機械学習の基礎

正規化

正規化の必要性

多項式最小二乗法: $f(x) = \sum_{k=0}^{n} a_k x^k$

$$\begin{pmatrix} n & \sum x_i & \sum {x_i}^2 & \cdots & \sum {x_i}^p \\ \sum x_i & \sum {x_i}^2 & \sum {x_i}^3 & & \sum {x_i}^{p+1} \\ \sum {x_i}^2 & \sum {x_i}^3 & \sum {x_i}^4 & & \sum {x_i}^{p+2} \\ \vdots & & & \ddots & \\ \sum {x_i}^p & \sum {x_i}^{p+1} & \sum {x_i}^{p+2} & & \sum {x_i}^{2p} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_p \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum y_i x_i \\ \sum y_i {x_i}^2 \\ \vdots \\ \sum y_i {x_i}^p \end{pmatrix}$$

X行列の要素は 最小次数は 定数 n、最高次数は ${x_i}^{2p}$

・データが $|x_i| >> 1$ の場合、オーバーフロー、$|x_i| << 1$ ではアンダーフロー
丸め誤差など、計算誤差が大きくなる

=> 入力データを 1 のオーダーの数値に変換する必要がある

正規化の種類

よく使う最小二乗法: 正規化はしないことが多い

64bit CPUで誤差が問題になるケースは少ない

- 高次の関数が必要な物理モデルは少ない (せいぜい6次)
- それほど高次の多項式を使うと、過適合の問題がでる
- 回帰した係数に単位を含んだ意味があることが多い (活性化エネルギー [eV] など)
  標準化すると元の係数に戻すのが面倒

機械学習: 正規化あるいは標準化はほぼ必須

比較しようのない記述子 (猫の体重、身長、年齢など) から、
単位依存性が無いように目的関数を作る必要がある

正規化 (normalization): $x_i' = \frac{x_i - x_{\min}}{x_{\max} - x_{\min}}$: データを 0 ~ 1の間に変換

ほかの範囲に変換してもいいが、計算誤差を減らすには
1のオーダーの範囲が望ましい

正規化: $x_i' = 2\frac{x_i - x_{\mathrm{mid}}}{x_{\max} - x_{\min}}$: データを –1 ~ 1の間に変換

標準化 (standardization): $x_i{}' = 2\frac{x_i - \langle x \rangle}{\sigma_x}$: 平均$\langle x \rangle$と標準偏差$\sigma_x$で変換

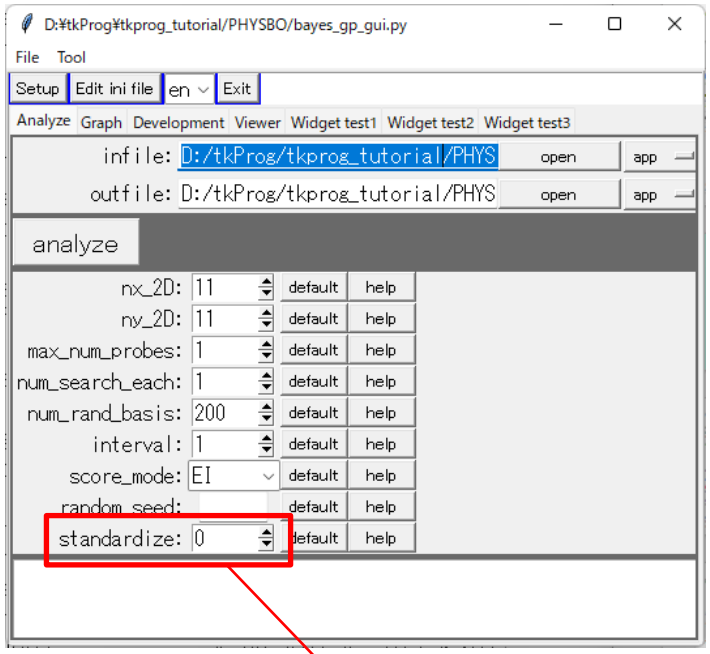
アルゴリズムによっては、平均 0、標準偏差 1 を前提としている場合がある
標準化は必須

GP回帰 (bayes_gp_gui.py) での例

入力ファイル: [tkProg]/tkprog_tutorial/PHYSBO/example.xlsx

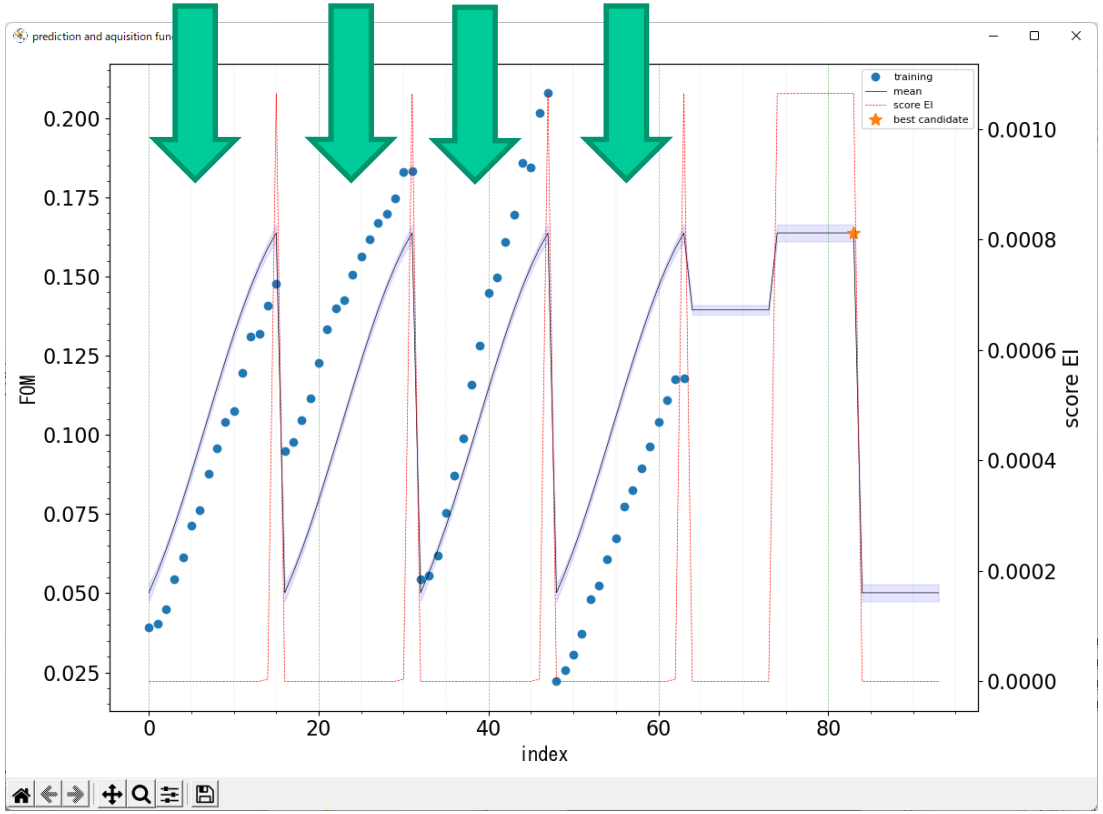
記述子: 化学組成 x、温度 T [℃]

x と T の値は 3桁違う　=> 最小二乗法では誤差の重みが 6桁変わるため、Tのフィッティングしか効かなくなる



記述子を標準化しない

異なる組成の試料の Figure-of-Merit の温度依存性

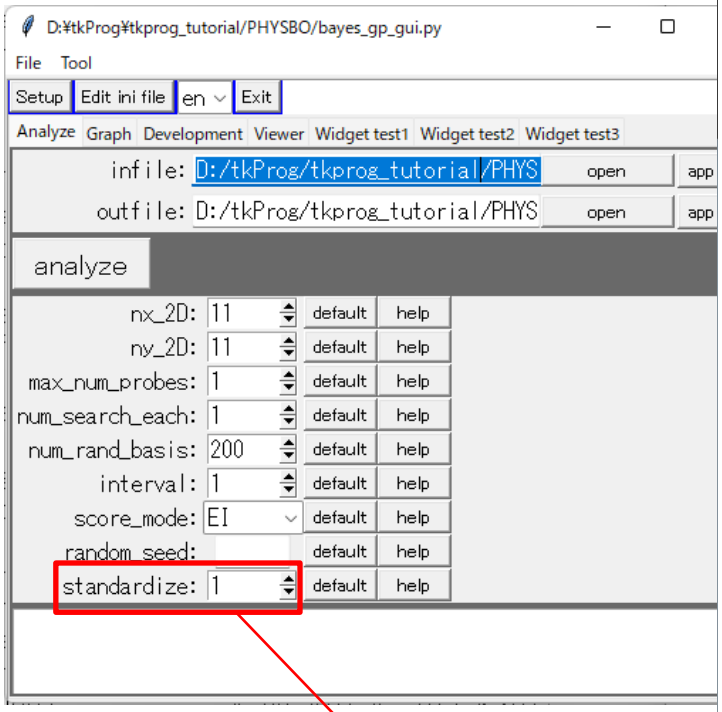


GP回帰 (bayes_gp_gui.py) での例

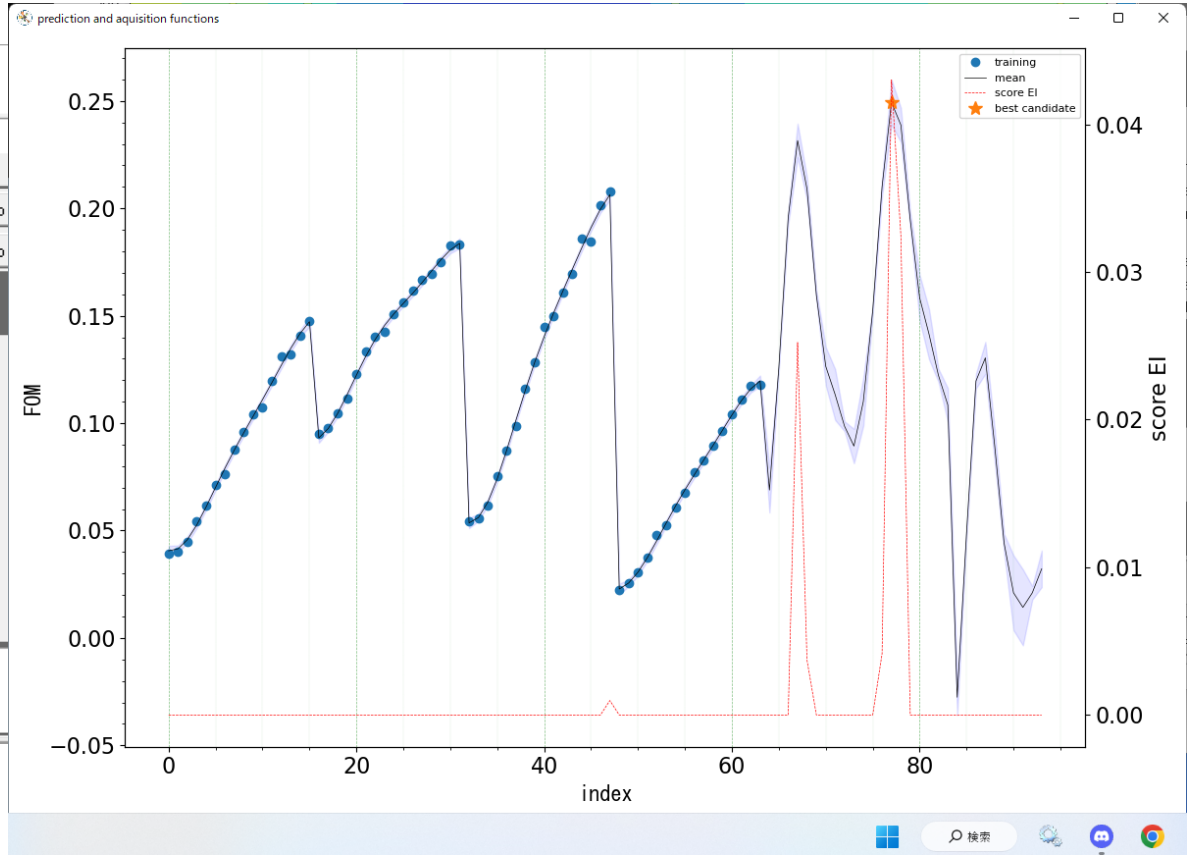
入力ファイル: [tkProg]/tkprog_tutorial/PHYSBO/example.xlsx

記述子: 化学組成 x、温度 T [℃]

x と T の値は 3桁違う => 標準化して重みをそろえる



記述子を標準化する



最小二乗法から
機械学習 (線形回帰) へ

モデル: $f(x) = a_1 x^{(1)} + a_2 x^{(2)} + \cdots + a_p x^{(p)} = \sum_{k=0}^{p} a_k x^{(k)}$

$x_j{}^{(k)}$: j番目のデータのk番目の記述子

$$\boldsymbol{j = 1, 2, \cdots, n}$$

$$\boldsymbol{k = 1, 2, \cdots, p}$$

$$y_j = f(x_j) = a_1 x_j{}^{(1)} + a_2 x_j{}^{(2)} + \cdots + a_p x_j{}^{(p)} = \sum_{k=0}^{p} a_k x_j{}^{(p)}$$

重回帰 (多変量解析): 複数変数の線形回帰

多成分解析 (複数の変数(記述子))

記述子　$\{x^{(k)}\}=(x^{(1)}, x^{(2)}, \cdots, x^{(p)})$

データ　$(\{x^{(k)}\}_1, \{x^{(i)}\}_2, \cdots, \{x^{(i)}\}_n)$

$\{x^{(k)}\}_j$の成分を $x^{(k)}{}_j$ と書く

モデル: $f(x) = a_1 x^{(1)} + a_2 x^{(2)} + \cdots + a_p x^{(p)} = \sum_{k=0}^{p} a_k x^{(k)}$

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j)^2$ を最小化

$$\frac{dS}{\mathrm{d}a_{k\prime}} = \sum_{j=1}^{n} f_{k\prime}(x_j)(\sum_{k=1}^{p} a_k x^{(k)}{}_j - y_j) = 0 \qquad k'=0, 1, \cdots, p$$

$$\sum_{j=1}^{n} (\sum_{k=1}^{p} a_k x^{(k')}{}_j x^{(k)}{}_j - y_j x^{(k')}{}_j) = 0$$

※ $\sum_{k=1}^{p} a_k \sum_{j=1}^{n} x^{(k')}{}_j x^{(k)}{}_j = \sum_{j=1}^{n} y_j x^{(k)}{}_j$

重回帰 (多変量解析): 複数変数の線形回帰

$$f(x) = a_1 x^{(1)} + a_2 x^{(2)} + \cdots + a_p x^{(p)} = \sum_{k=0}^{p} a_k x^{(k)}$$

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j)^2$ を最小化

解法: $\sum_{k=1}^{p} a_k \sum_{j=1}^{n} x^{(k')}{}_j x^{(k)}{}_j = \sum_{j=1}^{n} y_j x^{(k)}{}_j$

行列 (ベクトル) で表示: XA=Y ($(\boldsymbol{x_{k'} \cdot x_k})(a_k) = (\boldsymbol{y \cdot x_k})$) を解く

$$\boldsymbol{x_k} = \left(x^{(k)}{}_1, x^{(k)}{}_2, \cdots, x^{(k)}{}_n\right)$$

$$\boldsymbol{y} = (y_1, y_2, \cdots, y_n)$$

$$\mathrm{X} = (X_{k'k}) = \left(\sum_{j=1}^{n} x^{(k')}{}_j x^{(k)}{}_j\right) = \left(\boldsymbol{x^{(k')^T} x^{(k)}}\right)$$

$$\mathrm{A} = (A_k) = (a_k)$$

$$\mathrm{Y} = (Y_k) = \left(\sum_{j=1}^{n} y_j x^{(k)}{}_j\right) = \boldsymbol{y^T x^{(k)}}$$

機械学習の入力データ: 重回帰 (線形)

一般的な最小二乗法の入力データ random-poly.xlsx

x	y
0	4.810154
0.05	32.30261
0.1	18.78473
0.15	1.707199
0.2	15.75602

フィッティング関数 $f_i(x) = \{1, x, x^2, x^3, \cdots\}$ はプログラム中で計算する

機械学習の回帰の入力データ random-poly-ML.xlsx

x	y	x^1	x^2	x^3
0	4.810154	0	0	0
0.05	32.30261	0.05	0.0025	0.000125
0.1	18.78473	0.1	0.01	0.001
0.15	1.707199	0.15	0.0225	0.003375
0.2	15.75602	0.2	0.04	0.008

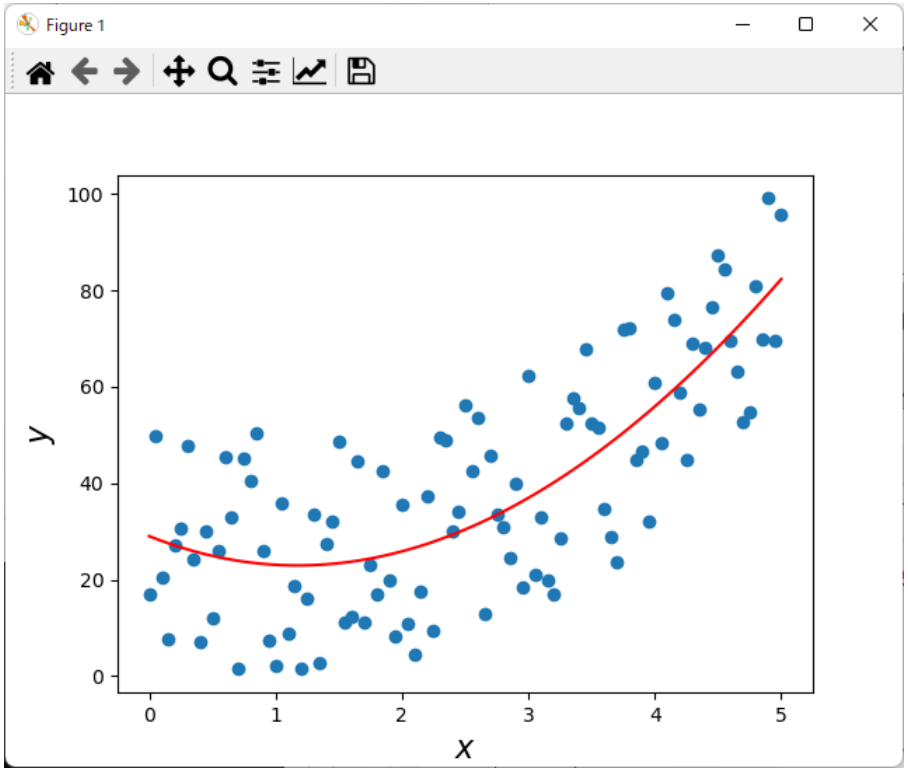
フィッティング関数 $f_i(x)$ の値は記述子として与える

LinearRegression() では定数部分は勝手に計算して intercept として返すので、
定数以外の項を記述子として与える
（x^1, x^2, x^3)

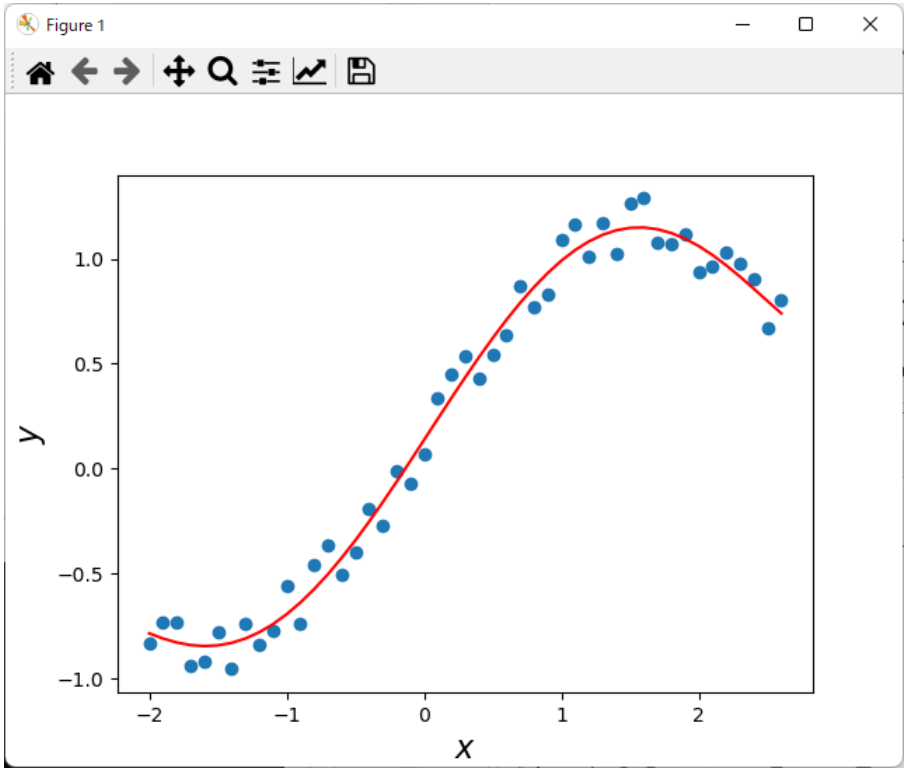
Program: lsq-polynomial-ML.py

Usage: python lsq-polynomial-ML.py input file

python lsq-polynomial-ML.py **random-poly-ML.xlsx**
3次多項式で回帰



python lsq-polynomial-ML.py **random-sin-ML.xlsx**
5次多項式で回帰



機械学習: カーネル法

任意関数の線形最小二乗法:

$f(\boldsymbol{x}) = a_1 f_1(x) + a_2 f_2(x) + \cdots + a_p f_p(x) = \sum_{k=0}^{p} a_k f_k(x)$

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k f_k(x_j) - y_j)^2$ を最小化

私たちが普段行う最小二乗法では、

$f_k(x)$には物理(科学)的意味がある

カーネル法:

$f(\boldsymbol{x}) = a_1 x^{(1)} + a_2 x^{(2)} + \cdots + a_p x^{(p)} = \sum_{k=0}^{p} a_k x^{(k)} = \boldsymbol{a} \cdot \boldsymbol{x}$

$\boldsymbol{x_{k'}} \cdot \boldsymbol{x_k} = (\mathrm{X}_{k',k})$を適当な関数 (カーネル関数) $k(\boldsymbol{x_{k'}}, \boldsymbol{x_k})$ で置き換える

$f(\boldsymbol{x}) = \sum_{k=0}^{p} a_k k(\boldsymbol{x}, \boldsymbol{x_k})$

カーネル関数に物理(科学)的な意味はなくてもいい。

必要な条件: 対称性 $k(\boldsymbol{x_{k'}}, \boldsymbol{x_k}) = k(\boldsymbol{x_k}, \boldsymbol{x_{k'}})$

正定値性 $\sum_{k,k'}^{p} a_{k'} a_k k(\boldsymbol{x_{k'}}, \boldsymbol{x_k}) \geq 0$

$(k(\boldsymbol{x_{k'}}, \boldsymbol{x_k}))(a_k) = (\boldsymbol{y} \cdot \boldsymbol{x_k})$ を解いて最小値が得られるための条件

機械学習: カーネル法

モデル (**線形回帰**): $f(x) = a_1 x^{(1)} + a_2 x^{(2)} + \cdots + a_p x^{(p)} = \sum_{k=0}^{p} a_k x^{(k)}$

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j)^2$ を最小化

解法: XA=Y ($(\boldsymbol{x}_{k'} \cdot \boldsymbol{x}_k)(a_k) = (\boldsymbol{y} \cdot \boldsymbol{x}_k)$) を解く

機械学習 (回帰、分類、クラスター分析・・・、要するに「距離」を最小化するアルゴリズム)
には頻繁に内積 $\boldsymbol{x}_{k'} \cdot \boldsymbol{x}_k = \boldsymbol{X}_{k,k'} = \sum_{j=1}^{n} x^{(k')}{}_j x^{(k)}{}_j$ (**カーネル**) **が現れる**

=> $\boldsymbol{x}_{k'} \cdot \boldsymbol{x}_k$を適当な非線形な関数で置き換えれば、
線形回帰で非線形問題を解ける

カーネル法:

$\boldsymbol{x}_{k'} \cdot \boldsymbol{x}_k = \mathrm{X}_{k,k'}$ (記述子のベクトルの内積) を
他の関数で置き換える

$\boldsymbol{x}_{k'} \cdot \boldsymbol{x}_k$ => $k(\boldsymbol{x}_{k'}, \boldsymbol{x}_k)$: カーネル関数 ($p \times p$正方行列)

カーネル ＝ 波動関数の基底関数

カーネル法:

$x_{k'} \cdot x_k = \mathrm{X}_{k,k'}$ (記述子のベクトルの内積) を

“なんでもいいから適当な” 他の関数で置き換える

$x_{k'} \cdot x_k \Rightarrow k(x_{k'}, x_k)$: カーネル関数 ($p \times p$正方行列)

思い出しましょう:

機械学習では、予測性能が確認されれば、中身がわからなくても、どのようないいかげんな手順で得たモデルでも利用できる

他の例:

量子化学では分子や結晶の波動関数を展開する「基底関数」は何でもいい

- 原子の波動関数
- 平面波
- 球面波
- **ガウス基底 (GTO)**
  - **=> データの分布に対してもガウス基底を使ってもいいでしょう？**
- スレーター基底 (STO)

カーネル関数の例

ガウシアンカーネル: $k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}) = \exp\left(-\frac{|\boldsymbol{x_k}-\boldsymbol{x_{k'}}|^2}{2\sigma^2}\right)$

多項式カーネル　　: $k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}) = (1 + \boldsymbol{x_k} \cdot \boldsymbol{x_{k'}})^p$

$\boldsymbol{x_{k'}}$を定数 (ハイパーパラメータ) $\boldsymbol{x_{c,i}}$と書き換えるとわかりやすい

例えば: $k(\boldsymbol{x_k}, \boldsymbol{x_{c,i}}) = \exp\left(-\frac{|\boldsymbol{x_k}-\boldsymbol{x_{c,i}}|^2}{2\sigma^2}\right)$

“記述子空間”のデータ点 $\boldsymbol{x_c}^{(k)} = (x_1{}^{(k)}, x_2{}^{(k)}, \cdots, x_p{}^{(k)})$
一つずつにガウス関数を置いた関数

カーネル回帰: 行列 $\mathrm{K} = (K_{k,k'}) = (k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}))$ に対して

$\mathbf{y} = f(\boldsymbol{x_k}) = \sum_{k=1}^{p} a_k\, k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}) = \mathrm{K}\boldsymbol{A}$

$S = (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y}) \cdot (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y}) = (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y})^T (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y})$を$A$の成分について最小化

$\Rightarrow \mathrm{K}^T(\mathrm{K}\boldsymbol{A} - \boldsymbol{Y}) = \boldsymbol{0}$ を解けばよい

Kは正定値行列なので $\boldsymbol{A} = K^{-1}\boldsymbol{Y}$ と、カーネルの逆行列だけで解ける

線形最小二乗法 vs. カーネル法

任意関数の線形最小二乗法:

$f(\boldsymbol{x}) = a_1 f_1(x) + a_2 f_2(x) + \cdots + a_p f_p(x) = \sum_{k=0}^{p} a_k f_k(x)$

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k f_k(x_j) - y_j)^2$ を最小化

私たちが普段行う最小二乗法では、

$f_k(x)$には物理(科学)的意味がある

カーネル法:

$f(\boldsymbol{x}) = a_1 x^{(1)} + a_2 x^{(2)} + \cdots + a_p x^{(p)} = \sum_{k=0}^{p} a_k x^{(k)} = \boldsymbol{a} \cdot \boldsymbol{x}$

$\boldsymbol{x_{k'}} \cdot \boldsymbol{x_k} = (\mathrm{X}_{k',k})$を適当な関数 (カーネル関数) $k(\boldsymbol{x_{k'}}, \boldsymbol{x_k})$ で置き換える

$f(\boldsymbol{x}) = \sum_{k=0}^{p} a_k k(\boldsymbol{x}, \boldsymbol{x_k})$

カーネル関数に物理(科学)的な意味はなくてもいい。

必要な条件: 対称性 $k(\boldsymbol{x_{k'}}, \boldsymbol{x_k}) = k(\boldsymbol{x_k}, \boldsymbol{x_{k'}})$

正定値性 $\sum_{k,k'}^{p} a_{k'} a_k k(\boldsymbol{x_{k'}}, \boldsymbol{x_k}) \geq 0$

$(k(\boldsymbol{x_{k'}}, \boldsymbol{x_k}))(a_k) = (\boldsymbol{y} \cdot \boldsymbol{x_k})$ を解いて最小値が得られるための条件

任意関数の線形最小二乗法 ～ カーネル回帰

任意関数 $f_k(x)$ の線形最小二乗法はカーネル法と(ほぼ)同じ

モデル: $y_j = \sum_{k=1}^{p} a_k f_k(x_j)$　　$\boldsymbol{Y} = \mathrm{F}\boldsymbol{A}$

解法: $\sum_{k=1}^{p} a_k \sum_{j=0}^{n} f_{k'}(x_j) f_k(x_j) = \sum_{j=0}^{n} y_j f_k(x_j)$

$$\begin{pmatrix} \sum f_1(x_i)f_1(x_i) & \sum f_1(x_i)f_2(x_i) & \sum f_1(x_i)f_3(x_i) & \cdots & \sum f_1(x_i)f_p(x_i) \\ \sum f_2(x_i)f_1(x_i) & \sum f_2(x_i)f_2(x_i) & \sum f_2(x_i)f_3(x_i) & & \sum f_2(x_i)f_p(x_i) \\ \sum f_3(x_i)f_1(x_i) & \sum f_3(x_i)f_2(x_i) & \sum f_3(x_i)f_3(x_i) & & \sum f_3(x_i)f_p(x_i) \\ \vdots & & & \ddots & \\ \sum f_p(x_i)f_1(x_i) & \sum f_p(x_i)f_2(x_i) & \sum f_N(x_i)f_3(x_i) & & \sum f_p(x_i)f_p(x_i) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_p \end{pmatrix} = \begin{pmatrix} \sum y_i f_1(x_i) \\ \sum y_i f_2(x_i) \\ \sum y_i f_3(x_i) \\ \vdots \\ \sum y_i f_p(x_i) \end{pmatrix}$$

$$X_{k,k'} = \sum_{j=1}^{n} f_{k'}(x_j) f_k(x_j) \qquad A_k = a_k$$

$$Y_k = \sum_{j=1}^{n} y_j f_k(x_j)$$

$\mathrm{X}\boldsymbol{A}=\boldsymbol{Y}$ を解く　$\boldsymbol{A}=\mathrm{X}^{-1}\boldsymbol{Y}$

カーネル法: $f_k(x_j)$ をハイパーパラメータ X_k をいれて

カーネル関数 $k(x_j, X_k)$ で置き換える

$\boldsymbol{Y} = \mathrm{K}\boldsymbol{A}$　=> $\boldsymbol{A} = \mathrm{K}^{-1}\boldsymbol{Y}$ を解く

つまり、カーネル回帰　＝ $f_k(x)$ にある関係 (カーネルの対称性・正定値性) がある

任意関数の線形最小二乗法

カーネル回帰とカーネル関数の例

最少化問題の解: $\mathrm{K} = (k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}))$ に対して

$\mathbf{y} = f(\boldsymbol{x_k}) = \sum_{k'=1}^{p} a_{k'}\, k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}) = \mathrm{K}\boldsymbol{A}$ として y を表せる

証明: $S = (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y}) \cdot (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y}) = (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y})^T (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y})$を$A$の成分について最小化

$=> \mathrm{K}^T(\mathrm{K}\boldsymbol{A} - \boldsymbol{Y}) = \boldsymbol{0}$ を解けばよい

Kは正定値行列なので $\mathrm{K}\boldsymbol{A} = \boldsymbol{Y}$

$\boldsymbol{A} = K^{-1}\boldsymbol{Y}$ と、カーネルの逆行列だけで $\boldsymbol{A}$ が求められる

カーネル関数の例:

ガウシアンカーネル: $k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}) = \exp\left(-\frac{|x_k - x_{k'}|^2}{2\sigma^2}\right)$

多項式カーネル　　: $k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}) = (1 + \boldsymbol{x_k} \cdot \boldsymbol{x_{k'}})^p$

$\boldsymbol{x_{k'}}$を定数 (ハイパーパラメータ) $\boldsymbol{x_{c,i}}$と書き換えるとわかりやすい

例えば: $k(\boldsymbol{x_k}, \boldsymbol{x_{c,i}}) = \exp\left(-\frac{|x_k - x_{c,i}|^2}{2\sigma^2}\right)$

“記述子空間”のデータ点 $\boldsymbol{x_c}^{(k)} = (x_1{}^{(k)}, x_2{}^{(k)}, \cdots, x_p{}^{(k)})$ 一つずつに

ガウス関数を置いた関数

カーネル法: 行列表示

$$\mathbf{y} = f(\boldsymbol{x_k}) = \sum_{k=1}^{p} a_k \, k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}) = \mathrm{K}\boldsymbol{A}$$

$\boldsymbol{Y} = \mathrm{K}\boldsymbol{A}$を解く

$$(\boldsymbol{x_k}) = \left({x_1}^{(k)}, {x_2}^{(k)}, \cdots, {x_n}^{(k)}\right)$$

${x_j}^{(k)}$: j番目のデータのk番目の記述子

$$\begin{pmatrix} k(\boldsymbol{x_1}, \boldsymbol{x_1}) & k(\boldsymbol{x_1}, \boldsymbol{x_2}) & k(\boldsymbol{x_1}, \boldsymbol{x_3}) & \cdots & k(\boldsymbol{x_1}, \boldsymbol{x_p}) \\ k(\boldsymbol{x_2}, \boldsymbol{x_1}) & k(\boldsymbol{x_2}, \boldsymbol{x_2}) & k(\boldsymbol{x_2}, \boldsymbol{x_3}) & & k(\boldsymbol{x_2}, \boldsymbol{x_p}) \\ k(\boldsymbol{x_3}, \boldsymbol{x_1}) & k(\boldsymbol{x_2}, \boldsymbol{x_2}) & k(\boldsymbol{x_3}, \boldsymbol{x_3}) & & k(\boldsymbol{x_3}, \boldsymbol{x_p}) \\ \vdots & & & \ddots & \\ k(\boldsymbol{x_p}, \boldsymbol{x_1}) & k(\boldsymbol{x_p}, \boldsymbol{x_2}) & k(\boldsymbol{x_p}, \boldsymbol{x_3}) & & k(\boldsymbol{x_p}, \boldsymbol{x_p}) \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_p \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_p \end{pmatrix}$$

注意: 回帰するデータ数 n は関数の数 p に等しくなければいけない

変数の数 ＝ 方程式の数

=> データ点の再現はできるが、それ以外の点の予測性が悪くなる

カーネル法: 1次元 $y = f(x)$ への回帰

データ点 x_j 一つづつにカーネル関数を置き、データ点自身を

を記述子とみなす

(“記述子”ではなく、”基底関数をx_j に置く” の方がわかりやすい)

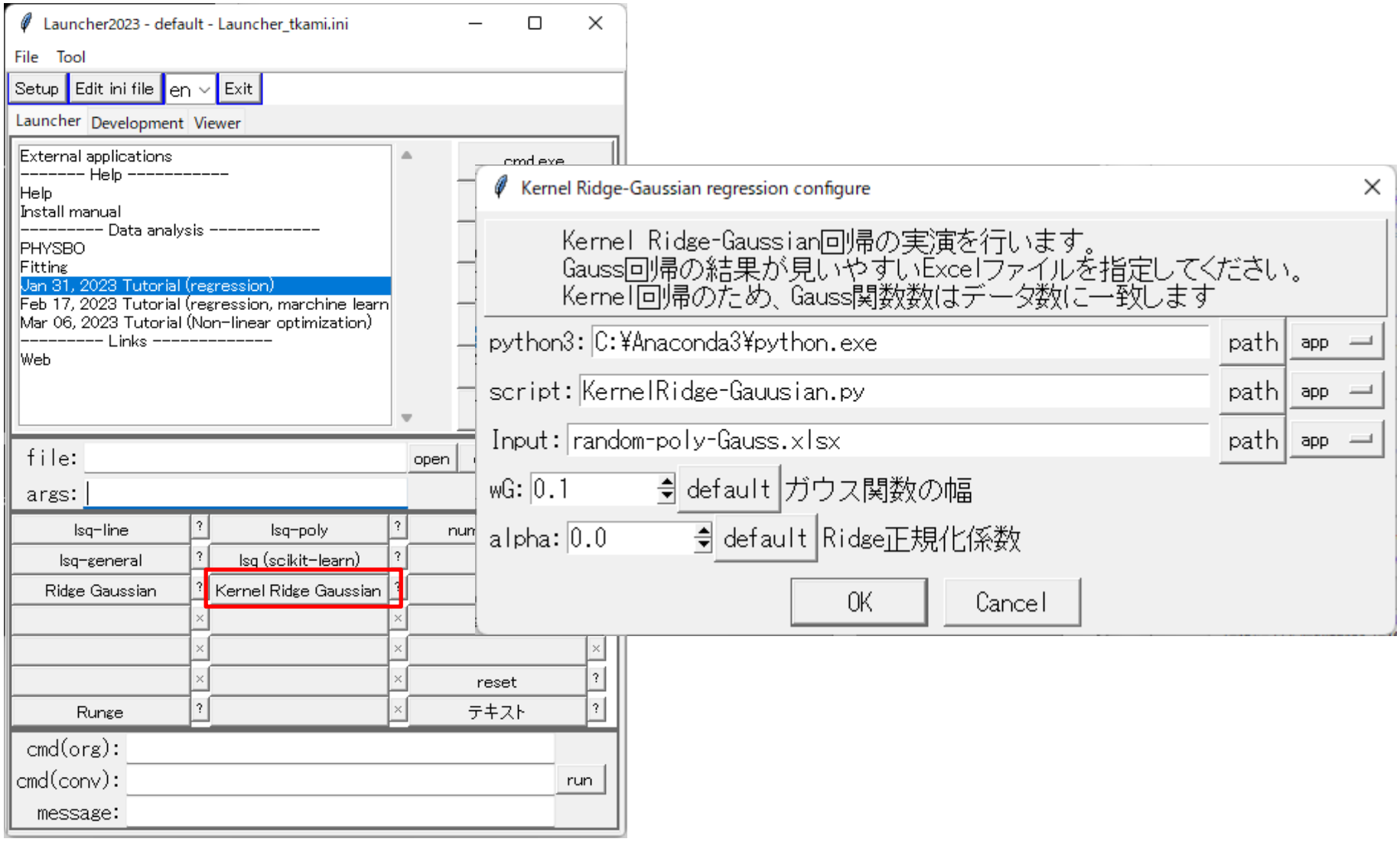
$$y_j = f(x_j) = \sum_{k=1}^{p} a_k \, k(x_j, x_k) \qquad j = 1,2,\cdots,n = p$$

ガウシアンカーネルの例:

$$\left(\exp\left(-\frac{(x_j - x_{j'})^2}{2\sigma^2}\right)\right)\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix}$$

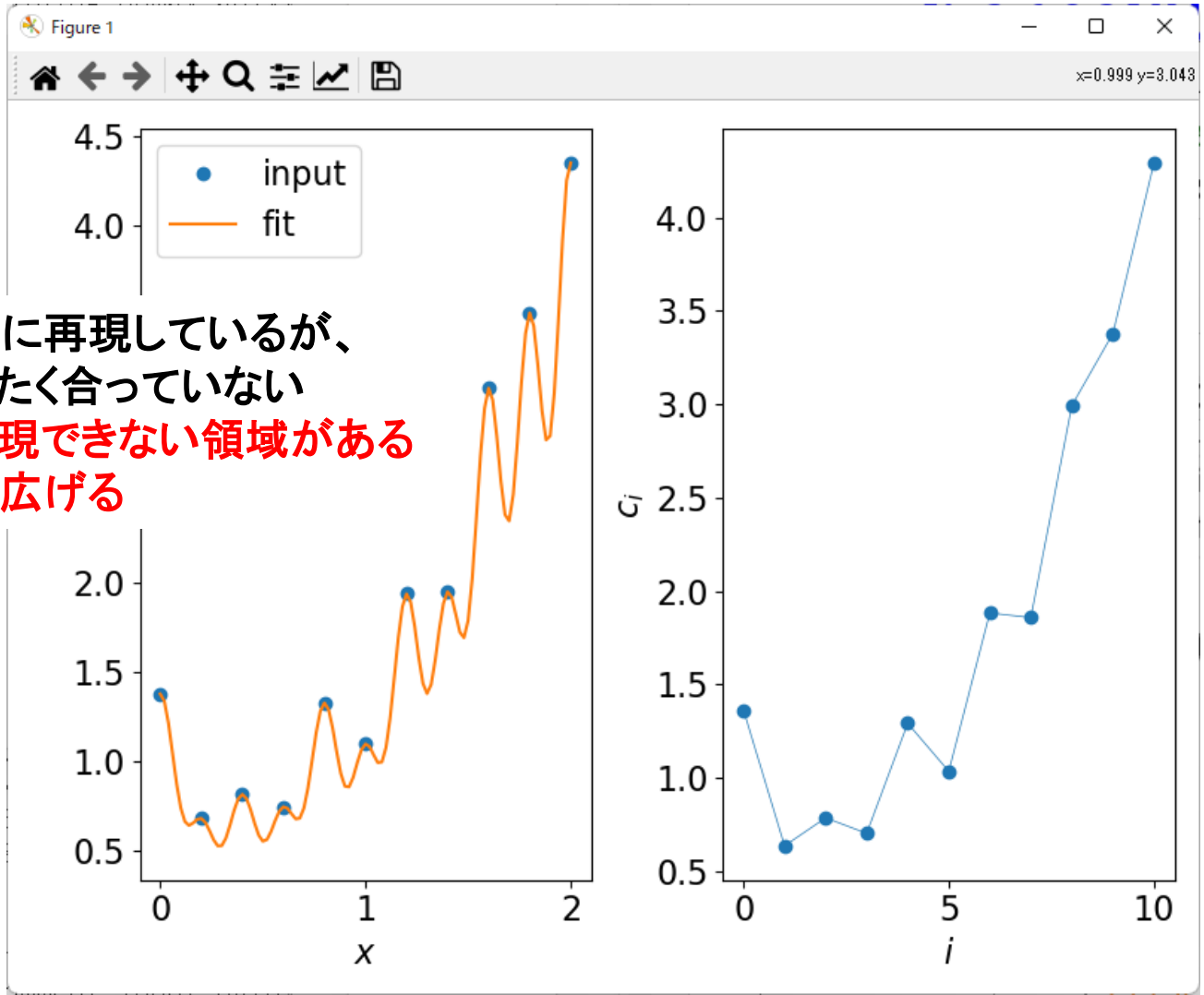
を解く。σ (あるいは $\sigma_{jj'}$) はハイパーパラメータ

Kernel回帰


Launcher2023 - default - Launcher_tkami.ini
File Tool
Setup Edit ini file en Exit
Launcher Development Viewer
External applications
-------- Help ------------
Help
Install manual
---------- Data analysis -------------
PHYSBO
Fitting
Jan 31, 2023 Tutorial (regression)
Feb 17, 2023 Tutorial (regression, marchine learn
Mar 06, 2023 Tutorial (Non-linear optimization)
---------- Links --------------
Web
cmd.exe
file: open
args:
lsq-line lsq-poly
lsq-general lsq (scikit-learn)
Ridge Gaussian Kernel Ridge Gaussian
reset
Runge テキスト
cmd(org):
cmd(conv): run
message:
Kernel Ridge-Gaussian regression configure
Kernel Ridge-Gaussian回帰の実演を行います。
Gauss回帰の結果が見いやすいExcelファイルを指定してください。
Kernel回帰のため、Gauss関数数はデータ数に一致します
python3: C:¥Anaconda3¥python.exe path app
script: KernelRidge-Gauusian.py path app
Input: random-poly-Gauss.xlsx path app
wG: 0.1 default ガウス関数の幅
alpha: 0.0 default Ridge正規化係数
OK Cancel

Kernel回帰はデータを再現できるが・・・

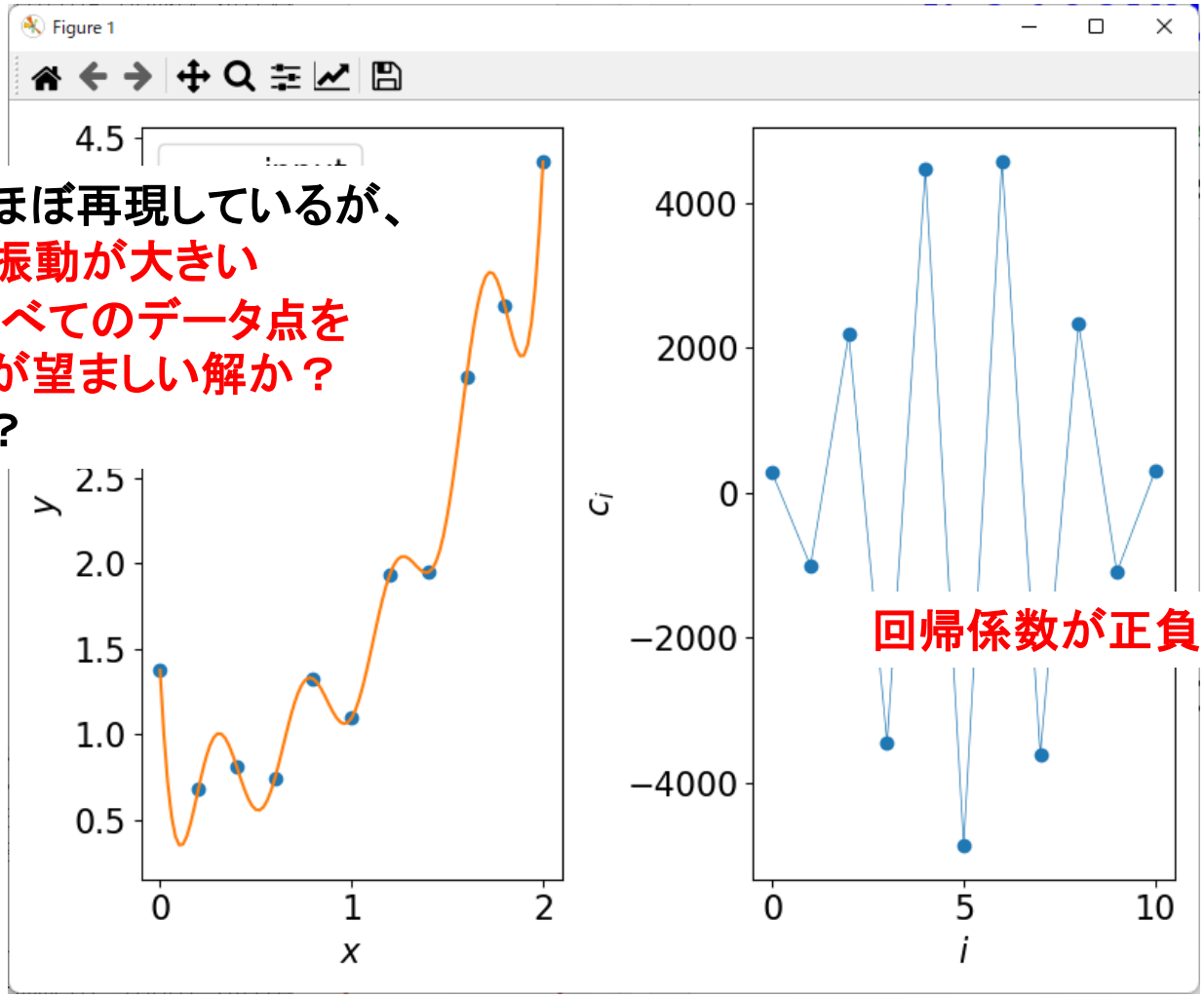
Usage: python KernelRidge-Gauusian.py infile nG wG alpha
ex: python KernelRidge-Gauusian.py random-poly-Gauss .xlsx 11 0.1 0.0
random-poly-Gauss .xlsx (データ点数11) を読み込み、11個の幅0.1のガウス関数でフィッティング。L2正規化項は入れない (Ridge解析をしない)



データ点は完全に再現しているが、
他の x ではまったく合っていない
基底関数で表現できない領域がある
=> 幅 wG を広げる

Kernel回帰はデータを再現できるが・・・

Usage: python KernelRidge-Gauusian.py infile nG wG alpha
ex: python KernelRidge-Gauusian.py random-poly-Gauss .xlsx 11 0.5 0.0
random-poly-Gauss .xlsx (データ点数11) を読み込み、11個の幅0.5のガウス関数でフィッティング。L2正規化項は入れない (Ridge解析をしない)

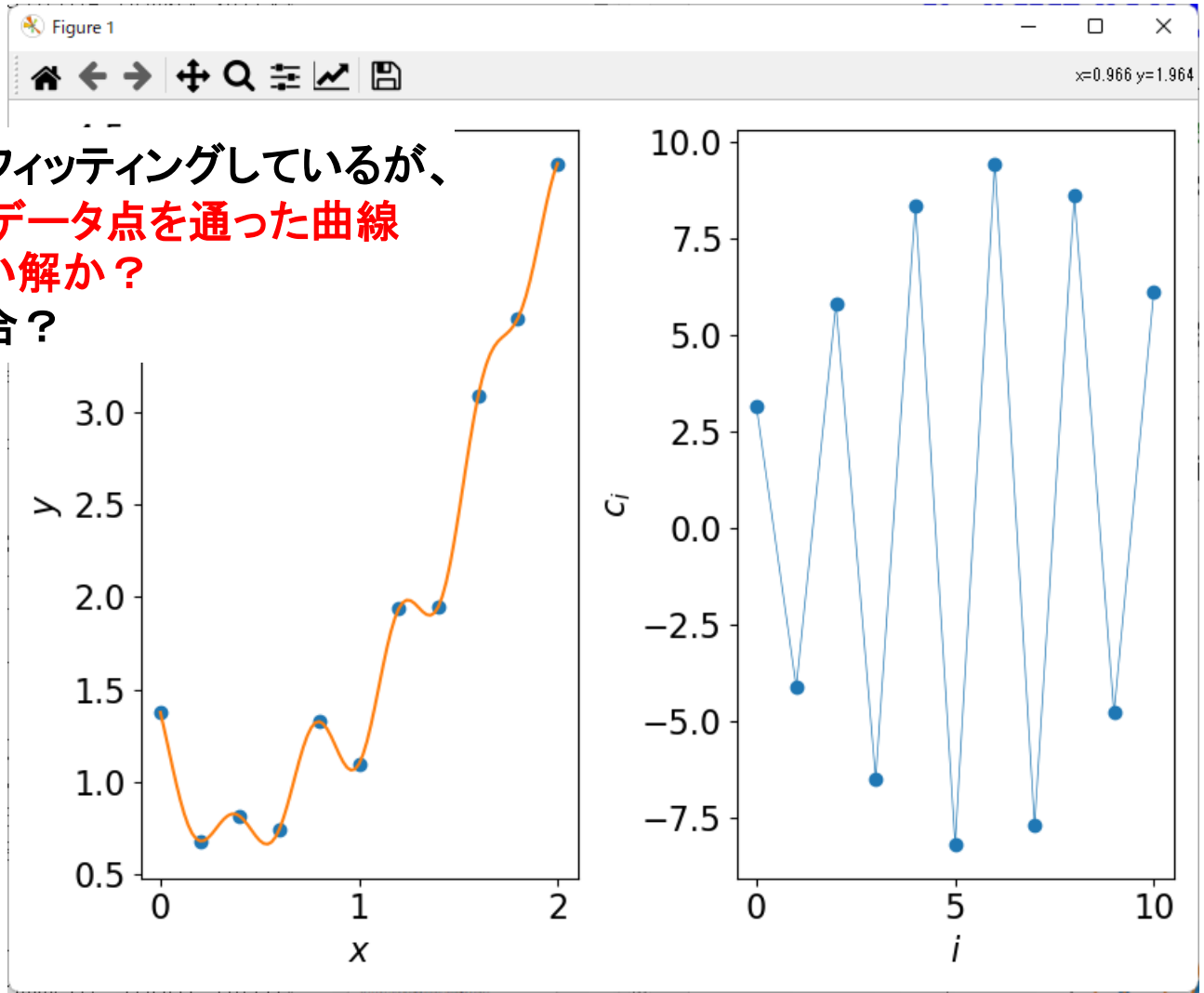


データ点はほぼ再現しているが、
他の x では振動が大きい
そもそも、すべてのデータ点を
通った曲線が望ましい解か？
　過剰適合？

回帰係数が正負に大きく振動

ハイパーパラメータで合う解は見つかるが・・・

Usage: python KernelRidge-Gauusian.py infile nG wG alpha
ex: python KernelRidge-Gauusian.py random-poly-Gauss .xlsx 11 0.3 0.0
random-poly-Gauss .xlsx (データ点数11) を読み込み、11個の幅0.3のガウス関数でフィッティング。L2正規化項は入れない (Ridge解析をしない)



きれいにフィッティングしているが、
すべてのデータ点を通った曲線が望ましい解か？
過剰適合？

共線性 (collinearity)

共線性

2つの記述子A, Bに線形的な関係がある

例：生年Aと数え年B

生年A	数え年B
2023	1
2022	2
2021	3

$$B = 2024 - A$$

多重共線性

2つ以上の記述子A, B, C, ･･･に線形的な関係がある

例：お小遣いを、テストで80点以上取ったときに100円、
大会で入賞したときに50円ずつ上げる

80点以上の回数A	入賞回数B	お小遣いC
1		100
	1	150
2		250
3		350
	2	400

$$C = 100A + 50B$$

記述子 $\alpha A + (1 - \alpha)(2024 - B)$
は α が違っても同じ

- 記述子の重みがα分不定になる
- 重みを計算する線形回帰の方程式
  $XA = Y$ で 行列 X が逆行列を持たない

正則でない X について
$X + \lambda I$ が正定値行列になるように
正数 λ を選ぶことができる: 正則化

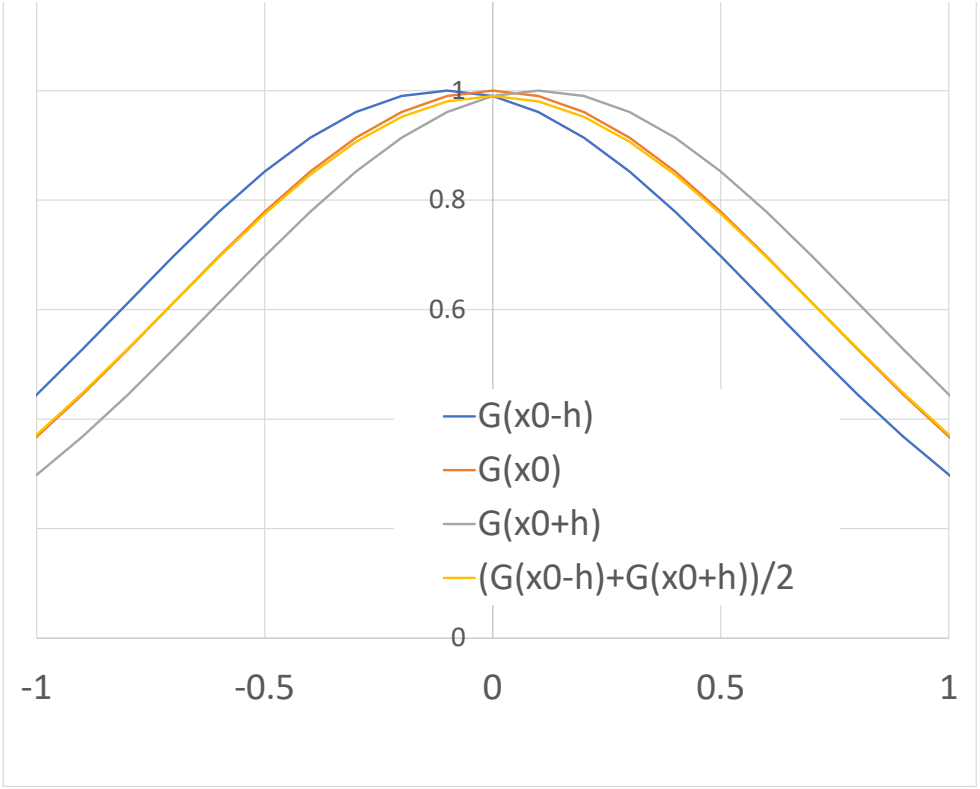
※ 正則: 素性の良い性質を持っていること。
微分可能、逆数を持つ、など

関数ベクトルの共線性

関数 $f(x_i)$: N個のデータの組 **(x_i) を N次元ベクトル**と考えると、
$f(x_i)$ の組 **$(f(x_i))$ も N次元ベクトル**として考えられる
連続関数 $f(x)$: **(x)、$f(x)$ は無限次元ベクトル**

ファイル: 関数ベクトルの共線性.xlsx
x_0 の前後 $x_0 - h$、$x_0 + h$ $(h << w)$ のガウス基底関数 (動径基底関数 RBF) $G(x)$ の平均として x_0 のRBFは近似できる

$$G(x_0) \sim (G(x_0 - h) + G(x_0 + h)) / 2$$

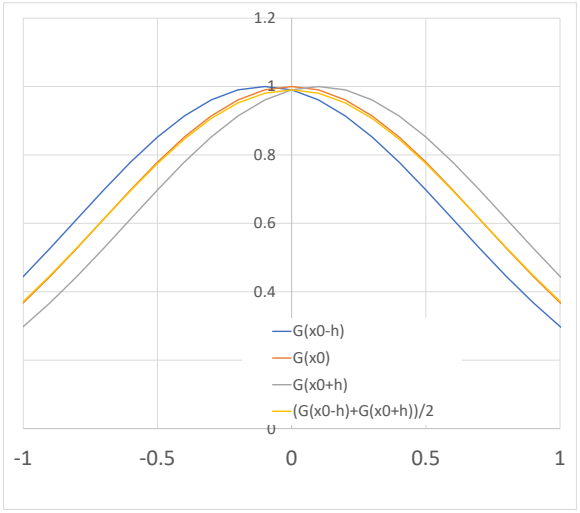


相関係数

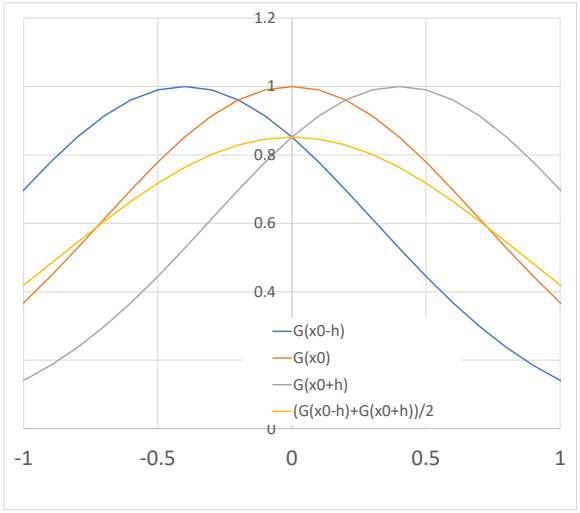
k番目の記述子ベクトル $A^{(k)}$ と $B^{(k)}$ (要素はデータのk番目の記述子の値)

相関係数: $r = \frac{A^{(k)} \cdot B^{(k)}}{|A^{(k)}||B^{(k)}|} = \cos(A^{(k)}$ と $B^{(k)}$の成す角)

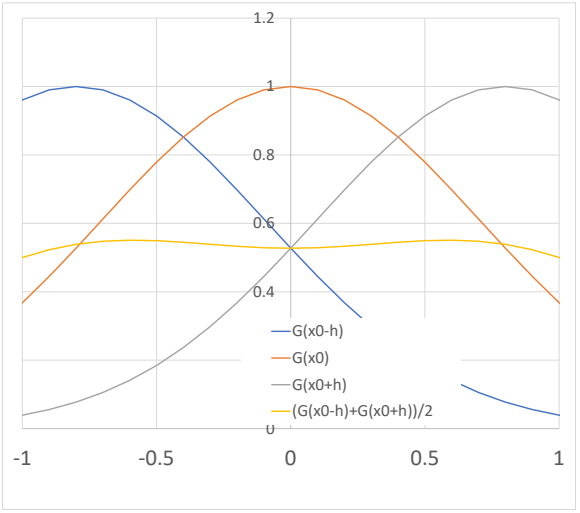
h = 0.1, w = 1.0
R = 0.980



h = 0.4, w = 1.0
R = 0.726



h = 0.8, w = 1.0
R = 0.278



相関係数の絶対値が1に近い記述子: 相関が大きい
取り除くか、正則化する必要がある
(多重共線性がある場合は相関係数からはわからない)

正則化

正則: 素性の良い性質を持っていること。微分可能、逆数を持つ、など

係数 a_i の絶対値が大きくならないようにペナルティ (正則化項) を加える

モデル: $f(x) = \sum_{k=0}^{p} a_k x^{(k)}$

LASSO回帰 (L1正則化) (Least Absolute Shrinkage and Selection Operator)

目的関数 $S = \underbrace{\sum_{j=1}^{n} (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j)^2}_{\text{誤差項}} + \underbrace{p\alpha \sum_{k=0}^{p} |a_k|}_{\text{正則化項}}$

- **適切に $\alpha \geq 0$ を選ぶと、フィッティングは悪くなるが不要な係数が0になる**

Ridge回帰 (L2正則化)

目的関数 $S = \underbrace{\sum_{j=1}^{n} (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j)^2}_{\text{誤差項}} + \underbrace{p\alpha \sum_{k=0}^{p} a_k{}^2}_{\text{正則化項}}$

- **$\alpha \geq 0$ を大きくすると、フィッティングは悪くなるが振動が抑えられる**

Elastic Net回帰 (L1正則化+ L2正則化)

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j)^2 + Kp\alpha \sum_{k=0}^{p} |a_k| + (1-K)p\alpha \sum_{k=0}^{p} a_k{}^2$

Ridge回帰

係数 a_i に制約をつける。Ridge回帰では a_i の絶対値が大きくならないように
目的関数にL2ノルムの二乗のペナルティ (正則化項) を加える

$$f(x) = \sum_{k=0}^{p} a_k x^{(k)}$$

目的関数 $S = \sum_{j=1}^{n} (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j)^2 + \boldsymbol{\lambda} \sum_{k=0}^{p} \boldsymbol{a_k}^2 \qquad \boldsymbol{\lambda \geq 0}$

$$\frac{dS}{da_{k'}} = 2 \sum_{j=1}^{n} x^{(k')}{}_j (\sum_{k=1}^{p} a_k \, x^{(k)}{}_j - y_j) + \mathbf{2}\boldsymbol{\lambda a_{k'}} = \mathbf{0}$$

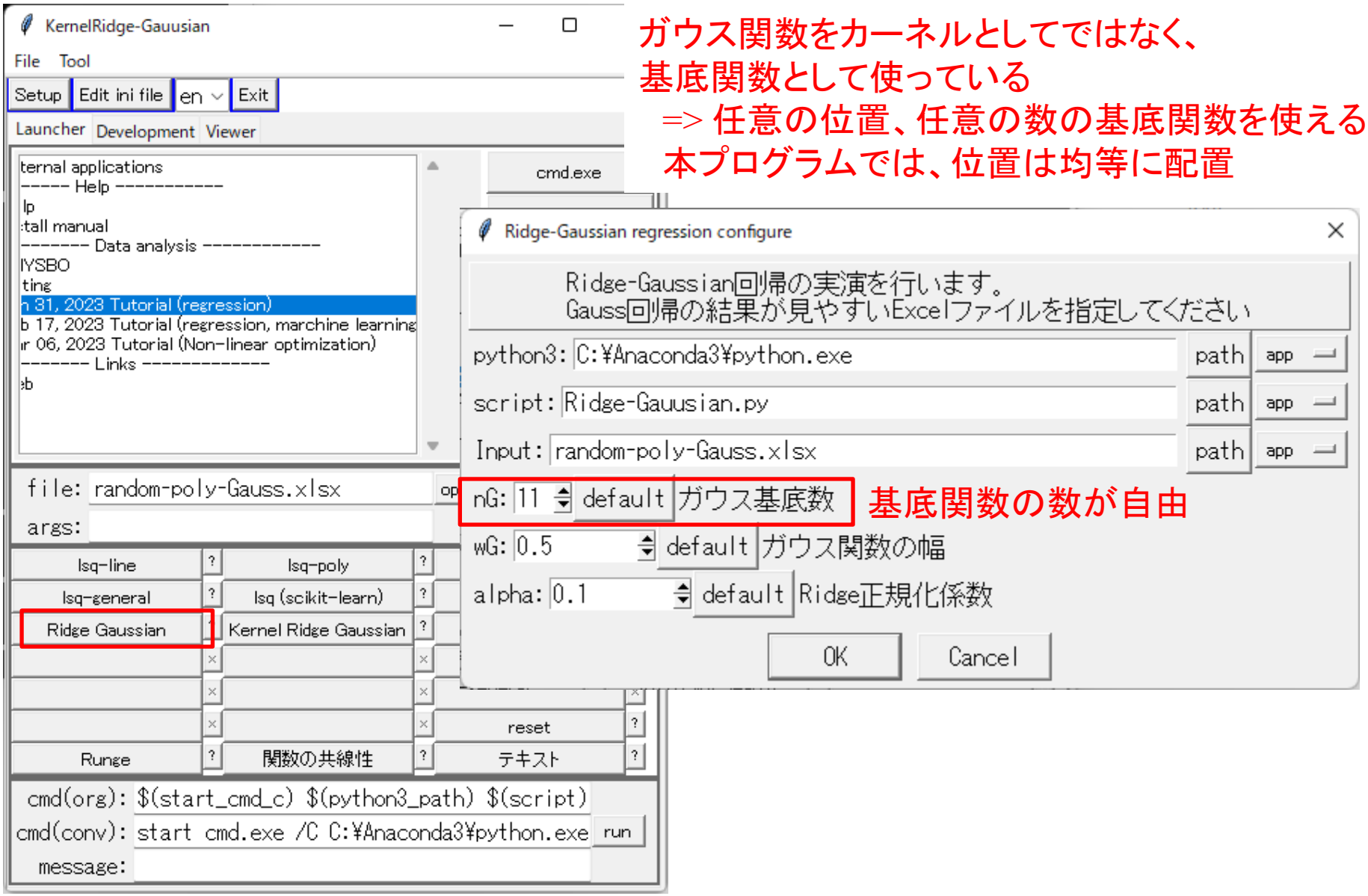
$$\sum_{k=1}^{p} a_k \sum_{j=1}^{n} x^{(k')}{}_j \, x^{(k)}{}_j + \boldsymbol{\lambda a_{k'}} = \sum_{k=1}^{p} \sum_{j=1}^{n} x^{(k')}{}_j y_j$$

$$\begin{pmatrix} \sum x^{(1)}x^{(1)} + \boldsymbol{\lambda} & \sum x^{(1)}x^{(2)} & \sum x^{(1)}x^{(3)} & \cdots & \sum x^{(1)}x^{(p)} \\ \sum x^{(2)}x^{(1)} & \sum x^{(2)}x^{(2)} + \boldsymbol{\lambda} & \sum x^{(2)}x^{(3)} & & \sum x^{(2)}x^{(p)} \\ \sum x^{(3)}x^{(1)} & \sum x^{(3)}x^{(2)} & \sum x^{(3)}x^{(3)} + \boldsymbol{\lambda} & & \sum x^{(3)}x^{(p)} \\ \vdots & & & \ddots & \\ \sum x^{(p)}x^{(1)} & \sum x^{(p)}x^{(2)} & \sum x^{(p)}x^{(3)} & & \sum x^{(p)}x^{(p)} + \boldsymbol{\lambda} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_p \end{pmatrix} = \begin{pmatrix} \sum x^{(1)}y_i \\ \sum x^{(2)}y_i \\ \sum x^{(3)}y_i \\ \vdots \\ \sum x^{(p)}y_i \end{pmatrix}$$

$$X_{k,k'} = \sum_{j=1}^{n} x^{(k)}{}_j x^{(k')}{}_j + \lambda \delta_{kk'} = \boldsymbol{x}^{(\boldsymbol{k})} \cdot \boldsymbol{x}^{(\boldsymbol{k'})} + \lambda I_p$$

XA=Y　を解く

リッジ回帰



ガウス関数をカーネルとしてではなく、
基底関数として使っている
=> 任意の位置、任意の数の基底関数を使える
本プログラムでは、位置は均等に配置

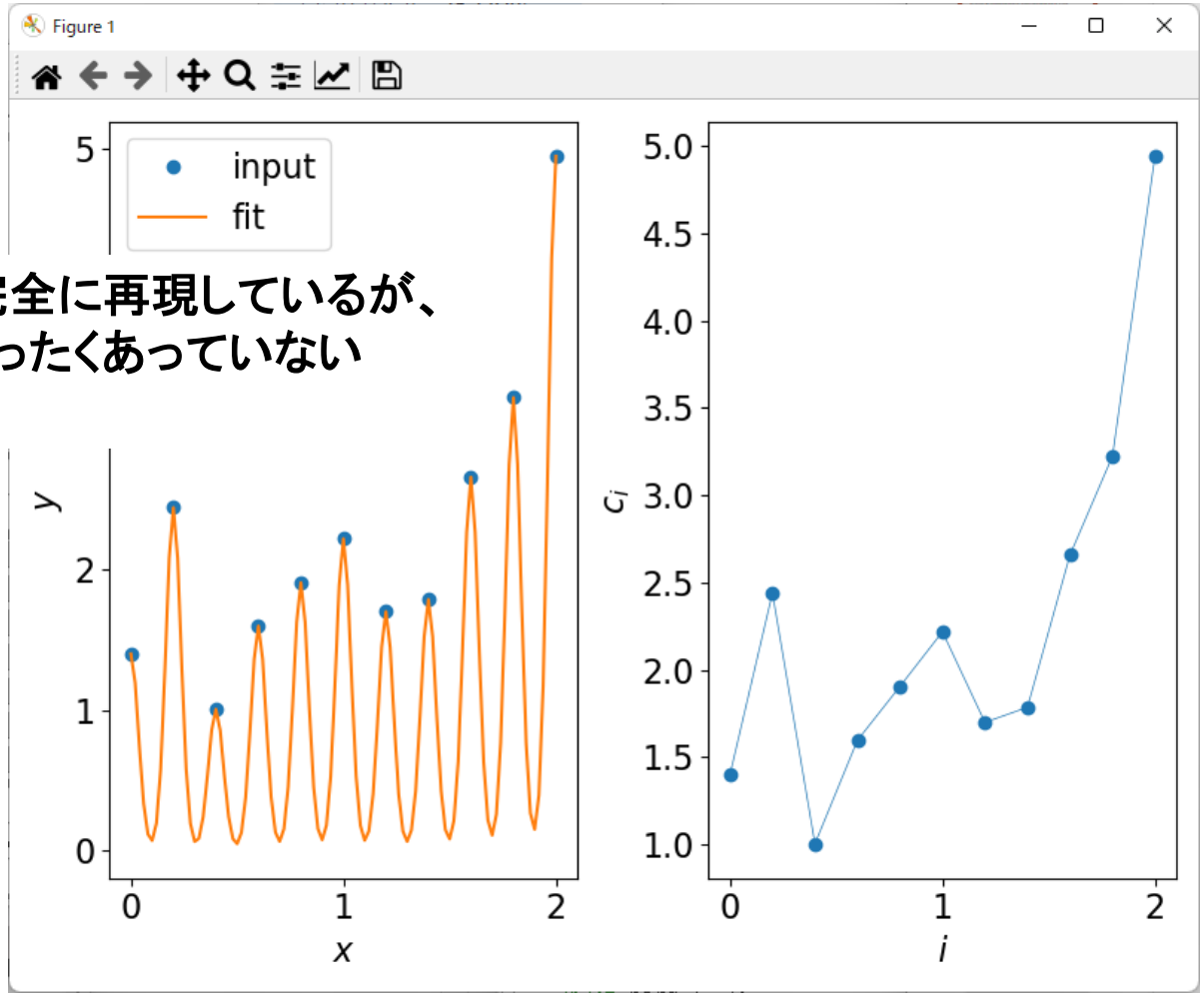
基底関数の数が自由

過剰適合: Program: Ridge-Gauusian.py

Usage: python KernelRidge-Gauusian.py infile nG wG lambda

ex: python KernelRidge-Gauusian.py random-poly-Gauss .xlsx 11 0.1 0.0

random-poly-Gauss .xlsx (データ点数11) を読み込み、11個の幅0.1のガウス関数でフィッティング。L2正規化項は入れない (Ridge解析をしない)



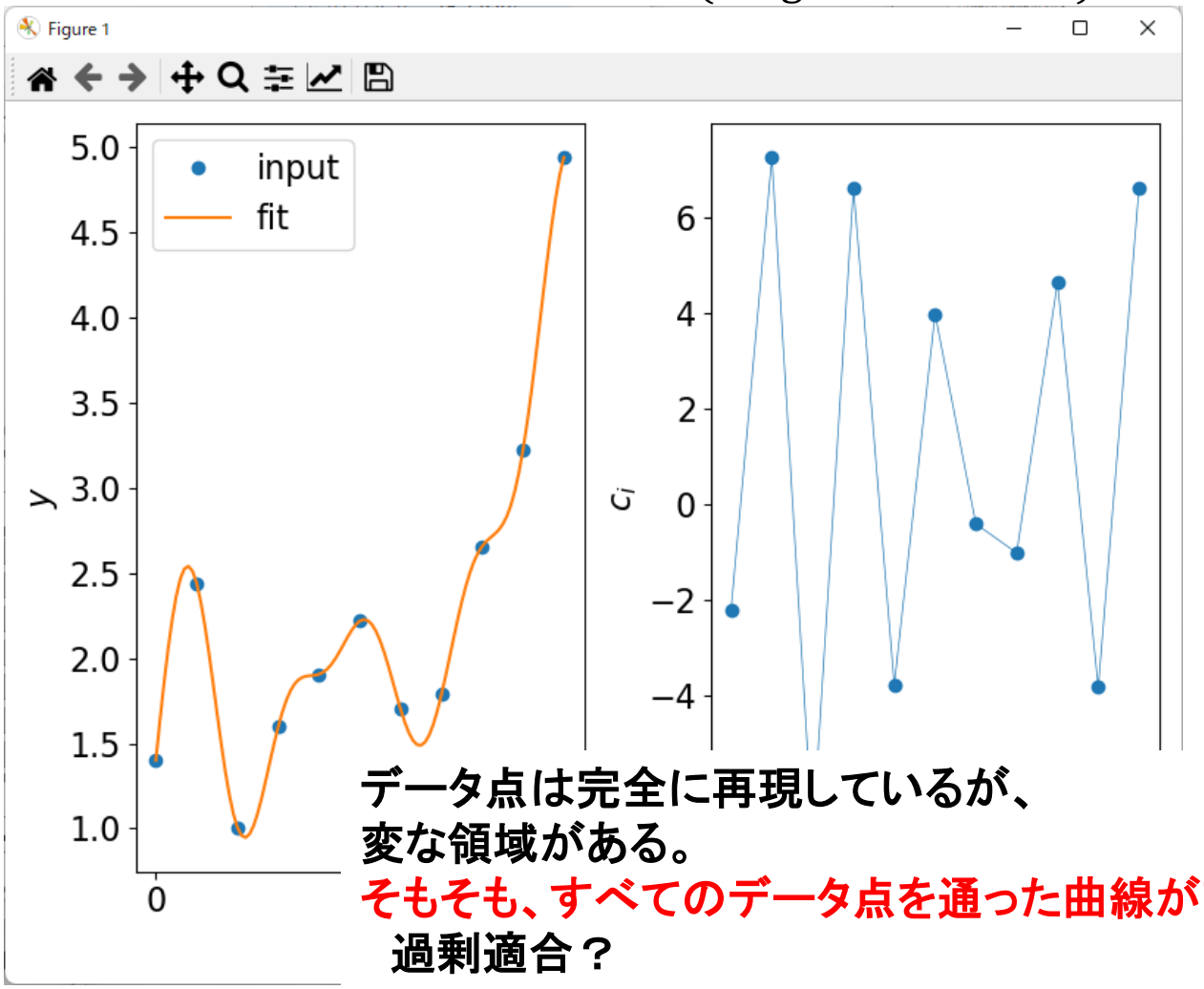
データ点は完全に再現しているが、
他のxではまったくあっていない
過剰適合

過剰適合: Program: Ridge-Gauusian.py

Usage: python KernelRidge-Gauusian.py infile nG wG lambda

ex: python KernelRidge-Gauusian.py random-poly-Gauss .xlsx 11 0.3 0.0

random-poly-Gauss .xlsx (データ点数11) を読み込み、11個の幅0.3のガウス関数でフィッティング。L2正規化項は入れない (Ridge解析をしない)



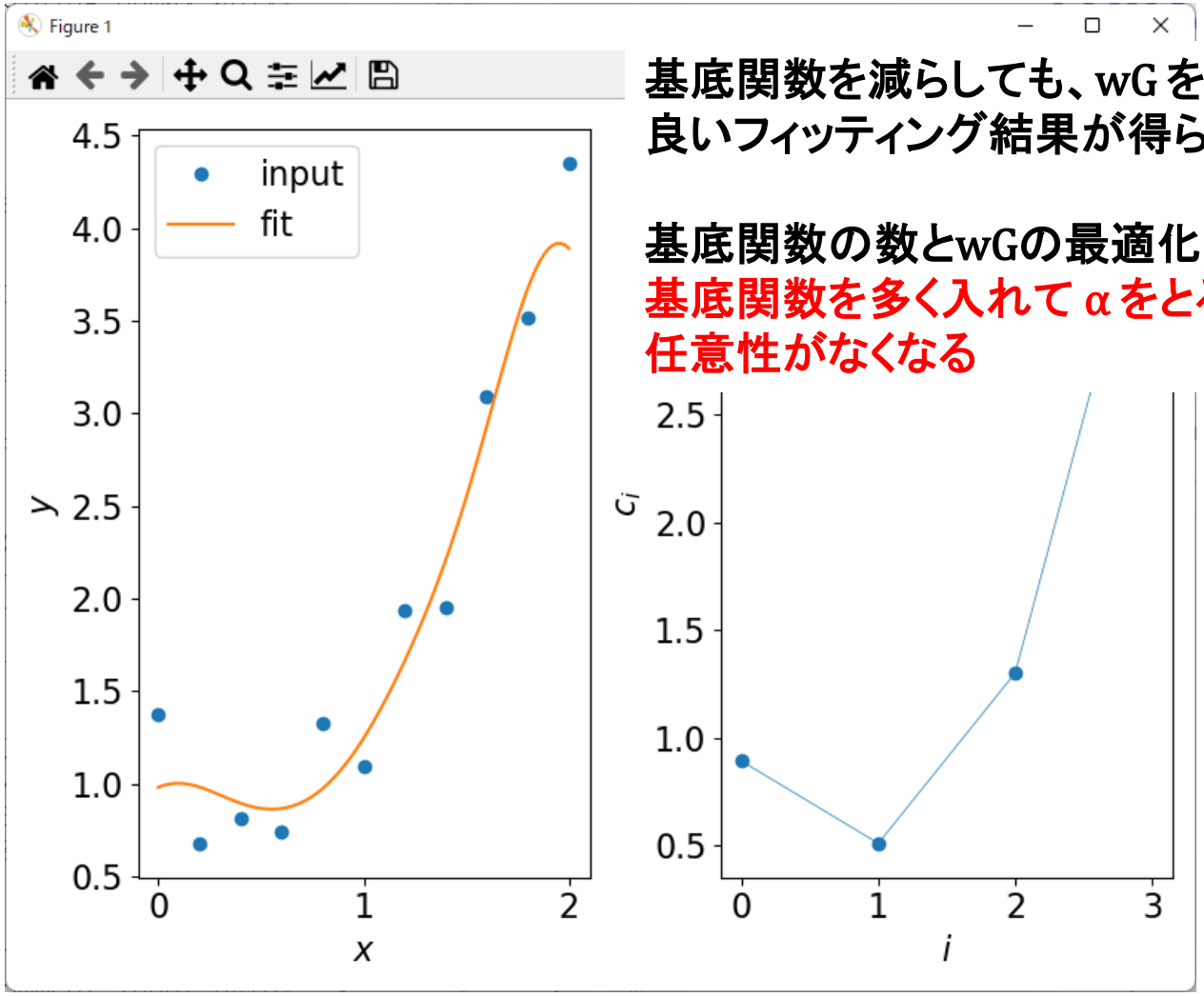
データ点は完全に再現しているが、
変な領域がある。
そもそも、すべてのデータ点を通った曲線が望ましい解か？
過剰適合？

リッジ回帰: 基底関数の数を減らした場合

Usage: python KernelRidge-Gauusian.py infile nG wG alpha

ex: python KernelRidge-Gauusian.py random-poly-Gauss .xlsx 4 0.5 0.1

random-poly-Gauss .xlsx (データ点数11) を読み込み、4個の幅0.5のガウス関数でフィッティング。L2正規化項の係数 0.1 を入れる (Ridge回帰)



基底関数を減らしても、wG を調整すれば
良いフィッティング結果が得られる

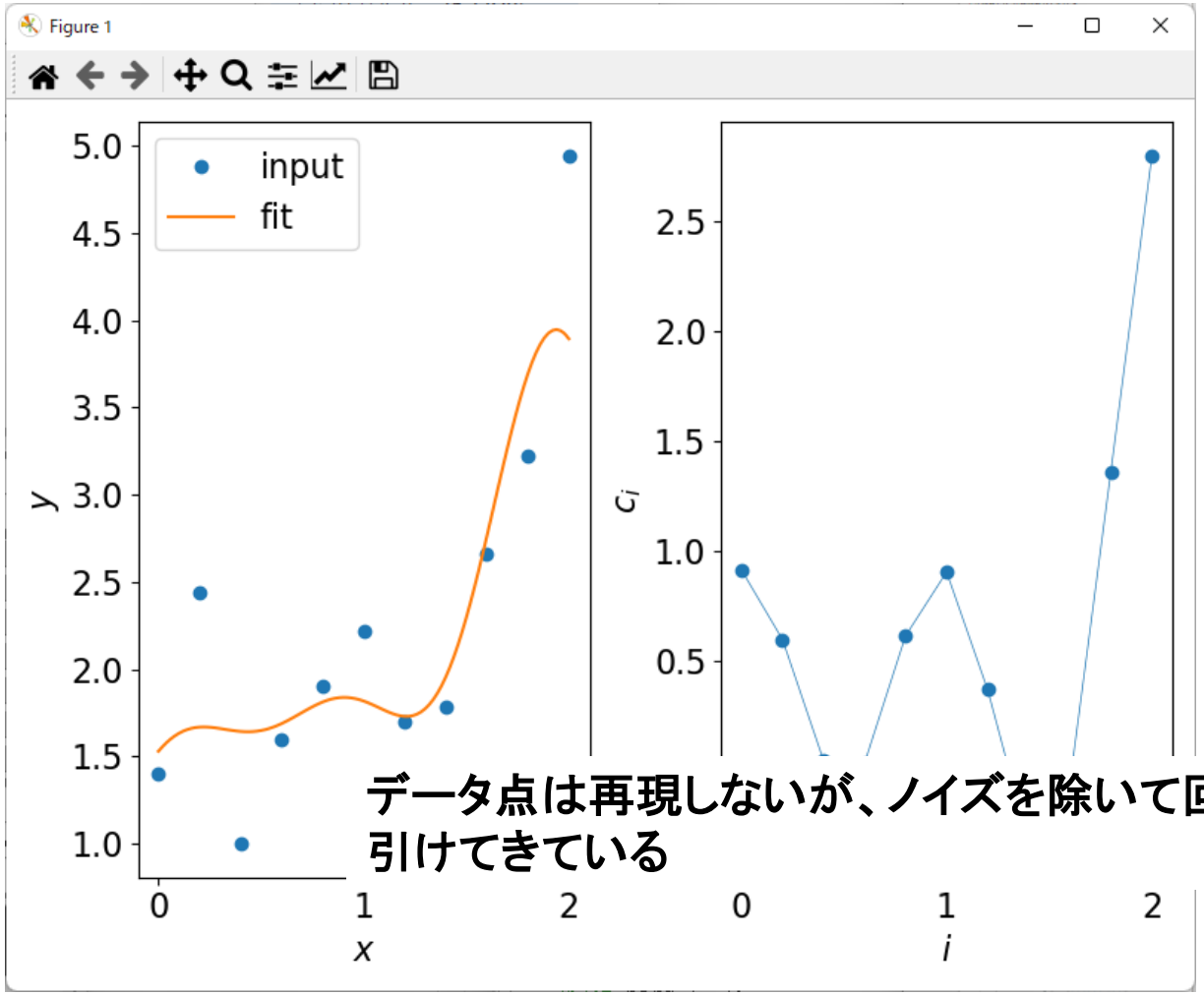
基底関数の数とwGの最適化に任意性が入る
基底関数を多く入れて α をとる方が
任意性がなくなる

Program: Ridge-Gauusian.py

Usage: python KernelRidge-Gauusian.py infile nG wG lambda

ex: python KernelRidge-Gauusian.py random-poly-Gauss .xlsx 11 0.5 0.2

random-poly-Gauss .xlsx (データ点数11) を読み込み、11個の幅0.5のガウス関数でフィッティング。L2正規化項の係数 0.2 を入れる (Ridge回帰)



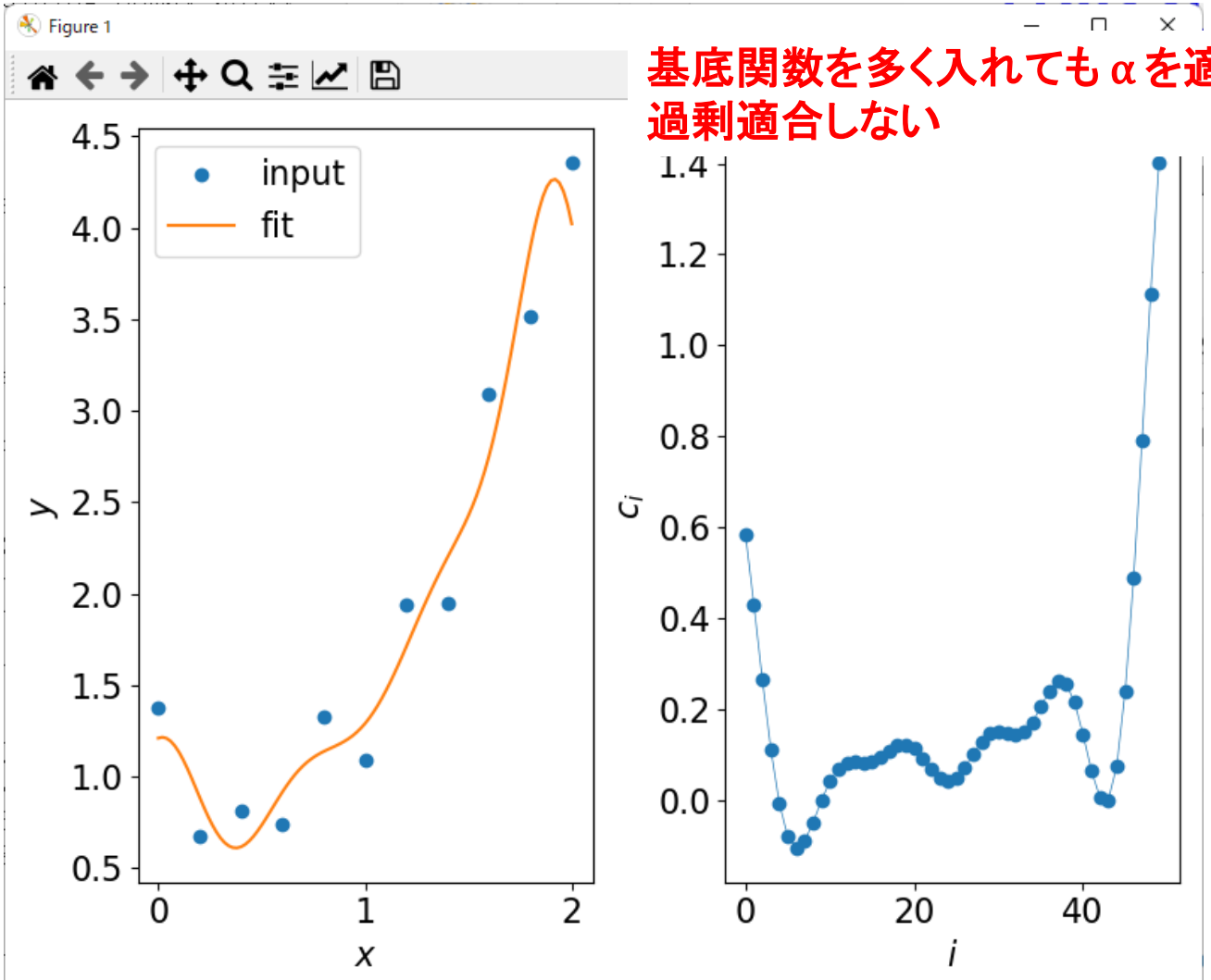
データ点は再現しないが、ノイズを除いて回帰曲線を引けてきている

リッジ回帰: 基底関数の数を増やした場合

Usage: python KernelRidge-Gauusian.py infile nG wG alpha

ex: python KernelRidge-Gauusian.py random-poly-Gauss .xlsx 50 0.3 0.1

random-poly-Gauss .xlsx (データ点数11) を読み込み、50個の幅0.3のガウス関数でフィッティング。L2正規化項の係数 0.1 を入れる (Ridge回帰)



カーネルリッジ回帰

カーネル法の最小化問題: $\mathrm{K} = (k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}))$ に対して

$S = (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y}) \cdot (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y}) = (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y})^T (\mathrm{K}\boldsymbol{A} - \boldsymbol{Y})$を最小化

$\Rightarrow \boldsymbol{A} = K^{-1}\boldsymbol{Y}$を解けばよい

L^2正規化項を加えて過学習を抑える: Ridge Kernel Regression

$S = (\mathrm{K}\boldsymbol{A} - Y) \cdot (\mathrm{K}\boldsymbol{A} - Y) + p\alpha \boldsymbol{A} \cdot \mathrm{K}\boldsymbol{A}$　を最小化

(pは正規化係数が記述子の次元に依存しないように導入)

正則化項に K を入れているために次のように簡単になる

$\boldsymbol{A} = \left(K + p\alpha \boldsymbol{I_p}\right)^{-1} \boldsymbol{Y}$　　$\boldsymbol{I_p}$はp次元単位行列

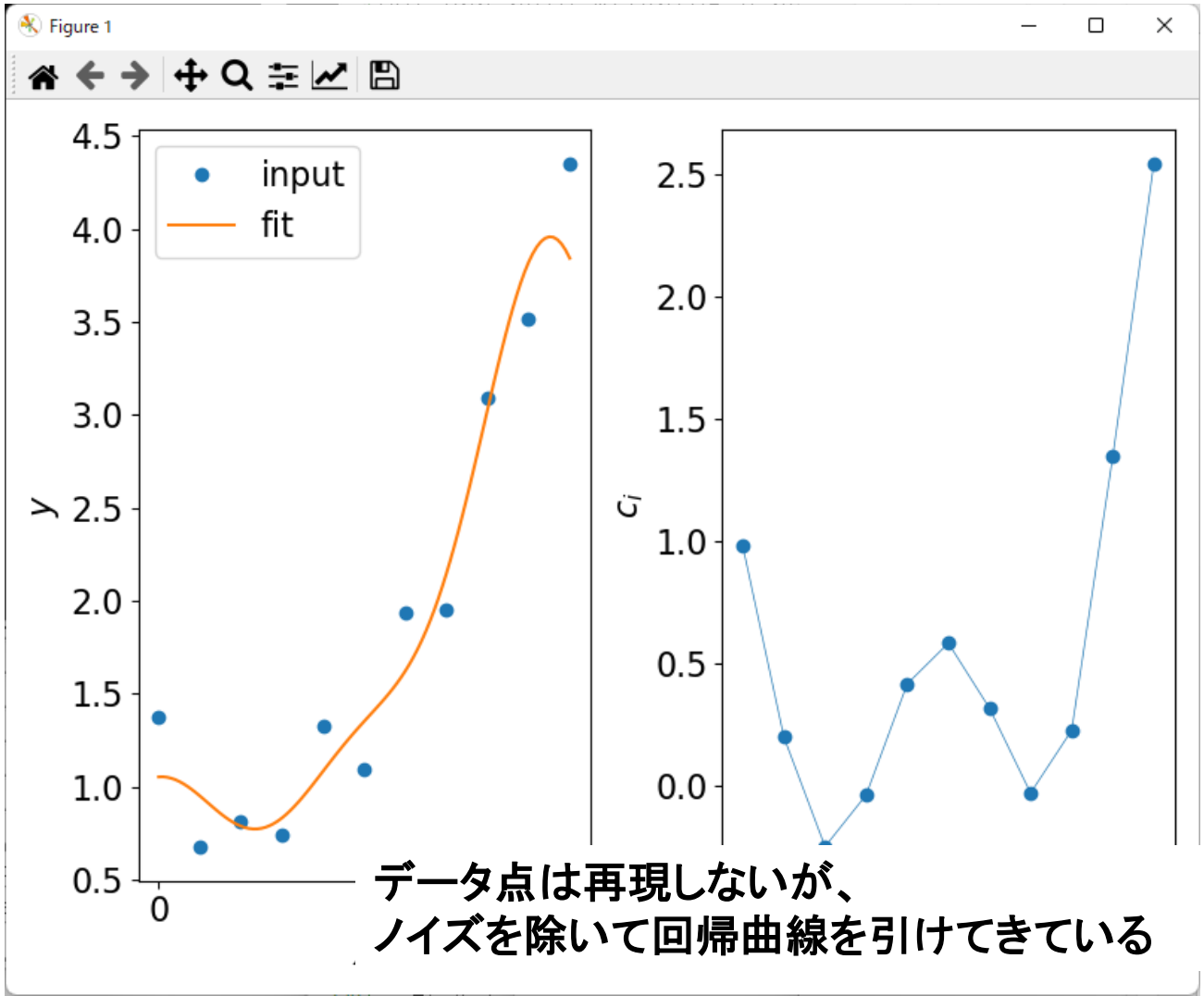
$f(\boldsymbol{x_k}) = \sum_{k'=1}^{p} a_{k'}\, k(\boldsymbol{x_k}, \boldsymbol{x_{k'}}) = \mathrm{K}\boldsymbol{A} = \mathrm{K}\left(K + p\alpha \boldsymbol{I_p}\right)^{-1} \boldsymbol{Y}$

カーネルリッジ回帰: KernelRidge-Gauusian.py

Usage: python KernelRidge-Gauusian.py infile nG wG alpha

ex: python KernelRidge-Gauusian.py random-poly-Gauss .xlsx 11 0.5 0.1

random-poly-Gauss .xlsx (データ点数11) を読み込み、11個の幅0.5のガウス関数でフィッティング。L2正規化項の係数 0.1 を入れる (Ridge回帰)



機械学習 (回帰)
ハイパーパラメータ
検定
カテゴリー変数

scikit-learn: 回帰の手順

lsq-polynomial-ML.py :

```
#Import
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression                    # 方法 (アルゴリズム) をimport
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score        # 評価関数をimport
#データ読み込み
  df = pd.read_excel(infile, engine = 'openpyxl‘)
  x = df[labels[2:]]
  y = df[labels[1]]
#記述子の標準化
   scaler = StandardScaler()
   scaler.fit(x)
   x_scaled = scaler.transform(x)
#フィッティング
   model = LinearRegression()                                        # 方法 (アルゴリズム) を選択
   model.fit(x_scaled, y)
#フィッティング結果から予測値を計算
   y_cal = model.predict(x_scaled)
#評価
   mae  = mean_absolute_error(y, y_cal)
   mse  = mean_squared_error(y, y_cal)
   rmse = sqrt(mse)
#パラメータ
   print(f“   intercept: {model.intercept_}”)                        #  定数項。決定木、NNなどには無い
   for iv in range(len(x_labels)):
      print(f“   {x_labels[iv]:>10}: {model.coef_[iv]:12.4g}”)   # 多項式回帰などの場合、記述子の係数。
                                                                    # 決定木、NNなどには無い
```

scikit-learnで使える回帰アルゴリズム

線形回帰系:

多重線形回帰

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

多項式回帰

```
from sklearn.preprocessing import PolynomialFeatures
model = PolynomialFeatures(degree = 2, interaction_only = False, include_bias = True,  order = 'C’)
```

Ridge回帰

```
from sklearn.linear_model import Ridge
model = Ridge(alpha = 0.05)
```

LASSO回帰

```
from sklearn.linear_model import Lasso
model = Lasso(alpha = 0.05)
```

Elastic Net回帰

```
from sklearn.linear_model import ElasticNet
model = ElasticNet(alpha = 0.05)
```

ノンパラメトリック系:

Kernal Ridge回帰

```
from sklearn.kernel_ridge import KernelRidge
model = KernelRidge(alpha = 1.0, kernel = 'rbf’)
```

Gauss過程回帰

```
from sklearn.gaussian_process import GaussianProcessRegressor
model = GaussianProcessRegressor()
```

多層パーセプトロン (多層ニューラルネットワーク)

```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(max_iter = 1000, hidden_layer_sizes = (10,), activation = 'logistic', solver = 'sgd’,
                      learning_rate_init = 0.01)
```

scikit-learn: 回帰

分類器系:

Random Forest回帰

```
from sklearn.ensemble import RandomForestRegressor
model = RandomForestRegressor()
```

勾配ブースティング木 回帰

```
from sklearn.ensemble import GradientBoostingRegressor
model = GradientBoostingRegressor()
```

サポートベクターマシーン 回帰

```
from sklearn.svm import SVR
model = SVR(kernel='linear', C=1, epsilon=0.1, gamma='auto’)
```

scikit-learnで使える回帰アルゴリズム (Web検索より)

```
# sklearnの回帰モデルを片っ端から試す
# https://qiita.com/futakuchi0117/items/72ce4afae9adcccd6e18

from sklearn.linear_model import LinearRegression, Ridge, Lasso, ElasticNet, SGDRegressor
from sklearn.linear_model import PassiveAggressiveRegressor, ARDRegression, RidgeCV
from sklearn.linear_model import TheilSenRegressor, RANSACRegressor, HuberRegressor
from sklearn.neural_network import MLPRegressor
from sklearn.svm import SVR, LinearSVR
from sklearn.neighbors import KNeighborsRegressor
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.tree import DecisionTreeRegressor
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor, ExtraTreesRegressor, HistGradientBoostingRegressor
from sklearn.ensemble import BaggingRegressor, GradientBoostingRegressor, VotingRegressor, StackingRegressor
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.cross_decomposition import PLSRegression

reg_dict = {"LinearRegression": LinearRegression(),
            "Ridge": Ridge(),
            "Lasso": Lasso(),
            "ElasticNet": ElasticNet(),
            "Polynomial_deg2": Pipeline([('poly', PolynomialFeatures(degree=2)),('linear', LinearRegression())]),
            "Polynomial_deg3": Pipeline([('poly', PolynomialFeatures(degree=3)),('linear', LinearRegression())]),
            "Polynomial_deg4": Pipeline([('poly', PolynomialFeatures(degree=4)),('linear', LinearRegression())]),
            "Polynomial_deg5": Pipeline([('poly', PolynomialFeatures(degree=5)),('linear', LinearRegression())]),
            "KNeighborsRegressor": KNeighborsRegressor(n_neighbors=3),
            "DecisionTreeRegressor": DecisionTreeRegressor(),
            "RandomForestRegressor": RandomForestRegressor(),
            "SVR": SVR(kernel='rbf', C=1e3, gamma=0.1, epsilon=0.1),
            "GaussianProcessRegressor": GaussianProcessRegressor(),
            "SGDRegressor": SGDRegressor(),
            "MLPRegressor": MLPRegressor(hidden_layer_sizes=(10,10), max_iter=100, early_stopping=True, n_iter_no_change=5),
            "ExtraTreesRegressor": ExtraTreesRegressor(n_estimators=100),
            "PLSRegression": PLSRegression(n_components=10),
            "PassiveAggressiveRegressor": PassiveAggressiveRegressor(max_iter=100, tol=1e-3),
            "TheilSenRegressor": TheilSenRegressor(random_state=0),
            "RANSACRegressor": RANSACRegressor(random_state=0),
            "HistGradientBoostingRegressor": HistGradientBoostingRegressor(),
            "AdaBoostRegressor": AdaBoostRegressor(random_state=0, n_estimators=100),
            "BaggingRegressor": BaggingRegressor(base_estimator=SVR(), n_estimators=10),
            "GradientBoostingRegressor": GradientBoostingRegressor(random_state=0),
            "VotingRegressor": VotingRegressor([('lr', LinearRegression()), ('rf', RandomForestRegressor(n_estimators=10))]),
            "StackingRegressor": StackingRegressor(estimators=[('lr', RidgeCV()), ('svr', LinearSVR())], final_estimator=RandomForestRegressor(n_estimators=10)),
            "ARDRegression": ARDRegression(),
            "HuberRegressor": HuberRegressor(),
                }
```

機械学習回帰の例: scikit-regressions.py

入力ファイル: 一変数関数 $f(x)$の回帰

ファイル: [tkProg]¥tkprog_tutorial¥regression¥random-poly-ML_with_blank.xlsx

プログラム: scikit-regressions.py

y	x^0	x^1	x^2	x^3
	1	0	0	0
	1	0.05	0.0025	0.000125
	1	0.1	0.01	0.001
11.13943	1	0	0	0
8.412969	1	0.05	0.0025	0.000125
4.850587	1	0.1	0.01	0.001
4.118663	1	0.15	0.0225	0.003375
6.692228	1	0.2	0.04	0.008
4.904099	1	0.25	0.0625	0.015625
0.635779	1	0.3	0.09	0.027
13.57628	1	0.35	0.1225	0.042875
3.65209	1	0.4	0.16	0.064
8.525117	1	0.45	0.2025	0.091125
14.0056	1	0.5	0.25	0.125
12.57216	1	0.55	0.3025	0.166375
8.362779	1	0.6	0.36	0.216
12.62163	1	0.65	0.4225	0.274625

ラベルの制御子 (bayes_gp_plain.pyに類似)

- 先頭が'-'の場合、記述子にも目的関数にも含めない
- 先頭が 'o:', 't:', 'max:', 'min:' の場合、目的関数とする
- その他の行は記述子とする

第１列: 目的関数となる

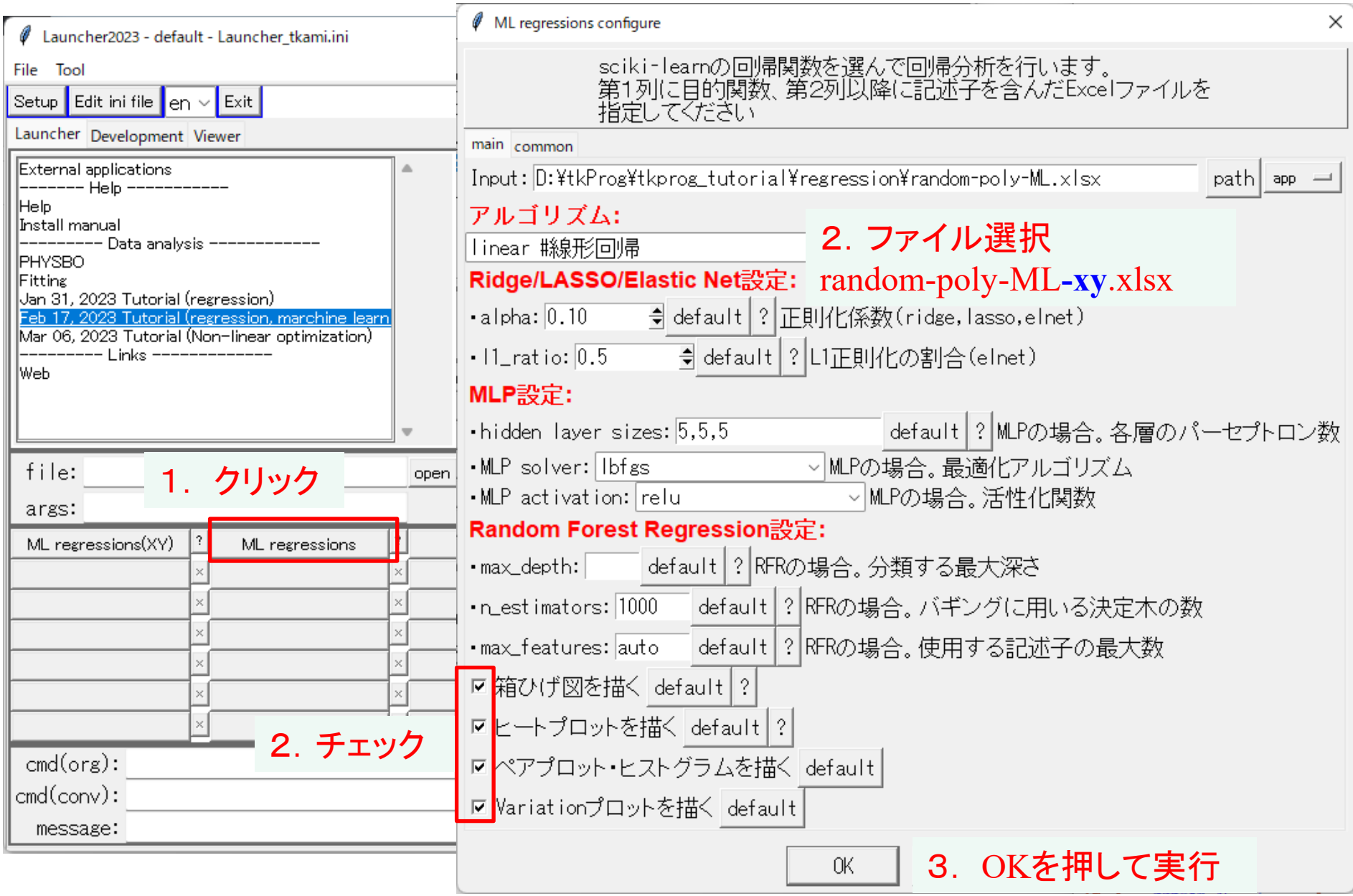
目的関数が入力されているデータは、

学習データ、検証データに使われる

目的関数がブランクの部分は、予測値を計算する

第２列以降: 記述子

機械学習の例: scikit-regressions.py



model=linear　# 線形回帰

コンソール出力

infile=D:/tkProg/tkprog_tutorial/regression/random-poly-ML-xy.xlsx

method=linear

Fraction of test data=0.3

--cut--

Covariances of standardized values　　# 共分散

(0, 1) (-x, o:y): 0.1117
(0, 2) (-x, x^1): 0.1080
(0, 3) (-x, x^2): 0.1036

--cut--

Scores:　　# 評価関数

Mean absolute error (MAE): training 3.63 test: 3.79
Mean squared error (MSE) : training 17.8 test: 22.7
Root MSE (RMSE) : training 4.22 test: 4.76
R^2 score : training 0.931 test: 0.93

--cut--

Parameters:　　# 線形回帰の係数

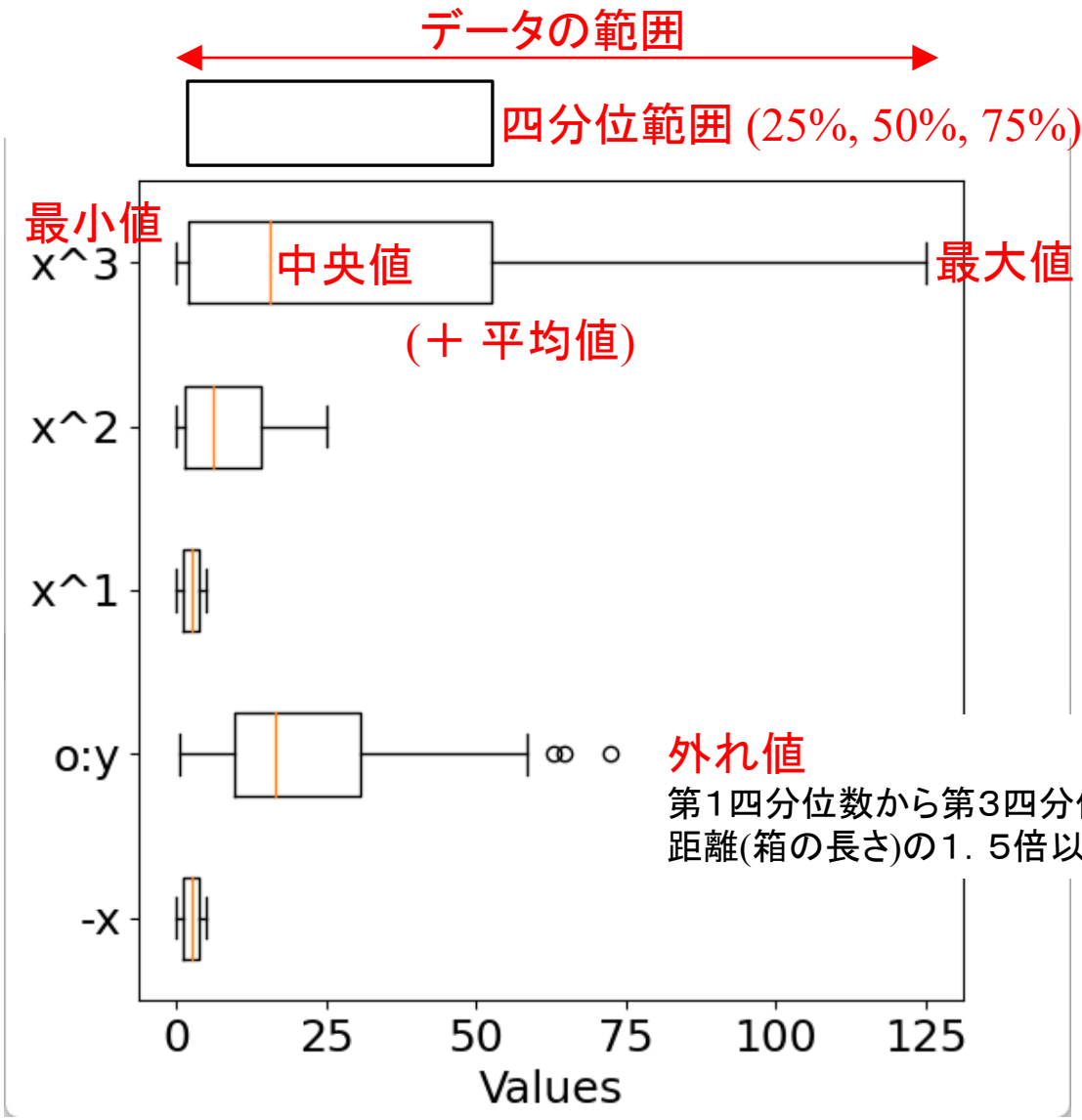
intercept: 21.702580366119125
coefficients
x^1: 2.159
x^2: -7.523
x^3: 20.96

--cut--

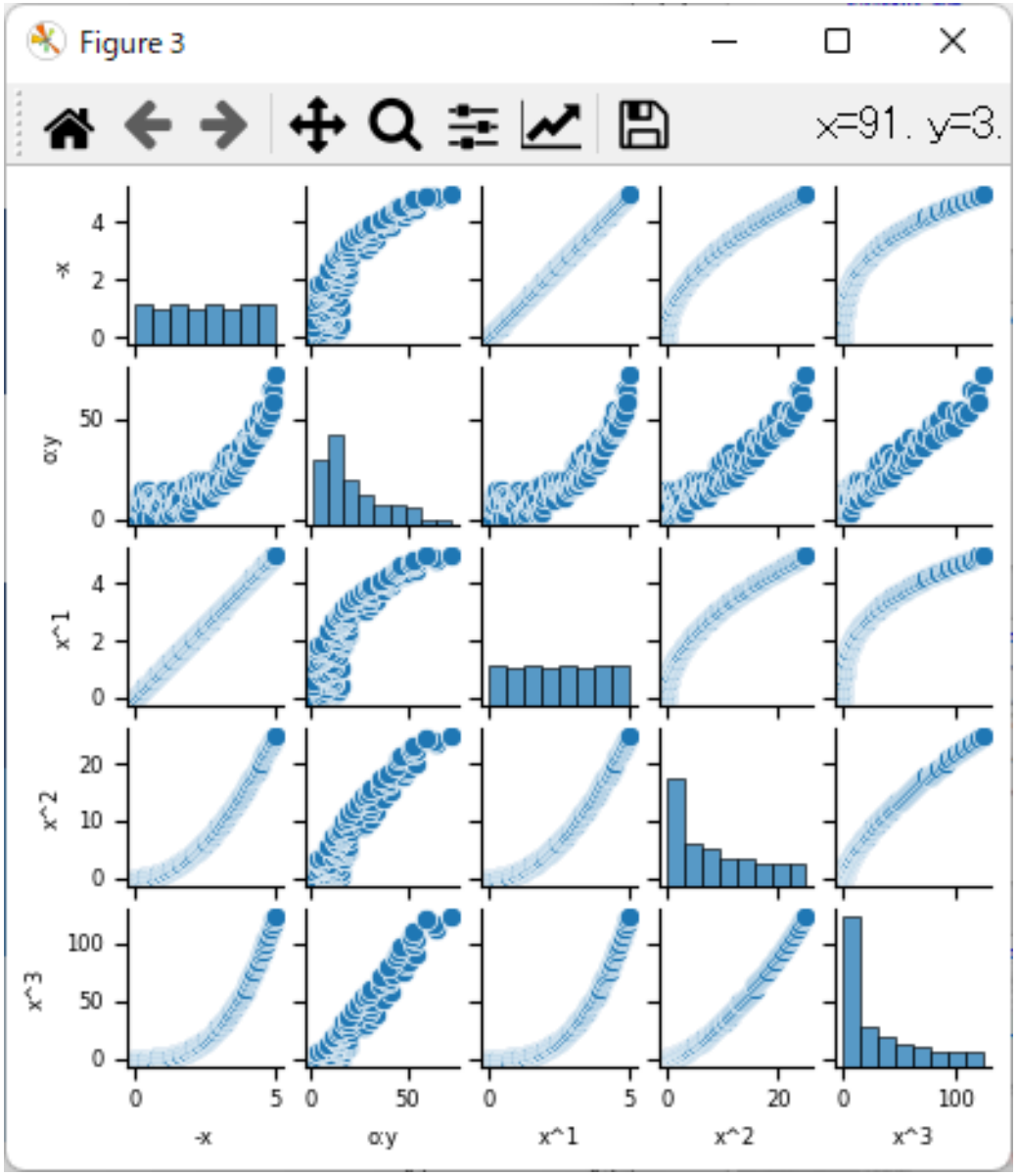
Correlation heatmap　　# 相関係数

	-x	o:y	x^1	x^2	x^3
-x	1.000000	0.882608	1.000000	0.967650	0.915525
o:y	0.882608	1.000000	0.882608	0.950931	0.965255
x^1	1.000000	0.882608	1.000000	0.967650	0.915525
x^2	0.967650	0.950931	0.967650	1.000000	0.985951
x^3	0.915525	0.965255	0.915525	0.985951	1.000000

箱ひげ図


データの範囲
四分位範囲 (25%, 50%, 75%)
最小値
中央値
最大値
(＋ 平均値)
x^3
x^2
x^1
o:y
-x
外れ値
第1四分位数から第3四分位数の
距離(箱の長さ)の1．5倍以上離れた値
0
25
50
75
100
125
Values

分布図 (非対角成分) とヒストグラム (対角成分)



ヒートマップ: 相関係数

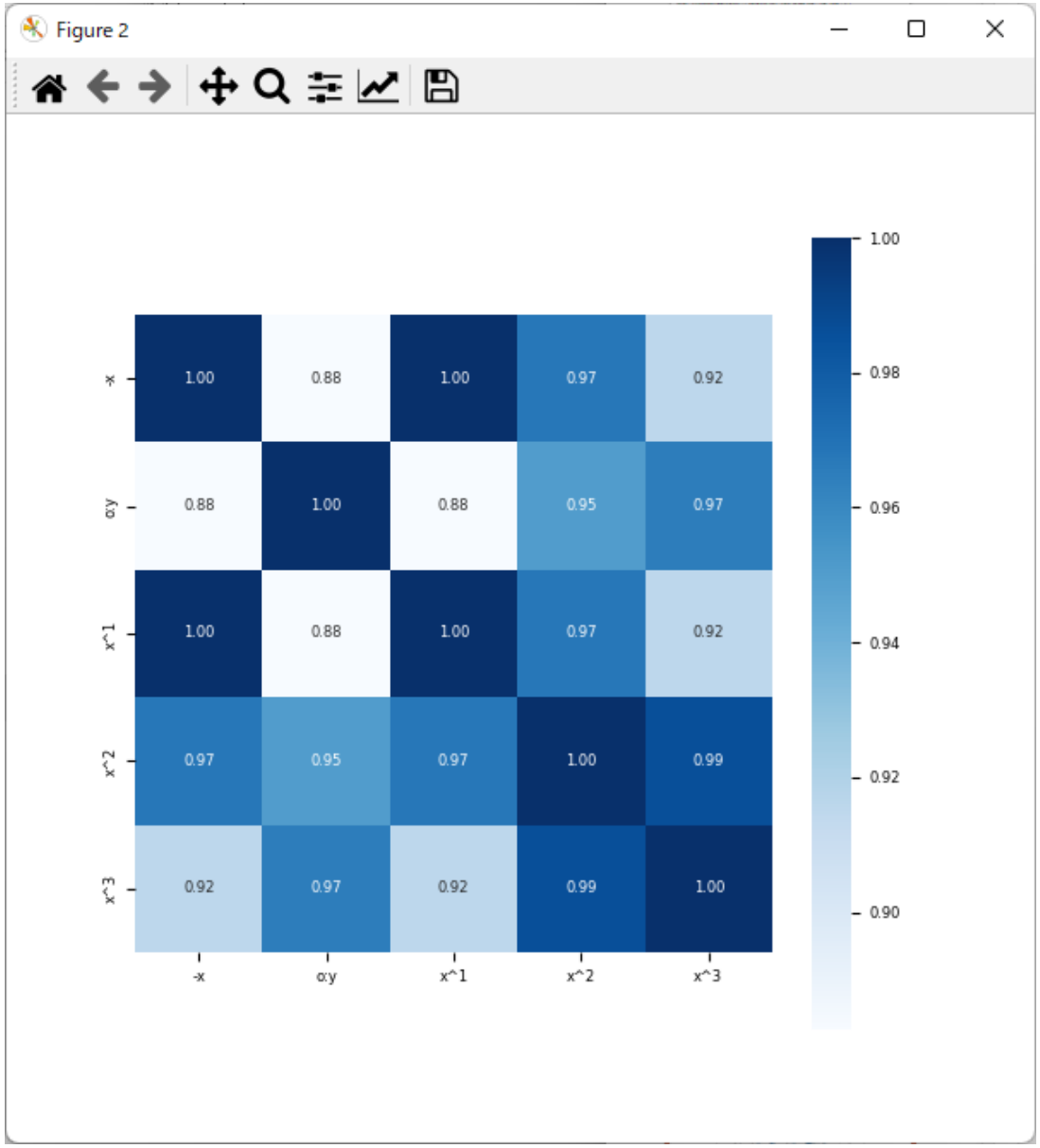
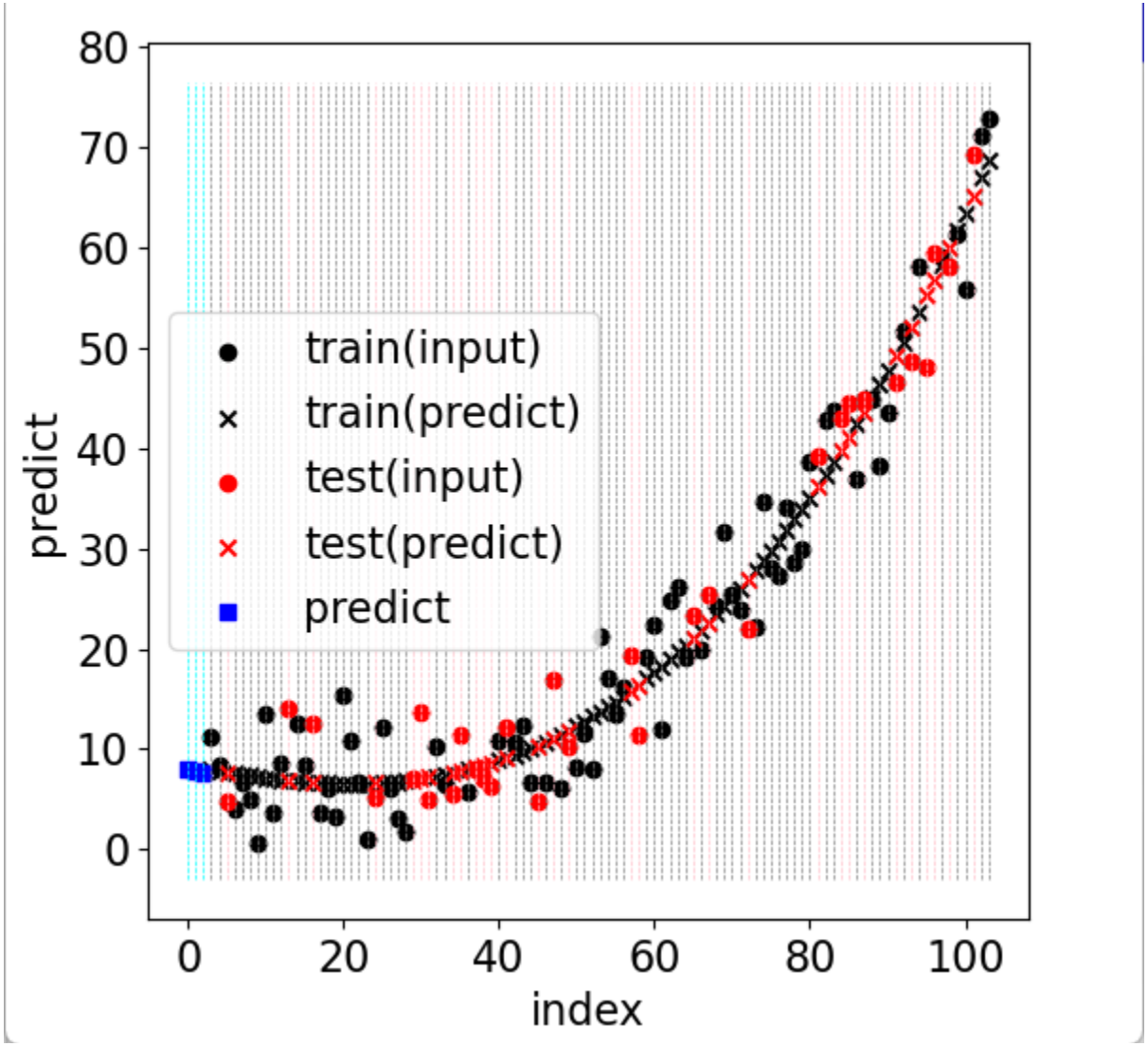


Figure 2
-x
α:y
x^1
x^2
x^3
1.00
0.88
1.00
0.97
0.92
0.88
1.00
0.88
0.95
0.97
1.00
0.88
1.00
0.97
0.92
0.97
0.95
0.97
1.00
0.99
0.92
0.97
0.92
0.99
1.00
1.00
0.98
0.96
0.94
0.92
0.90

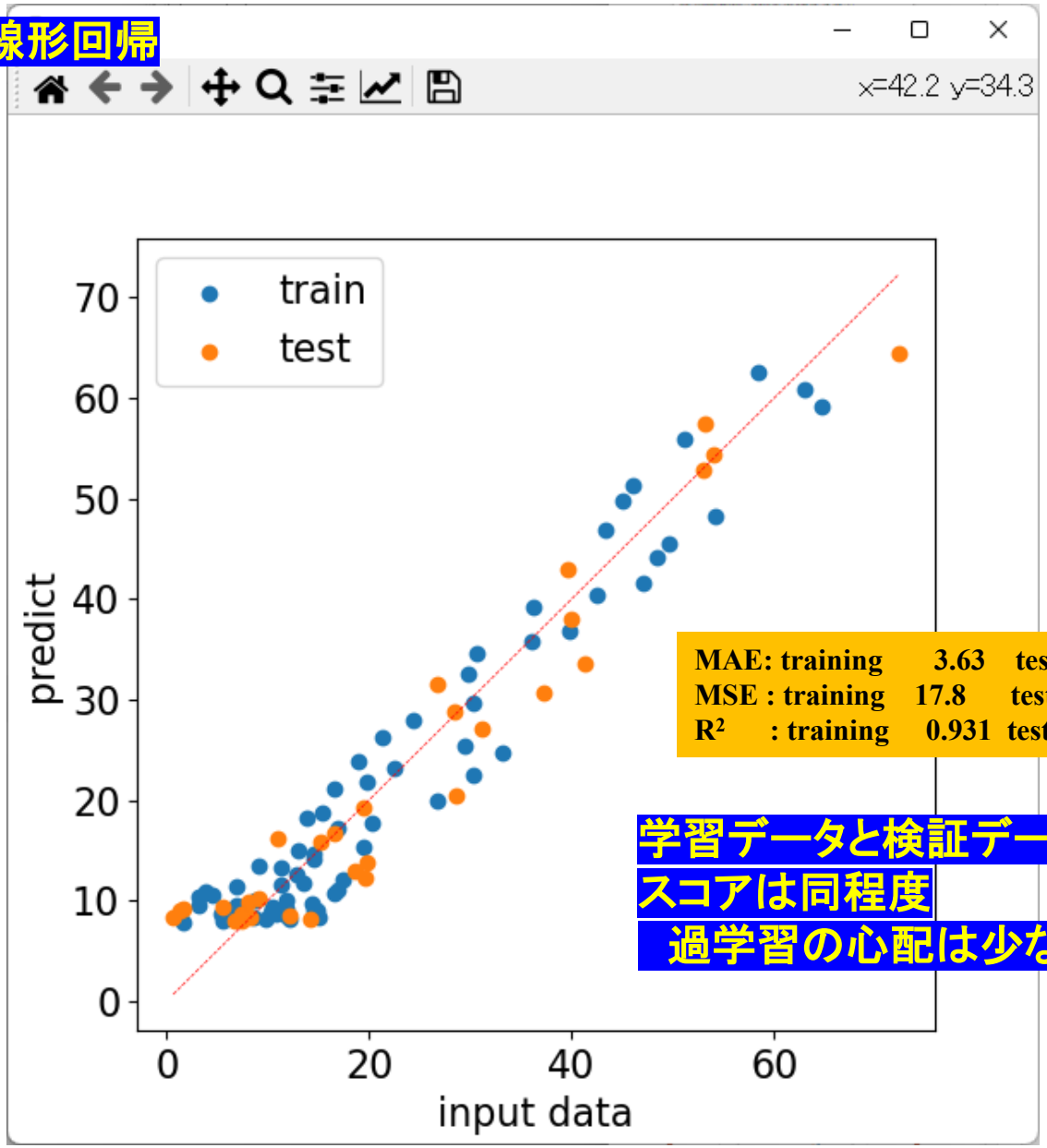
回帰結果: 入力値と予測値

model=linear　# 線形回帰



回帰結果の評価: 入力値ー予測値

model=linear　# 線形回帰



MAE: training 3.63 test: 3.79
MSE : training 17.8 test: 22.7
R^2 : training 0.931 test: 0.930

学習データと検証データの
スコアは同程度
過学習の心配は少ない

出力ファイル

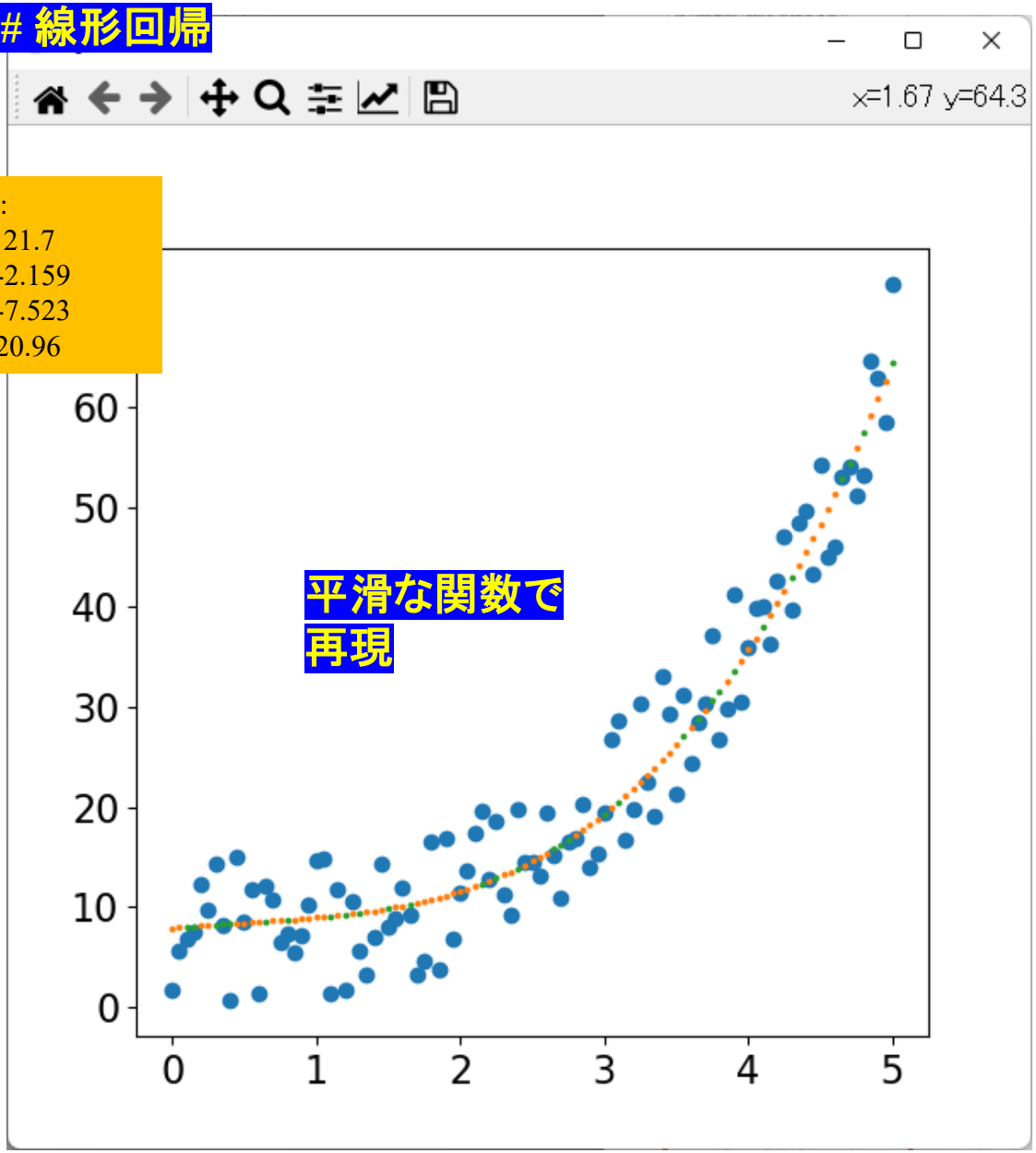
出力ファイル: random-poly-ML_with_blank-out.xlsx
コンソール出力

出力ファイル: random-poly-ML_with_blank-out.xlsx
入力ファイルの左端に predict を追加

	predict	y	x^0	x^1	x^2	x^3
0	7.996834		1	0	0	0
1	7.852176		1	0.05	0.0025	0.000125
2	7.713591		1	0.1	0.01	0.001
3	7.996834	11.13943	1	0	0	0
4	7.852176	8.412969	1	0.05	0.0025	0.000125
5	7.713591	4.850587	1	0.1	0.01	0.001
6	7.581361	4.118663	1	0.15	0.0225	0.003375
7	7.455764	6.692228	1	0.2	0.04	0.008
8	7.33708	4.904099	1	0.25	0.0625	0.015625
9	7.225589	0.635779	1	0.3	0.09	0.027
10	7.121569	13.57628	1	0.35	0.1225	0.042875
11	7.025301	3.65209	1	0.4	0.16	0.064
12	6.937064	8.525117	1	0.45	0.2025	0.091125
13	6.857138	14.0056	1	0.5	0.25	0.125
14	6.785802	12.57216	1	0.55	0.3025	0.166375
15	6.723335	8.362779	1	0.6	0.36	0.216

X-Yプロット: １変数関数への回帰の場合のみ

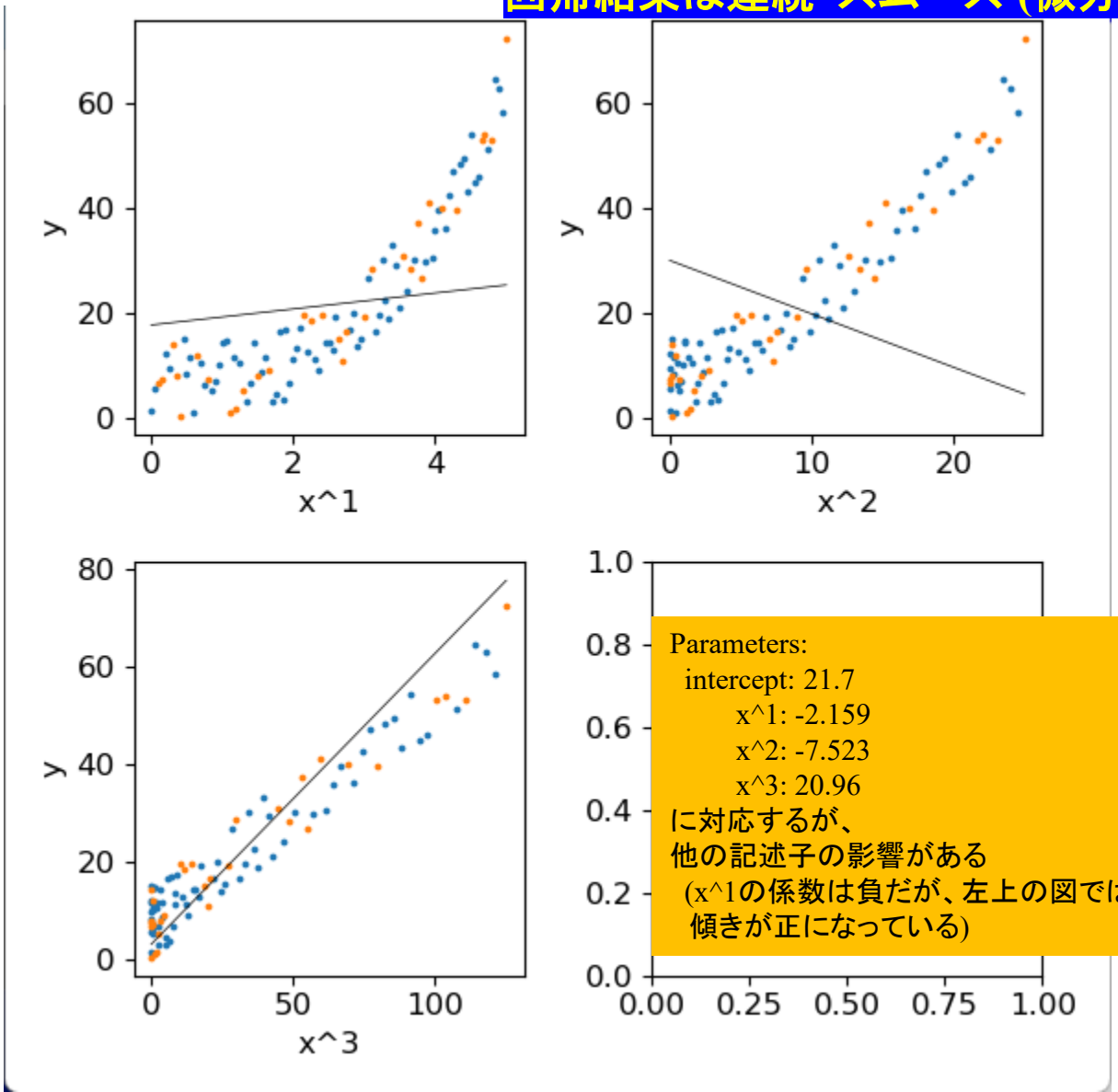
model=linear　# 線形回帰



X-Yプロット: １変数関数への回帰の場合のみ

model=linear　# 線形回帰

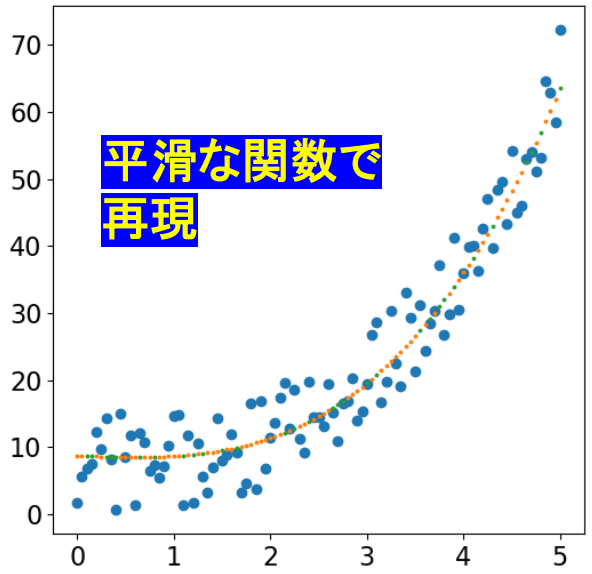
回帰結果は連続・スムーズ (微分可)



Parameters:
  intercept: 21.7
　　x^1: -2.159
　　x^2: -7.523
　　x^3: 20.96
に対応するが、
他の記述子の影響がある
 (x^1の係数は負だが、左上の図では
 傾きが正になっている)

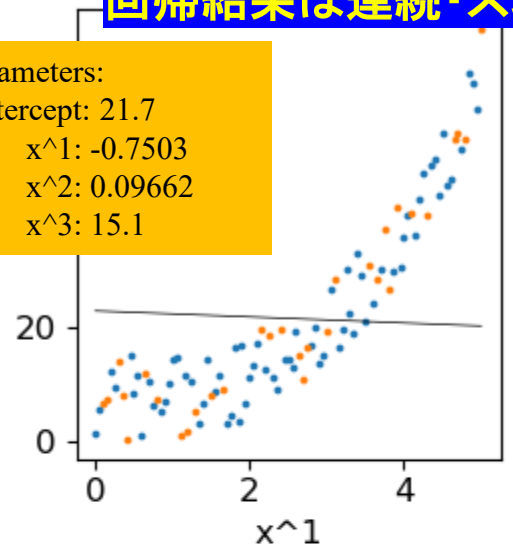
Ridge回帰

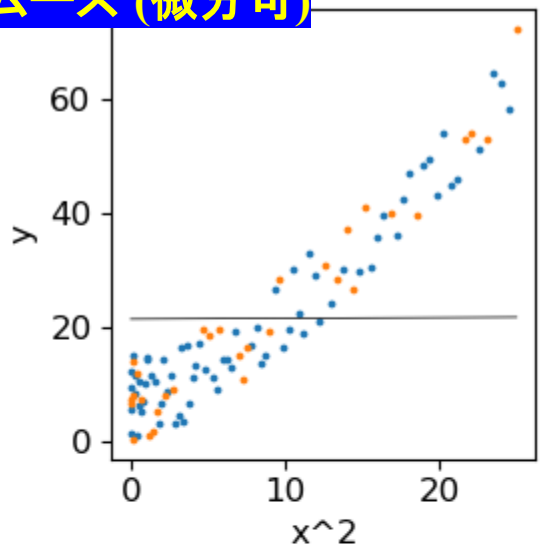
model=ridge
α=0.1

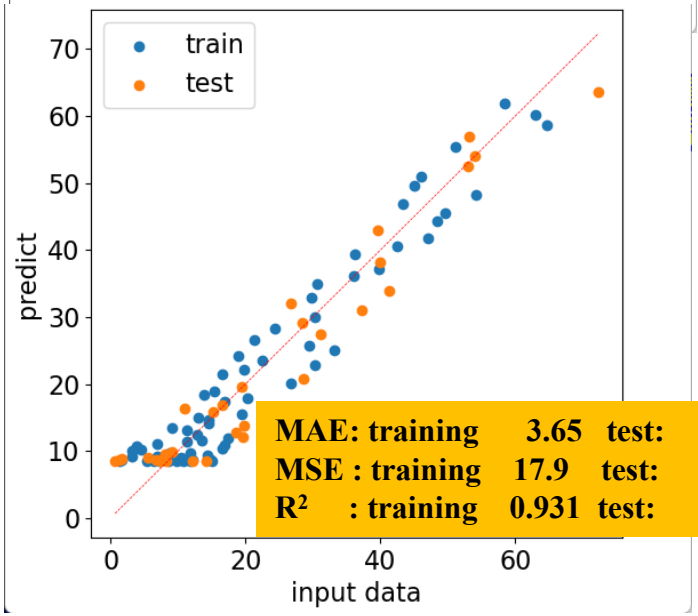


回帰結果は連続・スムーズ (微分可)

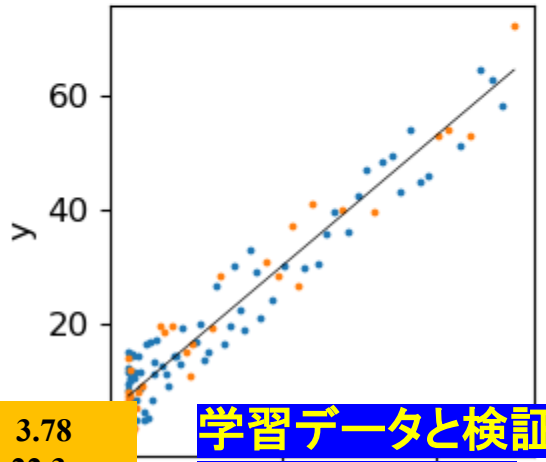
Parameters:
intercept: 21.7
x^1: -0.7503
x^2: 0.09662
x^3: 15.1

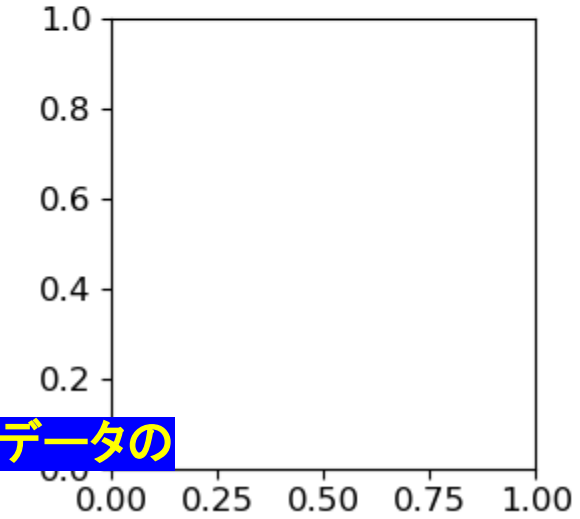






MAE: training 3.65 test: 3.78
MSE : training 17.9 test: 22.3
R^2 : training 0.931 test: 0.931

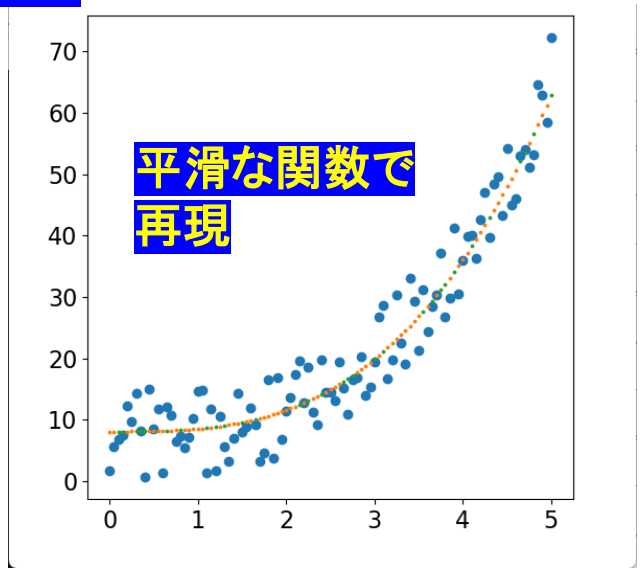




学習データと検証データの
スコアは同程度
過学習の心配は少ない

LASSO回帰

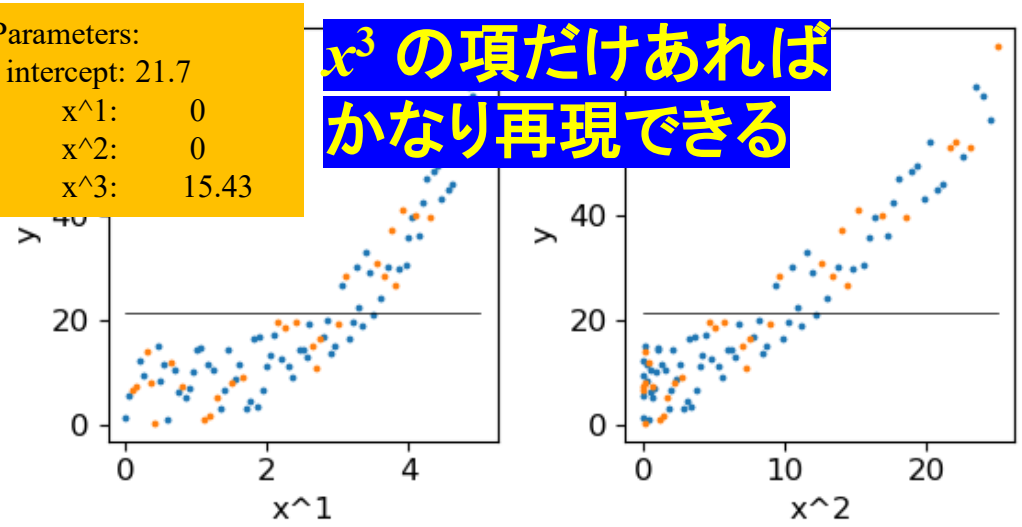
model=lasso
α=0.1

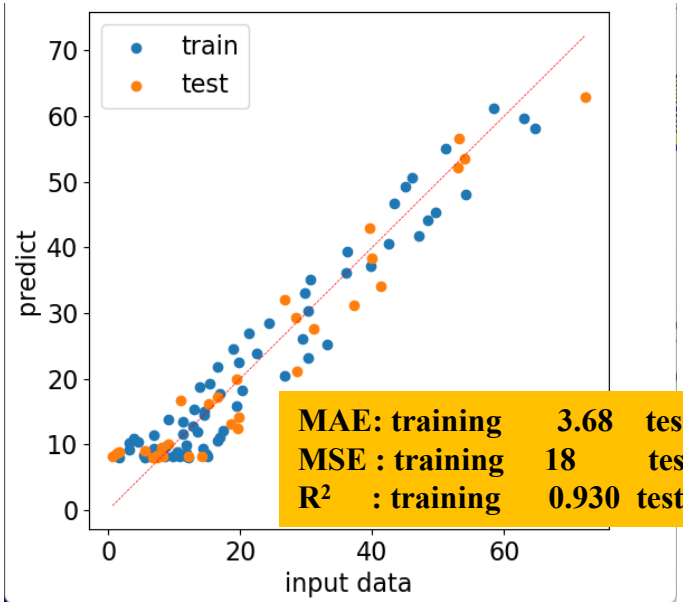


回帰結果は連続・スムーズ (微分可)

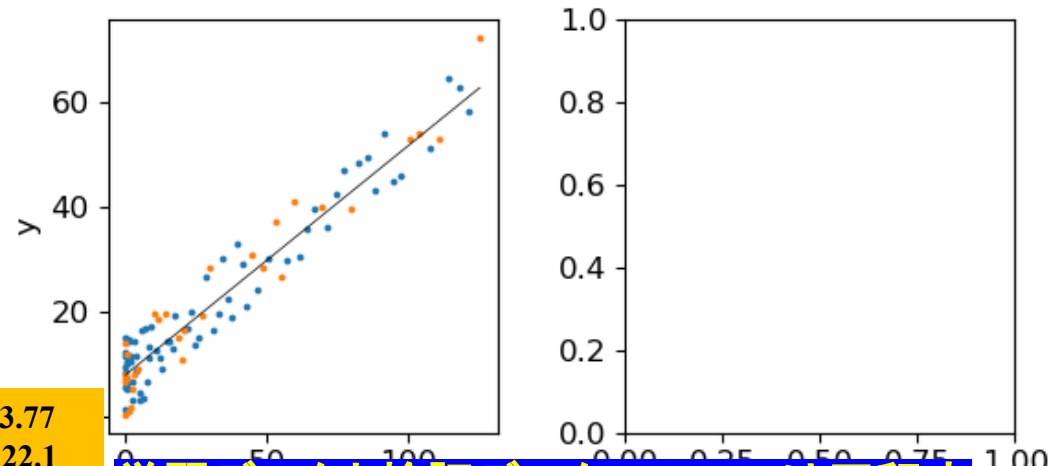
Parameters:
intercept: 21.7
x^1: 0
x^2: 0
x^3: 15.43

x^3 の項だけあれば
かなり再現できる





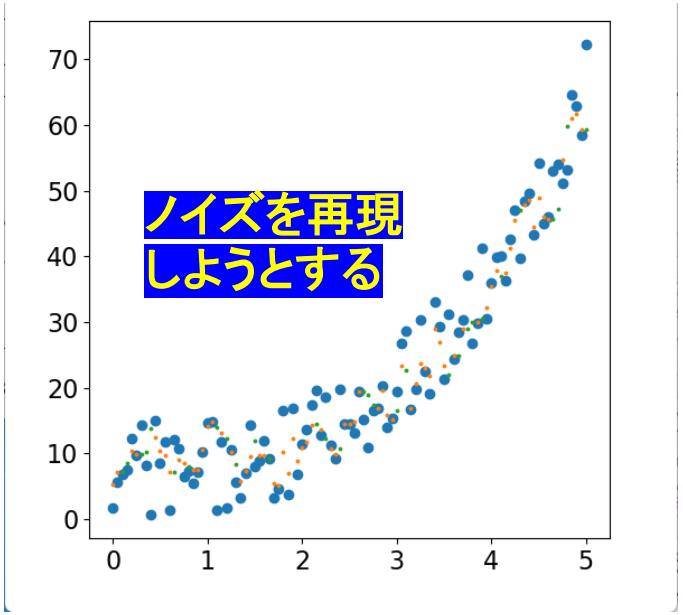
MAE: training 3.68 test: 3.77
MSE : training 18 test: 22.1
R^2 : training 0.930 test: 0.932



学習データと検証データのスコアは同程度
過学習の心配は少ない

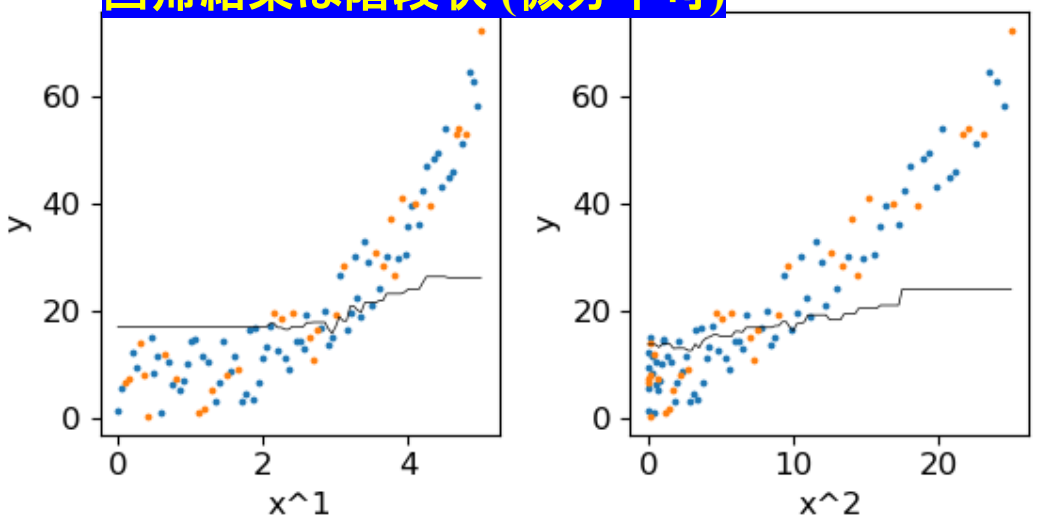
ランダムフォレスト回帰

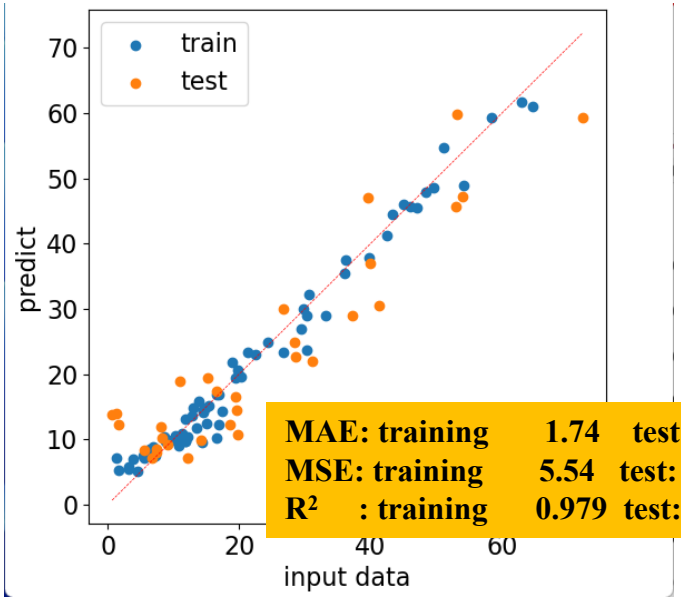
model=rfr　# ランダムフォレスト回帰



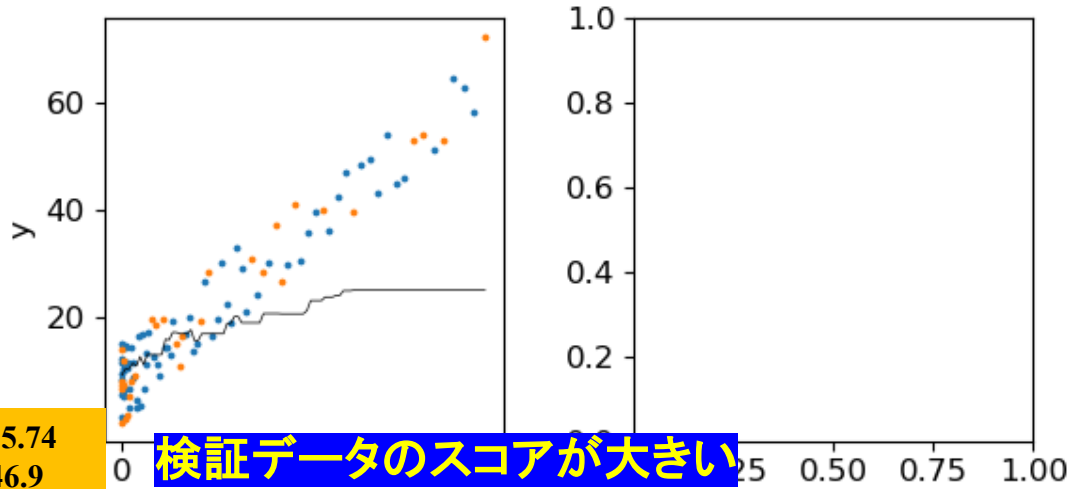
ノイズを再現しようとする

ランダムフォレストは分類器なので、回帰結果は階段状 (微分不可)





MAE: training	1.74	test:	5.74
MSE: training	5.54	test:	46.9
R^2 : training	0.979	test:	0.855



検証データのスコアが大きい

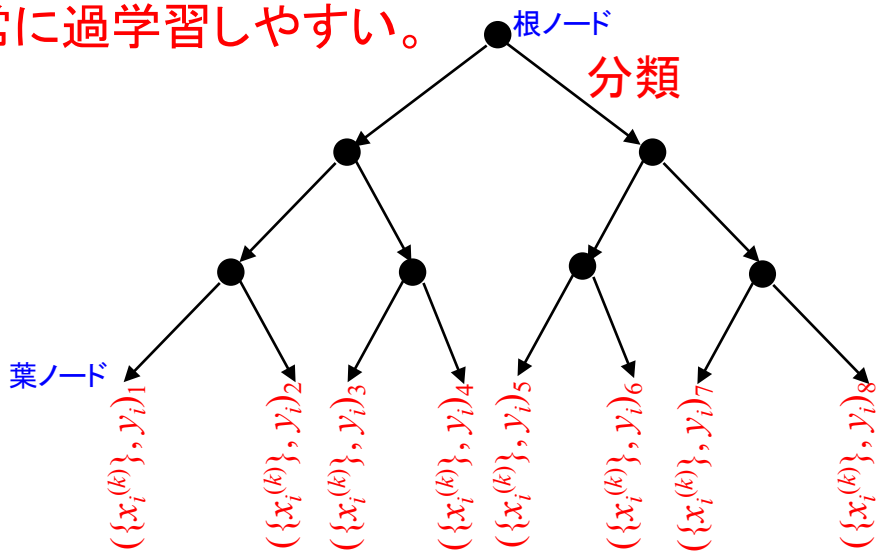
=> 過学習

ランダムフォレスト

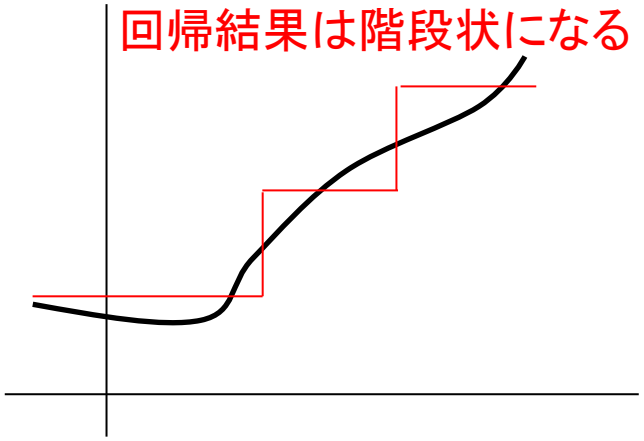
決定木 (Decision Tree):

データを記述子間の距離に応じて2グループに分類することを繰り返す分類器。
分類を何度繰り返すかを「深さ」と呼ぶ。

非常に過学習しやすい。



数値を分類して回帰に使える
回帰結果は階段状になる



ランダムフォレスト (Random Forest): 多くの小さい決定木でアンサンブル学習

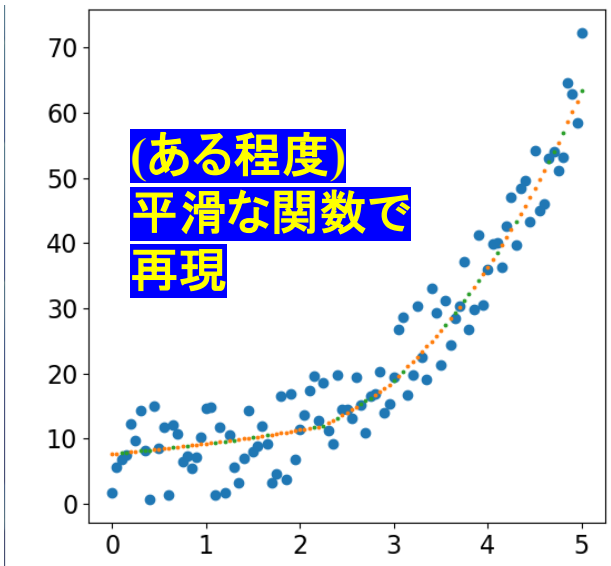
1. データをN個のサブデータセットに分割
   (ブートストラップ法: 重複を許してランダムに抽出)
2. それぞれのグループでランダムに記述子を選択
3. それぞれのグループで決定木による評価を行う (バギング)

N個の結果を 平均 (回帰) あるいは 多数決 (分類) で最終評価を出す。

=> 過学習を抑制。グループと記述子を減らしているので高速

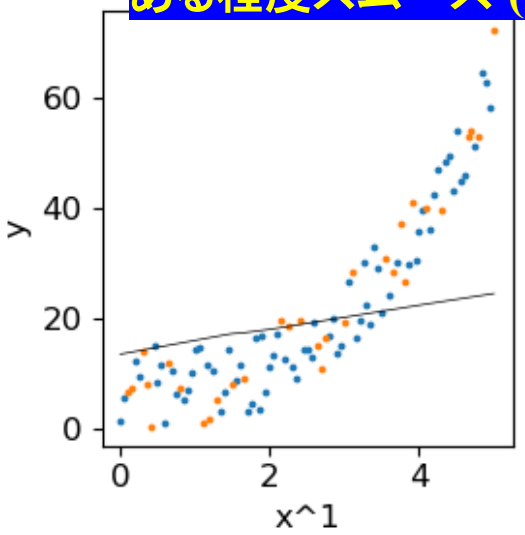
3層パーセプトロン回帰

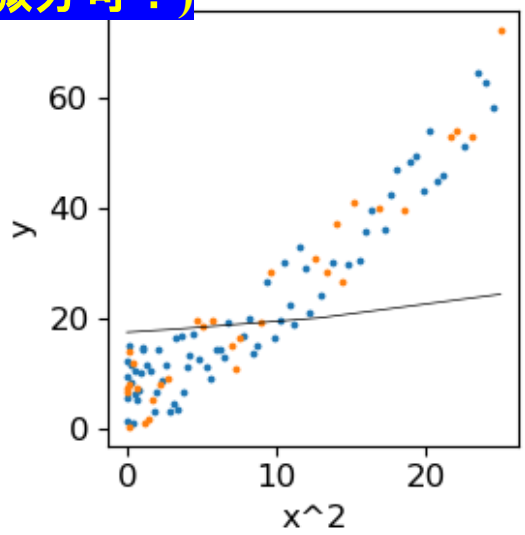
model=mlp　# 多層パーセプトロン回帰
5ノード × 3層、ReLU

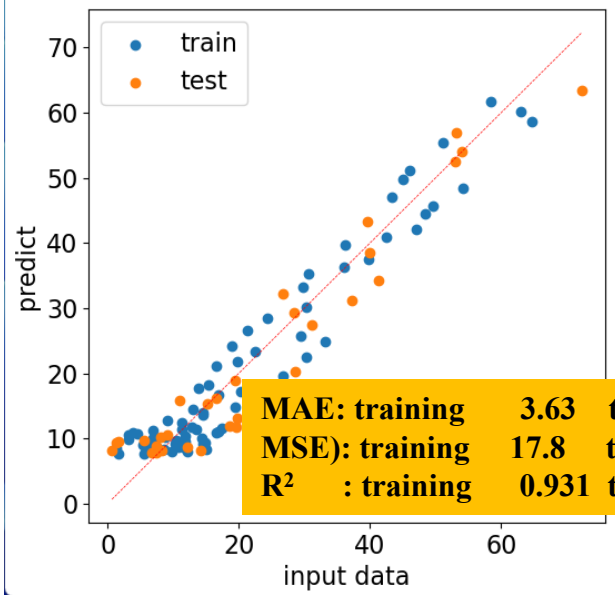


(ある程度)
平滑な関数で
再現

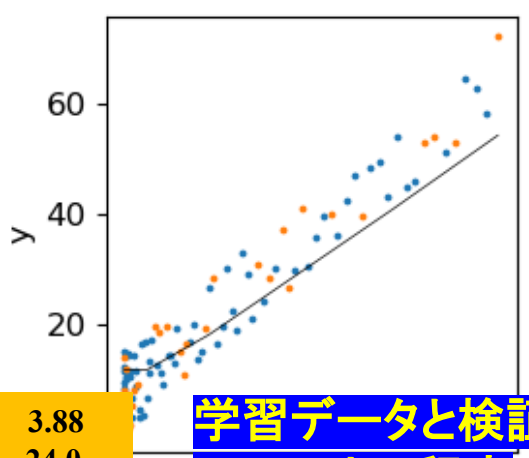
回帰結果は連続
ある程度スムーズ (微分可？)

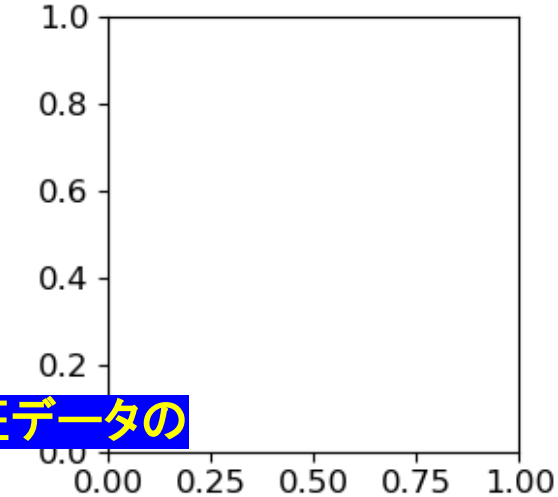






MAE: training　3.63　test:　3.88
MSE): training　17.8　test:　24.0
R^2　: training　0.931　test:　0.926





学習データと検証データの
スコアは同程度
　過学習の心配は少ない

多層パーセプトロン (ニューラルネットワークの1種)

パーセプトロン: 複数の入力データ $\{x_i\}$ に対して一つの値 y を出力する関数

(神経網のニューロンに対応)

1. 変換 $y = \sum w_i x_i + b$ (w_i: 重み　b: バイアス)
2. 非線形性を取り入れたり、分類器に使うため (出力を0か1にする)

活性化関数を使う $y = \varphi(\sum w_i x_i + b)$

identity	$\varphi(x) = 1$	
ReLU	$\varphi(x) = 0 \ (x < 0)$	
	$= x \ (x \geq 0)$	微分不可
tanh (シグモイド関数)	$\varphi(x) = \tanh \alpha x$	分類可能
ロジスティック関数 (シグモイド関数)	$\varphi(x) = \frac{1}{1+\exp(-\alpha x)} \frac{\exp(\alpha x/2)+1}{2}$	分類可能
ソフトサイン関数 (シグモイド関数)	$\varphi(x) = \frac{x}{1+\alpha\lvert x\rvert}$	微分不可

＊ シグモイド関数 $\lim_{x\to-\infty} f(x) = a, \lim_{x\to\infty} f(x) = b, \lim_{x\to\pm\infty} f'(x) = 0$

a と b の 2値に漸近するので、分類器に使える

機械学習の例: 一般的な入力フォーマット

一般的な入力ファイル: 複数の記述子と目的関数

ファイル: [tkProg]¥tkprog_tutorial¥regression¥boston.xlsx

プログラム: scikit-regressions.py

MEDV	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
24	0.00632	18	2.31	0	0.538	6.575	65.2	4.09	1	296	15.3	396.9	4.98
21.6	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.9	9.14
34.7	0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03
33.4	0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94
36.2	0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.9	5.33
28.7	0.02985	0	2.18	0	0.458	6.43	58.7	6.0622	3	222	18.7	394.12	5.21
22.9	0.08829	12.5	7.87	0	0.524	6.012	66.6	5.5605	5	311	15.2	395.6	12.43
27.1	0.14455	12.5	7.87	0	0.524	6.172	96.1	5.9505	5	311	15.2	396.9	19.15

ラベルの制御子 (bayes_gp_plain.pyに類似) は使えるが、
この場合はデフォルトで読み込めるので不要

第１列: 目的関数

第２列以降: 記述子

Boston: ボストンの住宅価格のサンプルデータ

住宅価格の中央値（1000ドル単位）(MEDV): 目的関数
犯罪率（CRIM）
広い家の割合（ZN）
非小売業の割合（INDUS）
チャールズ川の近くか（CHAS）
一酸化窒素濃度（NOX）
平均部屋数（RM）
家の古さ（AGE）
主要施設への距離（DIS）
高速道路へのアクセス性（RAD）
固定資産税（TAX）
生徒と先生の比率（PTRATIO）
黒人の割合（B）
低所得者の割合（LSTAT）

機械学習の例: Bostonデータで試す

箱ひげ図

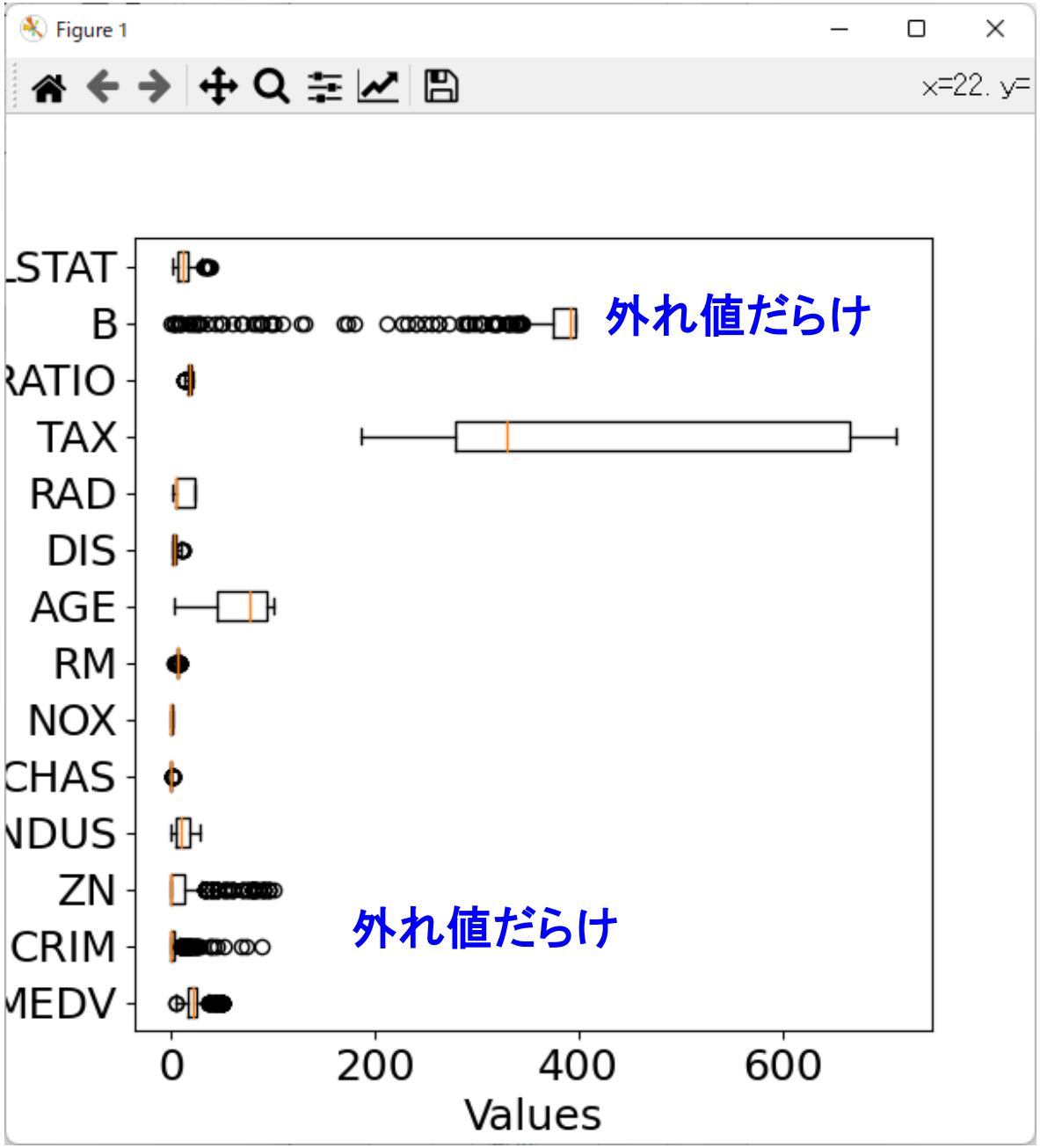


Figure 1
x=22. y=
LSTAT
B
RATIO
TAX
RAD
DIS
AGE
RM
NOX
CHAS
NDUS
ZN
CRIM
MEDV
外れ値だらけ
外れ値だらけ
0
200
400
600
Values

分布図とヒストグラム

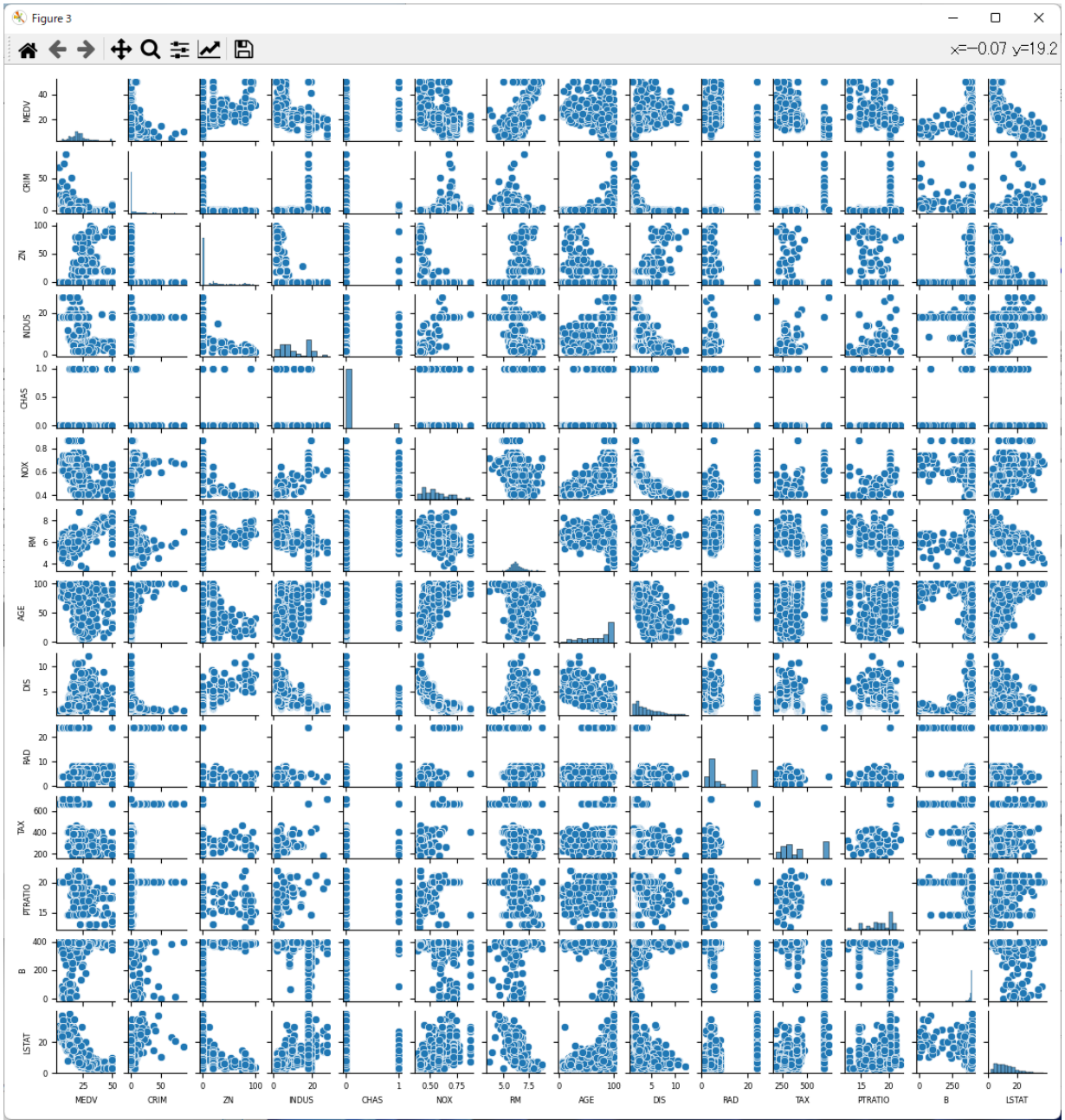
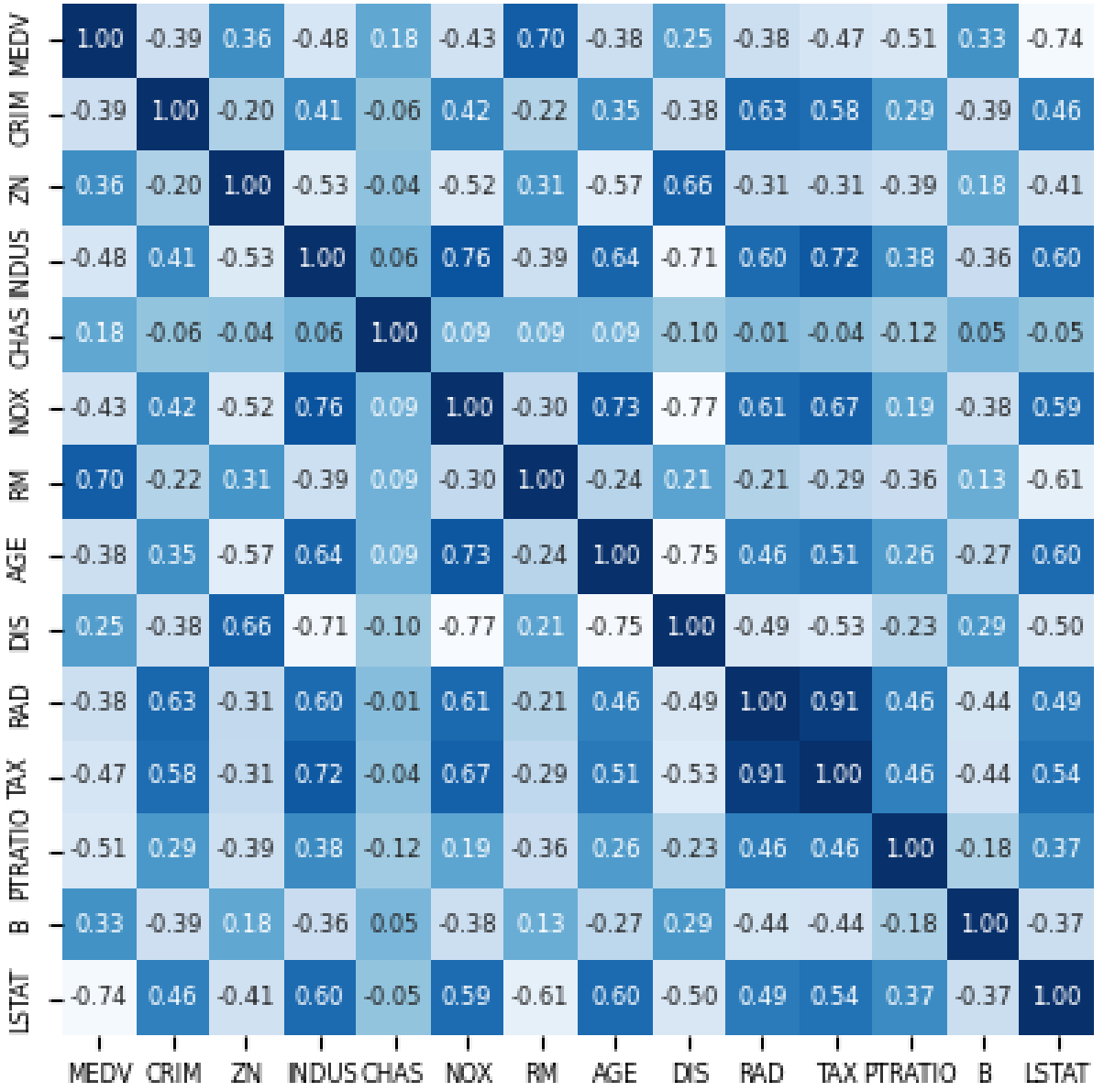


Figure 3
x=−0.07 y=19.2
MEDV
CRIM
ZN
INDUS
CHAS
NOX
RM
AGE
DIS
RAD
TAX
PTRATIO
B
LSTAT

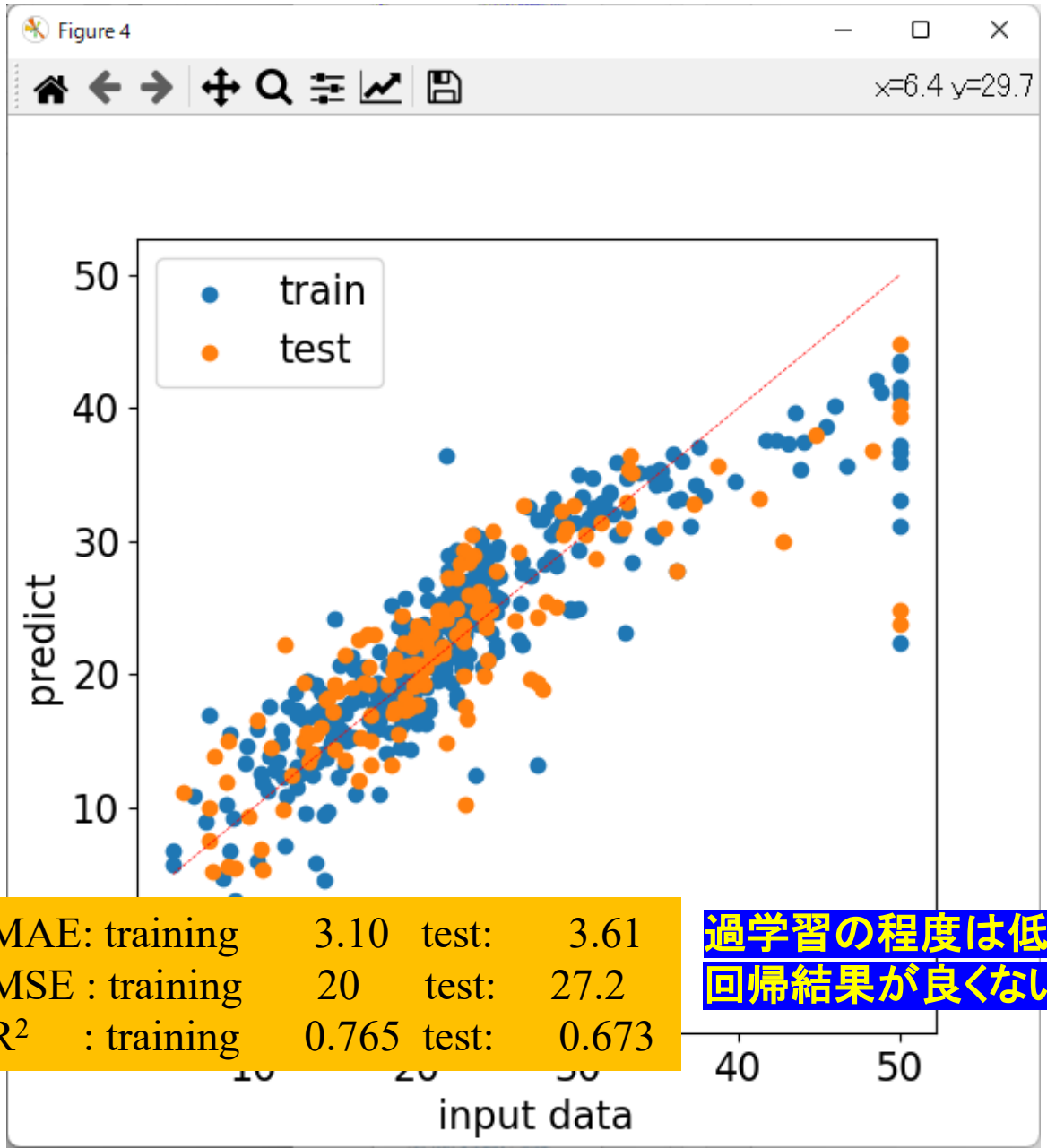
ヒートマップ: 相関係数

住宅価格の中央値（1000ドル単位）(MEDV)
犯罪率（CRIM）
広い家の割合（ZN）
非小売業の割合（INDUS）
チャールズ川の近くか（CHAS）
一酸化窒素濃度（NOX）
平均部屋数（RM）
家の古さ（AGE）
主要施設への距離（DIS）
高速道路へのアクセス性（RAD）
固定資産税（TAX）
生徒と先生の比率（PTRATIO）
黒人の割合（B）
低所得者の割合（LSTAT）



入力値 vs 予測値

線形回帰

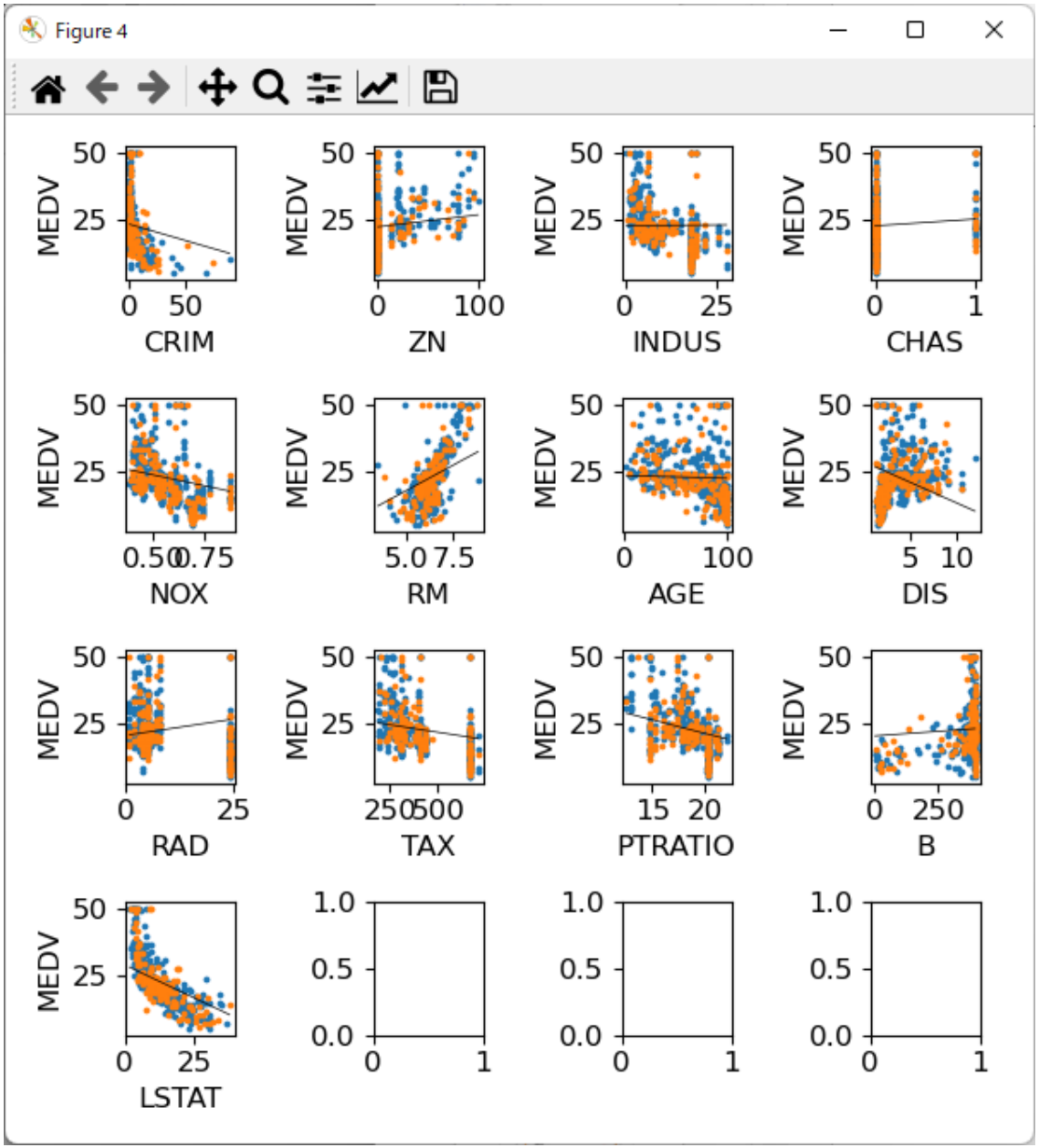


MAE: training	3.10	test:	3.61
MSE : training	20	test:	27.2
R^2 : training	0.765	test:	0.673

過学習の程度は低いが、
回帰結果が良くない (R_2が低い)

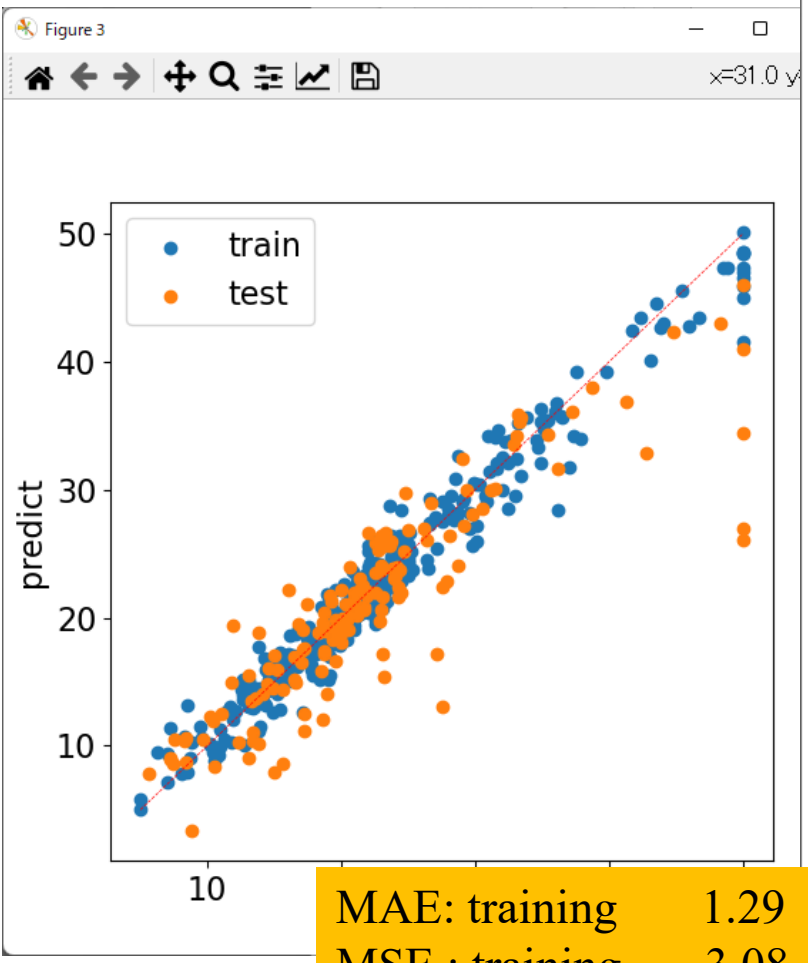
各記述子 vs 目的関数

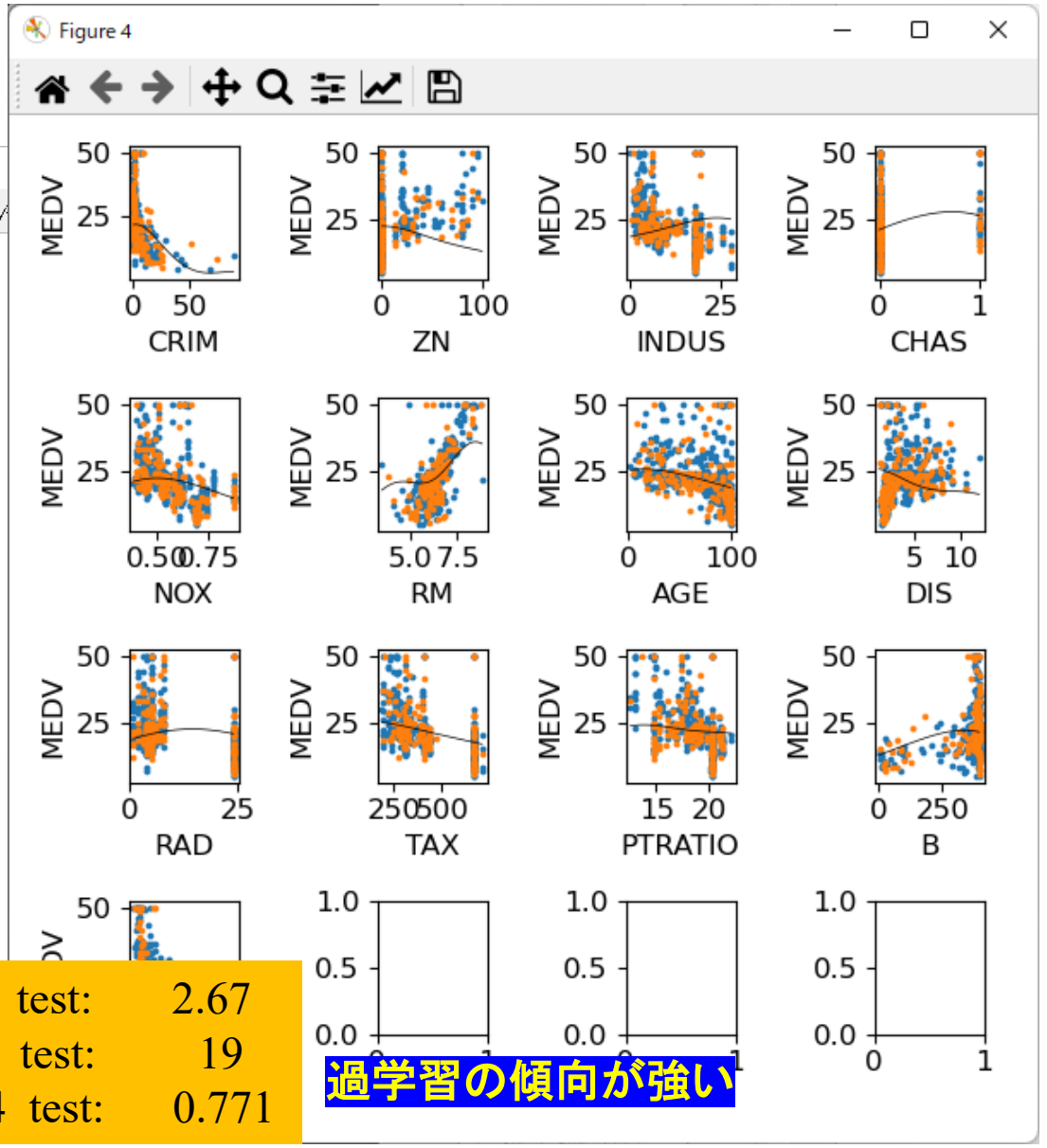
線形回帰



ガウス過程回帰

Model=gpr　# ガウス過程回帰



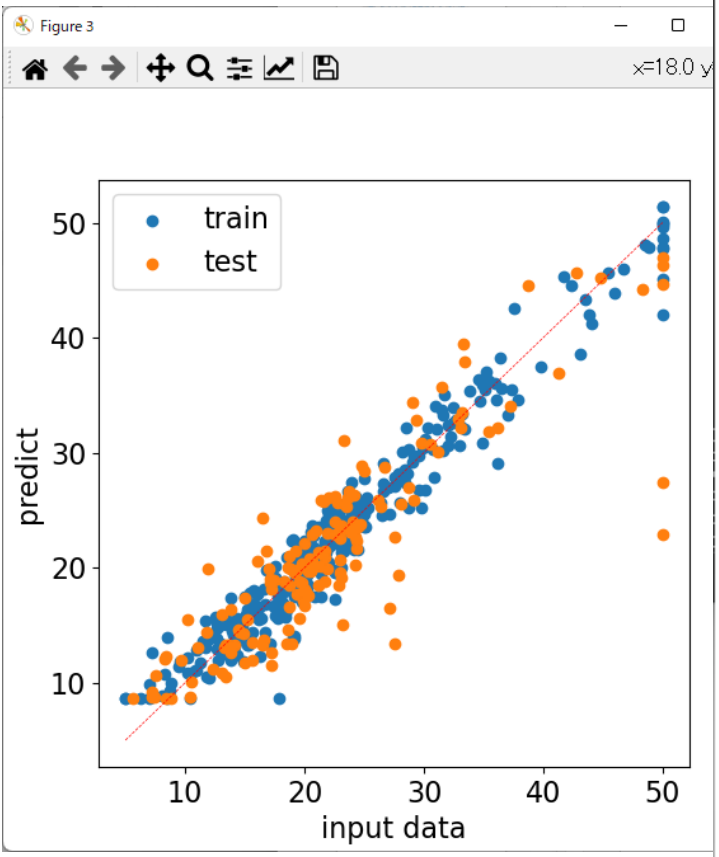


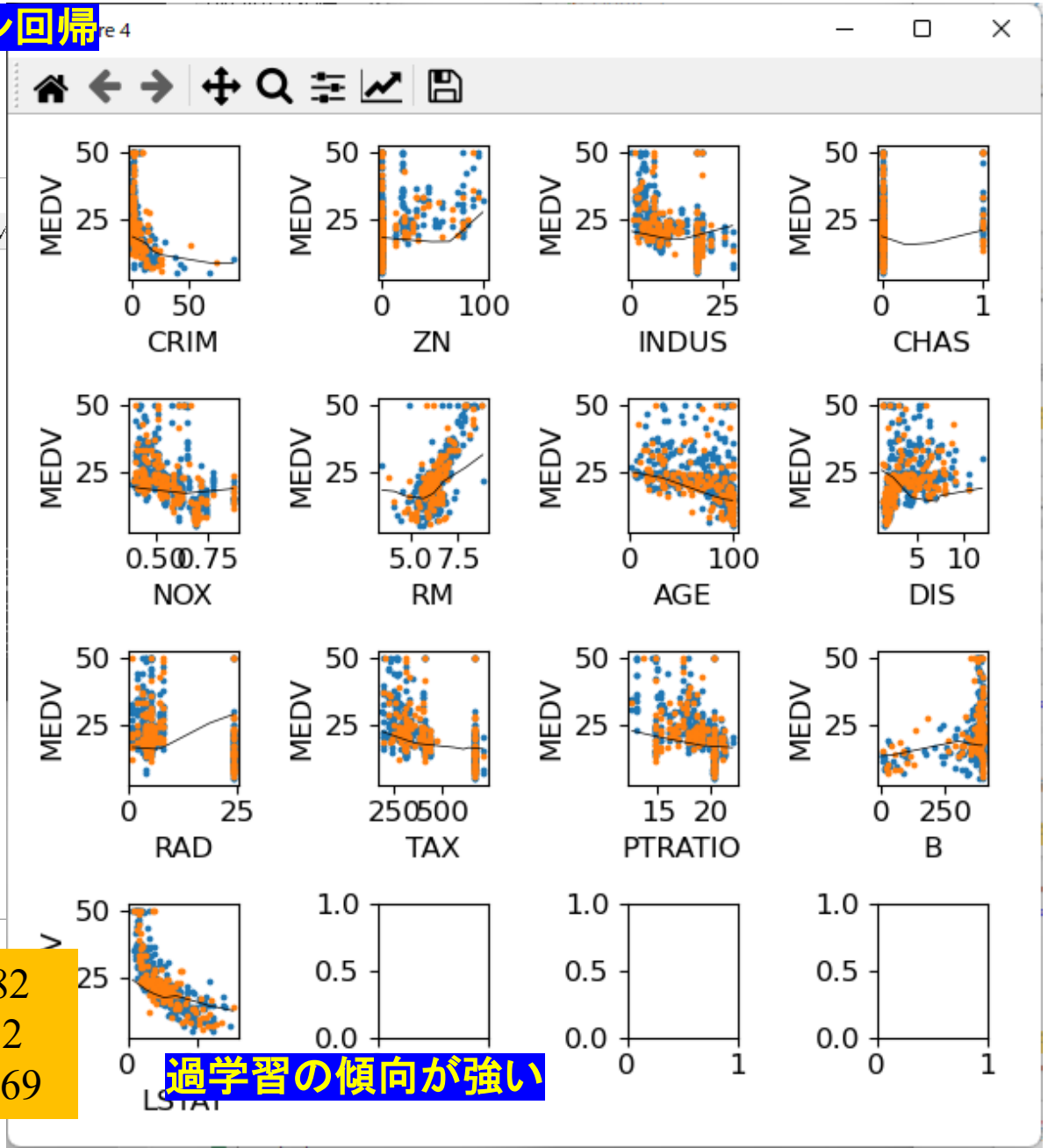
MAE: training	1.29	test:	2.67
MSE : training	3.08	test:	19
R^2 : training	0.964	test:	0.771

過学習の傾向が強い

多層パーセプトロン回帰

Model=mlp　# 多層パーセプトロン回帰
5ノード5層





MAE: training	1.50	test:	2.82
MSE : training	4.01	test:	19.2
R^2 : training	0.959	test:	0.769

過学習の傾向が強い

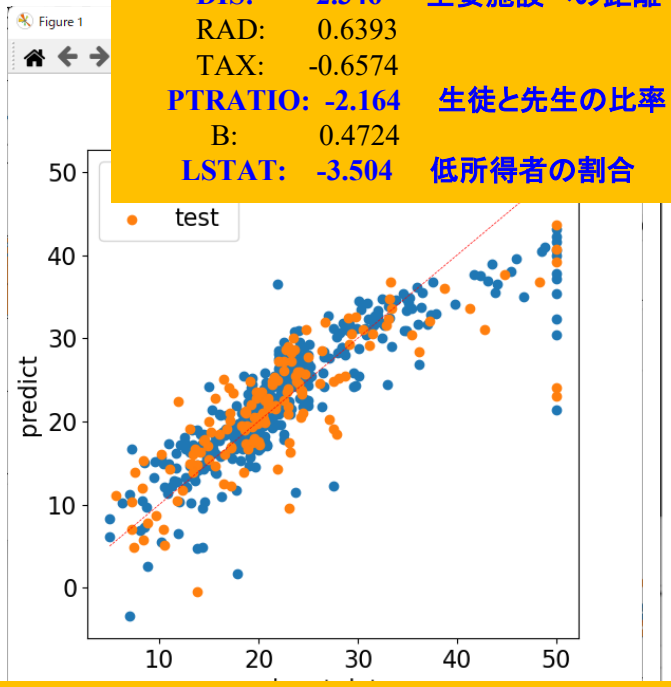
LASSO回帰

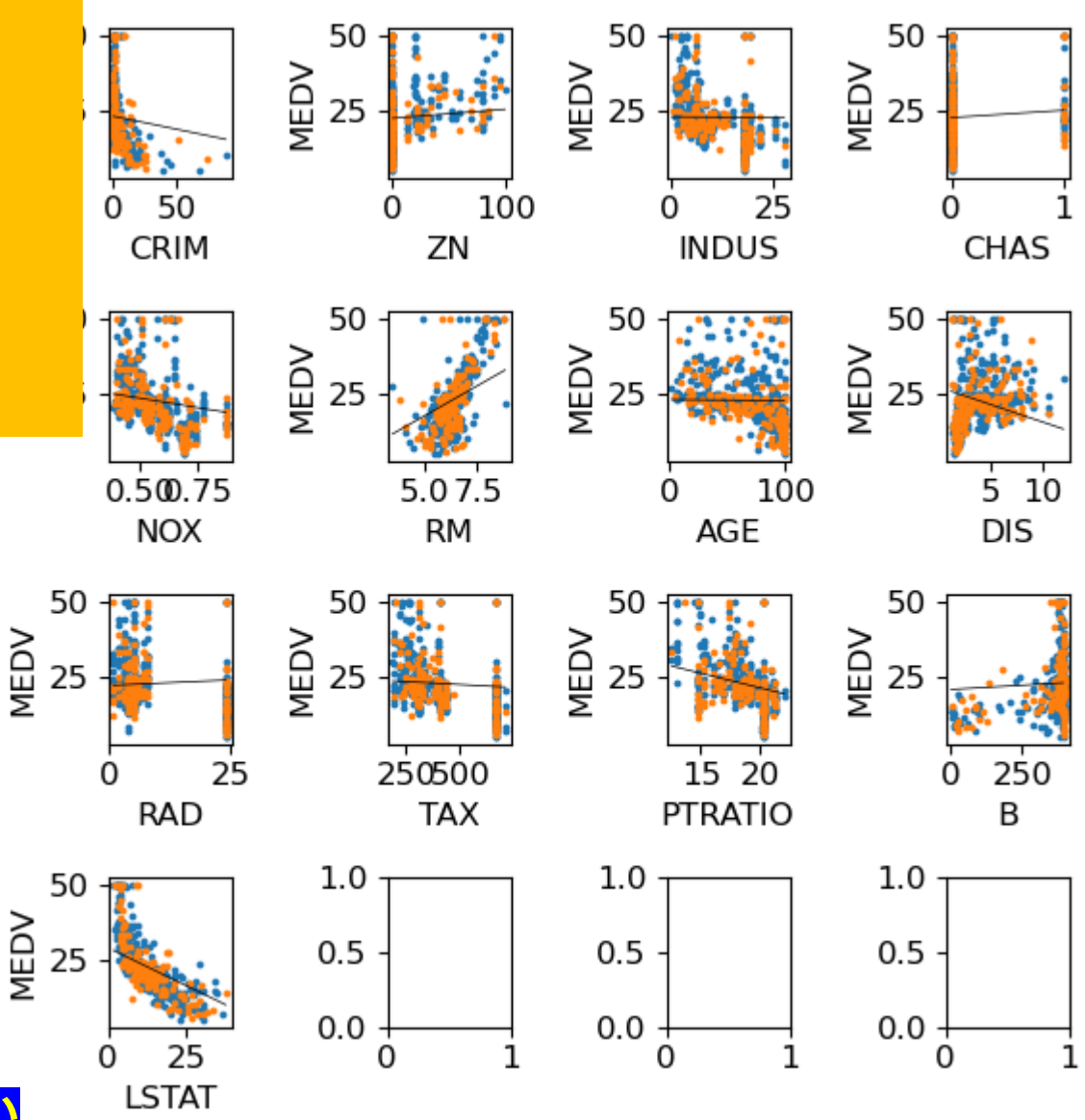
Model=lasso

α=0.1

Parameters:
intercept: 22.7454802259887
coefficients

CRIM:	-0.7133	
ZN:	0.7016	
INDUS:	**-0.06951**	**非小売業の割合**
CHAS:	0.604	
NOX:	-1.444	
RM:	**2.834**	**平均部屋数**
AGE:	**-0.08965**	**家の古さ**
DIS:	**-2.346**	**主要施設への距離**
RAD:	0.6393	
TAX:	-0.6574	
PTRATIO:	**-2.164**	**生徒と先生の比率**
B:	0.4724	
LSTAT:	**-3.504**	**低所得者の割合**





MAE: training 3.11 test: 3.62
MSE : training 20.4 test: 28.3
R^2 : training 0.759 test: 0.660

過学習の傾向が強い

LASSO回帰: αの影響

Model=lasso

α=0
intercept: 22.7
coefficients

CRIM:	-1.012
ZN:	1.05
INDUS:	0.0792
CHAS:	0.619
NOX:	-1.874
RM:	2.705
AGE:	-0.2796
DIS:	-3.098
RAD:	2.097
TAX:	-1.886
PTRATIO:	-2.261
B:	0.5826
LSTAT:	-3.44

α=0.1
intercept: 22.7
coefficients

CRIM:	-0.7133
ZN:	0.7016
INDUS:	-0.06951
CHAS:	0.604
NOX:	-1.444
RM:	2.834
AGE:	-0.08965
DIS:	-2.346
RAD:	0.6393
TAX:	-0.6574
PTRATIO:	-2.164
B:	0.4724
LSTAT:	-3.504

α=0.3
intercept: 22.7
coefficients

CRIM:	-0.439
ZN:	0.08883
INDUS:	**-0**
CHAS:	0.4736
NOX:	-0.7514
RM:	2.958
AGE:	**-0**
DIS:	-1.021
RAD:	**-0**
TAX:	-0.1392
PTRATIO:	-2.156
B:	0.3785
LSTAT:	-3.472

α=0.5
intercept: 22.7
coefficients

CRIM:	-0.2981
ZN:	**0**
INDUS:	**-0**
CHAS:	0.338
NOX:	**-0**
RM:	2.958
AGE:	**-0**
DIS:	-0.1101
RAD:	**-0**
TAX:	-0.2205
PTRATIO:	-2.015
B:	0.3117
LSTAT:	-3.393

ハイパーパラメータ
検定
カテゴリー変数

まだ機械学習は終わっていない

回帰モデルは得られたとする (検証もパス、予測性能も確認した)

やらないと意味のないこと: 後処理
- 回帰モデルに記述子を入力して予測値を計算
- 可視化
- 分析

その他、今後の対応予定
- 前処理: データの加工
  - 欠落データの処理
  - カテゴリー変数の量的変数化
  - 記述子、データの選択(分類、クラスター化)
- 解析: 複数目的関数への対応

回帰と機械学習

機械学習: 記述子、データが非常に多い

- 過剰適合の危険性が高い
- ブラックボックスでも、予測性能が高ければよい

過剰適合していないか、予測性能はあるかを検証 (Validation)

1. 既知データを 学習データ (training) と 検証データ (test) に分ける
2. 学習データでモデルを作る
3. 学習データと検証データの予測値 (prediction) と入力値から、評価を行う
4. 評価は MAE (Mean Absolute Error), MSE (Mean Squared Error), R^2 (決定係数) などで行う。

$MAE = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - f(\{x_i^{(k)}\})\right|$ 小さいほどいい

$MAE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\{x_i^{(k)}\})\right)^2$ 小さいほどいい

$RMAE = \mathrm{sqrt}(MAE)$ 小さいほどいい

$R^2 = 1 - \frac{\sum_{i=1}^{n}\left(y_i - f(\{x_i^{(k)}\})\right)^2}{\sum_{i=1}^{n}(y_i - \langle y_i \rangle))^2}$ 1に近いほどいい

ハイパーパラメータ最適化と検証

- **ハイパーパラメータ最適化** sklearn.model_selection.GridSearchCV()
  ハイパーパラメータを変えながら score (評価指標) を計算し、
  最適パラメータを求める

- 検証: ホールドアウト法
  学習データで学習したモデルを使い、検証データで予測して score を求める

- １通りの検証では足りないので、複数の検証を組み合わせる
  ・ホールドアウト法
  学習・検証データに分割する際の乱数のシードを設定しなければ、
  run毎にデータが変わる => 複数回 run して score や回帰結果を比較する

  ・交差検証 (Cross Validation)
  ・k-分割交差検証 sklearn.model_selection.cross_val_score()
  データを k 分割し、１つの組を検証データ、残りを学習データとして、
  k 通りのscoreを求め、socreの分布から評価
  ・分割の仕方はいろいろなバリエーションがありえる

カテゴリー変数 (質的変数)

記述子 (および目的関数) の種類

- 量的変数: 数値で大小関係が定義されている

- 質的変数 (カテゴリー変数): 大小関係がない。
  質的変数の値毎に 0 か 1 かを割り当てる
  “質的変数の異なる値の和” だけ記述子が増える

連続関数の回帰 (線形回帰、MLP、GPなど) にカテゴリー変数を入れると:

・カテゴリー変数同志を連続関数でつなごうとするが、そもそもそんな関係はない
  => 問題を複雑にするだけ。
  カテゴリー変数の組ごとに別の回帰を行っても結果は同じ
  原子種などを入れる場合、
  電子配置、族、周期などで置き換えれば量的変数として扱える

カテゴリー変数が多い場合は、すべての組み合わせの回帰を人力でやるのは無理がある ＝＞ 記述子に入れる。

・カテゴリー変数を入れる場合は分類器 (ランダムフォレストなど) ベースの方が整合性が高い

カテゴリー変数: ダミー変数化

質的変数を量的変数に変換する

質的変数 → ダミー変数化

Index	時間帯
0	朝
1	昼
2	夕方
3	夜

Index	朝	昼	夕方	夜
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

注意: 多重共線性

ダミー変数4つのうち独立なのは3つだけ

朝、昼、夕方のいずれもが0であれば、
夜であると決まってしまう

Pandasでデータフレーム df をダミー化して df_dum に返すプログラム

```
df_dum = pandas.get_dummies(data = df,
                            drop_first = True)
```

ダミー変数としては

昼、夕方、夜 の3つのみを使う