COGSCI131–Spring 2019 — Homework 4Solutions

Yuchen Zhou, SID 3034489901

1.

Some basic rules that must be satisfied:

- If $similarity = 1$, $distance = 0$
- If $similarity_{12} > similarity_{23}$, $distance_{12} < distance_{23}$

Here are three possible ways:

1. $dist = 1 - simi \in [0, 1]$
2. $dist = 1 - simi^{\frac{1}{2}} \in [0, 1]$
3. $dist = \frac{1}{simi} - 1 \in [0, \infty]$

Observe the fact that people are more sensitive to differences than similarities. So distance from 0.0 to 0.1 ought to be more than that from 0.9 to 1.0.
Thus, we choose $dist = 1 - simi^{\frac{1}{2}}$ so that the output of function ranges from 0 to 1, which is better for convergence.

2.

1. Code:

```python
import numpy
import matplotlib.pyplot as plt

# what each data point is called:
names = ["football", "baseball", "basketball", "tennis", "softball", "canoeing
                                        ", "handball", "rugby", "hockey",
         "ice hockey", "swimming", "track", "boxing", "volleyball", "lacrosse"
                                                  , "skiing", "golf", "polo",
                                                   "surfing",
         "wrestling", "gymnastics"]

# load the csv provided on bcourses
similarities = numpy.loadtxt(open("similarities.csv", "rb"), delimiter=",",
                                        skiprows=1)

distances =1-similarities**0.5
#- How should we convert similarities to distances?

D = 2  # How many dimensions we are going to use
N = distances.shape[0]  # the number of items
assert (distances.shape[1] == N and N == len(names))  # be sure we loaded as
                                        many items as we have names for


def dist(a, b):
    return pow((a[0]-b[0])**2+(a[1]-b[1])**2,0.5)

# 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


def stress(p):
    sum=0
    for i in range(N):
        for j in range(i,N):
            sum+=(distances[i][j]-dist(p[i],p[j]))**2
    return sum

# Take a matrix of positions (called here "p") and return the stress


def add_delta(p, i, d, 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)
    v = numpy.array(p)
    v[i, d] += delta
    return v


def compute_gradient(p, i, d, delta=0.001):
    return (stress(add_delta(p,i,d,delta))-stress(add_delta(p,i,d,-delta)))/(2
                                            *delta)
```

```
# compute the gradient of the stress function with repect to the [i,d] entry
                                        of a position matrix p
# (e.g. 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

def compute_full_gradient(p):
    q=numpy.zeros((N,D))
    for i in range(N):
        for d in range(D):
            q[i][d]=compute_gradient(p,i,d)
    return q
# 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

# Pick a position for each point. Note this is an NxD matrix
# so that pos[11,1] is the y coordinate for the 11th item
# and pos[11] is a (row) vector for the position of the 11th item

pos = numpy.random.normal(0.0, 1.0, size=(N, D))

# Now go through and adjust the position to minimize the stress
print(pos)
for steps in range(100):
    pos=pos-compute_full_gradient(pos)*0.04
    print("step",steps)
    print(stress(pos))
fig=plt.figure()
ax=fig.add_subplot(111)
"""
for i in range(N):
    ax.plot(pos[i][0],pos[i][1],"*")
    ax.text(pos[i][0]-0.01,pos[i][1]+0.01,names[i])
plt.show()
"""
k=1
for i in range(N):
    for j in range(i+1,N):
        #ax.scatter(k,distances[i][j],color="red",s=10)
        #ax.scatter(k,dist(pos[i],pos[j]),color="blue",s=10)
        ax.scatter(k, dist(pos[i],pos[j])-distances[i][j], color="green", s=10
                                        )
        print(k,distances[i][j]-dist(pos[i],pos[j]))
        k+=1
plt.show()
```

2. Final stress $s = 6.64$.

3. The result basically agree with my intuitions. One example is ball games cluster together. Another example is skiing, swimming and surfing are very close. Also, different kinds of sports are far from each other.

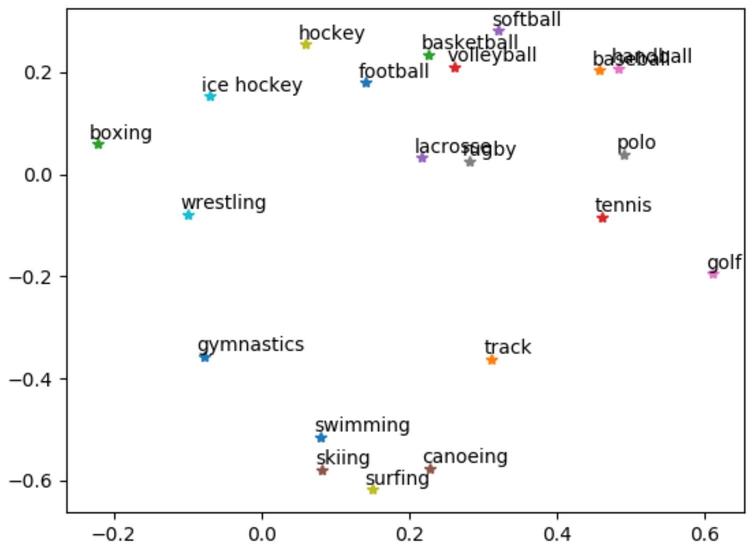

Figure 1: dots of sports in two dimensional space

3.

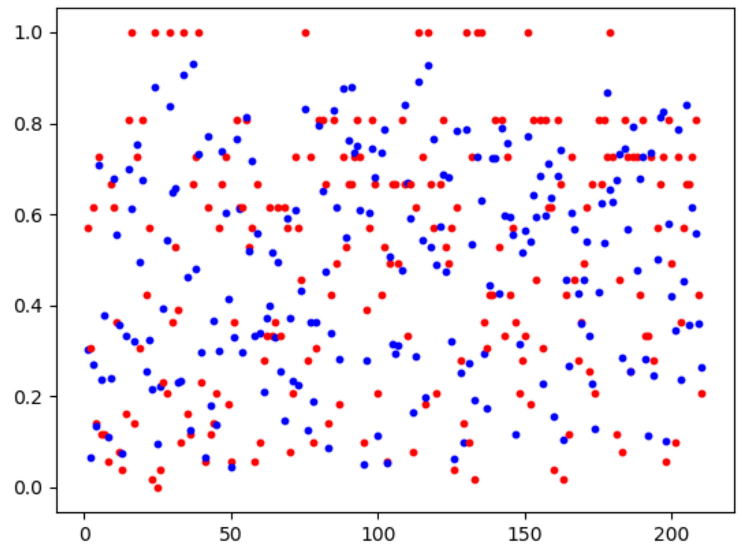

Figure 2: distances of MDS vs. people reported

1. In figure 2, red dots are distances judged by people and blue dots are distances by MDS. As shown in the figure, it looks not very perfect, so I draw loss of distances as figure 3.

2. Generally speaking, most pairs are simulated pretty good, for the distance losses are mostly less than 0.2.

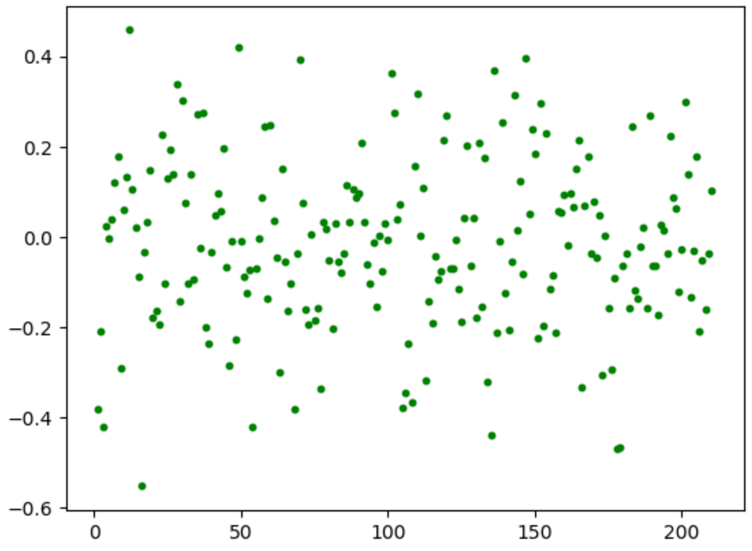

Figure 3: loss of distances for every pair

3.

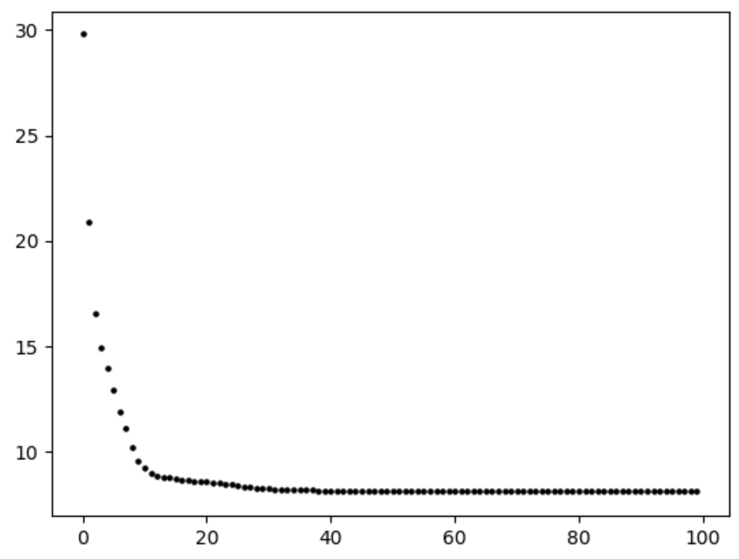

Figure 4: Stress over iterations

1. We can find that stress goes down very quickly during first few iterations and then derivative goes bigger (negative) to about 0.
2. When there is no distinct changes on stress, we can stop iterations. In this case, we can stop looping after 40 iterations.

4.

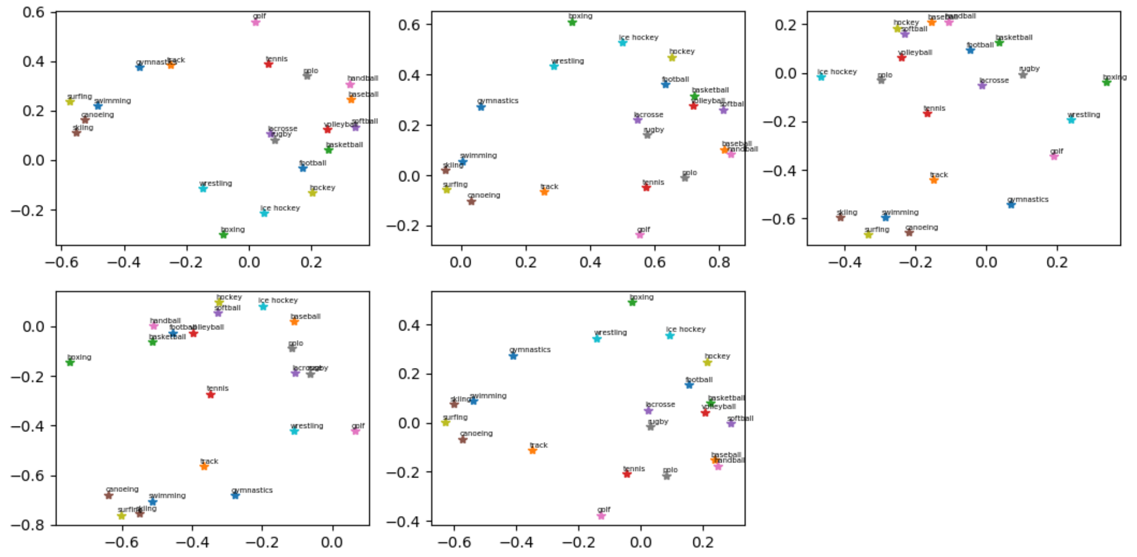

Figure 5: run MDS five times

1. As shown in the figure, five results are not the same.
2. One reason is that initial position is randomized. That is, even there are no losses for all distances and the stress is 0, relative distances of every pairs are all the same in different trails but absolute position may change.
3. However, we cannot reach a state that stress equals to 0 because the original dots are not strictly in one plain. So we take losses and this makes differences.
4. Moreover, the result may get caught in a trap so it cannot reach the global minimum point but goes towards the local minimum point, which highly depends on initial positions of every dots.

5.

1. I would pick the one with minimal stress. If stress is less, it means the loss is less, which suggests that the model performs good.

2. Code as follows. The definition of functions are same to previous code so I just show main program.

```
beststress=99999
bestpos=numpy.zeros((N,D))
fig=plt.figure()
ax=fig.add_subplot(111)
for ii in range(5):
    print("i:",ii)
    pos = numpy.random.normal(0.0, 1.0, size=(N, D))
    for steps in range(100):
        pos=pos-compute_full_gradient(pos)*0.04
    print(stress(pos))
    if stress(pos)<beststress:
        beststress=stress(pos)
        bestpos=pos
for i in range(N):
    ax.plot(bestpos[i][0],bestpos[i][1],"*")
    ax.text(bestpos[i][0]-0.01,bestpos[i][1]+0.01,names[i],fontsize=12)
plt.show()
```

3. Five results of stresses are: 7.44 7.19 7.51 6.50 7.42.
   Choose the fourth trail as the "best" answer.

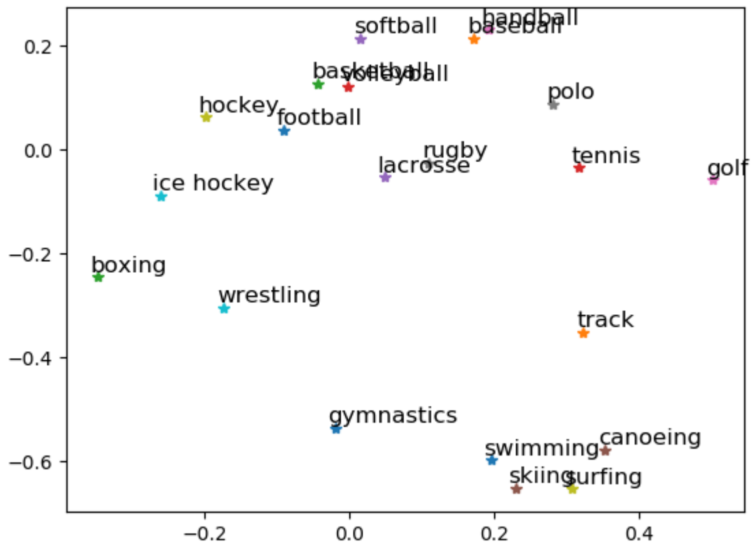

Figure 6: "best" answer