応用計量分析２（第4回）

線形代数

担当教員: 梶野 洸（かじの ひろし）

本日の目標

- 線形代数を思い出す
- Python で線形代数の数値計算をやる
- 主成分分析（PCA）を実装する

線形代数の登場人物

- ベクトル $x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
- 行列 $A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
- リストで書けばいい？

In [2]:

```
x = [1, 0] # x をリストで書いてみた
print(x + x) # x + x は、ベクトルでは [2, 0] になってほしいが...
```

```
[1, 0, 1, 0]
```

In [3]:

```
# ベクトルどうしの足し算を関数として定義する必要がある
def add_vectors(x, y):
    return [x[i] + y[i] for i in range(len(x))]
add_vectors(x, x)
```

Out[3]:

```
[2, 0]
```

線形代数で使う計算

色々あるので全部書くのはめんどくさい...

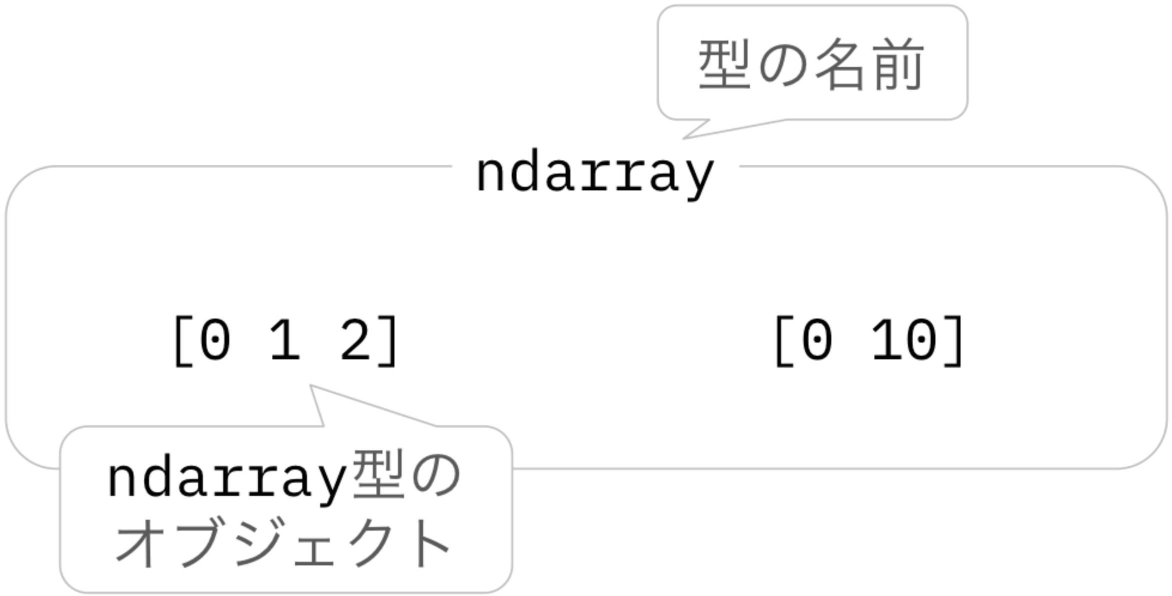
- ベクトル/行列どうしの足し算引き算
- 行列とベクトルの積、行列どうしの積
- ノルム、内積などなど

`numpy` 使う

- 基本的な線形代数の計算が実装されているライブラリ
- リストではなく `numpy.ndarray` というオブジェクトでベクトルを定義する
    - リストどうしの足し算だとリストの連結になってしまう
    - `numpy.ndarray` どうしの足し算だとベクトルの足し算となる！

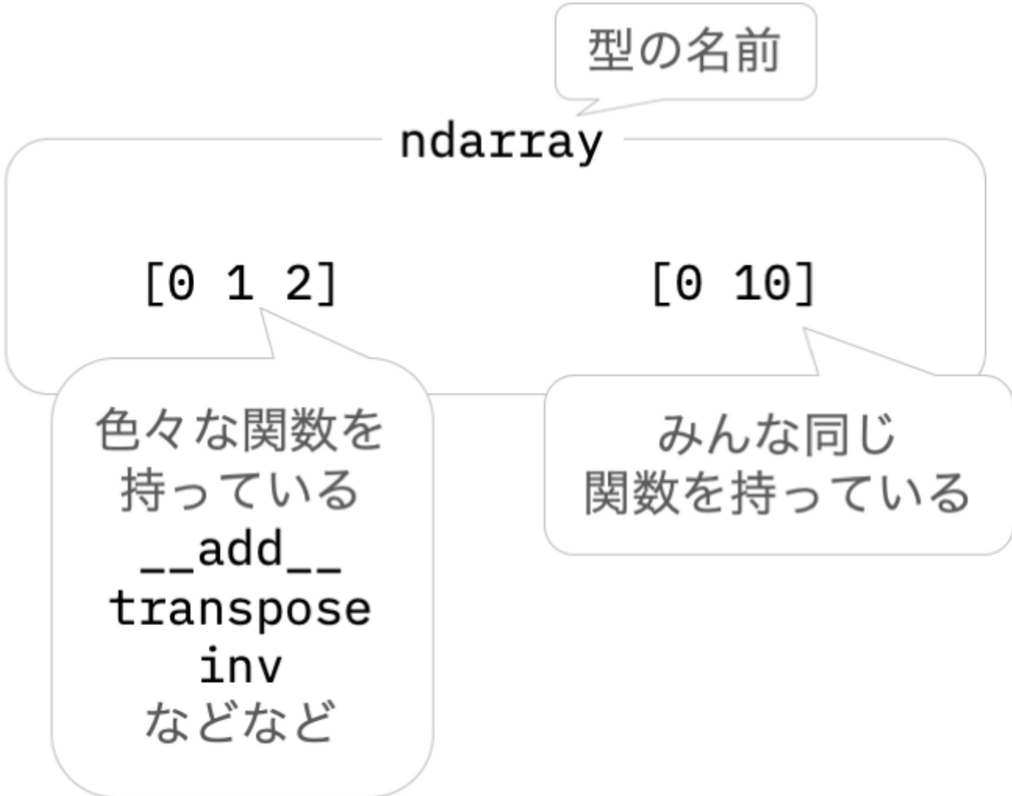
オブジェクト？ `numpy.ndarray`？

- 全てのものはオブジェクトと呼ばれる
  - `1`, `[1,0,3]`, `'hello world'` など
- 数字, 文字列などは、オブジェクトの「型」と呼ばれる
  - 例1. `1` の型は、数字
  - 例2. `[1,0,3]` の型は、リスト
  - `numpy.ndarray` も型
  - オブジェクトの「種類」と理解できる
  - `type` で型を調べられる


型の名前
ndarray
[0 1 2]
[0 10]
ndarray型の
オブジェクト

In [4]:
```
type(1.0)
```
Out[4]:
```
float
```
In [5]:
```
type('1.0')
```
Out[5]:
```
str
```
In [6]:
```
type([1,0])
```
Out[6]:
```
list
```

- それぞれの型には特有の関数がある
  - `<オブジェクト>.<関数名>(<引数>)` と書くのが基本
  - 読み方: <オブジェクト> に、そのオブジェクトに対して定義されている <関数名> を適用する。その時の引数は <引数> で指定する
  - 別の読み方: <オブジェクト> の中の <関数名> を実行する。その時の引数は <引数> で指定する
  - `A.B` は、 `A` の中の `B` という意味合いで広く使われる
    - `A` がオブジェクトではないこともあるし、 `B` が関数でないこともある
  - 何か値が返ってくる場合もあるし、オブジェクトが変更されるだけで何も返ってこない場合もある


型の名前
ndarray
[0 1 2]
[0 10]
色々な関数を
持っている
__add__
transpose
inv
などなど
みんな同じ
関数を持っている

In [7]:
```
x = [1,0] # x にリストを入れる
x.append(3) # リストには `append` という関数が用意されている。引数のオブジェクトを末尾に付け足す機能
# x というオブジェクトに対して、 `append` という関数を適用する。その時に引数として 3 を取る
print(x)
```

```
[1, 0, 3]
```

In [8]:
```
x = [1,0]
print(x + x) # 足し算も、リスト用に特別に用意されている
print(x.__add__(x)) # 上の書き方っぽくするとこう書ける
# x というオブジェクトに対して、 `__add__` という関数を適用する。その時に引数として x を取る
```

```
[1, 0, 1, 0]
[1, 0, 1, 0]
```

ここまで踏まえた上で numpy を使ってみる

- numpy.ndarray 型のオブジェクトを作りたい
  - まず numpy を使えるようにしないといけない
    - 標準ライブラリでないのでそのままでは使えないことがある
    - 今回はインストールは不要（Anacondaに入っている）
  - numpy.array(<リスト>) を実行すると <リスト> を numpy.ndarray に変換したものが返ってくる
    - numpy の中の array という関数を実行している
    - numpy はライブラリ

In [9]:
```
import numpy # numpy というライブラリを使うという宣言
x = numpy.array([1, 0]) # numpy.array という関数を使う。リストを入力するとベクトルを出力する（ベクトルは色々線形計算が定義されている）
print(x)
print(type(x))
```

```
[1 0]
<class 'numpy.ndarray'>
```

In [10]:
```
import numpy as np # numpy を使いたいけど、 numpy という名前だと長いので np という短い名前で呼びたい
x = np.array([1, 0])
print(x)
```

```
[1 0]
```

ベクトル

$x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, y = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ とする

In [11]:

```
x = np.array([1.0, 0])
y = np.array([0, 1.0])
print(x, y)
```

```
[1. 0.] [0. 1.]
```

- ベクトルのスカラー倍: $3x$

In [12]:
```
x = np.array([1.0, 0])
print(3.0 * x) # 実数との掛け算も自然に定義されている
```

```
[3. 0.]
```

- ベクトル同士の足し算・引き算: $x + y, x - y$
- より一般的に線形結合: $3x - 10y$

In [13]:
```
x = np.array([1.0, 0])
y = np.array([0, 1.0])
# ベクトル演算が自然に定義されている
print(x + y)
print(x - y)
print(3 * x - 10 * y)
```

```
[1. 1.]
[ 1. -1.]
[  3. -10.]
```

- 内積: $x \cdot y, (3x - y) \cdot (x + 2y)$

```
In [14]: print(x @ y)
         print((3 * x - y) @ (x + 2 * y))

         print(np.dot(x, y)) # 関数の形で書くこともできる
```

```
0.0
1.0
0.0
```

※内積は、2つベクトルを受け取って、1つのスカラーを返す**関数**としても書ける

- ノルム: $\|2x - y\|_2$

In [15]:
```
print(((2 * x - y) @ (2 * x - y)) ** (0.5)) # 内積を使って計算した場合
print(np.linalg.norm(2 * x - y)) # numpyの関数を使って計算した場合
```

```
2.23606797749979
2.23606797749979
```

- 要素積（アダマール積）: $x \circ y$

各次元で積を取る演算

In [16]:
```
print(x * y)
```

```
[0. 0.]
```

ここまでのまとめ

- ベクトルはnumpyの `ndarray` というオブジェクトで定義する
- 普通の数値と同じような演算ができる
- 内積やノルムなど、線形代数特有の計算の関数もある
    - 内積: `np.dot`, `@`
    - ノルム: `np.linalg.norm`
    - アダマール積: `*`

想定QA

Q. 欲しい関数があるかどうか調べたい

A. ググるかライブラリのAPIを見る（numpyは[ここ (https://docs.scipy.org/doc/numpy/reference/)](https://docs.scipy.org/doc/numpy/reference/)）

- "numpy <ほしい機能>" みたいにググる
    - "numpy 内積" とか "numpy inner product" とか
    - 基本的には公式ドキュメントがもっとも正しいはず
    - プログラミングには英語は必須

Q. `np.dot` とか `np.linalg.norm` とかなんやねん

A. ライブラリは階層構造になっている。

- `np.dot` は、`numpy` (`np`と書いてる)直下に定義された `dot` という関数、
- `np.linalg.norm` は、`numpy` の下の `linalg` (linear algebra; 線形代数)という線形代数の関数をまとめた集まりのなかの `norm` という関数と解釈する

行列

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}$$

In [17]:

```
# np.array にリストのリストを渡すと行列
A = np.array([[1.0, 1.0], [0.0, 2.0]])
print(A)
```

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

行列とベクトルの積

Ax

$x^\top A$

In [18]:

```
print(A @ x)
print(x @ A)
```

```
[1. 0.]
[1. 1.]
```

In [19]:

```
print(A @ np.array([1,2,3,4])) # 2x2の行列に4次元のベクトルは掛けられない
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-a386da54a8ef> in <module>
----> 1 print(A @ np.array([1,2,3,4])) # 2x2の行列に4次元のベクトルは掛けられない

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, wi
th gufunc signature (n?,k),(k,m?)->(n?,m?) (size 4 is different from 2)
```

行列と行列の積

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 2 \end{bmatrix}, B = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$$

In [20]:

```python
A = np.array([[1.0, 1.0],
              [0.0, 2.0]])
B = np.array([[0.0, 1.0],
              [1.0, 2.0]])
print(A @ B)
print(B @ A)
```

```
[[1. 3.]
 [2. 4.]]
[[0. 2.]
 [1. 5.]]
```

演習

- $A = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$ の絶対値が最大の固有値とそれに対応する固有ベクトルを計算せよ

指針

1. 手計算
2. `numpy` で実装されているのをつかう
3. べき乗法
    - ヒント: A を掛け続けるとどうなるか？

In [21]:
```
np.linalg.eigvals(np.array([[2,1],[1,2]]))
```

Out[21]:
```
array([3., 1.])
```

手計算

- 固有値、固有ベクトルの定義: $Av = \lambda v$ を満たす $\lambda, v (\neq 0)$
- 線形方程式 $(A - \lambda I)v = 0$ が非自明な解（つまり$v \neq 0$）を持つような λ を見つければよい
- 特性方程式 $\det(A - \lambda I) = 0$ の解が固有値
- $\det(A - \lambda I) = (2 - \lambda)(2 - \lambda) - 1$

  $= \lambda^2 - 4\lambda + 3 = (\lambda - 3)(\lambda - 1) = 0$
- 絶対値が最大の固有値は 3
- $(A - \lambda I)v = \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = 0$
- $v_1 = v_2$ を満たせばよいので、例えば $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ が固有ベクトル。

numpy で実装されているのを使う

In [22]:
```
A = np.array([[2, 1], [1, 2]])
np.linalg.eig(A)
```

Out[22]:
```
(array([3., 1.]), array([[ 0.70710678, -0.70710678],
        [ 0.70710678,  0.70710678]]))
```

絶対値最大の固有値は 3、対応する固有ベクトルは $\begin{bmatrix} 0.70710678 \\ 0.70710678 \end{bmatrix}$

（返り値の解釈は、[API (https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html)を見ましょう）

べき乗法

- $A \in \mathbb{R}^{N \times N}$ の固有値と対応する固有ベクトルを $\lambda_1, \ldots, \lambda_N, v_1, \ldots, v_N$ とする。
- $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_N|$ とする。

任意のベクトル $x \in \mathbb{R}^N$ は、固有ベクトルで展開できる（固有ベクトルは基底を成す）：

$$x = \sum_{n=1}^{N} c_n v_n$$

A を掛け続けると絶対値最大の固有値に対応する固有ベクトルが（相対的に）強調される:

$$\begin{aligned} A^k x &= \sum_{n=1}^{N} c_n A^k v_n = \sum_{n=1}^{N} c_n \lambda_n^k v_n \\ &= \lambda_1^k \sum_{n=1}^{N} c_n \left(\frac{\lambda_n}{\lambda_1} \right)^k v_n \\ &\approx \lambda_1^k c_1 v_1 \end{aligned}$$

- 適当なベクトルに行列 A を掛け続けると v_1 が求まる
- $\lambda_1 = \dfrac{v_1^\top A v_1}{v_1^\top v_1}$

In [23]:

```
A = np.array([[2, 1], [1, 2]])
x = np.array([1, 2])
for _ in range(100):
    x = A @ x
    x = x / np.linalg.norm(x) # ベクトルを正規化しないと発散する
print(x@A@x / (x@x), x)
```

```
3.0 [0.70710678 0.70710678]
```

線形方程式

$A \in \mathbb{R}^{N\times N}, b \in \mathbb{R}^N$ としたとき、$Ax = b$ を満たす $x \in \mathbb{R}^N$ を求める。

A が正則行列（＝逆行列を持つ）のとき、$x = A^{-1}b$ が解。

二通りの実装方法がある

- 逆行列を求めるアルゴリズム(Gauss-Jordanなど)を利用
- 直接線形方程式を解くアルゴリズム(LU分解)を利用

！！なるべく直接線形方程式を解くアルゴリズムを利用すべき！！

- LU 分解の方がそもそも速い
    - A の形によっては更に速くなる
- `numpy` では逆行列を求めるのに$AX = I$を解いている（＝線形方程式を解くのと同じ計算時間がここで必要）
    - さらに $A^{-1}b$ を計算しないといけないので計算時間的に損

（参考）伊理正夫, 藤野和建: 数値計算の常識

ここまでのまとめ

- ベクトル、行列は numpy を使う
- 固有値・固有ベクトルなどの計算もできる
- 線形方程式も解ける
- 逆行列を求める必要があるか考える（線形方程式を解けばいい場合は線形方程式を解く）

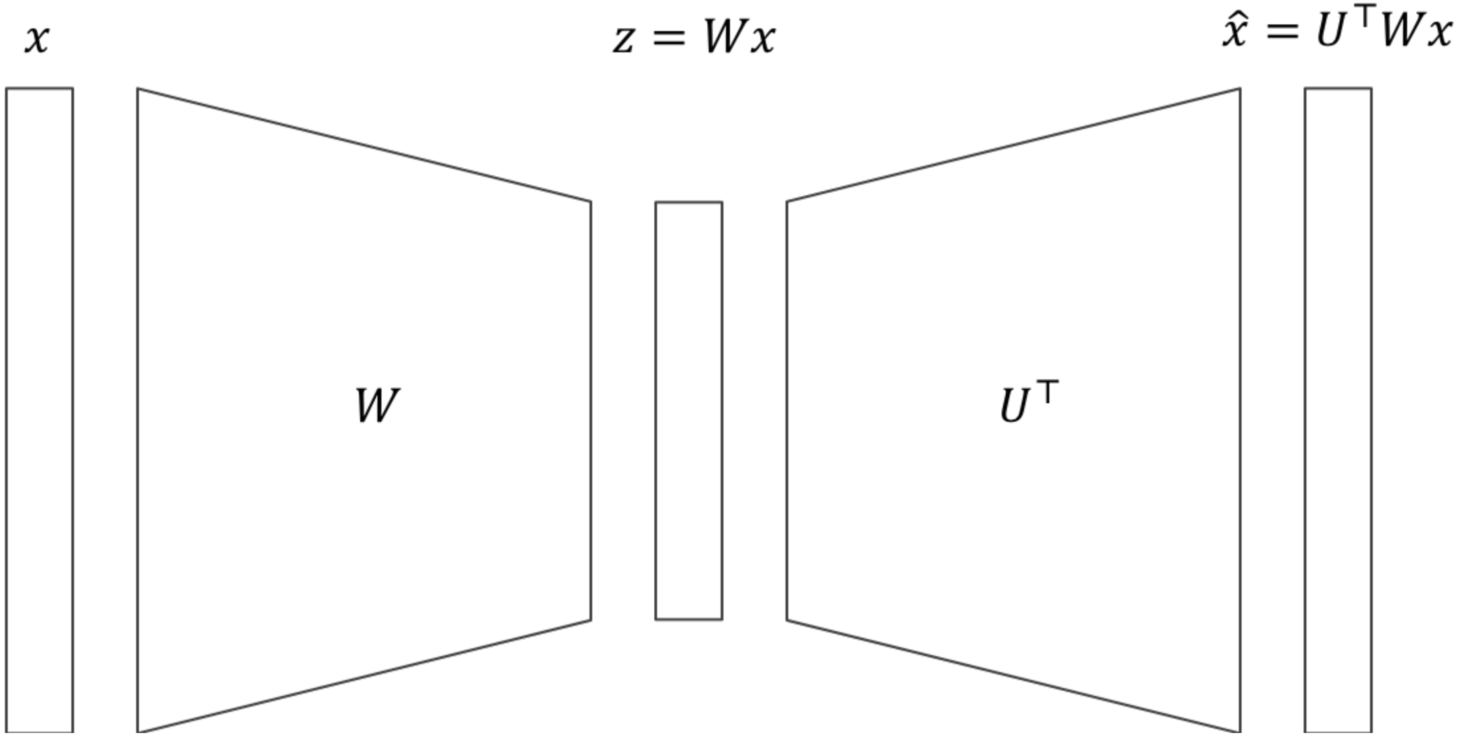
主成分分析, PCA

データ $x_1, \ldots, x_N \in \mathbb{R}^D$ があったとき、その"特性"を保ったまま低次元表現を得たい。

- データを目で見たい（100次元だと見られないけど2次元なら）
- 同じ情報量ならば低次元の方が学習しやすい
- 特性の定義によって様々な手法がある
- $K\ (< D)$次元表現を得る

PCA は、次のような線形変換のペア $\boldsymbol{W}, \boldsymbol{U}$ を求めるという気持ち:

- 入力 $x \in \mathbb{R}^D$
- 低次元表現 $z = \boldsymbol{W}x$ ($\boldsymbol{W} \in \mathbb{R}^{K \times D}$)
- 低次元表現から入力を復元 $\hat{x} = \boldsymbol{U}^\top z = \boldsymbol{U}^\top \boldsymbol{W}x$ ($\boldsymbol{U} \in \mathbb{R}^{K \times D}$)
- $\hat{x} \approx x$ となってほしい



定式化

- $X = \begin{bmatrix} x_1 & x_2 & \dots & x_N \end{bmatrix}^\top$
- 平均0であるとする: $\sum_{n=1}^{N} x_n = 0$
- 以下を満たす W, U を求める

$$\min_{W,U \in \mathbb{R}^{K \times D}} \sum_{n=1}^{N} \|x_n - U^\top W x_n\|_2^2 \tag{1}$$

補題1

式(1)の最適値は

$$\min_{V \in \mathbb{R}^{K \times D},\, VV^\top = I} \sum_{n=1}^{N} \|x_n - V^\top V x_n\|_2^2 \tag{2}$$

の最適値と等しい

証明

- $\boldsymbol{R} = \{\boldsymbol{U}^\top \boldsymbol{W} \boldsymbol{x} \mid \boldsymbol{x} \in \mathbb{R}^D\}$ とすると、$\boldsymbol{R}$は$\mathbb{R}^D$中の$\boldsymbol{K}$次元線型部分空間
- $\boldsymbol{R}$ の正規直交基底を$\boldsymbol{V} = \begin{bmatrix} \boldsymbol{v}_1 & \dots & \boldsymbol{v}_K \end{bmatrix}^\top \in \mathbb{R}^{D\times K}$とする。
- $\boldsymbol{V}^\top \boldsymbol{V} \boldsymbol{x} = \arg\min_{\tilde{\boldsymbol{x}}\in \boldsymbol{R}} \|\boldsymbol{x} - \tilde{\boldsymbol{x}}\|$が成り立つ。
  - $\boldsymbol{R}$ の元は $\boldsymbol{y} \in \mathbb{R}^K$ を用いて$\boldsymbol{V}^\top \boldsymbol{y}$と書けることを利用して示す（演習）
- 式(1)の最適解 $\boldsymbol{U}^\star, \boldsymbol{W}^\star$ と、それに対応する $\boldsymbol{R}^\star, \boldsymbol{V}^\star$ を持ってくると、

$$\|\boldsymbol{x}_n - \boldsymbol{U}^{\star\top} \boldsymbol{W}^\star \boldsymbol{x}_n\|^2 \geq \min_{\tilde{\boldsymbol{x}}_n \in \boldsymbol{R}^\star} \|\boldsymbol{x}_n - \tilde{\boldsymbol{x}}_n\|^2 = \|\boldsymbol{x}_n - \boldsymbol{V}^{\star\top} \boldsymbol{V}^\star \boldsymbol{x}_n\|^2$$

- $\boldsymbol{U}^\star, \boldsymbol{W}^\star$ も $\boldsymbol{V}^\star$ も最適解なので、上式は等号成立。

補題2

最適化問題(2)は

$$\max_{V \in \mathbb{R}^{K \times D}, VV^\top = I} \mathrm{tr}\left(V\left(\sum_{n=1}^{N} x_n x_n^\top\right)V^\top\right) \tag{3}$$

と同値。

証明

展開すれば良い。

$$\begin{aligned}\sum_{n=1}^{N} \|x_n - V^\top V x_n\|^2 &= \sum_{n=1}^{N} \|x_n\|^2 - 2\sum_{n=1}^{N} x_n^\top V^\top V X_n + \sum_{n=1}^{N} x_n^\top V^\top V V^\top V x_n \\ &= \sum_{n=1}^{N} \|x_n\|^2 - \sum_{n=1}^{N} x_n^\top V^\top V x_n\end{aligned}$$

第二項は、

$$\begin{aligned}\sum_{n=1}^{N} x_n^\top V^\top V x_n &= \sum_{n=1}^{N} \mathrm{tr}(x_n^\top V^\top V x_n) = \sum_{n=1}^{N} \mathrm{tr}(x_n x_n^\top V^\top V) \\ &= \mathrm{tr}\left(\left(\sum_{n=1}^{N} x_n x_n^\top\right) V^\top V\right) = \mathrm{tr}\left(V\left(\sum_{n=1}^{N} x_n x_n^\top\right) V^\top\right)\end{aligned}$$

補題3

$A = \sum_{n=1}^{N} x_n x_n^\top$ として、その固有値、固有ベクトルを $\lambda_1, \ldots, \lambda_D, u_1, \ldots, u_D$ とする（$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_D$）。この時任意の $V \in \mathbb{R}^{K \times D}, VV^\top = I$ について

$$\mathrm{tr}\left(VAV^\top\right) \leq \max_{w \in [0,1]^D, \sum_{d=1}^{D} w_d \leq K} \sum_{d=1}^{D} \lambda_d w_d = \sum_{d=1}^{K} \lambda_d$$

が成立

証明

$A = U^\top \Lambda U$ と固有値分解できる（$U \in \mathbb{R}^{D\times D}$）。$V \in \mathbb{R}^{K\times D}, VV^\top = I$ を一つ持ってくる。

$W = VU^\top \in \mathbb{R}^{K\times D}$ と置くと、

- $VAV^\top = VU^\top \Lambda UV^\top$.
  $= W\Lambda W^\top$
- W の各行はD次元空間の正規直交基底: $WW^\top = UV^\top VU^\top = I$

$$\mathrm{tr}(W\Lambda W^\top) = \sum_{d=1}^{D} \lambda_d \sum_{k=1}^{K} w_{k,d}^2$$

$w_d := \sum_{k=1}^{K} w_{k,d}^2$ と置くと、

$$\operatorname{tr}(W\Lambda W^\top) = \sum_{d=1}^{D} \lambda_d \sum_{k=1}^{K} w_{k,d}^2$$
$$= \sum_{d=1}^{D} \lambda_d w_d$$

ここで $0 \leq \sum_{k=1}^{K} w_{k,d}^2 \leq 1 \ (d = 1, \dots, D), \sum_{d=1}^{D} \sum_{k=1}^{K} w_{k,d}^2 = K$ なので、

$$\operatorname{tr}(W\Lambda W^\top) = \sum_{d=1}^{D} \lambda_d w_d$$
$$\leq \max_{w_d \in [0,1], \sum_{d=1}^{D} w_d \leq K} \sum_{d=1}^{D} \lambda_d w_d$$

が成立.

定理

$A = \sum_{n=1}^{N} x_n x_n^\top$ として、その固有値、固有ベクトルを $\lambda_1, \ldots, \lambda_D, u_1, \ldots, u_D$ とする（$\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_D$）。

この時 $V = \left[u_1 \ \ldots \ u_K \right]^\top$ が最適化問題(2), (3)の解

証明

$$\mathrm{tr}(VAV^\top) = \mathrm{tr}(VU^\top \Lambda U V^\top) = \mathrm{tr}(\mathbb{1}_K \Lambda \mathbb{1}_K) = \sum_{d=1}^{K} \lambda_d$$

が成り立つ。ここで

$$[\mathbb{1}_K]_{i,j} = \begin{cases} \delta_{i,j} & \text{if } i = j \leq K \\ 0 & \text{otherwise.} \end{cases}$$

補題3より、これは最適値。

アルゴリズム

- 入力: $x_1, \dots, x_N \in \mathbb{R}^D$, $K \in \mathbb{N}$
- 出力: $z_1, \dots, z_N \in \mathbb{R}^K$

1. $A = \sum_{n=1}^{N} x_n x_n^\top$
2. A の固有値と対応する固有ベクトル $(\lambda_1, u_1), \dots, (\lambda_D, u_D)$ を求める（$\lambda_1 \geq \cdots \geq \lambda_D$）
3. $V = \begin{bmatrix} u_1 \dots u_K \end{bmatrix}^\top$ として、$z_n = V x_n$ を計算

ここまでのまとめ

- PCA はデータを低次元空間に落とす方法
- 再構成時の損失を最小化するようにする

PCA の実装

- PCA でデータを2次元で見てみる
- 主成分を見てみる
- 再構成してみる

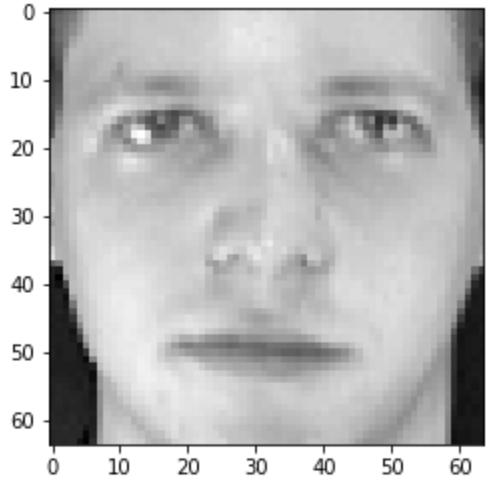
In [24]:

```python
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_olivetti_faces

# データを取得
dataset = fetch_olivetti_faces()
num_examples, row_size, col_size = dataset['images'].shape
X = dataset['data']

# 平均0にしておく（しなくてもまあ大丈夫だけど）
X_mean = X.mean(axis=0)
X_centered = X - X_mean
```

```
/Users/kjn/.pyenv/versions/anaconda3-5.1.0/lib/python3.6/site-packages/sklearn
/externals/joblib/__init__.py:15: DeprecationWarning: sklearn.externals.joblib
is deprecated in 0.21 and will be removed in 0.23. Please import this function
ality directly from joblib, which can be installed with: pip install joblib. I
f this warning is raised when loading pickled models, you may need to re-seria
lize those models with scikit-learn 0.21+.
  warnings.warn(msg, category=DeprecationWarning)
```

In [25]:

```
# 顔データを表示してみる
plt.imshow(dataset['images'][0], cmap=plt.cm.gray)
plt.show()
X_centered.shape
```



Out[25]:

```
(400, 4096)
```

演習

PCA を実行する関数を書け

- 入力: データ $X \in \mathbb{R}^{N \times D}$, 次元 K
- 出力: 変換されたデータ $Z \in \mathbb{R}^{N \times K}$, 変換にもちいる線形変換 $V \in \mathbb{R}^{D \times K}$

ヒント: 対称行列の固有値分解を行う関数 https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh (https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eigh.html#scipy.linalg.eigh)

In [26]:
```
from scipy.linalg import eigh
def pca(X, K):
    A = X.T @ X
    eig_val, eig_vec = eigh(A)
    V = eig_vec[:, -K:]
    z = X @ V
    return z, V
```

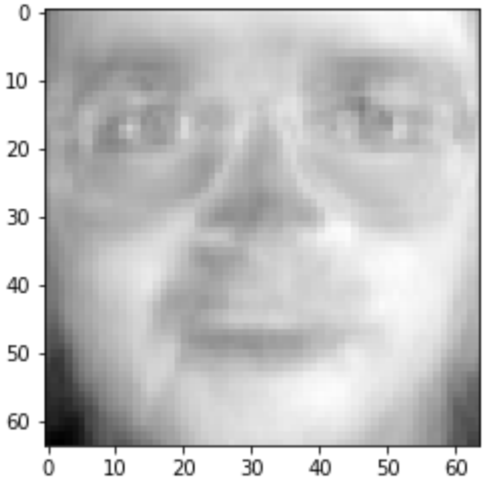
In [27]:
```
# pca を実行
K=20
z, V = pca(X_centered, K)
```

In [28]:
```
# V の行ベクトルが正規直交基底であることを確認
print('distance from the identity:', np.abs(V.T @ V - np.identity(K)).max())

print('mean reconstruction loss: ',((X_centered - z @ V.T) * (X_centered - z @ V.T
)).mean())
```

```
distance from the identity: 1.5404075384140015e-06
mean reconstruction loss:  0.0045594834
```

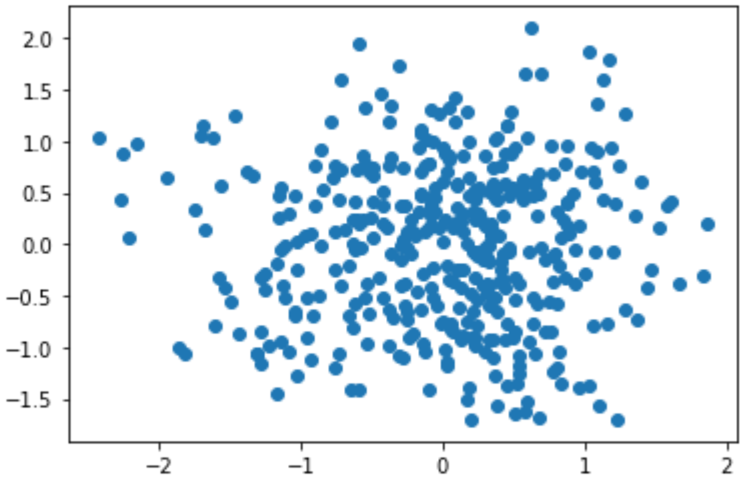
In [29]:

```
# データの貼る空間の固有ベクトルを見てみる
plt.imshow(-V[:, -1].reshape(row_size, col_size), cmap=plt.cm.gray)
plt.show()
```



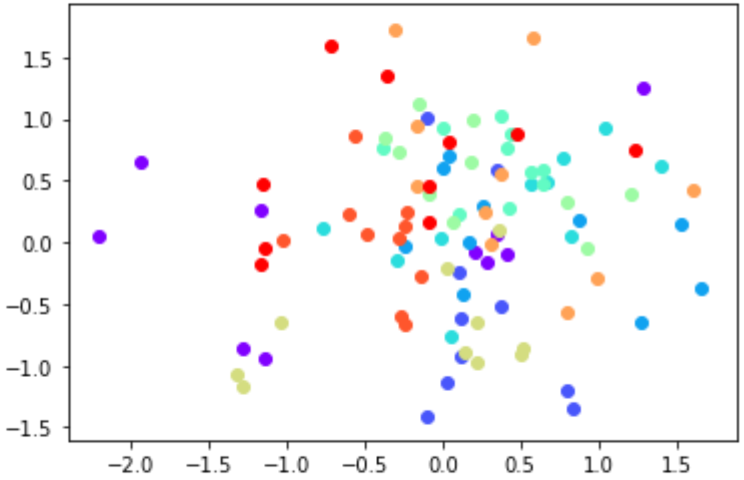
In [30]:
```
plt.scatter(z[:, 0], z[:, 1])
```

Out[30]:
```
<matplotlib.collections.PathCollection at 0x11087aa90>
```



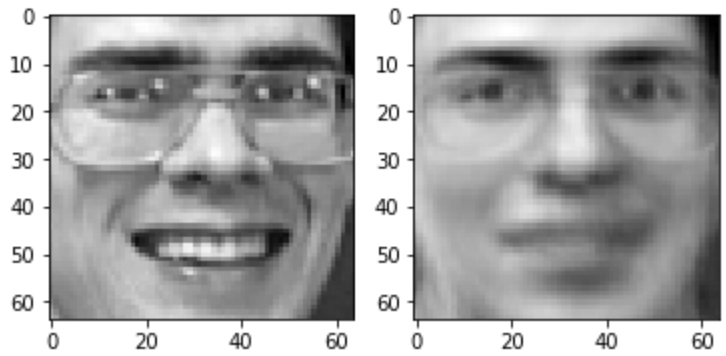
In [31]:

```
# この場合は2次元に落としてもよくわからない...
import matplotlib.cm as cm
import numpy as np

colors = cm.rainbow(np.linspace(0, 1, 10))
for each_idx in range(100):
    plt.scatter(z[each_idx, 0], z[each_idx, 1], color=colors[dataset['target'][eac
h_idx]])
plt.show()
```



In [32]:

```
# 再構成
X_rec = z @ V.T + X_mean
idx = 190

f, (ax1, ax2) = plt.subplots(1, 2)
ax1.imshow(dataset['images'][idx], cmap=plt.cm.gray) # 左が元の画像
ax2.imshow(X_rec[idx].reshape(row_size, col_size), cmap=plt.cm.gray) # 右が再構成画像
plt.show()
```



まとめ

- 行列、ベクトルは `numpy` を使って実装する
- 線形代数の操作は `numpy` の API を探せば実装されていることが多い
- 主成分分析(principle component analysis; PCA) を実装した
    - データを低次元空間に射影するアルゴリズム
    - 再構成時の損失を最小化する
    - 固有値分解に帰着される