応用計量分析２（第5回）

担当教員: 梶野 洸（かじの ひろし）

本日の目標

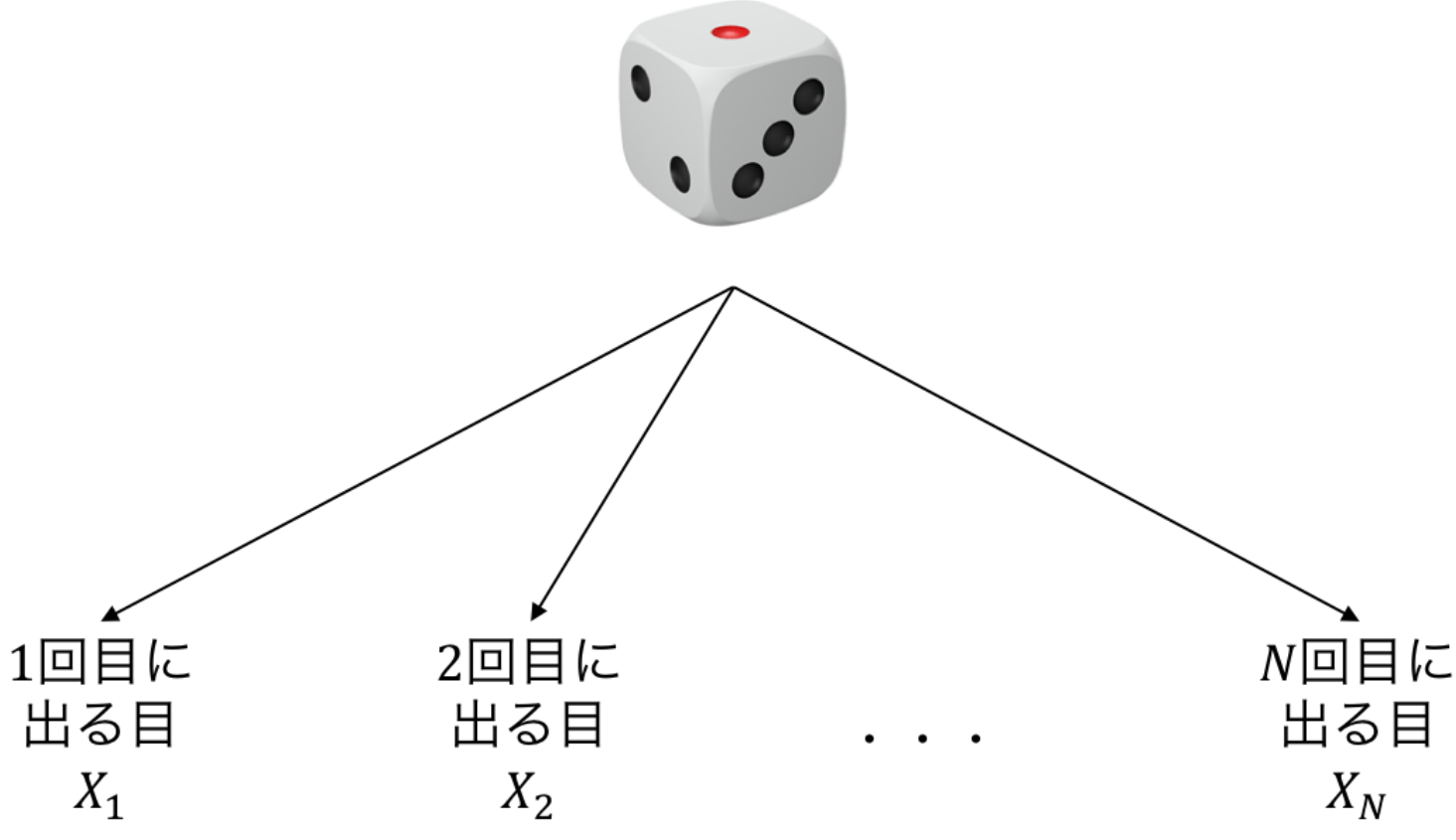
- 確率・統計を思い出す
- オブジェクト指向の書き方で最尤推定してみる

確率変数

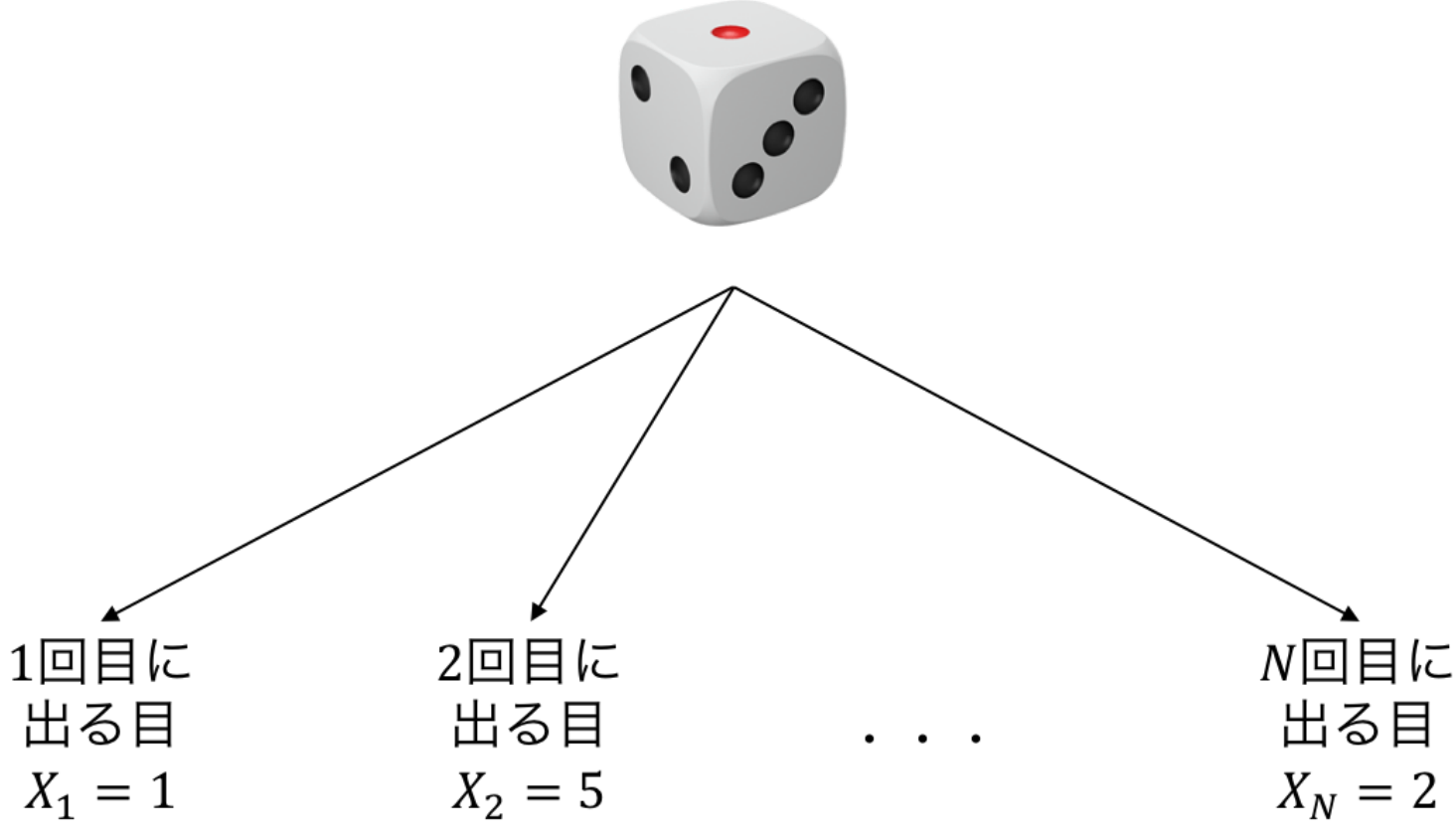
決定的には定まらない観測値をモデル化するのに用いる

- 例1: サイコロを振って出る目
    - $X_n \in \{1, 2, \ldots, 6\}$: n回目に出た目
    - 1~6の目のどれが出るかわからない
- 例2: 私の身長 $X_n \in \mathbb{R}_+$
    - X_n: n回目の測定値
    - 測るたびに微妙に値が異なる（測定誤差）
- 例3: 人類の身長 $X_n \in \mathbb{R}_+$
    - X_n: n人目の測定値
    - 人によって身長は異なる

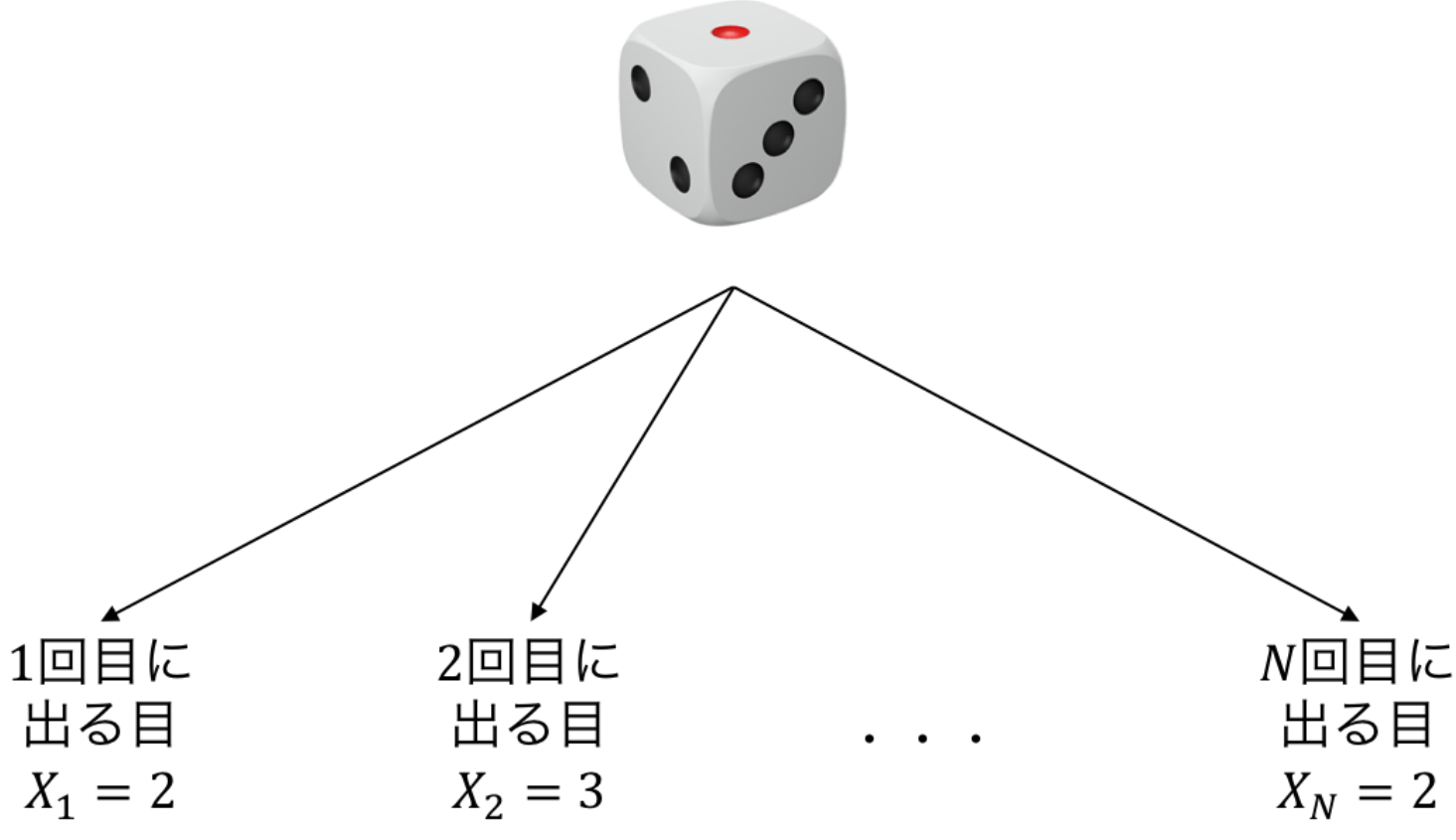
さいころを振る前にも各回で出る目を確率変数として書ける



N 回振るとそれぞれの確率変数の観測値が得られる



試行のたびに得られる観測値は変わる



統計でやりたいこと

得られた観測値から、確率変数の従う規則を推測したい

- 得られた観測値 = N回さいころを振ったときに $\{1, 5, \ldots, 2\}$ という目が出た

確率変数の従う規則=確率分布

- $p(X)$: 確率変数 X の従う分布
  - 観測値を入力すると確率を返す関数
  - 例えば $p(x) = 1/6$ $(\forall x \in \{1, 2, \ldots, 6\})$ だとすべての目が1/6の確率で出る、ということを表す
  - 未知のパラメタを用意しておいて、それを観測値から決定する

確率変数の従う規則は色々な決め方がある

事前知識や、手元にあるデータから推定できるか否か、という観点で選ぶ

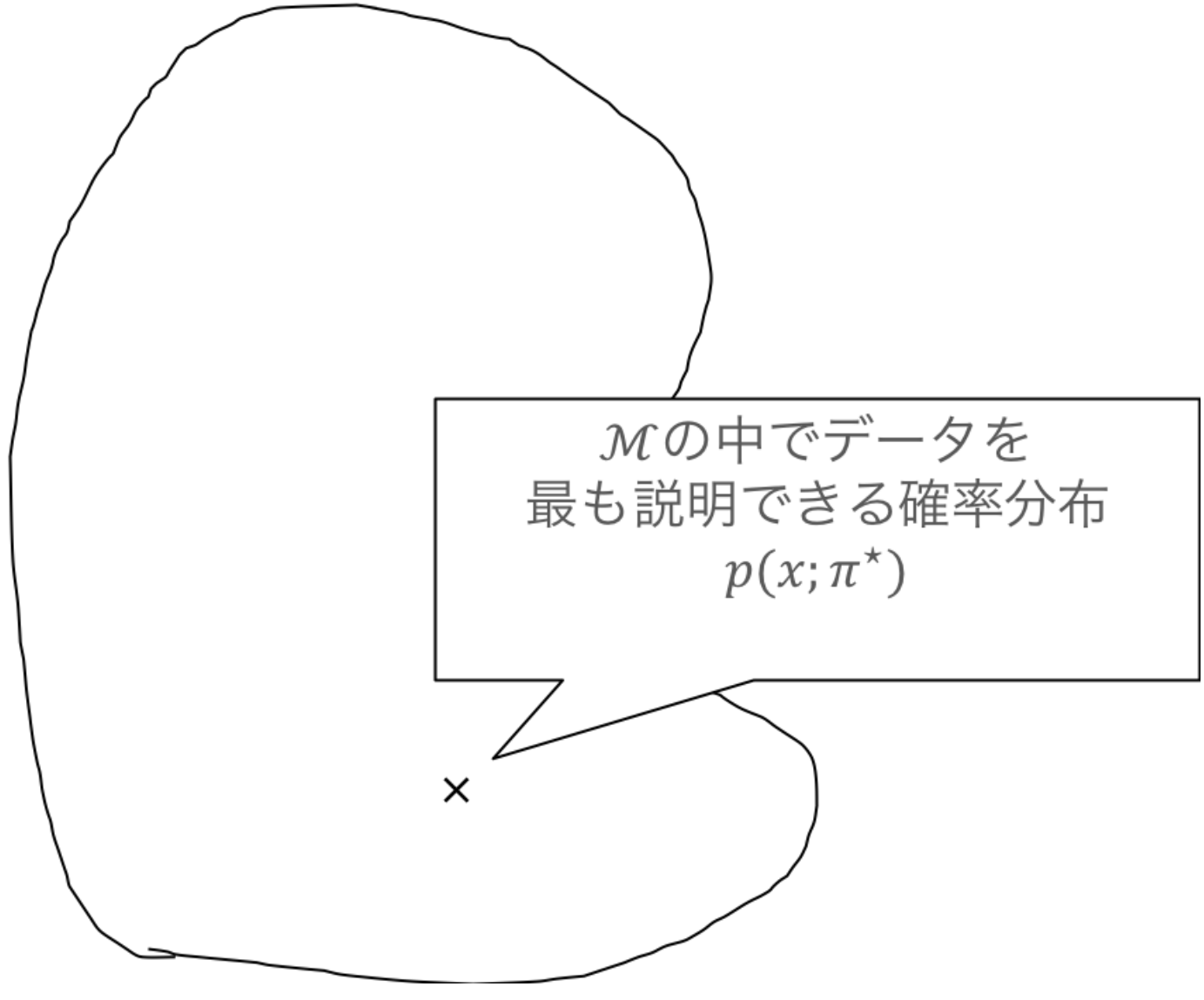
1. 何回目に振ったかは関係なく、すべての目は同一の分布 $p(X)$ に独立に従う（だいたいこれ）
2. 奇数回目は $p_o(X)$, 偶数回目は $p_e(X)$ に独立に従う
3. 回数や過去の履歴に依存して目が決まる。つまり $p(X_1, X_2, \ldots, X_N)$ に従う。
    - N 回振るという試行を何回もやっているならこの分布は推定可能

統計でやりたいこと

得られた観測値から、確率変数の従う規則を推測したい

- 得られた観測値 = N回さいころを振ったときに $\{1, 5, \ldots, 2\}$ という目が出た
- 仮定する確率分布（の一例）: すべての目は同一の分布 $p(X; \pi)$ に独立に従う
    - $p(X = k; \pi) = \pi_k \ (k = 1, 2, \ldots, 6)$ という分布（多項分布）
    - π を未知なものとして、データから決めたい！

$$\mathcal{M} = \{p(x;\pi) \mid \pi \in \Pi\}$$



確率モデルのパラメタの決め方

- 最尤推定（これをやる）
    - 手持ちのサンプルが得られる確率が最大になるようにパラメタを定める
    - 気持ち: 手持ちのデータが出てくる確率が高くなかったらどのデータの確率が高いんだ！？
- ベイズ推定（これはやらないので参考程度に）
    - 推定したいパラメタに事前分布 $p(\theta)$ を置き、サンプルが得られた元での事後分布 $p(\theta \mid \mathcal{D})$ を計算する
    - 事前分布は固定
    - 気持ち: サンプルで条件付けを行うとパラメタの範囲がそれっぽいところに狭まる

最尤推定

- モデル: $\mathcal{M} = \{p(X;\theta) \mid \theta \in \Theta\}$
- サンプル: $\mathcal{D} = \{x_1, \ldots, x_N\}$
    - **独立に同じ分布に従っていると仮定**

サンプル $\mathcal{D}$ が得られる確率は

$$\ell(\theta) = p(\mathcal{D};\theta) = \prod_{n=1}^{N} p(x_n;\theta)$$

（独立に同じ分布に従っていると仮定したため）

- θ の関数$\ell\left(\theta\right)$としてみることができる
- この時、$\ell\left(\theta\right)$ を尤度と呼ぶ

最尤推定量 $\hat{\theta}$ は

$$\hat{\theta} = \arg\max_{\theta} \ell(\theta) = \arg\max_{\theta} \prod_{n=1}^{N} p(x_n; \theta)$$

と定義される

ただし計算の簡単のため＆実装上の問題から負の対数尤度

$$\mathcal{L}(\theta) = -\log \ell(\theta) = -\sum_{n=1}^{N} \log p(x_n; \theta)$$

を使うことが多い。

$$\hat{\theta} = \arg\max_{\theta} \ell(\theta) = \arg\min_{\theta} \mathcal{L}(\theta)$$

であることに注意。

なぜ負の対数尤度を使うか

- 対数尤度を使う理由
    - 掛け算がたし算になる（微分するのが楽になる）
    - 対数を取ると小さな値でもコンピュータで扱いやすい（丸め誤差が乗りにくい）
- 特に負の対数を使う理由（強いて言えば...）
    - 最適化問題は minimize の形で書かれることが多い
    - 情報理論的な解釈（負の対数尤度は、データを送るのに必要なビット数に相当する）

最尤推定まとめ

1. サンプルが得られる確率（≒尤度）を書き下す
2. 負の対数をとって、負の対数尤度を書き下す
3. 負の対数尤度が最小になるパラメタ $\hat{\theta}$ を求める

統計モデルを用いた解析の流れ

1. 確率変数を定める（どの量に対する確率分布が欲しいか？）
2. 確率変数の観測値と確率分布の関係を仮定する（すべての観測値が独立同一分布に従う、など）
3. 確率分布の集合を仮定する（正規分布、多項分布、など）
4. 確率分布の集合から良さげなものを取ってくる（最尤推定、ベイズ推定、など）

演習（再掲）

- サンプル: $x_1, \ldots, x_N \in \mathbb{R}$
- パラメトリックモデル: $\mathcal{M} = \{\mathcal{N}(\mu, 1) \mid \mu \in \mathbb{R}\}$
    - $\mathcal{N}(\mu, \sigma^2)$: 平均$\mu$、分散$\sigma^2$の正規分布
    - $\mathcal{N}(\mu, \sigma^2)$の確率密度関数は
      $$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

としたとき、最尤推定量 $\hat{\mu}$ を求めよ。

統計モデルの実装

オブジェクト指向の技法で実装することが多い

- オブジェクト指向とは？
- Python ではどうやるの？
- 結局どうすればいいのか？

オブジェクト指向プログラミング

- （とても雑にいうと）オブジェクト単位でプログラムを考える技法
    - クラス ≒ 設計図
    - オブジェクト = クラスの設計図に従って作られたもの

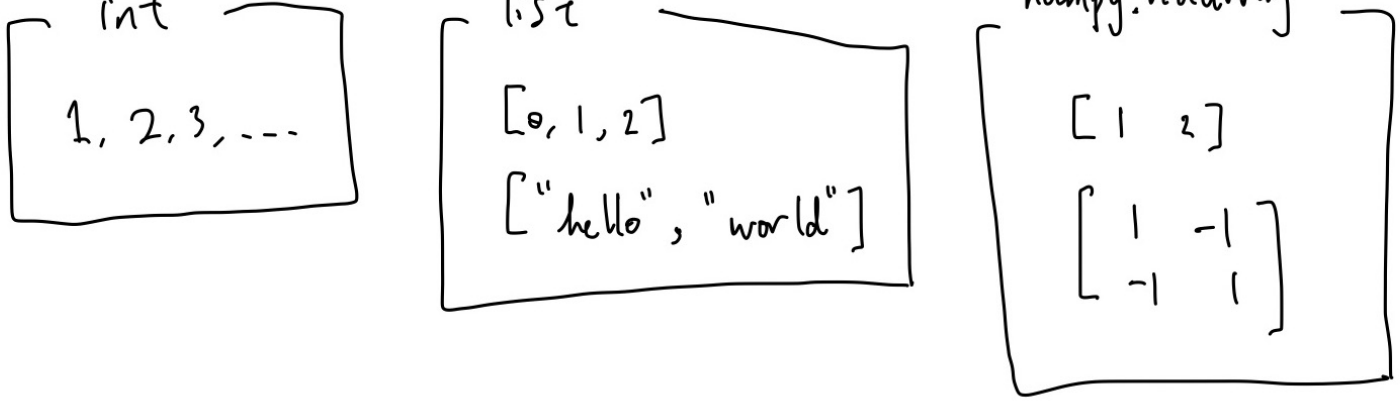
- クラスはオブジェクトの内部状態やオブジェクトに対して使える命令を規定する
- あるクラスから作られたオブジェクトは、
    1. オブジェクト固有の内部状態の値を持つ
    2. クラスで規定された命令を使える

In [9]:

```
import numpy as np
A = np.array([[1, -1], [1, 1]])
#print(A)
#print(A.transpose())
B = np.array([[1, -1], [0, 1]])
print(B)
print(B.transpose())
```

```
[[ 1 -1]
 [ 0  1]]
[[ 1  0]
 [-1  1]]
```

○ オブジェクトは、クラスで定められる。



オブジェクト固有の内部状態の値を持つ

例） `numpy.ndarray`

例えば $A = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$ という行列に対応する `numpy.ndarray` の場合

1. 要素が 1 -1 -1 1
2. サイズが 2 x 2

という内部状態

クラスで規定された命令を使える

例） `numpy.ndarray`

`transpose` という命令が使える。行列の転置に対応する。数学の世界で、ベクトルや行列であれば転置ができる、ということに対応する。

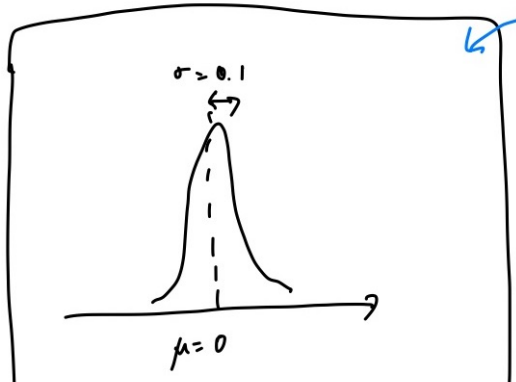
In [19]:

```
A = np.array([[1, 1], [-1, 1]])
print(A)
print(A.transpose()) # numpy.ndarray クラスのオブジェクトである A に対して `transpose` という命令をしている
```

```
[[ 1  1]
 [-1  1]]
[[ 1 -1]
 [ 1  1]]
```

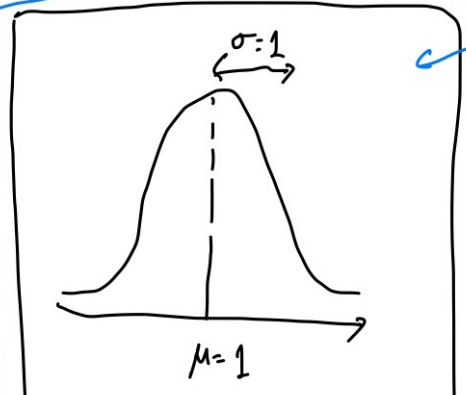
機械学習に当てはめてみると

- クラス=モデル
    - 内部状態 = パラメタ
    - 命令 = モデルの推定方法
- オブジェクト
    - 具体的なパラメタの値を持ったモデル
    - データを使ってモデルを推定して内部状態を変える
    - 推定し終わった内部状態で予測



正規分布 クラスの object

- 名前： "normal 1"
- 内部状態： $(\mu, \sigma) = (0, 0.1)$

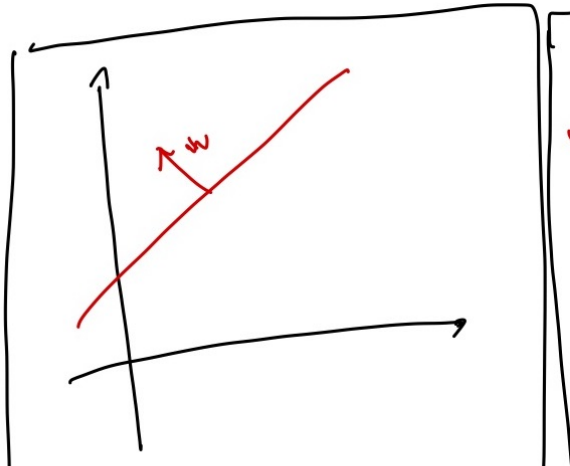


正規分布 クラスの object

- 名前： "normal 2"
- 内部状態： $(\mu, \sigma) = (1, 1)$

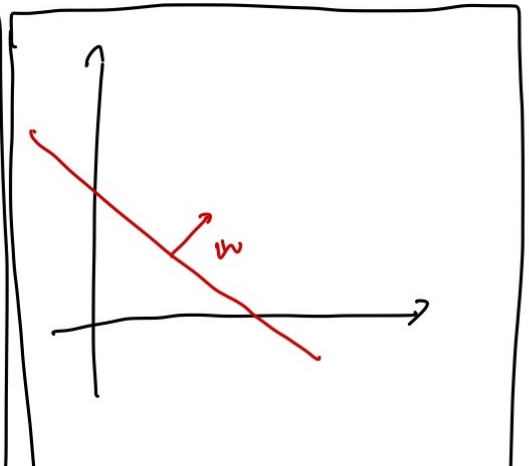
これらは
異なる object だけど、
共通した命令を持たせたい

- fit ： データを入力して
  パラメタを更新したい。
- sample ： 現在のパラメタを用いて
  データの新規生成をしたい。



線形分類器 クラスのobject

- 名前：“classifier 1”
- 内部状態：$w = [-1 \ \ 1]$



線形分類器 クラスのobject

- 名前：“classifier 2”
- 内部状態：$w = [1 \ \ 1]$

線形分類器など、

他のモデルも

オブジェクト指向っぽく書ける！

演習

正規分布の場合、

- 内部状態
- 欲しい命令（専用の関数）

は何か？

一つの答え

- 内部状態
    - 平均
    - 分散共分散行列
- 命令
    - データを入力して、パラメタを最尤推定する
    - データを入力して、各データの確率密度を計算する

他の答え

補助的なものを持っていてもよい

- 内部状態
    - 平均
    - 分散共分散行列
    - **次元**
- 命令
    - データを入力して、パラメタを最尤推定する
    - データを入力して、各データの確率密度を計算する
    - **入力の次元のチェック**
    - **内部状態の更新**

これらはバグに気づくやすくするために使う命令。今回は扱わないことにする。

Python での実装

まずは正規分布クラスを実装してみよう

In [6]:

```
import numpy as np

class Gaussian:

    def __init__(self, dim):
        self.dim = dim
        self.mean = np.zeros(dim)
        self.cov = np.identity(dim)

    def set_mean(self, x):
        self.mean = x
```

In [8]:

```
model = Gaussian(3)
print(model.mean)
model.set_mean(np.array([1,2,3]))
print(model.mean)
```

```
[0. 0. 0.]
[1 2 3]
```

In [10]:

```python
import numpy as np

class Gaussian:
    def __init__(self, dim):
        '''コンストラクタ（みたいなもの）
        オブジェクトを作るときに初めに実行される。
        内部状態の初期化に使う
        '''
        self.dim = dim
        '''
        self = オブジェクトを指す。 self.dim は、オブジェクトの dim という変数を指す。
        上の命令は、 self.dim に dim の値を代入することを表す
        '''
        self.mean = np.random.randn(dim) # オブジェクトの mean という変数をランダムに初期化
        self.cov = np.identity(dim)

    def log_pdf(self, X):
        ''' 確率密度関数の対数を返す

        Parameters
        ----------
        X : numpy.array, shape (sample_size, dim)

        Returns
        -------
        log_pdf : array, shape (sample_size,)
        '''
        return 0

    def fit(self, X):
        ''' X を使って最尤推定をする

        Parameters
        ----------
        X : numpy.array, shape (sample_size, dim)
        '''
        pass

    def sample(self, sample_size):
        ''' 現状のパラメタを使って `sample_size` のサイズのサンプルを生成する

        Parameters
        ----------
        sample_size : int

        Returns
        -----------
        X : numpy.array, shape (sample_size, dim)
            各行は平均 `self.mean`, 分散 `self.cov` の正規分布に従う
        '''
        pass
```

In [13]:

```
# クラスの使い方
normal1 = Gaussian(10) # dim = 10 を代入して正規分布オブジェクトを生成する
# 内部で __init__ メソッドが呼ばれる

# __init__ が実行されたので内部状態が設定されている
print(normal1.mean)
print(normal1.cov)

# 内部状態を後から直接変更することができる
normal1.mean = np.zeros(10)
print(normal1.mean)
```

```
[-0.18302306  0.62171581 -0.27611264 -0.55353582 -1.068
57271 -0.60349039
  0.374729   -0.20262547 -1.00334568  0.49501396]
[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

演習5.1: クラスを定義してみよう

以下の仕様を満たすクラスを書いてください

- クラス名: LinearRegression
- 内部変数: w
- 命令1: `__init__`
    - 引数: dim
    - 中でやること: 内部変数 w を、要素が全て1で長さ `dim` の `numpy.ndarray` で初期化
- 命令2: `predict`
    - 引数: X (`sample_size` x `dim` の `numpy.ndarray` と想定)
    - 出力: Xw

`log_pdf` を書く手順を解説します

- 入力は `sample_size` x `dim` の array
- 出力は長さ `sample_size` の array
    - 各データの確率密度の対数を計算したい

`log_pdf(self, X)` の実装

$$\log p(x \mid \mu, \Sigma) = -\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu) - \frac{D}{2}\log 2\pi - \frac{1}{2}\log|\Sigma|$$

1. $-\frac{D}{2}\log 2\pi - \frac{1}{2}\log|\Sigma|$ の計算
2. $-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)$ の計算
    - A. X がベクトルの時（データが一個の時）
    - B. X が行列の時（データが複数個あってそれぞれについて計算したい時）

1. $-\frac{D}{2}\log 2\pi - \frac{1}{2}\log|\Sigma|$ の計算

$\log$ は `np.log` , log determinant は `np.linalg.slogdet` で計算できるので

In [11]:

```
dim = 5
cov = np.identity(dim) # とりあえず単位行列で計算できるか確かめる

- 0.5 * dim * np.log(2 * np.pi) - 0.5 * np.linalg.slogdet(cov)[1]
```

Out[11]:

```
-4.594692666023363
```

2. $-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)$の計算（データが一個）

(1) $x-\mu$ を計算する

(2) $\Sigma^{-1}(x-\mu)$ を計算する

(3) $-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)$を計算する

In [12]:

```
dim = 10
x = np.ones(dim)
mean = np.zeros(dim)
cov = 2.0 * np.identity(dim)

centered_x = x - mean
print(centered_x)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

In [13]:

```
cov_inv_centered_x = np.linalg.solve(cov, centered_x)
print(cov_inv_centered_x)
```

```
[0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5]
```

In [14]:

```
-0.5 * (centered_x @ cov_inv_centered_x)
```

Out[14]:

```
-2.5
```

2. $-\frac{1}{2}(x-\mu)^{\top}\Sigma^{-1}(x-\mu)$の計算（データが複数個）

(1) $x-\mu$ を計算する

(2) $\Sigma^{-1}(x-\mu)$ を計算する

(3) $-\frac{1}{2}(x-\mu)^{-1}\Sigma^{-1}(x-\mu)$を計算する

In [15]:

```
dim = 3
sample_size = 10

X = np.arange(sample_size * dim).reshape(sample_size, dim)
mean = np.ones(dim)
cov = 2.0 * np.identity(dim)
print(X, mean)
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]
 [15 16 17]
 [18 19 20]
 [21 22 23]
 [24 25 26]
 [27 28 29]] [1. 1. 1.]
```

In [16]:

```
centered_X = X - mean
print(centered_X)
```

```
[[-1.  0.  1.]
 [ 2.  3.  4.]
 [ 5.  6.  7.]
 [ 8.  9. 10.]
 [11. 12. 13.]
 [14. 15. 16.]
 [17. 18. 19.]
 [20. 21. 22.]
 [23. 24. 25.]
 [26. 27. 28.]]
```

In [48]:

```
cov_inv_centered_X = np.linalg.solve(cov, centered_X.T).T
print(cov_inv_centered_X)
```

```
[[-0.5  0.   0.5]
 [ 1.   1.5  2. ]
 [ 2.5  3.   3.5]
 [ 4.   4.5  5. ]
 [ 5.5  6.   6.5]
 [ 7.   7.5  8. ]
 [ 8.5  9.   9.5]
 [10.  10.5 11. ]
 [11.5 12.  12.5]
 [13.  13.5 14. ]]
```

In [50]:

```
-0.5 * np.sum(centered_X * cov_inv_centered_X, axis=1)
```

Out[50]:

```
array([-5.0000e-01, -7.2500e+00, -2.7500e+01, -6.1250e+
01, -1.0850e+02,
       -1.6925e+02, -2.4350e+02, -3.3125e+02, -4.3250e+
02, -5.4725e+02])
```

それぞれの項をまとめて関数を作る

In [2]:

```python
import numpy as np

class Gaussian:
    def __init__(self, dim):
        '''コンストラクタ（みたいなもの）
        オブジェクトを作るときに初めに実行される。
        内部状態の初期化に使う
        '''
        self.dim = dim
        '''
        self = オブジェクトを指す。 self.dim は、オブジェクトの dim という変数を指す。
        上の命令は、 self.dim に dim の値を代入することを表す
        '''
        self.mean = np.random.randn(dim) # オブジェクトの mean という変数をランダムに初期化
        self.cov = np.identity(dim) # オブジェクトの cov という変数を単位行列に初期化

    def log_pdf(self, X):
        ''' 確率密度関数の対数を返す

        Parameters
        ----------
        X : numpy.array, shape (sample_size, dim)

        Returns
        -------
        log_pdf : array, shape (sample_size,)
        '''
        centered_X = X - self.mean
        cov_inv_centered_X = np.linalg.solve(self.cov, centered_X.T).T
        log_pdf = - 0.5 * self.dim * np.log(2 * np.pi) - 0.5 * np.linalg.slogdet(self.cov)[1] - 0.5 * np.sum(centered_X * co
        return log_pdf
```

In [4]:

```
my_gaussian = Gaussian(dim=2)
X = np.zeros((10, 2))
my_gaussian.mean = np.zeros(2)

my_gaussian.log_pdf(X)
```

Out[4]:

```
array([-2.65847588, -2.65847588, -2.65847588, -2.65847588, -2.65847588,
       -2.65847588, -2.65847588, -2.65847588, -2.65847588, -2.65847588])
```

実装の手順まとめ

1. 実装したいものを数式で書き起こす
2. 数式を計算するには何をどの順番で計算したらいいかを考える
    - なるべく細かくする
    - 全然わからないときは、実装したいものを単純化したもので考えるのも手
3. それぞれの手順を実装して、手元で動くか確かめる
    - 配列の大きさのチェックをする
    - できれば検算も
4. 組み合わせてクラスに実装する

課題5.2

1. `fit` を完成させよ。
    - 入力:
        - `X`: `sample_size` x `dim` の `numpy.ndarray`
    - 出力: なし
    - 中でやること
        - `self.mean` に `X` で計算した最尤推定量を代入する
        - `self.cov` に `X` で計算した最尤推定量を代入する
2. `sample` を完成させよ。
    - 入力:
        - `sample_size`: 整数
    - 出力:
        - `X`: `sample_size` x `dim` の `numpy.ndarray` で、各行は平均 `self.mean`、分散 `self.cov` の正規分布に従う乱数

ポイント

1. 逆行列を使わず、線型方程式を解く
2. for文を使わず、行列演算で頑張る

$$\sum_{n=1}^{N} x_n x_n^\top = \begin{bmatrix} x_1 & x_2 & \dots & x_N \end{bmatrix} \begin{bmatrix} x_1^\top \\ x_2^\top \\ \vdots \\ x_N^\top \end{bmatrix}$$

以下のように分解してみるとわかる

$$\begin{aligned}\begin{bmatrix} x_1 & x_2 & \dots & x_N \end{bmatrix} = &\begin{bmatrix} x_1 & 0 & \dots & 0 \end{bmatrix} + \begin{bmatrix} 0 & x_2 & \dots & 0 \end{bmatrix} + \cdots \\ &+ \begin{bmatrix} 0 & 0 & \dots & x_N \end{bmatrix}\end{aligned}$$

クラスの使い方

- オブジェクトを作る

In [13]:

```
my_model = Gaussian(2) # my_model というオブジェクトが出来た
```

In [3]:

```
print(my_model.mean, my_model.cov) # 平均、共分散行列を持っている
```

```
[ 0.84661088 -1.49408612] [[1. 0.]
 [0. 1.]]
```

In [6]:

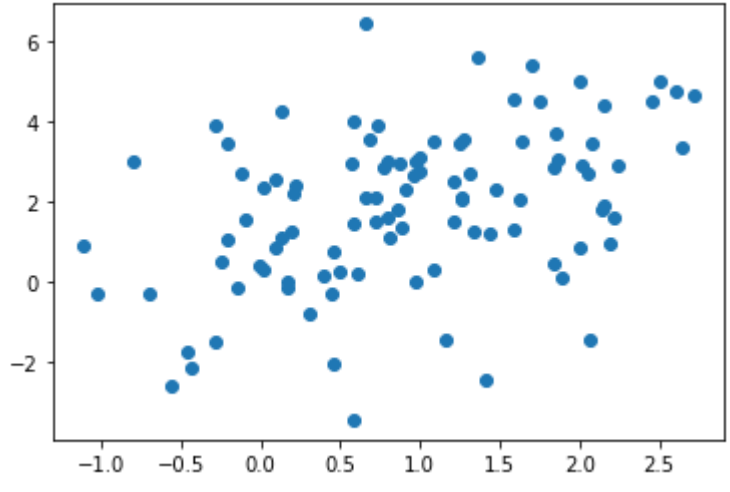
```
my_model_1 = Gaussian(2) # 他のオブジェクトも作れる
print(my_model_1.mean, my_model_1.cov) # 平均はランダムに初期化されるため my_model とは異なる
```

```
[-0.81790845  2.75674636] [[1. 0.]
 [0. 1.]]
```

- 命令する（メソッドを実行する）

In [7]:

```
X = np.random.multivariate_normal(np.array([1.0, 2.0]), np.array([[1.0, 0.9], [0.9, 4.0]]), size=100)
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1])
plt.show()
```



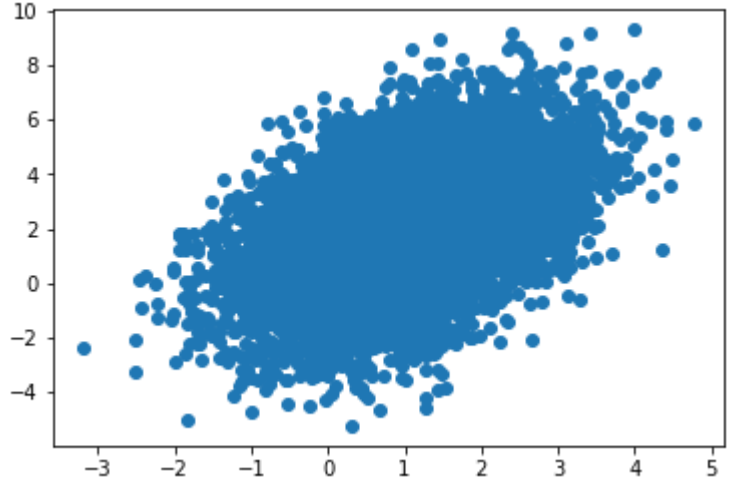
In [8]:

```
my_model.fit(X) # X で最尤推定をして、 mean, cov を更新する
print(my_model.mean)
print(my_model.cov)
```

```
[0.92658078 1.8651461 ]
[[0.80284483 0.74208342]
 [0.74208342 3.86674044]]
```

In [9]:

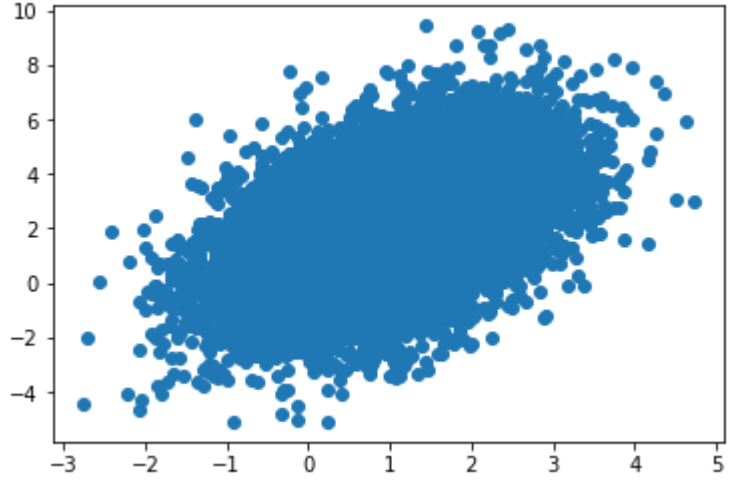
```
# サンプルサイズを大きくすると、真値に近くなる
X = np.random.multivariate_normal(np.array([1.0, 2.0]), np.array([[1.0, 0.9], [0.9, 4.0]]), size=10000)
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1])
plt.show()
my_model.fit(X)
print(my_model.mean, my_model.cov)
```




```
[0.99314407 2.02147572] [[0.9975764  0.88461421]
 [0.88461421 3.99847826]]
```

In [10]:

```
# サンプリングを試してみる
sample = my_model.sample(10000)
plt.scatter(sample[:, 0], sample[:, 1])
plt.show()
```



In [14]:

```
my_model.mean = np.array([0, 0])
my_model.cov = np.identity(2)
np.exp(my_model.log_pdf(np.array([[0,0]]))) # 1次元の Normal distribution だと 0.4 くらいなので、二次元だと0.16くらいのはず
```

Out[14]:

```
array([0.15915494])
```

まとめ

- 統計モデルはクラスを使って書くと便利
    - 内部状態 = モデルのパラメタ、ハイパーパラメタ
    - メソッド = パラメタ推定、予測、入力チェック、内部状態更新
- 最尤推定などを実装するときは
    - まず手続きを数式で書き下す
    - 数式全てを一度に書こうとせず、書けそうなところから書いてみる
    - 組み合わせて完成させる

In [ ]: