2023.05.17

応用基礎データサイエンス・AI 第一

第5回 データサイエンス・AI 数学基礎

補足資料

データサイエンス・AI全学教育機構

線形代数と Python ライブラリ

線形代数の基本的な問題を解くために以下の Python ライブラリが利用できる。

- NumPy: numpy, **numpy.linalg** (https://numpy.org/doc/stable/reference/routines.linalg.html)
- SciPy: scipy, **scipy.linalg** (https://docs.scipy.org/doc/scipy/reference/linalg.html)

scipy.linalg は numpy.linalg が提供する関数を含みつつ、行列の LU 分解、擬似逆行列の複数解法などの拡張が行われている。
一方、 numpy.linalg のいくつかの関数では、scipy.linalg より柔軟なオプション設定ができるという面もあるので、詳細は各ライブラリのマニュアルを参照のこと。

本補足資料では線形代数における NumPy の利用例を示す。

ライブラリのインポート

In [ ]:
```
import numpy as np
from numpy import linalg as LA
```

ベクトル

ベクトルの内積

内積の定義によれば、二つの ndarray オブジェクトの乗算（要素ごとの乗算）と sum() 関数を使って値を求めることができる。

In [ ]:
```
a = np.array([1,2,3])
b = np.array([3,2,1])
print(a,b)
print(sum(a * b))
```

内積を求める関数として numpy.**inner**() が用意されている。

In [ ]:
```
print(np.inner(a,b))
```

ndarray型の一次元配列であれば numpy.**dot**() 関数でも内積を求めることができる。

In [ ]:
```
print(np.dot(a,b))
```

ndarray型のオブジェクトが持つ **ndarray.dot**() メソッドを使うこともできる。

In [ ]:
```
print(a.dot(b))
```

ベクトルのノルム

ベクトル自身の内積の平方根によりノルムを計算できる。

In [ ]:
```
c = np.array([1,2,4,6,8])
print(c)
print(np.sqrt(np.inner(c,c)))
```

ベクトルのノルムを求める関数として numpy.**linalg.norm**() がある。

In [ ]:
```
print(LA.norm(c))
```

行列

行列の転置 (transepose)

行列が ndarray 型の二次元配列で表されている場合、そのオブジェクトの '**T**' アトリビュートが転置行列を表す。

In [ ]:
```
A = np.array([[1,2,3],
              [4,5,6]])
print(A)
B = A.T
print(B)
```

ここで注意すべき点は、T アトリビュートはビュー (view) を返す（元のオブジェクトを参照する）だけで、転置された行列（ndarray 型二次元配列）の新たなオブジェクトが生成されるわけではないということである。
上の例で、元の ndarray 型オブジェクト A とその T アトリビュートを代入したオブジェクト B が同じメモリ上の実体を参照していることが numpy.**shares_memory**() により確認できる。

In [ ]:
```
print(type(A), type(B), np.shares_memory(A,B))
```

このため、どちらか一方の要素を変更すると、他方の要素も同時に変化してしまう。

In [ ]:
```
B[2,0] = -1  # B を書き換えた時点で元の A も書き換わる
print(A)
print(B)
```

新たなオブジェクトを生成したい場合は **copy**() メソッドを利用する。

In [ ]:
```
A = np.array([[1,2,3],
              [4,5,6]])
B = A.T.copy()
print(np.shares_memory(A,B)) # A,B が異なるオブジェクトであることを確認
B[2,0] = -1                  # B を書き換えても A は変化しない
print(A)
print(B)
```

行列の転置は ndarray 型オブジェクトの **transpose**() メソッドでも得られる。

In [ ]:
```
B = A.transpose()
print(np.shares_memory(A,B))
print(B)
```

この場合もビューを返すので、新たなオブジェクトを生成するには copy() メソッドを利用する。

In [ ]:
```
B = A.transpose().copy()
print(np.shares_memory(A,B))
print(B)
```

行列の転置に numpy.**transpose**() 関数を利用することもできる。
この関数も T アトリビュートや transpose() メソッドと同様にビューを返す。

In [ ]:
```
B = np.transpose(A)
print(np.shares_memory(A,B))
print(B)
B = np.transpose(A).copy()
print(np.shares_memory(A,B))
print(B)
```

ベクトルの直積（外積）

直積 (direct product) または外積 (outer product) を求めるには numpy.**outer**() を利用する。

In [ ]:
```
a = np.array(['a','b','c'],dtype=object)
b = np.array([1,2,3])
C = np.outer(a,b)
print(a)
print(b)
print(C)
```

In [ ]:
```
print(b)
print(np.outer(b,b))
```

行列の積

ndarray 型二次元配列で表された行列の積の演算には、演算子 '@' を用いる。
numpy.**matmul**() 関数を用いても同じ。

In [ ]:
```
A = np.array([[1,-1],
              [3,2]])
B = np.array([[1,0],
              [2,-1]])
print(A); print(B,'\n')
AB = A @ B
print(AB,'\n')
print(np.matmul(A,B)) # matmul() 関数による行列の積
```

二次元配列の場合には numpy.dot() 関数を用いても同じ結果が得られるが、引数オブジェクトの次元に依存して処理内容が異なることに注意して使用すること。

In [ ]:
```
print(np.dot(A,B))
```

行列とベクトルの積も演算子 '@' を用いて同じように求められる。

In [ ]:
```
A = np.array([[1,-1],
              [3,2]])
b = np.array([1,2])
print(A); print(b,'\n')
c = A @ b
print(c)
```

行列の積の演算子 '@' は複数個同時に使ってもよい。

In [ ]:
```
C = np.array([[2,1],
              [1,-1]])
print(A @ B @ C,'\n')    # 3つの行列の積
AB = A @ B
print(AB @ C)            # 2段階による演算結果と一致することを確認
```

［演習］
上記の例で用いた行列 A, B に対して、$(AB)^\top = B^\top A^\top$ となることを確認する。

単位行列と逆行列

単位行列

n 次の単位行列は numpy.**identity**() により生成できる。

In [ ]:
```
I_3 = np.identity(3)                # 3次の単位行列を生成、デフォルトで float型
print(I_3,'\n')
I_5int = np.identity(5, dtype=int) # 5次の単位行列、int型を指定
print(I_5int)
```

numpy.**eye**() によっても単位行列が生成できる。
両者の違いは、numpy.**eye**() では正方行列以外の自由な形状にできることである。

In [ ]:
```
E_3 = np.eye(3)                     # np.identity(3)と同じ
print(E_3,'\n')
E_3x4int = np.eye(3, 4, dtype=int) # 3x4の行列、int型を指定
print(E_3x4int,'\n')
E_5u1 = np.eye(5, k=1)              # 5x5の行列、'1'の要素を対角の一つ上に配置
print(E_5u1,'\n')
E_5l1 = np.eye(5, k=-1)             # 5x5の行列、'1'の要素を対角の一つ下に配置
print(E_5l1)
```

逆行列

任意の正方行列 A が正則であるための必要十分条件は、A の行列式 (determinant) を $\det(A)$ とするとき、$\det(A) \neq 0$ が成り立つことである。
そこで、与えられた正方行列が逆行列を持つかどうかを調べるには、行列式を計算すればよい。
行列式の値は numpy.linalg.**det**() により、 また正則行列の逆行列は numpy.linalg.**inv**() により求めることができる。

In [ ]:
```
A = np.array([[1,2],
              [2,-1]])

detA = LA.det(A)         # 行列式の計算
print('det(A)=', detA,'\n')
if detA != 0:            # 正則性の判定
    A_I = LA.inv(A)      # 逆行列の計算
    print(A_I,'\n')
    print(A @ A_I,'\n') # 逆行列になっていることの確認
    print(A_I @ A)
```

対角行列の作成

対角行列は numpy.**diag**() を利用して作ることができる。

In [ ]:
```
d = [1,2,4,8]
D_d = np.diag(d)
print(D_d,'\n')
print(LA.inv(D_d))
```

固有値と固有ベクトル

正則な正方行列に対して関数 numpy.linalg.**eig**() は固有値と固有ベクトルを返す。

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

v, H = LA.eig(A)  # 固有値と固有ベクトルの計算
print(v,type(v))  # 固有値の表示
print(H,type(H))  # 固有ベクトルの表示
```

行列の対角化

前例で示した正則な実対称行列の固有ベクトルを用いて元の行列を対角化してみる。

In [ ]:
```
A_diag = H.T @ A @ H
print(A_diag)
```

行列の固有値分解

同様に、前例で示した正則な実対称行列を固有値と固有ベクトルを用いて表してみる。

In [ ]:
```
A_dcmp = H @ np.diag(v) @ H.T
print(A_dcmp,'\n')
print(A)
```

ベクトルの描画

matplotlib の plt.axes.Axes.**quiver**() 関数を利用するとベクトルを可視化することができる。
以下は、上記の固有値分解で用いた行列を構成するベクトルとその固有ベクトルを描画した例である。

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

# ベクトルの始点と終点の座標を与えてベクトルをプロットする関数
def vector_draw(ax, x, y, name, lc='black', sinf=True):
    ax.quiver(0,0,x,y,color=lc,angles ='xy',scale_units='xy',scale=1)
    ax.text(x,y,name, color=lc)
    if sinf: # 座標軸への射影を表す線をプロット
        ax.vlines(x,0,y,lw=1,linestyle='dotted')
        ax.hlines(y,0,x,lw=1,linestyle='dotted')

A = np.array([[2,1],
              [1,2]])
_, H = LA.eig(A)
x_1 = A[:,0]; x_2 = A[:,1]              # 対称行列 A を構成するベクトル x_1,x_2
h_1 = H[:,0]; h_2 = H[:,1]              # 固有ベクトル h_1,h_2

fig, ax = plt.subplots(figsize=(3,3))
                                        # 描画範囲,目盛りの設定
ax.set_xlim(-1,2.5);  ax.set_ylim(-1,2.5)
ax.set_xticks([1,2]); ax.set_yticks([1,2])
                                        # 座標軸の描画設定
ax.spines[['left','bottom']].set_position('zero')
ax.spines[['top','right']].set_visible(False)
ax.text(-0.3,-0.3,'0')                  # 座標原点の表示
# x_1,x_2 の描画
vector_draw(ax,x_1[0],x_1[1],'$x_1$')
vector_draw(ax,x_2[0],x_2[1],'$x_2$')
 # h_1,h_2 の描画
vector_draw(ax,h_1[0],h_1[1],'$h_1$',lc='red',sinf=False)
vector_draw(ax,h_2[0],h_2[1],'$h_2$',lc='red',sinf=False)

plt.show()
```