

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)
        A = [[1, 0], [0, 1]]
        #print(A)
        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):
            z = []
            for i in range(len(x)):
                z.append(x[i] + y[i])
            return z
        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（`repl.it` 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))
print(type([1, 0]))
```

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

```
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
         # 呼びたい
x = np.array([1, 0])
print(x)
```

```
[1 0]
```

ベクトル

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

```
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, 2.0])  
         print(3.0 * x) # 実数との掛け算も自然に定義されている
```

```
[3. 6.]
```

- 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]: x = np.array([1.0, 10.0])  
y = np.array([0, 1.0])  
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
```

```
10.0  
351.0  
10.0
```

※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]: x = np.array([1.0, 10.0])
y = np.array([0, 1.0])
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
```

19.1049731745428
19.1049731745428

- 要素積（アダマール積）： $x \circ y$

各次元で積を取る演算

```
In [16]: x = np.array([1.0, 10.0])  
         y = np.array([0, 1.0])  
         print(x * y)
```

```
[ 0. 10.]
```

```
In [23]: x = np.array([0, 1, 2, 3, 4])
print(x[1]) # 1 番目の要素
print(x[0:2]) # 0 から2番目の要素 (2番目は含まない)
print(x[-1]) # 一番最後の要素
print(x[-3:-1]) # 最後から3番目～1番目の要素 (-1番目を含まない)
print(x[-3:]) # 最後から3番目～最後の要素
```

1

[0 1]

4

[2 3]

[2 3 4]

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: `*`
- 構成要素の取り出し方は色々ある

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

演習4.1

1. `input_list` を入力とし、それを `numpy.ndarray` に変換して出力する関数 `list2ndarray` を実装せよ

- `input_list` はリスト型のオブジェクトで、各要素は `int` または `float` 型と仮定する

2. `x_array, y_array` という二つの `numpy.ndarray` を入力とし、`x_array` と `y_array` の差のL2ノルムを出力する関数 `dist` を実装せよ

- `x_array, y_array` は `numpy.ndarray` 型のオブジェクトで、同じ系列長であると仮定する

3. `x_array` を入力とし、その一番はじめの要素と最後の要素を取り除いた `numpy.ndarray` を出力する関数 `extract` を実装せよ

- `x_array` は `numpy.ndarray` 型のオブジェクトで系列長は3以上だと仮定する

4. `x_array` と `idx` を入力とし、`x_array` の `idx` 番目の要素を0に書き換える関数 `drop` を実装せよ

- `x_array` は `numpy.ndarray` 型のオブジェクトであると仮定し、`x_array` の系列長を `L` とする
- `idx` は `int` 型のオブジェクトでかつ0以上 `L-1` 以下の値をとると仮定する

行列

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

```
In [24]: # 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

$x^T A$

```
In [18]: x = np.array([1, 0])
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]: 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.]]
```

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

演習4.2

1. $D \times D$ 正則行列 A と D 次元ベクトル x を入力として、 $\sqrt{x^T A^{-1} x}$ を出力する関数 `quadratic` を完成させよ。ただし A, x 共に `numpy.array` として与えられるとする。
2. $D \times D$ 対称行列 A を入力として、 A の第二固有値（固有値の中で二番目に大きいもの）を出力する関数 `second_eig` を完成させよ。
3. $D \times D$ 行列 A 、 D 次元ベクトル x 、自然数 k を入力として、

$$v_1 = \frac{A^k x}{\|A^k x\|}$$

$$\lambda_1 = v_1^T A v_1$$

で定義される λ_1 を出力する関数 `power_iter` を完成させよ。

解説: 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 $\lambda_1, \dots, \lambda_N, v_1, \dots, v_N$ とする。
- $|\lambda_1| > |\lambda_2| > \dots > |\lambda_N|$ 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:

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

$$= \lambda_1^k \sum_{n=1}^N c_n \left(\frac{\lambda_n}{\lambda_1} \right)^k v_n$$

$$\approx \lambda_1^k c_1 v_1$$

- 適当なベクトルに行列 A を掛け続けると v_1 が求まる

- $\lambda_1 = \frac{v_1^T A v_1}{v_1^T v_1}$

```
In [22]: import numpy as np
A = np.array([[2, 1], [1, 2]])
x = np.array([1, 2])
for i 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]
```

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

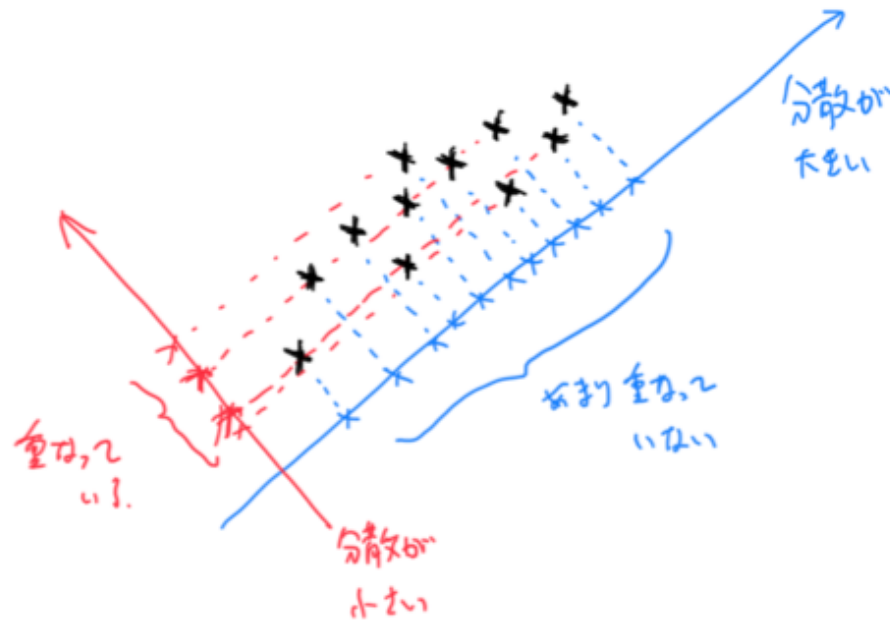
データ $x_1, \dots, x_N \in \mathbb{R}^D$ があったとき、その"特性"を保ったまま低次元表現を得たい。

- 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

主成分分析（1次元の場合）

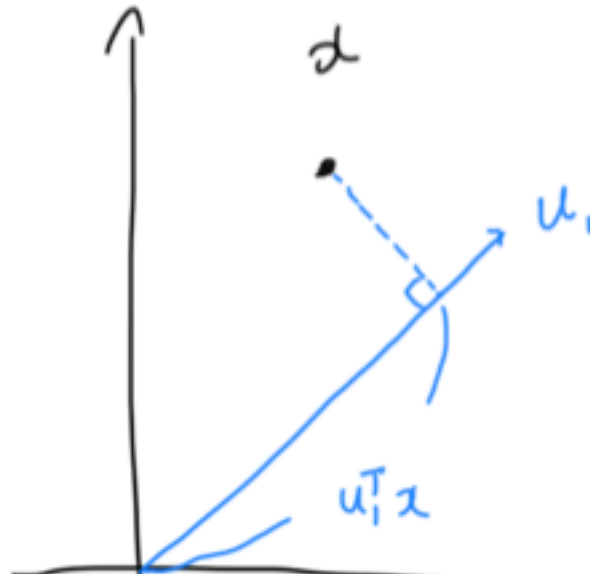
Q. データを1次元に射影するとき、どのように射影すれば一番データの特徴を保存できるか？

A. データの分散が最も大きくなる軸に射影すれば良さそう



主成分分析（1次元の場合）の定式化

- $X = [x_1 \ x_2 \ \dots \ x_N]^T$
 - データの平均を $\bar{x} = \frac{1}{N} \sum_{n=1}^N x_n$ とする
- $u_1 \in \mathbb{R}^D$ で定められる軸に射影することを考える
 - $u_1^T u_1 = 1$ とする
- u_1 で定められる軸上での x_n の座標は $u_1^T x_n$
- u_1 で定められる軸上での X の分散は $\frac{1}{N} \sum_{n=1}^N (u_1^T x_n - u_1^T \bar{x})^2$



主成分分析（1次元の場合）の定式化

分散が最大になる方向が知りたいので、以下の最適化問題を解く

$$\text{maximize}_{\mathbf{u}_1 \in \mathbb{R}^D} \frac{1}{N} \sum_{n=1}^N (\mathbf{u}_1^\top \mathbf{x}_n - \mathbf{u}_1^\top \bar{\mathbf{x}})^2 \text{ subject to } \mathbf{u}_1^\top \mathbf{u}_1 = 1$$

主成分分析（1次元の場合）の解法

まず目的関数を書き換える

$$\begin{aligned}\frac{1}{N} \sum_{n=1}^N (u_1^\top x_n - u_1^\top \bar{x})^2 &= \frac{1}{N} \sum_{n=1}^N u_1^\top (x_n - \bar{x})(x_n - \bar{x})^\top u_1 \\ &= u_1^\top \Sigma u_1\end{aligned}$$

where $\Sigma = \frac{1}{N} \sum_{n=1}^N (x_n - \bar{x})(x_n - \bar{x})^\top$.

すると最適化問題は以下のように書き換わる

$$\text{maximize}_{u_1 \in \mathbb{R}} u_1^\top \Sigma u_1 \text{ subject to } u_1^\top u_1 = 1$$

主成分分析（1次元の場合）の解法

ラグランジュ未定乗数法を使う。ラグランジアンは

$$L(u_1; \lambda_1) = u_1^T \Sigma u_1 + \lambda_1 (1 - u_1^T u_1)$$

最適解 u_1^* で停留点になっていることが必要なので、

$$\frac{\partial}{\partial u_1} L(u_1^*; \lambda_1) = \Sigma u_1^* - \lambda_1 u_1^* = 0$$

つまり λ_1 は Σ の固有値で u_1^* はそれに対応する（単位）固有ベクトルであることが必要。また目的関数は

$$u_1^T \Sigma u_1 = \lambda_1$$

となるため、 λ_1 は Σ の最大固有値で、 u_1^* は最大固有値に対応する長さ1の固有ベクトルである。

主成分分析（2次元以上）について

- 第一主成分は分散共分散行列 Σ の最大固有値に対応する固有ベクトルだった。
- Q. データを $K (\geq 2)$ 次元に落としたい場合はどうすればいいのか？
- A. K 次元空間に落とした時の分散を考えれば良さそう
 - Σ の固有値の大きい方から K 個とってきて、対応する固有ベクトルも持ってくる: $\{(\lambda_k, u_k)\}_{k=1}^K$
 - $U = [u_1 \dots u_K]^T \in \mathbb{R}^{K \times D}$ として、 U で K 次元空間に射影したらいい
 - 証明略

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. $\bar{x} = \frac{1}{N} \sum_{n=1}^N x_n$

2. $\Sigma = \frac{1}{N} \sum_{n=1}^N (x_n - \bar{x})(x_n - \bar{x})^\top$

3. Σ 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 ($\lambda_1 \geq \dots \geq \lambda_D$)

- ただし $\|u_d\| = 1$ for all $d = 1, \dots, D$.

4. $U = [u_1 \dots u_K]^\top$ として、 $z_n = Ux_n$ を計算

ここまでのまとめ

- PCA は分散共分散行列を固有値分解すればできる
- 固有値（+固有ベクトルも）の大きい方から順番にとってくればいい

PCA の実装

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

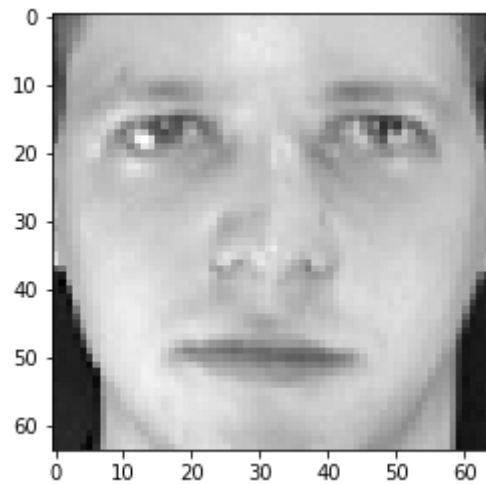
→ 見て楽しいので画像データを使ってみる

```
In [2]: import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_olivetti_faces

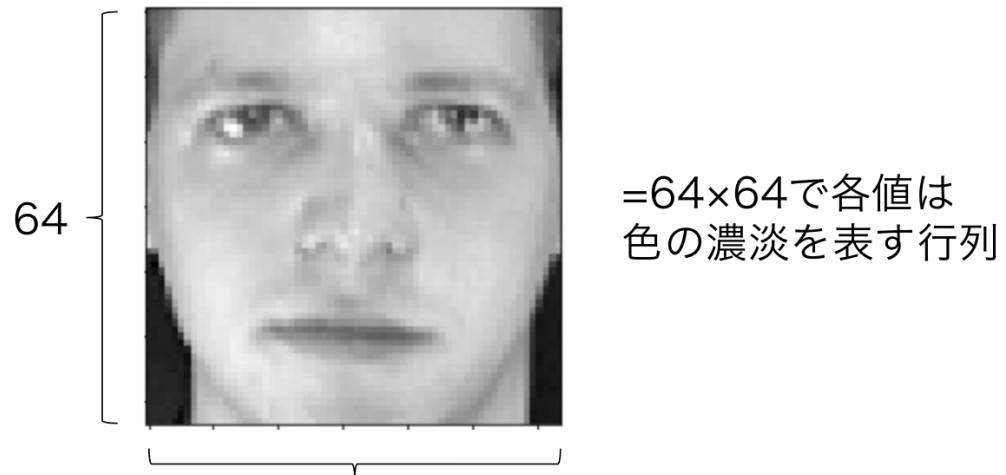
# データを取得
dataset = fetch_olivetti_faces()
num_examples, row_size, col_size = dataset['images'].shape
X = dataset['data']

# 平均0にしておく
X_mean = X.mean(axis=0)
X_centered = X - X_mean
```

```
In [3]: # 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[3]: (400, 4096)
```



=64×64で各値は
色の濃淡を表す行列

64

各行を連結して一本にするとベクトルになる



プレ演習

上で定義した $x_centered$ に対してPCAのアルゴリズムを適用した時に得られる低次元表現 $Z \in \mathbb{R}^{N \times K}$ と変換に用いる行列 $U \in \mathbb{R}^{K \times D}$ を求めるプログラムをかけ

小問

1. $\Sigma = \sum_{n=1}^N x_n x_n^T$ を計算せよ (x_n の平均は0に変換済み)
2. Σ の固有値と対応する固有ベクトル $(\lambda_1, u_1), \dots, (\lambda_D, u_D)$ を計算せよ ($\lambda_1 \geq \dots \geq \lambda_D$)
 - ヒント: 対称行列の固有値分解を行う関数 <https://numpy.org/doc/stable/reference/generated/numpy.linalg.eigh.html> (<https://numpy.org/doc/stable/reference/generated/numpy.linalg.eigh.html>)
3. $U = [u_1 \dots u_K]^T$ として、 $z_n = Ux_n$ を計算せよ
 - 例えば $K = 20$

1. $\Sigma = \sum_{n=1}^N x_n x_n^T$ を計算せよ

```
In [4]: sample_size = X_centered.shape[0]
dim = X_centered.shape[1]

# 定義通り地道に sigma を作ってもいい
sigma = np.zeros((dim, dim))
for each_example in range(sample_size):
    sigma = sigma + np.outer(X_centered[each_example], X_centered[each_example])
```

$X = [x_1, \dots, x_N]^T$ としたとき、

$$X^T X = \sum_{n=1}^N x_n x_n^T$$

という関係を使ってもいい（こっちの方が圧倒的に速い）

```
In [5]: # sigma には上で計算したものが入っている  
print(np.linalg.norm(sigma - X_centered.T @ X_centered))
```

```
0.0015574189977048496
```

1. A の固有値と対応する固有ベクトル $(\lambda_1, u_1), \dots, (\lambda_D, u_D)$ を計算せよ ($\lambda_1 \geq \dots \geq \lambda_D$)

```
In [6]: from numpy.linalg import eigh
eig_val, eig_vec = eigh(sigma)
print(eig_val)
print(eig_vec.shape)
```

```
[-1.40173184e-06 -1.39795718e-06 -1.38039418e-06 ...  2.51554128e+03
  4.41763308e+03  7.51723015e+03]
(4096, 4096)
```

1. $U = [u_1 \dots u_K]^T$ として、 $z_n = Ux_n$ を計算せよ

```
In [7]: K = 20  
U = eig_vec[:, -K:].T  
print(U.shape)  
z = (U @ X.T).T  
print(z.shape)
```

```
(20, 4096)
```

```
(400, 20)
```

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 $U \in \mathbb{R}^{K \times D}$

```
In [8]: from numpy.linalg import eigh
def pca(X, K):
    X = X - np.mean(X, axis=0)
    sigma = X.T @ X
    eig_val, eig_vec = eigh(sigma)
    U = eig_vec[:, -K:].T
    z = (U @ X.T).T
    return z, U
```

```
In [9]: # pca を実行
K=20
z, U = pca(X_centered, K)
print(z.shape, U.shape)
```

```
(400, 20) (20, 4096)
```

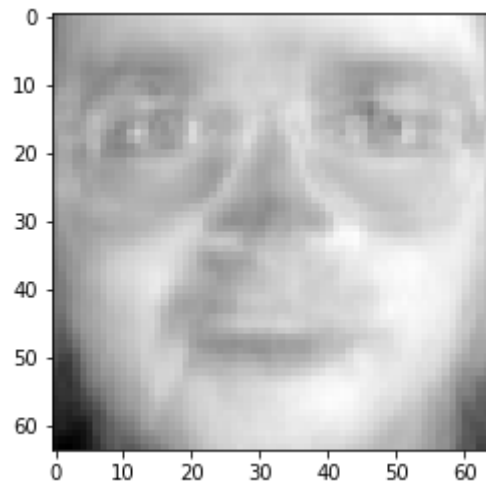


```
In [10]: # v の行ベクトルが正規直交基底であることを確認
print('distance from the identity:', np.abs(U @ U.T - np.identity(K)).max())
print('mean reconstruction loss: ', ((X_centered - (U.T @ z.T).T) * (X_centered -
(U.T @ z.T).T)).mean())
```

distance from the identity: 1.1920928955078125e-07

mean reconstruction loss: 0.0045594834

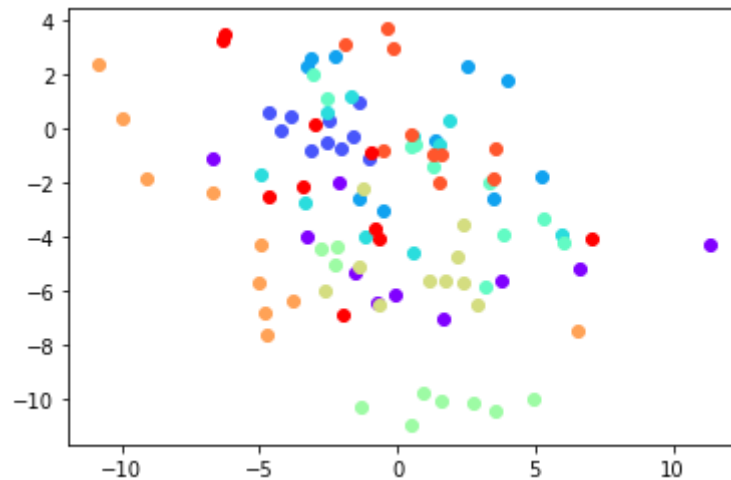
```
In [11]: # データの貼る空間の固有ベクトルを試みる  
plt.imshow(-U[-1].reshape(row_size, col_size), cmap=plt.cm.gray)  
plt.show()
```



```
In [12]: # 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

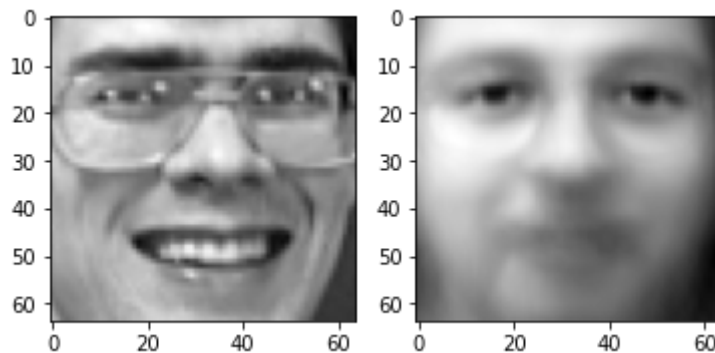
K = 2
z, U = pca(X_centered, K)

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'][each_idx]])
plt.show()
```



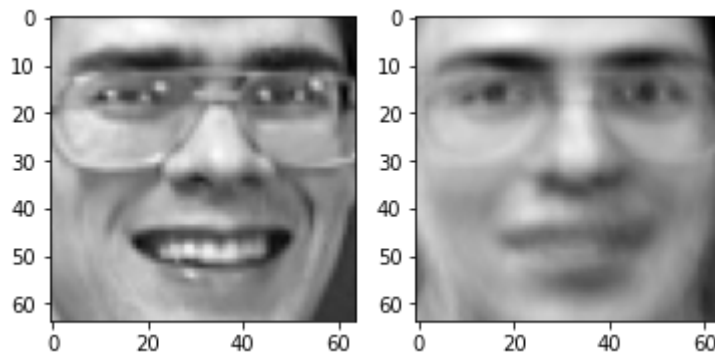
```
In [17]: # 再構成 (K=2)
K = 2
z, U = pca(X_centered, K)
X_rec = (U.T @ z.T).T + X_mean
idx = 190

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()
```



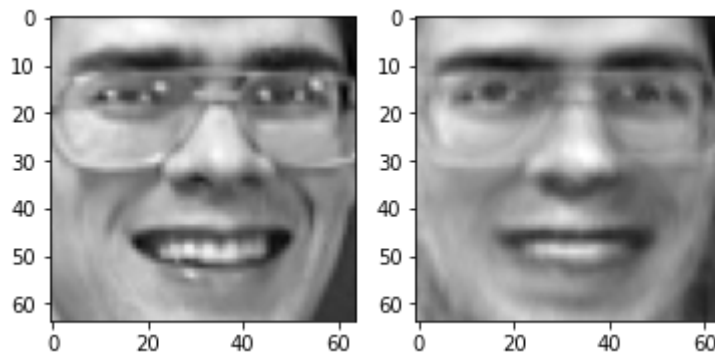
```
In [18]: # 再構成 (K=20)
K = 20
z, U = pca(X_centered, K)
X_rec = (U.T @ z.T).T + X_mean
idx = 190

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()
```



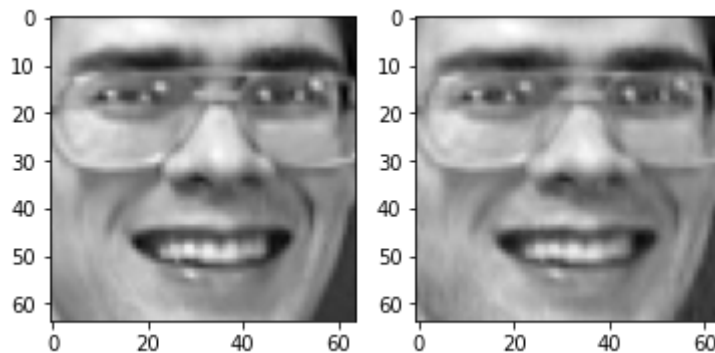
```
In [19]: # 再構成 (K=50)
K = 50
z, U = pca(X_centered, K)
X_rec = (U.T @ z.T).T + X_mean
idx = 190

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()
```



```
In [20]: # 再構成 (K=200)
K = 200
z, U = pca(X_centered, K)
X_rec = (U.T @ z.T).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
 - 低次元空間での分散を最小化する
 - 固有値分解に帰着される

以下、2020年の講義では用いない資料