機械学習の Python との出会い

リリース *2017-01-15 12:10:49 +0900*

神嶌 敏弘 (Toshihiro Kamishima)

2017-01-15 12:10:49 +0900

目次

第 1 章　はじめに 1

1.1 本チュートリアルの方針 1
1.2 ソースコード 2
1.3 バグリポート 2

第 2 章　単純ベイズ：入門編 3

2.1 NumPy 配列の基礎 3
2.2 単純ベイズ：カテゴリ特徴の場合 9
2.3 入力データとクラスの仕様 10
2.4 学習メソッドの実装（1） 12
2.5 予測メソッドの実装 14
2.6 実行 18
2.7 「単純ベイズ：入門編」まとめ 19

第 3 章　単純ベイズ：上級編 21

3.1 クラスの再編成 21
3.2 単純ベイズの実装 （2） 23
3.3 配列の次元数や大きさの操作 25
3.4 ブロードキャスト 28
3.5 クラスの分布の学習 32
3.6 特徴の分布の学習 36
3.7 実行速度の比較 40
3.8 「単純ベイズ：上級編」まとめ 41

第 4 章　ロジスティック回帰 43

4.1 ロジスティック回帰の形式的定義 43
4.2 シグモイド関数 44
4.3 非線形最適化関数 50
4.4 学習メソッドの実装 52
4.5 損失関数とその勾配 56
4.6 実行と予測 59
4.7 「ロジスティック回帰」まとめ 61

第 5 章　おわりに 63

5.1 謝辞 63

索引 64

第 1 章

はじめに

機械学習の基本的な手法の実装を通じて，Python による科学技術計算プログラミングについて知ることができるように，このチュートリアルを執筆しました．

1.1 本チュートリアルの方針

このチュートリアルでは，いろいろな機械学習の手法を Python で実装する過程をつうじて，NumPy や SciPy など科学技術計算に関連したモジュールの具体的な使い方を説明します．機械学習の手法についてはごく簡単な説明に留めますので，詳細は他の本を参考にして下さい．また，クラスなどのプログラミングに関する基礎知識や，Python の基本的な文法については知っているものとして説明します．

プログラム言語やライブラリの解説の多くは，背景にある概念の説明，ソフトウェアのコア部分の仕様，そして，拡張部分の仕様といった順に，その機能の説明が中心となっています．ここでは，これらとは違うアプローチで Python を用いた数値計算プログラミングを説明します．まず，数値計算プログラミングの中でも，機械学習のアルゴリズム[*1] の実装について述べます．そして，これらのアルゴリズムを実装する過程で用いた関数やクラスの機能について順次説明します．

今までのように，概念や機能を中心とした説明は，ソフトウェアの機能を体系的に知るには良い方法です．しかし，どう使うのかが分からないまま，多くの機能についての説明を読み続けるのは，やや忍耐を要します．さらに，さまざまな機能を，どういう場面でどのように使うのかを具体的に知ることはあまりできません．そこで，具体的にアルゴリズムの実装し，ソフトウェアを完成させてゆくことで，興味深く Python を使った科学技術計算プログラミングについて，具体的に知ることができるように工夫しました．

ただ一方で，このような方針では，網羅的にパッケージやソフトウェアの機能を説明することは難しくなります．NumPy や SciPy には，体系的なリファレンスマニュアルやサンプルも整備されています．また SciPy や EuroSciPy などの国際会議 でも各種の優れたチュートリアルが公開されています．これらの体系的な情報に，このチュートリアルの具体的な実装例を補うことで，Python を用いた数値計算プログラミングについてより深く知ることができるようになればと思います．

[*1] 著者の専門が機械学習なので対象として選びました．NumPy や SciPy は機械学習専用というわけではなく，数値計算プログラミング全般について使いやすい機能を提供しています．

1.2 ソースコード

本稿のソースコード関する情報と注意点を最初にまとめておきます．

実行可能なソースコードは次の URL より参照できます．

https://github.com/tkamishima/mlmpy/tree/master/source

本稿では，以下のソフトウェアを利用します．

- まず Python を利用します．このチュートリアルでは，Python3 系統を対象とします．バージョン 3.5 で動作検証を行いましたので，これ以降のバージョンが望ましいです．Python2 との相違点については脚注などで補足しますが，これから新しく Python を学ばれるのであれば，Python3 を対象とされることを強くお薦めします．
- ここで紹介する NumPy と SciPy も利用します．
- 機械学習のライブラリ scikit-learn の API 仕様に従ってクラスを設計します．ただし，scikit-learn 自体の利用法については紹介しません．

これらのモジュールは以下のように `import` されていることを前提とします:

```
import numpy as np
import scipy as sp
```

1.3 バグリポート

TYPO や記述の誤りを見つけられた場合は，ご連絡いただけると今後の改善に役立てることができます．ご協力いただける場合は，このチュートリアルのソースファイル公開している `GitHub` の pull request か issues の機能を使ってお知らせ下さい．

https://github.com/tkamishima/mlmpy

できれば，issues より， pull request をいただける方が助かります．なお，事情によりすぐには対処できない場合もありますことを，あらかじめご承知おき下さい．

第 2 章

単純ベイズ：入門編

最初に実装するのは，特徴量がカテゴリ変数である場合の単純ベイズ (Naive Bayes) です．この単純ベイズの実装を通じて，NumPy / SciPy を用いた行列・ベクトルの初歩的な扱いについて説明します．

2.1 NumPy 配列の基礎

ここでは，NumPy で最も重要なクラスである `np.ndarray` について，**本チュートリアルの方針** の方針に従い，最低限必要な予備知識について説明します．

`np.ndarray` は，*N*-d Array すなわち，N 次元配列を扱うためのクラスです．NumPy を使わない場合，Python ではこうした N 次元配列を表現するには，多重のリストが利用されます．`np.ndarray` と多重リストには以下のような違いがあります．

- 多重リストはリンクでセルを結合した形式でメモリ上に保持されますが，`np.ndarray` は C や Fortran の配列と同様にメモリの連続領域上に保持されます．そのため，多重リストは動的に変更可能ですが，`np.ndarray` の形状変更には全体の削除・再生成が必要になります．
- 多重リストはリスト内でその要素の型が異なることが許されますが，`np.ndarray` は，基本的に全て同じ型の要素で構成されていなければなりません．
- 多重リストとは異なり，`np.ndarray` は各次元ごとの要素数が等しくなければなりません．すなわち，行ごとに列数が異なるような 2 次元配列などは扱えません．
- `np.ndarray` は，行や列を対象とした多くの高度な数学的操作を，多重リストより容易かつ高速に適用できます．また，配列中の全要素，もしくは一部の要素に対してまとめて演算や関数を適用することで，高速な処理が可能です．

2.1.1 NumPy 配列の生成

それでは，`np.ndarray` の生成方法を説明します．N 次元配列 `np.ndarray` は，数学の概念で言えば，1 次元の場合はベクトルに，2 次元の場合は行列に，そして 3 次元以上の場合はテンソルに該当します．

np.array() 関数による生成

np.ndarray にもコンストラクタはありますが，通常は， *np.array()* 関数[*1] によって生成します．

np.**array**(*object*, *dtype=None*)

Create an array.

最初の引数 object には，配列の内容を，array_like という型で与えます．この array_like という型は，配列を *np.ndarray* の他，（多重）リストや（多重）タプルで表現したものです．リストの場合は，ネストしていない直線状のリストでベクトルを表します．行列は，直線状リストで表した行を要素とするリスト，すなわち 2 重にネストしたリストで表します．もう一つの引数 dtype は，配列の要素の型を指定しますが，詳細は *np.ndarray* の属性のところで述べます．

要素が 1, 2, 3 である長さ 3 のベクトルの例です:

```
In [10]: a = np.array([1, 2, 3])
In [11]: a
Out[11]: array([1, 2, 3])
```

タプルを使った表現も可能です:

```
In [12]: a = np.array((10, 20, 30))
In [13]: a
Out[13]: array([10, 20, 30])
```

2 重にネストしたリストで表した配列の例です:

```
In [14]: a = np.array([[1.5, 0], [0, 3.0]])
In [15]: a
Out[15]:
array([[ 1.5,  0. ],
       [ 0. ,  3. ]])
```

リストの要素に *np.ndarray* やタプルを含むことも可能です:

```
In [16]: a = np.array([1.0, 2.0, 3.0])
In [17]: b = np.array([a, (10, 20, 30)])
In [18]: b
Out[18]:
array([[  1.,   2.,   3.],
       [ 10.,  20.,  30.]])
```

その他の関数による生成

np.ndarray を作るための関数は， *np.array()* 以外にも数多くありますが，それらのうちよく使うもの

[*1] 関数の引数は他にもありますが，ここでの説明に必要なもののみを示します．他の引数についてはライブラリのリファレンスマニュアルを参照して下さい．

を紹介します．

np.zeros() と *np.ones()* は，それぞれ要素が全て 0 である 0 行列と，全て 1 である 1 行列を生成する関数です．

np.**zeros**(*shape*, *dtype=None*)

Return a new array of given shape and type, filled with zeros.

np.**ones**(*shape*, *dtype=None*)

Return a new array of given shape and type, filled with ones.

shape は，スカラーや，タプルによって配列の各次元の長さを表したものです．大きさが 5 のベクトルはスカラー 5 によって，2×3 の行列はタプル (2, 3) によって表現します．

長さが 3 の 0 ベクトルの例です:

```
In [20]: np.zeros(3)
Out[20]: array([ 0.,  0.,  0.])
```

3×4 の 1 行列の例です．引数をタプルにすることを忘れないようにして下さい:

```
In [21]: np.ones((3, 4))
Out[21]:
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
```

配列を生成した後，その内容をすぐ後で書き換える場合には，配列の要素を全て 0 や 1 にするのは無駄な処理になってしまいます．そこで，要素の値が不定の状態のままで，指定した大きさの配列を生成する関数 *np.empty()* があります．

np.**empty**(*shape*, *dtype=None*)

Return a new array of given shape and type, without initializing entries.

この例では，2×3 の行列 a と同じ大きさの 0 行列を生成します:

```
In [18]: a = np.array([[1,2,3], [2,3,4]])
In [19]: np.zeros_like(a)
Out[19]:
array([[0, 0, 0],
       [0, 0, 0]])
```

最後に，*np.identity()* は，単位行列を生成する関数です．

np.**identity**(*n*, *dtype=None*)

Return the identity array.

n は行列の大きさを表します．例えば，4 と指定すると，単位行列は正方行列なので，大きさ 4×4 の行列を指定したことになります:

```
In [30]: np.identity(4)
Out[30]:
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

その他，連続した数列を要素とする配列，対角行列，三角行列などを生成するものや，文字列など他の型のデータから配列を生成するものなど，多種多様な関数が用意されています．これらの関数については，実装で必要になったときに随時説明します．

2.1.2 NumPy 配列の属性と要素の参照

ここでは，前節で生成した *np.ndarray* の属性を説明したのち，配列の要素を参照する方法について述べます．

np.ndarray には多数の属性がありますが，よく使われるものをまとめました．

class np.**ndarray**

An array object represents a multidimensional, homogeneous array of fixed-size items. An associated data-type object describes the format of each element in the array (its byte-order, how many bytes it occupies in memory, whether it is an integer, a floating point number, or something else, etc.)

変数

- ***dtype*** – Data-type of the array's elements
- **ndim** – Number of array dimensions
- **shape** – Tuple of array dimensions

最初の属性 dtype は配列の要素の型を表し，これまでに紹介した関数の引数でも使われていました．*np.ndarray* は，配列の中の全要素の型は基本的に同じです[*2]．二番目の属性 ndim は，次元数を表します．ベクトルでは 1 に，行列では 2 になります．三番目の属性 shape は，各次元ごとの配列の大きさをまとめたタプルで指定します．例えば，大きさが 5 のベクトルは (5,)[*3] となり，2×3 行列では (2, 3) となります．

これらの属性のうち，型を表す dtype について詳しく述べます．まず，Python のビルトイン型の真理値型，整数型，浮動小数点型，複素数型に対応する型があります．加えて，これらの型については，それを格納するのに必要なメモリをビット数で明示的に指定するものもあります．例えば， np.uint8 は 8 ビットの符号なし整数で $0, \ldots, 255$ の値を， np.float32 は 32bit，いわゆる単精度の浮動小数点数を表します．ビット数を指定する型は，GPU 利用のために 32bit の浮動小数点数を用いる，メモリを特に節約したい，また C や Fortran で書いた関数とリンクするといった場合に用います．これらの基本的な型についてまとめておきます．

[*2] オブジェクトを要素とする型 np.object や，行ごとに同じ構造である制限の下，いろいろな型を混在できる structured array を用いると，異なる型の要素を混在させることは可能です．

[*3] Python では， (5) と表記すると，スカラー量 5 を括弧でくくった数式とみなされるため，要素数が 1 個のタプルは (5,) となります．

- Python のビルトインの bool，int，float，および complex に対応するのは，それぞれ np.bool，np.int，np.float，および np.complex です．これらは同じオブジェクトなので，ビルトイン型のものを用いておけばよいでしょう．
- 最後に _ が付いた型は，ビット数を指定した型に対応します．int は C 言語の long 型に対応し np.int32 か np.int64 と同じものです．np.float_ と np.complex_ は，それぞれ np.float64 と np.complex128 と同じものです．
- *np.dtype()* でビルトイン型を変換すると _ の付いた型になります．例えば，np.dtype(float).type は np.float_ と同じオブジェクトになります．

文字列型については，ビルトイン型の str とは，少し異なります．*np.ndarray* では，要素の大きさが同じである必要があるため，文字列は固定長になります．Python の文字列型に対応する NumPy での文字列型は，NumPy の型を表すクラス *np.dtype* のコンストラクタを用いて，np.dtype('U<**文字列長**>')[*4] のように指定します．例えば，最大長が 16 である文字列を扱う場合は np.dtype("U16") のように指定します．なお，整数型や浮動小数点型にも同様の文字列を用いた指定方法があります．

配列の dtype 属性を指定するには，(1) *np.array()* などの配列生成関数の dtype 引数で指定する方法と，(2) *np.ndarray* の np.ndarray.astype() メソッドを使う方法とがあります．

まず，(1) の dtype 引数を指定する方法について述べます．*np.array()* では要素が全て整数の場合は，要素の型は整数になりますが，それを浮動小数点にするには，次のように指定します:

```
In [41]: a = np.array([1, 2, 3])
In [42]: a.dtype
Out[42]: dtype('int64')
In [43]: a = np.array([1, 2, 3], dtype=float)
In [44]: a.dtype
Out[44]: dtype('float64')
```

浮動小数点型の配列を複素数型で作り直す場合は，次のようになります:

```
In [45]: a = np.array([1.0, 1.5, 2.0])
In [46]: a.dtype
Out[46]: dtype('float64')
In [47]: a = np.array(a, dtype=complex)
In [48]: a.dtype
Out[48]: dtype('complex128')
In [49]: a
Out[49]: array([ 1.0+0.j,  1.5+0.j,  2.0+0.j])
```

(2) メソッドを使う方針でも，メソッド np.ndarray.astype() が同様に利用できます．

[*4] Python3 では文字列は，Python2 での Unicode 文字列に相当します．そのため，Python3 での文字列型として扱うためには，Unicode 文字列として構造化配列では定義する必要があります．S10 のように文字列として定義すると，b'string' のようなバイト列として扱われます．

```
In [50]: a = np.array([1, 2, 3])
In [51]: a.dtype
Out[51]: dtype('int64')
In [52]: a = a.astype(float)
In [53]: a.dtype
Out[53]: dtype('float64')
In [54]: a
Out[54]: array([ 1.,  2.,  3.])
```

次は *np.ndarray* の要素の参照方法について述べます．非常に多様な要素の参照方法があるため，最も基本的な方法のみを述べ，他の方法については順次紹介することにします．最も基本的な要素の参照方法とは，各次元ごとに何番目の要素を参照するかを指定します．1 次元配列であるベクトル a の要素 3 である a[3] を参照すると，次のような結果が得られます．

```
In [60]: a = np.array([1, 2, 3, 4, 5], dtype=float)
In [61]: a[3]
Out[61]: 4.0
```

ここで注意すべきは，添え字の範囲は，数学の規則である $1, \ldots, 5$ ではなく，Python の規則に従って $0, \ldots, 4$ となることです．a.shape[0] とすると，第 1 次元の要素の長さ，すなわちベクトルの長さとして 5 が得られますが，添え字の範囲はそれより一つ前の 4 までとなります．同様に，2×3 の行列では，行は $0, \ldots, 1$ の範囲で，列は $0, \ldots, 2$ の範囲で指定します．

```
In [62]: a = np.array([[11, 12, 13], [21, 22, 23]])
In [63]: a.shape
Out[63]: (2, 3)
In [64]: a[1,2]
Out[64]: 23
```

最後に，*np.ndarray* の 1 次元と 2 次元の配列と，数学の概念であるベクトルと行列との関係について補足します．線形代数では，縦ベクトルや横ベクトルという区別がありますが，1 次元の *np.ndarray* 配列にはそのような区別はありません．そのため，1 次元配列を転置することができず，数学でいうところのベクトルとは厳密には異なります．

そこで，縦ベクトルや横ベクトルを区別して表現するには，それぞれ列数が 1 である 2 次元の配列と，行数が 1 である 2 次元配列を用います．縦ベクトルは次のようになり:

```
In [65]: np.array([[1], [2], [3]])
Out[65]:
array([[1],
       [2],
       [3]])
```

横ベクトルは次のようになります（リストが 2 重にネストしていることに注意）:

```
In [66]: np.array([[1, 2, 3]])
Out[66]: array([[1, 2, 3]])
```

以上，NumPy の配列 `np.ndarray` について基本的なことを述べました．ここで紹介した基本事項を使い，NumPy / SciPy のいろいろな機能を，機械学習のアルゴリズムの実装を通じて紹介してゆきます．

2.2 単純ベイズ：カテゴリ特徴の場合

実装の前に，特徴がカテゴリ変数である場合の単純ベイズ法による分類について，ごく簡単に復習します．

変数を次のように定義します．

- 特徴量 $\mathbf{x}_i = (x_{i1}, \ldots, x_{iK})$ の要素 x_{ij} はカテゴリ変数で， F_j 個の値のうちの一つをとります．ただし， K は特徴の種類数です．
- クラス y は， C 個の値のうちの一つをとります．

ここで，特徴 $\mathbf{X}$ は，クラス Y が与えられたとき条件付き独立であるとする単純ベイズの仮定を導入すると，$\mathbf{X}$ と Y の同時分布は次式で与えられます．

$$\Pr[\mathbf{X}, Y] = \Pr[Y] \prod_{j=1}^{K} \Pr[X_j|Y] \tag{2.1}$$

$\Pr[Y]$ と $\Pr[X_j|Y]$ の分布がカテゴリ分布（離散分布）である場合，学習すべき単純ベイズのパラメータは次のとおりです．

$$\begin{aligned} \Pr[y], &\quad y = 1, \ldots, C \\ \Pr[x_j|y], &\quad y = 1, \ldots, C,\ x_j = 1, \ldots, F_j,\ j = 1, \ldots, K \end{aligned} \tag{2.2}$$

さらに，ここでは実装を容易にするために， C や F_j は全て 2 に固定します．すなわち，クラスや各特徴量は全て 2 値変数となり，これにより $\Pr[Y]$ と $\Pr[X_j|Y]$ は，カテゴリ分布の特殊な場合であるベルヌーイ分布に従うことなります．

ここで，大きさ N のデータ集合 $\mathcal{D} = \{\mathbf{x}_i, y_i\}, i = 1, \ldots, N$ が与えられると，対数尤度は次式になります．

$$\mathcal{L}(\mathcal{D}; \{\Pr[y]\}, \{\Pr[x_j|y]\}) = \sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}} \ln \Pr[\mathbf{x}_i, y_i] \tag{2.3}$$

この対数尤度を最大化する最尤推定により 式 (2.2) のパラメータを求めます．クラスの分布のパラメータ群 $\Pr[y]$ は次式で計算できます．

$$\Pr[y] = \frac{N[y_i = y]}{N}, \quad y \in \{0, 1\} \tag{2.4}$$

ただし， $N[y_i = y]$ は，データ集合 $\mathcal{D}$ のうち，クラス y_i が値 y である事例の数です．もう一つのパラメータ群 $\Pr[x_{ij} = x_{ij}|y_i = y]$ は次式となります．

$$\Pr[x_j|y] = \frac{N[x_{ij} = x_j, y_i = y]}{N[y_i = y]}, \quad y \in \{0, 1\},\ x_j \in \{0, 1\},\ j = 1, \ldots, K \tag{2.5}$$

ただし，$N[x_{ij} = x_j, y_i = y]$ は，データ集合 $\mathcal{D}$ のうち，クラス y_i の値が y であり，かつ特徴 x_{ij} の値が x_j である事例の数です．以上のパラメータの計算に必要な値 N ， $\Pr[x_{ij} = x_{ij}|y_i = y]$ ，および $N[x_{ij} = x_j, y_i = y]$ は，データ集合 $\mathcal{D}$ に対する分割表を作成すれば計算できます．

予測をするときには，入力ベクトル $\mathbf{x}^{\mathrm{new}}$ が与えられたときのクラス事後確率を最大にするクラスを，次式で求めます．

$$
\begin{aligned}
\hat{y} &= \arg\max_y \Pr[y|\mathbf{x}^{\mathrm{new}}] \\
&= \arg\max_y \frac{\Pr[y]\Pr[\mathbf{x}^{\mathrm{new}}|y]}{\sum_{y'}\Pr[y']\Pr[\mathbf{x}^{\mathrm{new}}|y']} \\
&= \arg\max_y \Pr[y]\Pr[\mathbf{x}^{\mathrm{new}}|y] \\
&= \arg\max_y \left(\Pr[y]\prod_j \Pr[x_j^{\mathrm{new}}|y]\right) \\
&= \arg\max_y \left(\log\Pr[y] + \sum_j \log\Pr[x_j^{\mathrm{new}}|y]\right)
\end{aligned}
\tag{2.6}
$$

この式は，式 (2.4) と (2.5) で求めたパラメータを利用して計算できます．最後に対数をとっているのは，浮動小数点計算では小さな値のかけ算を繰り返すことにより計算結果が不安定になる場合がありますが，この不安定さを避けるためです．

2.3 入力データとクラスの仕様

単純ベイズ法を実装するために，入力データと，単純ベイズ法のクラスの仕様を定めます．

2.3.1 入力データの仕様

単純ベイズ：カテゴリ特徴の場合 では，単純ベイズの入力データは， $\mathcal{D} = \{\mathbf{x}_i, y_i\}, i = 1, \ldots, N$ であると述べました．これを NumPy 配列で表現します．

入力の特徴ベクトル $\mathbf{x}_i$ は，長さが K のベクトルです．NumPy 配列の多様な配列操作を利用できるようにするため，このベクトルを N 個集めたものをまとめて一つの $N \times K$ の大きさの行列で表現します．すなわち，入力特徴ベクトル集合を表す変数 `X` は 2 次元なので， `ndim` 属性は `2` に， `shape` 属性は `(N, K)` とします．また，特徴は全て二値変数に制限したので，配列 `X` の要素は 0 か 1 の整数となり，変数 `X` の要素の型である `dtype` 属性は整数型となります．

もう一方の入力のクラス変数 y_i は， 0 か 1 の整数をとるスカラーです．これを N 個集めて，長さ N の 1 次元配列 `y` で表現します．この変数の属性 `ndim` ， `shape` ，および `dtype` 属性は，それぞれ `2` ， `N` ，および整数型となります．この特徴ベクトル `X` とクラス変数 `y` との組が，学習時の入力となります．

パラメータの学習後，クラスが未知である特徴ベクトル $\mathbf{x}_{\mathrm{new}}$ のクラスを予測する場合の入力を考えます．この特徴ベクトルも， M 個まとめて与えられるものとし，それらを集めて `shape` 属性が (M, K) の変数 `X` で表すことにします．予測時には，クラス変数はないので，この変数 `X` のみが入力となります．

2.3.2 単純ベイズクラスの仕様

次に，単純ベイズ法のためのクラスを設計します．機械学習のアルゴリズムは，関数を用いるだけでも実装できますが，クラスを定義して実装することには，次のような利点があります．

- クラスの継承を利用して，モデルと予測メソッドだけを共有し，学習アルゴリズムだけを変えて部分的に改良するといったことが容易になります．
- `cPickle` などの，オブジェクトのシリアライズを利用して，学習したモデルのオブジェクトを，ファイルに保存しておくことで再利用できるようになります．

このチュートリアルでは，Python の機械学習パッケージ scikit-learn の API 仕様 (APIs of scikit-learn objects) に従ってクラスを設計します．主な仕様は次のとおりです．

- データに依存しないアルゴリズムのパラメータは，クラスのコンストラクタの引数で指定する．
- 学習は `fit()` メソッドで行う．訓練データと，データに依存したパラメータを，このメソッドの引数で指定する．
- 予測は `predict()` メソッドで行う．新規の入力データを，このメソッドの引数で指定する．
- モデルのデータへのあてはめの良さの評価は， `score()` メソッドで行う．評価対象のデータを，このメソッドの引数で指定する．
- 次元削減などのデータ変換は， `transform()` メソッドで行う．

単純ベイズクラスの名前は `NaiveBayes1` とします．単純ベイズは教師あり学習であるため，パラメータの初期化を行うコンストラクタ，学習を行う `fit()` メソッド，および予測を行う `predict()` メソッドが最低限必要になります．

まず，クラスの定義は次のとおりです．

```
class NaiveBayes1(object):
    """
    Naive Bayes class (1)
    """
```

ここで実装する単純ベイズクラスは，他のクラスを継承してその機能を利用する必要はないので，親クラスを `object` とします．

コンストラクタの定義は次のとおりです．

```
def __init__(self):
    """
    Constructor
    """
    self.pY_ = None
    self.pXgY_ = None
```

単純ベイズ：カテゴリ特徴の場合 の単純ベイズには，データに依存しないパラメータはないので，コンストラクタ `__init__()` の引数は `self` だけです．このコンストラクタの中では，学習すべきモデルのパラメータを格納するためのインスタンス変数を作成します．**単純ベイズ：カテゴリ特徴の場合** の式 (4) と (5) がモデルのパラメータです．式 (4) の $\Pr[y]$ はインスタンス変数 `self.pY_` に，式 (5) の $\Pr[x_j|y]$ はインスタンス変数 `self.pXgY_` に格納します．モデルパラメータを格納するインスタンス変数の名前は，scikit-learn の慣習に従って，その最後を _ としました．これらのモデルパラメータを格納する配列の大きさは，データに依存して決まるため，コンストラクタでは `None` で初期化します．

学習を行う `fit()` メソッドの枠組みは次のとおりです．

```
def fit(self, X, y):
    """
    Fitting model
    """
    pass
```

`fit()` メソッドの引数 `X` と `y` は，前節で述べたように訓練データと特徴ベクトルとクラスラベルの集合を表します．具体的な学習アルゴリズムの実装は **学習メソッドの実装（1）** で述べます．

クラスを予測する `predict()` メソッドの枠組みは次のとおりです．

```
def predict(self, X):
    """
    Predict class
    """
    pass
```

この `predict` メソッドの引数 `X` は，前節で述べたように未知のデータを表します．このメソッドの具体的な実装は **予測メソッドの実装** で述べます．

2.4 学習メソッドの実装（1）

モデルパラメータを，訓練データから学習する `fit()` メソッドを，単純に多次元配列として，NumPy 配列を利用する方針で実装します．実は，この実装方針では NumPy の利点は生かせませんが，後の **単純ベイズ：上級編** 章で，NumPy のいろいろな利点を順に紹介しながら，この実装を改良してゆきます．

2.4.1 定数の設定

まず，メソッド内で利用する定数を定義します．このメソッドの引数は，訓練データの特徴ベクトル集合 `X` とクラスラベル集合 `y` であると **単純ベイズクラスの仕様** で定義しました．最初に，この引数から特徴数や訓練事例数などの定数を抽出します．`X` は，行数が訓練事例数に，列数が特徴数に等しい行列に対応した 2 次元配列です．そこでこの変数の `shape` 属性のタプルから訓練事例数と特徴数を得ます．

```
n_samples = X.shape[0]
n_features = X.shape[1]
```

実装する単純ベイズは，クラスも特徴も全て二値としましたが，このことを定義する定数も定義しておきます．

```
n_classes = 2
n_fvalues = 2
```

特徴の事例数とクラスラベルの事例数は一致していなくてはならないので，そうでない場合は `ValueError` を送出するようにします．`y` の `shape` 属性を調べてもよいのですが，これは 1 次元配列なので長さを得る関数 `len()`[*1] を用いて実装してみます．

```
if n_samples != len(y):
    raise ValueError('Mismatched number of samples.')
```

以上で，モデルパラメータを学習する準備ができました．

2.4.2 クラスの分布の学習

単純ベイズ：カテゴリ特徴の場合 の式 (4) のクラスの分布のパラメータを求めます．計算に必要な量は総事例数 N とクラスラベルが y である事例数 $N[y_i = y]$ です．N はすでに `n_samples` として計算済みです．$N[y_i = y]$ は，$y \in \{0, 1\}$ について計算する必要があります．よって，大きさ `n_classes` の大きさのベクトル `nY` を作成し，各クラスごとに事例を数え上げます．

```
nY = np.zeros(n_classes, dtype=int)
for i in range(n_samples):
    nY[y[i]] += 1
```

モデルパラメータ `self.pY_` は式 (4) に従って計算します．なお，後で値を書き換えるので *`np.empty()`* で初期化したあと，各クラスの確率を計算して格納します[*2]．

```
self.pY_ = np.empty(n_classes, dtype=float)
for i in range(n_classes):
    self.pY_[i] = nY[i] / n_samples
```

2.4.3 特徴の分布の学習

単純ベイズ：カテゴリ特徴の場合 の式 (5) の特徴の分布のパラメータを求めます．計算に必要な量のうち $N[y_i = y]$ は，すでに式 (4) の計算で求めました．もう一つの量 $N[x_{ij} = x_j, y_i = y]$ は，特徴 $j = 1, \ldots, K$ それぞれについて，特徴の値 $x_j \in \{0, 1\}$ とクラス $y \in \{0, 1\}$ について計算する必要があります．よって，こ

[*1] 2 次元以上の NumPy 配列に `len()` を適用すると `shape` 属性の最初の要素を返します．

[*2] Python3 では，整数同士の除算の結果も実数ですが，Python2 では切り捨てした整数となります．これを避けるために，Python2 では `n_samples` を `float(n_samples)` のように実数型に変換しておく必要があります．

の量を保持する配列は 3 次元で，その shape 属性は (n_features, n_fvalues, n_classes) とする必要があります．この大きさの 0 行列を確保し，各特徴それぞれについて，各特徴値と各クラスごとに事例を数え上げます．

```
nXY = np.zeros((n_features, n_fvalues, n_classes), dtype=int)
for i in range(n_samples):
    for j in range(n_features):
        nXY[j, X[i, j], y[i]] += 1
```

モデルパラメータ self.pXgY_ は式 (5) に従って計算します．

```
self.pXgY_ = np.empty((n_features, n_fvalues, n_classes),
                      dtype=float)
for j in range(n_features):
    for xi in range(n_fvalues):
        for yi in range(n_classes):
            self.pXgY_[j, xi, yi] = nXY[j, xi, yi] / float(nY[yi])
```

以上で，単純ベイズのモデルパラメータの学習を完了しました．

2.5 予測メソッドの実装

学習したモデルパラメータを使って，未知の事例のクラスを予測する predict() メソッドを，できるだけ NumPy 配列の利点を活用生かす方針で実装します．

このメソッドは，未知の特徴ベクトルをいくつか集めた配列 X を引数とします．そして， X 中の各特徴ベクトルに対する予測ベクトルをまとめた配列 y を返します．最初に， fit() メソッドと同様に， n_samples や n_features などの定数を設定します．

2.5.1 未知ベクトルの抽出

次に， X から未知ベクトルを一つずつ抽出します．*NumPy* **配列の属性と要素の参照** では，配列の要素を一つずつ参照する方法を紹介しました．これに加え，NumPy 配列は，リストや文字列などのスライスと同様の方法により，配列の一部分をまとめて参照することもできます．

1 次元配列の場合は，リストのスライス表記と同様の **開始 : 終了 : 増分** の形式を用います．

```
In [10]: x = np.array([0, 1, 2, 3, 4])
In [11]: x[1:3]
Out[11]: array([1, 2])
In [12]: x[0:5:2]
Out[12]: array([0, 2, 4])
In [13]: x[::-1]
Out[13]: array([4, 3, 2, 1, 0])
```

```
In [14]: x[-3:-1]
Out[14]: array([2, 3])
```

NumPy 配列やリストを使って複数の要素を指定し，それらをまとめた配列を作ることもできます．これは，配列 x からリストと NumPy 配列を使って選んだ要素を並べた配列を作る例です．

```
In [20]: x = np.array([10, 20, 30, 40, 50])
In [21]: x[[0, 2, 1, 2]]
Out[21]: array([10, 30, 20, 30])
In [22]: x[np.array([3, 3, 1, 1, 0, 0])]
Out[22]: array([40, 40, 20, 20, 10, 10])
```

2 次元以上の配列でも同様の操作が可能です．特に， : のみを使って，行や列全体を取り出す操作はよく使われます．

```
In [30]: x = np.array([[11, 12, 13], [21, 22, 23]])
In [31]: x
Out[31]:
array([[11, 12, 13],
       [21, 22, 23]])
In [32]: x[0, :]
Out[32]: array([11, 12, 13])
In [33]: x[:, 1]
Out[33]: array([12, 22])
In [34]: x[:, 1:3]
Out[34]:
array([[12, 13],
       [22, 23]])
```

それでは，配列 X から一つずつ行を取り出してみます．そのために for ループで i 行目を順に取り出します．

```
for i in range(n_samples):
    xi = X[i, :]
```

np.ndarray は，最初の次元を順に走査するイテレータの機能も備えています．具体的には，1 次元配列なら要素を順に返し，2 次元配列なら行列の行を順に返し，3 次元配列なら 2 次元目と 3 次元目で構成される配列を順に返します．次の例では，行のインデックスを変数 i に，行の内容を変数 xi に同時に得ることができます．

```
for i, xi in enumerate(X):
    pass
```

なお，リスト内包表記や np.apply_along_axis() を利用する方法もありますが，どの実装の実行速度が速いかは事例数や特徴数に依存するようです．

2.5.2 対数同時確率の計算

方針（1）

次に，この未知データ xi のクラスラベルを，**単純ベイズ：カテゴリ特徴の場合** の式 (6) を用いて予測します．すなわち，xi に対し，y が 0 と 1 それぞれの場合の対数同時確率を計算し，その値が大きな方を予測クラスラベルとします．

まず y が 0 と 1 の場合を個別に計算するのではなく，NumPy の利点の一つであるユニバーサル関数を用いてまとめて計算する方針で実装します．ユニバーサル関数は，入力した配列の各要素に関数を適用し，その結果を入力と同じ形の配列にします．式 (6) の最初の項 $\log \Pr[y]$ は，クラスの事前分布のパラメータ self.pY_ に対数関数を適用して計算します．このとき，対数関数として math パッケージの対数関数 math.log() ではなく，ユニバーサル関数の機能をもつ NumPy の対数関数 np.log()[*1] を用います．

```
logpXY = np.log(self.pY_)
```

式 (6) の第 2 項の総和の中 $\log \Pr[x_j^{\mathrm{new}}|y]$ の計算に移ります．計算に必要な確率関数は，モデルパラメータ self.pXgY の j 番目の要素で，もう一方の未知特徴ベクトルの値は，xi の j 番目の要素で得られます．最後の y については，: を使うことで 0 と 1 両方の値を同時に得ます．これを全ての特徴 j について求め，それらを logpXY に加えます．

```
for j in range(n_features):
    logpXY = logpXY + np.log(self.pXgY_[j, xi[j], :])
```

np.log() と同様に，+ や * などの四則演算もユニバーサル関数としての機能を持っています．同じ大きさの配列 a と b があるとき，a + b は要素ごとの和をとり，入力と同じ大きさの配列を返します．* については，内積や行列積ではなく，要素ごとの積が計算されることに注意して下さい．

```
In [40]: a = np.array([1, 2])
In [41]: b = np.array([3, 4])
In [42]: a + b
Out[42]: array([4, 6])
In [43]: a * b
Out[43]: array([3, 8])
```

方針（2）

以上のような for ループを用いた実装をさらに改良し，NumPy の機能をさらに生かした実装を紹介します．具体的には，(1) NumPy 配列 self.pXgY_ の要素を，一つずつではなくまとめて取り出して (2) それらの総和を計算します．

まず (1) には，NumPy 配列やリストを使って複数の要素を指定し，それらをまとめた配列を作る機能を利用します．for 文によって j を変化させたとき self.pXgY_[j, xi[j], :] の 1 番目の添え字は 0 から

[*1] np.log() や np.sin() などの NumPy の初等関数は，math のものと比べて，ユニバーサル関数であることの他に，np.seterr() でエラー処理の方法を変更できたり，複素数を扱えるといった違いもあります．

n_features - 1 の範囲で変化します．2 番目の引数は， xi の要素を最初から最後まで並べたもの，すなわち xi そのものになります．以上のことから， self.pXgY_ の要素をまとめて取り出すとき，2 番目の添え字には xi を与え，3 番目の引数は : でこの軸の全要素を指定できるので，あとは 1 番目の添え字が指定できれば目的を達成できます．1 番目の添え字は 0 から n_features - 1 の整数を順にならべたものです．このような，等差級数の数列を表す配列は *np.arange()* 関数で生成できます．

np.**arange**([*start*], *stop*[, *step*], *dtype=None*)

Return evenly spaced values within a given interval.

使い方はビルトインの range() 関数と同様で，開始，終了，増分を指定します．ただし，リストではなく 1 次元の配列を返すことや，配列の dtype 属性を指定できる点が異なります．NumPy 配列の添え字として与える場合には dtype 属性は整数でなくてはなりません．ここでは， np.arange(n_features) と記述すると，引数が整数ですので，規定値で整数型の配列がちょうど得られます．以上のことから self.pXgY_[np.arange(n_features), xi, :] によって，各行が， j を 0 から n_features - 1 まで変化させたときの， self.pXgY_[j, xi[j], :] の結果になっている配列が得られます．なおこの配列の shape 属性は (n_features, n_classes) となっています．

この配列の各要素ごとに対数をとり， j が変化する方向，すなわち列方向の和をとれば目的のベクトルが得られます．まず， np.log() を適用すれば，ユニバーサル関数の機能によって，配列の全要素について対数をとることができます．

列方向の和をとるには *np.sum()* 関数を利用します．

np.**sum**(*a*, *axis=None*, *dtype=None*)

Sum of array elements over a given axis.

引数 a で指定された配列の，全要素の総和を計算します．ただし， axis を指定すると，配列の指定された次元方向の和を計算します．dtype は，返り値配列の dtype 属性です．

axis 引数について補足します．axis は， 0 から ndim で得られる次元数より 1 少ない値で指定します．行列に相当する 2 次元配列では， axis=0 は列和に， axis=1 は行和になります．計算結果の配列は，指定した次元は和をとることで消えて次元数が一つ減ります．指定した次元以外の shape 属性はそのまま保存されます．

対数同時確率は，これまでの手順をまとめた次のコードで計算できます．

```
logpXY = np.log(self.pY_) + \
         np.sum(np.log(self.pXgY_[np.arange(n_features), xi, :]),
                axis=0)
```

2.5.3 予測クラスの決定

以上で， y が 0 と 1 に対応する値を含む配列 logpXY が計算できました．このように計算した logpXY のうち最も大きな値をとる要素が予測クラスになります．これには，配列中で最大値をとる要素の添え字を返す関

数 *np.argmax()* を用います[*2]．

np.**argmax**(*a*, *axis=None*)

Indices of the maximum values along an axis.

逆に最小値をとる要素の添え字を返すのは *np.argmin()* です．

np.**argmin**(*a*, *axis=None*)

Return the indices of the minimum values along an axis.

予測クラスを得るコードは次のとおりです．

```
y[i] = np.argmax(logpXY)
```

この例では，予め確保しておいた領域 y に予測クラスを順に格納しています．

以上で， NaiveBayes1 クラスの実装は完了しました．実行可能な状態のファイルは，以下より取得できます．

https://github.com/tkamishima/mlmpy/blob/master/source/nbayes1.py

2.6 実行

最後に，データをファイルから読み込み，そのデータに対して，実装した NaiveBayes1 クラスを用いて学習と予測を行います．

最初に， NaiveBayes1 クラスを import します．

```
from nbayes1 import NaiveBayes1
```

次にファイルからデータを読み込みます．NumPy と SciPy にはいろいろな形式のファイルを読み込む関数がありますが[*1]，テキスト形式のファイルの読み込みをする *np.genfromtxt()*[*2] を用います．

np.**genfromtxt**(*fname*, *dtype=<type 'float'>*, *comments='#'*, *delimiter=None*)

Load data from a text file, with missing values handled as specified.

この関数は，カンマ区切り形式や，タブ区切り形式のテキストファイルを読み込み，それを NumPy 配列に格納します．fname は，読み込むファイルを，ファイル名を示す文字列か， open() 関数で得たファイルオブジェクトで指定します．dtype は，関数が返す NumPy 配列の dtype 属性を指定します．comments で指定した文字が，ファイル中の行の先頭にある場合，その行はコメント行として読み飛ばされます．delimiter は，列の区切りを指定します．規定値では，タブを含むホワイトスペースの位置で区切ります．カンマ区切り

[*2] NumPy 配列のメソッド np.ndarray.argmax() を使う方法もあります．

[*1] 代表的な読み込み関数には，バイナリの npy 形式 np.load()， matlab 形式 sp.io.loadmat()， Weka の arff 形式 sp.io.loadarff() などがあります．ファイルの読み込みについては，Scipy.org にある Cookbook / InputOutput が参考になります．

[*2] np.loadtxt() という同様の機能をもつ関数もあります．*np.genfromtxt()* は， np.loadtxt() の機能に加えて，欠損値処理の機能が加えられているので，こちらを紹介します．

csv ファイルの場合は，カンマ ",," を区切り文字列として指定します．区切り文字ではなく，数値や数値のタプルを指定することで，文字数で区切ることもできます．引数の種類が非常に多い関数なので，ごく一部のみをここでは紹介しました．その他の機能については Importing data with genfromtxt などを参照して下さい．

NaiveBayes1 のテスト用データとして， vote_filled.tsv を用意しました[*3]．

https://github.com/tkamishima/mlmpy/blob/master/source/vote_filled.tsv

このデータはタブ区切り形式です．また， NaiveBayes1 クラスでは，入力訓練データの dtype 属性が整数であることを前提としています．よって，次のようにファイルを読み込みます．

```
data = np.genfromtxt('vote_filled.tsv', dtype=int)
```

このファイルは，最終列がクラスラベル，それ以外に特徴量を格納しています．このため，変数 data の最終列をクラスラベルの配列 y に，それ以外を特徴量の配列 X に格納します．

```
X = data[:, :-1]
y = data[:, -1]
```

データが揃ったので，いよいよ NaiveBayes1 クラスを使うことができます．設計どおり，コンストラクタで分類器を作り， fit() メソッドに訓練データを与えてモデルパラメータを学習させます．

```
clr = NaiveBayes1()
clr.fit(X, y)
```

テスト用のデータは， X の最初の 10 個分を再利用します．予測クラスは，分類器の predict() メソッドで得られます．結果が正しいかどうかを調べるため，元のクラスと予測クラスを表示してみます．

```
predict_y = clr.predict(X[:10, :])
for i in range(10):
    print(i, y[i], predict_y[i])
```

結果を見ると，ほぼ正しく予測出来ていますが，6 番のデータについては誤って予測しているようです．

実行可能な状態の NaiveBayes1 の実行スクリプトは，以下より取得できます．実行時には nbayes1.py と vote_filled.tsv がカレントディレクトリに必要です．

https://github.com/tkamishima/mlmpy/blob/master/source/run_nbayes1.py

2.7 「単純ベイズ：入門編」まとめ

単純ベイズ：入門編 の章では，単純ベイズ法の実装を通じて以下の内容を紹介しました．

[*3] vote_filled.tsv は UCI Repository の Congressional Voting Records Data Set をタブ区切り形式にしたファイルです．アメリカ議会での 16 種の議題に対する投票行動を特徴とし，議員が共和党 (0) と民主党 (1) のいずれであるかがクラスです．元データには欠損値が含まれていますが，各クラスの最頻値で補完しました．

- *NumPy* 配列の基礎
  - NumPy 配列 `np.ndarray` の特徴
  - `np.array()` による NumPy 配列の生成
  - `np.zeros()` など，その他の関数による NumPy 配列の生成
  - NumPy 配列 `np.ndarray` クラスの属性
  - NumPy 値の型 `np.dtype`
  - NumPy 配列の値の参照方法
  - NumPy 配列と，数学のベクトルや行列との対応
- 単純ベイズ：カテゴリ特徴の場合
  - 特徴がカテゴリ変数である場合の単純ベイズ法
- 入力データとクラスの仕様
  - 入力データの仕様例
  - 機械学習アルゴリズムをクラスとして実装する利点
  - scikit-learn モジュールの API 基本仕様
  - 機械学習アルゴリズムのクラスの仕様例
- 学習メソッドの実装（1）
  - NumPy 配列の基本的な参照を用いたアルゴリズムの実装
- 予測メソッドの実装
  - NumPy 配列のスライスを使った参照
  - ユニバーサル関数によるベクトル化演算
  - `for` ループを用いない実装の例
  - `np.sum()` の紹介．特に， `axis` 引数について
  - `np.argmax()`， `np.argmin()`
- 実行
  - `np.genfromtxt()` を用いたテキスト形式ファイルの読み込み
  - scikit-learn API 基本仕様に基づくクラスの利用

第 3 章

単純ベイズ：上級編

この章では，**単純ベイズ：入門編** で実装した NaiveBayes1 クラスを，NumPy のより高度な機能を利用して改良します．その過程で，NumPy の強力な機能であるブロードキャストの機能と，この機能を活用する手順を紹介します．

3.1 クラスの再編成

この章では単純ベイズ法の学習のいろいろな実装を比較するのに便利になるように，**単純ベイズ：入門編** の NaiveBayes1 クラスを再編成します．NaiveBayes1 クラスには，コンストラクタの他には，学習を行う fit() メソッドと，予測を行う predict() メソッドがありました．predict() メソッドはどの実装でも共通にする予定ですが，学習メソッドのいろいろな実装をこれから試します．そこで，予測メソッドなど共通部分含む抽象クラスを新たに作成し，各クラスで異なる学習メソッドは，その抽象クラスを継承した下位クラスに実装することにします．

3.1.1 二値単純ベイズの抽象クラス

二値単純ベイズの共通部分を含む抽象クラス BaseBinaryNaiveBayes を作成します．抽象クラスを作るための abc モジュールを利用して，次のようにクラスを定義しておきます[*1]．

```
from abc import ABCMeta, abstractmethod

class BaseBinaryNaiveBayes(object, metaclass=ABCMeta):
"""
Abstract Class for Naive Bayes whose classes and features are binary.
"""
```

この抽象クラスでは実装しない fit() メソッドは，抽象メソッドとして次のように定義しておきます．このように定義しておくと，この抽象クラスの下位クラスで fit() メソッドが定義されていないときには例外が

[*1] 抽象クラスの定義の記述は Python2 では異なっています．Python3 と 2 の両方で動作するようにするには six などのモジュールが必要になります．

発生するので，定義し忘れたことが分かるようになります．

```
@abstractmethod
def fit(self, X, y):
    """
    Abstract method for fitting model
    """
    pass
```

最後に，今後の単純ベイズの実装で共通して使うコンストラクタと predict() メソッドを，今までの NaiveBayes1 からコピーしておきます．以上で，二値単純ベイズの抽象クラスは完成です．

3.1.2 新しい NaiveBayes1 クラス

新しい NaiveBayes1 クラスを，上記の BaseBinaryNaiveBayes の下位クラスとして次のように定義します．

```
class NaiveBayes1(BaseBinaryNaiveBayes):
    """
    Naive Bayes class (1)
    """
```

次に，このクラスのコンストラクタを作成します．ここでは単に上位クラスのコンストラクタを呼び出すように定義しておきます．

```
def __init__(self):
    super(NaiveBayes1, self).__init__()
```

最後にこのクラスに固有の fit() メソッドを，以前の NaiveBayes1 クラスからコピーしておきます．以上で， NaiveBayes1 クラスの再編成が完了しました．

3.1.3 実行

新しい NaiveBayes1 クラスの実行可能な状態のファイルは，以下より取得できます．

https://github.com/tkamishima/mlmpy/blob/master/source/nbayes1b.py

実行ファイルも， NaiveBayes1 クラスを読み込むファイルを変えるだけです．

https://github.com/tkamishima/mlmpy/blob/master/source/run_nbayes1b.py

データファイル vote_filled.tsv を作業ディレクトリに置いて実行すると，以前の run_nbayes1.py と同じ結果が得られます．

3.2 単純ベイズの実装 （2）

単純ベイズ：入門編 の **学習メソッドの実装（1）** 節では，NumPy 配列を単なる多次元配列として利用し，for ループで，訓練データを数え挙げて，単純ベイズの学習を実装しました．ここでは，*np.sum()* 関数などを用いて，NumPy の利点を生かして実装をします．

3.2.1 予測メソッドの実装の準備

それでは，NaiveBayes1 クラスとは，学習メソッドの実装だけが異なる NaiveBayes2 クラスの作成を始めます．コンストラクタや予測メソッドは NaiveBayes1 クラスと共通なので，この NaiveBayes2 クラスも，抽象クラス BaseBinaryNaiveBayes の下位クラスとして作成します．クラスの定義と，コンストラクタの定義は，クラス名を除いて NaiveBayes1 クラスと同じです．

```
class NaiveBayes2(BaseBinaryNaiveBayes):
    """
    Naive Bayes class (2)
    """

    def __init__(self):
        super(NaiveBayes2, self).__init__()
```

学習を行う fit() メソッドも，引数などの定義は NaiveBayes1 クラスのそれと全く同じです．さらに，サンプル数 n_samples などのメソッド内の定数の定義も，**定数の設定** 節で述べたものと共通です．

3.2.2 比較演算を利用したクラスごとの事例数の計算

単純ベイズ：カテゴリ特徴の場合 の式 (4) のクラスの分布のパラメータを求めるために，各クラスごとの事例数を NaiveBayes1 クラスでは，次のように求めていました．

```
nY = np.zeros(n_classes, dtype=int)
for i in range(n_samples):
    nY[y[i]] += 1
```

この実装では，クラスの対応する添え字の要素のカウンタを一つずつ増やしていました．これを，各クラスごとに，現在の対象クラスの事例であったらなら対応する要素のカウンタを一つずつ増やす実装にします．

```
nY = np.zeros(n_classes, dtype=int)
for yi in range(n_classes):
    for i in range(n_samples):
        if y[i] == yi:
            nY[yi] += 1
```

外側のループの添え字 yi は処理対象のクラスを指定し，その次のループの添え字 i は処理対象の事例を指定しています．ループの内部では，対象事例のクラスが，現在の処理対象クラスであるかどうかを，等号演算に

よって判定し，もし結果が真であれば，対応するカウンタの値を一つずつ増やしています．

ユニバーサル関数の利用

このコードの中で，内側のループでは全ての事例について等号演算を適用していますが，これを，ユニバーサル関数の機能を利用してまとめて処理します．等号演算 == を適用すると，次の関数が実際には呼び出されます．

np.equal(x1, x2[, out]) = <ufunc 'equal'>

Return (x1 == x2) element-wise.

この関数は x1 と x2 を比較し，その真偽値を論理型で返します．out が指定されていれば，結果をその配列に格納し，指定されていなければ結果を格納する配列を新たに作成します．

この関数はユニバーサル関数であるため，y == yi を実行すると，配列 y 各要素と，添え字 yi とを比較した結果をまとめた配列を返します．すなわち， y の要素が yi と等しいときには True ，それ以外は False を要素とする配列を返します．

この比較結果を格納した配列があれば，このうち True の要素の数を数え挙げれば，クラスが yi に等しい事例の数が計算できます．この数え挙げには，合計を計算する *np.sum()* を用います．論理型の定数 True は，整数型に変換すると 1 に，もう一方の False は変換すると 0 になります．このことを利用すると，*np.sum()* を y == yi に適用することで，配列 y のうち，その値が yi に等しい要素の数が計算できます．

以上のことを利用して，各クラスごとの事例数を数え挙げるコードは次のようになります．

```
nY = np.empty(n_classes, dtype=int)
for yi in range(n_classes):
    nY[yi] = np.sum(y ==yi)
```

なお，配列 nY は 0 で初期化しておく必要がなくなったので， *np.zeros()* ではなく， *np.empty()* で作成しています．

配列要素の一括処理の試み

コードは簡潔になりましたが，まだクラスについてのループが残っていますので，さらにこれを簡潔に記述できるか検討します．ここで， **対数同時確率の計算** 節の **方針（2）** で紹介した，配列の要素をまとめて処理する技法を利用します．これは，ループの添え字がとりうる値をまとめた配列を *np.arange()* 関数によって作成し，対応する添え字がある部分と置き換えるというものでした．

では，添え字 yi について検討します．この変数は，ループ内で 0 から n_classes - 1 まで変化するので， np.arange(n_classes) により，それらの値をまとめた配列を作成できます．この配列を導入した，クラスごとの事例数の数え挙げのコードは次のようになります．

```
nY = np.sum(y == np.arange(n_classes))
```

しかし，このコードは期待した動作をしません．ここでは， y 内の要素それぞれが， yi 内の要素それぞれと比較され，それらの和が計算されることを期待していました．しかし， y も yi も共に 1 次元の配列であるため，単純に配列の最初から要素同士を比較することになってしまいます．この問題を避けて， y の各要素と yi 内の各要素をそれぞれ比較するには，それぞれの配列を 2 次元にして，ブロードキャスト (broadcasting) という機能を利用する必要があります．次の節では，このブロードキャストについて説明します．

3.3 配列の次元数や大きさの操作

ブロードキャストを紹介する前に， *NumPy* **配列の基礎** で紹介した，NumPy の配列クラス *np.ndarray* の属性 ndim と shape を操作する方法を紹介します．

ndim は，配列の次元数を表す属性で，ベクトルでは 1 に，配列では 2 になります．shape は，スカラーや，タプルによって配列の各次元の大きさを表す属性です．例えば，大きさが 5 のベクトルはスカラー 5 によって， 2×3 の行列はタプル (2, 3) となります．

次元数を操作する必要がある例として配列の転置の例を紹介します．転置した配列を得るには，属性 T か，メソッド transpose() を用います．2 次元の配列である行列を転置してみましょう：

```
In [10]: a = np.array([[1, 3], [2, 1]])
In [11]: a
Out[11]:
array([[1, 3],
       [2, 1]])
In [12]: a.T
Out[12]:
array([[1, 2],
       [3, 1]])
In [13]: a.transpose()
Out[13]:
array([[1, 2],
       [3, 1]])
```

今度は，1 次元配列であるベクトルを転置してみます：

```
In [14]: b = np.array([10, 20])
In [15]: b
Out[15]: array([10, 20])
In [16]: b.T
Out[16]: array([10, 20])
```

転置しても，縦ベクトルになることはありません．属性 T やメソッド transpose() は，次元数 ndim が 1 以下であれば，元と同じ配列を返します．

3.3.1 np.newaxis による操作

縦ベクトルを得るには次元数や大きさを，転置する前に操作しておく必要があります．それには，定数 np.newaxis を使います[*1][*2]．np.newaxis は，添え字指定の表記の中に用います．元の配列の大きさを維持する次元には : を指定し，新たに大きさが 1 の次元を追加するところには np.newaxis を指定します．

```
In [17]: b
Out[17]: array([10, 20])
In [18]: b.ndim
Out[18]: 1
In [19]: b.shape
Out[19]: (2,)

In [20]: c = b[:, np.newaxis]
In [21]: c
Out[21]:
array([[10],
       [20]])
In [22]: c.ndim
Out[22]: 2
In [23]: c.shape
Out[23]: (2, 1)

In [24]: d = b[np.newaxis, :]
In [25]: d
Out[25]: array([[10, 20]])
In [26]: d.ndim
Out[26]: 2
In [27]: d.shape
Out[27]: (1, 2)
```

この例で，元の b の ndim は 1 で，その大きさは 2 です．20 行目では，第 0 次元[*3] は元のベクトルをコピーし，第 1 次元には大きさ 1 の新たな次元を追加しています．その結果，c の shape は (2, 1) となり，2×1 行列，すなわち縦ベクトルになっています．24 行目では，第 0 次元の方に新たな次元を追加し，第 1 次元は元ベクトルをコピーしており，その結果，配列 d の shape は (1, 2) となります．これは，1×2 行列，すなわち横ベクトルとなっています．

これら縦ベクトル c と横ベクトル d はそれぞれ 2 次元の配列，すなわち行列なので，次のように転置することができます．

```
In [28]: c.T
Out[28]: array([[10, 20]])
In [29]: d.T
```

[*1] np.newaxis の実体は None であり， np.newaxis の代わりに None と書いても全く同じ動作をします．ここでは，記述の意味を明確にするために， np.newaxis を用います．

[*2] 他にも np.expand_dims() や np.atleast_3d() などの関数を使う方法もありますが，最も自由度の高い np.newaxis を用いる方法を紹介します．

[*3] shape で示されるタプルの一番左側から第 0 次元，第 1 次元，… となります．

```
Out[29]:
array([[10],
       [20]])
```

転置により，縦ベクトル c は横ベクトルに，横ベクトル d は縦ベクトルになっています．

np.newaxis は，2 次元以上の配列にも適用できます．

```
In [30]: e = np.array([[1, 2, 3], [2, 4, 6]])
In [31]: e.shape
Out[31]: (2, 3)
In [32]: e[np.newaxis, :, :].shape
Out[32]: (1, 2, 3)
In [33]: e[:, np.newaxis, :].shape
Out[33]: (2, 1, 3)
In [34]: e[:, :, np.newaxis].shape
Out[34]: (2, 3, 1)
```

np.newaxis の挿入位置に応じて，大きさ 1 の新しい次元が shape に加わっていることが分かります．また，同時に 2 個以上の新しい次元を追加することも可能です．

```
In [35]: e[np.newaxis, :, np.newaxis, :].shape
Out[35]: (1, 2, 1, 3)
```

3.3.2 reshpe() による操作

ブロードキャストとは関連がありませんが，shape を変更する他の方法として *np.ndarray* の reshape() メソッドと，関数 *np.reshape()* をここで紹介しておきます．

np.**reshape**(*a*, *newshape*)

Gives a new shape to an array without changing its data.

この関数は，配列 a 全体の要素数はそのままで，その shape を newshape で指定したものに変更するものです．同様の働きをする reshape() メソッドもあります．

```
In [35]: np.arange(6)
Out[35]: array([0, 1, 2, 3, 4, 5])
In [36]: np.reshape(np.arange(6), (2, 3))
Out[36]:
array([[0, 1, 2],
       [3, 4, 5]])
In [37]: np.arange(6).reshape((3, 2))
Out[37]:
array([[0, 1],
       [2, 3],
       [4, 5]])
```

この例では，6 個の要素を含む shape が (6,) の配列を，それぞれ *np.reshape()* 関数で (2, 3) に， reshape() メソッドで (3, 2) に shape を変更しています．ただし， *np.reshape()* 関数や，reshape() メソッドでは，配列の総要素数を変えるような変更は指定できません．

```
In [38]: np.arange(6).reshape((3, 3))
ValueError: total size of new array must be unchanged
```

この例では，総要素数が 6 個の配列を，総要素数が 9 個の shape (3, 3) を指定したためエラーとなっています．

配列の総要素数が不明の場合は，大きさが不明な次元で -1 を指定すると適切な値が自動的に設定されます．

```
In [39]: np.arange(6).reshape((2, -1))
Out[39]:
array([[0, 1, 2],
       [3, 4, 5]])
```

この例では，次元 1 に -1 を指定すると，全体の要素数が 6 で，次元 0 の大きさに 2 なので，自動的に次元 1 の大きさが 3 に設定されています．

この記法は， shape に (1, -1) や (-1, 1) を指定すると，それぞれ 2 次元の横ベクトルや縦ベクトルを簡便に作ることができます．

```
In [40]: np.arange(6).reshape((1, -1))
Out[40]: array([[0, 1, 2, 3, 4, 5]])
In [41]: np.arange(6).reshape((-1, 1))
Out[41]:
array([[0],
       [1],
       [2],
       [3],
       [4],
       [5]])
```

3.4 ブロードキャスト

それでは，いよいよ本題のブロードキャストの説明に移ります．ブロードキャストとは， ndim や shape が異なる入力配列の間で，これらの属性を自動的に統一する機能です．ndim と shape を統一することで，それらの入力配列間で要素ごとの演算が可能になり，その次元数と大きさが統一された出力配列に演算結果を得ることができます．この機能により，例えば，次元数や大きさの異なる配列 a， b，および c があったとき，これらの配列の要素ごとの和を，明示的に変換を指示しなくても a + b + c のように簡潔な形で書けるようになります．

公式サイトのブロードキャストの規則 は次のとおりです．

1. 次元数 ndim が最大の入力配列より次元数 ndim が小さい全ての入力配列は， shape の先頭に 1 を加

えて次元数を統一する．

2. 出力配列の shape の各次元の大きさは，入力配列のその次元の大きさのうちの最大のものにします．
3. 入力配列の各次元の大きさが，出力配列の対応する次元の大きさと一致するか，1 である場合にその入力配列を計算で利用できます．
4. 入力配列のある次元の大きさが 1 であるとき，その最初の要素の値をその次元の全ての計算で利用します．すなわち，ユニバーサル関数の読み出し機構は，その軸では読み出し位置を動かしません（その次元の移動幅を 0 にします）．

これらの規則のうち，第 1 規則は次元数の統一に関するもので，それ以外は要素間の対応付けに関するものです．それぞれについて順に説明します．

3.4.1 次元数の統一

第 1 規則は，出力配列の次元数 ndim の値を決定し， ndim に変更のあった入力配列の大きさ shape を変更する方法を定めるものです．出力配列の ndim は，入力配列のうち最大のものになります．例えば，0 次元のスカラー，1 次元のベクトル，そして 2 次元の行列の三つの入力配列があったとき，出力配列の ndim はこれらの中で最大の 2 となります．

ndim を増やした入力配列では， shape の先頭，すなわち第 0 次元の位置に，大きさ 1 の次元を，必要な数だけ追加します．例えば，入力配列が，次のスカラー a ，ベクトル b ，そして行列 c の三つである場合，

```
In [38]: a = np.array(100)
In [39]: a
Out[39]: array(100)
In [40]: b = np.array([10, 20, 30])
In [41]: b
Out[42]: array([10, 20, 30])
In [43]: c = np.array([[1, 2, 3], [4, 5, 6]])
In [44]: c
Out[44]:
array([[1, 2, 3],
       [4, 5, 6]])
```

ndim は全て 2 に統一され， shape は次のようになります．

入力配列	統一前 shape	統一後 shape
a	()	(1, 1)
b	(3,)	(1, 3)
c	(2, 3)	(2, 3)

0 次元の a では， shape の先頭に 1 を 2 個追加して，全ての次元で大きさが 1 の shape になります．1 次元の b では，元の大きさ 3 の第 0 次元の前に，大きさ 1 の次元を挿入します．すなわち，統一後に ndim が 2 になる行列では，ベクトルは横ベクトルとなります．2 次元の c は， ndim は変更されないので， shape も変更されません．このように自動的に次元数を統一する機構が備わっていますが，コードが分かりにくくなる

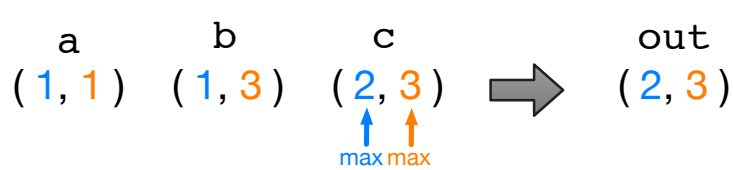
こともよくあります．そのため，実用的には，後述の **クラスの分布の学習** の例のように， `np.newaxis` などを用いて明示的に次元数を統一して利用することをお薦めします．

3.4.2 要素の対応付け

ここまでで，次元数を統一し， `shape` を修正しました．その後は，第 2 規則で出力配列の `shape` を決定し，第 3 規則で演算要素の対応付けが可能かどうかを判定し，第 4 規則で実際の演算でどの要素対応付けるかを決定します．

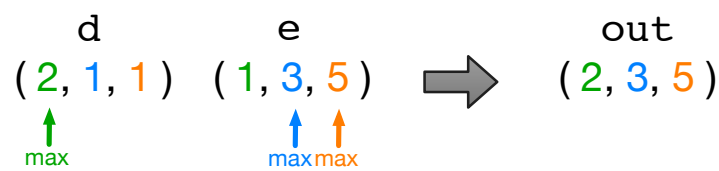
出力配列の shape の決定

第 2 規則により，入力配列の `shape` の同じ次元の配列の大きさを比較し，そのうち最大のものを出力配列のその次元の大きさとします．例えば，上記の入力配列 `a`， `b`，および `c` の場合は次の図のようになります．



青色の第 0 次元の大きさを比較すると `a` では 1， `b` も 1，そして `c` も 2 なので，これら三つのうちで最大の 2 が出力配列の第 0 次元の大きさとなります．同様にオレンジ色の第 1 次元では， `c` の 3 が最大なので，出力配列の第 1 次元の大きさは 3 となります．よって，出力配列の `shape` は `(2, 3)` となります．

3 次元以上の配列についても同様です．例えば， `shape` が `(2, 1, 1)` の配列 `d` と `(1, 3, 5)` の `e` があった場合には，次の図のように出力配列の `shape` は `(2, 3, 5)` となります．



ブロードキャスト可能性の判定

第 3 規則により，出力配列と各入力配列の `shape` を比較し，要素の対応付けが可能かどうかを判定します．全ての入力配列の，全ての次元で，その大きさが 1 であるか，もしくはその次元の大きさが出力配列の対応する次元と等しい場合にブロードキャスト可能 (broadcastable) であるといい，要素の対応付けが可能となります．

上記の `a`， `b`，および `c` の例では， `a` の `shape` は `(1, 1)` で，どの次元でも大きさが 1 なのでブロードキャスト可能です．`b` の `shape` は `(1, 3)` で，出力配列の `shape` `(2, 3)` と比較すると， `b` の第 0 次元の大きさは 1 なのでブロードキャスト可能性の条件を満たし，第 1 次元の 3 も出力配列の第 1 次元の大きさ 3 と等しいのでやはり条件を満たすためブロードキャスト可能です．`c` の `shape` は出力配列のそれと同じなのでブロードキャスト可能です．よって，全ての入力配列がブロードキャスト可能なため，全体でもブロードキャスト可能となります．

同様に，配列 d と e の例でも， e の shape (2, 1, 1) は出力配列の shape (2, 3, 5) と比べると，大きさは第 0 次元は一致し，入力配列のその他の次元では 1 なのでブロードキャスト可能です．e の shape (1, 3, 5) は第 0 次元では大きさが 1 で，その他は出力配列と一致するためブロードキャスト可能です．よって全ての入力配列がブロードキャスト可能なので，全体でもブロードキャスト可能になります．

ブロードキャスト可能でない例も挙げておきます．入力配列の shape が (1, 2, 5) で，出力配列の shape が (3, 3, 5) のときには，第 0 次元と第 2 次元はブロードキャスト可能性の条件を満たします．しかし，入力配列の第 1 次元の大きさは 2 であり，これは 1 でもなく，かつ出力配列の第 1 次元の大きさ 3 とも一致しないため条件を満たさないので，ブロードキャスト可能ではありません．**配列要素の一括処理の試み** 節の例は， y の shape が (n_samples,) であるのに対し，np.arange(n_classes) の shape は (n_classes,) ですので，出力配列の shape は一般に (n_samples,) となります．すると，np.arange(n_classes) の大きさは 1 でもなく，出力配列の大きさとも一致しないためブロードキャスト可能でなくなり，この実装は動作しませんでした．

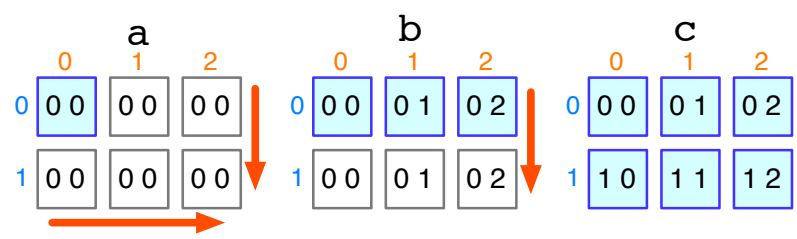
ブロードキャストの実行

最後の第 4 規則により，各入力配列の要素を対応付けます．この要素の対応付けは，次元数統一後の入力配列の：attr:*shape* に基づいて，次のように行います：

- 入力配列のある次元の大きさが，出力配列のそれと一致している場合では，統合後の入力配列の：attr:*shape* のその次元の大きさはそのまま変わりません．
- 入力配列のある次元の大きさが 1 である場合では，その一つの要素を，その次元では全ての演算で利用し続けます．

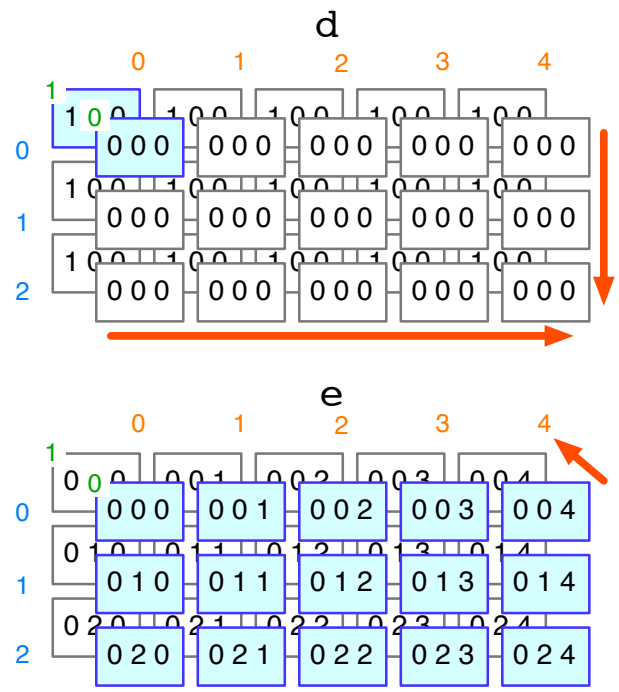
後者の場合，一つの要素の値を，その次元の全ての要素にブロードキャストして（拡散して）用いるので，この配列の自動操作の仕組みはブロードキャストと呼ばれています．

この規則について，上記の三つの配列 a， b，および c を例にとり，次の図を用いて説明します．



この図では，第 0 次元は行数，第 1 次元は列数に対応しています．出力配列の shape は (2, 3) であるため，どの入力配列もこの shape に統一されます．次元数統一後の a の shape は (1, 1) でしたが，これは図中の a の青色の箱で表示しています．第 0 次元の大きさは 1 なので，1 行目の値が 2 行目でも利用されます．第 1 次元の大きさも 1 なので，1 列目の値が 2 列目以降でも利用されます．よって， a では， a[0, 0] の値が，全ての要素に対する演算で利用されます．b では，第 0 次元の大きさは 1 なので，第 1 行目の値が第 2 行目でも利用されますが，第 1 次元の大きさは出力配列と同じ大きさなのでそれぞれの列の値が利用されます．よって， b では，第 0 行目の値が第 1 行目の演算でも利用されます．c は，第 0 次元と第 1 次元のどちらでも，その大きさは出力配列と等しいので，それぞれの要素の値がそのまま演算で利用されます．このように，全ての入力配列の shape を統一すれば，あとは同じ位置の要素ごとに演算ができます．

d と e の場合についても，次の図を用いて説明します．



この図では，第 0 次元は手前から奥に増加し，第 1 次元が行，第 2 次元が列に対応しています．この例では，出力配列の shape は (2, 3, 5) でした．次元数統一後の d の shape は (2, 1, 1) であるため，第 0 次元のみ要素の値をそのまま用い，第 1 次元と第 2 次元では d[:, 0, 0] の値を演算に用います．すなわち，手前と奥の両方のそれぞれ行列で，左上の要素の値を用いて演算します．e の shape は (1, 3, 5) であるため，今度は第 0 次元では手前の行列の値を奥の行列で用いて，他の次元ではその要素の値をそのまま用いて演算します．すなわち，手前の配列を奥の配列にあたかも複製して利用するようになります．こうして，入力配列の shape を統一した後は，要素ごとに演算ができるようになります．

以上で，配列の shape を操作する方法と，形式的なブロードキャストの規則とを説明しました．次の節では，ブロードキャストを利用して，分布の計算を実装します．

3.5 クラスの分布の学習

前節のブロードキャストの機能を用いると， for ループを用いなくても，複数の要素に対する演算をまとめて行うことができます．この節では，その方法を，単純ベイズの学習でのクラスの分布の計算の実装を通じて説明します．例として， **比較演算を利用したクラスごとの事例数の計算** 節で紹介した，クラス分布の計算の比較演算を用いた次の実装を，ブロードキャスト機能を用いた実装に書き換えます．

```
nY = np.zeros(n_classes, dtype=int)
for yi in range(n_classes):
    for i in range(n_samples):
        if y[i] == yi:
            nY[yi] += 1
```

3.5.1 書き換えの一般的な手順

for ループによる実装をブロードキャストを用いて書き換える手順について，多くの人が利用している方針は見当たりません．そこで，ここでは著者が採用している手順を紹介します．

1. 出力配列の次元数を for ループの数とします．
2. 各 for ループごとに，出力配列の次元を割り当てます．
3. 計算に必要な配列の生成します．このとき，ループ変数がループに割り当てた次元に対応するようにします．
4. 冗長な配列を整理統合します．
5. 要素ごとの演算をユニバーサル関数の機能を用いて実行します．
6. *np.sum()* などの集約演算を適用して，最終結果を得ます．

それでは，上記のコードを例として，これらの手順を具体的に説明します．

3.5.2 ループ変数の次元への割り当て

手順の段階 1 と 2 により，各ループを次元に割り当てます．例題のコードでは， for ループは 2 重なので，出力配列の次元数を 2 とします．ループは外側の yi と内側の i の二つで，これらに次元を一つずつ割り当てます．ここでは，第 0 次元に i のループを，第 1 次元に yi のループを割り当てておきます．表にまとめると次のようになります．

次元	ループ変数	大きさ	意味
0	i	n_samples	事例
1	yi	n_classes	クラス

3.5.3 計算に必要な配列の生成

段階 3 では，要素ごとの演算に必要な配列を生成します．for ループ内で行う配列の要素間演算は次の比較演算です．

```
y[i] == yi
```

左辺の y[i] と，右辺の yi に対応する配列が必要になります．これらについて，段階 2 で割り当てた次元にループ変数対応するようにした配列を作成します．

左辺の y[i] では，ループ変数 i で指定した位置の配列 y の値が必要になります．このループ変数 i に関するループを見てみます．

```
for i in range(n_samples):
```

このループ変数 i は 0 から n_samples - 1 までの整数をとります．これらの値を含む配列は np.arange(n_samples) により生成できます．次に，これらの値が，ループ変数 i に段階 2 で割り当てた次元 0 の要素になり，他の次元の大きさは 1 になるようにします．これは，**配列の次元数や大きさの操作** で紹介した shape の操作技法を用いて次のように実装できます．

```
ary_i = np.arange(n_samples)[:, np.newaxis]
```

第 0 次元の : により，np.arange(n_samples) の内容を第 0 次元に割り当て，第 1 次元は np.newaxis により大きさ 1 となるように設定します．

ループ変数 i で指定した位置の配列 y の値 y[i] は次のコードにより得ることができます．

```
ary_y = y[ary_i]
```

このコードにより， ary_i と同じ shape で，その要素が y[i] であるような配列を得ることができます．

右辺のループ変数 yi についての次のループも同様に処理します．

```
for yi in range(n_classes):
```

この変数は 0 から n_classes - 1 までの整数をとり，第 1 次元に割り当てられているので，この変数に対応する配列は次のようになります．

```
ary_yi = np.arange(n_classes)[np.newaxis, :]
```

第 0 次元には大きさ 1 の次元を設定し，第 1 次元の要素には np.arange(n_classes) の内容を割り当てています．以上で，比較演算に必要な配列 ary_y と ary_yi が得られました．

3.5.4 冗長な配列の整理

段階 4 では，冗長な配列を整理します．ary_y は， ary_i を展開すると次のようになります．

```
ary_y = y[np.arange(n_samples)[:, np.newaxis]]
```

配列の shape を変えてから y 中の値を取り出す代わりに，先に y の値を取り出してから shape を変更するようにすると次のようになります．

```
ary_y = (y[np.arange(n_samples)])[:, np.newaxis]
```

ここで y の大きさは n_samples であることから， y[np.arange(n_samples)] は y そのものです．このことをふまえると ary_y は，次のように簡潔に生成できます．

```
ary_y = y[:, np.newaxis]
```

以上のことから， ary_i を生成することなく目的の ary_y を生成できるようになりました．

この冗長なコードの削除は次のループの書き換えと対応付けて考えると分かりやすいかもしれません．次のループ変数 i を使って y 中の要素を取り出すコード

```
for i in range(n_samples):
    val_y = y[i]
```

は， for ループで y の要素を順に参照する次のコードと同じ val_y の値を得ることができます．

```
for val_y in y:
    pass
```

これらのコードは，それぞれ，ループ変数配列を用いた y[ary_i] と y の値を直接参照する y[:, np.newaxis] とに対応します．

3.5.5 要素ごとの演算と集約演算

段階 5 では要素ごとの演算を行います．元の実装では要素ごとの演算は y[i] == yi の比較演算だけでした．この比較演算を，全ての i と yi について実行した結果をまとめた配列は次のコードで計算できます．

```
cmp_y = (ary_y == ary_yi)
```

ary_y と ary_yi の shape はそれぞれ (n_samples, 1) と (1, n_classes) で一致していません．しかし，ブロードキャストの機能により， ary_y[:, 0] の内容と， ary_yi[0, :] の内容を，繰り返して比較演算利用するため，明示的に繰り返しを記述しなくても目的の結果を得ることができます．

最後の段階 6 は集約演算です．集約 (aggregation) とは，複数の値の代表値，例えば総和，平均，最大などを求めることです．**ユニバーサル関数の利用** で述べたように，比較結果が真である組み合わせは *np.sum()* によって計算できます．ここで問題となるのは，単純に np.sum(cmp_y) とすると配列全体についての総和になってしまいますが，計算したい値は yi がそれぞれの値をとるときの，全ての事例についての和でであることです．そこで， *np.sum()* 関数の axis 引数を指定します．ここでは，事例に対応するループ変数 i を次元 0 に割り当てたので， axis=0 と指定します．

```
nY = np.sum(cmp_y, axis=0)
```

以上の実装をまとめて書くと次のようになります．

```
ary_y = y[:, np.newaxis]
ary_yi = np.arange(n_classes)[np.newaxis, :]
cmp_y = (ary_y == ary_yi)
nY = np.sum(cmp_y, axis=0)
```

途中での変数への代入をしないようにすると，次の 1 行のコードで同じ結果を得ることができます．

```
nY = np.sum(y[:, np.newaxis] == np.arange(n_classes)[np.newaxis, :],
            axis=0)
```

3.5.6 クラスの確率の計算

NaiveBayes1 の実装では，各クラスごとの標本数 nY を，総標本数 n_samples で割って，クラスの確率を計算しました．

```
self.pY_ = np.empty(n_classes, dtype=float)
for i in range(n_classes):
    self.pY_[i] = nY[i] / n_samples
```

この処理も，除算演算子 / にユニバーサル関数の機能があるため次のように簡潔に実装できます[*1]．

```
self.pY_ = nY / n_samples
```

3.6 特徴の分布の学習

クラスの分布と同様に，**特徴の分布の学習** の特徴の分布もブロードキャストの機能を用いて実装します．特徴ごとの事例数を数え上げる NaiveBayes1 の実装は次のようなものでした．

```
nXY = np.zeros((n_features, n_fvalues, n_classes), dtype=int)
for i in range(n_samples):
    for j in range(n_features):
        nXY[j, X[i, j], y[i]] += 1
```

クラスの分布の場合と同様に，各特徴値ごとに，対象の特徴値の場合にのみカウンタを増やすような実装にします．

```
nXY = np.zeros((n_features, n_fvalues, n_classes), dtype=int)
for i in range(n_samples):
    for j in range(n_features):
        for yi in range(n_classes):
            for xi in range(n_fvalues):
                if y[i] == yi and X[i, j] == xi:
                    nXY[j, xi, yi] += 1
```

それでは，この実装を，特徴の分布と同様に書き換えます．

[*1] 整数だけでなく浮動小数点に対する除算でも，切り捨てした整数で除算の結果を得るには np.floor_divide() を用います．

np.floor_divide(x1, x2[, out]) = <ufunc 'floor_divide'>

Return the largest integer smaller or equal to the division of the inputs.

Python2 では除算の結果は，整数同士の場合では，結果は切り捨てした整数でした．そのため，整数同士の除算で実数の結果を得たい場合には np.true_divide() 関数を用います．

np.true_divide(x1, x2[, out]) = <ufunc 'true_divide'>

Returns a true division of the inputs, element-wise.

3.6.1 ループ変数の次元への割り当て

まず，ループ変数は i， j， yi，および xj の四つがあります．よって，出力配列の次元数は 4 とし，各ループ変数を次元に次のように割り当てます．

次元	ループ変数	大きさ	意味
0	i	n_samples	事例
1	j	n_features	特徴
2	xi	n_fvalues	特徴値
3	yi	n_classes	クラス

この割り当てで考慮すべきは，最終結果を格納する nXY です．この変数 nXY の第 0 次元は特徴，第 1 次元は特徴値，そして第 3 次元はクラスなので，この順序は同じになるように割り当てています[*1]．最後に集約演算をしたあとに，次元の入れ替えも可能ですが，入れ替えが不要で，実装が簡潔になるように予め割り当てておきます．

3.6.2 計算に必要な配列の生成

ループ内での要素ごとの演算は y[i] == yi and X[i, j] == xi です．よって，必要な配列は y[i]，yi，X[i, j]，および xi となります．

ループ変数 yi と xi に対応する配列は次のようになります．

```
ary_xi = np.arange(n_fvalues)[np.newaxis, np.newaxis, :, np.newaxis]
ary_yi = np.arange(n_classes)[np.newaxis, np.newaxis, np.newaxis, :]
```

y[i] は， **クラスの分布の学習** の場合とは，次元数とループの次元への割り当てが異なるだけです．ループ変数 i は第 0 次元に対応するので，これに対応する変数は次のとおりです．

```
ary_i = np.arange(n_samples)[:, np.newaxis, np.newaxis, np.newaxis]
```

すると， y[i] に対応する配列は次のようになります．

```
ary_y = y[ary_i]
```

これは， **クラスの分布の学習** の場合と同様に次のように簡潔に実装できます．

```
ary_y = y[:, np.newaxis, np.newaxis, np.newaxis]
```

この実装では，全事例の y の値を，事例に対応する第 0 次元に割り当て，その他の次元の大きさを 1 である配列を求めています．

[*1] もしも軸の順序を揃えることができない場合は， *np.swapaxes()* 関数を用いて次元の順序を入れ換えます．

np.**swapaxes**(*a*, *axis1*, *axis2*)

Interchange two axes of an array.

X[i, j] はループ変数を 2 個含んでいるので，これまでとは状況が異なります．X[ary_ij] のような形式で，2 個以上のインデックスを含み，かつ np.newaxis による次元の追加が可能な ary_ij の作成方法を著者は知りません[*2]．そこで，ループ変数の値に対応した配列を考えず，X の要素を，ループを割り当てた次元に対応するように配置した配列を直接的に生成します．これは，全事例の X[:, j] の値を，事例に対応する第 0 次元に，そして全特徴の X[i, :] の値を，特徴に対応する第 1 次元に割り当て，その他の第 2 と第 3 次元の大きさを 1 にした配列となります．すなわち，ループ変数 xi と yi に対応する次元を X に追加します．

```
ary_X = X[:, :, np.newaxis, np.newaxis]
```

以上で演算に必要な値を得ることができました．

3.6.3 要素ごとの演算と集約演算

y[i] == yi and X[i, j] == xi の式のうち，比較演算を実行します．y[i] == yi と X[i, j] == xi に対応する計算は， == がユニバーサル関数なので，次のように簡潔に実装できます．

```
cmp_X = (ary_X == ary_xi)
cmp_y = (ary_y == ary_yi)
```

次にこれらの比較結果の論理積を求めますが， and は Python の組み込み関数で，ユニバーサル関数ではありません．そこで，ユニバーサル関数である np.logical_and() を用います[*3]．

np.logical_and(x1, x2[, out]) = <ufunc 'logical_and'>

Compute the truth value of x1 AND x2 elementwise.

実装は次のようになります．

```
cmp_Xandy = np.logical_and(cmp_X, cmp_y)
```

最後に，全ての事例についての総和を求める集約演算を行います．総和を求める *np.sum()* を，事例に対応する第 0 次元に適用します[*4]．

```
nXY = np.sum(cmp_Xandy, axis=0)
```

以上の配列の生成と，演算を全てをまとめると次のようになります．

[*2] もし np.newaxis による次元の追加が不要であれば， np.ix_() を用いて， ary_ij = np.ix_(np.arange(n_samples), np.arange(n_features)) のような記述が可能です．

np.ix_(*args)[source]

Construct an open mesh from multiple sequences.

[*3] 同様の関数に， or ， not ，および xor の論理演算に，それぞれ対応するユニバーサル関数 np.logical_or() ， np.logical_not() ，および np.logical_xor() があります．

[*4] もし同時に二つ以上の次元について同時に集約演算をする必要がある場合には， axis=(1,2) のようにタプルを利用して複数の次元を指定できます．また， *np.apply_over_axes()* を用いる方法もあります．

np.**apply_over_axes** (*func*, *a*, *axes*)

Apply a function repeatedly over multiple axes.

```
ary_xi = np.arange(n_fvalues)[np.newaxis, np.newaxis, :, np.newaxis]
ary_yi = np.arange(n_classes)[np.newaxis, np.newaxis, np.newaxis, :]
ary_y = y[:, np.newaxis, np.newaxis, np.newaxis]
ary_X = X[:, :, np.newaxis, np.newaxis]

cmp_X = (ary_X == ary_xi)
cmp_y = (ary_y == ary_yi)
cmp_Xandy = np.logical_and(cmp_X, cmp_y)

nXY = np.sum(cmp_Xandy, axis=0)
```

そして，中間変数への代入を整理します．

```
ary_xi = np.arange(n_fvalues)[np.newaxis, np.newaxis, :, np.newaxis]
ary_yi = np.arange(n_classes)[np.newaxis, np.newaxis, np.newaxis, :]
ary_y = y[:, np.newaxis, np.newaxis, np.newaxis]
ary_X = X[:, :, np.newaxis, np.newaxis]

nXY = np.sum(np.logical_and(ary_X == ary_xi, ary_y == ary_yi), axis=0)
```

以上で，各特徴，各特徴値，そして各クラスごとの事例数を数え上げることができました．

3.6.4 特徴値の確率の計算

最後に nXY と，クラスごとの事例数 nY を用いて，クラスが与えられたときの，各特徴値が生じる確率を計算します．それには nXY を，対応するクラスごとにクラスごとの総事例数 nY で割ります．nY を nXY と同じ次元数にし，そのクラスに対応する第 2 次元に割り当てるようにすると nY[np.newaxis, np.newaxis, :] となります．あとは，除算演算子 / を適用すれば，特徴値の確率を計算できます[*5]．

```
self.pXgY_ = nXY / nY[np.newaxis, np.newaxis, :]
```

計算済みの nY を使う代わりに，ここで総和を計算する場合は次のようになります．

```
self.pXgY_ = nXY / nXY.sum(axis=1, keepdims=True)
```

通常の sum() では総和の対象とした次元は消去されるため，元の配列とはその大きさが一致しなくなります．そこで，keepdims=True の指定を加えることで元の配列の次元が維持するようにすると，そのまま割り算できるようになります．確率の計算では，総和が 1 になるような正規化は頻繁に行うので，この記述は便利です．

[*5] Python2 ではこの除算にはユニバーサル関数の実数除算関数 np.true_divide() を用いる必要があります．

3.6.5 実行

以上の，ブロードキャスト機能を活用した訓練メソッド fit() を実装した NaiveBayes2 と，その実行スクリプトは，以下より取得できます．この NaiveBayes2 クラスの実行可能な状態のファイルは

https://github.com/tkamishima/mlmpy/blob/master/source/nbayes2.py

であり，実行ファイルは

https://github.com/tkamishima/mlmpy/blob/master/source/run_nbayes2.py

です．実行すると， NaiveBayes1 と NaiveBayes2 で同じ結果が得られます．

3.7 実行速度の比較

便利な実行環境である ipython を用いて，二つのクラス NaiveBayes1 と NaiveBayes2 の訓練の実行速度を比較します．そのために， ipython を起動し，訓練に必要なデータを読み込みます．

```
In [10]: data = np.genfromtxt('vote_filled.tsv', dtype=int)
In [11]: X = data[:, :-1]
In [12]: y = data[:, -1]
```

次に，クラスを読み込み，単純ベイズ分類器を実装した二つクラス NaiveBayes1 と NaiveBayes2 のインスタンスを生成します．

```
In [13]: from nbayes2 import *
In [14]: clr1 = NaiveBayes1()
In [15]: clr2 = NaiveBayes2()
```

最後に， %timeit コマンドを使って，訓練メソッド fit() の実行速度を測ります．

```
In [10]: %timeit clr1.fit(X, y)
100 loops, best of 3: 16.2 ms per loop

In [11]: %timeit clr2.fit(X, y)
1000 loops, best of 3: 499 us per loop
```

実行速度を見ると NaiveBayes1 の 16.2 ミリ秒に対し， NaiveBayes2 では 499 マイクロ秒と，後者が 32.5 倍も高速です．for ループを用いた実装では Python のインタプリタ内で実行されるのに対し， NumPy で配列の演算を用いて実装すると，ほとんどがネイティブコードで実行されるため非常に高速になります．このようにブロードキャストを活用した実装は，コードが簡潔になるだけでなく，実行速度の面でも有利になります．

3.8 「単純ベイズ：上級編」まとめ

単純ベイズ：上級編 の章では，単純ベイズ法の実装を改良することで，以下の内容を紹介しました．

- **クラスの再編成**
  - 抽象クラスを用いて，実装の一部だけを変更したクラスを設計する方法
- **単純ベイズの実装 （2）**
  - 比較演算を行うユニバーサル関数
  - *np.sum()* を用いた数え上げの方法
- **配列の次元数や大きさの操作**
  - `np.newaxis` による，配列の次元数と `shape` の変更
  - `reshape()` メソッドや *np.reshape()* 関数による `shape` の変更
  - `T` 属性や `np.transepose()` 関数による行列の転置
- **ブロードキャスト**
  - ブロードキャスト機能：次元数を統一する規則，出力配列の `shape` の決定方法，ブロードキャスト可能性の判定，および演算要素の対応付け
- **クラスの分布の学習**
  - ブロードキャスト機能を用いた実装例
  - 実数を返す割り算関数 `np.true_divide()`
- **特徴の分布の学習**
  - ブロードキャスト機能を用いた実装例
  - 論理演算のユニバーサル関数 `np.logical_and()`
- **実行速度の比較**
  - `ipython` 内での， `%timeit` コマンドによる関数の実行速度の計測

第 4 章

ロジスティック回帰

ここではロジスティック回帰 (logistic regression) を実装します．この実装を通じて， NumPy / SciPy の特徴を利用した数学関数の実装とエラー処理，最適化ルーチンの利用法，そして構造化配列などについて説明します．

4.1 ロジスティック回帰の形式的定義

ロジスティック回帰を実装する前に，この手法について簡単に復習します．

変数を定義します．**単純ベイズ：カテゴリ特徴の場合** とほぼ同じ表記を用いますが，特徴はカテゴリ値ではなく，実数値をとります．

- 特徴量 $\mathbf{x}_i = (x_{i1}, \ldots, x_{iK})$ の要素 x_{ij} は実変数です．ただし， K は特徴の種類数です．
- クラス y は， $\{0, 1\}$ のうちの一つをとります．
- データ集合は $\mathcal{D} = \{\mathbf{x}_i, y_i\}, i = 1, \ldots, N$ です．ただし， N はデータ数です．

ロジスティック回帰では， $\mathbf{x}$ が与えられたときの y の条件付き確率を次式でモデル化します．

$$\Pr[y{=}1|\mathbf{x}; \mathbf{w}, b] = \mathrm{sig}(\mathbf{w}^\top \mathbf{x} + b) \tag{4.1}$$

ただし， $\mathbf{w}$ は次元数 K の重みベクトル， b は切片（バイアス）と呼ばれるパラメータです．また， $\mathrm{sig}(a)$ は次のシグモイド関数です．

$$\mathrm{sig}(a) = \frac{1}{1 + \exp(-a)}$$

学習には，正則化の度合いを決める超パラメータ λ を導入した次の目的関数を用います．

$$\begin{aligned}
\mathcal{L}(\mathbf{w}, b; \mathcal{D}) &= -\log \sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}} \Pr[y_i|\mathbf{x}_i; \mathbf{w}, b] + \frac{\lambda}{2}\left(\|\mathbf{w}\|_2{}^2 + b^2\right) \\
&= -\sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}} \{(1 - y_i)\log(1 - \Pr[y_i{=}1|\mathbf{x}_i]) + \\
&\qquad y_i \log \Pr[y_i{=}1|\mathbf{x}_i]\} + \frac{\lambda}{2}\left(\|\mathbf{w}\|_2{}^2 + b^2\right)
\end{aligned} \tag{4.2}$$

なお，$\Pr[y|\mathbf{x}]$ 中のパラメータは簡潔のため省略しました．この目的関数は，訓練データ集合 $\mathcal{D}$ に対する負の対数尤度に L_2 正則化項を加えたものです．ロジスティック回帰モデルでの学習は，この目的関数を最小にするパラメータ $\mathbf{w}$ と b を求めることです．

$$\{\mathbf{w}^*, b^*\} = \arg\min_{\{\mathbf{w},b\}} \mathcal{L}(\mathbf{w}, b; \mathcal{D}) \tag{4.3}$$

この最小化問題は反復再重み付け最小二乗法 (iteratively reweighted least squares method) により求めるのが一般的です．しかし，本章では，他の多くの最適化問題として定式化された機械学習手法の実装の参考となるように， SciPy の非線形最適化用の関数を利用して解きます．非線形最適化では目的関数の勾配も利用するので，ここに追記しておきます．

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{w}}\mathcal{L}(\mathbf{w}, b; \mathcal{D}) &= \sum_{(\mathbf{x}_i, y)\in\mathcal{D}} (\Pr[y_i{=}1|\mathbf{x}_i] - y_i)\mathbf{x} + \lambda\,\mathbf{w} \\
\frac{\partial}{\partial b}\mathcal{L}(\mathbf{w}, b; \mathcal{D}) &= \sum_{(\mathbf{x}, y)\in\mathcal{D}} (\Pr[y_i{=}1|\mathbf{x}_i] - y_i) + \lambda\,b
\end{aligned} \tag{4.4}$$

学習したパラメータ $\mathbf{w}^*$ と b^* を式 (4.1) に代入した分布 $\Pr[y|\mathbf{x}; \mathbf{w}^*, b^*]$ 用いて，新規入力データ $\mathbf{x}^{\mathrm{new}}$ に対するクラス y は次式で予測できます．

$$y = \begin{cases} 1, \text{ if } \Pr[y|\mathbf{x}; \mathbf{w}^*, b^*] \geq 0.5 \\ 0, \text{otherwise} \end{cases} \tag{4.5}$$

4.2 シグモイド関数

ここでは **ロジスティック回帰の形式的定義** の式 (1) で用いる次のシグモイド関数を実装します．

$$\mathrm{sig}(a) = \frac{1}{1 + \exp(-a)} \tag{4.6}$$

この関数の実装を通じ，数値演算エラーの扱い，ユニバーサル関数の作成方法，数学関数の実装に便利な関数などについて説明します．

4.2.1 直接的な実装とその問題点

最初に，式 (4.6) をそのまま実装してみます．モジュール `lr1`[*1] 中のロジスティック回帰クラスの定義のうち，ここでは関数 `sigmoid()` の部分のみを示します．

```
@staticmethod
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
```

なお， *@staticmethod* のデコレータを用いて，静的メソッドとして定義してあります．`sigmoid()` は数学関数であり，値はその引数だけに依存し，オブジェクトやクラスの内容や状態には依存しないので，このように静的メソッドとして定義しました．

[*1] https://github.com/tkamishima/mlmpy/blob/master/source/lr1.py

それでは，実行してみましょう．sigmoid() は静的メソッドなので，オブジェクトを生成しなくても実行できます．

```
In [10]: from lr1 import LogisticRegression
In [11]: LogisticRegression.sigmoid(0.0)
Out[11]: 0.5

In [12]: LogisticRegression.sigmoid(1.0)
Out[12]: 0.7310585786300049
In [13]: 1.0 / (1.0 + 1.0 / np.e)
Out[13]: 0.7310585786300049

In [14]: LogisticRegression.sigmoid(-1.0)
Out[14]: 0.2689414213699951
In [15]: 1.0 / (1.0 + np.e)
Out[15]: 0.2689414213699951
```

いずれも正しく計算できています．なお， np.e はネピアの数[*2] を表す定数です．

さらに，いろいろな入力値で試してみます．

```
In [20]: LogisticRegression.sigmoid(1000.)
Out[20]: 1.0

In [21]: LogisticRegression.sigmoid(-1000.)
lr1.py:62: RuntimeWarning: overflow encountered in exp
  return 1.0 / (1.0 + np.exp(-x))
Out[21]: 0.0
```

シグモイド関数は 1.0 や 0.0 といった値になることは，式 (4.6) の定義からはありえません．しかし， NumPy での実数演算は，精度が有限桁の浮動小数点を用いて行っているため，絶対値が大きすぎるオーバーフローや，小さすぎるアンダーフローといった浮動小数点エラーを生じます．そのため，意図したとおりの計算結果を得ることができません．こうした問題を避けるため，浮動小数点演算の制限を意識して数値計算プログラムを実装する必要があります．

浮動小数点エラーの処理

意図した計算結果を得ることができないこの問題の他に，オーバーフローが生じていることの警告メッセージが表示されてしまう問題も起きています．もちろん，この警告メッセージは有用なものですが，浮動小数点エラーを，無視してかまわない場合や，例外として処理したい場合など，警告メッセージ表示以外の動作が望ましい場合もあります．これらの場合には，次の np.seterr() を用いて，浮動小数点演算のエラーに対する挙動を変更できます．

np.seterr(all=None, divide=None, over=None, under=None,

[*2] NumPy には，このネピアの数を表す np.e の他に，円周率を表す np.pi の定数があります．SciPy の sp.consants モジュール内には，光速や重力定数などの物理定数が定義されています．

invalid=None)

Set how floating-point errors are handled.

divide は 0 で割ったときの 0 除算，over は計算結果の絶対値が大きすぎる場合のオーバーフロー，under は逆に小さすぎる場合のアンダーフロー，そして invalid は対数の引数が負数であるなど不正値の場合です．all はこれら全ての場合についてまとめて挙動を変更するときに用います．

そして，np.seterr(all='ignore') のように，キーワード引数の形式で下記の値を設定することで挙動を変更します．

- warn: 警告メッセージを表示するデフォルトの挙動です．
- ignore: 数値演算エラーを無視します．
- raise: 例外 FloatingPointError を送出します．

その他 call， print，および log の値を設定できます．

4.2.2 浮動小数点エラー対策

それでは，シグモイド関数の実装に戻ります．ここでは，シグモイド関数の入力が小さすぎる場合や，大きすぎる場合に処理を分けることで浮動小数点エラーを生じないようにします．シグモイド関数の出力値の範囲を次のような区間に分けて処理することにします．

- 10^{-15} より小さくなる場合では 10^{-15} の定数を出力．
- 10^{-15} 以上 $1-10^{-15}$ 以下の場合では式 (4.6) のとおりの値を出力．
- $1-10^{-15}$ より大きくなる場合では $1-10^{-15}$ の定数を出力．

簡単な計算により， sigmoid_range = 34.538776394910684[*3] とすると，入力値が -sigmoid_range 以上， +sigmoid_range 以下の範囲であれば式 (4.6) に従って計算し，それ以外では適切な定数を出力すればよいことが分かります．これを実装すると次のようになります[*4]．

```
@staticmethod
def sigmoid(x):
    sigmoid_range = 34.538776394910684

    if x <= -sigmoid_range:
        return 1e-15
    if x >= sigmoid_range:
        return 1.0 - 1e-15

    return 1.0 / (1.0 + np.exp(-x))
```

それでは，大きな値や小さな値を入力して試してみます．

[*3] sigmoid_range の具体的な計算式は $\log((1-10^{-15})/10^{-15})$ です．
[*4] https://github.com/tkamishima/mlmpy/blob/master/source/lr2.py

```
In [30]: from lr2 import LogisticRegression
In [31]: LogisticRegression.sigmoid(1000.)
Out[31]: 0.999999999999999
In [32]: LogisticRegression.sigmoid(-1000.)
Out[32]: 1e-15
```

今度は，大きな入力に対しては 1 よりわずかに小さな数，逆に，小さな入力に対しては 0 よりわずかに大きな数が得られるようになりました．こうして，シグモイド関数で浮動小数点エラーを生じないようにすることができました．

4.2.3 ユニバーサル関数への変換

ここでは，シグモイド関数をユニバーサル関数にする方法を紹介します．**対数同時確率の計算** で紹介しましたが， NumPy 配列を引数に与えると，その要素ごとに関数を適用した結果を， `shape` が入力と同じ配列にまとめて返すのがユニバーサル関数です．

前節で作成したシグモイド関数はユニバーサル関数としての機能がありません．このことを確認してみます．

```
In [40]: from lr2 import LogisticRegression
In [41]: x = np.array([ -1.0, 0.0, 1.0 ])
In [42]: LogisticRegression.sigmoid(x)

... omission ...

ValueError: The truth value of an array with more than one element
is ambiguous. Use a.any() or a.all()
```

if 文は配列 `x` の要素を個別に処理できないので，このようにエラーとなってしまいます．

そこで，通常の関数をユニバーサル関数に変換する `np.vectorize()` があります．

`np.vectorize(pyfunc, otypes='', doc=None, excluded=None, cache=False)`

Define a vectorized function which takes a nested sequence of objects or numpy arrays as inputs and returns a numpy array as output. The vectorized function evaluates *pyfunc* over successive tuples of the input arrays like the python map function, except it uses the broadcasting rules of numpy.

この `np.vectorize()` は，通常の関数を入力すると，その関数を，引数の配列の各要素に適用するユニバーサル関数を返す関数です．簡単なステップ関数の例を見てみましょう．

```
 In [50]: def step(x):
...:     return 0.0 if x < 0.0 else 1.0
...:
```

三項演算子は入力配列の要素を個別に処理できないのでこの関数はユニバーサル関数ではありません．そこで次のように `np.vectorize()` を用いてユニバーサル関数に変換します．

```
In [51]: vstep = np.vectorize(step)
In [52]: x = np.arange(7) - 3
In [53]: x
Out[53]: array([-3, -2, -1,  0,  1,  2,  3])
In [54]: vstep(x)
Out[54]: array([ 0.,  0.,  0.,  1.,  1.,  1.,  1.])
```

関数を入力として関数を返す関数は Python のデコレータとして使うことができます．こ機能を利用して，先ほど定義したシグモイド関数の， `@staticmethod` デコレータの下に，関数 `np.vectorize()` を `@np.vectorize` のような形式でデコレータとして与えます[*5]．

```
@staticmethod
@np.vectorize
def sigmoid(x):
    sigmoid_range = 34.538776394910684

    if x <= -sigmoid_range:
        return 1e-15
    if x >= sigmoid_range:
        return 1.0 - 1e-15

    return 1.0 / (1.0 + np.exp(-x))
```

これでユニバーサル関数になっているかを確かめてみます．

```
In [60]: from lr3 import LogisticRegression
In [61]: x = np.array([-1.0, 0.0, 1.0])
In [62]: LogisticRegression.sigmoid(x)
Out[62]: array([ 0.26894142,  0.5       ,  0.73105858])
```

配列 `x` の各要素にシグモイド関数を適用した結果を配列として得ることができました．このようにしてユニバーサル関数を定義することができました．

なお，入力引数が複数の関数をユニバーサル関数にする `np.frompyfunc()` もあります．

`np..`**`frompyfunc`**(*func*, *nin*, *nout*)

Takes an arbitrary Python function and returns a Numpy ufunc.

4.2.4 便利な関数を用いた実装

ここまで，他の数学関数の実装にも使える汎用的な手法を紹介しました．さらに，NumPy にはシグモイド関数の実装に使える便利な関数があり，これらを使って実装することもできます．そうした関数として `np.pieceswise()` と `np.clip()` を紹介します．

[*5] https://github.com/tkamishima/mlmpy/blob/master/source/lr3.py

np.piecewise() は Huber 関数や三角分布・切断分布の密度関数など，入力の範囲ごとに異なる数式でその出力が定義される区分関数を実装するのに便利です．

np.**piecewise**(*x*, *condlist*, *funclist*, **args*, ***kw*)

Evaluate a piecewise-defined function.

浮動小数点エラー対策 で実装したシグモイド関数は，浮動小数点エラーを防ぐために入力の範囲に応じて出力を変えています．*np.piecewise()* を用いて実装したシグモイド関数は次のようになります[*6]．

```
@staticmethod
def sigmoid(x):
    sig_r = 34.538776394910684
    condlist = [x < -sig_r, (x >= -sig_r) & (x < sig_r), x >= sig_r]
    funclist = [1e-15, lambda a: 1.0 / (1.0 + np.exp(-a)), 1.0 - 1e-15]

    return np.piecewise(x, condlist, funclist)
```

np.piecewise() の，第 2 引数は区間を定義する条件のリスト[*7] で，第 3 引数はそれらの区間ごとの出力のリストを定義します．条件のリストで `True` になった位置に対応する出力値が *np.piecewise()* の出力になります．出力リストが条件のリストより一つだけ長い場合は，出力リストの最後はデフォルト値となります．条件リストが全て `False` であるときに，このデフォルト値が出力されます．

`np.clip()` は，区間の最大値大きい入力はその最大値に，逆に最小値より小さい入力はその最小値にする関数です[*8]．シグモイド関数はこの `np.clip()` を用いると容易に実装できます．

```
@staticmethod
def sigmoid(x):
    # restrict domain of sigmoid function within [1e-15, 1 - 1e-15]
    sigmoid_range = 34.538776394910684
    x = np.clip(x, -sigmoid_range, sigmoid_range)

    return 1.0 / (1.0 + np.exp(-x))
```

`np.clip()` 関数は，ユニバーサル関数であるため，特に `np.vectorize()` を用いる必要もありません．この実装を採用した，さらにこのあと説明する他のメソッドを含むロジスティック回帰のモジュールは次のとおりです．以降は，これを用います．

https://github.com/tkamishima/mlmpy/blob/master/source/lr.py

---

*6 複数の条件に対して対応する値を出力する関数は他にも *np.select()* などがあります．

np.**select**(*condlist*, *choicelist*, *default=0*)

Return an array drawn from elements in choicelist, depending on conditions.

しかし，条件が満たされるかどうかに関わらず，全ての場合の出力値を計算するため，この節のシグモイド関数場合は浮動小数点エラーを生じてしまいます．

*7 条件リスト中で `and` や `or` を使うと，これらはユニバーサル関数ではないため，`x` が配列の場合にうまく動作しません．代わりに NumPy の `np.logical_and()` や `np.logical_or()` を使うこともできます．

*8 最大値か最小値の一方だけで十分な場合はそれぞれ `np.min()` や `np.max()` を用います．

4.3 非線形最適化関数

ロジスティック回帰を解くには，**ロジスティック回帰の形式的定義** の式 (3) の非線形最適化問題を解く必要があります．ここでは，この最適化問題を `scipy.optimize` モジュールに含まれる関数 `minimize()` を用いて実装します．そこで，この節では `minimize()` などの最適化関数について俯瞰します．ロジスティック回帰モデルをあてはめるメソッドの実装については，次の **学習メソッドの実装** で述べます．

4.3.1 SciPy の非線形最適化関数

SciPy の非線形最適化関数には，`minimize_scalar()` と `minimize()` があります．これらを順に紹介します．

`sp.optimize.`**`minimize_scalar`**(*fun*, *args=()*, *method='brent'*)

Minimization of scalar function of one variable.

`minimize_scalar()` は，入力パラメータと出力が共にスカラーである目的関数 `fun` の最小値と，そのときのパラメータ値を求めます．関数 `fun` に，最小化するパラメータ以外の引数がある場合には `args` で指定します．最適化の方法は `method` で指定します．通常は，最小化するパラメータの範囲に制約がないときは `brent` を，制約がある場合は `bounded` を指定します．

`sp.optimize.`**`minimize`**(*fun*, *x0*, *args=()*, *method=None*, *jac=None*, *hess=None*, *hessp=None*, *bounds=None*, *constraints=()*, *tol=None*, *options=None*)

Minimization of scalar function of one or more variables.

`minimize()` は入力パラメータがベクトルで出力はスカラーである目的関数 `fun` の最小値と，そのときのパラメータの値を求めるという非線形計画問題を解きます．`x0` で指定したパラメータから解の探索を開始し，そこから到達できる局所解を見つけることができます[*1]．`args` にはパラメータ以外の関数 `fun` への入力，`method` ではあとで紹介する最適化手法を指定します．以降の引数は，最適化手法によって指定が必要かどうかやその意味が変わります．`jac` は目的関数 `fun` の勾配ベクトル（ヤコビベクトル）すなわち，目的関数を入力パラメータの各変数での 1 次導関数を要素とするベクトル返す関数を与えます．`hess` や `hessp` は 2 次導関数を要素とするヘシアンを指定します[*2]．制約付きの最適化手法で `bounds` や `constraints` はパラメータに対する制約条件を指定します．`tol` は終了条件の許容誤差で，必要な精度と計算時間のトレードオフを考慮して選びます[*3]．

`minimize_scalar()` と `minimize()` のいずれも，最適化の結果を次のクラスで返す

[*1] 局所最適解を異なる初期値から探索することを何度も繰り返して大域最適解を求める関数として `sp.optimize.basinhopping()` や `sp.optimize.brute()` が用意されています．

[*2] `hess` は通常のヘシアン，すなわち $f(\mathbf{x})$ の 2 次導関数が特定の値 $\mathbf{a}$ をとったときの行列 $\mathbf{H}(\mathbf{a}) = \left[\frac{\partial^2 f}{\partial x_i \partial x_j}\right]_{ij}\Bigg|_{\mathbf{x}=\mathbf{a}}$ を指定します．しかし，パラメータベクトル $\mathbf{x}$ の次元数が大きいときは，ヘシアンを保持するためには次元数の 2 乗という多くの記憶領域を必要としてしまいます．そのような場合に，`hessp` はヘシアンと特定のベクトル $\mathbf{p}$ との積 $\mathbf{H}(\mathbf{a})\mathbf{p}$ を計算する関数を指定することで記憶領域を節約することができます．

[*3] 非常に小さな値を指定すると，浮動小数点のまるめ誤差などの影響で最適化関数が停止しない場合があります．目安として 10^{-6} より小さな値を指定するときは，この点に注意した方がよいでしょう．

class `sp.optimize.`**`OptimizeResult`**

Represents the optimization result.

変数

- **`fun`** – Values of objective function.
- **`x`** – The solution of the optimization.
- **`success`** – Whether or not the optimizer exited successfully.
- **`nit`** – Number of iterations performed by the optimizer.

`fun` と `x` は，それぞれ関数の最小値と，そのときのパラメータの値です．`success` は最適化が成功したかどうか，`nit` は収束するまでの反復数です．

4.3.2 各種の最適化手法

ロジスティック回帰への適用について述べる前に，`minimize()` の `method` で指定できる最適化手法を一通り紹介します．最適化手法は，パラメータの範囲に制約がない場合とある場合に用いるものに分けられます．

パラメータの範囲に制約がない手法は次のとおりです．

1.**勾配ベクトルやヘシアンが不要**

- `Nelder-Mead` ：Nelder-Mead 法
- `Powell` ：Powell 法

2.**勾配ベクトルのみが必要**

- `CG` ：共役勾配法 (conjugate gradient method)
- `BFGS` ：BFGS 法 (Broyden–Fletcher–Goldfarb–Shanno method)

3.**勾配ベクトルとヘシアンの両方が必要**

- `Newton-CG` ：ニュートン共役勾配法 (Newton conjugate gradient method)
- `trust-ncg`：信頼領域ニュートン共役勾配法 (Newton conjugate gradient trust-region method)
- `dogleg` ：信頼領域 dog-leg 法 (dog-leg trust-region method)

1 から 3 になるにつれ，勾配やヘシアンなど引数として与える関数は増えますが，収束するまでの反復数は減ります．1 の `Nelder-Mead` と `Powell` では，ほとんどの場合で Powell 法が高速です．おおまかにいって，勾配を使う方法と比べて，1 回の反復で必要になる目的関数の評価階数はパラメータ数倍になるため，これらの方法は遅いです．勾配を解析的に計算出来ない場合にのみ使うべきでしょう．

勾配ベクトルのみを使う方法のうち，`BFGS` は近似計算したヘシアンを用いるニュートン法であるので，収束は `CG` に比べて速いです．しかし，ヘシアンの大きさはパラメータ数の 2 乗であるため，パラメータ数が多い

ときには多くの記憶領域と計算量が必要となるため， CG の方が速くなることが多いです．

3 の方法はヘシアンも必要なので，ヘシアンの実装の手間や，その計算に必要な計算量やメモリを考慮して採用してください．

パラメータの範囲に制約のある方法は次のとおりです．

1.**パラメータの範囲に制約がある場合**

- L-BFGS-B ：範囲制約付きメモリ制限 BFGS 法
- TNC ：切断ニュートン共役勾配法

2.**パラメータの範囲の制約に加えて，等式・不等式制約がある場合**

- COBYLA ：COBYLA 法 (constrained optimization by linear approximation method)
- SLSQP ：sequential least squares programming

パラメータの範囲は bounds に，パラメータそれぞれの値の最小値と最大値の対の系列を指定します．等式・不等式制約は， type ， fun ， jac の要素を含む辞書の系列で指定します．type には，等式制約なら文字列定数 eq を，不等式制約なら ineq を指定します．fun には制約式の関数を， jac にはその勾配を指定します．

4.4 学習メソッドの実装

それでは，**非線形最適化関数** で紹介した minimize() を用いて，学習メソッド fit() を実装します．minimize() とパラメータをやりとりするために，構造化配列を用いる方法についても紹介します．

4.4.1 学習メソッド

あてはめをおこなう fit() メソッドでは，まずデータ数と特徴数を設定しておきます．

```
def fit(self, X, y):
    """
    Fitting model
    """

    # constants
    self.n_samples_ = X.shape[0]
    self.n_features_ = X.shape[1]
```

そして，最適化関数 mimimze() で最適なパラメータを求めます．

```
# optimize
res = minimize(fun=self.loss,
               x0=np.zeros(self.n_features_ + 1, dtype=float),
```

```
                jac=self.grad_loss,
                args=(X, y),
                method='CG')
```

minimize() を呼び出して，ロジスティック回帰モデルをあてはめて，その結果を:class:*OptimizeResult* のインスタンスとして受け取り， :obj‘res‘ に保持しています．

最適化手法には method で CG，すなわち共役勾配降下法を指定しました．minimize() の引数 fun と jac には，それぞれロジスティック回帰の目的関数とその勾配ベクトル，すなわち **ロジスティック回帰の形式的定義** の式 (2) と式 (4) を計算するメソッドを与えています．これらのメソッドについては次の **損失関数とその勾配** で詳しく述べます．

最適解を探索する初期パラメータ x0 には *np.zeros()* で生成した実数の 0 ベクトルを与えています．目的関数のパラメータ配列の大きさは，この初期パラメータの大きさになります．ここでは，重みベクトル $\mathbf{w}$ の次元数，すなわち特徴数に，切片パラメータ (intercept) b のための 1 を加えた数にしています．

目的関数と勾配ベクトルを計算するにはモデルのパラメータの他にも訓練データの情報が必要です．そこで，これらの情報を args に指定して，目的関数・勾配ベクトルを計算するメソッドに引き渡されるようにしています．

最適化が終わったら， OptimizeResult のインスタンスである res の属性 x に格納されているパラメータを取り出します．

```
# get result
self.coef_ = res.x.view(self._param_dtype)['coef'][0, :].copy()
self.intercept_ = res.x.view(self._param_dtype)['intercept'][0]
```

このロジスティック回帰のクラスでは，重みベクトル $\mathbf{w}$ と切片 b のパラメータを，それぞれ属性 coef_ と intercept_ に保持します．しかし， これらのパラメータはまとめて 1 次元配列 res.x に格納されています．そこで，このあとすぐ紹介する view() と構造化配列を使って分離する必要があります．なお，ローカル変数である res は fit() メソッドの終了時にその内容が失われるので， copy() メソッドで配列の実体をコピーしていることに注意して下さい．

4.4.2 構造化配列

1 次元の配列にまとめて格納されている複数のパラメータを分離するために，ここでは構造化配列を利用します．そこで，まずこの構造化配列について紹介します．

構造化配列 (structured array) とは，通常の NumPy 配列と次のような違いがあります．

- 通常の NumPy 配列では要素が全て同じ型でなければならないのに対し，構造化配列では列ごとに型を変更可能
- 文字列による名前で列を参照可能
- 列の要素として配列を指定可能

構造化配列は今まで紹介した ndarray とは， dtype 属性の値が異なります．構造化配列では，列ごとにその要素が異なるので，各列ごとの型の定義をリストとして並べます．

```
[(field_name, field_dtype, field_shape), ...]
```

field_name は列を参照するときの名前で，辞書型のキーワードとして利用できる文字列を指定します．field_dtype はこの列の型で， *NumPy* **配列の属性と要素の参照** で紹介した NumPy の型を表すクラス *np.dtype* を指定します．field_shape は省略可能で，省略したり，単に 1 と指定すると通常の配列と同じ 0 次元配列，すなわちスカラーになります[*1]．2 以上の整数を指定すると，指定した大きさの 1 次元配列が，整数のタプルを指定すると， このタプルが shape 属性の値であるような ndarray がその列の要素になります．

それでは，実際に構造化配列を生成してみます[*2]．

```
In [1]: a = np.array(
   ...:     [('red', 0.2, (255, 0, 0)),
   ...:     ('yellow', 0.5, (255, 255, 0)),
   ...:     ('green', 0.8, (0, 255, 0))],
   ...:     dtype=[('label', 'U10'), ('state', float), ('color', int, 3) ])
```

np.array() を用いて構造化配列を生成しています．最初の引数は配列の内容で，各行の内容を記述したタプルのリストで表します．配列の型を dtype 属性で指定しています．最初の列は名前が label で，その型は長さ 10 の文字列です．次の列 state はスカラーの実数，そして最後の列 color は大きさ 3 の 1 次元の整数型配列です．

次は，生成した構造化配列の内容を参照します．型を指定した時の列の名前 field_name の文字列を使って，構造化配列 a の列は a[field_name] の記述で参照できます．それでは，上記の構造化配列 a の要素を参照してみます．

```
In [2]: a['label']
Out[2]:
array(['red', 'yellow', 'green'],
      dtype='<U10')
In [3]: a['color']
Out[3]:
array([[255,   0,   0],
       [255, 255,   0],
       [  0, 255,   0]])
In [4]: a['state'][1]
Out[4]: 0.5
```

最初の a['label'] は，名前が label の列，すなわち第 1 列を参照します．要素が文字列である 1 次元配列が得られています．2 番目の a['color'] は最後の列 color を参照しています．各行の要素が大きさ 3 の整数配列なので，それらを縦に連結した (3, 3) の配列が得られます．最後の a['state'][1] は，

[*1] 1 ではなく，(1,) と指定すると，スカラーではなく，1 次元の大きさ 1 の配列になります．

[*2] その他，構造化配列の dtype を指定する方法は他にも用意されています．詳細は NumPy マニュアルの Structured Array の項目を参照して下さい．

a['state'] で a の第 2 列 state で 1 次元の実数配列が得られ， [1] によってインデックスが 1 の要素，すなわち 2 番目の要素が抽出されます．

4.4.3 構造化配列を用いた実装

それでは，この構造化配列を使って，ロジスティック回帰のパラメータを表してみます．fit() メソッドで，最適化を実行する前に，次のように実装しました．

```
# dtype for model parameters to optimize
self._param_dtype = np.dtype([
    ('coef', float, self.n_features_),
    ('intercept', float)
])
```

第 1 列目の coef は重みベクトル $\mathbf{w}$ を表すものです．1 次元で大きさが特徴数 n_features_ に等しい実数ベクトルとして定義しています．第 2 列目の intercept は切片 b に相当し，スカラーの実数値としています．この構造化配列の型を dtype クラスのインスタンスとしてロジスティック回帰クラスの属性 _param_dtype 保持しておきます．

class np.**dtype**

Create a data type object.

変数 **obj** – Object to be converted to a data type object.

それでは， minimize() の結果を格納した res.x から，構造化配列を使ってパラメータを分離する次のコードをもう一度見てみましょう．

```
# get result
self.coef_ = res.x.view(self._param_dtype)['coef'][0, :].copy()
self.intercept_ = res.x.view(self._param_dtype)['intercept'][0]
```

view() は，配列自体は変更や複製をすることなく，異なる型の配列として参照するメソッドです．C 言語などの共用体と同様の動作をします．res.x は大きさが n_features_ + 1 の実数配列ですが，重みベクトルと切片のパラメータをまとめた _param_dtype 型の構造化配列として参照できます．

_param_dtype 型では，列 coef は大きさが n_features_ の 1 次元配列です．よって， res.x.view(self._param_dtype)['coef'] によって shape が (1, n_features_) の配列を得ることができます．その後の [0, :] によって，この配列の 1 行目の内容を参照し，これを重みベクトルとして取り出しています．もう一方の列 intercept はスカラーの実数なので， res.x.view(self._param_dtype)['intercept'] と記述することで，大きさが 1 の 1 次元実数配列を参照できます．この配列の最初の要素を参照し，これを切片として取り出しています．

以上で， **ロジスティック回帰の形式的定義** の式 (3) を解いて，得られた重みベクトル $mathbfw$ と切片 b を，ロジスティック回帰の属性 coef_ と intercept_ とにそれぞれ格納することができました．

次の **損失関数とその勾配** では， minimize() に fun と jac の引数として引き渡す損失関数とその勾配を

実装します．

4.5 損失関数とその勾配

学習メソッドの実装 では， minimize() を用いて，データにあてはめて，パラメータを推定しました．しかし， minimize() に引き渡す損失関数とその勾配はまだ実装していませんでした．ここでは，これらを実装して，ロジスティック回帰の学習部分を完成させます．

4.5.1 損失関数

まず **ロジスティック回帰の形式的定義** の式 (2) で示した損失関数を実装します．この損失関数を minimize() に引数 fun として渡すことで，この関数値が最小になるようなパラメータを求めます．

関数は，次のようにメソッドとして定義します．

```
def loss(self, params, X, y):
    """ A loss function
    """
```

この関数は minimize() から呼び出されます．このように引数として指定した関数が呼び出されることをコールバックといいます．このとき，self の次の第 1 引数には目的関数のパラメータが渡されます．このパラメータは minimize() の初期値パラメータ x0 と同じ大きさの実数型の 1 次元配列です．2 番目以降の引数は minimize() で引数 args で指定したものが渡されます．損失関数の計算には訓練データが必要なので， minimize() では X と y を渡していましたので，これらがこの損失関数に引き渡されています．

fit() メソッドでは，ロジスティック回帰モデルのパラメータは，構造化配列を使って 1 次元配列にまとめていました．これを再び，重みベクトル $mathbfw$ と 切片 b それぞれに相当する coef と intercept に分けます．

```
# decompose parameters
coef = params.view(self._param_dtype)['coef'][0, :]
intercept = params.view(self._param_dtype)['intercept'][0]
```

このように， view() メソッドを使って **構造化配列を用いた実装** で紹介したのと同じ方法で分けることができます．

これで損失関数の計算に必要なデータやパラメータが揃いました．あとは， **ロジスティック回帰の形式的定義** の式 (2) に従って損失を計算し， メソッドの返り値としてその値を返せば完成です．

```
# predicted probabilities of data
p = self.sigmoid(np.dot(X, coef) + intercept)

# likelihood
l = np.sum((1.0 - y) * np.log(1.0 - p) + y * np.log(p))
```

```
# L2 regularizer
r = np.sum(coef * coef) + intercept * intercept

return - l + 0.5 * self.C * r
```

p は， $\Pr[y{=}1|\mathbf{x};\mathbf{w},b]$ ， l は大数尤度，そして r は L_2 正則化項にそれぞれ該当します．

4.5.2 損失関数の勾配

今度は **ロジスティック回帰の形式的定義** の式 (4) で示した損失関数の勾配を実装し，これを minimize() に引数 jac として渡します．勾配関数に引き渡される引数は，損失関数のそれと同じになります．また，パラメータは重みベクトルと切片に，損失関数と同じ方法で分けます．

スカラーである損失とは異なり，勾配はパラメータと同じ大きさのベクトルです．そこでパラメータと同じ大きさの 1 次元配列を用意し，そこに重みベクトルと切片のための領域を割り当てます．

```
# create empty gradient
grad = np.empty_like(params)
grad_coef = grad.view(self._param_dtype)['coef']
grad_intercept = grad.view(self._param_dtype)['intercept']
```

入力パラメータ params と同じ大きさの配列を確保するのに，ここでは *np.empty_like()* を用います．*np.zeros_like()* ， *np.ones_like()* ，および *np.empty_like()* は，今までに生成した配列と同じ大きさの配列を生成する関数で，それぞれ *np.zeros()* ， *np.ones()* ，および *np.empty()* に対応しています．

np.**zeros_like**(*a*, *dtype=None*)

Return an array of zeros with the same shape and type as a given array.

np.**ones_like**(*a*, *dtype=None*)

Return an array of ones with the same shape and type as a given array.

np.**empty_like**(*a*, *dtype=None*)

Return a new array with the same shape and type as a given array.

この確保した領域 grad を，重みベクトルと切片にそれぞれ対応する， grad_coef と grad_intercept に分けます．これには view() メソッドを用いますが，今までのパラメータ値の読み出しだけの場合と異なり，値を後で代入する必要があります．そのため，最初の要素を取り出すことはせず，配列のまま保持します．

これで勾配の計算に必要なものが揃いましたので， **ロジスティック回帰の形式的定義** の式 (4) に従って勾配を計算します．

```
# predicted probabilities of data
p = self.sigmoid(np.dot(X, coef) + intercept)
```

```
# gradient of weight coefficients
grad_coef[0, :] = np.dot(p - y, X) + self.C * coef

# gradient of an intercept
grad_intercept[0] = np.sum(p - y) + self.C * intercept

return grad
```

p は，損失関数と同じく $\Pr[y{=}1|\mathbf{x};\mathbf{w},b]$ です．重みベクトルについての勾配を計算したあと，保持していた配列 grad_coef の第 1 行目に代入しています．切片についての勾配も，同様に grad_intercept の最初の要素に代入します．これら二つの勾配は grad にまとめて格納できているので，これを返します．

この勾配を計算するのに，*np.dot()* を用いていますので，この関数を最後に紹介します．

np.**dot**(a, b)

Dot product of two arrays.

3 次元以上の配列についても動作が定義されていますが，ここでは 2 次元までの配列についての動作について紹介します．1 次元配列同士では，ベクトルの内積になります．

```
In [10]: a = np.array([10, 20])
In [10]: b = np.array([[1, 2], [3, 4]])
In [11]: np.dot(a, a)
Out[11]: 500
```

2 次元配列同士では行列積になります．

```
In [12]: np.dot(b, b)
Out[12]:
array([[ 7, 10],
       [15, 22]])
```

1 次元配列と 2 次元配列では，横ベクトルと行列の行列積になります．

```
In [13]: np.dot(a, b)
Out[13]: array([ 70, 100])
```

2 次元配列と 1 次元配列では，行列と縦ベクトルの行列積になります．

```
In [14]: np.dot(b, a)
Out[14]: array([ 50, 110])
```

以上で，損失関数とその勾配を求めるメソッドが実装できました．これにより **学習メソッドの実装** で実装した fit() メソッドでロジスティック回帰モデルの学習ができるようになりました．

4.6 実行と予測

ここでは，学習したモデルを使って予測をするメソッドを実装してロジスティック回帰のクラスを完成させます．その後，このクラスを使って，最適化手法の実行速度を比較してみます．

4.6.1 予測

ロジスティック回帰では，**ロジスティック回帰の形式的定義** の式 (5) によってクラスを予測します．分類する事例の特徴ベクトルを各行に格納した 2 次元配列 X が入力です．このとき，クラスが 1 になる確率 $\Pr[y{=}1|\mathbf{x}; \mathbf{w}^*, b]$ を計算します．

```
# predicted probabilities of data
p = self.sigmoid(np.sum(X * self.coef_[np.newaxis, :], axis=1) +
                 self.intercept_)
```

学習により獲得した重みベクトルは self.coef_ に，切片は self.intercept_ に格納されています．X の各行の特徴ベクトルと重みベクトルの内積に，切片を加えて，シグモイド関数を適用することで，各事例のクラスが 1 になる確率を要素とする 1 次元の配列を得ます．

この確率が 0.5 未満かどうかでクラスを予測します．これには 3 項演算子に該当する *np.where()* を用います．

np.**where**(*condition*, *x*, *y*)

Return elements, either from x or y, depending on condition.

条件が成立したときは x を，そうでないときは y を関数の値として出力します．クラスが 1 になる確率が 0.5 未満であれば，クラス 0 に，それ以外で 1 に分類する実装は次のとおりです．

```
return np.where(p < 0.5, 0, 1)
```

np.where() はユニバーサル関数なので，このように全ての事例をまとめて分類することができます．

4.6.2 実行

実行可能な状態の LogisticRegression の実行スクリプトは，以下の場所から取得できます．実行時には lr.py と iris2.tsv[*1] がカレントディレクトリに必要です．

https://github.com/tkamishima/mlmpy/blob/master/source/run_lr.py

このスクリプトでは，データを *np.genfromtxt()* で読み込むときに，構造化配列を利用しました．

[*1] iris2.tsv は UCI Repository の Iris Data Set をもとに作成したものです．Fisher の判別分析の論文で用いられた著名なデータです．3 種類のアヤメのうち， Iris Versicolour と Iris Virginica の 2 種類を取り出しています．

```
# load data
data = np.genfromtxt('iris2.tsv',
                     dtype=[('X', float, 4), ('y', int)])
```

最初の 4 列は実数型の特徴ベクトルとして X で参照できるように，残りの 1 列は整数型のクラスとして y で参照できるようにしています．すると，次のように特徴ベクトルとクラスを分けて fit() メソッドに渡すことができます．

```
clr.fit(data['X'], data['y'])
```

4.6.3 最適化手法の比較

最後に **各種の最適化手法** で紹介した各種の最適化手法の違いについて調べてみます．LogisticRegression の fit() メソッドでの最適化関数 minimize() の呼び出しを次のように変更してみます．

```
res = minimize(fun=self.loss,
               x0=np.zeros(self.n_features_ + 1, dtype=float),
               jac=self.grad_loss,
               args=(X, y),
               method='Powell',
               options={'disp': True})
```

これは勾配情報を使わない Powell 法を指定し，さらに最適化の結果を表示するように変更しています．run_lr.py スクリプトを実行すると，勾配利用しなかった警告が表示されたあと，最適化の結果が次のように表示されます:

```
Optimization terminated successfully.
         Current function value: 31.685406
         Iterations: 18
         Function evaluations: 1061
```

収束するまでに 18 回の反復がおこわれ，損失関数の呼び出しは 1061 回です．次に，損失関数の勾配を用いる共役勾配法を試してみます．

```
res = minimize(fun=self.loss,
               x0=np.zeros(self.n_features_ + 1, dtype=float),
               jac=self.grad_loss,
               args=(X, y),
               method='CG',
               options={'disp': True})
```

十分に収束しなかった旨の警告が表示されますが，上記の Powell 法と同等の損失関数値が達成できています:

```
Warning: Desired error not necessarily achieved due to precision loss.
         Current function value: 31.685406
         Iterations: 21
         Function evaluations: 58
         Gradient evaluations: 46
```

収束までの反復数は 21 回と若干増えていますが，損失関数とその勾配の呼び出しはそれぞれ 58 回と 46 回と，Powell 法ずっと少なくなっています．最後に，二次の微分であるヘシアンも近似計算する BFGS 法を試してみます．

```
res = minimize(fun=self.loss,
               x0=np.zeros(self.n_features_ + 1, dtype=float),
               jac=self.grad_loss,
               args=(X, y),
               method='CG',
               options={'disp': True})
```

最適化は収束し，今までと同等の損失関数値が達成できています:

```
Optimization terminated successfully.
         Current function value: 31.685406
         Iterations: 11
         Function evaluations: 15
         Gradient evaluations: 15
```

反復数は 11 と最も速く収束しており，損失関数やその勾配の評価回数も，共役勾配法より減少しています．

以上の結果からすると，収束が速く，関数の評価回数も少ない BFGS 法が優れているように見えます．しかし，BFGS 法は 2 次微分であるヘシアン行列を計算するため，パラメータ数が多い場合には多くの記憶領域を必要とします．よって，問題の性質や規模に応じて最適化手法は選択する必要が生じます．

4.7 「ロジスティック回帰」まとめ

ロジスティック回帰 の章では，ロジスティック回帰法の実装を通じて以下の内容を紹介しました．

- **ロジスティック回帰の形式的定義**
  - ロジスティック回帰法
- **シグモイド関数**
  - 静的メソッドによる数値関数の実装
  - ネピアの数や円周率などの定数
  - `np.seterr()` による浮動小数点エラーの処理方法の設定
  - オーバーフロー・アンダーフローへの対策

  - `np.vectorize()` を用いたユニバーサル関数への変換
  - `np.piecewize()` による区分関数の定義
  - 数値を一定の範囲に収める `np.clip()` 関数
- **非線形最適化関数**
  - SciPy の非線形最適化関数 *`sp.optimize.minimize_scalar()`* と *`sp.optimize.minimize()`* の紹介
  - 最適化の結果を返すためのクラス `OptimizeResult` の紹介
  - 各種の最適化手法の特徴
- **学習メソッドの実装**
  - 最適化関数を用いた，モデルのパラメータの学習
  - 構造化配列
  - 構造化配列と `view()` メソッドによる同一領域の異なる参照方法
- **損失関数とその勾配**
  - *`sp.optimize.minimize()`* からのコールバック
  - *`np.empty_like()`* などを用いた行列の生成
  - *`np.dot()`* による内積と行列積
- **実行と予測**
  - 3 項演算を行う *`np.where()`* 関数
  - 構造化配列を用いたデータの読み込み
  - 最適化手法の実行結果の比較

第 5 章

おわりに

本チュートリアルは以上です．Python には，機械学習の scikit-learn，統計処理の statsmodels，データ整形の pandas など多数の数値計算関連派ケージが充実しています．NumPy / SciPy と併せてこれらのパッケージを利用して，問題解決に役立ててください．

5.1 謝辞

この文書を作成するにあたり，下記ソフトウェアと，各所のサイトの情報を利用させていただきました．感謝とともに，ここに記したいと思います．

- このチュートリアルは Sphinx を利用して執筆しています．
- Sphinx の関連の情報を参考にしました．
  - Sphinx-Users.jp
- Sphinx へのソーシャルボタンの設置の参考にしました．
  - Sphinx にソーシャルボタンを設置する @ 今日の Python
- IPython コンソールのハイライトのために，IPython のソースから `ipython_console_highlighting.py` を導入しています．
  - The IPython licensing terms
- UCI Machine Learning Repository のいくつかのデータ集合をサンプルとして利用しています．
  - A. Frank and A. Asuncion “UCI Machine Learning Repository” University of California, Irvine, School of Information and Computer Sciences (2010)

最後に，本チュートリアルのバグ等をご連絡いただきました方々に感謝します．

索引

abstract class, 21
aggregation, 35
apply_over_axes, 38
arange, 17
argmax, 18
argmin, 18
array, 4

basinhopping, 50
Brent method, 50
broadcastable, 30
broadcasting, **28**, 32
brute, 50

class
    BaseBinaryNaiveBayes, 21
    LogisticRegression, 49
    NaiveBayes1, **11**, 22
    NaiveBayes2, 40
clip, 49

dot, 58
dtype, **6**, 54

e, 45
empty, 5
empty_like, 57
equal, 24

floor_divide, 36
frompyfunc, 48

genfromtxt, **18**, 59

identity, 5
ipython, 40
ix_, 38

log, 16
logical_and, 38
logistic regression, 43

matmul, 58
minimize, **50**, 52, 56
minimize_scalar, 50

naive Bayes, **9**
    multinomial, 9
ndarray, **3**, 14
    astype, 7
    dtype, **6**, 53
    ndim, 6, **25**
    shape, 5, **25**
    view, 55
newaxis, 25
non-linear optimization, 50
np..frompyfunc() (組み込み関数), 48
np.apply_over_axes() (組み込み関数), 38
np.arange() (組み込み関数), 17
np.argmax() (組み込み関数), 18
np.argmin() (組み込み関数), 18
np.array() (組み込み関数), 4
np.dot() (組み込み関数), 58
np.dtype (組み込みクラス), 55
np.empty() (組み込み関数), 5
np.empty_like() (組み込み関数), 57
np.genfromtxt() (組み込み関数), 18
np.identity() (組み込み関数), 5
np.ndarray (組み込みクラス), 6
np.newaxis, 25
np.ones() (組み込み関数), 5
np.ones_like() (組み込み関数), 57
np.piecewise() (組み込み関数), 49
np.reshape() (組み込み関数), 27
np.select() (組み込み関数), 49
np.sum() (組み込み関数), 17
np.swapaxes() (組み込み関数), 37
np.where() (組み込み関数), 59
np.zeros() (組み込み関数), 5
np.zeros_like() (組み込み関数), 57

ones, 5
ones_like, 57
optimization, **50**, 60
OptimizeResult, 50

pi, 45
piecewise, 48

reshape, 27

sample
    lr.py, 49
    lr1.py, 44
    lr2.py, 46
    lr3.py, 48
    nbayes1.py, 18
    nbayes1b.py, 22
    nbayes2.py, 40
    run_lr.py, 59
    run_nbayes1.py, 19
    run_nbayes1b.py, 22
    run_nbayes2.py, 40
    vote_filled.tsv, 19
scikit-learn, 2, **11**
select, 49
seterr, 45
sigmoid function, 44
slice, 14
sp.constants, 45
sp.optimize.minimize() (組み込み関数), 50
sp.optimize.minimize_scalar() (組み込み関数), 50
sp.optimize.OptimizeResult (組み込みクラス), 50
staticmethod, 44
structured array, **53**, 57, 59
sum, 17
swapaxes, 37

timeit, 40
transpose, 25
true_divide, 36

ufunc, 47
universal function, **16**, 47

vectorize, 47

where, 59

zeros, 5
zeros_like, 57