**データ工学特論**

**レポート３**

　　　　　　　チョウ　インチェン

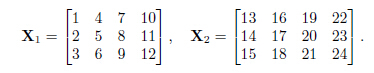
名前：趙　 寅琛

学籍番号：　14890521

**１．誤り訂正**

(1)．前回のHOSVDアルゴリズムの実験では、プログラミングに誤りがあるため、問題に対する解答は正しくなかった。ここでやり直す

問題：HOSVDアルゴリズムとHOOIアルゴリズムをそれぞれ使って、以下の**χ**∈R3\*4\*2テンソルを分解せよ



(2)．コード

コーディングの環境はPython 2.7

①　クラスを定義したファイルclass.py、

ページ数を短縮するのため関数名とその功能のみ示されている

from collections import defaultdict

class Matrix(defaultdict):

M = 0 #横

N = 0 #縦

def \_\_init\_\_(self,m,n,element\_list=[]):

'''初期化の方法 M=Matrix(横数,縦数)'''

'''あるいは 1 2 を初期化する場合を例として　M=Matrix(2,2,[[1,2],[3,4]])'''

''' 3 4'''

def mprint(self):

'''行列を　1 0 のような形式で出力'''

''' 0 1'''

def Transpose(self):

'''行列の転置を求める'''

class Tensor(defaultdict):

I, J, K= 0, 0, 0

def \_\_init\_\_(self,Matrix\_list=[]):

'''初期化の方法 T=Tensor(各スライスからなる行列のリスト)'''

self.I = Matrix\_list[0].M

self.J = Matrix\_list[0].N

self.K = len(Matrix\_list)

for i in range(1,self.K+1):

self[i] = Matrix\_list[i-1]

def tprint(self):

'''テンソルを各スライドで表示する'''

②　テンソル分解において使われる関数を定義したファイルfunctions.py、

ページ数を短縮するのため関数名とその功能のみ示されている

from classes import \*

from numpy import \*

def InnerProduct(M1,M2):

'''行列またはベクトルの内積を計算'''

def Norm(M):

'''行列またはベクトルのノルムを内積を用いて計算'''

def MatrixProduct(M1,M2):

'''行列の積を計算'''

def PInverse(M):

'''numpy.linalg.pinvを用いてMP一般逆行列を計算'''

def Unfold(T,n):

'''テンソルTをnモデルで展開''

def nModeProduct(U,T,n):

'''行列UとテンソルTのmodel-n積を計算'''

def mconvert1(M):

'''本プログラムにより定義された行列→numpyにより定義された行列'''

def mconvert2(M):

'''numpyにより定義された行列→本プログラムにより定義された行列'''

def left\_singular\_vectors(M):

'''固有ベクトルを計算'''

③　HOSVDを実行するプログラムのファイルHOSVD.py

from classes import \*

from functions import \*

def HOSVD(T):

'''HOSVDアルゴリズム'''

#A(n)を計算

X = {1:Unfold(T,1), 2:Unfold(T,2), 3:Unfold(T,3)}

A = {} #A = {1:A1, 2:A2, 3:A3}

for i in range(1,4):

A[i] = left\_singular\_vectors(X[i]) #固有ベクトルよりA(i)を計算

print "結果："

for i in range(1,4):

print "A(%d):"%(i)

A[i].mprint()

T = nModeProduct(T,A[i].Transpose(),i)

print

T.tprint()

#Y = X ×1A(1) ×2A(2) ×3A(3)より結果を検証

print "検証："

for i in range(1,4):

print "A(%d):"%(i)

A[i].mprint()

T = nModeProduct(T,A[i],i)

print

T.tprint()

if \_\_name\_\_ == "\_\_main\_\_":

X1 = Matrix(3,4,[[1,4,7,10],[2,5,8,11],[3,6,9,12]])

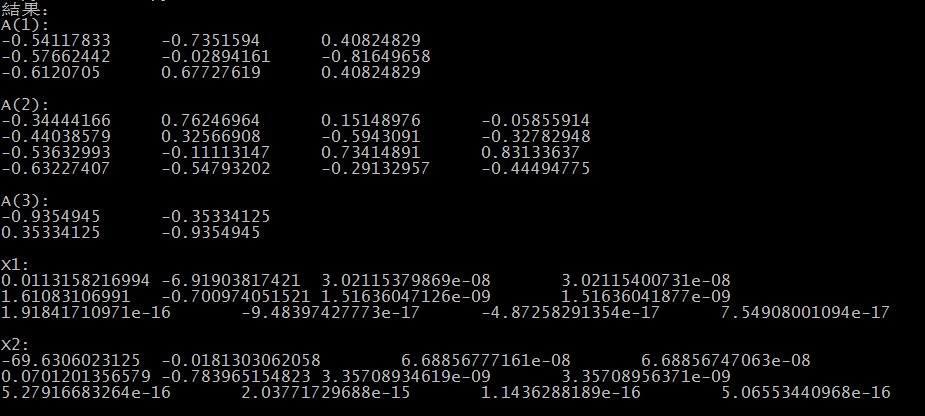
X2 = Matrix(3,4,[[13,16,19,22],[14,17,20,23],[15,18,21,24]])

T = Tensor([X1,X2])

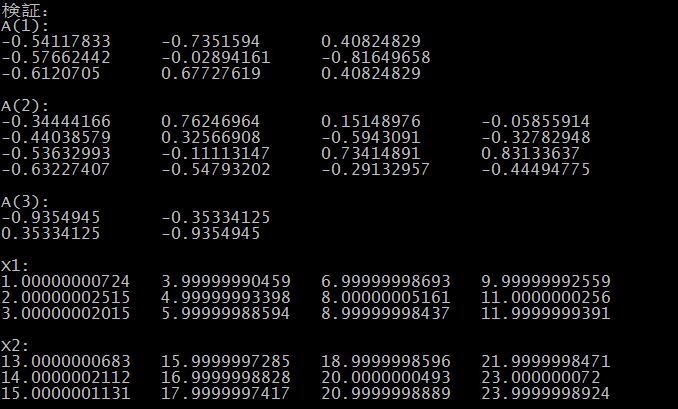
HOSVD(T)

(3)．結果

分解結果：



検証結果：



検証結果はもとテンソルと合っている。

**２．HOSVDアルゴリズムを用いて顔表情の分解**

(1) 計算量を減らすために、授業で配られた画像データを使わず、以下のように顔表情の分解の問題を以下のように設定し、簡単化する：

①　被験者はA、B二人だけとし、表情の種類はα、βだけとする

②　表情フィーチャは二次元ベクトル(x1,x2)とする

x1によって「誰の顔のか」を決定する。具体的に、Aの顔ならx1の値は０に近い、Bの顔ならx1の値は1に近い。x2によって「どの表情なのか」を決定する。具体的に、αの表情ならx2の値は０に近い、βの表情ならx2の値は1に近い。

ただし、これらのルールはシステムに入力しないこと

　③　(1)と(2)の設定により、学習データは以下のようとする

|  |  |  |
| --- | --- | --- |
|  | α | β |
| A | **(0.10,0.18)** | **(0.21,0.89)** |
| B | **(0.91,0.08)** | **(1.02,0.91)** |

したがって、入力テンソルは

X1 = [0.10 0.21 X2 = [0.18 0.89

0.91 1.02] 0.08 0.91 ]

④　問題１（顔認識）：ある顔画像のフィーチャベクトルは (0.18,0.79)、かつその顔画像の表情の種類はβだと分かったとすれば、それが誰の顔画像なのかを検出せよ

問題２（表情認識）：ある顔画像のフィーチャベクトルは (1.03,0.81)、かつその顔画像はBの顔だと分かったとすれば、それがどの表情なのかを検出せよ

(2)．コード

コーディングの環境はPython 2.7

import sys, math

sys.path.append("./")

from classes, functions, HOSVD import \*

def cos\_dist(a,b):

#cos類似度を計算

return InnerProduct(a,b)/math.sqrt(Norm(a)\*Norm(b))

def Expression\_i(Te,n):

# 特定した表情テンソル*T*expression(i)を抽出

temp\_list = []

for k in range(1,Te.K+1):

temp\_M = Matrix(Te.I,1)

for i in range(1,Te.I+1):

temp\_M[i][n] = T[k][i][n]

temp\_list.append(temp\_M)

return Tensor(temp\_list)

def Face\_Recognition(F, T, n):

'''顔認識 引数は学習データのテンソルT、入力画像の素性ベクトルF、既知表情種類n'''

# 表情テンソル*T*expression

temp = HOSVD(T)

S = temp[0]

U = {'p':temp[1], 'e':temp[2], 'f':temp[3]}

temp = nModeProduct(S,U['e'],2)

Te = nModeProduct(temp,U['f'],3)

# パーソンベクトル*u*p

temp = PInverse(Unfold(Expression\_i(Te,n),1))

up = MatrixProduct(F,temp)

# cos類似度によってパーソンを推定

Uperson = {}

temp\_max, temp\_i = -2, 0

for i in range(1,U['p'].N+1):

Uperson[i] = Matrix(1,up.N)

Uperson[i][1] = U['p'][i]

dist = cos\_dist(Uperson[i],up)

print dist

if dist > temp\_max:

temp\_max = dist

temp\_i = i

print temp\_i

def Face\_i(Te,n):

# 特定したパーソンテンソル*T*person(i)を抽出

temp\_list = []

for k in range(1,Te.K+1):

temp\_M = Matrix(1,Te.J)

for j in range(1,Te.J+1):

temp\_M[n][j] = T[k][n][j]

temp\_list.append(temp\_M)

return Tensor(temp\_list)

def Expression\_Recognition(E,T,n=1):

'''表情認識 引数は学習データのテンソルT、入力画像の素性ベクトルE、既知パーソン'''

# パーソンテンソル*T*person

temp = HOSVD(T)

S = temp[0]

U = {'p':temp[1], 'e':temp[2], 'f':temp[3]}

temp = nModeProduct(S,U['p'],1)

Tp = nModeProduct(temp,U['f'],3)

# 表情ベクトル*u*e

temp = PInverse(Unfold(Face\_i(Tp,n),2))

ue = MatrixProduct(E,temp)

# cos類似度によって表情を推定

Uperson = {}

temp\_max, temp\_i = -2, 0

for i in range(1,U['e'].M+1):

Uperson[i] = Matrix(1,ue.N)

Uperson[i][1] = U['e'][i]

dist = cos\_dist(Uperson[i],ue)

print dist

if dist > temp\_max:

temp\_max = dist

temp\_i = i

print temp\_i

if \_\_name\_\_ == "\_\_main\_\_":

#入力データ

X1 = Matrix(2,2,[[0.1,0.21],[0.91,1.02]])

X2 = Matrix(2,2,[[0.18,0.89],[0.08,0.91]])

T = Tensor([X1,X2])

F = Matrix(1,2,[[0.18,0.79]])

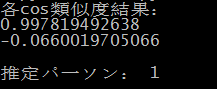
E = Matrix(1,2,[[0.56,0.79]])

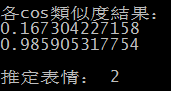
Face\_Recognition(F,T,2)

print

Expression\_Recognition(E,T,1)

(3)．結果

問題１に対する顔認識の結果：　Aの顔だと推定した

問題２に対する表情認識の結果：　βの表情だと推定した

(1)で述べた設定を参照すれば、結果の正しいことが分かった。

**３　まとめ**

本実験では、（Wang,Ahuja 2003）が提案した顔表情の分解手法に従って、顔認識および表情認識を行った。大規模データなら実験を行う端末が対応できない恐れがあるため、問題の本質が変わらない上で認識問題を適切に簡単化し再設定した。実験結果も予想通りに正しかった。

　しかし、こういう設定ですべての入力データに対して出力が正しいとは限らない。例えば、(0.5,0.5)のような中間値を入力したり、正しくない顔または表情情報を入力したりすれば、こういう手法は正しく検出できないと思われる。