COMPUTATIONAL METHODS FOR DATA SCIENCE

HOMEWORK 1

1. Basic Probability in Blackjack *(15 points)*
2. Given a single deck, calculate the probability that you get a blackjack.
3. What is the probability that you will bust if you take another card?
4. Explain when you will take a card if the face-up cards appear.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 莊家亮牌6 | 莊家亮牌6 | 莊家亮牌6 |
| 莊家是否要牌 | 7,8,9,10, J, Q, K, A | 6 | 2,3,4,5 |
| 要牌 (莊家總點數17) | 可能底牌  5,6,7,8,9  可能總點數範圍  11-16 | 可能底牌  2, 3,,.K  可能點數範圍  8-16 | 可能底牌  A,2, 3,,.K  可能點數範圍  3-16 |
| 不要牌 (莊家總點數17) | 可能底牌  10,J,Q,K  **可能總點數範圍**  **17-20** | 可能底牌  A  可能點數範圍  17 | 沒有可能底牌 |

要牌策略：當莊家亮出的牌大於6，而且莊家選擇不繼續要牌，而延續b小題的前提，我手上的牌有13點，我會選擇繼續要牌。

策略說明：如果莊家亮出的牌大於6，而且莊家選擇不繼續要牌，則莊家手上的總點數很有可能介於17-20點，此時我手上總點數只有13點，若不繼續要牌，在最終計算點數時，很有可能會輸給莊家，因為距離21點的距離，有很大的機會會比莊家的牌來得大。

1. Basic Statistics in Roulette *(10 points)*
2. Expectations and variances

* 每次賭本很低，但玩很多次的玩法

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Payout | Bet Amount | Expectation (A) | Variance (B) | Real Bet (D) | Times (C) | (A)\*(CD) | (B)\*(CD)^2 |
| Red or Black | 1-to-1 | A multiple of 8 | -0.053 | 0.997 | 8 | 12 | -5.088 | 9188.35 |
| Odd or Even | 1-to-1 | A multiple of 8 | -0.053 | 0.997 | 8 | 12 | -5.088 | 9188.35 |
| 1 to 18 or 19 to 36 | 1-to-1 | A multiple of 8 | -0.053 | 0.997 | 8 | 12 | -5.088 | 9188.35 |
| Dozen (1 to 12, …) | 2-to-1 | A multiple of 4 | -0.053 | 1.945 | 4 | 25 | -5.30 | 19450 |
| Column (on the right) | 2-to-1 | A multiple of 4 | -0.053 | 1.945 | 4 | 25 | -5.30 | 19450 |
| Single Number | 35-to-1 | A multiple of 1 | -0.053 | 33.21 | 1 | 100 | -5.30 | 332100 |

* 一次賭本比較高，玩最少次的玩法

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Payout | Bet Amount | Expectation (A) | Variance (B) | Real Bet (D) | Times (C) | (A)\*(CD) | (B)\*(CD)^2 |
| Red or Black | 1-to-1 | A multiple of 8 | -0.053 | 0.997 | 96 | 1 | -5.088 | 9188.35 |
| Odd or Even | 1-to-1 | A multiple of 8 | -0.053 | 0.997 | 96 | 1 | -5.088 | 9188.35 |
| 1 to 18 or 19 to 36 | 1-to-1 | A multiple of 8 | -0.053 | 0.997 | 96 | 1 | -5.088 | 9188.35 |
| Dozen (1 to 12, …) | 2-to-1 | A multiple of 4 | -0.053 | 1.945 | 100 | 1 | -5.30 | 19450 |
| Column (on the right) | 2-to-1 | A multiple of 4 | -0.053 | 1.945 | 100 | 1 | -5.30 | 19450 |
| Single Number | 35-to-1 | A multiple of 1 | -0.053 | 33.21 | 100 | 1 | -5.30 | 332100 |

在思考賭場測略時，基本上有兩種思考方向，每種玩法的期望值都是一樣的-0.053，但每種玩法的變異數會有不同，那相對應的每種玩法每次要下的最低賭本會有不一樣。第一種可以思考的玩法可以是每次玩的時候，都下最低賭本。而另一種玩法則是希望可以玩越少次越好，然後每次下的賭本就出到最大。兩種邏輯可以列出上面的兩張表。後來仔細一想，兩種玩法算出來的期望值及變異數，也會一樣，但在玩法上有些細微的不同。

1. 期望值最大化的策略：當我們整理出上面的表格後，可以看出其實賭場內不同玩法間的期望值會是一樣的，都是-0.053元，這個時候就要考慮當我們手上的賭本只有100元時，各種玩法可以進行的次數及每次可以下的賭本。現在假設前提是手上的100元需要全部用盡的情境下。在追求期望值最大化的策略中，基本上可以從Red or Black，Odd or Even，1 to 18 or 19 to 36這三種玩法中任選一種玩12次，每次都下最小賭本8元，或是乾脆一次下96元，就玩一次，兩種玩法的期望值基本上是一樣的，都是-5.088元，這時候還剩下4元，可以考慮再從Dozen或Column兩種玩法中，任選一種來玩，把剩下的4元賭本出完，最後的期望值算出來是-5.088-0.053\*4=-5.3。發現其實跟單純一次Single Number然後，一次就壓100元的期望值是一樣的。

並且其實當我們採用其他的玩法，例如Dozen或Column，不管每次壓最小賭本4元玩25次，或是一次壓100元論輸贏，期望值都是-5.30元。

因次在追求期望值最大的目標上，不管選哪種玩法，其實期望值都一樣。

1. 變異數最小化的策略：雖然各種玩法經過上面的試算後，發現基本上都一樣。但是各種玩法間的變異數卻都有差異。變異數最小的玩法依然有三種，分別是Red or Black，Odd or Even，1 to 18 or 19 to 36，不管是採用每次都最小賭本然後玩很多次，或是採用最高賭本，然後只玩一次的玩法，三種玩法在花了96元後，變異數都是9188.35，還有剩下4元，可以考慮玩一次賭本4元的玩法，即Dozen或Column，玩一次的變異數為1.945\*4^2，這樣就花了100元，變異數合計為9188.35+1.945\*4^2=9219.472，可以獲得最低的變異數。
2. 程式模擬結果
3. 期望值最大策略：

第一把賭 Red or Black，賭Red，押96元

第二把賭 Dozen，賭1-12，押4元

|  |  |
| --- | --- |
| 程式 | import numpy as np  import random  def spins():  slots = {'00': 'green', '0': 'green', '1': 'red', '2': 'black',  '3': 'red', '4': 'black', '5': 'red', '6': 'black',  '7': 'red','8': 'black', '9': 'red', '10': 'black',  '11': 'red',  '12': 'black', '13': 'red', '14': 'black', '15': 'red',  '16': 'black', '17': 'red', '18': 'black', '19': 'red',  '20': 'black', '21': 'red', '22': 'black', '23': 'red',  '24': 'black', '25': 'red', '26': 'black', '27': 'red',  '28': 'black', '29': 'red', '30': 'black', '31': 'red',  '32': 'black', '33': 'red', '34': 'black', '35': 'red',  '36': 'black'}  result = random.choice(list(slots.keys()))  return result  result = {}  for i in range(2):  result[i] = spins()  result |
| 程式模擬結果 | {0: '18', 1: '34'}  第一把結果是黑色，不是紅色，所以96元賠掉。  第二把結果是34，沒有落在1-12之間，所以4元也賠掉。 |
| Average Winning per Bet | -100元/2把 = -50元/把 |

1. 變異數最小策略：

前12把賭 Red or Black，賭Black，每把押8元

第13把賭 Dozen，賭13-24，押4元

|  |  |
| --- | --- |
| 程式 | import numpy as np  import random  def spins():  slots = {'00': 'green', '0': 'green', '1': 'red', '2': 'black',  '3': 'red', '4': 'black', '5': 'red', '6': 'black',  '7': 'red',  '8': 'black', '9': 'red', '10': 'black', '11': 'red',  '12': 'black', '13': 'red', '14': 'black', '15': 'red',  '16': 'black', '17': 'red', '18': 'black', '19': 'red',  '20': 'black', '21': 'red', '22': 'black', '23': 'red',  '24': 'black', '25': 'red', '26': 'black', '27': 'red',  '28': 'black', '29': 'red', '30': 'black', '31': 'red',  '32': 'black', '33': 'red', '34': 'black', '35': 'red',  '36': 'black'}  result = random.choice(list(slots.keys()))  return result  result = {}  for i in range(13):  result[i] = spins()  result |
| 程式模擬結果 | {0: '22', 1: '12', 2: '15', 3: '15', 4: '3', 5: '3', 6: '36', 7: '23', 8: '10', 9: '30', 10: '13', 11: '17', 12: '4'}  前12把中，有5把為Black，7把為Red，前12把結果為-16元。  第13把結果為4，沒有落在13-24中，再賠掉4元。 |
| Average Winning per Bet | -20元/13把＝-1.538元/把 |

1. 真實生活中，Steve若採用不斷加碼的策略，首先會面對的問題就是每個人在賭場內的成本通常都是有限的。如果Steve身上剛好就只有帶100元，這種不斷加碼的策略，當他連續輸掉三把：-8-16-32=-56，當下他的口袋其實只剩下44元，已經無法再加碼到64元了。當發生連續輸錢好幾次的情形時，基本上就無法採用題目中Steve原先預想的策略，而且現實中當人的運氣不好時，的確會有機會連續輸很多局。
2. Monthly Record-Breaking Temperature in California I: Matrix Calculation *(25 points)*
3. Run LU factorization on X to obtain matrix L and U.

|  |
| --- |
| import pandas as pd  data = pd.read\_csv('/content/drive/MyDrive/臺大資料科學博士班/資料科學計算/作業/CAmaxTemp.txt', sep = ' ').reset\_index(drop=False)  data['Station'] = data['index'].map(lambda x:x.split('\t')[0])  data['Period'] = data['index'].map(lambda x:x.split('\t')[1])  data.drop('index', axis=1, inplace=True)  data.columns = ['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC','MAX','Station','Period']  data = data[['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']]  def LU\_decomposition(a):  dim = a.shape #先知道送進來矩陣的維度  lower = np.identity(dim[0]) #產生Identity Matrix 當作下三角的初始  upper = np.zeros(dim) #產生Identity Matrix 當作上三角的初始  upper[0, :] = a[0, :]  lower[:, 0] = a[:, 0]/upper[0,0]  for i in range(1, dim[0]):  for j in range(1, dim[0]):  Z = sum(lower[i, k]\*upper[k, j] for k in range(i))  if i <= j:  upper[i,j] = a[i,j] - Z  elif i > j:  lower[i,j] = (a[i,j] - Z)/upper[j,j]  return lower, upper  L,U = LU\_decomposition(data.to\_numpy())  print(L)  print(U) |
| Lower triangular matrix L 結果如下 |
|  |
| Upper triangular matrix U 結果如下 |
|  |

1. Use Gram-Schmidt algorithm on X to obtain Q and R. Find the inverse of X.

|  |
| --- |
| # 3.b QR factorization using Gram-Schmidt  import pandas as pd  data = pd.read\_csv('/content/drive/MyDrive/臺大資料科學博士班/資料科學計算/作業/CAmaxTemp.txt', sep = ' ').reset\_index(drop=False)  data['Station'] = data['index'].map(lambda x:x.split('\t')[0])  data['Period'] = data['index'].map(lambda x:x.split('\t')[1])  data.drop('index', axis=1, inplace=True)  data.columns = ['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC','MAX','Station','Period']  data = data[['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']]  def QR\_factorization(a):  dim = a.shape  Q = np.zeros(dim)  R = np.zeros((dim[1], dim[1]))  R[0,0] = np.sqrt(sum(x\*\*2 for x in a[:, 0].reshape(-1,1)))  Q[:, 0] = a[:, 0]/R[0,0]  for k in range(1, dim[1]):  R[0:k, k] = [sum(Q[:, i]\*a[:,k]) for i in range(k)]  q = a[:, k] -sum(R[i,k]\*Q[:, i] for i in range(k))  R[k,k] = np.sqrt(sum([x\*\*2 for x in q.reshape(-1,1)]))  Q[:, k] = q/R[k,k]    return Q, R  Q, R = QR\_factorization(data.to\_numpy())  print(Q)  print(R) |
| Q矩陣的執行結果如下 |
|  |
| R矩陣的執行結果如下 |
|  |
| 有了Q,R矩陣後，就可以運用 來得到X的反矩陣 |
|  |

1. Use power iteration method to find the largest eigenvalue-eigenvector pair of X.

|  |
| --- |
| # 3.c Power Iteration  def power\_iteration(A, num\_simulations: int):  # Ideally choose a random vector  # To decrease the chance that our vector  # Is orthogonal to the eigenvector  b\_k = np.random.rand(A.shape[1])  for \_ in range(num\_simulations):  # calculate the matrix-by-vector product Ab  b\_k1 = np.dot(A, b\_k)  # calculate the norm  b\_k1\_norm = np.linalg.norm(b\_k1)  # re normalize the vector  b\_k = b\_k1 / b\_k1\_norm    eig\_val = np.dot(np.dot(b\_k.T,A),b\_k)/np.dot(b\_k.T, b\_k)  eig\_vec = b\_k.reshape(A.shape[1],1)  return eig\_val, eig\_vec  eig, eig\_vec = power\_iteration(data, 1000)  print(eig)  print(eig\_vec) |
| 可以找到最大的eigenvalue為1167.7759013704458，而特徵向量的結果如下方截圖所示 |
|  |

1. Use QR factorization to find all eigenvectors with REAL eigenvalues of X.

|  |
| --- |
| 本題預設迭代的門檻值為0.0001，迭代次數上限為1000次。怕會不收斂或是太早停止迭代，會將門檻值調成0.00001，迭代次數上限調成3000次做嘗試。程式碼如下： |
| # 3.d Use QR factorization to find all eigenvectors with REAL eigenvalues of X  def QR\_iteration(a, precision=0.0001, iter\_ceil=1000):  dim = a.shape  real\_value = []  for i in range(dim[0]):  times = 1  a\_0 = a.copy()  Q\_eigen = np.identity(dim[0]) # identity matrix    while True:  Q, R = QR\_factorization(a\_0)  Q\_eigen = np.dot(Q\_eigen, Q) # converges to eigenvector  ak = np.dot(R,Q) # converges to eigenvalue  if np.abs(ak[i][i] - a\_0[i][i]) < precision:  print(f"Eigenvalue number {i+1} is found. Iteration:", times)  real\_value.append(ak[i][i])  break  elif times > iter\_ceil:  print(f"Eigenvalue number {i+1} is unable to converge.")  break  else:  times = times + 1  a\_0 = ak    return Q\_eigen, real\_value  Q\_eigen, eig\_real = QR\_iteration(data.to\_numpy(), precision=0.00001, iter\_ceil=3000) |
| 程式執行的結果如下方所示，可以順利找到題目中說的6個實數eigenvalue。其他6個就無法順利收斂。 |
|  |
| 6個實數的eigenvalue如下所示： |
|  |

1. Monthly Record-Breaking Temperature in California II: PCA and SVD *(30 points)*
2. Standardize the data and compute the variance-covariance matrix

|  |
| --- |
| #%% 4. PCA and SVD  #read txt file as pandas dataframe  import pandas as pd  data = pd.read\_csv('/content/drive/MyDrive/臺大資料科學博士班/資料科學計算/作業/CAmaxTemp.txt', sep = ' ').reset\_index(drop=False)  data['Station'] = data['index'].map(lambda x:x.split('\t')[0])  data['Period'] = data['index'].map(lambda x:x.split('\t')[1])  data.drop('index', axis=1, inplace=True)  data.columns = ['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC','MAX','Station','Period']  data = data[['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']]  # Standardize the data  df\_standardized = (data - data.mean())/data.std()  # Compute the variance-covariance matrix  df\_cov1 = df\_standardized.cov() |
| 下方為標準化過後的資料表 |
|  |
| 下方為12個月份的共變異數矩陣 (Variance-Covariance Matrix) |
|  |

1. Find the top three principal components using power iteration. Calculate the cumulative percentage of the total eigenvalues that these three principal components cover.

|  |
| --- |
| def power\_iteration(A, num\_simulations: int):  b\_k = np.random.rand(A.shape[1])  for \_ in range(num\_simulations):  # calculate the matrix-by-vector product Ab  b\_k1 = np.dot(A, b\_k)  # calculate the norm  b\_k1\_norm = np.linalg.norm(b\_k1)  # re normalize the vector  b\_k = b\_k1 / b\_k1\_norm  eig\_val = np.dot(np.dot(b\_k.T,A),b\_k)/np.dot(b\_k.T, b\_k)  eig\_vec = b\_k.reshape(A.shape[1],1)  return eig\_val, eig\_vec  lambda1, eig\_v1 = power\_iteration(df\_cov1, 10000)  cov={}  lam={}  eig\_v = {}  cov[1] = df\_cov1  lam[1] = lambda1  eig\_v[1] = eig\_v1  for i in range(1,12):  cov[i+1] = cov[i] - lam[i]\*np.dot(eig\_v[i], eig\_v[i].T)  lam[i+1], eig\_v[i+1] = power\_iteration(cov[i+1], 10000)  top3\_cover =(lam[1]+lam[2]+lam[3])/(lam[1]+lam[2]+lam[3]+lam[4]+lam[5]+lam[6]+lam[7]+lam[8]+lam[9]+lam[10]+lam[11]+lam[12])  print(top3\_cover) |
| 前三大的eigenvalues在所有eigenvalues的佔比為95.68%，程式執行結果如下方截圖。 |
|  |

1. Plot the data on a 3D space with three principal component axes. Provide the coordinates of the recast data.

|  |
| --- |
| #plot the data on a 3D space with three PC axes  data\_centralized = data - data.mean()  eigv\_top3 = np.concatenate((eig\_v[1],eig\_v[2],eig\_v[3]), axis = 1)  eigv\_proj = np.dot(eigv\_top3.T, data\_centralized).T  fig = plt.figure(figsize = (10,10))  ax = plt.axes(projection='3d')  ax.grid()  # ax.set\_xlim([-4,1])  # ax.set\_ylim([-1,9])  # ax.set\_zlim([-1,14])  x = eigv\_proj[:,0]  y = eigv\_proj[:,1]  z = eigv\_proj[:,2]  ax.scatter(x, y, z, c = 'r', s = 50)  ax.set\_title('3D Scatter Plot')  # Set axes label  ax.set\_xlabel('x', labelpad=20)  ax.set\_ylabel('y', labelpad=20)  ax.set\_zlabel('z', labelpad=20)  plt.show()  12個點投影到三個特徵向量的3D散佈圖 |
|  |
| 12個點的X Y Z座標以下方矩陣表示 |
|  |

1. Find all principal components with their eigenvalues using SVD.

|  |
| --- |
| # 4.e  def SVD\_factorization(a):  m,n = a.shape  S = np.zeros([m,n])  U = np.zeros([m,m])  data = np.dot(a.T, a)  eigenvals, eigenvecs = np.linalg.eig(data)  eig\_index = eigenvals.argsort()[::-1]  eigenval\_desc = eigenvals[eig\_index]  V = eigenvecs[:, eig\_index]  #把 S 矩陣算出來  for i in range(m):  for j in range(n):  if i == j:  S[i,j] = np.sqrt(eigenval\_desc[i])    #把 U 矩陣算出來  for i in range(m):  if np.diagonal(S)[i] != 0:  u = (1/np.diagonal(S)[i])\*np.dot(a, V[:,i].reshape(n,1))  u = u.reshape(m,)  U[:,i] = u  else:  U[:,i] = np.zeros(m)    return U, S, V  U,S,V = SVD\_factorization(data)  V |
| 先以簡單的例子程式做出來SVD分解是正確的，如下圖所示，可以正確的將矩陣還原為原本的樣子。 |
|  |
| 在進行SVD分解的時候，過程中得到的V矩陣，即為所有Principal Component的向量，V的執行結果如下所示。 |
|  |

1. SVD provides an extra information on U that PCA does not usually have. Is it any interpretation of this U matrix? If yes, please state it.

給定一資料矩陣A，經SVD分解，可得 。

為 的特徵向量矩陣 ()， 為 的特徵向量矩陣 ()。若X為一 的矩陣，表示有p個feature及n筆sample，那麼V矩陣在描述的是feature與feature之間的相關性 (Covariance有相關性的意涵存在)，而U矩陣在表達的則sample與sample之間的相關性，以本題而言，V矩陣在描述的應該是月份與月份之間的關係，而U矩陣則是在描述各個地點與地點之間的關係。有些文獻會稱U矩陣為左奇異向量矩陣，V矩陣則是右奇異向量矩陣。

1. Conduct a rank-3 approximation (SVD version of X).

|  |
| --- |
| U,S,V = SVD\_factorization(data)  U\_3 = U[:,0:3]  S\_3 = S[0:3,0:3]  V\_3 = V.T[0:3,:]  A\_3 = np.dot(U\_3.dot(S\_3), V\_3) |
| Rank-3 approximation的結果如下圖所示 |
|  |
| 與跟原始的氣溫data比較，可以發現矩陣中各個的資料幾乎都可以還原的非常接近，顯示取前3個rank的rank-3 approximation效果已經非常好。 |
|  |

1. Monthly Record-Breaking Temperature in California III: ICA *(20 points)*
2. Explain why the data is unlikely to be Gaussian.

|  |
| --- |
| # 5.a  import scipy.stats as stats  from scipy.stats import kurtosis  import pandas as pd  import numpy as np  data = pd.read\_csv('/Users/wujhejia/Documents/Python/CAmaxTemp.txt', sep = ' ').reset\_index(drop=False)  data['Station'] = data['index'].map(lambda x:x.split('\t')[0])  data['Period'] = data['index'].map(lambda x:x.split('\t')[1])  data.drop('index', axis=1, inplace=True)  data.columns = ['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC','MAX','Station','Period']  data = data[['JAN','FEB','MAR','APR','MAY','JUN','JUL','AUG','SEP','OCT','NOV','DEC']]  #data = data.to\_numpy()  data\_prime = data[['FEB','JUN','OCT']]  results = {}  months = {0:'FEB',1:'JUN',2:'OCT'}  for i in range(3):  results[i] = kurtosis(data\_prime[months[i]]) |
| 運用Kurtosis是否等於0，來判斷三個月的變數是否符合常態分佈 |
|  |
| 從程式執行的結果可以發現，三個月的資料都不等於0，因此都不符合常態分佈。 |
|  |
| 上圖則是另外透過quantile-quantile plot來判斷是否接近常態，可以發現只有二月比較接近常態分佈，但六月跟十月都沒有接近常態，因為偏離紅色的標準線。 |

1. Run the three preprocessing steps of ICA on .

|  |
| --- |
| #5.b  #Preprocessing Steps - Step1:Centering  def centering(data):  output = data - data.mean()  return output  D = centering(data\_prime) |
|  |
| #Preprocessing Steps - Step2:Decorrelation  def decorrelation(D):  D = D.to\_numpy()  D\_cov = np.cov(D.T)  eigs, V = np.linalg.eigh(D\_cov)  lam = np.diag(eigs)  lam\_inv = np.sqrt(np.linalg.inv(lam))  U = np.dot(V, D.T).T  return lam\_inv, U  lam\_inv, U = decorrelation(D) |
|  |
| #Preprocessing Steps - Step3:Whitening  Z = np.dot(lam\_inv, U.T)  Z.T |
|  |

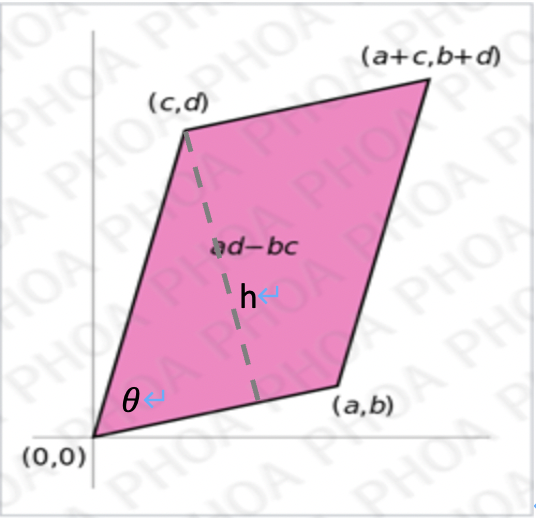
1. Provide the graphical illustration on the transformation, like the one in Lecture 04-2.

|  |  |  |  |
| --- | --- | --- | --- |
| 原始資料 Scatter Plots | | | |
| FEB vs JUN | | FEB vs OCT | JUN vs OCT |
|  | |  |  |
| 投影到PCA Space | | | |
| 1st PC vs 2nd PC | 1st PC vs 3rd PC | | 2nd PC vs 3rd PC |
|  |  | |  |
| 經過Whitening之後，其實整體資料呈現的分佈，與投影到PCA空間差不多，但整體尺度有變小。 | | | |
| 1st column vs 2nd column | 1st column vs 3rd column | | 2nd column vs 3rd column |
|  |  | |  |

1. Run the fast ICA on Kurtosis Maximization to find the three independent components.

|  |
| --- |
| def fast\_ICA1(a, precision=0.1\*\*15, iter\_ceil=100000):  m,n = a.shape  w0 = np.ones(n)  #w0 = np.random.random(n)  W = np.zeros(m)  Z = np.zeros([m,n])  times = 1  while True:  for i in range(m):  W[i] = (w0.dot(a[i,]))\*\*3  Z[i,] = a[i,]\*W[i]  w1 = np.mean(Z, axis = 0) - 3\*w0  w1 = w1/np.linalg.norm(w1)  if np.sum(np.abs(w1 - w0)) < precision:  print('w converges. Iteration times:', times)  break  elif times > iter\_ceil:  print('w cannot converge.')  break  else:  times = times + 1  w0 = w1  return w1  def fast\_ICA2(a, u1, precision=0.1\*\*15, iter\_ceil=100000):  m,n = a.shape  w0 = np.ones(n)  #w0 = np.random.random(n)  W = np.zeros(m)  Z = np.zeros([m,n])  u1 = u1  times = 1  while True:  for i in range(m):  W[i] = (w0.dot(a[i,]))\*\*3  Z[i,] = a[i,]\*W[i]  w1 = np.mean(Z, axis = 0) - 3\*w0  w1 = w1/np.linalg.norm(w1)  w1 = w1 - np.dot(w1,u1)\*u1  w1 = w1/np.linalg.norm(w1)  if np.sum(np.abs(w1 - w0)) < precision:  #if np.linalg.norm(w1 - w0) < precision:  print('w converges. Iteration times:', times)  break  elif times > iter\_ceil:  print('w cannot converge.')  break  else:  times = times + 1  w0 = w1  return w1  def fast\_ICA3(a, u1, u2, precision=0.1\*\*15, iter\_ceil=100000):  m,n = a.shape  w0 = np.ones(n)  #w0 = np.random.random(n)  W = np.zeros(m)  Z = np.zeros([m,n])  u1 = u1  u2 = u2  times = 1  while True:  for i in range(m):  W[i] = (w0.dot(a[i,]))\*\*3  Z[i,] = a[i,]\*W[i]  w1 = np.mean(Z, axis = 0) - 3\*w0  w1 = w1/np.linalg.norm(w1)  w1 = w1 - np.dot(w1,u1)\*u1 - np.dot(w1,u2)\*u2  w1 = w1/np.linalg.norm(w1)  if np.sum(np.abs(w1 - w0)) < precision or np.sum(np.abs(w1)-np.abs(w0)) < precision:  #if np.linalg.norm(w1 - w0) < precision:  print('w converges. Iteration times:', times)  break  elif times > iter\_ceil:  print('w cannot converge.')  break  else:  times = times + 1  w0 = w1  return w1 |
| 計算三組Independent Components的執行程式如下 |
|  |
| 執行結果如下 |
|  |

1. Determinant and Parallelogram *(Bonus 10 points)*



Proof.

令 , ， 及 的夾角為 ，平行四邊形的高為h

假設平行四邊形的面積為