

1. 線形回帰モデル

機械学習

- コンピュータプログラムは、タスクT（アプリケーションにさせてたいこと）を性能指標Pで測定し、その性能が経験E（データ）により改善される場合、タスクT及び性能指標Pに関して経験Eから学習するといわれている（トム・ミッセル 1997）
- 人がプログラムするのは認識の仕方ではなく学習の仕方（数学で記述）

線形とは

ざっくり言えば—比例関係

- $y = Ax + B$ (2次元)
- $z = Ax + By + C$ (3次元)

n次元空間における超平面の方程式

$$y = \sum_{i=0}^{n-1} a_i x_i$$
$$\star a^t = (a_0, a_1, \dots, a_{n-1})$$
$$\star x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

線形回帰モデル

- 回帰問題
 - ある入力（離散あるいは連続値）から出力（連続値）を予測する問題
 - 直線で予測 \Rightarrow 線形回帰
 - 曲線で予測 \Rightarrow 非線形回帰

■ (発展)* バクニックの原理 本来と期待問題より途中で難しい問題を解くべきではない。

- 回帰で扱うデータ
 - 入力 \Rightarrow m次元のベクトル ($m = 1$ の場合はスカラー)
 - 説明変数

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} \in \mathbb{R}^m$$

- 出力 \Rightarrow スカラー値
 - 目的変数

$$y \in \mathbb{R}^1$$

- 線形回帰モデル
 - 回帰問題を解くための機械学習モデルの1つ
 - 教師あり学習
 - パラメータ

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix} \in \mathbb{R}^m$$

* w : ウエイト

- 線形結合

$$\hat{y} = \mathbf{w}^T \mathbf{x} + w_0 = \sum_{j=0}^m w_j x_j + w_0$$

*ゴールは w_j を決めるこ

* 慣例として予測値には $\hat{}$ (ハット) をつける

- 線形結合（入力とパラメータの内積）
 - 入力ベクトルと道のパラメータの各要素を掛け算して足し合せたもの

$$\hat{y} = \mathbf{w}^T x + w_0 = \sum_{j=1}^m w_j x_j + w_0$$

重みの集合 \Rightarrow 最小二乗法により推定する

- 説明変数が 1 次元の場合 ($m = 1$)
 - モデル式

$$y = w_0 + w_1 x_1 + \epsilon$$

- 連立方程式では

$$y_1 = w_0 + w_1 x_1 + \epsilon_1$$

$$y_2 = w_0 + w_1 x_2 + \epsilon_2$$

\dots

$$y_n = w_0 + w_1 x_n + \epsilon_n$$

\Rightarrow ベクトルの内積で書く

- 行列変換

$$\mathbf{y} = X\mathbf{w} + \epsilon$$

式に展開

$$\begin{cases} y_1 = w_0 + w_1 x_1 + \epsilon_1 \\ y_2 = w_0 + w_1 x_2 + \epsilon_2 \\ \dots \\ y_n = w_0 + w_1 x_n + \epsilon_n \end{cases}$$

$$\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \dots \\ 1 & x_n \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \end{pmatrix}$$

$$n \text{ 行 } 1 \text{ 列} = n \text{ 行 } 2 \text{ 列 } 2 \text{ 行 } 1 \text{ 列}$$

- 説明変数が多次元の場合 ($m > 1$)

- 線形重回帰
- モデル式

$$y = w_0 + w_1 x_1 + w_2 x_2 + \epsilon$$

- 連立方程式

$$\begin{cases} y_1 = w_0 + w_1 x_{11} + w_2 x_{12} + \dots + w_n x_{1m} \epsilon_1 \\ y_2 = w_0 + w_1 x_{21} + w_2 x_{22} + \dots + w_n x_{2m} \epsilon_2 \\ \dots \\ y_n = w_0 + w_1 x_{n1} + w_2 x_{n2} + \dots + w_n x_{nm} \epsilon_n \end{cases}$$

- 行列

$$\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix} = \begin{pmatrix} 1 & x_{11} & \dots & x_{1m} \\ 1 & x_{21} & \dots & x_{2m} \\ \dots & & & \\ 1 & x_{n1} & \dots & x_{nm} \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \dots \\ w_m \end{pmatrix}$$

◦ * 行列がわからなくなったら一度連立方程式に直すとよい。

- * (参考書籍) 機械学習のエッセンス(第4版)
(numpyだけを使った機械学習の本)

- データの分割

- なぜ分割するのか \Rightarrow 汎化したいから

■ 学習データと検証データに分ける

- 線形回帰モデルのパラメータは最小二乗法で推定

- 平均 2 乗誤差

- データとモデル出力の二乗誤差の和
- パラメータのみに依存する関数
- データは既知でパラメータのみ未知
- 最小二乗法
- 学習データの平均時乗誤差を最小とするパラメータを探す
- 最小 \Rightarrow 微分

- 式

$$\sum_i (\hat{y}_i - y_i)^2 \Rightarrow \text{二乗誤差の総和}$$

◦ * 二乗損失は一般に外れ値に弱い

\Rightarrow Huber損失、Tukey損失を使うとよい

- MSE(平均二乗誤差)を最小にするようなW

- 式

$$\begin{aligned}
\hat{W} &= \arg_W \min MSE_{train} \\
\frac{\partial}{\partial W} \left\{ \frac{1}{n_{train}} \sum_{i=1}^{n_{train}} (\hat{y}_i^{(train)} - y_i^{(train)})^2 \right\} &= 0 \\
\frac{\partial}{\partial W} \left\{ \frac{1}{n_{train}} \sum_{i=1}^{n_{train}} (X_i^T W - y_i)^2 \right\} &= 0 \\
\frac{\partial}{\partial W} \left\{ \frac{1}{n_{train}} (XW - Y)^T (XW - Y) \right\} &= 0 \\
\frac{1}{n_{train}} \frac{\partial}{\partial W} \left\{ (W^T X^T - Y^T) (XW - y) \right\} &= 0 \\
\frac{1}{n_{train}} \frac{\partial}{\partial W} \left\{ W^T X^T XW - W^T X^T y - y^T XW + y^T y \right\} &= 0 \\
\frac{1}{n_{train}} \frac{\partial}{\partial W} \left\{ W^T X^T XW - 2W^T X^T y + y^T y \right\} &= 0 \\
\frac{1}{n_{train}} \frac{\partial}{\partial W} \left\{ 2X^T XW - 2W^T y \right\} &= 0 \\
\therefore \frac{\partial(W^T X)}{\partial W} = X &\quad \frac{\partial(W^T A X)}{\partial W} = (A + A^T)X = 2AX \\
2X^T XW - 2X^T y &= 0 \\
X^T XW - X^T y &= 0 \\
(X^T X)^{-1}(X^T X)W &= (X^T X)^{-1}X^T y \\
W &= (X^T X)^{-1}X^T y
\end{aligned}$$

- 回帰係数

$$\hat{w} = (X^{(train)^T} X^{(train)})^{-1} X^{(train)^T} y^{(train)}$$

- 予測値

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

▼ ハンズオン（ボストン住宅データセットを線形回帰で）

線形回帰モデル-Boston Housing Data

1. 必要モジュールとデータのインポート

```
#from モジュール名 import クラス名 (もしくは関数名や変数名)

from sklearn.datasets import load_boston
from pandas import DataFrame
import numpy as np

# ボストンデータを"boston"というインスタンスにインポート
boston = load_boston()

#print(boston)

#インポートしたデータを確認(data / target / feature_names / DESCR)
print(boston)

{'data': array([[6.3200e-03, 1.8000e+01, 2.3100e+00, ..., 1.5300e+01, 3.9690e+02,
   4.9800e+00],
 [2.7310e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9690e+02,
  9.1400e+00],
 [2.7290e-02, 0.0000e+00, 7.0700e+00, ..., 1.7800e+01, 3.9283e+02,
  4.0300e+00],
 ...,
 [6.0760e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02,
  5.6400e+00],
 [1.0959e-01, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9345e+02,
  6.4800e+00],
 [4.7410e-02, 0.0000e+00, 1.1930e+01, ..., 2.1000e+01, 3.9690e+02,
  7.8800e+00]], 'target': array([24., 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9, 15.,
  18.9, 21.7, 20.4, 18.2, 19.9, 23.1, 17.5, 20.2, 18.2, 13.6, 19.6,
  15.2, 14.5, 15.6, 13.9, 16.6, 14.8, 18.4, 21., 12.7, 14.5, 13.2,
  13.1, 13.5, 18.9, 20., 21., 24.7, 30.8, 34.9, 26.6, 25.3, 24.7,
  21.2, 19.3, 20., 16.6, 14.4, 19.4, 19.7, 20.5, 25., 23.4, 18.9,
  35.4, 24.7, 31.6, 23.3, 18.6, 18.7, 16., 22.2, 25., 33., 23.5,
  19.4, 22., 17.4, 20.9, 24.2, 21.7, 22.8, 23.4, 24.1, 21.4, 20.,
  20.8, 21.2, 20.3, 28., 23.9, 24.8, 22.9, 23.9, 26.6, 22.5, 22.2,
  23.6, 28.7, 22.6, 22., 22.9, 25., 20.6, 28.4, 21.4, 38.7, 43.8,
  33.2, 27.5, 26.5, 18.6, 19.3, 20.1, 19.5, 19.5, 20.4, 19.8, 19.4,
  21.7, 22.8, 18.8, 18.7, 18.5, 18.3, 21.2, 19.2, 20.4, 19.3, 22.,
  20.3, 20.5, 17.3, 18.8, 21.4, 15.7, 16.2, 18., 14.3, 19.2, 19.6,
  23., 18.4, 15.6, 18.1, 17.4, 17.1, 13.3, 17.8, 14., 14.4, 13.4,
  15.6, 11.8, 13.8, 15.6, 14.6, 17.8, 15.4, 21.5, 19.6, 15.3, 19.4,
  17., 15.6, 13.1, 41.3, 24.3, 23.3, 27., 50., 50., 50., 50., 22.7,
  25., 50., 23.8, 23.8, 22.3, 17.4, 19.1, 23.1, 23.6, 22.6, 29.4,
  23.2, 24.6, 29.9, 37.2, 39.8, 36.2, 37.9, 32.5, 26.4, 29.6, 50.,
  32., 29.8, 34.9, 37., 30.5, 36.4, 31.1, 29.1, 50., 33.3, 30.3,
  34.6, 34.9, 32.9, 24.1, 42.3, 48.5, 50., 22.6, 24.4, 22.5, 24.4,
  20., 21.7, 19.3, 22.4, 28.1, 23.7, 25., 23.3, 28.7, 21.5, 23.,
  26.7, 21.7, 27.5, 30.1, 44.8, 50., 37.6, 31.6, 46.7, 31.5, 24.3,
  31.7, 41.7, 48.3, 29., 24., 25.1, 31.5, 23.7, 23.3, 22., 20.1,
  22.2, 23.7, 17.6, 18.5, 24.3, 20.5, 24.5, 26.2, 24.4, 24.8, 29.6,
  42.8, 21.9, 20.9, 44., 50., 36., 30.1, 33.8, 43.1, 48.8, 31.,
  36.5, 22.8, 30.7, 50., 43.5, 20.7, 21.1, 25.2, 24.4, 35.2, 32.4,
  32., 33.2, 33.1, 29.1, 35.1, 45.4, 35.4, 46., 50., 32.2, 22.,
  20.1, 23.2, 22.3, 24.8, 28.5, 37.3, 27.9, 23.9, 21.7, 28.6, 27.1,
  20.3, 22.5, 29., 24.8, 22., 26.4, 33.1, 36.1, 28.4, 33.4, 28.2,
  22.8, 20.3, 16.1, 22.1, 19.4, 21.6, 23.8, 16.2, 17.8, 19.8, 23.1,
  21., 23.8, 23.1, 20.4, 18.5, 25., 24.6, 23., 22.2, 19.3, 22.6,
  19.8, 17.1, 19.4, 22., 20.7, 21., 19.5, 18.5, 20.6, 19., 18.7,
  32.7, 16.5, 23.9, 31.2, 17.5, 17.2, 23.1, 24.5, 26.6, 22.9, 24.1,
  18.6, 30.1, 18.2, 20.6, 17.8, 21.7, 22.7, 22.6, 25., 19.9, 20.8,
  16.8, 21.9, 27.5, 21.9, 23.1, 50., 50., 50., 50., 13.8,
  13.8, 15., 13.9, 13.3, 13.1, 10.2, 10.4, 10.9, 11.3, 12.3, 8.8,
  7.2, 10.5, 7.4, 10.2, 11.5, 15.1, 23.2, 9.7, 13.8, 12.7, 13.1,
  12.5, 8.5, 5., 6.3, 5.6, 7.2, 12.1, 8.3, 8.5, 5., 11.9,
  27.9, 17.2, 27.5, 15., 17.2, 17.9, 16.3, 7., 7.2, 7.5, 10.4,
  8.8, 8.4, 16.7, 14.2, 20.8, 13.4, 11.7, 8.3, 10.2, 10.9, 11.,
  9.5, 14.5, 14.1, 16.1, 14.3, 11.7, 13.4, 9.6, 8.7, 8.4, 12.8,
  10.5, 17.1, 18.4, 15.4, 10.8, 11.8, 14.9, 12.6, 14.1, 13., 13.4,
  15.2, 16.1, 17.8, 14.9, 14.1, 12.7, 13.5, 14.9, 20., 16.4, 17.7,
  19.5, 20.2, 21.4, 19.9, 19., 19.1, 19.1, 20.1, 19.9, 19.6, 23.2,
  29.8, 13.8, 13.3, 16.7, 12., 14.6, 21.4, 23., 23.7, 25., 21.8,
  20.6, 21.2, 19.1, 20.6, 15.2, 7., 8.1, 13.6, 20.1, 21.8, 24.5,
  23.1, 19.7, 18.3, 21.2, 17.5, 16.8, 22.4, 20.6, 23.9, 22., 11.9]), 'feature names': array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV'])}
```

```
#DESCR変数の中身を確認
print(boston['DESCR'])

... _boston_dataset:
Boston house prices dataset
```

```

**Data Set Characteristics:**

:Number of Instances: 506

:Number of Attributes: 13 numeric/categorical predictive. Median Value (attribute 14) is usually the target.

:Attribute Information (in order):
 - CRIM    per capita crime rate by town
 - ZN      proportion of residential land zoned for lots over 25.000 sq.ft.
 - INDUS   proportion of non-retail business acres per town
 - CHAS    Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
 - NOX     nitric oxides concentration (parts per 10 million)
 - RM      average number of rooms per dwelling
 - AGE     proportion of owner-occupied units built prior to 1940
 - DIS     weighted distances to five Boston employment centres
 - RAD     index of accessibility to radial highways
 - TAX     full-value property-tax rate per $10,000
 - PTRATIO pupil-teacher ratio by town
 - B       1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
 - LSTAT% lower status of the population
 - MEDV    Median value of owner-occupied homes in $1000's

:Missing Attribute Values: None

:Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/

```

This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic prices and the demand for clean air', J. Environ. Economics & Management, vol. 5, 81-102, 1978. Used in Belsley, Kuh & Welsch, 'Regression diagnostics ...', Wiley, 1980. N.B. Various transformations are used in the table on pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression problems.

.. topic:: References

- Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Collinearity', Wiley, 1980. 244-261.
- Quinlan, R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tenth International Conference of Machine Learning,

```

#feature_names変数の中身を確認
#カラム名
print(boston['feature_names'])

['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD', 'TAX', 'PTRATIO',
 'B', 'LSTAT']

#data変数(説明変数)の中身を確認
print(boston['data'])

[[6.3200e-03 1.8000e+01 2.3100e+00 ... 1.5300e+01 3.9690e+02 4.9800e+00]
 [2.7310e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9690e+02 9.1400e+00]
 [2.7290e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9283e+02 4.0300e+00]
 ...
 [6.0760e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+02 5.6400e+00]
 [1.0959e-01 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9345e+02 6.4800e+00]
 [4.7410e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+02 7.8800e+00]]

#target変数(目的変数)の中身を確認
print(boston['target'])

[24. 21.6 34.7 33.4 36.2 28.7 22.9 27.1 16.5 18.9 15. 18.9 21.7 20.4
 18.2 19.9 23.1 17.5 20.2 18.2 13.6 19.6 15.2 14.5 15.6 13.9 16.6 14.8
 18.4 21. 12.7 14.5 13.2 13.1 13.5 18.9 20. 21. 24.7 30.8 34.9 26.6
 25.3 24.7 21.2 19.3 20. 16.6 14.4 19.4 19.7 20.5 25. 23.4 18.9 35.4
 24.7 31.6 23.3 19.6 18.7 16. 22.2 25. 33. 23.5 19.4 22. 17.4 20.9
 24.2 21.7 22.8 23.4 24.1 21.4 20. 20.8 21.2 20.3 28. 23.9 24.8 22.9
 23.9 26.6 22.5 22.2 23.6 28.7 22.6 22. 22.9 25. 20.6 28.4 21.4 38.7
 43.8 33.2 27.5 26.5 18.6 19.3 20.1 19.5 19.5 20.4 19.8 19.4 21.7 22.8
 18.8 18.7 18.5 18.3 21.2 19.2 20.4 19.3 22. 20.3 20.5 17.3 18.8 21.4
 15.7 16.2 18. 14.3 19.2 19.6 23. 18.4 15.6 18.1 17.4 17.1 13.3 17.8
 14. 14.4 13.4 15.6 11.8 13.8 15.6 14.6 17.8 15.4 21.5 19.6 15.3 19.4
 17. 15.6 13.1 41.3 24.3 23.3 27. 50. 50. 50. 22.7 25. 50. 23.8
 23.8 22.3 17.4 19.1 23.1 23.6 22.6 29.4 23.2 24.6 29.9 37.2 39.8 36.2
 37.9 32.5 26.4 29.6 50. 32. 29.8 34.9 37. 30.5 36.4 31.1 29.1 50.
 33.3 30.3 34.6 34.9 32.9 24.1 42.3 48.5 50. 22.6 24.4 22.5 24.4 20.
 21.7 19.3 22.4 28.1 23.7 25. 23.3 28.7 21.5 23. 26.7 21.7 27.5 30.1
 44.8 50. 37.6 31.6 46.7 31.5 24.3 31.7 41.7 48.3 29. 24. 25.1 31.5
 23.7 23.3 22. 20.1 22.2 23.7 17.6 18.5 24.3 20.5 24.5 26.2 24.4 24.8
 29.6 42.8 21.9 20.9 44. 50. 36. 30.1 33.8 43.1 48.8 31. 36.5 22.8
 30.7 50. 43.5 20.7 21.1 25.2 24.4 35.2 32.4 32. 33.2 33.1 29.1 35.1]

```

```

45.4 35.4 46. 50. 32.2 22. 20.1 23.2 22.3 24.8 28.5 37.3 27.9 23.9
21.7 28.6 27.1 20.3 22.5 29. 24.8 22. 26.4 33.1 36.1 28.4 33.4 28.2
22.8 20.3 16.1 22.1 19.4 21.6 23.8 16.2 17.8 19.8 23.1 21. 23.8 23.1
20.4 18.5 25. 24.6 23. 22.2 19.3 22.6 19.8 17.1 19.4 22.2 20.7 21.1
19.5 18.5 20.6 19. 18.7 32.7 16.5 23.9 31.2 17.5 17.2 23.1 24.5 26.6
22.9 24.1 18.6 30.1 18.2 20.6 17.8 21.7 22.7 22.6 25. 19.9 20.8 16.8
21.9 27.5 21.9 23.1 50. 50. 50. 50. 13.8 13.8 15. 13.9 13.3
13.1 10.2 10.4 10.9 11.3 12.3 8.8 7.2 10.5 7.4 10.2 11.5 15.1 23.2
9.7 13.8 12.7 13.1 12.5 8.5 5. 6.3 5.6 7.2 12.1 8.3 8.5 5.
11.9 27.9 17.2 27.5 15. 17.2 17.9 16.3 7. 7.2 7.5 10.4 8.8 8.4
16.7 14.2 20.8 13.4 11.7 8.3 10.2 10.9 11. 9.5 14.5 14.1 16.1 14.3
11.7 13.4 9.6 8.7 8.4 12.8 10.5 17.1 18.4 15.4 10.8 11.8 14.9 12.6
14.1 13. 13.4 15.2 16.1 17.8 14.9 14.1 12.7 13.5 14.9 20. 16.4 17.7
19.5 20.2 21.4 19.9 19. 19.1 19.1 20.1 19.9 19.6 23.2 29.8 13.8 13.3
16.7 12. 14.6 21.4 23. 23.7 25. 21.8 20.6 21.2 19.1 20.6 15.2 7.
8.1 13.6 20.1 21.8 24.5 23.1 19.7 18.3 21.2 17.5 16.8 22.4 20.6 23.9
22. 11.9]

```

2. データフレームの作成

```
# 説明変数をDataFrameへ変換
df = DataFrame(data=boston.data, columns = boston.feature_names)
```

```
# 目的変数をDataFrameへ追加
df['PRICE'] = np.array(boston.target)
```

```
# 最初の5行を表示⇒12行に変更
df.head(12)
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2
5	0.02985	0.0	2.18	0.0	0.458	6.430	58.7	6.0622	3.0	222.0	18.7	394.12	5.21	28.7
6	0.08829	12.5	7.87	0.0	0.524	6.012	66.6	5.5605	5.0	311.0	15.2	395.60	12.43	22.9
7	0.14455	12.5	7.87	0.0	0.524	6.172	96.1	5.9505	5.0	311.0	15.2	396.90	19.15	27.1
8	0.21124	12.5	7.87	0.0	0.524	5.631	100.0	6.0821	5.0	311.0	15.2	386.63	29.93	16.5
9	0.17004	12.5	7.87	0.0	0.524	6.004	85.9	6.5921	5.0	311.0	15.2	386.71	17.10	18.9
10	0.22489	12.5	7.87	0.0	0.524	6.377	94.3	6.3467	5.0	311.0	15.2	392.52	20.45	15.0
11	0.11747	12.5	7.87	0.0	0.524	6.009	82.9	6.2267	5.0	311.0	15.2	396.90	13.27	18.9

線形単回帰分析

```
# カラムを指定してデータを表示
df[['RM']].head()
```

	RM
0	6.575
1	6.421
2	7.185
3	6.998
4	7.147

```
# 説明変数
data = df.loc[:, ['RM']].values
```

```
# dataリストの表示(1-5)
data[0:5]
```

```
array([[6.575,
       [6.421],
       [7.185],
       [6.998],
       [7.147]])
```

```

# 目的変数
target = df.loc[:, 'PRICE'].values

target[0:5]
array([24. , 21.6, 34.7, 33.4, 36.2])

## sklearnモジュールからLinearRegressionをインポート
from sklearn.linear_model import LinearRegression

# オブジェクト生成
model = LinearRegression()
#model.get_params()
#model = LinearRegression(fit_intercept = True, normalize = False, copy_X = True, n_jobs = 1)

# fit関数でパラメータ推定
model.fit(data, target)

LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

#予測
model.predict([[6]])
array([19.94203311])

```

重回帰分析(2変数)

```

#カラムを指定してデータを表示
df[['CRIM', 'RM']].head()

      CRIM      RM
0  0.00632  6.575
1  0.02731  6.421
2  0.02729  7.185
3  0.03237  6.998
4  0.06905  7.147

# 説明変数
data2 = df.loc[:, ['CRIM', 'RM']].values
# 目的変数
target2 = df.loc[:, 'PRICE'].values

# オブジェクト生成
model2 = LinearRegression()

# fit関数でパラメータ推定
model2.fit(data2, target2)

LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

model2.predict([[0.2, 6]])
array([21.04870738])

```

回帰係数と切片の値を確認

```

# 単回帰の回帰係数と切片を出力
print(' 推定された回帰係数: %.3f, 推定された切片 : %.3f' % (model.coef_, model.intercept_))

推定された回帰係数: 9.102, 推定された切片 : -34.671

# 重回帰の回帰係数と切片を出力
print(model.coef_)
print(model.intercept_)

[9.10210898]
-34.67062077643857

```

モデルの検証

1. 決定係数

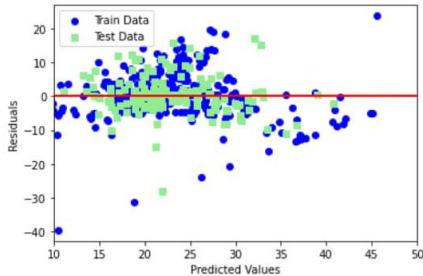
▼ 決定係数

```
print('単回帰決定係数: %.3f, 重回帰決定係数 : %.3f' % (model.score(data,target), model2.score(data2,target2)))
```

```
# train_test_splitをインポート
from sklearn.model_selection import train_test_split
```

```
# 70%を学習用、30%を検証用データにするよう分割
X_train, X_test, y_train, y_test = train_test_split(data, target,
test_size = 0.3, random_state = 666)
# 学習用データでパラメータ推定
model.fit(X_train, y_train)
# 作成したモデルから予測（学習用、検証用モデル使用）
y_train_pred = model.predict(X_train)
y_test_pred = model.predict(X_test)

# matplotlibをインポート
import matplotlib.pyplot as plt
# Jupyterを利用していたら、以下のまじないを書くとnotebook上に図が表示
%matplotlib inline
# 学習用、検証用それぞれで残差をプロット
plt.scatter(y_train_pred, y_train_pred - y_train, c = 'blue', marker = 'o', label = 'Train Data')
plt.scatter(y_test_pred, y_test_pred - y_test, c = 'lightgreen', marker = 's', label = 'Test Data')
plt.xlabel('Predicted Values')
plt.ylabel('Residuals')
# 凡例を左上に表示
plt.legend(loc = 'upper left')
# y = 0に直線を引く
plt.hlines(y = 0, xmin = -10, xmax = 50, lw = 2, color = 'red')
plt.xlim([-10, 50])
plt.show()
```



```
# 平均二乗誤差を評価するためのメソッドを呼び出し
from sklearn.metrics import mean_squared_error
# 学習用、検証用データに関して平均二乗誤差を出力
print('MSE Train : %.3f, Test : %.3f' % (mean_squared_error(y_train, y_train_pred), mean_squared_error(y_test, y_test_pred)))
# 学習用、検証用データに関してR^2を出力
print('R^2 Train : %.3f, Test : %.3f' % (model.score(X_train, y_train), model.score(X_test, y_test)))

MSE Train : 44.983, Test : 40.412
R^2 Train : 0.500, Test : 0.434
```

▼ 実装演習（線形回帰）

▼ 回帰分析

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

▼ 単回帰

▼ 訓練データ生成

```
n_sample = 100
var = .2

def linear_func(x):
    return 2 * x + 5

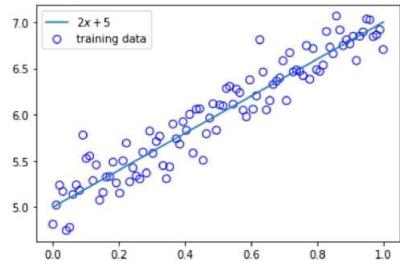
def add_noise(y_true, var):
    return y_true + np.random.normal(scale=var, size=y_true.shape)

def plt_result(xs_train, ys_train, ys_train):
    plt.scatter(xs_train, ys_train, facecolor="none", edgecolor="b", s=50, label="training data")
    plt.plot(xs_train, ys_train, label="2 x + 5")
    plt.legend()

#データの作成
xs = np.linspace(0, 1, n_sample)
ys_true = linear_func(xs)
ys = add_noise(ys_true, var)

print("xs: {}".format(xs.shape))
print("ys_true: {}".format(ys_true.shape))
print("ys: {}".format(ys.shape))

#結果の描画
plt_result(xs, ys_true, ys)
```



▼ 学習

1次関数 $y(x) = ax + b$ における、 a と b を求める。

訓練データ $X = [x_1, x_2, \dots, x_n]^T$, $\mathbf{y} = [y_1, y_2, \dots, y_n]^T$ に対して、最小化する目的関数は $L = \sum_{i=1}^n (y_i - (ax_i + b))^2$ と書け、

$$\frac{\partial L}{\partial a} = -2 \sum_{i=1}^n (y_i - (ax_i + b)) x_i = 0$$

$$\frac{\partial L}{\partial b} = -2 \sum_{i=1}^n (y_i - (ax_i + b)) = 0 \cdots (1)$$

より、目的関数を最小にする a, b は以下のように求まる。

```


$$\begin{pmatrix} \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & n \end{pmatrix} \begin{pmatrix} \hat{a} \\ \hat{b} \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n x_i y_i \\ \sum_{i=1}^n y_i \end{pmatrix}$$


#numpy実装の回帰
def train(xs, ys):
    cov = np.cov(xs, ys, ddof=0)
    a = cov[0, 1] / cov[0, 0]
    b = np.mean(ys) - a * np.mean(xs)
    return cov, a, b

cov, a, b = train(xs, ys)
print("cov: {}".format(cov))
print("coef: {}".format(a))
print("intercept: {}".format(b))

cov: [[0.08501684 0.16939243]
       [0.16939243 0.37276216]]
coef: 1.992457443995857
intercept: 5.010395745959691

#skl実装の回帰
from sklearn.linear_model import LinearRegression
model = LinearRegression()
reg = model.fit(xs.reshape(-1, 1), ys.reshape(-1, 1))

print("coef_: {}".format(reg.coef_))
print("intercept_: {}".format(reg.intercept_))

coef_: [[1.99245744]]
intercept_: [5.01039575]

```

▼ 予測

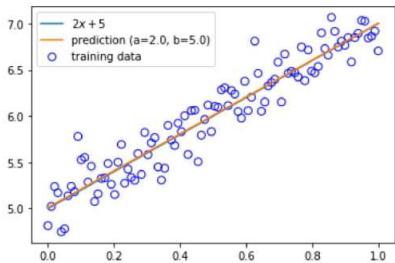
入力に対する値を $y(x) = ax + b$ で予測する

```

xs_new = np.linspace(0, 1, n_sample)
ys_pred = a * xs_new + b

plt.scatter(xs, ys, facecolor="none", edgecolor="b", s=50, label="training data")
plt.plot(xs_new, ys_true, label="$2x + 5$")
plt.plot(xs_new, ys_pred, label="prediction (a={:.2}, b={:.2})".format(a, b))
plt.legend()
plt.show()

```



▼ 多項式回帰

訓練データ生成

```

n_sample = 10
var = .25

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

def add_noise(y_true, var):
    return y_true + np.random.normal(scale=var, size=y_true.shape)

def plt_result(xs, ys_true, ys):
    plt.scatter(xs, ys, facecolor="none", edgecolor="b", s=50, label="training data")
    plt.plot(xs, ys_true, label="$\sin(2\pi x)$")
    plt.plot(xs, ys, label="fit result")
    plt.legend()

```

```

#データの作成
xs = np.linspace(0, 1, n_sample)
ys_true = sin_func(xs)
ys = add_noise(ys_true, var)

print("xs: {}".format(xs.shape))
print("ys_true: {}".format(ys_true.shape))
print("ys: {}".format(ys.shape))

#結果の描画
plt_result(xs, ys_true, ys)

xs: (10,)
ys_true: (10,)
ys: (10,)


$$\sin(2\pi x)$$

○ training data

```

▼ 学習

モデルとして以下を用いる。

$$y(x) = \sum_{i=0}^d w_i x^i = \mathbf{w}^T \phi(\mathbf{x})$$

ただし、 $\mathbf{w} = [w_0, w_1, \dots, w_d]^T$, $\phi(\mathbf{x}) = [1, x, x^2, \dots, x^d]^T$ である。

訓練データ X, \mathbf{y} に対しては $\mathbf{y} = \Phi \mathbf{w}$ と書ける。

ただし、 $\Phi = [\phi(x_1), \phi(x_2), \dots, \phi(x_n)]^T$ である。

よって、最小化する目的関数は $L = \|\mathbf{y} - \Phi \mathbf{w}\|^2$ と書け、

$\frac{\partial L}{\partial \mathbf{w}} = -2\Phi^T (\mathbf{y} - \Phi \mathbf{w}) = 0$ より、求める回帰係数 \mathbf{w} は以下のように書ける。

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

```

def polynomial_features(xs, degree=3):
    """多項式特徴ベクトルに変換
    X = [[1, x1, x1^2, x1^3],
         [1, x2, x2^2, x2^3],
         ...
         [1, xn, xn^2, xn^3]]"""
    X = np.ones((len(xs), degree+1))
    X_t = X.T #(100, 4)
    for i in range(1, degree+1):
        X_t[i] = X_t[i-1] * xs
    return X_t.T

Phi = polynomial_features(xs)
Phi_inv = np.dot(np.linalg.inv(np.dot(Phi.T, Phi)), Phi.T)
w = np.dot(Phi_inv, ys)

```

▼ 予測

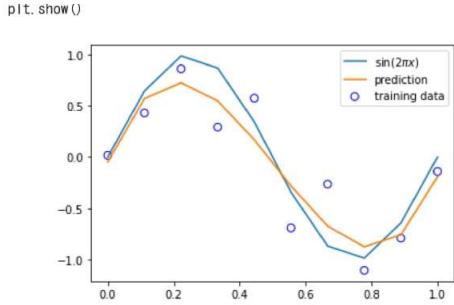
入力を多項式特徴ベクトル $\phi(\mathbf{x})$ に変換し、 $y = \hat{\mathbf{w}} \phi(\mathbf{x})$ ($\mathbf{y}(x) = \Phi \hat{\mathbf{w}}$) で予測する。

```

Phi_test = polynomial_features(xs)
ys_pred = np.dot(Phi_test, w)

plt.scatter(xs, ys, facecolor="none", edgecolor="b", s=50, label="training data")
plt.plot(xs, ys_true, label="$\sin(2\pi x)$")
plt.plot(xs, ys_pred, label="prediction")
# for i in range(0, 4):
#     plt.plot(xs, Phi[:, i], label="basis")
plt.legend()

```



▼ 重回帰分析

▼ 訓練データ生成 (3次元入力)

```
np.random.random((10, 3))

array([[0.95293486, 0.43671772, 0.68539984],
       [0.75742888, 0.77198894, 0.94513097],
       [0.85284069, 0.46821687, 0.2518513 ],
       [0.5197923 , 0.14078174, 0.43133792],
       [0.30846117, 0.89868625, 0.13482976],
       [0.04893669, 0.84315826, 0.23933467],
       [0.85567888, 0.60629944, 0.46001933],
       [0.44752095, 0.21283905, 0.638274 ],
       [0.09223676, 0.6911691 , 0.2216999 ],
       [0.83896078, 0.97844714, 0.56648099]])
```

```
n_sample = 100
var = .2

def mul_linear_func(x):
    ww = [1., 0.5, 2., 1.]
    return ww[0] + ww[1] * x[:, 0] + ww[2] * x[:, 1] + ww[3] * x[:, 2]

def add_noise(y_true, var):
    return y_true + np.random.normal(scale=var, size=y_true.shape)

def plt_result(xs_train, ys_true, ys_train):
    plt.scatter(xs_train, ys_train, facecolor="none", edgecolor="b", s=50, label="training data")
    plt.plot(xs_train, ys_true, label="$2 x + 5$")
    plt.legend()

x_dim = 3

X = np.random.random((n_sample, x_dim))
ys_true = mul_linear_func(X)
ys = add_noise(ys_true, var)
```

▼ 学習

モデルとして以下を用いる。

$$y(x) = \sum_{i=0}^d w_i x_i = \mathbf{w}^T \mathbf{x}$$

ただし、陽には書かないが、 \mathbf{x} には定数項のための1という要素があることを仮定する。

訓練データ X, \mathbf{y} に対しては $\mathbf{y} = X\mathbf{w}$ と書ける。

よって、最小化する目的関数は $L = \|\mathbf{y} - X\mathbf{w}\|^2$ と書け、

$$\frac{\partial L}{\partial \mathbf{w}} = -2X^T(\mathbf{y} - X\mathbf{w}) = 0$$

$$\hat{\mathbf{w}} = (X^T X)^{-1} X^T \mathbf{y}$$

```
def add_one(x):
    return np.concatenate([np.ones(len(x))[:, None], x], axis=1)

X_train = add_one(X)
# pinv = np.dot(np.linalg.inv(np.dot(X_train.T, X_train)), X_train.T)
# w = np.dot(pinv, y_train)
```

▼ 予測

入力に対する値を $y(x) = \hat{w}^T x$ ($y = X\hat{w}$)で予測する

▼ パラメータ推定結果

```
ww = [1., 0.5, 2., 1.]
for i in range(len(w)):
    print("w[0]_true: [1:>5.2]   w[0]_estimated: [2:>5.2]".format(i, ww[i], w[i]))
w0_true: 1.0   w0_estimated: -0.047
w1_true: 0.5   w1_estimated:  8.0
w2_true: 2.0   w2_estimated: -2.4e+01
w3_true: 1.0   w3_estimated: 1.6e+01
```

2. 非線形回帰モデル

- 単回帰/重回帰

$$\begin{aligned} y &= w_0 + w_1 x \\ y &= w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n \\ &\Rightarrow \text{非線型な回帰を考えたい} \\ \text{例)} \quad y &= w_0 + w_1 x + w_2 x^2 + w_3 x^3 \\ y &= w_0 + w_1 \sin x + w_2 \cos x + w_3 \log x \\ &\Rightarrow x \text{の代わりに } \varphi(x) \text{ を用いる (}x\text{ の関数)} \end{aligned}$$

* x が $\varphi(x)$ に変わるだけ (パラメータは線型のまま)

- 基底展開法

 - 式

$$y_i = c_0 + \sum_{j=1}^m w_j \varphi_j(x_i) + \varepsilon_i$$

- よく使われる基底関数

 - 多項式関数
 - ガウス関数
 - ...

- 未学習と過学習

 - 学習データに対して、十分市イサナ誤差が得られないモデル \Rightarrow 未学習
 - 小さな誤差は得られたが、テスト集合誤差との差が大きいモデル \Rightarrow 過学習
 \Rightarrow いかにして過学習を減らすか
 - 学習データの数を増やす
 - 不要な変数の削除 (表現力の抑止) \Rightarrow 実際には難しい
 - 正則化 (表現力の抑止) \Rightarrow こちらならできる

- 正則化法

 - モデルの複雑さに伴って、その値が大きくなる正則化項を課した関数を最小化

$$\begin{aligned} \text{予測: } \hat{y} &= X * (X^T X)^{-1} X^T y \\ X &= \begin{pmatrix} 1 & 2 & 4 \\ 1 & 3 & 5.9 \\ 1 & 4 & 8.1 \end{pmatrix} \Rightarrow (X^T X)^{-1} \text{の要素はメチャメチャ大きくなる} \\ \Rightarrow E(W) &= \underbrace{J(W)}_{MSE(\text{これが小さくなるようにする})} + \overbrace{\lambda W^T W}^{\text{罰則項}} \\ \text{解きたいのは、 } \min MSE &\text{s.t. } R(W) \leq r \\ \Rightarrow (\text{最適化}): \text{KKT条件より} & \\ \min MSE &+ \underbrace{\lambda R(W)}_{\text{おまけをつけると不等式条件をなくすことができる}} \end{aligned}$$

- 正則化項の役割（どう入れるか）
 - なし ⇒ 最小二乗推定量
 - L2ノルムを利用 ⇒ Ridge推定量
 - L1ノルムを利用 ⇒ Lasso推定量
- ホールドアウト法
 - 有限個のデータを学習用、テスト用に分けて、精度や誤り率を推定する。
 - 外れ値が学習データに入るとまずい。
- クロスバリデーション（交差検証）
 - 例えばデータを6つに分けて（それぞれa,b,c,d,e,fとする）
 - 1回目 訓練用 : a,b,c,d,e 検証用 : f
 - 2回目 訓練用 : a,b,c,d,f 検証用 : e
 - ...
 - 6回目 訓練用 : b,c,d,e,f 検証用 : a

として6回の評価で検証する。
 - ホールドアウト法の欠点を補う。

▼ 実装演習（非線形回帰分析）

Google ドライブのマウント

▼ 新しいセクション

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline

#seaborn設定
sns.set()
#背景変更
sns.set_style("darkgrid", {'grid.linestyle': '-'})
#大きさ(スケール変更)
sns.set_context("paper")

n=100

def true_func(x):
    z = 1-48*x+218*x**2-315*x**3+145*x**4
    return z

def linear_func(x):
    z = x
    return z

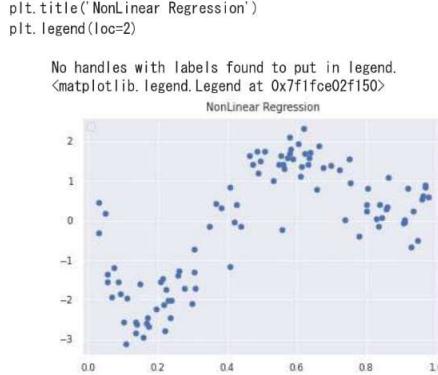
# 真の関数からノイズを伴うデータを生成

# 真の関数からデータ生成
data = np.random.rand(n).astype(np.float32)
data = np.sort(data)
target = true_func(data)

# ノイズを加える
noise = 0.5 * np.random.randn(n)
target = target + noise

# ノイズ付きデータを描画

plt.scatter(data, target)
```



```
from sklearn.linear_model import LinearRegression

clf = LinearRegression()
data = data.reshape(-1, 1)
target = target.reshape(-1, 1)
clf.fit(data, target)

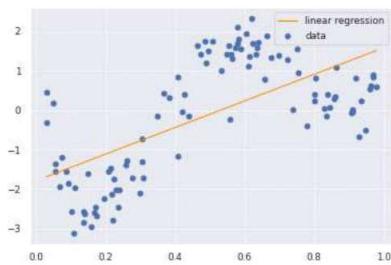
p_lin = clf.predict(data)
```

```

plt.scatter(data, target, label='data')
plt.plot(data, p_lin, color='darkorange', marker='', linestyle='-', linewidth=1, markersize=6, label='linear regression')
plt.legend()
print(clf.score(data, target))

```

0.41902085771948194



- 線形回帰では十分ではない

```

from sklearn.kernel_ridge import KernelRidge
clf = KernelRidge(alpha=0.0002, kernel='rbf')
clf.fit(data, target)

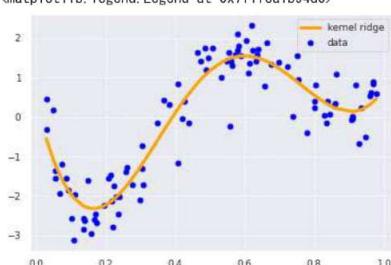
p_kridge = clf.predict(data)

plt.scatter(data, target, color='blue', label='data')

plt.plot(data, p_kridge, color='orange', linestyle='-', linewidth=3, markersize=6, label='kernel ridge')
plt.legend()
# plt.plot(data, p, color='orange', marker='o', linestyle='-', linewidth=1, markersize=6)

<matplotlib.legend.Legend at 0x7f1fc1b94d0>

```



#Ridge

```

from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import Ridge

kx = rbf_kernel(X=data, Y=data, gamma=50)
#KX = rbf_kernel(X, x)

#clf = LinearRegression()
clf = Ridge(alpha=30)
clf.fit(kx, target)

p_ridge = clf.predict(kx)

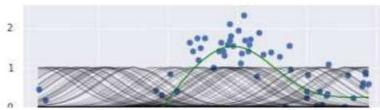
plt.scatter(data, target, label='data')
for i in range(len(kx)):
    plt.plot(data, kx[i], color='black', linestyle='-', linewidth=1, markersize=3, label='rbf', alpha=0.2)

plt.plot(data, p, color='green', marker='o', linestyle='-', linewidth=0.1, markersize=3)
plt.plot(data, p_ridge, color='green', linestyle='-', linewidth=1, markersize=3, label='ridge regression')
# plt.legend()

print(clf.score(kx, target))

```

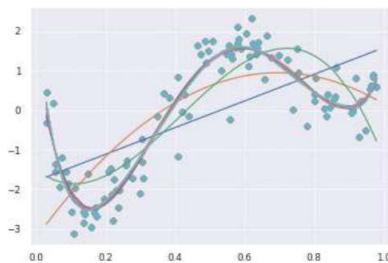
```
0.865333482171628
```



```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline

#PolynomialFeatures(degree=1)

deg = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for d in deg:
    regr = Pipeline([
        ('poly', PolynomialFeatures(degree=d)),
        ('linear', LinearRegression())
    ])
    regr.fit(data, target)
    # make predictions
    p_poly = regr.predict(data)
    # plot regression result
    plt.scatter(data, target, label='data')
    plt.plot(data, p_poly, label='polynomial of degree %d' % (d))
```



```
#Lasso
```

```
from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import Lasso

kx = rbf_kernel(X=data, Y=data, gamma=5)
#KX = rbf_kernel(X, x)

#lasso_clf = LinearRegression()
lasso_clf = Lasso(alpha=10000, max_iter=1000)
lasso_clf.fit(kx, target)

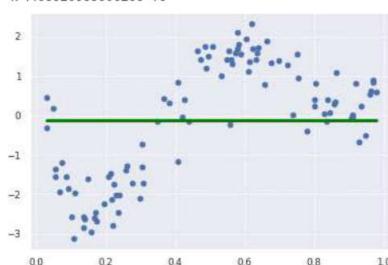
p_lasso = lasso_clf.predict(kx)

plt.scatter(data, target)

plt.plot(data, p, color='green', marker='o', linestyle='-', linewidth=0.1, markersize=3)
plt.plot(data, p_lasso, color='green', linestyle='-', linewidth=3, markersize=3)

print(lasso_clf.score(kx, target))
```

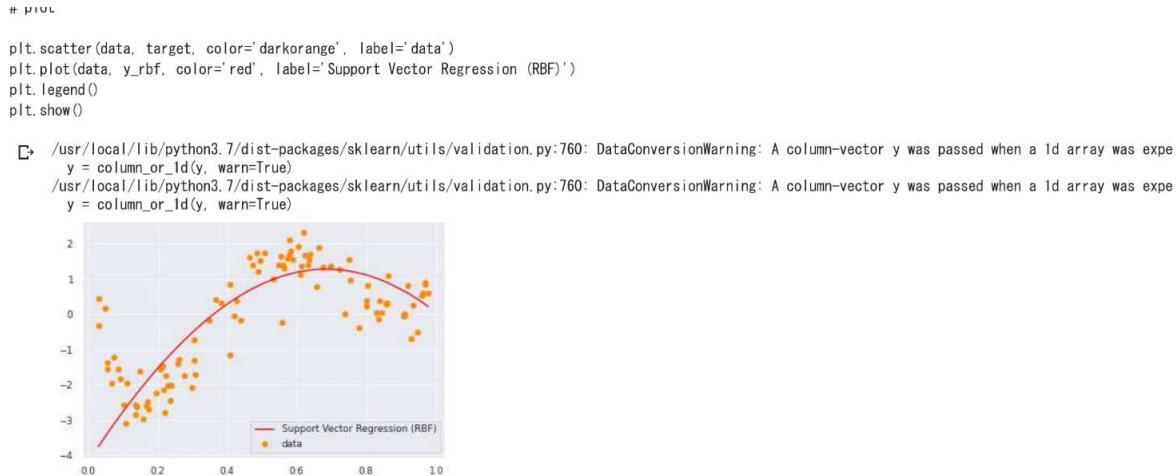
```
-4.440892098500626e-16
```



```
from sklearn import model_selection, preprocessing, linear_model, svm
```

```
# SVR-rbf
clf_svr = svm.SVR(kernel='rbf', C=1e3, gamma=0.1, epsilon=0.1)
clf_svr.fit(data, target)
y_rbf = clf_svr.fit(data, target).predict(data)

# rbf+
```



```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(data, target, test_size=0.1, random_state=0)
```

以下では、Googleドライブのマイドライブ直下にstudy_ai_mlフォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
from keras.callbacks import EarlyStopping, TensorBoard, ModelCheckpoint

cb_cp = ModelCheckpoint('/content/drive/My Drive/study_ai_ml/out/checkpoints/weights.{epoch:02d}-{val_loss:.2f}.hdf5', verbose=1, save_weights_only=True)
cb_tf = TensorBoard(log_dir='/content/drive/My Drive/study_ai_ml/out/tensorBoard', histogram_freq=0)

def relu_reg_model():
    model = Sequential()
    model.add(Dense(10, input_dim=1, activation='relu'))
    model.add(Dense(1000, activation='linear'))
    #    model.add(Dense(100, activation='relu'))
    #    model.add(Dense(100, activation='relu'))
    #    model.add(Dense(100, activation='relu'))
    #    model.add(Dense(100, activation='relu'))
    model.add(Dense(1))

    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

from keras.models import Sequential
from keras.layers import Input, Dense, Dropout, BatchNormalization
from keras.wrappers.scikit_learn import KerasRegressor

# use data split and fit to run the model
estimator = KerasRegressor(build_fn=relu_reg_model, epochs=100, batch_size=5, verbose=1)

history = estimator.fit(x_train, y_train, callbacks=[cb_cp, cb_tf], validation_data=(x_test, y_test))

Epoch 00086: saving model to /content/drive/My Drive/study_ai_ml/skl_ml/out/checkpoints/weights.86-0.57.hdf5
Epoch 87/100
18/18 [=====] - 0s 5ms/step - loss: 0.4125 - val_loss: 0.3816

Epoch 00087: saving model to /content/drive/My Drive/study_ai_ml/skl_ml/out/checkpoints/weights.87-0.38.hdf5
Epoch 88/100
18/18 [=====] - 0s 6ms/step - loss: 0.3979 - val_loss: 0.2080

Epoch 00088: saving model to /content/drive/My Drive/study_ai_ml/skl_ml/out/checkpoints/weights.88-0.21.hdf5
Epoch 89/100
18/18 [=====] - 0s 8ms/step - loss: 0.2264 - val_loss: 0.3992

Epoch 00089: saving model to /content/drive/My Drive/study_ai_ml/skl_ml/out/checkpoints/weights.89-0.40.hdf5
Epoch 90/100
18/18 [=====] - 0s 7ms/step - loss: 0.3585 - val_loss: 0.4991
```

3.ロジスティック回帰モデル

分類問題を解く種の教師あり機械学習モデル

- 分類問題（クラス分類）

- ある数値からクラスを分類する問題
- 分類で扱うデータ
 - 入力に対して出力 {0 or 1}
 - 有名なもの タイタニックのデータ、IRIS (あやめ) のデータ
- アプローチ
 - 識別的アプローチ (ロジスティック回帰)
 - 生成的アプローチ
- パラメータ

$$w = (w_1, w_2, \dots, w_m)^T \in R^m$$

- 線形結合

$$\hat{y} = \sum_{j=1}^m w_j x_j + w_0$$

$* X^T W \in R$
(実数全体を取りらず) \Rightarrow シグモイド関数をかませて $[0, 1]$ につぶすことができる。

- シグモイド関数

- 入力 : 実数
- 出力 : $[0, 1]$
- シグモイド関数の微分

シグモイド関数の微分—シグモイド関数で書ける

$$\begin{aligned}\sigma(z) &= \frac{1}{1 + \exp(-z)} \\ \frac{\partial \sigma(z)}{\partial z} &= -1 \{1 + \exp(-z)\}^{-2} \exp(-z)(-1) \\ &= \sigma(z)(1 - \sigma(z))\end{aligned}$$

- 求めたいこと

- シグモイド関数の出力を $Y = 1$ になる確率に対応させる

- 数式

$$P(Y = 1|x) = \sigma(w_0 + w_1 x_1)$$

- ベルヌーイ分布のパラメータ推定

- 分布からデータ推定
- データから分布を推定 \Rightarrow こっちをやりたい

- 尤度関数

データを固定し、パラメータを変化させる

尤度関数を最大化する \Rightarrow 最尤推定という

- 式

$$\begin{aligned}P(y_1, y_2, \dots, y_n; w_0, w_1, \dots, w_m) &= \prod_{i=1}^n p_i^{y_i} (1 - p_i)^{1-y_i} \\ &= \prod_{i=1}^n \sigma(w^T x_i)^{y_i} (1 - \sigma(w^T x_i))^{1-y_i} \\ &= L(w)\end{aligned}$$

- 計算を簡単にするため

- 対数をとると微分の計算が簡単になる
- 実装上は \log を取らないと、桁落ちが発生
 \Rightarrow 桁落ちしないように \log をとる

- 勾配降下法

- 対数尤度関数をパラメータで微分して 0 になる値を解析的に求めるのが困難なため。
- 反復学習によりパラメータを逐次的に更新するアプローチ

$$w(k+1) = w^k - \eta \frac{\partial E(w)}{\partial w}$$

$$\begin{aligned}Loss : E(W) &= -\log L(w) \\ &= - \sum_{i=1}^n \{y_i \log P_i + (1 - y_i) \log(1 - P_i)\} \\ &\Rightarrow \frac{\partial E(W)}{\partial W} = - \sum_{i=1}^n (y_i - P_i) X_i\end{aligned}$$

- パラメータが更新されなくなった場合、勾配が 0 になったということ

- 勾配降下法では、パラメータを更新するのにN個すべてのデータを求める必要がある
⇒ メモリーが足りない。
⇒ミニバッチ法（確率的勾配降下法）
- 確率的勾配降下法
 - データを1つづつランダムに選んでパラメータを更新する
*全体から1つだけとってきて更新する。

▼ ハンズオン（タイタニック）

▼ Google ドライブのマウント

```
from google.colab import drive  
drive.mount('/content/drive')
```

Mounted at /content/drive

0. データ表示

```
#from モジュール名 import クラス名 (もしくは関数名や変数名)
import pandas as pd
from pandas import DataFrame
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

#matplotlibをinlineで表示するためのおまじない (plt.show() しなくていい)
%matplotlib inline
```

以下では、Google ドライブのマイドライブ直下にstudy_ai_mlフォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
# titanic data csvファイルの読み込み  
titanic_df = pd.read_csv('/content/drive/My Drive/study_ai_ml/data/titanic_train.csv')
```

```
# ファイルの先頭部を表示し、データセットを確認する  
titanic_df.head(5)
```

1. ロジスティック回帰

不要なデータの削除・欠損値の補完

```
#予測に不要と考えるからうをドロップ (本当はこの情報もしっかり使うべきだと思っています)
titanic_df.drop(['PassengerId', 'Name', 'Ticket', 'Cabin'], axis=1, inplace=True)
```

```
#一部カラムをドロップしたデータを表示  
titanic_df.head()
```

	Survived	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked
0	0	3	male	22.0	1	0	7.2500	S
1	1	1	female	38.0	1	0	71.2833	C
2	1	3	female	26.0	0	0	7.9250	S
3	1	1	female	35.0	1	0	53.1000	S
4	0	3	male	35.0	0	0	8.0500	S

```
#nullを含んでいる行を表示  
titanic_df[titanic_df.isnull().any(1)].head(10)
```

```

      Survived Pclass   Sex   Age SibSp  Parch     Fare Embarked
      5         0     3 male  NaN      0      0    8.4583        Q
     17        1     2 male  NaN      0      0   13.0000        S
     19        1     3 female  NaN      0      0    7.2250        C
     26        0     3 male  NaN      0      0    7.2250        C
     28        1     3 female  NaN      0      0    7.8792        Q

#Ageカラムのnullを中央値で補完
titanic_df['AgeFill'] = titanic_df['Age'].fillna(titanic_df['Age'].mean())

#再度nullを含んでいる行を表示（Ageのnullは補完されている）
titanic_df[titanic_df.isnull().any(1)]

#titanic_df.dtypes

      Survived Pclass   Sex   Age SibSp  Parch     Fare Embarked  AgeFill
      5         0     3 male  NaN      0      0    8.4583        Q  29.699118
     17        1     2 male  NaN      0      0   13.0000        S  29.699118
     19        1     3 female  NaN      0      0    7.2250        C  29.699118
     26        0     3 male  NaN      0      0    7.2250        C  29.699118
     28        1     3 female  NaN      0      0    7.8792        Q  29.699118
     ...
     859       0     3 male  NaN      0      0    7.2292        C  29.699118
     863       0     3 female  NaN      8      2   69.5500        S  29.699118
     868       0     3 male  NaN      0      0    9.5000        S  29.699118
     878       0     3 male  NaN      0      0    7.8958        S  29.699118
     888       0     3 female  NaN      1      2   23.4500        S  29.699118

```

179 rows × 9 columns

1. ロジスティック回帰

実装(チケット価格から生死を判別)

```

#運賃だけのリストを作成
data1 = titanic_df.loc[:, ["Fare"]].values

#生死フラグのみのリストを作成
label1 = titanic_df.loc[:, ["Survived"]].values

from sklearn.linear_model import LogisticRegression

model=LogisticRegression()

model.fit(data1, label1)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760: DataConversionWarning: A column-vector y was passed when a 1d array was expected.
y = column_or_1d(y, warn=True)
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)

```

*ハイパーパラメータ C = 1.0 (正則化が入っている) *L2ノルムを利用

```

model.predict([[80]])
array([1])

model.predict_proba([[80]])
array([[0.43182354, 0.56817646]])

```

```

X_test_value = model.decision_function(data1)

# # 決定閾値 (絶対値が大きいほど識別境界から離れている)
# X_test_value = model.decision_function(X_test)
# # 決定閾値をシグモイド関数で確率に変換
# X_test_prob = normal_sigmoid(X_test_value)

print (model.intercept_)
print (model.coef_)

[-0.94131796]
[[0.01519666]]

w_0 = model.intercept_[0]
w_1 = model.coef_[0,0]

# def normal_sigmoid(x):
#     return 1 / (1+np.exp(-x))

def sigmoid(x):
    return 1 / (1+np.exp(-(w_1*x+w_0)))

x_range = np.linspace(-1, 500, 3000)

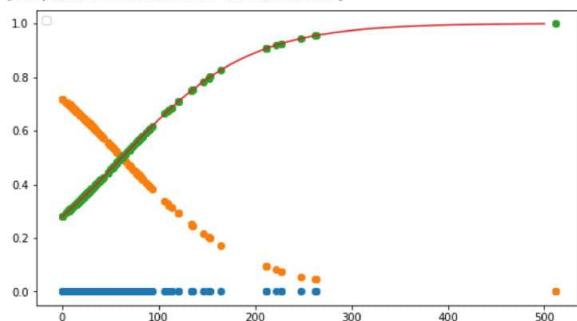
plt.figure(figsize=(9, 5))
#plt.xkcd()
plt.legend(loc=2)

# plt.ylim(-0.1, 1.1)
# plt.xlim(-10, 10)

# plt.plot([-10,10], [0,0], "k", lw=1)
# plt.plot([0,0], [-1,1.5], "k", lw=1)
plt.plot(data1,np.zeros(len(data1)), 'o')
plt.plot(data1, model.predict_proba(data1), 'o')
plt.plot(x_range, sigmoid(x_range), '-')
#plt.plot(x_range, normal_sigmoid(x_range), '-')
#

```

No handles with labels found to put in legend.
[<matplotlib.lines.Line2D at 0x7f93c7725990>]



1. ロジスティック回帰

実装(2変数から生死を判別)

```

#AgeFillの欠損値を埋めたので
titanic_df = titanic_df.drop(['Age'], axis=1)

titanic_df['Gender'] = titanic_df['Sex'].map({'female': 0, 'male': 1}).astype(int)

titanic_df.head(3)

```

```

Survived Pclass   Sex   Age  SibSp  Parch    Fare Embarked  AgeFill  Gender
0         0      3 male  22.0     1     0  7.2500      S    22.0     1
titanic_df['Pclass_Gender'] = titanic_df['Pclass'] + titanic_df['Gender']
2         1      3 female  26.0     0     0  7.9250      S    26.0     0
titanic_df.head()

Survived Pclass   Sex   Age  SibSp  Parch    Fare Embarked  AgeFill  Gender  Pclass_Gender
0         0      3 male  22.0     1     0  7.2500      S    22.0     1       4
1         1      1 female  38.0     1     0 71.2833      C    38.0     0       1
2         1      3 female  26.0     0     0  7.9250      S    26.0     0       3
3         1      1 female  35.0     1     0 53.1000      S    35.0     0       1
4         0      3 male  35.0     0     0  8.0500      S    35.0     1       4

titanic_df = titanic_df.drop(['Pclass', 'Sex', 'Gender', 'Age'], axis=1)

titanic_df.head()

Survived SibSp  Parch    Fare Embarked  AgeFill  Pclass_Gender
0         0     1     0  7.2500      S    22.0       4
1         1     1     0 71.2833      C    38.0       1
2         1     0     0  7.9250      S    26.0       3
3         1     1     0 53.1000      S    35.0       1
4         0     0     0  8.0500      S    35.0       4

# 重要なだよ！！！
# 境界線の式
#  $w_1 \cdot x + w_2 \cdot y + w_0 = 0$ 
#  $\Rightarrow y = (-w_1 \cdot x - w_0) / w_2$ 

# # 境界線 プロット
# plt.plot([-2, 2], map(lambda x: (-w_1 * x - w_0) / w_2, [-2, 2]))

# # データを重ねる
# plt.scatter(X_train_std[y_train==0, 0], X_train_std[y_train==0, 1], c='red', marker='x', label='train 0')
# plt.scatter(X_train_std[y_train==1, 0], X_train_std[y_train==1, 1], c='blue', marker='x', label='train 1')
# plt.scatter(X_test_std[y_test==0, 0], X_test_std[y_test==0, 1], c='red', marker='o', s=60, label='test 0')
# plt.scatter(X_test_std[y_test==1, 0], X_test_std[y_test==1, 1], c='blue', marker='o', s=60, label='test 1')

np.random.seed = 0

xmin, xmax = -5, 85
ymin, ymax = 0.5, 4.5

index_survived = titanic_df[titanic_df["Survived"]==0].index
index_notsurvived = titanic_df[titanic_df["Survived"]==1].index

from matplotlib.colors import ListedColormap
fig, ax = plt.subplots()
cm = plt.cm.RdBu
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
sc = ax.scatter(titanic_df.loc[index_survived, 'AgeFill'],
                titanic_df.loc[index_survived, 'Pclass_Gender']+(np.random.rand(len(index_survived))-0.5)*0.1,
                color='r', label='Not Survived', alpha=0.3)
sc = ax.scatter(titanic_df.loc[index_notsurvived, 'AgeFill'],
                titanic_df.loc[index_notsurvived, 'Pclass_Gender']+(np.random.rand(len(index_notsurvived))-0.5)*0.1,
                color='b', label='Survived', alpha=0.3)
ax.set_xlabel('AgeFill')
ax.set_ylabel('Pclass_Gender')
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
ax.legend(bbox_to_anchor=(1.4, 1.03))

```

```

<matplotlib.legend.Legend at 0x7f93c6005d50>

#運賃だけのリストを作成
data2 = titanic_df.loc[:, ["AgeFill", "Pclass_Gender"]].values
= | |
data2

array([[22.      , 4.      ],
       [38.      , 1.      ],
       [26.      , 3.      ],
       ...,
       [29.69911765, 3.      ],
       [26.      , 2.      ],
       [32.      , 4.      ]])

#生存フラグのみのリストを作成
label2 = titanic_df.loc[:, ["Survived"]].values

model2 = LogisticRegression()

model2.fit(data2, label2)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760: DataConversionWarning: A column-vector y was passed when a 1d array was expected. y = column_or_1d(y, warn=True)
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='auto', n_jobs=None, penalty='l2',
                   random_state=None, solver='lbfgs', tol=0.0001, verbose=0,
                   warm_start=False)

model2.predict([[10, 1]])
array([1])

model2.predict_proba([[10, 1]])
array([[0.03754749, 0.96245251]])

titanic_df.head(3)

   Survived SibSp Parch     Fare Embarked AgeFill Pclass_Gender
0         0      1      0    7.2500        S    22.0          4
1         1      1      0   71.2833        C    38.0          1
2         1      0      0    7.9250        S    26.0          3

h = 0.02
xmin, xmax = -5, 85
ymin, ymax = 0.5, 4.5
xx, yy = np.meshgrid(np.arange(xmin, xmax, h), np.arange(ymin, ymax, h))
Z = model2.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
Z = Z.reshape(xx.shape)

fig, ax = plt.subplots()
levels = np.linspace(0, 1.0)
cm = plt.cm.RdBu
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
#contour = ax.contourf(xx, yy, Z, cmap=cm, levels=levels, alpha=0.5)

sc = ax.scatter(titanic_df.loc[index_survived, 'AgeFill'],
                titanic_df.loc[index_survived, 'Pclass_Gender']+np.random.rand(len(index_survived))-0.5)*0.1,
                color='r', label='Not Survived', alpha=0.3)
sc = ax.scatter(titanic_df.loc[index_notsurvived, 'AgeFill'],
                titanic_df.loc[index_notsurvived, 'Pclass_Gender']+np.random.rand(len(index_notsurvived))-0.5)*0.1,
                color='b', label='Survived', alpha=0.3)

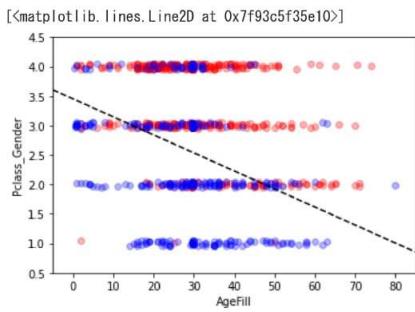
ax.set_xlabel('AgeFill')
ax.set_ylabel('Pclass_Gender')
ax.set_xlim(xmin, xmax)
ax.set_ylim(ymin, ymax)
#fig.colorbar(contour)

x1 = xmin
-
```

```

x2 = xmax
y1 = -1*(model2.intercept_[0]+model2.coef_[0][0]*xmin)/model2.coef_[0][1]
y2 = -1*(model2.intercept_[0]+model2.coef_[0][0]*xmax)/model2.coef_[0][1]
ax.plot([x1, x2], [y1, y2], 'k--')

```



2. モデル評価

混同行列とクロスバリデーション

```

from sklearn.model_selection import train_test_split

traindata1, testdata1, trainlabel1, testlabel1 = train_test_split(data1, label1, test_size=0.2)
traindata1.shape
trainlabel1.shape
(712, 1)

testdata2, trainlabel2 = train_test_split(data2, label2, test_size=0.2)
traindata2.shape
trainlabel2.shape
#本来は同じデータセットを分割しなければいけない。(簡単に別々に分割している。)
(712, 1)

data = titanic_df.loc[:, :].values
label = titanic_df.loc[:, ["Survived"]].values
traindata, testdata, trainlabel, testlabel = train_test_split(data, label, test_size=0.2)
traindata.shape
trainlabel.shape
(712, 1)

eval_model1=LogisticRegression()
eval_model2=LogisticRegression()
#eval_model=LogisticRegression()

predictor_eval1=eval_model1.fit(traindata1, trainlabel1).predict(testdata1)
predictor_eval2=eval_model2.fit(traindata2, trainlabel2).predict(testdata2)
#predictor_eval=eval_model.fit(traindata, trainlabel).predict(testdata)

/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760: DataConversionWarning: A column-vector y was passed when a 1d array was expected.
y = column_or_1d(y, warn=True)
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:760: DataConversionWarning: A column-vector y was passed when a 1d array was expected.
y = column_or_1d(y, warn=True)

eval_model1.score(traindata1, trainlabel1)
0.6671348314606742

eval_model1.score(testdata1,testlabel1)
0.6536312849162011

eval_model2.score(traindata2, trainlabel2)
0.7710674157303371

eval_model2.score(testdata2,testlabel2)

```

```
0.7821229050279329
```

```
from sklearn import metrics
print(metrics.classification_report(testlabel1, predictor_eval1))
print(metrics.classification_report(testlabel2, predictor_eval2))

          precision    recall  f1-score   support

           0       0.66      0.91      0.76      109
           1       0.64      0.26      0.37       70

   accuracy                           0.65      179
  macro avg       0.65      0.58      0.56      179
weighted avg       0.65      0.65      0.61      179

          precision    recall  f1-score   support

           0       0.77      0.88      0.82      103
           1       0.80      0.64      0.72       76

   accuracy                           0.78      179
  macro avg       0.79      0.76      0.77      179
weighted avg       0.78      0.78      0.78      179
```

```
from sklearn.metrics import confusion_matrix
confusion_matrix1=confusion_matrix(testlabel1, predictor_eval1)
confusion_matrix2=confusion_matrix(testlabel2, predictor_eval2)
```

```
confusion_matrix1
```

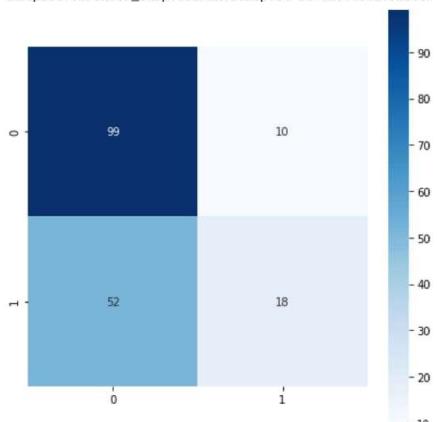
```
array([[99, 10],
       [52, 18]])
```

```
confusion_matrix2
```

```
array([[91, 12],
       [27, 49]])
```

```
fig = plt.figure(figsize = (7,7))
#plt.title(title)
sns.heatmap(
    confusion_matrix1,
    vmin=None,
    vmax=None,
    cmap="Blues",
    center=None,
    robust=False,
    annot=True, fmt='%.2g',
    annot_kws=None,
    linewidths=0,
    linecolor='white',
    cbar=True,
    cbar_kws=None,
    cbar_ax=None,
    square=True, ax=None,
    #xticklabels=columns,
    #yticklabels=columns,
    mask=None)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f93cb17b910>
```

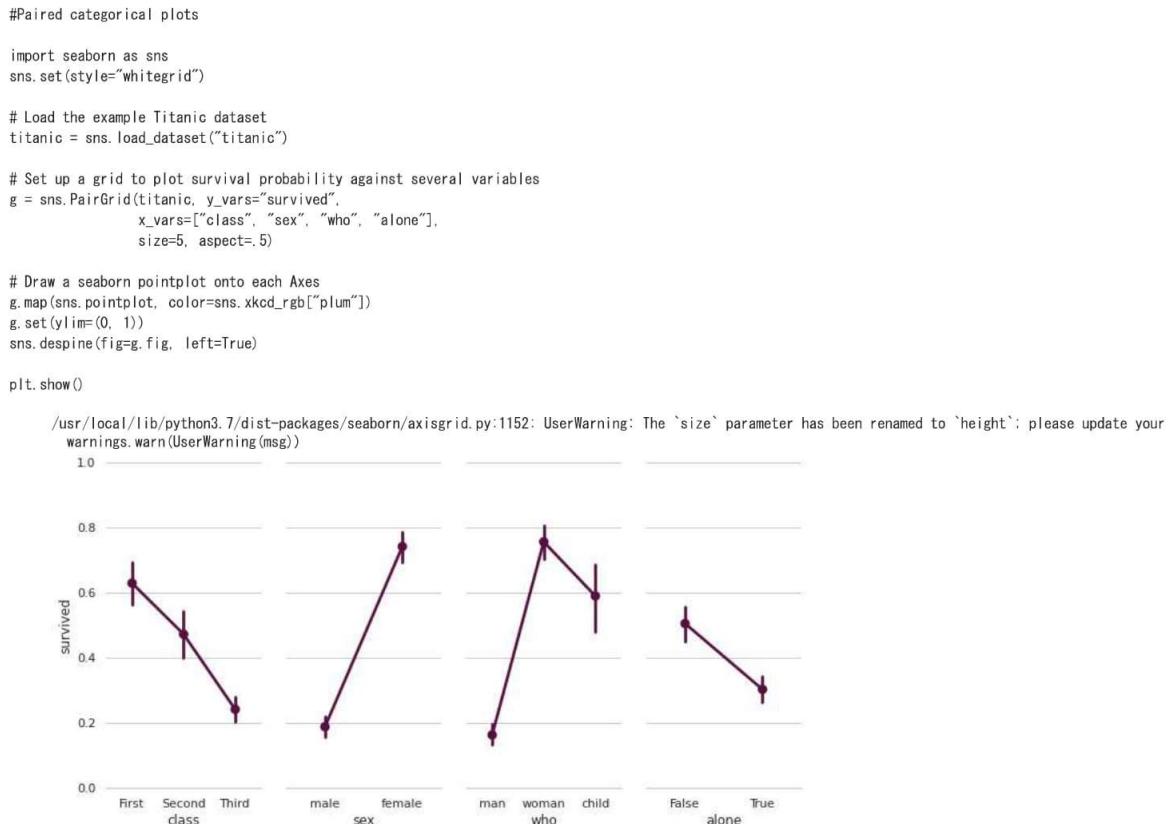


```
fig = plt.figure(figsize = (7,7))
```

```
#plt.title(title)
sns.heatmap(
    confusion_matrix2,
    vmin=None,
    vmax=None,
    cmap="Blues",
    center=None,
    robust=False,
    annot=True, fmt=' .2g',
    annot_kws=None,
    linewidths=0,
    linecolor='white',
    cbar=True,
    cbar_kws=None,
    cbar_ax=None,
    square=True, ax=None,
    #xticklabels=columns,
    #yticklabels=columns,
    mask=None)

<matplotlib.axes._subplots.AxesSubplot at 0x7f93bd613610>
```

	0	1
0	27	91
1	49	12



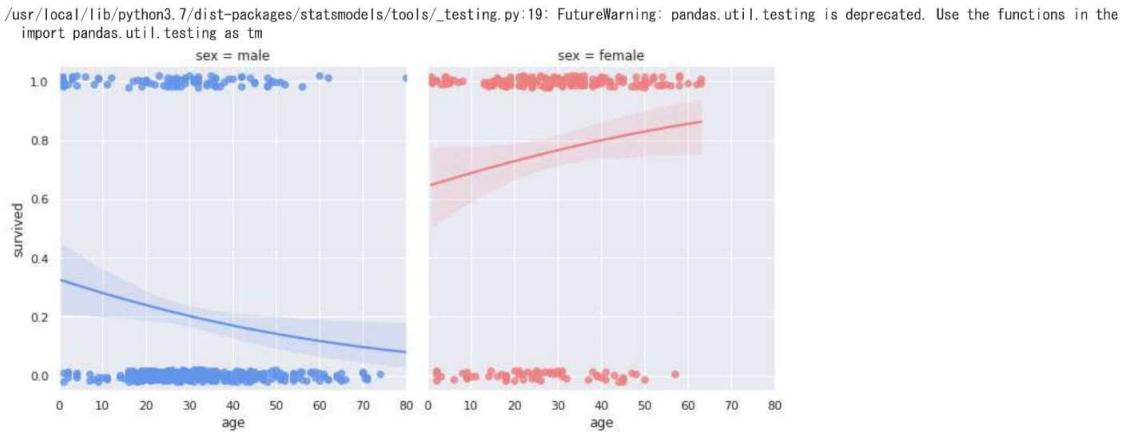
```
#Faceted logistic regression

import seaborn as sns
sns.set(style="darkgrid")

# Load the example titanic dataset
df = sns.load_dataset("titanic")

# Make a custom palette with gendered colors
pal = dict(male="#6495ED", female="#F08080")

# Show the survival probability as a function of age and sex
g = sns.lmplot(x="age", y="survived", col="sex", hue="sex", data=df,
                 palette=pal, y_jitter=.02, logistic=True)
g.set(xlim=(0, 80), ylim=(-.05, 1.05))
plt.show()
```



実装演習（ロジスティック回帰）

▼ ロジスティック回帰

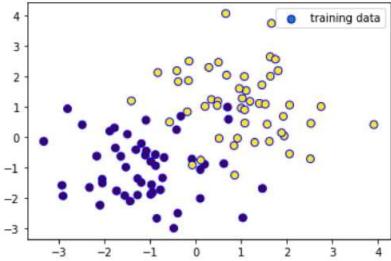
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

n_sample = 100
harf_n_sample = 50
var = .2

def gen_data(n_sample, harf_n_sample):
    x0 = np.random.normal(size=n_sample).reshape(-1, 2) - 1.
    x1 = np.random.normal(size=n_sample).reshape(-1, 2) + 1.
    x_train = np.concatenate([x0, x1])
    y_train = np.concatenate([np.zeros(harf_n_sample), np.ones(harf_n_sample)]).astype(np.int)
    return x_train, y_train

def plt_data(x_train, y_train):
    plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train, facecolor="none", edgecolor="b", s=50, label="training data")
    plt.legend()

#データ作成
x_train, y_train = gen_data(n_sample, harf_n_sample)
#データ表示
plt_data(x_train, y_train)
```



▼ ロジスティック回帰モデル

識別モデルとして $p(y=1|\mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x})$ を用いる。

ただし、 $\sigma(\cdot)$ はシグモイド関数であり、 $\sigma(h) = \frac{1}{1+\exp(-h)}$ で定義される。

また、陽には書かないが、 \mathbf{x} には定数項のための1という要素があることを仮定する。

▼ 学習

訓練データ $X = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n]^T$, $\mathbf{y} = [y_1, y_2, \dots, y_n]^T$ ($y_i = \{0, 1\}$)に対して尤度関数 L は以下のように書ける。

$$L(\mathbf{w}) = \prod_{i=1}^n p(y_i = 1|\mathbf{x}_i; \mathbf{w})^{y_i} (1 - p(y_i = 1|\mathbf{x}_i; \mathbf{w}))^{1-y_i}$$

負の対数尤度関数は

$$-\log L(\mathbf{w}) = -\sum_{i=1}^n [y_i \log p(y_i = 1|\mathbf{x}_i; \mathbf{w}) + (1 - y_i) \log (1 - p(y_i = 1|\mathbf{x}_i; \mathbf{w}))]$$

のように書ける。これを最小化する \mathbf{w} を求める。

$$\frac{d\sigma(h)}{dh} = \sigma(h)(1 - \sigma(h))$$
と書けることを考慮し、負の対数尤度関数を \mathbf{w} で偏微分すると、

$$\begin{aligned} \frac{\partial}{\partial \mathbf{w}} (-\log L(\mathbf{w})) &= -\sum_{i=1}^n [y_i(1 - \sigma(\mathbf{w}^T \mathbf{x}_i)) - (1 - y_i)\sigma(\mathbf{w}^T \mathbf{x}_i)] \mathbf{x}_i \\ &= \sum_{i=1}^n (\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i) \mathbf{x}_i \end{aligned}$$

この式が0となる \mathbf{w} は解析的に求められないので、今回は $-\log L(\mathbf{w})$ の最小化問題を最急降下法を用いて解く。

最急降下法では学習率を η とすると、以下の式で \mathbf{w} を更新する。

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial}{\partial \mathbf{w}} (-\log L(\mathbf{w}))$$

```

def add_one(x):
    return np.concatenate([np.ones(len(x))[:, None], x], axis=1)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sgd(X_train, max_iter, eta):
    w = np.zeros(X_train.shape[1])
    for _ in range(max_iter):
        w_prev = np.copy(w)
        sigma = sigmoid(np.dot(X_train, w))
        grad = np.dot(X_train.T, (sigma - y_train))
        w -= eta * grad
        if np.allclose(w, w_prev):
            return w
    return w

X_train = add_one(x_train)
max_iter=100
eta = 0.01
w = sgd(X_train, max_iter, eta)

```

▼ 予測

入力に対して、 $y = 1$ である確率を出力する。よって

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

0.5より大きければ1に、小さければ0に分類する。

```

xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

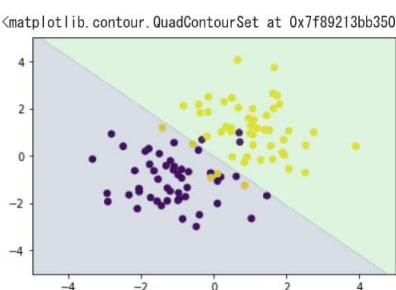
```

```

X_test = add_one(xx)
proba = sigmoid(np.dot(X_test, w))
y_pred = (proba > 0.5).astype(np.int)

plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
plt.contourf(xx0, xx1, proba.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))

```



#numpy実装

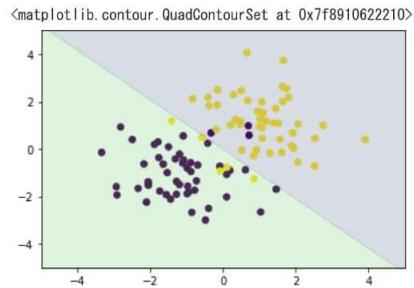
```

from sklearn.linear_model import LogisticRegression
model=LogisticRegression(fit_intercept=True)
model.fit(x_train, y_train)
proba = model.predict_proba(xx)
y_pred = (proba > 0.5).astype(np.int)

plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
plt.contourf(xx0, xx1, proba[:, 0].reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))

```

□



▼ 4. 主成分分析 (PCA)

- 2次元のデータを1次元に圧縮したい。
- 式を追うより何をしているのかを第一に考える
- 分散が最大になるように1次元をとる
- 式

- 以下の制約付き最適化問題を解く

- 目的関数

$$\arg \max_{\substack{a \in R^m}} a_j^T \text{Var}(\bar{X}) a_j$$

- 制約条件

$$a_j^T a_j = 1$$

- 制約付き最適化問題の解き方

- ラグランジュ関数を最大にする係数ベクトルを探す（微分して0）

$$E(a_j) = a_j^T \text{Var}(\bar{X}) a_j - \lambda(a_j^T a_j - 1)$$

$$\frac{\partial E(a_j)}{\partial a_j} = 2 \text{Var}(\bar{X}) a_j - 2\lambda a_j = 0$$

$$\Rightarrow \text{Var}(\bar{X}) a_j = \lambda a_j$$

主成分分析ハンズオン（乳がん検査データ）

▼ Google ドライブのマウント

```
from google.colab import drive
drive.mount('/content/drive')

Mounted at /content/drive

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegressionCV
from sklearn.metrics import confusion_matrix
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
%matplotlib inline
```

▼ sys.pathの設定

以下では、Google ドライブのマイドライブ直下にstudy_ai_ml フォルダを置くことを仮定しています。必要に応じて、パスを変更してください。

```
cancer_df = pd.read_csv('/content/drive/My Drive/study_ai_ml/data/cancer.csv')

print('cancer df shape: {}'.format(cancer_df.shape))

cancer df shape: (569, 33)

cancer_df

   id diagnosis radius_mean texture_mean perimeter_mean area_mean smoothness_mean compactness_mean concavity_mean
0  842302      M       17.99      10.38     122.80    1001.0      0.11840      0.27760      0.30010
1  842517      M       20.57      17.77     132.90    1326.0      0.08474      0.07864      0.08690
2  84300903     M       19.69      21.25     130.00    1203.0      0.10960      0.15990      0.19740
3  84348301     M       11.42      20.38      77.58     386.1      0.14250      0.28390      0.24140
4  84358402     M       20.29      14.34     135.10    1297.0      0.10030      0.13280      0.19800
...
564 926424      M       21.56      22.39     142.00    1479.0      0.11100      0.11590      0.24390
565 926682      M       20.13      28.25     131.20    1261.0      0.09780      0.10340      0.14400
566 926954      M       16.60      28.08     108.30     858.1      0.08455      0.10230      0.09251
567 927241      M       20.60      29.33     140.10    1265.0      0.11780      0.27700      0.35140
568 92751       B        7.76      24.54      47.92     181.0      0.05263      0.04362      0.00000

569 rows × 33 columns
```

```
cancer_df.drop('Unnamed: 32', axis=1, inplace=True)
cancer_df
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010

・ diagnosis: 診断結果(良性がB / 悪性がM) ・ 説明変数は3列以降、目的変数を2列目としロジスティック回帰で分類

```
# 目的変数の抽出
y = cancer_df.diagnosis.apply(lambda d: 1 if d == 'M' else 0)

# 説明変数の抽出
X = cancer_df.loc[:, 'radius_mean']

# 学習用とテスト用でデータを分離
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# 標準化
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

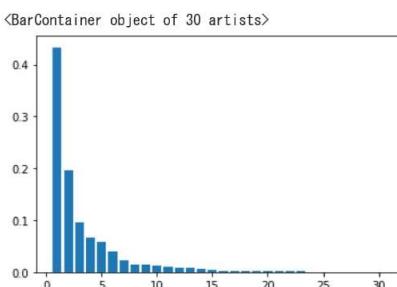
# ロジスティック回帰で学習
logistic = LogisticRegressionCV(cv=10, random_state=0, max_iter=1000)
logistic.fit(X_train_scaled, y_train)

# 検証
print('Train score: {:.3f}'.format(logistic.score(X_train_scaled, y_train)))
print('Test score: {:.3f}'.format(logistic.score(X_test_scaled, y_test)))
print('Confusion matrix:\n{}'.format(confusion_matrix(y_true=y_test, y_pred=logistic.predict(X_test_scaled))))
```

Train score: 0.988
Test score: 0.972
Confusion matrix:
[[89 1]
 [3 50]]

・検証スコア97%で分類できることを確認

```
pca = PCA(n_components=30)
pca.fit(X_train_scaled)
plt.bar([n for n in range(1, len(pca.explained_variance_ratio_)+1)], pca.explained_variance_ratio_)
```

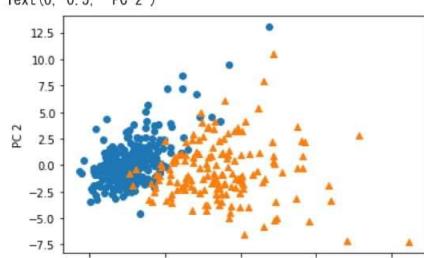


```
# PCA
# 次元数2まで圧縮
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
print('X_train_pca shape: {}'.format(X_train_pca.shape))
# X_train_pca shape: (426, 2)

# 寄与率
print('explained variance ratio: {}'.format(pca.explained_variance_ratio_))
# explained variance ratio: [ 0.43315126  0.19586506]

# 散布図にプロット
temp = pd.DataFrame(X_train_pca)
temp['Outcome'] = y_train.values
b = temp[temp['Outcome'] == 0]
m = temp[temp['Outcome'] == 1]
plt.scatter(x=b[0], y=b[1], marker='o') # 良性は○でマーク
plt.scatter(x=m[0], y=m[1], marker='^') # 悪性は△でマーク
plt.xlabel('PC 1') # 第1主成分をx軸
plt.ylabel('PC 2') # 第2主成分をy軸
```

```
X_train_pca shape: (426, 2)
explained variance ratio: [0.43315126 0.19586506]
Text(0, 0.5, 'PC 2')
```



▼ 主成分分析実装演習

▼ 主成分分析

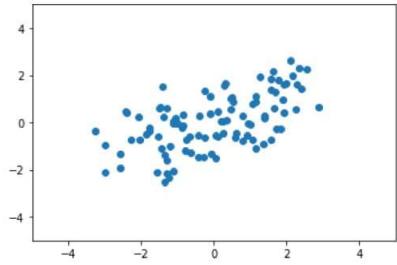
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

n_sample = 100

def gen_data(n_sample):
    mean = [0, 0]
    cov = [[2, 0.7], [0.7, 1]]
    return np.random.multivariate_normal(mean, cov, n_sample)

def plt_data(X):
    plt.scatter(X[:, 0], X[:, 1])
    plt.xlim(-5, 5)
    plt.ylim(-5, 5)

X = gen_data(n_sample)
plt_data(X)
```



▼ 学習

訓練データ $X = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n]^T$ に対して $\mathbb{E}[\mathbf{x}] = \mathbf{0}$ となるように変換する。
すると、不偏共分散行列は $Var[\mathbf{x}] = \frac{1}{n-1} X^T X$ と書ける。
 $Var[\mathbf{x}]$ を固有値分解し、固有値の大きい順に対応する固有ベクトルを第1主成分(\mathbf{w}_1)、第2主成分(\mathbf{w}_2)、...とよぶ。

```
n_components=2

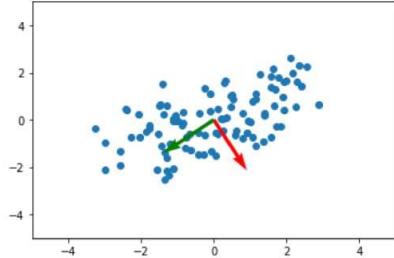
def get_moments(X):
    mean = X.mean(axis=0)
    stan_cov = np.dot((X - mean).T, X - mean) / (len(X) - 1)
    return mean, stan_cov

def get_components(eigenvectors, n_components):
    # W = eigenvectors[:, :n_components]
    # return W.T[::1]
    W = eigenvectors[:, ::-1][:, :n_components]
    return W.T

def plt_result(X, first, second):
    plt.scatter(X[:, 0], X[:, 1])
    plt.xlim(-5, 5)
    plt.ylim(-5, 5)
    # 第1主成分
    plt.quiver(0, 0, first[0], first[1], width=0.01, scale=6, color='red')
    # 第2主成分
    plt.quiver(0, 0, second[0], second[1], width=0.01, scale=6, color='green')

# 分散共分散行列を標準化
mean, stan_cov = get_moments(X)
# 固有値と固有ベクトルを計算
eigenvalues, eigenvectors = np.linalg.eigh(stan_cov)
components = get_components(eigenvectors, n_components)

plt_result(X, eigenvectors[0, :], eigenvectors[1, :])
```

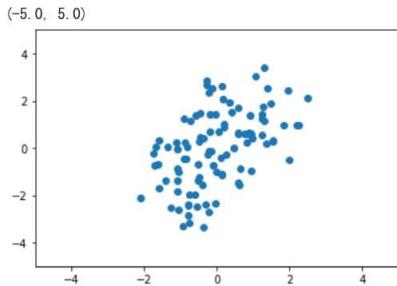


▼ 変換（射影）

元のデータを m 次元に変換(射影)するときは行列 W を $W = [w_1, w_2, \dots, w_m]$ とし、データ点 x を $z = W^T x$ によって変換(射影)する。よって、データ X に対しては $Z = X^T W$ によって変換する。

```
def transform_by_pca(X, pca):
    mean = X.mean(axis=0)
    return np.dot(X-mean, components)
```

```
Z = transform_by_pca(X, components.T)
plt.scatter(Z[:, 0], Z[:, 1])
plt.xlim(-5, 5)
plt.ylim(-5, 5)
```

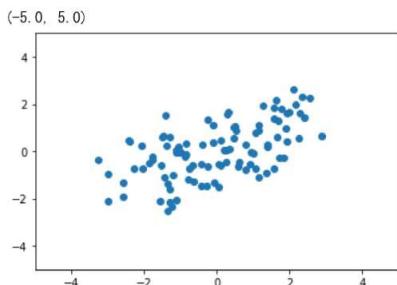


▼ 逆変換

射影されたデータ点 z を元のデータ空間へ逆変換するときは $\bar{x} = (W^T)^{-1} z = W z$ によって変換する。よって、射影されたデータ Z に対しては $\bar{X} = Z W^T$ によって変換する。

```
mean = X.mean(axis=0)
X_ = np.dot(Z, components.T) + mean
```

```
plt.scatter(X_[:, 0], X_[:, 1])
plt.xlim(-5, 5)
plt.ylim(-5, 5)
```



```
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)
```

```

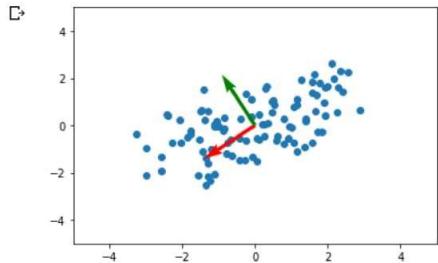
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
    svd_solver='auto', tol=0.0, whiten=False)

print('components: {}'.format(pca.components_))
print('mean: {}'.format(pca.mean_))
print('covariance: {}'.format(pca.get_covariance()))

components: [[-0.83671721 -0.5476352]
 [-0.5476352  0.83671721]]
mean: [-0.01956398  0.02653667]
covariance: [[2.29360975 1.00340809]
 [1.00340809 1.35996444]]

```

```
plt_result(X, pca.components_[0, :], pca.components_[1, :])
```



```

from sklearn.decomposition import PCA
pca = PCA(n_components=2)
pca.fit(X)
plt_result(X, pca.components_[0, :], pca.components_[1, :])

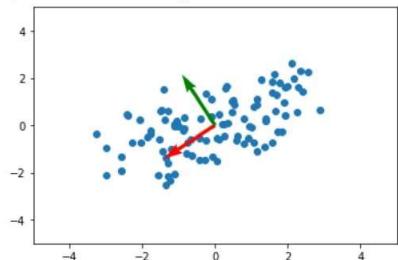
print('components: {}'.format(pca.components_))
print('mean: {}'.format(pca.mean_))
print('covariance: {}'.format(pca.get_covariance()))

```

```

components: [[-0.83671721 -0.5476352]
 [-0.5476352  0.83671721]]
mean: [-0.01956398  0.02653667]
covariance: [[2.29360975 1.00340809]
 [1.00340809 1.35996444]]

```



5.アルゴリズム

- k-近傍法 (kNN)
 - 分類問題のための機械学習
 - 教師なし学習
 - 最近傍のデータを k つとってきて、それらが最も多く所属するクラスに識別
 - 新たなデータからの距離が近い順に k 個データをとってきてどちらのほうが多いかで新たなデータの分類を決定する。
 - k を変化させると値が変わってしまう。（ k の値の決め方が問題）
 - k を大きくすると決定境界はなめらかになる。

▼ k 近傍法実装演習

▼ k近傍法

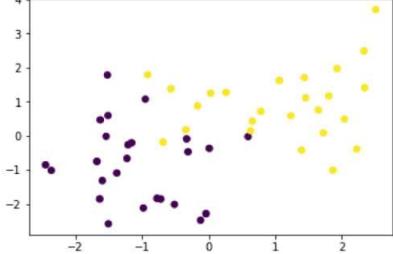
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

def gen_data():
    x0 = np.random.normal(size=50).reshape(-1, 2) - 1
    x1 = np.random.normal(size=50).reshape(-1, 2) + 1.
    x_train = np.concatenate([x0, x1])
    y_train = np.concatenate([np.zeros(25), np.ones(25)]).astype(np.int)
    return x_train, y_train
```

▼ 訓練データ生成

```
X_train, ys_train = gen_data()
plt.scatter(X_train[:, 0], X_train[:, 1], c=ys_train)

<matplotlib.collections.PathCollection at 0x7efe363cebd0>
```



▼ 学習

陽に訓練ステップはない

▼ 予測

予測するデータ点との、距離が最も近いk個の、訓練データのラベルの最頻値を割り当てる

```
def distance(x1, x2):
    return np.sum((x1 - x2)**2, axis=1)

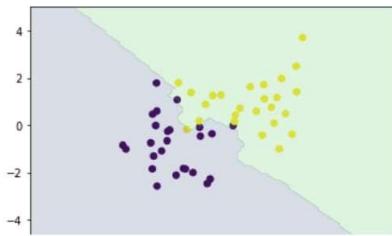
def knc_predict(n_neighbors, x_train, y_train, X_test):
    y_pred = np.empty(len(X_test), dtype=y_train.dtype)
    for i, x in enumerate(X_test):
        distances = distance(x, X_train)
        nearest_index = distances.argsort()[:n_neighbors]
        mode, _ = stats.mode(y_train[nearest_index])
        y_pred[i] = mode
    return y_pred

def plt_resut(x_train, y_train, y_pred):
    xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
    xx = np.array([xx0, xx1]).reshape(2, -1)
    plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
    plt.contourf(xx0, xx1, y_pred.reshape(100, 100).astype(dtype=np.float), alpha=0.2, levels=np.linspace(0, 1, 3))

n_neighbors = 3

xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
X_test = np.array([xx0, xx1]).reshape(2, -1)

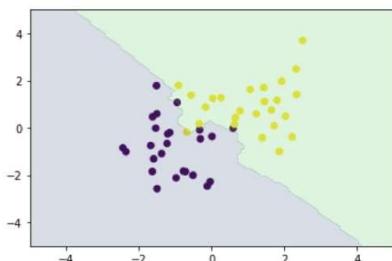
y_pred = knc_predict(n_neighbors, X_train, ys_train, X_test)
plt_resut(X_train, ys_train, y_pred)
```



▼ numpy実装

```
xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T
```

```
from sklearn.neighbors import KNeighborsClassifier
knc = KNeighborsClassifier(n_neighbors=n_neighbors).fit(X_train, ys_train)
plt_result(X_train, ys_train, knc.predict(xx))
```



- k-平均法 (k-means)
 - 教師なし学習
 - クラスタリング（特徴の似ている者同士をグループ化する）
 - 与えられたデータを k 個のクラスタに分類する
 - アルゴリズム
 - 1.各クラスタ中心の初期値を設定する
 - 2.各データ点に対して、各クラスタ中心との距離を計算し、最も距離が近いクラスタへ割り当てる
 - 3.各クラスタの平均ベクトル（中心）を計算
 - 4.収束するまで2.3の処理を繰り返す
 - 利点
 - 初期値が離れる ⇒ うまくクラスタリングができる
 - 欠点
 - 初期値が近い ⇒ うまくクラスタリングができない

k-meansハンズオン

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn import cluster, preprocessing, datasets

from sklearn.cluster import KMeans

wine = datasets.load_wine()

X = wine.data

X.shape
(178, 13)

y=wine.target

y.shape
(178,)

wine.target_names
array(['class_0', 'class_1', 'class_2'], dtype='|<U7')

model = KMeans(n_clusters=3)

labels = model.fit_predict(X)

df = pd.DataFrame({'labels': labels})
type(df)
pandas.core.frame.DataFrame

def species_label(theta):
    if theta == 0:
        return wine.target_names[0]
    if theta == 1:
        return wine.target_names[1]
    if theta == 2:
        return wine.target_names[2]

df['species'] = [species_label(theta) for theta in wine.target]

pd.crosstab(df['labels'], df['species'])

species  class_0  class_1  class_2
labels
0          0       50      19
1         46       1       0
2         13      20      29

```

▼ k-means実装演習

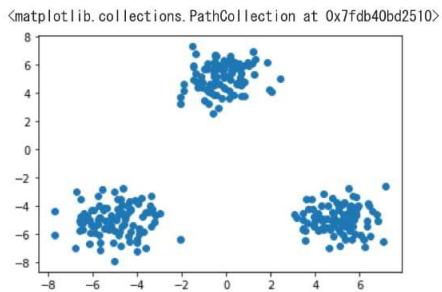
▼ k平均クラスタリング(k-means)

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

▼ データ生成

```
def gen_data():
    x1 = np.random.normal(size=(100, 2)) + np.array([-5, -5])
    x2 = np.random.normal(size=(100, 2)) + np.array([5, -5])
    x3 = np.random.normal(size=(100, 2)) + np.array([0, 5])
    return np.vstack((x1, x2, x3))
```

```
#データ作成
X_train = gen_data()
#データ描画
plt.scatter(X_train[:, 0], X_train[:, 1])
```



▼ 学習

k-meansアルゴリズムは以下のとおりである

- 1) 各クラスタ中心の初期値を設定する
- 2) 各データ点に対して、各クラスタ中心との距離を計算し、最も距離が近いクラスタを割り当てる
- 3) 各クラスタの平均ベクトル（中心）を計算する
- 4) 収束するまで2, 3の処理を繰り返す

```
def distance(x1, x2):
    return np.sum((x1 - x2)**2, axis=1)

n_clusters = 3
iter_max = 100

# 各クラスタ中心をランダムに初期化
centers = X_train[np.random.choice(len(X_train), n_clusters, replace=False)]

for _ in range(iter_max):
    prev_centers = np.copy(centers)
    D = np.zeros((len(X_train), n_clusters))
    # 各データ点に対して、各クラスタ中心との距離を計算
    for i, x in enumerate(X_train):
        D[i] = distance(x, centers)
    # 各データ点に最も距離が近いクラスタを割り当てる
    cluster_index = np.argmin(D, axis=1)
    # 各クラスタの中心を計算
    for k in range(n_clusters):
        index_k = cluster_index == k
        centers[k] = np.mean(X_train[index_k], axis=0)
    # 収束判定
    if np.allclose(prev_centers, centers):
        break
```

▼ クラスタリング結果

```

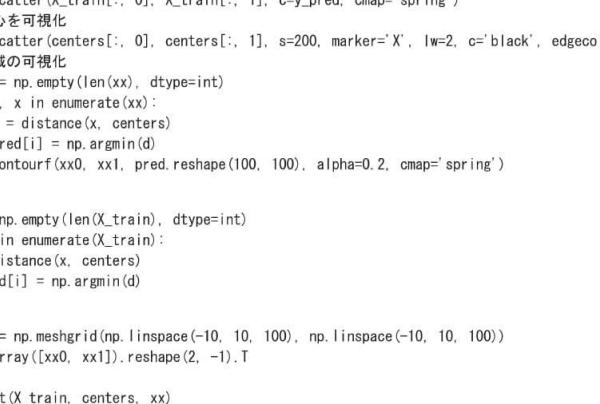
def plt_result(X_train, centers, xx):
    # データを可視化
    plt.scatter(X_train[:, 0], X_train[:, 1], c=y_pred, cmap='spring')
    # 中心を可視化
    plt.scatter(centers[:, 0], centers[:, 1], s=200, marker='X', lw=2, c='black', edgecolor="white")
    # 領域の可視化
    pred = np.empty(len(xx), dtype=int)
    for i, x in enumerate(xx):
        d = distance(x, centers)
        pred[i] = np.argmin(d)
    plt.contourf(xx0, xx1, pred.reshape(100, 100), alpha=0.2, cmap='spring')

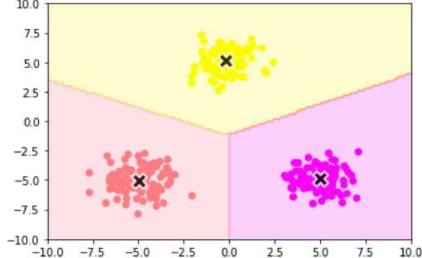
y_pred = np.empty(len(X_train), dtype=int)
for i, x in enumerate(X_train):
    d = distance(x, centers)
    y_pred[i] = np.argmin(d)

xx0, xx1 = np.meshgrid(np.linspace(-10, 10, 100), np.linspace(-10, 10, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

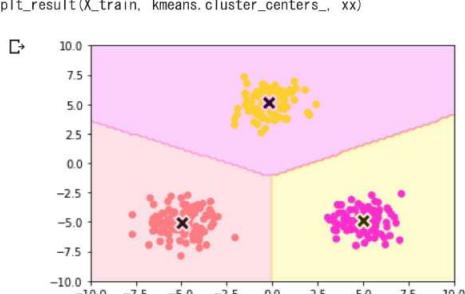
plt_result(X_train, centers, xx)

```





▼ numpy実装



6. サポートベクターマシン

- 2 クラス分類のための機械学習手法
 - 線形モデルの正負で 2 値に分類

- 線形判別関数と、最も期待データ点との距離を「マージン」

⇒ マージンが最大となる線形判別関数を求める。

- 目的関数の導出（準備）

$$\begin{aligned}\rho(w, b) &= \min_{x \in C_{t_i}=+1} \frac{w^T x_i}{\|w\|} - \max_{x \in C_{t_i}=-1} \frac{w^T x_i}{\|w\|} \\ &= \frac{1-b}{\|w\|} - \frac{-1-b}{\|w\|} \\ &= \frac{2}{\|w\|} \Rightarrow w \text{が決まればマージンが決まる。}\end{aligned}$$

- 各クラスのデータを w の方向 k へ射影した点 w 軸に座標を変換

$$w^T s + b$$

- 線形判別関数のマージンを k としたときにすべてのデータ点で成り立つ条件は

$$|w^T s + b| \geq k$$

- 各点と決定境界との距離は、点と直線と距離の公式から

$$\frac{|w^T x_i + b|}{\|w\|} = \frac{t_i(w^T x_i + b)}{\|w\|}$$

- マージンとは決定境界と最も距離の近い点との距離なので

$$\min_i \frac{t_i(w^T x_i + b)}{\|w\|}$$

- SVMはマージンを最大化することを目標なので目的関数は

$$\max_{w,b} [\min_i \frac{t_i(w^T x_i + b)}{\|w\|}]$$

- 式（主問題）

$$\min_{w,b} \frac{1}{2} \|w\|^2$$

$$t_i(w^T s_i + b) \geq 1 (i = 1, 2, \dots, n)$$

これをラグランジュ未定乗数法で解く

- ラグランジュ未定乗数法

- 制約付き最適化問題を解くための手法

$$\begin{aligned}L(w, b, a) &= \frac{1}{2} \|w\|^2 - \sum_{i=1}^n a_i (t_i(w^T x_i + b) - 1) \\ * a_i &\geq 0 (i = 1, 2, \dots, n) : \text{ラグランジュ乗数}\end{aligned}$$

最小となる w, b は次を満たす

$$\begin{aligned}\frac{\partial L}{\partial w} &= w - \sum_{i=1}^n a_i t_i x_i = 0 \\ \frac{\partial L}{\partial b} &= - \sum_{i=1}^n a_i t_i = 0\end{aligned}$$

⇒

$$w = \sum_{i=1}^n a_i t_i x_i$$

$$\sum_{i=1}^n a_i t_i = 0$$

- 双対問題

$$\begin{aligned}\max_a \sum_{t=1}^n a_t - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n a_i a_j t_i t_j x_i^T x_j \\ \sum_{i=1}^n a_i t_i = 0 \\ a_i \geq 0 (i = 1, 2, \dots, n)\end{aligned}$$

- 主問題と双対問題の最適解は1対1

- SVMの決定関数

$$y(x) = w^T x + b = \sum_{i=1}^n a_i t_i x^T x + b$$

$t_i(w^T x_i + b - 1) > 0$ の時、つまりマージンの外側のデータでは $a_i = 0$ となり予測に影響を与えない

$t_i(w^T x_i + b - 1) = 0$ の時、つまりマージン上のデータでは $a_i > 0$ となり予測に影響をあたえる（サポートベクター）

- サポートベクター

- 分離超平面を構成する学習データは、サポートベクターだけで残りのデータは不要。

実装演習（サポートベクターマシン）

▼ サポートベクターマシン(SVM)

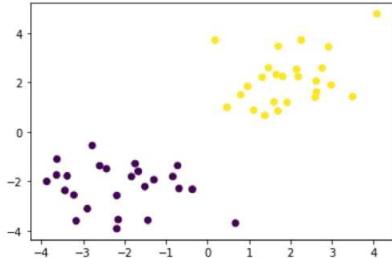
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

▼ 訓練データ生成①（線形分離可能）

```
def gen_data():
    x0 = np.random.normal(size=50).reshape(-1, 2) - 2.
    x1 = np.random.normal(size=50).reshape(-1, 2) + 2.
    X_train = np.concatenate([x0, x1])
    ys_train = np.concatenate([np.zeros(25), np.ones(25)]).astype(np.int)
    return X_train, ys_train
```

```
X_train, ys_train = gen_data()
plt.scatter(X_train[:, 0], X_train[:, 1], c=ys_train)
```

↳ <matplotlib.collections.PathCollection at 0x7f341ceda150>



▼ 学習

特徴空間上で線形なモデル $y(\mathbf{x}) = \mathbf{w}\phi(\mathbf{x}) + b$ を用い、その正負によって2値分類を行うことを考える。

サポートベクターマシンではマージンの最大化を行うが、それは結局以下の最適化問題を解くことと同じである。

ただし、訓練データを $X = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n]^T, \mathbf{t} = [t_1, t_2, \dots, t_n]^T (t_i = \{-1, +1\})$ とする。

$$\min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2$$

$$\text{subject to } t_i(\mathbf{w}\phi(\mathbf{x}_i) + b) \geq 1 \quad (i = 1, 2, \dots, n)$$

ラグランジュ乗数法を使うと、上の最適化問題はラグランジュ乗数 $\mathbf{a} (\geq 0)$ を用いて、以下の目的関数を最小化する問題となる。

$$L(\mathbf{w}, b, \mathbf{a}) = \frac{1}{2} \|\mathbf{w}\|^2 - \sum_{i=1}^n a_i t_i (\mathbf{w}\phi(\mathbf{x}_i) + b - 1) \quad (1)$$

目的関数が最小となるのは、 \mathbf{w}, b に関して偏微分した値が0となるときなので、

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^n a_i t_i \phi(\mathbf{x}_i) = \mathbf{0}$$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^n a_i t_i = \mathbf{a}^T \mathbf{t} = 0$$

これを式(1)に代入することで、最適化問題は結局以下の目的関数の最大化となる。

$$\begin{aligned} \tilde{L}(\mathbf{a}) &= \sum_{i=1}^n a_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n a_i a_j t_i t_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \\ &= \mathbf{a}^T \mathbf{1} - \frac{1}{2} \mathbf{a}^T H \mathbf{a} \end{aligned}$$

ただし、行列 H の i 行 j 列成分は $H_{ij} = t_i t_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) = t_i t_j k(\mathbf{x}_i, \mathbf{x}_j)$ である。また制約条件は、 $\mathbf{a}^T \mathbf{t} = 0 (\frac{1}{2} \|\mathbf{a}^T \mathbf{t}\|^2 = 0)$ である。

この最適化問題を最急降下法で解く。目的関数と制約条件を \mathbf{a} で微分すると、

$$\frac{d\tilde{L}}{d\mathbf{a}} = \mathbf{1} - H\mathbf{a}$$

$$\frac{d}{d\mathbf{a}} (\frac{1}{2} \|\mathbf{a}^T \mathbf{t}\|^2) = (\mathbf{a}^T \mathbf{t}) \mathbf{t}$$

なので、 \mathbf{a} を以下の二式で更新する。

$$\mathbf{a} \leftarrow \mathbf{a} + \eta_1 (\mathbf{1} - H\mathbf{a})$$

$$\mathbf{a} \leftarrow \mathbf{a} - \eta_2 (\mathbf{a}^T \mathbf{t}) \mathbf{t}$$

+ = np.where(ys_train == 1, 1, 0, -1, 0)

```

L = np.where(yb_train == 1, 0, -1, 0)

n_samples = len(X_train)
# 線形カーネル
K = X_train.dot(X_train.T)

eta1 = 0.01
eta2 = 0.001
n_iter = 500

H = np.outer(t, t) * K

a = np.ones(n_samples)
for _ in range(n_iter):
    grad = 1 - H.dot(a)

```

▼ 予測

新しいデータ点 \mathbf{x} に対しては、 $y(\mathbf{x}) = \mathbf{w}\phi(\mathbf{x}) + b = \sum_{i=1}^n a_i t_i k(\mathbf{x}, \mathbf{x}_i) + b$ の正負によって分類する。

ここで、最適化の結果得られた a_i ($i = 1, 2, \dots, n$) の中で $a_i = 0$ に対応するデータ点は予測に影響を与えないで、 $a_i > 0$ に対応するデータ点は正しく予測される。逆に $a_i < 0$ のデータ点は予測を誤らせる。

タ点 (サポートベクトル) のみ保持しておく。 b はサポートベクトルのインデックスの集合を S とすると、 $b = \frac{1}{n} \sum_{s \in S} (t_s - \sum_{i=1}^n a_i t_i k(x_s, x_i))$ によって求める。

$b = \frac{1}{S} \sum_{s \in S} (t_s - \sum_{i=1}^n a_i t_i k(\mathbf{x}_i, \mathbf{x}_s))$ によって求める。

```

index = a > 1e-6
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]

term2 = K[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()

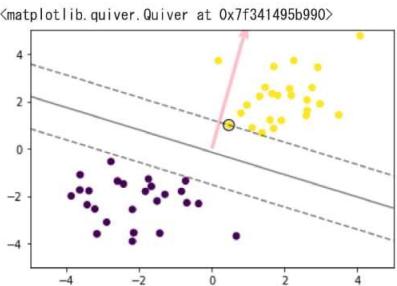
xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

X_test = xx
y_project = np.ones(len(X_test)) * b
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * sv.dot(X_test[i])
y_pred = np.sign(y_project)

# 訓練データを可視化
plt.scatter(X_train[:, 0], X_train[:, 1], c=ys_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1],
            s=100, facecolors='none', edgecolors='k')
# 領域を可視化
# plt.contourf(xx0, xx1, y_pred.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))
# マージンと決定境界を可視化
plt.contour(xx0, xx1, y_project.reshape(100, 100), colors='k',
            levels=[-1, 0, 1], alpha=0.5, linestyles=['-', '--', '-'])

# マージンと決定境界を可視化
plt.quiver(0, 0, 0.1, 0.35, width=0.01, scale=1, color='pink')

```



▼ 訓練データ生成②（線形分離不可能）

```

factor = .2
n_samples = 50
linspace = np.linspace(0, 2 * np.pi, n_samples // 2 + 1)[-1]
outer_circ_x = np.cos(linspace)
outer_circ_y = np.sin(linspace)
inner_circ_x = outer_circ_x * factor
inner_circ_y = outer_circ_y * factor

X = np.vstack((np.append(outer_circ_x, inner_circ_x),
               np.append(outer_circ_y, inner_circ_y))).T
y = np.hstack([np.zeros(n_samples // 2, dtype=np.intp),
               np.ones(n_samples // 2, dtype=np.intp)])
X += np.random.normal(scale=0.15, size=X.shape)
x_train = X
y_train = y

plt.scatter(x_train[:,0], x_train[:,1], c=y_train)

<matplotlib.collections.PathCollection at 0x7f341470fed0>

```

▼ 学習

元のデータ空間では線形分離は出来ないが、特徴空間上で線形分離することを考える。
今回はカーネルとしてRBFカーネル（ガウシアンカーネル）を利用する。

```

def rbf(u, v):
    sigma = 0.8
    return np.exp(-0.5 * ((u - v)**2).sum() / sigma**2)

X_train = x_train
t = np.where(y_train == 1.0, 1.0, -1.0)

n_samples = len(X_train)
# RBFカーネル
K = np.zeros((n_samples, n_samples))
for i in range(n_samples):
    for j in range(n_samples):
        K[i, j] = rbf(X_train[i], X_train[j])

eta1 = 0.01
eta2 = 0.001
n_iter = 5000

H = np.outer(t, t) * K

a = np.ones(n_samples)
for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.where(a > 0, a, 0)

```

▼ 予測

```

index = a > 1e-6
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]

term2 = K[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()

xx0, xx1 = np.meshgrid(np.linspace(-1.5, 1.5, 100), np.linspace(-1.5, 1.5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

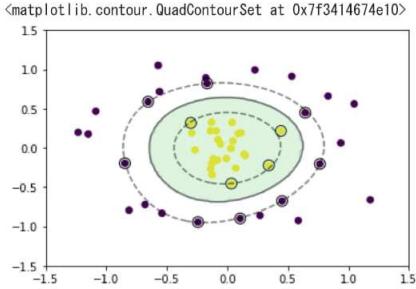
```

```

X_test = xx
y_project = np.ones(len(X_test)) * b
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * rbf(X_test[i], sv)
y_pred = np.sign(y_project)

# 訓練データを可視化
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1], s=100, facecolors='none', edgecolors='k')
# 領域を可視化
plt.contourf(xx0, xx1, y_pred.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))
# マージンと決定境界を可視化
plt.contour(xx0, xx1, y_project.reshape(100, 100), colors='k',
            levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-.', '--'])

```



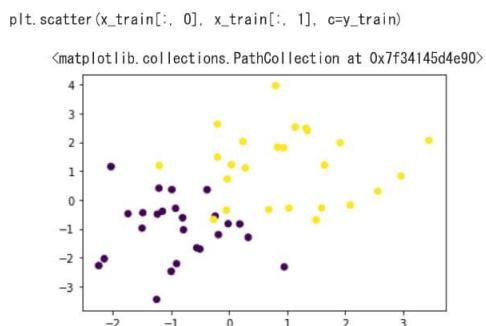
▼ ソフトマージンSVM

▼ 訓練データ生成③（重なりあり）

```

x0 = np.random.normal(size=50).reshape(-1, 2) - 1.
x1 = np.random.normal(size=50).reshape(-1, 2) + 1.
x_train = np.concatenate([x0, x1])
y_train = np.concatenate([[np.zeros(25), np.ones(25)]]).astype(np.int)

```



▼ 学習

分離不可能な場合は学習できないが、データ点がマージン内部に入ることや誤分類を許容することでその問題を回避する。

スラック変数 $\xi_i \geq 0$ を導入し、マージン内部に入ったり誤分類された点に対しては、 $\xi_i = |1 - t_i y(\mathbf{x}_i)|$ とし、これらを許容する代わりにに対して、ペナルティを与えるように、最適化問題を以下のように修正する。

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i \\ \text{subject to} \quad & t_i (\mathbf{w}^\top \phi(\mathbf{x}_i) + b) \geq 1 - \xi_i \quad (i = 1, 2, \dots, n) \end{aligned}$$

ただし、パラメータ C はマージンの大きさと誤差の許容度のトレードオフを決めるパラメータである。この最適化問題をラグランジュ乗数法などを用いると、結局最大化する目的関数はハードマージンSVMと同じとなる。

$$\tilde{L}(\mathbf{a}) = \sum_{i=1}^n a_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n a_i a_j t_i t_j \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j)$$

ただし、制約条件が $a_i \geq 0$ の代わりに $0 \leq a_i \leq C (i = 1, 2, \dots, n)$ となる。（ハードマージンSVMと同じ $\sum_{i=1}^n a_i t_i = 0$ も制約条件）

```
X_train = x_train
t = np.where(y_train == 1.0, 1.0, -1.0)

n_samples = len(X_train)
# 線形カーネル
K = X_train.dot(X_train.T)

C = 1
eta1 = 0.01
eta2 = 0.001
n_iter = 1000

H = np.outer(t, t) * K

a = np.ones(n_samples)
for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.clip(a, 0, C)
```

▼ 予測

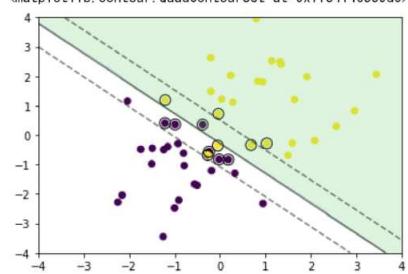
```
index = a > 1e-8
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]

term2 = K[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()

xx0, xx1 = np.meshgrid(np.linspace(-4, 4, 100), np.linspace(-4, 4, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

X_test = xx
y_project = np.ones(len(X_test)) * b
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * sv.dot(X_test[i])
y_pred = np.sign(y_project)

# 訓練データを可視化
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
# サポートベクトルを可視化
plt.scatter(support_vectors[:, 0], support_vectors[:, 1],
            s=100, facecolors='none', edgecolors='k')
# 領域を可視化
plt.contourf(xx0, xx1, y_pred.reshape(100, 100), alpha=0.2, levels=np.linspace(0, 1, 3))
# マージンと決定境界を可視化
plt.contour(xx0, xx1, y_project.reshape(100, 100), colors='k',
            levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-.', '--'])
```



終了テストについて

- ・終了テストを合格してから、機械学習のレポートをまとめていくつか気付いた点についてまとめました。

- 講義資料の表記と、出題の表記が一致していない。
こちらについては、本番の試験も表記が違う可能性があるとの認識で、細かい記述よりも、本質的に何が問われているのかを考えて解くようにしました。
- レポート範囲と、動画解説が一致していない。
こちらは事前に申し込みの際、自己学習が必要との記載がありましたので、洗礼を受けたとの認識です。
- テスト後に改めてレポートをまとめていて気付いたこと
テスト受講時には何を調べていいのかわからない問題や、設問の意味があったが、テスト合格後に改めて動画と講義資料を確認したら、かなりの設問事項について、きちんと記載してあり、確認が足りていないことを痛感した。
- 検索してもわからない問題がある
例えば1回目のパラメータ更新の後の値などは、検索してもわからないので、数式とPythonのトレースを行い、いまのところは克服できた。