```
#Name: Abhishek N. Singh
#Course: Matrix Decomposition for Data Analysis
#Entry No.: 322079
#
```

```
from util import *
#from utils import * #Note this is not needed
import numpy as np
from scipy.sparse import coo_matrix
import matplotlib.pyplot as plt
import time
```

```
#!pip install utils #This is not needed
```

```
Collecting utils
  Downloading https://files.pythonhosted.org/packages/55/e6/c2d2b2703e7debc8b501caae0e6
Installing collected packages: utils
Successfully installed utils-1.0.1
```

```
#Task 1: Matrix completion for movie ratings
#Download the data and code stub files from course's Moodle page. Among the files, there are
#that we use in this task: ratings.csv, ratings_validation.csv, and movies.txt. This data is
#movie ratings data from GroupLens Research.1
#It contains ratings (in 0 to 5 starts scale) of 9724 movies by
#610 users. Most of these ratings are in ratings.csv. The last rating (by timestamp) of every
#in the ratings_validation.csv file. We use this to evaluate how good our prediction of the r
#Do not use the validation data to train your model! The final file, movies.txt, contains the
#movies in the same order as they appear in the data (i.e. the first movie name corresponds t
#1, etc.).
#First, load the data and normalize the ratings so that they're first in scale from 1 to 6 st
#them by 6 so that the final ratings are in interval (0, 1]. We will use 0 to denote a missir
#can use standard sparse matrix storage formats.
#The code stub contains an implementation of an SGD algorithm. For this and the next task you
#this implementation so that the results are comparable. The code can be slow, so reserve enc
#complete these tasks!
#Now, get some basic idea how SGD algorithm works: run the provided SGD algorithm with this c
#using learning rate 0.001 and 100 epochs and no regularization, and observe how the error be
```

```
#epochs. Plot errors over epochs. Has the algorithm converged in 100 epochs? Argue!
#The algorithm does not do early stopping if the change between two consecutive epochs is sma
#Would this make sense for an SGD algorithm? Why or why not?
#You can also print the top-10 and bottom-10 movies per factor. Can you identify any movies c
#patterns? Include some examples in your report (but not all).
#In the previous run you used a fixed learning rate. A common heuristic to set dynamic learni
#the so-called bold driver heuristic. In bold driver heuristic, we increase the learning rate
#decreases over epochs. If the error increases, we drop the learning rate back to the base le
#provided function implements the bold driver heuristic with parameter bold_driver which is m
#with the learning rate when error decreases.3 Run SGD again, this time with learning rate 0.
#driver multiplier of 2. How does the behaviour change? Do you get better results? How is the
#Now, try around different step sizes, bold driver multipliers, and epoch numbers. Try only f
#values so that this part doesn't take too long; we will return to this topic in the next ass
#converge to (essentially) equally good training results with smaller number of epochs? How c
#the validation error?
#Next, we will work with biases. Start by running SGD with learning rate 0.0001 and bold driv
#2 but computing only the global, user, and movie bias. Hence, the prediction of a rating for
#i is μ + b
#user
#i + b
#movie
#j
#, where μ is the average rating. How are the results? Why? Argue!



#Now, using the same settings, compute both the factor matrices and the bias terms. Now the p
#rating for element (i, j) becomes
#μ + b
#user
#i + b
#movie
#j + li
#· rj = μ + b
#user
#i + b
#movie
#j + (LRT
#)ij ,
#where li
#is the ith row of left-hand factor matrix L and rj
#is the jth row or right-hand factor matrix R (here,
#R is transposed). Are the results better? Also print the movies with highest and lowest bias
#any patterns here?
#As the final part, we add an L2-regularizer for all terms. The error we now try to minimize
#∑
#(i,j)∈Ω
#(Di,j − μ − b
#user
#i − b
#movie
#j − li
```

```
#· rj)
#2 + λf kLk
#2
#F + λf kRk
#2
#F + λbkb
#userk
#2
#2 + λbkb
#moviek
#2
#2
#,
#where λf and λb are the factor and bias term regularizer constants, respectively (the provic
#more using regularizer coefficient value 0.001 for both factors and biases. How do the resul
#do the validation error results change? Why? Argue!
#Use the parameters (epoch number, step size, and bold driver multiplier) you found earlier t
#bias terms and regularizers. How are the results now?
#As the final task, compute the SVD of the rating matrix (assuming 0s mean 0s, not missing va
#compare the error in observed entries and the validation error.


tmp = np.genfromtxt('ratings.csv', delimiter=',', dtype='int32')
#Reading the data


tmp

     array([[   1,    1,    4],
            [   1,    2,    4],
            [   1,    3,    4],
            ...,
            [ 610, 9722,    3],
            [ 610, 9723,    3],
            [ 610, 9724,    3]], dtype=int32)


#Since the rating is on the column number 3 which has column index 2, we modify it
#That is normalization is done between value (0,6]
val = np.array(tmp[:,2]+1, dtype='float')/6


val

     array([0.83333333, 0.83333333, 0.83333333, ..., 0.66666667, 0.66666667,
            0.66666667])


#coo_matrix((data, (i, j)), [shape=(M, N)])
#to construct from three arrays:
#data[:] the entries of the matrix, in any order

#i[:] the row indices of the matrix entries
```

```
#j[:] the column indices of the matrix entries

#Where A[i[k], j[k]] = data[k]. When shape is not specified, it is inferred from the index ar
#So now the modified data is:
data = coo_matrix( (val, (tmp[:,0]-1, tmp[:,1]-1)) )
# data is 1-indexed, so we need to reduce 1 to get it 0-indexed


data
```

```
<610x9724 sparse matrix of type '<class 'numpy.float64'>'
        with 100226 stored elements in COOrdinate format>
```

```
data.toarray() #This is how we can see the data visually
```

```
array([[0.83333333, 0.83333333, 0.83333333, ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       ...,
       [0.5       , 0.5       , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [0.66666667, 0.        , 0.        , ..., 0.        , 0.        ,
        0.        ],
       [1.        , 0.        , 1.        , ..., 0.66666667, 0.66666667,
        0.66666667]])
```

```
n = data.get_shape()[0]
m = data.get_shape()[1]
avg = data.sum()/data.count_nonzero()
print(f'Data has {n} rows and {m} columns')
print(f'Average rating is {avg:.4F}')
```

```
Data has 610 rows and 9724 columns
Average rating is 0.7249
```

```
#Similarly, we prepare the validation data matrix
tmp = np.genfromtxt('ratings_validation.csv', delimiter=',', dtype='int32')
val = np.array(tmp[:,2]+1, dtype='float')/6
validation = coo_matrix( (val, (tmp[:,0]-1, tmp[:,1]-1)) )


validation
```

```
<610x9341 sparse matrix of type '<class 'numpy.float64'>'
        with 610 stored elements in COOrdinate format>
```

```
#SGD is provided in the additional file util.py which has been uploaded
# Compute just factors
res = sgd(data, 10, factor_learning_rate=0.001, epochs=100, verbose=True ) #factor_learning_r
```

```
Epoch  41: error 3098.03        factor_learning rate = 0.001
Epoch  42: error 3053.61        factor_learning rate = 0.001
Epoch  43: error 3012.08        factor_learning rate = 0.001
Epoch  44: error 2970.45        factor_learning rate = 0.001
Epoch  45: error 2931.56        factor_learning rate = 0.001
Epoch  46: error 2890.29        factor_learning rate = 0.001
Epoch  47: error 2856.78        factor_learning rate = 0.001
Epoch  48: error 2821.09        factor_learning rate = 0.001
Epoch  49: error 2784.41        factor_learning rate = 0.001
Epoch  50: error 2753.31        factor_learning rate = 0.001
Epoch  51: error 2719.67        factor_learning rate = 0.001
Epoch  52: error 2688.62        factor_learning rate = 0.001
Epoch  53: error 2658.55        factor_learning rate = 0.001
Epoch  54: error 2629.06        factor_learning rate = 0.001
Epoch  55: error 2601.25        factor_learning rate = 0.001
Epoch  56: error 2571.48        factor_learning rate = 0.001
Epoch  57: error 2545.28        factor_learning rate = 0.001
Epoch  58: error 2518.87        factor_learning rate = 0.001
Epoch  59: error 2495.94        factor_learning rate = 0.001
Epoch  60: error 2471.06        factor_learning rate = 0.001
Epoch  61: error 2445.79        factor_learning rate = 0.001
Epoch  62: error 2421.83        factor_learning rate = 0.001
Epoch  63: error 2401.31        factor_learning rate = 0.001
Epoch  64: error 2377.63        factor_learning rate = 0.001
Epoch  65: error 2355.29        factor_learning rate = 0.001
Epoch  66: error 2336.52        factor_learning rate = 0.001
Epoch  67: error 2314.03        factor_learning rate = 0.001
Epoch  68: error 2296.60        factor_learning rate = 0.001
Epoch  69: error 2276.61        factor_learning rate = 0.001
Epoch  70: error 2257.62        factor_learning rate = 0.001
Epoch  71: error 2239.59        factor_learning rate = 0.001
Epoch  72: error 2223.42        factor_learning rate = 0.001
Epoch  73: error 2204.06        factor_learning rate = 0.001
Epoch  74: error 2188.74        factor_learning rate = 0.001
Epoch  75: error 2172.01        factor_learning rate = 0.001
Epoch  76: error 2157.77        factor_learning rate = 0.001
Epoch  77: error 2139.62        factor_learning rate = 0.001
Epoch  78: error 2126.47        factor_learning rate = 0.001
Epoch  79: error 2110.57        factor_learning rate = 0.001
Epoch  80: error 2094.65        factor_learning rate = 0.001
Epoch  81: error 2082.63        factor_learning rate = 0.001
Epoch  82: error 2067.80        factor_learning rate = 0.001
Epoch  83: error 2053.95        factor_learning rate = 0.001
Epoch  84: error 2041.58        factor_learning rate = 0.001
Epoch  85: error 2028.66        factor_learning rate = 0.001
Epoch  86: error 2015.64        factor_learning rate = 0.001
Epoch  87: error 2003.03        factor_learning rate = 0.001
Epoch  88: error 1991.94        factor_learning rate = 0.001
Epoch  89: error 1979.44        factor_learning rate = 0.001
```

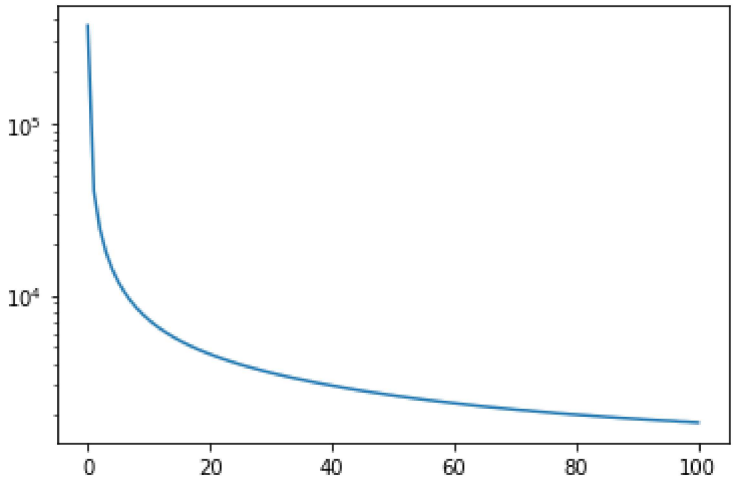
```
Epoch  90: error 1969.31        factor_learning rate = 0.001
Epoch  91: error 1958.02        factor_learning rate = 0.001
Epoch  92: error 1948.35        factor_learning rate = 0.001
Epoch  93: error 1936.20        factor_learning rate = 0.001
Epoch  94: error 1927.27        factor_learning rate = 0.001
Epoch  95: error 1915.00        factor_learning rate = 0.001
Epoch  96: error 1904.43        factor_learning rate = 0.001
Epoch  97: error 1895.47        factor_learning rate = 0.001
Epoch  98: error 1885.35        factor_learning rate = 0.001
Epoch  99: error 1877.05        factor_learning rate = 0.001
```

```
#We can see by the verbose that the error keeps on decreasing


val_err = squared_error(validation, res.L, res.R)
print('Only factors, fixed rate, no regularization')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Only factors, fixed rate, no regularization
Final training error = 1817.814 Relative error = 0.032481741082324427
    Validation error = 229.815  Relative error = 0.6027504825942747
```

```
# Plot errors over epocs
plt.plot(res.err)
plt.gca().set_yscale('log') # log scale makes these plots easier to read
plt.show()
plt.close()
```




```
#This confirms that the error keeps on decreasing as we saw in verbose


#However, the error has not converged at 100 epochs given that the curve values are still drc
#we can agree to a convergence value when the difference is very small.
```

```
#The algorithm does not do early stopping if the change between two
#consecutive epochs is small enough. In stochastic gradient descent
#since the gradient vector is chosen randomly and so it is very much possible
#that the algorithm might not reach at exact global minima. Thus, it would be
#very much sensible to stop at a a minimum difference in consecutive epoch value


# Read movies
movies = []
with open('movies.txt') as f:
    for row in f:
        movies.append(row.strip())


# Print the top-10 and bottom-10 movies
print_factors(res.R, movies)
```

```
       2     Caddyshack II (1988)    -0.240
       3     Curly Sue (1991)        -0.229
       4     Always Watching: A Marble Hornets Story (2015)  -0.190
       5     Assassin's Creed (2016) -0.183
       6     Candyman 3: Day of the Dead (1999)      -0.181
       7     Hard Ticket to Hawaii (1987)    -0.176
       8     Detroit Rock City (1999)        -0.174
       9     What the #$*! Do We Know!? (a.k.a. What the Bleep Do We Know!?) (2004)  -0.16
      10     Vie en Rose, La (Môme, La) (2007)       -0.169

     Factor 8 top-10
       1     Dam Busters, The (1955) 0.374
       2     Cincinnati Kid, The (1965)      0.364
       3     Infinity (1996) 0.362
       4     Used People (1992)      0.361
       5     Thin Man Goes Home, The (1945)  0.361
       6     Sea of Love (1989)      0.360
       7     Angst  (1983)   0.360
       8     In Love and War (1996)  0.360
       9     Losin' It (1983)        0.359
      10     Jazz Singer, The (1927) 0.358

     Factor 8 bottom-10
       1     Baby Geniuses (1999)    -0.212
       2     Angel Heart (1987)      -0.208
       3     Unknown (2006)  -0.202
       4     They Shoot Horses, Don't They? (1969)   -0.200
       5     Wagons East (1994)      -0.190
       6     Edge of Heaven, The (Auf der anderen Seite) (2007)      -0.186
       7     Asphyx, The (1973)      -0.180
       8     Babylon 5       -0.179
       9     Whale Rider (2002)      -0.176
      10     Devil and Daniel Johnston, The (2005)   -0.173

     Factor 9 top-10
       1     Comic-Con Episode IV: A Fan's Hope (2011)       0.382
       2     Returner (Ritaanaa) (2002)      0.381
       3     Revolution (1985)       0.375
       4     Sleepwalkers (1992)     0.374
       5     The Flash 2 - Revenge of the Trickster (1991)   0.373
```

```
 6     Afro Samurai (2007)     0.371
 7     Mutant Aliens (2001)    0.367
 8     V. I. Warshawski (1991) 0.366
 9     Lionheart (1990)        0.365
10     Danny Deckchair (2003)  0.365

Factor 9 bottom-10
 1     Amityville: A New Generation (1993)     -0.254
 2     Doug's 1st Movie (1999) -0.246
 3     Loverboy (1989) -0.227
 4     The Star Wars Holiday Special (1978)    -0.221
 5     Pollyanna (1960)        -0.211
 6     Ringu 0: Bâsudei (2000) -0.209
 7     Turbo: A Power Rangers Movie (1997)     -0.202
 8     La vérité si je mens ! (1997)   -0.200
 9     Strangeland (1998)      -0.195
10     Newton Boys, The (1998) -0.194
```

```
#The pattern is that the top movies have score abouve 0.3 while bottom ones have scores
# in negative


# Compute just factors, but use bold driver and slower learning
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001)
```

```
Epoch  41: error 1417.77        factor_learning rate = 0.0001
Epoch  42: error 1414.60        factor_learning rate = 0.0002
Epoch  43: error 1412.80        factor_learning rate = 0.0004
Epoch  44: error 1412.27        factor_learning rate = 0.0008
Epoch  45: error 1414.10        factor_learning rate = 0.0001
Epoch  46: error 1411.39        factor_learning rate = 0.0002
Epoch  47: error 1409.80        factor_learning rate = 0.0004
Epoch  48: error 1409.35        factor_learning rate = 0.0008
Epoch  49: error 1411.00        factor_learning rate = 0.0001
Epoch  50: error 1408.58        factor_learning rate = 0.0002
Epoch  51: error 1407.08        factor_learning rate = 0.0004
Epoch  52: error 1406.59        factor_learning rate = 0.0008
Epoch  53: error 1408.69        factor_learning rate = 0.0001
Epoch  54: error 1406.12        factor_learning rate = 0.0002
Epoch  55: error 1404.52        factor_learning rate = 0.0004
Epoch  56: error 1404.16        factor_learning rate = 0.0008
Epoch  57: error 1406.26        factor_learning rate = 0.0001
Epoch  58: error 1403.55        factor_learning rate = 0.0002
Epoch  59: error 1402.00        factor_learning rate = 0.0004
Epoch  60: error 1401.61        factor_learning rate = 0.0008
Epoch  61: error 1403.09        factor_learning rate = 0.0001
Epoch  62: error 1400.81        factor_learning rate = 0.0002
Epoch  63: error 1399.53        factor_learning rate = 0.0004
Epoch  64: error 1399.33        factor_learning rate = 0.0008
Epoch  65: error 1402.02        factor_learning rate = 0.0001
Epoch  66: error 1398.83        factor_learning rate = 0.0002
Epoch  67: error 1397.27        factor_learning rate = 0.0004
Epoch  68: error 1397.07        factor_learning rate = 0.0008
Epoch  69: error 1398.37        factor_learning rate = 0.0001
Epoch  70: error 1396.30        factor_learning rate = 0.0002
```

```
Epoch  71: error 1394.99        factor_learning rate = 0.0004
Epoch  72: error 1394.69        factor_learning rate = 0.0008
Epoch  73: error 1396.56        factor_learning rate = 0.0001
Epoch  74: error 1394.22        factor_learning rate = 0.0002
Epoch  75: error 1392.76        factor_learning rate = 0.0004
Epoch  76: error 1392.53        factor_learning rate = 0.0008
Epoch  77: error 1394.88        factor_learning rate = 0.0001
Epoch  78: error 1392.23        factor_learning rate = 0.0002
Epoch  79: error 1390.66        factor_learning rate = 0.0004
Epoch  80: error 1390.36        factor_learning rate = 0.0008
Epoch  81: error 1392.31        factor_learning rate = 0.0001
Epoch  82: error 1389.88        factor_learning rate = 0.0002
Epoch  83: error 1388.38        factor_learning rate = 0.0004
Epoch  84: error 1388.20        factor_learning rate = 0.0008
Epoch  85: error 1389.61        factor_learning rate = 0.0001

Epoch  86: error 1387.55        factor_learning rate = 0.0002
Epoch  87: error 1386.33        factor_learning rate = 0.0004
Epoch  88: error 1385.97        factor_learning rate = 0.0008
Epoch  89: error 1388.05        factor_learning rate = 0.0001
Epoch  90: error 1385.76        factor_learning rate = 0.0002
Epoch  91: error 1384.35        factor_learning rate = 0.0004
Epoch  92: error 1383.98        factor_learning rate = 0.0008
Epoch  93: error 1385.41        factor_learning rate = 0.0001
Epoch  94: error 1383.56        factor_learning rate = 0.0002
Epoch  95: error 1382.23        factor_learning rate = 0.0004
Epoch  96: error 1381.97        factor_learning rate = 0.0008
Epoch  97: error 1384.69        factor_learning rate = 0.0001
Epoch  98: error 1381.82        factor_learning rate = 0.0002
Epoch  99: error 1380.31        factor_learning rate = 0.0004
```

```
val_err = squared_error(validation, res.L, res.R)
print('Only factors, fixed rate, no regularization')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Only factors, fixed rate, no regularization
Final training error = 1380.074 Relative error = 0.024659964429735608
    Validation error = 221.044  Relative error = 0.5797447581574955
```

```
#So we see that the validation error has reduced this time with bold driver heuristic


# Plot errors over epocs
plt.plot(res.err)
plt.gca().set_yscale('log') # log scale makes these plots easier to read
plt.show()
plt.close()
```


```
#The plot above clearly shows that the convergence to a more straight line takes place
#much faster this time


# Compute just factors, but use bold driver and slower learning
res = sgd(data, 10, bold_driver=3, factor_learning_rate=0.001)
```

```
Epoch  41: error 1358.33        factor_learning rate = 0.003
Epoch  42: error 1385.69        factor_learning rate = 0.001
Epoch  43: error 1354.12        factor_learning rate = 0.003
Epoch  44: error 1382.99        factor_learning rate = 0.001
Epoch  45: error 1349.12        factor_learning rate = 0.003
Epoch  46: error 1377.30        factor_learning rate = 0.001
Epoch  47: error 1346.17        factor_learning rate = 0.003
Epoch  48: error 1378.10        factor_learning rate = 0.001
Epoch  49: error 1341.68        factor_learning rate = 0.003
Epoch  50: error 1371.94        factor_learning rate = 0.001
Epoch  51: error 1339.61        factor_learning rate = 0.003
Epoch  52: error 1370.26        factor_learning rate = 0.001
Epoch  53: error 1333.97        factor_learning rate = 0.003
Epoch  54: error 1359.49        factor_learning rate = 0.001
Epoch  55: error 1330.27        factor_learning rate = 0.003
Epoch  56: error 1362.01        factor_learning rate = 0.001
Epoch  57: error 1327.28        factor_learning rate = 0.003
Epoch  58: error 1358.79        factor_learning rate = 0.001
Epoch  59: error 1324.17        factor_learning rate = 0.003
Epoch  60: error 1349.38        factor_learning rate = 0.001
Epoch  61: error 1320.25        factor_learning rate = 0.003
Epoch  62: error 1348.73        factor_learning rate = 0.001
Epoch  63: error 1317.52        factor_learning rate = 0.003
Epoch  64: error 1345.52        factor_learning rate = 0.001
Epoch  65: error 1314.02        factor_learning rate = 0.003
Epoch  66: error 1344.71        factor_learning rate = 0.001
Epoch  67: error 1311.98        factor_learning rate = 0.003
Epoch  68: error 1339.31        factor_learning rate = 0.001
Epoch  69: error 1307.83        factor_learning rate = 0.003
Epoch  70: error 1335.77        factor_learning rate = 0.001
Epoch  71: error 1305.64        factor_learning rate = 0.003
Epoch  72: error 1336.85        factor_learning rate = 0.001
Epoch  73: error 1302.57        factor_learning rate = 0.003
Epoch  74: error 1329.09        factor_learning rate = 0.001
Epoch  75: error 1299.83        factor_learning rate = 0.003
Epoch  76: error 1331.98        factor_learning rate = 0.001
Epoch  77: error 1296.26        factor_learning rate = 0.003
Epoch  78: error 1327.48        factor_learning rate = 0.001
```

```
Epoch  79: error 1295.43        factor_learning rate = 0.003
Epoch  80: error 1320.53        factor_learning rate = 0.001
Epoch  81: error 1291.63        factor_learning rate = 0.003
Epoch  82: error 1323.90        factor_learning rate = 0.001
Epoch  83: error 1289.14        factor_learning rate = 0.003
Epoch  84: error 1320.91        factor_learning rate = 0.001
Epoch  85: error 1287.71        factor_learning rate = 0.003
Epoch  86: error 1315.20        factor_learning rate = 0.001
Epoch  87: error 1284.06        factor_learning rate = 0.003
Epoch  88: error 1314.30        factor_learning rate = 0.001
Epoch  89: error 1282.15        factor_learning rate = 0.003
Epoch  90: error 1310.41        factor_learning rate = 0.001
Epoch  91: error 1279.72        factor_learning rate = 0.003
Epoch  92: error 1310.53        factor_learning rate = 0.001
Epoch  93: error 1278.29        factor_learning rate = 0.003

Epoch  94: error 1307.28        factor_learning rate = 0.001
Epoch  95: error 1275.29        factor_learning rate = 0.003
Epoch  96: error 1303.02        factor_learning rate = 0.001
Epoch  97: error 1273.62        factor_learning rate = 0.003
Epoch  98: error 1301.70        factor_learning rate = 0.001
Epoch  99: error 1271.54        factor_learning rate = 0.003
```

```
val_err = squared_error(validation, res.L, res.R)
print('Only factors, fixed rate, no regularization')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Only factors, fixed rate, no regularization
Final training error = 1297.298 Relative error = 0.023180870199845993
    Validation error = 191.577  Relative error = 0.5024611010341171
```

```
#So now I manage to reduce the error further


# Plot errors over epocs
plt.plot(res.err)
plt.gca().set_yscale('log') # log scale makes these plots easier to read
plt.show()
plt.close()
```

```
# Compute just factors, but use bold driver and slower learning
res = sgd(data, 10, bold_driver=3, factor_learning_rate=0.01)
```

```
Epoch  41: error 1508.07        factor_learning rate = 0.01
Epoch  42: error 1498.04        factor_learning rate = 0.01
Epoch  43: error 1506.67        factor_learning rate = 0.01
Epoch  44: error 1482.92        factor_learning rate = 0.01
Epoch  45: error 1451.89        factor_learning rate = 0.01
Epoch  46: error 1483.03        factor_learning rate = 0.01
Epoch  47: error 1470.73        factor_learning rate = 0.01
Epoch  48: error 1437.58        factor_learning rate = 0.01
Epoch  49: error 1469.46        factor_learning rate = 0.01
Epoch  50: error 1459.17        factor_learning rate = 0.01
Epoch  51: error 1441.51        factor_learning rate = 0.01
Epoch  52: error 1466.80        factor_learning rate = 0.01
Epoch  53: error 1438.28        factor_learning rate = 0.01
Epoch  54: error 1446.03        factor_learning rate = 0.01
Epoch  55: error 1432.32        factor_learning rate = 0.01
Epoch  56: error 1430.43        factor_learning rate = 0.01
Epoch  57: error 1447.81        factor_learning rate = 0.01
Epoch  58: error 1437.31        factor_learning rate = 0.01
Epoch  59: error 1453.65        factor_learning rate = 0.01
Epoch  60: error 1415.71        factor_learning rate = 0.01
Epoch  61: error 1418.99        factor_learning rate = 0.01
Epoch  62: error 1435.39        factor_learning rate = 0.01
Epoch  63: error 1422.10        factor_learning rate = 0.01
Epoch  64: error 1446.08        factor_learning rate = 0.01
Epoch  65: error 1387.13        factor_learning rate = 0.01
Epoch  66: error 1401.12        factor_learning rate = 0.01
Epoch  67: error 1416.84        factor_learning rate = 0.01
Epoch  68: error 1402.72        factor_learning rate = 0.01
Epoch  69: error 1429.64        factor_learning rate = 0.01
Epoch  70: error 1398.91        factor_learning rate = 0.01
Epoch  71: error 1409.83        factor_learning rate = 0.01
Epoch  72: error 1391.43        factor_learning rate = 0.01
Epoch  73: error 1388.94        factor_learning rate = 0.01
Epoch  74: error 1393.11        factor_learning rate = 0.01
Epoch  75: error 1403.41        factor_learning rate = 0.01
Epoch  76: error 1382.72        factor_learning rate = 0.01
Epoch  77: error 1414.56        factor_learning rate = 0.01
Epoch  78: error 1394.75        factor_learning rate = 0.01
Epoch  79: error 1390.33        factor_learning rate = 0.01
Epoch  80: error 1388.20        factor_learning rate = 0.01
Epoch  81: error 1378.10        factor_learning rate = 0.01
Epoch  82: error 1365.83        factor_learning rate = 0.01
Epoch  83: error 1380.85        factor_learning rate = 0.01
Epoch  84: error 1354.04        factor_learning rate = 0.01
Epoch  85: error 1392.73        factor_learning rate = 0.01
Epoch  86: error 1383.27        factor_learning rate = 0.01
Epoch  87: error 1401.98        factor_learning rate = 0.01
Epoch  88: error 1391.88        factor_learning rate = 0.01
Epoch  89: error 1381.46        factor_learning rate = 0.01
Epoch  90: error 1352.71        factor_learning rate = 0.01
```

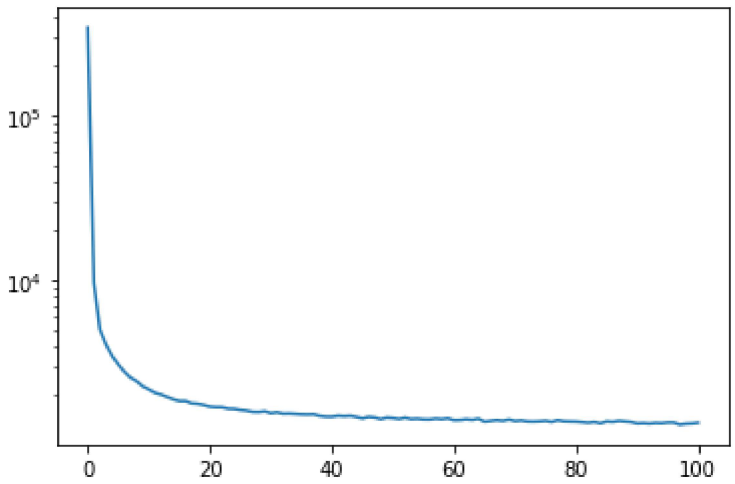
```
Epoch  91: error 1358.28        factor_learning rate = 0.01
Epoch  92: error 1349.60        factor_learning rate = 0.01
Epoch  93: error 1361.25        factor_learning rate = 0.01
Epoch  94: error 1356.77        factor_learning rate = 0.01
Epoch  95: error 1369.15        factor_learning rate = 0.01
Epoch  96: error 1370.30        factor_learning rate = 0.01
Epoch  97: error 1331.42        factor_learning rate = 0.01
Epoch  98: error 1347.75        factor_learning rate = 0.01
Epoch  99: error 1352.07        factor_learning rate = 0.01
```

```
val_err = squared_error(validation, res.L, res.R)
print('Only factors, fixed rate, no regularization')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Only factors, fixed rate, no regularization
Final training error = 1365.280 Relative error = 0.024395606035237705
    Validation error = 188.997  Relative error = 0.49569468928693805
```

```
#The validation error drops further :)
```

```
# Plot errors over epocs
plt.plot(res.err)
plt.gca().set_yscale('log') # log scale makes these plots easier to read
plt.show()
plt.close()
```




```
# Compute factors and bias
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, ve
```

```
Epoch  41: error 1315.61        factor_learning rate = 0.0008
Epoch  42: error 1317.74        factor_learning rate = 0.0001
Epoch  43: error 1315.01        factor_learning rate = 0.0002
Epoch  44: error 1313.44        factor_learning rate = 0.0004
Epoch  45: error 1313.07        factor_learning rate = 0.0008
Epoch  46: error 1315.36        factor_learning rate = 0.0001
```

```
Epoch  47: error 1312.50        factor_learning rate = 0.0002
Epoch  48: error 1310.95        factor_learning rate = 0.0004
Epoch  49: error 1310.76        factor_learning rate = 0.0008
Epoch  50: error 1314.17        factor_learning rate = 0.0001
Epoch  51: error 1310.46        factor_learning rate = 0.0002
Epoch  52: error 1308.62        factor_learning rate = 0.0004
Epoch  53: error 1308.35        factor_learning rate = 0.0008
Epoch  54: error 1310.69        factor_learning rate = 0.0001
Epoch  55: error 1307.84        factor_learning rate = 0.0002
Epoch  56: error 1306.30        factor_learning rate = 0.0004
Epoch  57: error 1306.02        factor_learning rate = 0.0008
Epoch  58: error 1309.46        factor_learning rate = 0.0001
Epoch  59: error 1305.86        factor_learning rate = 0.0002
Epoch  60: error 1304.19        factor_learning rate = 0.0004
Epoch  61: error 1304.06        factor_learning rate = 0.0008

Epoch  62: error 1306.32        factor_learning rate = 0.0001
Epoch  63: error 1303.62        factor_learning rate = 0.0002
Epoch  64: error 1302.06        factor_learning rate = 0.0004
Epoch  65: error 1302.00        factor_learning rate = 0.0008
Epoch  66: error 1304.96        factor_learning rate = 0.0001
Epoch  67: error 1301.71        factor_learning rate = 0.0002
Epoch  68: error 1300.11        factor_learning rate = 0.0004
Epoch  69: error 1299.80        factor_learning rate = 0.0008
Epoch  70: error 1303.84        factor_learning rate = 0.0001
Epoch  71: error 1300.10        factor_learning rate = 0.0002
Epoch  72: error 1298.18        factor_learning rate = 0.0004
Epoch  73: error 1298.01        factor_learning rate = 0.0008
Epoch  74: error 1300.33        factor_learning rate = 0.0001
Epoch  75: error 1297.63        factor_learning rate = 0.0002
Epoch  76: error 1296.15        factor_learning rate = 0.0004
Epoch  77: error 1296.06        factor_learning rate = 0.0008
Epoch  78: error 1298.64        factor_learning rate = 0.0001
Epoch  79: error 1295.76        factor_learning rate = 0.0002
Epoch  80: error 1294.26        factor_learning rate = 0.0004
Epoch  81: error 1294.04        factor_learning rate = 0.0008
Epoch  82: error 1298.13        factor_learning rate = 0.0001
Epoch  83: error 1294.34        factor_learning rate = 0.0002
Epoch  84: error 1292.48        factor_learning rate = 0.0004
Epoch  85: error 1292.34        factor_learning rate = 0.0008
Epoch  86: error 1295.09        factor_learning rate = 0.0001
Epoch  87: error 1292.25        factor_learning rate = 0.0002
Epoch  88: error 1290.71        factor_learning rate = 0.0004
Epoch  89: error 1290.41        factor_learning rate = 0.0008
Epoch  90: error 1294.25        factor_learning rate = 0.0001
Epoch  91: error 1290.72        factor_learning rate = 0.0002
Epoch  92: error 1289.01        factor_learning rate = 0.0004
Epoch  93: error 1289.00        factor_learning rate = 0.0008
Epoch  94: error 1291.50        factor_learning rate = 0.0001
Epoch  95: error 1288.82        factor_learning rate = 0.0002
Epoch  96: error 1287.22        factor_learning rate = 0.0004
Epoch  97: error 1287.01        factor_learning rate = 0.0008
Epoch  98: error 1290.86        factor_learning rate = 0.0001
Epoch  99: error 1287.33        factor_learning rate = 0.0002
```

```
val_err = squared_error(validation, res.L, res.R, res.global_bias, res.row_bias, res.col_bias
print('Bias only, no regularization')
```

```
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Bias only, no regularization
Final training error = 1285.582 Relative error = 0.02297151945956205
    Validation error = 222.025  Relative error = 0.5823179086533342
```

```
# Print the movies based on their bias top 10 and bottom 10
print_factors(res.col_bias, movies)
```

```
Factor 0 top-10
  1     Pulp Fiction (1994)     0.299
  2     Cell 211 (Celda 211) (2009)     0.279
  3     Reservoir Dogs (1992)   0.264
  4     Hearts of Darkness: A Filmmakers Apocalypse (1991)      0.259
  5     Watchmen: Tales of the Black Freighter (2009)   0.257
  6     Syrup (2013)    0.255
  7     Stalker (1979)  0.248
  8     Léon: The Professional (a.k.a. The Professional) (Léon) (1994)  0.241
  9     And the Band Played On (1993)   0.236
 10     Fist Fight (2017)       0.235

Factor 0 bottom-10
  1     No Small Affair (1984)  -0.455
  2     Infernal Affairs 2 (Mou gaan dou II) (2003)     -0.388
  3     Stefan Zweig: Farewell to Europe (2016) -0.362
  4     Urban Legends: Final Cut (2000) -0.353
  5     Grind (2003)    -0.351
  6     Aelita: The Queen of Mars (Aelita) (1924)       -0.347
  7     My Bloody Valentine (1981)      -0.344
  8     Boat Trip (2003)        -0.344
  9     The Crew (2016) -0.343
 10     3 Days to Kill (2014)   -0.343
```

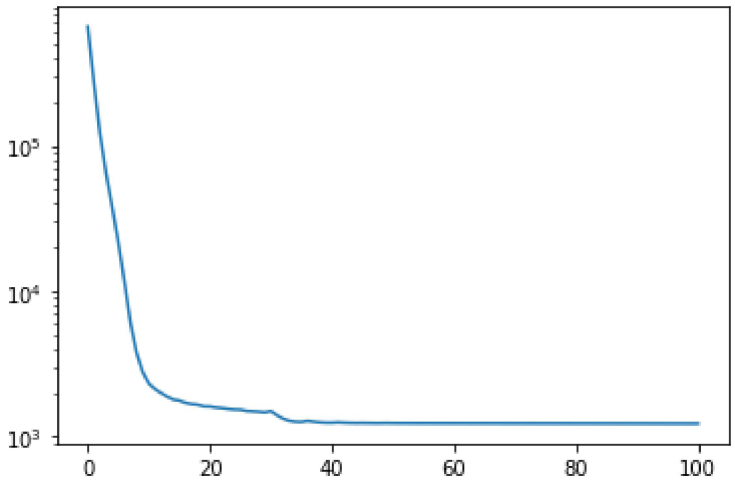
```
# Plot errors over epocs
plt.plot(res.err)
plt.gca().set_yscale('log') # log scale makes these plots easier to read
plt.show()
plt.close()
#The results arn't better because the validation error is 222 while I was already
#able to achieve a validation error of 188 earlier.
```


```
#Now doing SGD with regularization L2-Regularization
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, fa
```

```
val_err = squared_error(validation, res.L, res.R, res.global_bias, res.row_bias, res.col_bias
print('Factors and bias, regularized')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
     Factors and bias, regularized
     Final training error = 1231.810 Relative error = 0.022010698393192092
         Validation error = 203.652  Relative error = 0.5341297631399506
```

```
# Plot errors over epocs
plt.plot(res.err)
plt.gca().set_yscale('log') # log scale makes these plots easier to read
plt.show()
plt.close()
#The results arn't better because the validation error is 222 while I was already
#able to achieve a validation error of 188 earlier.
```




```
egularization L2-Regularization, bold driver of 3, factor learning rate of 0.01
old_driver=3, factor_learning_rate=0.01, bias_learning_rate=0.0001, factor_regu=0.001, bias_r
```

```
     Epoch   0: error 653219.96      factor_learning rate = 0.01
     Epoch   1: error 8972.64        factor_learning rate = 0.01
     Epoch   2: error 4457.27        factor_learning rate = 0.01
     Epoch   3: error 3431.22        factor_learning rate = 0.01
```

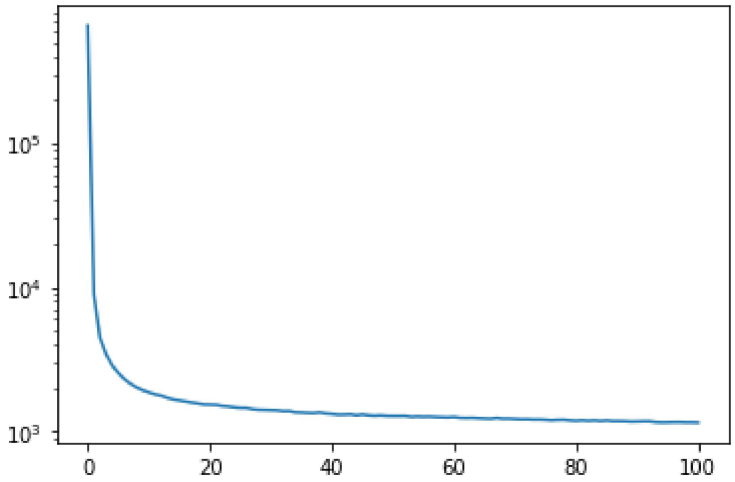
```
Epoch   4: error 2870.53        factor_learning rate = 0.01
Epoch   5: error 2546.66        factor_learning rate = 0.01
Epoch   6: error 2306.88        factor_learning rate = 0.01
Epoch   7: error 2141.87        factor_learning rate = 0.01
Epoch   8: error 2019.36        factor_learning rate = 0.01
Epoch   9: error 1932.13        factor_learning rate = 0.01
Epoch  10: error 1866.20        factor_learning rate = 0.01
Epoch  11: error 1807.71        factor_learning rate = 0.01
Epoch  12: error 1771.94        factor_learning rate = 0.01
Epoch  13: error 1712.39        factor_learning rate = 0.01
Epoch  14: error 1667.06        factor_learning rate = 0.01
Epoch  15: error 1640.71        factor_learning rate = 0.01
Epoch  16: error 1613.60        factor_learning rate = 0.01
Epoch  17: error 1586.59        factor_learning rate = 0.01
Epoch  18: error 1564.48        factor_learning rate = 0.01
Epoch  19: error 1537.26        factor_learning rate = 0.01
Epoch  20: error 1533.13        factor_learning rate = 0.01
Epoch  21: error 1528.86        factor_learning rate = 0.01
Epoch  22: error 1502.74        factor_learning rate = 0.01
Epoch  23: error 1490.45        factor_learning rate = 0.01
Epoch  24: error 1469.40        factor_learning rate = 0.01
Epoch  25: error 1456.77        factor_learning rate = 0.01
Epoch  26: error 1455.65        factor_learning rate = 0.01
Epoch  27: error 1426.81        factor_learning rate = 0.01
Epoch  28: error 1412.73        factor_learning rate = 0.01
Epoch  29: error 1409.39        factor_learning rate = 0.01
Epoch  30: error 1404.42        factor_learning rate = 0.01
Epoch  31: error 1399.30        factor_learning rate = 0.01
Epoch  32: error 1383.29        factor_learning rate = 0.01
Epoch  33: error 1384.88        factor_learning rate = 0.01
Epoch  34: error 1355.22        factor_learning rate = 0.01
Epoch  35: error 1351.55        factor_learning rate = 0.01
Epoch  36: error 1343.42        factor_learning rate = 0.01
Epoch  37: error 1340.98        factor_learning rate = 0.01
Epoch  38: error 1352.54        factor_learning rate = 0.01
Epoch  39: error 1332.35        factor_learning rate = 0.01
Epoch  40: error 1322.87        factor_learning rate = 0.01
Epoch  41: error 1307.07        factor_learning rate = 0.01
Epoch  42: error 1308.47        factor_learning rate = 0.01
Epoch  43: error 1315.10        factor_learning rate = 0.01
Epoch  44: error 1294.51        factor_learning rate = 0.01
Epoch  45: error 1310.83        factor_learning rate = 0.01
Epoch  46: error 1292.40        factor_learning rate = 0.01
Epoch  47: error 1283.86        factor_learning rate = 0.01
Epoch  48: error 1290.73        factor_learning rate = 0.01
Epoch  49: error 1280.16        factor_learning rate = 0.01
Epoch  50: error 1278.88        factor_learning rate = 0.01
Epoch  51: error 1278.71        factor_learning rate = 0.01
Epoch  52: error 1280.26        factor_learning rate = 0.01
Epoch  53: error 1262.37        factor_learning rate = 0.01
Epoch  54: error 1268.39        factor_learning rate = 0.01
Epoch  55: error 1263.37        factor_learning rate = 0.01
Epoch  56: error 1266.02        factor_learning rate = 0.01
Epoch  57: error 1259.22        factor_learning rate = 0.01
Epoch  58: error 1254.53        factor_learning rate = 0.01
```

```
val_err = squared_error(validation, res.L, res.R, res.global_bias, res.row_bias, res.col_bias
print('Factors and bias  regularized')
```

```
print( Factors and bias, regularized )
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Factors and bias, regularized
Final training error = 1151.517 Relative error = 0.020575962304935237
    Validation error = 267.460  Relative error = 0.7014830460905507
```

```
# Plot errors over epocs
plt.plot(res.err)
plt.gca().set_yscale('log') # log scale makes these plots easier to read
plt.show()
plt.close()
```




```
#Regularization comes with a cost of increasing the error rate. The validation error has incr
#But perhaps that is what is needed.
```

```
fullArray = data.toarray()
```

```
from numpy.linalg import svd
## Compute SVD of data
U, S, V = svd(fullArray, full_matrices=False, compute_uv=True)
```

```
#Squared Error
se = S @ V
```

```
se
```

```
array([ 7.90568756, -8.37391345, -0.05052415, ..., -0.73988401,
       -0.73988401, -0.73988401])
```

```
#To get truncated SVD
from sklearn.utils.extmath import randomized_svd

Utruncated, Sigma, VTtruncated = randomized_svd(fullArray,n_components=10,n_iter=5,random_sta


seTruncated = Sigma @ VTtruncated


seTruncated
```

```
array([0.06608784, 0.64533515, 4.51407857, ..., 0.26432449, 0.26432449,
       0.26432449])
```

```
#Now doing the same for validation data


validationFull = validation.toarray()


validationFull
```

```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

```
UtruncatedValid, SigmaValid, VTtruncatedValid = randomized_svd(validationFull,n_components=10


seValidationTruncated = SigmaValid @ VTtruncatedValid


seValidationTruncated
```

```
array([-1.66403198e-16, -2.38323818e-16,  3.52371195e-16, ...,
        0.00000000e+00,  0.00000000e+00, -9.59328450e-07])
```

```
len(seValidationTruncated)
```

```
9341
```

```
len(seTruncated)
```

```
9724
```

```
#Since the lenghts are differenct, they cannot be 1 to 1 compared.


#Task 2#
#Task 2: Initial solutions and hyperparameter optimization
#We continue using the same data and code as above. We start by looking at the effect of the
#solutions. By default, the code uses uniform distribution from unit interval for factor matr
#normalized, and normal distribution with 0.1 standard deviation for biases. Try following ch
#solutions and report the results. Explain what happened and why you think it happened (e.g.
#better because something or the code crashed because of something else).
#a) Factor matrices and bias vectors are all-1s matrices and vectors. (No normalization.)
#b) Factor matrices and bias vectors are all-0s matrices and vectors. (No normalization.)
#c) Factor matrices are drawn from normal distribution with 0.1 standard deviation. (No norma
#d) Factor matrices are random integer matrices with values from 0 to 6, normalized.
#Next we try to optimize the hyperparameters. The algorithm has the following hyperparameters
#a) rank
#b) number of epochs
#c) learning rates (for simplicity, let’s assume they’re same for factors and biases)
#d) bold driver multiplier
#e) regularization coefficients (again, same for factors and biases).
#
#The standard way to optimize these is to select some values for each hyperparameter and try
#combinations of the potential values. This is called a grid search. To test how good each co
#one should also run cross validation; a 5-fold cross validation, for example, would split th
#into five random subsets, find the decomposition based on four of them, and test the result
#repeat four more times so that each subset would get left out once. The final quality value
#average error over the five runs. Assume we want to try 5 different values for each hyperpar
#5-fold cross validation. Time the algorithm running once over the data set and estimate how
#grid search would take. You can assume 5-fold cross validation takes five times the time one
#that run time grows linearly with number of epochs and rank. Report what you calculated.
#If the expected run time exceeds the time you have before the DL, design an experiment where
#at least the following (you can change the values if you want, but explain why in your repor
#a) rank = 10
#b) number of epochs = 100 (or smaller if you think you converge)
#c) learning rates in {10−5
#, 10−4
#, 10−3}
#d) bold driver multiplier = 2 (or some other coefficient)
#e) regularization coefficients in {10−6
#, 10−5
#, 10−4}.
#Notice that this still leaves you with nine combinations that you must be able to run. Be pr
#your computer running overnight!
#After you’ve found the optimum combination of hyperparameters, run SGD with full training da
#those parameters and compute the validation error. Is the error better than the best you’ve
#Argue!


#a) Factor matrices and bias vectors are all-1s matrices and vectors. (No normalization.)
n = data.get_shape()[0]
m = data.get_shape()[1]
avg = data.sum()/data.count_nonzero()
```

```
# Initial factors are given as a tuple (L, R); R is transposed
init_LR = (np.ones(shape=(n,10)), np.ones(shape=(m,10)))
# Initial biases are a tuple (global_bias, row_bias, col_bias
# Give the true global bias.
init_biases = (avg, np.ones(shape=n), np.ones(shape=m))

# You can change the learning rates and bold driver factor if you want
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, ve
```

```
Epoch   0: error 14435843.09    factor_learning rate = 0.0001
Epoch   1: error 3687950.83     factor_learning rate = 0.0002
Epoch   2: error 803754.06      factor_learning rate = 0.0004
Epoch   3: error 327210.90      factor_learning rate = 0.0008
Epoch   4: error 143830.17      factor_learning rate = 0.0016
Epoch   5: error 63452.22       factor_learning rate = 0.0032
Epoch   6: error 27742.48       factor_learning rate = 0.0064
Epoch   7: error 12648.93       factor_learning rate = 0.01
Epoch   8: error 7855.12        factor_learning rate = 0.01
Epoch   9: error 4951.88        factor_learning rate = 0.01
Epoch  10: error 3806.94        factor_learning rate = 0.01
Epoch  11: error 3460.73        factor_learning rate = 0.01
Epoch  12: error 3345.87        factor_learning rate = 0.01
Epoch  13: error 3127.46        factor_learning rate = 0.01
Epoch  14: error 3115.84        factor_learning rate = 0.01
Epoch  15: error 2973.42        factor_learning rate = 0.01
Epoch  16: error 2926.97        factor_learning rate = 0.01
Epoch  17: error 2875.61        factor_learning rate = 0.01
Epoch  18: error 2770.24        factor_learning rate = 0.01
Epoch  19: error 2828.46        factor_learning rate = 0.0001
Epoch  20: error 2355.66        factor_learning rate = 0.0002
Epoch  21: error 2122.54        factor_learning rate = 0.0004
Epoch  22: error 2001.82        factor_learning rate = 0.0008
Epoch  23: error 1947.30        factor_learning rate = 0.0016
Epoch  24: error 1947.04        factor_learning rate = 0.0032
Epoch  25: error 2009.12        factor_learning rate = 0.0001
Epoch  26: error 1940.80        factor_learning rate = 0.0002
Epoch  27: error 1910.30        factor_learning rate = 0.0004
Epoch  28: error 1899.52        factor_learning rate = 0.0008
Epoch  29: error 1902.07        factor_learning rate = 0.0001
Epoch  30: error 1891.75        factor_learning rate = 0.0002
Epoch  31: error 1889.24        factor_learning rate = 0.0004
Epoch  32: error 1890.69        factor_learning rate = 0.0001
Epoch  33: error 1887.37        factor_learning rate = 0.0002
Epoch  34: error 1886.69        factor_learning rate = 0.0004
Epoch  35: error 1888.54        factor_learning rate = 0.0001
Epoch  36: error 1885.73        factor_learning rate = 0.0002
Epoch  37: error 1885.34        factor_learning rate = 0.0004
Epoch  38: error 1887.79        factor_learning rate = 0.0001
Epoch  39: error 1884.48        factor_learning rate = 0.0002
Epoch  40: error 1884.36        factor_learning rate = 0.0004
Epoch  41: error 1887.01        factor_learning rate = 0.0001
Epoch  42: error 1883.49        factor_learning rate = 0.0002
Epoch  43: error 1883.26        factor_learning rate = 0.0004
Epoch  44: error 1886.21        factor_learning rate = 0.0001
Epoch  45: error 1882.58        factor_learning rate = 0.0002
```

```
Epoch  46: error 1882.29        factor_learning rate = 0.0004
Epoch  47: error 1883.97        factor_learning rate = 0.0001
Epoch  48: error 1881.58        factor_learning rate = 0.0002
Epoch  49: error 1881.37        factor_learning rate = 0.0004
Epoch  50: error 1883.04        factor_learning rate = 0.0001
Epoch  51: error 1880.74        factor_learning rate = 0.0002
Epoch  52: error 1880.64        factor_learning rate = 0.0004
Epoch  53: error 1884.17        factor_learning rate = 0.0001
Epoch  54: error 1879.83        factor_learning rate = 0.0002
Epoch  55: error 1879.67        factor_learning rate = 0.0004
Epoch  56: error 1881.30        factor_learning rate = 0.0001
Epoch  57: error 1878.88        factor_learning rate = 0.0002
Epoch  58: error 1878.64        factor_learning rate = 0.0004
```

```
val_err = squared_error(validation, res=res)
print('Initial solutions as all-1s vectors')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Initial solutions as all-1s vectors
Final training error = 1868.113 Relative error = 0.03338051972473484
    Validation error = 5438.536 Relative error = 14.263972215757216
```

```
#The validation error is far higher than in other approaches
```

```
#b) Factor matrices and bias vectors are all-0s matrices and vectors. (No normalization.)
# Initial factors are given as a tuple (L, R); R is transposed
init_LR = (np.zeros(shape=(n,10)), np.zeros(shape=(m,10)))
# Initial biases are a tuple (global_bias, row_bias, col_bias
# Give the true global bias.
init_biases = (avg, np.zeros(shape=n), np.zeros(shape=m))

# You can change the learning rates and bold driver factor if you want
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, ve
```

```
Epoch  41: error NAN    factor_learning rate = 0.0001
Epoch  42: error NAN    factor_learning rate = 0.0001
Epoch  43: error NAN    factor_learning rate = 0.0001
Epoch  44: error NAN    factor_learning rate = 0.0001
Epoch  45: error NAN    factor_learning rate = 0.0001
Epoch  46: error NAN    factor_learning rate = 0.0001
Epoch  47: error NAN    factor_learning rate = 0.0001
Epoch  48: error NAN    factor_learning rate = 0.0001
Epoch  49: error NAN    factor_learning rate = 0.0001
Epoch  50: error NAN    factor_learning rate = 0.0001
Epoch  51: error NAN    factor_learning rate = 0.0001
Epoch  52: error NAN    factor_learning rate = 0.0001
Epoch  53: error NAN    factor_learning rate = 0.0001
Epoch  54: error NAN    factor_learning rate = 0.0001
Epoch  55: error NAN    factor_learning rate = 0.0001
Epoch  56: error NAN    factor_learning rate = 0.0001
Epoch  57: error NAN    factor_learning rate = 0.0001
```

```
Epoch  57: error NAN    factor_learning rate = 0.0001
Epoch  58: error NAN    factor_learning rate = 0.0001
Epoch  59: error NAN    factor_learning rate = 0.0001
Epoch  60: error NAN    factor_learning rate = 0.0001
Epoch  61: error NAN    factor_learning rate = 0.0001
Epoch  62: error NAN    factor_learning rate = 0.0001
Epoch  63: error NAN    factor_learning rate = 0.0001
Epoch  64: error NAN    factor_learning rate = 0.0001
Epoch  65: error NAN    factor_learning rate = 0.0001
Epoch  66: error NAN    factor_learning rate = 0.0001
Epoch  67: error NAN    factor_learning rate = 0.0001

Epoch  68: error NAN    factor_learning rate = 0.0001
Epoch  69: error NAN    factor_learning rate = 0.0001
Epoch  70: error NAN    factor_learning rate = 0.0001
Epoch  71: error NAN    factor_learning rate = 0.0001
Epoch  72: error NAN    factor_learning rate = 0.0001
Epoch  73: error NAN    factor_learning rate = 0.0001
Epoch  74: error NAN    factor_learning rate = 0.0001
Epoch  75: error NAN    factor_learning rate = 0.0001
Epoch  76: error NAN    factor_learning rate = 0.0001
Epoch  77: error NAN    factor_learning rate = 0.0001
Epoch  78: error NAN    factor_learning rate = 0.0001
Epoch  79: error NAN    factor_learning rate = 0.0001
Epoch  80: error NAN    factor_learning rate = 0.0001
Epoch  81: error NAN    factor_learning rate = 0.0001
Epoch  82: error NAN    factor_learning rate = 0.0001
Epoch  83: error NAN    factor_learning rate = 0.0001
Epoch  84: error NAN    factor_learning rate = 0.0001
Epoch  85: error NAN    factor_learning rate = 0.0001
Epoch  86: error NAN    factor_learning rate = 0.0001
Epoch  87: error NAN    factor_learning rate = 0.0001
Epoch  88: error NAN    factor_learning rate = 0.0001
Epoch  89: error NAN    factor_learning rate = 0.0001
Epoch  90: error NAN    factor_learning rate = 0.0001
Epoch  91: error NAN    factor_learning rate = 0.0001
Epoch  92: error NAN    factor_learning rate = 0.0001
Epoch  93: error NAN    factor_learning rate = 0.0001
Epoch  94: error NAN    factor_learning rate = 0.0001
Epoch  95: error NAN    factor_learning rate = 0.0001
Epoch  96: error NAN    factor_learning rate = 0.0001
Epoch  97: error NAN    factor_learning rate = 0.0001
Epoch  98: error NAN    factor_learning rate = 0.0001
Epoch  99: error NAN    factor_learning rate = 0.0001
```

```
val_err = squared_error(validation, res=res)
print('Initial solutions as all-1s vectors')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Initial solutions as all-1s vectors
Final training error =   nan    Relative error = nan
    Validation error =   nan    Relative error = nan
```

```
#Thus we see that in the zero case we do not have any solution
```

```
#np.random.normal(0, 1, (3, 3))
# Initial factors are given as a tuple (L, R); R is transposed
init_LR = (np.random.normal(0, 1, (n, 10)), np.random.normal(0, 1, (m, 10)))
# Initial biases are a tuple (global_bias, row_bias, col_bias
# Give the true global bias.
#init_biases = (avg, np.random.normal(0, 1, (n, 10)), np.random.normal(0, 1, (m, 10)))
```

```
init_LR
```

```
(array([[-1.17425096,  0.19918981, -0.60222315, ...,  0.94614608,
          0.72020795, -0.18975039],
        [ 0.12938449,  2.18551624,  1.17862941, ..., -1.60806639,
         -0.12300904, -0.36229804],
        [-1.12518146,  0.20165542, -0.70840878, ..., -0.99945708,
          0.18227732,  0.28327222],
        ...,
        [-0.95893046, -0.5124027 , -0.95621689, ...,  0.33078595,
          0.21427449, -0.35070937],
        [ 1.05471018, -1.59608667,  1.17445045, ...,  0.33615226,
          2.22510379, -0.94514869],
        [-0.5575909 ,  0.30100682, -2.3460474 , ...,  0.22825616,
         -1.41118481, -1.94459559]]),
 array([[-1.05958836, -0.22664738, -0.96912525, ..., -2.11702163,
         -1.40406968, -1.03730845],
        [-0.97952757,  1.242295  , -0.54230601, ..., -0.63007592,
          0.15228372,  0.97380255],
        [-0.33820309,  0.31873166,  0.1123456 , ..., -0.30794351,
         -0.75334118, -1.86813542],
        ...,
        [ 0.44960496,  0.24646594,  0.66482954, ..., -0.21475797,
         -0.69779164,  0.72185682],
        [-0.36235038,  0.39441899, -0.51329037, ...,  0.57037241,
         -0.70571621, -0.28587571],
        [ 0.00957147, -0.37847174,  0.22947165, ..., -0.30678886,
          1.90335051,  1.95844878]]))
```

```
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, ve
```

```
Epoch  41: error NAN    factor_learning rate = 0.0001
Epoch  42: error NAN    factor_learning rate = 0.0001
Epoch  43: error NAN    factor_learning rate = 0.0001
Epoch  44: error NAN    factor_learning rate = 0.0001
Epoch  45: error NAN    factor_learning rate = 0.0001
Epoch  46: error NAN    factor_learning rate = 0.0001
Epoch  47: error NAN    factor_learning rate = 0.0001
Epoch  48: error NAN    factor_learning rate = 0.0001
```

```
Epoch  49: error NAN    factor_learning rate = 0.0001
Epoch  50: error NAN    factor_learning rate = 0.0001
Epoch  51: error NAN    factor_learning rate = 0.0001
Epoch  52: error NAN    factor_learning rate = 0.0001
Epoch  53: error NAN    factor_learning rate = 0.0001
Epoch  54: error NAN    factor_learning rate = 0.0001
Epoch  55: error NAN    factor_learning rate = 0.0001
Epoch  56: error NAN    factor_learning rate = 0.0001
Epoch  57: error NAN    factor_learning rate = 0.0001
Epoch  58: error NAN    factor_learning rate = 0.0001
Epoch  59: error NAN    factor_learning rate = 0.0001

Epoch  60: error NAN    factor_learning rate = 0.0001
Epoch  61: error NAN    factor_learning rate = 0.0001
Epoch  62: error NAN    factor_learning rate = 0.0001
Epoch  63: error NAN    factor_learning rate = 0.0001
Epoch  64: error NAN    factor_learning rate = 0.0001
Epoch  65: error NAN    factor_learning rate = 0.0001
Epoch  66: error NAN    factor_learning rate = 0.0001
Epoch  67: error NAN    factor_learning rate = 0.0001
Epoch  68: error NAN    factor_learning rate = 0.0001
Epoch  69: error NAN    factor_learning rate = 0.0001
Epoch  70: error NAN    factor_learning rate = 0.0001
Epoch  71: error NAN    factor_learning rate = 0.0001
Epoch  72: error NAN    factor_learning rate = 0.0001
Epoch  73: error NAN    factor_learning rate = 0.0001
Epoch  74: error NAN    factor_learning rate = 0.0001
Epoch  75: error NAN    factor_learning rate = 0.0001
Epoch  76: error NAN    factor_learning rate = 0.0001
Epoch  77: error NAN    factor_learning rate = 0.0001
Epoch  78: error NAN    factor_learning rate = 0.0001
Epoch  79: error NAN    factor_learning rate = 0.0001
Epoch  80: error NAN    factor_learning rate = 0.0001
Epoch  81: error NAN    factor_learning rate = 0.0001
Epoch  82: error NAN    factor_learning rate = 0.0001
Epoch  83: error NAN    factor_learning rate = 0.0001
Epoch  84: error NAN    factor_learning rate = 0.0001
Epoch  85: error NAN    factor_learning rate = 0.0001
Epoch  86: error NAN    factor_learning rate = 0.0001
Epoch  87: error NAN    factor_learning rate = 0.0001
Epoch  88: error NAN    factor_learning rate = 0.0001
Epoch  89: error NAN    factor_learning rate = 0.0001
Epoch  90: error NAN    factor_learning rate = 0.0001
Epoch  91: error NAN    factor_learning rate = 0.0001
Epoch  92: error NAN    factor_learning rate = 0.0001
Epoch  93: error NAN    factor_learning rate = 0.0001
Epoch  94: error NAN    factor_learning rate = 0.0001
Epoch  95: error NAN    factor_learning rate = 0.0001
Epoch  96: error NAN    factor_learning rate = 0.0001
Epoch  97: error NAN    factor_learning rate = 0.0001
Epoch  98: error NAN    factor_learning rate = 0.0001
Epoch  99: error NAN    factor_learning rate = 0.0001
```

```
#So we see that although we are able to generate L R initial solutio matrices, the final
# solution is a not a number
```

```
#Factor matrices are random integer matrices with values from 0 to 6, normalized.
L= np.random.randint(7, size=(n,10)) #Create a random matrix of size n by 10 and having value
R = np.random.randint(7, size=(m,10))
print("Original Matrix:")
print(L,R)
Lmax, Lmin = L.max(), L.min()
Rmax, Rmin = R.max(), R.min()
L = (L - Lmin)/(Lmax - Lmin)
R = (R - Rmin) / (Rmax -  Rmin)
print("After normalization:")
print(L,R)
```

```
Original Matrix:
[[6 2 5 ... 3 3 5]
 [0 3 4 ... 2 0 1]
 [6 3 1 ... 3 0 4]
 ...
 [5 2 3 ... 2 4 0]
 [1 3 0 ... 5 1 0]
 [6 2 5 ... 0 3 2]] [[2 5 4 ... 0 1 1]
 [5 0 4 ... 3 1 0]
 [0 4 1 ... 3 3 6]
 ...
 [5 6 4 ... 6 0 3]
 [3 2 5 ... 1 1 4]
 [4 4 6 ... 0 3 5]]
After normalization:
[[1.         0.33333333 0.83333333 ... 0.5        0.5        0.83333333]
 [0.         0.5        0.66666667 ... 0.33333333 0.         0.16666667]
 [1.         0.5        0.16666667 ... 0.5        0.         0.66666667]
 ...
 [0.83333333 0.33333333 0.5        ... 0.33333333 0.66666667 0.        ]
 [0.16666667 0.5        0.         ... 0.83333333 0.16666667 0.        ]
 [1.         0.33333333 0.83333333 ... 0.         0.5        0.33333333]] [[0.33333333
 [0.83333333 0.         0.66666667 ... 0.5        0.16666667 0.        ]
 [0.         0.66666667 0.16666667 ... 0.5        0.5        1.        ]
 ...
 [0.83333333 1.         0.66666667 ... 1.         0.         0.5       ]
 [0.5        0.33333333 0.83333333 ... 0.16666667 0.16666667 0.66666667]
 [0.66666667 0.66666667 1.         ... 0.         0.5        0.83333333]]
```

```
# Initial factors are given as a tuple (L, R); R is transposed
init_LR = (L, R)
#
```

```
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, ve
```

```
Epoch  41: error 1280.91        factor_learning rate = 0.0008
Epoch  42: error 1282.83        factor_learning rate = 0.0001
Epoch  43: error 1280.21        factor_learning rate = 0.0002
Epoch  44: error 1278.92        factor_learning rate = 0.0004
Epoch  45: error 1278.72        factor_learning rate = 0.0008
```

```
Epoch  46: error 1280.45        factor_learning rate = 0.0001
Epoch  47: error 1278.12        factor_learning rate = 0.0002
Epoch  48: error 1276.82        factor_learning rate = 0.0004
Epoch  49: error 1276.88        factor_learning rate = 0.0001
Epoch  50: error 1275.91        factor_learning rate = 0.0002
Epoch  51: error 1275.43        factor_learning rate = 0.0004
Epoch  52: error 1275.53        factor_learning rate = 0.0001
Epoch  53: error 1274.84        factor_learning rate = 0.0002
Epoch  54: error 1274.48        factor_learning rate = 0.0004
Epoch  55: error 1274.74        factor_learning rate = 0.0001
Epoch  56: error 1273.97        factor_learning rate = 0.0002
Epoch  57: error 1273.60        factor_learning rate = 0.0004
Epoch  58: error 1274.03        factor_learning rate = 0.0001
Epoch  59: error 1273.11        factor_learning rate = 0.0002
Epoch  60: error 1272.72        factor_learning rate = 0.0004

Epoch  61: error 1273.40        factor_learning rate = 0.0001
Epoch  62: error 1272.32        factor_learning rate = 0.0002
Epoch  63: error 1271.91        factor_learning rate = 0.0004
Epoch  64: error 1272.21        factor_learning rate = 0.0001
Epoch  65: error 1271.43        factor_learning rate = 0.0002
Epoch  66: error 1271.02        factor_learning rate = 0.0004
Epoch  67: error 1271.39        factor_learning rate = 0.0001
Epoch  68: error 1270.61        factor_learning rate = 0.0002
Epoch  69: error 1270.21        factor_learning rate = 0.0004
Epoch  70: error 1270.37        factor_learning rate = 0.0001
Epoch  71: error 1269.76        factor_learning rate = 0.0002
Epoch  72: error 1269.39        factor_learning rate = 0.0004
Epoch  73: error 1269.58        factor_learning rate = 0.0001
Epoch  74: error 1269.00        factor_learning rate = 0.0002
Epoch  75: error 1268.67        factor_learning rate = 0.0004
Epoch  76: error 1268.86        factor_learning rate = 0.0001
Epoch  77: error 1268.21        factor_learning rate = 0.0002
Epoch  78: error 1267.82        factor_learning rate = 0.0004
Epoch  79: error 1268.14        factor_learning rate = 0.0001
Epoch  80: error 1267.45        factor_learning rate = 0.0002
Epoch  81: error 1267.11        factor_learning rate = 0.0004
Epoch  82: error 1267.33        factor_learning rate = 0.0001
Epoch  83: error 1266.69        factor_learning rate = 0.0002
Epoch  84: error 1266.34        factor_learning rate = 0.0004
Epoch  85: error 1266.95        factor_learning rate = 0.0001
Epoch  86: error 1266.01        factor_learning rate = 0.0002
Epoch  87: error 1265.60        factor_learning rate = 0.0004
Epoch  88: error 1265.78        factor_learning rate = 0.0001
Epoch  89: error 1265.19        factor_learning rate = 0.0002
Epoch  90: error 1264.86        factor_learning rate = 0.0004
Epoch  91: error 1265.12        factor_learning rate = 0.0001
Epoch  92: error 1264.45        factor_learning rate = 0.0002
Epoch  93: error 1264.19        factor_learning rate = 0.0004
Epoch  94: error 1264.43        factor_learning rate = 0.0001
Epoch  95: error 1263.78        factor_learning rate = 0.0002
Epoch  96: error 1263.44        factor_learning rate = 0.0004
Epoch  97: error 1263.84        factor_learning rate = 0.0001
Epoch  98: error 1263.07        factor_learning rate = 0.0002
Epoch  99: error 1262.73        factor_learning rate = 0.0004
```

```
val_err = squared_error(validation, res=res)
```

```
print('Initial solutions as all-1s vectors')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Initial solutions as all-1s vectors
Final training error = 1262.930 Relative error = 0.022566763306068317
    Validation error = 289.776  Relative error = 0.7600123874348135
```

```
learning_rates = [1e-5,1e-4,1e-3]

start = time.process_time()
err = []
for rate in learning_rates:
    err.append( cv(data, folds=5, k=10, bold_driver=2, factor_learning_rate=rate, bias_learni
print('Rate\terr')
for i in range(len(learning_rates)):
    print(f'{learning_rates[i]}\t{err[i]}')
print(time.process_time() - start)
```

```
Fold 1/5
        Error = 2522.5388802632397
Fold 2/5
        Error = 2489.1905454788107
Fold 3/5
        Error = 2504.4814770818143
Fold 4/5
        Error = 2603.2898195405733
Fold 5/5
        Error = 2436.963867852763
Fold 1/5
        Error = 2606.133391562352
Fold 2/5
        Error = 2515.1371980337613
Fold 3/5
        Error = 2425.8187872819076
Fold 4/5
        Error = 2356.2555134840804
Fold 5/5
        Error = 2641.0567237722935
Fold 1/5
        Error = 2543.449346384727
Fold 2/5
        Error = 2723.9636613255407
Fold 3/5
        Error = 2485.382732583863
Fold 4/5
        Error = 2605.006823579612
Fold 5/5
        Error = 2428.441760061842
Rate    err
1e-05   2511.2929180434403
```

```
0.0001  2508.880322826879
0.001   2557.248864787117
2956.489249872


#The time taken in code above is 2956.48 SECs.The best parameter is for
# learning rate 0.0001 for which the average error is the lowest.
#data = [data]
#k = [10]

#from sklearn.model_selection import GridSearchCV
#param_grid = {'factor_regu':factor_regu, 'bias_regu': bias_regu,'k':k,'factor_learning_rates



regularization_coeff = [1e-6,1e-5,1e-4]

start = time.process_time()
err = []
for coeff in regularization_coeff:
    err.append( cv(data, folds=5, k=10, bold_driver=2, factor_regu=coeff, factor_learning_rat
print('Rate\terr')
for i in range(len(regularization_coeff)):
    print(f'{regularization_coeff[i]}\t{err[i]}')
print(time.process_time() - start)

 Fold 1/5
        Error = 2743.658330049852
 Fold 2/5
        Error = 2961.9016783817465
 Fold 3/5
        Error = 2748.7784574101943
 Fold 4/5
        Error = 2969.309548115035
 Fold 5/5
        Error = 2633.911537871388
 Fold 1/5
        Error = 2935.336981518091
 Fold 2/5
        Error = 2720.21982641188
 Fold 3/5
        Error = 2864.97255927242
 Fold 4/5
        Error = 2854.3106561156783
 Fold 5/5
        Error = 2835.735287572032
 Fold 1/5
        Error = 2574.718204069597
 Fold 2/5
        Error = 2654.184835630396
 Fold 3/5
        Error = 2780.8703271832537
 Fold 4/5
        Error = 2764.8582985545318
 Fold 5/5
```

```
        Error = 2651.355910728593
Rate    err
1e-06   2811.511910365643
1e-05   2842.1150621780203
0.0001  2685.1975152332743
2495.0275192910003


#We get the lowest error for REG COEFF 0.0001 . Total time taken was 2495 secs.


#Now doing SGD with regularization L2-Regularization, bold driver of 2, factor learning rate
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, fa

    Epoch   0: error 628968.60      factor_learning rate = 0.0001
    Epoch   1: error 270501.48      factor_learning rate = 0.0002
    Epoch   2: error 122283.66      factor_learning rate = 0.0004
    Epoch   3: error 64240.54       factor_learning rate = 0.0008
    Epoch   4: error 37100.29       factor_learning rate = 0.0016
    Epoch   5: error 21236.32       factor_learning rate = 0.0032
    Epoch   6: error 11513.69       factor_learning rate = 0.0064
    Epoch   7: error 6153.06        factor_learning rate = 0.01
    Epoch   8: error 3830.75        factor_learning rate = 0.01
    Epoch   9: error 2860.91        factor_learning rate = 0.01
    Epoch  10: error 2420.47        factor_learning rate = 0.01
    Epoch  11: error 2190.95        factor_learning rate = 0.01
    Epoch  12: error 2031.73        factor_learning rate = 0.01
    Epoch  13: error 1925.51        factor_learning rate = 0.01
    Epoch  14: error 1833.90        factor_learning rate = 0.01
    Epoch  15: error 1790.35        factor_learning rate = 0.01
    Epoch  16: error 1753.30        factor_learning rate = 0.01
    Epoch  17: error 1692.71        factor_learning rate = 0.01
    Epoch  18: error 1692.06        factor_learning rate = 0.01
    Epoch  19: error 1670.95        factor_learning rate = 0.01
    Epoch  20: error 1622.03        factor_learning rate = 0.01
    Epoch  21: error 1638.80        factor_learning rate = 0.0001
    Epoch  22: error 1516.26        factor_learning rate = 0.0002
    Epoch  23: error 1424.12        factor_learning rate = 0.0004
    Epoch  24: error 1367.12        factor_learning rate = 0.0008
    Epoch  25: error 1338.52        factor_learning rate = 0.0016
    Epoch  26: error 1333.34        factor_learning rate = 0.0032
    Epoch  27: error 1355.56        factor_learning rate = 0.0001
    Epoch  28: error 1333.34        factor_learning rate = 0.0002
    Epoch  29: error 1317.81        factor_learning rate = 0.0004
    Epoch  30: error 1309.73        factor_learning rate = 0.0008
    Epoch  31: error 1308.74        factor_learning rate = 0.0016
    Epoch  32: error 1313.39        factor_learning rate = 0.0001
    Epoch  33: error 1306.68        factor_learning rate = 0.0002
    Epoch  34: error 1302.11        factor_learning rate = 0.0004
    Epoch  35: error 1300.16        factor_learning rate = 0.0008
    Epoch  36: error 1301.27        factor_learning rate = 0.0001
    Epoch  37: error 1298.98        factor_learning rate = 0.0002
    Epoch  38: error 1297.56        factor_learning rate = 0.0004
    Epoch  39: error 1297.17        factor_learning rate = 0.0008
    Epoch  40: error 1300.09        factor_learning rate = 0.0001
    Epoch  41: error 1296.70        factor_learning rate = 0.0002
    Epoch  42: error 1295.04        factor_learning rate = 0.0004
```

```
Epoch  43: error 1295.07        factor_learning rate = 0.0001
Epoch  44: error 1294.03        factor_learning rate = 0.0002
Epoch  45: error 1293.50        factor_learning rate = 0.0004
Epoch  46: error 1293.77        factor_learning rate = 0.0001
Epoch  47: error 1292.93        factor_learning rate = 0.0002
Epoch  48: error 1292.46        factor_learning rate = 0.0004
Epoch  49: error 1292.73        factor_learning rate = 0.0001
Epoch  50: error 1291.84        factor_learning rate = 0.0002
Epoch  51: error 1291.45        factor_learning rate = 0.0004
Epoch  52: error 1291.72        factor_learning rate = 0.0001
Epoch  53: error 1290.90        factor_learning rate = 0.0002
Epoch  54: error 1290.47        factor_learning rate = 0.0004
Epoch  55: error 1290.61        factor_learning rate = 0.0001
Epoch  56: error 1289.90        factor_learning rate = 0.0002
Epoch  57: error 1289.52        factor_learning rate = 0.0004
Epoch  58: error 1289.59        factor_learning rate = 0.0001
```

```
val_err = squared_error(validation, res=res)
print('Initial solutions as all-1s vectors')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Initial solutions as all-1s vectors
Final training error = 1277.875 Relative error = 0.02283381383939651
    Validation error = 233.456  Relative error = 0.6122996982054809
```

```
#This result is lower than the best we got earlier, because the bold driver factor used there
#So we need to work for that, and see the best parameters corresponding to it.
```

```
learning_rates = [1e-5,1e-4,1e-3]

start = time.process_time()
err = []
for rate in learning_rates:
    err.append( cv(data, folds=5, k=10, bold_driver=3, factor_learning_rate=rate, bias_learni
print('Rate\terr')
for i in range(len(learning_rates)):
    print(f'{learning_rates[i]}\t{err[i]}')
print(time.process_time() - start)
```

```
Fold 1/5
        Error = 2722.5288581181385
Fold 2/5
        Error = 2548.302374176806
Fold 3/5
        Error = 2531.347277367209
Fold 4/5
        Error = 2540.238496009842
Fold 5/5
```

```
        Error = 2748.2896215624846
Fold 1/5
        Error = 2633.391336263878
Fold 2/5
        Error = 2650.69334040362
Fold 3/5
        Error = 2542.0456009362965
Fold 4/5
        Error = 2656.0247062022363
Fold 5/5
        Error = 2449.8824910680437
Fold 1/5
        Error = 2312.145161770416
Fold 2/5
        Error = 2552.8925029346537
Fold 3/5
        Error = 2418.1880469407265
Fold 4/5
        Error = 2353.8519240647397
Fold 5/5
        Error = 2659.890620634777
Rate    err
1e-05   2618.141325446896
0.0001  2586.4074949748147
0.001   2459.3936512690625
2978.851456674999
```

```
#The error is minimum for 0.001 value. So I use this to get the next parameter optimum value


regularization_coeff = [1e-6,1e-5,1e-4]

start = time.process_time()
err = []
for coeff in regularization_coeff:
    err.append( cv(data, folds=5, k=10, bold_driver=3, factor_regu=coeff, factor_learning_rat
print('Rate\terr')
for i in range(len(regularization_coeff)):
    print(f'{regularization_coeff[i]}\t{err[i]}')
print(time.process_time() - start)
```

```
Fold 1/5
        Error = 2801.487380575274
Fold 2/5
        Error = 2820.085083191071
Fold 3/5
        Error = 2947.297234022993
Fold 4/5
        Error = 3017.180038227253
Fold 5/5
        Error = 3011.9426937494577
Fold 1/5
        Error = 2956.9260771417444
Fold 2/5
        Error = 2868.551920255637
```

```
Fold 3/5
        Error = 2764.358354477345
Fold 4/5
        Error = 3131.893527184753
Fold 5/5
        Error = 2957.233345347008
Fold 1/5
        Error = 2743.44515320909
Fold 2/5
        Error = 2595.991925187045
Fold 3/5
        Error = 2684.655141344161
Fold 4/5
        Error = 2688.8406079075107
Fold 5/5
        Error = 2750.029991512247
Rate    err
1e-06   2919.5984859532095
1e-05   2935.7926448812973
0.0001  2692.5925638320105
2531.3175626440006


#So it took 2531 seconds to complete and the lowest error came for 0.0001 as regularization c


#Now doing SGD with regularization L2-Regularization, bold driver of 3, factor learning rate
res = sgd(data, 10, bold_driver=3, factor_learning_rate=0.001, bias_learning_rate=0.001, fact
Epoch  27: error 1291.74        factor_learning rate = 0.003
Epoch  28: error 1322.85        factor_learning rate = 0.001
Epoch  29: error 1286.62        factor_learning rate = 0.003
Epoch  30: error 1317.55        factor_learning rate = 0.001
Epoch  31: error 1283.19        factor_learning rate = 0.003
Epoch  32: error 1323.15        factor_learning rate = 0.001
Epoch  33: error 1278.04        factor_learning rate = 0.003
Epoch  34: error 1315.06        factor_learning rate = 0.001
Epoch  35: error 1274.85        factor_learning rate = 0.003
Epoch  36: error 1309.03        factor_learning rate = 0.001
Epoch  37: error 1272.15        factor_learning rate = 0.003
Epoch  38: error 1305.50        factor_learning rate = 0.001
Epoch  39: error 1269.99        factor_learning rate = 0.003
Epoch  40: error 1302.88        factor_learning rate = 0.001
Epoch  41: error 1265.51        factor_learning rate = 0.003
Epoch  42: error 1295.56        factor_learning rate = 0.001
Epoch  43: error 1262.78        factor_learning rate = 0.003
Epoch  44: error 1294.25        factor_learning rate = 0.001
Epoch  45: error 1259.57        factor_learning rate = 0.003
Epoch  46: error 1293.88        factor_learning rate = 0.001
Epoch  47: error 1257.07        factor_learning rate = 0.003
Epoch  48: error 1292.61        factor_learning rate = 0.001
Epoch  49: error 1254.53        factor_learning rate = 0.003
Epoch  50: error 1287.86        factor_learning rate = 0.001
Epoch  51: error 1251.21        factor_learning rate = 0.003
Epoch  52: error 1286.20        factor_learning rate = 0.001
Epoch  53: error 1249.37        factor_learning rate = 0.003
Epoch  54: error 1277.64        factor_learning rate = 0.001
Epoch  55: error 1246.74        factor_learning rate = 0.003
```

```
Epoch  56: error 1280.83        factor_learning rate = 0.001
Epoch  57: error 1245.58        factor_learning rate = 0.003
Epoch  58: error 1281.63        factor_learning rate = 0.001
Epoch  59: error 1241.89        factor_learning rate = 0.003
Epoch  60: error 1277.92        factor_learning rate = 0.001
Epoch  61: error 1239.59        factor_learning rate = 0.003
Epoch  62: error 1277.06        factor_learning rate = 0.001
Epoch  63: error 1238.46        factor_learning rate = 0.003
Epoch  64: error 1272.13        factor_learning rate = 0.001
Epoch  65: error 1236.15        factor_learning rate = 0.003
Epoch  66: error 1269.85        factor_learning rate = 0.001
Epoch  67: error 1233.26        factor_learning rate = 0.003
Epoch  68: error 1265.91        factor_learning rate = 0.001
Epoch  69: error 1232.96        factor_learning rate = 0.003
Epoch  70: error 1265.74        factor_learning rate = 0.001
Epoch  71: error 1229.62        factor_learning rate = 0.003
Epoch  72: error 1262.56        factor_learning rate = 0.001
Epoch  73: error 1227.98        factor_learning rate = 0.003
Epoch  74: error 1266.34        factor_learning rate = 0.001
Epoch  75: error 1226.77        factor_learning rate = 0.003
Epoch  76: error 1261.01        factor_learning rate = 0.001
Epoch  77: error 1223.48        factor_learning rate = 0.003
Epoch  78: error 1257.90        factor_learning rate = 0.001
Epoch  79: error 1224.11        factor_learning rate = 0.003
Epoch  80: error 1257.64        factor_learning rate = 0.001
Epoch  81: error 1221.53        factor_learning rate = 0.003
Epoch  82: error 1255.34        factor_learning rate = 0.001
Epoch  83: error 1218.77        factor_learning rate = 0.003
Epoch  84: error 1253.05        factor_learning rate = 0.001

Epoch  85: error 1218.98        factor_learning rate = 0.003
```

```
val_err = squared_error(validation, res=res)
print('Initial solutions as all-1s vectors')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Initial solutions as all-1s vectors
Final training error = 1240.353 Relative error = 0.02216334060456452
    Validation error = 240.222  Relative error = 0.6300442533451417
```

```
#However, this validation score is greater than 188 what we obtained earlier:

#res = sgd(data, 10, bold_driver=3, factor_learning_rate=0.01)
#So we try those learning rates as well


learning_rates = [0.01,0.1,1e-5,1e-4,1e-3]

start = time.process_time()
err = []
for rate in learning_rates:
```

```
    err.append( cv(data, folds=5, k=10, bold_driver=3, factor_learning_rate=rate, bias_learni
print('Rate\terr')
for i in range(len(learning_rates)):
    print(f'{learning_rates[i]}\t{err[i]}')
print(time.process_time() - start)
```

```
Error is 5.727265654196042Se+20, you should probably use slower learning rate
Error is 1.3163441052078707e+294, you should probably use slower learning rate
Error is 9.416417444054082e+224, you should probably use slower learning rate
Error is 3.2342781242768822e+227, you should probably use slower learning rate
Error is 3.586297209233935e+124, you should probably use slower learning rate
Error is 1.7155035737386879e+47, you should probably use slower learning rate
Error is 4.55041345993541e+20, you should probably use slower learning rate

Error is 5503149314228485.0, you should probably use slower learning rate
Error is 1827605033038075.5, you should probably use slower learning rate
Error is 1.472385389577297e+299, you should probably use slower learning rate
Error is 734219046760867.5, you should probably use slower learning rate
Error is 1.249741888653658e+164, you should probably use slower learning rate
Error is 322695.57768617227, you should probably use slower learning rate
Error is 2.4200710706305106e+202, you should probably use slower learning rate
Error is 1.101113170840056e+131, you should probably use slower learning rate
Error is 3.623921722199731e+144, you should probably use slower learning rate
Error is 402062.1226076005, you should probably use slower learning rate
Error is 1.8596667068632105e+202, you should probably use slower learning rate
Error is 8.69015602505699e+201, you should probably use slower learning rate
Error is 6.716854006823664e+201, you should probably use slower learning rate
        Error = nan
Fold 1/5
        Error = 2467.853681477283
Fold 2/5
        Error = 2527.732967914114
Fold 3/5
        Error = 2360.838935521543
Fold 4/5
        Error = 2622.725077740915
Fold 5/5
        Error = 2617.765652226571
Fold 1/5
        Error = 2464.9599944767897
Fold 2/5
        Error = 2639.924315673732
Fold 3/5
        Error = 2386.4413731964187
Fold 4/5
        Error = 2471.631361123579
Fold 5/5
        Error = 2555.01243837131
Fold 1/5
        Error = 2621.1776178656646
Fold 2/5
        Error = 2396.846637590643
Fold 3/5
        Error = 2813.4069896910764
Fold 4/5
        Error = 2439.9967283026454
Fold 5/5
        Error = 2362.993120233712
```

```
Rate    err
0.01    2315.183788809335
0.1     nan
1e-05   2519.383262976085
0.0001  2503.5938965683663
0.001   2526.884218736748
5002.484504351001
```

```
#Thus, now we can clearly see that the least error is for step value 0.01
#The total time taken was 5002 seconds


#Finding the regularizaiton hyper-parameter which is optimal for step value 0.01
regularization_coeff = [1e-6,1e-5,1e-4]

start = time.process_time()
err = []
for coeff in regularization_coeff:
    err.append( cv(data, folds=5, k=10, bold_driver=3, factor_regu=coeff, factor_learning_rat
print('Rate\terr')
for i in range(len(regularization_coeff)):
    print(f'{regularization_coeff[i]}\t{err[i]}')
print(time.process_time() - start)
```

```
 Fold 1/5
        Error = 2723.9522335410074
 Fold 2/5
        Error = 2479.8903454863907
 Fold 3/5
        Error = 2701.477978900012
 Fold 4/5
        Error = 2620.651166682969
 Fold 5/5
        Error = 2770.3373583415414
 Fold 1/5
        Error = 2896.6547637955846
 Fold 2/5
        Error = 2697.6782914859987
 Fold 3/5
        Error = 2697.3500426837727
 Fold 4/5
        Error = 2666.9147475426907
 Fold 5/5
        Error = 2700.9933094694256
 Fold 1/5
        Error = 2578.160316807287
 Fold 2/5
        Error = 2634.4587197658807
 Fold 3/5
        Error = 2378.3291158020434
 Fold 4/5
        Error = 2418.9902713865345
 Fold 5/5
        Error = 2738.4165664106777
```

```
Rate    err
1e-06   2659.261816590384
1e-05   2731.918230995495
0.0001  2549.6709980344845
2285.068527286


#So the least error was found for coeff 0.0001 and time taken is 2285 secs


#Now applying the optimal hyper-parameters
#Now doing SGD with regularization L2-Regularization, bold driver of 2, factor learning rate
res = sgd(data, 10, bold_driver=3, factor_learning_rate=0.01, bias_learning_rate=0.01, factor

Epoch  42: error 1407.03        factor_learning rate = 0.01
Epoch  43: error 1390.47        factor_learning rate = 0.01
Epoch  44: error 1386.31        factor_learning rate = 0.01
Epoch  45: error 1366.78        factor_learning rate = 0.01
Epoch  46: error 1390.11        factor_learning rate = 0.01
Epoch  47: error 1380.69        factor_learning rate = 0.01
Epoch  48: error 1406.17        factor_learning rate = 0.01
Epoch  49: error 1369.86        factor_learning rate = 0.01
Epoch  50: error 1371.19        factor_learning rate = 0.01
Epoch  51: error 1386.71        factor_learning rate = 0.01
Epoch  52: error 1355.18        factor_learning rate = 0.01
Epoch  53: error 1351.27        factor_learning rate = 0.01
Epoch  54: error 1362.63        factor_learning rate = 0.01
Epoch  55: error 1357.35        factor_learning rate = 0.01
Epoch  56: error 1349.00        factor_learning rate = 0.01
Epoch  57: error 1311.33        factor_learning rate = 0.01
Epoch  58: error 1336.14        factor_learning rate = 0.01
Epoch  59: error 1348.36        factor_learning rate = 0.01
Epoch  60: error 1346.03        factor_learning rate = 0.01
Epoch  61: error 1330.36        factor_learning rate = 0.01
Epoch  62: error 1352.84        factor_learning rate = 0.01
Epoch  63: error 1321.40        factor_learning rate = 0.01
Epoch  64: error 1355.92        factor_learning rate = 0.01
Epoch  65: error 1325.09        factor_learning rate = 0.01
Epoch  66: error 1331.01        factor_learning rate = 0.01
Epoch  67: error 1318.82        factor_learning rate = 0.01
Epoch  68: error 1334.22        factor_learning rate = 0.01
Epoch  69: error 1307.41        factor_learning rate = 0.01
Epoch  70: error 1318.32        factor_learning rate = 0.01
Epoch  71: error 1325.00        factor_learning rate = 0.01
Epoch  72: error 1314.91        factor_learning rate = 0.01
Epoch  73: error 1304.17        factor_learning rate = 0.01
Epoch  74: error 1300.56        factor_learning rate = 0.01
Epoch  75: error 1293.69        factor_learning rate = 0.01
Epoch  76: error 1306.56        factor_learning rate = 0.01
Epoch  77: error 1282.80        factor_learning rate = 0.01
Epoch  78: error 1295.34        factor_learning rate = 0.01
Epoch  79: error 1301.15        factor_learning rate = 0.01
Epoch  80: error 1288.67        factor_learning rate = 0.01
Epoch  81: error 1284.35        factor_learning rate = 0.01
Epoch  82: error 1279.61        factor_learning rate = 0.01
Epoch  83: error 1293.37        factor_learning rate = 0.01
Epoch  84: error 1279.02        factor_learning rate = 0.01
```

```
Epoch  85: error 1286.47        factor_learning rate = 0.01
Epoch  86: error 1265.21        factor_learning rate = 0.01
Epoch  87: error 1297.84        factor_learning rate = 0.01
Epoch  88: error 1277.86        factor_learning rate = 0.01
Epoch  89: error 1279.30        factor_learning rate = 0.01
Epoch  90: error 1269.20        factor_learning rate = 0.01
Epoch  91: error 1246.06        factor_learning rate = 0.01
Epoch  92: error 1247.97        factor_learning rate = 0.01
Epoch  93: error 1261.16        factor_learning rate = 0.01
Epoch  94: error 1255.30        factor_learning rate = 0.01
Epoch  95: error 1266.86        factor_learning rate = 0.01
Epoch  96: error 1241.27        factor_learning rate = 0.01
Epoch  97: error 1267.68        factor_learning rate = 0.01


Epoch  98: error 1267.82        factor_learning rate = 0.01
Epoch  99: error 1309.23        factor_learning rate = 0.01
```

```
val_err = squared_error(validation, res=res)
print('Initial solutions as all-1s vectors')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')
```

```
Initial solutions as all-1s vectors
Final training error = 1287.652 Relative error = 0.02300850059005928
    Validation error = 216.934  Relative error = 0.5689668332814921
```

```
#So now we have a validation score which is just above 188 which might be the
# case for over-fitting. The regularization causes the error to increase
# a bit to 216. This seems to be the optimal model.
```

```
############
## TASK 3
#Task 3: ICA for housing prices
#In this task, we study the applications of independent component analysis to housing price d
#US. The data set housing_prices.csv4 contains the monthly house price index for twenty metrc
#areas in the US from January 1987 to June 2014. As is common to time series data, the rows c
#the locations and the columns to the time stamps. The areas are in housing_prices_locations.
#the time stamps are in housing_prices_times.txt.
#Get yourself familiar with the data by studying which locations it covers and by plotting th
#series.
#This data contains missing values. We start by using k-nearest neighbour imputer. You can pl
#again. Did it change?
#Compute the ICA of the data. First, compute only top-4 components and plot them. How would y
#interpret the results?
#The US housing bubble made the housing prices climb heavily from around year 2000 until 200€
#they started to decline. By late 2008, the decline had turned into a crisis with significant
#housing prices; the houses would not start to fully recover until 2012. Can you identify the
#independent components? Repeat ICA with higher number of components (say, 8). Can you identi
#events now? Remember that some signals might have their sign changed, so you need to multipl
```

```
#-1 to see the signal the right way (though of course you need to first decide which signals
#More common way to impute values is to replace the missing values with a mean over the signa
#this and re-compute ICA. Did the results change? What does it mean if the results changed/di
#############
```

```
from sklearn.impute import KNNImputer
from sklearn.decomposition import FastICA
```

```
# Read data
data = np.genfromtxt('housing_prices.csv', delimiter=',', skip_header=1, missing_values="NA",
```

```
# We'd like to have cities as columns and time stamps as rows, so transpose
#data = data.T
```

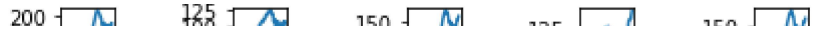
```
# Read locations
loc=[]
with open('housing_prices_locations.txt', 'r') as f:
    for row in f:
        loc.append(row.strip())
```

```
# Read times
times=[]
with open('housing_prices_times.txt', 'r') as f:
    for row in f:
        times.append(row.strip())

# Plot data
for i in range(data.shape[0]):
    plt.subplot(4, 5, i+1)
    plt.title(loc[i])
    plt.plot(data[i,:])
plt.tight_layout()
plt.show()
```


```
# Remove missing values using k-nearest neighbour imputer
imputer = KNNImputer(n_neighbors=2)
imp_data=imputer.fit_transform(data)
```

```
#So now we see how the data is imputed
```



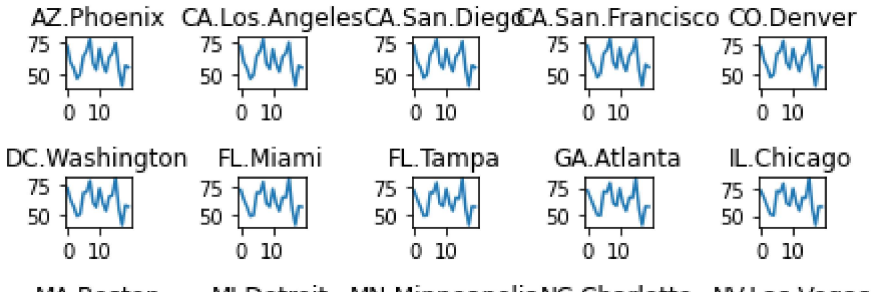
```
data
```

```
array([[   nan,    nan,    nan, ..., 145.4 , 146.05, 146.9 ],
       [ 59.33,  59.65,  59.99, ..., 219.63, 221.94, 223.33],
       [ 54.67,  54.89,  55.16, ..., 200.72, 201.94, 203.32],
       ...,
       [ 41.05,  41.28,  41.06, ..., 165.2 , 167.09, 168.97],
       [   nan,    nan,    nan, ..., 136.73, 138.54, 140.12],
       [   nan,    nan,    nan, ..., 165.74, 168.1 , 169.96]])
```

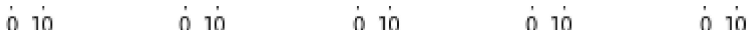
```
imp_data
```

```
array([[ 71.845,  72.48 ,  72.55 , ..., 145.4  , 146.05 , 146.9  ],
       [ 59.33 ,  59.65 ,  59.99 , ..., 219.63 , 221.94 , 223.33 ],
       [ 54.67 ,  54.89 ,  55.16 , ..., 200.72 , 201.94 , 203.32 ],
       ...,
       [ 41.05 ,  41.28 ,  41.06 , ..., 165.2  , 167.09 , 168.97 ],
       [ 56.795,  56.95 ,  57.16 , ..., 136.73 , 138.54 , 140.12 ],
       [ 55.545,  55.68 ,  55.53 , ..., 165.74 , 168.1  , 169.96 ]])
```

```
# Plot imputed data
for i in range(imp_data.shape[0]):
    plt.subplot(4, 5, i+1)
    plt.title(loc[i])
    plt.plot(imp_data[:,i])
plt.tight_layout()
plt.show()
```

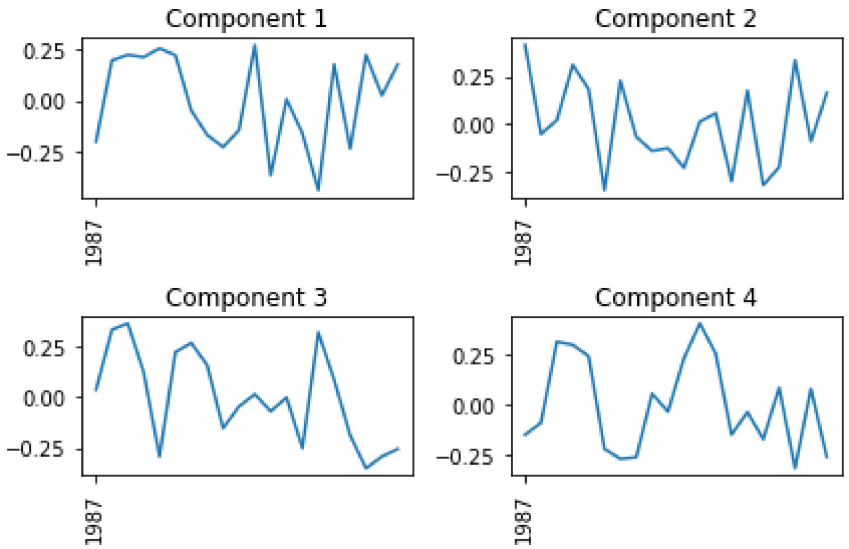



```
#We can clearly see that the plots have changed significantly.
```




```
# Compute ICA
ica = FastICA(n_components=4)
all_comps = ica.fit_transform(imp_data)
```

```
# Plot components
for i in range(all_comps.shape[1]):
    ax = plt.subplot(2, 2, i+1)
    ax.set_xticks(np.arange(0,len(times),36))
    ax.set_xticklabels(np.arange(1987, 2015, 3), rotation='vertical')
    plt.title(f'Component {i+1}')
    plt.plot(all_comps[:,i])
plt.tight_layout()
plt.show()
```

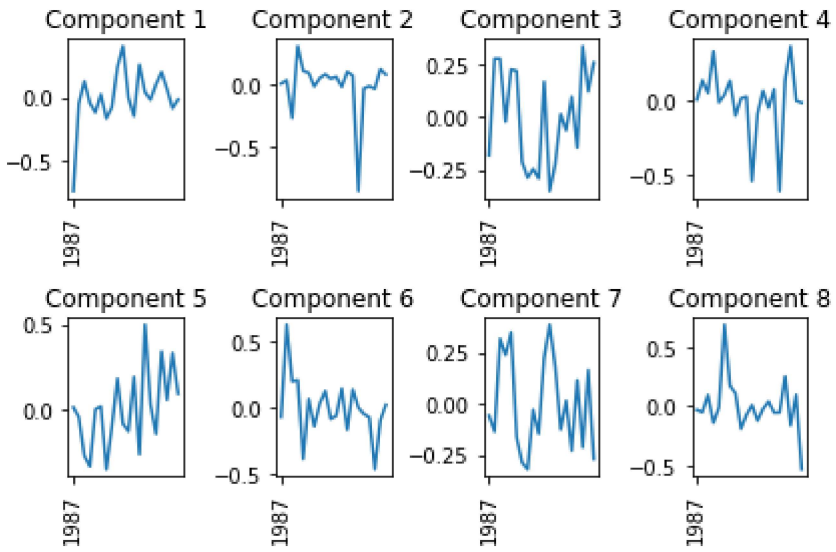



```
#From the data we can see that the cities can be classified into 4 different categories
# In the Component 1 type of cities we can see that the prices of houses show great abberatic
# zig zag pattern . The rise in price of houses in each of the component can be seen from the
# 2000 till 2006 and then sudden decline in prices since 2008 is observed in all components,
# particularly in component 3 . Compoenent 1 houses still has their price rise in most parts
# indicating the cities where investments can be highly profitable.
```

```
# Compute ICA 8 components
ica = FastICA(n_components=8)
all_comps = ica.fit_transform(imp_data)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/decomposition/_fastica.py:119: Convergen
  ConvergenceWarning)
```

```
 # Plot 8 components
for i in range(all_comps.shape[1]):
    ax = plt.subplot(2, 4, i+1)
    ax.set_xticks(np.arange(0,len(times),36))
    ax.set_xticklabels(np.arange(1987, 2015, 3), rotation='vertical')
    plt.title(f'Component {i+1}')
    plt.plot(all_comps[:,i])
plt.tight_layout()
plt.show()
```

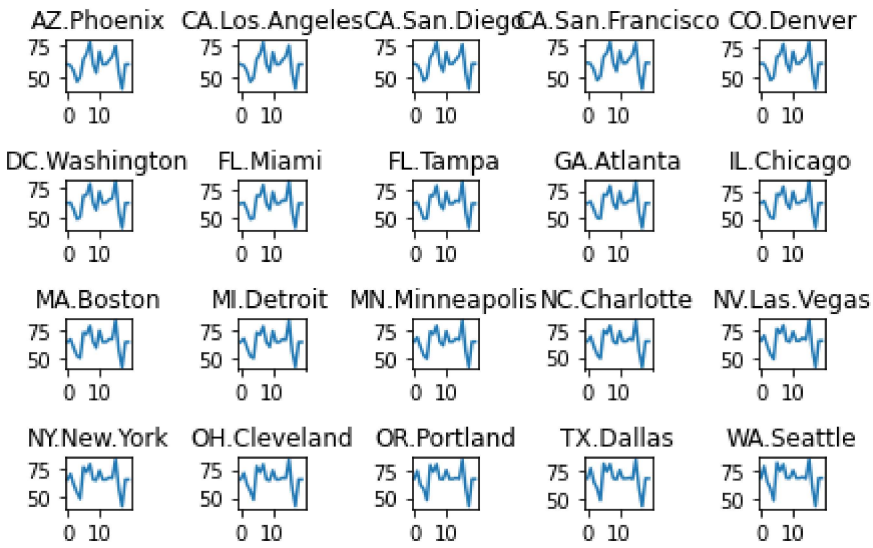



```
#Now we can see more components where the rise in price from 2000 to 2006 can be
#observed and then subsequent crisis fall from 2008 onwards. Component 2 and 4 needs
# to have a flip in sign by multiplying by -1 in order to observe the results in
# the expected way.
```

```
# To impute mean of columns, we use SimpleImputer
from sklearn.impute import SimpleImputer
mean_imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
mean_imp_data = mean_imputer.fit_transform(data)
```

```
# Plot imputed data
for i in range(mean_imp_data.shape[0]):
    plt.subplot(4, 5, i+1)
    plt.title(loc[i])
    plt.plot(mean_imp_data[:,i])
```

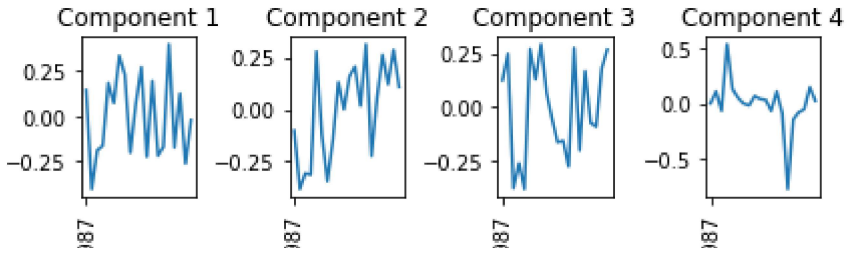
```
plt.tight_layout()
plt.show()
```




```
# Compute ICA 8 components
ica = FastICA(n_components=8)
all_comps = ica.fit_transform(mean_imp_data)


# Plot imputed data
for i in range(mean_imp_data.shape[0]):
    plt.subplot(4, 5, i+1)
    plt.title(loc[i])
    plt.plot(mean_imp_data[:,i])
plt.tight_layout()
plt.show()


# Plot 8 components
for i in range(all_comps.shape[1]):
    ax = plt.subplot(2, 4, i+1)
    ax.set_xticks(np.arange(0,len(times),36))
    ax.set_xticklabels(np.arange(1987, 2015, 3), rotation='vertical')
    plt.title(f'Component {i+1}')
    plt.plot(all_comps[:,i])
plt.tight_layout()
plt.show()
```



le see similar observation even with mean imputation, although, the drop in price is more prominent when we use k-nn. The ICA can help us see which components to invest in.

```
tmp = np.genfromtxt('ratings.csv', delimiter=',', dtype='int32')
val = np.array(tmp[:,2]+1, dtype='float')/6
# ratings are from 0 to 5, but we can't have 0 entries, as they'd be confused
# with missing ratings.
data = coo_matrix( (val, (tmp[:,0]-1, tmp[:,1]-1)) )
# data is 1-indexed, so we need to reduce 1 to get it 0-indexed

tmp = np.genfromtxt('ratings_validation.csv', delimiter=',', dtype='int32')
val = np.array(tmp[:,2]+1, dtype='float')/6
validation = coo_matrix( (val, (tmp[:,0]-1, tmp[:,1]-1)) )

n = data.get_shape()[0]
m = data.get_shape()[1]
avg = data.sum()/data.count_nonzero()
print(f'Data has {n} rows and {m} columns')
print(f'Average rating is {avg:.4F}')

# Read movies
movies = []
with open('movies.txt') as f:
    for row in f:
        movies.append(row.strip())

# Compute just factors
res = sgd(data, 10, factor_learning_rate=0.001)
val_err = squared_error(validation, res.L, res.R)
```

```
print('Only factors, fixed rate, no regularization')
print(f'Final training error = {res.err[-1]:5.3f}'
      f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
      f'\tRelative error = {val_err/squared_error(validation)}')

# Plot errors over epocs
plt.plot(res.err)
plt.gca().set_yscale('log') # log scale makes these plots easier to read
plt.show()
plt.close()

# Print the top-10 and bottom-10 movies
print_factors(res.R, movies)

# Compute just factors, but use bold driver and slower learning
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001)
val_err = squared_error(validation, res.L, res.R)
print('Only factors, bold driver, no regularization')
print(f'Final training error = {res.err[-1]:5.3f}'
      f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
      f'\tRelative error = {val_err/squared_error(validation)}')

# Plot errors over epocs
plt.plot(res.err)
plt.gca().set_yscale('log') # log scale makes these plots easier to read
plt.show()
plt.close()

#####
## YOUR PART STARTS HERE
## Play around with different learning rates & bold driver multipliers
#####

#####
## YOUR PART ENDS HERE
#####

# Compute only bias; factor_learning_rate=None means don't compute factors
res = sgd(data, 10, bold_driver=2, factor_learning_rate=None, bias_learning_rate=0.0001, verb
val_err = squared_error(validation, res.L, res.R, res.global_bias, res.row_bias, res.col_bias
print('Bias only, no regularization')
print(f'Final training error = {res.err[-1]:5.3f}'
      f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
      f'\tRelative error = {val_err/squared_error(validation)}')


# Compute factors and bias
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, ve
```

```
val_err = squared_error(validation, res.L, res.R, res.global_bias, res.row_bias, res.col_bias
print('Factors and bias, no regularization')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')

# Print the movies based on their bias
print_factors(res.col_bias, movies)

# Add regularization
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, fa
val_err = squared_error(validation, res.L, res.R, res.global_bias, res.row_bias, res.col_bias
print('Factors and bias, regularized')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')

# To be able to compute the SVD, you can convert the data into full array with
#data.toarray()
# To use squared_error(), you must multiply Sigma to U or V and use a transpose of V.
#####
## YOUR PART STARTS HERE
## Use the parameters from earlier and compare your errors (training and validation)
## to those you get with rank-10 truncated SVD
#####

#####
## YOUR PART ENDS HERE
#####

############
## TASK 2 ##
############

# Initial factors are given as a tuple (L, R); R is transposed
init_LR = (np.ones(shape=(n,10)), np.ones(shape=(m,10)))
# Initial biases are a tuple (global_bias, row_bias, col_bias
# Give the true global bias.
init_biases = (avg, np.ones(shape=n), np.ones(shape=m))

# You can change the learning rates and bold driver factor if you want
res = sgd(data, 10, bold_driver=2, factor_learning_rate=0.0001, bias_learning_rate=0.0001, ve
val_err = squared_error(validation, res=res)
print('Initial solutions as all-1s vectors')
print(f'Final training error = {res.err[-1]:5.3f}'
          f'\tRelative error = {res.err[-1]/squared_error(data)}')
print(f'    Validation error = {val_err:5.3f}'
          f'\tRelative error = {val_err/squared_error(validation)}')

#####
```

```
## YOUR PART STARTS HERE
## Try the other initial solutions
#####



#####
## YOUR PART ENDS HERE
#####



# try two different values for learning rates using 5-fold cross validation
learning_rates = [1e-5, 1e-3]
err = []
for rate in learning_rates:
    err.append( cv(data, folds=5, k=10, bold_driver=2, factor_learning_rate=rate, bias_learni
print('Rate\terr')
for i in range(len(learning_rates)):
    print(f'{learning_rates[i]}\t{err[i]}')

#####
## YOUR PART STARTS HERE
## Implement the grid search. After you've found the optimum combination of
## hyperparameters, run sgd with full training data using those parameters and
## compute the validation error.
#####



#####
## YOUR PART ENDS HERE
#####



############
## TASK 3 ##
############

from sklearn.impute import KNNImputer
from sklearn.decomposition import FastICA

# Read data
data = np.genfromtxt('housing_prices.csv', delimiter=',', skip_header=1, missing_values="NA",
# Read locations
loc=[]
with open('housing_prices_locations.txt', 'r') as f:
    for row in f:
        loc.append(row.strip())
# Read times
times=[]
with open('housing_prices_times.txt', 'r') as f:
    for row in f:
        times.append(row.strip())
```

```
# Plot data
for i in range(data.shape[0]):
    plt.subplot(4, 5, i+1)
    plt.title(loc[i])
    plt.plot(data[i,:])
plt.tight_layout()
plt.show()

# We'd like to have cities as columns and time stamps as rows, so transpose
data = data.T

# Remove missing values using k-nearest neighbour imputer
imputer = KNNImputer(n_neighbors=2)
imp_data=imputer.fit_transform(data)

#Plot data again

# Compute ICA
ica = FastICA(n_components=4)
all_comps = ica.fit_transform(imp_data)

# Plot components
for i in range(all_comps.shape[1]):
    ax = plt.subplot(2, 2, i+1)
    ax.set_xticks(np.arange(0,len(times),36))
    ax.set_xticklabels(np.arange(1987, 2015, 3), rotation='vertical')
    plt.title(f'Component {i+1}')
    plt.plot(all_comps[:,i])
plt.tight_layout()
plt.show()

# To impute mean of columns, we use SimpleImputer
from sklearn.impute import SimpleImputer
mean_imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
mean_imp_data = mean_imputer.fit_transform(data)

#####
## YOUR PART STARTS HERE
## Do further analysis
#####



#####
## YOUR PART ENDS HERE
#####
```