**import** numpy **as** np, pandas **as** pd, random

**import** matplotlib.pyplot **as** plt

**def** load\_csv(**dir**):

"""

Load the csv provided on bcourses.

This function makes sure that we have valid column name and data array size inputs.

"""

data **=** pd.read\_csv(dir)

names **=** [s.strip() **for** s **in** data.columns]

similarities **=** data.as\_matrix()

**assert** len(names) **==** len(similarities) **and** len(similarities) **==** len(similarities[0]), "Data formatted incorrectly!"

**return** names, similarities

**def** sim\_to\_dis\_val(**similarity**):

**return** 1**-**similarity

**def** convert\_similarities\_to\_distances(**similarities**):

"""

Converts similarity values to distance values based off of the system of our choosing.

@param similarities: A matrix of similarity values to be converted into distance relations

@return Returns a matrix of distance relations.

We begin with a pointer to our matrix of similarity values, newly named distances.

We then iterate through with a nested for-loop. As it is a pointer, this function has O(1) for space efficiency.

This method is also fast as any np map functions are at least as slow as this.

**>>>** convert\_similarities\_to\_distances([[0.5, 1], [1, 2]])

[[0.5, 0], [0, -1]]

"""

distances **=** similarities

**for** i **in** range(len(distances)):

**for** j **in** range(len(distances[0])):

distances[i][j] **=** sim\_to\_dis\_val(distances[i][j])

**return** distances

**def** dist(**pos1**, **pos2**):

"""

Compute the Euclidean distance between two locations (numpy arrays) a and b

Thus, dist(pos[1], pos[2]) gives the distance between the locations for items 1 and 2

@param pos1: Position 1

@param pos2: Position 2

@return Returns the Euclidean distance between these two positions.

This function is not the most efficient, but the data we are working with is rather small.

Also, this makes the code much more readable!

**>>>** dist([0, 0], [3, 4])

5.0

**>>>** dist([1, 2, 3], [3, 4 ,4])

3.0

"""

**assert** len(pos1) **==** len(pos2), "Position1 and Position2 must be of the same dimension"

differences **=** [pos1[i] **-** pos2[i] **for** i **in** range(len(pos1))]

squared\_diff **=** [val**\***val **for** val **in** differences]

**return** pow(sum(squared\_diff), 0.5)

**def** mean\_squared\_error(**expected\_value**, **actual\_value**):

**return** pow(expected\_value**-**actual\_value, 2)

**def** stress(**distances**, **positions**):

"""

Takes in an matrix of similarity values, and calculates the current amount of stress based off of the positions.

@param distances: The (NxN) matrix of goal distance values between each pair of positions

@param positions: An (NxD) array of current positions of each point

@return Returns the current amount of "error" of these position values given the actual values (called stress).

"""

stress **=** 0

**for** i **in** range(len(distances)):

**for** j **in** range(len(distances[0])):

**if** i **!=** j:

stress **+=** mean\_squared\_error(distances[i][j], dist(positions[i], positions[j]))

**return** stress

**def** add\_delta(**positions**, **row**, **col**, **delta**):

"""

This is a helper function that will make a new vector which is the same as p (a position matrix), except that

p[i,d] has been increased by delta (which may be positive or negative).

@param positions: An (NxD) array of current positions of each point

@param row: The respective row value of positions to increase

@param col: The respective col value of positions to increase

@param delta: The value to add to positions[i][j]

@return Returns an (NxD) array of the current positions with value (row, col) changed by delta.

"""

new\_positions **=** np.array(positions)

new\_positions[row][col] **=** new\_positions[row][col] **+** delta

**return** new\_positions

**def** compute\_gradient(**distances**, **positions**, **row**, **col**, **delta**):

"""

Compute the gradient of the stress function with repect to the [row, col] entry of a position matrix p.

@param distances: The (NxN) matrix of goal distance values between each pair of positions

@param positions: An (Nx2) array of current positions of each point

@param row: The respective row to calculate the gradient at

@param col: The respective col to calculate the gradient at

@param delta: The step size of the gradient

@return Returns the gradient of the stress function at the given (row, col)

Compute the derivative of stress with respect to the i'th coordinate of the x'th dimension

Here, to compute numerically, you can use the fact that:

f'(x) = (f(x+delta)-f(x-delta))/(2 delta) as delta -> 0

"""

x\_plus\_delta, x\_minus\_delta **=** add\_delta(positions, row, col, delta), add\_delta(positions, row, col, **-**delta)

f\_plus, f\_minus **=** stress(distances, x\_plus\_delta), stress(distances, x\_minus\_delta)

**return** **-**((f\_plus**-**f\_minus)**/**(2**\***delta))

**def** normalize\_vector(**vector**):

magnitude **=** pow(sum([val**\***val **for** val **in** vector]), 0.5)

**return** np.array([val**/**magnitude **for** val **in** vector])

**def** scale\_vector(**vector**, **scale\_factor**):

**return** np.array([val**\***scale\_factor **for** val **in** vector])

**def** compute\_full\_gradient(**distances**, **positions**, **delta**):

"""

Numerically compute the full gradient of stress at a position p

This should return a matrix whose elements are the gradient of stress at p with respect to each [i,d] coordinate

@param distances: The (NxN) matrix of goal distance values between each pair of positions

@param positions: An (Nx2) array of current positions of each point

@return Returns an (Nx2) array of gradient values for each position.

"""

gradient **=** np.zeros(positions.shape)

**for** row **in** range(positions.shape[0]):

**for** col **in** range(2):

gradient[row][col] **=** compute\_gradient(distances, positions, row, col, delta)

gradient[row] **=** normalize\_vector(gradient[row])

gradient[row] **=** scale\_vector(gradient[row], delta)

**return** gradient

**def** matrix\_add(**matrix1**, **matrix2**):

"""

Destructively adds matrix1 and matrix2 together.

@param matrix1: Matrix to add

@param matrix2: Matrix to add

@return Returns the sum of the two matrices by element.

**>>>** matrix1 = np.array([[1, 2], [3, 4]])

**>>>** matrix2 = np.array([[10, 20], [30, 40]])

**>>>** matrix\_sum = matrix\_add(matrix1, matrix2)

**>>>** matrix\_sum[0][0]==11 and matrix\_sum[0][1]==22 and matrix\_sum[1][0]==33 and matrix\_sum[1][1]==44

True

"""

**assert** matrix1.shape **==** matrix2.shape, "Matrices must be of same size!"

**if** len(list(matrix1.shape)) **==** 1:

**return** np.array([matrix1[i] **+** matrix2[i] **for** i **in** range(matrix1.shape[0])])

**for** i **in** range(len(matrix1)):

matrix1[i] **=** matrix\_add(matrix1[i], matrix2[i])

**return** matrix1

**def** get\_step\_size(**current\_stress**, **min\_stress**, **delta**):

**return** current\_stress**\***delta**/**min\_stress

**def** round\_to\_micro(**val**):

**return** int(val**\***1e6)**/**1e6

**def** main():

**def** record\_history():

*nonlocal* history, current\_confidence

**if** abs(current\_stress**-**history) **<** DELTA:

current\_confidence, history **=** current\_confidence **+** 1, (current\_stress**+**history)**/**2

**else**:

current\_confidence, history **=** 1, current\_stress

**def** plot\_positions():

xVals, yVals **=** [pos[0] **for** pos **in** positions], [pos[1] **for** pos **in** positions]

\_, ax **=** plt.subplots()

ax.scatter(xVals, yVals)

**for** i, txt **in** enumerate(names):

ax.annotate(txt, (xVals[i], yVals[i]))

plt.savefig('positions\_plot\_5.jpg')

plt.show()

**def** plot\_distaces\_positions():

xVals, yVals **=** [], []

**for** row **in** range(21):

**for** col **in** range(21):

xVals.append(distances[row][col])

yVals.append(dist(positions[row], positions[col]))

plt.plot(xVals, yVals, 'ro')

plt.ylabel **=** 'Distance between positions'

plt.xlabel **=** 'Psychological Distance'

plt.savefig('distances\_positions.jpg')

plt.show()

**def** plot\_stress\_iterations():

plt.plot([i **for** i **in** range(len(stress\_history))], stress\_history)

plt.ylabel **=** 'Stress'

plt.xlabel **=** 'Iteration'

plt.savefig('stress\_iterations.jpg')

plt.show()

MIN\_STRESS, DIMENSIONS, DELTA, CONFIDENCE, MAX\_STEPS **=** 10, 2, 1e-2, 10, 2500

names, similarities **=** load\_csv('hw04.csv')

distances **=** convert\_similarities\_to\_distances(similarities)

positions **=** np.random.normal(0.0,1.0,**size=**(len(names),DIMENSIONS))

i, current\_stress **=** 0, MIN\_STRESS**+**1

current\_confidence, history **=** 1, **-**1

stress\_history **=** []

**while** current\_stress **>** MIN\_STRESS **and** current\_confidence **<** CONFIDENCE **and** i **<** MAX\_STEPS:

gradient **=** compute\_full\_gradient(

distances,

positions,

get\_step\_size(current\_stress, MIN\_STRESS, DELTA)

)

positions **=** matrix\_add(positions, gradient)

i, current\_stress **=** i**+**1, round\_to\_micro(stress(distances, positions))

stress\_history.append(current\_stress)

record\_history()

**if** i**%**10 **==** 0:

print('Iteration: {0} | Confidence: {1} / {2} | Stress: {3}'.format(i, current\_confidence, CONFIDENCE, current\_stress))

plot\_positions()

plot\_distaces\_positions()

*# plot\_stress\_iterations()*

**if** \_\_name\_\_ **==** '\_\_main\_\_':