Machine Learning: Theory and Implementation using Python

Keishi Okudera

2018 年 7 月 14 日

目次

第 1 章

線形代数

数学的に厳密ではない箇所があるが，ご容赦いただきたい．後で追記する予定．

1.1 行列の定義

まずは行列を定義する．

定義 1.1 (行列)**.** 実数の長方形配列を**行列 (matrix)** と呼ぶ．すなわち，行列は一般的な形として以下のように配列された実数 $a_{11}, a_{12}, \ldots, a_{1n}, \ldots, a_{m1}, a_{m2}, \ldots, a_{mn} \in \mathbb{R}$ の集まりである．

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

なお，上記のように m 個の行と n 個の列からなる行列を $m \times n$ 行列と呼び，m と n をこの行列の**次元 (dimension)** と呼ぶ．行列の第 i 行と第 j 列にある a_{ij} をその行列の第 ij **要素 (element)** または**成分 (entry)** と呼ぶ．たびたび，行列はその成分 a_{ij} として $A = \{a_{ij}\}$ と表記する．なお，行列に対して，単なる数のことを，**スカラー (scalar)** と呼ぶ．

注意 1.1. 断りのない限り，行列の成分は実数に限定する．

1.2 基礎演算

定義した行列に対して，相等，和・差，積 (スカラー積と行列の積)，そして転置という行列特有の操作を導入する．

定義 1.2 (行列の相等)**.** 同じ次元を持つ 2 つの行列 A,B について，もし A の全ての要素が B のその対応する各要素に等しいならば，行列は**等しい (equal)** といい，$A = B$ と書く．

定義 1.3 (スカラー乗法)**.** 任意のスカラー k と任意の $m \times n$ 行列 $A = \{a_{ij}\}$ に対して，k と A の**スカラー積 (scalar product)** を，その第 ij 要素が ka_{ij} である $m \times n$ 行列と定義し，kA と書く．特に，-1 と A のスカラー積 $(-1)A$ を，A の**負 (negative)** と呼び，$-A$ と略記する．

定理 1.1. 任意のスカラー c, k，任意の行列 A に対して次式が成り立つ．

$$c(kA) = (ck)A = (kc)A = k(cA) \tag{1.1}$$

証明. 省略 (任意の ij 成分に対して成り立つことを示せばよい)． □

定義 1.4 (行列の加法と減法)**.** 同じ行の数 m と同じ列の数 n をもつ任意の行列 $A = \{a_{ij}\}, B = \{b_{ij}\}$ に対して，A と B の**和 (sum)** を，その第 ij 要素が $a_{ij} + b_{ij}$ である $m \times n$ 行列と定義し，$A + B$ と書く．また，和 $A + (-B)$，すなわちその第 ij 要素が $a_{ij} - b_{ij}$ である $m \times n$ 行列を $A - B$ と書くこととし，この行列を A と B の**差 (difference)** と呼ぶ．同じ数の行と列をもつ行列は加法あるいは減法に対して**共形的 (conformal)** であるという．

定理 1.2. 任意のスカラー c, k，任意の共形的である行列 A, B, C に対して次式が成り立つ．

$$A + B = B + A \tag{1.2}$$
$$A + (B + C) = (A + B) + C \tag{1.3}$$
$$c(A + B) = cA + cB \tag{1.4}$$
$$(c + k)A = cA + kA \tag{1.5}$$

証明. 略 (任意の ij 成分に対して成り立つことを示せばよい)． □

定義 1.5 (行列の積)**.** $m \times n$ 行列 A と $p \times q$ 行列 B について，$p = n$，すなわち A の列と B の行の数が等しいとき，行列の**積 (matrix product)** AB は，$AB = \{c_{ij}\}$ としたとき，その第 ij 要素 c_{ij} が次式で表される $m \times q$ 行列と定義する．

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{1.6}$$

なお，AB は，A による B への**前からの乗法 (premultiplication)** または B の A による**後ろからの乗法 (postmultiplication)** と呼ぶ．

定理 1.3. 任意のスカラー c，任意の $m \times n$ 行列 A，$n \times q$ 行列 B，$q \times t$ 行列 C に対して次式が成り立つ．

$$A(BC) = (AB)C \tag{1.7}$$
$$A(B + C) = AB + AC \tag{1.8}$$
$$(A + B)C = AC + BC \tag{1.9}$$
$$cAB = (cA)B = A(cB) \tag{1.10}$$

証明. 略 (任意の ij 成分に対して成り立つことを示せばよい)． □

問題 1.1. 任意の $m \times n$ 行列 A, B，$n \times q$ 行列 C, D に対して次式が成り立つことを示せ．

$$(A + B)(C + D) = AC + AD + BC + BD \tag{1.11}$$

解答. 式 (1.8) において $(A + B)$ を A だとして適用し，その後に式 (1.9) を順に適用して展開したあと，式 (1.2) で交換操作をすることで示せる．

$$\begin{aligned}(A + B)(C + D) &= (A + B)C + (A + B)D \\ &= AC + BC + AD + BD \\ &= AC + AD + BC + BD\end{aligned}$$

□

定義 1.6 (転置)**.** $m \times n$ 行列 A の**転置 (transposition)** は記号 A^T で表し，各 ij 要素が A の第 ji 要素である $n \times m$ 行列を意味する．すなわち，行列の転置は行を列として，列を行として書き直すことで作る．

定理 1.4. 任意の行列 A に対して，次式が成り立つ．

$$(A^T)^T = A \tag{1.12}$$

また，加法に関して共形的である任意の行列 A, B に対して，次式が成り立つ．

$$(A+B)^T = A^T + B^T \tag{1.13}$$

また，積が定義される任意の行列 A, B に対して，次式が成り立つ．

$$(AB)^T = B^T A^T \tag{1.14}$$

証明. 任意の ij 成分に対して成り立つことを示せばよいので省略するが，式 (1.14) のみ示す．$m \times n$ 行列 $A = \{a_{ij}\}$，$n \times q$ 行列 $B = \{b_{ij}\}$ とする．$q \times m$ 行列 $(AB)^T = \{d_{ij}\}$ の第 ij 成分は，$m \times q$ 行列 $AB = \{c_{ij}\}$ の第 ji 成分であるため，式 (1.6) より

$$d_{ij} = c_{ji} = \sum_{k=1}^{n} a_{jk} b_{ki}$$

となる．一方，$q \times n$ 行列 $B^T = \{b'_{ij}\}$，$n \times m$ 行列 $A^T = \{a'_{ij}\}$ としたとき，$q \times m$ 行列 $B^T A^T = \{d'_{ij}\}$ の第 ij 成分は，式 (1.6) より

$$d'_{ij} = \sum_{k=1}^{n} b'_{ik} a'_{kj}$$

となるが，$b'_{ik} = b_{ki}, a'_{kj} = a_{jk}$ であるため，結局，

$$d'_{ij} = \sum_{k=1}^{n} b'_{ik} a'_{kj} = \sum_{k=1}^{n} b_{ki} a_{jk} = \sum_{k=1}^{n} a_{jk} b_{ki} = d_{ij}$$

□

1.3　正方行列，ベクトル

行列の中でも，特別な名前がつけられているものがある．ここでは正方行列と対称行列，ベクトルを定義する．

定義 1.7 (正方行列)**.** 行の数と列の数が同じ行列を**正方行列 (square matrix)** と呼び，$n \times n$ 正方行列における n を**次数 (order)** という．また，$n \times n$ 正方行列において $a_{11}, a_{22}, \ldots, a_{nn}$ を（第 1，第 2，等の）**対角要素 (diagonal element)** と呼び，対角要素以外の要素を**非対角要素 (off-diagonal element)** と呼ぶ．

定義 1.8 (対称行列)**.** $A^T = A$ である行列 A，すなわち第 ij 要素が第 ji 要素と等しい行列を**対称行列 (symmetric matrix)** という．

定理 1.5. 任意の行列 X に対して，$X^T X$ は対称行列である．

証明. 式 (1.14)，式 (1.12) より，$(X^TX)^T = X^T(X^T)^T = X^TX$. □

定義 1.9 (ベクトル)**.** ただ 1 個の列をもつ行列，すなわち以下の行列を**列ベクトル (column vector)** と呼ぶ.

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}$$

同様に，ただ 1 個の行をもつ行列を**行ベクトル (row vector)** と呼ぶ．なお，上記のように m 個の要素からなるベクトルを m 次元ベクトルと呼ぶ．たびたび，ベクトルはその第 i 成分 a_i として $\boldsymbol{a} = \{a_i\}$ と表記する．また，上記の列ベクトルを文章中で書く場合，行ベクトルと転置を用いて $\boldsymbol{a} = (a_1, a_2, \ldots, a_m)^T$ とすることが多い.

1.4 内積，ノルム，角度

ここでは，内積，(通常の) ノルム，角度を定義する.

定義 1.10 (内積)**.** 2 個の n 次元列ベクトル $\boldsymbol{a} = (a_1, a_2, \ldots, a_n)^T$，$\boldsymbol{b} = (b_1, b_2, \ldots, b_n)^T$ に対して，**内積 (inner product)** $\boldsymbol{a} \cdot \boldsymbol{b}$ は次式で定義される.

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1b_1 + a_2b_2 + \cdots + a_nb_n = \boldsymbol{a}^T\boldsymbol{b} \tag{1.15}$$

注意 1.2. 本書では $\boldsymbol{a} \cdot \boldsymbol{b}$ という表記はあまり使用せず，$\boldsymbol{a}^T\boldsymbol{b}$ を使用する.

定理 1.6. 内積について次式が成り立つ.

$$\boldsymbol{a}^T\boldsymbol{b} = \boldsymbol{b}^T\boldsymbol{a} \tag{1.16}$$
$$\boldsymbol{a}^T\boldsymbol{a} \geq 0 \quad (\text{等号成立は } \boldsymbol{a} = \boldsymbol{0} \text{ のとき}) \tag{1.17}$$
$$(k\boldsymbol{a})^T\boldsymbol{b} = k\boldsymbol{a}^T\boldsymbol{b} \tag{1.18}$$
$$(\boldsymbol{a} + \boldsymbol{b})^T(\boldsymbol{c} + \boldsymbol{d}) = \boldsymbol{a}^T\boldsymbol{c} + \boldsymbol{b}^T\boldsymbol{c} + \boldsymbol{a}^T\boldsymbol{d} + \boldsymbol{b}^T\boldsymbol{d} \tag{1.19}$$

証明. 略. □

定義 1.11 ((通常の) ノルム)**.** n 次元列ベクトル $\boldsymbol{a} = (a_1, a_2, \ldots, a_n)$ に対して，**(通常の) ノルム ((usual) norm)** $||\boldsymbol{a}||$ は次式で定義される.

$$||\boldsymbol{a}|| = \sqrt{\boldsymbol{a}^T\boldsymbol{a}} = \sqrt{a_1^2 + a_2^2 + \cdots + a_n^2} \tag{1.20}$$

定理 1.7. 通常のノルムについて次式が成り立つ.

$$||\boldsymbol{a}|| \geq 0 \quad (\text{等号成立は } \boldsymbol{a} = \boldsymbol{0} \text{ のとき}) \tag{1.21}$$
$$||k\boldsymbol{a}|| = |k|||\boldsymbol{a}|| \tag{1.22}$$

証明. 略. □

定義 1.12 (角度)**.** 2 つの $\boldsymbol{0}$ でない n 次元列ベクトル $\boldsymbol{x}, \boldsymbol{y}$ に対して，そのなす角度 θ $(0 \leq \theta \leq \pi)$ を，その余弦を用いて次式で定義する.

$$\cos\theta = \frac{\boldsymbol{x}^T\boldsymbol{y}}{||\boldsymbol{x}||||\boldsymbol{y}||} \tag{1.23}$$

1.5　分割ベクトル

行列はベクトルを横だったり縦だったりにくっつけたものともいえる．行列においてその要素をベクトルに分割したものを分割ベクトルという．

定義 1.13 (分割ベクトル)**.** 任意の $m \times n$ 行列 $A = \{a_{ij}\}$ を行方向に分割した場合，すなわち m 個の列ベクトル $\boldsymbol{a}_i = (a_{i1}, a_{i2}, \ldots, a_{in})^T$ $(i = 1, 2, \ldots, m)$ で分割したとき，その列ベクトル $\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_m$ を**分割行ベクトル (partitioned row vector)** という．また，列方向に分割した場合，すなわち n 個の列ベクトル $\boldsymbol{a}'_j = (a_{1j}, a_{2j}, \ldots, a_{mj})^T$ $(j = 1, 2, \ldots, n)$ で分割した時，その列ベクトル $\boldsymbol{a}'_1, \boldsymbol{a}'_2, \ldots, \boldsymbol{a}'_n$ を**分割列ベクトル (partitioned column vector)** という．これらを用いて，A は次式で表記される．

$$A = \begin{bmatrix} \text{---} & \boldsymbol{a}_1^T & \text{---} \\ \text{---} & \boldsymbol{a}_2^T & \text{---} \\ & \vdots & \\ \text{---} & \boldsymbol{a}_m^T & \text{---} \end{bmatrix} = \begin{bmatrix} | & | & \cdots & | \\ | & | & \cdots & | \\ \boldsymbol{a}'_1 & \boldsymbol{a}'_2 & \cdots & \boldsymbol{a}'_n \\ | & | & \cdots & | \\ | & | & \cdots & | \end{bmatrix} \tag{1.24}$$

定理 1.8. $m \times n$ 行列 $A = \{a_{ij}\}$ と n 次元ベクトル $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ との積 $A\boldsymbol{x}$ は，分割行ベクトル $\boldsymbol{a} = (a_{i1}, a_{i2}, \ldots, a_{in})^T$ $(i = 1, 2, \ldots, m)$ または分割列ベクトル $\boldsymbol{a}'_j = (a_{1j}, a_{2j}, \ldots, a_{mj})^T$ $(j = 1, 2, \ldots, n)$ を用いてそれぞれ以下で表すことができる．

$$A\boldsymbol{x} = \begin{bmatrix} \text{---} & \boldsymbol{a}_1^T & \text{---} \\ \text{---} & \boldsymbol{a}_2^T & \text{---} \\ & \vdots & \\ \text{---} & \boldsymbol{a}_m^T & \text{---} \end{bmatrix} \begin{bmatrix} | \\ | \\ \boldsymbol{x} \\ | \\ | \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}_1^T \boldsymbol{x} \\ \boldsymbol{a}_2^T \boldsymbol{x} \\ \vdots \\ \boldsymbol{a}_m^T \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}^T \boldsymbol{a}_1 \\ \boldsymbol{x}^T \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{x}^T \boldsymbol{a}_m \end{bmatrix} \tag{1.25}$$

$$= \begin{bmatrix} | & | & \cdots & | \\ | & | & \cdots & | \\ \boldsymbol{a}'_1 & \boldsymbol{a}'_2 & \cdots & \boldsymbol{a}'_n \\ | & | & \cdots & | \\ | & | & \cdots & | \end{bmatrix} \begin{bmatrix} | \\ | \\ \boldsymbol{x} \\ | \\ | \end{bmatrix} = \boldsymbol{a}'_1 x_1 + \boldsymbol{a}'_2 x_2 + \cdots + \boldsymbol{a}'_n x_n \tag{1.26}$$

証明. $A\boldsymbol{x}$ を書き下すことで容易に示すことができる．式 (1.25) は内積の定義，性質よりすぐ分かる．式 (1.26) は和を分解していく操作をすると示される．

$$A\boldsymbol{x} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}_1^T \boldsymbol{x} \\ \boldsymbol{a}_2^T \boldsymbol{x} \\ \vdots \\ \boldsymbol{a}_m^T \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}^T \boldsymbol{a}_1 \\ \boldsymbol{x}^T \boldsymbol{a}_2 \\ \vdots \\ \boldsymbol{x}^T \boldsymbol{a}_m \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} x_1 + \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} x_2 + \cdots + \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} x_n = \boldsymbol{a}'_1 x_1 + \boldsymbol{a}'_2 x_2 + \cdots + \boldsymbol{a}'_n x_n$$

□

1.6　線形形式，二次形式

行列やベクトルを演算した結果としてスカラー値を返す関数はいくらでもあるが，その中に重要な関数がいくつかある．代表的なものとして線形形式，二次形式がある．

定義 1.14 (線形形式)**.** $\boldsymbol{a} = (a_1, a_2, \ldots, a_n)^T$ を任意の n 次元列ベクトルとする．このとき，任意の n 次元列ベクトル $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$ に対してスカラー値 $\boldsymbol{a}^T\boldsymbol{x}$ を返す関数を，$\boldsymbol{x}$ に関する**線形形式 (linear form)** と呼ぶ．ここで，習慣的に $\boldsymbol{a}$ を**係数ベクトル (coefficient vector)** と呼ぶ．

定義 1.15 (二次形式)**.** $A = \{a_{ij}\}$ を任意の $n \times n$ 行列とする．このとき，任意の n 次元列ベクトル $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)^T$ に対してスカラー値 $\boldsymbol{x}^T A\boldsymbol{x}$ を返す関数を，$\boldsymbol{x}$ に関する**二次形式 (quadratic form)** と呼ぶ．ここで，習慣的に A を**二次形式の行列 (matrix of quadratic form)** と呼ぶ．$\boldsymbol{x}^T A\boldsymbol{x}$ は次式で展開される．2 つ目の式は，a_{ii} を起点にそれ自身の項，行方向に見た項，列方向に見た項，それ以外の項に分けた形で，x_i での微分をする場合等に役立つ．

$$\boldsymbol{x}^T A\boldsymbol{x} = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_i x_j \tag{1.27}$$

$$= a_{ii}x_i^2 + \sum_{\substack{j=1 \\ j\neq i}}^{n} a_{ij}x_i x_j + \sum_{\substack{k=1 \\ k\neq i}}^{n} a_{ki}x_k x_i + \sum_{\substack{j,k \\ j\neq i \\ k\neq i}} a_{kj}x_k x_j \tag{1.28}$$

1.7 行列の特殊な積

式 (1.6) とは別に，要素同士の積をとった演算も存在する．それをアダマール積という．

定義 1.16 (アダマール積)**.** 2 つの $m \times n$ 行列 $A = \{a_{ij}\}, B = \{b_{ij}\}$ について，**アダマール積 (hadamard product)** $A * B$ は，$A * B = \{c_{ij}\}$ としたとき，その第 ij 要素が次式で表される $m \times n$ 行列と定義する．

$$c_{ij} = a_{ij}b_{ij} \tag{1.29}$$

第 2 章

Python

2.1 標準モジュール

2.1.1 print 関数

Python にはあらかじめ入っている関数がいくつかある．最も基本的な関数が print 関数である．

Grammer 2.1.

- print(<object>): <object>の表示結果を返す．print 関数は少々特殊で，表示結果を変数に代入するということはできない (代入文を書いても表示結果が出るだけで，その変数には None が格納される)．<str>の場合，<str>の中に\n を書くと，そこで改行されて表示される (<str>自身から\n がなくなるわけではないので注意)．

Code 2.1 (py1.py)**.**

```
print('~~~Hello␣world!~~~')
print('---Hello\nworld!---')
```

```
~~~Hello world!~~~
---Hello
world!---
```

Code 2.2 (py2.py)**.**

```
x = 10
print('---')
p = print(x)
print('---')
print(p)
```

```
---
10
---
None
```

2.1.2 type 関数

Python は多種多様な型を持つため，最初のうちは型をいろいろ確認しながらやるのが良い．

Grammer 2.2.

- type(<object>): <object>の型を返す．type(<object>) の返り値自体も type 型を持つ．

Code 2.3 (py3.py).

```
x = 'Hello world!'
type_x = type(x)
print(type_x)
print(type(type_x))
```

```
<class 'str'>
<class 'type'>
```

2.1.3 format メソッド

特に print 文を使うときに便利な format メソッドを紹介する．これは<str>に作用し，文字列中に任意の変数等の中身を文字列として挿入できるものである．

Grammer 2.3.

- <str>.format(<object>,<object>,...): <str>の中に記入した {} に，<object>の中身を文字列として代入する．なお，{} は何個あってもよく，{} には<object>,<object>,...の順にそれぞれ代入されていく．
- <str>の中に記入した {} を {:f} に書き換える: 小数表記 (ただし小数点以下第 6 位まで) で表示する．デフォルトの場合，数値の小ささや大きさ等によって，自動的に指数表記への変換が行われるが，これをやめて欲しい場合，このように指定することで少数で表示できる．

Code 2.4 (py4.py).

```
p = {'x':3, 'y':5}
string = 'p={}, type={}'.format(p, type(p))
print(string)

alpha= 0.000001
print('alpha={}'.format(alpha))
print('alpha={:f}'.format(alpha))
```

```
p={'x': 3, 'y': 5}, type=<class 'dict'>
alpha=1e-06
alpha=0.000001
```

2.1.4 関数の作成

Python は型の定義や渡し方等を意識して記述することなく，非常にシンプルな書き方で関数を作成できる．また，返り値は複数でもよく，出力先の変数が 1 つならタプル，複数ならそれぞれの要素をそれぞれの変数に格納する．通常，関数を作る際は引数の型として何をとるかなどを考えながら組むことが多いが，実際のコードは型を指定しないので，関数を作るときに想定していなかった型の引数を渡した場合にどうなるかということを考えてみると，型を指定しない関数の便利さが分かってくる．

Grammer 2.4 (関数の定義).

```
def <func_name>(<object>,<object>,....):
       <expr>
       <expr>
       ...
       return <object>, <object>,....
```

このコード例は，なんとなく数値の引数を思い浮かべながら組んだ関数の例であるが，この関数は数値だけではなく<ndarray>でも動作する (numpy については後述).

Code 2.5 (py5.py).

```
import numpy as np

def four_arithmetic(a, b):
    c = a + b
    d = a - b
    e = a * b
    f = a / b
    return c, d, e, f

add, sub, mul, div = four_arithmetic(10, -1)
tpl = four_arithmetic(5, -5)

print('add={},sub={},mul={},div={}'.format(add, sub, mul, div))
print('tpl={}\n'.format(tpl))

lst1 = np.array([1, 2, 3])
lst2 = np.array([-1, -2, -3])

add, sub, mul, div = four_arithmetic(lst1, lst2)
print('add={},sub={},mul={},div={}'.format(add, sub, mul, div))
```

```
add=9,sub=11,mul=-10,div=-10.0
tpl=(0, 10, -25, -1.0)

add=[0 0 0],sub=[2 4 6],mul=[-1 -4 -9],div=[-1. -1. -1.]
```

2.2 numpy

2.2.1 任意の ndarray オブジェクトの作成

<ndarray>は<list>から作成できる．普通の<list>の場合は 1 次元の<ndarray>(以下<1d_ndarray>と書く)，入れ子の<list>の場合は 2 次元の<ndarray>(以下<2d_ndarray>と書く) となる．ここで，<1d_ndarray>の<shape>は，要素が 1 つのタプルで返ってくるので注意.

Grammer 2.5.

- np.array(<list>): <list>から<ndarray>を作成する.

- <ndarray>.shape: <ndarray>の行数と列数を（行数，列数）の形のタプルで出力する．ここで，<1d_ndarray>の場合は要素が 1 つのタプルで出力され，行や列という概念がそもそもないことに注意．

Code 2.6 (num1.py).

```
import numpy as np

lst_1d = [1,3,5,9]
array_1d = np.array(lst_1d)
print('array_1d=\n{}\nshape={}\ntype={}\n'.format(array_1d, array_1d.shape, type(array_1d)))

lst_2d = [[1,3,5],[2,4,6],[3,6,9],[5,10,15]]
array_2d = np.array(lst_2d)
print('array_2d=\n{}\nshape={}\ntype={}\n'.format(array_2d, array_2d.shape, type(array_1d)))
```

```
array_1d=
[1 3 5 9]
shape=(4,)
type=<class 'numpy.ndarray'>

array_2d=
[[ 1 3 5]
 [ 2 4 6]
 [ 3 6 9]
 [ 5 10 15]]
shape=(4, 3)
type=<class 'numpy.ndarray'>
```

2.2.2 <ndarray>の一部分の取り出し

<ndarray>はスライスで要素の一部取り出しができる．ここで，<2d_ndarray>から 1 行または 1 列だけ抜き出したときは，<2d_ndarray>ではなく<1d_ndarray>に変わってしまうので注意する (例えば，<2d_ndarray>の特定の行を 1 行抜き出して代入するような場合，<1d_ndarray>を代入する必要がある)．また，要素 1 つだけ取り出した場合は<ndarray>ではなく普通の数値になることも注意する．

Grammer 2.6.

- <1d_ndarray>[a:b:c]: <1d_ndarray>からスライス a:b:c で抜き出す．抜き出した結果は，要素が 1 つの場合は数値，複数ある場合は<1d_ndarray>となる．
- <2d_ndarray>[a:b:c, d:e:f]: <2d_ndarray>から行方向についてスライス a:b:c，列方向についてスライス d:e:f で抜き出す．抜き出した結果は，要素が 1 つの場合は数値，1 列または 1 行だけの場合は<1d_ndarray>，それ以外の場合は<2d_ndarray>となる．なお，d:e:f を省略して行だけの指定で抜き出しをすることが可能だが，d:e:f を省略することはできない．列だけの指定で抜き出したい場合は<2d_ndarray>[:, d:e:f] という指定とする必要がある．

Code 2.7 (num5.py).

```
import numpy as np
```

```
array_1d = np.array([1,2,3,4,5])
array_2d = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])

print('array_1d[1::2]=\n{}'.format(array_1d[1::2]))
print('array_2d[::2,::2]=\n{}'.format(array_2d[::2,::2]))
print('array_2d[3]=\n{},shape={}'.format(array_2d[3],array_2d[3].shape))
print('array_2d[:,1]=\n{},shape={}'.format(array_2d[:,1],array_2d[:,1].shape))
print('array_2d[0,0]=\n{}'.format(array_2d[0,0]))
```

```
array_1d[1::2]=
[2 4]
array_2d[::2,::2]=
[[1 3]
 [7 9]]
array_2d[3]=
[10 11 12],shape=(3,)
array_2d[:,1]=
[ 2 5 8 11],shape=(4,)
array_2d[0,0]=
1
```

2.2.3 <1d_ndarray>と列ベクトル・行ベクトルとしての<2d_ndarray>

<1d_ndarray>には列や行といった概念はないため，<1d_ndarray>をそのまま列ベクトルや行ベクトルとして線形代数計算に使うことはできない (うまく動くときもあるが，そうではない場合のほうが多い)．よって，列ベクトルや行ベクトルは必ず<2d_ndarray>で定義する必要がある (普通のリストではなく入れ子リストを与える必要がある)．

なお，<1d_ndarray>を列ベクトルとしての<2d_ndarray>に変換するには，np.c_[<1d_ndarray>] を使用する．また，<1d_ndarray>を行ベクトルとしての<2d_ndarray>に変換するには，一旦 np.c_[<1d_ndarray>] で列ベクトルを作った上で，<2d_ndarray>を転置させるメソッド<2d_ndarray>.T を使用する．

Grammer 2.7.

- np.c_[<1d_ndarray>]: <1d_ndarray>から列ベクトルとしての<2d_ndarray>を作成する．ここで，行ベクトルとしての<2d_ndarray>にこれを適用しても列ベクトルとしての<2d_ndarray>にはならず，そのままになってしまうので注意する．
- <2d_ndarray>.T: <2d_ndarray>を転置する (<1d_ndarray>に使用しても何も起こらないので注意．また，列ベクトルとしての<2d_ndarray>を転置しても<1d_ndarray>にはならずに単に行ベクトルとしての<2d_ndarray>になることに注意)．

Code 2.8 (num3.py)**.**

```
import numpy as np

lst_1d = [1,3,5,9]
array_1d = np.array(lst_1d)
print('array_1d=\n{}\nshape={}\ntype={}\n'.format(array_1d, array_1d.shape, type(array_1d)))
```

```
lst_cv = [[1],[3],[5],[9]]
array_cv = np.array(lst_cv)
print('array_cv=\n{}\nshape={}\ntype={}\n'.format(array_cv,array_cv.shape, type(array_cv)))

lst_rv = [[1,3,5,9]]
array_rv = np.array(lst_rv)
print('array_cv=\n{}\nshape={}\ntype={}\n'.format(array_rv,array_rv.shape, type(array_rv)))


array_1d_modc = np.c_[array_1d]
print('array_1d_modc=\n{}\nshape={}\ntype={}\n'.format(array_1d_modc,
                                array_1d_modc.shape, type(array_1d_modc)))

array_1d_modr = np.c_[array_1d].T
print('array_1d_modr=\n{}\nshape={}\ntype={}\n'.format(array_1d_modr,
                                array_1d_modr.shape, type(array_1d_modr)))

array_rv_modc = np.c_[array_rv]
print('array_rv_modc=\n{}\nshape={}\ntype={}\n'.format(array_rv_modc,
                                array_rv_modc.shape, type(array_rv_modc)))
```

```
array_1d=
[1 3 5 9]
shape=(4,)
type=<class 'numpy.ndarray'>

array_cv=
[[1]
 [3]
 [5]
 [9]]
shape=(4, 1)
type=<class 'numpy.ndarray'>

array_cv=
[[1 3 5 9]]
shape=(1, 4)
type=<class 'numpy.ndarray'>

array_1d_modc=
[[1]
 [3]
 [5]
 [9]]
shape=(4, 1)
type=<class 'numpy.ndarray'>

array_1d_modr=
[[1 3 5 9]]
shape=(1, 4)
```

```
type=<class 'numpy.ndarray'>

array_rv_modc=
[[1 3 5 9]]
shape=(1, 4)
type=<class 'numpy.ndarray'>
```

2.2.4 <2d_ndarray>を<1d_ndarray>に展開する

<2d_ndarray>を<1d_ndarray>に展開するには，<2d_ndarray>.flatten(<str>) を使う．これを使えば，上記の列ベクトルや行ベクトルとしての<2d_ndarray>を<1d_ndarray>に変換することも可能になる．

Grammer 2.8.

- <2d_ndarray>.flatten(<str>): <2d_ndarray>を<2d_ndarray>に展開する．<str>を省略した場合，行方向に展開し，<str>='F' とした場合，列方向に展開する．

Code 2.9 (num8.py)**.**

```
import numpy as np

A = np.array([[1,2,3,4]])
B = np.array([[1],[2],[3],[4]])
C = np.array([[1,2,3],[4,5,6],[7,8,9]])
D = np.array([[3]])

print('A.flatten()=\n{}\nA.flatten().shape={}\n'.format(A.flatten(),A.flatten().shape))
print('B.flatten()=\n{}\nB.flatten().shape={}\n'.format(B.flatten(),B.flatten().shape))
print('C.flatten()=\n{}\nC.flatten().shape={}\n'.format(C.flatten(),C.flatten().shape))
print('C.flatten(F)=\n{}\nC.flatten(F).shape={}\n'.format(C.flatten('F'),
                                                    C.flatten('F').shape))
print('D.flatten()=\n{}\nD.flatten().shape={}'.format(D.flatten(),D.flatten().shape))
```

```
A.flatten()=
[1 2 3 4]
A.flatten().shape=(4,)

B.flatten()=
[1 2 3 4]
B.flatten().shape=(4,)

C.flatten()=
[1 2 3 4 5 6 7 8 9]
C.flatten().shape=(9,)

C.flatten(F)=
[1 4 7 2 5 8 3 6 9]
C.flatten(F).shape=(9,)

D.flatten()=
```

```
[3]
D.flatten().shape=(1,)
```

2.2.5 スカラーと 1 要素の<1d_ndarray>と 1 要素の<2d_ndarray>

numpy では，1 要素しかない<1d_ndarray>や<2d_ndarray>の概念がスカラーとは別に存在するため，これらは区別される．numpy で計算を行うとき，結果がスカラーだと思いきや 1 要素の<2d_ndarray>だったりして，その後の計算エラーになってしまう場合等があるため注意する．

Code 2.10 (num4.py)**.**

```
import numpy as np

scl = 3
array_1d_scl = np.array([3])
array_2d_scl = np.array([[3]])

print('scalar={},␣type={}'.format(scl,type(scl)))
print('array_1d_scl={},␣type={},␣shape={}'.format(array_1d_scl,type(array_1d_scl),
                                                  array_1d_scl.shape))
print('array_2d_scl={},␣type={},␣shape={}'.format(array_2d_scl,type(array_2d_scl),
                                                  array_2d_scl.shape))

print('array_1d_scl[0]={},␣type={}'.format(array_1d_scl[0],type(array_1d_scl[0])))
print('array_2d_scl[0]={},␣type={},␣shape={}'.format(array_2d_scl[0],
                                                    type(array_2d_scl[0]),array_2d_scl[0].shape))
print('array_2d_scl[0,0]={},␣type={}'.format(array_2d_scl[0,0],type(array_2d_scl[0,0])))
```

```
scalar=3, type=<class 'int'>
array_1d_scl=[3], type=<class 'numpy.ndarray'>, shape=(1,)
array_2d_scl=[[3]], type=<class 'numpy.ndarray'>, shape=(1, 1)
array_1d_scl[0]=3, type=<class 'numpy.int64'>
array_2d_scl[0]=[3], type=<class 'numpy.ndarray'>, shape=(1,)
array_2d_scl[0,0]=3, type=<class 'numpy.int64'>
```

2.2.6 数学の定数，関数など

ネイピア数 e は np.e で使用できる．また，e^x は np.exp(x) で計算できる．以下に重要なものをまとめる．これらの関数の引数には<2d_ndarray>などの配列を与えてもよく，この場合は各成分ごとに計算される．

Grammer 2.9.

- np.e: ネイピア数 e を表す．
- np.exp(x): e^x を計算する．なお，x に<2d_ndarray>などの配列を与えた場合は配列の各成分に対して計算する．
- np.log(x): $\log x$ を計算する．なお，x に<2d_ndarray>などの配列を与えた場合は配列の各成分に対して計算する．

- np.log10(x): $\log_{10} x$ を計算する．なお，x に<2d_ndarray>などの配列を与えた場合は配列の各成分に対して計算する．

Code 2.11 (num11.py).

```
import numpy as np

x = np.array([np.e, np.log(3), 10, np.log10(3)])

print('e^x={}'.format(np.exp(x)))
print('log(x)={}'.format(np.log(x)))
print('10^x={}'.format(10 ** x))
print('log10(x)={}'.format(np.log10(x)))
```

```
e^x=[1.51542622e+01 3.00000000e+00 2.20264658e+04 1.61142883e+00]
log(x)=[ 1. 0.09404783 2.30258509 -0.73998462]
10^x=[5.22735300e+02 1.25490916e+01 1.00000000e+10 3.00000000e+00]
log10(x)=[ 0.43429448 0.04084445 1. -0.32137124]
```

2.2.7 規則的な<1d_ndarray>の作成

<1d_ndarray>をリスト直打ちで作成するのは，大きい配列を作ろうとする場合は不便であるが，規則的な配列であれば，別のプログラムで書くことができる．

Grammer 2.10.

- np.zeros(a): 要素が全て 0 である a 個の要素を持つ<1d_ndarray>を作成する．
- np.arange(a,b,c): a から b の手前まで，c ずつ増加していく<1d_ndarray>を作成する．
- np.arange(a): 0 から a の手前まで，1 ずつ増加していく<1d_ndarray>を作成する．
- np.arange(a,b): a から b の手前まで，1 ずつ増加していく<1d_ndarray>を作成する．
- np.random.randn(a): 標準正規分布に従う乱数から発生する a 個の要素の<1d_ndarray>を作成する．
- np.linspace(a,b,c): a から始まり、$\frac{b-a}{c-1}$ ずつ変化して b で終わる長さ c の<1d_ndarray>を作成する．これは，グラフを描画するときに等間隔に各点が欲しくなる場合などによく活用される．
- np.logspace(a,b,c,base=x): x^a から始まり，a の部分が $\frac{b-a}{c-1}$ ずつ変化して x^b で終わる長さ c の<1d_ndarray>を作成する．なお，base を省略した場合は，10 の累乗となる．これは，指数的に増加するようなグラフを対数目盛で描画したい場合などによく活用される．

Code 2.12 (num2.py).

```
import numpy as np

v = np.zeros(5)
w = np.arange(0,30,2)
x = np.arange(10)
y = np.arange(-10,1)
z = np.random.randn(10)
u = np.linspace(-10,10,10)
t = np.logspace(0,10,11,base=2)
```

```
print('v={}'.format(v))
print('w={}'.format(w))
print('x={}'.format(x))
print('y={}'.format(y))
print('z={}'.format(z))
print('u={}'.format(u))
print('t={}'.format(t))
```

```
v=[0. 0. 0. 0. 0.]
w=[ 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28]
x=[0 1 2 3 4 5 6 7 8 9]
y=[-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0]
z=[-0.38558831 1.20391439 1.18496279 1.50358118 0.93103295 -0.02233168
  0.04894498 0.20757944 -2.67874618 -1.90888007]
u=[-10. -7.77777778 -5.55555556 -3.33333333 -1.11111111
   1.11111111 3.33333333 5.55555556 7.77777778 10. ]
t=[1.000e+00 2.000e+00 4.000e+00 8.000e+00 1.600e+01 3.200e+01 6.400e+01
 1.280e+02 2.560e+02 5.120e+02 1.024e+03]
```

2.2.8 規則的な<2d_ndarray>の作成

規則的な<2d_ndarray>も作成可能.

Grammer 2.11.

- np.zeros((a,b)): 全ての要素が 0 である a 行 b 列の<2d_ndarray>を作成する.

Code 2.13 (num7.py).

```
import numpy as np

A = np.zeros((4,3))

print('A=\n{}\nshape={}'.format(A,A.shape))
```

```
A=
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
shape=(4, 3)
```

2.2.9 線形代数計算

基本的には数値と<2d_ndarray>で計算する．四則演算等について，その計算が定義されるのであれば通常の数値のように式を書けば問題ないが，掛け算記号は，線形代数計算においてはアダマール積であるため，行列積を行うときは<2d_ndarray>.dot(<2d_ndarray>) で行う必要がある．なお，線形代数計算の結果スカラーになる場合でも，数値ではなく 1 要素しかない<2d_ndarray>で結果が保持されることに注意する.

Grammer 2.12.

- <A_2d_ndarray>.dot(<B_2d_ndarray>): 行列積 AB を計算する.
- <A_2d_ndarray>∗<B_2d_ndarray>: アダマール積 $A * B$ を計算する.
- np.linalg.pinv(<A_2d_ndarray>): 行列 A のムーア・ペンローズ一般逆行列 A^+ を計算する.

Code 2.14 (num6.py).

```
import numpy as np

A = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
B = np.array([[1,2],[3,4],[5,6]])
C1 = np.array([[1],[2]])
C2 = np.array([[3],[4]])
D = np.array([[1,2,3,4]])
E = np.array([[-6,5],[4,-3],[-2,1]])

F = A.dot(B).dot(C1).dot(D)
G = D.dot(A.dot(B).dot(C1 + C2))
H = B * E

I = np.array([[1,1],[1,1],[1,1]])
J = np.linalg.pinv(I)

print('F=\n{}'.format(F))
print('G=\n{}'.format(G))
print('H=\n{}'.format(H))
print('J=\n{}'.format(J))
```

```
F=
[[ 78 156 234 312]
 [ 177 354 531 708]
 [ 276 552 828 1104]
 [ 375 750 1125 1500]]
G=
[[9040]]
H=
[[ -6 10]
 [ 12 -12]
 [-10 6]]
J=
[[0.16666667 0.16666667 0.16666667]
 [0.16666667 0.16666667 0.16666667]]
```

2.2.10 複数の<1d_ndarray>を連結し<2d_ndarray>を作る

np.c_[<1d_ndarray>,<1d_ndarray>,...] とすることで，各<1d_ndarray>を列ベクトルとしての<2d_ndarray>とした上でそれらを連結し行列化できる.

Grammer 2.13.

- np.c_[<1d_ndarray>,<1d_ndarray>,...]: 各<1d_ndarray>を列ベクトルの<2d_ndarray>と変換した上でそれらを連結し行列とする.

Code 2.15 (num9.py).

```
import numpy as np

array1 = np.array([1,2,3,4])
array2 = np.array([-1,-2,-3,-4])

matrix = np.c_[array1, array2]

print('matrix=\n{},\nshape={}'.format(matrix, matrix.shape))
```

```
matrix=
[[ 1 -1]
 [ 2 -2]
 [ 3 -3]
 [ 4 -4]],
shape=(4, 2)
```

2.2.11 等間隔の 2 次元の格子点を表現する<2d_ndarray>の作成

等間隔の 1 次元の点は np.linspace(a,b,c) で作成可能であるが，等間隔の 2 次元の格子点は，2 つの np.linspace(a,b,c) の組み合わせとして表現される．例えば，等間隔 1 次元の点 $0,1,2$ と $0,1,2$ からなる等間隔の 2 次元の格子点は，9 個の点からなり，$(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)$ となる．この格子点は，np.meshgrid(<1d_ndarray>,<1d_ndarray>) を用いて効率的に作成することができる．が，出力が少々独特なので，よく理解して使用する必要がある．

Grammer 2.14.

- np.meshgrid(<1d_ndarray>,<1d_ndarray>): 格子点の第 1 成分，第 2 成分のペアを，[<2d_ndarray>,<2d_ndarray>] という<list>で出力する．[<1d_ndarray>,<1d_ndarray>] のような感じで出るのではと想像しがちだが，そうではないので注意.

ここから例えば $\{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$ のような 2 次元配列を作りたければ，成分ごとに別々の変数で受けた上で，それぞれ<2d_ndarray>.flatten() で<1d_ndarray>に変換し，結合することで作成することになる．

Code 2.16 (num10.py).

```
import numpy as np

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

grid = np.meshgrid(ary, ary)
print('grid=\n{}\ntype={}'.format(grid, type(grid)))

grid1, grid2 = np.meshgrid(ary, ary)
grid1 = grid1.flatten()
grid2 = grid2.flatten()
grid = np.c_[grid1, grid2]
print('grid=\n{}\ntype={}'.format(grid, type(grid)))
```

```
grid=
[array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]]), array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2]])]
type=<class 'list'>
grid=
[[0 0]
 [1 0]
 [2 0]
 [0 1]
 [1 1]
 [2 1]
 [0 2]
 [1 2]
 [2 2]]
type=<class 'numpy.ndarray'>
```

なお，matplotlib の一部の描画では，<1d_ndarray>ではなく行ベクトルや列ベクトルでもない，行列としての<2d_ndarray>を与えなければならないものもあり，そういう場合は np.meshgrid(<1d_ndarray>,<1d_ndarray>) で生成された<2d_ndarray>はそのまま使用することになる．

2.3 pandas

2.3.1 カンマ区切りテキストを DataFrame オブジェクトに格納 (read_csv 関数)

機械学習の元となるデータは基本的に表形式データであるため，まずそれを読み込むところから始まる．pandas の<DataFrame>は，表形式データを格納できる．カンマ区切りのテキストファイル (txt,csv) は，pd.read_csv() で読み込める．また，読み込んだ表の行数と列数は，<DataFrame>.shape で取り出せる．また，読み込む<DataFrame>は巨大であることが多いので，読み込んだあとはその中身の最初の数行を表示して読み込まれているかどうかを確認することが多い．このような処理は<DataFrame>.head(<int>) で行うことができる．

Grammer 2.15.

- pd.read_csv(<str>, names=<list>): <str>で指定したファイルパスのデータ（カンマ区切り）を読み込み<DataFrame>に格納する．names は，列名をつけるときに指定する．
- <DataFrame>.shape: <DataFrame>の行数と列数を (行数，列数) の形のタプルで出力する．ここで，列数には index 列はカウントされないことに注意する．
- <DataFrame>.head(<int>): <DataFrame>の最初の<int>行を<DataFrame>で取り出す．

Code 2.17 (pd1.py)．ここで読み込む ex1data1.txt は，以下のようなデータである (97 行 2 列，カンマ区切り，列名ヘッダーなし．以下では最初の 5 行だけ表示している)．データは，左は population of city in 10,000s，右は profit in $10,000s を表している．ヘッダーがないので，読み込む際は列名をつけている．

```
6.1101,17.592
5.5277,9.1302
8.5186,13.662
7.0032,11.854
5.8598,6.8233
```

```
import pandas as pd

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])
m, n = df.shape
print('df=\n{}\ntype={}\nlow_num={},␣col_num={}'.format(df.head(10), type(df), m, n))
```

```
df=
   population profit
0 6.1101 17.5920
1 5.5277 9.1302
2 8.5186 13.6620
3 7.0032 11.8540
4 5.8598 6.8233
5 8.3829 11.8860
6 7.4764 4.3483
7 8.5781 12.0000
8 6.4862 6.5987
9 5.0546 3.8166
type=<class 'pandas.core.frame.DataFrame'>
low_num=97, col_num=2
```

2.3.2 <2d_ndarray>から<DataFrame>の作成

<2d_ndarray>から<DataFrame>を作成できる．このときの列名は<list>で指定する．

Grammer 2.16.

- pd.DataFrame(<2d_ndarray>, columns=<list>): <2d_ndarray>から<DataFrame>を作成する．columns は，列名をつけるときに指定する．

Code 2.18 (pd6.py).

```
import numpy as np
import pandas as pd

array_2d = np.array([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
colnames = ['col1', 'col2', 'col3']

df = pd.DataFrame(array_2d, columns=colnames)
print('df=\n{}'.format(df))
```

```
df=
   col1 col2 col3
0 1 2 3
1 4 5 6
```

```
2 7 8 9
3 10 11 12
```

2.3.3 列の追加

機械学習では，新たに特徴量をデータに追加することが頻繁に行われる．全ての行の数値が同じ列を追加するには，新しい列を指定して数値を代入する文を書く．また，行数と要素数が等しい場合は，新しい列を指定して<1d_ndarray>や<list>を代入する文を書くことでその配列を追加することができる．

Grammer 2.17.

- <DataFrame>[<str>] = a: <DataFrame>の新しい列<str>の全要素に a を代入する．
- <DataFrame>[<str>] = <1d_ndarray or list>: <DataFrame>の新しい列<str>に<1d_ndarray or list>を代入する (列ベクトルとしての<2d_ndarray>でも問題ないが，行ベクトルとしての<2d_ndarray>はエラーとなるので注意)．

Code 2.19 (pd3.py)**.**

```
import pandas as pd
import numpy as np

df = pd.read_csv('../data/ex1data1_test.txt', names=['population','profit'])
m, n = df.shape

df['all_1'] = 1
df['ndarray'] = np.arange(m)
df['list'] = range(m)

print('df=\n{}'.format(df))
```

```
df=
   population profit all_1 ndarray list
0 6.1101 17.5920 1 0 0
1 5.5277 9.1302 1 1 1
2 8.5186 13.6620 1 2 2
3 7.0032 11.8540 1 3 3
4 5.8598 6.8233 1 4 4
```

2.3.4 <DataFrame>の一部分の取り出し

トレーニングデータとテストデータに分ける，あるいは特徴量と目的変数に分ける等，機械学習では<DataFrame>から一部分を取り出す操作を頻繁に行う．取り出したものの列が 1 つしかない場合，<DataFrame>ではなく<Series>で取り出されるので注意．

注意 2.1. 取り出したものの列が 1 つしかない場合，<DataFrame>ではなく<Series>で取り出される．

Grammer 2.18.

- <DataFrame>[<str>]: <DataFrame>の列<str>を<Series>で抜き出す．

- <DataFrame>[<list>]: <DataFrame>の<list>で指定した複数列を<DataFrame>で抜き出す.
- <DataFrame>[a:b:c]: <DataFrame>のスライス a:b:c で指定した行を<DataFrame>で抜き出す.(1 行だけの抜き出しでも<Series>ではなく<DataFrame>となる.)
- <DataFrame>.loc[a:b:c,<str or list>]: <DataFrame>のスライス a:b:c で指定した行かつ<str or list>で指定した列で選ばれる表データを<DataFrame>で抜き出す.

Code 2.20 (pd4.py).

```
import pandas as pd
import numpy as np

df = pd.read_csv('../data/ex1data1_test.txt', names=['population','profit'])
m, n = df.shape

df['all_1'] = 1
df['ndarray'] = np.arange(m)
df['list'] = range(m)

df_sub1 = df['profit']
df_sub2 = df[['list','population']]

df_sub3 = df[0:1]
df_sub4 = df[2:4]

df_sub5 = df.loc[3:6,'list']
df_sub6 = df.loc[::2,['profit','ndarray']]

print('df_sub1=\n{}\ntype={}\n'.format(df_sub1,type(df_sub1)))
print('df_sub2=\n{}\ntype={}\n'.format(df_sub2,type(df_sub2)))
print('df_sub3=\n{}\ntype={}\n'.format(df_sub3,type(df_sub3)))
print('df_sub4=\n{}\ntype={}\n'.format(df_sub4,type(df_sub4)))
print('df_sub5=\n{}\ntype={}\n'.format(df_sub5,type(df_sub5)))
print('df_sub6=\n{}\ntype={}\n'.format(df_sub6,type(df_sub6)))
```

```
df_sub1=
0 17.5920
1 9.1302
2 13.6620
3 11.8540
4 6.8233
Name: profit, dtype: float64
type=<class 'pandas.core.series.Series'>

df_sub2=
   list population
0 0 6.1101
1 1 5.5277

2 2 8.5186
3 3 7.0032
4 4 5.8598
```

```
type=<class 'pandas.core.frame.DataFrame'>

df_sub3=
   population profit all_1 ndarray list
0 6.1101 17.592 1 0 0
type=<class 'pandas.core.frame.DataFrame'>

df_sub4=
   population profit all_1 ndarray list
2 8.5186 13.662 1 2 2
3 7.0032 11.854 1 3 3
type=<class 'pandas.core.frame.DataFrame'>

df_sub5=
3 3
4 4
Name: list, dtype: int64
type=<class 'pandas.core.series.Series'>

df_sub6=
    profit ndarray
0 17.5920 0
2 13.6620 2
4 6.8233 4
type=<class 'pandas.core.frame.DataFrame'>
```

2.3.5 <Series>と 1 列しか持たない<DataFrame>の違い

<Series>と<DataFrame>の関係は，<1d_ndarray>と<2d_ndarray>と同じである．すなわち，1 列しか持たない<DataFrame>と<Series>は shape が異なる．なお，<Series>を<DataFrame>に変換する場合は，pd.DataFrame(<Series>) とすればよい．

Grammer 2.19.

- pd.DataFrame(<Series>): <Series>を<DataFrame>に変換する．

Code 2.21 (pd5.py).

```
import pandas as pd
import numpy as np

df = pd.read_csv('../data/ex1data1_test.txt', names=['population','profit'])

df_sub1 = df['profit']
df_sub2 = df[2:3]

print('df_sub1=\n{}\ntype={}\nshape={}\n'.format(df_sub1,type(df_sub1),df_sub1.shape))
print('df_sub2=\n{}\ntype={}\nshape={}\n'.format(df_sub2,type(df_sub2),df_sub2.shape))

df_sub3 = pd.DataFrame(df_sub1)
```

```
print('df_sub3=\n{}\ntype={}\nshape={}\n'.format(df_sub3,type(df_sub3),df_sub3.shape))
```

```
df_sub1=
0 17.5920
1 9.1302
2 13.6620
3 11.8540
4 6.8233
Name: profit, dtype: float64
type=<class 'pandas.core.series.Series'>
shape=(5,)

df_sub2=
   population profit
2 8.5186 13.662
type=<class 'pandas.core.frame.DataFrame'>
shape=(1, 2)

df_sub3=
    profit
0 17.5920
1 9.1302
2 13.6620
3 11.8540
4 6.8233
type=<class 'pandas.core.frame.DataFrame'>
shape=(5, 1)
```

2.3.6 values メソッド

<DataFrame or Series>から，index や列名を除いた値のみを取り出したいときは<DataFrame or Series>.values メソッドを使う．<DataFrame>の場合は<2d_ndarray>，<Series>の場合は<1d_ndarray>となる．

Grammer 2.20.

- <DataFrame or Series>.values: <DataFrame or Series>から，index や列名を除いた値のみを取り出して<ndarray>として格納する．ここで，<DataFrame>の場合は<2d_ndarray>，<Series>の場合は<1d_ndarray>である．特に，1 行しかない<DataFrame>は行ベクトル，1 列しかない<DataFrame>は列ベクトルとなることに注意．

Code 2.22 (pd2.py).

```
import pandas as pd

df = pd.read_csv('../data/ex1data1_test.txt', names=['population','profit'])

array_1 = df.values
array_2 = df['population'].values
```

```
array_3 = df[0:1].values
array_4 = pd.DataFrame(df['profit']).values

print('array_1=\n{}\ntype={}\nshape={}\n'.format(array_1,type(array_1),array_1.shape))
print('array_2=\n{}\ntype={}\nshape={}\n'.format(array_2,type(array_2),array_2.shape))
print('array_3=\n{}\ntype={}\nshape={}\n'.format(array_3,type(array_3),array_3.shape))
print('array_4=\n{}\ntype={}\nshape={}'.format(array_4,type(array_4),array_4.shape))
```

```
array_1=
[[ 6.1101 17.592 ]
 [ 5.5277 9.1302]
 [ 8.5186 13.662 ]
 [ 7.0032 11.854 ]
 [ 5.8598 6.8233]]
type=<class 'numpy.ndarray'>
shape=(5, 2)

array_2=
[6.1101 5.5277 8.5186 7.0032 5.8598]
type=<class 'numpy.ndarray'>
shape=(5,)

array_3=
[[ 6.1101 17.592 ]]
type=<class 'numpy.ndarray'>
shape=(1, 2)

array_4=
[[17.592 ]
 [ 9.1302]
 [13.662 ]
 [11.854 ]
 [ 6.8233]]
type=<class 'numpy.ndarray'>
shape=(5, 1)
```

2.4 matplotlib

2.4.1 figure オブジェクトと axes オブジェクトの作成

一つ一つのグラフの本体は，matplotlib では<axes>として表され，<axes>は，<figure>の中で管理される．イメージとしては、<figure>は白のキャンバスであり，そこにグラフの素である<axes>を置いていく．空の<axes>は，デフォルトの軸だけがセットされている．

Grammer 2.21.

- plt.figure(): 空の<figure>を作成する．
- <figure>.add_subplot(a,b,c): <figure>を a 行 b 列に分割した上で，c 番目の部分に空の<axes>を作成する．
- <figure>.savefig('XXXX.XXX'): <figure>を XXXX.XXX として保存する．

Code 2.23 (fig1.py).

```
import matplotlib.pyplot as plt

fig = plt.figure()
ax1 = fig.add_subplot(2,2,1)
ax2 = fig.add_subplot(2,2,2)
ax3 = fig.add_subplot(2,2,3)

fig.savefig('fig1.eps')
```

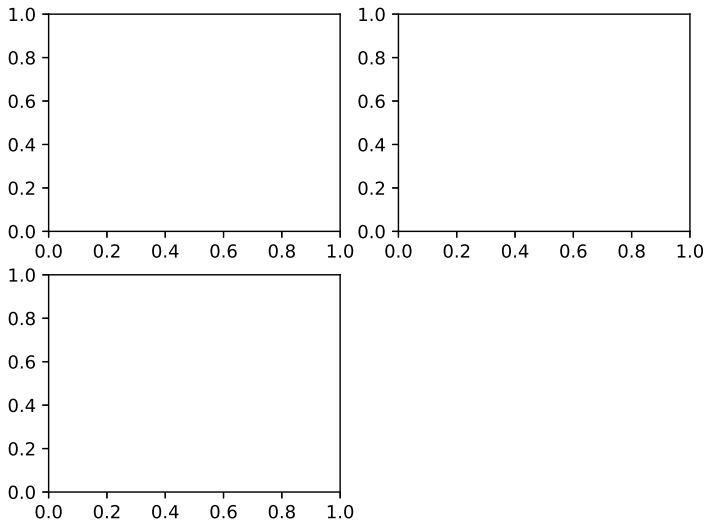

図 2.1 fig1.eps

2.4.2 <axes>間の余白調整

fig1.eps をみると，<axes>間の余白が詰まっている．デフォルトのまま，これに軸ラベルやグラフのタイトルをつけていくと，大体の場合で重なってしまう．ので，<figure>.subplots_adjust() で余白を調整する必要がある．

Grammer 2.22.

- <figure>.subplots_adjust(wspace=a, hspace=b): <figure>.subplot() での<axes>の余白を調整する．wspace=a は，<axes>の横並び間隔を a インチ広げる．hspace=b は，<axes>の縦並び間隔を b インチ広げる．

Code 2.24 (fig7.py).

```
import matplotlib.pyplot as plt

fig = plt.figure()
ax1 = fig.add_subplot(2,2,1)
ax2 = fig.add_subplot(2,2,2)
ax3 = fig.add_subplot(2,2,3)
```

```
fig.subplots_adjust(wspace=0.3, hspace=0.5)

fig.savefig('fig7.eps')
```

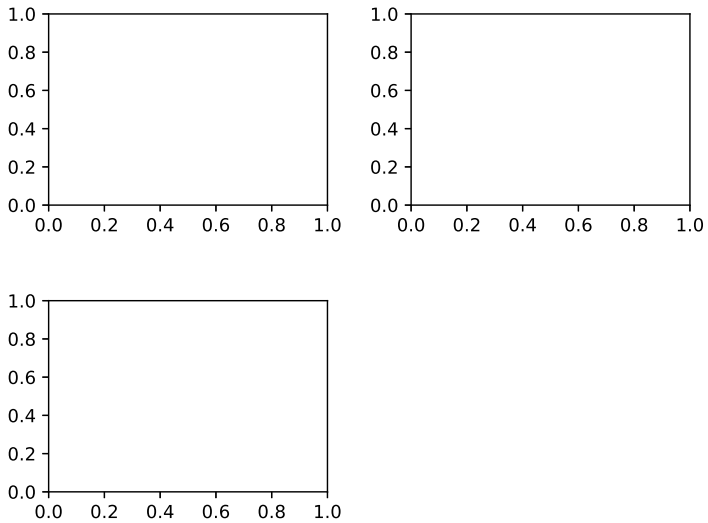

図 2.2 fig7.eps

2.4.3 2 次元の散布図

2 次元の散布図は，<axes>.scatter() で描画することができる．散布図の場合は一つ一つの点がどれくらいの数値なのかがわかりにくいので，<axes>.grid() でグリッド補助線を追加する．また，各点についてどっちが x でどっちが y かわからないので横軸と縦軸に<axes>.set_xlabel(<str>)，<axes>.set_ylabel(<str>) でラベルをつける．

Grammer 2.23.

- <axes>.scatter(x,y): 2 次元データ $(\boldsymbol{x}, \boldsymbol{y})$ の散布図を描画する．ここで，$\boldsymbol{x}, \boldsymbol{y}$ は<1d_ndarray>であり，左のデータが横軸，右のデータが縦軸である．
- <axes>.grid(): <axes>にグリッド補助線を追加する．
- <axes>.set_xlabel(<str>): <axes>の横軸にラベルをつける．
- <axes>.set_ylabel(<str>): <axes>の縦軸にラベルをつける．

注意 2.2. 基本的に matplotlib に渡すデータは，基本的には<1d_ndarray>であることが必要なので，以降はその前提とする．

Code 2.25 (fig2.py)**.**

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

fig = plt.figure()
ax = fig.add_subplot(1,1,1)

x = np.arange(30)
```

```
y = np.arange(30) + 3 * np.random.randn(30)

ax.scatter(x, y)

ax.grid()
ax.set_xlabel('x')
ax.set_ylabel('y')

fig.savefig('fig2.eps')
```

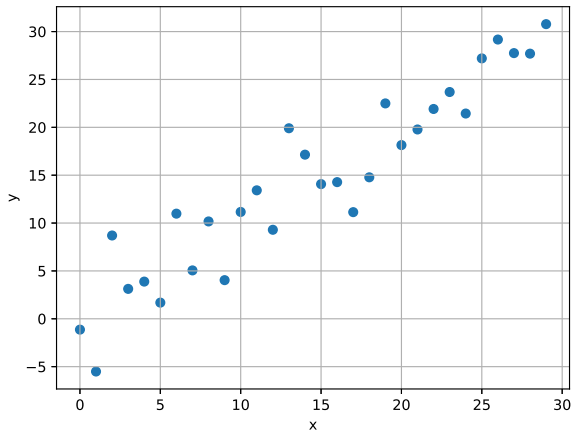

図 2.3 fig2.eps

2.4.4 2 次元の折れ線図

2 次元の折れ線図は、<axes>.plot() で描画することができる．各点を結ぶようにして線が描画される．

Grammer 2.24.

- <axes>.plot(x,y): 2 次元データ $(\boldsymbol{x}, \boldsymbol{y})$ の各点を結んだ線を描画する．ここで，$\boldsymbol{x}, \boldsymbol{y}$ は<1d_ndarray>であり，左のデータが横軸，右のデータが縦軸である．

Code 2.26 (fig3.py).

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

x = np.array([1,2,8,10])
y = np.array([-3,5,-2,3])

fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(x, y)
ax.set_xlabel('x')
ax.set_ylabel('y')

fig.savefig('fig3.eps')
```

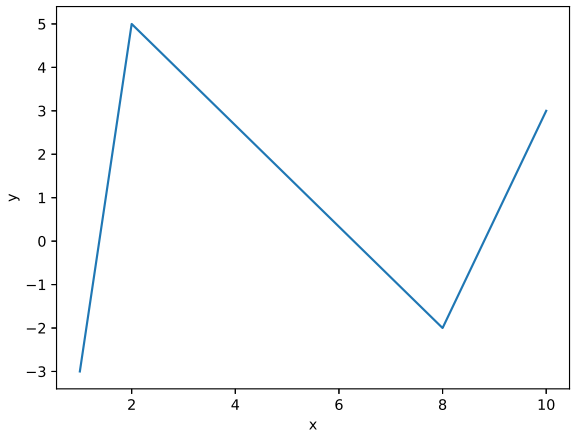

図 2.4 fig3.eps

2.4.5 複数の図の重ね合わせ

<axes>にどんどんメソッドを重ねていくイメージで，一つの描画スペースに複数の図を重ねて置くことができる．また，凡例は，各 plot に label=<str>を設定し，<axes>.legend() で表示させることができる (plot 以外の描画系関数でも同様のことは可能).

Grammer 2.25.

- <axes>.plot(x,y,label=<str>): 2 次元データ $(\boldsymbol{x}, \boldsymbol{y})$ の各点を結んだ線を描画する．また，オプション label=<str>で plot に名前をつける．
- <axes>.legend(): <axes>の凡例をつける．ここで，凡例は描く描画系関数で設定した label=<str>となる．

Code 2.27 (fig6.py).

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

fig = plt.figure()
ax = fig.add_subplot(1,1,1)

x = np.arange(30)
y = np.arange(30) + 3 * np.random.randn(30)

ax.scatter(x, y)
ax.plot(x, x, label='y=x')
ax.plot(x, x / 2, label='y=x/2')

ax.grid()
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()

fig.savefig('fig6.eps')
```

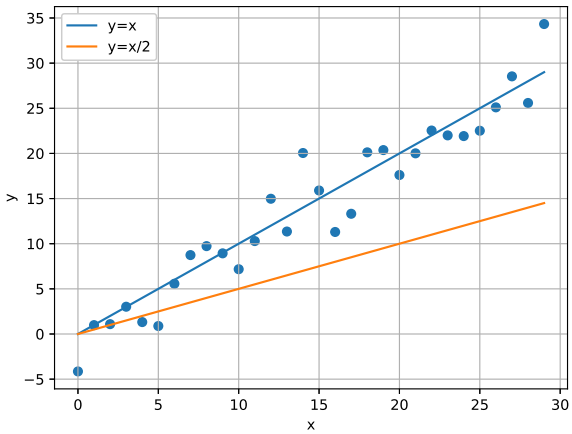

図 2.5 fig6.eps

2.4.6 3 次元の曲面図

$z = f(x, y)$ の 3 次元の曲面図は<axes>.plot_surface() で描画することができる．これを使うためには，matplotlib の mpl_toolkits.mplot3d から Axes3D をインポートした上で，<figure>.add_subplots() による<axes>生成時に 3 次元の図を書くということを明示的に指定するため projection='3d' を指定してあげる必要がある．また，<axes>.plot_surface() で指定する配列は全て行列としての<2d_ndarray>でなければならない．そのため，np.meshgrid() により<2d_ndarray>を生成して使うのが一般的である．

Grammer 2.26.

- <axes>.plot_surface(x,y,z): $z = f(x, y)$ の型の曲面を描画する．ここで，x,y,z は行列としての<2d_ndarray>であることが必要である．なお，このメソッドを使うためには，matplotlib の mpl_toolkits.mplot3d から Axes3D をインポートし，<axes>を<figure>.add_subplots() で作成するときに projection='3d' を指定する必要がある．
- <axes>.set_zlabel(<str>): z 軸のラベルを設定する．
- <axes>.set_title(<str>): 描画した<axes>にタイトルをつける．
- latex 記法: matplotlib において<str>に latex の数式を使いたい場合，r<str>と書く．

Code 2.28 (fig4.py).
$f(x, y) = x^2 - y^2$ $(-5 \leq x \leq 5, -5 \leq y \leq 5)$ のグラフを描画した例．Python の関数をシンプルに書いているが，<ndarray>の四則演算は各成分ごとの計算となることを利用して，<2d_ndarray>の全成分に対して f の値をサクッと計算させて<2d_ndarray>のまま保持するようにしている．

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

def f(x, y):
    return x ** 2 - y ** 2
```

```
ary = np.linspace(-5,5,50)
grid1, grid2 = np.meshgrid(ary, ary)
f_val = f(grid1, grid2)

fig = plt.figure()
ax = fig.add_subplot(1,1,1, projection='3d')
ax.plot_surface(grid1, grid2, f_val)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.set_title(r'$z=f(x,y)$')

fig.savefig('fig4.eps')
```

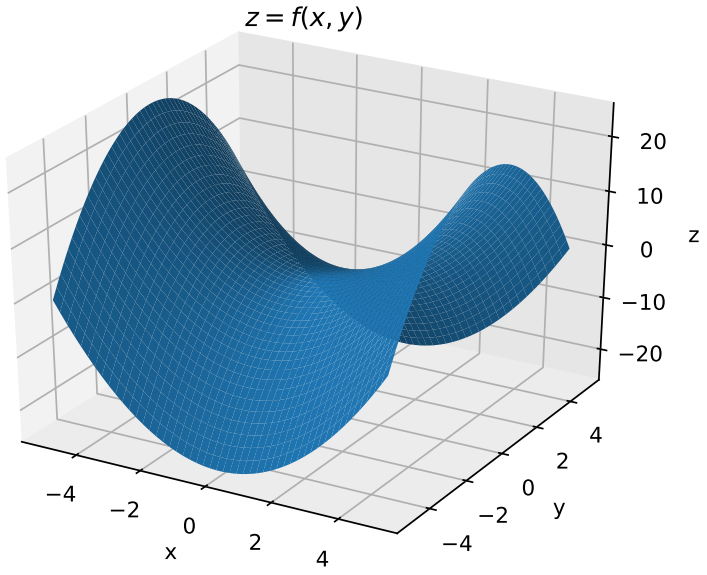

図 2.6 fig4.eps

2.4.7 2 次元の等高線図

$z = f(x,y)$ の 2 次元の等高線図は<axes>.contour() で描画する．これも配列には行列としての<2d_ndarray>を指定しなければならないので注意する．

Grammer 2.27.

- <axes>.contour(x,y,z,levels=<1d_ndarray>): $z = f(x,y)$ の型の等高線図を描画する．ここで，x,y,z は行列としての<2d_ndarray>であることが必要である．等高線は，levels=<1d_ndarray>を指定しない場合は z の値について等間隔に描画されるが，例えば z の値が指数的に増加する場合などは少しの x,y の変動で大きく z が動いてしまい等高線同士が詰まってしまうので，levels=<1d_ndarray>に直接 z の値を指定して，等高線を直接定めることもで

きる．

- `plt.figure(figsize=(a,b))`: `figsize` は<figure>領域のサイズ (縦 a インチ，横 b インチ) を指定する．`figsize` を省略した場合は `(8,6)`，すなわち横 8 インチ，縦 6 インチの指定となる．`subplot` をたくさん横につなげる場合でも `figsize` は変わらないので，つぶれたりゆがんだりする．ので，複数の図を描くときは `figsize` を検討する必要がある．

Code 2.29 (`fig5.py`).
$f(x,y) = \frac{e^x+e^{-x}+e^y+e^{-y}}{4}$ $(-5 \leq x \leq 5, -5 \leq y \leq 5)$ について曲面図と等高線図を描画した例．指数関数なので，等高線を f の値で等間隔でとっても x, y で見れば等間隔ではない．ので，右側の図では等高線の間隔を，対数をとったら等間隔になるように指定している．

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

def f(x, y):
    return ( np.exp(x) + np.exp(-x) + np.exp(y) + np.exp(-y) ) / 4

ary = np.linspace(-5,5,50)
grid1, grid2 = np.meshgrid(ary, ary)
f_val = f(grid1, grid2)

fig = plt.figure(figsize=(15,5))

ax1 = fig.add_subplot(1,3,1, projection='3d')
ax1.plot_surface(grid1, grid2, f_val)
ax1.set_xlabel('x')
ax1.set_ylabel('y')
ax1.set_zlabel('z')

ax2 = fig.add_subplot(1,3,2)
ax2.contour(grid1, grid2, f_val)
ax2.set_xlabel('x')
ax2.set_ylabel('y')

ax3 = fig.add_subplot(1,3,3)
ax3.contour(grid1, grid2, f_val, levels=np.logspace(0,5,11,base=np.e))
ax3.set_xlabel('x')
ax3.set_ylabel('y')

fig.savefig('fig5.eps')
```

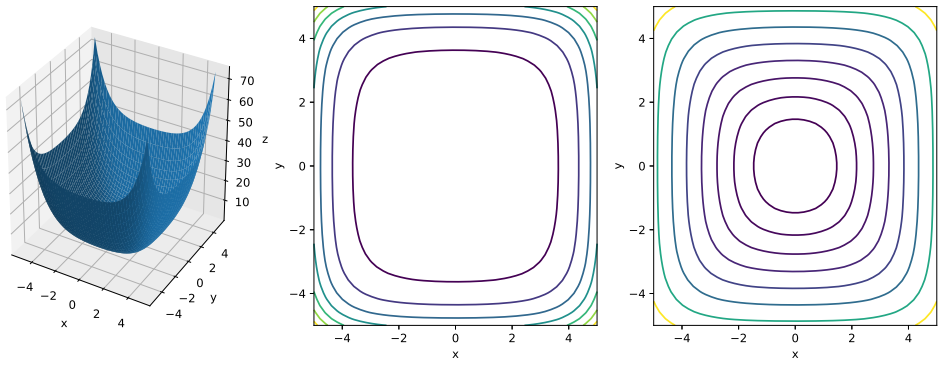

図 2.7 fig5.eps

2.4.8 <axes>間での y 軸の共有

matplotlib は，データに応じて軸の目盛りは自動調整されるが，そのせいで，グラフの見た目 (傾き等) がどれも同じように調整されてしまい，グラフを比較するときに大きな誤認をしてしまう可能性がある．<figure>.subplot() で<axes>間の y 軸を同じくする場合，<axes>を生成するときのオプションに sharey=<axes>を指定する．

Grammer 2.28.

- <figure>.add_subplot(a,b,c, sharey=<axes2>): <figure>を a 行 b 列に分割した上で，c 番目の部分に，y 軸を<axes2>と共有した空の<axes>を作成する．

Code 2.30 (fig8.py).

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

x = np.linspace(0,10,11)
y1 = x
y2 = x * 3

fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(1,2,1)
ax1.plot(x, y1)
ax2 = fig.add_subplot(1,2,2)
ax2.plot(x, y2)

fig.savefig('fig8-1.eps')

fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(1,2,1)
ax1.plot(x, y1)
ax2 = fig.add_subplot(1,2,2, sharey=ax1)
```

```
ax2.plot(x, y2)

fig.savefig('fig8-2.eps')
```

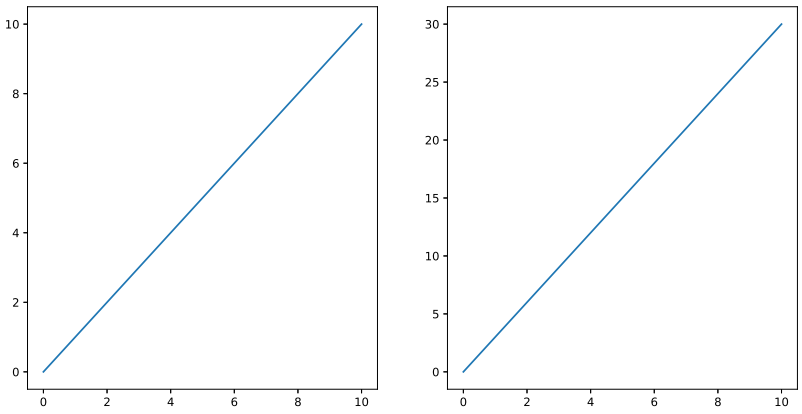

図 2.8 fig8-1.eps

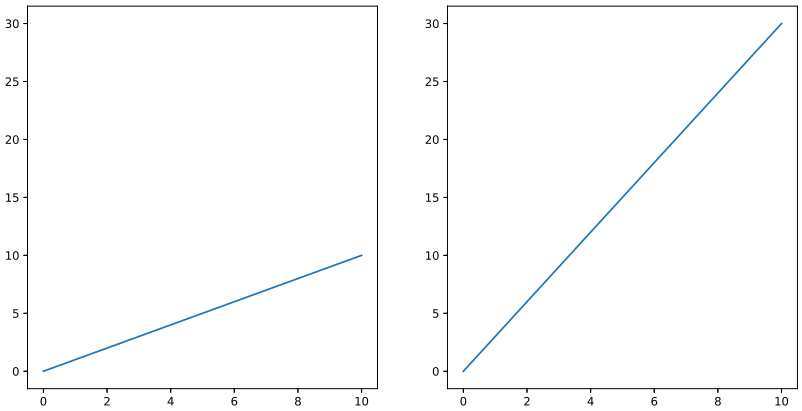

図 2.9 fig8-2.eps

第 3 章

機械学習の概念

3.1　機械学習の定義

Arthur Samuel は，機械学習を以下のとおり定義している．

定義 3.1 (機械学習 (Arthur Samuel(1959)))**．機械学習 (machine learning)** とは，明示的にプログラミングしなくてもコンピュータに学習能力を与える研究分野のことである．

では，学習とは何か．Tom Mitchell は，学習を，タスク T，経験 E，性能指標 P を用いて，以下の通り定義している．

定義 3.2 (学習，タスク T，経験 E，性能指標 P(Tom Mitchell(1998)))**．性能指標 (performance measure)**P で測定される，**タスク (task)**T における性能が**経験 (experience)**E により改善されることを，そのタスク T のクラスおよび性能指標 P に関して経験 E から**学習 (learn)** するという．

問題 3.1. 以下の機械学習の各事例においてタスク T，経験 E，性能指標 P を答えなさい．

1. 将棋プログラムが，自身を相手に数万回もの対局を行い，どのような棋譜が勝つまたは負ける傾向になるかを学習していき，人間よりも将棋が強くなった．
2. 電子メールクライアントが，どの電子メールをスパムとしてフラグを立てるかどうかを判断しようとしている．人間がどの電子メールがスパムかを電子メールクライアントに逐一報告していくことにより，より正確にスパムであるかどうかの判断を行えるようになっていった．
3. 過去の天気データから，将来の天気を予測する．

解答.

1. T は「将棋をさすこと」，E は「自身を相手に数万回もの対局を行なった経験」，P は「次の対戦で勝利する確率」.
2. T は「電子メールをスパムかどうか判断すること」，E は「各電子メールがスパムかどうかの報告内容」，P は「正しくスパムと判断できる確率」.
3. T は「将来の天気を予測すること」，E は「過去の天気データ」，P は「正しく天気を当てられる確率」.

□

3.2 教師あり学習と教師なし学習

機械学習には，教師あり学習と教師なし学習がある．それぞれ以下の通り定義される．

定義 3.3 (教師あり学習)**. 教師あり学習 (supervised learning)** とは，入力に対して正しい出力がわかるデータを用いた機械学習のことである．

定義 3.4 (教師なし学習)**. 教師なし学習 (unsupervised learning)** とは，単にデータが与えられ，そのデータから何らかの構造関係を導出する機械学習のことである．

教師あり学習は，回帰問題と分類問題に分けられる．

定義 3.5 (回帰問題)**. 回帰問題 (regression problem)** とは，出力が連続値である教師あり学習のことである．

定義 3.6 (分類問題)**. 分類問題 (classification problem)** とは，出力が離散値である教師あり学習のことである．

問題 3.2. 以下の機械学習の各事例において教師あり学習か教師なし学習か答えなさい．また，教師あり学習の場合，それが回帰問題であるか分類問題であるかを答えなさい．

1. 大量に在庫がある商品を抱えている．それが 3 ヶ月以内に何個売れるか予測する．
2. あるソフトウェアに使われている各ライセンスについて，それが正規ライセンスか不正ライセンスかを予測する．
3. 毎日何万ものニュースを集めてきて，関連する記事にグループ分けする．
4. ある人物の絵を見て，その人物の年齢を予測する．
5. 電子メールのやりとり履歴から，自動的にどれが密接な友人のグループかを特定する．
6. 2 つのマイクで拾った 2 人の声を解析して分離する (Cocktail Party Problem)．

解答.

1. 教師あり学習の回帰問題.
2. 教師あり学習の分類問題.
3. 教師なし学習.
4. 教師あり学習の回帰問題.
5. 教師なし学習.
6. 教師なし学習.

□

第 4 章

教師あり学習

4.1 トレーニングセットと仮説関数

教師あり学習は，特徴量と目的変数の組のデータを用いて学習する．学習に使用するデータをトレーニングセットという．

定義 4.1 (トレーニングセット)**.** $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{m} \subset ((\mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n) \times \mathcal{Y})^m$ を**トレーニングセット (training set)** という．ここで，m は**トレーニングサンプル数 (number of training examples)**，$\boldsymbol{x}^{(i)} \in \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n$ は i 番目の n 個の**入力変数 (input variables)** または**特徴量 (features)** で，$\boldsymbol{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)})^T$ と表す．また，$y^{(i)} \in \mathcal{Y}$ は i 番目の**出力変数 (output variable)** または**目的変数 (target variable)** である．また，トレーニングセットの i 番目の要素 $(\boldsymbol{x}^{(i)}, y^{(i)}) \in (\mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n) \times \mathcal{Y}$ を**トレーニングサンプル (training example)** という．また，入力変数のとる空間 $\mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n$ を**入力変数空間 (space of input values)**，出力変数のとる空間 $\mathcal{Y}$ を**出力変数空間 (space of output values)** という．

問題 4.1. 次のトレーニングセットにおいて，$x_3^{(4)}, y^{(2)}$ を答えよ．

i	$x_1^{(i)}$	$x_2^{(i)}$	$x_3^{(i)}$	$x_4^{(i)}$	$y^{(i)}$
1	2104	5	1	45	460
2	1416	3	2	40	232
3	1534	3	2	30	315
4	852	2	1	36	178

解答. $x_3^{(4)} = 1,\ y^{(2)} = 232.$ □

教師あり学習を使って解きたいタスク T は，入力変数から出力変数を予測することであるが，それは言い換えると入力変数を引数として出力変数を出力する写像を設定することである．この写像を仮説関数という．仮説関数を以下で定義する．

定義 4.2 (仮説関数)**.** 入力変数 $\boldsymbol{x}$ から出力変数 y への写像 $h : \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n \to \mathcal{Y}$，すなわち $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ を**仮説関数 (hypothesis function)** という．ここで，$\boldsymbol{\theta}$ は仮説関数の**パラメータ**である (パラメータは複数あることがほとんどなのでベクトルとしている)．

すなわち，教師あり学習とは，仮説関数を設定し，トレーニングセットを用いて仮説関数の最適なパラメータを決定することといえる．最適なパラメータを決定できれば，そのパラメータをセットした仮説関

数にデータを流し込むことで，そのデータに対する予測値を計算できる．

4.2 線形回帰

仮説関数は自らで与える必要があるが，仮説関数の形によって，教師あり学習に特別な名前がつくものがある．例えば，仮説関数を線形関数とし，出力変数空間を $\mathcal{Y} = \mathbb{R}$ としたときは線形回帰という．

定義 4.3 (線形回帰)**.** トレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{m} \subset ((\mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n) \times \mathcal{Y})^m$ について，$\mathcal{Y} = \mathbb{R}$ であり，かつ仮説関数が式 (4.1) である教師あり学習を，特に**線形回帰 (linear regression)** という．ここで，特徴量 $\boldsymbol{x}^{(i)}$ は，常に 1 の値をとるような特徴量 $x_0^{(i)} = 1$ を付して $\boldsymbol{x}^{(i)} = (x_0^{(i)}, x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)})^T \in 1 \times \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n$ と置き直すこととする．また，$\boldsymbol{\theta} = (\theta_0, \theta_1, \ldots, \theta_n)^T \in \mathbb{R}^{n+1}$ とする．

$$\begin{aligned} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) &= \theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \cdots + \theta_n x_n^{(i)} \\ &= \sum_{j=0}^{n} \theta_j x_j^{(i)} \\ &= \boldsymbol{\theta}^T \boldsymbol{x}^{(i)} \end{aligned} \tag{4.1}$$

問題 4.2. ある大学生について，1 年次の成績で優をとった個数から 2 年次にいくつ優をとるのか予測したい．そこで，何人かの大学生の 1 年次の成績の優の個数 x と 2 年次の成績の優の個数 y を集めた．その結果が次表である．このとき，次の問いに答えよ．

x	y
3	4
2	1
4	3
0	1

1. m はいくつか．
2. 仮説関数として $h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x$ を設定し，本トレーニングセットを用いて線形回帰を行なった結果，パラメータは $\theta_0 = -1$, $\theta_1 = 2$ となった．このとき，1 年次の優の個数が 6 だった大学生の 2 年次の優の個数を予測せよ．

解答.

1. $m = 4$.
2. $h_{\boldsymbol{\theta}}(x) = -1 + 2x$ なので，$h_{\boldsymbol{\theta}}(6) = -1 + 2 \cdot 6 = 11$.

□

4.2.1 目的関数

さて，仮説関数のパラメータをどう決めるかという問題がある．パラメータをでたらめに与えても良い予測値を返さないので，経験 E のトレーニングセットを使って，性能指標 P を高めるように仮説関数のパラメータをアップデートしていくことが必要になる．この手順を学習アルゴリズムといい，性能指標を測る関数を目的関数という．

定義 4.4 (学習アルゴリズム，目的関数)**.** 性能指標を測る関数 $J(\boldsymbol{\theta})$ を**目的関数**または**コスト関数 (cost function)** といい，この目的関数で測った性能指標が良くなるように仮説関数 (のパラメータ $\boldsymbol{\theta}$) をアップデートしていく手順を**学習アルゴリズム (learning algorithm)** という．

回帰問題における性能指標としては，各トレーニングサンプル $(\boldsymbol{x}^{(i)}, y^{(i)})$ での仮説関数 $h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)})$ と $y^{(i)}$ の二乗誤差平均が考えられる．この二乗誤差平均を計算する目的関数を最小二乗誤差関数という．

定義 4.5 (最小二乗誤差関数)**.** トレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{m} \subset ((1 \times \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_n) \times \mathcal{Y})^m$ について，式 (4.2) で表される目的関数 $J(\boldsymbol{\theta})$ を**最小二乗誤差関数 (squared error function)** という．

$$J(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} (h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 \tag{4.2}$$

注意 4.1. m ではなく $2m$ で割っているのは，微分したときに出てくる 2 が消えるようにしているからである．m でも特段の問題はない (同じ結果が得られる)．

問題 4.3. あるトレーニングセット $\{(x^{(i)}, y^{(i)})\}_{i=1}^{3}$ をプロットしたところ以下の散布図となった (簡単のため，特徴量 x に $x_0 = 1$ である特徴量は追加していない)．仮説関数を $h_\theta(x) = \theta x$，目的関数 $J(\theta)$ を最小二乗誤差関数としたとき，$J(0)$ を求めよ．

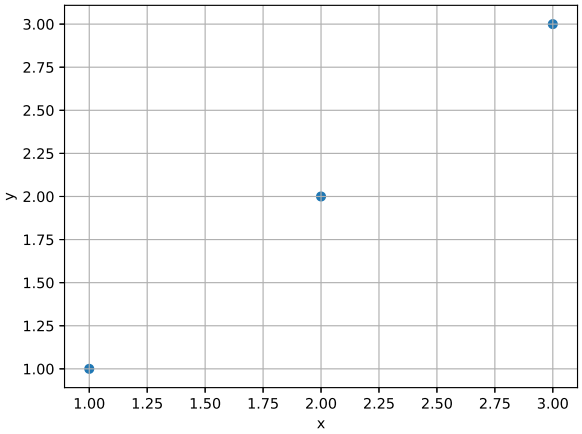

図 4.1　`lrfig1.eps`

解答. 目的関数 $J(\theta)$ は次式となる．

$$\begin{aligned} J(\theta) &= \frac{1}{6} \sum_{i=1}^{3} (h_\theta(x^{(i)}) - y^{(i)})^2 \\ &= \frac{1}{6} \sum_{i=1}^{3} (\theta x^{(i)} - y^{(i)})^2 \end{aligned}$$

$\theta = 0$ を代入し，散布図から $y^{(i)}$ を読み取ると，

$$\begin{aligned} J(0) &= \frac{1}{6} \sum_{i=1}^{3} (-y^{(i)})^2 \\ &= \frac{1}{6} ((-1)^2 + (-2)^2 + (-3)^2) \\ &= \frac{14}{6} \end{aligned}$$

□

Code 4.1 (lrfig1.eps 作成プログラム (lrfig1.py))**.**

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

fig = plt.figure()
ax = fig.add_subplot(1,1,1)

x = np.array([1,2,3])
y = x

ax.scatter(x, y)

ax.grid()
ax.set_xlabel('x')
ax.set_ylabel('y')

fig.savefig('lrfig1.eps')
```

4.2.2 最急降下法

「性能指標が良くなるように仮説関数のパラメータ $\boldsymbol{\theta}$ をアップデートしていく」とはどういうことか．目的関数が最小二乗誤差関数の場合，その最小二乗誤差がどんどん小さくなっていくことが，性能指標が良くなっていくといえる．すなわち，目的関数を最小にするパラメータ $\boldsymbol{\theta}$ を見つければよい．それを見つけるための手法が学習アルゴリズムである．

学習アルゴリズムの中で一般的なものとして最急降下法がある．最急降下法とは，目的関数 $J(\boldsymbol{\theta})$ のグラフ上に適当に点を打ち (すなわちパラメータ $\boldsymbol{\theta}$ として適当に初期値を決め)，その点からあたりを見渡してもっとも勾配が急な方向に一定程度進み，進んだ後の点からまたあたりを見渡してもっとも勾配が急な方向に一定程度進み... を繰り返して，どこを見渡しても勾配がないような点を探す方法である．

最急降下法を行うためには，関数上のある点について勾配が急な方向はどの方向であるかを計算しなければならない．勾配が最も急な方向を向くベクトルを勾配ベクトルといい，以下で定義される．

定義 4.6 (勾配ベクトル)**.** k 次元ベクトル $\boldsymbol{\theta} = (\theta_1, \theta_2, \ldots, \theta_k)^T$ からスカラー値に写る関数 $f(\boldsymbol{\theta})$，すなわち $f : \boldsymbol{\theta} \in \mathbb{R}^k \to \mathbb{R}$ である関数 $f(\boldsymbol{\theta})$ において，**勾配ベクトル** $\nabla_{\boldsymbol{\theta}} f = \frac{\partial f}{\partial \boldsymbol{\theta}}$ は次式で定義される．

$$\nabla_{\boldsymbol{\theta}} f = \frac{\partial f}{\partial \boldsymbol{\theta}} = \begin{bmatrix} \frac{\partial f}{\partial \theta_1} \\ \frac{\partial f}{\partial \theta_2} \\ \vdots \\ \frac{\partial f}{\partial \theta_k} \end{bmatrix} \tag{4.3}$$

問題 4.4. $f : \mathbb{R}^2 \to \mathbb{R}$ の関数 $f(x, y)$ のある点 $\mathrm{P}(x, y)$ における最大勾配方向が勾配ベクトルの方向と同方向であることを示せ．

解答. 勾配とは，関数 f の変化度合いであり，勾配が最大ということは，関数の変化度合いが最も大きいということである．点 $\mathrm{P}(x, y)$ と，そこから微小量 Δx，Δy だけ動かした点 $\mathrm{Q}(x + \Delta x, y + \Delta y)$ においてそれぞれ関数値を求めて差をとったものを変化度合い Δf とすると，$\overrightarrow{PQ} = \overrightarrow{OQ} - \overrightarrow{OP} = (\Delta x, \Delta y)$ に

注意して，以下の通り変形できる．

$$\begin{aligned}\Delta f &= f(x+\Delta x, y+\Delta y) - f(x,y)\\ &= f(x+\Delta x, y+\Delta y) - f(x, y+\Delta y) + f(x, y+\Delta y) - f(x,y)\\ &= \frac{f(x+\Delta x, y+\Delta y) - f(x, y+\Delta y)}{\Delta x}\Delta x + \frac{f(x, y+\Delta y) - f(x,y)}{\Delta y}\Delta y\\ &\fallingdotseq \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y\\ &= \nabla f \cdot (\Delta x, \Delta y)\\ &= \nabla f \cdot \overrightarrow{PQ}\end{aligned}$$

すなわちこれは，関数の変化度合いは勾配ベクトルと点 P から点 Q への方向ベクトル，すなわち微小量を動かした方向のベクトルの内積となっている．ここで，角度の定義より，

$$\Delta f = ||\nabla f|| ||\overrightarrow{PQ}|| \cos\theta$$

となる．ここで，θ は，∇f と $\overrightarrow{PQ}$ のなす角である．関数の変化度合い Δf が最大となるのは，$\cos\theta = 1$，すなわち $\theta = 0$ となる場合である．これはつまり ∇f と $\overrightarrow{PQ}$ が同じ方向を向いているときに関数の変化度合い Δf が最大となるということである．以上より，点 P から関数の変化度合い Δf が最大となるように進むためには (点 Q をとるためには)，勾配ベクトルの方向に進めばよいということである．　□

これで勾配が最も急な方向が勾配ベクトル方向であることがわかったので，それを用いて最急降下法を以下の通り定義する．

定義 4.7 (最急降下法)**.** 関数 $J(\boldsymbol{\theta})$ を最小とする $\boldsymbol{\theta}$ を次の手順で見つけるアルゴリズムを，**最急降下法 (gradient descent algorithm)** という．ここで，$\alpha(>0)$ を**学習率**といい，勾配が最大の方向にどの程度移動させるかの強さを表す．

Algorithm 1 最急降下法

1: トレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^m$，仮説関数 $h_{\boldsymbol{\theta}}(\boldsymbol{x})$，目的関数 $J(\boldsymbol{\theta})$ を用意
2: $\alpha \leftarrow$ 初期値
3: $\boldsymbol{\theta} \leftarrow$ 初期値
4: **while** $\boldsymbol{\theta}$ が収束または有限回繰り返し **do**
5: 　$\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^m$ を代入して $J(\boldsymbol{\theta})$ を計算
6: 　$\nabla_{\boldsymbol{\theta}} J$ を計算
7: 　$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha\nabla_{\boldsymbol{\theta}} J$　▷ パラメータ $\theta_0, \theta_1 \ldots$ は同タイミングで更新
8: **end while**

問題 4.5. $x_0^{(i)} = 1$ も含め特徴量が $n+1$ 個のトレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^m$ の線形回帰において，目的関数 $J(\boldsymbol{\theta})$ を最小二乗誤差関数としたとき，$\nabla_{\boldsymbol{\theta}} J$ を計算せよ．

解答. $\nabla_{\boldsymbol{\theta}} J$ の j 番目の要素 $\frac{\partial J}{\partial \theta_j}$ を計算すると以下となる.

$$\begin{aligned}\frac{\partial J}{\partial \theta_j} &= \frac{1}{2m}\sum_{i=1}^{m}\frac{\partial}{\partial \theta_j}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 \\ &= \frac{1}{2m}\sum_{i=1}^{m} 2(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})\frac{\partial}{\partial \theta_j} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) \\ &= \frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)}\end{aligned}$$

$j = 0$ のときは $\frac{\partial}{\partial \theta_0} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) = 1$ であることに注意してまとめると,

$$\nabla_{\boldsymbol{\theta}} J = \frac{1}{m}\begin{bmatrix} \sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)}) \\ \sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_1^{(i)} \\ \sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_2^{(i)} \\ \vdots \\ \sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)} \\ \vdots \\ \sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})x_n^{(i)} \end{bmatrix} \tag{4.4}$$

となる（$\nabla_{\boldsymbol{\theta}} J$ は $n+1$ 次元ベクトルである）. □

4.2.3 デザイン行列

最急降下法アルゴリズムは, 式 (4.4) を用いて $\nabla_{\boldsymbol{\theta}} J$ の各要素に対して逐次計算を行っていけば単純に実装できるが, 各要素の和の計算を各要素に対して行い, 反復していくことは少々ややこしい. 行列計算を容易に行えるプログラミング言語で実装する場合には, できるだけ逐次計算をしないように, 行列計算やベクトル計算を用いて工夫して実装することが簡潔かつバグも少ない. ここでは, デザイン行列を定義し, 行列計算により $\nabla_{\boldsymbol{\theta}} J$ を計算できることを示す.

定義 4.8 (デザイン行列). $x_0^{(i)} = 1$ も含め特徴量が $n+1$ 個のトレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ において, 式 (4.5) で定義する行列 $X \in \mathbb{R}^{m\times(n+1)}$ を**デザイン行列 (design matrix)** という. デザイン行列は, 特徴量 $\boldsymbol{x}^{(i)}$ を転置してサンプル数の分縦に並べたもので表される. ここで, 特徴量として $x_0 = 1$ が加わっていることに注意する.

$$X = \begin{bmatrix} \text{---} & \boldsymbol{x}^{(1)T} & \text{---} \\ \text{---} & \boldsymbol{x}^{(2)T} & \text{---} \\ & \vdots & \\ \text{---} & \boldsymbol{x}^{(m)T} & \text{---} \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix} \tag{4.5}$$

問題 4.6. 次のトレーニングセットにおいて, デザイン行列 X を答えよ.

i	$x_1^{(i)}$	$x_2^{(i)}$	$x_3^{(i)}$	$x_4^{(i)}$	$y^{(i)}$
1	2104	5	1	45	460
2	1416	3	2	40	232
3	1534	3	2	30	315
4	852	2	1	36	178

解答. サンプル数が 4 つで，特徴量が $x_0 = 1$ を含めると 5 つなので，X は 4×5 次元の行列となり，

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix}$$

□

デザイン行列を使うと，予測値である線形回帰の仮説関数のベクトルを簡潔に表すことができ，1 発の線形代数計算で予測値を計算できる．

問題 4.7. $x_0^{(i)} = 1$ も含め特徴量が $n+1$ 個のトレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^m$ における線形回帰問題を考える．このとき，$(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)}), h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(2)}), \ldots, h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(m)}))^T$ を X，$\boldsymbol{\theta}$ を用いて表せ．

解答. 式 (1.25) より，以下となる．

$$\begin{bmatrix} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)}) \\ h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(2)}) \\ \vdots \\ h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(m)}) \end{bmatrix} = \begin{bmatrix} \boldsymbol{\theta}^T\boldsymbol{x}^{(1)} \\ \boldsymbol{\theta}^T\boldsymbol{x}^{(2)} \\ \vdots \\ \boldsymbol{\theta}^T\boldsymbol{x}^{(m)} \end{bmatrix} = \begin{bmatrix} {\boldsymbol{x}^{(1)}}^T\boldsymbol{\theta} \\ {\boldsymbol{x}^{(2)}}^T\boldsymbol{\theta} \\ \vdots \\ {\boldsymbol{x}^{(m)}}^T\boldsymbol{\theta} \end{bmatrix} = \begin{bmatrix} \text{---} & {\boldsymbol{x}^{(1)}}^T & \text{---} \\ \text{---} & {\boldsymbol{x}^{(2)}}^T & \text{---} \\ & \vdots & \\ \text{---} & {\boldsymbol{x}^{(m)}}^T & \text{---} \end{bmatrix} \begin{bmatrix} | \\ \boldsymbol{\theta} \\ | \end{bmatrix} = X\boldsymbol{\theta} \tag{4.6}$$

□

この結果を用いることで，目的関数を最小二乗誤差関数とした線形回帰問題において，目的関数をより簡潔に表すことができる．

問題 4.8. $x_0^{(i)} = 1$ も含め特徴量が $n+1$ 個のトレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^m$ における線形回帰問題において，目的関数 $J(\boldsymbol{\theta})$ を最小二乗誤差関数とする．このとき，$J(\boldsymbol{\theta})$ を X，$\boldsymbol{\theta}$，$\boldsymbol{y}$ を用いて表せ．ここで，$\boldsymbol{y} = (y^{(1)}, y^{(2)}, \ldots, y^{(m)})^T$ とする．

解答.

$$\begin{aligned} J(\boldsymbol{\theta}) &= \frac{1}{2m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) - y^{(i)})^2 \\ &= \frac{1}{2m}\begin{bmatrix} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)}) - y^{(1)} & h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(2)}) - y^{(2)} & \cdots & h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(m)}) - y^{(m)} \end{bmatrix} \begin{bmatrix} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)}) - y^{(1)} \\ h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(2)}) - y^{(2)} \\ \vdots \\ h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(m)}) - y^{(m)} \end{bmatrix} \end{aligned}$$

となるが，ここで，

$$\begin{bmatrix} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)}) - y^{(1)} \\ h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(2)}) - y^{(2)} \\ \vdots \\ h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(m)}) - y^{(m)} \end{bmatrix} = \begin{bmatrix} h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)}) \\ h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(2)}) \\ \vdots \\ h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(m)}) \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} = X\boldsymbol{\theta} - \boldsymbol{y}$$

より，

$$J(\boldsymbol{\theta}) = \frac{1}{2m}(X\boldsymbol{\theta} - \boldsymbol{y})^T(X\boldsymbol{\theta} - \boldsymbol{y}) \tag{4.7}$$

□

これで目的関数を簡潔に書けたので，最後にそれを微分して $\nabla_{\boldsymbol{\theta}} J$ を求める．ここで，線形形式，二次形式の勾配ベクトルを使用するので事前に述べておく．

定理 4.1 (線形形式，二次形式の勾配ベクトル)**.** n 次元列ベクトル $\boldsymbol{x}$，$\boldsymbol{a}$，$n \times n$ 行列 A に対して，次式が成り立つ．

$$\nabla_{\boldsymbol{x}}(\boldsymbol{a}^T\boldsymbol{x}) = \nabla_{\boldsymbol{x}}(\boldsymbol{x}^T\boldsymbol{a}) = \boldsymbol{a} \tag{4.8}$$

$$\nabla_{\boldsymbol{x}}(\boldsymbol{x}^T A\boldsymbol{x}) = (A + A^T)\boldsymbol{x} \tag{4.9}$$

証明. 線形形式の勾配ベクトルの証明は略．二次形式の勾配ベクトルは式 (1.28) から計算すると楽である．式 (1.28) において，x_i における偏微分を計算すると以下となる．変形の最後は，$A = \{a_{ij}\}$ の転置行列を $A^T = \{a'_{ij}\}$ としたとき，$a_{ki} = a'_{ik}$ であることを使用している．

$$\begin{aligned}
\frac{\partial}{\partial x_i}(\boldsymbol{x}^T A\boldsymbol{x}) &= \frac{\partial}{\partial x_i}\left(a_{ii}x_i^2 + \sum_{\substack{j=1\\ j\neq i}}^{n} a_{ij}x_i x_j + \sum_{\substack{k=1\\ k\neq i}}^{n} a_{ki}x_k x_i + \sum_{\substack{j,k\\ j\neq i\\ k\neq i}} a_{kj}x_k x_j \right) \\
&= 2a_{ii}x_i + \sum_{\substack{j=1\\ j\neq i}}^{n} a_{ij}x_j + \sum_{\substack{k=1\\ k\neq i}}^{n} a_{ki}x_k \\
&= \sum_{j=1}^{n} a_{ij}x_j + \sum_{k=1}^{n} a_{ki}x_k \\
&= \sum_{j=1}^{n} a_{ij}x_j + \sum_{k=1}^{n} a'_{ik}x_k
\end{aligned}$$

第 1 項について，x_1〜x_n についての結果を列ベクトルとして並べると，式 (1.26) の形が現れ，その結果は $A\boldsymbol{x}$ となる．第 2 項についても同様であり，その結果は $A^T\boldsymbol{x}$ となる．以上より，$\nabla_{\boldsymbol{x}}(\boldsymbol{x}^T A\boldsymbol{x}) = A\boldsymbol{x} + A^T\boldsymbol{x} = (A + A^T)\boldsymbol{x}$ である． □

問題 4.9. $x_0^{(i)} = 1$ も含め特徴量が $n+1$ 個のトレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^m$ における線形回帰問題において，目的関数 $J(\boldsymbol{\theta})$ を最小二乗誤差関数とする．このとき，$\nabla_{\boldsymbol{\theta}} J$ を m，X，$\boldsymbol{\theta}$，$\boldsymbol{y}$ で表せ．

解答. X^TX は対称行列であることに注意すると，

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J &= \nabla_{\boldsymbol{\theta}} \left(\frac{1}{2m} (X\boldsymbol{\theta} - \boldsymbol{y})^T (X\boldsymbol{\theta} - \boldsymbol{y}) \right) \\
&= \nabla_{\boldsymbol{\theta}} \left(\frac{1}{2m} \left((X\boldsymbol{\theta})^T (X\boldsymbol{\theta}) - (X\boldsymbol{\theta})^T \boldsymbol{y} - \boldsymbol{y}^T (X\boldsymbol{\theta}) + \boldsymbol{y}^T \boldsymbol{y} \right) \right) \\
&= \nabla_{\boldsymbol{\theta}} \left(\frac{1}{2m} \left((X\boldsymbol{\theta})^T (X\boldsymbol{\theta}) - 2(X\boldsymbol{\theta})^T \boldsymbol{y} + \boldsymbol{y}^T \boldsymbol{y} \right) \right) \\
&= \nabla_{\boldsymbol{\theta}} \left(\frac{1}{2m} \left(\boldsymbol{\theta}^T X^T X\boldsymbol{\theta} - 2\boldsymbol{\theta}^T X^T \boldsymbol{y} + \boldsymbol{y}^T \boldsymbol{y} \right) \right) \\
&= \frac{1}{2m} \left(\nabla_{\boldsymbol{\theta}} (\boldsymbol{\theta}^T X^T X\boldsymbol{\theta}) - 2\nabla_{\boldsymbol{\theta}} (\boldsymbol{\theta}^T X^T \boldsymbol{y}) + \nabla_{\boldsymbol{\theta}} (\boldsymbol{y}^T \boldsymbol{y}) \right) \\
&= \frac{1}{2m} \left(2X^T X\boldsymbol{\theta} - 2X^T \boldsymbol{y} \right) \\
&= \frac{1}{m} \left(X^T X\boldsymbol{\theta} - X^T \boldsymbol{y} \right) \qquad (4.10)
\end{aligned}$$

□

4.2.4　Python による実装

以上で線形回帰で必要なパーツは揃った．これらのパーツを実際に Python にて実装していきながら，線形回帰とはどういうものかを掴んでいく．

問題 4.10. `ex1data1.txt`(2 列，カンマ区切り，列名ヘッダーなし，1 列目は population of city in 10,000s，2 列目は profit in $10,000s であるデータ．取得元は [1]) を散布図でプロットせよ．

解答.

Code 4.2 (`lr5.py`)**.**

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.grid()
ax.scatter(df['population'].values, df['profit'].values)
ax.set_xlabel('Population␣of␣City␣in␣10,000s')
ax.set_ylabel('Profit␣in␣$10,000s')

fig.savefig('lrfig2.eps')
```

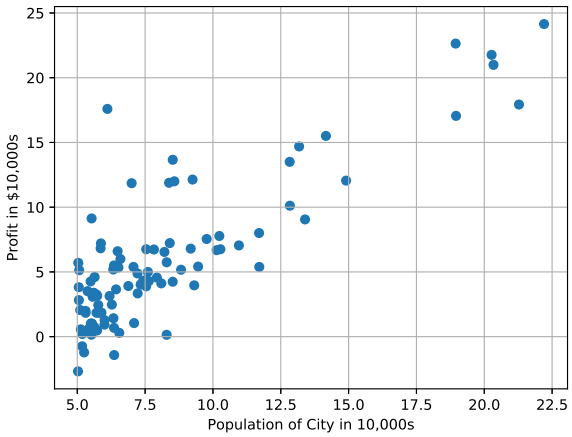

図 4.2 lrfig2.eps

問題 4.11. (問題 4.10 の続き) ex1data1.txt から，デザイン行列 X と目的変数 $\boldsymbol{y}$ をそれぞれ<2d_ndarray>で作成せよ．ここで，目的変数は profit in $10,000s とし，デザイン行列には $x_0^{(i)} = 1$ の項も付け加えよ．

解答. 表形式データは，行方向にデータが並び，列方向が特徴量が並ぶものが多いので，デザイン行列は表形式データそのものとして与えてあげればよい．また，目的変数は 1 列のデータであるが，<DataFrame>から 1 列だけ読み込む場合は<1d_ndarray>になってしまうので，列ベクトルとしての<2d_ndarray>に変換する．なお，データ数が多いので，以下のコードにおいてはデザイン行列や目的変数は最初の 5 行分のみ表示するようにしている．

Code 4.3 (lr1.py).

```
import pandas as pd
import numpy as np

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
y = np.c_[y]

print('X=\n{}\n......\ntype={},shape={}'.format(X[:5],type(X),X.shape))
print('y=\n{}\n......\ntype={},shape={}'.format(y[:5],type(y),y.shape))
```

```
X=
[[1.  6.1101]
 [1.  5.5277]
 [1.  8.5186]
 [1.  7.0032]
 [1.  5.8598]]
......
type=<class 'numpy.ndarray'>,shape=(97, 2)
y=
```

```
[[17.592 ]
 [ 9.1302]
 [13.662 ]
 [11.854 ]
 [ 6.8233]]
......
type=<class 'numpy.ndarray'>,shape=(97, 1)
```

□

問題 4.12. (問題 4.11 の続き) 仮説関数のパラメータを $\boldsymbol{\theta} = (1, -2)^T$ とする．このとき，問題 4.11 で作成したデザイン行列 X に対して予測値である仮説関数ベクトル $(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(1)}), h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(2)}), \ldots, h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(m)}))^T$ を計算し，問題 4.11 で作成した目的変数 $\boldsymbol{y}$ と並べて表示せよ．なお，仮説関数は関数として定義せよ．

解答. 式 (4.6) を実装すれば良い．

Code 4.4 (lr2.py)**.**

```
import pandas as pd
import numpy as np

def hypo_function(X, theta):
    y_pred = X.dot(theta)
    return y_pred

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
y = np.c_[y]

THETA = np.array([[1],[-2]])

y_pred = hypo_function(X, THETA)

y_matrix = np.c_[y, y_pred]
y_df = pd.DataFrame(y_matrix, columns=['y','y_pred'])

print('y_df=\n{}'.format(y_df.head(5)))
```

```
y_df=
         y y_pred
0 17.5920 -11.2202
1 9.1302 -10.0554
2 13.6620 -16.0372
3 11.8540 -13.0064
4 6.8233 -10.7196
```

予測結果より，$\boldsymbol{\theta} = (1, -2)^T$ では全然予測になっていないことがわかる．　□

問題 4.13. (問題 4.12 の続き) 仮説関数のパラメータを $\boldsymbol{\theta} = (1, -2)^T$ とする．このとき，問題 4.11 で作

成したデザイン行列 X と目的変数 $\boldsymbol{y}$ に対して目的関数 $J(\boldsymbol{\theta})$ の値を計算せよ．なお，目的関数 $J(\boldsymbol{\theta})$ は関数として定義せよ．

解答. 式 (4.7) を実装すれば良い．numpy での線形代数計算は，結果がスカラーだとしても<2d_ndarray>のままなので，要素を指定してあげて数値を取り出すことが必要となる．

Code 4.5 (lr3.py)**.**

```
import pandas as pd
import numpy as np


def cost_function(X, y, theta):
    m, n = X.shape
    J = 1 / (2 * m) * (X.dot(theta) - y).T.dot(X.dot(theta) - y)
    return J[0,0]

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
y = np.c_[y]

THETA = np.array([[1],[-2]])

J = cost_function(X, y, THETA)
print('J={}'.format(J))
```

```
J=303.8795839611464
```

目的関数の数値の大きさからも，$\boldsymbol{\theta}$ は最適化できていないことがわかる． □

問題 4.14. (問題 4.13 の続き) 仮説関数のパラメータを $\boldsymbol{\theta} = (1, -2)^T$ とする．このとき，ex1data1.txt の散布図と，問題 4.12 で計算した予測値を線でプロットせよ．

解答.

Code 4.6 (lr6.py)**.**

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

def hypo_function(X, theta):
    y_pred = X.dot(theta)
    return y_pred

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
```

```
THETA = np.array([[1],[-2]])

y_pred = hypo_function(X, THETA)
y_pred = y_pred.flatten()

fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.grid()
ax.scatter(df['population'].values, y)
ax.set_xlabel('Population␣of␣City␣in␣10,000s')
ax.set_ylabel('Profit␣in␣$10,000s')

ax.plot(df['population'].values, y_pred)

fig.savefig('lrfig3.eps')
```

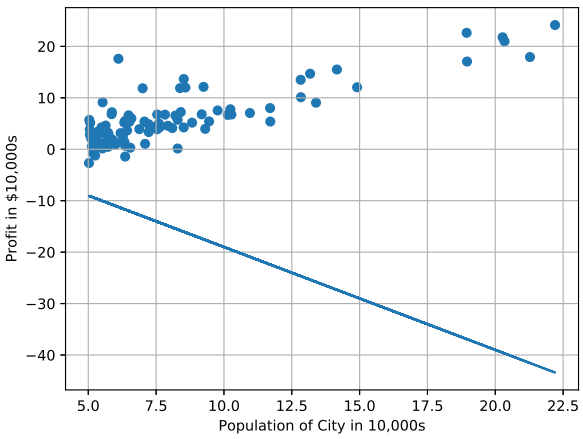

図 4.3　lrfig3.eps

図示すると予測が全然できていないことが一目瞭然. □

問題 4.15. (問題 4.14 の続き) ex1data1.txt のデータを全て用いて目的変数 $\boldsymbol{y}$ を特徴量から予測するモデルを線形回帰で構築するとしたとき，次の問いに答えよ.

1. 線形回帰の予測値である仮説関数 $h_{\boldsymbol{\theta}}(\boldsymbol{x})$ のパラメータ $\boldsymbol{\theta}$ を求めよ．ここで，目的関数 $J(\boldsymbol{\theta})$ は最小二乗誤差関数とし，$\boldsymbol{\theta}$ を求めるときに使用するアルゴリズムは最急降下法とする．また，最急降下法の学習率 α は $\alpha = 0.01$，$\boldsymbol{\theta}$ の初期値は $\boldsymbol{\theta} = (1, -2)^T$ とし，収束判定は行わず 1500 回パラメータの更新を行ったらアルゴリズムは終了するように実装せよ.
2. 求めた $\boldsymbol{\theta}$ に対する目的関数 $J(\boldsymbol{\theta})$ の値を求めよ.
3. 新しく population of city in 10,000s のデータ $x = 7.532$ を得たとする．この x に対して profit in $10,000s がいくらになるか，上で構築した線形回帰モデルを用いて予測せよ.

解答. 最急降下法における $J(\boldsymbol{\theta})$ の勾配ベクトルは式 (4.10) を実装すればよい．その他，仮説関数や目的関数は問題 4.12，問題 4.13 で実装したものをベースに組めばよい.

Code 4.7 (lr4.py).

```
import pandas as pd
import numpy as np

def gradient_descent(X, y, theta, alpha, itera):
    m, n = X.shape
    for i in range(itera):
        J_grad = 1 / m * (X.T.dot(X).dot(theta) - X.T.dot(y))
        theta = theta - alpha * J_grad
    return theta

def cost_function(X, y, theta):
    m, n = X.shape
    J = 1 / (2 * m) * (X.dot(theta) - y).T.dot(X.dot(theta) - y)
    return J[0,0]

def hypo_function(X, theta):
    y_pred = X.dot(theta)
    return y_pred

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
y = np.c_[y]

ALPHA = 0.01
ITERA = 1500
THETA = np.array([[1],[-2]])

theta = gradient_descent(X, y, THETA, ALPHA, ITERA)
print('1:␣theta=\n{}'.format(theta))

J = cost_function(X, y, theta)
print('2:␣J={}'.format(J))

X_pred = np.array([[1, 7.532]])
y_pred = hypo_function(X_pred, theta)
y_pred = y_pred[0,0]
print('3:␣y_pred={}'.format(y_pred))
```

```
1: theta=
[[-3.55089376]
 [ 1.15838599]]
2: J=4.4878002526614615
3: y_pred=5.174069533566031
```

$\boldsymbol{\theta} = (1, -2)^T$ の場合の $J(\boldsymbol{\theta})$ の値からかなり減少していることがわかる． □

問題 4.16. (問題 4.15 の続き) `ex1data1.txt` の散布図と，問題 4.15 で最適化したパラメータ $\boldsymbol{\theta}$ で `ex1data1.txt` のデータに対する線形回帰の予測値を線でプロットせよ．

解答.

Code 4.8 (lr7.py).

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

def gradient_descent(X, y, theta, alpha, itera):
    m, n = X.shape
    for i in range(itera):
        J_grad = 1 / m * (X.T.dot(X).dot(theta) - X.T.dot(y))
        theta = theta - alpha * J_grad
    return theta

def hypo_function(X, theta):
    y_pred = X.dot(theta)
    return y_pred

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
y = np.c_[y]

ALPHA = 0.01
ITERA = 1500
THETA = np.array([[1],[-2]])

theta = gradient_descent(X, y, THETA, ALPHA, ITERA)
y_pred = hypo_function(X, theta)

fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.grid()
ax.scatter(df['population'].values, y.flatten())
ax.set_xlabel('Population␣of␣City␣in␣10,000s')
ax.set_ylabel('Profit␣in␣$10,000s')
ax.plot(df['population'].values, y_pred.flatten())

fig.savefig('lrfig4.eps')
```

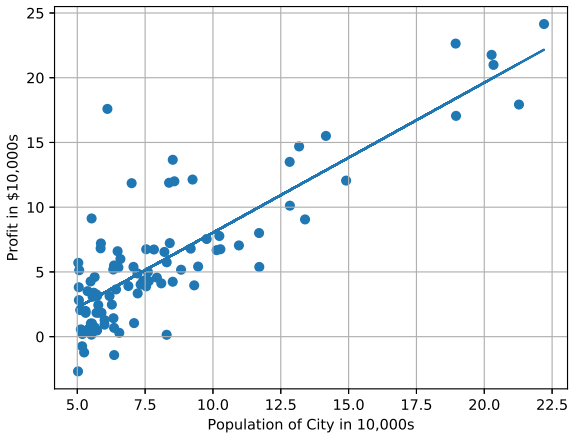

図 4.4 lrfig4.eps

データに対しては良さそうな予測ができていることが確認できる (未知のデータに対しても良い予測かどうかは分からないことに注意). □

問題 4.17. (問題 4.16 の続き) 目的関数 $J(\boldsymbol{\theta})$ について，3 次元の曲面図と 2 次元の等高線図を図示し，等高線図について最適化したパラメータ $\boldsymbol{\theta}$ をプロットして最適化ができていること (最小値付近にいること) を確認せよ．なお，図示の際の定義域は，$-15 \leq \theta_0 \leq 10$, $-1 \leq \theta_1 \leq 4$ とし，等高線図の図示の際は等高線ができるだけ等間隔に並ぶように `levels` を適切に設定せよ．

解答.

Code 4.9 (`lr8.py`).

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

def cost_function(X, y, theta):
    m, n = X.shape
    J = 1 / (2 * m) * (X.dot(theta) - y).T.dot(X.dot(theta) - y)
    return J[0,0]

def gradient_descent(X, y, theta, alpha, itera):
    m, n = X.shape
    for i in range(itera):
        J_grad = 1 / m * (X.T.dot(X).dot(theta) - X.T.dot(y))
        theta = theta - alpha * J_grad
    return theta

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
y = np.c_[y]
```

```
theta0 = np.linspace(-15, 10, 50)
theta1 = np.linspace(-1, 4, 50)
theta_grid0, theta_grid1 = np.meshgrid(theta0, theta1)
m, n = theta_grid0.shape
J_val = np.zeros((m, n))

for i in range(m):
    for j in range(n):
        theta = np.array([[theta_grid0[i,j]], [theta_grid1[i,j]]])
        J_val[i,j] = cost_function(X, y, theta)

fig = plt.figure(figsize=(10,5))

ax1 = fig.add_subplot(1,2,1, projection='3d')
ax1.plot_surface(theta_grid0, theta_grid1, J_val)
ax1.set_xlabel(r'$\theta_0$')
ax1.set_ylabel(r'$\theta_1$')
ax1.set_zlabel(r'$J(\theta)$')

ax2 = fig.add_subplot(1,2,2)
ax2.contour(theta_grid0, theta_grid1, J_val, levels=np.logspace(-2, 3, 20))
ax2.set_xlabel(r'$\theta_0$')
ax2.set_ylabel(r'$\theta_1$')

ALPHA = 0.01
ITERA = 1500
THETA = np.array([[1],[-2]])
theta_opt = gradient_descent(X, y, THETA, ALPHA, ITERA)

ax2.scatter(theta_opt[0,0], theta_opt[1,0])

fig.savefig('lrfig5.eps')
```

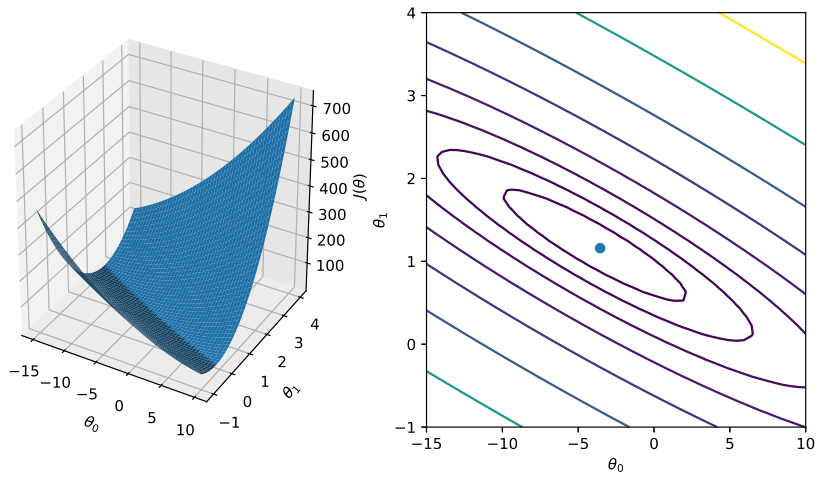

図 4.5　lrfig5.eps

確かに最小値っぽいところに落ち着いているので，最急降下法はうまく動いてくれたようだ． □

問題 4.18. (問題 4.17 の続き) 最急降下法がうまくいっていることを確認するため，繰り返し計算が進むにつれて $\boldsymbol{\theta}$ が最小値付近に移動することや，仮説関数がデータをよく予測できていく様子を描画したい．最急降下法の $\boldsymbol{\theta}$ の更新ごとに $\boldsymbol{\theta}$ をストックするようなプログラムを作り，それを使用して繰り返し回数が 0,1,10,1000 回それぞれにおける仮説関数と $J(\boldsymbol{\theta})$ の等高線図における $\boldsymbol{\theta}$ のプロットを描画せよ．なお，最急降下法の初期値は $\boldsymbol{\theta} = (0, 0)^T$ とすること．

解答.

Code 4.10 (lr9.py)**.**

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

def gradient_descent(X, y, theta, alpha, itera):
    m, n = X.shape
    theta_n, dummy = theta.shape
    theta_hist = np.zeros((itera+1, theta_n))
    theta_hist[0] = theta.flatten()
    for i in range(itera):
        J_grad = 1 / m * (X.T.dot(X).dot(theta) - X.T.dot(y))
        theta = theta - alpha * J_grad
        theta_hist[i+1] = theta.flatten()
    return theta, theta_hist

def cost_function(X, y, theta):
    m, n = X.shape
    J = 1 / (2 * m) * (X.dot(theta) - y).T.dot(X.dot(theta) - y)
    return J[0,0]

def hypo_function(X, theta):
    y_pred = X.dot(theta)
    return y_pred

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
y = np.c_[y]

ALPHA = 0.01
ITERA = 1500
THETA = np.array([[0],[0]])

theta_opt, theta_hist = gradient_descent(X, y, THETA, ALPHA, ITERA)

theta0 = np.linspace(-15, 10, 50)
```

```
theta1 = np.linspace(-1, 4, 50)
theta_grid0, theta_grid1 = np.meshgrid(theta0, theta1)
m, n = theta_grid0.shape
J_val = np.zeros((m, n))

for i in range(m):
    for j in range(n):
        theta = np.array([[theta_grid0[i,j]], [theta_grid1[i,j]]])
        J_val[i,j] = cost_function(X, y, theta)

NUM = [0,1,10,1000]
fig = plt.figure(figsize=(7, 12))

for i in range(len(NUM)):
    theta_tmp = theta_hist[NUM[i]]
    y_pred = hypo_function(X, theta_tmp)

    ax1 = fig.add_subplot(len(NUM), 2, i*2+1)
    ax1.scatter(df['population'].values, df['profit'].values)
    ax1.plot(df['population'].values, y_pred)
    ax1.set_title('iter={}'.format(NUM[i]))

    ax2 = fig.add_subplot(len(NUM), 2, i*2+2)
    ax2.contour(theta_grid0, theta_grid1, J_val, levels=np.logspace(-2, 3, 20))
    ax2.scatter(theta_tmp[0], theta_tmp[1])

fig.subplots_adjust(hspace=0.5)

fig.savefig('lrfig6.eps')
```

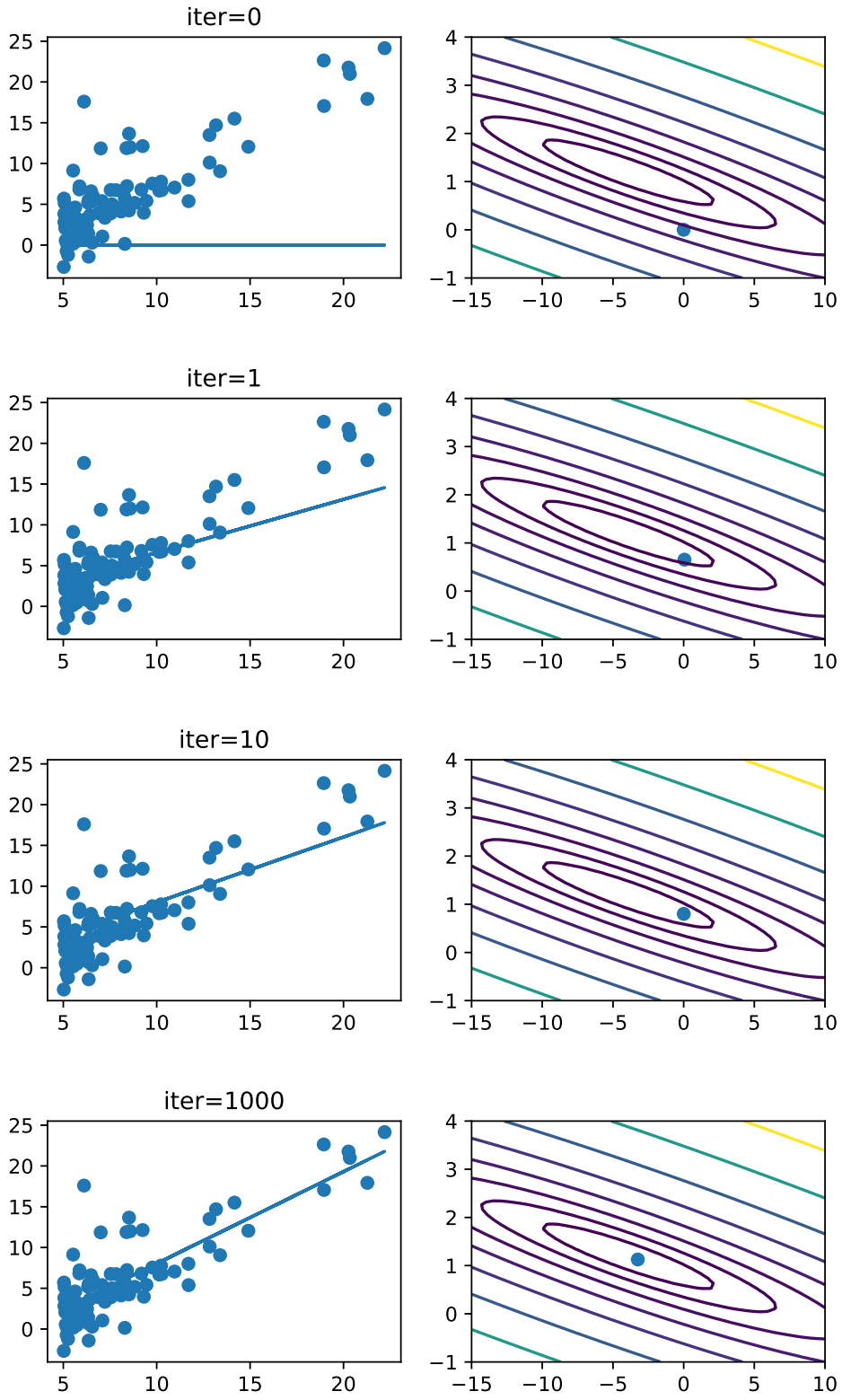

図 4.6 lrfig6.eps

最初の 1 回の更新でかなり学習が進んでいることがわかる．　□

4.2.5　デバッグ

さて，これまで最急降下法の学習率は $\alpha = 0.01$ 固定でずっと行ってきた．この学習率を別の数値にした場合，最急降下法はどういう挙動を示すのか知りたいとする．最急降下法は目的関数 $J(\boldsymbol{\theta})$ の数値をどんどん小さくするためのアルゴリズムであるから，最急降下法がうまく収束しているかを確認する手段としては，ループ 1 回ごとに $J(\boldsymbol{\theta})$ の値をプロットしていき，値が順調に減少していっているかどうかをみるという方法がある．これをデバッグという．

定義 4.9 (デバッグ)**.** 最急降下法において，横軸に繰り返し回数 (number of iterations)，縦軸に目的関数値 $J(\boldsymbol{\theta})$ をとり図示することを**デバッグ (debugging)** という．

問題 4.19. (問題 4.18 の続き) 学習率 $\alpha = 0.00001, 0.0001, 0.024324$ の 3 パターンそれぞれで最急降下法のデバッグを行い，目的関数 $J(\boldsymbol{\theta})$ の変化具合を図示することで比較し，最も適している学習率は 3 つのうちどれか答えなさい．なお，最急降下法の初期値は $\boldsymbol{\theta} = (1, -2)^T$ とすること．

解答.

Code 4.11 (lr10.py)**.**

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

def cost_function(X, y, theta):
    m, n = X.shape
    J = 1 / (2 * m) * (X.dot(theta) - y).T.dot(X.dot(theta) - y)
    return J[0,0]

def gradient_descent(X, y, theta, alpha, itera):
    m, n = X.shape
    theta_n, dummy = theta.shape
    theta_hist = np.zeros((itera+1, theta_n))
    theta_hist[0] = theta.flatten()
    J_hist = np.zeros(itera+1)
    J_hist[0] = cost_function(X, y, theta)
    for i in range(itera):
        J_grad = 1 / m * (X.T.dot(X).dot(theta) - X.T.dot(y))
        theta = theta - alpha * J_grad
        J_hist[i+1] = cost_function(X, y, theta)
    return theta, theta_hist, J_hist

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
y = np.c_[y]
```

```
ITERA = 1500
ALPHA = [0.00001, 0.0001, 0.024324]
THETA = np.array([[0],[0]])

fig = plt.figure(figsize=(12, 4))
axs = list()
flag = 0

for i in range(len(ALPHA)):
    theta_opt, theta_hist, J_hist = gradient_descent(X, y, THETA, ALPHA[i], ITERA)
    if flag == 0:
        ax = fig.add_subplot(1, len(ALPHA), i+1)
        flag = 1
    else:
        ax = fig.add_subplot(1, len(ALPHA), i+1, sharey=axs[0])
    ax.plot(np.arange(ITERA+1), J_hist)
    ax.set_title('alpha={:f}'.format(ALPHA[i]))
    ax.set_xlabel('iter')
    ax.set_ylabel(r'$J(\theta)$')
    axs.append(ax)
fig.subplots_adjust(wspace=0.25)

fig.savefig('lrfig7.eps')
```

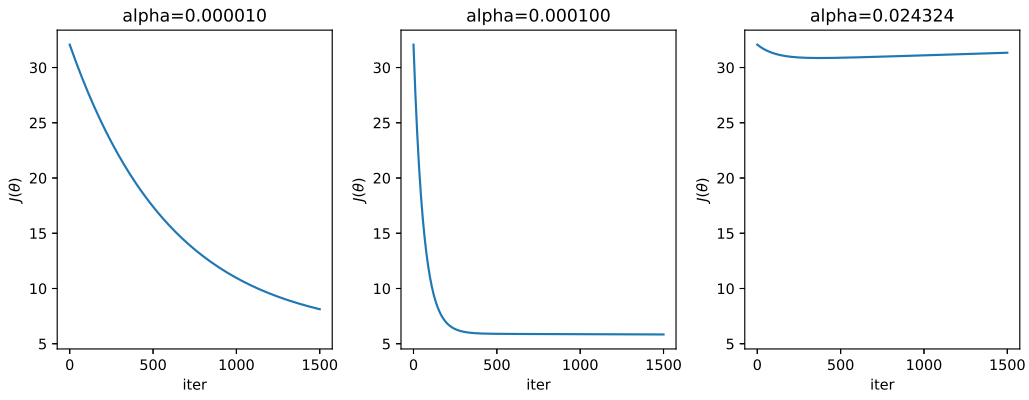

図 4.7 lrfig7.eps

最も適している学習率は，図より $\alpha = 0.0001$ である．学習率とは，パラメータ更新の強さを表すので，弱いとなかなか最小にたどり着かず，かといって強すぎると最小値を超えてパラメータを更新してしまうので，いつまでたっても最小にはたどり着かず，むしろどんどん遠ざかることなる． □

4.2.6 正規方程式

最急降下法で収束して得られた $\boldsymbol{\theta}$ は，目的関数 $J(\boldsymbol{\theta})$ の最小値である点であるためどこを見渡しても勾配がない状態となっている．これは，言い換えると目的関数 $J(\boldsymbol{\theta})$ の勾配ベクトルがゼロベクトルとなる

点 $\boldsymbol{\theta}$ は，方程式 $\nabla_{\boldsymbol{\theta}} J = \mathbf{0}$ の解であることと等しい．この方程式は正規方程式といわれる．

定義 4.10 (正規方程式)**.** 目的関数 $J(\boldsymbol{\theta})$ の回帰問題について，次式を**正規方程式 (normal equation formula)** という．

$$\nabla_{\boldsymbol{\theta}} J = \mathbf{0} \tag{4.11}$$

問題 4.20. 特徴量が n であるトレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ における線形回帰問題において，目的関数 $J(\boldsymbol{\theta})$ を最小二乗誤差関数とする．このとき，$\nabla_{\boldsymbol{\theta}} J = \mathbf{0}$ を簡単に表せ．

解答. 式 (4.10) より，

$$X^T X \boldsymbol{\theta} = X^T \boldsymbol{y} \tag{4.12}$$

□

これを $\boldsymbol{\theta}$ について解けば最適なパラメータ $\boldsymbol{\theta}$ を得ることができる．ここで，$X^T X$ が正則であれば，逆行列 $(X^T X)^{-1}$ が存在するのでそれを左から掛けることによって解けるが，正則でない場合（非正則，非可逆，特異の場合ともいう），逆行列を持たず，正規方程式は解を持たない（不能）もしくは複数または無数の解（不定）となる．不能や不定だからそれでお手上げ，というわけにはいかないので，「いい感じの」の解を設定したいとする．この「いい感じ」の解は，$X^T X$ の**ムーア・ペンローズ一般逆行列** $(X^T X)^{+}$ を用いて，次式で書ける．

$$\boldsymbol{\theta} = (X^T X)^{+} X^T \boldsymbol{y} \tag{4.13}$$

この議論の詳細は，まだ著者が理解できていないため，今の所は割愛する．

問題 4.21. (問題 4.19 の続き) 正規方程式を解いて最適なパラメータ $\boldsymbol{\theta}$ を求め，そのパラメータでの予測値と，最急降下法による予測値を図示することによって比較せよ．

解答. 式 (4.13) を実装すればよい．

Code 4.12 (`lr11.py`)**.**

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

def cost_function(X, y, theta):
    m, n = X.shape
    J = 1 / (2 * m) * (X.dot(theta) - y).T.dot(X.dot(theta) - y)
    return J[0,0]

def gradient_descent(X, y, theta, alpha, itera):
    m, n = X.shape
    theta_n, dummy = theta.shape
    theta_hist = np.zeros((itera+1, theta_n))
    theta_hist[0] = theta.flatten()
    J_hist = np.zeros(itera+1)
    J_hist[0] = cost_function(X, y, theta)
    for i in range(itera):
        J_grad = 1 / m * (X.T.dot(X).dot(theta) - X.T.dot(y))
```

```
        theta = theta - alpha * J_grad
        J_hist[i+1] = cost_function(X, y, theta)
    return theta, theta_hist, J_hist

def hypo_function(X, theta):
    y_pred = X.dot(theta)
    return y_pred

df = pd.read_csv('../data/ex1data1.txt', names=['population','profit'])

df['all_1'] = 1
X = df[['all_1','population']].values
y = df['profit'].values
y = np.c_[y]

ALPHA = 0.01
ITERA = 1500
THETA = np.array([[1],[-2]])

theta, theta_hist, J_hist = gradient_descent(X, y, THETA, ALPHA, ITERA)
y_pred = hypo_function(X, theta)

theta_ne = np.linalg.pinv(X.T.dot(X)).dot(X.T).dot(y)
y_pred_ne = hypo_function(X, theta_ne)

fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.grid()
ax.scatter(df['population'].values, y.flatten())
ax.set_xlabel('Population␣of␣City␣in␣10,000s')
ax.set_ylabel('Profit␣in␣$10,000s')
ax.plot(df['population'].values, y_pred.flatten(), label='GD')
ax.plot(df['population'].values, y_pred_ne.flatten(), label='NE')
ax.legend()

fig.savefig('lrfig8.eps')
```

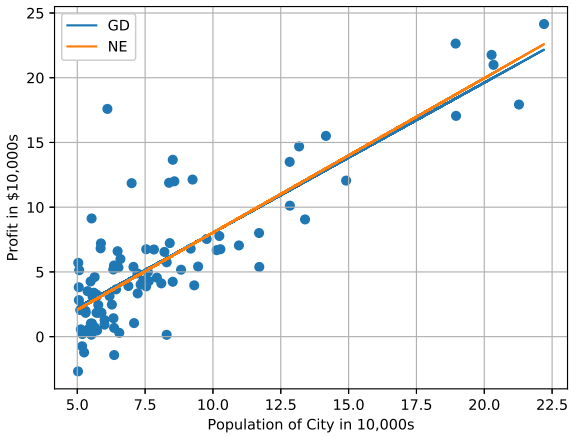

図 4.8　`lrfig8.eps`

□

では，どのような場合に X^TX は非正則なのか．ここは著者がまだ理解できていないが，[1] によれば，以下 2 つの場合を念頭に置いておけばとりあえず良いとのこと．

1. **特徴量が冗長**：例えば，住宅価格の予測についての特徴量で，縦の長さ，横の長さ，面積の 3 つを考えた場合，面積は縦の長さと横の長さですでに捉えられているので，面積という特徴量が冗長である．
2. **データ数より特徴量が多い** $(m \leq n)$：$n = 100$ 個の特徴量を，$m = 10$ サンプルでフィッティングするのは，うまくいくこともあるかもしれないが良いアイデアではない．

教師あり学習の回帰問題について，パラメータを探す方法として最急降下法と正規方程式を解く方法の 2 種類を取り上げた．それぞれの手法のメリットやデメリットについて，[1] によれば以下の通り．

- 最急降下法は，学習率 α を適切に選択する必要があるため，良さげな数値を何度か試行することが必要となってくる場合がある．一方で，正規方程式はその手間がない．
- 最急降下法の場合は，アルゴリズムがちゃんと機能しているか，ちゃんと収束しているかなどを確認しなければならないが，正規方程式はその手間がない．
- 正規方程式の場合，特徴量スケーリングを行う必要はない．
- 正規方程式は，逆行列を求める必要があるが，逆行列を計算するコストが非常に大きい．具体的には，逆行列の計算コストは特徴量 n の 3 乗のオーダーとなる．一方，最急降下法は特徴量 n 本に対する計算を繰り返すだけなので，計算コストは n のオーダーと非常に少ない．つまり，最急降下法は特徴量が数百万個あるような場合でも正しく機能する．だいたい $n = 10000$ が正規方程式ではなく最急降下法を選ぶ目安．

4.2.7　特徴量スケーリング

これまでの実例では，特徴量は 1 つの場合しか取り扱ってこなかった．しかし，特徴量は複数あることの方が普通である．複数であっても，上記の実例と同様に線形代数計算と最急降下法で同じように実装で

きる．ここで，最急降下法を適用するにあたり，複数の特徴量がある場合，各特徴量のとりうる範囲がだいたい同じような範囲にあると，収束速度は速くなる．特徴量を変換してとりうる範囲の調整を行うことを，特徴量スケーリングという．

定義 4.11 (特徴量スケーリング)**.** トレーニングセット $\{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{m}$ について，$\boldsymbol{x}^{(i)}$ の各特徴量のとりうる範囲を同じような範囲に変換することを**特徴量スケーリング (feature scalling)** という．特に，特徴量 x_j の平均 μ_j と標準偏差 s_j を用いて $\frac{x_j-\mu_j}{s_j}$ のようにスケーリングすることを，**平均標準化 (mean normalization)** という．

注意 4.2. 特徴量スケーリングを行うと $\boldsymbol{x}^{(i)} \to \boldsymbol{x}'^{(i)}$ と変換されるが，その場合は変換後のトレーニングセット $\{(\boldsymbol{x}'^{(i)}, y^{(i)})\}_{i=1}^{m}$ に対して線形回帰を行う (パラメータ $\boldsymbol{\theta}$ を定める) ことになるため，$y^{(i)}$ の予測値である仮説関数は $h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) = \boldsymbol{\theta}^T\boldsymbol{x}^{(i)}$ ではなく，$h_{\boldsymbol{\theta}}(\boldsymbol{x}'^{(i)}) = \boldsymbol{\theta}^T\boldsymbol{x}'^{(i)}$ の形となる．よって，トレーニングセットにはない新しいデータ $\boldsymbol{x}^{(m+1)}$ が得られた場合，その予測値を計算するためには新しいデータに対しても特徴量スケーリングを施し $\boldsymbol{x}'^{(m+1)}$ と変換した上で仮説関数 $h_{\boldsymbol{\theta}}(\boldsymbol{x}'^{(m+1)}) = \boldsymbol{\theta}^T\boldsymbol{x}'^{(m+1)}$ を計算し予測することになる．よって，特徴量スケーリングに用いた平均値や標準偏差等の数値や，その変換方法は学習が終わっても保持しておく必要がある．

問題 4.22. データ `ex1data2.txt` は，オレゴン州ポートランドの住宅価格のデータ (最初の列は住宅の面積 (単位は square feet)，2 番目の列は寝室の数，3 番目の列は住宅価格) である．`ex1data2.txt` を用いて，オレゴン州ポートランドの住宅価格の予測モデルを線形回帰を用いて構築したい．このとき，次の問いに答えなさい．

1. データ `ex1data2.txt` をそのまま使用して学習し，最適なパラメータ $\boldsymbol{\theta}$ を求め，新しいデータ $\boldsymbol{x}^{(m+1)}$: 住宅面積 $= 2500$, 寝室の数 $= 3$ に対する住宅価格の予測値を求めよ．ここで，学習率はデバッグをしながら自身で定めよ．また，繰り返し回数は 50 回で打ち切り，$\boldsymbol{\theta} = \boldsymbol{0}$ を初期値とする．
2. データ `ex1data2.txt` に特徴量スケーリングを施して学習し，最適なパラメータ $\boldsymbol{\theta}$ を求め，新しいデータ $\boldsymbol{x}^{(m+1)}$: 住宅面積 $= 2500$, 寝室の数 $= 3$ に対する住宅価格の予測値を求めよ．ここで，学習率はデバッグをしながら自身で定めよ．また，繰り返し回数は 50 回で打ち切り，$\boldsymbol{\theta} = \boldsymbol{0}$ を初期値とする．
3. 特徴量スケーリングによりどう変わったのか確認せよ．

解答.

1. これまで組んできたプログラムは特徴量が 2 つ以上の場合でも問題なく動くように書いてきたので，特段新しいことは必要としない．α を変えながらひらすらデバッグをしていい感じに収束していくようなパターンを見つけていく．その結果，$\alpha = 0.00000003$ の場合がいい感じであった．

Code 4.13 (`lr12.py`)**.**

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

def cost_function(X, y, theta):
    m, n = X.shape
    J = 1 / (2 * m) * (X.dot(theta) - y).T.dot(X.dot(theta) - y)
```

```
    return J[0,0]

def gradient_descent(X, y, theta, alpha, itera):
    m, n = X.shape
    theta_n, dummy = theta.shape
    theta_hist = np.zeros((itera+1, theta_n))
    theta_hist[0] = theta.flatten()
    J_hist = np.zeros(itera+1)
    J_hist[0] = cost_function(X, y, theta)
    for i in range(itera):
        J_grad = 1 / m * (X.T.dot(X).dot(theta) - X.T.dot(y))
        theta = theta - alpha * J_grad
        J_hist[i+1] = cost_function(X, y, theta)
        theta_hist[i+1] = theta.flatten()
    return theta, theta_hist, J_hist

def hypo_function(X, theta):
    y_pred = X.dot(theta)
    return y_pred

df = pd.read_csv('../data/ex1data2.txt', names=['area','bedroom','price'])

df['all_1'] = 1
X = df[['all_1','area','bedroom']].values
y = df['price'].values
y = np.c_[y]

ALPHA = 0.00000003
ITERA = 50
THETA = np.array([[0],[0],[0]])

theta, theta_hist, J_hist = gradient_descent(X, y, THETA, ALPHA, ITERA)

fig = plt.figure()
ax = fig.add_subplot(1,1,1)

ax.plot(np.arange(ITERA+1), J_hist)
ax.set_xlabel('iter')
ax.set_ylabel(r'$J(\theta)$')

fig.savefig('lrfig9.eps')

X_pred = np.array([[1,2500,3]])
y_pred = hypo_function(X_pred, theta)

print('y_pred={}'.format(y_pred[0,0]))
```

```
y_pred=413220.18984796316
```

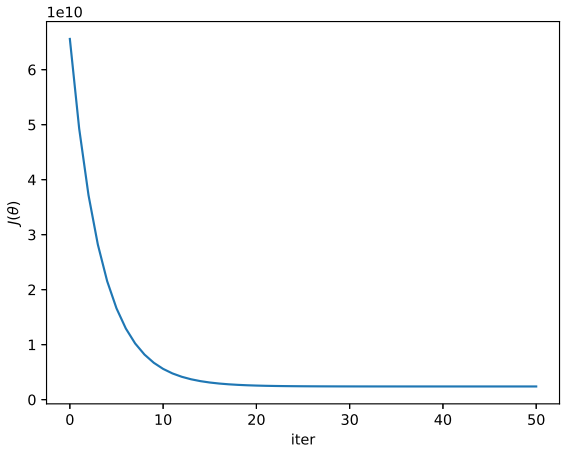

図 4.9 lrfig9.eps

2. 特徴量スケーリングを行う関数は，スケーリングに使用する平均値や標準偏差も併せて返すようにし，新しいデータに対する特徴量スケーリングを行う関数の引数として使えるようにする．あとは，α を変えながらひらすらデバッグをしていい感じに収束していくようなパターンを見つけていく．その結果，$\alpha = 0.3$ の場合がいい感じであった．

Code 4.14 (lr13.py).

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

def cost_function(X, y, theta):
    m, n = X.shape
    J = 1 / (2 * m) * (X.dot(theta) - y).T.dot(X.dot(theta) - y)
    return J[0,0]

def gradient_descent(X, y, theta, alpha, itera):
    m, n = X.shape
    theta_n, dummy = theta.shape
    theta_hist = np.zeros((itera+1, theta_n))
    theta_hist[0] = theta.flatten()
    J_hist = np.zeros(itera+1)
    J_hist[0] = cost_function(X, y, theta)
    for i in range(itera):
        J_grad = 1 / m * (X.T.dot(X).dot(theta) - X.T.dot(y))
        theta = theta - alpha * J_grad
        J_hist[i+1] = cost_function(X, y, theta)
        theta_hist[i+1] = theta.flatten()
    return theta, theta_hist, J_hist

def hypo_function(X, theta):
    y_pred = X.dot(theta)
    return y_pred

def train_mean_normalize(X):
```

```
    mean = X[:,1:].mean(axis=0)
    std = X[:,1:].std(axis=0)
    tmp = (X[:,1:] - mean) / std
    new_X = np.c_[X[:,0], tmp]
    return new_X, mean, std

def test_mean_normalize(X, mean, std):
    tmp = (X[:,1:] - mean) / std
    new_X = np.c_[X[:,0], tmp]
    return new_X

df = pd.read_csv('../data/ex1data2.txt', names=['area','bedroom','price'])

df['all_1'] = 1
X = df[['all_1','area','bedroom']].values
y = df['price'].values
y = np.c_[y]

X, mean, std = train_mean_normalize(X)

ALPHA = 0.3
ITERA = 50
THETA = np.array([[0],[0],[0]])

theta, theta_hist, J_hist = gradient_descent(X, y, THETA, ALPHA, ITERA)

fig = plt.figure()
ax = fig.add_subplot(1,1,1)

ax.plot(np.arange(ITERA+1), J_hist)
ax.set_xlabel('iter')
ax.set_ylabel(r'$J(\theta)$')

fig.savefig('lrfig10.eps')

X_pred = np.array([[1,2500,3]])
X_pred = test_mean_normalize(X_pred, mean, std)
y_pred = hypo_function(X_pred, theta)

print('y_pred={}'.format(y_pred[0,0]))
```

```
y_pred=411368.42281033465
```

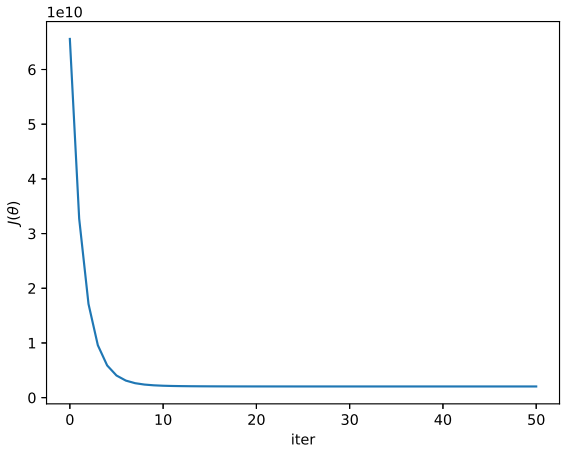

図 4.10　`lrfig10.eps`

3. 特徴量スケーリングをすると学習率はありうる自然な値とすることができる程度で，学習率として適切なものを設定できれば特徴量スケーリングをしてもしなくても少ない繰り返し回数で収束させることができるのではと考えられる．とはいえ，探索する学習率の範囲が広大だとデバッグに苦労するため，範囲を狭くするという意味で特徴量スケーリングを施す意義はあると思われる．なお，特徴量スケーリングはあくまでも学習効率を上げるための手法であり，予測精度を高める手法ではないことに注意する．実際，上記の例でも予測値はどちらも同じような値を示している．

□

4.3　多項式回帰

仮説関数として，線形関数を選ぶ必要はない．仮説関数として多項式を設定した場合は多項式回帰と呼ばれる．

定義 4.12 (多項式回帰)**.**
回帰問題において，仮説関数を多項式とした場合，特に**多項式回帰 (polynomial regression)** という．

問題 4.23. トレーニングセット $\{(x_1^{(i)}, y^{(i)})\}_{i=1}^{1}0$ をプロットしたところ，下図となった．これについての回帰問題を解きたい．仮説関数としてどのようなものが考えられるか．

Proof. 新しく特徴量として $x_2^{(i)} = (x_1^{(i)})^2$ を設定し，仮説関数として $h_{\boldsymbol{\theta}}(x^{(i)}) = \theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} = \theta_0 + \theta_1 x_1^{(i)} + \theta_2 (x_1^{(i)})^2$ とする． □

問題 4.24. 住宅の平米数から住宅価格を予測する回帰問題を解くことを考える．集めたデータの平米数はおおむね 1 から 1000 フィートの範囲となっている．平米数と価格をプロットしたところ，以下の仮説関数があてはまりがよさそうと考えた．

$$h_{\boldsymbol{\theta}}(x^{(i)}) = \theta_0 + \theta_1 \text{平米数}^{(i)} + \theta_2 \sqrt{\text{平米数}^{(i)}}$$

ここで，特徴量スケーリングをしてより適切に回帰問題を解くために，新しく仮説関数を

$$h_{\boldsymbol{\theta}}(x^{(i)}) = \theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)}$$

としたとき，$x_1^{(i)}$ と $x_2^{(i)}$ はそれぞれどのようにおくのがよいか．ここで，$\sqrt{1000} \fallingdotseq 32$ とする．

Proof. $x_1^{(i)} = \frac{\text{平米数}^{(i)}}{1000}$，$x_2^{(i)} = \frac{\sqrt{\text{平米数}^{(i)}}}{32}$. □

特徴量は，与えられたものをそのまま使う必要はなく，それらを組み合わせるなどして自分で新しく作っても良い．うまく特徴量を作り出して，より単純な仮説関数にするという選択肢もある．

問題 4.25. 今，手元に特徴量 $x_1^{(i)}$:間口，$x_2^{(i)}$:奥行き，出力変数 $y^{(i)}$:土地の価格がある．このとき，土地の価格を予測する問題を線形回帰で解くことを考えると，仮説関数としてまず $h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}) = \theta_0 + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)}$ が考えつくが，もっと簡単に仮説関数を定めるにはどうすれば良いか．

Proof. 新しい特徴量 $x_3^{(i)} = x_1^{(i)} x_2^{(i)}$:面積を設定し，仮説関数として $h_{\boldsymbol{\theta}}(x^{(i)}) = \theta_0 + \theta_1 x_3^{(i)}$ とする． □

参考文献

[1] Andrew Ng, “Machine Learning”, Stanford University, https://www.coursera.org/learn/machine-learning.

[2] Charles Severance, “Programming for Everybody (Getting Started with Python)”, University of Michigan, https://www.coursera.org/learn/python.

[3] Christopher Brooks, “Introduction to Data Science in Python”, University of Michigan, https://www.coursera.org/learn/python-data-analysis.

[4] Christopher Brooks, “Applied Plotting, Charting & Data Representation in Python”, University of Michigan, https://www.coursera.org/learn/python-plotting.

[5] McKinney W., “Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython”, 2nd Edition, O’Reilly Media, 2017.

[6] Jake VanderPlas, “Python データサイエンスハンドブック”, O’Reilly Media, 2018.

[7] 涌井良幸, “高校生からわかるベクトル解析”, ベレ出版, 2017.

[8] David A. Harville, “統計のための行列代数 上”, 丸善出版, 2007.

[9] C. R. Rao, “統計的推測とその応用”, 東京図書, 1977.