機械学習の Python との出会い

リリース *2016-10-20 05:08:56 +0900*

神嶌 敏弘 (Toshihiro Kamishima)

2016/10/20

目次

Chapter 1

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 [2] 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[1] 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

[2] 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] http://conference.scipy.org/

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．

1.2 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．

- 言うまでもなく Python[3] を利用します．Python3 ではなく，Python2 系統の最終版とされるバージョン 2.7 を想定しています．
- ここで紹介する NumPy[4] と SciPy[5] も利用します．NumPy は 1.6.0 以降， SciPy は 0.10.0 以降のバージョンを想定しています．
- 機械学習のライブラリである scikit-learn[6] の利用や連携についても紹介します．scikit-learn はバージョンが 0.10 以降を想定しています．
- Matplotlib[7] はグラフ（チャート）を描くためのライブラリで，入力データや学習結果の表示に利用します．バージョンは 1.1.0 以降を想定しています．

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
import sklearn
```

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．

[3] http://www.python.org/
[4] http://numpy.scipy.org/
[5] http://www.scipy.org/
[6] http://scikit-learn.org/
[7] http://matplotlib.sourceforge.net/

Chapter 2

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[8] 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を紹介します．

[8] 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．

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.

np.zeros() ，*np.ones()* ，および *np.empty()* には，それぞれ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 *np.zeros_like()* ，*np.ones_like()* ，および *np.empty_like()* があります．

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 [9]．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,) [10] 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

[9] 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．

[10] 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．

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．

- 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 の型を返す関数 np.dtype() を用いて，np.dtype('S<文字列長>') [11] 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("S16") のように指定します．Unicode 文字列の場合は，この S を U に置き換えます．

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=np.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．

[11] 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．

```
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[12]) 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
```

[12] http://scikit-learn.org/stable/developers/index.html#apis-of-scikit-learn-objects

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()` [13] 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 xrange(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()* で初期化します．また，割り算の結果を実数で得るため，`float` 型への変換も行います．

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

[13] 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．

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 xrange(n_samples):
    for j in xrange(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 xrange(n_features):
    for xi in xrange(n_fvalues):
        for yi in xrange(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 xrange(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()` [14] 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 xrange(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 `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．

[14] `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．

`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 [15] ．

`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．

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

```
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 [17] が，テキスト形式のファイルの読み込みをする *np.genfromtxt()* [19] 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 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[16] 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 [20] ．

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

[17] 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．

[19] 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．

[16] http://docs.scipy.org/doc/numpy/user/basics.io.genfromtxt.html

[20] 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．

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 xrange(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 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．

- *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

Chapter 3

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．

```
from abc import ABCMeta, abstractmethod

class BaseBinaryNaiveBayes(object):
"""
Abstract Class for Naive Bayes whose classes and features are␣
↪binary.
"""

__metaclass__ = ABCMeta
```

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 xrange(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 xrange(n_classes):
    for i in xrange(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 xrange(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 [22] [23]．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 [24] 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
Out[29]:
```

[22] 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．

[23] 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．

[24] 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．

```
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[25] 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.

[25] http://docs.scipy.org/doc/numpy/reference/ufuncs.html#broadcasting

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)

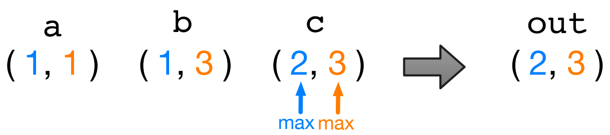
0 次元の a では， shape の先頭に 1 を 2 個追加して，全ての次元で大きさが 1 の shape になります．1 次元の b では，元の大きさ 3 の第 0 次元の前に，大きさ 1 の次元を挿入します．すなわち，統一後に ndim が 2 になる行列では，ベクトルは横ベクトルとなります．2 次元の c は， ndim は変更されないので， shape も変更されません．このように自動的に次元数を統一する機構が備わっていますが，コードが分かりにくくなることもよくあります．そのため，実用的には，後述の **クラスの分布の学習** の例のように，np.newaxis などを用いて明示的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．

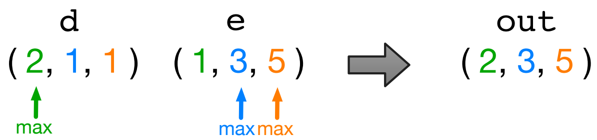
出力配列の shape の決定

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．

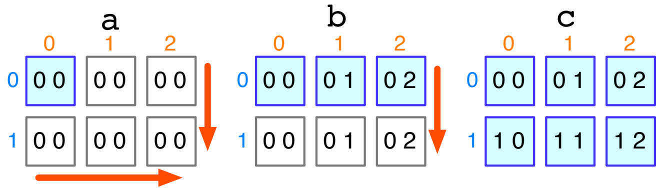
ブロードキャストの実行

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．

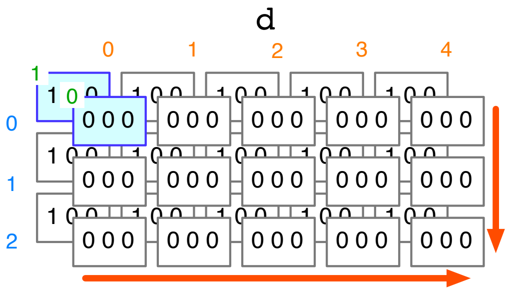
後者の場合，一つの要素の値を，その次元の全ての要素にブロードキャストして（拡散して）用いるので，この配列の自動操作の仕組みはブロードキャストと呼ばれています．

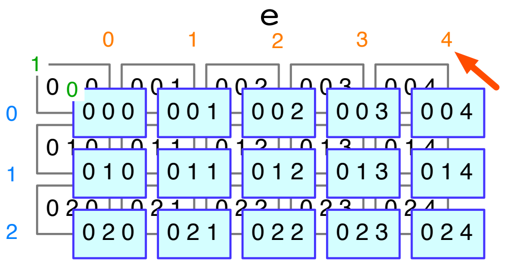
この規則について，上記の三つの配列 `a`，`b`，および `c` を例にとり，次の図を用いて説明します．



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 xrange(n_classes):
    for i in xrange(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 xrange(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 xrange(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 xrange(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 xrange(n_classes):
    self.pY_[i] = nY[i] / float(n_samples)
```

この処理も，ユニバーサル関数の機能を使うと次のように簡潔に実装できます．

```
self.pY_ = np.true_divide(nY, n_samples)
```

Python では整数同士の割り算の解は切り捨ての整数になります [26]．しかし，ここでは実数の解を得たいので np.true_divide() 関数を用いて，切り捨てではない実数の解を得ます．

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

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

この関数はユニバーサル関数なので， nY の各要素は，それぞれ n_samples で割られます．

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 xrange(n_samples):
    for j in xrange(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 xrange(n_samples):
    for j in xrange(n_features):
        for yi in xrange(n_classes):
            for xi in xrange(n_fvalues):
                if y[i] == yi and X[i, j] == xi:
                    nXY[j, xi, yi] += 1
```

[26] 逆に浮動小数点に対する場合でも，切り捨てした割り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.

それでは，この実装を，特徴の分布と同様に書き換えます．

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 [27]．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．

[27] 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 [28]．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 [29]．

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 [30]．

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

[28] 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.

[29] 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 logical_or()，logical_not()，および logical_xor() があります．

[30] 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.

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．

```
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，実数の結果を返す割り算のユニバーサル関数 np.true_divide() を適用すれば，特徴値の確率を計算できます [31]．

```
self.pXgY_ = np.true_divide(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_ = np.true_divide(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．

[31] Python3 のように割り算を整数ではなく実数で行う指定 from __future__ import division がある場合には *np.ndarray* に対する演算子 / による割り算でも，その解は実数となります．

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 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

Chapter 4

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[32] 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[33]
- 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[34]
- 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[35]
- UCI Machine Learning Repository[36] 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．

[32] http://sphinx.pocoo.org/
[33] http://sphinx-users.jp/
[34] http://blog1.erp2py.com/2011/09/sphinx.html
[35] https://github.com/ipython/ipython/blob/master/COPYING.txt
[36] http://archive.ics.uci.edu/ml

索引