

第3章 線形回帰

観測点が与えられたときに、それを構成する複数の変数間の関係をあらわすことは、歴史的に古くから研究されてきたテーマである。ここでは起点として、 x, y 2つの変数に対して2組の観測点が与えられた場合に、それを結ぶ直線を引くために連立方程式を解くところから始める。そしてそこから、観測点を一つ増やして、連立方程式では解けないケースにして、最小二乗法で解いたのち、数値微分法、誤差逆伝播法、と解法を発展させていく。そして、線形回帰を軸にして機械学習の原理を、それを支える数学の基礎とともに発展的に学んでいくことにする。プログラムを作成して動かし、結果を確かめながら進めることは理解の助けになるので、適宜 python のプログラムのコードを掲載する。実際にプログラムを動かしながら学んでいくことで、使える知識にして欲しい。

[関数と変数について]

関数の厳密な定義は専門書に譲ることとするが、これからの説明で使う用語として、関数と変数について、それに割り当てる記号とともに次のように定めておく。

数値の組が与えられたとする。そしてその数値の組の一方から一方への何らかの対応関係が想定されていて、その一方を説明変数(独立変数)としたとき、それと対応するもう一方を目的変数(従属変数)として、その対応関係を関数とする。このとき、説明変数を x 、目的変数を y 、関数を f であらわして $y = f(x)$ 。そしてここで、観測値ないしは測定値として与えられる目的変数の値については、関数の値 $f(x)$ と区別するために、 y のかわりに t であらわすことにする。さらに必要に応じて、複数の説明変数をまとめたベクトルないしは行列を X のように太字の大文字であらわすこととする。なお、ベクトルなのか行列なのかということとを区別する必要がある場合については、その点を言及することとして、以下の説明ではベクトルも行列も太字の大文字であらわすこととする。

3.1 連立方程式

説明変数(独立変数) x と目的変数(従属変数) t の組が $[3, 2], [5, 3]$ で与えられたとする。つまり $x = 3$ のとき $t = 2$ 、 $x = 5$ のとき $t = 3$ という値が与えられた。

そしてこのとき、この2点を結ぶ直線が引けることは明らか。

そこで切片 w_0 傾き w_1 の直線を、

$$y = f(x) = w_0 + w_1 x \quad (3.1.1)$$

とおく。(こうおいただけでは、直線が2点の上を通るかどうかはわからないが...)

与えられた x と t にしたがって表にすると、

表 3.1.1

X	T	$Y = F(X)$
3	2	$w_0 + 3w_1$
5	3	$w_0 + 5w_1$

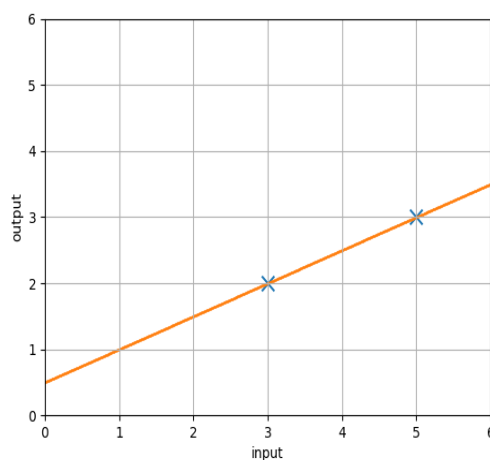


図 3.1.1 2点を通る直線

X と T はグラフ上では、図中に×で示すようになる。このとき、直線 $y = w_0 + w_1x$ がこの2点を通るようにひかれているとすれば、この2点の上では、 $Y = T$ であり、次式が成り立つ。

$$\begin{cases} w_0 + 3w_1 = 2 \\ w_0 + 5w_1 = 3 \end{cases} \quad (3.1.2)$$

この関係は w_0 と w_1 を変数とする連立方程式とみることができる。そして簡単に解くことができる。

2つ目の式から1つ目の式を引いて、

$$\begin{aligned} 2w_1 &= 1 \\ \therefore w_1 &= \frac{1}{2} = 0.5 \end{aligned} \quad (3.1.3)$$

1つ目の式に代入して、

$$\begin{aligned} w_0 + 3 \times \frac{1}{2} &= 2 \\ \therefore w_0 &= 2 - \frac{3}{2} = \frac{1}{2} = 0.5 \end{aligned} \quad (3.1.4)$$

直線が与えられた2点[3, 2], [5, 3]の上を通るときの w_0 と w_1 の値がわかった。

説明変数と目的変数の関係をあらわす式は次のようになる.

$$y = f(x) = w_0 + w_1x = 0.5 + 0.5x \quad (3.1.5)$$

簡単に答えは出せるが、ここでは敢えて元の連立方程式を行列を使ってあらわすと、

$$\begin{pmatrix} w_0 + 3w_1 \\ w_0 + 5w_1 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 1 & 5 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \quad (3.1.6)$$

$$\therefore \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 1 & 5 \end{pmatrix}^{-1} \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 2.5 & -1.5 \\ -0.5 & 0.5 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \quad (3.1.7)$$

あらためてここで現れる行列やベクトルを、

$$\Phi = \begin{pmatrix} 1 & 3 \\ 1 & 5 \end{pmatrix} \quad W = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \quad T = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \quad (3.1.8)$$

とおけば、

$$Y = \Phi W \quad (3.1.9)$$

であり、元の方程式は、

$$\Phi W = T \quad (3.1.10)$$

とあらわすことができ、その解は、

$$W = \Phi^{-1}T \quad (3.1.11)$$

である.

W はすでに求まっているので検算しておこう.

$$Y = \Phi W = \begin{pmatrix} 1 & 3 \\ 1 & 5 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix} \quad (3.1.12)$$

もちろん、次のように $y = f(x) = w_0 + w_1x = 0.5 + 0.5x$ で計算するのと同じである.

$$\begin{cases} y_0 = 0.5 + 0.5x_0 = 0.5 + 0.5 \times 3 = 2 \\ y_1 = 0.5 + 0.5x_1 = 0.5 + 0.5 \times 5 = 3 \end{cases} \quad (3.1.13)$$

[基底関数と線形結合]

線形結合を平たく言えば、与えられたものを何倍かして足すという意味である. 与えられたものにあたる、何倍かして足される元となるものを基底関数とすれば、基底関数の線形結合ということになる. そしてその何倍かを重みという. 本節では、 x と y がグラフで表現すれば、切片 w_0 傾き w_1 の直線となっているものを取り上げてきた. すなわち、式 3.1.1 に示すと

おり、関数 $y = f(x) = w_0 + w_1x$ である。これを基底関数とその線形結合という観点で眺めなおすとどうなるだろうか？ まず基底関数だが、 x の 0 乗と 1 乗を並べてベクトルにして、

$$\Phi(x) = (\varphi_0(x), \varphi_1(x)) = (x^0, x^1) = (1, x) \quad (3.1.14)$$

とおくことができる。いっぽう重みは、式 3.1.8 でベクトルにしたものをそのまま用い、

$$\mathbf{W} = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \quad (3.1.8 \text{ 一部再掲})$$

重みと基底関数の対応に気をつけて、 $\Phi(x)$ の線形結合で $f(x)$ をあらわして、式 3.1.5 の関数は次式となる。

$$y = f(x) = \Phi(x) \mathbf{W} = w_0\varphi_0(x) + w_1\varphi_1(x) = w_0 + w_1x \quad (3.1.15)$$

重みの要素の数、すなわち重みのベクトルの次数は、基底関数の要素の数、すなわち基底関数のベクトルの次数と同じだが、これをここでは線形結合の次数と呼ぶ。したがって、ここで取り上げた直線は、べき乗の 2 次の線形結合ということになる。

ここであらためて、式 3.1.8 の Φ を見直してみると、これは $\Phi(x)$ に説明変数 x の 2 つの観測値を入れて、これを行方向に並べたものである。そして同式 3.1.8 の \mathbf{T} は目的変数の観測値の組である。つまり式 3.1.9～式 3.1.11 は、複数の観測値として得られる説明変数と目的変数の組から、両者の関係性を線形結合の重み \mathbf{W} として導き出す過程にほかならない。この基本的な考え方は、次節以降で扱う観測値が必ずしも関数の値に一致しないような場合においても同様である。

また本節で扱っている、べき乗の 2 次の線形結合という単純な線形結合で表現できることは、せいぜい平面上の様々な直線に過ぎないけれども、逆に考えるならば、このように単純な線形結合でも平面上のあらゆる直線があらわせるのである。だから平面に直線状に並ぶ現象であれば何でも 2 つの重みの組でその関係性をあらわせるのである。世の中の様々な事象も、単純な事柄が折り重なっているもの、すなわち、線形結合として表すことができる場合が決して少なくはなく、それはとりもなおさずこの章のテーマである線形回帰によって、世の中の多くの事柄を表すことができる可能性を示唆するものである。

この直線の基底関数について、以下に python でクラスとしてインプリメントしておく。

リスト 3.1.1 : 基底関数 $\phi(x) = (1, x)$ の定義

```
# -- 基底関数: 直線(1 次式)基底 Rectilinear Basis [1, x]
class RectilinearBasis:
    def __init__(self):
        print('Initialize RectilinearBasis')

    def __call__(self, x):
        y = np.array([x**0, x])          # x**0=1 だが、x がベクトルのため
        y = y.T                          # 軸 0: データ x、軸 1: 基底次元
        return y
```

このクラスは、`__init__()`と`__call__()`の2つの特殊メソッドから構成される。

`__init__()`メソッドでは、`RectilinearBasis`が基底関数となることを `print` 文で表示している。いっぽう、`__call__()`メソッドを定義しているのは、呼出す際の利便性のためである。ここで要となるのは、引数 x に対して `y = np.array([x**0, x])` を作っている部分である。コメントにもある通り、「1」を作るのにわざわざ `x**0` としているが、これは引数 x が複数要素からなるベクトルに対応するためである。この `__call__()`メソッド中で、引数 x には観測値のデータを入力しているので、返回值 y は基底関数に観測値を代入した定数の行列となる。そして返回值は転置して、軸 0: データ x 、軸 1: 基底次元とした。すなわち、返回值 y は形状も含めて ϕ そのものである。

続いて、2 点を通る直線の切片と傾きを求めるプログラムを作っておこう。先にクラスとして定義した直線の基底関数 `RectilinearBasis` を使い、① 連立方程式で説明した手順で逆行列を使って答えを導き出す。

リスト 3.1.2 : 2点を通る直線の切片と傾き

```
import numpy as np
import matplotlib.pyplot as plt
from ufiesia import LCBF

data = np.array([[3, 2], [5, 3]])
X = data[:, 0]    # 入力
T = data[:, 1]    # 正解値

# -- 基底関数に直線を指定 --
phi = LCBF.RectilinearBasis()
Pi = phi(X)
print('Pi\n', Pi)

PiI = np.linalg.inv(Pi)    # PiI = np.linalg.pinv(Pi) # Singular matrix でエラーする場合
print('PiI\n', PiI)

W = np.dot(PiI, T)
print('算出したパラメタ', W)

# -- 入力値とともにグラフを描く --
plt_x = np.linspace(0, max(X)+1)
phi_plt_x = phi(plt_x)
plt_y = np.dot(phi_plt_x, W)

# -- グラフ表示 --
plt.grid()
plt.scatter(X, T, marker='x')
plt.scatter(plt_x, plt_y, marker='.')
plt.xlabel('input')
plt.ylabel('output')
plt.show()
```

この例では、基底関数 RectilinearBasis は、ufiesia の LCBF の中に定義されている。はじめにデータとして [3, 2] と [5, 3] の 2 つを用意し、これをいったん numpy の array にまとめてから説明変数 X と目的変数 T として切り出す。その後で、次の 2 行で基底関数を指定し Φ を得る。

```
phi = LCBF.RectilinearBasis()
Pi = phi(X)
```

ここでは、クラス RectilinearBasis を phi としてインスタンス化した上で、それをそのまま関数のように $Pi = \phi(x)$ により呼び出して実行している。プログラム上は使える文字の制約から、 ϕ を phi とし、 Φ を Pi としている。

あとは、numpy の $\text{linalg.inv}()$ を使って、 Φ の逆行列 Φ^{-1} を得る。そして、numpy の $\text{dot}()$ により、 $W = \Phi^{-1}T$ の計算を行って W を得る。最後に、結果を matplotlib でグラ

```
plt_y = np.dot(phi_plt_x, W)
```

重み

基底関数

とし、その線形結合の式で関数を定義する.

ここで、 x の観測値あるいは測定値が得られて、 x は何らかの値をとるものとする。それらを

$$\mathbf{X} = (x_0, x_1, \dots, x_n) \quad (3.1.19)$$

とする. そうすると x に対応して、

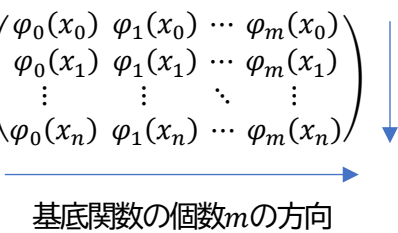
$$\begin{cases} y_0 = f(x_0) = w_0\varphi_0(x_0) + w_1\varphi_1(x_0) + \cdots + w_m\varphi_m(x_0) \\ y_1 = f(x_1) = w_0\varphi_0(x_1) + w_1\varphi_1(x_1) + \cdots + w_m\varphi_m(x_1) \\ \quad \vdots \qquad \qquad \qquad \vdots \qquad \qquad \qquad \qquad \vdots \\ y_n = f(x_n) = w_0\varphi_0(x_n) + w_1\varphi_1(x_n) + \cdots + w_m\varphi_m(x_n) \end{cases} \quad (3.1.20)$$

これらの式をひとまとめにしてあらわすと、

$$\begin{aligned}
\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix} &= \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix} = \begin{pmatrix} w_0\varphi_0(x_0) + w_1\varphi_1(x_0) + \cdots + w_m\varphi_m(x_0) \\ w_0\varphi_0(x_1) + w_1\varphi_1(x_1) + \cdots + w_m\varphi_m(x_1) \\ \vdots \\ w_0\varphi_0(x_n) + w_1\varphi_1(x_n) + \cdots + w_m\varphi_m(x_n) \end{pmatrix} \\
&= \begin{pmatrix} \varphi_0(x_0) & \varphi_1(x_0) & \cdots & \varphi_m(x_0) \\ \varphi_0(x_1) & \varphi_1(x_1) & \cdots & \varphi_m(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_0(x_n) & \varphi_1(x_n) & \cdots & \varphi_m(x_n) \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_m \end{pmatrix} \tag{3.1.21}
\end{aligned}$$

この式にあらわれる、説明変数 x の観測値あるいは測定値を代入した基底関数の並びを、 Φ であらわすこととする。すなわち、

$$\Phi = \begin{pmatrix} \varphi_0(x_0) & \varphi_1(x_0) & \cdots & \varphi_m(x_0) \\ \varphi_0(x_1) & \varphi_1(x_1) & \cdots & \varphi_m(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_0(x_n) & \varphi_1(x_n) & \cdots & \varphi_m(x_n) \end{pmatrix} \tag{3.1.22}$$



とする。そして目的変数 y の並びもまとめて Y とおけば、この関係は次式であらわされる。

$$Y = \Phi W \tag{3.1.9 再掲}$$

説明変数 x の観測値あるいは測定値を代入した基底関数の並びである Φ は以降の説明でしばしば登場するが、これは関数そのものの並んだものではなくて、あくまでも関数に何らかの値が代入された数値が並ぶ行列であることに注意されたし。いずれにせよ、以下の説明では、記号 Φ を使ってこの行列を表すこととする。

注：行列 Φ は、計画行列、配置行列、デザイン行列などと呼ばれている場合も見受けられるが、計画行列という呼び方は実験計画法に由来するようである。重回帰分析を行うような場合、複数項目の観測によって、説明変数は多次元のベクトルとなる。そして複数の観測点に対応して、この説明変数のベクトルを行方向に並べた行列を計画行列と呼ぶ。これは実験で取得するデータの測定項目をデータ方向に並べた表になるから計画行列という呼び方はしっくりくる。いっぽうここで示した Φ は、一つの説明変数 x を、 $0 \sim m$ の複数の基底関数

に代入して得られるベクトルを、データ方向に並べて行列にしたものだから、ちょっと違う。説明変数のベクトルであれ、一つの説明変数を複数の基底関数で変換して生成したベクトルであれ、できた行列の扱いにおいて本質的な違いはないかもしれないが、今一つしっくりこない。なお、基底関数に x を代入して並べた $(\varphi_0(x) \ \varphi_1(x) \ \cdots \ \varphi_m(x))$ については特徴ベクトルと呼ばれるようであり、それならばそれをさらにデータ方向に並べた Φ は特徴行列とでも呼べば良さそうなものだが、残念ながらそう呼んでいる例は見当たらない。

3.2 最小二乗法

表 3.1.1 に、変数と目的変数の組[4, 3]を 1 つ加え、これを表 3.2.1 に示す。

表 3.2.1

X	T	$Y = F(X)$	$E = Y - T$
3	2	$w_0 + 3w_1$	$w_0 + 3w_1 - 2$
5	3	$w_0 + 5w_1$	$w_0 + 5w_1 - 3$
4	3	$w_0 + 4w_1$	$w_0 + 4w_1 - 3$

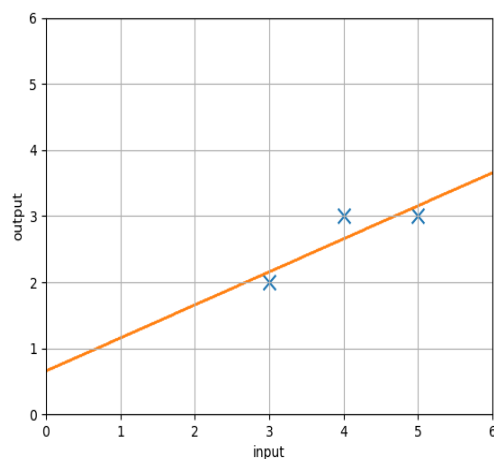


図 3.2.1 3点の近くを通る直線

表 3.1.1 に対して式 3.1.8 を求めたのと同様に、表 3.2.1 に対して、

$$\Phi = \begin{pmatrix} 1 & 3 \\ 1 & 5 \\ 1 & 4 \end{pmatrix} \quad W = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} \quad T = \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} \quad (3.2.1)$$

データの組が 1 つ増えたことで、連立方程式では解けなくなってしまう。グラフ上でもこのことは明らかである。すなわち 3 つの点を通る直線は存在しない。言い換えれば、与えられたデータに対して $y = f(x) = w_0 + w_1x = t$ を同時に満たす W を求めることができないということである。線形結合の計算結果 Y とデータとして与えられた目的変数 T の値は一致しないから、

$$Y = \Phi W \neq T \quad (3.2.2)$$

だが、その隔たり $Y - T$ を

$$E = \begin{pmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} \quad (3.2.3)$$

とおけば、 $Y = T + E$ であって、

$$Y = \Phi W = T + E \quad (3.2.4)$$

という等式が書ける。しかし、 Φ^{-1} は存在せず、やはり連立方程式で解くことはできない。そこでまず、 Y が T にどれだけ近いのか遠いのかの尺度として二乗和誤差を定義する。すなわち、式3.2.3の E を2乗した総和を求める。

$$2L = \sum_{t=1}^n \varepsilon^2 = \sum_{t=1}^n (y - t)^2 \quad (3.2.5)$$

そして二乗和誤差を最小にする W を求める。このために、二乗和誤差の式を変形して W に対して平方完成させる。 $y = f(x) = w_0 + w_1x$ だから

$$\begin{aligned} 2L &= \sum_{t=1}^n (y - t)^2 = \sum_{t=1}^n (w_0 + w_1x - t)^2 \\ &= (w_0 + 3w_1 - 2)^2 + (w_0 + 5w_1 - 3)^2 + (w_0 + 4w_1 - 3)^2 \\ &= w_0^2 + 2w_0(3w_1 - 2) + (3w_1 - 2)^2 + w_0^2 + 2w_0(5w_1 - 3) + (5w_1 - 3)^2 \\ &\quad + w_0^2 + 2w_0(4w_1 - 3) + (4w_1 - 3)^2 \\ &= 3w_0^2 + 24w_0w_1 - 16w_0 + 50w_1^2 - 66w_1 + 22 \\ &= 3\left(w_0^2 + 8w_0w_1 - \frac{16}{3}w_0\right) + 50w_1^2 - 66w_1 + 22 \\ &= 3\left(w_0 + 4w_1 - \frac{8}{3}\right)^2 - 3\left(4w_1 - \frac{8}{3}\right)^2 + 50w_1^2 - 66w_1 + 22 \\ &= 3\left(w_0 + 4w_1 - \frac{8}{3}\right)^2 - 3\left(16w_1^2 - \frac{64}{3}w_1 + \frac{64}{9}\right) + 50w_1^2 - 66w_1 + 22 \\ &= 3\left(w_0 + 4w_1 - \frac{8}{3}\right)^2 + 2w_1^2 - 2w_1 + \frac{2}{3} \\ &= 3\left(w_0 + 4w_1 - \frac{8}{3}\right)^2 + 2\left(w_1 - \frac{1}{2}\right)^2 - 2\left(\frac{1}{2}\right)^2 + \frac{2}{3} \\ &\doteq 3(w_0 + 4w_1 - 2.6667)^2 + 2(w_1 - 0.5)^2 + 0.1667 \end{aligned} \quad (3.2.6)$$

平方完成させた式の値が最小となるのは、2乗の部分が0のときだから、

$$\begin{cases} w_0 + 4w_1 - 2.6667 = 0 \\ w_1 - 0.5 = 0 \end{cases} \quad (3.2.7)$$

これを解いて

$$\therefore \begin{cases} w_1 = 0.5 \\ w_0 = 2.6667 - 4 \times 0.5 = 0.6667 \end{cases} \quad (3.2.8)$$

線形結合の重み \mathbf{W} を算出することが出来た。この \mathbf{W} により説明変数と目的変数の関係をあらわす線形結合の式は次のようになる。

$$y = f(x) = w_0 + w_1 x = 0.6667 + 0.5x \quad (3.2.9)$$

また、求まった \mathbf{W} で目的変数の値を算出すると、

$$\mathbf{Y} = \Phi \mathbf{W} = \begin{pmatrix} 1 & 3 \\ 1 & 5 \\ 1 & 4 \end{pmatrix} \begin{pmatrix} 0.6667 \\ 0.5 \end{pmatrix} = \begin{pmatrix} 2.1667 \\ 3.1667 \\ 2.6667 \end{pmatrix} \quad (3.2.10)$$

もちろん、次のように計算するのと同じである。

$$\begin{cases} y_0 = 0.6667 + 0.5x_0 = 0.6667 + 0.5 \times 3 = 2.1667 \\ y_1 = 0.6667 + 0.5x_1 = 0.6667 + 0.5 \times 5 = 3.1667 \\ y_2 = 0.6667 + 0.5x_2 = 0.6667 + 0.5 \times 4 = 2.6667 \end{cases} \quad (3.2.11)$$

またこのとき二乗和誤差は2乗の部分が0 だから、

$$2L = 0.1667 \quad (3.2.12)$$

いちおう検算しておくと、

$$\mathbf{E} = \mathbf{Y} - \mathbf{T} = \begin{pmatrix} 2.1667 \\ 3.1667 \\ 2.6667 \end{pmatrix} - \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} = \begin{pmatrix} 0.1667 \\ 0.1667 \\ -0.3333 \end{pmatrix} \quad (3.2.13)$$

$$2L = \sum_{i=1}^n \varepsilon^2 = 0.1667^2 + 0.1667^2 + (-0.3333)^2 = 0.1667 \quad (3.2.14)$$

先に求めた2点を通る連立方程式の解では、 $w_0 = 0.5$ 、 $w_1 = 0.5$ だったが、その場合には3点のうち2点は直線上に乗る。そこで、比較のためその場合の二乗和誤差を求めると、

$$\begin{aligned}
2L &= (0.5 + 0.5 * 3 - 2)^2 + (0.5 + 0.5 * 5 - 3)^2 + (0.5 + 0.5 * 4 - 3)^2 \\
&= 0.25 > 0.1667
\end{aligned}
\tag{3.2.15}$$

このように、ここで解とした場合に比べて二乗和誤差が大きくなる。つまり二乗和誤差では悪化=全体として隔たりが大きいという評価になる。

二乗和誤差の平方完成による解法は原理は単純だが、たった 3 組の説明変数と目的変数でも計算が大変で、データが多くなると手計算ではとても計算できなくなる。コンピュータのプログラムを作成して解きたいところだが、逐次状況を判別しながら解いていくやり方は、あまりプログラミングに向いているとは思われない。

注：ここでは二乗和誤差として、

$$2L = \sum_{i=1}^n \varepsilon^2$$

と定義した。ここで L ではなく $2L$ としたのは、後で二乗和誤差として、

$$L = \frac{1}{2} \sum_{i=1}^n \varepsilon^2$$

を定義しなおすため、これと矛盾しないようにするためである。

3.3 勾配の可視化

観測値ないしは測定値に、線形結合の計算結果が近くなるような重み \mathbf{W} を求めてきた。先の「3.2 最小二乗法」では、二乗和誤差 $2L$ を最小にする \mathbf{W} を求めるために、式を変形して平方完成させた。ここからは、基底関数によらずに重み \mathbf{W} を求めるための、より一般的な方法を模索していくことにする。そこでまず、二乗和誤差をあらためて次のように定義する。

$$L = \frac{1}{2} \sum_{i=1}^n \varepsilon^2 = \frac{1}{2} \sum_{i=1}^n (y - t)^2 \quad (3.3.1)$$

これを python の関数としてインプリメントしておこう。

リスト 3.3.1：二乗和誤差

```
def sum_squared_error(y, t):  
    return 1/2 * np.sum(np.square(y - t))
```

ここで定義した二乗和誤差 L を最小にする \mathbf{W} を求めるにあたり、両者の関係がどうなっているのかを見ることができれば、たとえそれが限定的な場合に限られていたとしても、大いに助けになるのではないだろうか？ 幸い、扱ってきた表 3.2.1 のケースは非常に単純であり、 \mathbf{W} は 2 つの要素からなる 2 次元ベクトルである。リスト 3.3.3 のようにデータの数を増やしたとしても、この点は変わらない。そこで \mathbf{W} と二乗和誤差 L で、2+1=3 次元グラフを描いて、両者の関係を見ることにしよう。

まずは3次元グラフを描くプログラムを作る。これをリスト 3.3.2 に示す。

リスト 3.3.2 : 3次元グラフ

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def graph3d(func, rx=(0,1), ry=(0,1), label=None):
    qx = np.linspace(*rx, 100)
    qy = np.linspace(*ry, 100)
    X, Y, Z = [], [], []
    for x in qx:
        for y in qy:
            z = func(x, y)
            X.append(x)
            Y.append(y)
            Z.append(z)

    fig = plt.figure()
    ax = Axes3D(fig)
    if label is not None:
        ax.set_xlabel(label[0])
        ax.set_ylabel(label[1])
        ax.set_zlabel(label[2])
    ax.scatter(X, Y, Z, marker='.', s=2, c='red')
    plt.show()
```

このプログラムは引数で関数 `func` を指定して、`x`, `y` の値の範囲 `rx`, `ry` にわたり `z = func(x, y)` を求めて `z` 軸にあらわす。すなわち、`z = func(x, y)` を 3 軸で描く。3次元のグラフを描画するため、冒頭で `numpy` と `matplotlib.pyplot` に加えて、`mpl_toolkits.mplot3d.axes3d` というライブラリを追加でインポートしている。そしてここでは、`graph3d` という名前の関数を定義して、その関数の中でグラフを描くところまで行う。この関数は必ずしも汎用的である必要はないが、`z` 軸に描く評価対象の関数、`x`, `y` 軸の値の範囲、それにグラフの軸ラベルは指定したいので、これらを `graph3d` の引数 `func`, `rx`, `ry`, `label` で与えるようにした。そして `x`, `y` 軸の値の範囲は引数 `rx`, `ry` に応じて、それぞれ `x` 軸、`y` 軸のデータとして、

```
qx = np.linspace(*rx, 100)
qy = np.linspace(*ry, 100)
```

によって、指定範囲の 100 個の値に展開する。そして 2 重の `for` 文で 1 つずつ取り出される。いっぽう引数で与えられた `func` は `x`, `y` の値の組を引数として与え、その返り値が `z` となる。少々まだらっこしいが、そうしてできた `x`, `y`, `z` の 3 つの値を、それぞれ `X`, `Y`, `Z` のリストにループごとに追加していく。2 重のループを抜けると、 $100 \times 100 = 10000$ 組の 3 軸データが揃うので散布図として描画する。

それではリスト 3.3.2 で作った 3 次元グラフを描く関数を使って、 W と二乗和誤差 L の関係を見るプログラムを作ろう。

リスト 3.3.3 : 重み W と損失 L の関係

```
import numpy as np
from ufiesia import LCBF

data = np.array([[3, 2], [5, 3], [4, 3]])
X = data[:, 0]    # 入力
T = data[:, 1]    # 正解値

# -- 基底関数に直線を指定 --
phi = LCBF.RectilinearBasis()
Pi = phi(X)
print('Pi\n', Pi)

def target_function(*w): # Pi と T は global 変数
    y = np.dot(Pi, w)
    l = LCBF.sum_squared_error(y, T)
    return l

rw0 = 0, 1 #0.6667, 0.6667
rw1 = 0, 1 #0.5, 0.5
label = 'W0', 'W1', 'L'
LCBF.graph3d(target_function, rw0, rw1, label)
```

「リスト 3.1.2: 2 点を通る直線の切片と傾き」とデータを用意して Φ を求めるまでは同じ。プログラム中で Φ を P_i としていることなども同じ。ここではその後に、評価対象となる関数 `target_function` を定義する。この引数は重み w とし、 P_i と T はグローバル変数として関数 `target_function` の外の値をそのまま使う。ここで `*w` のように `*` をつけて指定するのは、 w が w_0 と w_1 の 2 次元ベクトルだからである。この指定によって `y = np.dot(Pi, w)` を行う際に、2 つの要素からなるタプルとして numpy の `dot` 関数に渡される。リスト 3.3.1 で定義した二乗和誤差は基底関数と同じく LCBF の中に置いたので、`LCBF.sum_squared_error` で呼出す。そして得られた二乗和誤差を関数 `target_function` の戻り値としている。最後にリスト 3.3.2 で定義した `graph3d` を実行するが、これも LCBF の中に置いたので、`LCBF.graph3d` で呼出す。その際の引数だが、`graph3d` の `func` には、引数として `target_function` を渡し、値の範囲 `rw0`, `rw1` とグラフの軸ラベルも与えている。重み W の 2 つのベクトルのうちいずれかを固定したい場合には、`rw0 = 0.6667, 0.6667` のように始点終点に同じ値を指定すれば良い。なおコメントアウトした 0.6667 と 0.5 は式 3.2.8 ですでに求めた W の解である。

リスト 3.3.3 のプログラムの実行結果を次に示す。

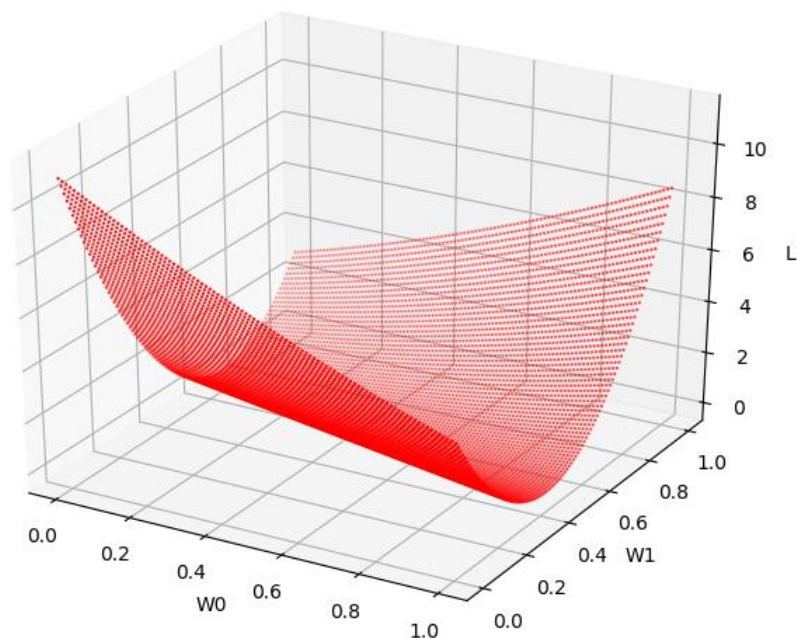


図 3.3.1 W の値と L

図 3.3.1 に示すように、重み W とそれに対する二乗和誤差 L の値の関係を描くことができた。グラフの形は、全体として $w_1 = 0.5$ くらいを底にして、 w_1 がそれよりも大きくても小さくても二乗和誤差 L が大きくなっている。最小二乗法で求めた W の解が $w_1 = 0.5$ であることとよく一致している。いっぽう w_0 の値の変化に対しては、すこしよじれたようになっていることはわかるものの、この図からはっきりとした傾向は読み取れない。

次にリスト 3.3.3 で、 $rw_1 = 0.5, 0.5$ として、 w_1 の値を固定してみることにする。そして、図 3.3.2 を得る。今度は、 w_0 の値が $0.6 \sim 0.8$ のあたりで L が最小となることがはっきりとわかる。やはり最小二乗法で求めた W の解 $w_0 \doteq 0.6667$ と合っているようである。図 3.3.2 のグラフの z 軸= L の値を見ると、 $0.1 \sim 0.7$ で 0.1 刻みの範囲で描かれていて、図 3.3.1 で L が $0 \sim 10$ で 2 刻みだったことと比べると一桁小さい。これで、 w_0 の値に対する二乗和誤差 L の変化の傾向が、図 3.3.1 で読み取れなかったわけがわかった。ともかくも w_0 についても、最小二乗法で求めていた解は、二乗和誤差が最小となるグラフの底のところの W だったことがわかる。

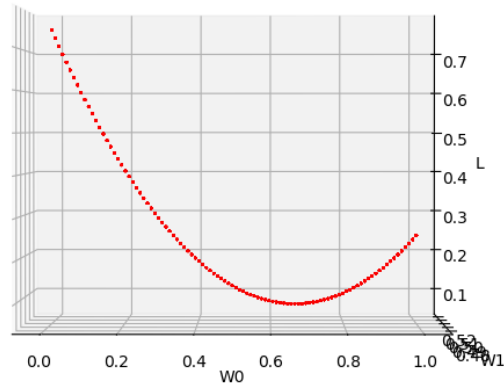


図 3.3.2 w_0 に対する L の変化

さらに念のため、リスト 3.3.3 で、 $rw_0 = 0.6667$, 0.6667 として、 w_0 の値を固定した場合が図 3.3.3 である。やはり $w_1 = 0.5$ で L が最小となっている。

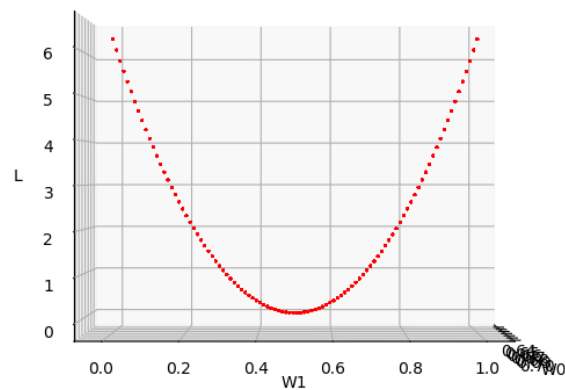


図 3.3.3 w_1 に対する L の変化

これらの観察から、二乗和誤差 L を最小にする重み W が、グラフの底の時の W の値として得られることがわかった。これを、 W の変化に対する L の変化、すなわち勾配という観点でとらえるならば、どうなるだろうか？

たとえば、図 3.3 w_1 に対する L の変化で、 $w_1 = 0$ から見ていくと、はじめは急な勾配で w_1 が大きくなるほど L が小さくなっていき、 $w_1 = 0.5$ に近づくにつれて次第に勾配がゆるくな

っていき、 $w_1 = 0.5$ のあたりでは w_1 の変化に対する L の変化は小さく、すなわち勾配は 0 になって、そこからさらに w_1 が大きくなると、今度は次第に勾配を増しながら、 L が大きくなっていく。これは図 3.3.2 で示す w_0 についても同様で $w_0 \doteq 0.6667$ で勾配が 0 となり、そこから離れると勾配が大きくなる。つまり \mathbf{W} の要素が L を小さくする方向にそれぞれ変化していった、 L がこれ以上小さくならない最下点に達すると、そこで勾配が 0 になるのである。だから、二乗和誤差を最小にする \mathbf{W} を求める方法として、勾配が 0 ということが利用できる。勾配は微分によって求められるが、その際に複数ある \mathbf{W} の要素それぞれの一つに注目して注目した以外の要素を定数とみなして微分をおこなうこと、すなわち偏微分を行うことで、 \mathbf{W} の要素ごとの勾配が得られる。そしてそれらが同時に 0 となるような \mathbf{W} を求めるのである。 \mathbf{W} の要素はこれまでの例では w_0 と w_1 の 2 個だが、 $w_0 \sim w_m$ の $m+1$ 個あるとして、これをあらわすならば、次式となる。

$$\begin{cases} \frac{\partial L}{\partial w_0} = 0 \\ \frac{\partial L}{\partial w_1} = 0 \\ \vdots \\ \frac{\partial L}{\partial w_m} = 0 \end{cases} \quad (3.3.2)$$

これらをまとめて次式であらわすこととする。

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{0} \quad (3.3.3)$$

3.4 線形回帰(直接法)

前節では線形結合の重み \mathbf{w} の変化に対する二乗和誤差 L の変化を可視化し、そして二乗和誤差を最小にするときに、勾配が 0 になることを見てきた。そして、二乗和誤差 L の重み \mathbf{w} の要素ごとの偏微分でそれぞれの勾配を求めて、それらが同時にいずれも 0 となるような \mathbf{w} が求める解であることがわかった。

ではさっそく最小二乗法で答えを求めたのと同じ表 3.2.1 の場合について、この方法で解を求めてみよう。まず二乗和誤差は、

$$\begin{aligned} L &= \frac{1}{2}(\varepsilon_0^2 + \varepsilon_1^2 + \varepsilon_2^2) \\ &= \frac{1}{2}(w_0 + 3w_1 - 2)^2 + \frac{1}{2}(w_0 + 5w_1 - 3)^2 + \frac{1}{2}(w_0 + 4w_1 - 3)^2 \end{aligned} \quad (3.4.1)$$

二乗和誤差 L の \mathbf{w} に対する勾配を求める、まず w_0 について

$$\begin{aligned} \frac{\partial L}{\partial w_0} &= \frac{\partial}{\partial w_0} \left\{ \frac{1}{2}(\varepsilon_0^2 + \varepsilon_1^2 + \varepsilon_2^2) \right\} \\ &= \frac{\partial}{\partial w_0} \left\{ \frac{1}{2}(\varepsilon_0^2) \right\} + \frac{\partial}{\partial w_0} \left\{ \frac{1}{2}(\varepsilon_1^2) \right\} + \frac{\partial}{\partial w_0} \left\{ \frac{1}{2}(\varepsilon_2^2) \right\} \\ &= \frac{\partial}{\partial \varepsilon_0} \left\{ \frac{1}{2}(\varepsilon_0^2) \right\} \frac{\partial \varepsilon_0}{\partial w_0} + \frac{\partial}{\partial \varepsilon_1} \left\{ \frac{1}{2}(\varepsilon_1^2) \right\} \frac{\partial \varepsilon_1}{\partial w_0} + \frac{\partial}{\partial \varepsilon_2} \left\{ \frac{1}{2}(\varepsilon_2^2) \right\} \frac{\partial \varepsilon_2}{\partial w_0} \end{aligned} \quad (3.4.2)$$

ここで、

$$\begin{cases} \frac{\partial \varepsilon_0}{\partial w_0} = \frac{\partial}{\partial w_0} (w_0 + 3w_1 - 2) = 1 \\ \frac{\partial \varepsilon_1}{\partial w_0} = \frac{\partial}{\partial w_0} (w_0 + 5w_1 - 3) = 1 \\ \frac{\partial \varepsilon_2}{\partial w_0} = \frac{\partial}{\partial w_0} (w_0 + 4w_1 - 3) = 1 \end{cases} \quad (3.4.3)$$

したがって、

$$\frac{\partial L}{\partial w_0} = \varepsilon_0 + \varepsilon_1 + \varepsilon_2 \quad (3.4.4)$$

同様に w_1 について

$$\begin{aligned}
\frac{\partial L}{\partial w_1} &= \frac{\partial}{\partial w_1} \left\{ \frac{1}{2} (\varepsilon_0^2 + \varepsilon_1^2 + \varepsilon_2^2) \right\} \\
&= \frac{\partial}{\partial w_1} \left\{ \frac{1}{2} (\varepsilon_0^2) \right\} + \frac{\partial}{\partial w_1} \left\{ \frac{1}{2} (\varepsilon_1^2) \right\} + \frac{\partial}{\partial w_1} \left\{ \frac{1}{2} (\varepsilon_2^2) \right\} \\
&= \frac{\partial}{\partial \varepsilon_0} \left\{ \frac{1}{2} (\varepsilon_0^2) \right\} \frac{\partial \varepsilon_0}{\partial w_1} + \frac{\partial}{\partial \varepsilon_1} \left\{ \frac{1}{2} (\varepsilon_1^2) \right\} \frac{\partial \varepsilon_1}{\partial w_1} + \frac{\partial}{\partial \varepsilon_2} \left\{ \frac{1}{2} (\varepsilon_2^2) \right\} \frac{\partial \varepsilon_2}{\partial w_1}
\end{aligned} \tag{3.4.5}$$

ここで、

$$\begin{cases} \frac{\partial \varepsilon_0}{\partial w_1} = \frac{\partial}{\partial w_1} (w_0 + 3w_1 - 2) = 3 \\ \frac{\partial \varepsilon_1}{\partial w_1} = \frac{\partial}{\partial w_1} (w_0 + 5w_1 - 3) = 5 \\ \frac{\partial \varepsilon_2}{\partial w_1} = \frac{\partial}{\partial w_1} (w_0 + 4w_1 - 3) = 4 \end{cases} \tag{3.4.6}$$

したがって、

$$\frac{\partial L}{\partial w_1} = 3\varepsilon_0 + 5\varepsilon_1 + 4\varepsilon_2 \tag{3.4.7}$$

$\frac{\partial L}{\partial w_0}$ と $\frac{\partial L}{\partial w_1}$ が求まったので、これらをまとめると、

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_0} \\ \frac{\partial L}{\partial w_1} \end{pmatrix} = \begin{pmatrix} \varepsilon_0 + \varepsilon_1 + \varepsilon_2 \\ 3\varepsilon_0 + 5\varepsilon_1 + 4\varepsilon_2 \end{pmatrix} \tag{3.4.8}$$

また元の関係式に戻って、

$$\mathbf{Y} = \boldsymbol{\Phi} \mathbf{W} = \mathbf{T} + \mathbf{E} \tag{3.2.4 再掲}$$

この各辺の各項の左から ${}^t\boldsymbol{\Phi}$ をかけて、

$${}^t\boldsymbol{\Phi} \mathbf{Y} = {}^t\boldsymbol{\Phi} \boldsymbol{\Phi} \mathbf{W} = {}^t\boldsymbol{\Phi} \mathbf{T} + {}^t\boldsymbol{\Phi} \mathbf{E} \tag{3.4.9}$$

ここで、この式の ${}^t\Phi E$ だが、

$${}^t\Phi E = \begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} = \begin{pmatrix} \varepsilon_0 + \varepsilon_1 + \varepsilon_2 \\ 3\varepsilon_0 + 5\varepsilon_1 + 4\varepsilon_2 \end{pmatrix} \quad (3.4.10)$$

これは式 3.4.8 に示す $\frac{\partial L}{\partial W}$ そのものである。

いっぽう L を最小とする W では、その勾配は 0 となるはずだから、

$$\frac{\partial L}{\partial W} = \mathbf{0} \quad (3.3.3 \text{ 再掲})$$

したがって、

$${}^t\Phi E = \mathbf{0} \quad (3.4.11)$$

となって式 3.4.9 から、

$${}^t\Phi\Phi W = {}^t\Phi T \quad (3.4.12)$$

$$\therefore W = \left({}^t\Phi\Phi \right)^{-1} {}^t\Phi T \quad (3.4.13)$$

これを計算してみると、

$$\begin{aligned} W &= \left(\begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} 1 & 3 \\ 1 & 5 \\ 1 & 4 \end{pmatrix} \right)^{-1} \begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} \\ &= \begin{pmatrix} 3 & 12 \\ 12 & 50 \end{pmatrix}^{-1} \begin{pmatrix} 8 \\ 33 \end{pmatrix} \\ &= \begin{pmatrix} \frac{25}{3} & -2 \\ -2 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 8 \\ 33 \end{pmatrix} = \begin{pmatrix} \frac{2}{3} \\ \frac{1}{2} \end{pmatrix} \end{aligned} \quad (3.4.14)$$

以上の結果から、

$$\begin{cases} w_0 \doteq 0.6667 \\ w_1 \doteq 0.5000 \end{cases} \quad (3.4.15)$$

最小二乗法と同じ結果が得られた。いささか回り道をしたが、式 3.4.13 に示す

$W = \left({}^t\Phi\Phi \right)^{-1} {}^t\Phi T$ によって解が得られる. それではさっそく、この解法をプログラムにしておこう.

リスト 3.4.1：線形回帰によって解く

```
import numpy as np
import matplotlib.pyplot as plt
from ufiesia import LCBF

data = np.array([[3, 2],[5, 3],[4, 3]])
X = data[:, 0]    # 入力
T = data[:, 1]    # 正解値

# -- 基底関数を指定 --
phi = LCBF.RectilinearBasis()
Pi = phi(X)
print('Pi\n', Pi)

# -- パラメタの算出 --
tPiPi = np.dot(Pi.T, Pi)
tPiPiI = np.linalg.inv(tPiPi)
print('tPiPiI\n', tPiPiI)
tPiT = np.dot(Pi.T, T)
print('PiTT\n', tPiT)
W = np.dot(tPiPiI, tPiT)
print('算出したパラメタ', W)

# -- 入力値とともにグラフを描く --
plt_x = np.linspace(min(X)-1, max(X)+1, 1000)
phi_plt_x = phi(plt_x)
plt_y = np.dot(phi_plt_x, W)

# -- グラフ表示 --
plt.grid()
plt.scatter(X, T, marker='x', s=100)
plt.scatter(plt_x, plt_y, marker='.', s=1)
plt.xlabel('input')
plt.ylabel('output')
plt.show()
```

「リスト 3.1.2:2 点を通る直線の切片と傾き」に示した連立方程式のプログラムと全体の構成は同じであるから、それとの違いに着目して説明する. データは[4, 3]を加えている. 説明変数 X と目的変数 T として切り出してから、基底関数を指定し、 Φ を得るまでは連立方程式のプログラムと同じ. いっぽう、連立方程式のプログラムで W を求める式は $W = \Phi^{-1}T$ だったが、ここでは $W = \left({}^t\Phi\Phi \right)^{-1} {}^t\Phi T$ により W を算出する. そこですぐ $tPiPi = np.dot(Pi.T, Pi)$ により、 ${}^t\Phi\Phi$ を求めてから、 $tPiPiI = np.linalg.inv(tPiPi)$ で、

$({}^t\Phi\Phi)^{-1}$ を得る。また $tPiT = np.dot(Pi.T, T)$ で ${}^t\Phi T$ を作っておいて、最後に

$W = np.dot(tPiPiI, tPiT)$ で両者の積(ここでは $tPiPiI$ は行列、 $tPiT$ はベクトル)をとって W を得る。最後に連立方程式のプログラム同様 matplotlib でグラフ表示する。なお図の都合から、`linespace` で要素数 1000、`plt.scatter` のマーカのサイズ(×は $s=100$, 線は $s=1$)など、指定を追加しているが、これらはなくても良い。

これまでたった 2 点や 3 点のデータを対象としてきたが、ここでデータを増やしてみよう。scikit-learn にアヤメのデータがあるが、この中から花卉の長さと花卉の幅のデータのさらにその一部 20 個を切り出してみた。

リスト 3.4.2 : scikit-learn のアヤメのデータの一部

```
data = [[4.7, 1.4],
        [4.5, 1.5],
        [4.9, 1.5],
        [4.0, 1.3],
        [4.6, 1.5],
        [4.5, 1.3],
        [4.7, 1.6],
        [3.3, 1.0],
        [4.6, 1.3],
        [3.9, 1.4],
        [3.5, 1.0],
        [4.2, 1.5],
        [4.0, 1.0],
        [4.7, 1.4],
        [3.6, 1.3],
        [4.4, 1.4],
        [4.5, 1.5],
        [4.1, 1.0],
        [4.5, 1.5],
        [3.9, 1.1]]
```

このデータを使うため、「リスト 3.4.2」は、`data_iris.py` としてプログラムと同じディレクトリに保存する。

そして、「リスト 3.4.1:線形回帰によって解く」のデータを指定する以下の行

```
data = np.array([[3, 2], [5, 3], [4, 3]])
```

を次のように書き換える

```
import data_iris
```

```
data = np.array(data_iris.data)
```

これによって、「リスト 3.4.1:線形回帰によって解く」のデータを「リスト 3.4.2:scikit-learn

にアヤメのデータの一部」で置き換えて実行すると図3.4.1を得る。

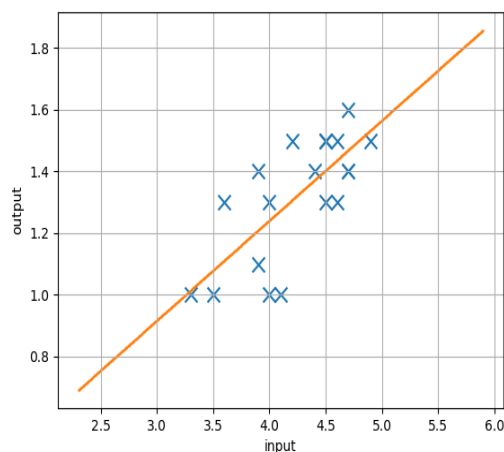


図 3.4.1 アヤメの花弁の長さと幅

データが多くなってもプログラムが問題なく動作し妥当な結果が得られるはずだ。むしろこのような場合に、人が引いたのではそれぞれの主観によってさまざまな答えがありうるのに対し、数学的な根拠をもった、ただ一つの答えを得られることにこそ、価値が認められるのではないだろうか？

[補足]

基底関数の種類とその次数によらず答えが求められるように一般化しておく

$$L = \frac{1}{2} \sum_{i=1}^n \varepsilon^2 \quad (3.3.1 \text{ 再掲})$$

これを w_0 で微分すると、

$$\frac{\partial L}{\partial w_0} = \frac{\partial}{\partial w_0} \left(\frac{1}{2} \sum_{i=1}^n \varepsilon^2 \right) = \frac{\partial}{\partial \varepsilon} \left(\frac{1}{2} \sum_{i=1}^n \varepsilon^2 \right) \frac{\partial \varepsilon}{\partial w_0} \quad (3.4.16)$$

ここで、

$\varepsilon = y - t = w_0 \varphi_0(x) + w_1 \varphi_1(x) + \cdots + w_m \varphi_m(x) - t$ を w_0, w_1, \dots, w_m で微分すると、

$$\frac{\partial \varepsilon}{\partial w_0} = \varphi_0(x), \quad \frac{\partial \varepsilon}{\partial w_1} = \varphi_1(x), \quad \cdots, \quad \frac{\partial \varepsilon}{\partial w_m} = \varphi_m(x) \quad (3.4.17)$$

また、

$$\frac{\partial}{\partial \varepsilon} \left(\frac{1}{2} \sum^n \varepsilon^2 \right) = \sum^n \varepsilon \quad (3.4.18)$$

したがって、

$$\frac{\partial L}{\partial w_0} = \sum^n \varepsilon \varphi_0(x) \quad (3.4.19)$$

同様に、

$$\frac{\partial L}{\partial w_1} = \sum^n \varepsilon \varphi_1(x), \quad \dots, \quad \frac{\partial L}{\partial w_m} = \sum^n \varepsilon \varphi_m(x) \quad (3.4.20)$$

まとめると、

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}} &= \begin{pmatrix} \frac{\partial L}{\partial w_0} \\ \vdots \\ \frac{\partial L}{\partial w_m} \end{pmatrix} = \begin{pmatrix} \sum^n \varepsilon \varphi_0(x) \\ \vdots \\ \sum^n \varepsilon \varphi_m(x) \end{pmatrix} \\ &= \begin{pmatrix} \varepsilon_0 \varphi_0(x_0) + \dots + \varepsilon_n \varphi_0(x_n) \\ \vdots \\ \varepsilon_0 \varphi_m(x_0) + \dots + \varepsilon_n \varphi_m(x_n) \end{pmatrix} \\ &= \begin{pmatrix} \varphi_0(x_0) & \dots & \varphi_0(x_n) \\ \vdots & \ddots & \vdots \\ \varphi_m(x_0) & \dots & \varphi_m(x_n) \end{pmatrix} \begin{pmatrix} \varepsilon_0 \\ \vdots \\ \varepsilon_n \end{pmatrix} \end{aligned} \quad (3.4.21)$$

あらためてここで、

$$\mathbf{E} = \begin{pmatrix} \varepsilon_0 \\ \vdots \\ \varepsilon_n \end{pmatrix} \quad (3.4.22)$$

とおけば、

$$\therefore \frac{\partial L}{\partial \mathbf{W}} = {}^t \Phi \mathbf{E} \quad (3.4.23)$$

L を最小にする \mathbf{W} で $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{0}$ だから、

$${}^t\Phi E = 0 \quad (3.4.11 \text{ 再掲})$$

ここで、 $Y = \Phi W = T + E$ の各辺の左から ${}^t\Phi$ をかけて、

$${}^t\Phi Y = {}^t\Phi \Phi W = {}^t\Phi T + {}^t\Phi E = {}^t\Phi T \quad (3.4.24)$$

$$\therefore W = \left({}^t\Phi \Phi \right)^{-1} {}^t\Phi T \quad (3.4.13 \text{ 再掲})$$

これで一般化した場合にも W を求めることができる。この式は線形回帰において解を求める重要な式であり、正規方程式と呼ばれる。

ここまでは直線近似に限定して、基底関数として「リスト 3.1.1」で定義した RectilinearBasis を使用してきた。しかしもちろん線形回帰の基底関数は、この限りではない。ここまでに、基底関数の次数を $0 \sim m$ とした説明を加えてきた。そこで、RectilinearBasis を拡張して、多項式基底 Polynormal Basis を新たに定義しよう。

リスト 3.4.3：基底関数 $\Phi(x) = (x^0, x^1, x^2, \dots, x^m)$ を定義

```
# -- 基底関数：多項式基底 Polynormal Basis [x**0, x**1, ..., x**(m-1)]
class PolynormalBasis:
    def __init__(self, m=2):
        self.m = m
        print('Initialize PolynormalBasis')

    def __call__(self, x):
        m = self.m
        y = np.array([x**j for j in range(m)])          # x の冪乗のベクトル
        y = y.T                                         # 軸0:データ x、軸1:基底次元
        return y
```

基本的な構成は、RectilinearBasis と同じだが、`__init__()` メソッドには、引数 m で多項式の次数を指定する。 m を指定しない場合には、 $m=2$ となって、RectilinearBasis と同じになる。 m は、`self.m = m` によってインスタンス変数に保存し `__call__()` で使用する。基底関数は、python の内包表記 `[x**j for j in range(m)]` で生成して、numpy の array にしている。これを転置して返り値としているのは RectilinearBasis と同じ。なお、このコードでは $0 \sim m$ の $m+1$ 次ではなく、 $0 \sim m-1$ の m 次となることに注意。（よって $m=2$ で x^0, x^1 の直線となる）

さっそく「リスト 3.4.3：基底関数 $\phi(x) = (x^0, x^1, x^2, \dots, x^m)$ を定義」を使って、基底関数を x^2 までに広げてみる。すなわち、

$\phi(x) = (x^0, x^1, x^2) = (1, x, x^2)$ としたとき、

3 点を通る 2 次曲線が引けるはずである。

そこで、「リスト 3.4.1：線形回帰によって解く」で、基底関数を指定する以下の行

```
phi = LCBF.RectilinearBasis()
```

これを次のように書き換える

```
phi = LCBF.PolynormalBasis(3)
```

そして実行すると、3 点を通る 2 次曲線が確認できる。

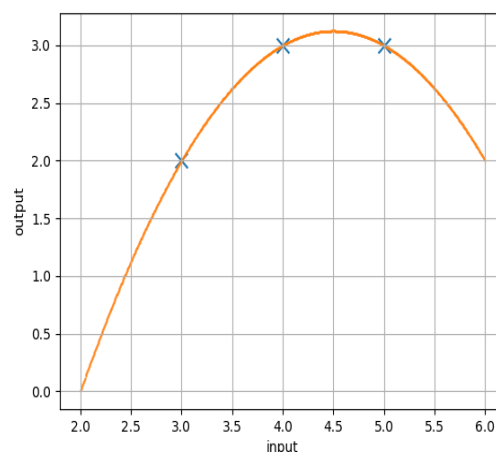


図 3.4.2 3 点を通る 2 次曲線

さらに、 $x^m (m > 3)$ までに広げて $\phi(x) = (x^0, x^1, \dots, x^m)$ としたときにも、3 点を通る線が描けるはずである。2 次以上の曲線が 3 点の上を通るからといって、はたして直線よりも正確だと言えるだろうか？それを仮にその通りだとしても、それならば 3 次曲線は 2 次曲線よりも正確だろうか？そもそも観測値ないしは測定値には誤差が含まれることもわすれてはならない。その誤差の見極めも含めて、ここでいえることは、答えは一つではなく、むしろ無数にあるどれが本当かわからない中から、もっともらしい答を導き出すようなものだということである。データの数が多ければある程度は軽減されることとはいえ、機械学習が扱う問題の多くは、どんなにたくさんデータを集めたつもりでも、有限個のかぎられたデータからそれらしい答を導き出しているにすぎないことを忘れてはならない。

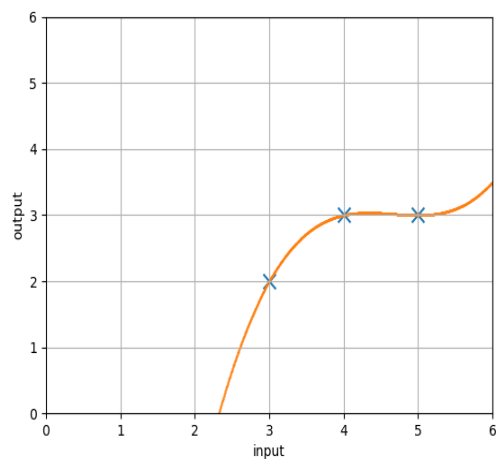


図 3.4.3 3点を通る3次曲線

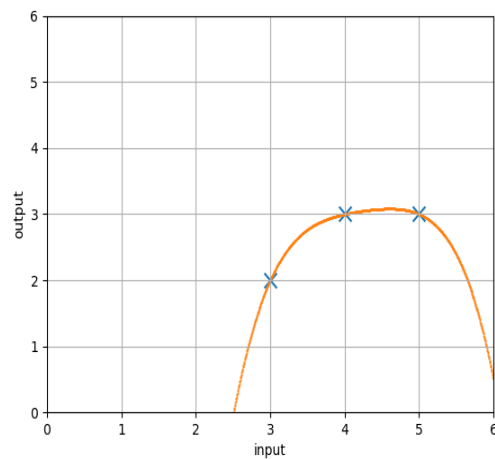


図 3.4.4 3点を通る4次曲線

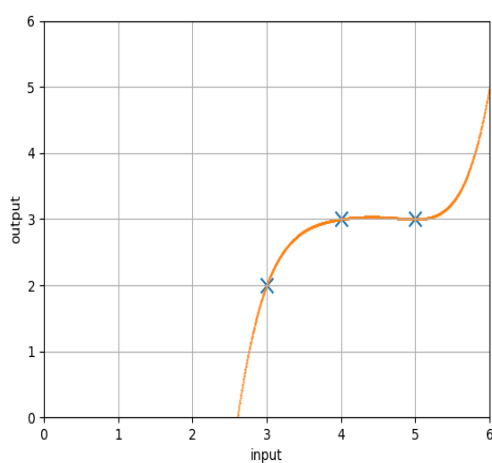


図 3.4.5 3点を通る5次曲線

注：これらの図を描く際には、2次曲線を描く際に加えた指定に、さらにデータの正規化を行い、重み W の正則化項を加えているが、本題の議論と離れるためここでは説明を省略する。

データの数を増やした場合として、「リスト 3.4.5:scikit-learn のアヤメのデータの一部」を線形回帰の対象として結果を得た。しかし、この場合は説明変数と目的変数の関係は直線であらわすのが妥当であり、せっかく多項式基底を定義しても、無駄な曲線を描くばかりで意味があるとは思われない。そこで少し素性の異なるデータを探したところ、インフルエンザの流行を数値化データが入手できた。これは経過週と定点当たりの報告数の対となった 20 個のデータとなっている。

リスト 3.4.4：インフルエンザの流行を数値化データ

```
data = [[ 1, 0.26],
        [ 2, 0.25],
        [ 3, 0.19],
        [ 4, 0.34],
        [ 5, 0.89],
        [ 6, 1.78],
        [ 7, 2.51],
        [ 8, 4.81],
        [ 9, 10.97],
        [10, 18.02],
        [11, 19.28],
        [12, 32.42],
        [13, 62.12],
        [14, 62.18],
        [15, 62.6 ],
        [16, 49.18],
        [17, 32.57],
        [18, 25.93],
        [19, 21.17],
        [20, 14.68]]
```

さっそくこのデータを対象に線形回帰を試みよう。ここでも、この「リスト 3.4.4」は data_flu.py として保存し、「リスト 3.4.1：線形回帰によって解く」のデータを指定する以下の行を次のように書き換えて実行すると次図を得る。

```
import data_flu
data = np.array(data_flu.data)
```

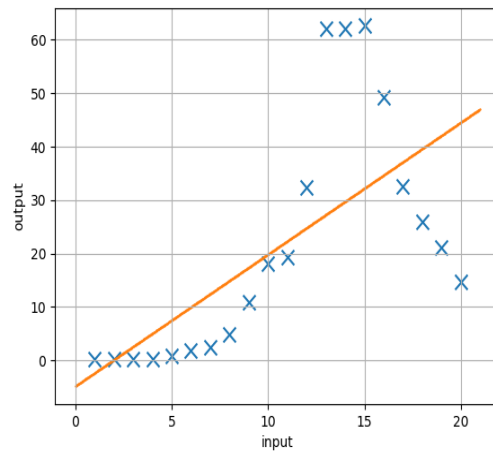


図 3.4.6 インフルエンザの流行～直線

直線ではデータを近似しているとはいいいがたいのは明らかなので、多項式基底にして 3 次曲線で線形回帰を試みる。そのために、基底関数を指定する行を次のように書き換える

```
phi = LCBF.PolynomialBasis(4)
```

そして実行すると次図を得る。

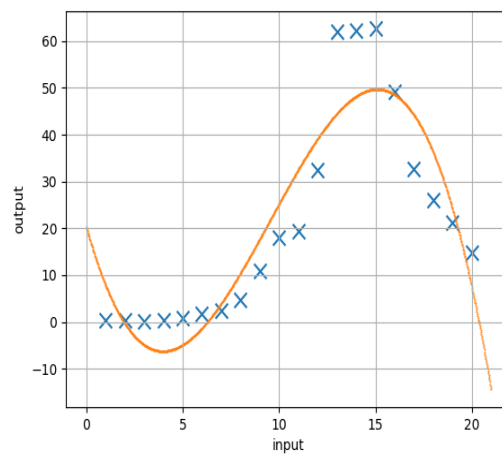


図 3.4.7 インフルエンザの流行～3 次曲線

やはりあまり感心しないが、データをなぞろうとしていることは確認できる。ここで肝心なのは、指定する基底関数の次数によらずプログラムが動作して、それなりの結果が得られることである。

インフルエンザの流行なのだから 0 以下になるのはおかしいと思うのは、データ以外の情報を総合的に判断するからであって、データだけから得られるのは、このようなことなのである。しかし、自然現象のように確率的に起きる事象であることを踏まえて、基底関数をそれに即したものに変わること、はるかに上手にデータをなぞることができるので、それを次節で取り上げよう。

3.5 正規分布、ガウス分布

確率的に起きる事象の、もっとも典型的で、もっともありふれた分布は、正規分布ないしはガウス分布とされる。この分布は、統計学や自然科学、社会科学の様々な場面で複雑な現象を簡単に表すモデルとして用いられていることが多い。そして、この分布を基底関数にして線形回帰をおこなうことは、説明変数と目的変数の間の関係がわからない場合に良く行われる。ガウス分布をグラフで描くと次のようになり、その形のごとく「ベルカーブ」とか、「鐘形曲面」とも呼ばれる。

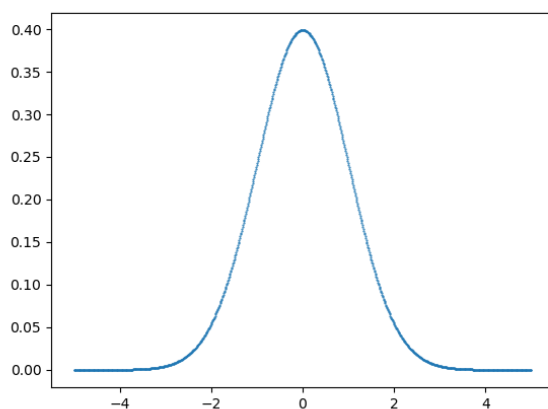


図 3.5.1 ガウス分布

ガウス分布を数式であらわすと、平均 0、分散 1 のとき、

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \quad (3.5.1)$$

もう少し一般的に、平均 μ 、分散 σ^2 のとき、

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \quad (3.5.2)$$

注：ガウス分布の先頭の定数項 $\frac{1}{\sqrt{2\pi}\sigma}$ により、ガウス分布を積分した場合の値は 1 となつて、ガウス分布は確率密度関数の定義に合致する。

$$\int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} = \sqrt{2\pi}\sigma \quad (3.5.3)$$

であるから、 $f(x)$ がガウス分布の時、

$$\int_{-\infty}^{\infty} f(x) = 1 \quad (3.5.4)$$

ガウス分布の式を見てもよくわからないと思うので、この式であらわされるガウス分布をプログラミングしてベルカーブを描いてみよう。

リスト 3.5.1：ガウス分布を描く

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

x = np.linspace(-5, 5, 1000)
mu = 0; s = 1
y = np.exp(-(x - mu)**2 / (2 * s**2))
y /= np.sqrt(2 * np.pi) * s
plt.scatter(x, y, marker='.', s=1)
plt.show()
```

きれいにつながった線を描くために、

```
x = np.linspace(-5, 5, 1000)
```

として-5~+5の範囲を 1000 個のデータを用意している。そして平均と分散は、

```
mu = 0; s = 1
```

で指定している。この μ と s の値を変えると、ベルカーブが変化するので試してみると良い。

ガウス分布は次の 2 行で計算する。

```
y = np.exp(-(x - mu)**2 / (2 * s**2))
```

```
y /= np.sqrt(2 * np.pi) * s
```

1 行目でガウス分布のベルカーブの形状を計算して、2 行目でガウス分布の先頭の定数項の部分を担っている。この 2 行目の定数項は、全体を積分して 1 である必要がなければ不要で、コメントアウトできる。

ガウス基底として、この分布を使う際には、この平均と分散をパラメータとして与える。そして作ったガウス分布を次数に応じて複数用意して、その分布の重ね合わせで近似する。それをふまえて、ガウス基底を定義することにする。

リスト 3.5.2：ガウス基底を定義する

```
# — 基底関数：ガウス基底 Gaussian Basis
class GaussianBasis:
    def __init__(self, m=2, s=1.0, *mu):
        self.m = m
        self.s = s
        self.mu = mu
        print('Initialize GaussianBasis')
        if len(self.mu)>0 and len(self.mu)!=m:
            raise Exception('Inconsistent specification of "m" and "mu"')

    def __call__(self, x):
        m = self.m
        s = self.s
        if len(self.mu)==0:
            mu = np.linspace(np.min(x), np.max(x), m+2, endpoint=True)
            mu = mu[1:-1] # 次数+2 で一旦生成して両端を除く
            self.mu = mu
        else:
            mu = self.mu
        y_shape = len(x), m # 軸0:データ x、軸1:基底次元
        mu = np.broadcast_to(mu, y_shape) # データ x 方向に拡張
        x = np.broadcast_to(x.reshape(-1,1), y_shape) # 基底 mu 方向に拡張
        y = np.exp(-(x - mu)**2 / (2 * s**2)) # ガウス分布 mu:中心、s:広がり
        return y
```

`__init__()` メソッドの引数 `m` は次数であり、何本のベルカーブを用意するかを指定する。いっぽう引数 `s` でベルカーブの広がりを指定するが、 s^2 (`s**2`) がガウス分布における分散に相当する。また、引数 `mu` でガウス分布の中心を指定するが省略できる。指定する場合には、複数のベルカーブの複数の中心を、`' , '` で区切った数列で指定する。引数は `*mu` となっていて変数名に `*` が付加されているが、これにより引数の数列が `tuple` でメソッドに渡される一方、省略してもエラーせずに空の `tuple` となる。空の `tuple` は長さ=0 なので、`len()` で `mu` が指定されたか否かが判定できる。また、引数 `mu` で指定するベルカーブの中心の数 `len(self.mu)` はベルカーブの本数と一致していなければならないから、不一致の場合には例外処理とした。

ベルカーブの中心が `__init__()` で指定されなかった場合には、`__call__()` メソッドを最初に実行する際に生成する。その場合には、`__init__()` の引数 `m` で指定されたベルカーブの本数に対し、`m+2` 個の要素からなる `mu` を、`x` の最小値から最大値の範囲に均等に並ぶリストとし

て生成してから、両端の1つずつを捨てる。これは m で指定された数の μ を、対象となる説明変数の範囲に均等に割り当てるためである。ベルカーブの中心 μ が一旦確定すると、インスタンス変数として保持して、その後は保持された μ を使う。ここまで準備ができると、`__call__()` メソッドの役割として Φ を生成する。生成するにあたり、 μ と x は `numpy` の `broadcast_to` で明示的に返り値 y と同じ形状に拡張する。それから、 x の値の範囲に中心を持つベルカーブをガウス分布の定義の式（定数項は除いている）

```
y = np.exp(-(x - mu)**2 / (2 * s**2))
```

で計算して生成する。生成されるベルカーブは、 x の値の範囲の各 μ を中心とした複数本となる。そしてこれが返り値 y であり Φ そのものである。

さてそれではガウス基底を使って、多項式でうまくいかなかったインフルエンザの流行を線形回帰してみよう。

リスト 3.5.3：ガウス基底を用いた線形回帰によって解く

```
import numpy as np
import matplotlib.pyplot as plt
from ufiesia import LCBF
import data_flu

data = np.array(data_flu.data)
X = data[:, 0]    # 入力
T = data[:, 1]    # 正解値

# -- 基底関数にガウス基底を指定 --
phi = LCBF.GaussianBasis(1, 2.5, 14)
Pi = phi(X)
print('Pi\n', Pi)

tPiPi = np.dot(Pi.T, Pi)
tPiPiI = np.linalg.inv(tPiPi)
print('PiT PiI\n', tPiPiI)
tPiT = np.dot(Pi.T, T)
print('PiTT\n', tPiT)
W = np.dot(tPiPiI, tPiT)
print('算出したパラメタ', W)

# -- 入力値とともにグラフを描く --
plt_x = np.linspace(0, max(X)+1, 1000)
phi_plt_x = phi(plt_x)
plt_y = np.dot(phi_plt_x, W)

# -- グラフ表示 --
plt.grid()
plt.scatter(X, T, marker='x', s=100)
plt.scatter(plt_x, plt_y, marker='.', s=1)
plt.xlabel('input')
plt.ylabel('output')
plt.show()
```

データは先に提示したインフルエンザのそれであり、基底関数にガウス基底を指定する以外は、「リスト 3.4.1：線形回帰によって解く」と基本的には同じ。

基底関数の次数と広がり幅とベルカーブの中心は、引数(1, 2.5, 14)で指定した。すなわち、中心が14、広がり幅が2.5の1本のベルカーブである。もともとこのような現象はガウス分布するのは当たり前と言ってしまうればそれまでだが、納得性のあるカーブでデータをなぞっている。

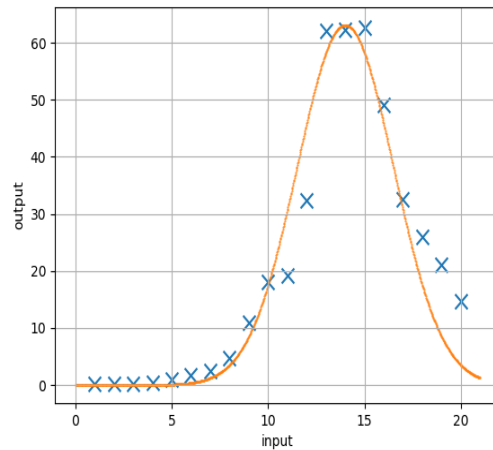


図 3.5.2 インフルエンザの流行～ガウス基底

データの素性に応じて基底関数を選ぶことの重要性を垣間見ることができたのではないだろうか？ もう少し広く一般的に言うならば、解くべき問題に応じたモデルの選択が重要ということである。

[ガウス分布と二乗和誤差]

線形回帰では二乗和誤差を最小にする重み W を求めるが、「二乗和誤差を最小にする」ということについて、あらためて考えてみよう。一例として、「リスト 3.4.2」のアヤメのデータを使って線形回帰を行った結果を見てみよう。誤差を最小にと言っても、誤差がまったくなくなったわけではない。あらためて、 $Y = \Phi W$ そして、 $E = Y - T$ で誤差を算出して、この誤差をプロットしてみよう。

「リスト 3.4.1」のプログラムでデータを「リスト 3.4.2: scikit-learn のアヤメのデータの一部」としたものに、リスト 3.5.4 の 5 行を加える。

リスト 3.5.4：誤差のプロット

```
Y = np.dot(Pi, W)
E = Y - T
plt.plot(E)
plt.show()
print(np.sum(E))
```

これで下図を得る.

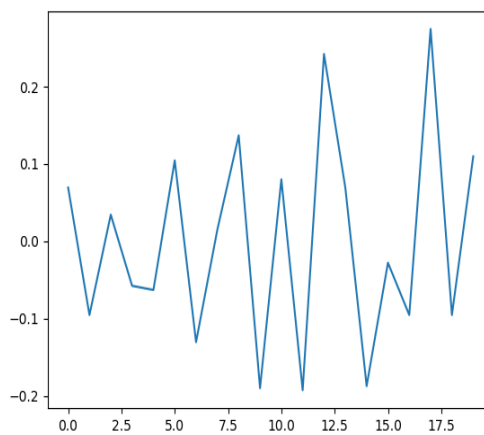


図 3.5.3 誤差のプロット

このグラフは横軸が 20 個のデータに対応し、縦軸がそのデータに対する誤差の大きさとなっている。これを見ても、誤差が値 0 を中心にプラス・マイナスに程よく割り振られていることがわかる。実際に最後の print 文で $\sum \varepsilon \approx 0$ となっていることが確かめられる。

今度は、誤差のヒストグラムを作ってみよう。以下のように、「リスト 3.5.4」を少し変えて「リスト 3.5.2」にするだけで、あとはmatplotlibに任せる。

リスト 3.5.5：誤差のヒストグラム

```
Y = np.dot(Pi, W)
E = Y - T
plt.hist(E)
plt.show()
```

先に得ている結果の図と併せて、「リスト 3.5.2」の追加により得た図を示す。

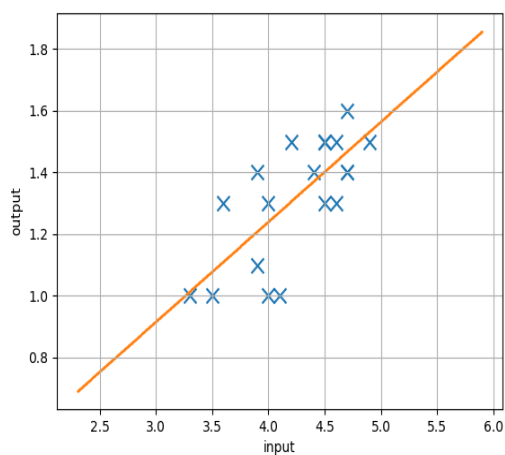


図 3.5.4 アヤメの花弁の長さ と幅(再掲)

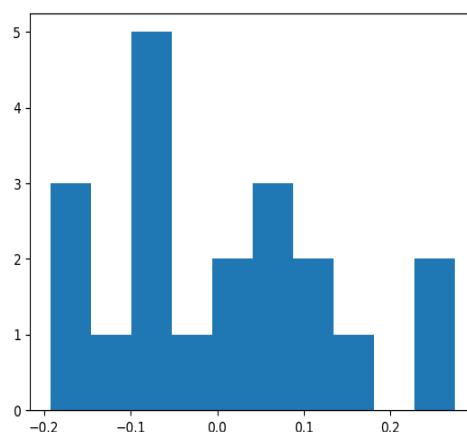


図 3.5.5 花弁の長さ と幅の誤差

データの数が多く凸凹していて、なんとも言えない。そこで、もう少し数を増やしてみよう。sklearn にはアヤメのデータが 150 個あるので、それを全部使ってみよう。python のコードは省略するが、その結果だけを示す。

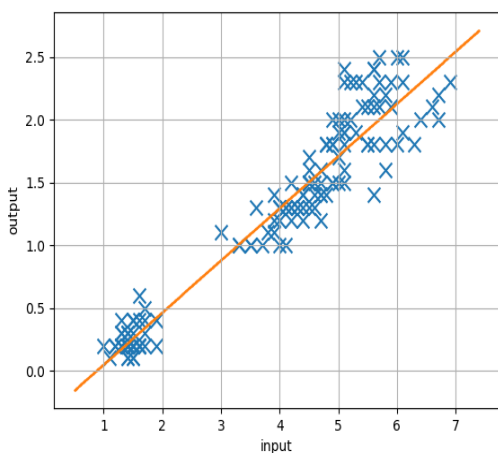


図 3.5.6 アヤメの花弁の長さ と幅 150 点

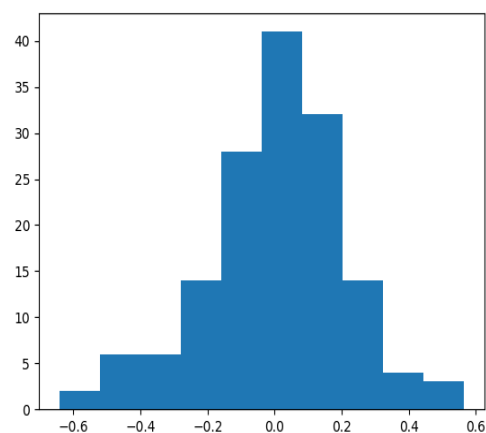


図 3.5.7 花弁の長さ と幅の誤差 150 点

これでもまだ数が少なく凸凹してはいるが、データを増やしたことでガウス分布のベルカーブに近づいていることが見てとれるのではないだろうか？

ヒストグラムの横軸は値の範囲、縦軸は起きた回数である。ヒストグラムを描く際にはデータの分布する範囲を 10 に区切って、その範囲にあるデータの数棒グラフにしているが、これを読み取ると、 $-0.16 \sim -0.04$ が 28 個、 $-0.04 \sim 0.08$ が 41 個、 $0.08 \sim 0.20$ が 32 個などとなっている。確率に直すと、データは全部で 150 個だから、 $-0.16 \sim -0.04$ が $28/150 \doteq 0.19$ 、 $-0.04 \sim 0.08$ が $41/150 \doteq 0.27$ 、 $0.08 \sim 0.20$ が $32/150 \doteq 0.21$ などとなる。

さて、横軸は本来は連続する値を範囲を決めて区切ったものであり、縦軸を確率にすることを考えると、エラー ε 、確率 $p(\varepsilon)$ として、 $p(\varepsilon)$ が平均 0 のガウス分布(さらに分散 $\sigma = 1$ ならば標準正規分布)になっていると仮定できそうである。すなわち、

$$p(\varepsilon) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}\varepsilon^2} \quad (3.5.5)$$

逆に言うならば、エラーは無くせないものとして、そのエラーが平均 0 のガウス分布に従うように追い込んだ結果が得られている、ということができるのではないだろうか？この点をもう少し考えてみよう。個々のデータに対する確率が、 $p(\varepsilon)$ だからデータ全体では、確率の積の法則から、

$$p(E) = \prod_{i=1}^n p(\varepsilon_i) = p(\varepsilon_0) \times p(\varepsilon_1) \times \cdots \times p(\varepsilon_n) \quad (3.5.6)$$

この両辺の対数をとると、

$$\begin{aligned} \log(p(E)) &= \log\left(\prod_{i=1}^n p(\varepsilon_i)\right) = \log(p(\varepsilon_0) \times p(\varepsilon_1) \times \cdots \times p(\varepsilon_n)) \\ &= \log(p(\varepsilon_0)) + \log(p(\varepsilon_1)) + \cdots + \log(p(\varepsilon_n)) = \sum_{i=1}^n \log(p(\varepsilon_i)) \end{aligned} \quad (3.5.7)$$

ここで、エラーが平均 0 のガウス分布になっていると仮定すると、

$$\begin{aligned} \log(p(\varepsilon)) &= \log\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}\varepsilon^2}\right) \\ &= \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) + \log\left(e^{-\frac{1}{2\sigma^2}\varepsilon^2}\right) \\ &= -\log(\sqrt{2\pi}\sigma) - \frac{1}{2\sigma^2}\varepsilon^2 \end{aligned} \quad (3.5.8)$$

$$\therefore \log(p(E)) = -n \log(\sqrt{2\pi}\sigma) - \frac{1}{2\sigma^2} \sum_{i=1}^n \varepsilon_i^2 \quad (3.5.9)$$

$\log(p(E))$ は単調増加なので、 $p(E)$ の最大化と $\log(p(E))$ の最大化は等価。

また、 $-n \log(\sqrt{2\pi}\sigma)$ は定数項。したがって、 $p(E)$ の最大化は $-\frac{1}{2\sigma^2} \sum_{i=1}^n \varepsilon_i^2$ の最大化、すなわち、 $\frac{1}{2\sigma^2} \sum_{i=1}^n \varepsilon_i^2$ の最小化と等価であって、二乗和誤差 $L = \frac{1}{2} \sum_{i=1}^n \varepsilon_i^2$ の最小化そのものである。ここで $p(E)$ の最大化が意味するのは、平均 0 のガウス分布を仮定した E の起きる確率の最大化であるから、まさしく E は平均 0 のガウス分布になるべくしてなるのである。つまり二乗和誤差を最小化するというのは、エラーのばらつきがガウス分布していることを仮定して最適化すること、そのものなのである。

参考：中心極限定理

独立な同一の分布に従う確率変数の算術平均（確率変数の合計を変数の数で割ったもの）の分布は、元の確率変数に標準偏差が存在するならば、元の分布の形状に関係なく、変数の数が多数になったとき、正規分布に収束する。これを中心極限定理という。

3.6 線形結合の拡張、そして、データの正規化と正則化項

ここまで基底関数として、直線基底 RectilinearBasis、多項式基底 PolynormalBasis、ガウス基底 GaussianBasis と 3 つのクラスを定義してきたが、この先の拡張も踏まえ、これら基底関数を束ねて線形結合を作る LinearCombination クラスを定義しておこう。

リスト 3.6.1：線形結合で定義する関数 $f(x) = w_0\phi_0(x) + w_1\phi_1(x) + \dots + w_m\phi_m(x)$

```
class LinearCombination: # Linear Combination of Basis Functions
    def __init__(self, basis_function=None):
        self.phi = basis_function
        self.w = None
        print('Initialize Linear Combination of', self.phi.__class__.__name__)

    def __call__(self, x):
        if self.w is None:
            raise Exception('Need to fix parameters')
        self.Pi = self.phi(x)
        y = np.dot(self.Pi, self.w)
        return y

    def regression(self, x, t):
        if self.phi is None:
            raise Exception('Need to initialize specifying basis function')
        Pi = self.phi(x)
        tPiPi = np.dot(Pi.T, Pi)
        tPiPiI = np.linalg.inv(tPiPi)
        PiI = np.dot(tPiPiI, Pi.T)
        w = np.dot(PiI, t)
        print('Parameters are fixed.')
        self.w = w
        return w
```

このクラスは `__init__()` と `__call__()` と `regression()` の 3 つのメソッドを持つ。 `__init__()` で初期化の際に引数 `basis_function` を基底関数にインスタンス変数 `phi` として設定する。後から基底関数を呼出す際は `self.phi` で呼ぶ。初期化の段階では線形結合の重みは定まっていないから、`self.w = None` だ。 `__call__()` メソッドは、その重みが定まってから実行すべきだから、はじめにこれをチェックしてする。インスタンス変数 `Pi` は基底関数に説明変数 x を与えた返り値 ϕ であり、これと重みを numpy の `dot` により積をとった y は、線形結合の式の値 Y すなわち目的変数の値である。 `regression()` メソッドは、初期化の際に基底関数が指定されていることをチェックした上で、正規方程式によって重みを求める計算を行う。リスト 3.4.1 やリスト 3.5.3 で行っていた計算部分をそのままメソッドにした。

補足：LinearCombination クラスでは、`__call__()`メソッドの引数として説明変数 x を与えて、その中で `self.Pi = self.phi(x)`によって、 Φ を得てインスタンス変数として保存している。しかし説明変数の値は変わらないから、`__init__()`メソッドで、 Φ を得るようにするか、`regression()`メソッドの中で算出する Pi を保存しても良さそうである。たしかに本稿の範囲ではデータを全部使って1回の学習を行うーこれをバッチ学習と呼ぶーので、その通りだが、データが大量にある場合には、切り出す部分を変えながらデータの一部を切り出して学習を進めるーこれをミニバッチ学習と呼ぶー方法が一般的である。そこでここでは、このようにつくりにしている。ちなみに、データを1つずつ取り出して学習を進める方法はオンライン学習と呼ばれる。

いささか脱線したが、「リスト 3.6.1」で定義した LinearCombination を使って線形回帰を行ってみよう。

リスト 3.6.2：LinearCombination を使った線形回帰

```
import numpy as np
import matplotlib.pyplot as plt
from ufiesia import LCBF
import data_flu

m = 9          # 基底関数の次数
s = 2.0        # ガウス基底の広がり係数

data = np.array(data_flu.data)
X = data[:, 0]  # 入力
T = data[:, 1]  # 正解値

# -- 対象を定義し回帰 --
model = LCBF.LinearCombination(LCBF.GaussianBasis(m, s))
model.regression(X, T)

# -- 入力値とともにグラフを描く --
plt_x = np.linspace(min(X), max(X), 1000)
plt_y = model(plt_x)

# -- グラフ表示 --
plt.grid()
plt.scatter(X, T, marker='x', s=100)
plt.scatter(plt_x, plt_y, marker='.', s=1)
plt.xlabel('input')
plt.ylabel('output')
plt.show()
```

この例ではLCBFをufiesiaに置き、その中にLinearCombinationクラスを、基底関数を定義するクラスとともに記述した。LinearCombinationはクラスとして定義したので、これをインスタンス化してmodelとしている。そして入力と正解値を与えregression()メソッドで線形結合の重み W を求める。あとは結果のグラフのためにnumpyの関数linspace()でplt_xを入力として用意し、modelに入れてplt_yを求める。そして元のデータとともにグラフ描画する。

model = LCBF.LinearCombination(LCBF.GaussianBasis(m, s)) のところを
model = LCBF.LinearCombination(LCBF.PolynormalBasis(m)) に書き換えて
多項式基底で基底関数の次数 m を大きくしても、それなりの結果が得られる。

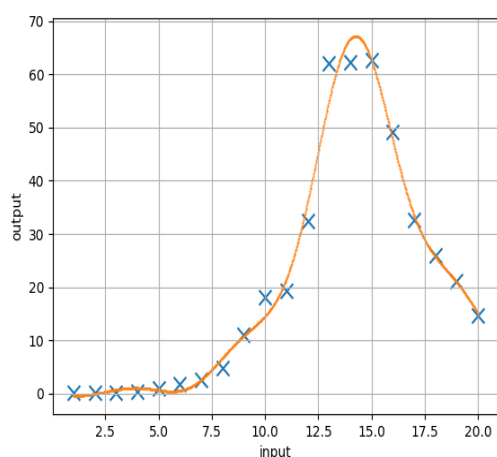


図 3.6.1 インフルエンザ～ガウス基底9次

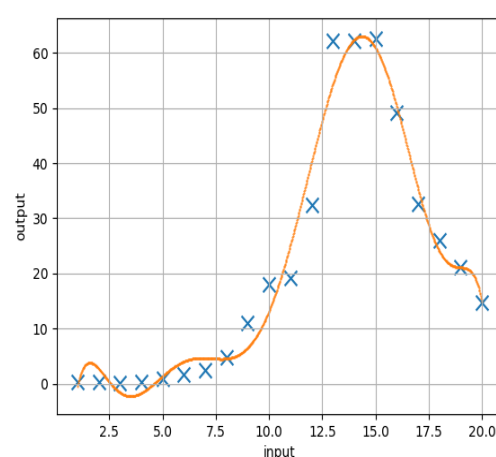


図 3.6.2 インフルエンザ～多項式9次

ガウス基底がうまくデータをなぞるのは期待通り。過学習かどうかはさておき、 $m=9$ 、 $s=2$ あたりがとても上手になぞる。そこでその結果を多項式基底で同じく $m=9$ とした場合とともに掲載する。

補足：逆行列が求まらずにエラーすることがあるが、クラスLinearCombinationのregression()メソッドの中の計算で、np.linalg.inv()にかえてnp.linalg.pinv()を使えば回避できる。

さて、ここでデータの正規化と重みの正規化について取り上げよう。この両者は、線形回帰に限らず、ニューラルネットワークの学習など、機械学習において広く必要な技術である。

[データの正規化]

ここで、3点のデータを多項式基底を使って様々な曲線でなぞる際に説明を省略したデータの正規化を簡単に説明し、そのインプリメントを行う。

一例として、先に取り上げた[3, 2], [5, 3], [4, 3]のデータの組を多項式基底の線形結合で近似しようとする際に、4次以上、すなわち3次曲線以上の場合には、そのままではうまくデータをなぞってくれない。データが少ないのはその通りだが、データをなぞらない原因の一つとして考えられることを述べておきたい。多項式基底では説明変数の値が1を超える場合に、高次のべき乗を取ると値が大きくなるが、0乗や1乗の比較的小さな値と同じく線形結合の項となる。そうすると、各項の重み w の変化に対し、高次の項ばかりに敏感になり、逆に低次の項の重みの変化には鈍くなってしまう。しかし、例えば全体が上昇傾向か下降傾向かなど、重要な全体にわたる情報はえてして低次の項が担う場合が多い。このために、コンピュータにおける計算誤差も関与して、的外れなものを解としてしまうのではないだろうか？

理由はともあれ、説明変数 x を

3 → -1

5 → 1

4 → 0

と変えてみると、今度は3点の上を通る曲線が描けるようになる。＜注＞

注：そのままではnumpy.linalg.LinAlgError: Singular matrixのエラーが出てしまう。これは逆行列が存在しないためで、これを回避するために、numpyのlinalg.inv()をlinalg.pinv()に置き換えれば結果を得られる。そして描いたものが、先に示した3点をなぞる、図3.4.3の3次曲線、図3.4.4の4次曲線、図3.4.5の5次曲線である。

より一般的に、説明変数の値を最小値 = -1、最大値 = 1 にすることで、高次のべき乗を1以下にすることができ、計算の可用性というべきか、計算できる場合を拡大することが期待できる。この計算は、

$$\hat{x} = \frac{x - \min(x)}{\max(x) - \min(x)} * 2 - 1 \quad (3.6.1)$$

いっぽうデータを正規化した場合には元に戻す必要があるが、この逆変換は次式となる。

$$x = \frac{\hat{x} + 1}{2} (\max(x) - \min(x)) + \min(x) \quad (3.6.2)$$

リスト 3.6.3：正規化とその逆変換

```

if normalize: # データの正規化
    xmax = np.max(X); xmin = np.min(X)
    X = (X - xmin) / (xmax - xmin) * 2 - 1

    :
    :   (途中略)
    :

if normalize: # 正規化の逆変換
    X = (X + 1) / 2 * (xmax - xmin) + xmin

```

解を求める計算をはじめる前に正規化を行うが、このとき x の最大値と最小値を求めて、変数 x_{\max} と x_{\min} にそれぞれ入れておく。結果を表示する前に逆変換を行うが、正規化の際に作った x_{\max} , x_{\min} を使えば良い。

ここで正規化を適用した場合の計算を確認しておこう。

$X = (3, 5, 4)$ だから、最大値 5、最小値 3 であり、正規化後の値 \hat{X} は次表の値となる。

表 3.6.1

X	\hat{X}	T	Y	$E = Y - T$
3	-1	2	$w_0 + 3w_1$	$w_0 + 3w_1 - 2$
5	1	3	$w_0 + 5w_1$	$w_0 + 5w_1 - 3$
4	0	3	$w_0 + 4w_1$	$w_0 + 4w_1 - 3$

X を \hat{X} で置き換えて、 Φ をつくれば、

$$\Phi = \begin{pmatrix} 1 & -1 \\ 1 & 1 \\ 1 & 0 \end{pmatrix} \quad (3.6.3)$$

正規方程式は、

$$\begin{aligned}
 W &= \left({}^t\Phi\Phi \right)^{-1} {}^t\Phi T \\
 &= \left(\begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & -1 \\ 1 & 1 \\ 1 & 0 \end{pmatrix} \right)^{-1} \begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} \\
 &= \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}^{-1} \begin{pmatrix} 8 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 8 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{8}{3} \\ \frac{1}{2} \end{pmatrix} \doteq \begin{pmatrix} 2.6667 \\ 0.5000 \end{pmatrix}
 \end{aligned} \tag{3.6.4}$$

W が得られたので、これを使って Y を計算すると、

$$Y = \Phi W \doteq \begin{pmatrix} 1 & -1 \\ 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 2.6667 \\ 0.5000 \end{pmatrix} = \begin{pmatrix} 2.1667 \\ 3.1667 \\ 2.6667 \end{pmatrix} \tag{3.6.5}$$

Y の値は式 3.2.10 と一致する。すなわち、説明変数 X を正規化してもしなくても、目的変数の計算値 Y が一致することが確かめられた。ただし、 X の値が違うのだから、当然、正規化した場合としない場合で W の値は異なる。

正規化するとこのケースは手計算が非常に楽なので、3 次の多項式基底、すなわち 2 次曲線の場合についても計算してみる。 Φ に x^2 の項が加わって、

$$\Phi = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix} \tag{3.6.6}$$

正規方程式は、

$$\begin{aligned}
 W &= \left({}^t\Phi\Phi \right)^{-1} {}^t\Phi T \\
 &= \left(\begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix} \right)^{-1} \begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} \\
 &= \begin{pmatrix} 3 & 0 & 2 \\ 0 & 2 & 0 \\ 2 & 0 & 2 \end{pmatrix}^{-1} \begin{pmatrix} 8 \\ 1 \\ 5 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 0.5 & 0 \\ -1 & 0 & 1.5 \end{pmatrix} \begin{pmatrix} 8 \\ 1 \\ 5 \end{pmatrix} = \begin{pmatrix} 3 \\ 0.5 \\ -0.5 \end{pmatrix}
 \end{aligned} \tag{3.6.7}$$

検算してみると

$$Y = \Phi W = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 3 \\ 0.5 \\ -0.5 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} \quad (3.6.8)$$

表 3.6.1 と見比べると、たしかに $Y = T$ となっており、3 点の上を通ることが確認できる。
この場合の X と Y の関係式は、得られた W の値から、

$$y = 3 + 0.5x - 0.5x^2 \quad (3.6.9)$$

さらに 4 次の多項式基底、すなわち 3 次曲線の場合について計算してみる。
 Φ にさらに x^3 の項が加わって、

$$\Phi = \begin{pmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad (3.6.10)$$

正規方程式は、

$$\begin{aligned} W &= \left({}^t\Phi\Phi \right)^{-1} {}^t\Phi T \\ &= \left(\begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 0 \\ -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \right)^{-1} \begin{pmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \\ 1 & 1 & 0 \\ -1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} \\ &= \begin{pmatrix} 3 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \\ 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \end{pmatrix}^{-1} \begin{pmatrix} 8 \\ 1 \\ 5 \\ 1 \end{pmatrix} \end{aligned} \quad (3.6.11)$$

ここで、

$${}^t\Phi\Phi = \begin{pmatrix} 3 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \\ 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \end{pmatrix} \quad (3.6.12)$$

には困ったことに逆行列がない。

これは 2 行目と 4 行目、あるいは、2 列目と 4 列目が同じだから明らかである。（ ${}^t\Phi\Phi$ のランクは 3）したがって、このままでは W は求まらない。

参考：ムーア・ペンローズ一般逆行列

逆行列を求める際にうまくいかないことへの対処として `np.linalg.inv()` にかえて `np.linalg.pinv()` を使えばよいとしてきた。この `np.linalg.pinv()` が返すのは、ムーア・ペンローズ一般逆行列と呼ばれる。

もう少し説明しよう。もともと解こうとしている方程式は、

$$Y = \Phi W = T + E \quad \text{であって、} {}^t\Phi E = 0 \quad \text{とすることで、} {}^t\Phi\Phi W = {}^t\Phi T$$

これを解くために、 $({}^t\Phi\Phi)^{-1}$ が欲しかったのである。逆行列がない場合、そのかわりに、

ムーア・ペンローズ一般逆行列を持ってきたのだが、その場合ムーア・ペンローズ一般逆行列には都合が良いことに、両辺の差、この場合には ${}^t\Phi\Phi W - {}^t\Phi T$ 、の二乗和を最小にする性質があるということである。もともと、 $E = \Phi W - T$ だから、これは誤差 E を最小にすることにほかならず、妥当な解が得られるのである。

[正則化項]

正則化項について説明しておきたい。

線形結合の式を振り返ると、

$$f(x) = w_0\varphi_0(x) + w_1\varphi_1(x) + \cdots + w_m\varphi_m(x) \quad (3.1.16 \text{ 再掲})$$

ここで重み w_0, w_1, \dots, w_m のいずれかが極端な値になることを考えると、それは線形結合の基底関数のそれに対応するものに極端な重みづけがされているに他ならない。その状態は与えられたデータに適合するために生じたことであつたとしても、好ましいかといえば決してそうではない場合が多い。そこでこの極端な状態が生じることを抑制するために、正則化項を加える方法がある。

また振り返るならば、線形結合の重み \mathbf{W} は、二乗和誤差 L を定義し、その \mathbf{W} に対する

勾配 $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{0}$ から、 ${}^t\Phi\mathbf{E} = \mathbf{0}$ を導いて正規方程式

$$\mathbf{W} = \left({}^t\Phi\Phi \right)^{-1} {}^t\Phi\mathbf{T} \quad (3.4.13 \text{ 再掲})$$

によって \mathbf{W} を求めた。

ここで二乗和誤差 L に、重み \mathbf{W} の大きさそのものを何らかの形で加えることを考えて、それで重み \mathbf{W} が極端な値に陥るのを抑制する。その加え方だが、重み \mathbf{W} の二乗和（この平方根を L2 ノルムという）を係数 λ をかけて加える方法（L2 正則化）がよく行われる。この加える項を正則化項という。

すなわち二乗和誤差に正則化項を加えて、

$$L = \frac{1}{2} \sum^n \varepsilon^2 + \frac{1}{2} \lambda \sum^m w^2 = \frac{1}{2} \sum^n (y - t)^2 + \frac{1}{2} \lambda \sum^m w^2 \quad (3.6.13)$$

正則化項がある場合の二乗和誤差 L の \mathbf{W} に対する勾配を求めていこう。 L の w_0 に対する勾配は、

$$\begin{aligned} \frac{\partial L}{\partial w_0} &= \frac{\partial}{\partial w_0} \left(\frac{1}{2} \sum^n \varepsilon^2 + \frac{1}{2} \lambda \sum^m w^2 \right) \\ &= \frac{\partial}{\partial w_0} \left(\frac{1}{2} \sum^n \varepsilon^2 \right) + \frac{\partial}{\partial w_0} \left(\frac{1}{2} \lambda \sum^m w^2 \right) \end{aligned}$$

$$= \frac{\partial}{\partial \varepsilon} \left(\frac{1}{2} \sum^n \varepsilon^2 \right) \frac{\partial \varepsilon}{\partial w_0} + \frac{\partial}{\partial w_0} \left(\frac{1}{2} \lambda \sum^m w^2 \right) \quad (3.6.14)$$

ここで、

$\varepsilon = y - t = w_0 \varphi_0(x) + w_1 \varphi_1(x) + \cdots + w_m \varphi_m(x) - t$ を w_0 で微分すると、

$$\frac{\partial \varepsilon}{\partial w_0} = \varphi_0(x) \quad (3.6.15)$$

また、

$$\frac{\partial}{\partial \varepsilon} \left(\frac{1}{2} \sum^n \varepsilon^2 \right) = \sum^n \varepsilon \quad (3.6.16)$$

そして、式 3.6.14 の右辺第 2 項の

$\frac{\partial}{\partial w_0} \left(\frac{1}{2} \lambda \sum^m w^2 \right)$ は、偏微分の際に \sum^m の中の w のうち、 w_0 以外を定数とみなして、

$$\frac{\partial}{\partial w_0} \left(\frac{1}{2} \lambda \sum^m w^2 \right) = \lambda w_0 \quad (3.6.17)$$

式 3.6.14 の右辺に、式 3.6.15、式 3.6.16、式 3.6.17 を代入して、

$$\frac{\partial L}{\partial w_0} = \sum^n \varepsilon \varphi_0(x) + \lambda w_0 \quad (3.6.18)$$

L の w_0 に対する勾配が得られた。同様に w_1, \dots, w_m について L の勾配を求めると、

$$\frac{\partial L}{\partial w_1} = \sum^n \varepsilon \varphi_1(x) + \lambda w_1, \quad \dots, \quad \frac{\partial L}{\partial w_m} = \sum^n \varepsilon \varphi_m(x) + \lambda w_m \quad (3.6.19)$$

まとめると、

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{W}} &= \begin{pmatrix} \frac{\partial L}{\partial w_0} \\ \vdots \\ \frac{\partial L}{\partial w_m} \end{pmatrix} = \begin{pmatrix} \sum^n \varepsilon \varphi_0(x) + \lambda w_0 \\ \vdots \\ \sum^n \varepsilon \varphi_m(x) + \lambda w_m \end{pmatrix} \\ &= \begin{pmatrix} \varepsilon_0 \varphi_0(x_0) + \cdots + \varepsilon_n \varphi_0(x_n) + \lambda w_0 \\ \vdots \\ \varepsilon_0 \varphi_m(x_0) + \cdots + \varepsilon_n \varphi_m(x_n) + \lambda w_m \end{pmatrix} \end{aligned}$$

$$\begin{aligned}
&= \begin{pmatrix} \varphi_0(x_0) & \cdots & \varphi_0(x_n) \\ \vdots & \ddots & \vdots \\ \varphi_m(x_0) & \cdots & \varphi_m(x_n) \end{pmatrix} \begin{pmatrix} \varepsilon_0 \\ \vdots \\ \varepsilon_n \end{pmatrix} + \lambda \begin{pmatrix} w_0 \\ \vdots \\ w_m \end{pmatrix} \\
&= {}^t\Phi E + \lambda W
\end{aligned} \tag{3.6.20}$$

L を最小にする W でその勾配 $\frac{\partial L}{\partial W} = \mathbf{0}$ だから、式 3.6.20 により、

$${}^t\Phi E = -\lambda W \tag{3.6.21}$$

$Y = \Phi W = T + E$ の各辺の左から ${}^t\Phi$ をかけて、

$${}^t\Phi Y = {}^t\Phi \Phi W = {}^t\Phi T + {}^t\Phi E \tag{3.4.9 再掲}$$

式 3.6.21 により、

$${}^t\Phi \Phi W = {}^t\Phi T - \lambda W \tag{3.6.22}$$

$${}^t\Phi \Phi W + \lambda W = {}^t\Phi T \tag{3.6.23}$$

式 3.6.23 で W が要素数 m 、すなわち m 次元のベクトルだとするならば、 m 行 m 列の単位行列を I とおく、

$$I = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ \vdots & & & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & 1 \end{pmatrix} \tag{3.6.24}$$

この単位行列を用いて、

$$\lambda W = \lambda I W \tag{3.6.25}$$

だから

$$\left({}^t\Phi \Phi + \lambda I \right) W = {}^t\Phi T \tag{3.6.26}$$

$$\therefore W = \left({}^t\Phi \Phi + \lambda I \right)^{-1} {}^t\Phi T \tag{3.6.27}$$

これで正則化項がある場合にも W を求めることができる。

ところで正則化項なしの場合は、

$$W = \left({}^t\Phi\Phi \right)^{-1} {}^t\Phi T \quad (3.4.13 \text{ 再掲})$$

だったから、 ${}^t\Phi\Phi$ の逆行列のかわりに、 ${}^t\Phi\Phi + \lambda I$ の逆行列を求めれば良いことになる。
 ここで、 ${}^t\Phi\Phi$ に、 λI が加えられることを考えると、もともと3点を4次の多項式基底の線形結合、すなわち、3次曲線でなぞるような場合に、行列 ${}^t\Phi\Phi$ はランク落ちしていて、逆行列がないのだが、対角要素にそれぞれ λ を加えることでフルランクにすることができる。すなわち「正則化」していることにほかならない。正則でないために逆行列がない場合に正則化項を加えることで、逆行列が求められるようになって解が得られるようになる。
 ここで一例として次の場合、

$${}^t\Phi\Phi = \begin{pmatrix} 3 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \\ 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \end{pmatrix} \quad (3.6.12 \text{ 再掲})$$

この場合に2行目と4行目、あるいは、2列目と4列目が同じ値であり逆行列がないが、ここで、対角要素にそれぞれ0.1を加えると、

$${}^t\Phi\Phi + 0.1 * I = \begin{pmatrix} 3.1 & 0 & 2 & 0 \\ 0 & 2.1 & 0 & 2 \\ 2 & 0 & 2.1 & 0 \\ 0 & 2 & 0 & 2.1 \end{pmatrix} \quad (3.6.28)$$

2行目と4行目、あるいは、2列目と4列目は同じ値ではなくなり逆行列が得られる。正則でなかった行列を正則にする、まさに正則化項の「ゆえん」である。実際にプログラムで確かめておこう。

リスト 3.6.4：正則化と逆行列

```
import numpy as np
tPiPi = np.array([[3, 0, 2, 0], [0, 2, 0, 2], [2, 0, 2, 0], [0, 2, 0, 2]])
print(tPiPi)

# 逆行列の有無の確認
try:
    print(np.linalg.inv(tPiPi))
except:
    print('No inverse matrix exists.')
    print(np.linalg.pinv(tPiPi))

# 正則化して逆行列を求める
tPiPi = tPiPi + 0.1 * np.eye(4)
print(tPiPi)
print(np.linalg.inv(tPiPi))
```

ここでは上記の例をそのまま `tPiPi` としている。 `np.linalg.inv()` は、そのまま実行するとエラーしてしまうので `try:` の中で実行し、逆行列が得られる場合はそれを `print` する。 いっぽう逆行列が求まらずにエラーする場合は、 `except:` の `print` 文を実行するが、その場合には `np.linalg.pinv()` で求められるムーアペンローズの一般逆行列が `print` される。 その後、 `np.eye()` で単位行列を作って 0.1 倍して `tPiPi` に足しこんでいる。 今度は逆行列が得られる。 このプログラムでは逆行列が求まらない場合にはムーアペンローズの一般逆行列が正則化項を加えて得られた逆行列とともに表示される。 前に逆行列を求める際にうまくいかないことへの対処として `np.linalg.inv()` にかえて `np.linalg.pinv()` を使えばよいとしてきたが、ムーアペンローズの一般逆行列は、正則化して得られる逆行列とは、とても似ているとは言えないことがわかる。

注：ここで説明してきた正則化項を加えて行う方法は「リッジ回帰」 (Ridge Regression) と呼ばれる。 リッジ回帰では重みの二乗の合計、すなわち L_2 ノルムを加えるが、このほかに学習した重みの絶対値の合計 (これを L_1 ノルムという) を加える方法もあり、こちらは「ラッソ回帰」 (Lasso Regression) と呼ばれる。

さて、 \mathbf{W} の求め方がわかったので、最小二乗法で答えを求めたのと同じケースについて確認しておこう。 正則化項を加えた二乗和誤差は、

$$L = \frac{1}{2}(\varepsilon_0^2 + \varepsilon_1^2 + \varepsilon_2^2) + \frac{\lambda}{2}(w_0^2 + w_1^2) \quad (3.6.29)$$

L の w_0 に対する勾配は、

$$\begin{aligned}
\frac{\partial L}{\partial w_0} &= \frac{\partial}{\partial w_0} \left\{ \frac{1}{2} (\varepsilon_0^2 + \varepsilon_1^2 + \varepsilon_2^2) \right\} + \frac{\partial}{\partial w_0} \left\{ \frac{\lambda}{2} (w_0^2 + w_1^2) \right\} \\
&= \frac{\partial}{\partial w_0} \left\{ \frac{1}{2} (\varepsilon_0^2) \right\} + \frac{\partial}{\partial w_0} \left\{ \frac{1}{2} (\varepsilon_1^2) \right\} + \frac{\partial}{\partial w_0} \left\{ \frac{1}{2} (\varepsilon_2^2) \right\} + \frac{\partial}{\partial w_0} \left(\frac{\lambda}{2} w_0^2 \right) + \frac{\partial}{\partial w_0} \left(\frac{\lambda}{2} w_1^2 \right) \\
&= \frac{\partial}{\partial \varepsilon_0} \left\{ \frac{1}{2} (\varepsilon_0^2) \right\} \frac{\partial \varepsilon_0}{\partial w_0} + \frac{\partial}{\partial \varepsilon_1} \left\{ \frac{1}{2} (\varepsilon_1^2) \right\} \frac{\partial \varepsilon_1}{\partial w_0} + \frac{\partial}{\partial \varepsilon_2} \left\{ \frac{1}{2} (\varepsilon_2^2) \right\} \frac{\partial \varepsilon_2}{\partial w_0} + \lambda w_0 \\
&= \varepsilon_0 + \varepsilon_1 + \varepsilon_2 + \lambda w_0
\end{aligned} \tag{3.6.30}$$

L の w_1 に対する勾配は、

$$\begin{aligned}
\frac{\partial L}{\partial w_1} &= \frac{\partial}{\partial w_1} \left\{ \frac{1}{2} (\varepsilon_0^2 + \varepsilon_1^2 + \varepsilon_2^2) \right\} + \frac{\partial}{\partial w_1} \left\{ \frac{\lambda}{2} (w_0^2 + w_1^2) \right\} \\
&= \frac{\partial}{\partial w_1} \left\{ \frac{1}{2} (\varepsilon_0^2) \right\} + \frac{\partial}{\partial w_1} \left\{ \frac{1}{2} (\varepsilon_1^2) \right\} + \frac{\partial}{\partial w_1} \left\{ \frac{1}{2} (\varepsilon_2^2) \right\} + \frac{\partial}{\partial w_1} \left(\frac{\lambda}{2} w_0^2 \right) + \frac{\partial}{\partial w_1} \left(\frac{\lambda}{2} w_1^2 \right) \\
&= \frac{\partial}{\partial \varepsilon_0} \left\{ \frac{1}{2} (\varepsilon_0^2) \right\} \frac{\partial \varepsilon_0}{\partial w_1} + \frac{\partial}{\partial \varepsilon_1} \left\{ \frac{1}{2} (\varepsilon_1^2) \right\} \frac{\partial \varepsilon_1}{\partial w_1} + \frac{\partial}{\partial \varepsilon_2} \left\{ \frac{1}{2} (\varepsilon_2^2) \right\} \frac{\partial \varepsilon_2}{\partial w_1} + \lambda w_1 \\
&= 3\varepsilon_0 + 5\varepsilon_1 + 4\varepsilon_2 + \lambda w_1
\end{aligned} \tag{3.6.31}$$

これをまとめて、

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \varepsilon_0 + \varepsilon_1 + \varepsilon_2 + \lambda w_0 \\ 3\varepsilon_0 + 5\varepsilon_1 + 4\varepsilon_2 + \lambda w_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} + \begin{pmatrix} \lambda w_0 \\ \lambda w_1 \end{pmatrix} \tag{3.6.32}$$

いっぽう式 3.2.4 は $\Phi \mathbf{W} = \mathbf{T} + \mathbf{E}$ すなわち、

$$\begin{pmatrix} 1 & 3 \\ 1 & 5 \\ 1 & 4 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} + \begin{pmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} \tag{3.6.33}$$

各項に左から ${}^t\Phi$ をかけて

$$\begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} 1 & 3 \\ 1 & 5 \\ 1 & 4 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} + \begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} \tag{3.6.34}$$

式 3.6.32 で L を最小にする \mathbf{w} で $\frac{\partial L}{\partial \mathbf{w}} = \mathbf{0}$ だから、

$$\begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \varepsilon_2 \end{pmatrix} = - \begin{pmatrix} \lambda w_0 \\ \lambda w_1 \end{pmatrix} \quad (3.6.35)$$

式 3.6.34 と式 3.6.35 から、

$$\begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} 1 & 3 \\ 1 & 5 \\ 1 & 4 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} + \begin{pmatrix} \lambda w_0 \\ \lambda w_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 3 & 5 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 3 \end{pmatrix} \quad (3.6.36)$$

ここで、 $\begin{pmatrix} \lambda w_0 \\ \lambda w_1 \end{pmatrix} = \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \end{pmatrix}$ だから

$$\therefore \begin{pmatrix} w_0 \\ w_1 \end{pmatrix} = \left(\begin{pmatrix} 3 & 12 \\ 12 & 50 \end{pmatrix} + \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right)^{-1} \begin{pmatrix} 8 \\ 33 \end{pmatrix} \quad (3.6.37)$$

λ に具体的な値を与えれば、 \mathbf{w} を算出できる。

では、正則化項を入れる場合の必要部分を次に示そう。 ${}^t\Phi\Phi$ を ${}^t\Phi\Phi + \lambda I$ で置き換えるだけだから次のようになる。

リスト 3.6.5：正則化項を加える

```
class LinearCombination: # Linear Combination of Basis Functions
    def regression(self, x, t, r=0.0):
        if self.phi is None:
            raise Exception('Need to initialize specifying basis function')
        Pi = self.phi(x)
        tPiPi = np.dot(Pi.T, Pi)
        I = np.eye(len(tPiPi))
        tPiPi += r * I          # 正則化項を加える
        tPiPiI = np.linalg.inv(tPiPi)
        PiI = np.dot(tPiPiI, Pi.T)
        w = np.dot(PiI, t)
        print('Parameters are fixed.')
        self.w = w
        return w
```

「リスト 3.6.1：線形結合で定義する関数」の regression() メソッドの計算部分で tPiPi で計算した後、次の 2 行を書き加えるだけである。すなわち、

```
I = np.eye(len(tPiPi))
```

で、numpy の関数を使って単位行列をつくる。このとき必要な大きさは tPiPi から知ることができる。それから、

```
tPiPi += r * I
```

で tPiPi に正則化項を加える。

材料がそろったので、ここまでのところを整理する意味も含めて、データの正規化と重みの正則化を入れて線形回帰を汎用化したプログラムを示そう。

リスト 3.6.6：線形回帰(直接法)

```
import numpy as np
import matplotlib.pyplot as plt
from ufiesia import LCBF
import data_flu

m = 9                                # 基底関数の次数
s = 2.0                              # ガウス基底の広がり係数
normalize = True                      # 正規化の指定
r = 0.0001                           # 正則化係数

data = np.array(data_flu.data)
X = data[:, 0]                       # 入力
T = data[:, 1]                       # 正解値

# -- 入力データの正規化 --
if normalize:
    xmax = np.max(X); xmin = np.min(X)
    X = (X - xmin) / (xmax - xmin) * 2 - 1
    s = s / (xmax - xmin) * 2 # 広がり幅もデータに合わせてスケールリング

# -- 対象を定義し回帰 --
model = LCBF.LinearCombination(LCBF.PolynormalBasis(m))
model.regression(X, T, r)

# -- グラフ描画用 --
plt_x = np.linspace(min(X), max(X), 1000)
plt_y = model(plt_x)

# -- 逆変換で x 軸をもとに戻す --
if normalize:
    X = (X + 1) / 2 * (xmax - xmin) + xmin
    plt_x = (plt_x + 1) / 2 * (xmax - xmin) + xmin

# -- グラフ表示 --
plt.grid()
plt.scatter(X, T, marker='x', s=100)
plt.scatter(plt_x, plt_y, marker='.', s=1)
plt.xlabel('input')
plt.ylabel('output')
plt.show()
plt.ylabel('output')
plt.show()
```

「リスト 3.6.2」と基本的には同じなので、それとの違いについて説明する。冒頭で正規化と正則化の指定のため、次の 2 行が追加されている。

```
normalize = True          # 正規化の指定
```

```
r = 0.0001               # 正則化係数
```

また入力データの正規化は、次の 4 行で行う。

```
if normalize:
```

```
    xmax = np.max(X); xmin = np.min(X)
```

```
    X = (X - xmin) / (xmax - xmin) * 2 - 1
```

```
    s = s / (xmax - xmin) * 2 # 広がり幅もデータに合わせてスケーリング
```

ここで、入力データの正規化で X が $-1 \sim +1$ の範囲にスケーリングされるので、これに合わせてガウス基底のベルカーブの広がり係数 s もスケーリングしている。しかし、ガウス基底の場合には、多項式基底の場合と異なり計算途中で極大値が出ることもないから、正規化をやめてしまっても問題ない。

regression() メソッドに正則化項の足しこみを指定するため、

```
model.regression(X, T, r)
```

で先に指定した r を渡す。

正規化を行ったので、グラフを描く前に逆変換で戻している。 X そのものはさておき、グラフ表示用の `plt_x` についても逆変換するのを忘れてはならない。この `plt_x` は `np.linspace()` で生成する際、正規化後の X を基準にしているから、逆変換だけが必要になる。

```
if normalize:
```

```
    X = (X + 1) / 2 * (xmax - xmin) + xmin
```

```
    plt_x = (plt_x + 1) / 2 * (xmax - xmin) + xmin
```

それではデータは先のインフルエンザの流行のものとして実行結果を示そう。図 3.6.3 は多項式基底で、 $m=9$ 、`normalize='True'`、 $r=0.0001$ の場合である。

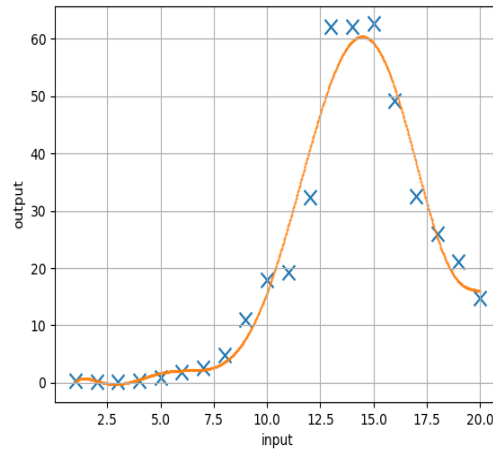


図 3.6.3 インフルエンザ～多項式9次正規化正則化

かわりばえはしないと見えるであろうが、図 3.6.2 の多項式基底であればいた流行の始まりのあたりがきれいになった。

ここで、次のデータを試してみよう。

リスト 3.6.7：8 点のデータ

```
data = [[4, 7], [6, 18], [8, 10], [11, 7], [13, 11], [15, 5], [17, 14], [19, 12]]
data = np.array(data, dtype='float')
```

「リスト 3.6.6：線形回帰(直接法)」データを指定する行を「リスト 3.6.7」で置き換えるか、さもなくば、その後に追記して変数 `data` を上書きすればよい。なおここではデータ中の数値がすべて整数なので、numpy の array にする際に、浮動小数点形式を指定した。

多項式基底でも観測値の入力データを正規化し、 \mathbf{w} を求める際に正則化項を加えることによって、基底関数の次数 m を大きくして、データの数 n を上回るようにした場合も含めて、それなりの結果が得られるようになる。

ここで実行結果の一例を示す。

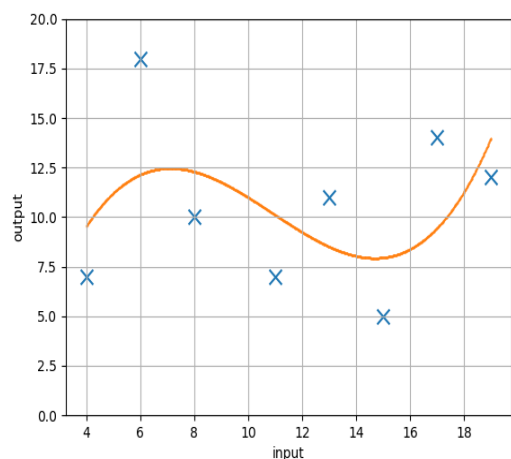


図 3.6.4 8点のデータ～多項式基底

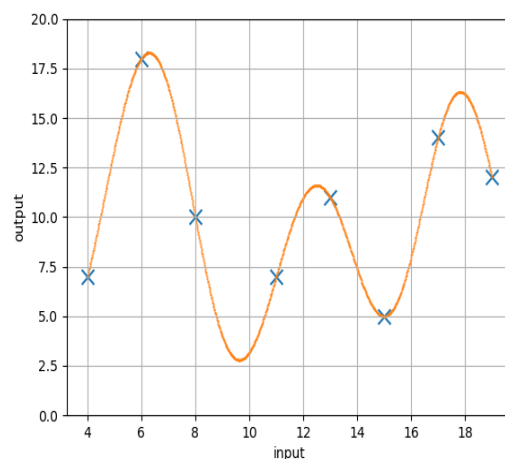


図 3.6.5 8点のデータ～ガウス基底

右の図はデータが 1 点でも増えるとそのたびに線の形状がころころ変わり感心しないが、こんなものも計算できてしまうことも含めて、基底関数の種類や次数、正規化のありなし、正則化など、いろいろ変えてプログラムを走らせて、それらの影響を見てほしい。そういうことを見るためにも、プログラムがエラーせずに結果を出せることはありがたいものである。

左図：多項式基底($m=4$ 、 $\text{normalize}='False'$ 、 $r=0.0$ 、)

右図：ガウス基底($m=13$ 、 $s=1.5$ 、 $\text{normalize}='False'$ 、 $r=0.00001$)

このデータに限らず、他のデータでもやってみてほしい。また、基底関数も多項式基底とガウス基底の両方とも、その次数の大小を変えて試してみてほしい。先のプログラムでは歯が立たなかった多項式基底で、この例に限らず、思い切って次数を高くても結果が得られるはずである。もちろん、それが好ましい回帰といえるかといえば、そうではないだろう。しかし、ここで線形回帰がちゃんと動くことを確認しておきたい。それによってディープラーニングのように、大量データを扱い、はるかに複雑なモデルを作って行う機械学習への道程で経験しておくべきことが多々あると考えるからである。

3.7 正規化と正則化を含めた勾配の可視化

本節のテーマである可視化の前に、正則化に関連してやっておきたいことがある。「リスト 3.6.6:線形回帰(直接法)」では、式 3.6.27 に従い、リスト 3.6.5 に示すように ${}^t\Phi\Phi$ を ${}^t\Phi\Phi + \lambda I$ で置き換えることで、重み W を得る際に正則化項の効果をいれた。しかし、もともと正則化項は式 3.6.13 に示すように、誤差関数の値 L を評価するにあたって二乗和誤差に加えたものである。すなわち、

$$L = \frac{1}{2} \sum^n \varepsilon^2 + \frac{1}{2} \lambda \sum^m w^2 = \frac{1}{2} \sum^n (y - t)^2 + \frac{1}{2} \lambda \sum^m w^2 \quad (3.6.13 \text{ 再掲})$$

そこで前節とすこし話は前後するのだが、あらためて正則化項を加えた平均二乗和誤差を定義する。二乗和誤差は、すでに「リスト 3.4.1」で関数として用意したが、あらためて正則化項を含めて、今度はクラスとして作っておくことにする。

リスト 3.7.1：正則化項付きの平均二乗誤差

```
class MeanSquaredErrorDecay:
    def forward(self, y, t, r, w):
        loss = 0.5 * np.sum((y - t) ** 2)
        decay = 0.5 * r * np.sum(np.square(w))
        return (loss + decay) / len(y)
```

forward メソッドの引数として正則化項に必要な変数として重み w そのものと正則化係数 r を渡す。正則化項の誤差関数に対する影響項は decay として算出して、返り値を作る際に足しこんでいる。なお返り値を作る際に $\text{len}(y)$ によるデータ点の数で割って、誤差の二乗の和ではなくて平均にして、返り値の絶対値がデータ点の数によって変わるのを避けるようにした。

正則化項を入れた誤差関数を定義した理由は、いうまでもなく誤差関数の値 L そのものが必要だからである。いっぽう前節 3.6 では W が得られれば良かった。今後、「3.8 数値微分法による回帰」や「3.9 誤差逆伝播法による回帰」でも誤差関数の値 L が必要であり、正則化項を考慮する際には、ここで定義する MeanSquaredErrorDecay クラスを用いることになる。それらの場合、誤差関数の値 L を最小にする W を得る際に、誤差関数に正則化項が加えられたもので勾配を算出することによってはじめて、正則化項の効果の含まれる W の最適値を得ることができる。

もうひとつ勾配を可視化するにあたって、「リスト 3.3.2」で作った 3 次元グラフの色分けもしておこう。

リスト 3.7.2 : 3 次元グラフの色分け

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

def graph3dc(func, rx=(0,1), ry=(0,1), label=None, cm='gnuplot'):
    qx = np.linspace(*rx, 100)
    qy = np.linspace(*ry, 100)
    X, Y, Z = [], [], []
    for x in qx:
        for y in qy:
            z = func(x, y)
            X.append(x)
            Y.append(y)
            Z.append(z)

    fig = plt.figure()
    ax = Axes3D(fig)
    if label is not None:
        ax.set_xlabel(label[0])
        ax.set_ylabel(label[1])
        ax.set_zlabel(label[2])
    cm = plt.cm.get_cmap(cm)
    ax.scatter(X, Y, Z, c=Z, cmap=cm, marker='.')
    plt.show()
```

これは「リスト 3.3.2 : 3 次元グラフ」で定義した関数 `graph3d()` とほとんどまったく同じだから違いだけを説明する。関数は一応名前を変えて `graph3dc()` とした。引数にはカラーマップ `cm` を加えるが、ここではデフォルトを `cm='gnuplot'` とした。もちろん関数を呼出す際に別のカラーマップを指定することもできる。カラーマップはいろいろ用意されていて、詳細は下記を参照。

<https://matplotlib.org/stable/tutorials/colors/colormaps.html>

あとは `ax.scatter()` の前に、

```
cm = plt.cm.get_cmap(cm)
```

をによって引数の `cm` で指定されたカラーマップを取得してから、

```
ax.scatter(X, Y, Z, c=Z, cmap=cm, marker='.')
```

を実行すればよい。 `plt.show()` でグラフが描画されるのも同じ。

さてそれでは勾配を可視化するプログラムの本体を次に示す。基本的にやることは、「リスト 3.3.3：重み W と損失 L の関係」と同じだが、すでに「リスト 3.6.1」で基底関数を束ねたものとして線形結合を LinearCombination クラスで定義し、これを使って「リスト 3.6.2」で線形回帰を行っている。そこでこれにならって、LinearCombination クラスを使うことにする。そして本節のテーマとなるデータの正規化と正則化項を含めて、重み W と損失 L の関係を見られるようにする。

リスト 3.7.3：重み W と損失 L の関係～正規化・正則化

```
import numpy as np
from ufiesia import LCBF

r = 0.0
normalize = False

data = np.array([[3, 2], [5, 3], [4, 3]])
X = data[:, 0]    # 入力
T = data[:, 1]    # 正解値

# ー 入力データの正規化 ー
if normalize:
    xmax = np.max(X); xmin = np.min(X)
    X = (X - xmin) / (xmax - xmin) * 2 - 1

# ー 対象となる関数 ー
model = LCBF.LinearCombination(LCBF.RectilinearBasis())
loss = LCBF.MeanSquaredErrorDecay()

def target_function(*w): # XとTとrはglobal変数
    model.w = w
    y = model(X)
    l = loss.forward(y, T, r, w)
    return l

rw0 = -10, 10 #0.6667, 0.6667
rw1 = -10, 10 #0.5, 0.5
label = 'W0', 'W1', 'L'
LCBF.graph3dc(target_function, rw0, rw1, label)
```

「リスト 3.3.3」や「リスト 3.6.6」と比べながら見ていくことにする。冒頭で正則化係数 r を与える。ここでは $r = 0.0$ としていて、この場合は正則化項は 0 になり正則化しない。それから、normalize = True、False を指定して、入力データの正規化をやる、やらないを決める。データを用意し、正規化が指定されたら適用し、その後に、次の 2 行で線形結合と損失関数をインスタンス化する。


```
model = LCBF.LinearCombination(LCBF.RectilinearBasis())
```

```
loss = LCBF.MeanSquaredErrorDecay()
```

それから、インスタンス化したmodelとlossを使って可視化対象の関数

target_function()を定義して、「リスト 3.3.3」と同様、w0とw1の範囲を指定してグラフを描く。このとき線形結合modelの重みmodel.wをtarget_function()の引数から設定する必要があるから、model.w = wの一文を忘れてはならない。

それでは結果を見ていこう。図 3.7.1 に示すのは、 $r = 0.0$ 、`normalize = False` で正規化しないし、正規化もしない場合である。ただし他の場合との比較の都合でWの値の範囲は-10, 10に広げた。

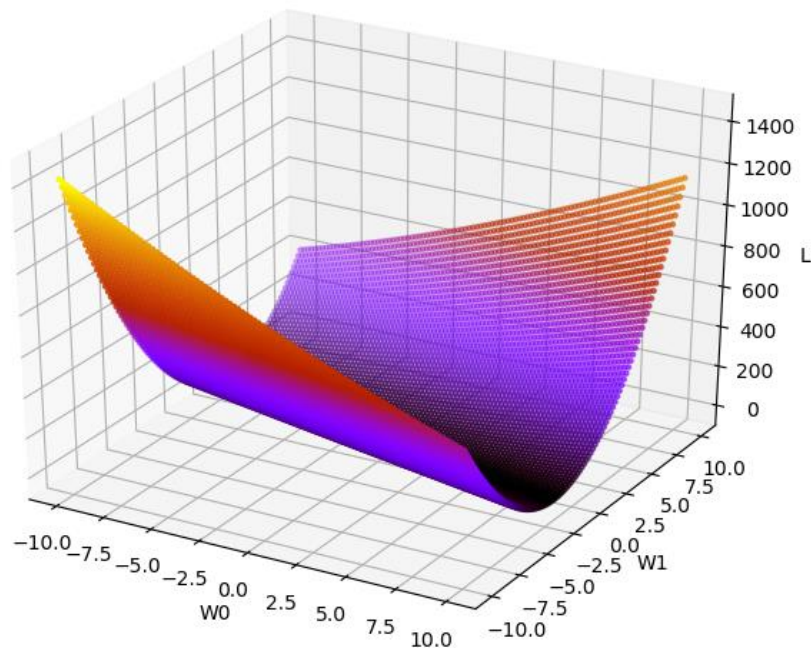


図 3.7.1 w の値と $L(r = 0.0, \text{normalize} = \text{False})$

図 3.3.1 と値の範囲は違うが形状はほとんど変わらない。このグラフで言えるのは、次の点である。すなわち、w1によってLが大きく変化しw1=0付近で最小となる。いっぽう、w0に対してLの変化はw1に依存してねじれたようになっているものの、どこで最小になるか分かりにくく、Lを最小にするw0の値が読み取れない。この点は、図 3.3.2 で示したように、w1の値を固定してw0とLの関係を描けば見えてくる。しかしそれでも、w0とw1の組として答えを得ようとする場合にw0の方に困難さがあることが見てとれるのであり、節 3.6 で述べたデータの正規化が必要な理由が、あながち当たっていないとは言えない。

次にデータの正規化を行った場合を見てみよう。normalize = True として正規化を行った場合を図 3.7.2 に示す。グラフの形状には顕著な変化がみられる。今度は w_0 、 w_1 ともに最適な値を底として、そこから大小どちらに外れても L が大きくなることがわかる。正規化を行った場合の W の解は、式 3.6.4 に示すように $w_0=2.6667$ 、 $w_1=0.5000$ だった。グラフから目視で細かい数値を読み取るのは困難だが、グラフと解が合致することはわかる。そしてなによりも、図 3.7.1 の正規化しない場合と違って、 w_0 と w_1 の L への影響が均等なバランスとなっており、その解が自然に導かれることがグラフの形状から明らかである。

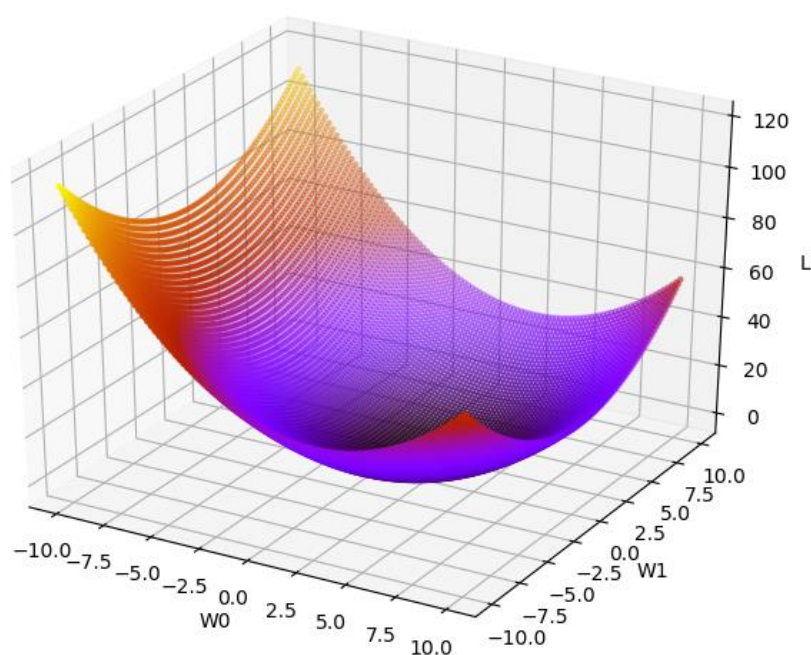


図 3.7.2 w の値と $L(r = 0.0$ 、normalize=True)

最後に正則化項を加えた場合を見てみよう。normalize = False に戻し、かわりに $r = 1.0$ として正規化を行った場合を図 3.7.3 に示す。今度はグラフの形状はほとんど図 3.7.1 に戻ってしまった。これを見る限りは、正規化に見られるような w の要素間のバランスといった点での効果は無いようだ。しかしもちろん $r = 100$ のように極端に大きくすれば、図 3.7.4 に示すように w_0 も w_1 も 0 から離れていくと L が大きくなるのは式 3.6.13 から明らかである。ともあれ正則化項については、むしろ正則化項を加えても勾配に対する影響が小さい、つまり変な影響を与えて悪さをしない、ということである。そして効果としては、すでに式 3.6.28 に示すように逆行列を求めたい行列が正則化されるという点である。

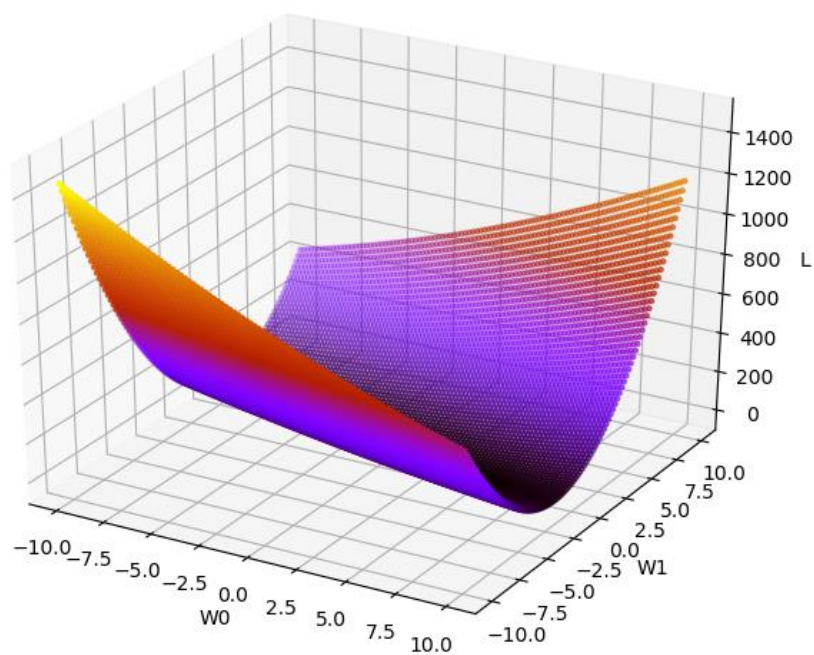


図 3.7.3 w の値と L ($r=1.0$, $\text{normalize} = \text{False}$)

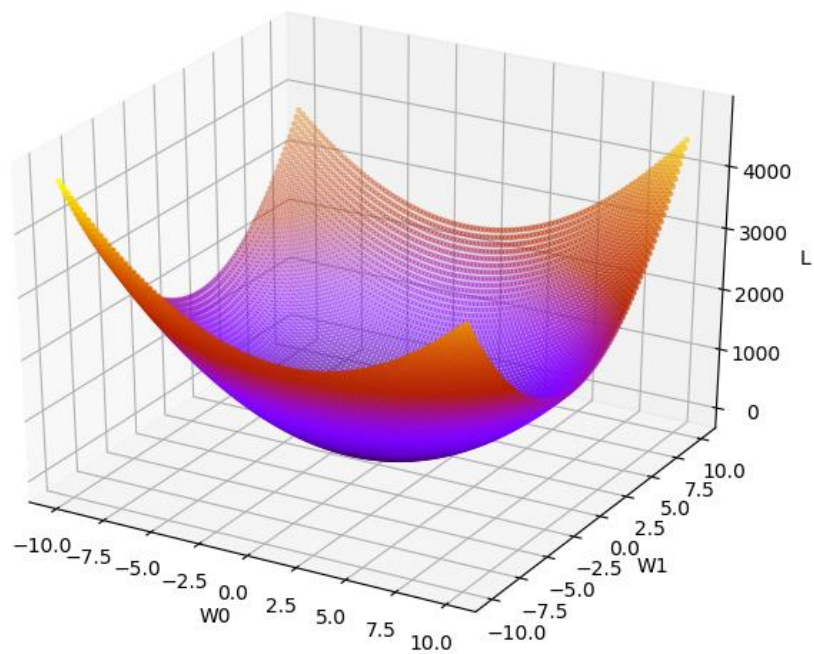


図 3.7.4 w の値と L ($r=100$, $\text{normalize} = \text{False}$)

ここまでデータの正規化と正則化項の影響について見てきたが、両者を併せて行うことで、線形回帰の解を得ることがうまくできるようになることがわかった。節 3.4 において、図 3.4.3、図 3.4.4、図 3.4.5 を描くにあたって、説明を省略してきたのは、実はこの点なのであった。これは、線形回帰に限らず機械学習のさまざまな場面で正規化と正則化が求められる理由であって、ディープラーニングでも同様であり、なくてはならない技術なのである。

3.8 数値微分法による回帰

線形回帰(直接法)では、誤差関数の値 L を最小にする \mathbf{W} を求めるために、 $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{0}$ を解いた。たとえば、図 3.3.1~図 3.3.3 や図 3.7.1~図 3.7.3 のような場合であれば、 \mathbf{W} に対して L は凹型になっていて、その一番低くなっている底の平らなところを解として求めたのである。これに対して、どこが底か分からない場合に、どこか適当なところから、勾配を少しずつ下っていき、最後に底の一番低いところに至るという方法が考えられる。このやり方を勾配法といい、はじめに適当な $\mathbf{W} = \mathbf{W}^{(0)}$ を与えて、誤差関数の値 L の、その時点(時刻 t) の重み $\mathbf{W}^{(t)}$ に対する勾配 $\frac{\partial L}{\partial \mathbf{W}^{(t)}}$ を求め、その勾配に従って \mathbf{W} を調節して、徐々に L を小さくしていき、この繰り返し処理によって最適な \mathbf{W} を得る。すなわち、学習率を ε として、次式(時刻 $t \rightarrow$ 時刻 $t+1$)を繰り返す。

$$\mathbf{W}^{(t+1)} = \mathbf{W}^{(t)} - \varepsilon \frac{\partial L}{\partial \mathbf{W}^{(t)}} \quad (3.8.1)$$

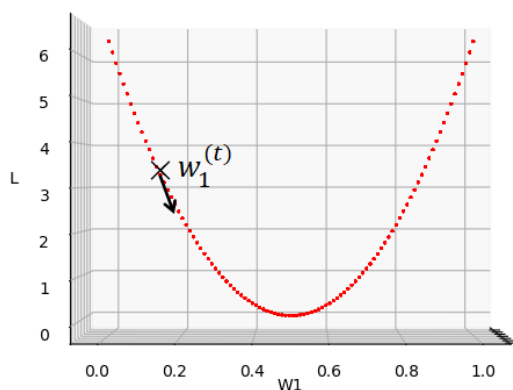


図 3.8.1 w_1 に対する L の変化(図 3.3.3 から引用)

具体例で説明しよう。図 3.8.1 は、図 3.3.3 に説明のため加筆したものだが、時刻 t で w_1 が 0.1 より少し大きい値で \times のところにいるとする。ここでの勾配 $\frac{\partial L}{\partial w_1^{(t)}}$ は矢印で示すように負の値となっている。すなわち w_1 が大きくなるほど L は小さくなる。そこでこの勾配に ε

をかけた値を w_1 から差し引く、つまりその分 w_1 を大きくすることで、 L が最小の底に近づいていくのである。ここで例として挙げた図 3.3.1～図 3.3.3 や図 3.7.1～図 3.7.3 のような場合であれば可視化も容易で、底の在処も明らかであるから、勾配法は回りくどい方法に思われるかもしれない。しかし $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{0}$ を直接解くことができないような場合でも、勾配法では最適な \mathbf{W} を得ることができる。実はディープラーニングではデータの正規化を行ってもなお、図 3.7.2 に示すような単純な凹型とならず、学習が難しい場合が多々ある。そのような場合には、式 3.8.1 を基本としつつ \mathbf{W} を最適値に寄せていくやり方に様々な工夫がなされることが多い。

勾配 $\frac{\partial L}{\partial \mathbf{W}^{(t)}}$ は数値微分によって求めることができる。すなわち、 $\mathbf{W} = (w_0, w_1, \dots, w_m)$ の要素を一つずつ取り出して、関数 L をその \mathbf{W} の要素 w_j ($j = 0 \sim m$) の関数とみなし、また計算の都合から、 $h = 1 \times 10^{-7}$ 程度の固定値にして、

$$\left\{ \begin{array}{l} \frac{\partial L}{\partial w_0^{(t)}} = \lim_{h \rightarrow 0} \frac{L(w_0^{(t)} + h) - L(w_0^{(t)} - h)}{2h} \doteq \frac{L(w_0^{(t)} + h) - L(w_0^{(t)} - h)}{2h} \\ \vdots \\ \frac{\partial L}{\partial w_m^{(t)}} = \lim_{h \rightarrow 0} \frac{L(w_m^{(t)} + h) - L(w_m^{(t)} - h)}{2h} \doteq \frac{L(w_m^{(t)} + h) - L(w_m^{(t)} - h)}{2h} \end{array} \right. \quad (3.8.2)$$

注意： \mathbf{W} の要素一つずつ w_j ($j = 0 \sim m$) について算出するが、その際に注目している要素 $w_j \pm h$ とし、これ以外の \mathbf{W} の要素は、その時点の(元の)値を持つようにする必要がある。すなわち、 $j = 0, j = 1, \dots$ とずらしながら、 $\mathbf{W} = (w_0^{(t)}, \dots, w_{j-1}^{(t)}, w_j^{(t)} \pm h, w_{j+1}^{(t)}, \dots, w_m^{(t)})$ として、上記を $m + 1$ 回計算する。そしてその結果が \mathbf{W} の各要素の勾配となる。これをまとめて次のように表記する。

$$\frac{\partial L}{\partial \mathbf{W}^{(t)}} = \frac{L(\mathbf{W}^{(t)} + h) - L(\mathbf{W}^{(t)} - h)}{2h} \quad (3.8.3)$$

数値微分を python の関数として定義する。関数の引数として、微分対象の関数、その微分を行う変数を与え、微分結果を返り値とする。

リスト 3.8.1：数値微分

```
def numerical_gradient(func, x, h=1e-7): # 数値微分、変数 x は配列に対応
    grad = np.empty(x.shape)
    for i, xi in enumerate(x):          # x の要素を順に選んで±h
        x[i] = xi + h                    # 要素 x[i]を+h、他はそのまま
        fxph = func(x)
        x[i] = xi - h                    # 要素 x[i]を-h、他はそのまま
        fxmh = func(x)
        grad[i] = (fxph - fxmh) / (2 * h)
        x[i] = xi                        # x[i]をもとに戻す
    return grad
```

ここでは、 $\frac{\partial L}{\partial \mathbf{W}}$ を求めたいので、この関数を呼び出す際に、引数として func には L を、 \mathbf{x} には \mathbf{W} を与える。for 文の中を、引数で渡された \mathbf{W} の要素分繰り返して、個々の要素すなわち、 w_0, \dots, w_m による微分の近似値を求めて、 \mathbf{W} と同じ形状の変数 grad に並べる。返り値 grad は $\frac{L(\mathbf{W}+h)-L(\mathbf{W}-h)}{2h}$ となる。

ではさっそく、数値微分法による線形回帰のプログラムを作ろう。これを「リスト 3.8.2：線形回帰数値微分法」に示す。「リスト 3.6.6：線形回帰(直接法)」と基本的な流れは同じで、回帰のための計算の部分を勾配法にすげかえる。

リスト 3.8.2：線形回帰数値微分法

```

import numpy as np
import matplotlib.pyplot as plt
from ufiesia import LCBF
import data_flu

m = 9 # 基底関数の次数
s = 2.0 # ガウス基底の広がり係数
normalize = False
r = 0.0 #5e-4 # フィッティングの正則化係数

data = np.array(data_flu.data)
X = data[:, 0] # 入力
T = data[:, 1] # 正解値

# -- 入力データの正規化 --
if normalize:
    xmax = np.max(X); xmin = np.min(X)
    X = (X - xmin) / (xmax - xmin) * 2 - 1
    s = s / (xmax - xmin) * 2 # 広がり幅もデータに合わせてスケーリング

# -- 対象となる関数と損失(正則化項つき) --
model = LCBF.LinearCombination(LCBF.GaussianBasis(m, s))
loss_func = LCBF.MeanSquaredErrorDecay()

# -- パラメタの初期値は乱数 --
model.w = np.random.randn(m)

# -- 勾配降下法によるパラメタのチューニング --
lr = 0.1 # 学習係数
for i in range(1000):
    func = lambda w: loss_func.forward(model(X), T, r, w)
    grad_w = LCBF.numerical_gradient(func, model.w)
    model.w -= lr * grad_w

print('算出したパラメタ\n', model.w)

# -- グラフ描画用 --
plt_x = np.linspace(min(X), max(X), 1000)
plt_y = model(plt_x)

# -- 逆変換で x 軸をもとに戻す --
if normalize:
    X = (X + 1) / 2 * (xmax - xmin) + xmin
    plt_x = (plt_x + 1) / 2 * (xmax - xmin) + xmin

# -- グラフ表示 --
plt.grid()
plt.scatter(X.tolist(), T.tolist(), marker='x', s=100) # 入力
plt.scatter(plt_x.tolist(), plt_y.tolist(), marker='.', s=1) # 近似
plt.xlabel('input')
plt.ylabel('output')
plt.show()

```


データを用意して正規化するところまでは、「リスト 3.6.6」と同じ。その後に対象となる関数の部分で、線形結合だけでなく、平均二乗誤差も含め次の2行でインスタンス化する。誤差関数の平均二乗誤差は「リスト 3.7.1」で定義した正則化項を加えたものである。正則化項を含めて数値微分によって勾配を求めることによって、正則化項の効果の含まれる W の最適値を得る。

```
model = LCBF.LinearCombination(LCBF.GaussianBasis(m, s))
```

```
loss_func = LCBF.MeanSquaredErrorDecay()
```

続いて、 W の初期値を乱数で与え、学習係数 lr を設定してから、for ループで学習を繰り返す（この場合1000回）。

学習に際して勾配を求めて、求めた勾配に学習率をかけて W を更新するのだが、ここで数値微分の変数と関数の関係を λ 式によって再定義している。その λ の前に

```
loss_func.forward(model(X), T, r, w)
```

の部分を見ておこう。まず $\text{model}(X)$ で線形結合 model に入力を与えられて順伝播に相当する `__call__()` メソッドが実行される。それから、その結果が、正解値 T 、正則化係数 r 、重み w とともに、損失関数 loss_func の `forward()` メソッドに与えられて順伝播されるのだ。そしてこの線形結合と損失関数をつなげた $\text{loss_func.forward}(\text{model}(X), T, r, w)$ に対して、 λ 式で再定義する func の引数を w としている。そして $\text{func} = \lambda \dots$ の1行は、次の3行で置き換えても同じだ。

```
def func(w):
```

```
    y = model(X)
```

```
    return loss_func.forward(y, T, r, w)
```

ともかく w を引数として再定義した func を、 w に線形結合の重み model.w を与えて、 $\text{LCBF.numerical_gradient}$ によって勾配 grad_w を得る。数値微分に渡す際に $\text{func}(w)$ のようにみせて、その w に model.w を入れていることに注意されたし。いずれの表現をしたとしても、引数のひとつ w を、数値微分の関数 $\text{numerical_gradient}$ を呼び出す際に、対象の関数 func とともに明示する必要がある。いっぽう、他の引数 X と T と r は、 $\text{numerical_gradient}$ を実行するのに必要であるが、その引数ではないように見せる必要がある。このために λ を使うか否かはさておき、新たに別の関数として定義しなおす必要がある。そしてこの X と T と r は、 func の定義の外で定義されて、グローバル変数として扱われる。なお func はループの外で定義しても動くが、たとえばオンライン学習やミニバッチ学習のように X とそれに対応する T を一部を切り出しながら学習するような場合を想定して、それに備えて、ここではこのようにしている。

学習が終わると、得られた W を表示しグラフを描くのは、「リスト 3.6.6」と同じ。

では「リスト 3.8.2」の実行結果を示そう。

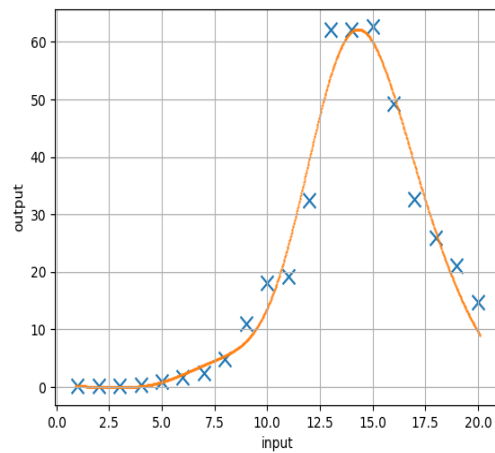


図 3.8.1 インフルエンザ～数値微分法

図は、ガウス基底で、 $m=9$ 、 $s=2$ 、 $\text{normalize}='False'$ 、 $r=0.0$ とした場合のものである。

学習回数を増やしていくと、直接法で行った図 3.6.1 や図 3.6.2 に示す結果にさらに近づく。しかしかえってこれくらいの方が自然なのではないだろうか？

多項式基底では観測値の入力データを正規化しないと、高次の基底関数の値が巨大になって学習しない場合があることを含め、基底関数の種類やその次数をはじめとして、データとの組み合わせにおいて、どういう選択をすれば、うまくデータをなぞってくれるのかを様々な試してみると良い。

3.9 誤差逆伝播法による回帰

数値微分により勾配 $\frac{\partial L}{\partial \mathbf{W}^{(l)}}$ を求めることは、基底関数の次元数＝重みの数にしたがって計算

回数が多くなり、時間もかかってしまう。そこで勾配 $\frac{\partial L}{\partial \mathbf{W}^{(l)}}$ を求めるのに、目的関数を定義

する際に、逆伝播を含めて定義しておき、これを利用することで、計算回数を大幅に節約することができる。ディープラーニングのように層を積み重ねるような場合、その出力から入力へと順に層を遡って誤差を伝播していく。このような場合に最適解を得るために誤差逆伝播法はなくてはならない。ただし、この逆伝播の定義はえてして厄介であり、バグが入り込む余地が大きい。そこで、順伝播だけを使う数値微分法を検証手段として使うことはたいへん有用である。そこでこれから逆伝播の式を導くことにする。

[二乗和誤差の逆伝播]

二乗和誤差の定義＝順伝播を振り返ると、

$$L = \frac{1}{2} \sum \varepsilon^2 = \frac{1}{2} \sum (y - t)^2 = \frac{1}{2} (y_0 - t_0)^2 + \dots + \frac{1}{2} (y_n - t_n)^2 \quad (3.9.1)$$

これを Y について微分する。

$$\frac{\partial L}{\partial y_0} = y_0 - t_0, \dots, \frac{\partial L}{\partial y_n} = y_n - t_n \quad (3.9.2)$$

これらをまとめて gY とおくと、

$$gY = \frac{\partial L}{\partial Y} = Y - T = (y_0 - t_0, y_1 - t_1, \dots, y_n - t_n) \quad (3.9.3)$$

二乗和誤差を最小にするのが最適化の目的なので、二乗和誤差の出力側(下流)は終端となっているが、一般化するために、逆伝播の際に gl が下流から伝播してくるものとする、二乗和誤差の逆伝播の式は次のようになる。

$$gY = gl(Y - T) \quad (3.9.4)$$

ただし通常は $gl = 1$

逆伝播の式がわかったので、これをプログラミングしておこう。「リスト 3.7.1：正則化項付きの平均二乗誤差」で MeanSquaredErrorDecay が python のクラスとして定義済みなので、これに backward() メソッドを追加する。

リスト 3.9.1：二乗和誤差に逆伝播を追加

```
class MeanSquaredErrorDecay:
    def forward(self, y, t, r, w):
        self.inputs = y, t, r, w
        loss = 0.5 * np.sum((y - t) ** 2)
        decay = 0.5 * r * np.sum(np.square(w))
        return (loss + decay) / len(y)

    def backward(self, gl=1):
        y, t, r, w = self.inputs
        gl /= len(y)
        gy = gl * (y - t)
        gw = r * gl * w
        return gy, gw
```

まず、backward() メソッドの前に forward() メソッドに、次の 1 行を追加する。

```
self.inputs = y, t, r, w
```

これは順伝播の時の値を逆伝播でも使用するためである。さて backward() メソッドだが、式の算出の際に説明したように、引数は下流からの勾配 gl=1 としている。しかしここでは、順伝播で返り値を作るときに len(y) で割っていてデータ数によらない平均にしているから、それに対応して、gl も len(y) で割っている。このあたりのところは間違えやすいので、数値微分の結果と照合して確かめるのが良い。gy = gl * (y - t) は勾配の式そのまま。いっぽう逆伝播の式の算出の説明の中では説明を省略したが、順伝播で正則化項が加えられているので、逆伝播では w に対する勾配も求める必要がある。順伝播の decay = r * 0.5 * np.sum(np.square(w)) を w で微分するのだから、gw = r * gl * w となる。

[線形結合の関数の逆伝播]

線形結合の関数の順伝播を振り返ると、

$$\mathbf{Y} = \Phi \mathbf{W} = \begin{pmatrix} \varphi_0(x_0) & \cdots & \varphi_m(x_0) \\ \vdots & \ddots & \vdots \\ \varphi_0(x_n) & \cdots & \varphi_m(x_n) \end{pmatrix} \begin{pmatrix} w_0 \\ \vdots \\ w_m \end{pmatrix} \quad (3.1.9 \text{ および } 3.1.21 \text{ 再掲})$$

あるいは、

$$\begin{cases} y_0 = f(x_0) = w_0\varphi_0(x_0) + w_1\varphi_1(x_0) + \cdots + w_m\varphi_m(x_0) \\ \vdots \\ y_n = f(x_n) = w_0\varphi_0(x_n) + w_1\varphi_1(x_n) + \cdots + w_m\varphi_m(x_n) \end{cases} \quad (3.1.20 \text{ 再掲})$$

これを W の各要素について微分すると

$$\begin{cases} \frac{\partial y_0}{\partial w_0} = \varphi_0(x_0), & \frac{\partial y_0}{\partial w_1} = \varphi_1(x_0), & \cdots & \frac{\partial y_0}{\partial w_m} = \varphi_m(x_0) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial w_0} = \varphi_0(x_n), & \frac{\partial y_n}{\partial w_1} = \varphi_1(x_n), & \cdots & \frac{\partial y_n}{\partial w_m} = \varphi_m(x_n) \end{cases} \quad (3.9.5)$$

これは Φ そのものである。すなわち、

$$\frac{\partial Y}{\partial W} = \Phi \quad (3.9.6)$$

逆伝播の際に gY が下流から伝播してくるものとする、線形結合の逆伝播の式は次のようになる。

$$gW = gY \cdot \Phi \quad (3.9.7)$$

これで線形結合の逆伝播の式がわかったのでプログラミングしておこう。「リスト 3.6.1：線形結合で定義する関数 $f(x) = w_0\varphi_0(x) + w_1\varphi_1(x) + \cdots + w_m\varphi_m(x)$ 」で LinearCombination が python のクラスとして定義済みなので、これに backward() メソッドを追加する。

リスト 3.9.2 : 線形結合で定義する関数 $f(x) = w_0\phi_0(x) + w_1\phi_1(x) + \dots + w_m\phi_m(x)$
に逆伝播を追加

```
class LinearCombination: # Linear Combination of Basis Functions
# 線形回帰の関数: phi0(x)*W[0] + phi1(x)*W[1] + . . . + phi(m-1)(x)*W[m-1]
# Pi = Σphi(x)
    def __init__(self, BF=None, dim=2, s=1.0):
        if BF in ('Linear', 'linear'):
            self.phi = RectilinearBasis()
        elif BF in ('Gauss', 'gauss', 'Gaussian', 'gaussian'):
            self.phi = GaussianBasis(dim, s)
        else:
            self.phi = PolynorminalBasis(dim)
        print('Initialize Linear Combination of Basis Functions', self.phi)

    def forward(self, x, w):
        self.Pi = self.phi(x)
        y = np.dot(self.Pi, w)
        return y

    def backward(self, gy=1):
        gw = np.dot(gy, self.Pi)
        return gw

    def regression(self, x, t, r=0.0):
        if self.phi is None:
            raise Exception('Need to initialize specifying basis function')
        Pi = self.phi(x).astype('float')
        tPiPi = np.dot(Pi.T, Pi)
        I = np.eye(len(tPiPi))
        tPiPi += r * I # 正則化項を加える
        tPiPiI = np.linalg.inv(tPiPi)
        PiI = np.dot(tPiPiI, Pi.T)
        w = np.dot(PiI, t)
        print('Parameters are fixed.')
        self.w = w
        return w
```

引数は下流からの勾配 $gy=1$ 。そして $gw = np.dot(gy, self.Pi)$ は勾配の式そのまま。このとき、 ϕ は順伝播でそのデータによって生成して $self.Pi$ としてインスタンス変数に記憶しておいたもの。backward メソッドの引数 $gy=1$ としているが、この gy には下流からの勾配を渡す。ここでは二乗和誤差の MeanSquaredErrorDecay クラスを下流に配置する。したがって逆伝播の際には、MeanSquaredErrorDecay クラスの backward メソッドの戻り値 gy を、この LinearCombination クラスの backward メソッドの引数として渡すことになる。

ここで、節 3.4 線形回帰(直接法)で求めた勾配との関係について少し整理しておこう。二乗和誤差は、それそのものが評価関数であって、通常は $gl = 1$ で逆伝播するから、これと、線形結合の逆伝播で得られる勾配の積は、

$$\begin{aligned}
 (Y - T) \Phi &= (y_0 - t_0 \quad \cdots \quad y_n - t_n) \begin{pmatrix} \varphi_0(x_0) & \cdots & \varphi_m(x_0) \\ \vdots & \ddots & \vdots \\ \varphi_0(x_n) & \cdots & \varphi_m(x_n) \end{pmatrix} \\
 &= (\varepsilon_0 \quad \cdots \quad \varepsilon_n) \begin{pmatrix} \varphi_0(x_0) & \cdots & \varphi_m(x_0) \\ \vdots & \ddots & \vdots \\ \varphi_0(x_n) & \cdots & \varphi_m(x_n) \end{pmatrix} \\
 &= \begin{pmatrix} \varepsilon_0 \varphi_0(x_0) + \cdots + \varepsilon_n \varphi_0(x_n) \\ \vdots \\ \varepsilon_0 \varphi_m(x_0) + \cdots + \varepsilon_n \varphi_m(x_n) \end{pmatrix} \tag{3.9.8}
 \end{aligned}$$

これは式 3.4.23 の w の勾配と同じものである。目的変数の計算値と観測値との差の並びのベクトル $(\varepsilon_0 \quad \cdots \quad \varepsilon_n)$ は、ここでは横ベクトルであるが、3.4 線形回帰(直接法)では縦ベクトルとして次のように定義した。

$$E = \begin{pmatrix} \varepsilon_0 \\ \vdots \\ \varepsilon_n \end{pmatrix} \tag{3.4.22 再掲}$$

そこで横ベクトルを縦ベクトルと区別して、

$${}^tE = (\varepsilon_0 \quad \cdots \quad \varepsilon_n) \tag{3.9.9}$$

とおくならば、

$$(Y - T) \Phi = {}^tE \Phi \tag{3.9.10}$$

式 3.9.10 は、二乗和誤差の逆伝播で得られた勾配に線形結合の逆伝播で得られた勾配を掛け合わせたものであるが、転置行列の積の公式によって、

$${}^tE \Phi = {}^t\Phi E \tag{3.9.11}$$

であるから式 3.4.25 により、

$$(3.9.12)$$

したがって、3.4 線形回帰(直接法)で求めた勾配と、ここで誤差逆伝播法により得られた勾配は同じである。言い換えるならば、二乗和誤差を評価関数として数学的に解いて得られた勾配と、二乗和誤差の逆伝播で得られた勾配に線形結合の逆伝播で得られた勾配を掛け合わせたものは同じである。これは誤差逆伝播法に根拠を与えるものであるが、同時に微分の連鎖律そのものにほかならない。

$$(3.9.13)$$

ここでは線形結合と二乗和誤差が連結されている簡単な場合について考察したが、このような簡単な例に限らず一般的に、各層の逆伝播で得られる勾配を連鎖律によって上流側へと伝え、掛け合わせていくことで、必要な勾配を求めることができるのである。ディープラーニングのように層を幾重にも積み重ねる場合には、系全体にわたり数学的に勾配を求めることは困難だが、層ごとに逆伝播をもとめることは比較的容易である。層ごとに勾配を求めてはそれを下流から上流へと伝えていくことで必要な勾配が得られるため、誤差逆伝播法は層が深い場合に有効な手段となるのであり、これなしにはディープラーニングは成立しえなかったのである。

念のために確認しておく、

$$(3.9.14)$$

注意：行列やベクトルの形状に注意が必要である。 上記の場合、 $\frac{\partial L}{\partial \mathbf{y}}$ は n 次元のベクトル、

$\frac{\partial Y}{\partial W}$ は n 行 m 列の行列であり、これらの積である $\frac{\partial L}{\partial W}$ は m 次元のベクトルとなっている

観測値が行列で与えられるような場合には、積の転置行列の法則 ${}^t(AB) = {}^tB {}^tA$ も考慮して、各行列とベクトルの形状に注意して式を変形する必要がある。

それでは材料がそろったので、誤差逆伝播法による線形回帰をプログラミングしよう。少し長いので前半・後半に分けて説明する。

リスト 3.9.3：誤差逆伝播法(前半)

```
import numpy as np
import matplotlib.pyplot as plt
from ufiesia import LCBF
import data_flu

m = 9 # 基底関数の次数
s = 2.0 # ガウス基底の広がり係数
normalize = False
r = 0.0 # 正則化係数

data = np.array(data_flu.data)
X = data[:, 0] # 入力
T = data[:, 1] # 正解値

# -- 入力データの正規化+ --
if normalize:
    xmax = np.max(X); xmin = np.min(X)
    X = (X - xmin) / (xmax - xmin) * 2 - 1
    s = s / (xmax - xmin) * 2 # 広がり幅もデータに合わせてスケールリング

# -- 対象となる関数と損失(正則化項つき) --
model = LCBF.LinearCombination(LCBF.GaussianBasis(m, s))
loss_func = LCBF.MeanSquaredErrorDecay()

# -- パラメタの初期値は乱数 --
model.w = np.random.randn(m)
print('パラメタの初期値', model.w)
error = loss_func.forward(model(X), T, r, model.w)
print('初期の損失', error)
```

まずは前半部分だが、データを用意するまでは、これまでの例と同じ。回帰の対象となる関数だが、線形結合の関数と平均二乗誤差をインスタンス化するまでは「リスト 3.8.2」と同じだ。パラメタに初期値を与え損失を評価する。

リスト 3.9.4 : 誤差逆伝播法(後半)

```
# -- 勾配降下法によるパラメタのチューニング --
lr = 0.1 # 学習係数
for i in range(10000):
    loss_func.forward(model(X), T, r, model.w)
    gy, grad_w = loss_func.backward()
    grad_w += model.backward(gy)
    model.w -= lr * grad_w

print('完了時のパラメタ', model.w)
error = loss_func.forward(model(X), T, r, model.w)
print('完了時の損失', error)

# -- グラフ描画用 --
plt_x = np.linspace(min(X), max(X), 1000)
plt_y = model(plt_x)

# -- 逆変換で x 軸をもとに戻す --
if normalize:
    X = (X + 1) / 2 * (xmax - xmin) + xmin
    plt_x = (plt_x + 1) / 2 * (xmax - xmin) + xmin

# -- グラフ表示 --
plt.grid()
plt.scatter(X.tolist(), T.tolist(), marker='x', s=100) # 入力
plt.scatter(plt_x.tolist(), plt_y.tolist(), marker='.', s=1) # 近似
plt.xlabel('input')
plt.ylabel('output')
plt.show()
```

続いて後半だ。学習係数を指定し for ループで回して学習を繰り返す。

```
loss_func.forward(model(X), T, r, model.w)
```

で線形結合と損失関数をつなげているのは、リスト 3.8.2 と同じだ。いっぽう逆伝播はこれとは違って、損失関数の逆伝播、線形結合の逆伝播を分けている。そして逆伝播では重みの勾配 `grad_w` に少し細工が必要である。重みの勾配 `grad_w` は、損失関数と、線形結合の両方を考慮する必要があり、いったん損失関数の `backward()` メソッドから返された `grad_w` に、後から得られる線形結合の `backward()` メソッドの戻り値を足し合わせるようにする。そしてその足しあわされた値で、線形結合の重み `w` を更新する。

そして学習が完了したら再び損失を評価する。あとはリスト 3.6.6 やリスト 3.8.2 と同じだ。

では実行結果を示そう。

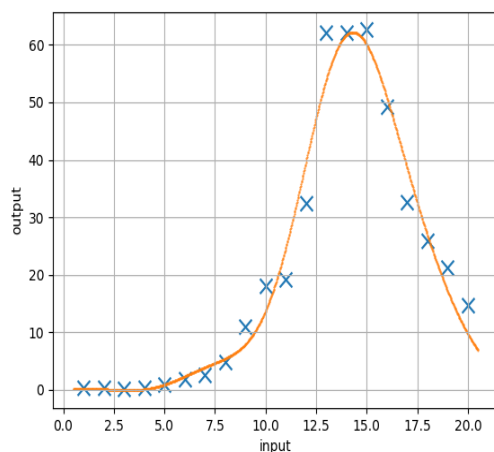


図 3.9.1 インフルエンザ～誤差逆伝播法

図は、ガウス基底で $m=9$ 、 $s=2$ 、 $\text{normalize}='False'$ 、 $r=0.0$ とした場合のものである。

「リスト 3.8.3」の数値微分法で実行したのと同じ結果になった。誤差逆伝播法によって勾配を求めても、数値微分法によって勾配を求めるのと同じになることが確かめられた。

明らかに過学習になっている例を多々示してきた。その示した結果それ自体は意味はない。しかし、やすやすと過学習することは、大きな表現力の裏返しである。またいっぽうで過学習を抑えることも、それを引き起こせるからこそ、できるのではないだろうか？

ここまで同じ問題を解くのに、いろいろなやり方を示してきた。ここで取り上げたような単純なケースでは、やり方をいろいろ知っている必要はないだろう。しかしディープラーニングのように、簡単には良い答えに行きつけない、もしくはそもそも何が良い答えなのかすらわからないような場合に、答えを導き出す「すべ」を持つと同時に、その意味を理解していることは、さまざまな問題に取り組むうえでなくてはならず、一通りではないアプローチは理解の助けになると確信している。

3.10 フーリエ級数

フーリエ級数は、複雑な変化をサイン波、コサイン波の重ね合わされたものとしてとらえる。式であらわせば、

$$f(x) = \frac{a^0}{2} + \sum_n (a_n \cos nx + b_n \sin nx) \quad (3.10.1)$$

このフーリエ係数 a_n や b_n は、 $f(x)$ が実数値関数で周期 2π の周期関数ならば、次式によって求められる。

$$\begin{aligned} a_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos nx \, dx, \quad (n = 0, 1, 2, \dots) \\ b_n &= \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin nx \, dx, \quad (n = 0, 1, 2, \dots) \end{aligned} \quad (3.10.2)$$

ここで式 3.10.2 によれば、フーリエ係数 a_n や b_n を求めることが出来る。しかしこれは結構大変なのではないだろうか？ それというのも、式 3.10.2 は $f(x)$ が数式で与えられても計算が大変だし、そもそもそれがわからないほうが普通で、数値積分で求めるということらしいが、それが測定値ならば誤差が含まれるのが普通だから、それをどうするのかも考えなくてはならない。そこでこれを追求することは、ここではちょっと置いておいて、フーリエ級数でデータを作ってみることにしよう。そしてちょっとその前に、式 3.10.1 に時間軸を調整する係数 T をいれて、周期 2π というのを調整するようにしておけば、

$$f(x) = \frac{a^0}{2} + \sum_n \left(a_n \cos \frac{nx}{T} + b_n \sin \frac{nx}{T} \right) \quad (3.10.3)$$

式 3.10.3 によってフーリエ級数をあらわすなら、式 3.10.2 も修正が必要だが、ここではどのみち使わないから放っておくことにする。

それですべてやっておきたいのが、フーリエ係数 a_n や b_n を当てずっぽうで与えてフーリエ級数を作ることだ。

リスト 3.10.1：フーリエ級数

```
import numpy as np

def fourier_series(N=5, T=1, min_ab=-5, max_ab=5, start=-np.pi, stop=np.pi, num=100):
    a = np.random.randint(min_ab, max_ab, N)
    b = np.random.randint(min_ab, max_ab, N)
    x = np.linspace(start, stop, num)
    y = a[0]
    print('a0 =', a[0], end=' ')
    for n in range(1, N, 1):
        y += (a[n]*np.cos(n*x/T) + b[n]*np.sin(n*x/T))
        print('a' + str(n), '=', a[n], 'b' + str(n), '=', b[n], end=' ')
    print()
    return np.vstack((x, y)).T

data = fourier_series()
```

リスト 3.10.1 は関数 `fourier_series()` を定義して、それを使って `data` を用意する。関数 `fourier_series()` はやたらと引数が多いが、順に見ていく。

`N`：フーリエ級数の次数

`T`：周期の調整

`min_ab, max_ab`：フーリエ係数の値の最小値と最大値

`start, stop, num`：作成する `data` の `x` の範囲と数

これらはいずれもデフォルト値を与えておいた。

そして、この関数の中だが、`numpy` の関数 `random.randint()` で引数で指定した範囲のランダムな整数を作ってフーリエ係数 a と b にする。このときフーリエ級数の次元数 N に応じた数を作る。もともと式 3.10.1 や式 3.10.3 で Σ の上限は ∞ だが、ここでは $N-1$ を上限にする。

だから関数 `fourier_series()` が作るのは、フーリエ級数もどきだ。ともかくもそれで必要なフーリエ係数の数も $N-1$ 個になり、式 3.10.1 や式 3.10.3 の $a^0/2$ の項を含めて N 個だ。

フーリエ係数が出来たら、次は `x` を `np.linspace()` で用意する。そしてはじめの `y` は `a[0]` として、あとは式 3.10.3 の Σ のところを `for` ループでそのまま行う。このときフーリエ係数を `print()` 文で出力しておく。返り値は、リスト 3.4.2 のアヤメやリスト 3.4.4 のインフルエンザの流行と同様に、別のプログラムから使うことを想定して、同じように `x` と `y` の組が並んだ形にする。最後に `data` にこの関数の結果を入れたら完了だ。このプログラムは `data_mfc.py` として保存しておこう。

これが出来たら続いて、結果を表示するプログラムで確かめておこう。

リスト 3.10.2：フーリエ級数で曲線を描く

```
import data_mfc
import matplotlib.pyplot as plt

data = data_mfc.data
X = data[:,0]
Y = data[:,1]
plt.plot(X,Y)
plt.show()
```

リスト 3.10.1 で作った data_mfc をインポートしてデータを取得し表示するだけだから、特に説明はいらないだろう。ともかくこの実行結果の一例を示す。

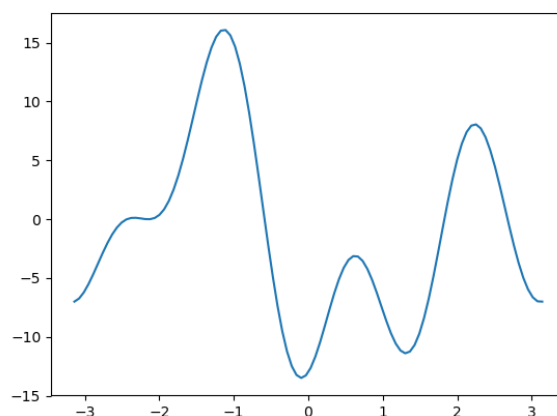


図 3.10.1 フーリエ級数で曲線を描く

このとき出力結果を見ると、

$a_0 = -1$ $a_1 = -2$ $b_1 = -5$ $a_2 = -5$ $b_2 = -5$ $a_3 = -1$ $b_3 = 3$ $a_4 = -4$ $b_4 = 4$

となっていたのだが、まあともかくもよくわからない曲線には違いない。何度もやってみると、さまざまな曲線が描かれて、どんな曲線も描けるような気がしなくもない。

data = data_mfc.data の1行を、data = data_mfc.fourier_series(N=100) と変えて、関数を直接呼出して、次元数を N=100 のように大きな値に指定すれば、さらにいろんな曲線が描ける。

リスト 3.10.1 とリスト 3.10.2 で描いた曲線は、もともと意図的に作ったものだ。しかし本来はなんらかの測定の結果得られたデータを解析するので、ここでは素性をしらないものとして、リスト 3.10.1 で作った曲線を扱っていこう。

さて、式 3.10.1 をもういちど良く見ると、変数 x の関数の線形結合となっている。そしてフーリエ係数 a_n や b_n を重みと考えれば、これまでやってきた線形回帰の問題そのものとみることができる。少しだけ工夫がいるのは、同じ n に対してそれぞれ別の係数 a_n や b_n を乗じたサインとコサインの 2 つの関数の和だから、基底関数にそれを踏まえてサイン、コサインが交互に現れるようにすることや、基底関数の次元と重みの数の辻褄を合うようにする必要があることだ。しかしいずれも簡単だ。それではさっそく、フーリエ級数の基底関数を定義しよう。

リスト 3.10.3：フーリエ級数の基底関数

```
class FourierBasis:
    def __init__(self, m=2, T=1):
        self.m = m
        self.func = lambda n, x: [np.cos(n*x/T), np.sin(n*x/T)]
        print('Initialize FourierBasis')

    def __call__(self, x):
        m = self.m
        y = np.array([self.func(n, x) for n in range(1, m//2+1, 1)])
        y = y.reshape(-1, len(x)).T      # 次元を1つ減らして転置
        ones = np.ones((len(x), 1))     # 定数項
        y = np.hstack((ones, y))        # 左端は定数項
        y = y[:, :m]
        return y
```

式 3.10.3 に従って、コントラクタでサインの項とコサインの項を並べたリストを `self.func` として作っておく。このとき[]を抜かすとうまくいかない。

そして `__call__()` では、これを呼び出して、

`[self.func(n, x) for n in range(1, m//2+1, 1)]` で

内包表記により並べる。このとき n の値の範囲は $n=1$ から始めて $n=m//2$ を含むようにする。そうすると基底関数はコサインとサインで 2 つずつ並ぶから、 m が偶数の場合は同数、 m が奇数の場合は $m-1$ 個の基底関数が並ぶ。これにあとで定数 1 の項を加えるから、基底関数の並びは、 m が偶数の場合は $m+1$ 個、 m が奇数の場合は m 個となるので、最後に `y = y[:, :m]` で不要な末尾を切り捨てる。少し先走ったので戻るが、内包表記で基底関数を並べた時点で y の形状は、 $(m//2, 2, \text{データ数})$ となっていて、データ x の並びが末尾の軸、サインとコサインの 2 つが中間の軸、先頭の軸は $m//2$ に対応する。そこでデータの並びを残して、初めの 2 つの軸を 1 つにした上で転置する。これで行方向にデータが、列方向に基底関数が並んだ行列になる。式 3.10.3 の Σ の外の $a^0/2$ に対応する定数項は別途用意して、numpy の `hstack()` で先頭の列に結合する。後は `y[:, m]` で基底関数の次数をはみ出す列を切り捨てて返り値にする。こうして返り値は式 3.1.22 で一般的に示す Φ の形になる。

では続いてフーリエ級数の基底関数を使った線形回帰をやってみよう。リスト 3.6.2 を少し書き換えればすぐに作れる。

リスト 3.10.4：フーリエ級数で線形回帰

```
import numpy as np
import matplotlib.pyplot as plt
from ufiesia import LCBF
import data_mfc as data

m = 21          # 基底関数の次数
period = 1.0    # 周期の調整

data = np.array(data.data)
X = data[:, 0]  # 入力
T = data[:, 1]  # 正解値

# -- 対象を定義し回帰 --
model = LCBF.LinearCombination(LCBF.FourierBasis(m, period))
model.regression(X, T)

# -- グラフ描画用 --
plt_x = np.linspace(min(X), max(X), 1000)
plt_y = model(plt_x)

# -- グラフ表示 --
plt.grid()
plt.scatter(X, T, marker='x', s=100)
plt.scatter(plt_x, plt_y, marker='.', s=1)
plt.xlabel('input')
plt.ylabel('output')
plt.show()

# -- 成分の表示 --
plt.stem(model.w)
plt.show()
```

冒頭からグラフ表示までは、リスト 3.6.2 と同じ流れだから、違う点を説明する。データは先に作ったリスト 10.3.1 の data_mfc だ。フーリエ級数で作ったデータをフーリエ級数で解析してどうなるというなかれ。data_mfc でデータを作る際のフーリエ係数を全く知らないプログラムが、それを解き明かすのだ。そしてもちろんリスト 3.4.4 など、ほかのデータも解析対象にできる。ともかくもここでは、答え合わせのできる data_mfc を使う。回帰モデルはリスト 3.10.4 で作った基底関数に FourierBasis() を指定する。周期の調整は period=1.0 で指定しなくても同じだが、ほかのデータでは調整が必要になる場合がある。あとはリスト 3.6.2 とまったく同じだが、せっかくだから最後に、matplotlib の stem plot で求めた重み w を表示することを追加した。重み w は式 3.10.3 のフーリエ係数 a_n や b_n を並べたものに他ならな

いから、周波数成分の大きさを可視化したものになる。つまり周波数成分がグラフ表示されるわけだ。そしてこれが data_mfc で曲線を作る際のフーリエ係数と一致すれば、プログラムは正しく曲線を解析したことになる。

ではこの実行の一例を示そう。

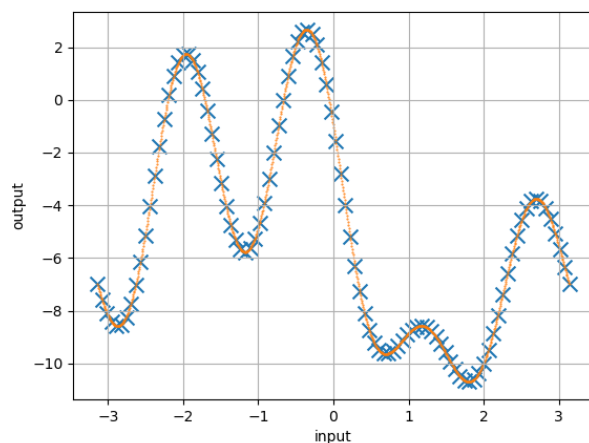


図 3.10.2 フーリエ級数で線形回帰～その1

図 3.10.2 は与えたデータと推定結果だ。みごとに×のデータを曲線がなぞっている。続く図 3.10.3 では解析結果のフーリエ係数をグラフ表示している。

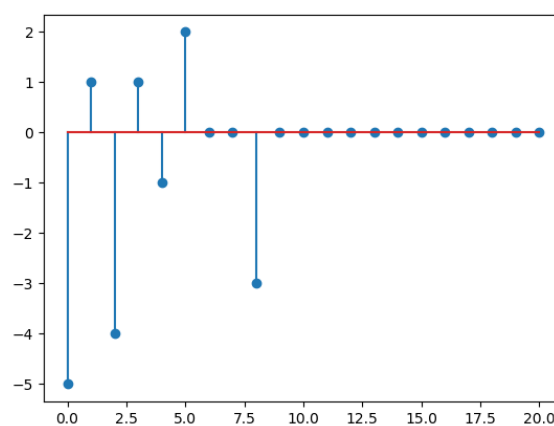


図 3.10.3 フーリエ級数で線形回帰～その2

data_mfc でデータを生成した際には、フーリエ係数=正解値が出力されている。これは次のようになっていた。

$a_0 = -5$ $a_1 = 1$ $b_1 = -4$ $a_2 = 1$ $b_2 = -1$ $a_3 = 2$ $b_3 = 0$ $a_4 = 0$ $b_4 = -3$

どうだろうか？グラフとびたり一致しているのではないだろうか？

これで狙い通りに、フーリエ係数が線形回帰のやり方で求めることができた。データとして与えた曲線をフーリエ級数であらわすことが可能となり、その周波数成分の大きさがわかるのだ。ここではもともとフーリエ級数で作った曲線を対象にしたが、もちろんそうである必要はない。たとえばリスト 3.4.4 のインフルエンザの流行のデータでもフーリエ係数を求めることができる。そしてデータとして与えられるどんな形の曲線も数式であらわせるならば、数学的な演算対象として処理を施すこともできることになる。

膨大で複雑なデータの中から何らかの真理を導き出す、そういう大変なテーマに取り組むときに、ここで学んだことが役に立つと期待している。