B 4 本読み第 5 回

内積と直交性

2020/05/08

実装担当：松林慶祐 / テスト担当：重見開

目次

環境

- 実装環境
    - macOS Catalina (10.15.4)
    - Python 3.7.7
- スライド制作環境
    - Windows Home 64bit (1903)
    - Jupyter Notebook 6.0.3
    - Python 3.7.7

ユークリッド空間

ユークリッド内積を導入した空間のこと． 一般的に知られている内積はユークリッド内積といい，\二つのベクトル $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{C}^N$ に対して，以下の式で表される．
$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{y}^H \boldsymbol{x} = x_1 \overline{y_1} + x_2 \overline{y_2} + \cdots + x_N \overline{y_N}$

また，内積はスカラであることから，順番を入れ替えると共役になる．
$\langle \boldsymbol{y}, \boldsymbol{x} \rangle = \boldsymbol{x}^H \boldsymbol{y} = (\boldsymbol{y}^H \boldsymbol{x})^H = \overline{\boldsymbol{y}^H \boldsymbol{x}} = \overline{\langle \boldsymbol{x}, \boldsymbol{y} \rangle}$

実数のＮ次元ユークリッド空間の場合は，
共役を考える必要がないため，簡潔に表記できる．
$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = \boldsymbol{y}^T \boldsymbol{x} = x_1 y_1 + x_2 y_2 + \cdots + x_N y_N$

ベクトルの直交性

二つのベクトル $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{C}^N$ の内積に対して,

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle = 0$$

が成り立つとき,
$\boldsymbol{x}$ と $\boldsymbol{y}$ は**直交する**という.

実装課題 1

ユーグリッド内積を求める（ついでに直交性も判定）関数を実装

関数の動作

1. 二つのベクトルを入力（縦ベクトルを想定）
2. 縦と横の場合はエラー，横同士の場合は縦に変換
3. 2つ目に入力されたベクトルを共役なものに変換
4. 成分同士を掛け合わせて加算
5. 内積の結果が 0 であったら直交であると判断
6. 結果を縦ベクトルで出力

行列ベクトルを表すためにnumpyを使用

In [1]:
```
import numpy as np
```

In [2]:

```python
def euclidean_inner_product(vector1, vector2, show_orth=False):
    # numpy配列変換
    x = np.array(vector1)
    y = np.array(vector2)
    # エラー処理
    try:
        if not x.shape == y.shape: #横×縦, 成分数違いを検知
            raise ValueError("ベクトルのサイズが違います")
        else:
            pass
    except ValueError as ve:
        print(ve)
        return  # 関数を抜けてNoneを返す
    # 共役をとる
    conj_y = y.conj()
    # 成分ごとの積をとって加算
    tmp = []
    for i in range(x.size):
        tmp.append(x[i][0] * conj_y[i][0])
    res = sum(tmp)
    # 直交性の判断
    if round(res, 7) == 0:
        is_orthogonal = True
    else:
        is_orthogonal = False
    # 直交性の判別を返すかどうか
    if show_orth == True:
        return res, is_orthogonal
    else:
        return res
```

In [3]:

```
x = [[2-1j], [1j], [-1j]]
y = [[3j], [1+1j], [4+1j]]
x_ip_y, xy_orth = euclidean_inner_product(x, y, show_orth=True)
y_ip_x, yx_orth = euclidean_inner_product(y, x, show_orth=True)
print("<x,y> = {0}, 直交かどうか={1}¥n<y,x> = {2}, 直交かどうか={3}".format(x_ip_y, xy_ort
h, y_ip_x, yx_orth))
x = [[2], [3], [-1]]
y = [[4], [-2], [2]]
x_ip_y, xy_orth = euclidean_inner_product(x, y, show_orth=True)
y_ip_x, yx_orth = euclidean_inner_product(y, x, show_orth=True)
print("<x,y> = {0}, 直交かどうか={1}¥n<y,x> = {2}, 直交かどうか={3}".format(x_ip_y, xy_ort
h, y_ip_x, yx_orth))
```

```
<x,y> = (-3-9j), 直交かどうか=False
<y,x> = (-3+9j), 直交かどうか=False
<x,y> = 0, 直交かどうか=True
<y,x> = 0, 直交かどうか=True
```

ノルム

ベクトルの長さを決める量のこと.
内積空間のベクトル $\boldsymbol{x}$ に対して

$$\|\boldsymbol{x}\| = \sqrt{\langle \boldsymbol{x}, \boldsymbol{x} \rangle}$$

をノルムと呼ぶ.

特に，ノルムが 1 のとき，$\boldsymbol{x}$ は**単位ベクトル**であるという.

ユークリッド空間のベクトル $\boldsymbol{x} \in \mathbb{C}^N$ の場合に，ノルムは

$$\|\boldsymbol{x}\| = \sqrt{\sum_{i=1}^{N} |x_i|^2}$$

のように要素で書き下すことができ，特にこのノルムを l_2 ノルム，または 2- ノルムと呼ぶ.

一方，

$$\|\boldsymbol{x}\|_1 = \sum_{i=1}^{N} |x_i|$$

のように定義されるノルムを，l_1 ノルム，または 1- ノルムとよび，信号処理や機械学習で疎性（スパース性）を測るのに広く使われる.

実装課題 2

n- ノルムを求める（引数はベクトルとn）関数を実装

関数の動作

1. ノルムを求めるベクトルとｎを入力する
2. ベクトルの成分ごとに絶対値をとり，ｎ乗する
3. ｎ乗したものの総和をとり，ｎ乗根を求める

In [4]:

```
def n_norm(vector, n=2):
    # 縦ベクトルに変換
    vec = np.array(vector).reshape(-1, 1)

    # 計算
    tmp = []
    for i in range(vec.size):
        # print(abs(vec[i][0]))
        tmp.append(abs(vec[i][0]) ** n)  # 絶対値のn乗(たぶん複素数もOK)

    # 全部加算してn乗根
    res = sum(tmp)**(1 / n)

    return res

print("整数のみ {0}".format(x))
x = [[-3], [2], [2], [3]]
norm1 = n_norm(x,1)
norm2 = n_norm(x)
print("1-ノルムは {0}¥n2-ノルムは {1}".format(norm1,norm2))
print("異なる型同士 {0}".format(x))
x = [[-3], [5.6], [4+1j]]
norm1 = n_norm(x,1)
norm2 = n_norm(x)
print("1-ノルムは {0}¥n2-ノルムは {1}".format(norm1,norm2))
```

```
整数のみ [[2], [3], [-1]]
1-ノルムは 10.0
2-ノルムは 5.0990195135927845
異なる型同士 [[-3], [2], [2], [3]]
1-ノルムは 12.72310562561766
2-ノルムは 7.573638491504595
```

コサイン類似度

ノルムと内積によって導入される，二つのベクトルの角度のことで，パターン間の距離を測るのに用いられる.

$$\cos\theta = \frac{\langle \boldsymbol{x}, \boldsymbol{y} \rangle}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|}$$

ベクトル同士の向きが近いほどなす角度が小さくなるので，コサイン類似度は小さくなり，向きが異なるほど，コサイン類似度は大きくなる.

コサイン類似度は -1 から 1 の値をとり，
コサイン類似度から $0 \leq \theta \leq \pi$ の範囲で決まる θ を**角度**と呼ぶ.

実装課題 3

コサイン類似度を求める関数を実装

関数の動作

1. 二つのベクトルを入力
2. 課題 1 の関数を使って内積を計算
3. 課題 2 の関数を使ってそれぞれのベクトルの 2- ノルムを計算
4. 定義に従ってコサイン類似度を計算

In [5]:

```python
def cosine_similarity(vector1, vector2):
    # 内積計算
    ip_xy = euclidean_inner_product(vector1, vector2)

    # ノルム計算
    norm_x = n_norm(vector1)
    norm_y = n_norm(vector2)

    # コサイン類似度計算
    res = ip_xy / (norm_x * norm_y)

    return res

x = [[0.4], [5.6], [-0.07]]
y = [[-1.1], [0.22], [-3.3]]
print("コサイン類似度は {0}".format(cosine_similarity(x, y)))
```

```
コサイン類似度は 0.05227442125730368
```

正規直交基底

N次元ベクトル空間の基底ベクトル $\{\boldsymbol{u_i}\}_{i=1}^{N}$ が

- すべて互いに直交
- すべての基底ベクトルのノルムが 1

のとき，この基底を**正規直交基底**と呼ぶ.

正規直交展開

N次元ベクトル空間 V の基底 $\{\boldsymbol{u_i}\}_{i=1}^{N}$ が正規直交性を満たすとき，V 内のベクトル $\boldsymbol{x} \in V$ は基底 $\{\boldsymbol{u_i}\}_{i=1}^{N}$ を用いて，

$$\boldsymbol{x} = \langle \boldsymbol{x}, \boldsymbol{u_1} \rangle \boldsymbol{u_1} + \langle \boldsymbol{x}, \boldsymbol{u_2} \rangle \boldsymbol{u_2} + \cdots + \langle \boldsymbol{x}, \boldsymbol{u_N} \rangle \boldsymbol{u_N}$$

と展開できる．これを**正規直交展開**という.

$\langle \boldsymbol{x}, \boldsymbol{u_N} \rangle$ が展開係数となっている.

ユニタリ行列

ある行列の逆行列とエルミート転置が一致しているような行列のこと．
すなわち，$\boldsymbol{A}^{-1} = \boldsymbol{A}^{H}$ より $\boldsymbol{A}\boldsymbol{A}^{H} = \boldsymbol{A}^{H}\boldsymbol{A} = \boldsymbol{I}$
が成り立つような行列のことをユニタリ行列という．

ユニタリ行列の列ベクトルは正規直交性を持つ．

正規直交展開の展開係数 $c_i = \langle \boldsymbol{x}, \boldsymbol{u_i} \rangle = \boldsymbol{u_i}^H \boldsymbol{x}$ を並べたベクトルを考えると，

$$\boldsymbol{c} = \begin{bmatrix} \boldsymbol{u_1}^H \boldsymbol{x} \\ \boldsymbol{u_2}^H \boldsymbol{x} \\ \vdots \\ \boldsymbol{u_N}^H \boldsymbol{x} \end{bmatrix} = [\boldsymbol{u_1}, \boldsymbol{u_2}, \cdots, \boldsymbol{u_N}]^H \boldsymbol{x} = \boldsymbol{U}^H \boldsymbol{x}$$

のように表せる．この時 $\boldsymbol{U}$ は正規直交基底が列ベクトルとなっている行列である．
つまり $\boldsymbol{U}$ の列ベクトルが正規直交性をもつので，$\boldsymbol{U}$ はユニタリ行列である．

ノルムについて考えると，
$\|\boldsymbol{c}\|^2 = \|\boldsymbol{U}^{\boldsymbol{H}} \boldsymbol{x}\|^2 = \|\boldsymbol{x}\|^2$
となっていることから，ユニタリ行列は**ノルム変えない**性質を持つ．

実装課題 4

ユニタリ行列かどうか判定する関数を実装

判別方法

1. $\boldsymbol{A}^{-1} = \boldsymbol{A}^H$ （逆行列とエルミート転置が一致する）
2. $\boldsymbol{A}\boldsymbol{A}^H = \boldsymbol{A}^H\boldsymbol{A} = \boldsymbol{I}$ （逆行列とエルミート転置の積が単位行列になる）

$\Rightarrow$ 2番の方法を採用

関数の動作

1. 行列を入力
2. 入力された行列のエルミート転置を取得
3. 行列とそのエルミート転置の積を計算
4. 用意しておいた単位行列と積が等しいか判別

In [6]:

```python
def unitary_matrix(matrix):

    uni = np.array(matrix)
    her_mat = np.conjugate(uni.T)
    ide_mat = uni @ her_mat
    ide_mat2 = np.identity(len(uni))

    for i in range(len(uni)):
        for j in range(len(uni)):
            if not round(ide_mat[i][j] - ide_mat2[i][j], 7) == 0:
                unitary = False
            else:
                unitary = True
    return unitary
```

In [7]:

```
mat = (1/math.sqrt(2)) * np.array([[1,1],[-1j,1j]])
print(unitary_matrix(mat))
mat = np.array([[1,1],[-1j,1j]])
print(unitary_matrix(mat))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-7-c8a67088e0fc> in <module>
----> 1 mat = (1/math.sqrt(2)) * np.array([[1,1],[-1j,1j]])
      2 print(unitary_matrix(mat))
      3 mat = np.array([[1,1],[-1j,1j]])
      4 print(unitary_matrix(mat))

NameError: name 'math' is not defined
```

グラム・シュミットの正規直交化

直交ではない基底から，正規直交基底を生成する方法のこと.
以下の手順により，直交しているとは限らない基底 $\{\boldsymbol{u_i}\}_{i=1}^{N}$ から 正規直交基底 $\{\boldsymbol{v_i}\}_{i=1}^{N}$ を得ることができる.

1. $\boldsymbol{v_1} = \dfrac{\boldsymbol{u_1}}{\|u_1\|}$
2. $i = 2, 3, \cdots, N$に対し以下の操作を繰り返す
   A. $\boldsymbol{\hat{u}_i} = \displaystyle\sum_{j=1}^{i-1} \langle \boldsymbol{u_i}, \boldsymbol{v_j} \rangle \boldsymbol{v_j}$
   B. $\boldsymbol{v_i} = \dfrac{\boldsymbol{u_i} - \boldsymbol{\hat{u}_i}}{\|\boldsymbol{u_i} - \boldsymbol{\hat{u}_i}\|}$

実装課題 5

基底をグラム・シュミットの正規直交化する関数を実装

関数の動作

1. 正規直交化したい基底を並べた行列を入力
2. 手順（1）のとおりに１本目の基底を計算
3. $i = 2, 3, \cdots, N$に対し for 文を回すことで手順（2）の A と B を計算し残りの基底を計算

In [ ]:

```
def Gram_Schmidt_orthonormalization(matrix):
    # numpy基底行列matrix_u
    matrix_u = np.array(matrix)
    u_len = len(matrix_u)

    # 第一手順
    u0 = np.array(matrix_u[:, 0]).reshape(-1, 1)  # 一つ目の基底を縦ベクトルで取得
    v0 = u0/(n_norm(u0))
    matrix_v = np.array(v0).reshape(-1, 1)

    # 第二手順
    for i in range(1, u_len):
        ui = np.array(matrix_u[:, i]).reshape(-1, 1)
        u_ = np.zeros((u_len, 1))
        for j in range(i):
            vj = np.array(matrix_v[:, j]).reshape(-1, 1)
            u_ = u_ + euclidean_inner_product(ui, vj) * vj
        vi = (ui - u_) / (n_norm((ui - u_)))
        matrix_v = np.hstack((matrix_v, vi))
    return matrix_v
```

In [ ]:

```
u = np.array([[1,1,1],[1,-1,2],[-1,1,3]]).T
print(Gram_Schmidt_orthonormalization(u))
#
```

正射影

N次元空間 V に，ある部分空間 W を定義したとき，任意の $\boldsymbol{x} \in V$ に対して
$\langle \boldsymbol{y}, \boldsymbol{x} - \boldsymbol{y} \rangle = 0$
となるような $\boldsymbol{y} \in W$ を，xのWへの**正射影（直交射影）**という.

V の基底 $\{\boldsymbol{u_i}\}_{i=1}^N$ が正規直交性を満たすとき，部分空間 W に対する $\boldsymbol{x}$ の正射影を特に $\boldsymbol{P}_w\boldsymbol{x}$ と表記し，以下のように表すことができる．

$$\boldsymbol{P}_w\boldsymbol{x} = \sum_{i=1}^{r} \langle \boldsymbol{x}, \boldsymbol{u_i} \rangle \boldsymbol{u_i} = \left(\sum_{i=1}^{r} \boldsymbol{u_i}\boldsymbol{u_i}^H \right) \boldsymbol{x}$$

実装課題 6

正射影を求める関数を実装

関数の動作

1. 基底を並べた行列と射影したいベクトルを入力する.
2. 1つ目の基底ベクトルと射影したいベクトルの内積を計算し，1つ目の基底ベクトルをその値でスカラ倍する.
3. 2つ目の基底ベクトルについても同様に行い，1つ目の結果とのベクトル和をとる.
4. それ以降も同様にして，基底ベクトルを内積でスカラ倍したものの総和をとる.
5. ベクトルが総和されたものを出力する.

In [ ]:

```python
def orthogonal_projection(matrix, vector):
    x = np.array(vector).reshape(-1, 1)
    matrix_u = np.array(matrix)
    u_len = len(matrix_u)
    pwx = np.zeros((u_len, 1))
    for i in range(u_len):
        ui = np.array(matrix_u[:, i]).reshape(-1, 1)
        pwx = pwx + euclidean_inner_product(x, ui) * ui
    return pwx
```

In [ ]:

```python
x = [[2], [3], [-1]]
Pw = [[1, 0, 0], [0, 1, 0], [0, 0, 0]]
print(orthogonal_projection(Pw, x))
```

まとめ

- ユークリッド内積や正規直交性について理解した
- Jupyter Notebook の使い方について理解した
    - OS の差を意識することなく使えた
    - RISE でのスライドづくりは Keynote,Powerpoint に比べて手間がかかった
- 例外処理の実装が難しかった