INF 552 HW2

Teammates:
He Chang 5670527576
Ziqiao Gao 2157371827
Fanlin Qin 5317973858

Part 1.

PCA

Data Structure：

Basically, we just read all the data points in and store them in an array. After computing the covariance matrix of the data points, we call the function **np.linalg.eig()** on the covariance matrix to get the set of eigenvectors and their corresponding eigenvalues.

Optimization:

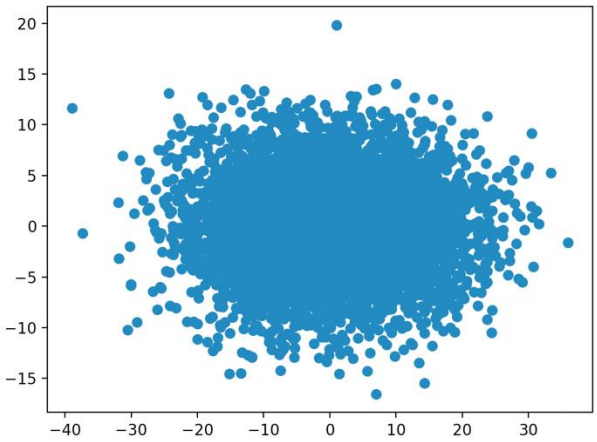
When sorting the eigenvalues, we use **ng.argsort()** to get the sorted order of the original eigenvalue indexes. Using the sorted indexes, we can get all sorted eigenvectors. Function **ng.argsort()** provides a quick method to sort eigenvectors based on the indexes of eigenvalues.

Challenge:

The major challenge is to find an efficient way of sorting eigenvectors based on the order of eigenvalues. Before using **ng.argsort()**, our self-defined method is pretty complicated in terms of code volume and logic of realization.

PCA Result:

```
[[ 10.87667009   7.37396173]
 [-12.68609992  -4.24879151]
 [  0.43255106   0.26700852]
 ...
 [ -2.92254009   2.41914881]
 [ 11.18317124   4.20349275]
 [ 14.2299014    5.64409544]]
```



FastMap

Data Structure

1. We use dictionary<point,dictionary<point, distance>> to store the original graph. For example, map[point_a][point_b] represents the distance between point_a and point_b. We only use this data structure to store the original distance that the txt file gives us, we are not going to use this structure to store any new distance or do any kind of iteration from it. This structure only helps us to do inquiry, constant time, when we need the original distance.
2. We use a two-dimensional list, *result_coordinate*, to store the xi value of each point. Every time we compute an xi for point i, we will put it into the result_coordinate[i-1], note that the index of result_coordinte starts with 0, but the index of actual point starts with 1. And this is efficient for us to inquiry the xi when computing new Distance, and the inquiry time is constant time.

Optimization

1. For xi distance, we use result_coodinate to keep track of it. This two-dimensional list provide an efficient inquiry and can be used to store the final result of this algorithm.
2. For original graph storage, we use dictionary<point,dictionary<point, distance>>. This structure can help us to do inquiry in constant time. It is very helpful, considering we need to get this value basically every time we need to calculate the distance.
3. For calculating new distance, we use recursion to calculate it, Distance_ith_iteration(i,j)^2 = Distance_i-1th_iteration(i,j)^2 - x(i,j)^2. The worst scenario of calculating a new distance is O(K), K is the dimension of new coordinate, in this HW, K = 2. And this recursion method will still make this algorithm a linear algorithm. For step

1, the time complexity is O(KN), step 2 is O(KN) either, therefore, it is still a linear time algorithm. This prevents us from calculating all D(i,j) every iteration which will make this algorithm a quadratic algorithm.

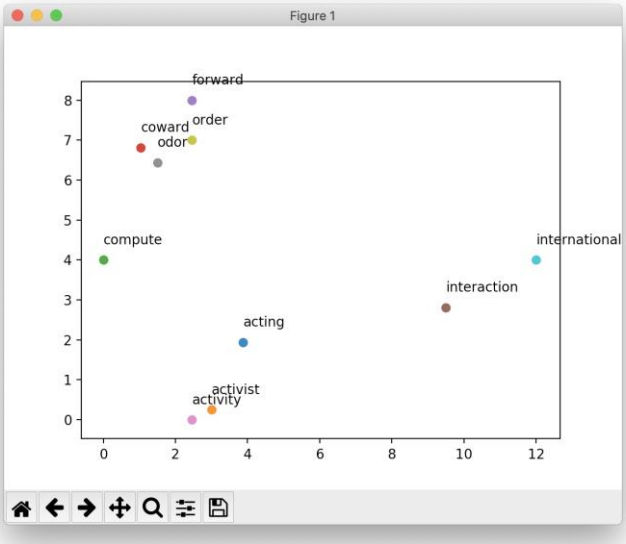
Challenge

1. We spent a long time implementing a linear time method to calculate new distances.
2. The optimized way to correctly store and retrieve generated coordinates took us some effort.

FastMap Result:

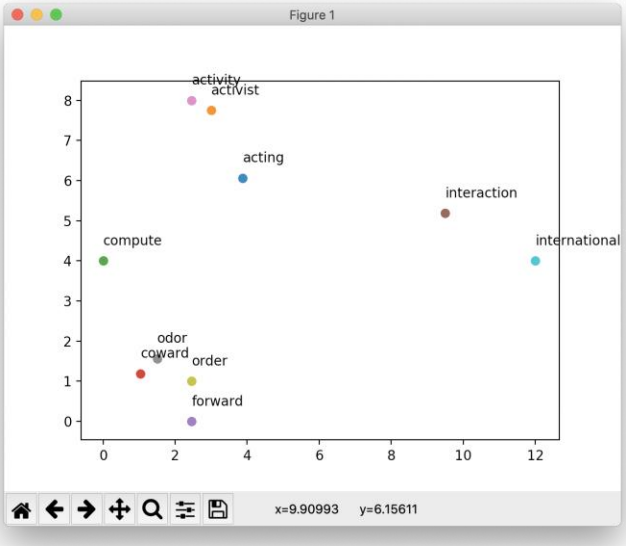
Result 1:

[3.875, 1.9375]
[3.0, 0.25]
[0.0, 4.0]
[1.04167, 6.8125]
[2.4583, 8.0]
[9.5, 2.8125]
[2.4583, 0.0]
[1.5, 6.4375]
[2.4583, 7.0]
[12.0, 4.0]



Result 2:

[3.875, 6.0625]
[3.0, 7.749999999999999]
[0.0, 4.0]
[1.0416666666666667, 1.1875]
[2.4583333333333335, 0.0]
[9.5, 5.1875]
[2.4583333333333335, 8.0]
[1.5, 1.5624999999999996]
[2.4583333333333335, 1.0]
[12.0, 4.0]



FastMap Summary:

1.We found two kinds of FastMap results, where the two different points distribution is axial symmetric along the line connected by object **compute** and **internationa**l. During the time when we pick the longest distance, the two end points **a** and **b** can both be used as the starting point for calculating Xi of other points, and the difference is caused by picking either **a** or **b**.

2.To see if our method is able to decrease the average distortion, we also embed objects to 3D and 4D planes and calculate distortion for each case.

Average Distortion:
2D plane: 1.3318311090338195
3D plane: 0.8195537489559723
4D plane: 0.3942941568522886

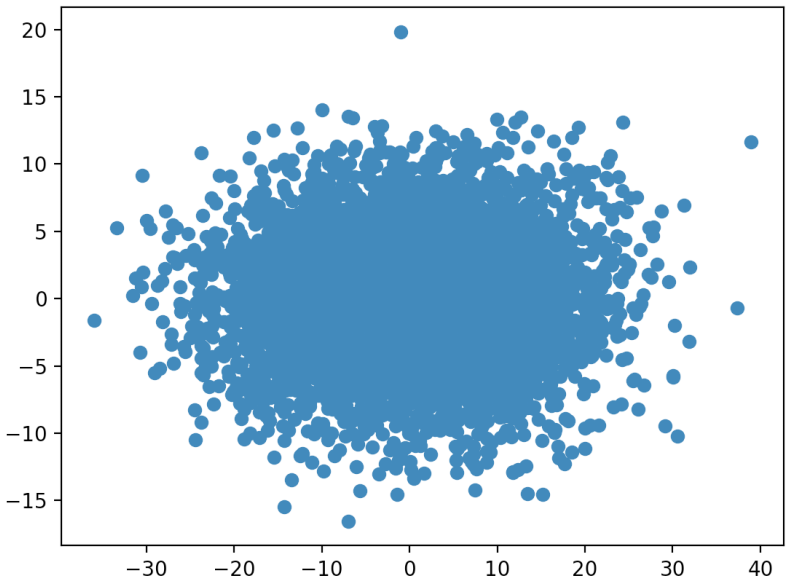
As the results show, the average distortion goes down as we go from 2D to 4D plane

Part 2.

PCA

Library used: from sklearn.decomposition import PCA
Result:

```
[[-10.87667009   7.37396173]
 [ 12.68609992  -4.24879151]
 [ -0.43255106   0.26700852]
 ...
 [  2.92254009   2.41914881]
 [-11.18317124   4.20349275]
 [-14.2299014    5.64409544]]
```



Summary:
This PCA function from sklearn is very simple to use with only a couple lines of code. Our discovery is that the results generated by the sklearn library are a little bit different from ours. The values of the first principle component are opposite. The research shows that the sklearn library uses Singular Value Decomposition to generate the eigenvalues and eigenvectors. The benefit of using SVD is that it does not require the shape of the covariance matrix to be in a square. Although there is a difference, the distance relationships among the points are the same.

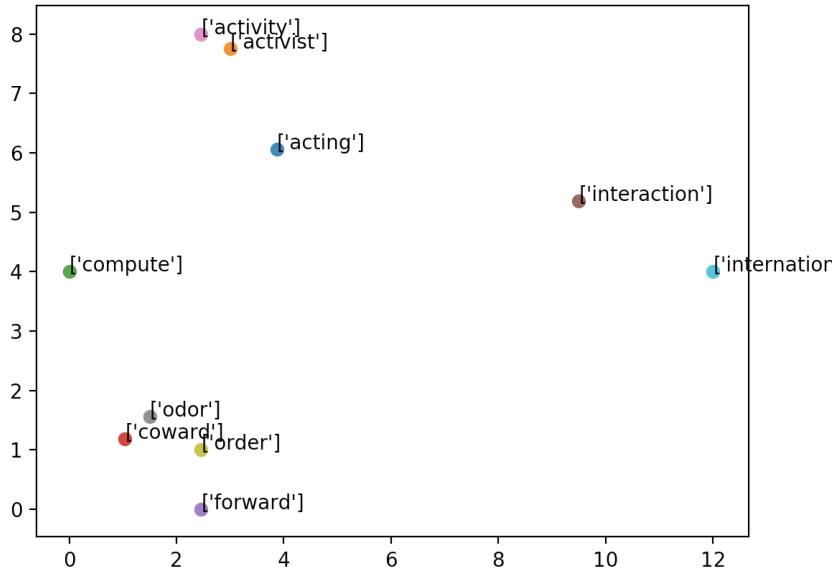
Fastmap

Library used: Unfortunately, we could not find any public library to implement FastMap algorithm, so we downloaded a FastMap implementation online and compare the results.

Citation: https://github.com/anupamish/FastMap/blob/master/FastMap.py

Result:

```
[[ 3.875      6.0625    ]
 [ 3.         7.75      ]
 [ 0.         4.        ]
 [ 1.04166667  1.1875    ]
 [ 2.45833333  0.        ]
 [ 9.5        5.1875    ]
 [ 2.45833333  8.        ]
 [ 1.5        1.5625    ]
 [ 2.45833333  1.        ]
 [12.         4.        ]]
```



Summary:
In general, the results are very similar. Sometimes our method will generate a slightly different points distribution, and we believe it normal. During the time when we pick the longest distance, the two endpoints **a** and **b** can both be used as the starting point for calculating Xi of other points, which will result in different coordinates. But the distance relationships among all the objects are the same.

Part 3.

1. Application of PCA to Image Compression

PCA can be used as a particle method to achieve dimensionality reduction of high-dimensional image feature data. It can be said that reconstructing the image can use less information based on the periodicity of the first principal component. However, the accuracy of image is not reduced after dimensionality reduction. The use of Singular Value Decomposition in the library reduces the complexity of algorithm processing.

2. Application of FastMap in Human Action Recognition

Human action recognition is one of the most popular topics in computer vision. However, it is very difficult to recognize human action accurately because of variable actions, speed. It is a complex task. In this paper, using FastMap mapping method associates the dynamic raw silhouettes with static intrinsic activity spaces, as well as dimensionality reduction, in order to obtain much of geometric structure of actions. FastMap algorithm can provide a fast and simple method in many situations.

[1]Wilmar H, Alfredo M, *“Application of Principle Component Analysis to Image Compression ”*, DOI: 10.5772/intechopen.72139, Nov. 2018
[2]Belhadj L.C, Mignotte M, “Spatio-temporal fastmap-based mapping for human action recognition”, *2016 IEEE International Conference on Image Processing(ICIP)*, 25-28 Sept. 2016

Individual contribution:
Ziqiao Gao: Optimize, implement and debug the PCA and FastMap algorithm for part 1

He Chang: Optimize, implement and debug the PCA and FastMap algorithm for part 1

Fanlin Qin: Research online and use open source python library to implement PCA and FastMap algorithm applications and complete part 2&3.