

Programming Assignment 3: PCA and Fastmap

1. Group member

- a) Hao Yang, USCID: 5284267375. (Contributions: All)

2. Part 1: Implementation

1. Programming language: Python 3.7
2. Libraries used: random, numpy, matplotlib.pyplot, mpl_toolkits.mplot3d
3. Clustering algorithm: PCA, Fastmap
4. Script description:

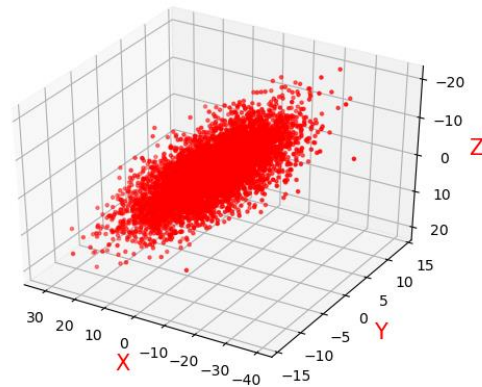
1. PCA:

1. Data processing: read txt file line by line, and save as list. Totally 6000 data points.

```
data_pca = []
with open('pca-data.txt', 'r') as file1:
    line = file1.readline().strip('\n').split('\t')
    line = list(map(float, line))
    data_pca.append(line)
    line = file1.readline().strip('\n').split('\t')
    while line != ['']:
        line = list(map(float, line))
        data_pca.append(line)
        line = file1.readline().strip('\n').split('\t')
datanp = np.array(data_pca)
```

```
▼ data_pca: [[5.906262853951832, -7.729464584682111, 9.144944874608196]
> 0000: [5.906262853951832, -7.729464584682111, 9.144944874608196]
> 0001: [-8.640323106971573, 1.7242604350569888, -10.696805187953952]
> 0002: [0.25854061492215674, 0.23062223968214718, 0.7674391638194876]
> 0003: [-5.234353794193943, 3.194685075162285, -1.8943847407605263]
> 0004: [12.62286294146571, -3.5078877906880543, 4.086258338102421]
> 0005: [0.7855670569156168, 3.007478450451627, 0.001893147740198886]
```

Data points in 3D space:



2. Implementation of PCA

Step 0: Data centralization, make all data point subtracted by their average value.

```
med = (xnp.sum(axis=0))/size # Step 0: centralization
xnp = xnp - med
x = xnp.tolist()
```

Step 1: compute covariance matrix $\Sigma_{n \times n}$ by doing matrix multiplication of dataset.

```
xm = np.mat(xnp).T # Step 1: compute covariance matrix E
E = xm * xm.T
E = E/size
```

Get the covariance matrix:

```
✓ E: matrix([[ 81.22845778, -15.83817402,  31.66312677],\n
✓ [0:3] : [matrix([[ 81.2284577...6312677]]), matrix([[ -15.83817402,\n
> 0: matrix([[ 81.22845778, -15.83817402,  31.66312677]])\n
> 1: matrix([[ -15.83817402,  13.69953054, -15.26190629]])\n
> 2: matrix([[ 31.66312677, -15.26190629,  31.36154358]])
```

Step 2: Compute the eigenvalues and eigenvectors of $\Sigma_{n \times n}$ by function `linalg.eig(E)`, and sort eigenvalues and corresponding eigenvectors in decreasing order.

```
eigval, feavec = np.linalg.eig(E) # Step 2: compute eigenvalues and featurevectors of E
idx = eigval.argsort()[::-1] # in decreasing order
ev = eigval[idx[0]]
fv = feavec[:,idx[0]]
for i in idx[1:]:
    fv = np.c_[fv, feavec[:,i]]
    ev = np.append(ev, eigval[i])
```

The sorted eigenvalues and eigenvectors:

```
✓ ev: array([101.60286375, 19.89589866, 4.79076949])
> [0:3] : [101.60286374952251, 19.895898658310898, 4.790769486190333]
```

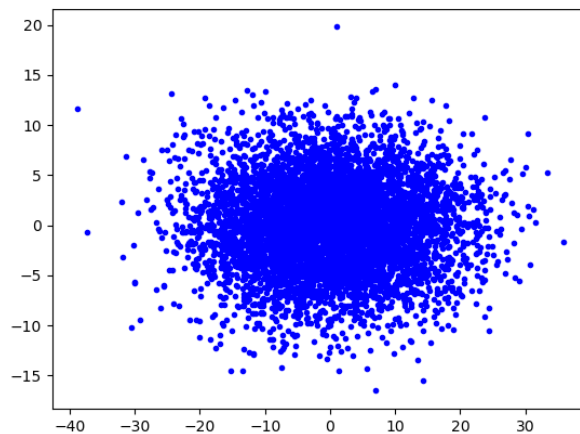
```
✓ fv: matrix([[ 0.86667137, -0.4962773 , -0.0508879 ],\n
✓ [0:3] : [matrix([[ 0.86667137...508879 ]]), matrix([[ -0.\n
> 0: matrix([[ 0.86667137, -0.4962773 , -0.0508879 ]])\n
> 1: matrix([[ -0.23276482, -0.4924792 , 0.83862076]])\n
> 2: matrix([[ 0.44124968, 0.71496368, 0.54233352]])
```

Step 3: Pick the first k column of eigenvectors as the transformation matrix

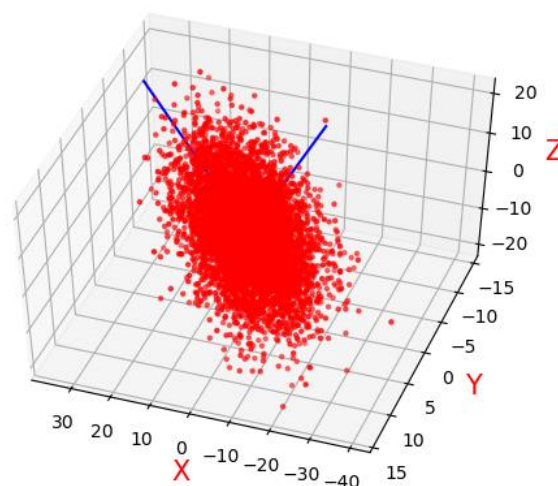
to compute the new coordinate system and new coordinates for data points.

```
Utr = fv[:,0:k] # Step 3: got U truncate
origin_direction = np.eye(dim)
new_direction = Utr.T * origin_direction
z = Utr.T * xm
```

Result of dimension reduction in 2D space:



The coordinate system of 2D space in 3D space:

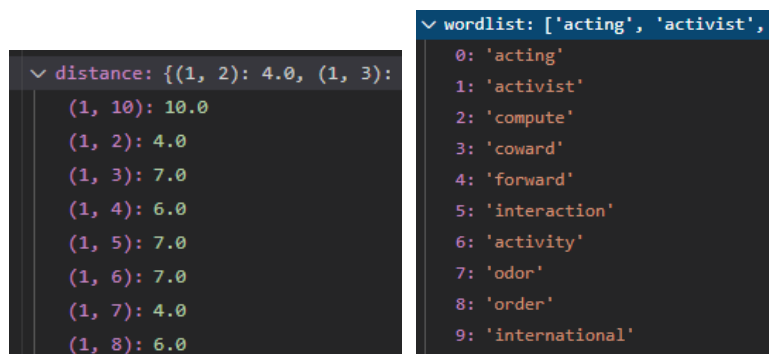


2. Fastmap:

1. Data processing: Read fastmap-data.txt file and save it as multi-key dictionary. Read fastmap-wordlist.txt file and save it as list.

```
wordlist = []
distance = {}
with open('fastmap-wordlist.txt','r') as wl:          # got wordlist
    line = wl.readline().strip('\n')
    while line:
        wordlist.append(line)
        line = wl.readline().strip('\n')
with open('fastmap-data.txt','r') as wd:             # got distance data
    line = wd.readline().strip('\n').split('\t')
    while line != ['']:
        distance[int(line[0]),int(line[1])] = float(line[2])
        line = wd.readline().strip('\n').split('\t')
```

Part of data:



```
distance: {(1, 2): 4.0, (1, 3): 7.0, (1, 10): 10.0, (1, 2): 4.0, (1, 3): 7.0, (1, 4): 6.0, (1, 5): 7.0, (1, 6): 7.0, (1, 7): 4.0, (1, 8): 6.0}
```

```
wordlist: ['acting', 'activist', 'acting', 'activist', 'compute', 'coward', 'forward', 'interaction', 'activity', 'odor', 'order', 'international']
```

Step 1: Find the farthest point pair O_a , O_b . For keeping the time complexity of implement in $O(N)$, can't simply read the entire distance table. We should start from a random object O_a and find another object O_b which is farthest from it.

```
def Farthestpoint(wd,size,pointA,loop,x):
    max = -1
    for i in range(1,size+1):
        if i < pointA:
            if NowDistance(wd,i,pointA,loop,x) > max:
                pointB = i
                max = NowDistance(wd,i,pointA,loop,x)
        elif i > pointA:
            if NowDistance(wd,pointA,i,loop,x) > max:
                pointB = i
                max = NowDistance(wd,pointA,i,loop,x)
    return pointB
```

Totally $N-1$ loop because need to check all other points. Takes $O(N)$

Then find the farthest object from O_b , check if it's O_a . If it is, we found the farthest pair. if not, iteratively do that for 5 iteration. As professor said in class, this works in 99% cases. I tested it more than 100 times, all work.

```
IterationMax = 5
Oa = random.randint(1,size)
Ob = Farthestpoint(wd,size,Oa,loop,x)
for i in range(IterationMax):
    O_next = Farthestpoint(wd,size,Ob,loop,x)
    if O_next == Oa:
        break
    Oa = O_next

    O_next = Farthestpoint(wd,size,Oa,loop,x)
    if O_next == Ob:
        break
    Ob = O_next
if Oa > Ob:
    return Ob,Oa,NowDistance(wd,Ob,Oa,loop,x)
else:
    return Oa,Ob,NowDistance(wd,Oa,Ob,loop,x)
```

It takes $O(N)$ to find the farthest point from one point, and we implement 5 loops of that, so totally takes $O(10N)$ which is still $O(N)$.

Step 2: We already found farthest pair O_a, O_b in $O(N)$ time, next we need to consider all other object O_i one by one, and compute the first coordinate of O_i in new space we are going to embed in:

```
coordinate = [[] for i in range(size)]
```

```
pointa, pointb, dis = Farthest(wd,size,loop,coordinate) # Step 1: find farth
for i in range(1,size+1): # Step 2: consider
    if i == pointa:
        coordinate[i-1].append(0)
    elif i == pointb:
        coordinate[i-1].append(dis)
    else:
        coordinate[i-1].append(xiCompute(wd,pointa,pointb,i,loop,coordinate))
```

This will go through all N objects, so I takes $O(N)$ time.

```
def xiCompute(wd,Oa,Ob,Oi,loop,x):
    Dab = NowDistance(wd,Oa,Ob,loop,x)
    if Oa < Oi:
        Dai = NowDistance(wd,Oa,Oi,loop,x)
    else:
        Dai = NowDistance(wd,Oi,Oa,loop,x)
    if Ob < Oi:
        Dbi = NowDistance(wd,Ob,Oi,loop,x)
    else:
        Dbi = NowDistance(wd,Oi,Ob,loop,x)
    xi = (Dab**2 + Dai**2 - Dbi**2)/([2 * Dab])
    return xi
```

We got the first coordinate for all object in $O(N+N)=O(N)$ time so far.

Step 3: Iteratively compute the remain coordinates for all object:

```
for loop in range(k):
    pointa, pointb, dis = Farthest(wd,size,loop,coordinate)
    for i in range(1,size+1):
        if i == pointa:
            coordinate[i-1].append(0)
        elif i == pointb:
            coordinate[i-1].append(dis)
        else:
            coordinate[i-1].append(xiCompute(wd,pointa,pointb,i,loop,coordinate))
```

Totally k loops for embedding in k-dimensional space, so Totally takes $O(kN)$ times.

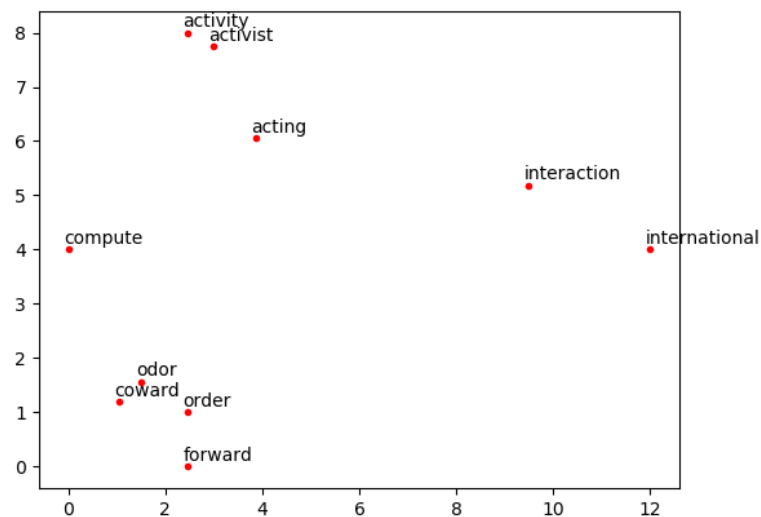
Notice: Because we need to keep the time complexity of implementation in $O(N)$. So, after computing the first coordinate of all object, we can't create a new table for all object to record their new distance between others because it will take $O(N^2)$ times. Therefore, we will compute the new distance of two object in specific loop based of the coordinates we got.

In function NowDistance(wd,Oa,Ob,loop,x), we compute the distance in specific loop iteratively:

```
def NowDistance(wd,Oa,Ob,loop,x):
    dis = wd[Oa,Ob]
    for i in range(loop):
        dis = (dis**2 - (x[Oa-1][i]-x[Ob-1][i])**2)**(1/2)
    if type(dis) == complex:
        return 0
    else:
        return dis
```

It takes $O(k)$ to compute new distance. So in step of finding farthest pair, we can only spend $O(10kN)=O(kN)$ instead of compute all distance pair first and use, which spends $O(N^2)$.

Result:



5. Optimizations

1. PCA: Used k as the part of parameter of the function instead of 2, and compute the information remaining rate by computing the proportion of first k eigenvalue. Therefore can see if it will lose too many information to reduce dimension to k .

```
e = (sum(ev[:k])/sum(ev)) * 100
```

```
e: 96.20651892968411
```

In this assignment, reduce dimension to 2 can remain 96.2% information.

2. Fastmap:

1. I did some researches and tests, find that choose 5 as the max iteration time when finding the farthest pair works better, to achieve result and avoid running too long.

2. Compute current distance only when need to use instead of computing all distance pair first to ensure the implementation run in $O(N)$ time.

3. One improvement: I combined PCA and fastmap when implement fastmap algorithm. When set k for implementing fastmap, we don't know whether this value of k is a good k -value to perform the dataset in k -dimension space. Therefore, after implement fastmap with value k , I also implement it again with value $2*k$ to embed the dataset in a higher dimension space, then implement PCA to do dimensional reduction and this can see if the first k eigenvalue is larger enough to remain majority information. If, for example, we implement fastmap to embed dataset into 2D space and found that the first 2 eigenvalue only contain 50% information comparing with embedding in 4D space, we should not choose k as 2, need to choose a larger k value. On the contrary, this can also check if the k value is too large.

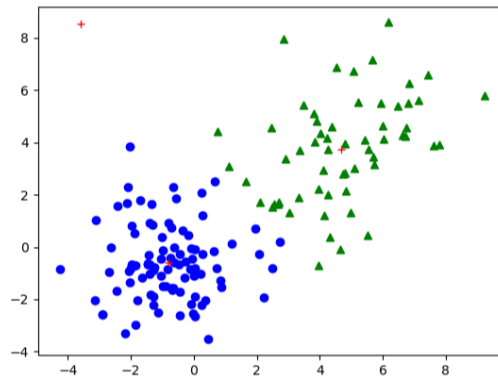
```
higher_space_check = Fastmap(wordlist,distance,4)
q, w, e, information_rate = PCA(higher_space_check,2)
print(information_rate)
```

In this assignment, the information contained in 2D space is 85% of that in 4D space, so we can say it's satisfactory to fit dataset in 2D space and doesn't lose too many information.

6. Challenges faced

1. PCA: find that using `np.linalg.eig` to compute eigenvalue and eigenvector will sometimes converge to a complex numbers result or contrary number result. It will generate a mirror result, but will not change the result in this

implementation.



2. Fastmap: Initially I compute all distance pair after every loop. But I found out that it will result implement in $O(N^2)$ time. So change it to compute current distance only when need to use distance. It can reduce time complexity to $O(kN)$.

I found that sometimes the distance we be calculated as complex value when two objects are too close. For fix this problem, I need to return 0 when returned complex distance value for doing comparison.

7. Result

1. PCA:

Run `k_eigenvector, dir, z = PCA(data_pca,2)` function with dataset `data_pca` above:

```
k_eigenvector, dir, z = PCA(data_pca,2)
print("the first k eigenvector is: ")
print(k_eigenvector)
print("the coordinate system of 2D spcae in 3D space is:" + str(dir))
```

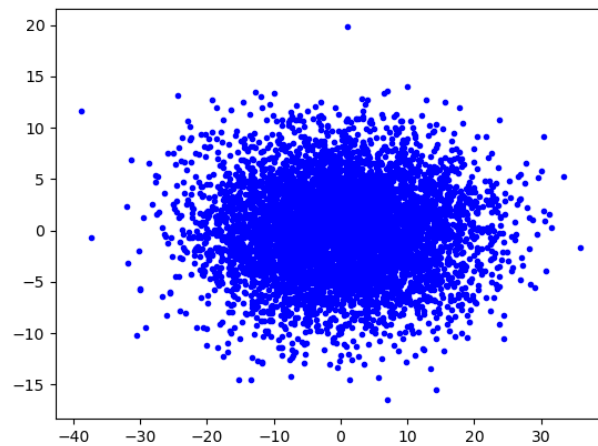
Print the result of first k eigenvector and the coordinate system of 2D space:

```

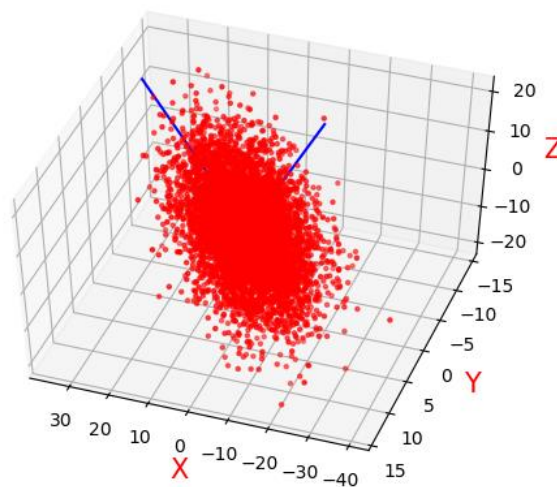
the first k eigenvector is:
[[ 0.86667137 -0.4962773 ]
 [-0.23276482 -0.4924792 ]
 [ 0.44124968  0.71496368]]
the coordinate system of 2D spcae in 3D space is:[[ 0.86667137 -0.23276482  0.44124968]
 [-0.4962773  -0.4924792  0.71496368]]

```

Dimension reduction result:



Result in original 3D space:



2. GMM:

Run Fastmap(wordlist,distance,2) function with dataset above:

```

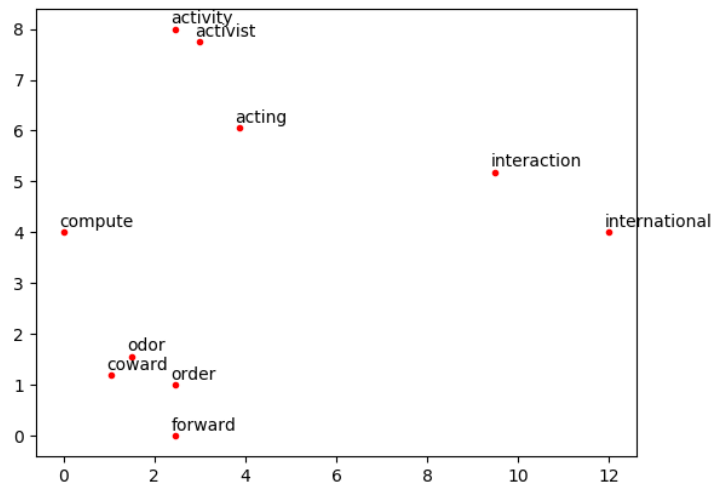
coordinates = Fastmap(wordlist,distance,2)
coordinates = np.array(coordinates)

```

Plot the result in 2D plane with the coordinates output:

```
plt.scatter(coordinates[:,0],coordinates[:,1],marker='.',color='r',label='1')
for i in range(len(wordlist)):
    plt.annotate(wordlist[i],xy = (coordinates[:,0][i],coordinates[:,1][i]), xytext = (coordinates[:,0][i]-0.1,coordinates[:,1][i]+0.1))
    """ plt.xlim(-1,13)
plt.ylim(-1,13) """
plt.show()
```

The 2D plane result:



3. Part 2: Software familiarization

1. Library function: sklearn.decomposition.PCA and code from China science

blog by Xu Shuo(pzczxs@gmail.com) <http://blog.sciencenet.cn/blog-611051-499019.html> or from another blog <http://gromgull.net/blog/2009/08/fastmap-in-python/>

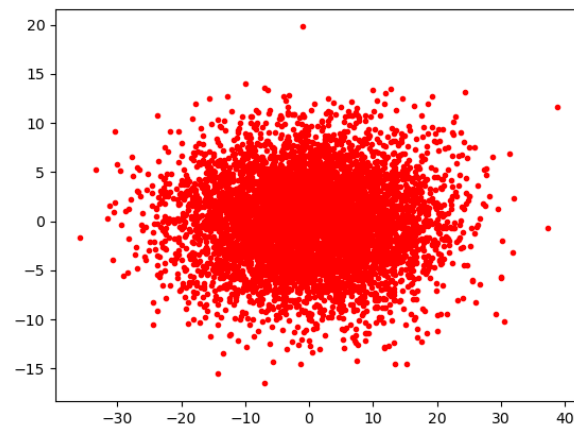
2. How to use:

1. PCA: Simply use `pca=PCA(n_components=k)` to choose the target dimension, Then use `pca.fit(data)` to implement pca, and use `pca.fit_transform` to get the dataset after dimension reduction.

```
pca = PCA(n_components=2)
ve = pca.fit(data)
print(pca.explained_variance_ratio_)
reduced_x=pca.fit_transform(data)    #dimen reduction

plt.plot(reduced_x[:,0],reduced_x[:,1], 'r.')
plt.show()
```

Result:

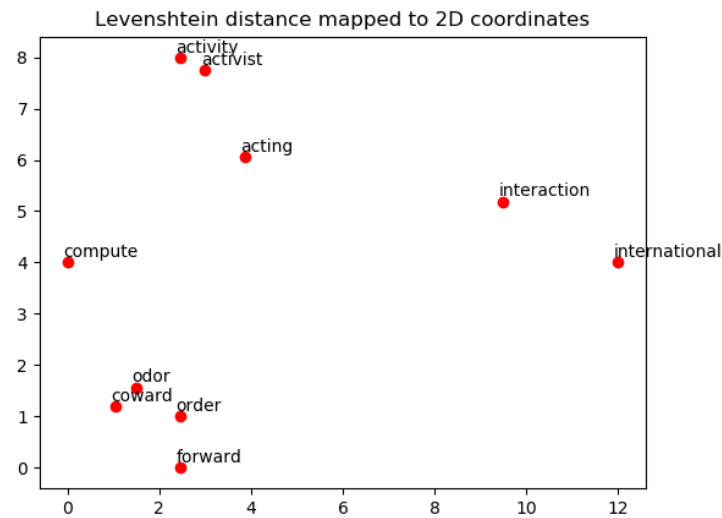


2. Fastmap: The code from blog I found is based on Python2.6 so some grammar should be modify to implement. But the logic is totally same as my implement.

Just directly change its dataset and use the dataset provided in this assignment.

```
def stringtest():  
    import Levenshtein  
  
    strings=["acting","activist","compute","coward","forward","interaction","activity","odor","order","international"]  
  
    dist=distmatrix(strings, c=lambda x,y: 1-Levenshtein.ratio(x,y))  
  
    p=fastmap(dist,2)  
    import pylab  
    pylab.scatter([x[0] for x in p], [x[1] for x in p], c="r")  
    for i,s in enumerate(strings):  
        pylab.annotate(s,p[i])  
  
    pylab.title("Levenshtein distance mapped to 2D coordinates")  
    pylab.show()  
  
if __name__=='__main__':  
    stringtest()  
    #distortiontest()
```

Result:



3. Comparison:

1. PCA: In `sklearn.decomposition.PCA`, it has many other parameters to provide more functions. For example, you can set parameter “whiten” or “svd_solver” to decide whether make dataset whiten or define the SVD method of singular value decomposition. These are what my implementation does have.

2. Fastmap: The code on blog set the max iteration as 2 and if can't found farthest pair after 2 iteration, it will restart with another random point. Maybe it will save some time in this part. But still in $O(N)$ time. And its plot has title.

3. Both 2 result between my implementation and code online is same.

4. Part 3: Applications

In data analysis problem, too high data dimension when the size of dataset is too small will cause the difficulty and the low precision of regression and finding the pattern of data. For example, too many other data like height, weight and if has any sprain

which has no obvious connection of coronavirus will cause difficulty of determining whether a patient infected by coronavirus. So we need to do some data filtering to simplify the data. PCA is a good choose because it can not only reduce the dimension, but also can see the information property which means the importance by checking the value of that eigenvalue. And Fastmap, which is part of MDS algorithm is usually used in data visualization by embedding data in to 3D space to see the approximate distribution. Because fastmap has a advantages of fast speed implementation, it's usually used to have a preliminary judgement of data distribution.