period-10

February 22, 2024

0.1 CS/ECE/ME532 Period 10 Activity

Estimated Time:

P1: 25 mins

P2: 25 mins

0.1.1 Preambles

[1]:
```
import numpy as np # numpy
from scipy.io import loadmat # load & save data
from scipy.io import savemat
import matplotlib.pyplot as plt # plot
np.set_printoptions(formatter={'float': lambda x: "{0:0.2f}".format(x)})
```

0.1.2 Q1. *K*-means

Let $A = \begin{bmatrix} 3 & 3 & 3 & -1 & -1 & -1 \\ 1 & 1 & 1 & -3 & -3 & -3 \\ 1 & 1 & 1 & -3 & -3 & -3 \\ 3 & 3 & 3 & -1 & -1 & -1 \end{bmatrix}$. Use the provided script to help you complete the problem.

[2]:
```
A = np.
↪array([[3,3,3,-1,-1,-1],[1,1,1,-3,-3,-3],[1,1,1,-3,-3,-3],[3,3,3,-1,-1,-1]],␣
↪float)
rows, cols = A.shape
print('A = \n', A)
```

```
A =
 [[3.00 3.00 3.00 -1.00 -1.00 -1.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [3.00 3.00 3.00 -1.00 -1.00 -1.00]]
```

[3]:
```
# numpy iterates over the 0th dimension first (over the rows)

for each_entry in A:
    print(each_entry) # This prints iterates the "rows" of A
```

```
for each_entry in A.transpose():
    print(each_entry) # This prints iterates the "columns" of A
```

```
[3.00 3.00 3.00 -1.00 -1.00 -1.00]
[1.00 1.00 1.00 -3.00 -3.00 -3.00]
[1.00 1.00 1.00 -3.00 -3.00 -3.00]
[3.00 3.00 3.00 -1.00 -1.00 -1.00]
[3.00 1.00 1.00 3.00]
[3.00 1.00 1.00 3.00]
[3.00 1.00 1.00 3.00]
[-1.00 -3.00 -3.00 -1.00]
[-1.00 -3.00 -3.00 -1.00]
[-1.00 -3.00 -3.00 -1.00]
```

a) Understand the following implementation of the k-means algorithm and fill in the blank to define the distance function.

[19]:
```
def dist(x, y):
    """
    this function takes in two 1-d numpy as input an outputs
    Euclidean the distance between them
    """
    sum_square = 0
    for i in range(len(x)):
        sum_square += (x[i] - y[i])**(2)

    return sum_square ## Fill in the blank: Recall the 'distance' function used␣
 ↪in the kMeans algorithm

# print(A[0])
# print(A[1])
# dist(A[0],A[1]) # 4*6=24

def kMeans(X, K, maxIters = 20):
    """
    this implementation of k-means takes as input (i) a matrix X
    (with the data points as columns) (ii) an integer K representing the number
    of clusters, and returns (i) a matrix with the K columns representing
    the cluster centers and (ii) a list C of the assigned cluster centers
    """
    X_transpose = X.transpose()
    centroids = X_transpose[np.random.choice(X.shape[0], K)] # choose k# data␣
 ↪points randomly
    for i in range(maxIters):
        # Cluster Assignment step
        C = np.array([np.argmin([dist(x_i, y_k) for y_k in centroids]) for x_i␣
 ↪in X_transpose]) # x_i and y_k are data points;
        # np.argmin: Returns the indices of the minimum values along an axis.
        # C save index with min dist
        # Update centroids step
        for k in range(K):
            if (C == k).any(): # compare to diff k and get T/F
                centroids[k] = X_transpose[C == k].mean(axis = 0)
            else: # if there are no data points assigned to this certain␣
 ↪centroid
                centroids[k] = X_transpose[np.random.choice(len(X))]
    return centroids.transpose() , C
```

b) Use the K-means algorithm to represent the columns of A with a single cluster.

[20]:
```
# k-means with 1 cluster
centroids, C = kMeans(A, 1) ## Fill in the blank: call the "kMeans" algorithm␣
 ↪with proper input arguments
print('A = \n', A)
print('centroids = \n', centroids)
print('centroid assignment = \n', C)
```

```
A =
 [[3.00 3.00 3.00 -1.00 -1.00 -1.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [3.00 3.00 3.00 -1.00 -1.00 -1.00]]
centroids =
 [[1.00]
 [-1.00]
 [-1.00]
 [1.00]]
centroid assignment =
 [0 0 0 0 0 0]
```

c) Construct a matrix $\hat{A}_{r=1}$ whose i-th column is the centroid corresponding to the i-th column of A. Note that this can be viewed as a rank-1 approximation to A. Compare the rank-1 approximation to the original matrix and explain the nature of the approximation in terms of the properties of the K-means algorithm.

[21]:
```
# Construct rank-1 approximation using cluster
#centroids_transposed = centroids.transpose() # transpose "centroids" to␣
 ↪iterate over columns
A_hat_1 = centroids[:, C] # Fill in the blank: pick the columns of centroids␣
 ↪indexed by C and then transpose it again
print('Rank-1 Approximation, \n A_hat_1 = \n', A_hat_1)
```

```
Rank-1 Approximation,
 A_hat_1 =
 [[1.00 1.00 1.00 1.00 1.00 1.00]
 [-1.00 -1.00 -1.00 -1.00 -1.00 -1.00]
 [-1.00 -1.00 -1.00 -1.00 -1.00 -1.00]
 [1.00 1.00 1.00 1.00 1.00 1.00]]
```

d) Repeat b) and c) with $K = 2$. Compare the rank-2 approximation to the original matrix and explain the nature of the approximation in terms of the properties of the K-means algorithm. The rank-2 approximation is better than rank-1, because when k increases, meaning we have more centroids and clusters, data points could find a closer centroid, making the total distance decreases.

[22]:
```
# k-means with 2 cluster
centroids, C = kMeans(A,2) ## Fill in the blank: call the "kMeans" method with␣
↪proper input arguments
print('A = \n', A)
print('centroids = \n', centroids)
print('centroid assignment = \n', C)

#centroids_transposed = centroids.transpose() # transpose "centroids" to␣
↪iterate over columns
A_hat_2 = centroids[:, C] # Fill in the blank: pick the columns of centroids␣
↪indexed by C and then transpose it again
print('Rank-2 Approximation \n', A_hat_2)
```

```
A =
 [[3.00 3.00 3.00 -1.00 -1.00 -1.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [3.00 3.00 3.00 -1.00 -1.00 -1.00]]
centroids =
 [[-1.00 3.00]
 [-3.00 1.00]
 [-3.00 1.00]
 [-1.00 3.00]]
centroid assignment =
 [1 1 1 0 0 0]
Rank-2 Approximation
 [[3.00 3.00 3.00 -1.00 -1.00 -1.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [3.00 3.00 3.00 -1.00 -1.00 -1.00]]
```

[35]:
```
# Write code to compare A_hat_1 and A_hat_2 to the original matrix A
# compare???
print(np.linalg.norm(np.subtract(A, A_hat_1), "fro"))
print(np.linalg.norm(np.subtract(A, A_hat_2), "fro"))

print(np.subtract(A, A_hat_1))
print(np.subtract(A, A_hat_2))
```

```
9.797958971132712
0.0
[[2.00 2.00 2.00 -2.00 -2.00 -2.00]
 [2.00 2.00 2.00 -2.00 -2.00 -2.00]
 [2.00 2.00 2.00 -2.00 -2.00 -2.00]
 [2.00 2.00 2.00 -2.00 -2.00 -2.00]]
[[0.00 0.00 0.00 0.00 0.00 0.00]
 [0.00 0.00 0.00 0.00 0.00 0.00]
 [0.00 0.00 0.00 0.00 0.00 0.00]
 [0.00 0.00 0.00 0.00 0.00 0.00]]
```

0.1.3 Q2. SVD

Again let $A = \begin{bmatrix} 3 & 3 & 3 & -1 & -1 & -1 \\ 1 & 1 & 1 & -3 & -3 & -3 \\ 1 & 1 & 1 & -3 & -3 & -3 \\ 3 & 3 & 3 & -1 & -1 & -1 \end{bmatrix}$. Now consider the singular value decomposition (SVD) $A = USV^T$

a) If the full SVD is computed, find the dimensions of U, S, and V.

b) Find the dimensions of U, S, and V in the economy or skinny SVD of A.

c) The Python and NumPy command `U, s, VT = np.linalg.svd(A, full_matrices=True)` computes the singular value decomposition, $A = USV^T$ where U and V are matrices with orthonormal columns comprising the left and right singular vectors and S is a diagonal matrix of singular values.

i. Compute the SVD of A. Make sure $A = USV^T$ holds.

ii. Find U^TU and V^TV. Are the columns of U and V orthonormal? Why? *Hint:* compute U^TU.

iii. Find UU^T and VV^T. Are the rows of U and V orthonormal? Why?

iv. Find the left and right singular vectors associated with the largest singular value.

v. What is the rank of A?

[45]:
```python
# i)
# A: 4x6
U, s, VT = np.linalg.svd(A, full_matrices=True)
print('U:\n', U, '\ns:\n', s, '\nVT:\n', VT)
S_matrix = np.zeros_like(A) ## Fill in the blank: Size of S should be equal to␣
  ↪size of A ???
# np.zeros_like(X): Return an array of zeros with the same shape and type as a␣
  ↪given array.
np.fill_diagonal(S_matrix, s) ## Fill in the diagonal entries of S_matrix with␣
  ↪s ???
print(S_matrix)
print(U@S_matrix@VT)
print(A)
```

```
U:
 [[-0.50 -0.50 -0.69 0.15]
 [-0.50 0.50 -0.15 -0.69]
 [-0.50 0.50 0.15 0.69]
 [-0.50 -0.50 0.69 -0.15]]
s:
 [9.80 4.90 0.00 0.00]
VT:
 [[-0.41 -0.41 -0.41 0.41 0.41 0.41]
 [-0.41 -0.41 -0.41 -0.41 -0.41 -0.41]
 [-0.81 0.50 0.31 0.00 0.00 0.00]
 [0.11 0.65 -0.76 0.00 -0.00 -0.00]
 [-0.00 0.00 -0.00 -0.58 0.79 -0.21]
 [-0.00 0.00 -0.00 -0.58 -0.21 0.79]]
[[9.80 0.00 0.00 0.00 0.00 0.00]
 [0.00 4.90 0.00 0.00 0.00 0.00]
 [0.00 0.00 0.00 0.00 0.00 0.00]
 [0.00 0.00 0.00 0.00 0.00 0.00]]
[[3.00 3.00 3.00 -1.00 -1.00 -1.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [3.00 3.00 3.00 -1.00 -1.00 -1.00]]
[[3.00 3.00 3.00 -1.00 -1.00 -1.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [3.00 3.00 3.00 -1.00 -1.00 -1.00]]
```

[40]:
```python
# ii) Are the columns of U and V orthonormal? Why?
# Yes, because after mutiplying with itself transpose, we get identity matrix
print('UTU: \n', U.T@U) # i. Printing U^T*U
print('VTV: \n', VT@VT.T) # i. Printing V^T*V

# iii) Are the rows of U and V orthonormal? Why?
# Yes, because after mutiplying with itself transpose, we get identity matrix
print('UUT: \n', U@U.T) # i. Printing U*U^T
print('VVT: \n', VT.T@VT) # i. Printing V*V^T

V = VT.transpose()
# iv)
print('First left singular vector: \n', U[:,[0]])
print('First left singular vector: \n', V[:,[0]])
# print('First right singular vector: \n', VT[[0],:])
print('Largest singular value:', s[0])

# v) What is the rank of A? 2
# print(np.linalg.matrix_rank(A))
print(np.sum(np.abs(s)>1e-6))
```

```
# check if si bigger than 0, so the left vectors really work.
```

```
UTU:
 [[1.00 0.00 0.00 0.00]
 [0.00 1.00 0.00 -0.00]
 [0.00 0.00 1.00 0.00]
 [0.00 -0.00 0.00 1.00]]
VTV:
 [[1.00 -0.00 0.00 0.00 0.00 0.00]
 [-0.00 1.00 0.00 -0.00 -0.00 -0.00]
 [0.00 0.00 1.00 -0.00 0.00 0.00]
 [0.00 -0.00 -0.00 1.00 -0.00 -0.00]
 [0.00 -0.00 0.00 -0.00 1.00 -0.00]
 [0.00 -0.00 0.00 -0.00 -0.00 1.00]]
UUT:
 [[1.00 0.00 0.00 -0.00]
 [0.00 1.00 0.00 0.00]
 [0.00 0.00 1.00 0.00]
 [-0.00 0.00 0.00 1.00]]
VVT:
 [[1.00 -0.00 -0.00 0.00 0.00 0.00]
 [-0.00 1.00 -0.00 0.00 0.00 0.00]
 [-0.00 -0.00 1.00 -0.00 0.00 0.00]
 [0.00 0.00 -0.00 1.00 0.00 0.00]
 [0.00 0.00 0.00 0.00 1.00 -0.00]
 [0.00 0.00 0.00 0.00 -0.00 1.00]]
First left singular vector:
 [[-0.50]
 [-0.50]
 [-0.50]
 [-0.50]]
First left singular vector:
 [[-0.41]
 [-0.41]
 [-0.41]
 [0.41]
 [0.41]
 [0.41]]
Largest singular value: 9.79795897113271
2
```

d) The Python and NumPy command `U, s, VT = np.linalg.svd(A, full_matrices=False)` computes the economy or skinny singular value decomposition, $A = USV^T$ where U and V are matrices with orthonormal columns comprising the left and right singular vectors and S is a square diagonal matrix of singular values.

i. Compute the SVD of A. Make sure $A = USV^T$ holds.

ii. Find U^TU and V^TV. Are the columns of U and V orthonormal? Why? *Hint:* compute U^TU.

iii. Find UU^T and VV^T. Are the rows of U and V orthonormal? Why?

[41]:
```
# i)
U, s, VT = np.linalg.svd(A, full_matrices=False)
S_matrix = np.diag(s)
print(U@S_matrix@VT)
print(A)
```

```
[[3.00 3.00 3.00 -1.00 -1.00 -1.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [3.00 3.00 3.00 -1.00 -1.00 -1.00]]
[[3.00 3.00 3.00 -1.00 -1.00 -1.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [1.00 1.00 1.00 -3.00 -3.00 -3.00]
 [3.00 3.00 3.00 -1.00 -1.00 -1.00]]
```

[44]:
```
print('U:\n', U, '\ns:\n', s, '\nVT:\n', VT)

# ii) Are the columns of U and V orthonormal? Why?
# Yes, because after mutiplying with itself transpose, we get identity matrix
print('UTU: \n', U.T@U) # i. Printing U^T*U
print('VTV: \n', VT@VT.T) # i. Printing V^T*V

# iii) Are the rows of U and V orthonormal? Why?
# For U, yes. However, for v, no, because VT is wider than it is tall.
print('UUT: \n', U@U.T) # i. Printing U*U^T
print('VVT: \n', VT.T@VT) # i. Printing V*V^T
```

```
U:
 [[-0.50 -0.50 -0.69 0.15]
 [-0.50 0.50 -0.15 -0.69]
 [-0.50 0.50 0.15 0.69]
 [-0.50 -0.50 0.69 -0.15]]
s:
 [9.80 4.90 0.00 0.00]
VT:
 [[-0.41 -0.41 -0.41 0.41 0.41 0.41]
 [-0.41 -0.41 -0.41 -0.41 -0.41 -0.41]
 [-0.81 0.50 0.31 0.00 0.00 0.00]
 [0.11 0.65 -0.76 0.00 -0.00 -0.00]]
UTU:
 [[1.00 0.00 0.00 0.00]
 [0.00 1.00 0.00 -0.00]
 [0.00 0.00 1.00 0.00]
 [0.00 -0.00 0.00 1.00]]
VTV:
 [[1.00 -0.00 0.00 0.00]
 [-0.00 1.00 0.00 -0.00]
 [0.00 0.00 1.00 -0.00]
 [0.00 -0.00 -0.00 1.00]]
UUT:
 [[1.00 0.00 0.00 -0.00]
 [0.00 1.00 0.00 0.00]
 [0.00 0.00 1.00 0.00]
 [-0.00 0.00 0.00 1.00]]
VVT:
 [[1.00 -0.00 -0.00 0.00 0.00 0.00]
 [-0.00 1.00 -0.00 0.00 0.00 0.00]
 [-0.00 -0.00 1.00 -0.00 0.00 0.00]
 [0.00 0.00 -0.00 0.33 0.33 0.33]
 [0.00 0.00 0.00 0.33 0.33 0.33]
 [0.00 0.00 0.00 0.33 0.33 0.33]]
```

e) Compare the singular vectors and singular values of the economy and full SVD. How do they differ?

They are the same.

f) Identify an orthonormal basis for the space spanned by the columns of A.|

[46]:
```
basis_col = U[:, :2]
print(basis_col)
```

```
[[-0.50 -0.50]
 [-0.50 0.50]
 [-0.50 0.50]
 [-0.50 -0.50]]
```

g) Identify an orthonormal basis for the space spanned by the rows of A.

[47]:
```
basis_row = VT[:2, :]
print(basis_row)
```

```
[[-0.41 -0.41 -0.41 0.41 0.41 0.41]
 [-0.41 -0.41 -0.41 -0.41 -0.41 -0.41]]
```

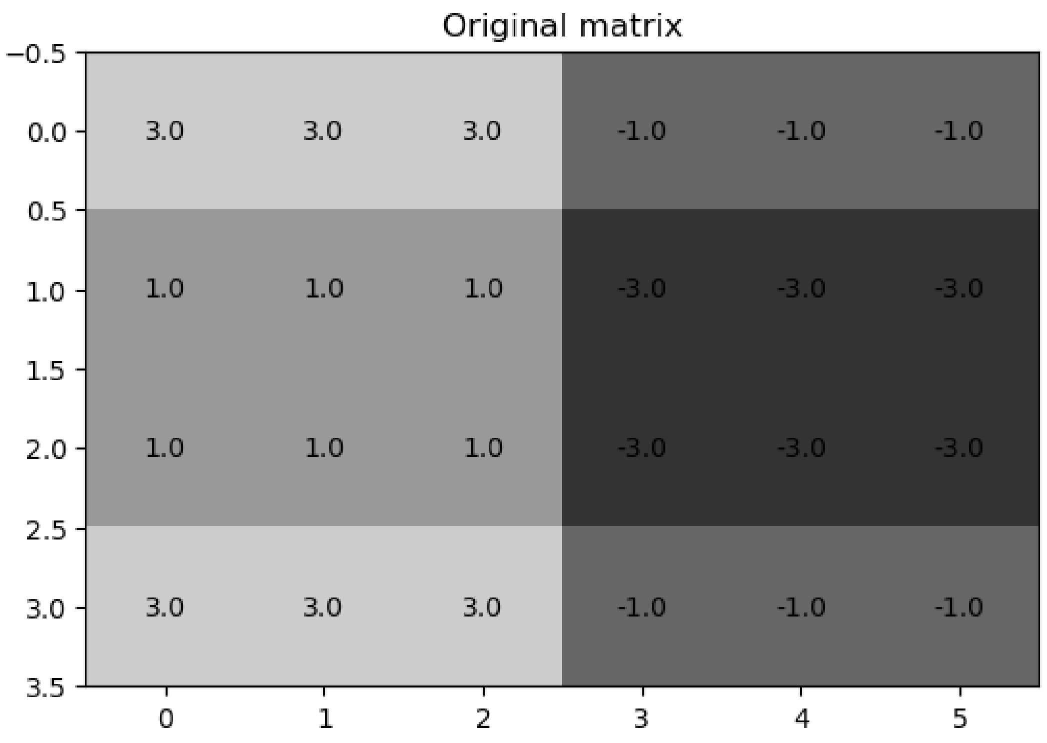
h) Define the rank-r approximation to A as $A_r = \sum_{i=1}^{r} \sigma_i u_i v_i^T$ where σ_i is the ith singular value with left singular vector u_i and right singular vector v_i.

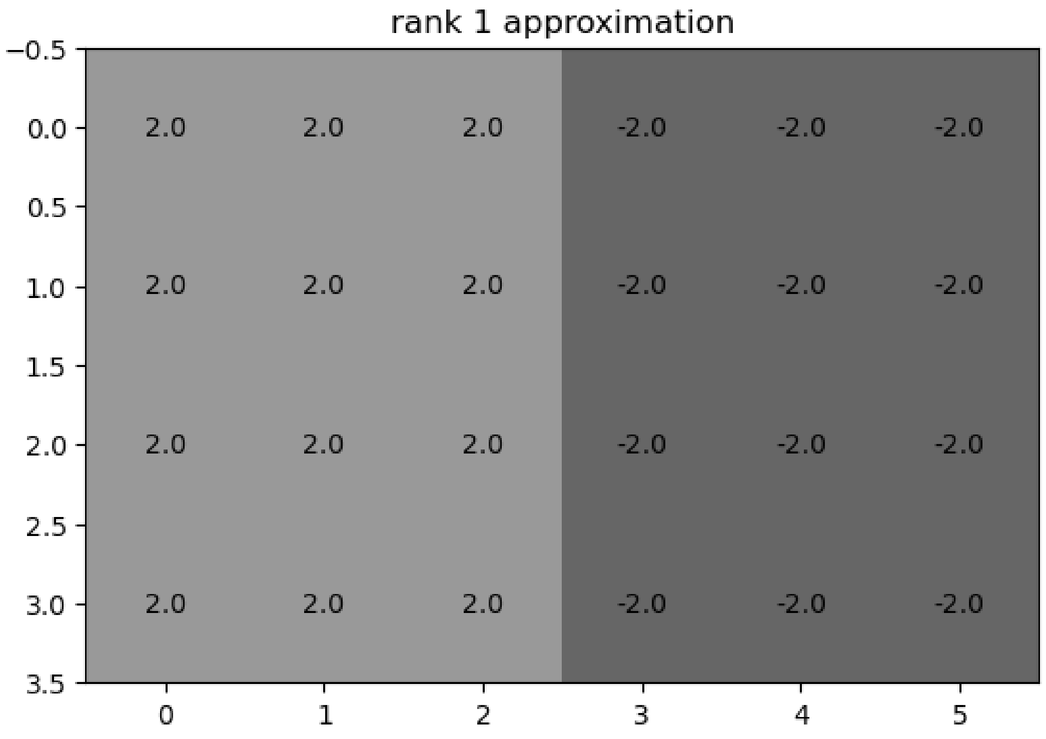
i. Find the rank-1 approximation A_1. How does A_1 compare to A?

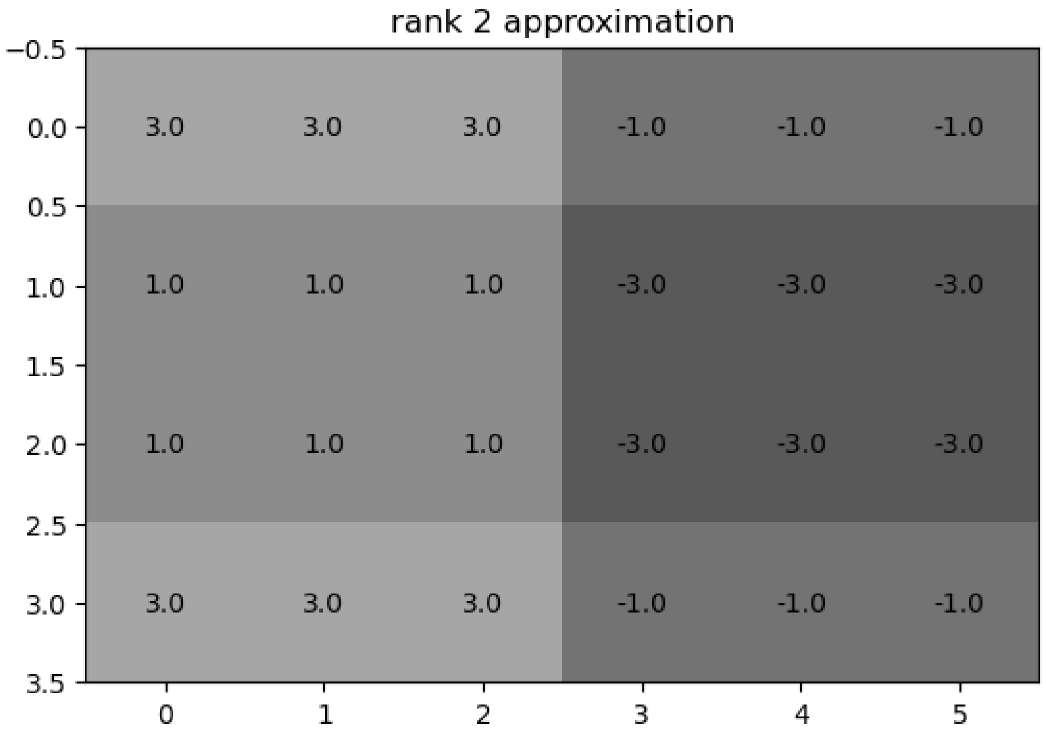
ii. Find the rank-2 approximation A_2. How does A_2 compare to A?

[48]:
```
## display the original matrix using a heatmap
plt.figure(num=None)
for (j,i),label in np.ndenumerate(A):
    plt.text(i,j,np.round(label,1),ha='center',va='center')
plt.imshow(A, vmin=-5, vmax=5, interpolation='none', cmap='gray')
plt.title('Original matrix' )

## display the rank-r approximations using a heatmap
for r in range(1,3):
    ## Fill in the blank: choose the first r colummns of U, first r singular␣
 ↪values, etc...
    A_rank_r_approx = U[:,:r]@S_matrix[:r,:r]@VT[:r,:]
    plt.figure(num=None)
    for (j,i),label in np.ndenumerate(A_rank_r_approx):
        plt.text(i,j,np.round(label,1),ha='center',va='center')
    plt.imshow(A_rank_r_approx, vmin=-10, vmax=10, interpolation='none',␣
 ↪cmap='gray')
    plt.title('rank ' + str(r) + ' approximation'  )
```




rank 1 approximation
2.0 2.0 2.0 -2.0 -2.0 -2.0
2.0 2.0 2.0 -2.0 -2.0 -2.0
2.0 2.0 2.0 -2.0 -2.0 -2.0
2.0 2.0 2.0 -2.0 -2.0 -2.0


rank 2 approximation
3.0 3.0 3.0 -1.0 -1.0 -1.0
1.0 1.0 1.0 -3.0 -3.0 -3.0
1.0 1.0 1.0 -3.0 -3.0 -3.0
3.0 3.0 3.0 -1.0 -1.0 -1.0

i) The economy SVD is based on the dimension of the matrices and does not consider the rank of the matrix. What is the smallest economy SVD (minimum dimension of the square matrix S) possible for the matrix A? Find U, S, and V for this minimal economy SVD.

According to , the smallest economy SVD would be that U is 4x2, s 2x2, and V 2x6.

```
[ ]:
```