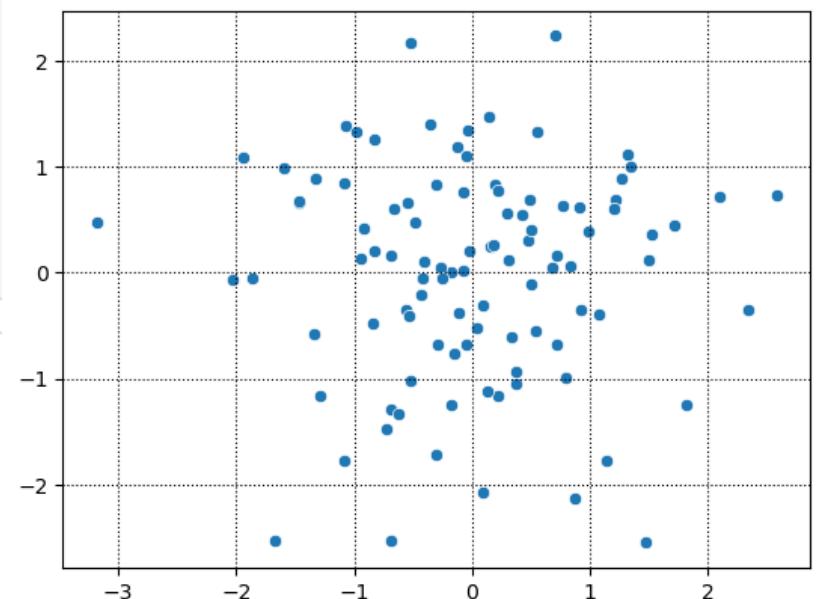
6-2 | 3. 白色化

```
# 標準化を行うためのクラス
stdsc = StandardScaler()
# データの平均と標準偏差を計算
stdsc.fit(data1_decorr)

# 無相関化したデータに、さらに標準化を実施
data1_whitening = stdsc.transform(data1_decorr)

# 相関係数を計算
print('相関係数: {:.3f}'.format(np.corrcoef(data1_whitening[:, 0], data1_whitening[:, 1])[0,1]))
# グリッド線の表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')
# 散布図の表示
sns.scatterplot(x=data1_whitening[:, 0], y=data1_whitening[:, 1], marker='o')
plt.show()
```

相関係数: -0.000



現場で使える 機械学習・データ分析基礎講座

第 7 章：特徴選択

ノートブック解説

■ 特徴選択

- 7-1_feature_selection_filter.ipynb
- 7-2_feature_selection_wrapper.ipynb
- 7-3_feature_selection_embedded.ipynb

7-1_feature_selection_filter

7-1_feature_selection_filter

■ 内容

- 相関係数に基づいて特徴の数を削減
- 特徴選択の前後でモデルの性能を比較

■ 手順

1. ライブラリの読み込み
2. 疑似データの作成
3. データの可視化
4. 特徴選択なしで学習
5. フィルタ法の実施
 1. 特徴の削除
 2. 特徴を削除した上で学習



7-1 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# 線形回帰モデル
from sklearn.linear_model import LinearRegression
# 回帰問題の評価指標
from sklearn.metrics import mean_squared_error, mean_absolute_error, mean_absolute_percentage_error
```

7-1 | 2. 疑似データの作成

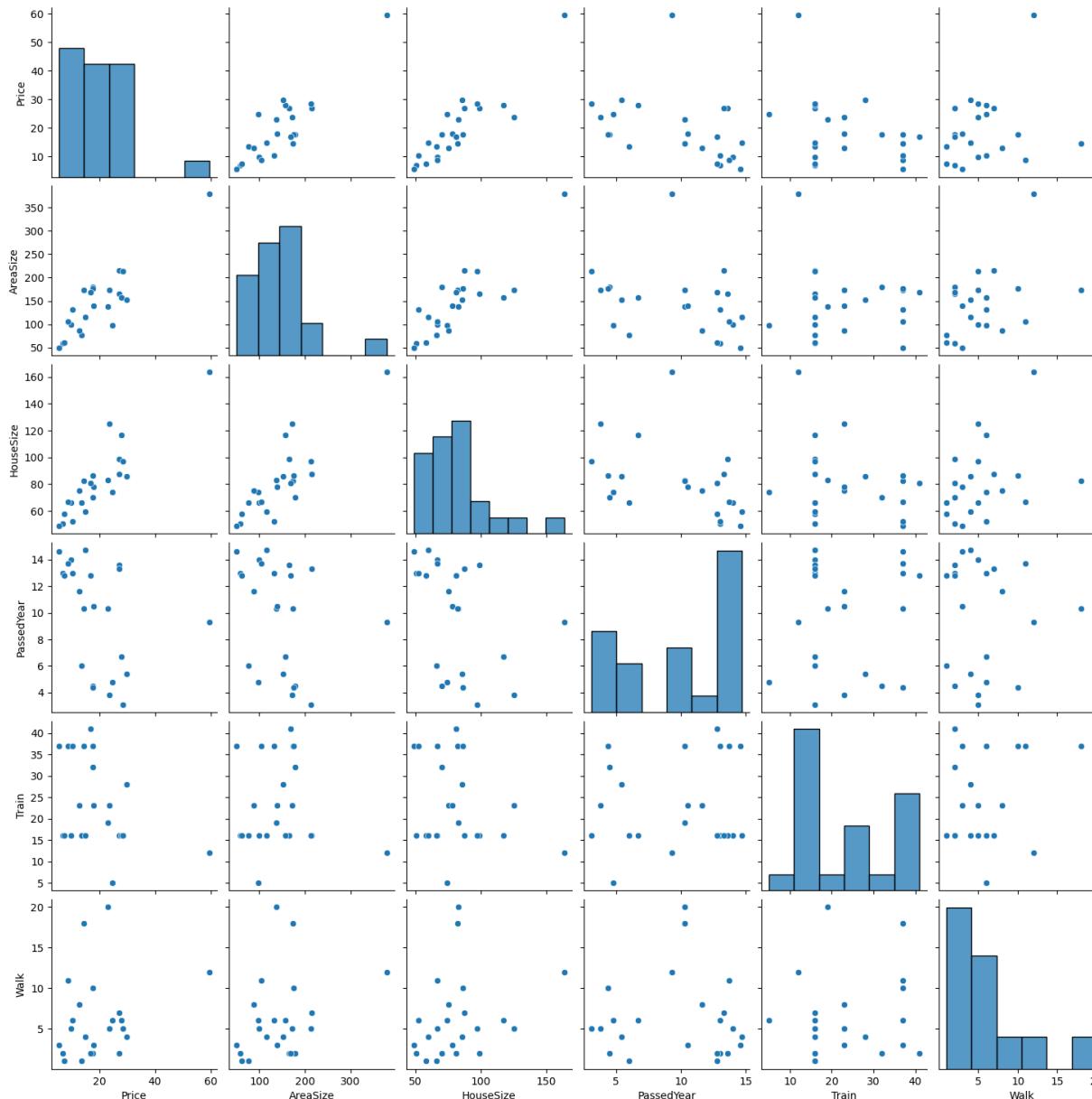
```
df_house = pd.DataFrame({  
    "Price": [24.8, 59.5, 7, 7.5, 9.8, 13.5, 14.9, 27, 27, 28, 28.5, 23, 12.9, 18, 23.7, 29.8, 17.8, 5.5, 8.7, 10.3, 14.5, 17.6, 16.8],  
    "AreaSize": [98.4, 379.8, 58.6, 61.5, 99.6, 76.2, 115.7, 165.2, 215.2, 157.8, 212.9, 137.8, 87.2, 139.6, 172.6, 151.9, 179.5, 50, 105, 132, 174, 176, 168.7],  
    "HouseSize": [74.2, 163.7, 50.5, 58, 66.4, 66.2, 59.6, 98.6, 87.4, 116.9, 96.9, 82.8, 75.1, 77.9, 125, 85.6, 70.1, 48.7, 66.5, 51.9, 82.3, 86.1, 80.8],  
    "PassedYear": [4.8, 9.3, 13, 12.8, 14, 6, 14.7, 13.6, 13.3, 6.7, 3.1, 10.3, 11.6, 10.5, 3.8, 5.4, 4.5, 14.6, 13.7, 13, 10.3, 4.4, 12.8],  
    "Train": [5, 12, 16, 16, 16, 16, 16, 16, 16, 19, 23, 23, 23, 28, 32, 37, 37, 37, 37, 37, 41],  
    "Walk": [6, 12, 2, 1, 5, 1, 4, 2, 7, 6, 5, 20, 8, 3, 5, 4, 2, 3, 11, 6, 18, 10, 2]  
})  
df_house.index.name="id"  
df_house.head()
```

	Price	AreaSize	HouseSize	PassedYear	Train	Walk
id						
0	24.8	98.4	74.2	4.8	5	6
1	59.5	379.8	163.7	9.3	12	12
2	7.0	58.6	50.5	13.0	16	2
3	7.5	61.5	58.0	12.8	16	1
4	9.8	99.6	66.4	14.0	16	5

- 目的変数
 - Price : 値段(百万円)
- 説明変数
 - AreaSize : 土地面積(m²)
 - HouseSize : 家面積(m²)
 - PassedYear : 経過年数(年)
 - Train : 電車での最寄り駅から主要駅までの所要時間(分)
 - Walk : 徒歩での最寄り駅から家までの所要時間(分)

7-1 | 3. データの可視化 1/3

```
# 散布図行列の表示  
sns.pairplot(df_house)  
plt.show()
```



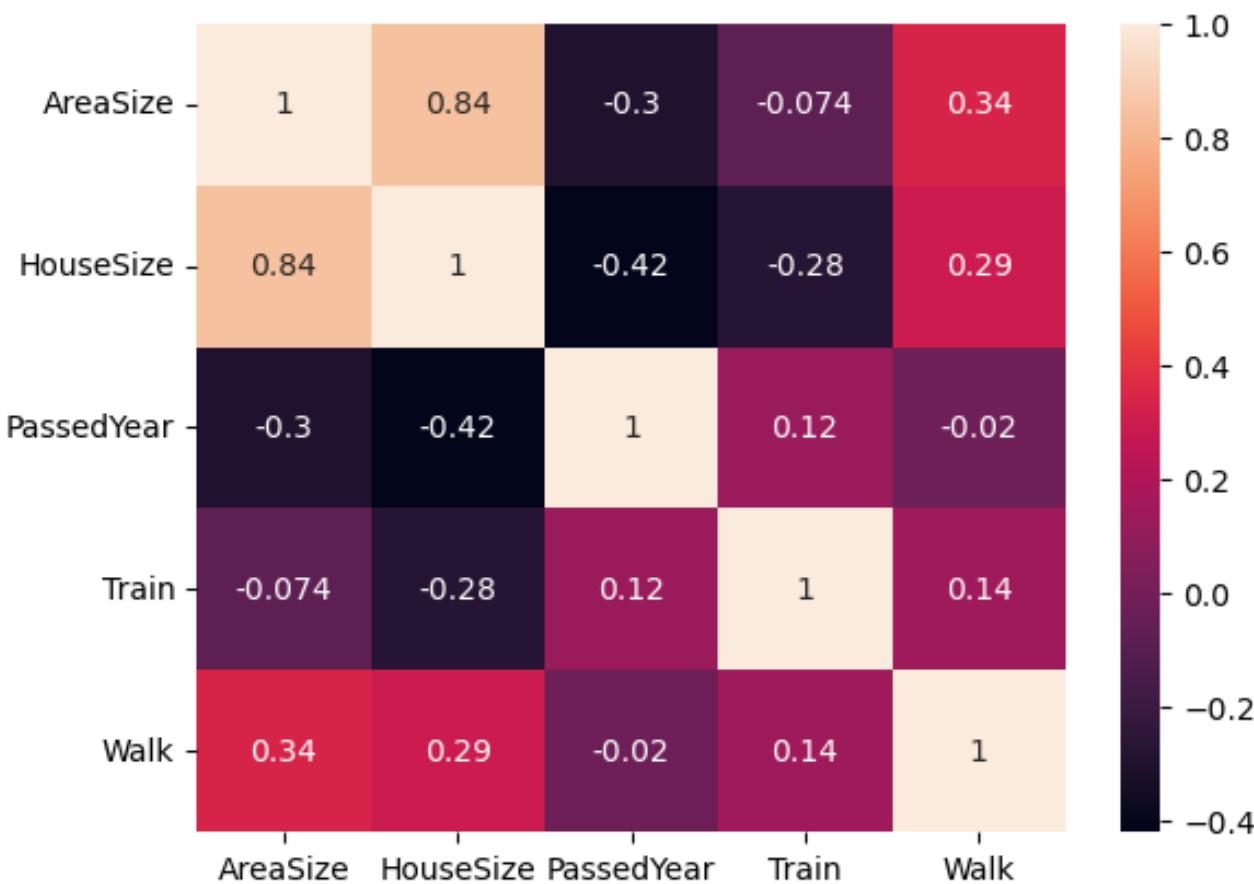
7-1 | 3. データの可視化 2/3

```
# 説明変数の相関係数を確認  
df_house[["AreaSize", "HouseSize", "PassedYear", "Train", "Walk"]].corr()
```

	AreaSize	HouseSize	PassedYear	Train	Walk
AreaSize	1.000000	0.843471	-0.303278	-0.074319	0.336687
HouseSize	0.843471	1.000000	-0.420226	-0.276636	0.291113
PassedYear	-0.303278	-0.420226	1.000000	0.124133	-0.020027
Train	-0.074319	-0.276636	0.124133	1.000000	0.138155
Walk	0.336687	0.291113	-0.020027	0.138155	1.000000

7-1 | 3. データの可視化 3/3

```
# 相関係数をヒートマップにして可視化
# annot=Trueでマスの中に値を表示
sns.heatmap(
    df_house[["AreaSize", "HouseSize", "PassedYear", "Train", "Walk"]].corr(),
    annot=True
)
plt.show()
```



7-1 | 4. 特徴選択なしで学習 1/2

```
# 説明変数  
X = df_house.drop("Price", axis=1).values  
  
# 目的変数  
y = df_house["Price"].values  
  
# 線形回帰モデルの構築  
regr = LinearRegression(fit_intercept=True)  
# モデルの学習  
regr.fit(X, y)
```

▼ LinearRegression

LinearRegression()

7-1 | 4. 特徴選択なしで学習 2/2

```
# 値を予測
y_pred = regr.predict(X)

# MAEを計算
mae = mean_absolute_error(y, y_pred)
print("MAE = %s"%round(mae,3))

# RMSEを計算
rmse = np.sqrt(mean_squared_error(y, y_pred) )
print("RMSE = %s"%round(rmse, 3))

# MAPEを計算
mape = mean_absolute_percentage_error(y, y_pred)
print("MAPE = %s"%round(mape*100,3), "%")
```

MAE = 2.411

RMSE = 3.297

MAPE = 15.998 %

7-1 | 5-1. 特徴の削除

```
# pandasのdrop関数を使って"AreaSize"を削除
# 列方向に削除するときは引数に特徴の名前とaxis=1を指定
selected_X = df_house.drop("AreaSize", axis=1)
display(selected_X.head())
```

	Price	HouseSize	PassedYear	Train	Walk
id					
0	24.8	74.2	4.8	5	6
1	59.5	163.7	9.3	12	12
2	7.0	50.5	13.0	16	2
3	7.5	58.0	12.8	16	1
4	9.8	66.4	14.0	16	5

7-1 | 5-2. 特徴を削除した上で学習

```
# モデルの学習
regr.fit(selected_X, y)

# 値を予測
y_pred = regr.predict(selected_X)

# MAEを計算
mae = mean_absolute_error(y, y_pred)
print("MAE = %s"%round(mae,3) )
# RMSEを計算
rmse = np.sqrt(mean_squared_error(y, y_pred) )
print("RMSE = %s"%round(rmse, 3))
# MAPEを計算
mape = mean_absolute_percentage_error(y, y_pred)
print("MAPE = %s"%round(mape*100,3), "%")
```

```
MAE = 0.0
RMSE = 0.0
MAPE = 0.0 %
```

7-2_feature_selection_wrapper

■ 内容

- ・ステップワイズ法による特徴選択を実施
- ・特徴選択の前後でモデルの性能を比較

■ 手順

1. ライブラリの読み込み
2. データの読み込み
3. 特徴選択なしで学習
4. ステップワイズ法による特徴選択
 1. ステップワイズ法の実行
 2. 特徴選択の結果を確認
 3. 特徴の削除
 4. 特徴を削除した上で学習

```
# 特徴のランキングを表示（1が最も重要な特徴）
print('Feature ranking: \n{}'.format(rfecv.ranking_))
```

```
Feature ranking:
[ 2  1 10  3  4  7  1 11  8  1  9  6  5 ]
```

7-2 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# ワインのデータセットを読み込むためのクラス
from sklearn.datasets import load_wine

# ロジスティック回帰モデル
from sklearn.linear_model import SGDClassifier

# ステップワイズ法を実行するためのクラス
# 再帰的な特徴選択を、交差検証法による評価に基づいて行う
# Recursive feature elimination with cross-validation
from sklearn.feature_selection import RFEcv

# 多クラス分類の評価指標
from sklearn.metrics import classification_report
```

7-2 | 2. データの読み込み

```
# データセットの読み込み
df_X, df_y = load_wine(return_X_y=True, as_frame=True)

# 説明変数を確認
display(df_X.head())
display(df_X.describe())

# 目的変数の内訳を確認
df_y.value_counts()
```

```
target
1    71
0    59
2    48
Name: count, dtype: int64
```

- alcohol : アルコール度数
- malic_acid : リンゴ酸
- ash : 灰分 (かいぶん)
- alcalinity_of_ash : 灰分のアルカリ度
- magnesium : マグネシウム
- total_phenols : 全フェノール含量
- flavanoids : フラボノイド (ポリフェノールの一種)
- nonflavanoid_phenols : 非フラボノイドフェノール
- proanthocyanins : プロアントシアニン (ポリフェノールの一種)
- color_intensity : 色の濃さ
- hue : 色相
- od280/od315_of_diluted_wines : 希釀ワイン溶液のOD280／OD315 (=280nmと315nmの吸光度の比)
- proline : プロリン (アミノ酸の一種)

	alcohol	malic_acid	ash	alcalinity_of_ash	magnesium	total_phenols	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue	od280/od315_of_diluted_wines	proline
0	14.23	1.71	2.43	15.6	127.0	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065.0
1	13.20	1.78	2.14	11.2	100.0	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050.0
2	13.16	2.36	2.67	18.6	101.0	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185.0
3	14.37	1.95	2.50	16.8	113.0	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480.0
4	13.24	2.59	2.87	21.0	118.0	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735.0

7-2 | 3. 特徴選択なしで学習 1/2

```
# 特徴選択用のもの (df_X, df_y) とは別に作成
# 説明変数
X = df_X.values
# 目的変数
y = df_y.values

# 分類問題であるためSGDClassifierを使用
estimator = SGDClassifier(loss='log_loss', max_iter=10000, fit_intercept=True,
                           random_state=1234, tol=1e-3, )

# 特徴選択なし
# モデルの学習
estimator.fit(X, y)
```

```
▼ SGDClassifier
SGDClassifier(loss='log_loss', max_iter=10000, random_state=1234)
```

7-2 | 3. 特徴選択なしで学習 2/3

```
# 予測結果の取得
y_pred = estimator.predict(X)

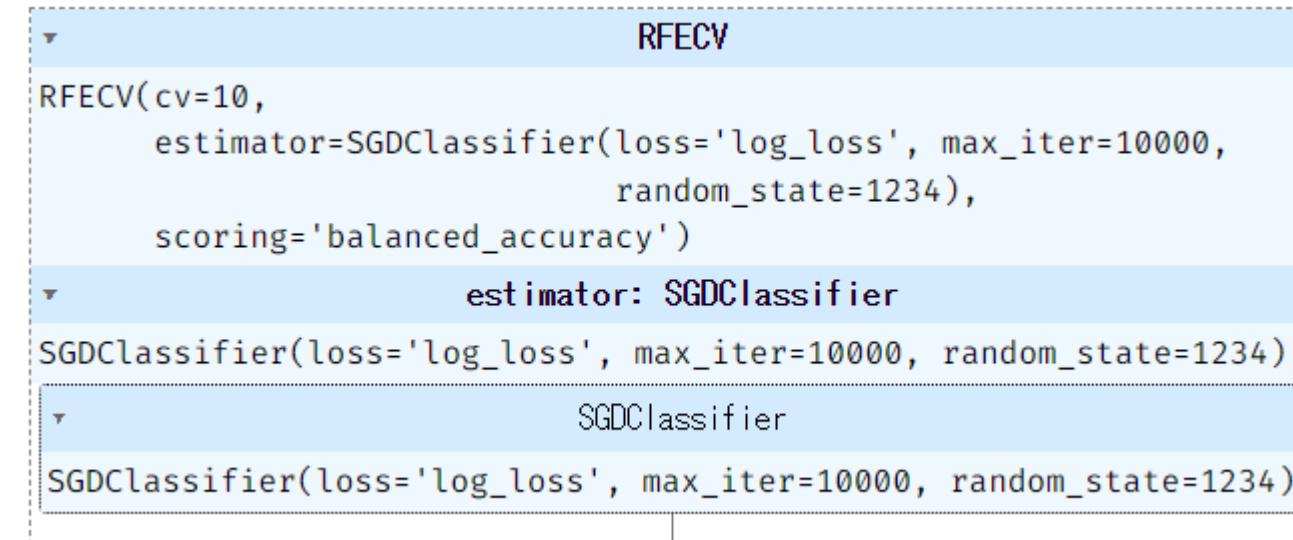
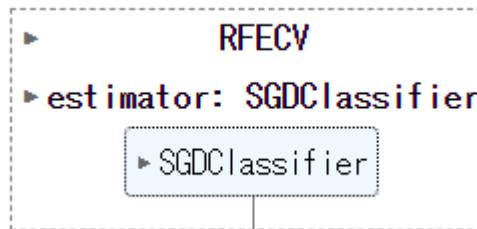
# 訓練性能の評価
scores = classification_report(y, y_pred)
print(scores)
```

	precision	recall	f1-score	support
0	1.00	0.75	0.85	59
1	0.58	0.94	0.72	71
2	0.42	0.17	0.24	48
accuracy			0.67	178
macro avg	0.67	0.62	0.60	178
weighted avg	0.68	0.67	0.63	178

7-2 | 4-1. ステップワイズ法の実行

```
# 交差検証を行いつつ、ステップワイズ法による特徴選択を行う
# cvにはFold (=グループ) の数、scoringには評価指標を指定する
# 今回は多クラス分類問題なのでbalanced_accuracyを評価指標に指定
rfecv = RFECV(estimator, cv=10, scoring='balanced_accuracy')

# fitで特徴選択と学習を実行
rfecv.fit(X, y)
```



7-2 | 4-2. 特徴選択の結果を確認

```
# 特徴のランキングを表示 (1が最も重要な特徴)
print('Feature ranking: \n{}'.format(rfecv.ranking_))
```

```
Feature ranking:
[ 2  1 10  3  4  7  1 11  8  1  9  6  5]
```

```
# rfecv.support_でランク1位以外はFalseとするindexを取得できる
# Trueになっている特徴を使用すれば汎化誤差は最小となる
rfecv.support_
```

```
array([False,  True, False, False, False, False,  True, False, False,
       True, False, False, False])
```

```
# bool型の配列に ~ をつけるとTrueとFalseを反転させることができる
# ここでTrueになっている特徴が、削除してもよい特徴
remove_idx = ~rfecv.support_
remove_idx
```

```
array([ True, False,  True,  True,  True,  True, False,  True,  True,
       False,  True,  True,  True])
```

7-2 | 4-3. 特徴の削除

```
# 削除してもよい特徴の名前を取得する
remove_feature = df_X.columns[remove_idx]
remove_feature
```

```
Index(['alcohol', 'ash', 'alcalinity_of_ash', 'magnesium', 'total_phenols',
       'nonflavanoid_phenols', 'proanthocyanins', 'hue',
       'od280/od315_of_diluted_wines', 'proline'],
      dtype='object')
```

```
# drop関数で特徴を削除
selected_X = df_X.drop(remove_feature, axis=1)
selected_X
```

	malic_acid	flavanoids	color_intensity
0	1.71	3.06	5.64
1	1.78	2.76	4.38
2	2.36	3.24	5.68
3	1.95	3.49	7.80
4	2.59	2.69	4.32
...
173	5.65	0.61	7.70
174	3.91	0.75	7.30
175	4.28	0.69	10.20
176	2.59	0.68	9.30
177	4.10	0.76	9.20

7-2 | 4-4. 特徴を削除した上で学習

```
# モデルの学習
estimator.fit(selected_X, y)

# 予測結果の取得
y_pred = estimator.predict(selected_X)

# 訓練性能の評価
scores = classification_report(y, y_pred)
print(scores)
```

	precision	recall	f1-score	support
0	0.78	0.98	0.87	59
1	0.98	0.73	0.84	71
2	0.94	1.00	0.97	48
accuracy			0.89	178
macro avg	0.90	0.91	0.89	178
weighted avg	0.90	0.89	0.89	178

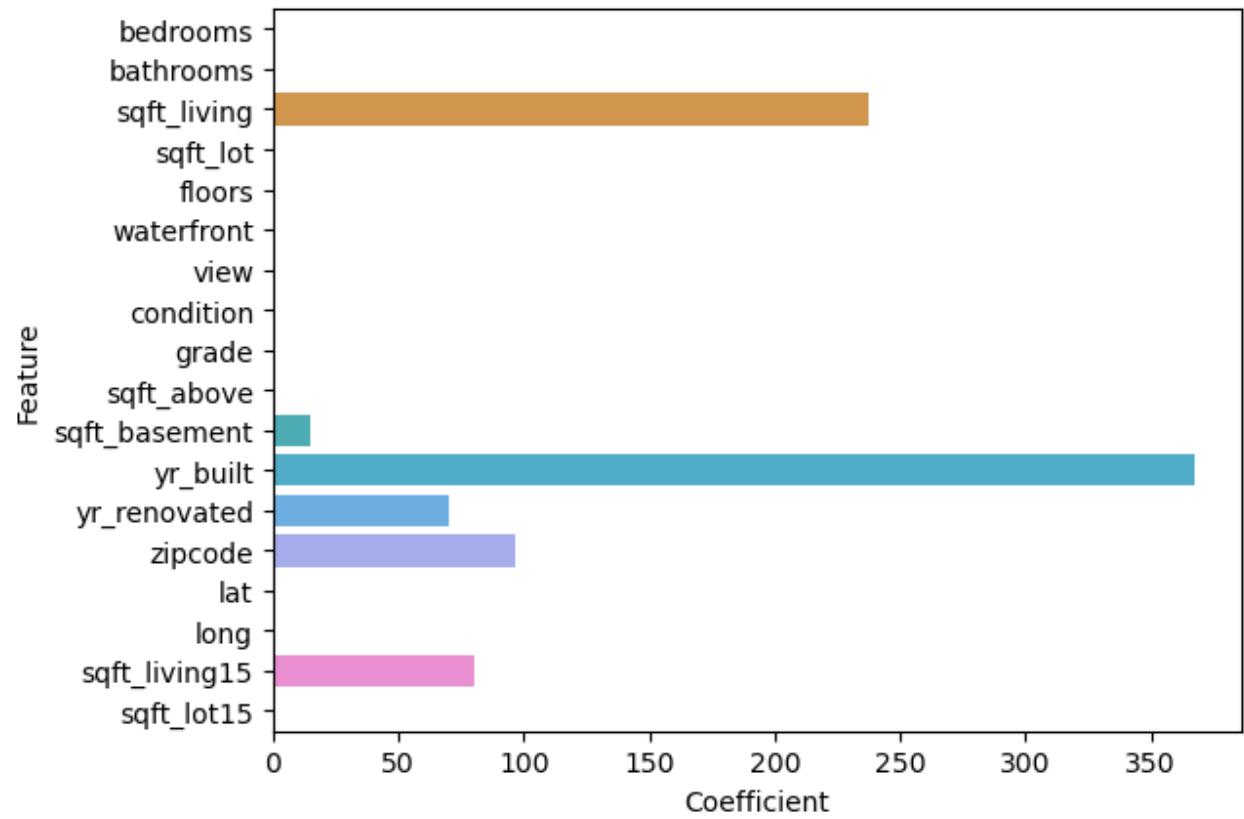
7-3_feature_selection_embedded

■ 内容

- Lasso による特徴選択の効果を確認
- 特徴選択の前後でモデルの性能を比較

■ 手順

1. ライブラリの読み込み
2. データの読み込み
3. 特徴選択なしで学習
4. Lasso による特徴選択
 1. モデルの構築・学習
 2. 特徴の係数を可視化
 3. 特徴選択の結果を確認
 4. 特徴の削除
 5. 特徴を削除した上で学習



7-3 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# 線形回帰モデル
from sklearn.linear_model import LinearRegression
# L1正則化つき線形回帰モデル
from sklearn.linear_model import LassoCV
# 回帰問題の評価指標
from sklearn.metrics import mean_squared_error, mean_absolute_error, mean_absolute_percentage_error

# モデルの情報を用いて特徴選択
from sklearn.feature_selection import SelectFromModel
```

7-3 | 2. データの読み込み

```
# CSVファイルの読み込み
df_house = pd.read_csv("../..../1_data/ch7/kc_house_data.csv").drop(["id", "date"], axis=1)

# 変数の確認
display(df_house.head())
df_house.describe()
```

- | | |
|----------------------------------|------------------------------------|
| ◦ bedrooms: ベッドルームの数 | ◦ sqft_above: 地下室を除いた延床面積 [平方フィート] |
| ◦ bathrooms: バスルームの数 | ◦ sqft_basement: 地下室の面積 [平方フィート] |
| ◦ sqft_living: 延床面積 [平方フィート] | ◦ yr_built: 建築年 |
| ◦ sqft_lot: 敷地面積 [平方フィート] | ◦ yr_renovated: 改装された年 |
| ◦ floors: 階数 (何階建てであるか) | ◦ zipcode: 郵便番号 |
| ◦ waterfront: 海や川、湖などが見える立地か | ◦ lat: 緯度座標 |
| ◦ view: 内見済みか | ◦ long: 経度座標 |
| ◦ condition: 全体的な状態の良し悪し | ◦ sqft_living15: 2015年における延床面積 |
| ◦ grade: King County の等級制度に基づく等級 | ◦ sqft_lot15: 2015年における敷地面積 |

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_basement	yr_built	yr_renovated	zipcode	lat	long	sqft_living15	sqft_lot15
0	221900.0	3	1.00	1180	5650	1.0	0	0	3	7	1180	0	1955	0	98178	47.5112	-122.257	1340	5650
1	538000.0	3	2.25	2570	7242	2.0	0	0	3	7	2170	400	1951	1991	98125	47.7210	-122.319	1690	7639
2	180000.0	2	1.00	770	10000	1.0	0	0	3	6	770	0	1933	0	98028	47.7379	-122.233	2720	8062
3	604000.0	4	3.00	1960	5000	1.0	0	0	5	7	1050	910	1965	0	98136	47.5208	-122.393	1360	5000
4	510000.0	3	2.00	1680	8080	1.0	0	0	3	8	1680	0	1987	0	98074	47.6168	-122.045	1800	7503

7-3 | 3. 特徴選択なしで学習 1/2

```
# 特徴選択前のデータを保持
train_label = df_house["price"]
train_data = df_house.drop("price", axis=1)

# 目的変数
y = train_label.values
# 説明変数
X = train_data.values

# 線形回帰モデルの構築
regr = LinearRegression(fit_intercept=True)
# モデルの学習
regr.fit(X, y)
```

▼ LinearRegression
LinearRegression()

7-3 | 3. 特徴選択なしで学習 2/2

```
# 値を予測
y_pred = regr.predict(X)

# MAEを計算
mae = mean_absolute_error(y, y_pred)
print("MAE = %s"%round(mae,3))

# RMSEを計算
rmse = np.sqrt(mean_squared_error(y, y_pred) )
print("RMSE = %s"%round(rmse, 3))

# MAPEを計算
mape = mean_absolute_percentage_error(y, y_pred)
print("MAPE = %s"%round(mape*100,3), "%")
```

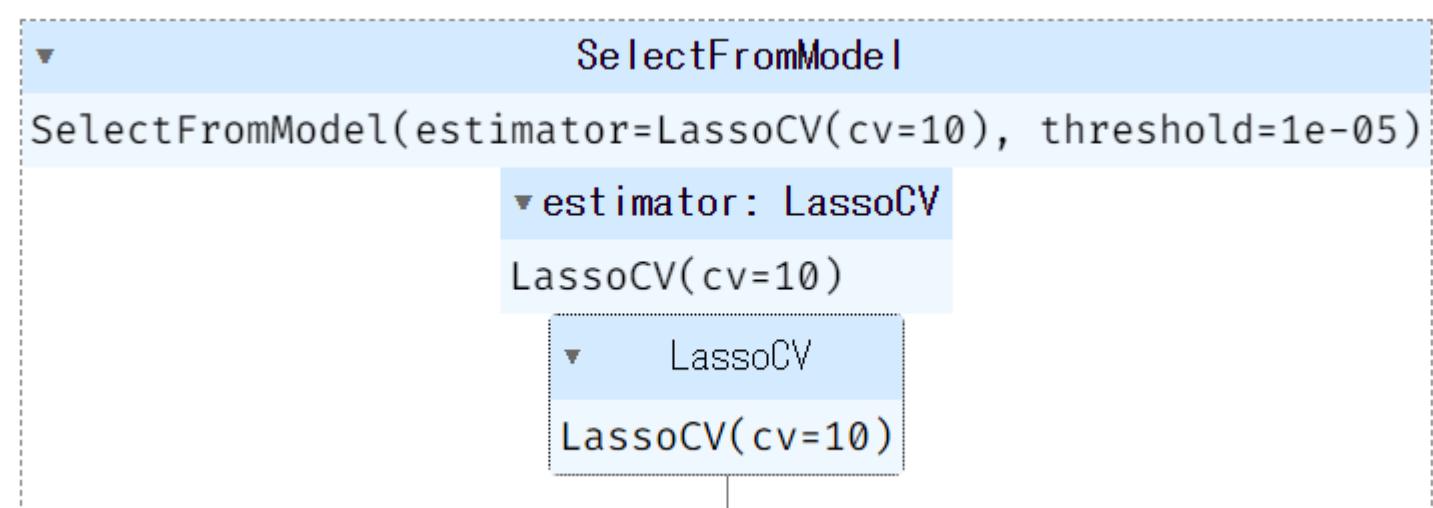
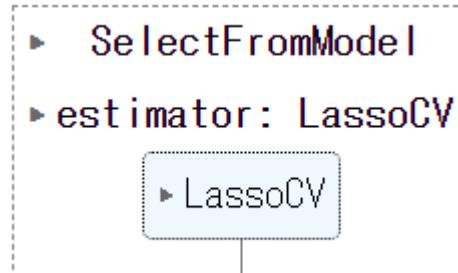
MAE = 125922.646
RMSE = 201163.902
MAPE = 25.58 %

7-3 | 4-1. モデルの構築・学習

```
# estimatorにモデル（LassoCV）をセット
# 正則化の強さを自動で調整しながら、交差検証を行う
estimator = LassoCV(cv=10)

# モデルの情報を使って特徴選択を行う
# 今回は係数が1e-5以下である特徴を削除する
# 係数のしきい値はthresholdで指定する
sfm = SelectFromModel(estimator, threshold=1e-5)

# 学習と特徴選択を実行
sfm.fit(X, y)
```



7-3 | 4-2. 特徴の係数を可視化 1/2

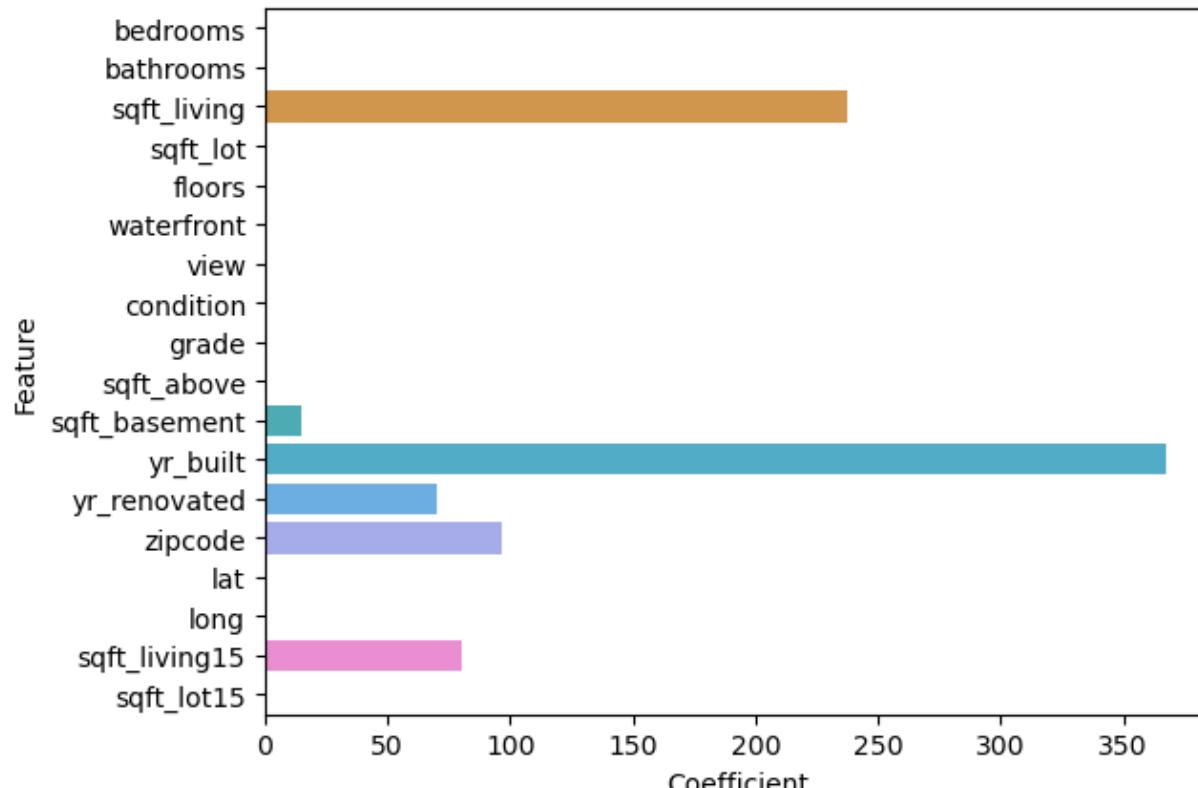
```
# Lassoで得た各特徴の係数の値を確認してみよう
# 係数の絶対値を取得
abs_coef = np.abs(sfm.estimator_.coef_)
abs_coef
```

```
array([0.0000000e+00, 0.0000000e+00, 2.37205685e+02, 7.46650294e-02,
       0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
       0.0000000e+00, 0.0000000e+00, 1.45700417e+01, 3.67698927e+02,
       6.97780917e+01, 9.62998902e+01, 0.0000000e+00, 0.0000000e+00,
       8.05758593e+01, 7.60240333e-01])
```

7-3 | 4-2. 特徴の係数を可視化 2/2

```
# データフレームを作成
df = pd.DataFrame({'Feature': train_data.columns.values, 'Coefficient': abs_coef})

# 棒グラフをプロット
sns.barplot(x='Coefficient', y='Feature', data=df)
plt.show()
```



7-3 | 4-3. 特徴の結果を確認

```
# get_support関数で使用する特徴のインデックスを取得
# Trueになっている特徴が使用する特徴
sfm.get_support()
```

```
array([False, False,  True,  True, False, False, False, False,
       False,  True,  True,  True,  True, False, False,  True,  True])
```

```
# 削除すべき特徴の名前を取得
removed_idx = ~sfm.get_support()
train_data.columns[removed_idx]
```

```
Index(['bedrooms', 'bathrooms', 'floors', 'waterfront', 'view', 'condition',
       'grade', 'sqft_above', 'lat', 'long'],
      dtype='object')
```

7-3 | 4-4. 特徴の削除

```
# 削除してもよい特徴の名前を取得する
removed_feature = train_data.columns[removed_idx]
removed_feature

# drop関数で特徴を削除
selected_X = train_data.drop(removed_feature, axis=1)
selected_X.head()
```

	sqft_living	sqft_lot	sqft_basement	yr_built	yr_renovated	zipcode	sqft_living15	sqft_lot15
0	1180	5650	0	1955	0	98178	1340	5650
1	2570	7242	400	1951	1991	98125	1690	7639
2	770	10000	0	1933	0	98028	2720	8062
3	1960	5000	910	1965	0	98136	1360	5000
4	1680	8080	0	1987	0	98074	1800	7503

7-3 | 4-5. 特徴を削除した上で学習

```
# 正則化なし線形回帰モデルの学習
regr.fit(selected_X, y)
# 予測結果の取得
y_pred = regr.predict(selected_X)

# MAEを計算
mae = mean_absolute_error(y, y_pred)
print("MAE = %s"%round(mae,3))
# RMSEを計算
rmse = np.sqrt(mean_squared_error(y, y_pred) )
print("RMSE = %s"%round(rmse, 3))
# MAPEを計算
mape = mean_absolute_percentage_error(y, y_pred)
print("MAPE = %s"%round(mape*100,3), "%")
```

MAE = 163452.352
RMSE = 247759.025
MAPE = 34.133 %

現場で使える 機械学習・データ分析基礎講座

第 8 章：決定木

ノートブック解説

■ 決定木

- 8-1_entropy_and_gini.ipynb
- 8-2_descision_tree.ipynb

8-1_entropy_and_gini

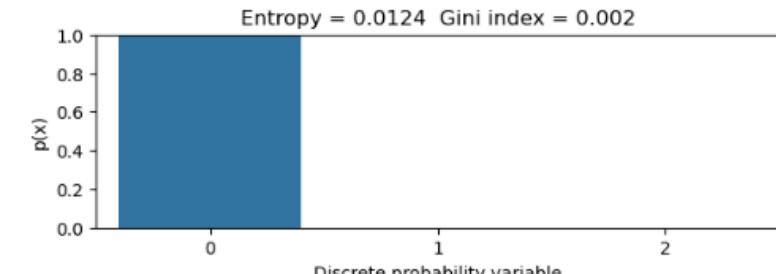
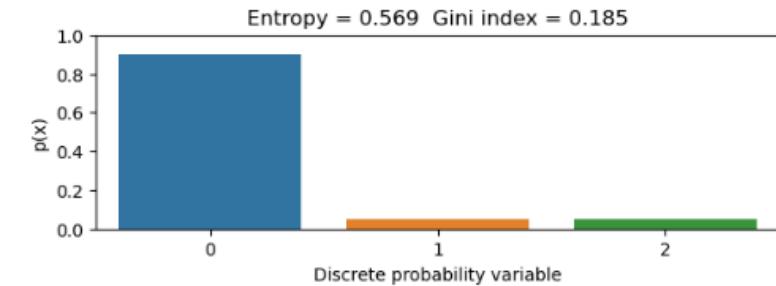
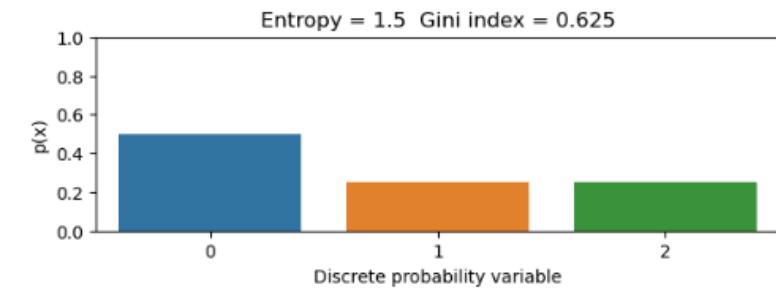
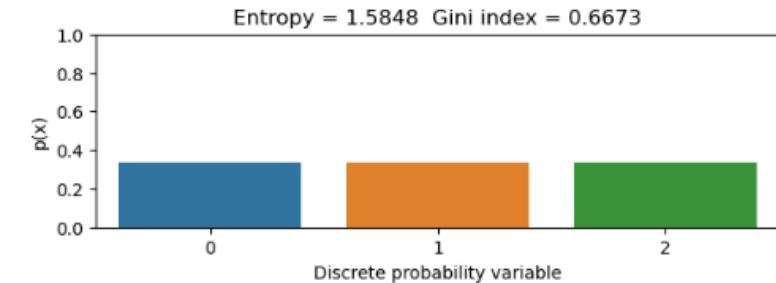
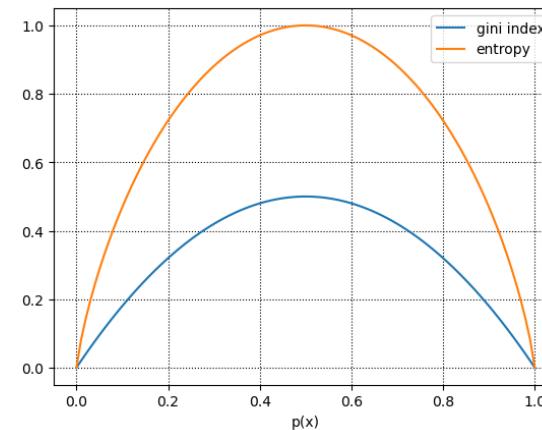
8-1_entropy_and_gini

■ 内容

- エントロピーとジニ係数（不純度の指標）を計算
- 2 クラス分類、多クラス分類における指標の変化を確認

■ 手順

1. ライブラリの読み込み
2. 指標を算出する関数の定義
3. 確率と指標の関係
4. 確率のばらつきと指標の関係
 1. グラフを表示する関数の定義
 2. 2 クラス分類の場合
 3. 3 クラス分類の場合
 4. 10 クラス分類の場合



8-1 | 1. ライブラリの読み込み

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

8-1 | 2. 指標を計算する関数の定義

```
def entropy(px):
    """
    エントロピーの計算
    px : 確率値のリスト
    """
    px = np.array(px).astype(float)
    # logx(0)は計算できないため、小さな値に置き換え
    px[px==0] = 1e-10
    return -1 * np.sum(px*np.log2(px))

def gini_index(px):
    """
    ジニ係数 (Gini Impurity) の計算
    px : 確率値のリスト
    """
    px = np.array(px).astype(float)
    return 1 - np.sum(px ** 2)
```

- エントロピー
 - $E = -\sum p(x) \log_2 p(x)$
- ジニ係数
 - $G = \sum p(x)(1 - p(x)) = 1 - \sum p(x)^2$

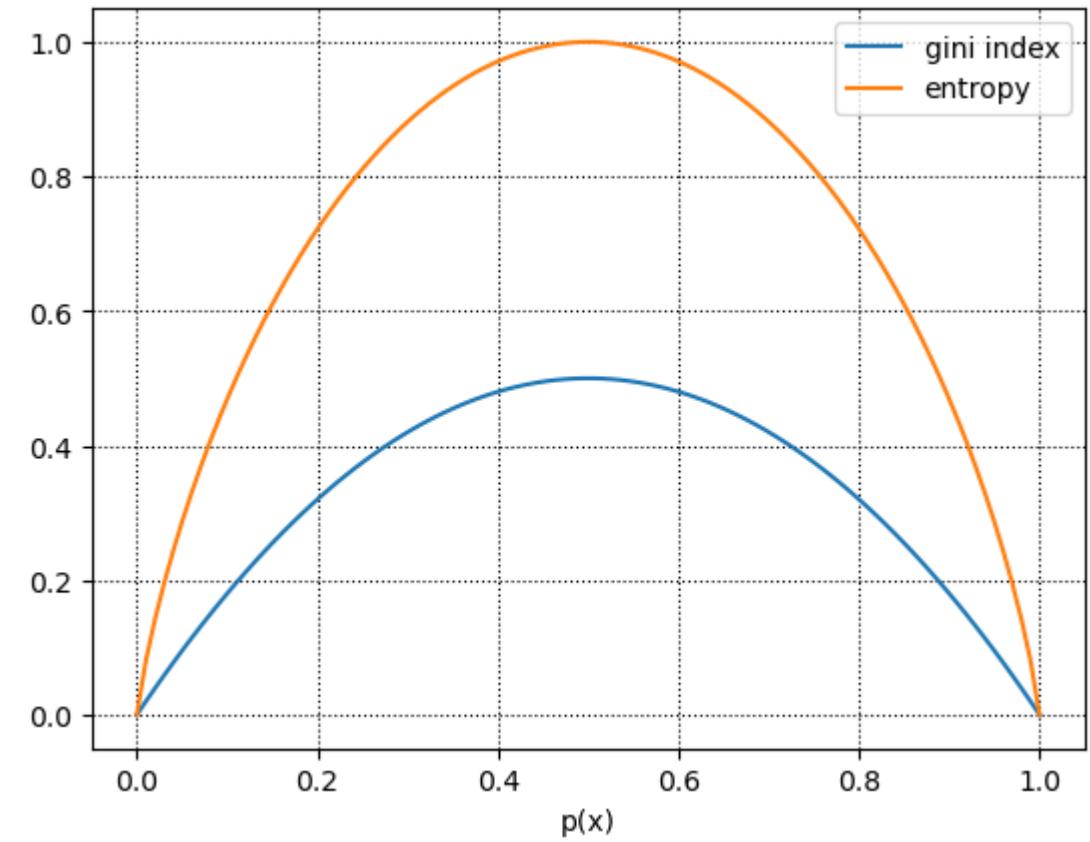
8-1 | 3. 確率と指標の関係

```
# 横軸の値を生成
p_list = np.linspace(0, 1, 100)
# 縦軸の値を格納するためのリスト
gini = np.zeros(len(p_list))
ent = np.zeros(len(p_list))

# 縦軸の値をG,Eそれぞれについて計算
for i, p in enumerate(p_list):
    gini[i] = gini_index(np.array([p, 1 - p]))
    ent[i] = entropy(np.array([p, 1 - p]))

# グラフの表示
sns.lineplot(x=p_list, y=gini, label='gini index')
sns.lineplot(x=p_list, y=ent, label='entropy')
# グリッド線の表示
plt.grid(which='major', color='black', linestyle=':')
plt.grid(which='minor', color='black', linestyle=':')
# ラベルの設定
plt.legend(loc='best')
plt.xlabel("p(x)")
plt.show()
```

$p(x) = 0.5$ のとき、最大となる



8-1 | 4-1. グラフを表示する関数の定義

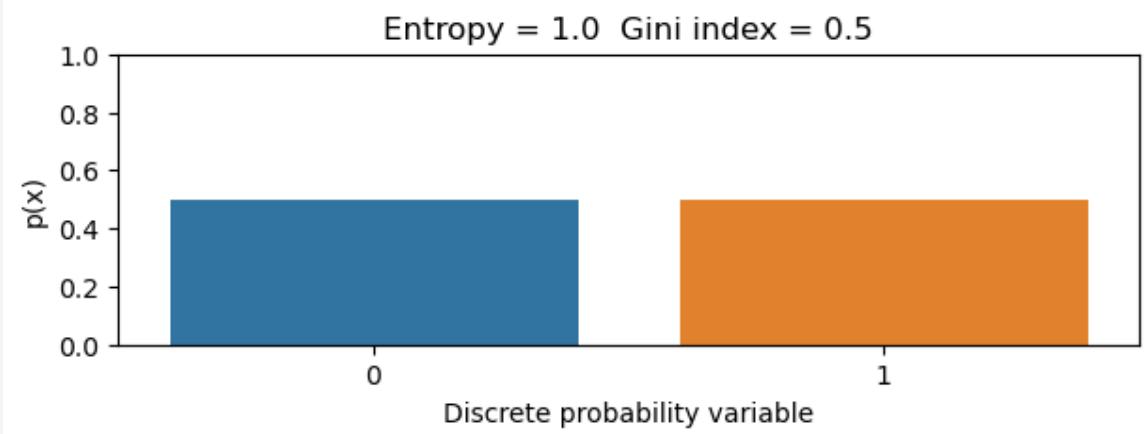
```
def entropy_and_gini_index_plot(px):
    """
    エントロピーとジニ係数のグラフを表示
    px : 確率値のリスト
    """
    # 確率値のデータフレームを作成
    df = pd.DataFrame(px, columns=['p(x)'])

    # グラフのサイズを設定
    plt.figure(figsize=(7,2))

    # EとGを計算し、グラフタイトルに設定
    title = "Entropy = %s Gini index = %s" % round(entropy(px),4) + "%s" % round(gini_index(px),4)
    plt.title(title)

    # 確率値を棒グラフとして表示
    sns.barplot(x=df.index, y='p(x)', data=df)

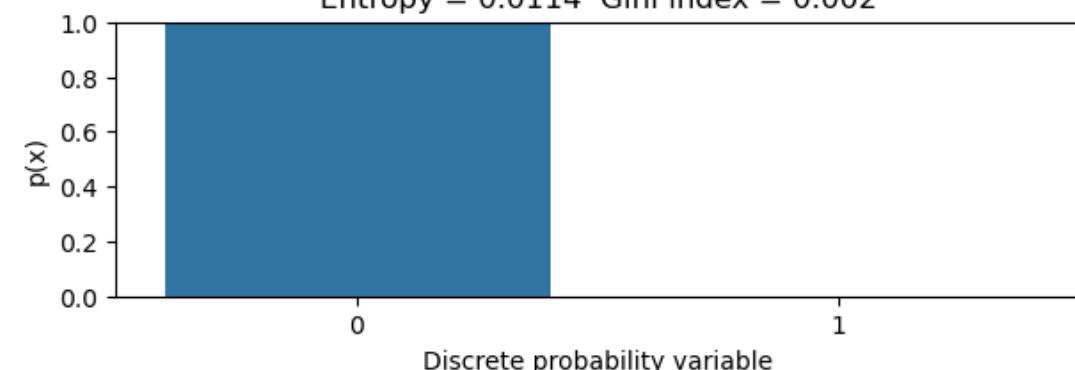
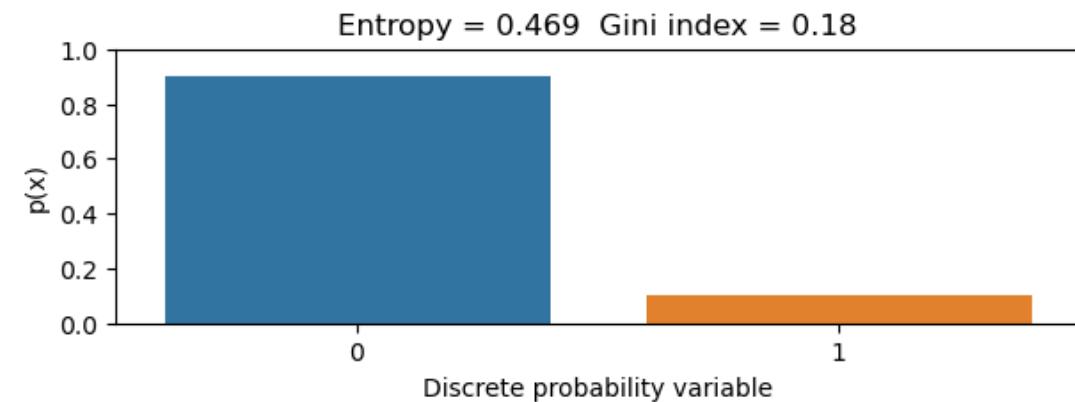
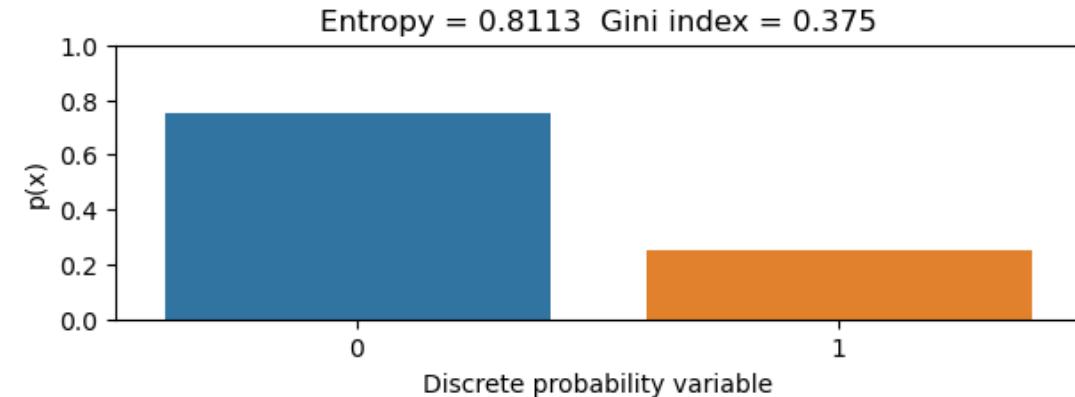
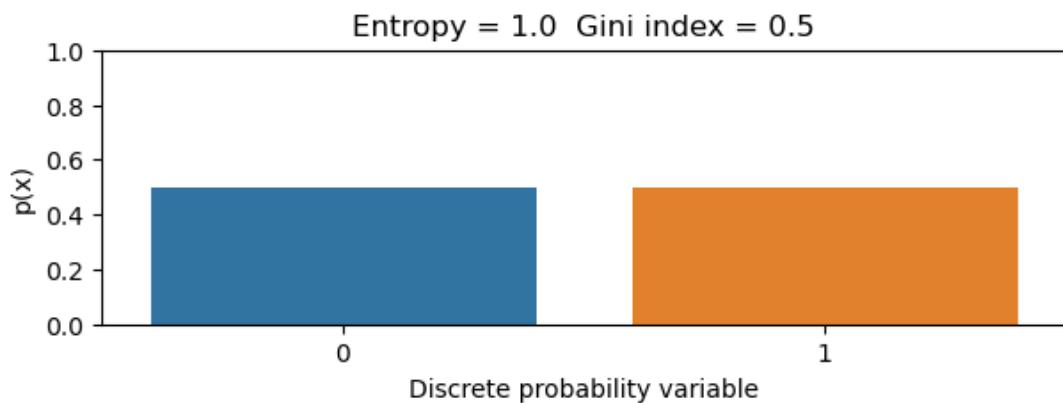
    # 軸のラベルと範囲を設定
    plt.xlabel('Discrete probability variable') # 離散確率変数
    plt.ylabel('p(x)')
    plt.ylim([0,1.0])
    plt.show()
```



8-1 | 4-2. 2 クラス分類の場合

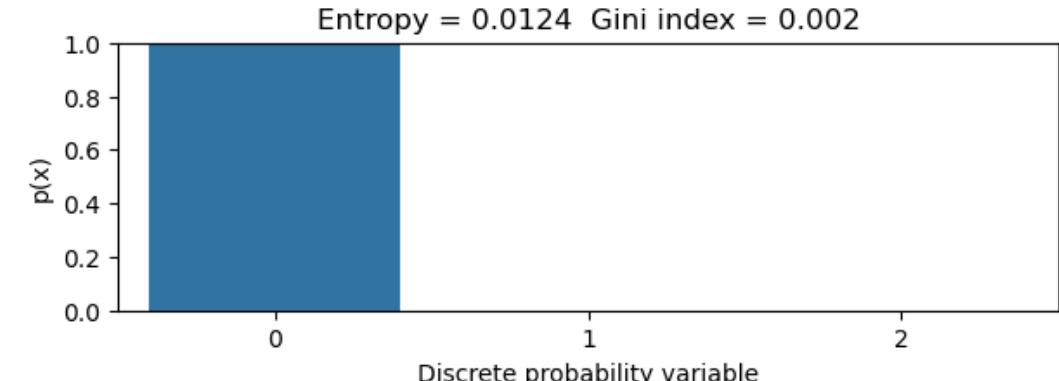
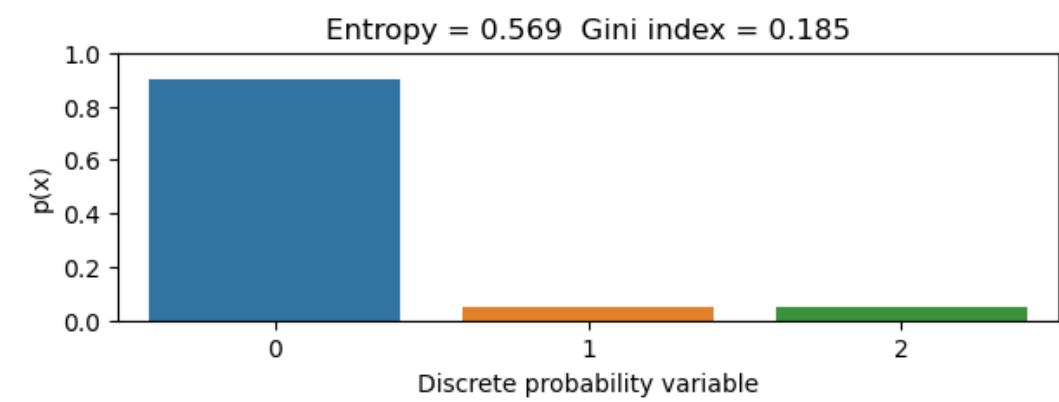
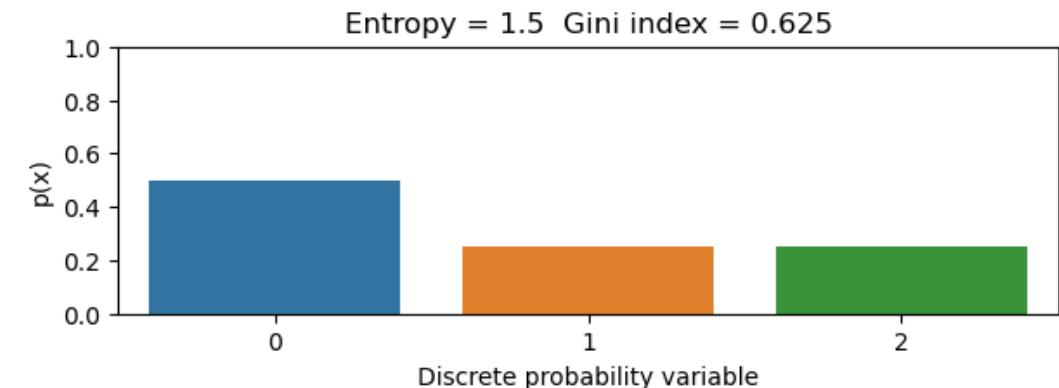
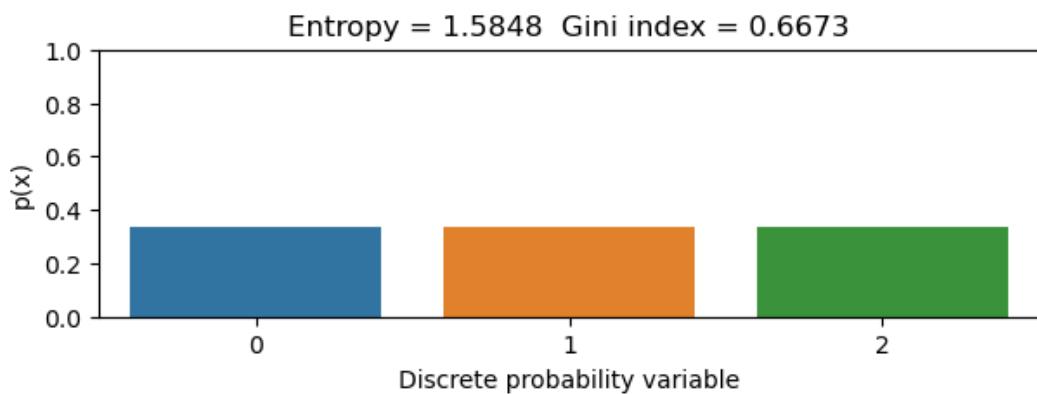
```
# クラス0の確率を徐々に大きくする
```

```
entropy_and_gini_index_plot([0.5, 0.5])
entropy_and_gini_index_plot([0.75, 0.25])
entropy_and_gini_index_plot([0.9, 0.1])
entropy_and_gini_index_plot([0.999, 0.001])
```



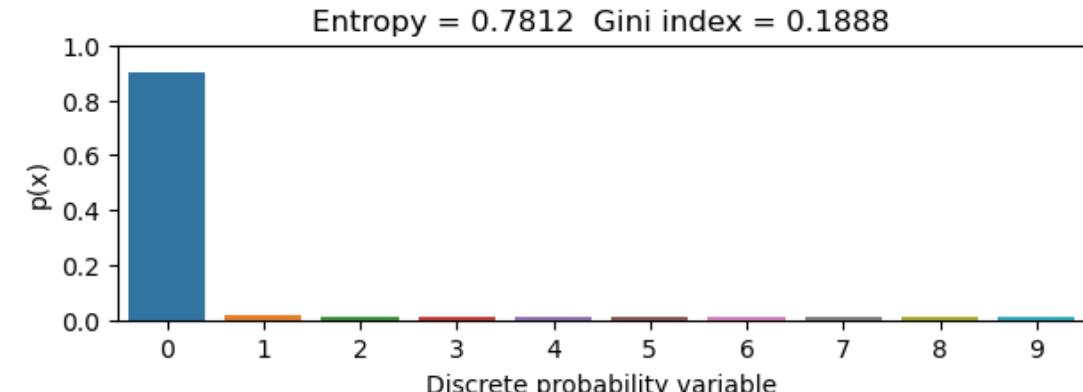
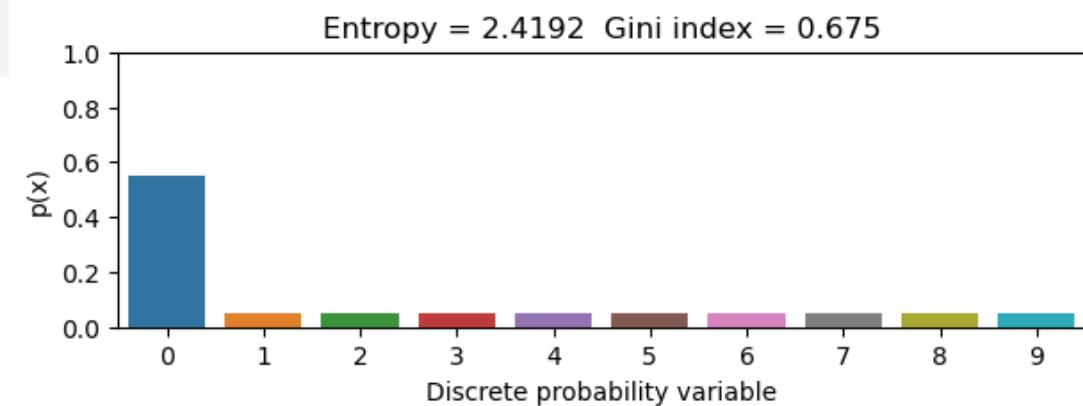
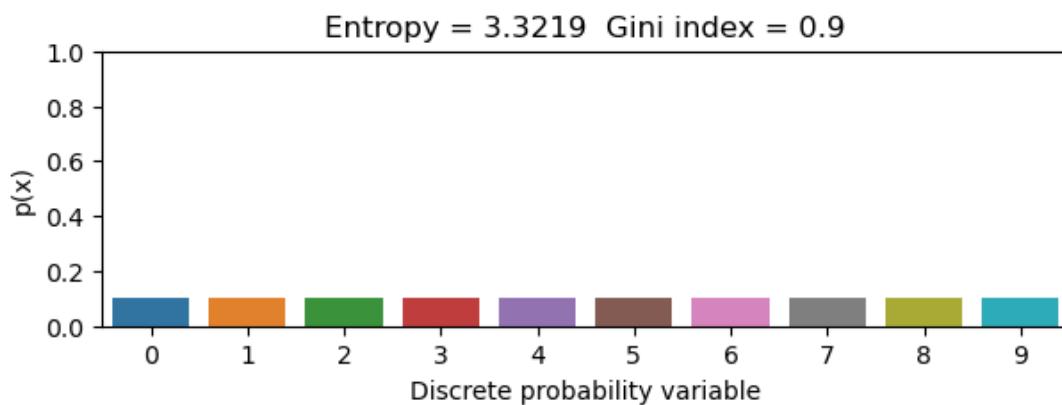
8-1 | 4-3. 3 クラス分類の場合

```
# クラス0の確率を徐々に大きくする  
# クラス1・2の確率は同じ値  
entropy_and_gini_index_plot([0.333, 0.333, 0.333])  
entropy_and_gini_index_plot([0.5, 0.25, 0.25])  
entropy_and_gini_index_plot([0.9, 0.05, 0.05])  
entropy_and_gini_index_plot([0.999, 0.0005, 0.0005])
```



8-1 | 4-4. 10 クラス分類の場合

```
# クラス0の確率を徐々に大きくする  
# クラス1~9の確率は同じ値  
entropy_and_gini_index_plot([0.1]*10) # 要素自体を掛け算で増やす  
entropy_and_gini_index_plot([0.55]+[0.05]*9) # リストの連結  
entropy_and_gini_index_plot([0.9, 0.02]+[0.01]*8)
```

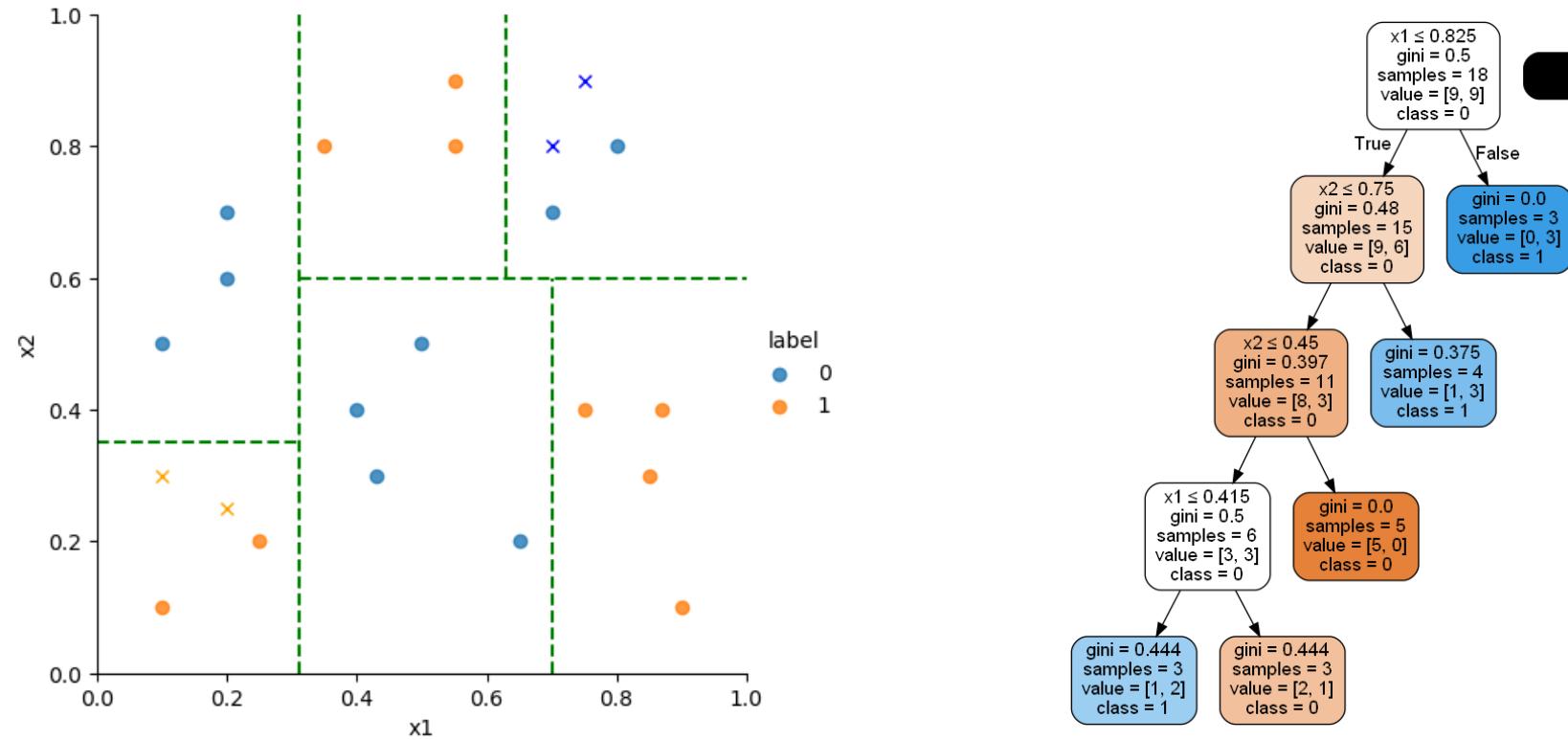


8-2_descision_tree

8-2_descision_tree

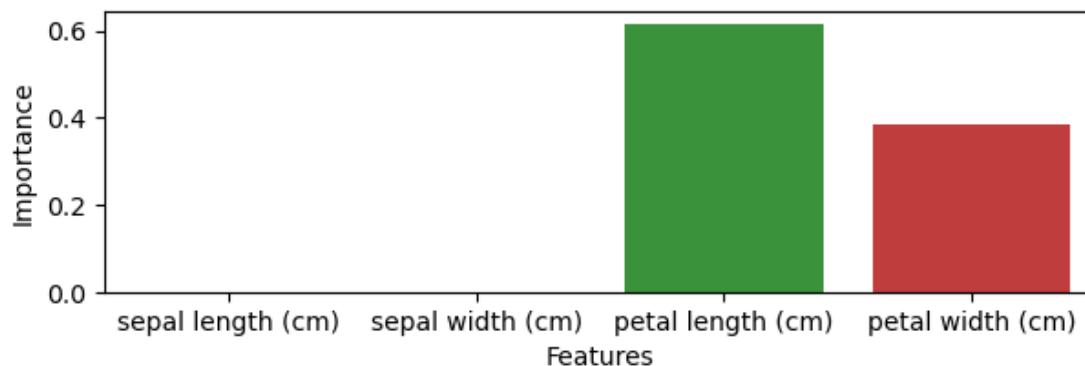
■ 内容

- 疑似データに対して人力で境界線を引いて、決定木のイメージを確認
- scikit-learn を用いて決定木を作成し、graphviz により可視化



■ 手順

1. ライブラリの読み込み
2. 疑似データの作成
3. 人力で境界線を設定
 1. 関数の定義
 2. [演習] 境界線の変更
 3. 汎化性能の確認
4. 疑似データを用いた決定木の学習
 1. モデルの構築・学習
 2. モデルの評価
 3. 変数の重要度を確認
 4. モデルの可視化
5. Iris データセットを用いた決定木の学習
 1. データの読み込み
 2. データの可視化
 3. モデルの構築・学習
 4. モデルの評価
 5. 変数の重要度を確認
 6. モデルの可視化



8-2 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

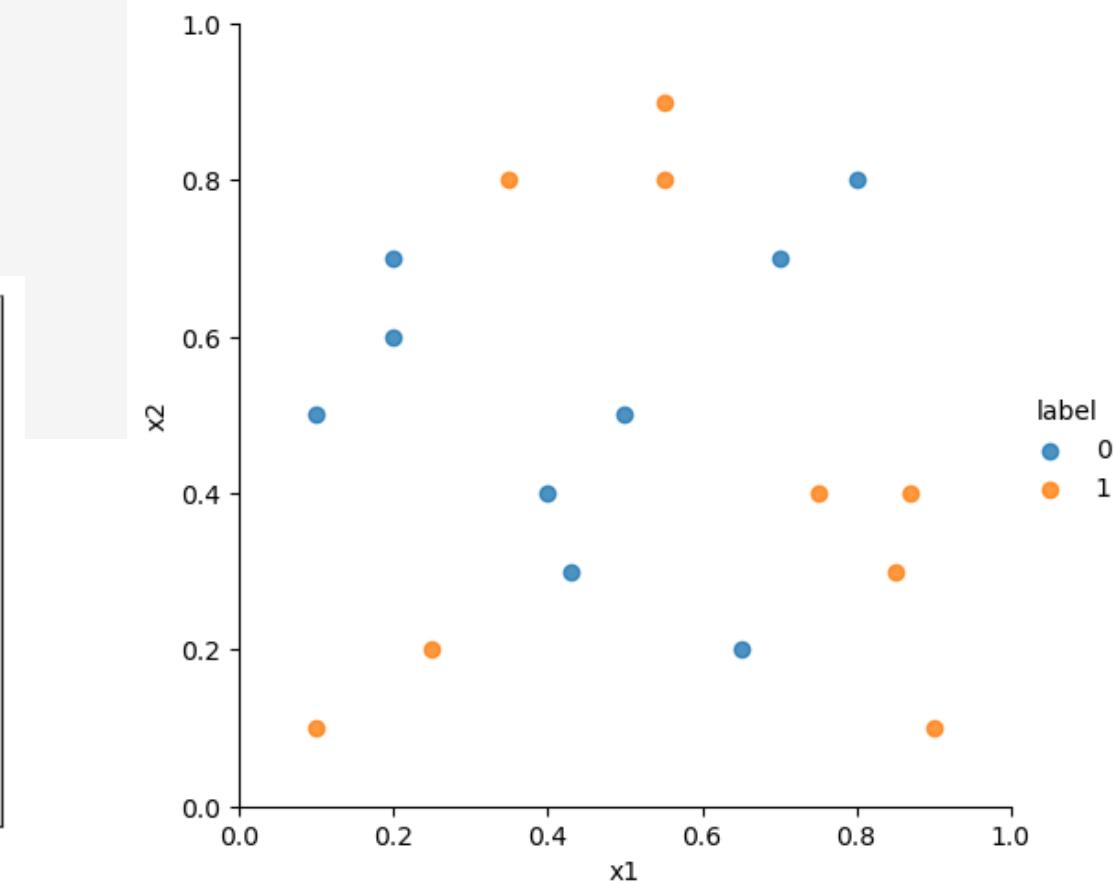
# アヤメの品種分類データセット
from sklearn.datasets import load_iris
# ホールドアウト法を実行するための関数
from sklearn.model_selection import train_test_split
# 分類問題の評価指標
from sklearn.metrics import accuracy_score, classification_report

# 決定木の実装、DOTファイルへの出力
from sklearn.tree import DecisionTreeClassifier, export_graphviz
# ファイルの入出力を文字列として保持
from io import StringIO
# グラフ構造を可視化
import graphviz
from pydotplus import graph_from_dot_data
# 画像データを表示
from IPython.display import Image
```

8-2 | 2. 疑似データの作成

```
df = pd.DataFrame({  
    "x1": [0.1, 0.1, 0.2, 0.25, 0.2, 0.35, 0.4, 0.43, 0.65, 0.55, 0.55, 0.55, 0.5, 0.75, 0.7, 0.85, 0.87, 0.8, 0.9],  
    "x2": [0.1, 0.5, 0.7, 0.2, 0.6, 0.8, 0.4, 0.3, 0.2, 0.9, 0.8, 0.5, 0.4, 0.7, 0.3, 0.4, 0.8, 0.1],  
    "label": [1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1]  
})  
display(df.head())  
  
# Seabornを用いて散布図を表示  
# fit_reg=Falseで回帰直線を非表示  
sns.lmplot(x="x1", y="x2", hue="label", data=df, fit_reg=False)  
  
# 表示範囲の調整  
plt.ylim([0,1]), plt.xlim([0,1])  
plt.show()
```

	x1	x2	label
0	0.10	0.1	1
1	0.10	0.5	0
2	0.20	0.7	0
3	0.25	0.2	1
4	0.20	0.6	0



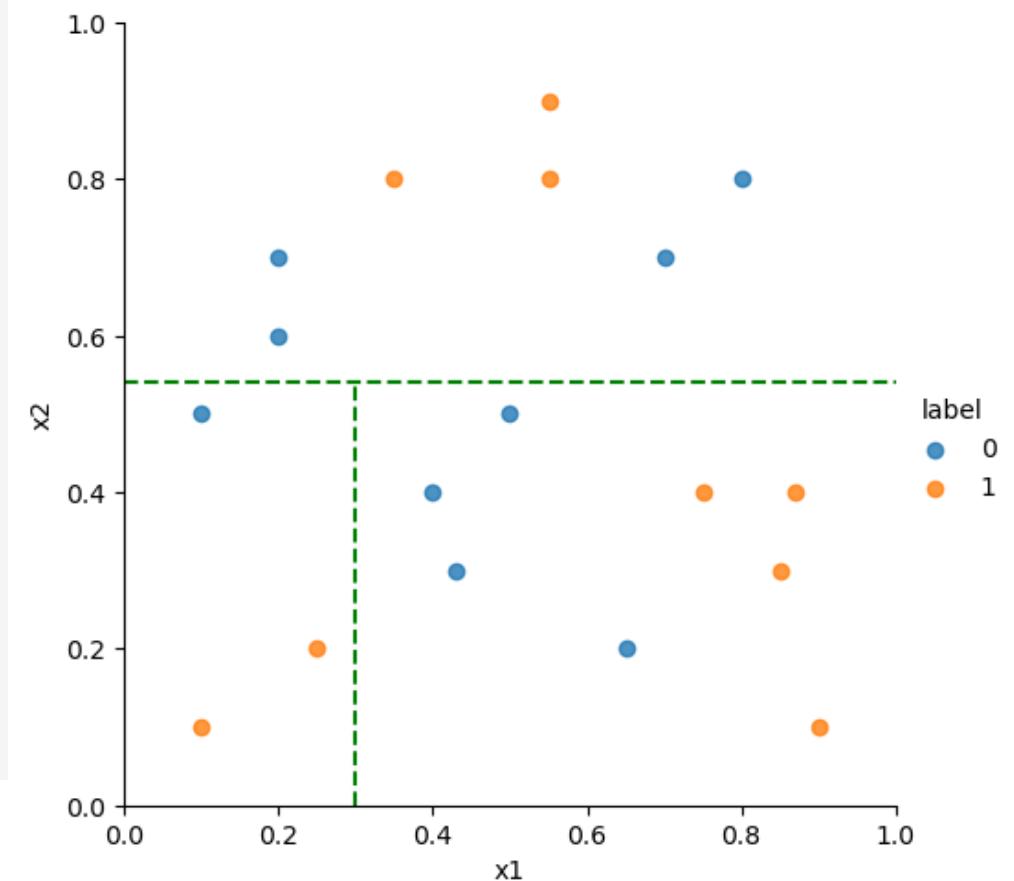
8-2 | 3-1. 関数の定義

```
# 散布図と境界線を表示する関数
def decision_line(start_points, end_points):
    # 散布図の表示
    sns.lmplot(x="x1", y="x2", hue="label", data=df, fit_reg=False)
    plt.ylim([0,1]),plt.xlim([0,1])

    # 境界線の表示
    for sp, ep in zip(start_points, end_points):
        X = [sp[0], ep[0]]
        Y = [sp[1], ep[1]]
        plt.plot(X, Y, ls = "--", color="g")

# 始点の座標
start_points = [[0.00, 0.54], [0.30, 0.00]]
# 終点の座標
end_points = [[1.00, 0.54], [0.30, 0.54]]

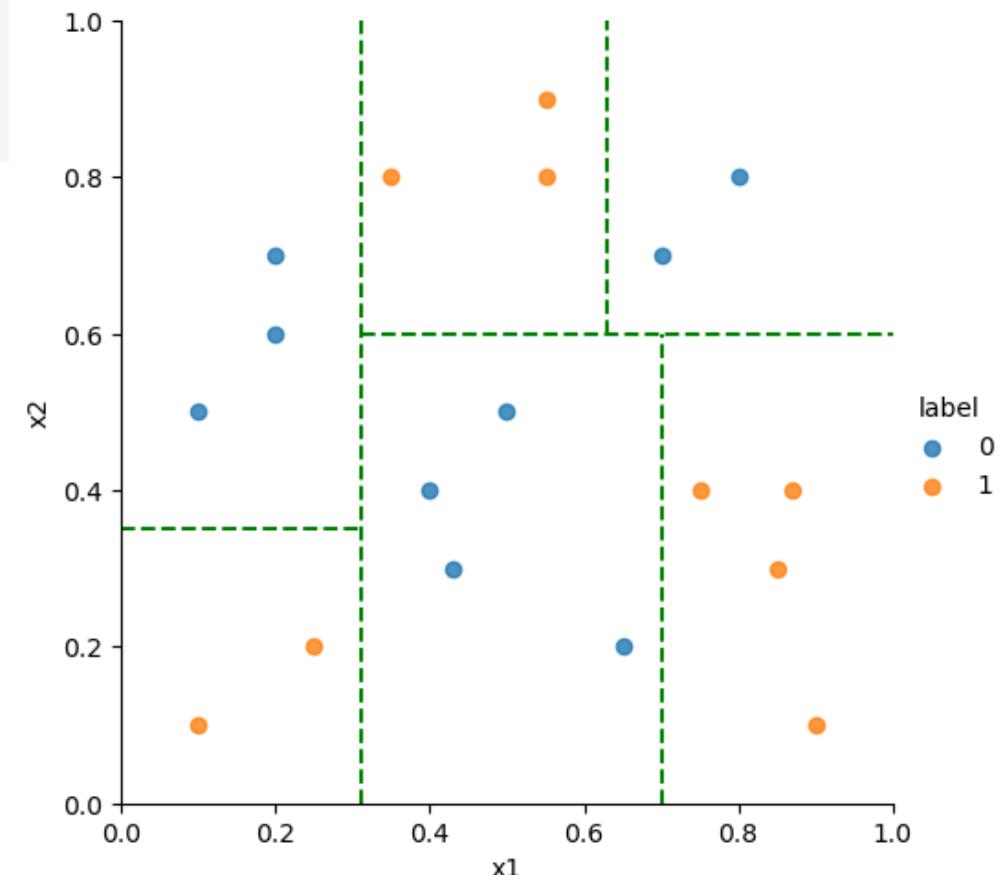
# 関数の実行
decision_line(start_points, end_points)
plt.show()
```



8-2 | 3-2. [演習] 境界線の変更

```
# 始点の座標（ここを編集）
start_points = [[0.31, 0.00], [0.31, 0.60], [0.70, 0.00], [0.63, 0.60], [0.00, 0.35]]
# 終点の座標（ここを編集）
end_points = [[0.31, 1.00], [1.00, 0.60], [0.70, 0.60], [0.63, 1.00], [0.31, 0.35]]

# 関数の実行
decision_line(start_points, end_points)
plt.show()
```



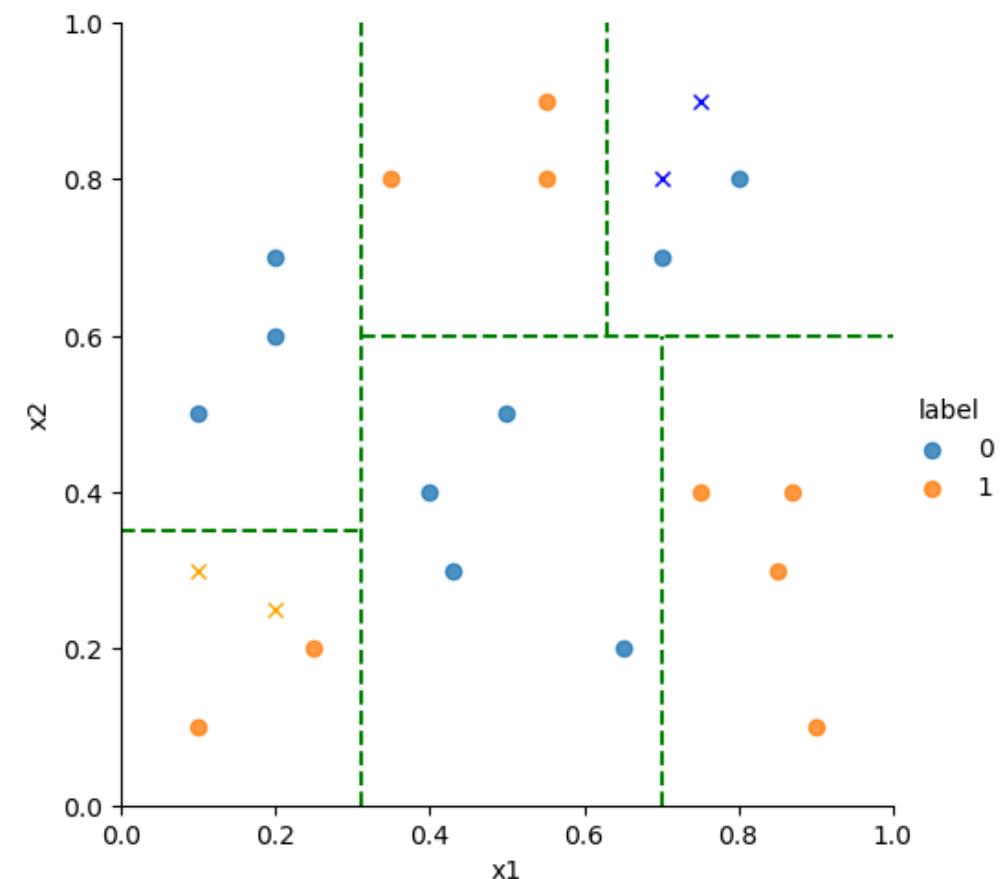
8-2 | 3-3. 汎化性能の確認

```
# クラス0のテストデータ
X_test_0 = [[0.70, 0.80], [0.75, 0.90]]
# クラス1のテストデータ
X_test_1 = [[0.10, 0.30], [0.20, 0.25]]

# 学習用データと境界線を表示
decision_line(start_points, end_points)

# クラス0のデータを、青色の×として表示
for point in X_test_0:
    plt.plot(point[0], point[1], marker="x", color="blue")
# クラス1のデータを、オレンジ色の×として表示
for point in X_test_1:
    plt.plot(point[0], point[1], marker="x", color="orange")

plt.show()
```



```
# 説明変数
X_train = df[["x1", "x2"]].values
# 目的変数
y_train = df["label"].values

# 決定木モデルの構築
clf = DecisionTreeClassifier(
    criterion="gini", max_depth=None,
    min_samples_split=3, min_samples_leaf=3,
    random_state=1234)

# モデルの学習
clf.fit(X_train, y_train)

# 訓練性能の確認
print("train_accuracy:", clf.score(X_train, y_train))
```

■ criterion

- 不純度の評価方法
- "gini"または"entropy"を設定

■ max_depth

- 決定木の最大深度
- 葉ノードに辿り着くまでに分岐を何回行うか

■ min_samples_split

- ノードを分割するために必要な最小サンプル数
- これを満たさないノードは分割できない

■ min_samples_leaf

- 葉ノードが持つべき最小サンプル数
- これを満たさないノードは作成できない

train_accuracy: 0.8333333333333334

```
# 説明変数
X_test = X_test_0 + X_test_1 # リストの結合

# 目的変数
y_test = [0, 0, 1, 1]

# テスト用データに対する予測
y_pred = clf.predict(X_test)
print("predicted_class:", y_pred)
print("test_accuracy:", accuracy_score(y_test, y_pred))
```

```
predicted_class: [1 1 1 1]
test_accuracy: 0.5
```

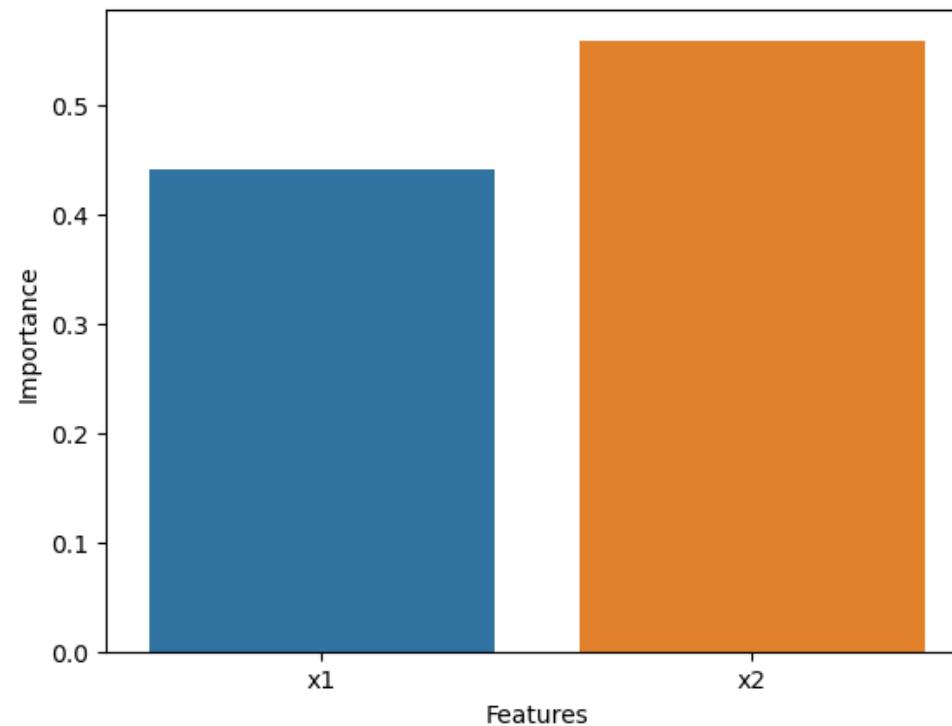
8-2 | 4-3. 変数の重要度を確認

```
# 説明変数の重要度を出力する
# 重要度=ある説明変数による不純度の減少量の合計
print(clf.feature_importances_)

# データフレームの作成
df = pd.DataFrame(clf.feature_importances_, index=["x1", "x2"])
df = df.reset_index()

# 棒グラフの表示
sns.barplot(x='index', y=0, data=df)
plt.ylabel("Importance")
plt.xlabel("Features")
plt.show()
```

[0.44137931 0.55862069]



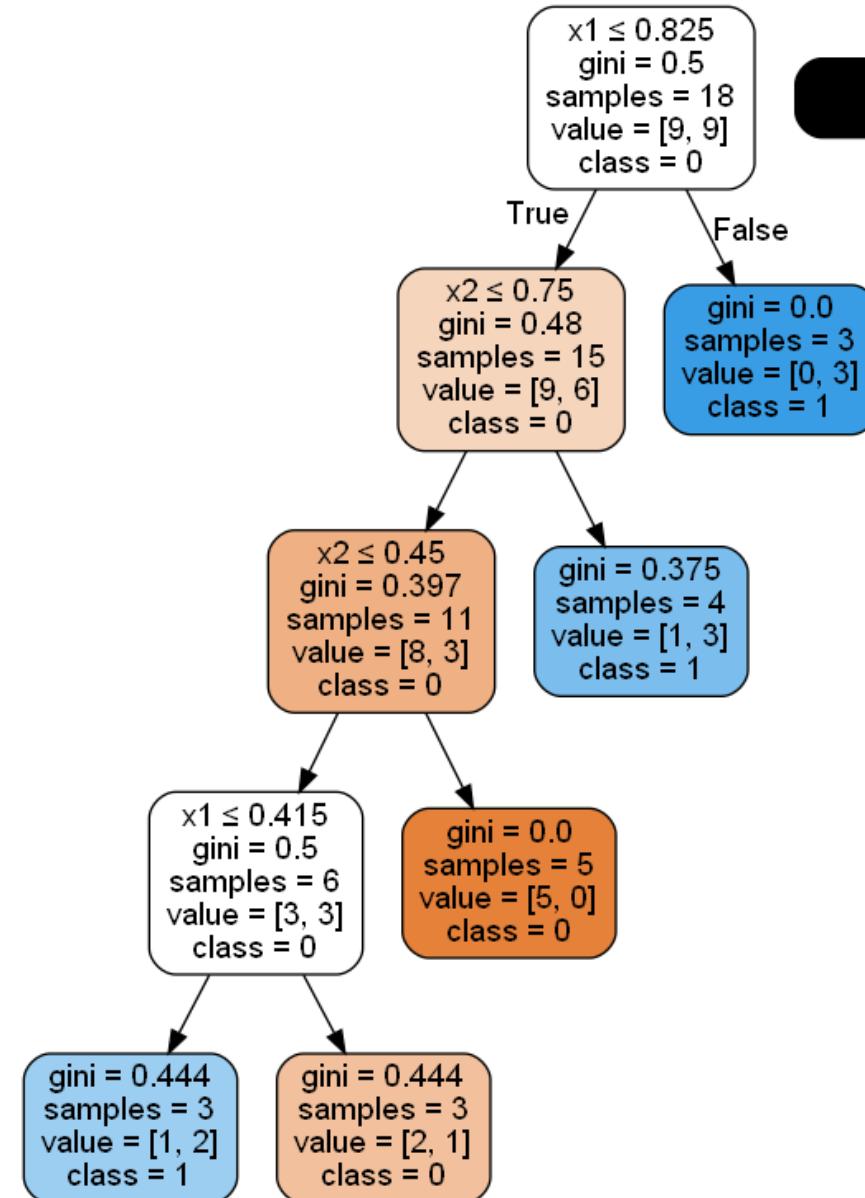
8-2 | 4-4. モデルの可視化

```
# DOTファイルの格納先（文字列として保持）
dot_data = StringIO()

# モデルをDOTファイルとして出力
export_graphviz(
    clf, out_file=dot_data,
    feature_names=[ "x1", "x2" ],
    class_names=[ "0", "1" ],
    filled=True, rounded=True,
    special_characters=True
)

# DOTファイルを読み込む
graph = graph_from_dot_data(dot_data.getvalue())

# PNG形式の画像として表示
Image(graph.create_png())
```



8-2 | 5-1. データの読み込み

```
# データセットの読み込み
iris = load_iris()

# 説明変数をデータフレームに変換
df_iris = pd.DataFrame(iris.data, columns=iris.feature_names)
# 目的変数をデータフレームに追加
df_iris["label"] = iris.target

# クラス番号を文字列に置き換え
df_iris["label"] = df_iris["label"].map(
    {0:iris.target_names[0],
     1:iris.target_names[1],
     2:iris.target_names[2]})

df_iris.head()
```

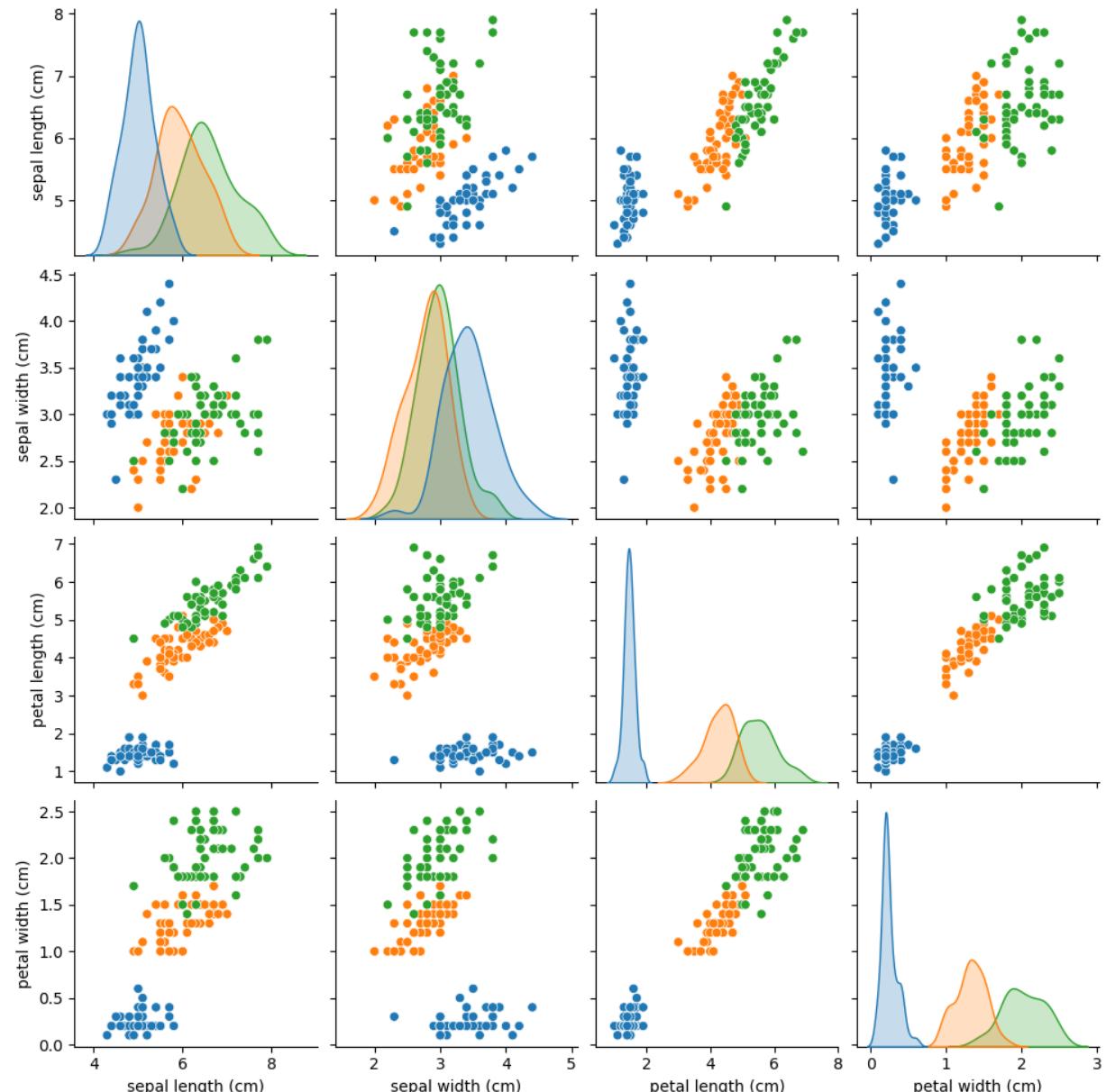
- 目的変数 (クラス)
 - Setosa
 - Versicolor
 - Virginica
- 説明変数
 - Sepal Length (萼片の長さ)
 - Sepal Width (萼片の幅)
 - Petal Length (花弁の長さ)
 - Petal Width (花弁の幅)

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

8-2 | 5-2. データの可視化

```
# 散布図行列の表示（クラスごとに色分け）
sns.pairplot(df_iris, hue="label")
plt.show()
```

label
● setosa
● versicolor
● virginica



8-2 | 5-3. モデルの構築・学習

```
# 学習用・テスト用の分割
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.4, random_state=1234
)

# モデルの構築
clf = DecisionTreeClassifier(
    criterion="gini", max_depth=None,
    min_samples_leaf=3, random_state=1234)

# モデルの学習
clf.fit(X_train, y_train)

# 訓練性能の確認
print("train_accuracy:", clf.score(X_train, y_train))
```

train_accuracy: 0.9777777777777777

```
# 汎化性能の確認
print("test_accuracy:", clf.score(X_test, y_test))

# 予測結果の取得
y_pred = clf.predict(X_test)

# 混同行列の確認
scores = classification_report(y_test, y_pred)
print(scores)
```

test_accuracy: 0.9833333333333333				
	precision	recall	f1-score	support
0	1.00	1.00	1.00	19
1	1.00	0.96	0.98	23
2	0.95	1.00	0.97	18
accuracy			0.98	60
macro avg	0.98	0.99	0.98	60
weighted avg	0.98	0.98	0.98	60

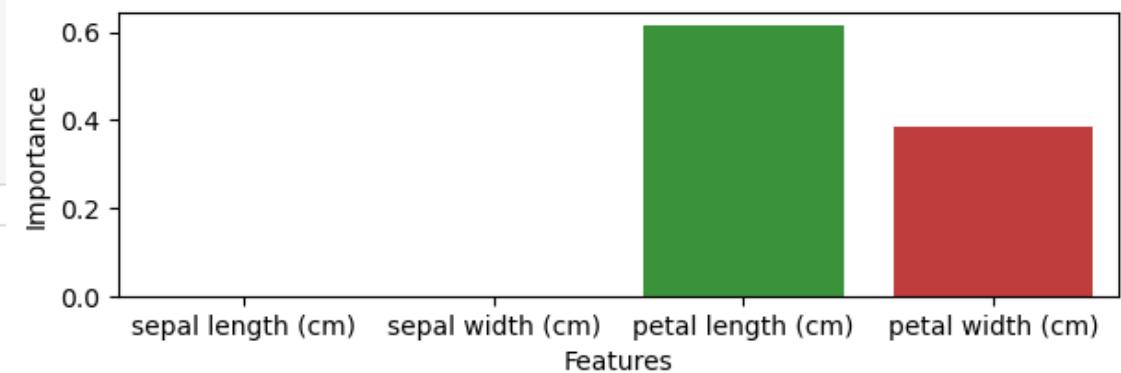
8-2 | 5-5. 変数の重要度を確認

```
# 説明変数の重要度を出力する
# 重要度=ある説明変数による不純度の減少量の合計
print(clf.feature_importances_)

# データフレームの作成
feature_imp = pd.DataFrame(
    clf.feature_importances_,
    index=iris.feature_names, columns=['Importance']
)

# 棒グラフの表示
plt.figure(figsize=(7,2))
sns.barplot(x=feature_imp.index, y=feature_imp['Importance'])
plt.ylabel("Importance")
plt.xlabel("Features")
plt.show()
```

```
[0.          0.          0.61454019  0.38545981]
```



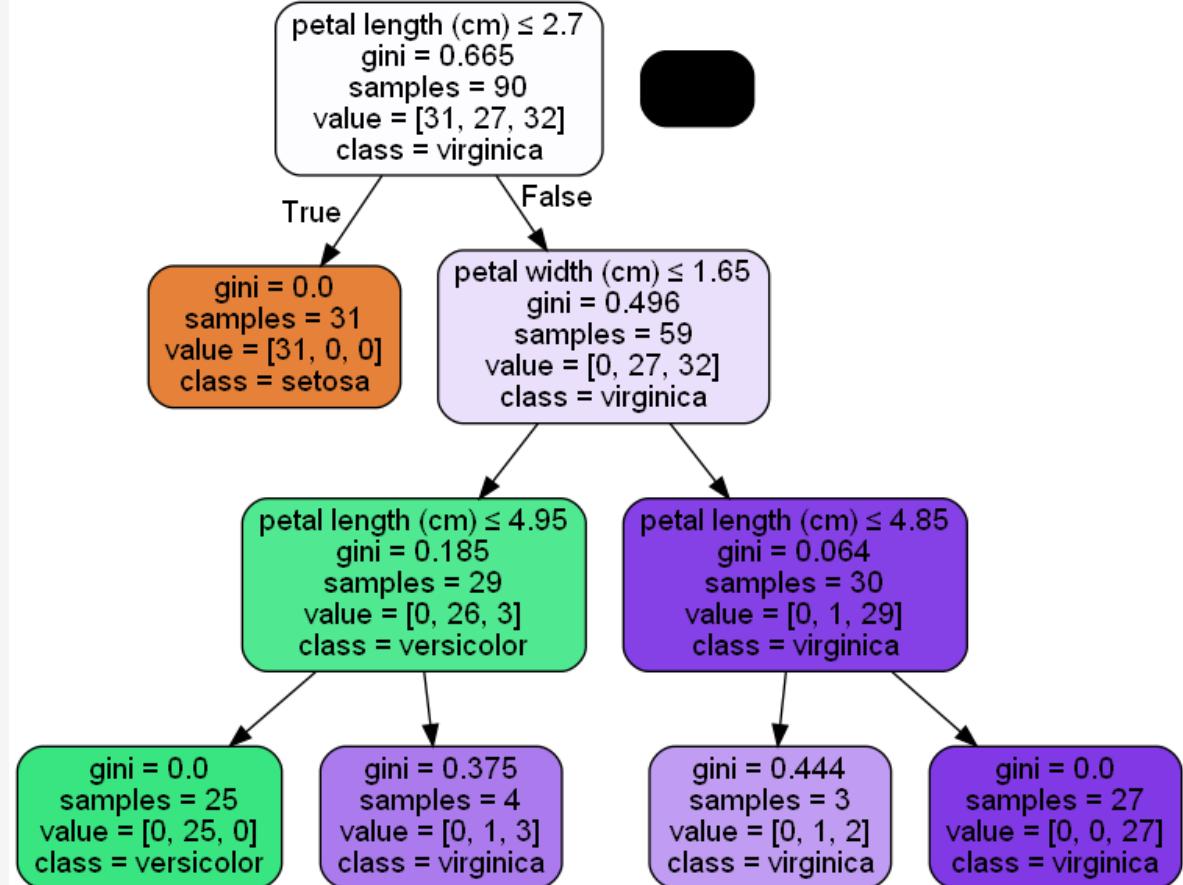
8-2 | 5-6. モデルの可視化

```
# DOTファイルの格納先（文字列として保持）
dot_data = StringIO()

# モデルをDOTファイルとして出力
export_graphviz(
    clf, out_file=dot_data,
    feature_names=iris.feature_names,
    class_names=iris.target_names,
    filled=True, rounded=True,
    special_characters=True
)

# DOTファイルを読み込む
graph = graph_from_dot_data(dot_data.getvalue())

# PNG形式の画像として表示
Image(graph.create_png())
```



現場で使える 機械学習・データ分析基礎講座

第9章：アンサンブル学習

ノートブック解説

■ アンサンブル学習

- 9-1_random_forest.ipynb
- 9-2_adaboost.ipynb
- 9-3_xgboost.ipynb

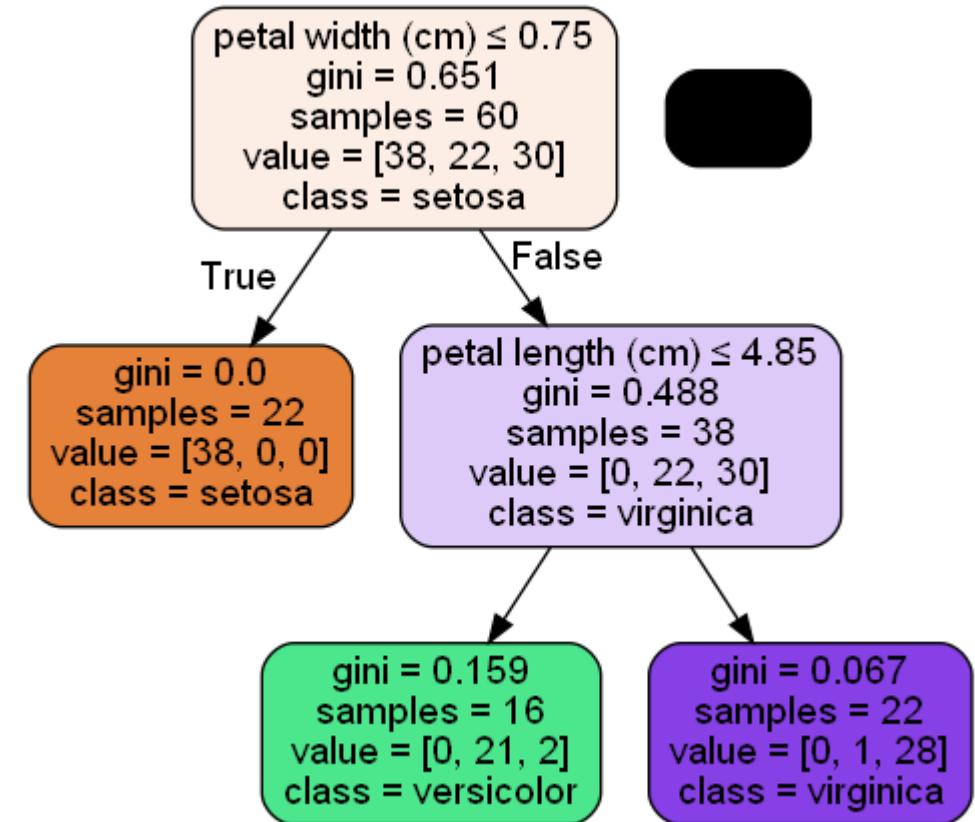
9-1_random_forest

■ 内容

- ランダムフォレストを多クラス分類に適用
- graphviz を用いて各決定木の中身を可視化

■ 手順

- ライブラリの読み込み
- データの読み込み
- データの可視化
- モデルの構築・学習
- モデルの評価
- 変数の重要度を確認
- モデルの可視化



9-1 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# データセット
from sklearn.datasets import load_iris
# データの分割
from sklearn.model_selection import train_test_split
# 多クラス分類の評価指標
from sklearn.metrics import accuracy_score, classification_report

# モデルの可視化
from io import StringIO
from sklearn.tree import export_graphviz
import graphviz
from pydotplus import graph_from_dot_data
from IPython.display import Image

# ランダムフォレストによる分類モデル
from sklearn.ensemble import RandomForestClassifier
```

9-1 | 2. データの読み込み

```
# データセットの読み込み
iris = load_iris()

# 説明変数をデータフレームに変換
df_iris = pd.DataFrame(iris.data, columns=iris.feature_names)
# 目的変数をデータフレームに追加
df_iris["label"] = iris.target

# クラス番号を文字列に置き換え
df_iris["label"] = df_iris["label"].map(
    {0:iris.target_names[0],
     1:iris.target_names[1],
     2:iris.target_names[2]})

df_iris.head()
```

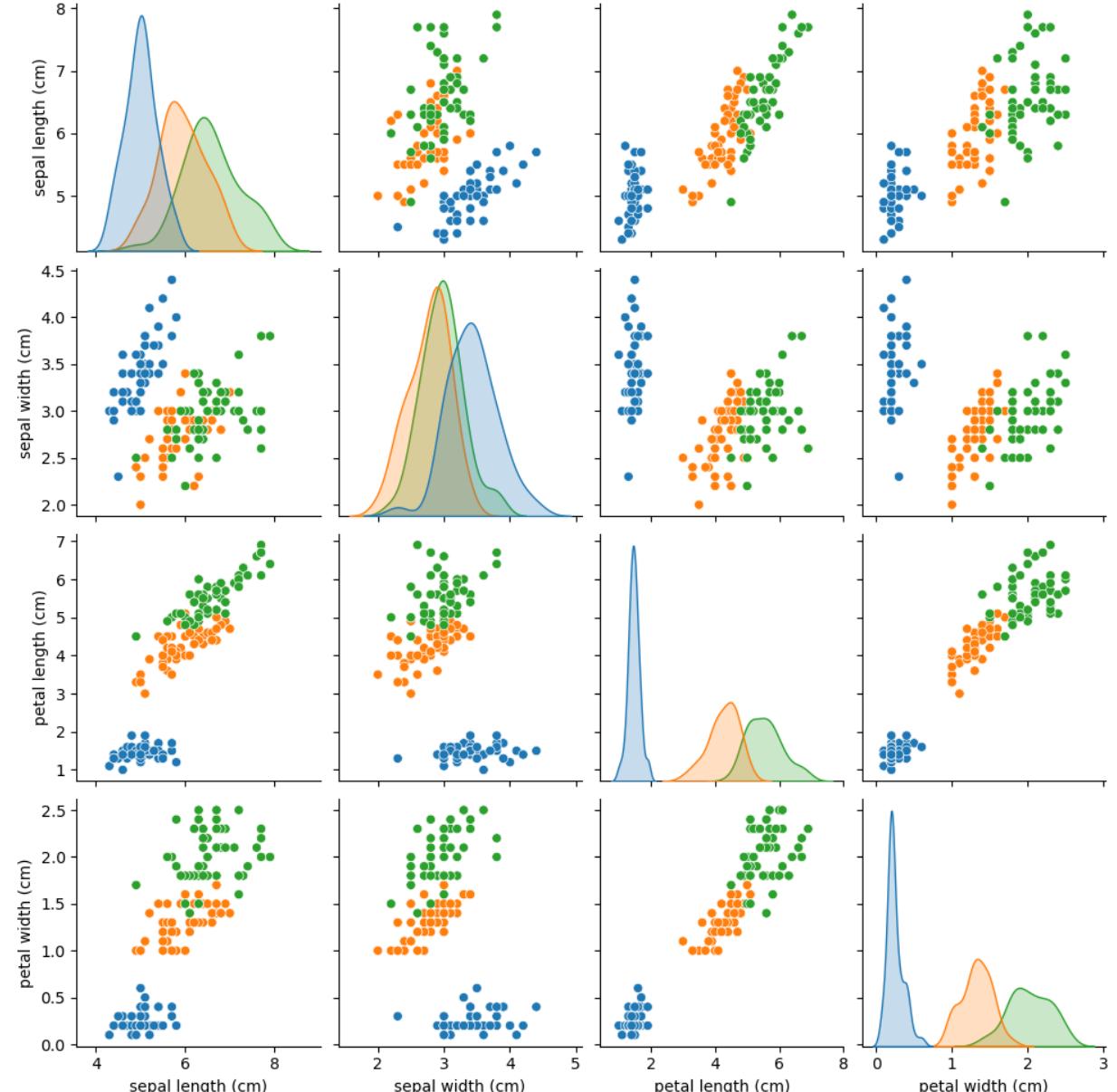
- 目的変数 (クラス)
 - Setosa
 - Versicolor
 - Virginica
- 説明変数
 - Sepal Length (萼片の長さ)
 - Sepal Width (萼片の幅)
 - Petal Length (花弁の長さ)
 - Petal Width (花弁の幅)

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

9-1 | 3. データの可視化

```
# 散布図行列の表示（クラスごとに色分け）
sns.pairplot(df_iris, hue="label")
plt.show()
```

label
● setosa
● versicolor
● virginica



9-1 | 4. モデルの構築・学習

```
# 学習用・テスト用の分割
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.4, random_state=1234
)

# ランダムフォレストの構築
# n_estimatorsで使用する決定木の数を指定
# それ以外のパラメータは単体の決定木と同じ
clf = RandomForestClassifier(
    n_estimators=10, max_depth=2, criterion="gini",
    min_samples_leaf=2, min_samples_split=2, random_state=1234
)

# モデルの学習
clf.fit(X_train, y_train)

# 訓練性能の確認
print("train_accuracy:", clf.score(X_train, y_train))
```

train_accuracy: 0.9777777777777777

9-1 | 5. モデルの評価

```
# 汎化性能の確認
print("test_accuracy:", clf.score(X_test, y_test))

# 予測結果の取得
y_pred = clf.predict(X_test)

# 評価指標の確認
scores = classification_report(y_test, y_pred)
print(scores)
```

```
test_accuracy: 0.9833333333333333
      precision    recall  f1-score   support
          0       1.00     1.00     1.00      19
          1       1.00     0.96     0.98      23
          2       0.95     1.00     0.97      18

      accuracy                           0.98      60
   macro avg       0.98     0.99     0.98      60
weighted avg       0.98     0.98     0.98      60
```

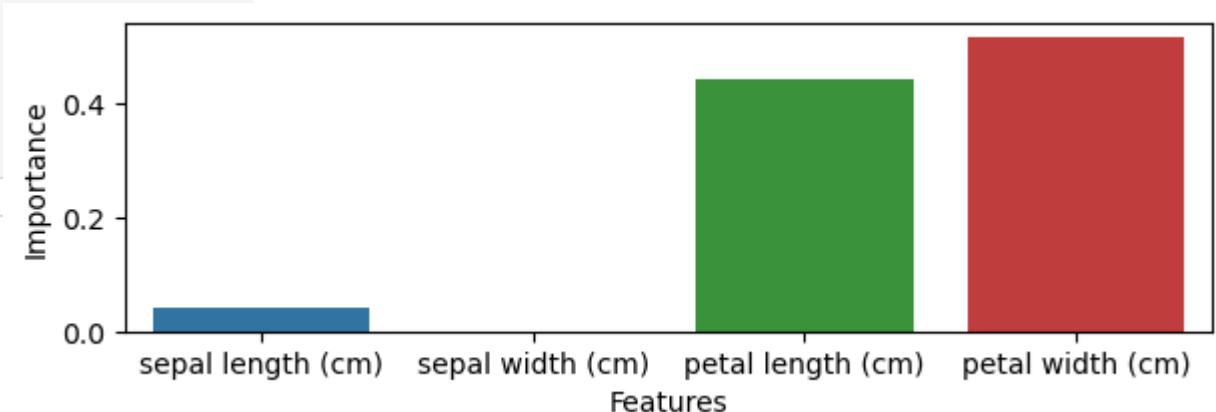
9-1 | 6. 変数の重要度を確認

```
# 説明変数の重要度を出力する
# 重要度=ある説明変数による不純度の減少量の合計
print(clf.feature_importances_)

# データフレームの作成
feature_imp = pd.DataFrame(
    clf.feature_importances_,
    index=iris.feature_names, columns=['Importance']
)

# 棒グラフの表示
plt.figure(figsize=(7,2))
sns.barplot(x=feature_imp.index, y=feature_imp['Importance'])
plt.ylabel("Importance")
plt.xlabel("Features")
plt.show()
```

[0.04003301 0. 0.4437946 0.5161724]



9-1 | 7. モデルの可視化

```
# ランダムフォレストから決定木を取り出す
for i, est in enumerate(clf.estimators_):
    print(i, ":", est)

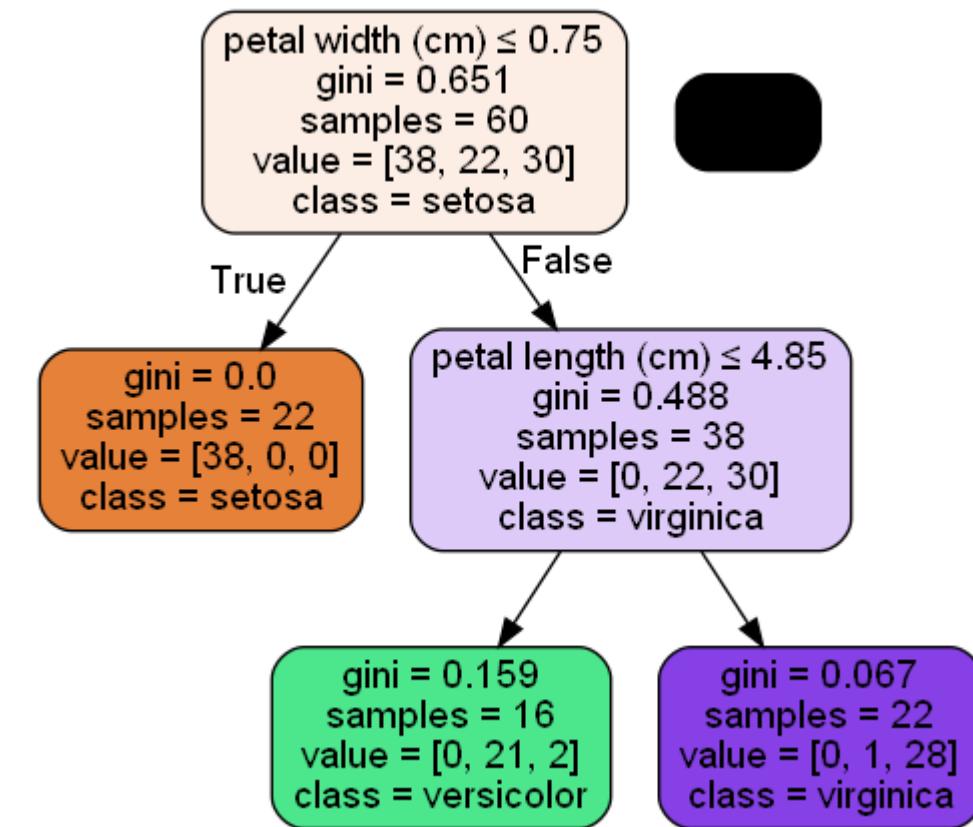
# DOTファイルの格納先（文字列として保持）
dot_data = StringIO()

# モデルをDOTファイルとして出力
export_graphviz(
    est, out_file=dot_data,
    feature_names=iris.feature_names,
    class_names=iris.target_names,
    filled=True, rounded=True,
    special_characters=True
)

# DOTファイルを読み込む
graph = graph_from_dot_data(dot_data.getvalue())

# PNG形式の画像として表示
display(Image(graph.create_png()))
```

```
0 : DecisionTreeClassifier(max_depth=2, max_features='sqrt', min_samples_leaf=2,
                           random_state=822569775)
```



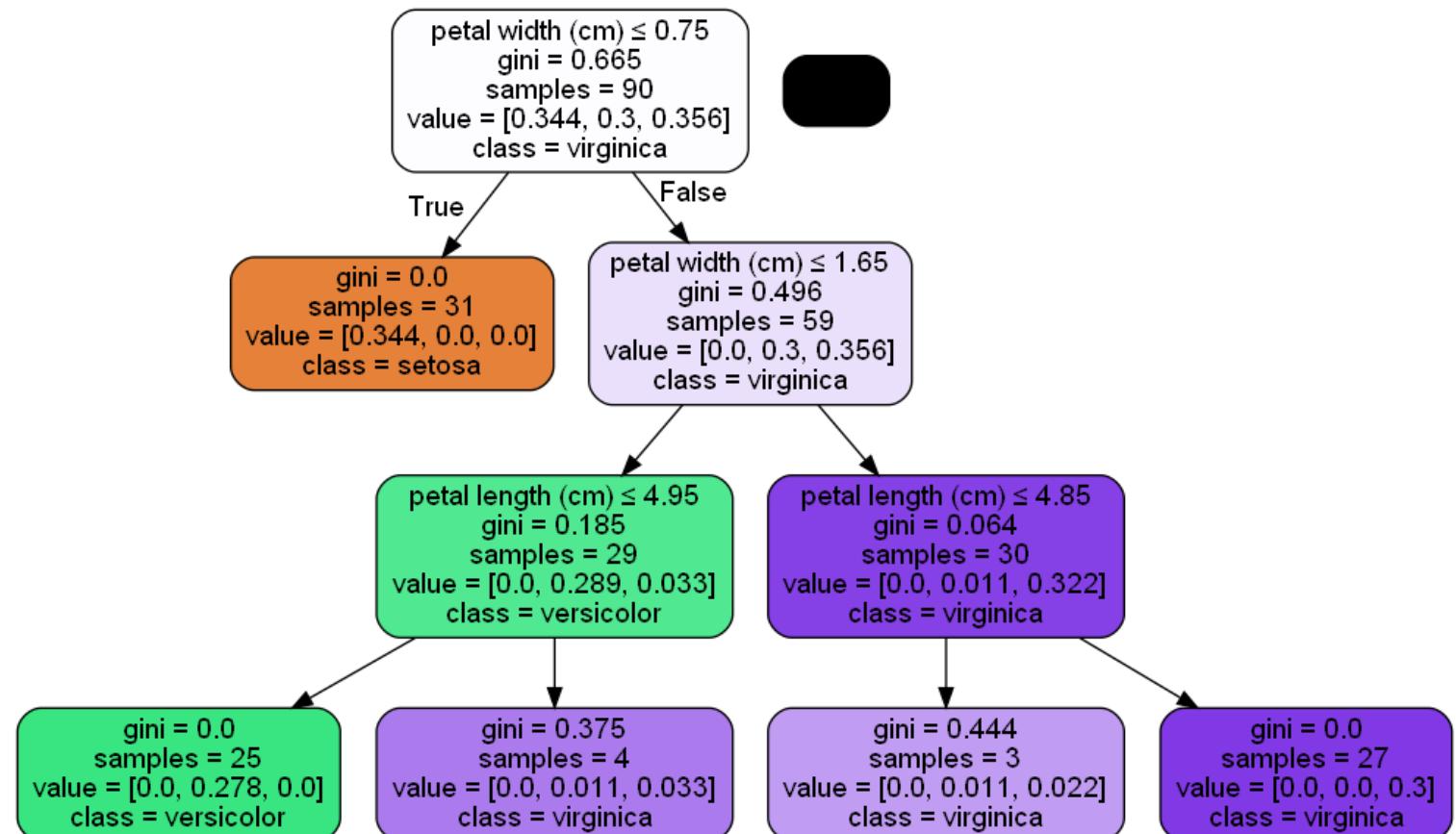
9-2_adaboost

■ 内容

- アダブースト+決定木を多クラス分類に適用
- graphviz を用いて各決定木の中身を可視化

■ 手順

- ライブラリの読み込み
- データの読み込み
- データの可視化
- モデルの構築・学習
- モデルの評価
- 変数の重要度を確認
- モデルの可視化



9-2 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# データセット
from sklearn.datasets import load_iris
# データの分割
from sklearn.model_selection import train_test_split
# 多クラス分類の評価指標
from sklearn.metrics import accuracy_score, classification_report

# モデルの可視化
from io import StringIO
from sklearn.tree import export_graphviz
import graphviz
from pydotplus import graph_from_dot_data
from IPython.display import Image

# 決定木+アダブーストによる分類モデル
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
```

9-2 | 2. データの読み込み

```
# データセットの読み込み
iris = load_iris()

# 説明変数をデータフレームに変換
df_iris = pd.DataFrame(iris.data, columns=iris.feature_names)
# 目的変数をデータフレームに追加
df_iris["label"] = iris.target

# クラス番号を文字列に置き換え
df_iris["label"] = df_iris["label"].map(
    {0:iris.target_names[0],
     1:iris.target_names[1],
     2:iris.target_names[2]})

df_iris.head()
```

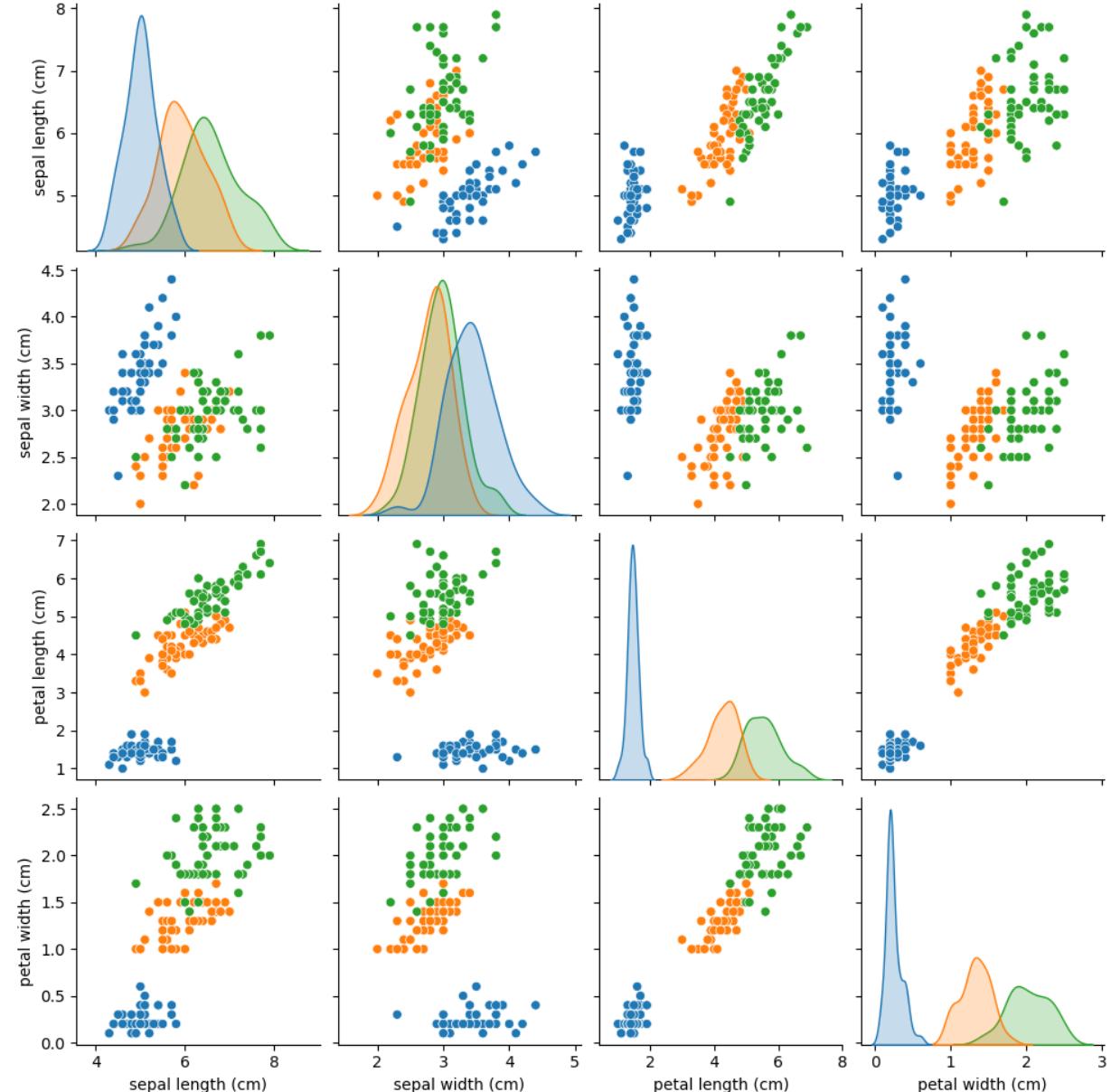
- 目的変数 (クラス)
 - Setosa
 - Versicolor
 - Virginica
- 説明変数
 - Sepal Length (萼片の長さ)
 - Sepal Width (萼片の幅)
 - Petal Length (花弁の長さ)
 - Petal Width (花弁の幅)

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

9-2 | 3. データの可視化

```
# 散布図行列の表示（クラスごとに色分け）
sns.pairplot(df_iris, hue="label")
plt.show()
```

label
● setosa
● versicolor
● virginica



9-2 | 4. モデルの構築・学習

```
# 学習用・テスト用の分割
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.4, random_state=1234
)

# アダブーストの構築
# 最初の引数で弱学習器を設定
# それ以外の引数はランダムフォレストと同様
clf = AdaBoostClassifier(
    DecisionTreeClassifier(
        max_depth=3,
        min_samples_leaf=2,
        min_samples_split=2,
        random_state=1234,
        criterion="gini"
    ),
    n_estimators=10,
    random_state=1234
)

# モデルの学習
clf.fit(X_train, y_train)

# 訓練性能の確認
print("train_accuracy: ", clf.score(X_train, y_train))
```

```
train_accuracy: 1.0
```

9-2 | 5. モデルの評価

```
# 汎化性能の確認
print("test_accuracy:", clf.score(X_test, y_test))

# 予測結果の取得
y_pred = clf.predict(X_test)

# 評価指標の確認
scores = classification_report(y_test, y_pred)
print(scores)
```

```
test_accuracy: 0.9833333333333333
      precision    recall  f1-score   support
          0       1.00     1.00     1.00      19
          1       1.00     0.96     0.98      23
          2       0.95     1.00     0.97      18

   accuracy                           0.98      60
  macro avg       0.98     0.99     0.98      60
weighted avg       0.98     0.98     0.98      60
```

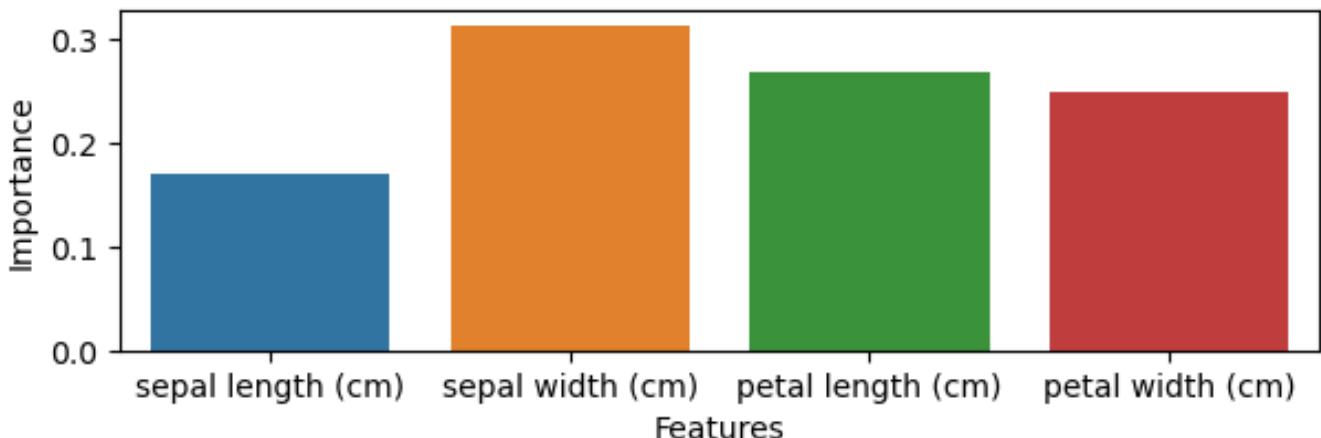
9-2 | 6. 変数の重要度を確認

```
# 説明変数の重要度を出力する
# 重要度=ある説明変数による不純度の減少量の合計
print(clf.feature_importances_)

# データフレームの作成
feature_imp = pd.DataFrame(
    clf.feature_importances_,
    index=iris.feature_names, columns=['Importance']
)

# 棒グラフの表示
plt.figure(figsize=(7,2))
sns.barplot(x=feature_imp.index, y=feature_imp['Importance'])
plt.ylabel("Importance")
plt.xlabel("Features")
plt.show()
```

[0.17003038 0.31232333 0.26811629 0.24952999]



9-2 | 7. モデルの可視化

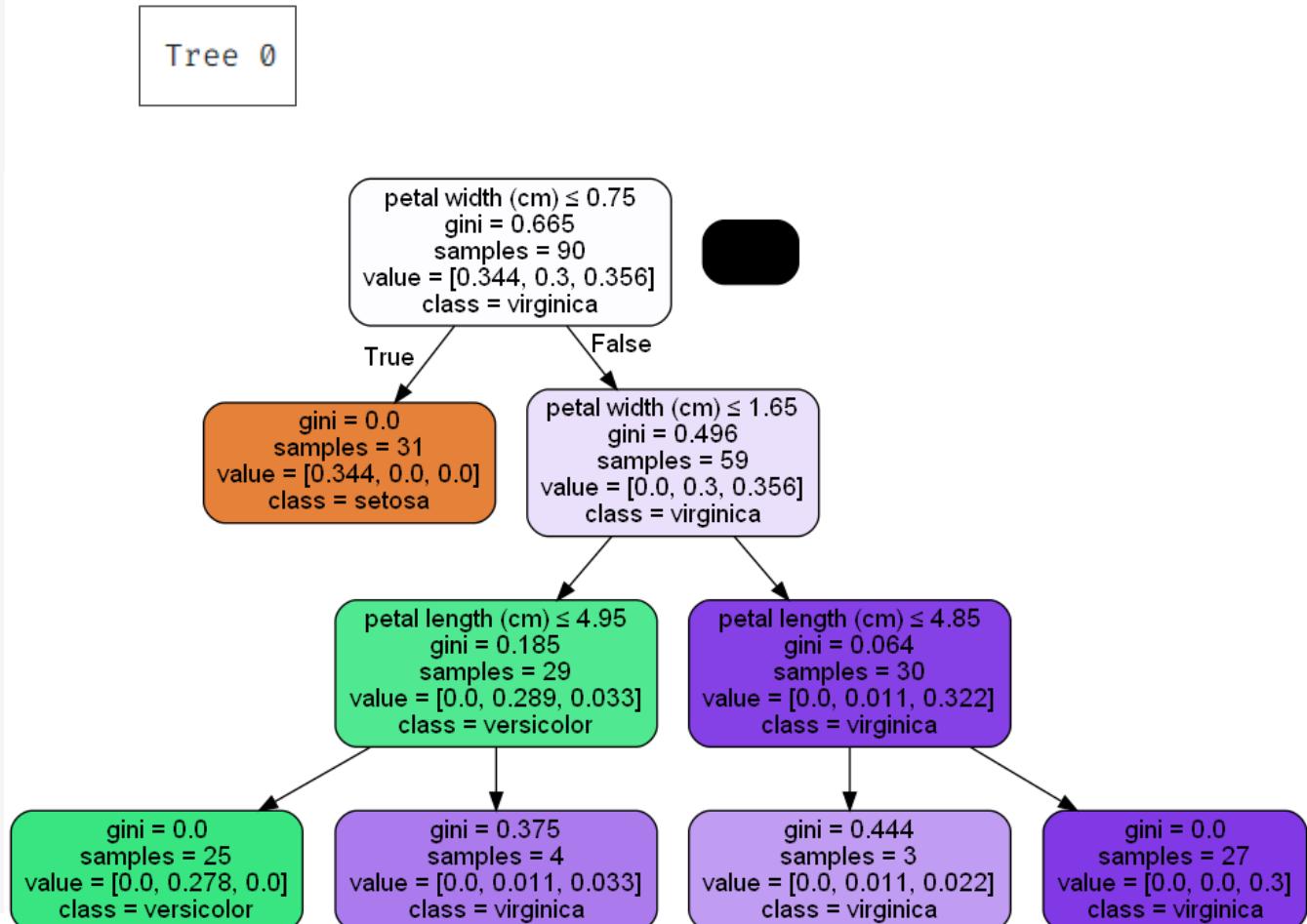
```
# アダブーストのモデルから決定木を取り出す
for i, est in enumerate(clf.estimators_):
    print("Tree", i)

# DOTファイルの格納先（文字列として保持）
dot_data = StringIO()

# モデルをDOTファイルとして出力
export_graphviz(
    est, out_file=dot_data,
    feature_names=iris.feature_names,
    class_names=iris.target_names,
    filled=True, rounded=True,
    special_characters=True
)

# DOTファイルを読み込む
graph = graph_from_dot_data(dot_data.getvalue())

# PNG形式の画像として表示
display(Image(graph.create_png()))
```



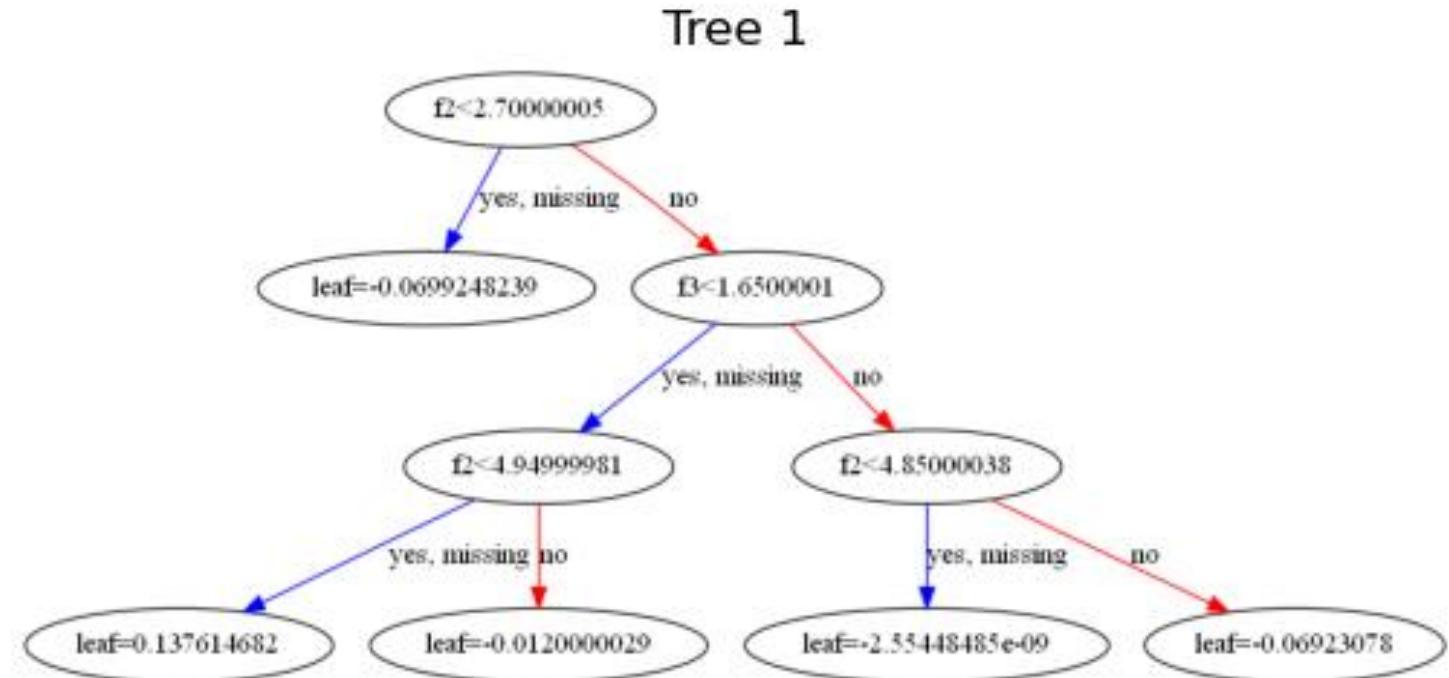
9-3_xgboost

■ 内容

- XGBoost を用いて、勾配ブースティング木を多クラス分類に適用
- graphviz を用いて各決定木の中身を可視化

■ 手順

1. ライブラリの読み込み
2. データの読み込み
3. データの可視化
4. モデルの構築・学習
5. モデルの評価
6. 変数の重要度を確認
7. モデルの可視化



9-3 | 1. ライブラリの読み込み

```
%matplotlib inline
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# データセット
from sklearn.datasets import load_iris
# データの分割
from sklearn.model_selection import train_test_split
# 多クラス分類の評価指標
from sklearn.metrics import accuracy_score, classification_report

# XGBoostの分類モデルと可視化用の関数
from xgboost import XGBClassifier, plot_tree

# Warningを非表示に設定
import warnings
warnings.simplefilter('ignore', UserWarning)
```

9-3 | 2. データの読み込み

```
# データセットの読み込み
iris = load_iris()

# 説明変数をデータフレームに変換
df_iris = pd.DataFrame(iris.data, columns=iris.feature_names)
# 目的変数をデータフレームに追加
df_iris["label"] = iris.target

# クラス番号を文字列に置き換え
df_iris["label"] = df_iris["label"].map(
    {0:iris.target_names[0],
     1:iris.target_names[1],
     2:iris.target_names[2]})

df_iris.head()
```

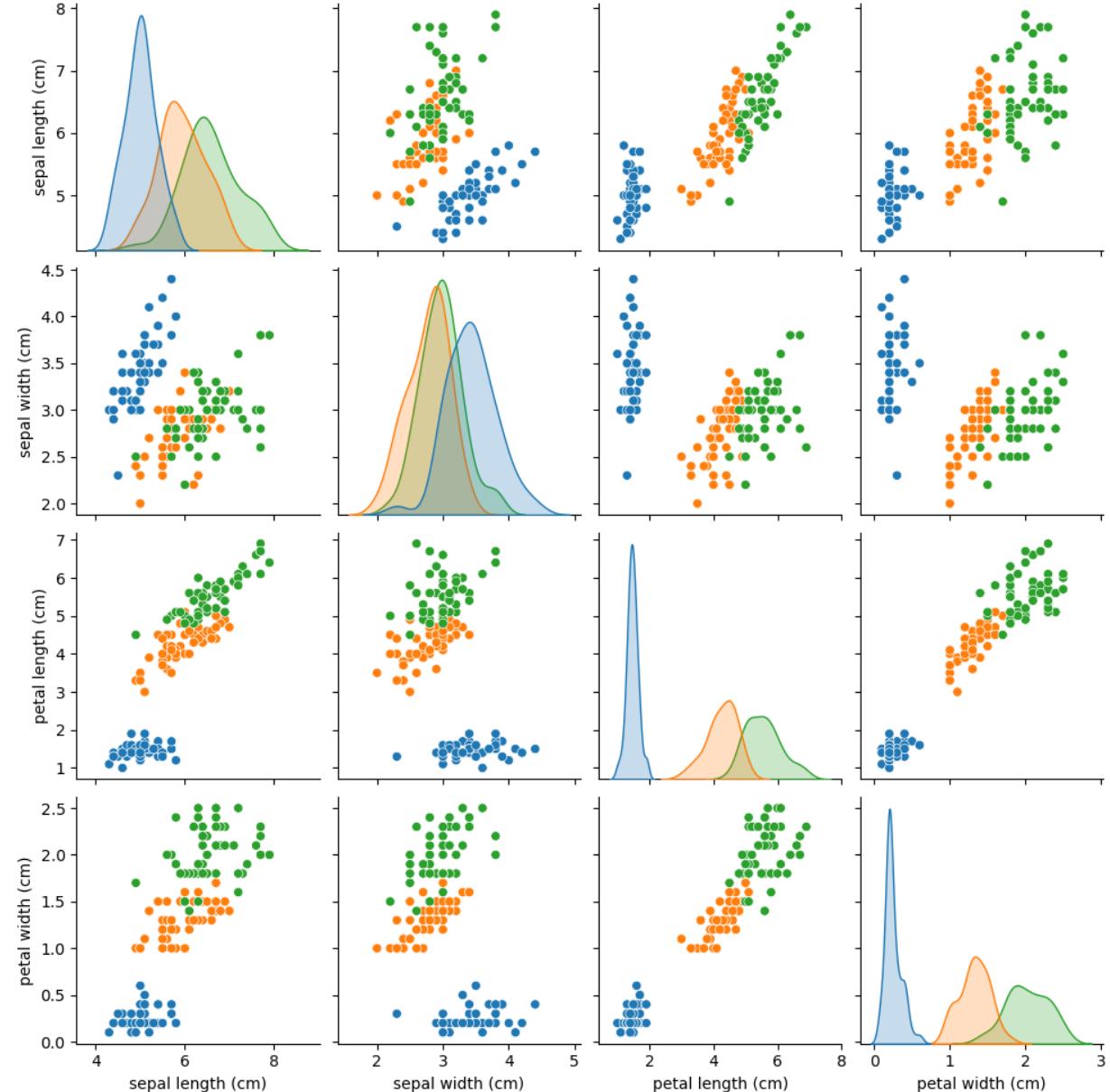
- 目的変数 (クラス)
 - Setosa
 - Versicolor
 - Virginica
- 説明変数
 - Sepal Length (萼片の長さ)
 - Sepal Width (萼片の幅)
 - Petal Length (花弁の長さ)
 - Petal Width (花弁の幅)

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	label
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

9-3 | 3. データの可視化

```
# 散布図行列の表示（クラスごとに色分け）
sns.pairplot(df_iris, hue="label")
plt.show()
```

label
● setosa
● versicolor
● virginica



9-3 | 4. モデルの構築・学習

```
# 学習用・テスト用の分割
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.4, random_state=1234
)

# XGBoostモデルの構築
# learning_rate: 学習率を指定
# objective: 出力時に適用する関数
clf = XGBClassifier(
    n_estimators=100, learning_rate=0.1, max_depth=10,
    objective="multi:softmax", random_state=1234)

# モデルの学習
clf.fit(X_train, y_train)

# 訓練性能の確認
print("train_accuracy:", clf.score(X_train, y_train.reshape(-1,1)))
```

- objective
 - モデルの出力形式
 - 多クラス場合は"multi:softmax" (ソフトマックス関数)
 - 二値分類の場合は"binary:logistic" (シグモイド関数)
- n_estimators
 - ブースティングのステージ数
 - 構築する弱学習器の数
- learning_rate
 - ブースティングのステージごとの学習率

```
train_accuracy: 1.0
```

9-3 | 5. モデルの評価

```
# 汎化性能の確認
print("test_accuracy:", clf.score(X_test, y_test))

# 予測結果の取得
y_pred = clf.predict(X_test)

# 評価指標の確認
scores = classification_report(y_test, y_pred)
print(scores)
```

```
test_accuracy: 0.9833333333333333
      precision    recall  f1-score   support
          0       1.00     1.00     1.00      19
          1       1.00     0.96     0.98      23
          2       0.95     1.00     0.97      18

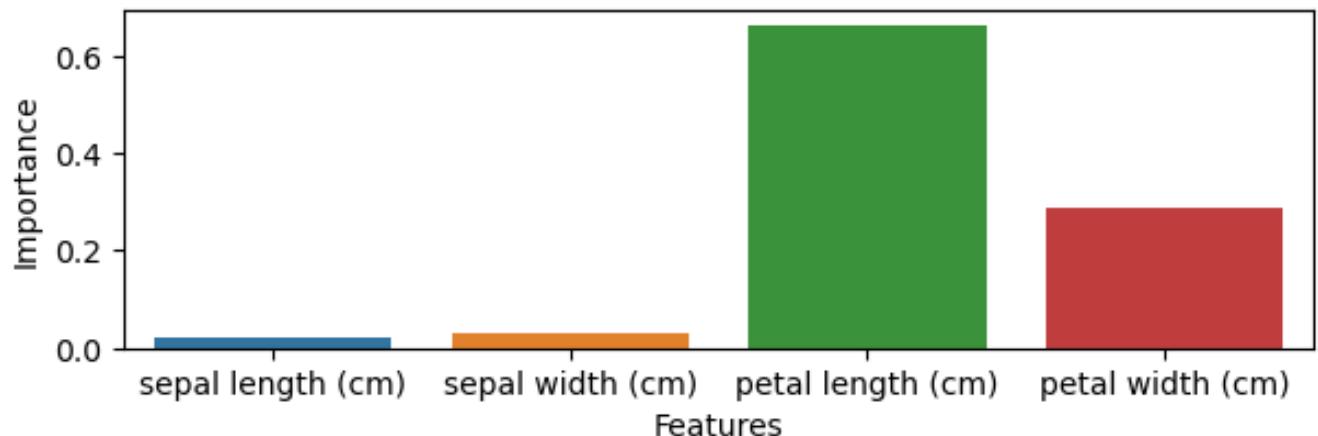
   accuracy                           0.98      60
  macro avg       0.98     0.99     0.98      60
weighted avg       0.98     0.98     0.98      60
```

9-3 | 6. 変数の重要度を確認

```
# 説明変数の重要度を出力する
# 重要度=ある説明変数による不純度の減少量の合計
print(clf.feature_importances_)

# データフレームの作成
feature_imp = pd.DataFrame(
    clf.feature_importances_,
    index=iris.feature_names, columns=['Importance']
)

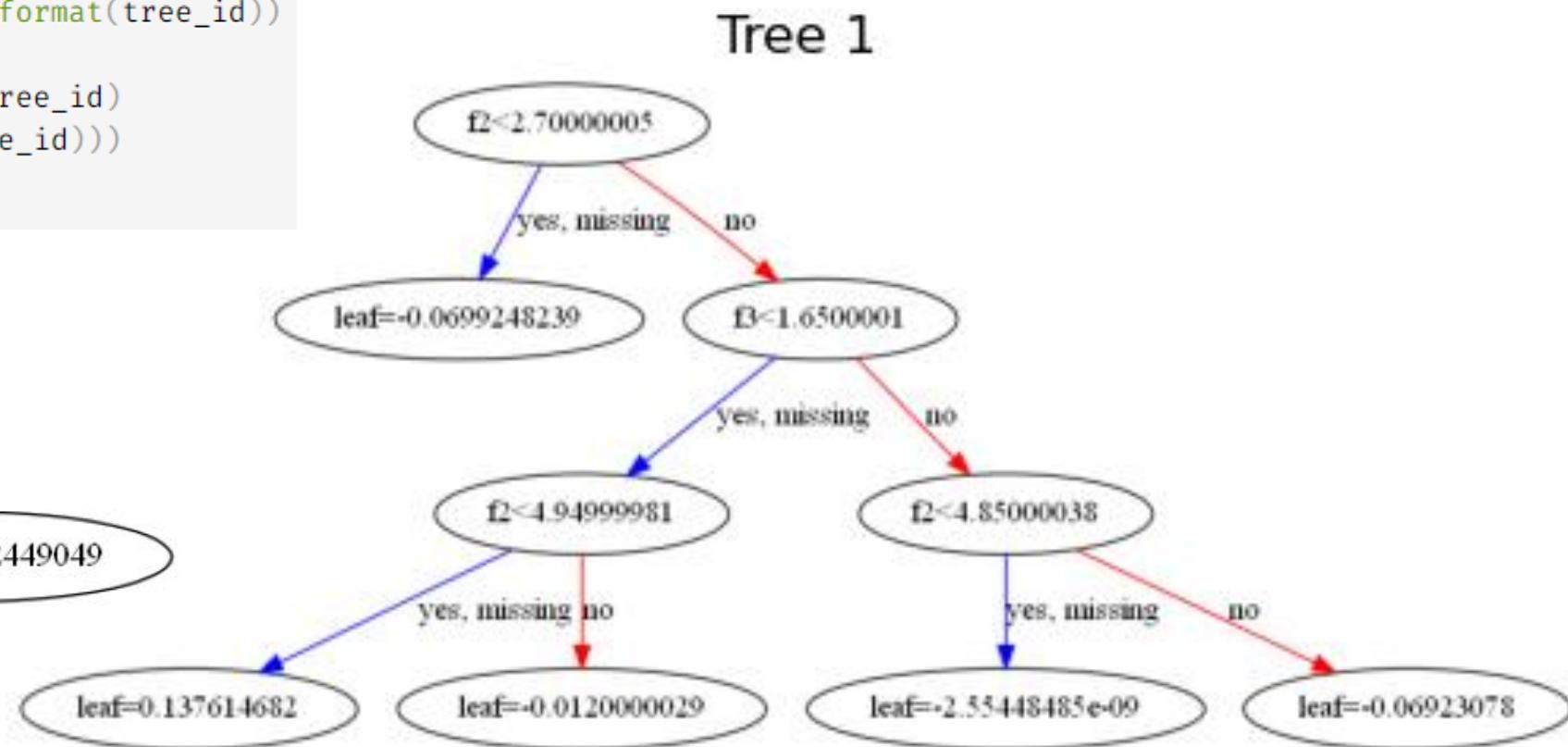
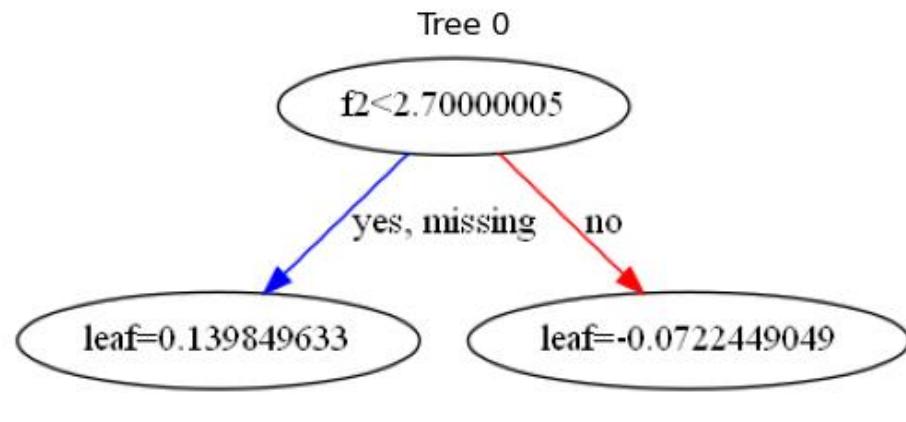
# 棒グラフの表示
plt.figure(figsize=(7,2))
sns.barplot(x=feature_imp.index, y=feature_imp['Importance'])
plt.ylabel("Importance")
plt.xlabel("Features")
plt.show()
```



9-3 | 7. モデルの可視化

```
# 木の描画本数
plot_tree_num = 10

# 木を可視化
for tree_id in range(plot_tree_num):
    print("Rendering Tree {:d} ... ".format(tree_id))
    # 図の作成・表示
    ax = plot_tree(clf, num_trees=tree_id)
    ax.set_title(("Tree " + str(tree_id)))
    plt.plot()
```



現場で使える 機械学習・データ分析基礎講座

第 10 章：サポートベクターマシン
ノートブック解説

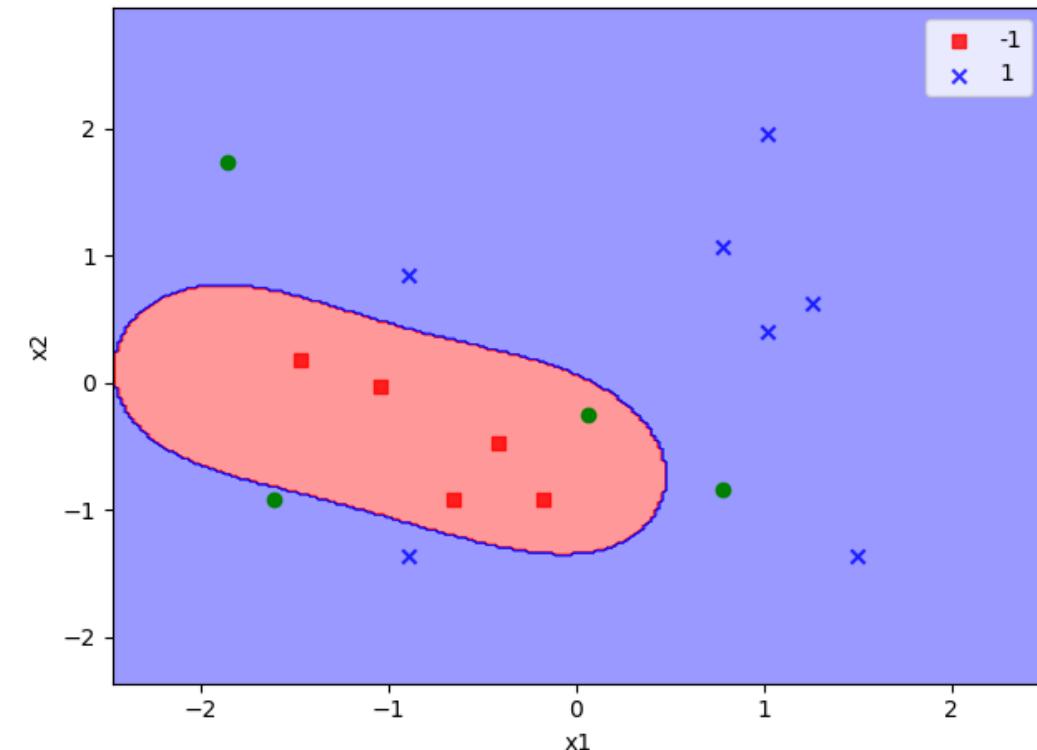
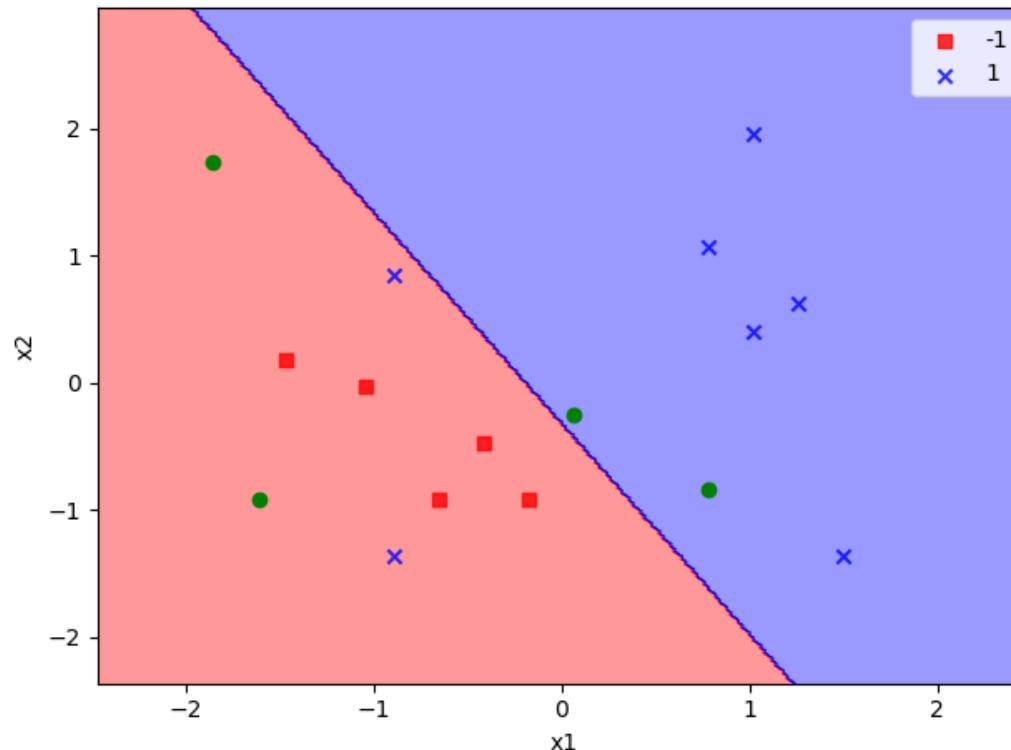
■ サポートベクターマシン

- 10-1_SVM_practice.ipynb
- 10-2_SVM_kernel_function.ipynb

10-1_SVM_practice

■ 内容

- ・ サポートベクターマシンを用いて、様々なデータを分類
- ・ カーネル関数の効果を確認
- ・ グリッドサーチ（ハイパーパラメータ探索）の実装方法を確認



■ 手順

1. ライブラリの読み込み
2. 線形分離可能なデータの場合
 1. 疑似データの作成
 2. モデルの構築・学習
 3. 予測結果の可視化
 4. 予測値の確認
3. 線形分離不可能なデータの場合
 1. 疑似データの作成
 2. モデルの構築・学習
 3. 予測結果の可視化
 4. 予測値の確認
4. カーネル関数の適用
 1. モデルの構築・学習
 2. 予測結果の可視化
5. ハイパーパラメータ探索
 1. データの読み込み
 2. データの前処理
 3. グリッドサーチの実行
 4. モデルの評価

10-1 | 1. ライブラリの読み込み

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

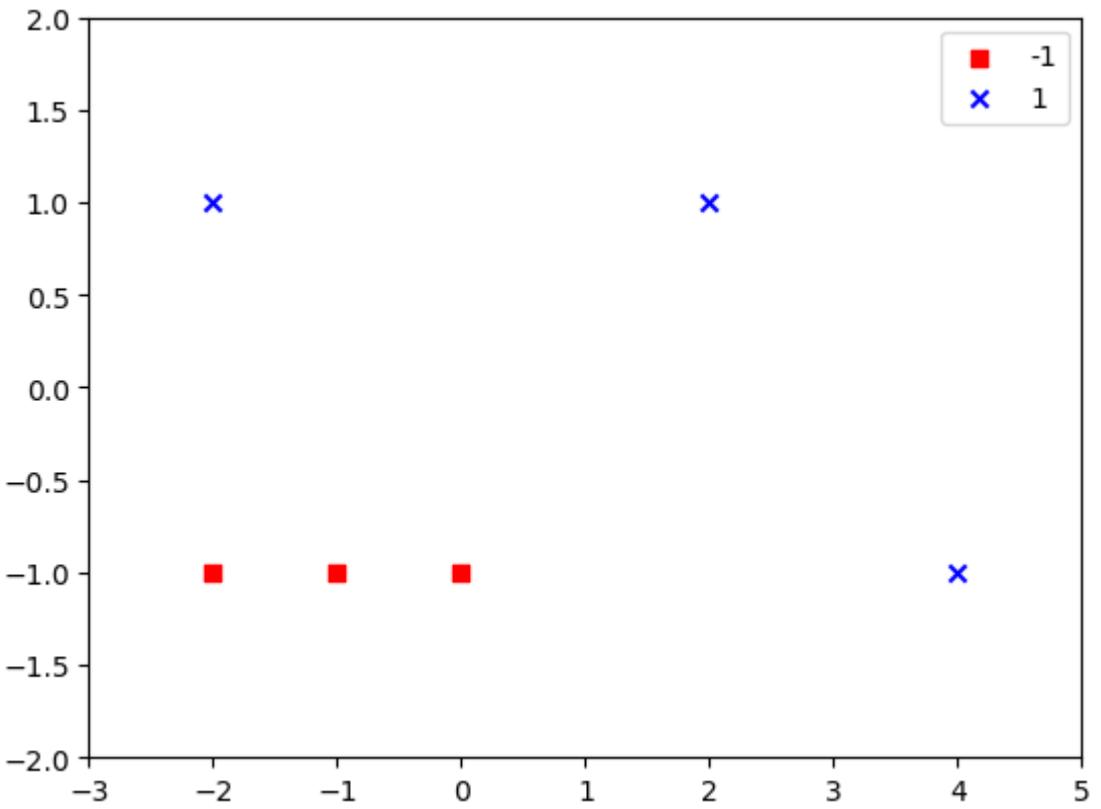
# 標準化
from sklearn.preprocessing import StandardScaler
# ホールドアウト法、グリッドサーチ
from sklearn.model_selection import train_test_split, GridSearchCV

# SVMを用いた分類モデル
from sklearn.svm import SVC
# 分類境界と各クラスの領域を表示
from common_func import plot_decision_regions
```

10-1 | 2-1. 疑似データの作成

```
# 疑似データの作成
X = np.array([
    [-1, -1], [-2, -1], [ 0, -1],
    [ 2,  1], [-2,  1], [ 4, -1]
])
y = np.array([-1, -1, -1,  1,  1,  1])

# クラス-1のデータを表示
plt.scatter(x=X[:3, 0], y=X[:3, 1],
            color="r", marker="s", label="-1")
# クラス1のデータを表示
plt.scatter(x=X[3:, 0], y=X[3:, 1],
            color="b", marker="x", label="1")
# 表示範囲の調整
plt.xlim(-3, 5)
plt.ylim(-2, 2)
plt.legend()
plt.show()
```



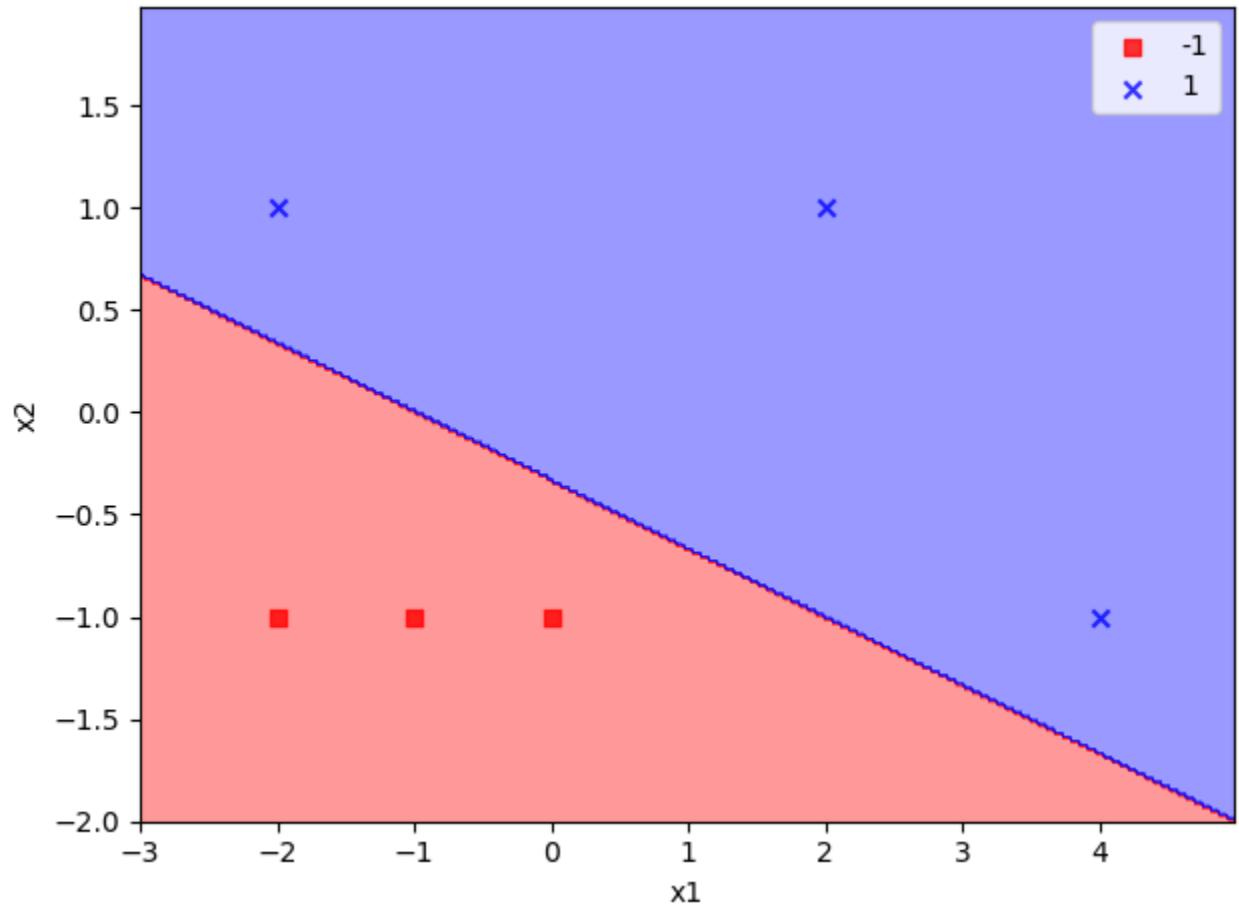
```
# ペナルティ項の強さを設定  
C = 10  
  
# モデルの構築  
# kernel:カーネル関数  
clf = SVC(C=C, kernel="linear")  
  
# モデルの学習  
clf.fit(X, y)
```

```
▼ SVC  
SVC(C=10, kernel='linear')
```

10-1 | 2-3. 予測結果の可視化

```
# 決定領域を描画
plot_decision_regions(X, y, classifier=clf)

# ラベルの設定
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(loc='upper right')
# 表示位置を調整
plt.tight_layout()
plt.show()
```

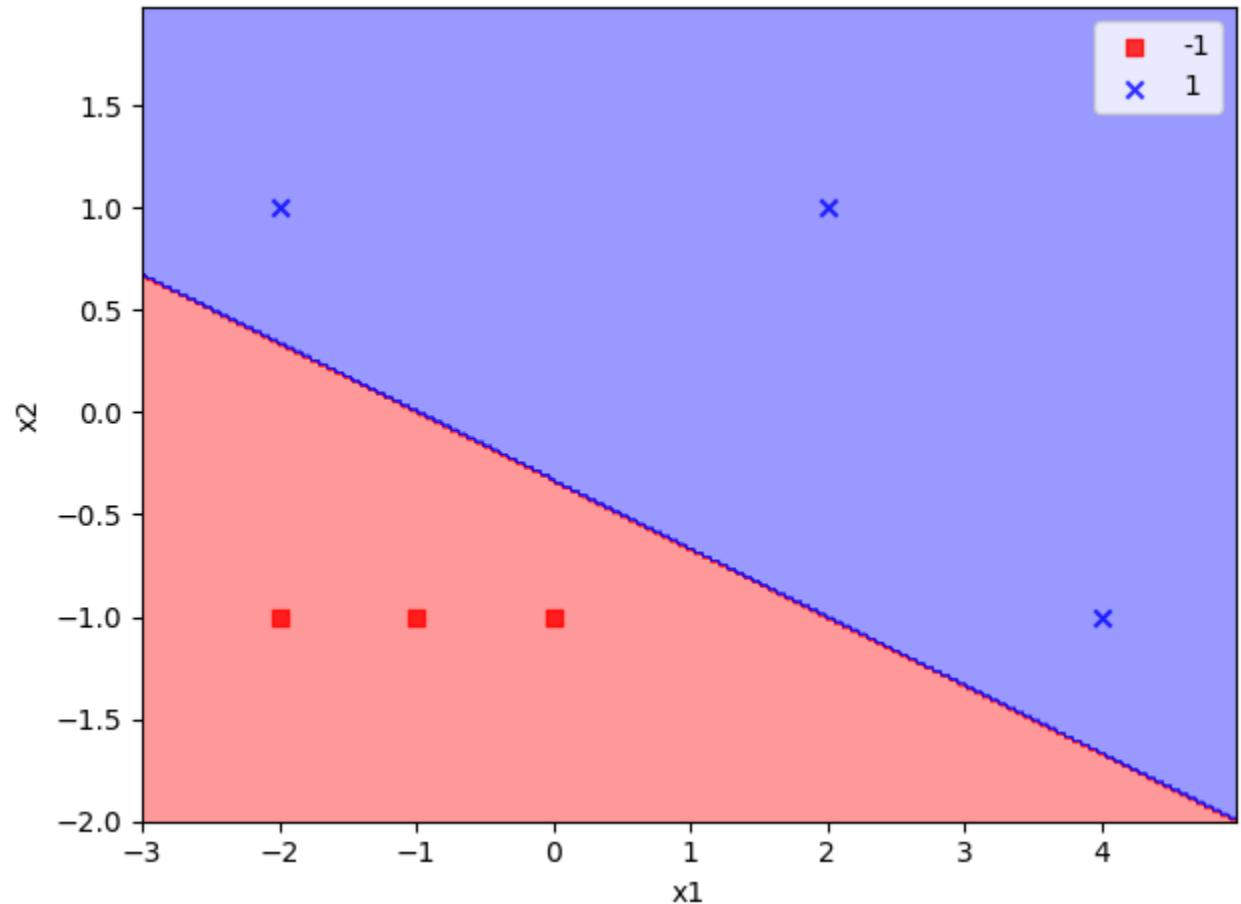


10-1 | 2-4. 予測値の確認

```
# 線形識別関数（超平面）上の値を確認
# 値が0未満ならば、y=-1と判定される領域にいる
# 値が0以上ならば、y=1と判定される領域にいる
y_pred = np.dot(X, clf.coef_.T) + clf.intercept_
y_pred
```

```
array([[-1.49973333],
       [-1.99973333],
       [-0.99973333],
       [ 2.99946667],
       [ 0.99946667],
       [ 1.00026667]])
```

```
# 疑似データの作成
X = np.array([
    [-1, -1], [-2, -1], [ 0, -1],
    [ 2,  1], [-2,  1], [ 4, -1]
])
y = np.array([-1, -1, -1,  1,  1,  1])
```



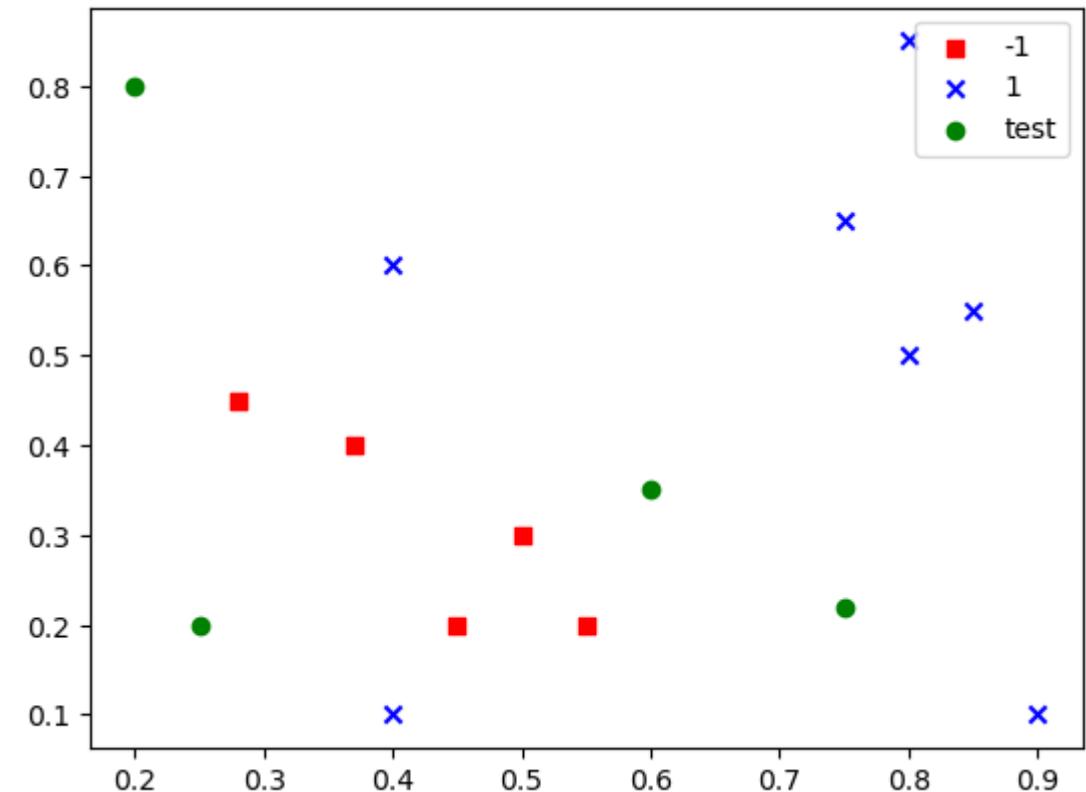
10-1 | 3-1. 疑似データの作成

```
# 学習用データの作成
df_simple = pd.DataFrame({
    "label": [1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1],
    "x1": [0.4, 0.9, 0.8, 0.75, 0.85, 0.4, 0.8, 0.5, 0.28, 0.37, 0.45, 0.55],
    "x2": [0.1, 0.1, 0.85, 0.65, 0.55, 0.6, 0.5, 0.3, 0.45, 0.4, 0.2, 0.2]
})
# 説明変数・目的変数に分割
X_train = df_simple[["x1", "x2"]].values
y_train = df_simple["label"].values

# テスト用データの作成
X_test = np.array([
    [0.6, 0.35], [0.75, 0.22], [0.2, 0.8], [0.25, 0.2]
])

# クラス-1のデータを表示
plt.scatter(x=X_train[7:, 0], y=X_train[7:, 1],
            color="r", marker="s", label="-1")
# クラス1のデータを表示
plt.scatter(x=X_train[:7, 0], y=X_train[:7, 1],
            color="b", marker="x", label="1")
# テスト用データを表示
plt.scatter(x=X_test[:, 0], y=X_test[:, 1],
            color="g", marker="o", label="test")

# 表示位置を調整
plt.legend(loc='upper right')
plt.show()
```



```
# 標準化
stdsc = StandardScaler()
X_train_s = stdsc.fit_transform(X_train)
X_test_s = stdsc.transform(X_test)

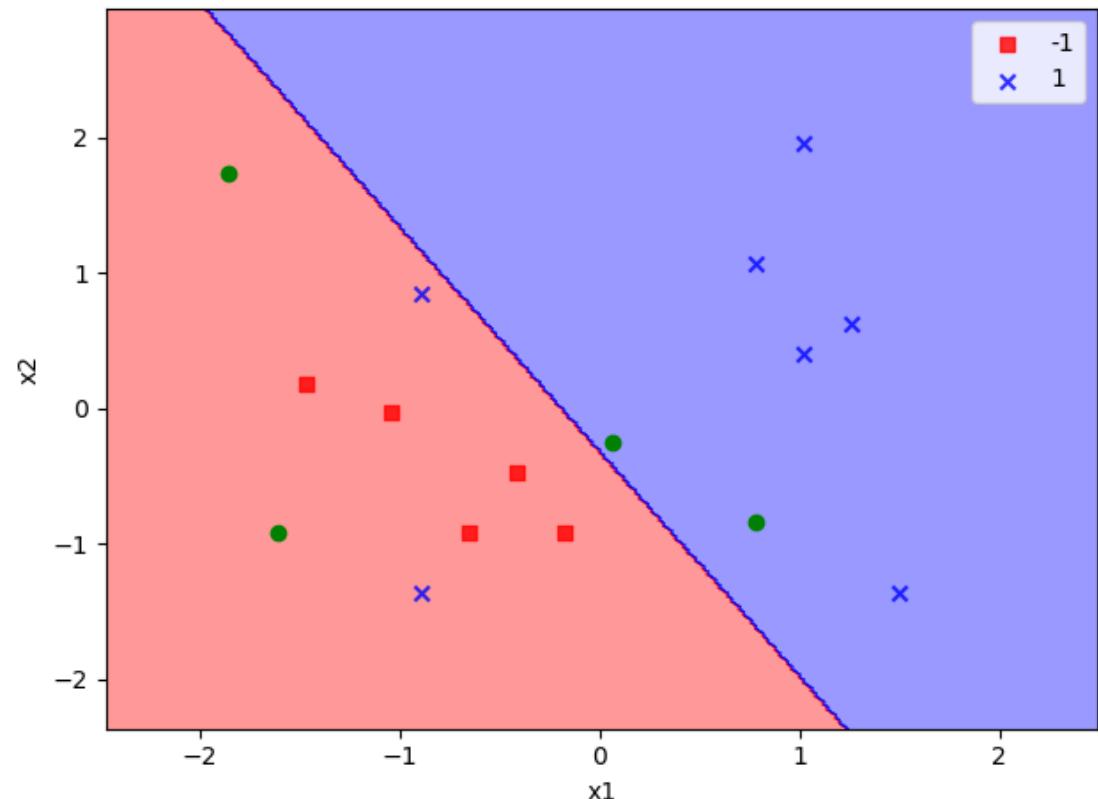
# モデルの構築
C = 5
clf = SVC(C=C, kernel="linear")
# モデルの学習
clf.fit(X_train_s, y_train)
```

```
▼ SVC
SVC(C=5, kernel='linear')
```

10-1 | 3-3. 予測結果の可視化

```
# 未知のデータを識別する
clf.predict(X_test_s)
# 決定領域を描画する
plot_decision_regions(X_train_s, y_train, classifier=clf)
# テスト用データを重ねて表示
plt.plot(X_test_s[:,0], X_test_s[:,1], color="g", marker="o", ls="")
```

```
# ラベルの設定
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(loc='upper right')
# 表示位置を調整
plt.tight_layout()
plt.show()
```

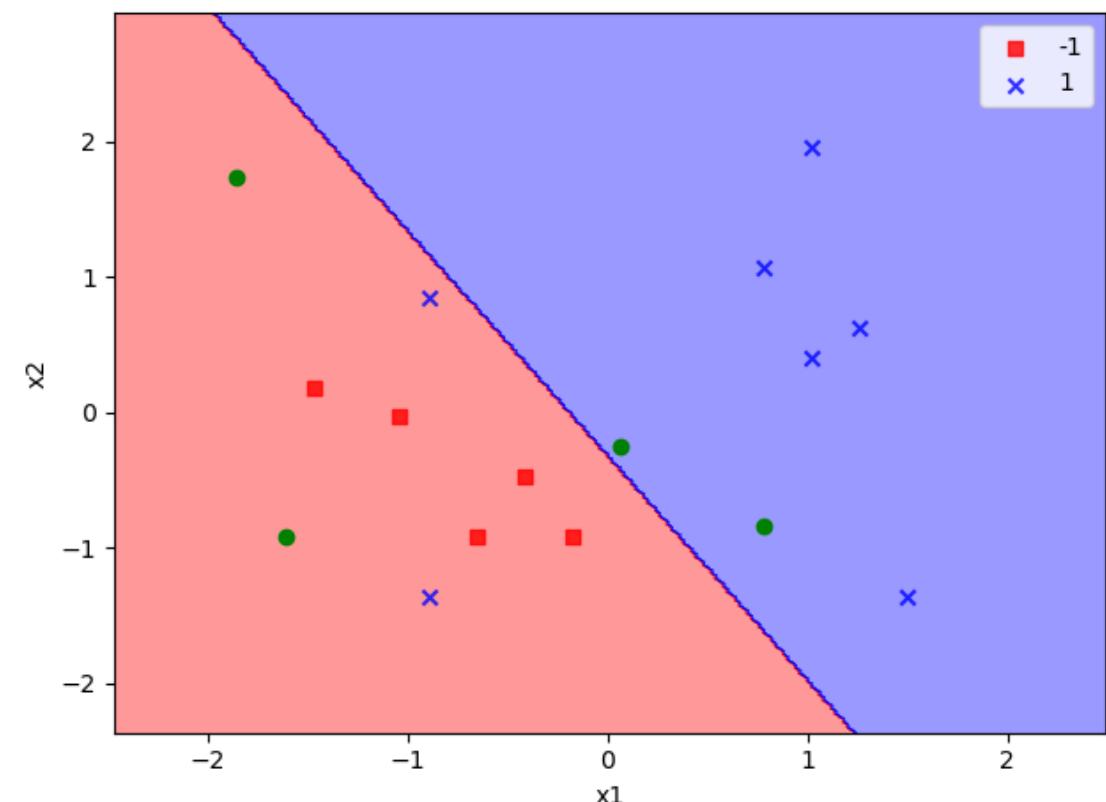


10-1 | 3-4. 予測値の確認

```
# 線形識別関数（超平面）上の値を確認
# 値が0未満ならば、y=-1と判定される領域にいる
# 値が0以上ならば、y=1と判定される領域にいる
np.dot(X_train, clf.coef_.T) + clf.intercept_
```

```
array([[-1.75799461],
       [ 0.99999971],
       [ 2.75708702],
       [ 1.86563794],
       [ 2.10941198],
       [-0.2188705 ],
       [ 1.67970014],
       [-0.5907461 ],
       [-1.34252637],
       [-0.9999998 ],
       [-1.17437036],
       [-0.62277149]])
```

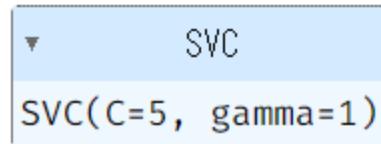
```
# 学習用データの作成
df_simple = pd.DataFrame({
    "label": [1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1],
    "x1": [0.4, 0.9, 0.8, 0.75, 0.85, 0.4, 0.8, 0.5, 0.28, 0.37, 0.45, 0.55],
    "x2": [0.1, 0.1, 0.85, 0.65, 0.55, 0.6, 0.5, 0.3, 0.45, 0.4, 0.2, 0.2]
})
```



```
# ペナルティ項の強さを設定
C = 5

# RBF: Radial Basis Function, 動径既定関数
# ガウスカーネルと同様のもの
kernel = "rbf"
# RBFのパラメータgammaを設定
gamma = 1

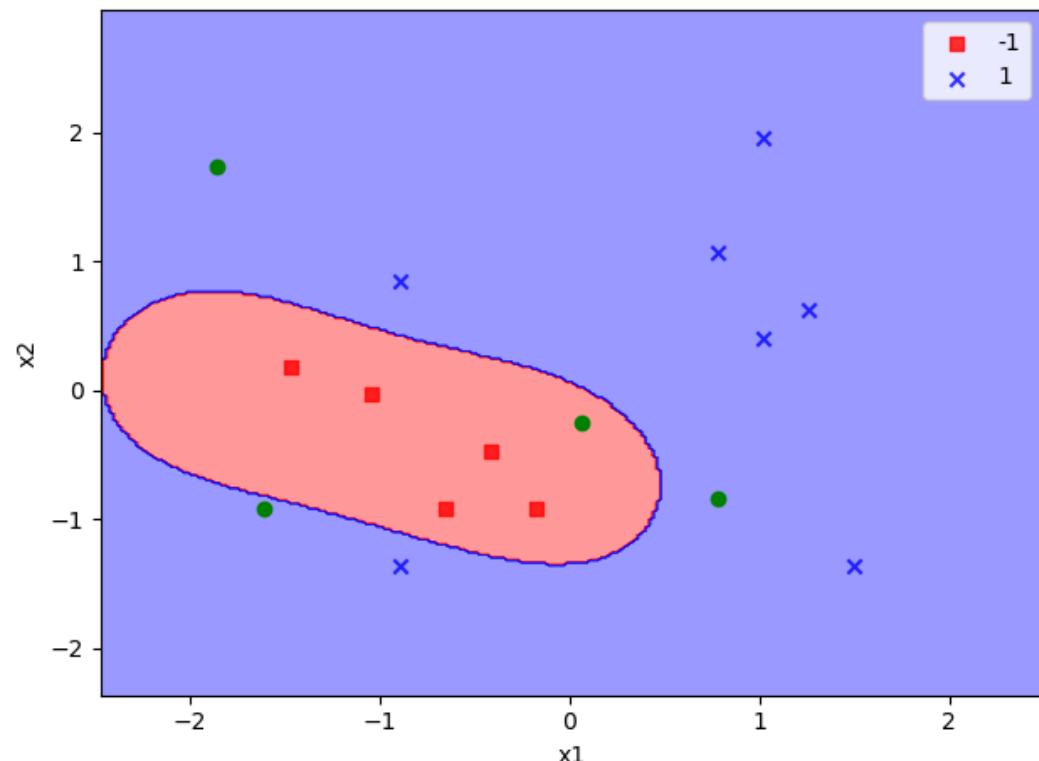
# モデルの構築
clf = SVC(C=C, kernel=kernel, gamma=gamma)
# モデルの学習
clf.fit(X_train_s, y_train)
```



10-1 | 4-2. 予測結果の可視化

```
# 未知のデータを識別する
clf.predict(X_test_s)
# 決定領域を描画する
plot_decision_regions(X_train_s, y_train, classifier=clf)
# テスト用データを重ねて表示
plt.plot(X_test_s[:,0], X_test_s[:,1], color="g", marker="o", ls="")

# ラベルの設定
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend(loc='upper right')
# 表示位置を調整
plt.tight_layout()
plt.show()
```



```
# CSVファイルの読み込み
df = pd.read_csv("../.. / 1_data/ch10/phishing_website_dataset.csv")
```

```
# データセットの中身を確認
print(df.shape)
df.head()
```

(11055, 31)

■ Phishing WebSite Dataset

- フィッシングサイトかどうかを識別する、二値分類のデータセット

■ 目的変数

- Result : あるサイトがフィッシングサイトであるか

■ 説明変数

- 30 個 (すべてカテゴリ変数)

	having IP Address	URL Length	Shortining Service	having At Symbol	double slash redirecting	Prefix Suffix	having Sub Domain	SSLfinal State	Domain registration length	Favicon	...	popUpWidnow	Iframe	age of domain	DNSRecord	web traffic	Page Rank	Google Index	Links pointing to page	Statistical report	Result
0	-1	1	1	1	-1	-1	-1	-1	-1	1	...	1	1	-1	-1	-1	-1	1	-1	-1	
1	1	1	1	1	1	-1	0	1	-1	1	...	1	1	-1	-1	-1	0	-1	1	-1	
2	1	0	1	1	1	-1	-1	-1	-1	1	...	1	1	1	-1	-1	1	-1	0	-1	
3	1	0	1	1	1	-1	-1	-1	1	1	...	1	1	-1	-1	-1	1	-1	1	-1	
4	1	0	-1	1	1	-1	1	1	-1	1	...	1	1	-1	-1	-1	0	-1	1	1	

```
# 説明変数/目的変数の分割
X = df.drop("Result", axis=1).values
y = df["Result"].values

# 学習用/テスト用の分割
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

# 明らかに有効な説明変数(7番目、13番目)を除いておく
indices = list(set(range(30)) - set([7, 13]))
```

7. SSLfinal_State
13. URL_of_Anchor

```
X_train = X_train[:, indices]
X_test = X_test[:, indices]
```

10-1 | 5-3. グリッドサーチの実行

```
# 探索するパラメータ（ここを編集）
parameters = {
    'kernel':['linear', 'rbf'],
    'C':[1, 5]
}

# モデルの構築
# gamma=scaleに設定すると、gammaが自動で計算される
model = SVC(gamma="scale")

# グリッドサーチ&交差検証を行う
# 交差検証のグループ数は3に設定
clf = GridSearchCV(model, parameters, cv=3,)

# ハイパーパラメータ探索の実行
clf.fit(X_train, y_train)

# 最良のパラメータと、そのときの評価値を表示
print(clf.best_params_, clf.best_score_)
```

```
{'C': 5, 'kernel': 'rbf'} 0.8904097696260695
```

```
# 最適なハイパーパラメータをもつモデルを取得
clf_best = clf.best_estimator_

# テスト用データで汎化性能を確認
print("test_accuracy: ", clf_best.score(X_test, y_test))
```

```
test_accuracy:  0.9023213747362074
```

10-2_SVM_kernel_function

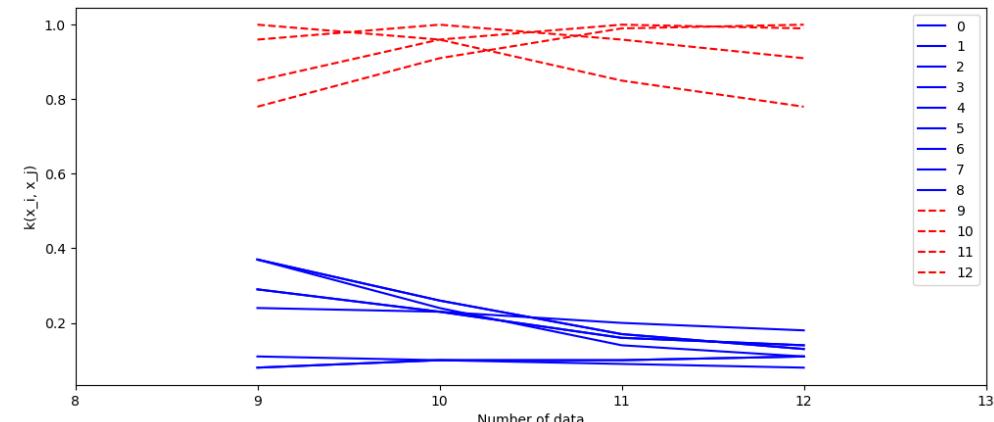
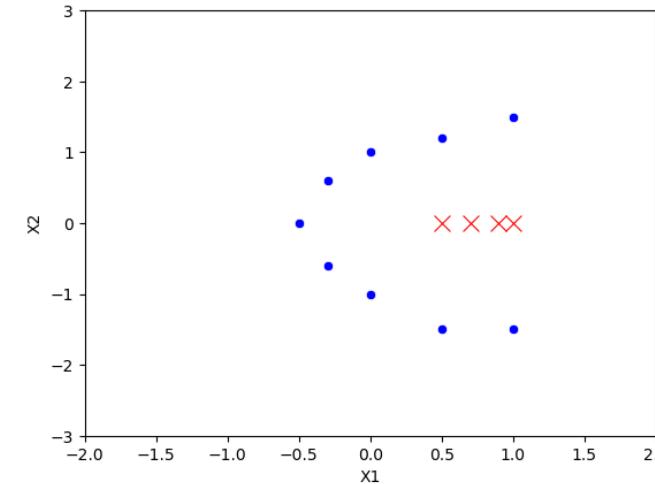
10-2_SVM_kernel_function

■ 内容

- 疑似データに対して線形カーネルとガウスカーネルを適用し、その性質を確認

■ 手順

1. ライブラリの読み込み
2. ガウスカーネル関数のグラフ
3. カーネル関数の適用 (2 次元)
 1. 疑似データの作成
 2. データの可視化
 3. 線形カーネル関数の適用
 4. 結果の確認 (線形カーネル)
 5. ガウスカーネル関数の適用
 6. 結果の確認 (ガウスカーネル)



```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

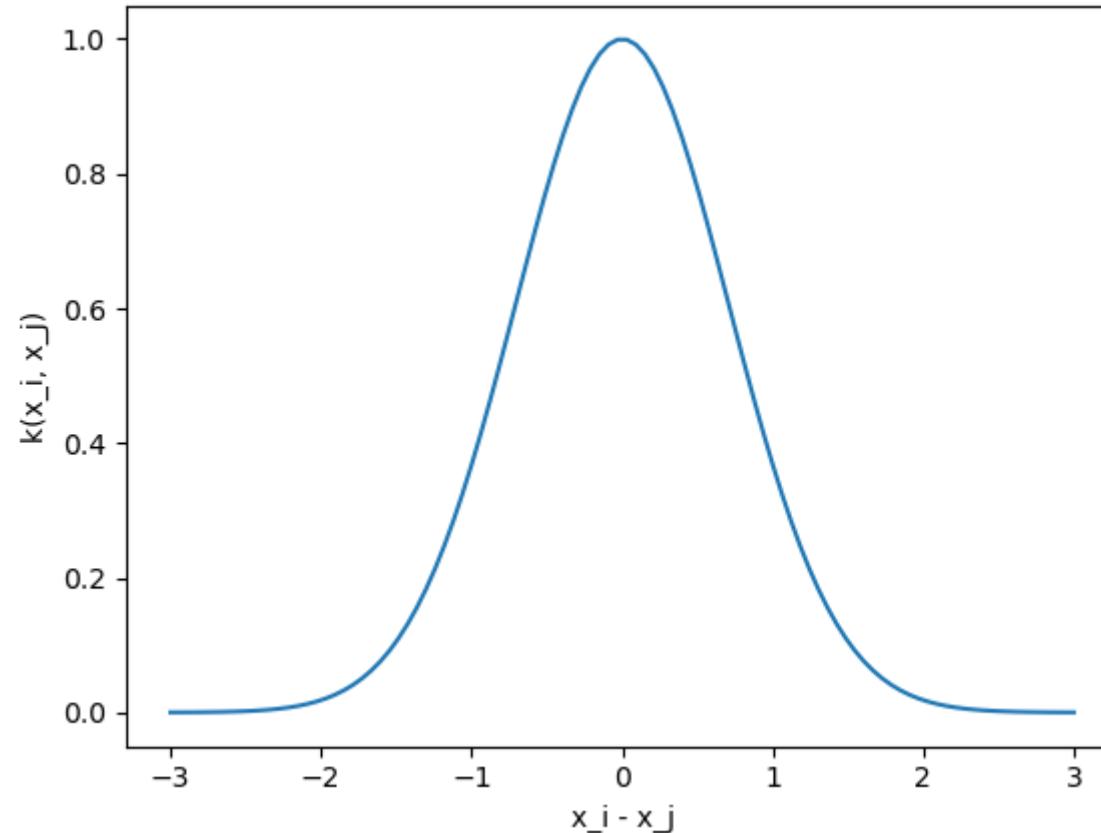
10-2 | 2. ガウスカーネル関数のグラフ

```
# パラメータの設定
gamma = 1

# 横軸の値を生成
x = np.linspace(-3,3,100)
# 縦軸の値を計算
y = np.exp(-gamma*np.power(x,2))

# グラフの表示
sns.lineplot(x=x,y=y)
plt.ylabel("k(x_i, x_j)")
plt.xlabel("x_i - x_j")
plt.show()
```

• $y = \exp(-\gamma x^2)$ とおく



10-2 | 3-1. 疑似データの作成

```
# データを作成する関数
def make_dataset(DATA_TYPE):
    if DATA_TYPE==1:
        """
        線形分離可能なデータセット
        """
        X1 = np.array([
            [-0.5, -0.3, -0.3, -0.1, -0.1, 0.1, 0.1, 0.3, 0.3],
            [ 0, -0.6, 0.6, -1.3, 1.3, -1.9, 1.9, -2.5, 2.5]])
        X2 = np.array([
            [ 0.5, 0.7, 0.9, 1.0],
            [ 0, 0, 0, 0]])
    elif DATA_TYPE==2:
        """
        線形分離不可能なデータセット
        """
        X1 = np.array([
            [-0.3, -0.3, 0.5, 0.5, 0, 0, -0.5, 1, 1],
            [-0.6, 0.6, -1.5, 1.2, -1, 1, 0, -1.5, 1.5]])
        X2 = np.array([
            [ 0.5, 0.7, 0.9, 1.0],
            [ 0, 0, 0, 0]])
    return X1, X2
```

10-2 | 3-1. 疑似データの作成

```
# 実際にデータを生成
X1, X2 = make_dataset(DATA_TYPE=2) # ここを変更

# 1つの配列として結合
X = np.hstack((X1,X2))

# データフレームに変換
pd.DataFrame(X)
```

DATA_TYPE=1

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	-0.5	-0.3	-0.3	-0.1	-0.1	0.1	0.1	0.3	0.3	0.5	0.7	0.9	1.0
1	0.0	-0.6	0.6	-1.3	1.3	-1.9	1.9	-2.5	2.5	0.0	0.0	0.0	0.0

DATA_TYPE=2

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	-0.3	-0.3	0.5	0.5	0.0	0.0	-0.5	1.0	1.0	0.5	0.7	0.9	1.0
1	-0.6	0.6	-1.5	1.2	-1.0	1.0	0.0	-1.5	1.5	0.0	0.0	0.0	0.0

10-2 | 3-2. データの可視化

```
# 散布図の表示 (X1=○、 X2=x)
sns.scatterplot(x=X1[0],y=X1[1],marker="o",color="b")
sns.scatterplot(x=X2[0],y=X2[1],marker="x",color="r",s=100)
```

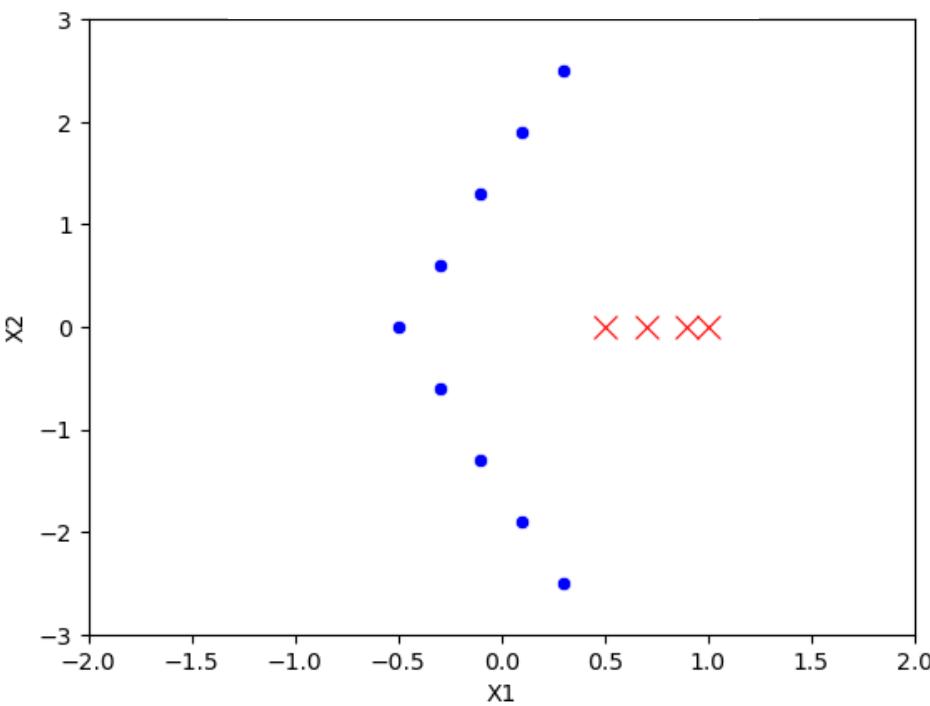
```
# 表示範囲の調整
```

```
plt.xlim([-2,2])
plt.ylim([-3,3])
```

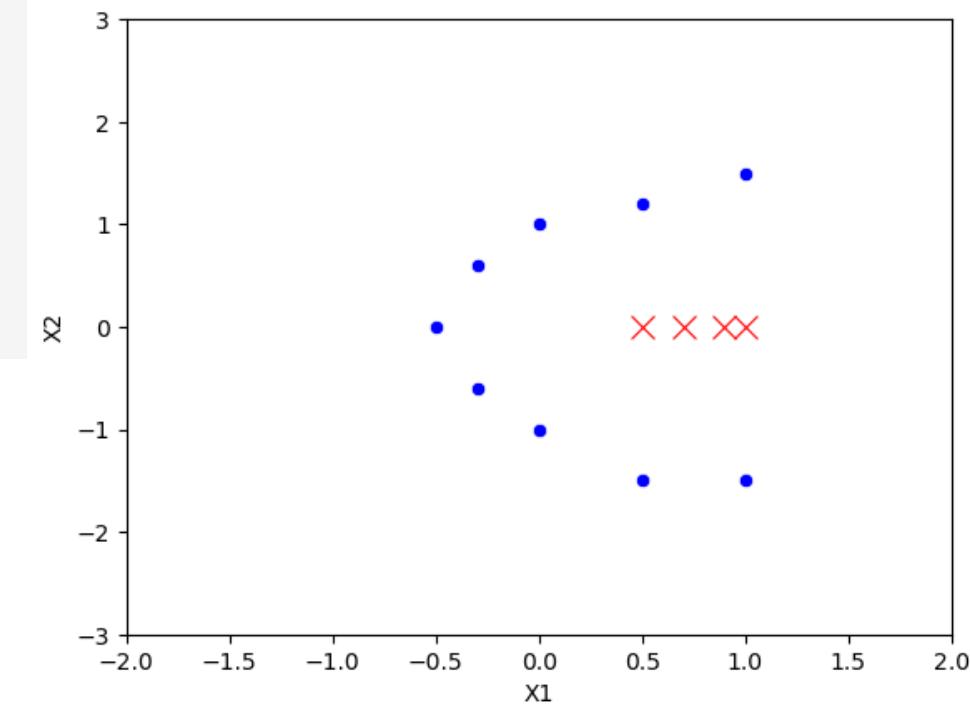
```
# 軸ラベルの設定
```

```
plt.xlabel("X1")
plt.ylabel("X2")
plt.show()
```

DATA_TYPE=1



DATA_TYPE=2



10-2 | 3-3. 線形カーネル関数の適用

```
# 関数の定義
def k_linear(x, x_):
    """
    線形カーネル関数
    """
    return np.dot(x, x_)

# データの個数を取得
N = X.shape[1]
# 空のグラム行列を作成
K = np.zeros([N, N])

# 関数の計算
for i in range(N):
    x = X[:, i]
    for j in range(N):
        x_ = X[:, j]
        k = k_linear(x, x_)
        K[i, j] = k

# グラム行列の表示
df = pd.DataFrame(K)
df
```

DATA_TYPE=2

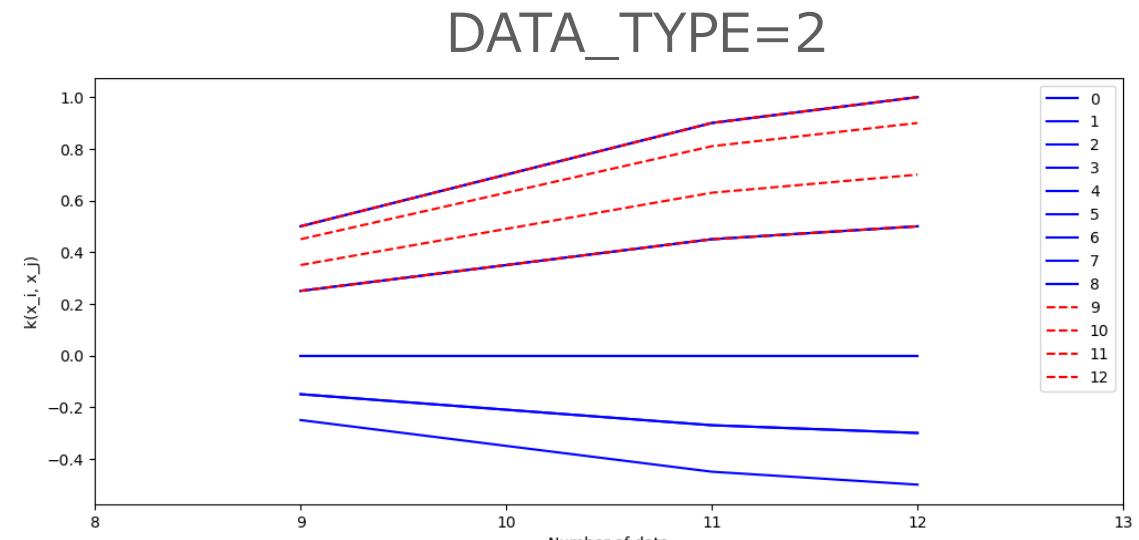
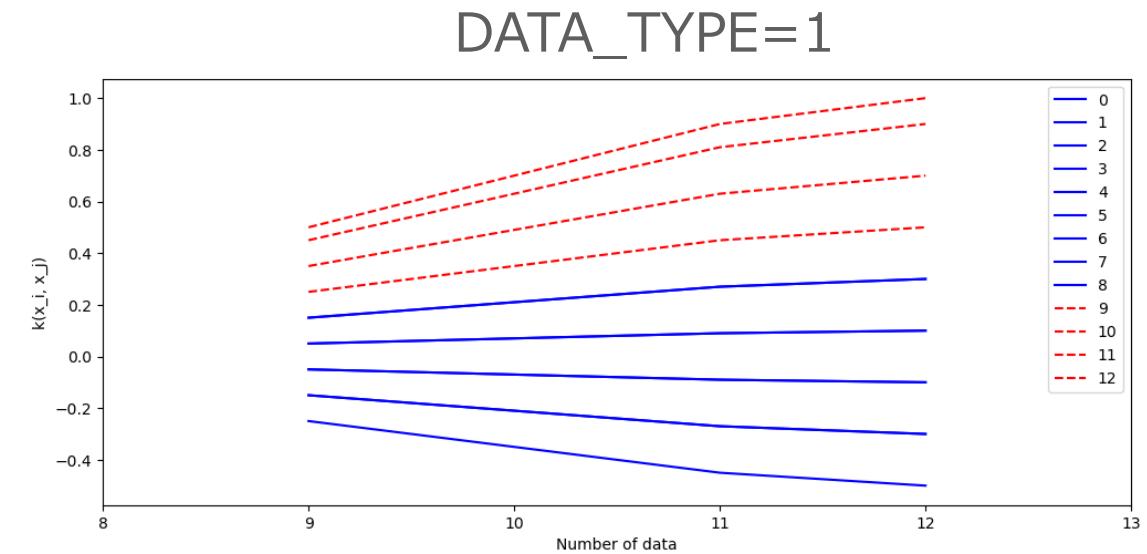
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0.45	-0.27	0.75	-0.87	0.6	-0.6	0.15	0.60	-1.20	-0.15	-0.21	-0.27	-0.3
1	-0.27	0.45	-1.05	0.57	-0.6	0.6	0.15	-1.20	0.60	-0.15	-0.21	-0.27	-0.3
2	0.75	-1.05	2.50	-1.55	1.5	-1.5	-0.25	2.75	-1.75	0.25	0.35	0.45	0.5
3	-0.87	0.57	-1.55	1.69	-1.2	1.2	-0.25	-1.30	2.30	0.25	0.35	0.45	0.5
4	0.60	-0.60	1.50	-1.20	1.0	-1.0	0.00	1.50	-1.50	0.00	0.00	0.00	0.0
5	-0.60	0.60	-1.50	1.20	-1.0	1.0	0.00	-1.50	1.50	0.00	0.00	0.00	0.0
6	0.15	0.15	-0.25	-0.25	0.0	0.0	0.25	-0.50	-0.50	-0.25	-0.35	-0.45	-0.5
7	0.60	-1.20	2.75	-1.30	1.5	-1.5	-0.50	3.25	-1.25	0.50	0.70	0.90	1.0
8	-1.20	0.60	-1.75	2.30	-1.5	1.5	-0.50	-1.25	3.25	0.50	0.70	0.90	1.0
9	-0.15	-0.15	0.25	0.25	0.0	0.0	-0.25	0.50	0.50	0.25	0.35	0.45	0.5
10	-0.21	-0.21	0.35	0.35	0.0	0.0	-0.35	0.70	0.70	0.35	0.49	0.63	0.7
11	-0.27	-0.27	0.45	0.45	0.0	0.0	-0.45	0.90	0.90	0.45	0.63	0.81	0.9
12	-0.30	-0.30	0.50	0.50	0.0	0.0	-0.50	1.00	1.00	0.50	0.70	0.90	1.0

10-2 | 3-4. 結果の確認（線形カーネル）

```
# X1-X2の類似度をグラフ化（青線）
# 9~12行目 (X2)、0~9列目 (X1)
ax1 = df.iloc[9:, :9].plot(color="b", figsize=(12,5))

# X2-X2の類似度を重ねて表示（赤線）
# 9~12行目 (X2)、9~12列目 (X2)
df.iloc[9:, 9: ].plot(color="r", ax=ax1, ls="--")

# ラベルの設定
plt.ylabel("k(x_i, x_j)") # 類似度
plt.xlabel("Number of data") # データの片方
plt.legend(loc="best")
# 表示範囲の調整
plt.xlim([8, 13])
plt.show()
```



10-2 | 3-5. ガウスカーネル関数の適用

```
# カーネル関数の定義
def k_gauss(x, x_, gamma):
    """
    ガウスカーネル関数
    """
    return np.exp(-gamma*np.power(np.linalg.norm(x - x_), 2))

# パラメータの設定
gamma = 1 # ここを変更

# データの個数を取得
N = X.shape[1]
# 空のグラム行列を作成
K = np.zeros([N,N])

# 関数の計算
for i in range(N):
    x = X[:,i]
    for j in range(N):
        x_ = X[:,j]
        k = k_gauss(x, x_, gamma=gamma)
        K[i, j] = k

# グラム行列の表示
df = pd.DataFrame(K.round(2))
df
```

• $y = \exp(-\gamma x^2)$ とおく

DATA_TYPE=2

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1.00	0.24	0.23	0.02	0.78	0.07	0.67	0.08	0.00	0.37	0.26	0.17	0.13
1	0.24	1.00	0.01	0.37	0.07	0.78	0.67	0.00	0.08	0.37	0.26	0.17	0.13
2	0.23	0.01	1.00	0.00	0.61	0.00	0.04	0.78	0.00	0.11	0.10	0.09	0.08
3	0.02	0.37	0.00	1.00	0.01	0.75	0.09	0.00	0.71	0.24	0.23	0.20	0.18
4	0.78	0.07	0.61	0.01	1.00	0.02	0.29	0.29	0.00	0.29	0.23	0.16	0.14
5	0.07	0.78	0.00	0.75	0.02	1.00	0.29	0.00	0.29	0.29	0.23	0.16	0.14
6	0.67	0.67	0.04	0.09	0.29	0.29	1.00	0.01	0.01	0.37	0.24	0.14	0.11
7	0.08	0.00	0.78	0.00	0.29	0.00	0.01	1.00	0.00	0.08	0.10	0.10	0.11
8	0.00	0.08	0.00	0.71	0.00	0.29	0.01	0.00	1.00	0.08	0.10	0.10	0.11
9	0.37	0.37	0.11	0.24	0.29	0.29	0.37	0.08	0.08	1.00	0.96	0.85	0.78
10	0.26	0.26	0.10	0.23	0.23	0.23	0.24	0.10	0.10	0.96	1.00	0.96	0.91
11	0.17	0.17	0.09	0.20	0.16	0.16	0.14	0.10	0.10	0.85	0.96	1.00	0.99
12	0.13	0.13	0.08	0.18	0.14	0.14	0.11	0.11	0.11	0.78	0.91	0.99	1.00

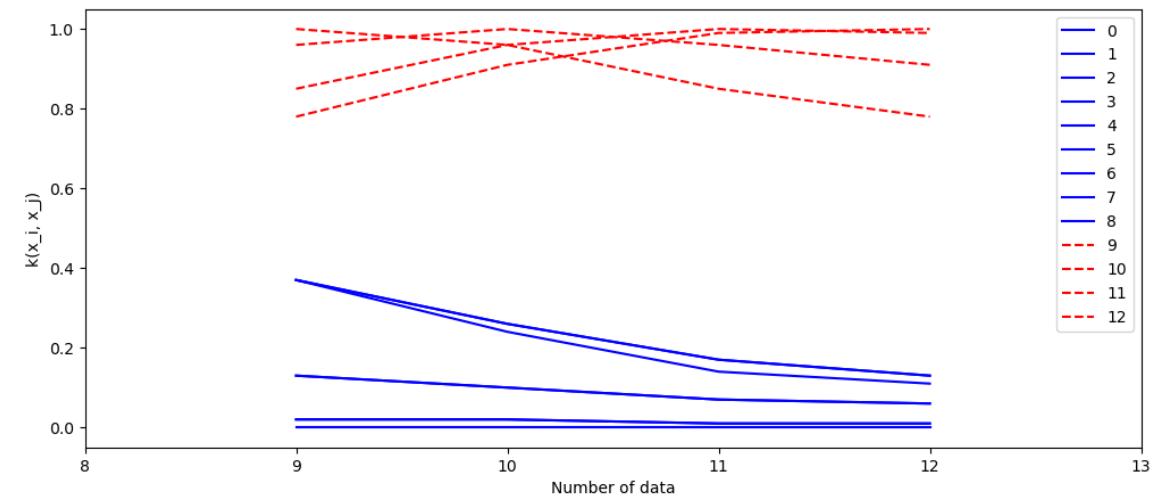
10-2 | 3-6. 結果の確認（ガウスカーネル）

```
# X1-X2の類似度をグラフ化（青線）
# 9~12行目 (X2)、0~9列目 (X1)
ax1 = df.iloc[9:, :9].plot(color="b", figsize=(12, 5))

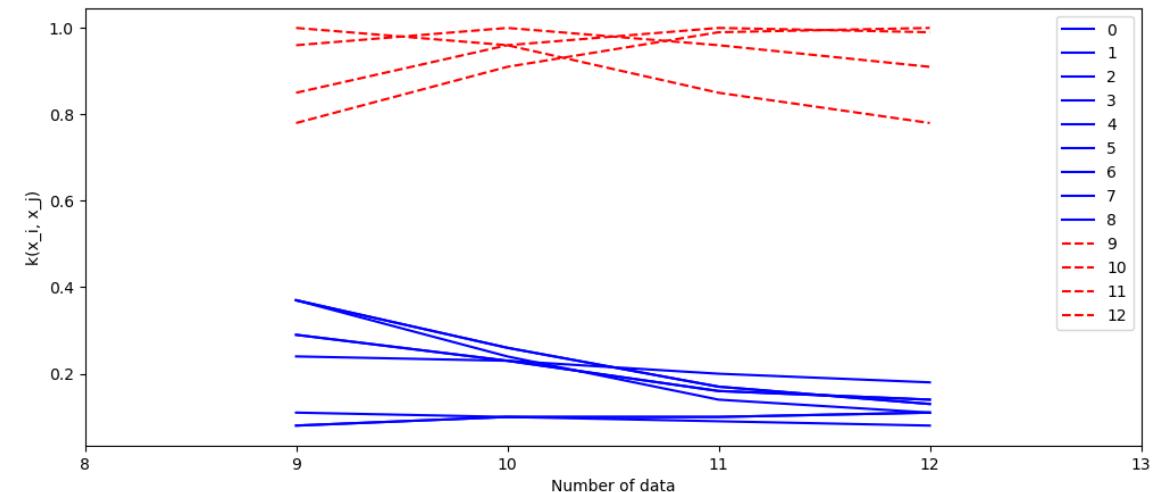
# X2-X2の類似度を重ねて表示（赤線）
# 9~12行目 (X2)、9~12列目 (X2)
df.iloc[9:, 9:12].plot(color="r", ax=ax1, ls="--")

# ラベルの設定
plt.ylabel("k(x_i, x_j)") # 類似度
plt.xlabel("Number of data") # データの片方
plt.legend(loc="best")
# 表示範囲の調整
plt.xlim([8, 13])
plt.show()
```

DATA_TYPE=1



DATA_TYPE=2



現場で使える 機械学習・データ分析基礎講座

第 11 章：深層学習の概要

ノートブック解説

■ 深層学習の概要

- 11-1_gradient_descent.ipynb
- 11-2_NN_practice.ipynb
- 11-3_activate_function.ipynb (_trainee あり)

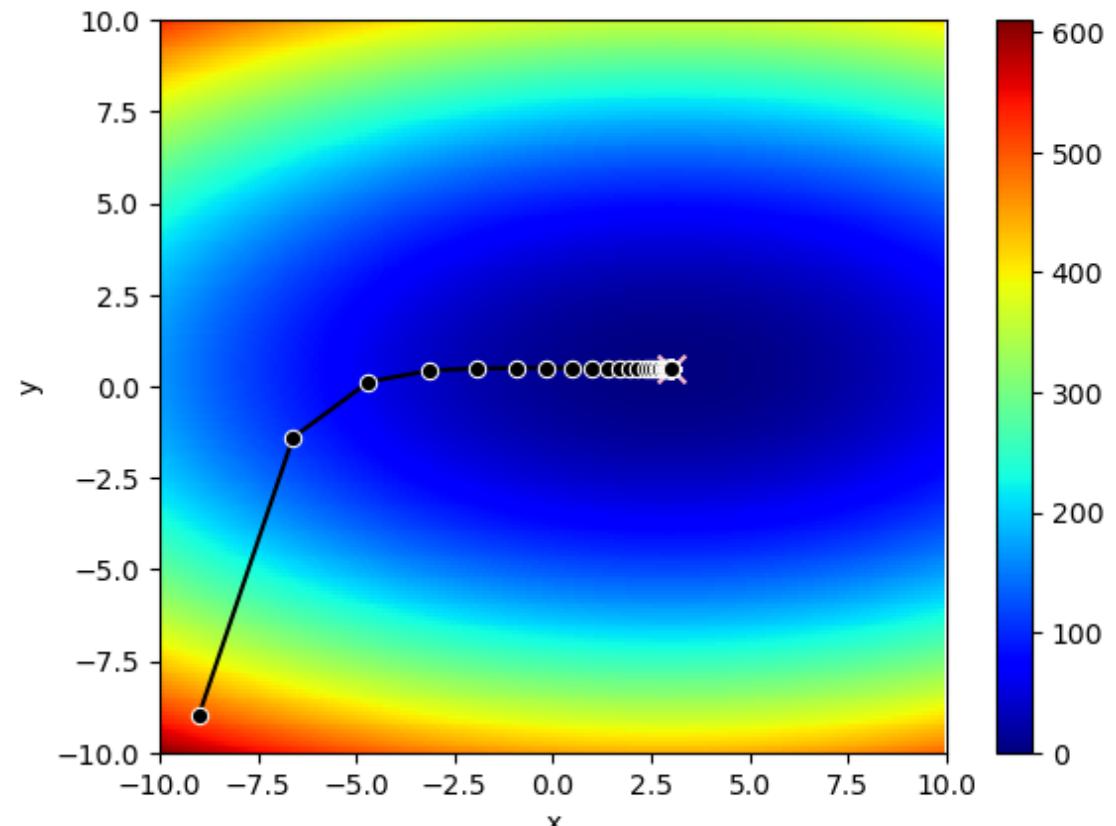
11-1_gradient_decent

■ 内容

- 勾配降下法の振る舞いを、グラフで可視化しながら確認
- 初期値や学習率を変更したとき、どのような変化が起こるか確認

■ 手順

- ライブラリの読み込み
- 目的関数のグラフを作成
- 勾配降下法の計算
- 結果の可視化



```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

11-1 | 2. 目的関数のグラフを作成

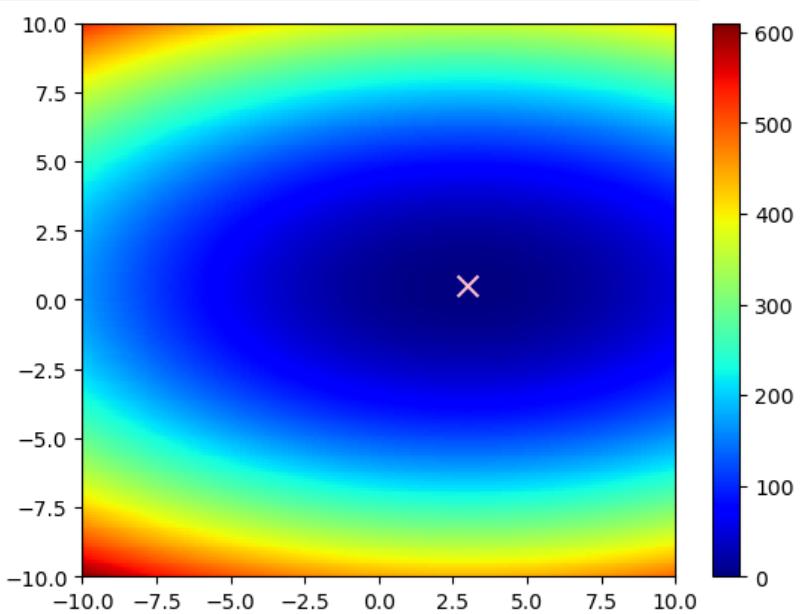
```
# 目的関数の定義
def func(x, y):
    z =  $f(x, y) = (x - 3)^2 + (2y - 1)^2$ 
    return (x-3)**2 + (2*y-1)**2

# x軸、y軸の値を生成
axis_x = np.arange(-10, 10, 0.1)
axis_y = np.arange(-10, 10, 0.1)
# 格子点の作成
mx, my = np.meshgrid(axis_x, axis_y)

# z軸（目的関数）の値を格納する配列
mz = np.ones([200, 200])
# 各点におけるzを計算
for i, y in enumerate(axis_y): # y軸のループ
    z = []
    for x in axis_x: # x軸のループ
        z.append(func(x, y))
    mz[i] = z
```

```
# 目的関数の値を色として表示
plt.pcolor(mx, my, mz, cmap="jet")
plt.colorbar()
# 目的関数の最小値を×で表示
plt.scatter(x=3, y=0.5, marker="x", s=100, c="pink")

# 表示範囲の調整
plt.xlim([-10, 10])
plt.ylim([-10, 10])
plt.show()
```



11-1 | 3. 勾配降下法の計算

```
# 条件設定
x = -9          # xの初期値
y = -9          # yの初期値
alpha = 0.1     # 学習率
norm = 1         # 勾配ベクトルのノルム
threshold = 1.0e-5 # 閾値（収束条件）
i = 0           # 現在の更新回数
maxIter = 1000 # 更新回数の上限

# 結果を格納するためのデータフレーム
df_re = pd.DataFrame()
df_re.loc[i, "x"] = x
df_re.loc[i, "y"] = y
df_re.loc[i, "norm"] = norm
i += 1
```

```
# 勾配の大きさが閾値以下になるまで繰り返す
while norm > threshold:
    dx = -2 * x + 6 # ∂f/∂xの計算
    dy = -8 * y + 4 # ∂f/∂yの計算
    x += dx * alpha # xの更新
    y += dy * alpha # yの更新
    norm = np.linalg.norm([dx, dy]) # 勾配の大きさ
    # データフレームに値を格納
    df_re.loc[i, "x"] = x
    df_re.loc[i, "y"] = y
    df_re.loc[i, "norm"] = norm
    i += 1
    # 更新回数が上限を越えたら終了
    if i > maxIter:
        break
iter_num: 68, (3.0, 0.5)
grad = 9.641628265555369e-06

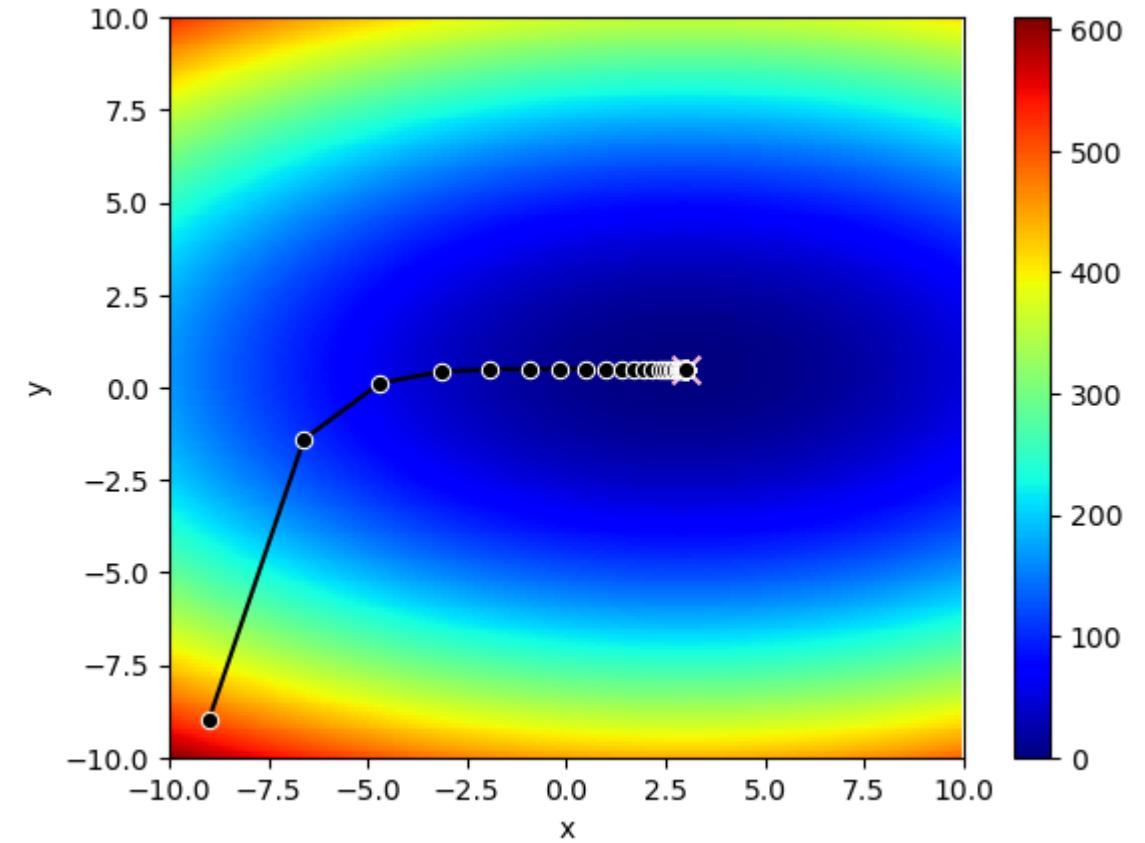
# 更新回数・最小点・勾配の大きさを表示
print(f"iter_num: {i}, ({round(x,3)}, {round(y,3)})")
print(f"grad = {norm}")
```

11-1 | 4. 結果の可視化

```
# 目的関数の値を色として表示
plt.pcolor(mx, my, mz, cmap="jet")
plt.colorbar()
# 目的関数の最小値をxで表示
plt.scatter(x=3, y=0.5, marker="x", s=100, c="pink")

# 探索点を●で表示
sns.lineplot(x=df_re["x"], y=df_re["y"],
              marker="o", color="k")

#表示範囲の調整
plt.xlim([-10,10])
plt.ylim([-10,10])
plt.show()
```



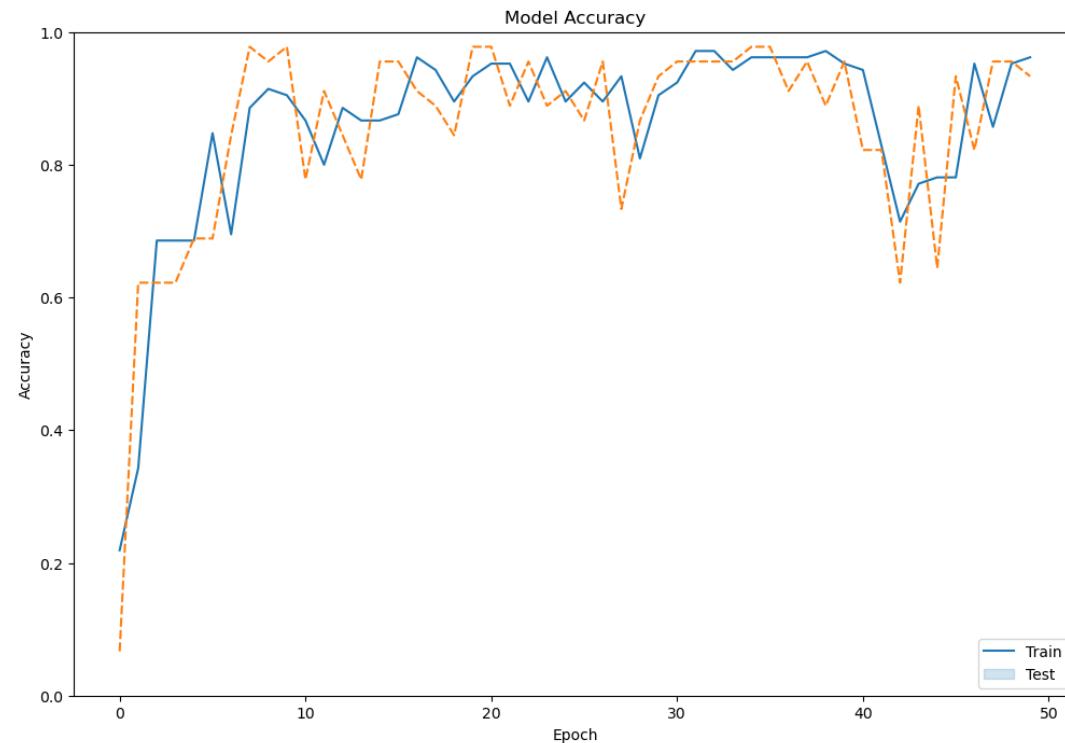
11-2_NN_practice

■ 内容

- TensorFlow を用いてニューラルネットワークを実装
- ハイパーパラメータを変更し、どのような変化が起こるか確認

■ 手順

- ライブラリの読み込み
- データの読み込み
- データの前処理
- モデルの構築
- モデルの学習
- 学習結果の確認
 - 学習曲線の表示
 - 学習後のパラメータを表示
 - 予測の実行



11-2 | 1. ライブラリの読み込み

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# one-hotベクトルへの変換
from tensorflow.keras.utils import to_categorical
# モデルの構築と読み込み
from tensorflow.keras.models import Sequential, load_model
# 全結合層
from tensorflow.keras.layers import Dense
# 最適化手法（パラメータの更新方法）
from tensorflow.keras.optimizers import SGD, RMSprop, Adagrad, Adadelta, Adam
```

11-2 | 2. データの読み込み

```
# データセットの読み込み
iris = load_iris()

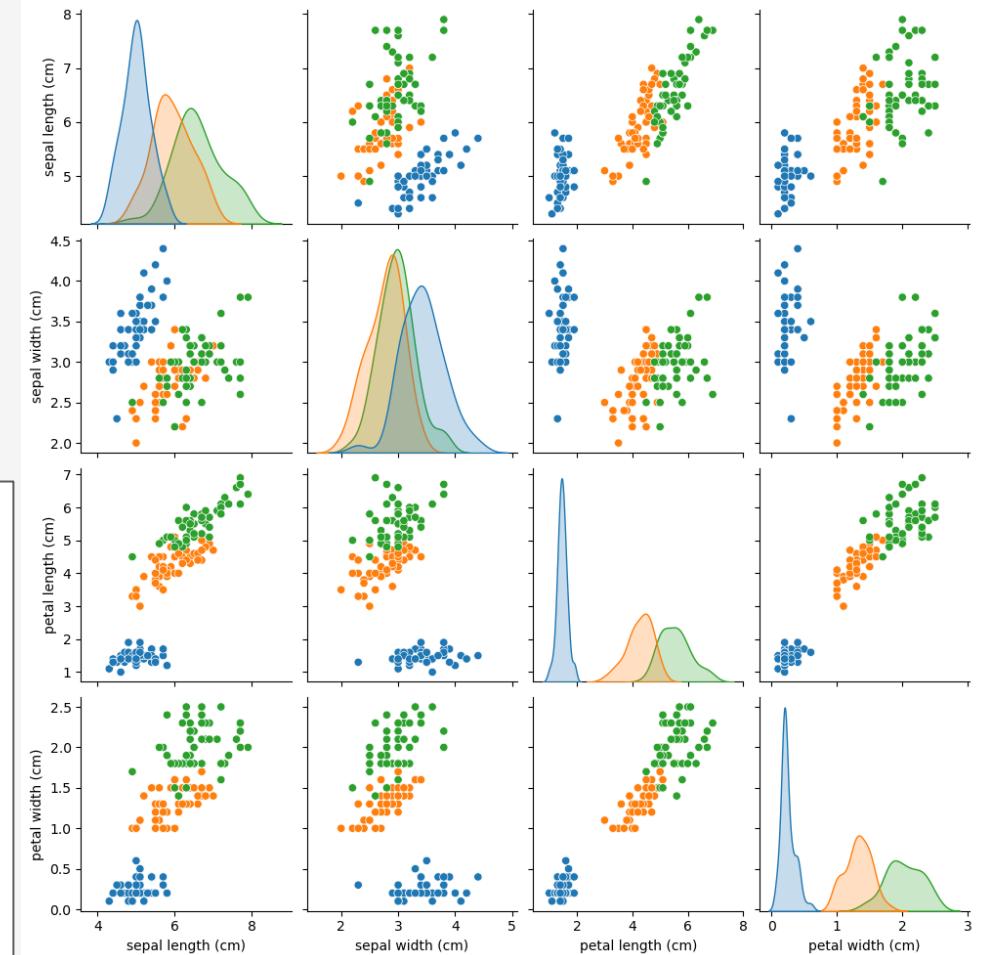
# 説明変数をデータフレームに変換
df_iris = pd.DataFrame(iris.data, columns=iris.feature_names)
# 目的変数をデータフレームに追加
df_iris["label"] = iris.target

# クラス番号を文字列に置き換え
df_iris["label"] = df_iris["label"].map(
    {0:iris.target_names[0],
     1:iris.target_names[1],
     2:iris.target_names[2]})

# データフレームの中身を確認
display(df_iris.head())

# 散布図行列の表示（クラスごとに色分け）
sns.pairplot(df_iris, hue="label")
plt.show()
```

- 目的変数（クラス）
 - Setosa
 - Versicolor
 - Virginica
- 説明変数
 - Sepal Length (萼片の長さ)
 - Sepal Width (萼片の幅)
 - Petal Length (花弁の長さ)
 - Petal Width (花弁の幅)



```
# 説明変数/目的関数の分割
X = iris.data
y = iris.target

# 訓練用/検証用の分割
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=1234
)

# 目的関数をone-hotベクトルに変換
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

# データサイズの確認
print("X_train:", X_train.shape)
print("X_test:", X_test.shape)
print("y_train:", y_train.shape)
print("y_test:", y_test.shape)
```

X_train: (105, 4)
X_test: (45, 4)
y_train: (105, 3)
y_test: (45, 3)

11-2 | 4. モデルの構築

```
# 空のモデルを作成
model = Sequential()
# 入力4、出力6の全結合層を追加
# activation: 活性化関数の種類
model.add(Dense(6, activation='relu', input_dim=4))
# 入力6、出力5の全結合層を追加
model.add(Dense(5, activation='relu', input_dim=6))
# 出力3の全結合層を追加
# 分類モデルの場合、最終層のactivationはsoftmax
model.add(Dense(3, activation='softmax'))

# ----- 最適化手法の設定 -----
sgd = SGD(learning_rate=0.01, momentum=0.9, nesterov=False)
# rms = RMSprop(learning_rate=0.01)
# adag = Adagrad(learning_rate=0.01)
# adad = Adadelta(learning_rate=0.01)
# adam = Adam(learning_rate=0.01)
# -----
```

```
# モデルの学習プロセスを設定
# 損失関数、最適化手法、評価指標を指定
model.compile(
    loss='categorical_crossentropy',
    optimizer=sgd,
    metrics=['accuracy']
)

# モデルの構造を確認
model.summary()
```

Model: "sequential"		
Layer (type)	Output Shape	Param #
dense (Dense)	(None, 6)	30
dense_1 (Dense)	(None, 5)	35
dense_2 (Dense)	(None, 3)	18
Total params: 83		
Trainable params: 83		
Non-trainable params: 0		

```

# 学習の実行
# epochs: 学習回数
# batch_size: 一度に計算するデータの数 (バッチサイズ)
fit = model.fit(
    X_train, y_train,
    epochs=50, batch_size=20,
    validation_data=(X_test, y_test)
)

# 各epochにおける損失と正解率をdfに格納
df = pd.DataFrame(fit.history)
  
```

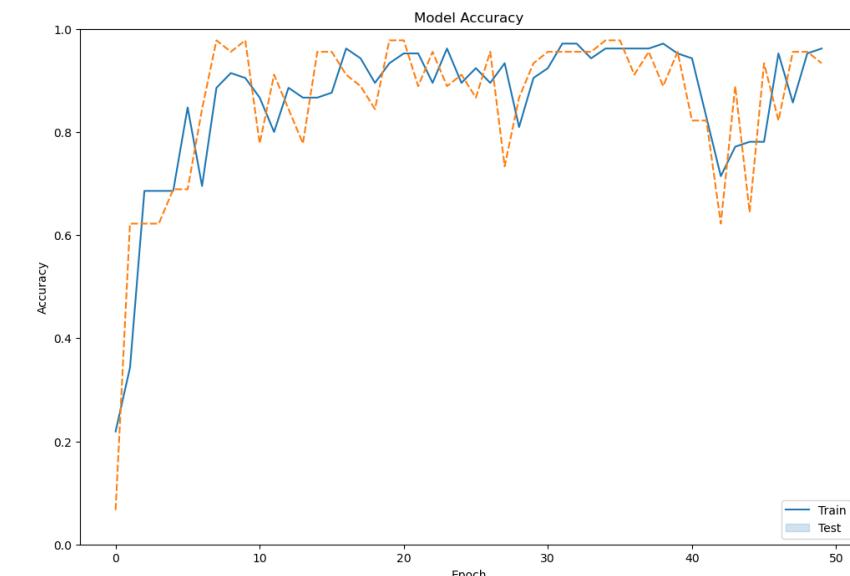
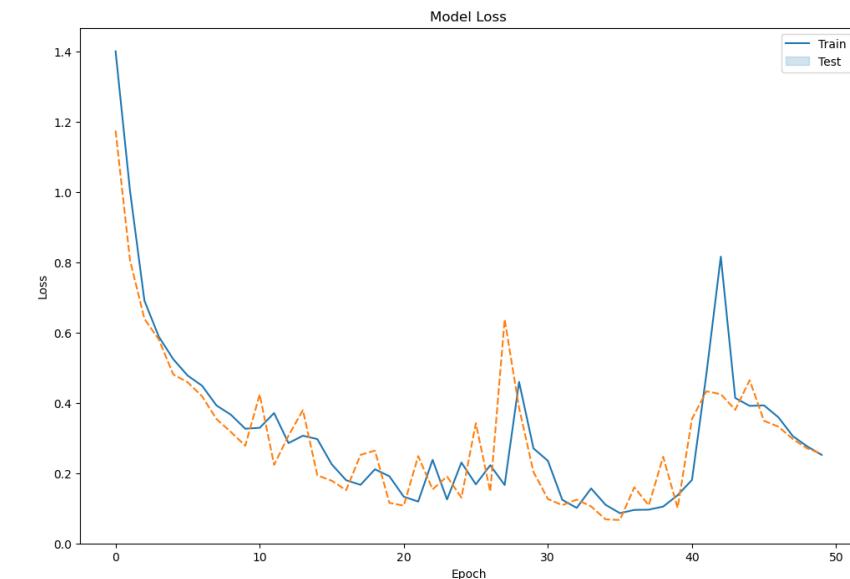
```

Epoch 1/50
6/6 [=====] - 1s 60ms/step - loss: 1.4009 - accuracy: 0.2190 - val_loss: 1.1752 - val_accuracy: 0.0667
Epoch 2/50
6/6 [=====] - 0s 11ms/step - loss: 1.0041 - accuracy: 0.3429 - val_loss: 0.8063 - val_accuracy: 0.6222
Epoch 3/50
6/6 [=====] - 0s 11ms/step - loss: 0.6913 - accuracy: 0.6857 - val_loss: 0.6393 - val_accuracy: 0.6222
Epoch 4/50
6/6 [=====] - 0s 10ms/step - loss: 0.5885 - accuracy: 0.6857 - val_loss: 0.5805 - val_accuracy: 0.6222
Epoch 5/50
6/6 [=====] - 0s 10ms/step - loss: 0.5245 - accuracy: 0.6857 - val_loss: 0.4810 - val_accuracy: 0.6889
Epoch 6/50
6/6 [=====] - 0s 10ms/step - loss: 0.4778 - accuracy: 0.8476 - val_loss: 0.4587 - val_accuracy: 0.6889
Epoch 7/50
6/6 [=====] - 0s 10ms/step - loss: 0.4489 - accuracy: 0.6952 - val_loss: 0.4189 - val_accuracy: 0.8444
Epoch 8/50
6/6 [=====] - 0s 9ms/step - loss: 0.3927 - accuracy: 0.8857 - val_loss: 0.3540 - val_accuracy: 0.9778
Epoch 9/50
6/6 [=====] - 0s 9ms/step - loss: 0.3663 - accuracy: 0.9143 - val_loss: 0.3172 - val_accuracy: 0.9556
Epoch 10/50
6/6 [=====] - 0s 8ms/step - loss: 0.3264 - accuracy: 0.9048 - val_loss: 0.2778 - val_accuracy: 0.9778
Epoch 11/50
6/6 [=====] - 0s 12ms/step - loss: 0.3290 - accuracy: 0.8667 - val_loss: 0.4242 - val_accuracy: 0.7778
Epoch 12/50
6/6 [=====] - 0s 11ms/step - loss: 0.3711 - accuracy: 0.8000 - val_loss: 0.2236 - val_accuracy: 0.9111
Epoch 13/50
6/6 [=====] - 0s 9ms/step - loss: 0.2855 - accuracy: 0.8857 - val_loss: 0.3066 - val_accuracy: 0.8444
Epoch 14/50
6/6 [=====] - 0s 9ms/step - loss: 0.3065 - accuracy: 0.8667 - val_loss: 0.3796 - val_accuracy: 0.7778
Epoch 15/50
6/6 [=====] - 0s 9ms/step - loss: 0.2969 - accuracy: 0.8667 - val_loss: 0.1937 - val_accuracy: 0.9556
  
```

11-2 | 6-1. 学習曲線の表示

```
# Loss (損失)をグラフ化
plt.figure(figsize=(12, 8))
sns.lineplot(data=df[["loss", "val_loss"]])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper right')
plt.show()
```

```
# Accuracy (正解率)をグラフ化
plt.figure(figsize=(12, 8))
sns.lineplot(data=df[["accuracy", "val_accuracy"]])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.ylim([0,1.0])
plt.legend(['Train', 'Test'], loc='lower right')
plt.show()
```



11-2 | 6-2. 学習後のパラメータを表示

```
# パラメータを取得
weights = model.get_weights()

# 全ての層のパラメータを表示
# 偶数番目は重み、奇数番目はバイアス
for i in range(len(weights)):
    print("weights[%s] = %i" % (i, len(weights[i])))
    print(weights[i])
    print("num:", weights[i].flatten().shape[0])
    print()
```

```
weights[0]=
[[-0.33299777  0.1173033   0.038746   -0.10054118 -0.16344285  1.0907683 ]
 [-0.17494386 -0.67514586 -0.6634359   -0.8541814  -0.10640674  0.24260029]
 [-0.57980067  0.9732491   0.24253978  1.6027133   0.65872127 -0.82134306]
 [-0.33228657  1.159295    0.888793    0.6010479   0.33845553 -1.1343031 ]]
num: 24

weights[1]=
[ 0.          -0.24175799 -0.16731378 -0.23823105 -0.20907304  0.33441857]
num: 6

weights[2]=
[[ 0.71263784 -0.51306874  0.46090442 -0.29739583  0.5323469 ]
 [-0.0885699  -0.7208868   0.1828951  -1.1672195  -0.22386928]
 [ 0.784331   -0.5992001   0.6346341  -0.40654185 -0.4639653 ]
 [ 0.42206967 -0.40078202  0.27658716 -1.3076733   0.34169447]
 [ 0.1741393  -0.04018056  0.26079604 -0.52118754 -0.4181909 ]
 [-0.7700956  -0.48460126 -0.0493258   1.1842515  -0.14052142]]
num: 30

weights[3]=
[-0.02496619  0.          -0.23395438  0.38751462 -0.05100761]
num: 5

weights[4]=
[[-0.20008425 -0.93574274  0.05117688]
 [-0.6182211   0.07643843  0.12773818]
 [-1.0879904   0.5110429   0.55438375]
 [ 0.9709561  -0.04388493 -1.3429043 ]
 [-0.8478299   0.11062877 -0.69409305]]
num: 15

weights[5]=
[ 0.05703399  0.3932075  -0.4502415 ]
num: 3
```

11-2 | 6-3. 予測の実行

```
# 各カテゴリに属する確率を算出
y_proba = model.predict(X_test)

# クラスラベルに変換
# 確率が最も大きい要素の番号を抽出
y_pred = np.argmax(y_proba, axis=1)
print(y_pred)
```

```
2/2 [=====] - 0s 3ms/step
[1 2 2 0 1 0 0 0 1 2 1 0 2 1 0 1 2 0 2 1 1 1 1 1 2 0 2 1 2 0 1 2 0 2 2 0 0
 0 0 1 0 2 0 2 2]
```

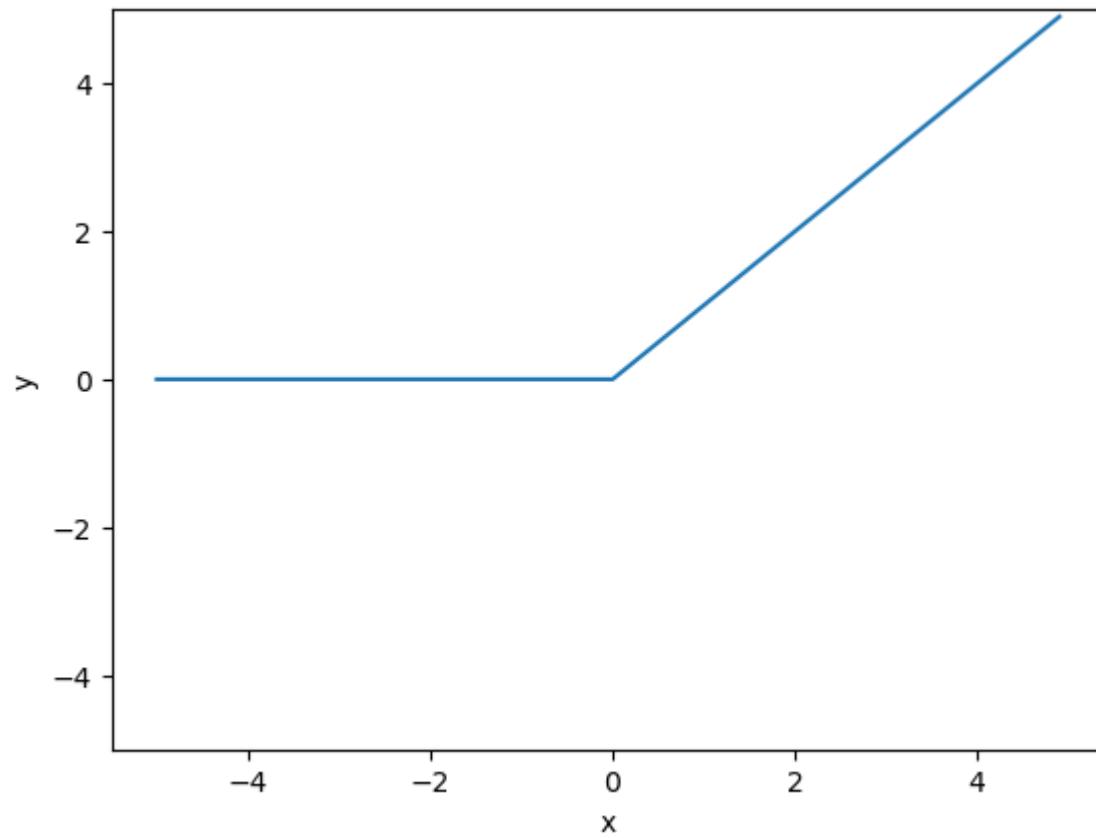
11-3_activate_function

■ 内容

- 活性化関数の実装方法とその形状を確認
- ReLU 関数を自分の手で実装

■ 手順

1. ライブラリの読み込み
2. ステップ関数
3. シグモイド関数
4. tanh 関数
5. ReLU 関数



```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

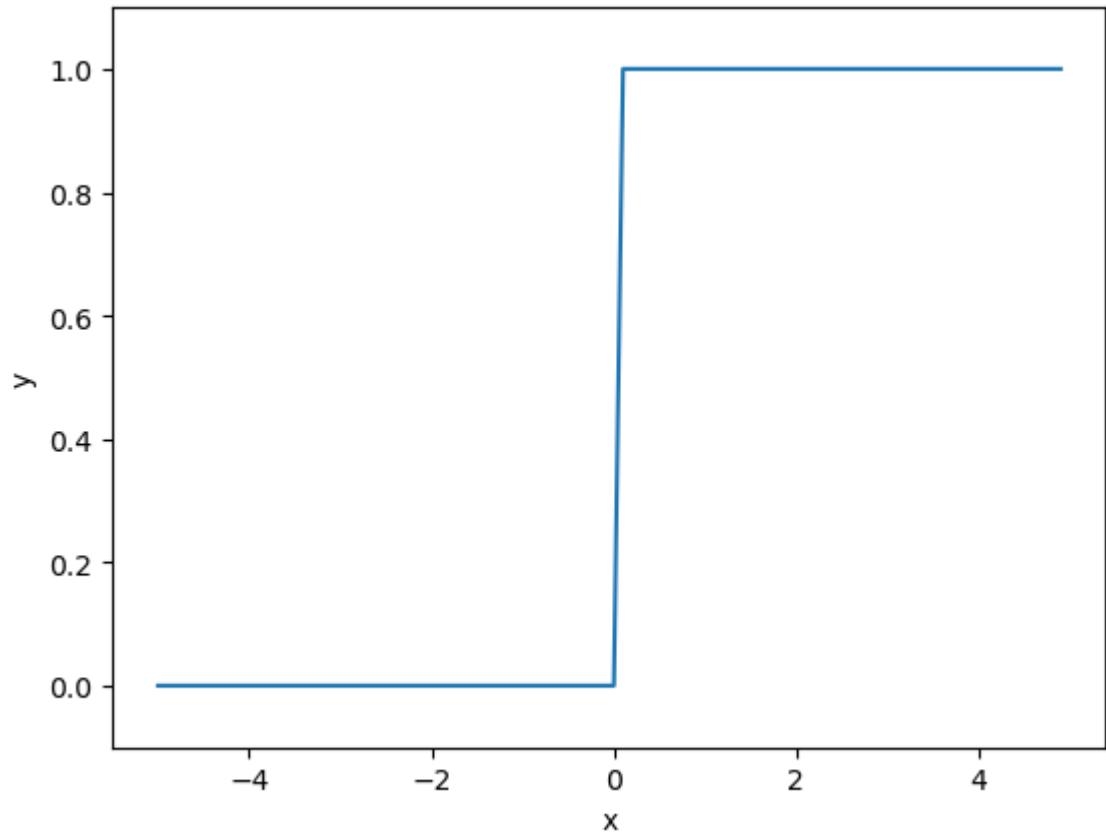
11-3 | 2. ステップ関数

```
# 関数の定義
def step(x):
    return np.array(x>0).astype(np.int32)

# 横軸の値を生成
x = np.arange(-5,5,0.1)
# 縦軸の値（関数の出力）を計算
y = step(x)

# グラフの表示
sns.lineplot(x=x,y=y)
# ラベルの設定
plt.xlabel("x")
plt.ylabel("y")
# 表示範囲の調整
plt.ylim([-0.1,1.1])
plt.show()
```

$$h(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$$



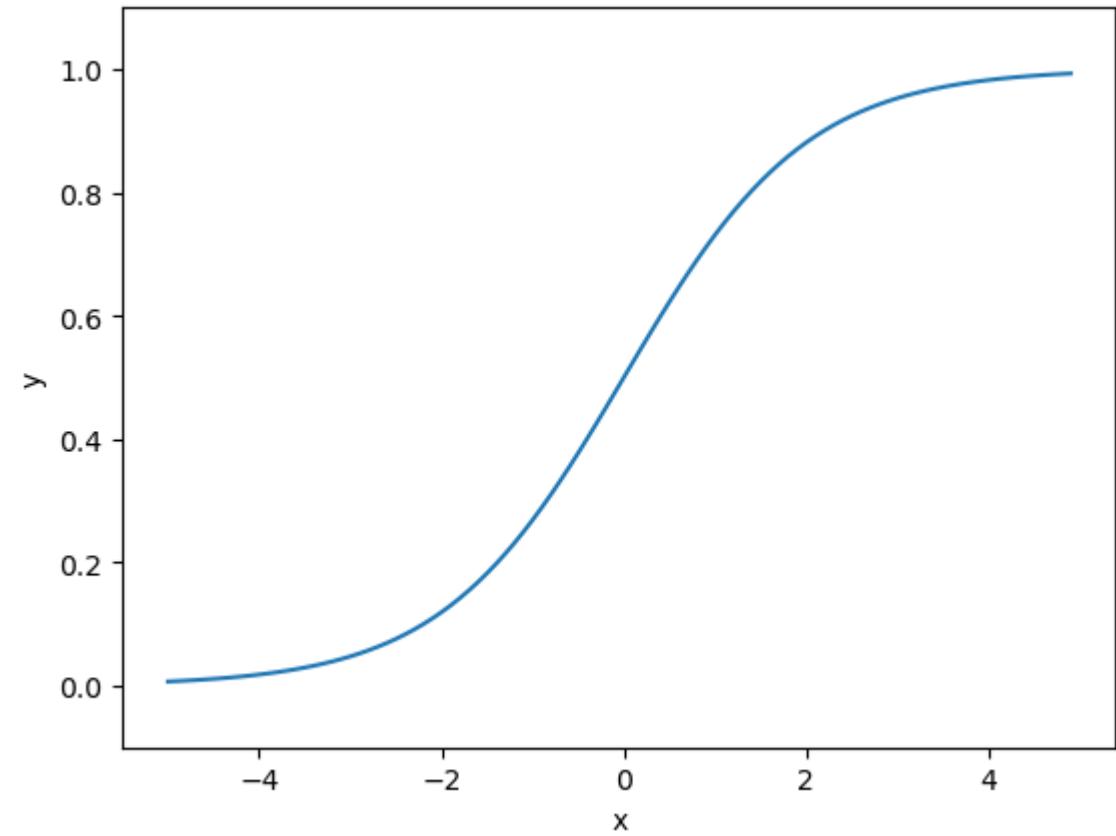
11-3 | 3. シグモイド関数

```
# 関数の定義
def sigmoid(x):
    return 1 / (1+ np.exp(-x))

# 縦軸の値（関数の出力）を計算
y = sigmoid(x)

# グラフの表示
sns.lineplot(x=x, y=y)
# ラベルの設定
plt.xlabel("x")
plt.ylabel("y")
# 表示範囲の調整
plt.ylim([-0.1,1.1])
plt.show()
```

$$y = \frac{1}{1+\exp(-x)}$$



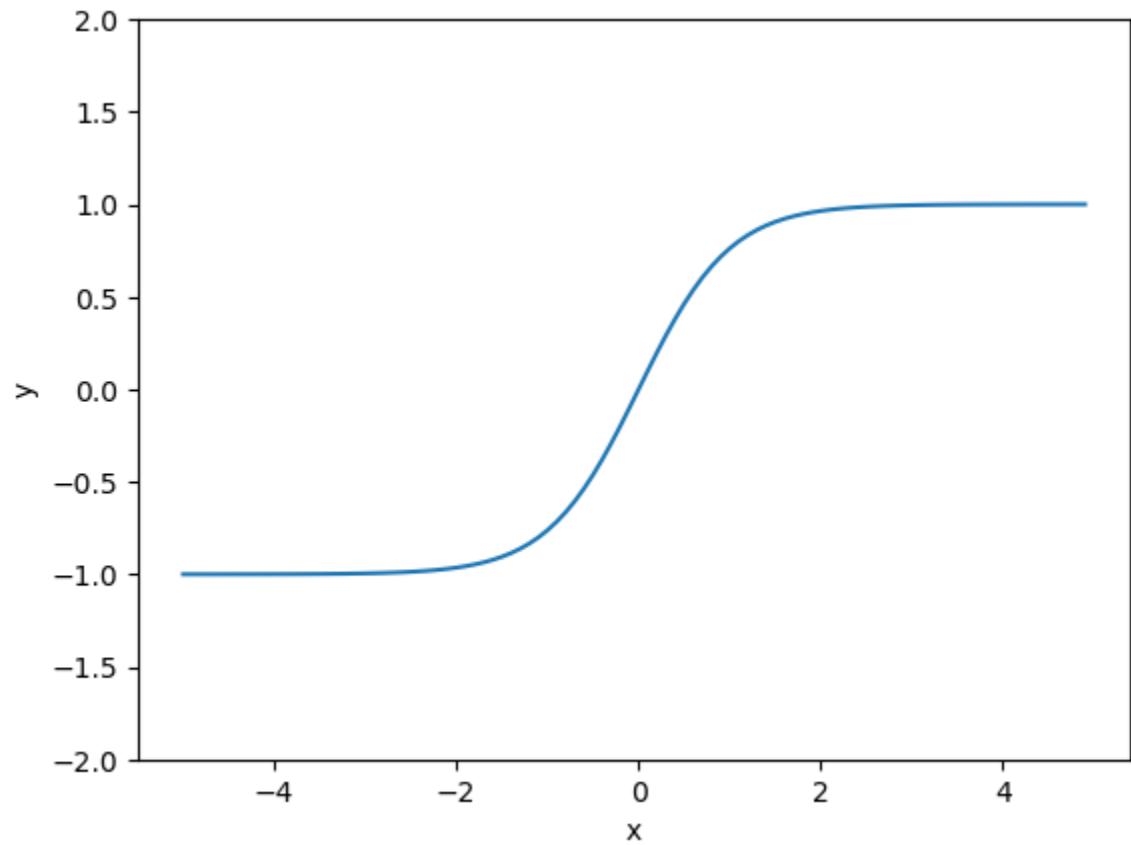
11-3 | 4. tanh 関数

```
# 関数の定義
def tanh(x):
    return np.tanh(x)

# 縦軸の値（関数の出力）を計算
y = tanh(x)

# グラフの表示
sns.lineplot(x=x, y=y)
# ラベルの設定
plt.xlabel("x")
plt.ylabel("y")
# 表示範囲の調整
plt.ylim([-2, 2])
plt.show()
```

$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



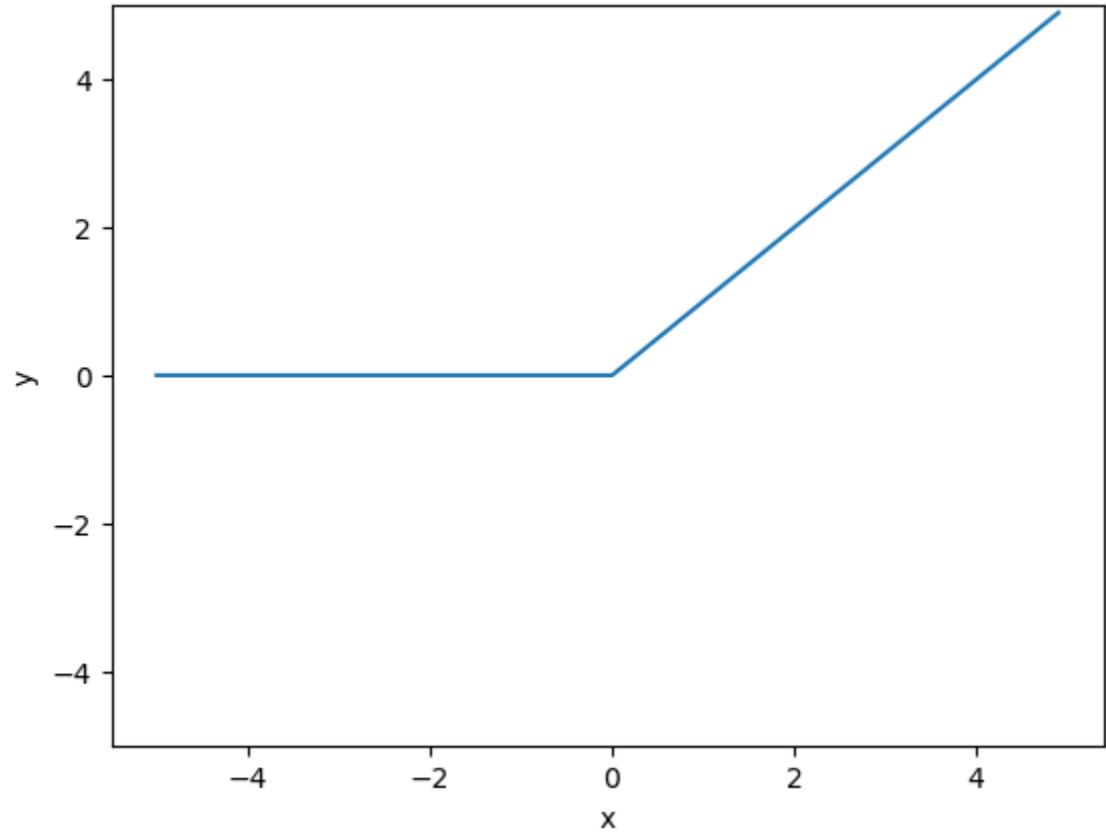
11-3 | 5. [演習] ReLU 関数

```
# 関数の定義
def relu(x):
    x = np.maximum(0, x)
    return x

# 縦軸の値（関数の出力）を計算
y = relu(x)

# グラフの表示
sns.lineplot(x=x, y=y)
# ラベルの設定
plt.xlabel("x")
plt.ylabel("y")
# 表示範囲の調整
plt.ylim([-5.0, 5])
plt.show()
```

$$\text{ReLU}(x) = \max(x, 0)$$



現場で使える 機械学習・データ分析基礎講座

第 12 章：畳み込みニューラルネットワーク
ノートブック解説

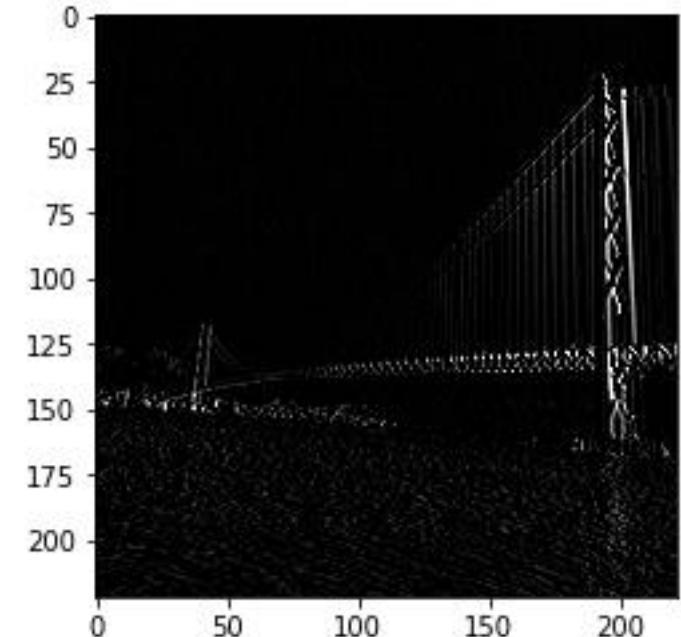
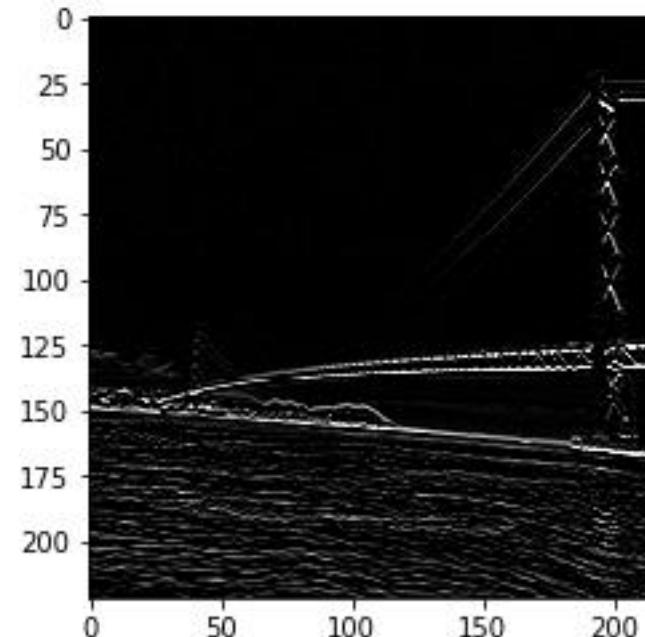
■ 置み込みニューラルネットワーク

- 12-1_convolution
- 12-2_pooling
- 12-3_CNN

12-1_convolution

■ 内容

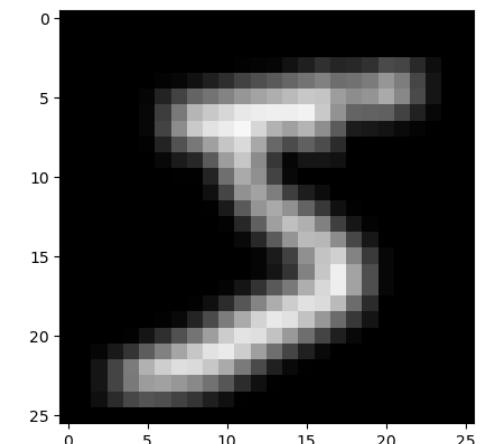
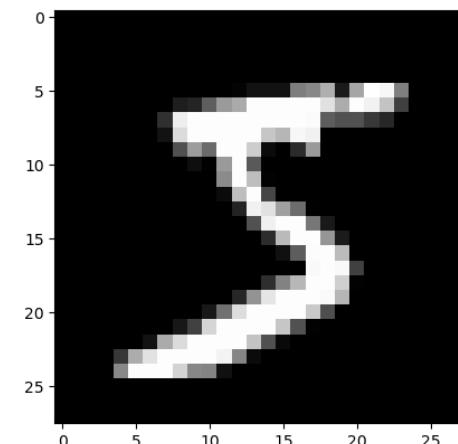
- 関数 `im2col()` を用いて畳み込み演算を実装
- 簡単な行列、モノクロ画像、カラー画像に対してそれぞれ処理を行う



■ 手順

1. ライブラリの読み込み
2. 疑似データに対する畳み込み
 1. データの作成
 2. データの配列を変換
 3. フィルタの作成
 4. フィルタの配列を変換
 5. 畳み込み演算の実行
3. モノクロ画像に対する畳み込み
 1. 畳み込み演算を関数として定義
 2. データの読み込み
 3. フィルタの作成
 4. 畳み込み演算の実行

4. 様々なフィルタ
 1. 垂直方向のエッジ抽出
 2. 水平方向のエッジ抽出
5. カラー画像に対する畳み込み
 1. 畳み込み演算を関数として定義
 2. データの読み込み
 3. 畳み込み演算の実行



```
import numpy as np
import matplotlib.pyplot as plt

# 画像データを扱うライブラリ
from PIL import Image

# MNISTデータセットの読み込み
from tensorflow.keras.datasets import mnist

# フィルタ演算を行列の積に変換する関数
from common.util import im2col
```

```
img = np.array([
    [1, 2, 3, 0],
    [0, 1, 2, 3],
    [3, 0, 1, 2],
    [2, 3, 0, 1]
])

# im2col()に入力するため、軸を増やす
# (画像の枚数, チャネル数, 縦幅, 横幅)
img_expand = img.reshape(1, 1, img.shape[0], img.shape[1])

# データの形状と中身を表示
print(img_expand.shape)
img_expand
```

(1, 1, 4, 4)

```
array([[[[1, 2, 3, 0],
        [0, 1, 2, 3],
        [3, 0, 1, 2],
        [2, 3, 0, 1]]]])
```

12-1 | 2-2. データの配列を変換

```
# 画像, フィルタサイズ(縦幅, 横幅), ストライド, パディング幅
col = im2col(img_expand, 3, 3, stride=1, pad=0)

# データの形状と中身を表示
print("shape:", col.shape)
col
```

shape: (4, 9)

```
array([[1., 2., 3., 0., 1., 2., 3., 0., 1.],
       [2., 3., 0., 1., 2., 3., 0., 1., 2.],
       [0., 1., 2., 3., 0., 1., 2., 3., 0.],
       [1., 2., 3., 0., 1., 2., 3., 0., 1.]])
```

12-1 | 2-3. フィルタの作成

```
flt = np.array([
    [2, 0, 1],
    [0, 1, 2],
    [1, 0, 2]
])

# フィルタの形状と中身を表示
print("shape:", flt.shape)
flt
```

shape: (3, 3)

```
array([[2, 0, 1],
       [0, 1, 2],
       [1, 0, 2]])
```

12-1 | 2-4. フィルタの配列を変換

```
# 一次元配列に変換
flt_flat = flt.reshape(-1)
flt_flat
```

```
array([2, 0, 1, 0, 1, 2, 1, 0, 2])
```

```
# 行列の積をとることで置み込む
img_y = np.dot(col, flt_flat)

# 二次元配列に戻す
img_y.reshape(2, 2)
```

```
array([[15., 16.],
       [ 6., 15.]])
```

```
# 出力サイズを計算する関数
def calc_output_size(input_size, filter_size, stride, padding):
    output_h = 1 + (input_size[0] + 2*padding - filter_size[0]) / stride
    output_w = 1 + (input_size[1] + 2*padding - filter_size[1]) / stride

    # 小数点以下を切り捨て
    output_h = np.floor(output_h).astype(int)
    output_w = np.floor(output_w).astype(int)
    return output_h, output_w
```

- 置み込み演算の出力サイズは、以下の式で計算できる

$$O_h = \frac{I_h + 2p - F_h}{s} + 1$$

$$O_w = \frac{I_w + 2p - F_w}{s} + 1$$

- O_h, O_w : 出力の縦幅、横幅
- I_h, I_w : 入力の縦幅、横幅
- F_h, F_w : フィルタの縦幅、横幅
- s : ストライド
- p : パディング幅

```
# 二次元配列imgとfltの置み込み演算
def conv2d(img, flt, stride, pad):
    # im2col()に入力するため、軸を増やす
    img_expand = img.reshape(1, 1, img.shape[0], img.shape[1])
    # im2col()でデータを変換
    # 画像, フィルタサイズ(縦幅, 横幅), ストライド, パディング幅
    col = im2col(img_expand, flt.shape[0], flt.shape[1], stride=stride, pad=pad)

    # フィルタを一次元配列に変換
    flt_flat = flt.reshape(-1)
    # 行列の積をとることで置み込む
    img_y = np.dot(col, flt_flat)

    # 出力サイズを計算
    output_size = calc_output_size(img.shape, flt.shape, stride, pad)

    # 出力データを二次元配列に戻す
    img_y = img_y.reshape(output_size[0], output_size[1])
    return img_y
```

12-1 | 3-2. データの読み込み

```
# 配列を画像として表示するための関数
def show_image(img):
    # 配列をImageオブジェクトに変換
    pil_img = Image.fromarray(img)
    # Imageオブジェクトを画像として表示
    plt.imshow(pil_img)
    # グレースケールで表示
    plt.gray()
    plt.show()
```

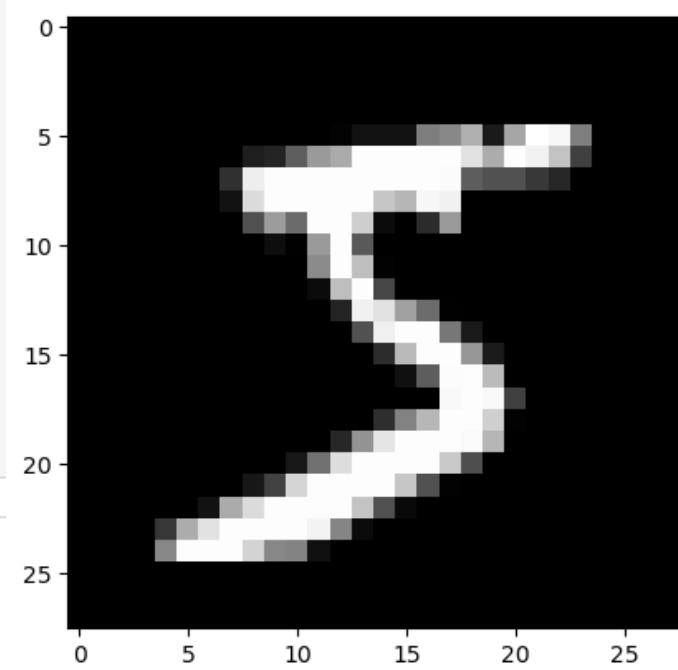
```
# MNISTデータセットの読み込み
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# データの形状を確認
print("X_train:", X_train.shape)

# クラスラベル
label = y_train[0]
# 画像データ（を表す配列）
img = X_train[0]

# クラスラベルの表示
print("label = %s" % label)
# 画像の表示
show_image(img)
```

```
X_train: (60000, 28, 28)
label = 5
```



12-1 | 3-3. フィルタの作成

```
# 平均化フィルタ
flt_mean = np.array([
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9],
    [1/9, 1/9, 1/9]
])

# 合計値を表示
print("filter values sum:", flt_mean.sum())
# 中身を表示
```

```
flt_mean
```

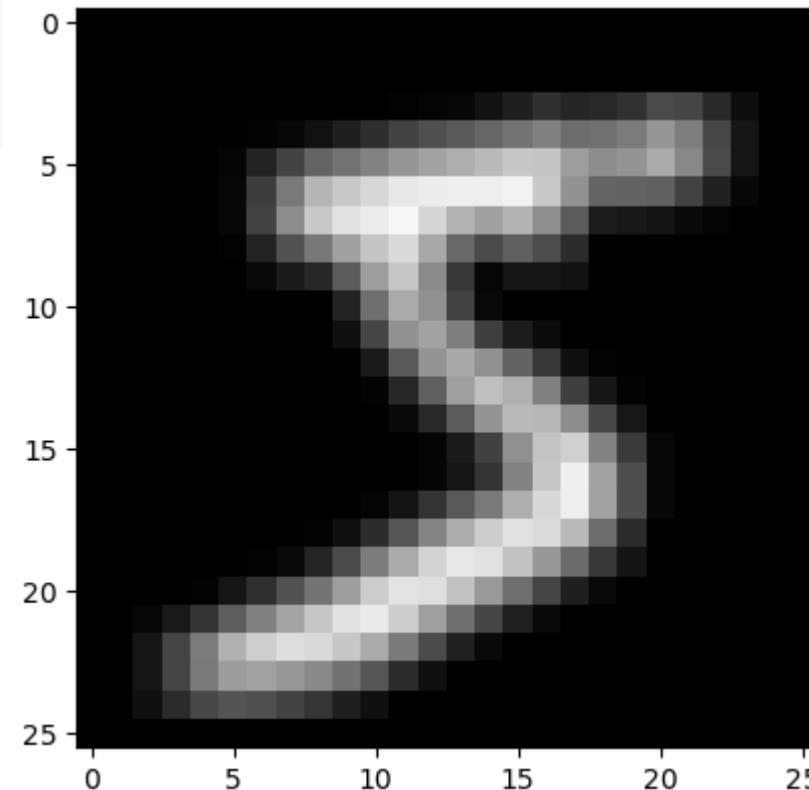
```
filter values sum: 1.0
```

```
array([[0.11111111, 0.11111111, 0.11111111],
       [0.11111111, 0.11111111, 0.11111111],
       [0.11111111, 0.11111111, 0.11111111]])
```

12-1 | 3-4. 置み込み演算の実行

```
# 置み込み演算
img_y = conv2d(img, flt_mean, stride=1, pad=0)
```

```
# 結果を画像として表示
show_image(img_y)
```



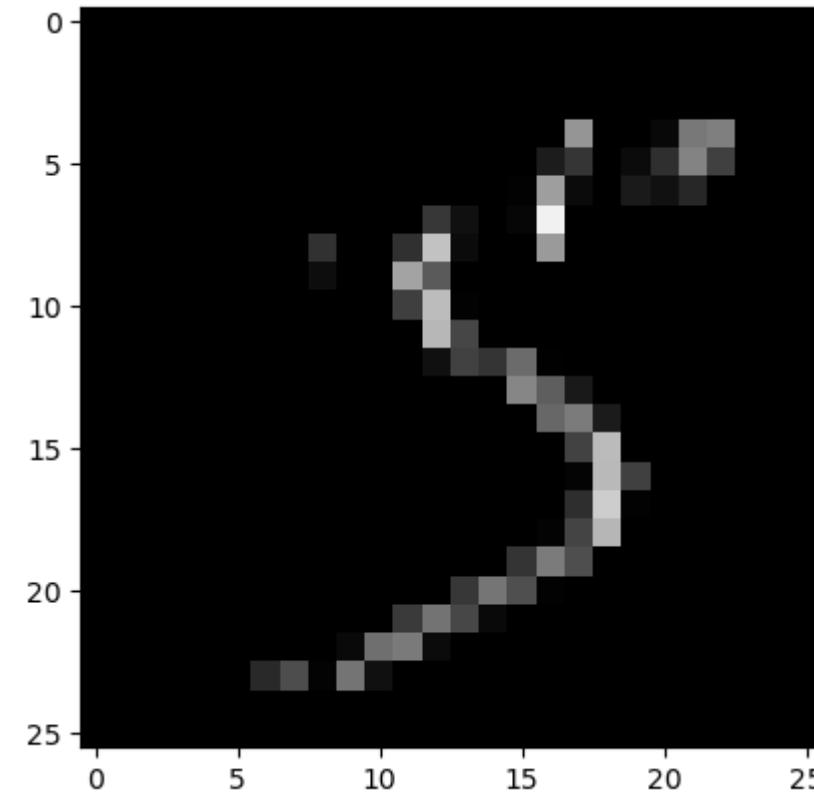
12-1 | 4-1. 垂直方向のエッジ抽出

```
# フィルタの作成
flt_dv = np.array([
    [0, 0, 0],
    [0, 1, -1],
    [0, 0, 0]
])

# 合計値を表示
print("filter values sum:", flt_dv.sum())

# 署み込み演算
img_y = conv2d(img, flt_dv, stride=1, pad=0)

# 結果を画像として表示
show_image(img_y)
```



filter values sum: 0

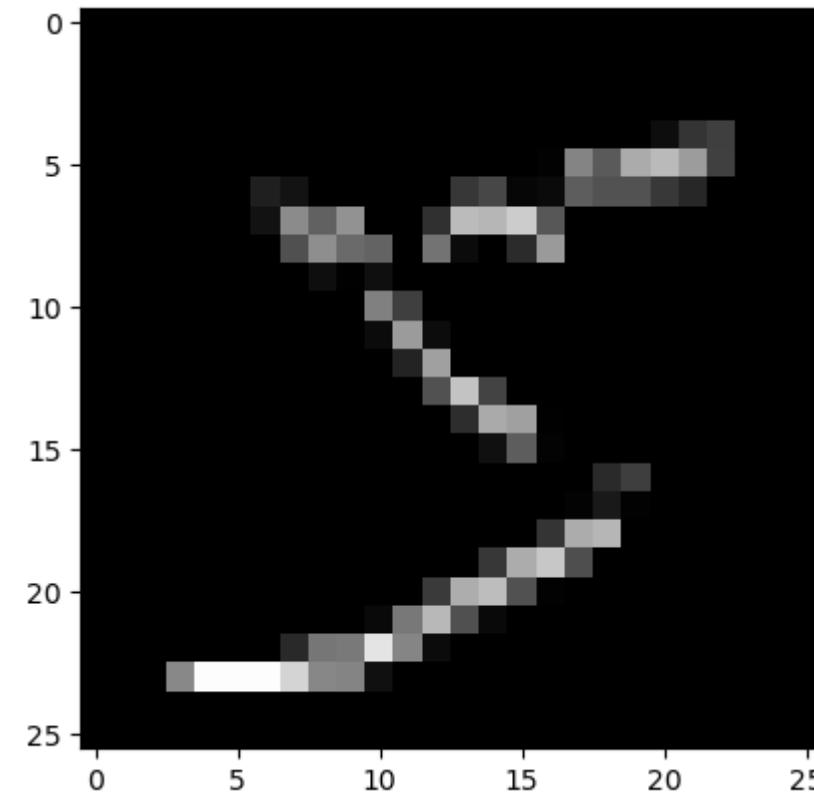
12-1 | 4-2. 水平方向のエッジ抽出

```
# フィルタの作成
flt_dh = np.array([
    [0, 0, 0],
    [0, 1, 0],
    [0, -1, 0]
])

# 合計値を表示
print("filter values sum:", flt_dh.sum())

# 署み込み演算
img_y = conv2d(img, flt_dh, stride=1, pad=0)

# 結果を画像として表示
show_image(img_y)
```



filter values sum: 0

12-1 | 5-1. 置み込み演算を関数として定義

```
# 複数のチャネルともつ二次元配列imgとfltの置み込み演算
def conv2d_multi_ch(img, flt, stride, pad):
    # 配列の軸を入れ替え
    # (高さ, 幅, チャネル数) → (チャネル数, 高さ, 幅)
    img_t = img.transpose(2, 0, 1)

    # im2col()に入力するため、軸を増やす
    img_expand = img_t.reshape(1, img_t.shape[0], img_t.shape[1], img_t.shape[2])
    # im2col()でデータを変換
    # 画像, フィルタサイズ(縦幅, 横幅), ストライド, パディング幅
    col = im2col(img_expand, flt.shape[0], flt.shape[1], stride=stride, pad=pad)

    # フィルタをチャネル数に合わせて増やす
    # 3チャネルの場合、flt_multi = np.array([flt, flt, flt])
    flt_multi = flt.copy()
    for i in range(img_t.shape[0]-1):
        flt_multi = np.vstack((flt_multi, flt))

    # フィルタを一次元配列に変換
    flt_flat = flt_multi.reshape(-1)
    # 行列の積をとることで畳み込む
    img_y = np.dot(col, flt_flat)
```

```
# 入力画像のサイズをタプルにまとめる
input_size = (img_t.shape[1], img_t.shape[2])

# 出力サイズを計算
output_size = calc_output_size(input_size, flt.shape, stride, pad)

# 出力データを二次元配列に戻す
img_y = img_y.reshape(output_size[0], output_size[1])
return img_y
```

12-1 | 5-2. データの読み込み

```
# Imageオブジェクトとして読み込み
img_ = Image.open("../.. /1_data/ch12/bridge.jpg")
img_
```

```
# NumPy配列に変換
img_ = np.asarray(img_)

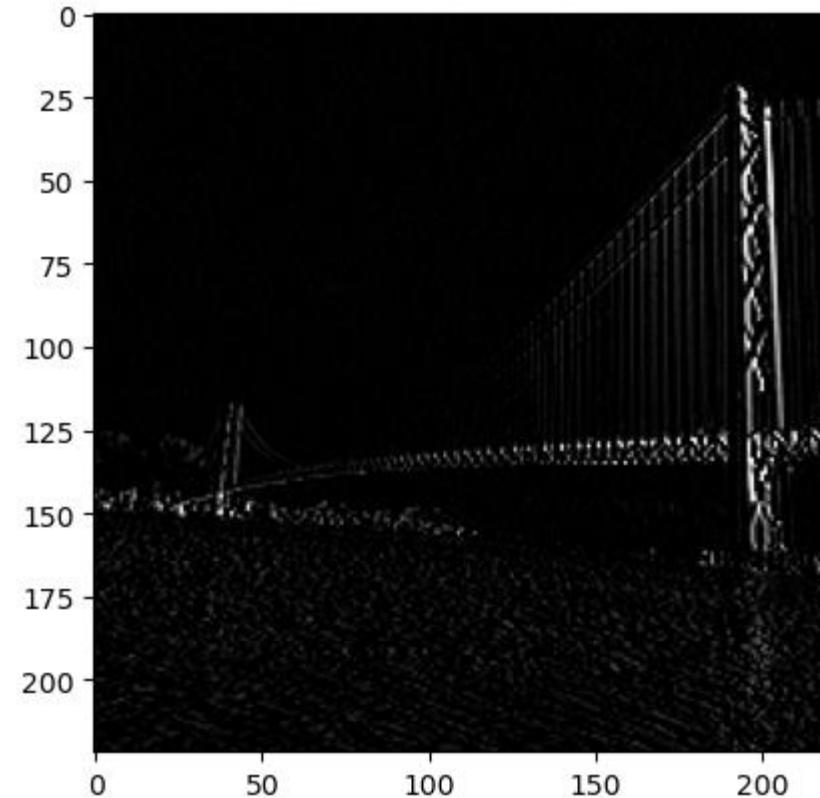
# データの形状を確認
# 3つ目の軸はチャネル（R・G・Bの三原色）
print("shape:", img_.shape)
```

shape: (224, 224, 3)



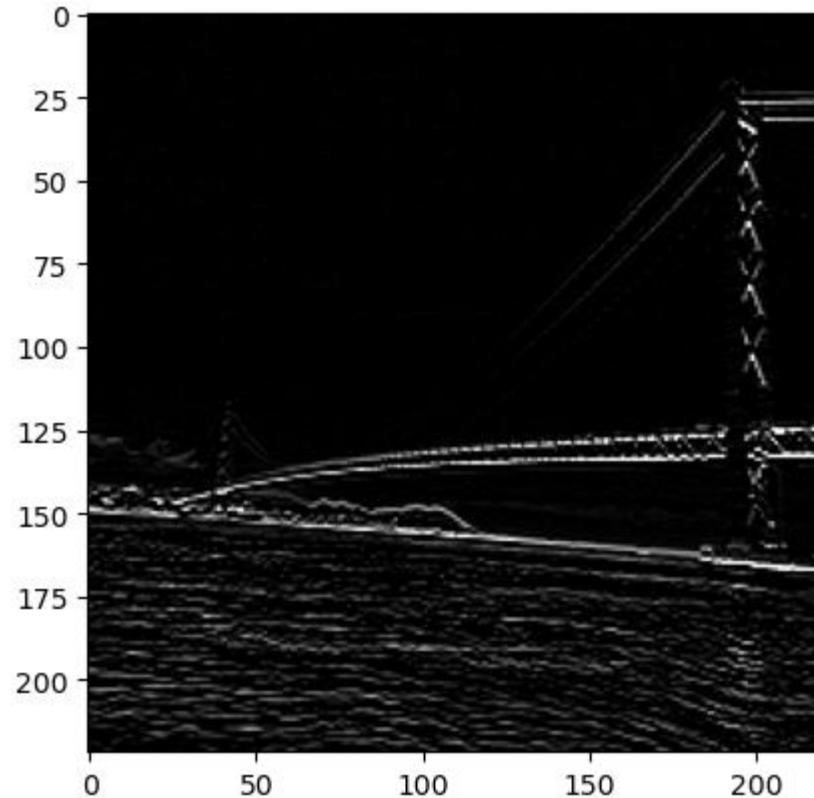
12-1 | 5-3. 置み込み演算の実行 1/2

```
# 垂直方向（縦方向）のエッジ抽出  
img_y = conv2d_multi_ch(img_, flt_dy, stride=1, pad=0)  
  
# 結果を画像として表示  
show_image(img_y)
```



12-1 | 5-3. 置み込み演算の実行 2/2

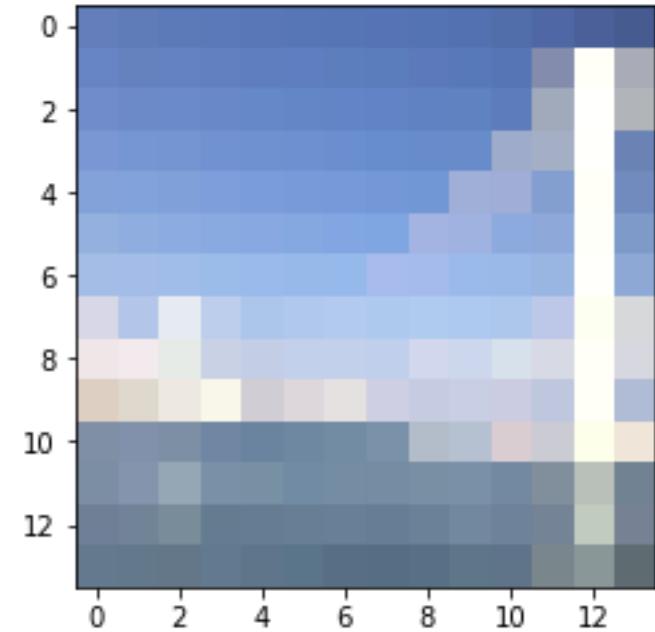
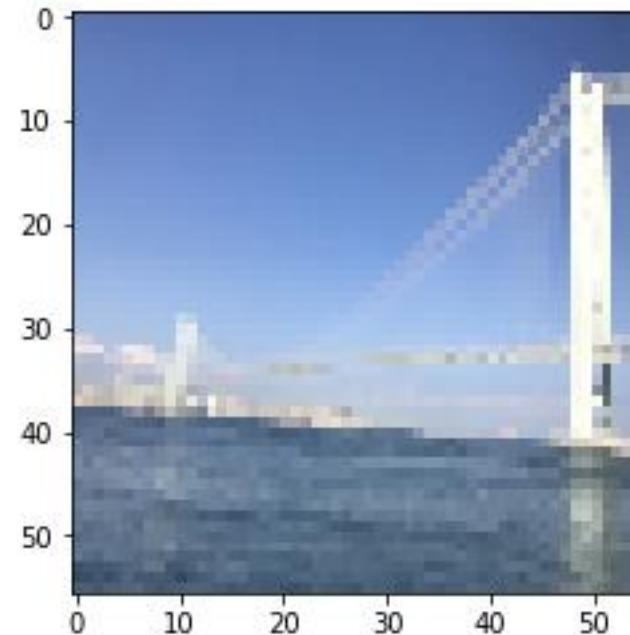
```
# 水平方向（横方向）のエッジ抽出  
img_y = conv2d_multi_ch(img_, flt_dh, stride=1, pad=0)  
  
# 結果を画像として表示  
show_image(img_y)
```



12-2_pooling

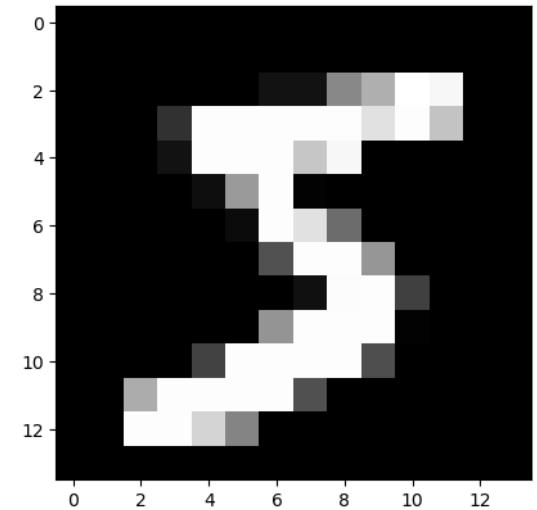
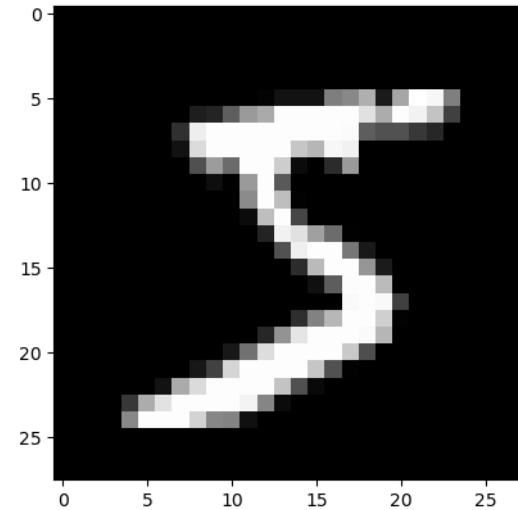
■ 内容

- ・ 関数 `im2col()` を用いて最大値プーリング (Max Pooling) を実装
- ・ 簡単な行列、モノクロ画像、カラー画像に対してそれぞれ処理を行う



■ 手順

1. ライブラリの読み込み
2. 疑似データに対するプーリング
 1. データの作成
 2. データの配列を変換
 3. プーリングの実行
3. モノクロ画像に対するプーリング
 1. プーリングを関数として定義
 2. データの読み込み
 3. プーリングの実行
4. カラー画像に対するプーリング
 1. プーリングを関数として定義
 2. データの読み込み
 3. プーリングの実行
 4. フィルタサイズを変えて実行



```
import numpy as np
import matplotlib.pyplot as plt

# 画像データを扱うライブラリ
from PIL import Image

# MNISTデータセットの読み込み
from tensorflow.keras.datasets import mnist

# フィルタ演算を行列の積に変換する関数
from common.util import im2col
```

```
img = np.array([
    [1, 2, 1, 0],
    [0, 1, 2, 3],
    [3, 0, 1, 2],
    [2, 4, 0, 1]
])

# im2col()に入力するため、軸を増やす
# (画像の枚数, チャネル数, 縦幅, 横幅)
img_expand = img.reshape(1, 1, img.shape[0], img.shape[1])

# データの形状と中身を表示
print(img_expand.shape)
img_expand
```

```
(1, 1, 4, 4)

array([[[[1, 2, 1, 0],
         [0, 1, 2, 3],
         [3, 0, 1, 2],
         [2, 4, 0, 1]]]])
```

12-2 | 2-2. データの配列を変換

```
# 画像, フィルタサイズ(縦幅, 横幅), ストライド, パディング幅
col = im2col(img_expand, 2, 2, stride=2, pad=0)
print(col.shape)
col
```

(1, 1, 4, 4)

```
array([[[[1, 2, 1, 0],  
        [0, 1, 2, 3],  
        [3, 0, 1, 2],  
        [2, 4, 0, 1]]]])
```



(4, 4)

```
array([[1., 2., 0., 1.],  
       [1., 0., 2., 3.],  
       [3., 0., 2., 4.],  
       [1., 2., 0., 1.]])
```

```
# 最大値を取得
out = np.max(col, axis=1)

# 出力を（画像の枚数, 縦幅, 横幅, チャネル数）に変形
img_y = out.reshape(1, 2, 2, 1)

# 配列の軸を（画像の枚数, チャネル数, 縦幅, 横幅）に変更
img_y = img_y.transpose(0, 3, 1, 2)
img_y
```

```
array([[[[2., 3.],
       [4., 2.]]]])
```

```
# 出力サイズを計算する関数
def calc_output_size(input_size, filter_size, padding):
    output_h = 1 + (input_size[0] + 2*padding - filter_size[0]) / filter_size[0]
    output_w = 1 + (input_size[1] + 2*padding - filter_size[1]) / filter_size[1]

# 小数点以下を切り捨て
output_h = np.floor(output_h).astype(int)
output_w = np.floor(output_w).astype(int)
return output_h, output_w
```

- プーリングの出力サイズは、以下の式で計算できる
(ストライド=フィルタの幅とする)

$$O_h = \frac{I_h + 2p - F_h}{F_h} + 1$$

$$O_w = \frac{I_w + 2p - F_w}{F_w} + 1$$

- O_h, O_w : 出力の縦幅、横幅
- I_h, I_w : 入力の縦幅、横幅
- F_h, F_w : フィルタの縦幅、横幅
- p : パディング幅

12-2 | 3-1. プーリングを関数として定義 2/2

```
def max_pool2d(img, filter_size, pad):
    # im2col()に入力するため、軸を増やす
    # (画像の枚数, チャネル数, 縦幅, 横幅)
    img_expand = img.reshape(1, 1, img.shape[0], img.shape[1])

    # im2col()で変換
    # 画像, フィルタサイズ(縦幅, 横幅), ストライド, パディング幅
    col = im2col(img_expand, filter_size[0], filter_size[1],
                  stride=filter_size[0], pad=pad)

    # 最大値を取得
    out = np.max(col, axis=1)

    # 出力サイズを計算
    output_size = calc_output_size(img.shape, filter_size, padding=pad)
    # 出力データを二次元配列に戻す
    img_y = out.reshape(output_size[0], output_size[1])

return img_y
```

12-2 | 3-2. データの読み込み

```
# 配列を画像として表示するための関数
def show_image(img):
    # 配列をImageオブジェクトに変換
    pil_img = Image.fromarray(img)
    # Imageオブジェクトを画像として表示
    plt.imshow(pil_img)
    # グレースケールで表示
    plt.gray()
    plt.show()
```

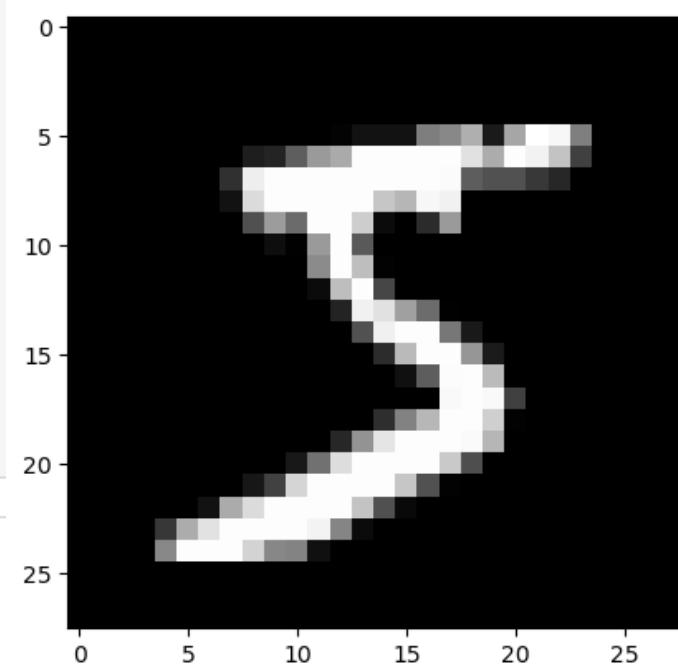
```
# MNISTデータセットの読み込み
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# データの形状を確認
print("X_train:", X_train.shape)

# クラスラベル
label = y_train[0]
# 画像データ（を表す配列）
img = X_train[0]

# クラスラベルの表示
print("label = %s" % label)
# 画像の表示
show_image(img)
```

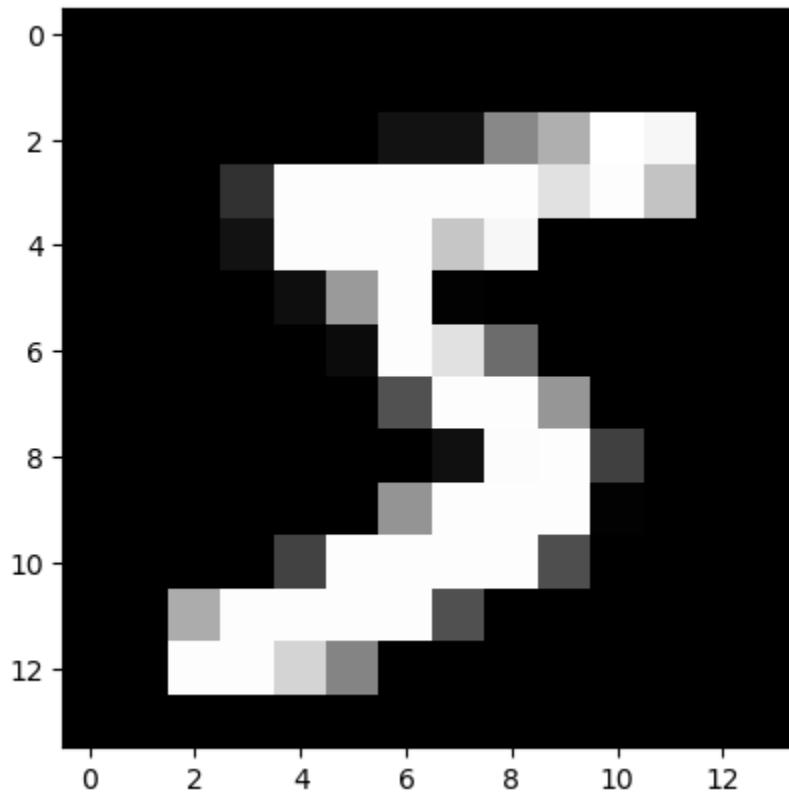
```
X_train: (60000, 28, 28)
label = 5
```



12-2 | 3-3. プーリングの実行

```
img_y = max_pool2d(img, (2, 2), pad=0)
print(img_y.shape)
show_image(img_y)
```

(14, 14)



12-2 | 4-1. プーリングを関数として定義

```
def max_pool2d_multi(img, filter_size, pad):
    # 配列の軸を入れ替え
    # (高さ, 幅, チャネル数) → (チャネル数, 高さ, 幅)
    img_t = img.transpose(2, 0, 1)

    # im2col()に入力するため、軸を増やす
    # (画像の枚数, チャネル数, 縦幅, 横幅)
    img_expand = img_t.reshape(1, img_t.shape[0], img_t.shape[1], img_t.shape[2])

    # im2col()で変換
    # 画像, フィルタサイズ(縦幅, 横幅), ストライド, パディング幅
    col = im2col(img_expand, filter_size[0], filter_size[1],
                  stride=filter_size[0], pad=pad)

    # col.shape = (..., フィルタサイズ*フィルタサイズ*チャネル数)
    # colの各行には3チャネル分の値が入っている
    # max(col, axis=1)を行うと、3チャネル全体でのmaxを取ることになる
    # 各チャンネルでmaxを取りたいため、チャンネルを別々の行に分離する
    col_ = col.reshape(-1, filter_size[0]*filter_size[1])

    # 最大値を取得
    out = np.max(col_, axis=1)
```

```
# 入力サイズをタプルにまとめる
input_size = (img_t.shape[1], img_t.shape[2])
# 出力サイズを計算
output_size = calc_output_size(input_size, filter_size, padding=pad)

# 出力データをカラーの二次元配列に戻す
img_y = out.reshape(output_size[0], output_size[1], img_t.shape[0])

return np.uint8(img_y)
```

12-2 | 4-2. データの読み込み

```
# Imageオブジェクトとして読み込み
img_ = Image.open("../.. /1_data/ch12/bridge.jpg")
img_
```

```
# NumPy配列に変換
img_ = np.asarray(img_)

# データの形状を確認
# 3つ目の軸はチャネル（R・G・Bの三原色）
print("shape:", img_.shape)
```

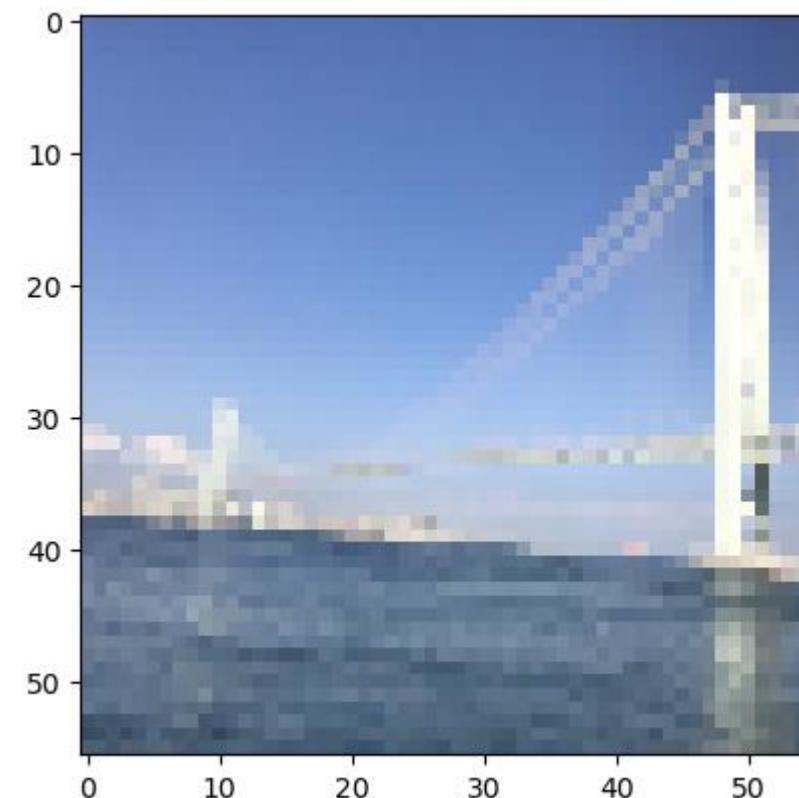
shape: (224, 224, 3)



12-2 | 4-3. プーリングの実行

```
img_y = max_pool2d_multi(img_, (4, 4), pad=0)
print(img_y.shape)
show_image(img_y)
```

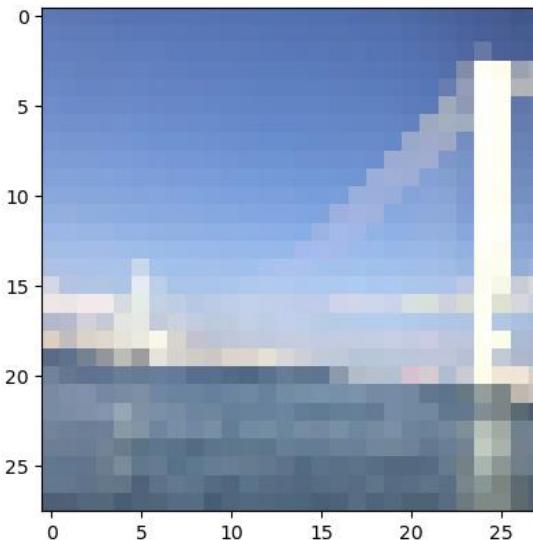
(56, 56, 3)



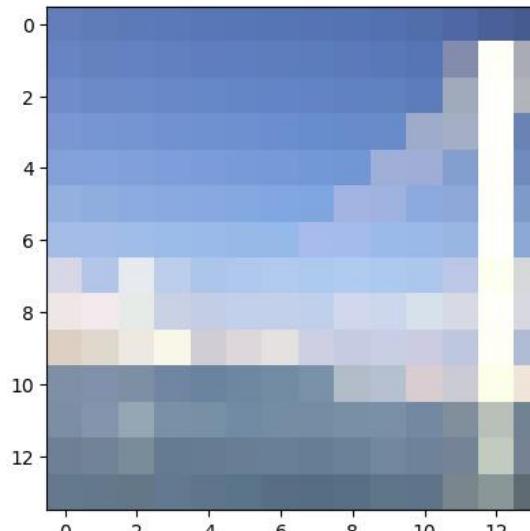
12-2 | 4-4. フィルタサイズを変えて実行

```
for size in [4,8,16,32]:  
    print("filter size = %s"%size)  
    img_y = max_pool2d_multi(img_, (size, size), pad=0)  
    print("output shape:", img_y.shape)  
    show_image(img_y)
```

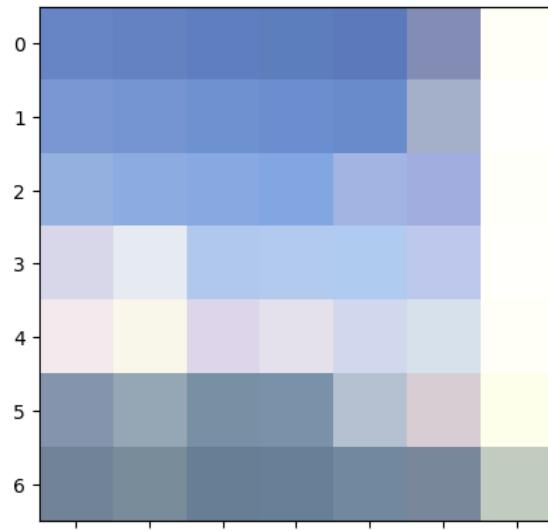
filter size = 8
output shape: (28, 28, 3)



filter size = 16
output shape: (14, 14, 3)



filter size = 32
output shape: (7, 7, 3)



12-3_CNN

■ 内容

- [EfficientNet](#) の学習済みモデルを読み込んで、画像分類を行う
- 深層学習ライブラリの一つである TensorFlow を用いる

■ 手順

1. ライブラリの読み込み
2. データの読み込み
3. データの前処理
4. モデルの読み込み
5. 予測の実行



```
[('n09332890', 'lakeside', 0.0895904),  
 ('n02782093', 'balloon', 0.08481371),  
 ('n09428293', 'seashore', 0.06655197),  
 ('n03888257', 'parachute', 0.04206922),  
 ('n03355925', 'flagpole', 0.036196813),  
 ('n03792972', 'mountain_tent', 0.030743118),  
 ('n03376595', 'folding_chair', 0.030592278),  
 ('n03733281', 'maze', 0.020027254),  
 ('n09421951', 'sandbar', 0.017280469),  
 ('n03127925', 'crate', 0.015329881)]
```

12-3 | 1. ライブラリの読み込み

```
import numpy as np
from PIL import Image

# データをテンソルとして読み込む
from tensorflow.io import read_file
# 画像のテンソルを加工
from tensorflow.image import decode_png, resize
# テンソルの軸を増やす
from tensorflow import expand_dims

# EfficientNetの学習済みモデル
from tensorflow.keras.applications import EfficientNetB0
# 予測結果を読み取りやすい形式に変換
from tensorflow.keras.applications.imagenet_utils import decode_predictions
```

12-3 | 2. データの読み込み

```
# 画像のパスを設定  
file_path = " ../../1_data/ch12/park.jpg"  
  
# 画像の確認  
img = Image.open(file_path)  
img
```

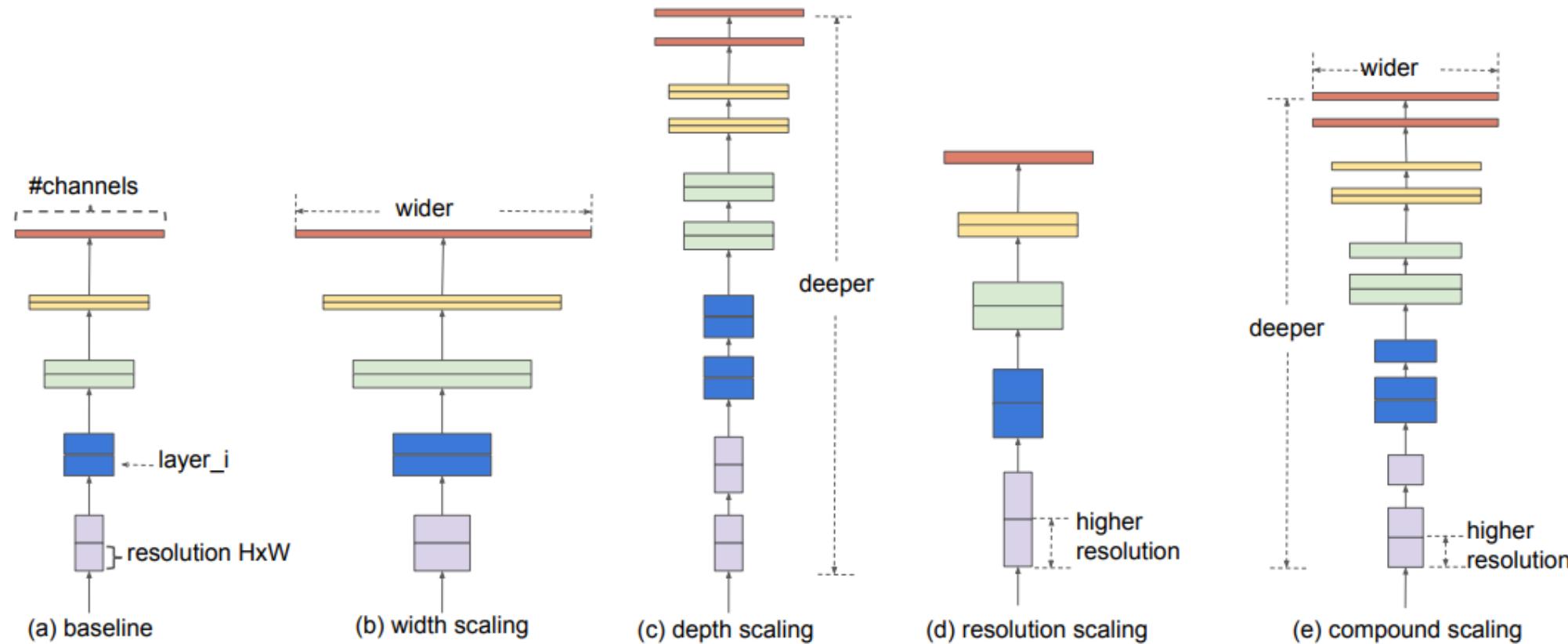


```
# 学習時の画像サイズ
IMAGE_SIZE = 224

def get_image(filename):
    # TensorFlow用の形式で読み込む
    img = read_file(filename)
    img = decode_png(img, channels=3)
    # 224×224にリサイズ
    img = resize(img, (IMAGE_SIZE, IMAGE_SIZE))
    # データの軸を増やす
    img = expand_dims(img, axis=0)
    return img
```

12-3 | 4. モデルの読み込み

```
# EfficientNetB0のうち、ImageNetを学習させたものを読み込む  
model = EfficientNetB0(weights="imagenet")
```



出典：[EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks](#)

12-3 | 5. 予測の実行

```
# データの読み込み・前処理
image = get_image(file_path)
print("input shape: ", image.shape)

# 1000クラス分の確率値を計算
result = model.predict(image)

# トップ10のクラス名とその確率を出力
decode_predictions(result, top=10)[0]
```

```
input shape: (1, 224, 224, 3)
1/1 [=====] - 2s 2s/step
```

```
[('n09332890', 'lakeside', 0.0895904),
 ('n02782093', 'balloon', 0.08481371),
 ('n09428293', 'seashore', 0.06655197),
 ('n03888257', 'parachute', 0.04206922),
 ('n03355925', 'flagpole', 0.036196813),
 ('n03792972', 'mountain_tent', 0.030743118),
 ('n03376595', 'folding_chair', 0.030592278),
 ('n03733281', 'maze', 0.020027254),
 ('n09421951', 'sandbar', 0.017280469),
 ('n03127925', 'crate', 0.015329881)]
```

現場で使える 機械学習・データ分析基礎講座

第 14 章：教師なし学習

ノートブック解説

■ 教師なし学習

- 14-1_k-means_artificial.ipynb
- 14-2_k-means_real.ipynb
- 14-3_PCA.ipynb
- 14-4_AutoEncoder.ipynb

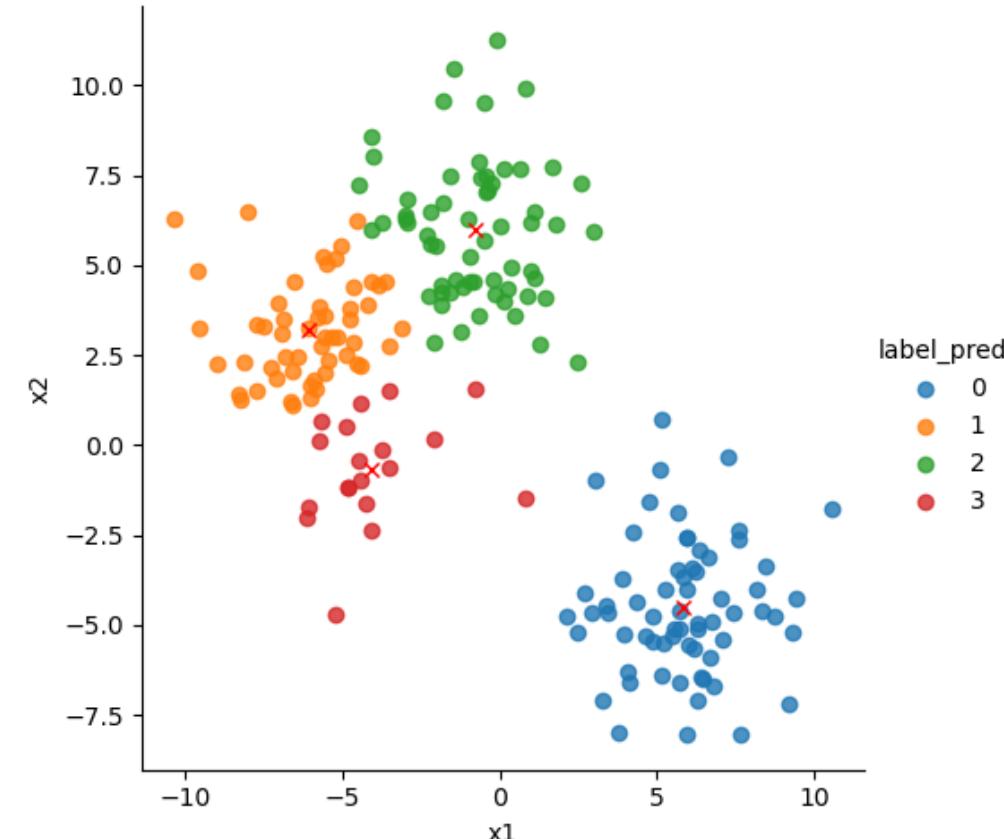
14-1_k-means_artificial

■ 内容

- 疑似データを用いて、k-means 法の学習と予測を実行
- クラスタ数 k を変化させて、結果がどのように変わるか確認

■ 手順

1. ライブラリの読み込み
2. 疑似データの生成
3. クラスタリングの実行
 1. グラフ表示用の関数を定義
 2. $k=2$ の場合
 3. $k=3$ の場合
 4. $k=4$ の場合
4. エルボー法の実行



14-1 | 1. ライブラリの読み込み

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# 疑似データの生成
from sklearn.datasets import make_blobs
# k-means法の実装
from sklearn.cluster import KMeans

# Warningを非表示にする
import warnings
warnings.simplefilter('ignore')
```

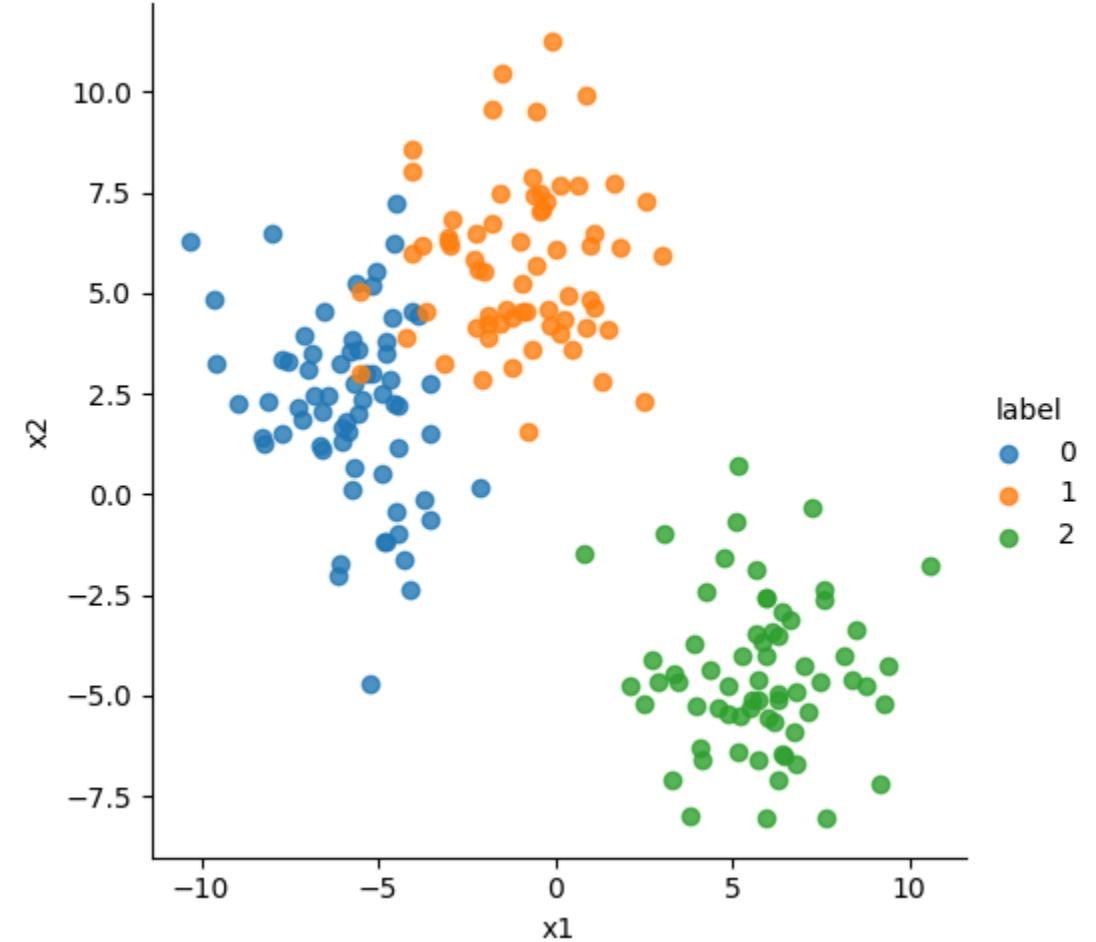
14-1 | 2. 疑似データの生成

```
# 特徴Xとラベルyを生成
# サンプル数、クラスタ数、特徴の数、標準偏差、乱数シードを指定
X, y = make_blobs(
    n_samples=200, centers=3, n_features=2,
    cluster_std=2.0, random_state=1234
)

# 特徴Xをデータフレームに変換
df_data = pd.DataFrame(X, columns=["x1", "x2"])
# ラベルy（クラスタ）をデータフレームに追加
df_data["label"] = y

# データの確認
display(df_data.head())
# データの散布図を表示
# クラスタで色分け
sns.lmplot(
    x="x1", y="x2", hue="label",
    data=df_data, fit_reg=False
)
plt.show()
```

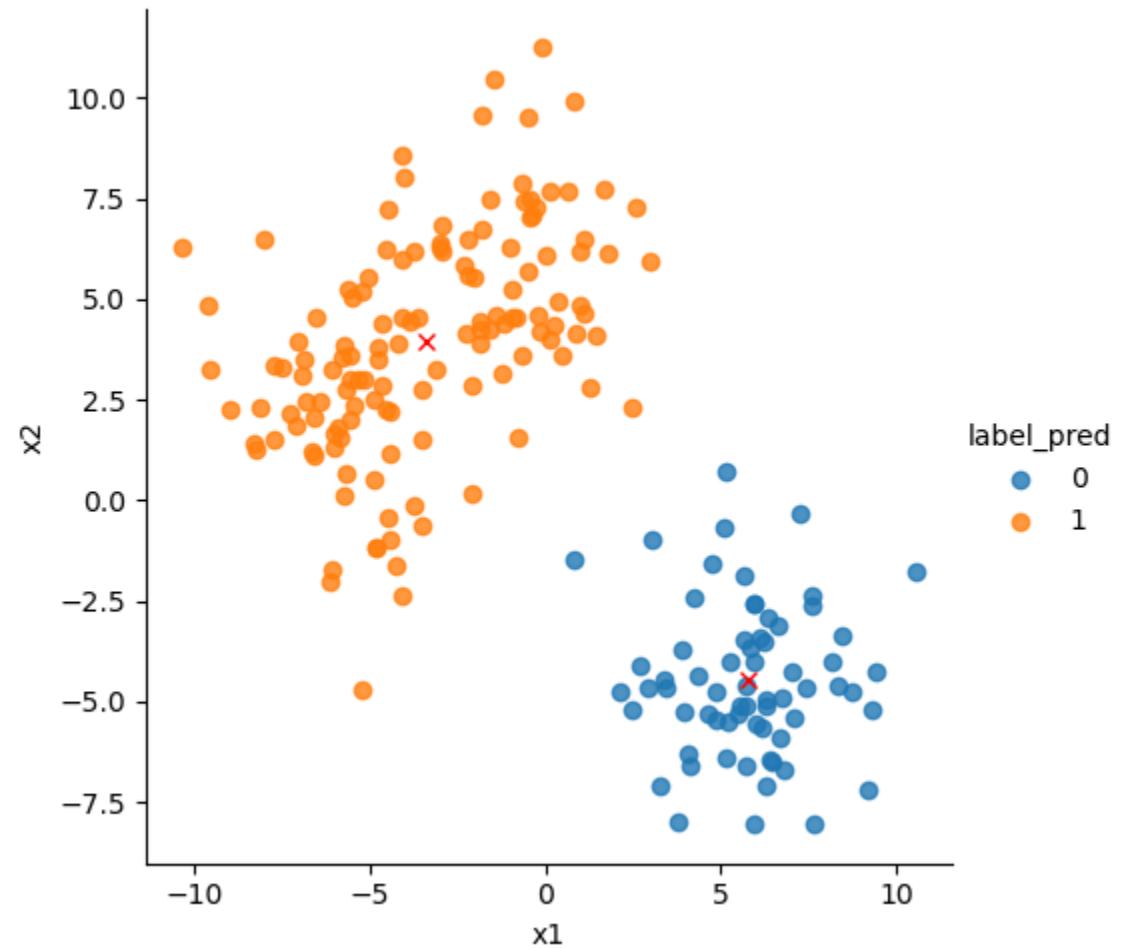
	x1	x2	label
0	0.252924	4.356129	1
1	-1.003102	6.305140	1
2	0.159679	4.006479	1
3	-6.577477	2.077825	0
4	0.012106	6.080160	1



14-1 | 3-1. グラフ表示用の関数を定義

```
def plot_clusters(df_data, y_pred, centers):
    # データフレームにラベル（クラスタ）の予測値を追加
    df_data[ "label_pred" ] = y_pred

    # データの散布図を表示
    # クラスタごとに色分け
    sns.lmplot(
        x="x1", y="x2", hue="label_pred",
        data=df_data, fit_reg=False
    )
    # クラスタの中心点を赤色の×で表示
    plt.plot(
        centers[:,0], centers[:,1],
        marker="x", ls="", color="r"
    )
```

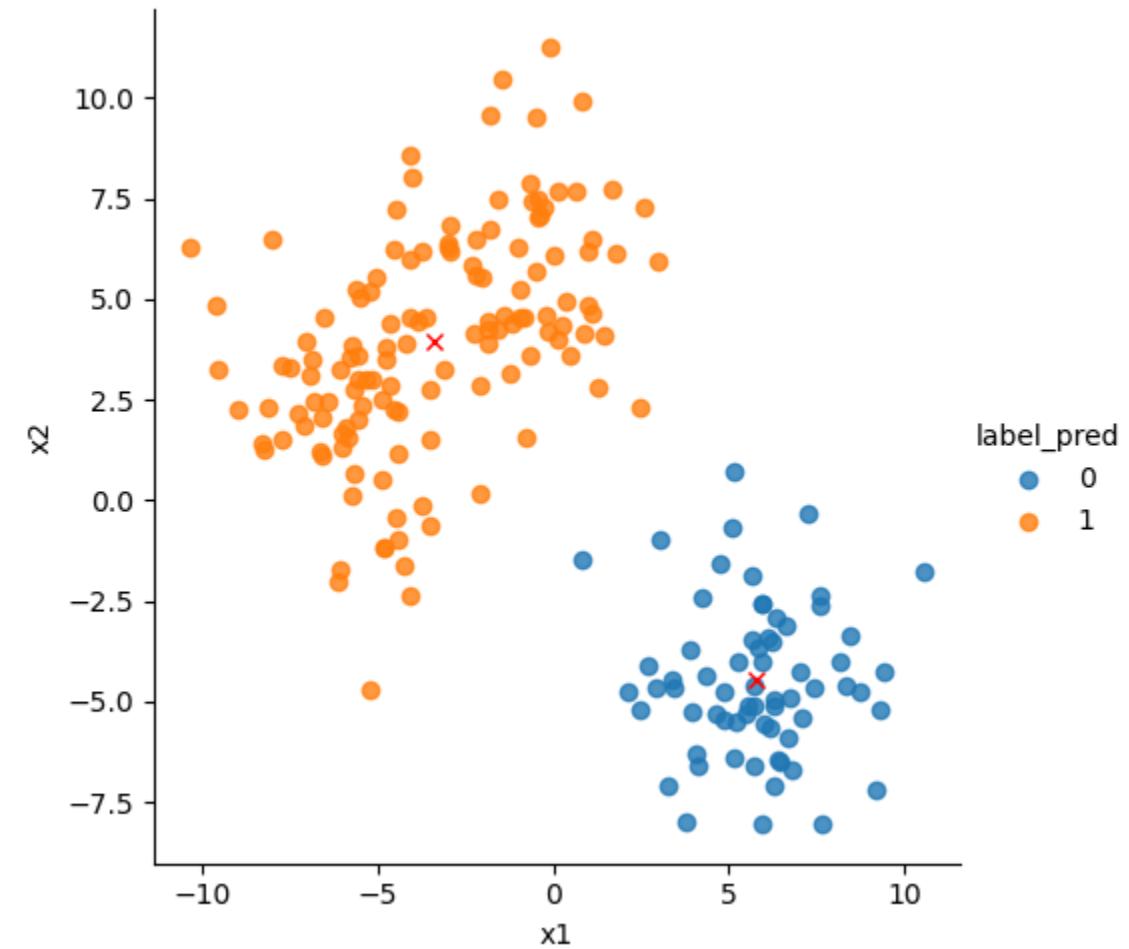


14-1 | 3-2. k=2 の場合

```
# クラスタ数を2に設定
k = 2
# k-meansのモデルを構築
clf = KMeans(n_clusters=k)
# モデルの学習
clf.fit(X)

# 予測値の計算
y_pred = clf.predict(X)
# 中心点を取得
centers = clf.cluster_centers_

# クラスタリング結果の描画
# 赤色の×は中心点
plot_clusters(df_data, y_pred, centers)
```

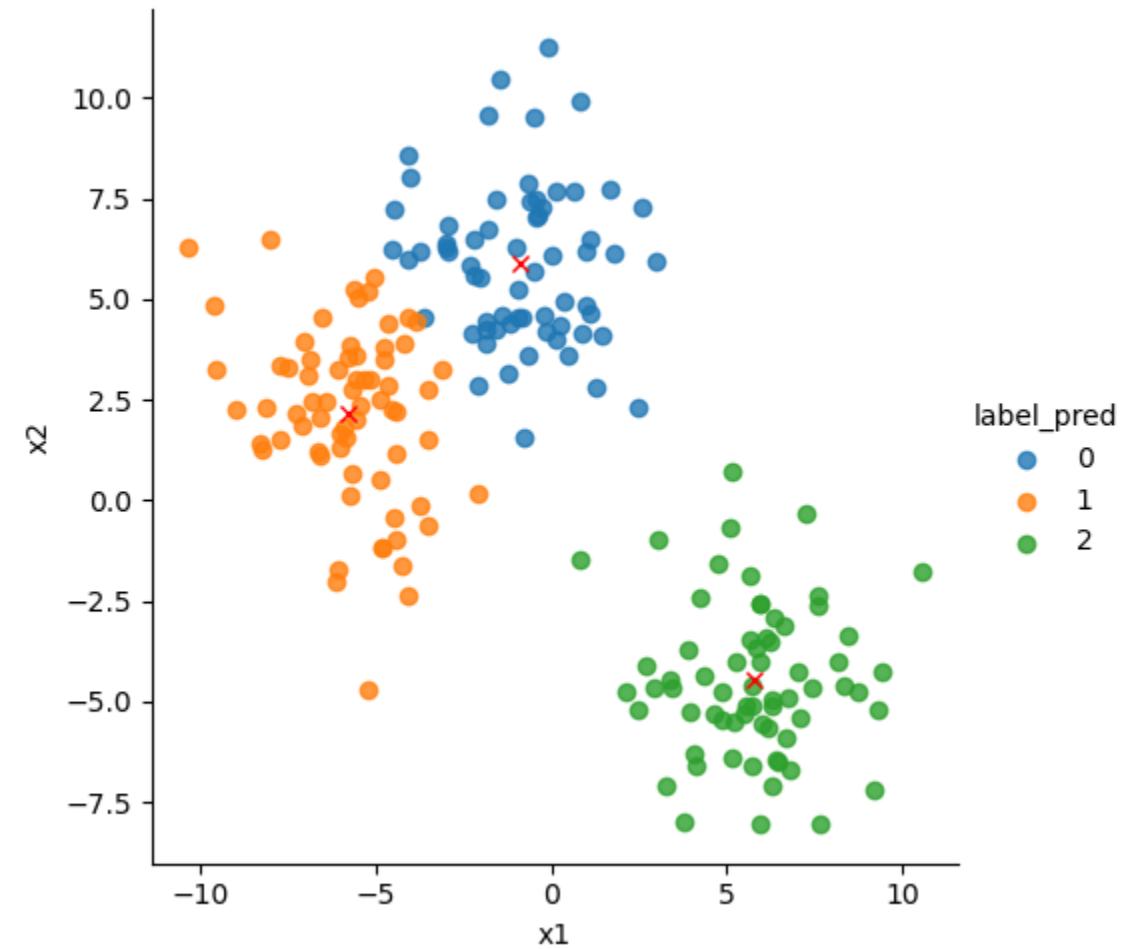


14-1 | 3-3. k=3 の場合

```
# クラスタ数を3に設定
k = 3
# k-meansのモデルを構築
clf = KMeans(n_clusters=k)
# モデルの学習
clf.fit(X)

# 予測値の計算
y_pred = clf.predict(X)
# 中心点を取得
centers = clf.cluster_centers_

# クラスタリング結果の描画
# 赤色の×は中心点
plot_clusters(df_data, y_pred, centers)
```

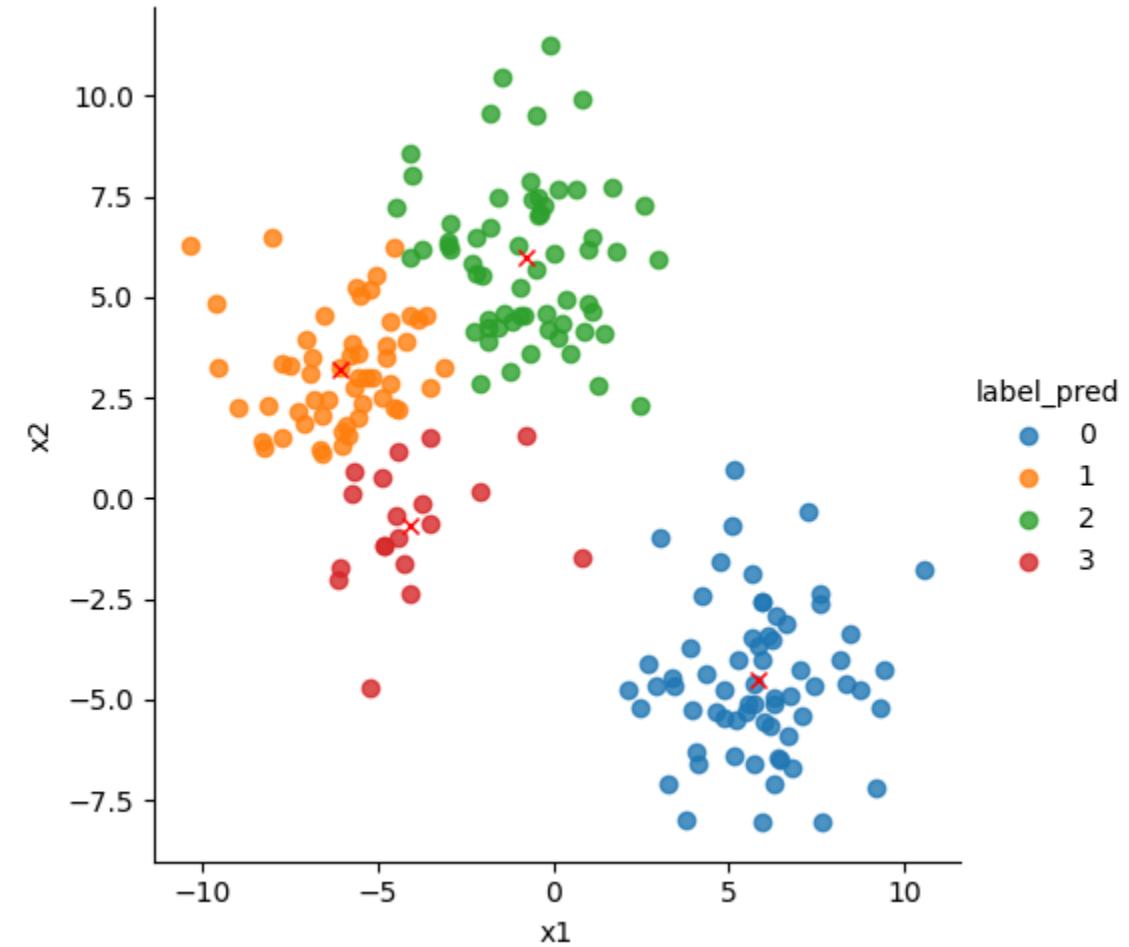


14-1 | 3-4. k=4 の場合

```
# クラスタ数を4に設定
k = 4
# k-meansのモデルを構築
clf = KMeans(n_clusters=k)
# モデルの学習
clf.fit(X)

# 予測値の計算
y_pred = clf.predict(X)
# 中心点を取得
centers = clf.cluster_centers_

# クラスタリング結果の描画
# 赤色の×は中心点
plot_clusters(df_data, y_pred, centers)
```



14-1 | 4. エルボー法の実行

```
# エルボー法を行い、その結果をグラフ化
def plot_elbow(X, k_start, k_end):
    # SSEを格納するための配列
    sse = []
    # クラスタ数をk_startからk_endまで増やす
    for i in range(k_start, k_end+1):
        # モデルの構築
        clf = KMeans(n_clusters=i, random_state=1234)
        # モデルの学習
        clf.fit(X)
        # SSEの値を取得して配列に追加
        sse.append(clf.inertia_)

    # クラスタ数とSSEをまとめたデータフレームを作成
    df = pd.DataFrame({
        'Number of clusters': range(k_start, k_end+1),
        'SSE': sse
    })
    # グラフの表示
    sns.lineplot(x='Number of clusters', y='SSE', data=df, marker="o")
    plt.show()

# 関数の実行
plot_elbow(X, k_start=1, k_end=10)
```

クラスタ内誤差平方和 (SSE)

$$SSE = \sum_{i=1}^n \sum_{j=1}^k w^{(i,j)} \|x^{(i)} - \mu^{(j)}\|_2^2, \quad w^{(i,j)} = \begin{cases} 1 & x^{(i)} \in C_j \\ 0 & x^{(i)} \notin C_j \end{cases}$$

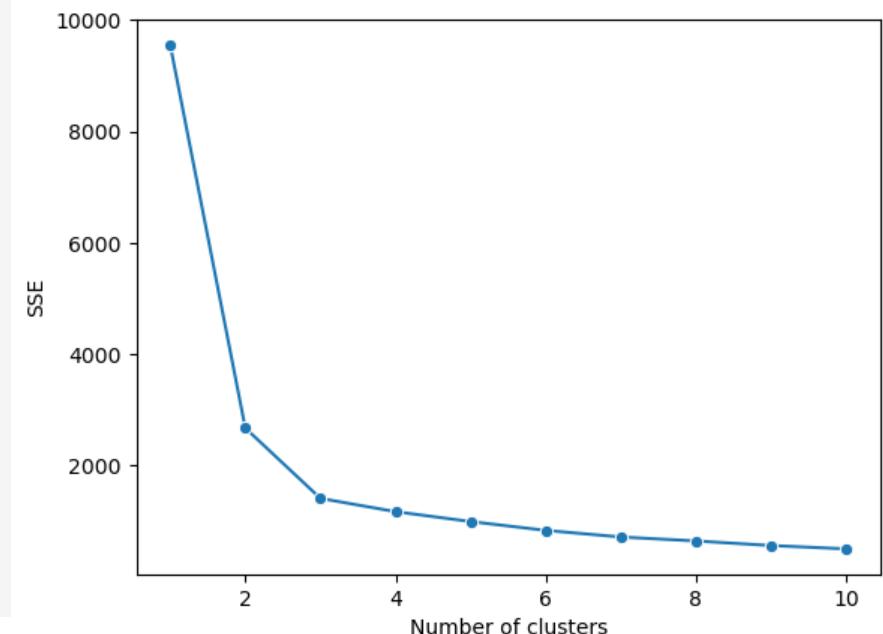
n : データ数

k : クラスタ数

$x^{(i)}$: i 番目のサンプル点

μ_j : j 番目のクラスタの中心点

C_j : j 番目のクラスタ



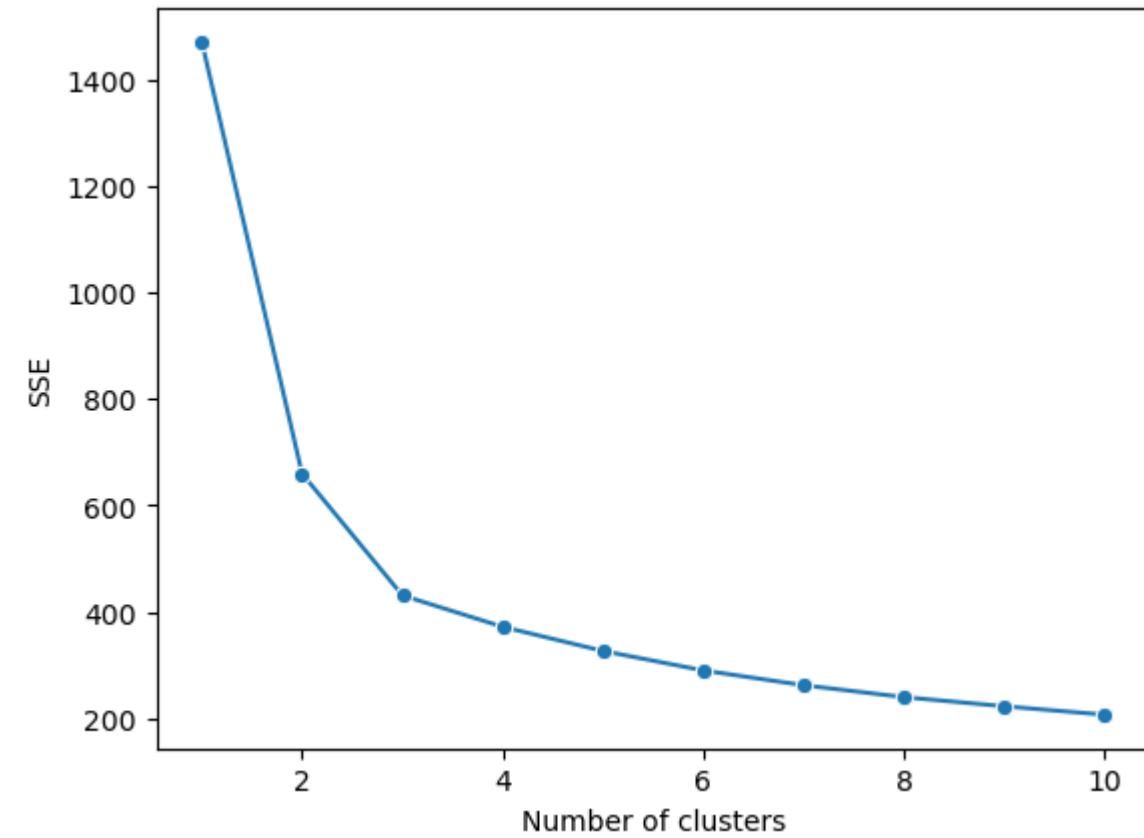
14-2_k-means_real

■ 内容

- クラスタ数が既知の疑似データに加えて、クラスタ数が未知の実データを学習

■ 手順

1. ライブラリの読み込み
2. クラスタ数が既知のデータ
 1. 疑似データの生成
 2. グラフ表示用の関数を定義
 3. クラスタリングの実行
 4. エルボー法の実行
3. クラスタ数が未知のデータ
 1. データの読み込み
 2. データの可視化
 3. エルボー法の実行



14-2 | 1. ライブラリの読み込み

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# 疑似データの生成
from sklearn.datasets import make_blobs
# k-means法の実装
from sklearn.cluster import KMeans

# Warningを非表示にする
import warnings
warnings.simplefilter('ignore')

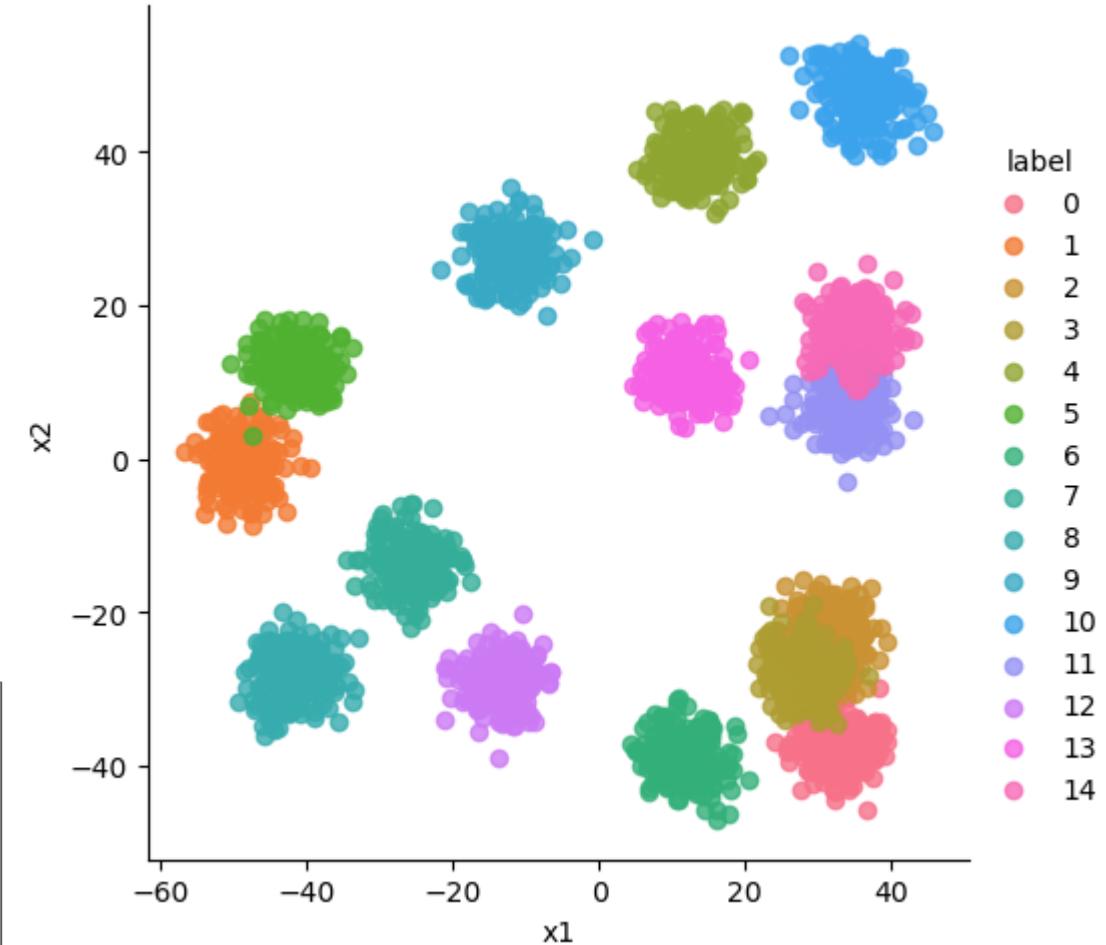
# 標準化を行う
from sklearn.preprocessing import StandardScaler
```

14-2 | 2-1. 疑似データの作成

```
# 特徴Xとラベルyを生成
# サンプル数、クラスタ数、特徴の数、標準偏差、乱数シードを指定
# さらに中心点の範囲も指定
X, y = make_blobs(
    n_samples=3000, centers=15, n_features=2,
    cluster_std=3, random_state=2532689,
    center_box=(-50, 50)
)
# 特徴Xをデータフレームに変換
df_data = pd.DataFrame(X, columns=["x1", "x2"])
# ラベルy（クラスタ）をデータフレームに追加
df_data["label"] = y

# データの確認
display(df_data.head())
# データの散布図を表示
# クラスタで色分け
sns.lmplot(
    x="x1", y="x2", hue="label",
    data=df_data, fit_reg=False
)
plt.show()
```

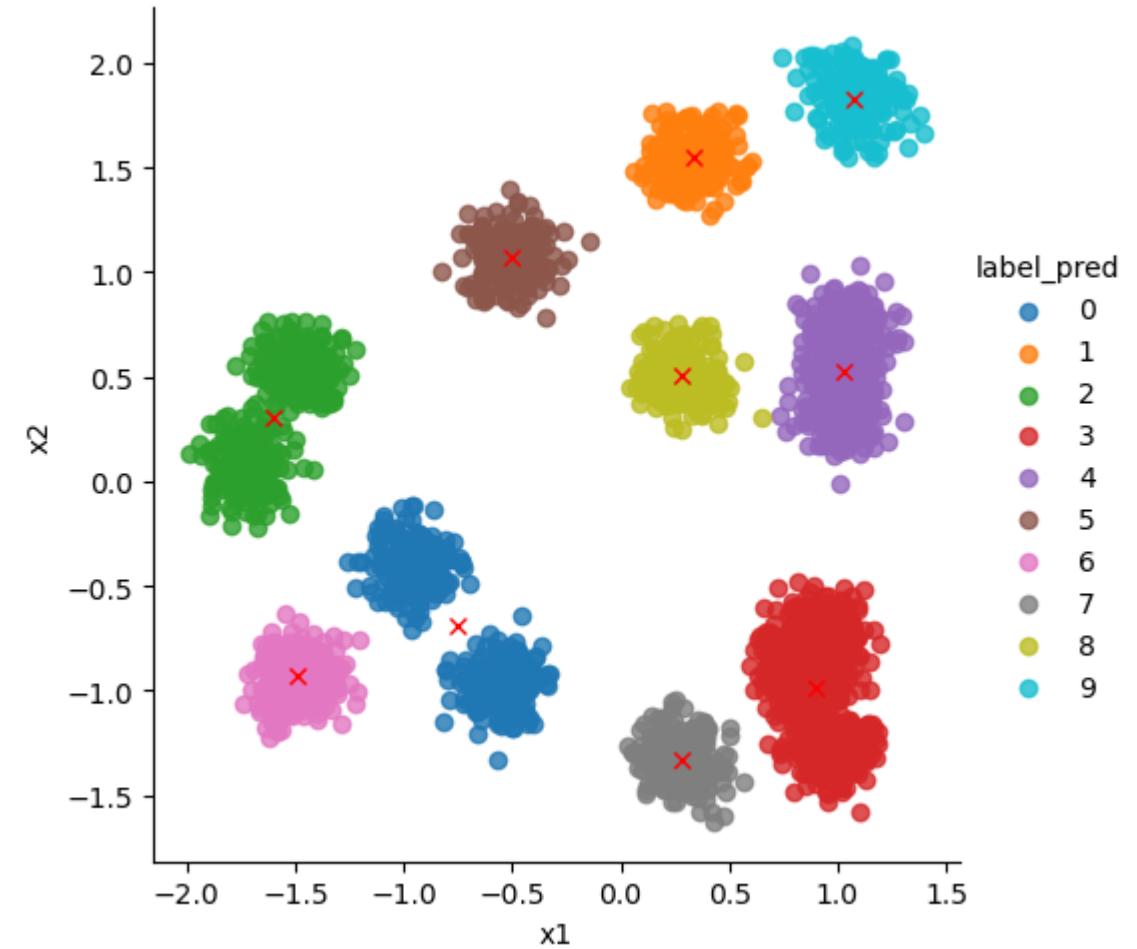
	x1	x2	label
0	-46.233528	-31.385499	8
1	26.591767	-30.851087	3
2	-23.305483	-14.692841	7
3	-24.405911	-11.207747	7
4	-23.820724	-12.611908	7



14-2 | 2-2. グラフ表示用の関数を定義

```
def plot_clusters(df_data, y_pred, centers):
    # データフレームにラベル（クラスタ）の予測値を追加
    df_data[ "label_pred" ] = y_pred

    # データの散布図を表示
    # クラスタごとに色分け
    sns.lmplot(
        x="x1", y="x2", hue="label_pred",
        data=df_data, fit_reg=False
    )
    # クラスタの中心点を赤色の×で表示
    plt.plot(
        centers[:,0], centers[:,1],
        marker="x", ls="", color="r"
    )
```



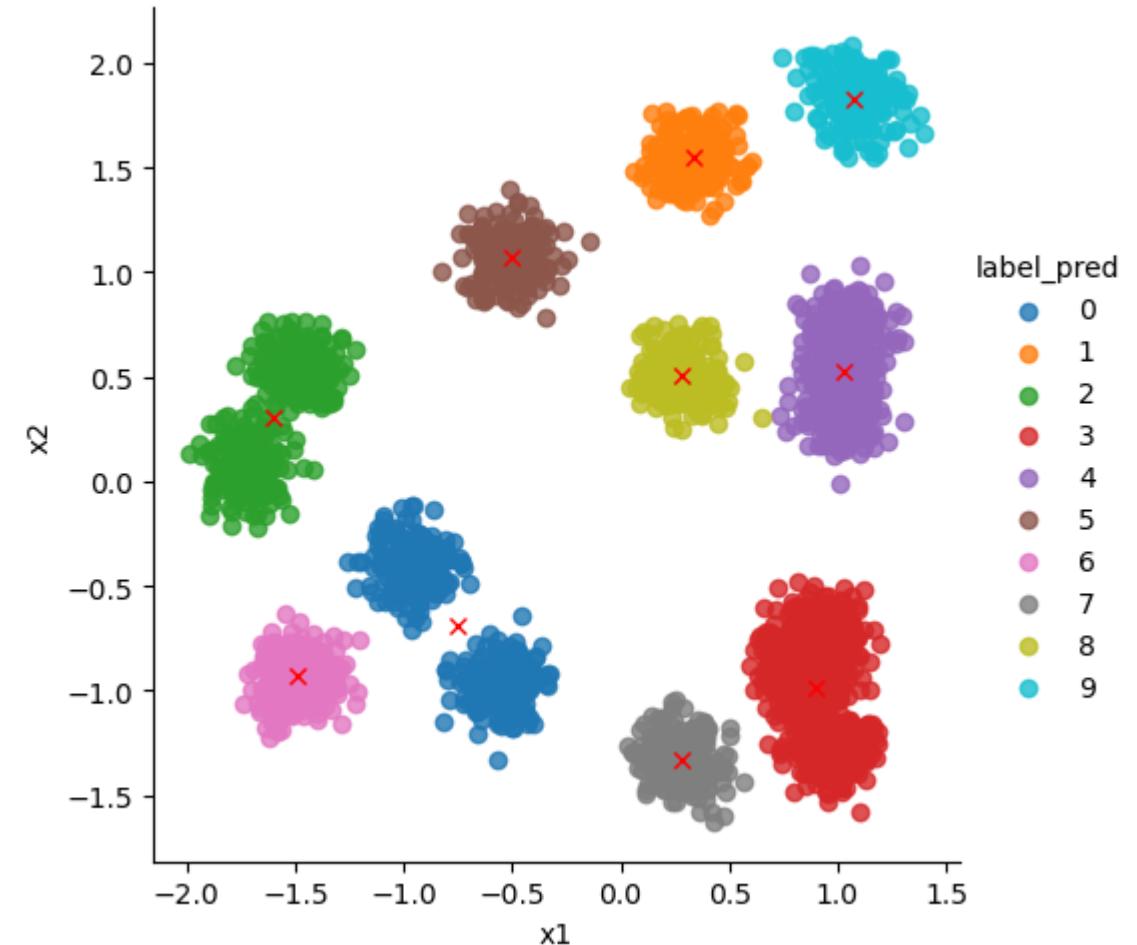
14-2 | 2-3. クラスタリングの実行

```
# 特徴Xの標準化
scaler = StandardScaler()
X_fake_std = scaler.fit_transform(X)
# データフレームを更新
df_data = pd.DataFrame(X_fake_std, columns= [ "x1" , "x2" ])

# クラスタ数10のモデルを構築
k = 10
clf = KMeans(n_clusters=k)
# モデルの学習
clf.fit(X_fake_std)

# 予測値の計算
y_pred = clf.predict(X_fake_std)
# 中心点を取得
centers = clf.cluster_centers_

# クラスタリング結果の描画
# 赤色の×は中心点
plot_clusters(df_data, y_pred, centers)
```

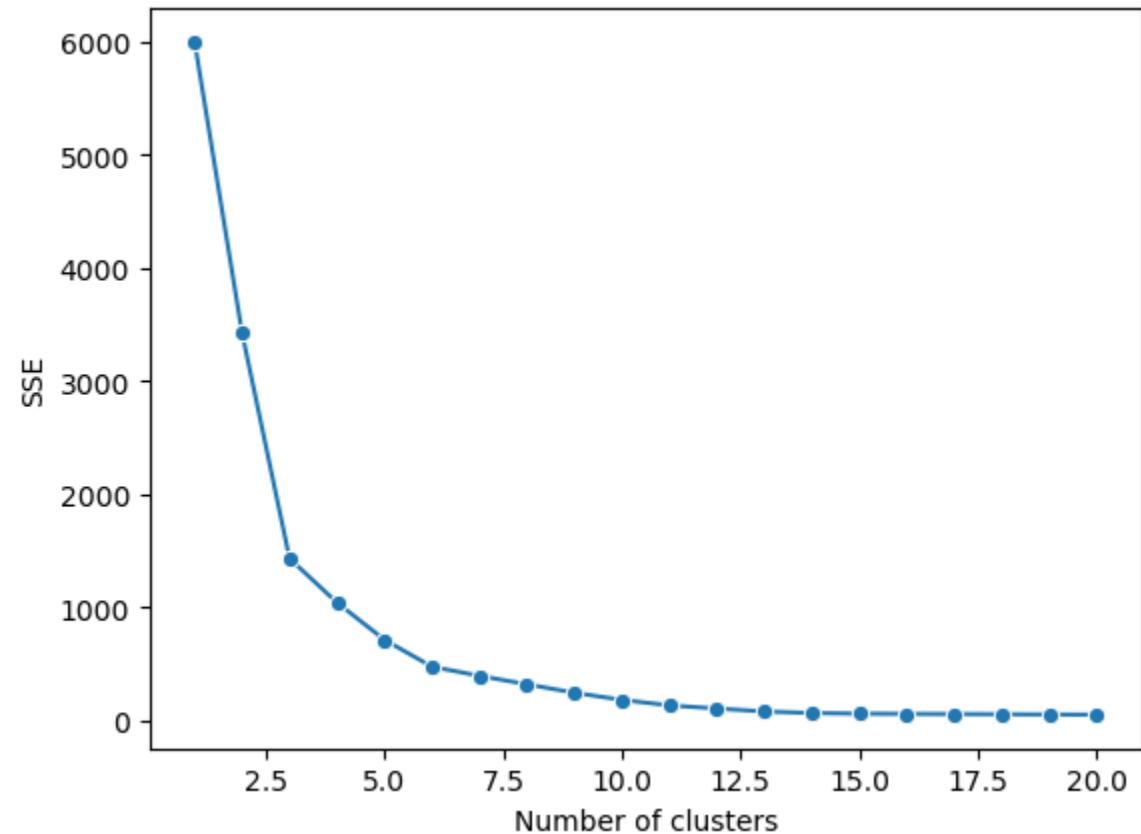


14-2 | 2-4. エルボー法の実行

```
# エルボー法を行い、その結果をグラフ化
def plot_elbow(X, k_start, k_end):
    # SSEを格納するための配列
    sse = []
    # クラスタ数をk_startからk_endまで増やす
    for i in range(k_start, k_end+1):
        # モデルの構築
        clf = KMeans(n_clusters=i, random_state=1234)
        # モデルの学習
        clf.fit(X)
        # SSEの値を取得して配列に追加
        sse.append(clf.inertia_)

    # クラスタ数とSSEをまとめたデータフレームを作成
    df = pd.DataFrame({
        'Number of clusters': range(k_start, k_end+1),
        'SSE': sse
    })
    # グラフの表示
    sns.lineplot(x='Number of clusters', y='SSE', data=df, marker="o")
    plt.show()

# 関数の実行
plot_elbow(X_fake_std, k_start=1, k_end=20)
```



14-2 | 3-1. データの読み込み

```
# CSVファイルの読み込み
df_seeds = pd.read_csv("../.. /1_data/ch14/seeds_dataset.csv", header=None, sep="\t")

# 列名を文字列に置き換え
feature_names = ["A", "P", "C", "LK", "WK", "A_coef", "LKG"]
df_seeds.columns = feature_names

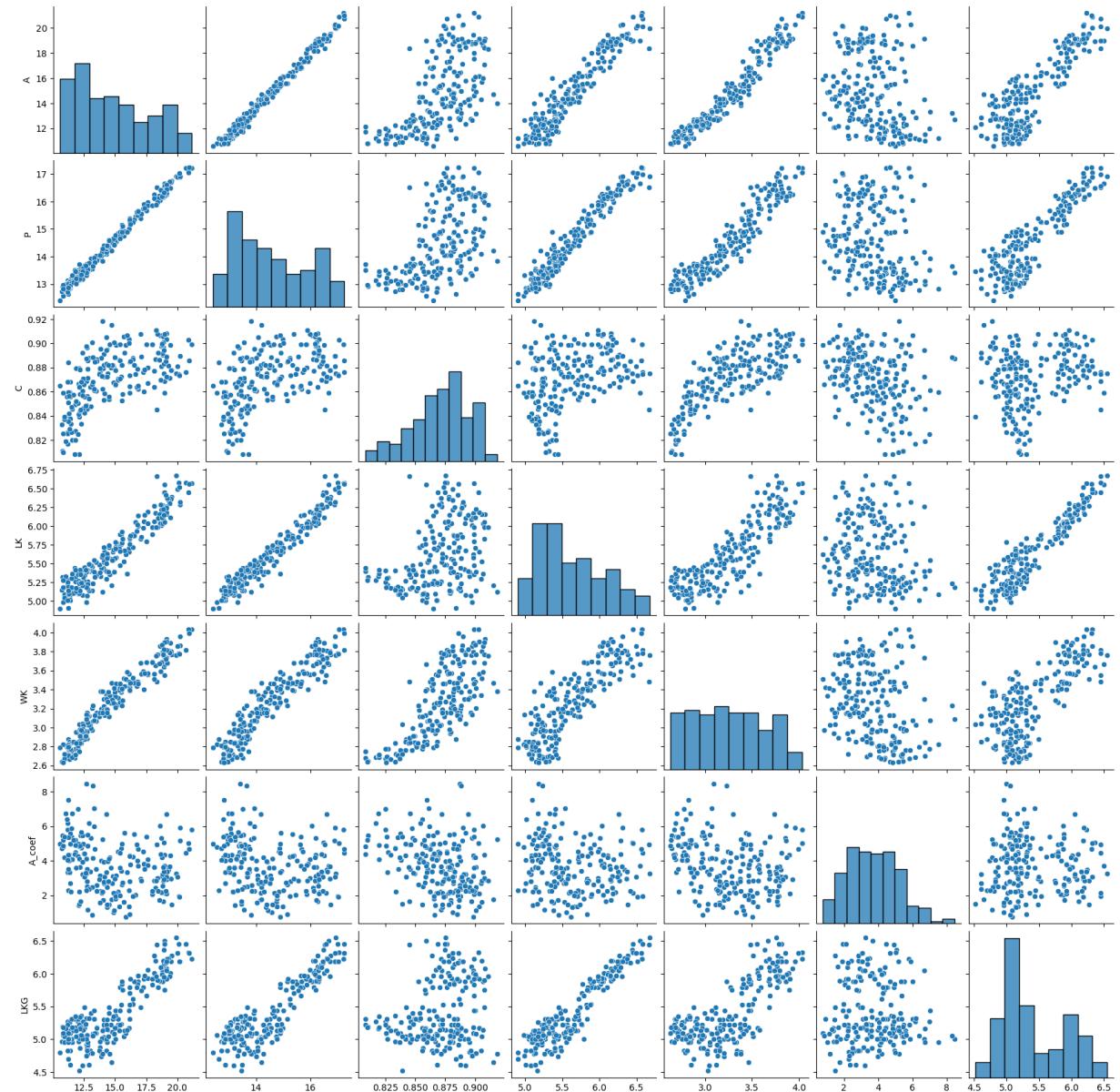
# データフレームの形状と中身を確認
print(df_seeds.shape)
display(df_seeds.head())
display(df_seeds.describe())
```

	A	P	C	LK	WK	A_coef	LKG
0	15.26	14.84	0.8710	5.763	3.312	2.221	5.220
1	14.88	14.57	0.8811	5.554	3.333	1.018	4.956
2	14.29	14.09	0.9050	5.291	3.337	2.699	4.825
3	13.84	13.94	0.8955	5.324	3.379	2.259	4.805
4	16.14	14.99	0.9034	5.658	3.562	1.355	5.175

- 説明変数：
 - i. area, A : 粒の断面積
 - ii. perimeter, P : 粒の外周
 - iii. compactness, C = $\frac{4\pi A}{P^2}$: 粒の緊密さ
 - iv. length of kernel, LK : 粒の長さ
 - v. width of kernel, WK : 粒の幅
 - vi. asymmetry coefficient, A_coef : 非対称性係数
 - vii. length of kernel groove, LKG : 粒の表面にある溝の長さ

14-2 | 3-2. データの可視化

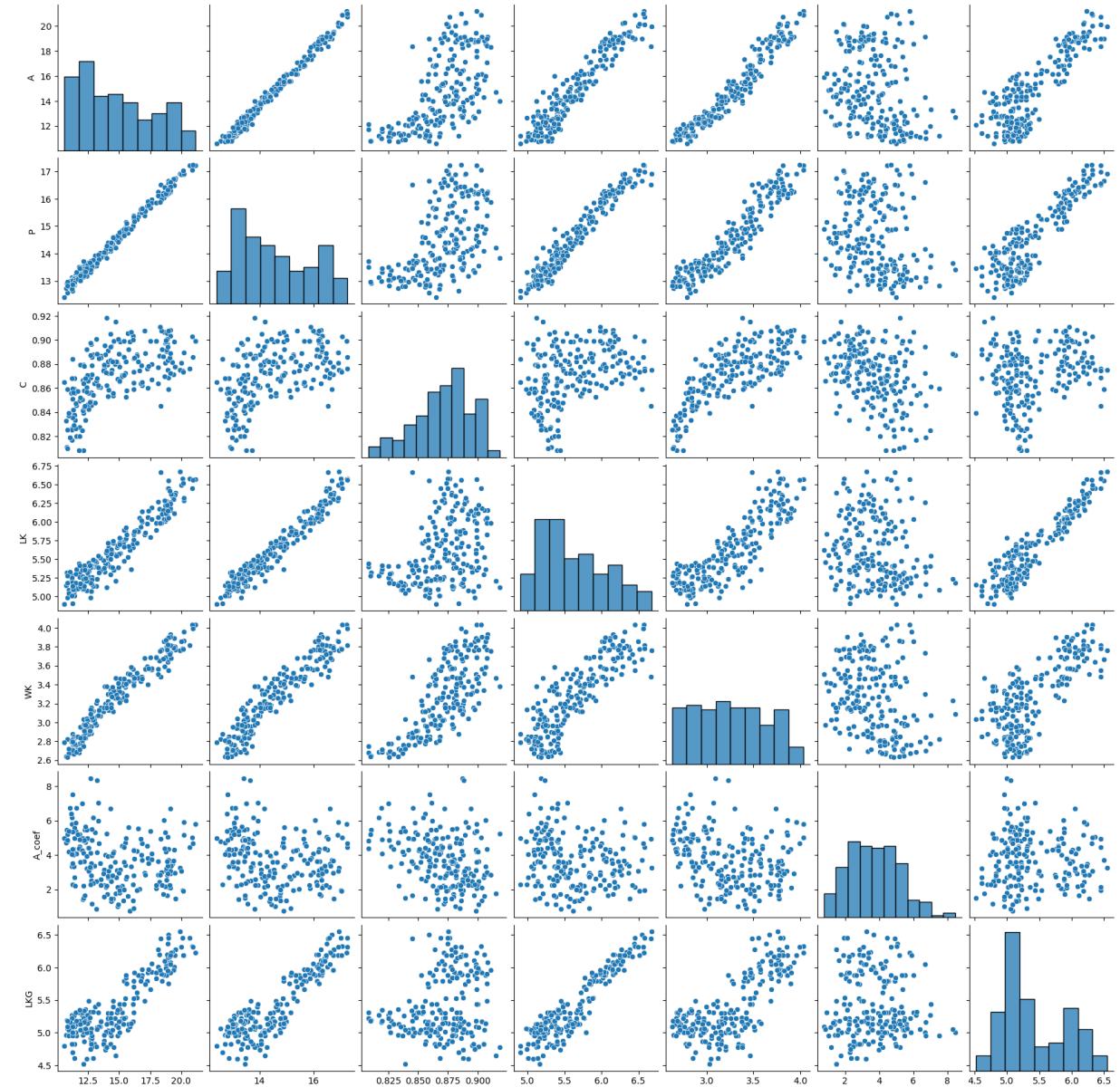
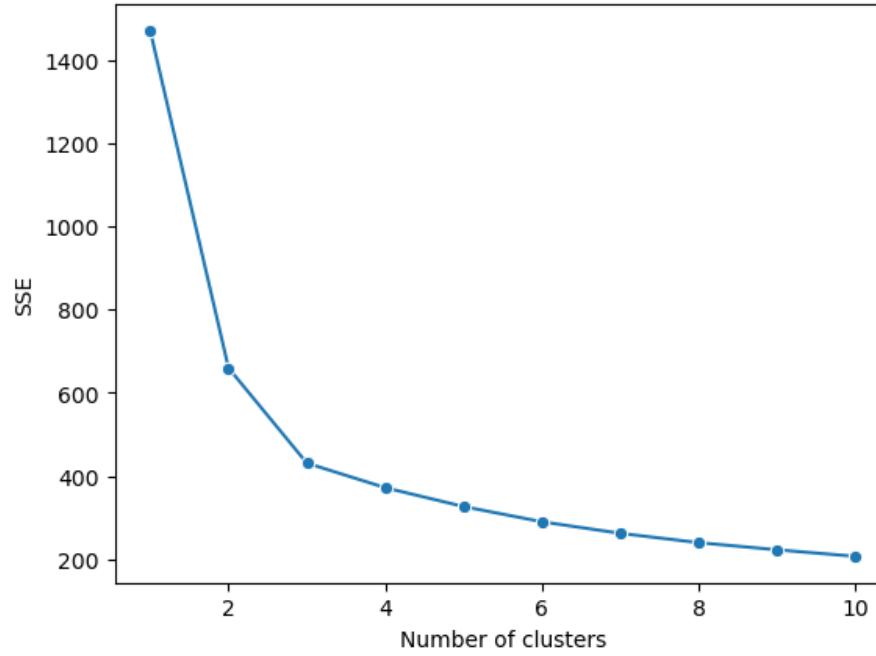
```
# 散布図行列の表示  
sns.pairplot(df_seeds)  
plt.show()
```



14-2 | 3-3. エルボー法の実行

```
# 特徴Xの標準化
X_real = df_seeds.values
scaler = StandardScaler()
X_real_std = scaler.fit_transform(X_real)

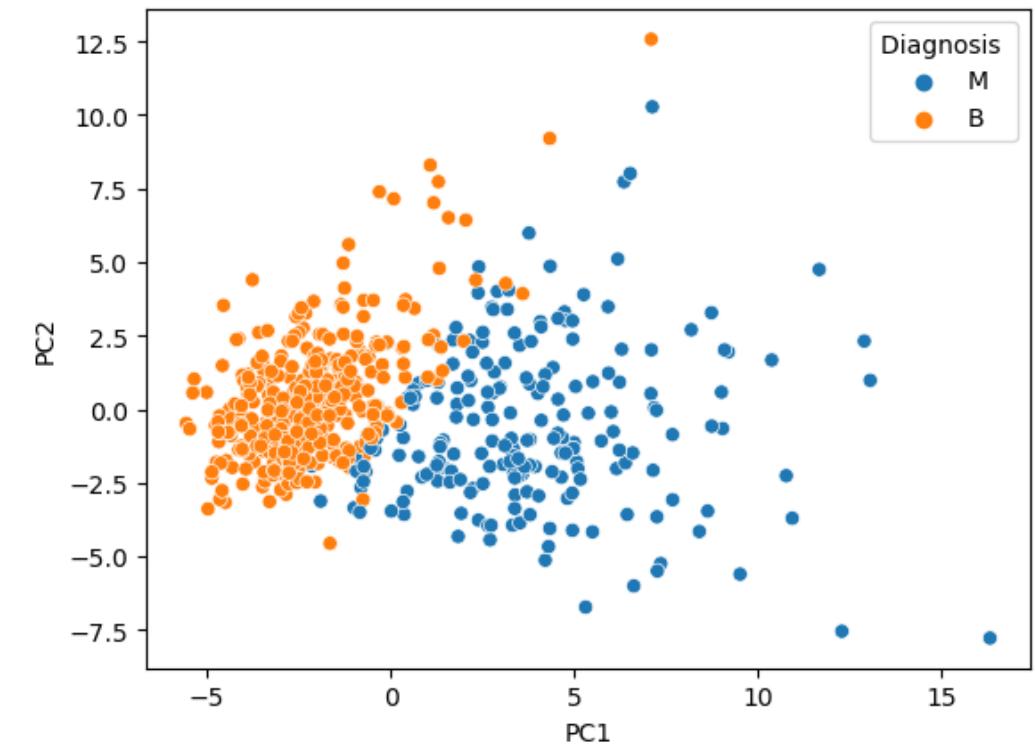
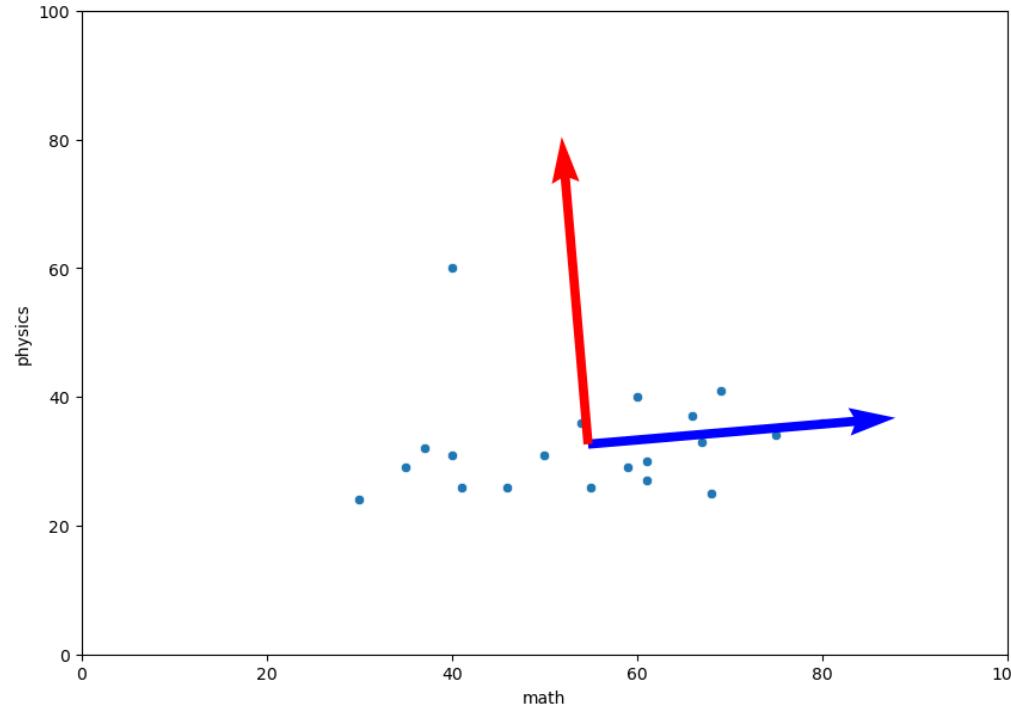
# 関数の実行
plot_elbow(X_real_std, k_start=1, k_end=10)
```



14-3_PCA

■ 内容

- 疑似データと実データに対して、主成分分析を実行
- 寄与率を確認しながら、データの次元削減を実施



■ 手順

1. ライブラリの読み込み
2. 疑似データに対する分析
 1. 疑似データの作成
 2. データの可視化
 3. 主成分分析の実行
 4. 平均値と固有ベクトルの確認
 5. 主成分軸の可視化
 6. 寄与率の確認
 7. 主成分軸への射影変換
3. 乳がんデータセットに対する分析
 1. データの読み込み
 2. 平均値の確認
 3. 主成分分析の実行
 4. 固有ベクトルの確認
 5. 寄与率の確認
 6. 主成分軸への射影変換

14-3 | 1. ライブラリの読み込み

```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# 標準化
from sklearn.preprocessing import StandardScaler
# 主成分分析 (Principal Component Analysis; PCA)
from sklearn.decomposition import PCA
```

14-3 | 2-1. 疑似データの作成

```
# データフレームの作成
df_test = pd.DataFrame({
    "math": [41, 37, 40, 30, 40, 60, 46, 61, 67, 68, 55, 61, 59, 66, 69, 54, 50, 35, 80, 75],
    "physics": [26, 32, 31, 24, 60, 40, 26, 27, 33, 25, 26, 30, 29, 37, 41, 36, 31, 29, 36, 34]
})

# 行番号をIDとして設定
df_test.index.name = "id"

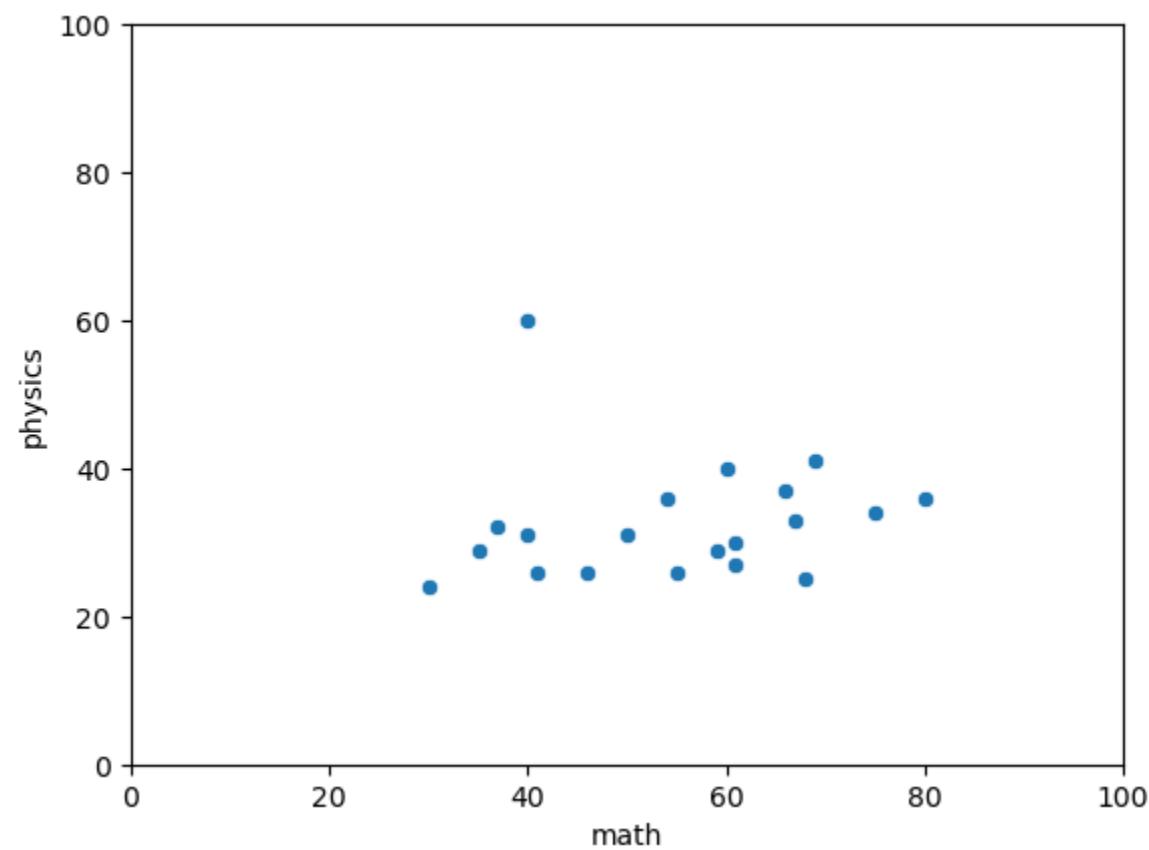
# 点数の合計を表す列を追加
df_test["total"] = df_test["math"] + df_test["physics"]
df_test.head()
```

	math	physics	total
id			
0	41	26	67
1	37	32	69
2	40	31	71
3	30	24	54
4	40	60	100

14-3 | 2-2. データの可視化

```
# 散布図の表示  
sns.scatterplot(x='math', y='physics', data=df_test)
```

```
# 表示範囲の調整  
plt.xlim(0, 100)  
plt.ylim(0, 100)  
plt.show()
```

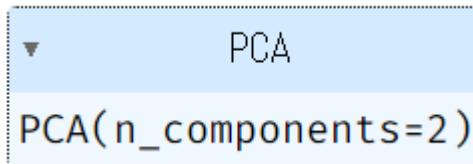


14-3 | 2-3. 主成分分析の実行

```
# 変数を配列に変換
X = df_test[["math", "physics"]].values

# 主成分分析用のオブジェクトを作成
# n_components: 削減後の次元数（主成分の数）
# 入力が2次元であるため、3以上に設定するとエラー
pca = PCA(n_components=2)

# 主成分分析の実行
pca.fit(X)
```



14-3 | 2-4. 平均値と固有ベクトルの確認

```
# 元データの平均値
print("X_mean:", pca.mean_, "\n")

# 分散共分散行列の固有ベクトル
# 元の変数に対する重み（主成分負荷量）
print("eigen vector:\n",
      pca.components_, "\n")

# 固有ベクトルの大きさ
print("norm:",
      np.linalg.norm(pca.components_[0]).round(5),
      np.linalg.norm(pca.components_[1]).round(5), "\n")

# 固有ベクトルの内積
print("dot product:",
      np.dot(pca.components_[0], pca.components_[1]))
```

```
X_mean: [54.7 32.65]

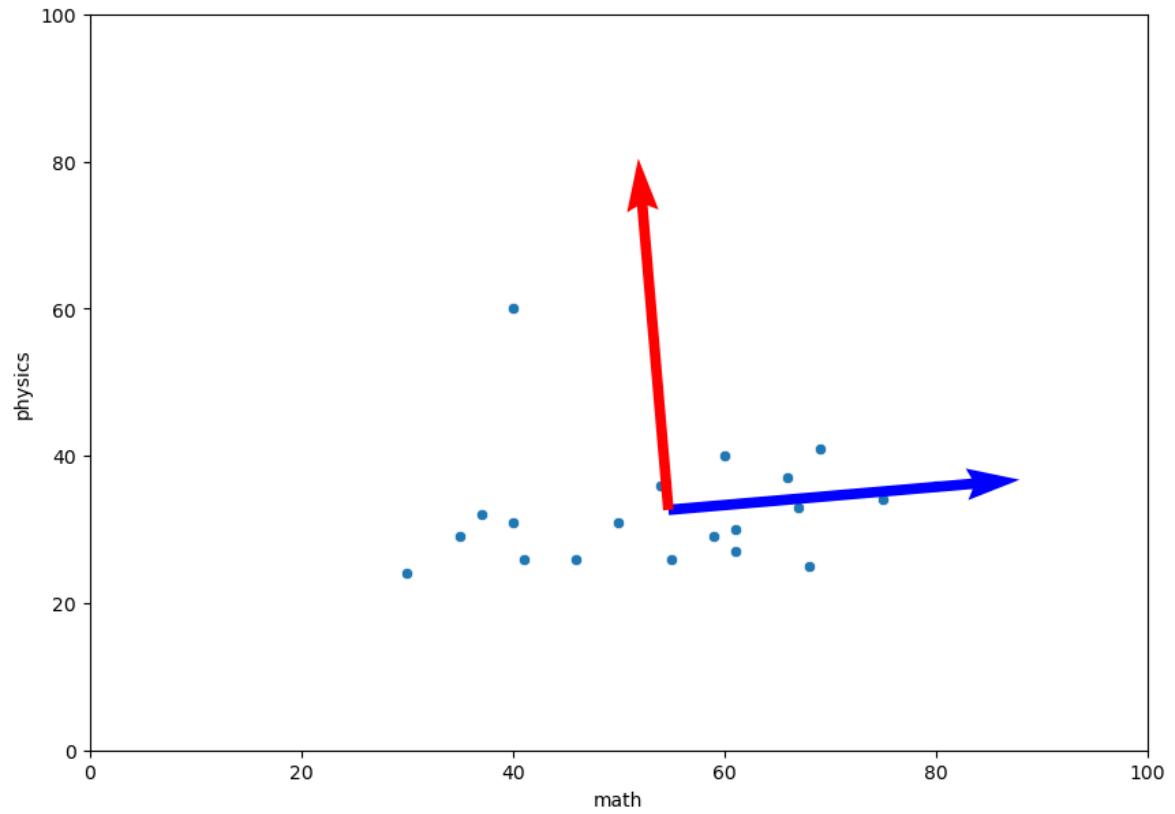
eigen vector:
[[ 0.99634532  0.0854167 ]
 [-0.0854167   0.99634532]]

norm: 1.0 1.0

dot product: 0.0
```

14-3 | 2-5. 主成分軸の可視化 1/2

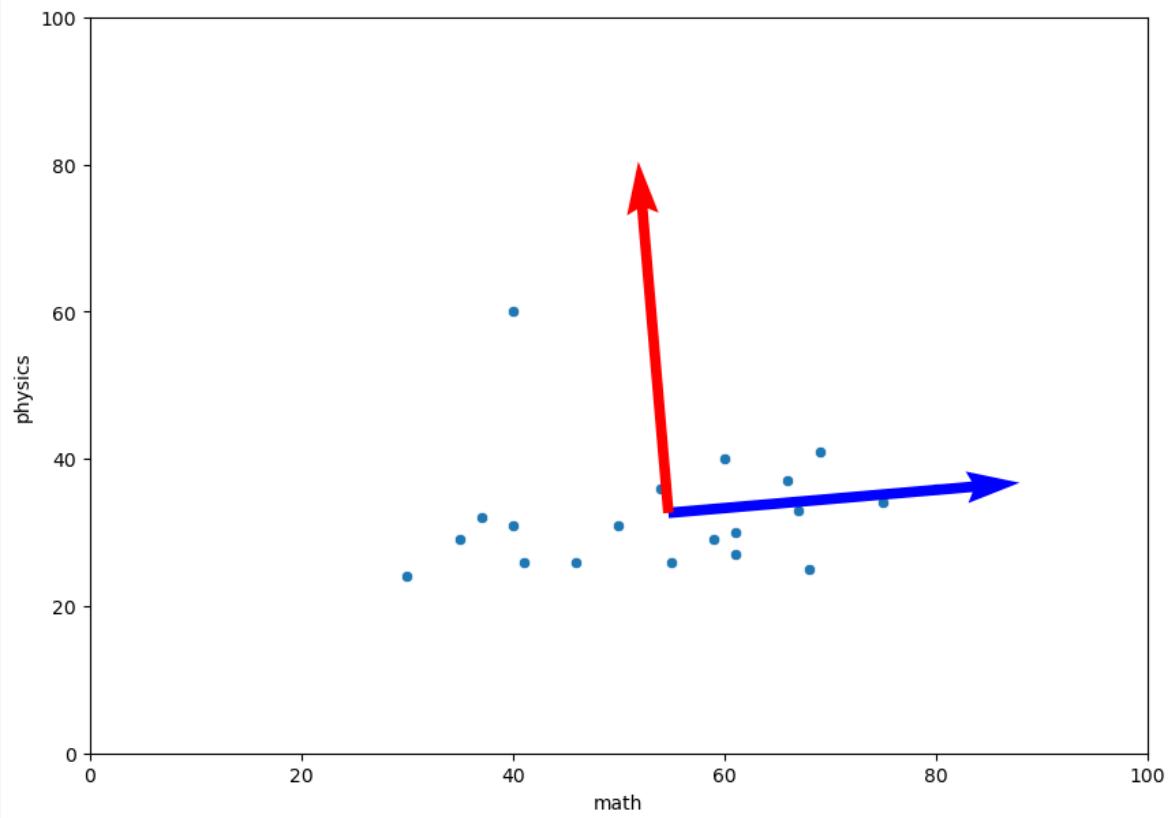
```
# 表示領域の作成
plt.figure(figsize=(10,7))
# 散布図の表示
sns.scatterplot(x=X[:,0], y=X[:,1])
# 表示範囲の調整
plt.xlim([0, 100])
plt.ylim([0, 100])
# 軸の設定
plt.xlabel('math')
plt.ylabel('physics')
```



14-3 | 2-5. 主成分軸の可視化 2/2

```
# 主成分軸を表すベクトルを矢印として表示
# 第一主成分軸
plt.quiver(
    # 始点 (元データの平均)
    pca.mean_[0], pca.mean_[1],
    # ベクトルの方向と長さ (固有ベクトル)
    pca.components_[0,0], pca.components_[0,1],
    # 色、太さ、長さ (表示領域の大きさに対して1/scale倍)
    color='b', width=0.01, scale=3
)
# 第二主成分軸
plt.quiver(
    pca.mean_[0], pca.mean_[1],
    pca.components_[1,0], pca.components_[1,1],
    color='r', width=0.01, scale=3
)

plt.show()
```



```
print("寄与率:", pca.explained_variance_ratio_)
print("累積寄与率:", pca.explained_variance_ratio_.sum())
```

寄与率: [0.75917911 0.24082089]
累積寄与率: 1.0

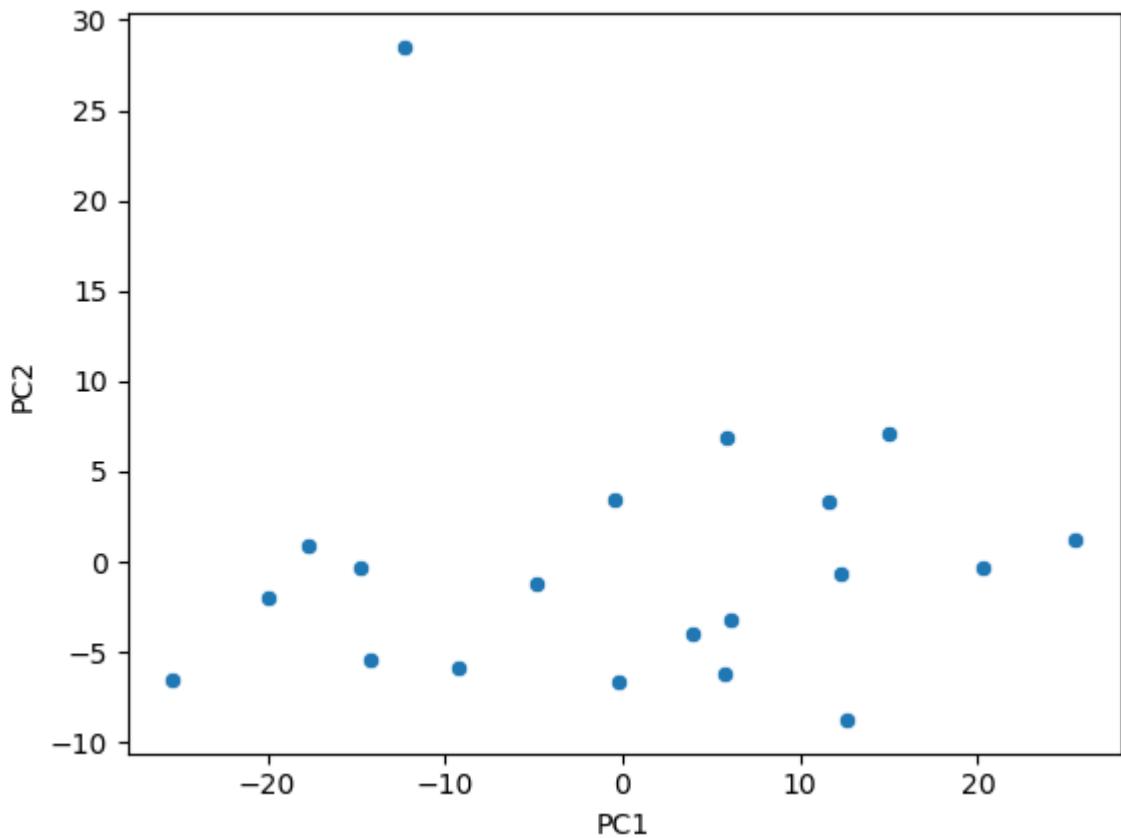
14-3 | 2-7. 主成分軸への射影変換

```
# 元データから平均を引き、固有ベクトルとの行列積をとる
# 固有ベクトルが、元の変数に対する重み（主成分負荷量）となる
Y = np.dot((X - pca.mean_), pca.components_.T)

# transform()でも同様の変換を行える
Y = pca.transform(X)

# 変換後の値（主成分得点）をデータフレームに格納
df_test["PC1"] = Y[:, 0]
df_test["PC2"] = Y[:, 1]

# 散布図の表示
sns.scatterplot(df_test, x="PC1", y="PC2")
plt.show()
```



```
# CSVファイルの読み込み
# 0列目をインデックスとして扱う
df_wdbc = pd.read_csv("../../../1_data/ch14/wdbc.csv", index_col=[0])
# データサイズの確認
print(df_wdbc.shape)

# データの中身を確認
# 2列目以降が説明変数
display(df_wdbc.head())

# 説明変数のみを取り出す
X = df_wdbc.iloc[:, 2:]
```

(569, 32)

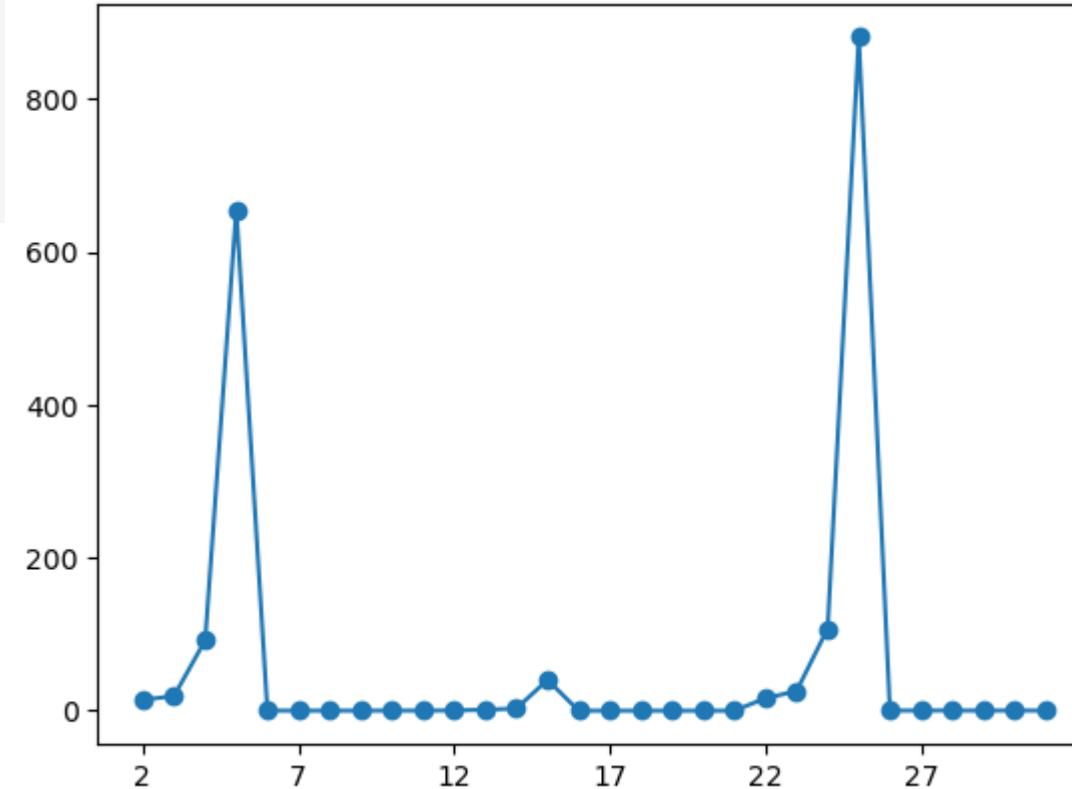
- 目的変数
 - Diagnosis : 診断結果
 - M = malignant (悪性)
 - B = benign (良性)
- 説明変数
 - ある患者の細胞核の画像から算出された30個の実数値
 - 10個の特徴それぞれに関する3つの統計量（標準偏差、標準誤差、絶対値の最大値）
 - radius : 中心から外周上の点に対する距離の平均
 - texture : グレースケール値の標準偏差
 - perimeter : 外周
 - area : 面積
 - smoothness : 半径の局所的なばらつき
 - compactness : 緊密さ $\frac{\text{perimeter}^2}{\text{area}-1}$
 - concavity : 輪郭の凹みの度合い
 - concave points : 輪郭の凹部分の数
 - symmetry : 対称性
 - fractal dimension : フラクタル次元("coastline approximation" - 1)

Breast Cancer Wisconsin (Diagnostic)
乳がんの診断を行う分類問題のデータセット

	ID	Diagnosis	2	3	4	5	6
0	842302	M	17.99	10.38	122.80	1001.0	0.11840
1	842517	M	20.57	17.77	132.90	1326.0	0.08474
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960
3	84348301	M	11.42	20.38	77.58	386.1	0.14250
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030

14-3 | 3-2. 平均値の確認

```
# 各説明変数の平均値をグラフ化  
# マーカー付き折れ線グラフ  
X.mean().plot(style='o-')  
plt.show()
```

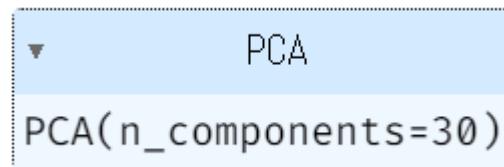


14-3 | 3-3. 主成分分析の実行

```
# 平均値の大きな変数が混じっているため、標準化しておく
stdsc = StandardScaler()
X_std = stdsc.fit_transform(X)

# 主成分分析用のオブジェクトを作成
# n_components: 削減後の次元数（主成分の数）
# ここでは、元データの次元数に合わせる
pca = PCA(n_components=30)

# 主成分分析の実行
pca.fit(X_std)
```



14-3 | 3-4. 固有ベクトルの確認

```
# 固有ベクトル
print("eigen vector:\n", pca.components_.round(3), "\n")

# 固有ベクトルの大きさ
print("norm:")
for i in range(len(pca.components_)):
    # 改行なしで表示
    print(np.linalg.norm(pca.components_[i]).round(5), end=' ', )
print("\n")

# 固有ベクトルの内積
print("dot product:")
for i in range(len(pca.components_)):
    for r in range(len(pca.components_)):
        if i==r: # 同じベクトル同士の場合は省略
            continue
        # 改行なしで表示
        print(np.dot(pca.components_[i], pca.components_[r]).round(5), end=' ', )
```

```
eigen vector:
[[ 0.219  0.104  0.228  0.221  0.143  0.239  0.258  0.261  0.138  0.064
   0.206  0.017  0.211  0.203  0.015  0.17   0.154  0.183  0.042  0.103
   0.228  0.104  0.237  0.225  0.128  0.21   0.229  0.251  0.123  0.132]
 [-0.234 -0.06  -0.215 -0.231  0.186  0.152  0.06  -0.035  0.19   0.367
  -0.106  0.09  -0.089 -0.152  0.204  0.233  0.197  0.13   0.184  0.28
  -0.22  -0.045 -0.2   -0.219  0.172  0.144  0.098 -0.008  0.142  0.275]
```

```
norm:
1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
dot product:
0.0, 0.0, 0.0, 0.0, 0.0, -0.0, -0.0, -0.0, -0.0, 0.0, -0.0, -0.0, 0.0,
```

```
# 何番目の主成分まで利用するか
max_component = 2

# 寄与率、累積寄与率を表示
print("寄与率=", pca.explained_variance_ratio_[:max_component])
print(max_component, "つの主成分の累積寄与率=", pca.explained_variance_ratio_[:max_component].sum())
```

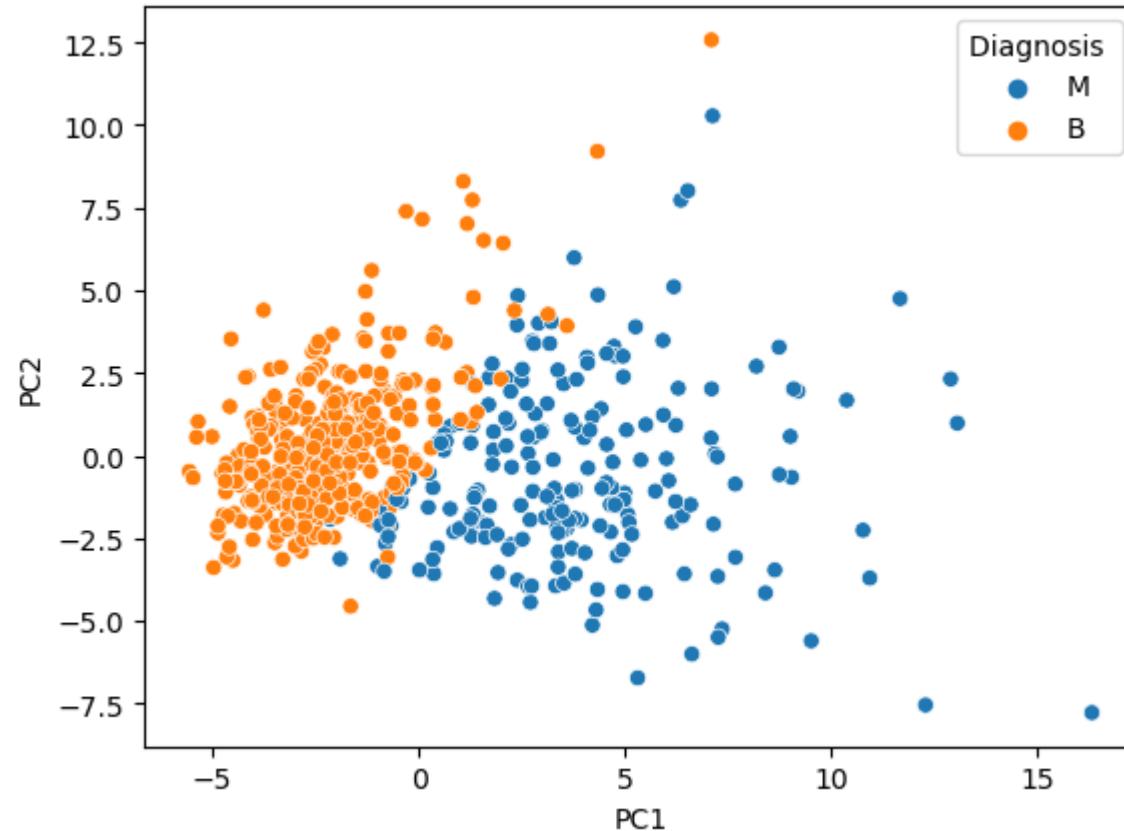
```
寄与率= [0.44272026 0.18971182]
2 つの主成分の累積寄与率= 0.6324320765155942
```

14-3 | 3-6. 主成分軸への射影変換

```
# 値を主成分得点に変換
Y = pca.transform(X_std)

# 変換後の値（主成分得点）をデータフレームに格納
df_wdbc["PC1"] = Y[:,0]
df_wdbc["PC2"] = Y[:,1]

# 散布図の表示（診断結果で色分け）
sns.scatterplot(
    df_wdbc, x="PC1", y="PC2",
    hue="Diagnosis"
)
plt.show()
```



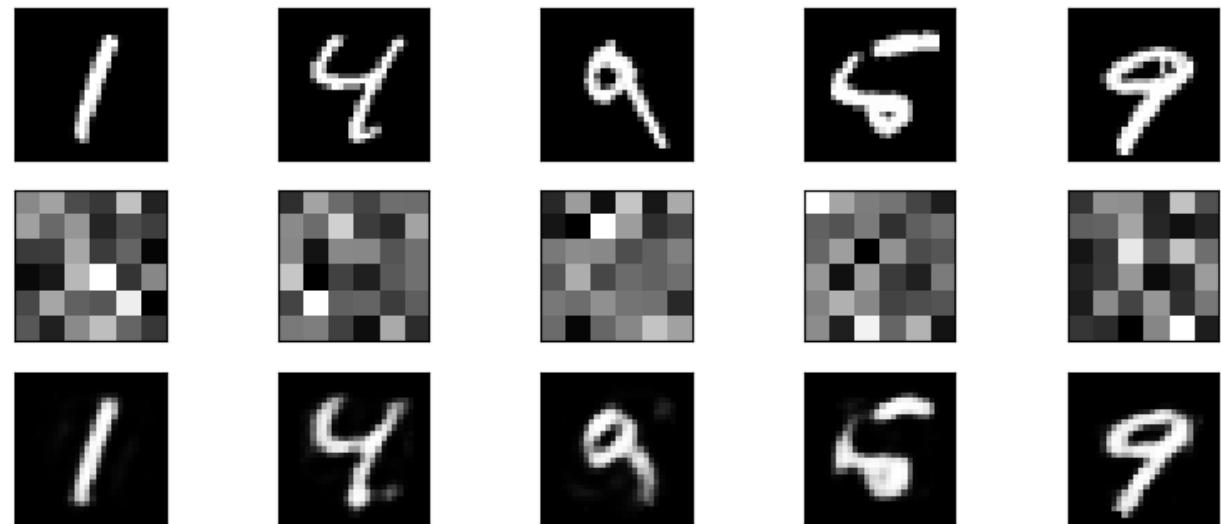
14-4_AutoEncoder

■ 内容

- ・自己符号化器を用いて、手書き数字画像のデータを圧縮・復元
- ・ハイパーパラメータを変更して、復元画像がどのように変化するか確認

■ 手順

1. ライブラリの読み込み
2. データの読み込み
3. モデルの構築
4. モデルの学習
5. 学習結果の確認
 1. 学習曲線の表示
 2. 予測の実行
 3. 予測結果の可視化



14-4 | 1. ライブラリの読み込み

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from PIL import Image

# データセットの読み込み
from tensorflow.keras.datasets import mnist
# ニューラルネットワークの層
from tensorflow.keras.layers import Input, Dense
# 層をまとめてネットワークを構築
from tensorflow.keras.models import Model
```

14-4 | 2. データの読み込み

```
# MNISTデータセットの読み込み
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# 28*28の画像データを、784次元のベクトルに変換
# 画素値を255で割ることで正規化（画素値=0～255）
X_train = X_train.reshape(-1, 784) / 255
X_test = X_test.reshape(-1, 784) / 255
```

14-4 | 3. モデルの構築 1/2

```
# 中間層の次元数
encoding_dim = 36

# 入力層の定義
input_img = Input(shape=(784,))
# 入力層～中間層を定義
# activationで活性化関数を指定
# 2つ目の () で前の層を指定
encoded = Dense(encoding_dim, activation='relu')(input_img)
# 中間層～出力層を定義
decoded = Dense(784, activation='sigmoid')(encoded)

# 入力層～出力層までをつなげて、ネットワークを完成させる
autoencoder = Model(inputs=input_img, outputs=decoded)
# 学習条件の定義
autoencoder.compile(
    # 最適化手法
    optimizer='adam',
    # 損失関数
    # 画素値の回帰問題として扱う
    loss='mean_squared_error'
)
# ネットワーク全体の中身を確認
autoencoder.summary()
```

Model: "model"		
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 784)]	0
dense (Dense)	(None, 36)	28260
dense_1 (Dense)	(None, 784)	29008
Total params: 57,268		
Trainable params: 57,268		
Non-trainable params: 0		

14-4 | 3. モデルの構築 2/2

```
# 計算結果を利用しやすくするために、エンコーダとデコーダを分離
# エンコーダ部分だけのモデル
# autoencoderと層を共有している
encoder = Model(input_img, encoded)
# エンコーダの中身を確認
print("★★ encoder ★★")
print(encoder.summary(), "\n")

# デコーダ部分だけのモデル
# autoencoderと層を共有している
encoded_input = Input(shape=(encoding_dim,))
decoder_layer = autoencoder.layers[-1]
decoder = Model(encoded_input, decoder_layer(encoded_input))
# デコーダの中身を確認
print("★★ decoder ★★")
decoder.summary()
```

★★ encoder ★★		
Model: "model_1"		
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 784)]	0
dense (Dense)	(None, 36)	28260

Total params: 28,260
Trainable params: 28,260
Non-trainable params: 0

None

★★ decoder ★★		
Model: "model_2"		
Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 36)]	0
dense_1 (Dense)	(None, 784)	29008

Total params: 29,008
Trainable params: 29,008
Non-trainable params: 0

14-4 | 4. モデルの学習

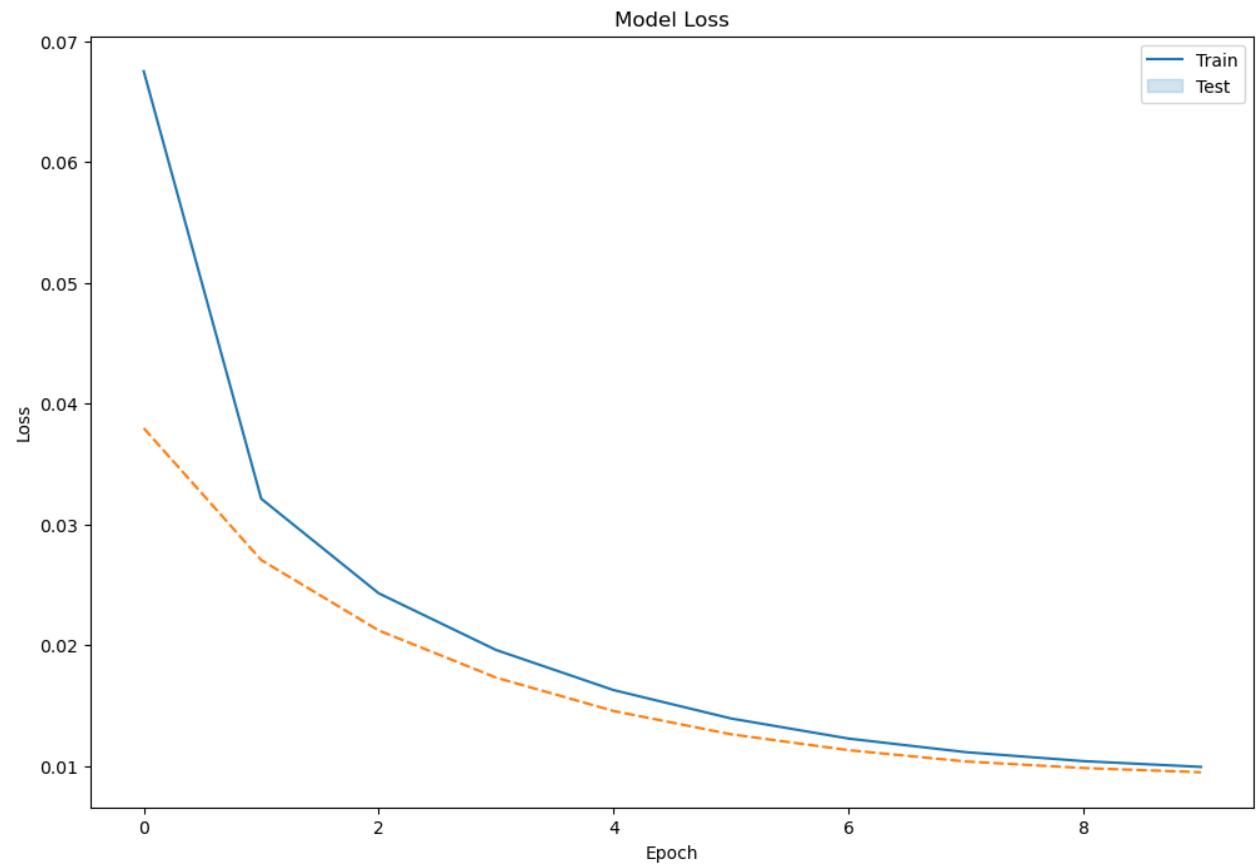
```
fit = autoencoder.fit(  
    # 訓練用データ  
    X_train, X_train,  
    # エポック数 (学習を行う回数)  
    epochs=10,  
    # バッチサイズ (一回の更新で利用するデータの数)  
    batch_size=256,  
    # 学習時にデータをシャッフル  
    shuffle=True,  
    # 検証用データ  
    validation_data=(X_test, X_test)  
)  
  
# 各epochにおける損失をdfに格納  
df = pd.DataFrame(fit.history)
```

```
Epoch 1/10  
235/235 [=====] - 2s 6ms/step - loss: 0.0675 - val_loss: 0.0380  
Epoch 2/10  
235/235 [=====] - 1s 5ms/step - loss: 0.0321 - val_loss: 0.0271  
Epoch 3/10  
235/235 [=====] - 1s 5ms/step - loss: 0.0243 - val_loss: 0.0212  
Epoch 4/10  
235/235 [=====] - 1s 6ms/step - loss: 0.0196 - val_loss: 0.0173  
Epoch 5/10  
235/235 [=====] - 1s 5ms/step - loss: 0.0163 - val_loss: 0.0146  
Epoch 6/10  
235/235 [=====] - 1s 5ms/step - loss: 0.0139 - val_loss: 0.0126  
Epoch 7/10  
235/235 [=====] - 1s 6ms/step - loss: 0.0123 - val_loss: 0.0113  
Epoch 8/10  
235/235 [=====] - 1s 5ms/step - loss: 0.0112 - val_loss: 0.0104  
Epoch 9/10  
235/235 [=====] - 1s 5ms/step - loss: 0.0104 - val_loss: 0.0099  
Epoch 10/10  
235/235 [=====] - 1s 5ms/step - loss: 0.0099 - val_loss: 0.0095
```

14-4 | 5-1. 学習曲線の表示

```
# 表示領域の作成
plt.figure(figsize=(12, 8))
# Loss (損失)をグラフ化
sns.lineplot(data=df[["loss", "val_loss"]])

# グラフタイトルの設定
plt.title('Model Loss')
# 軸ラベルの設定
plt.ylabel('Loss')
plt.xlabel('Epoch')
# 凡例の位置調整
plt.legend(['Train', 'Test'], loc='best')
plt.show()
```



```
# 中間層の値を計算
encoded_imgs = encoder.predict(X_test)

# 出力層の値を計算
decoded_imgs = decoder.predict(encoded_imgs)
# decoded_imgs = autoencoder.predict(X_test) # としても同じ
```

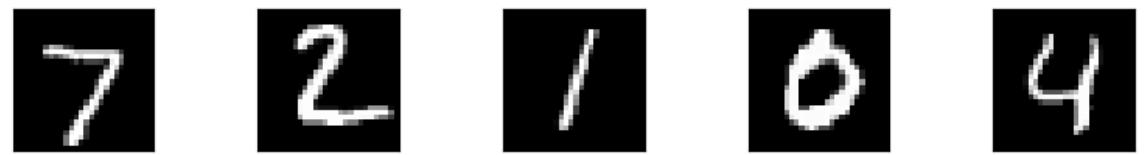
```
313/313 [=====] - 0s 1ms/step
```

```
313/313 [=====] - 0s 1ms/step
```

14-4 | 5-3. 予測結果の可視化 1/3

```
# 表示する画像の枚数
n = 10
# 表示領域の作成
plt.figure(figsize=(20, 4))

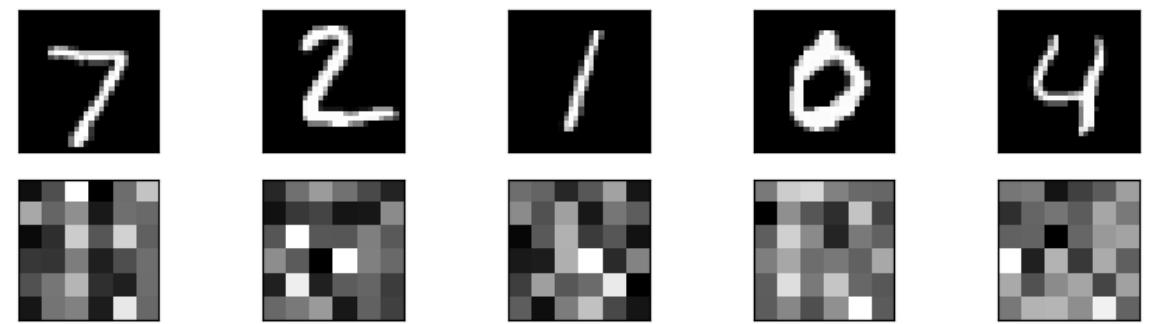
# n枚分繰り返す
for i in range(n):
    # 表示位置の設定
    ax = plt.subplot(3, n, i + 1)
    # 元の画像を表示
    plt.imshow(X_test[i].reshape(28, 28))
    plt.gray()
    # 軸を非表示にする
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```



14-4 | 5-3. 予測結果の可視化 2/3

```
# 表示する画像の枚数
n = 10
# 表示領域の作成
plt.figure(figsize=(20, 4))

# n枚分繰り返す
for i in range(n):
    # 表示位置の設定
    ax = plt.subplot(3, n, i + 1 + n)
    # 中間層の値を画像として表示
    # 画像サイズは、encoding_dimに合わせて変更する
    plt.imshow(encoded_imgs[i].reshape(6,6))
    plt.gray()
    # 軸を非表示にする
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```



14-4 | 5-3. 予測結果の可視化 3/3

```
# 表示する画像の枚数
n = 10
# 表示領域の作成
plt.figure(figsize=(20, 4))

# n枚分繰り返す
for i in range(n):
    # 表示位置の設定
    ax = plt.subplot(3, n, i + 1 + 2*n)
    # 出力層の値を画像として表示
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    # 軸を非表示にする
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

plt.show()
```

