DSCI 552: Machine Learning for Data Science

Programming Assignment 3: PCA & FastMap

Team members: Li An, Shengnan Ke

Due Date: 02/27/2022

Contribution:

We followed the same procedure as last week. We reviewed the lecture together, checked some online resources, and had some discussions to help both of us get a better understanding of PCA and FastMap. Then each of us worked on the implementation of one algorithm. Our individual contributions are listed below. Then we write the report together and help each other to check on the algorithms that each of us implemented.

Li An: FastMap

Shengnan Ke: PCA

Part 1: Implementation

- ### PCA

Step 1: Load the dataset.

Load each column from the txt file as x, y, z coordinates.

```
# Step-1: load data as x,y,z coordinates

pcadata = np.loadtxt('pca-data.txt')

x_data, y_data, z_data = pcadata[:, 0], pcadata[:, 1], pcadata[:, 2]
```

Step 2: Center the data.

Subtract the mean of each variable and create a new list for x,y,z coordinates.

```
# Step-2: Center of the data

x_mean, y_mean, z_mean = x_data.mean(), y_data.mean(), z_data.mean()

new_x_data, new_y_data, new_z_data = x_data - x_mean, y_data - y_mean, z_data - z_mean

# create the data matrix

data_matrix = np.vstack((new_x_data, new_y_data, new_z_data)).T
```

Step 3: Compute the covariance matrix to identify correlations.

```
# Step-3: Compute the Covariance Matrix

cov_Matrix = np.cov(data_matrix, rowvar = False)
```

Step 4: Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components.

```
# Step-4.1: Compute the Eigenvalues and Eigenvectors

eigen_values , eigen_vectors = np.linalg.eig(cov_Matrix)
```

Sort Eigenvalues in descending order.

```
# Step-4.2: Sort Eigenvalues in descending order

sorted_index = np.argsort(eigen_values)[::-1]

sort_eigen_value = eigen_values[sorted_index]

# pair Eigenvalues with Eigenvectors

sort_eigen_vectors = eigen_vectors[:,sorted_index]
```

Step 5: Create a feature vector to decide principal components to keep.

Select a subset from the rearranged Eigenvalue matrix

```
# Step-5: select the subset from sorted eigen-vectors

# since we want to reduce the dimension to 2D dataset - selected the first two
principal components

n_components = 2

eigen_vectors_subset = sort_eigen_vectors[:,0:n_components]
```

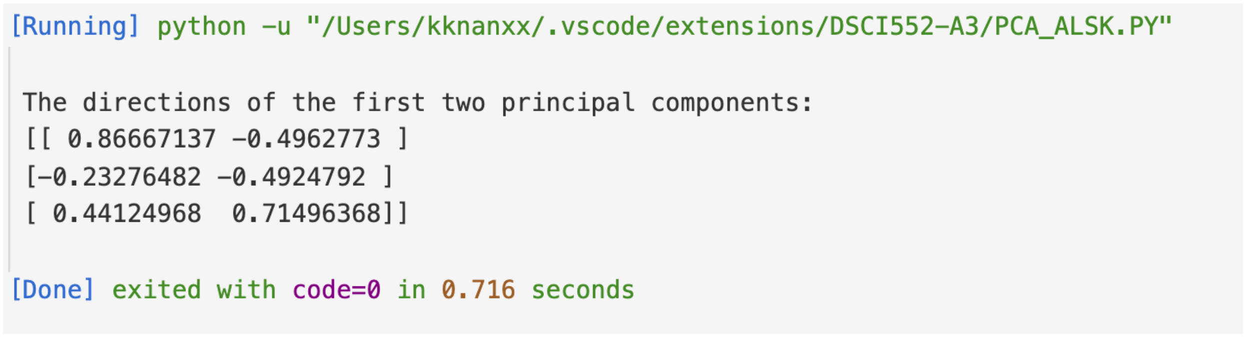
Step 6: Recast the data along the axes of the principal component - dimension reduction is done!

```
# Step-6: Constract the transformation matrix

Trans_matrix = np.dot(eigen_vectors_subset.transpose() , data_matrix.transpose()
).transpose()
```

Result:

the directions of the first two principal components.



```
[Running] python -u "/Users/kknanxx/.vscode/extensions/DSCI552-A3/PCA_ALSK.PY"

The directions of the first two principal components:
[[ 0.86667137 -0.4962773 ]
 [-0.23276482 -0.4924792 ]
 [ 0.44124968  0.71496368]]

[Done] exited with code=0 in 0.716 seconds
```

■ Fast Map

The pseudo-code below illustrates the main process of implementing the FastMap algorithm.

```
def fastMap(k):
    for k = 1, 2, ..., K:
        set a, b to the farthest pair of objects
        compute the kth dimension coordinate of each object
        update the distance function
```

Then we will dive into each function to see how it works.

Get domain-specific distance between object *i* and object *j* [`getDomainSpecificDist(self, i, j)`]:

We know that the first two columns in each line of the data file represent the IDs of the two objects, this function is to find out the domain-specific distance for each object.

Because that distance matrix is symmetric, we will first use an *if* statement to make *i* become the smaller index while *j* is the bigger one. Also, we add the condition when *i* is the same as *j* and the return distance will be zero.

Compute the distance between *i* and *j* [`computeDistance(self, i, j)`]:

As the distance function is changing after each iteration, we found the pattern of change. In each iteration, the distance function of two objects is subtracting all the differences on coordinates created by previous iterations from the domain-specific distance. We created an array to store coordinates already been calculated in the previous iteration, and use this follow the formula as the new distance function in the K^{th} iteration:

$$D^2_{new}(O'_i, O'_j) = D^2(O_i, O_j) - \sum_{k=1}^{K-1} (l_{ik} - l_{jk})^2$$

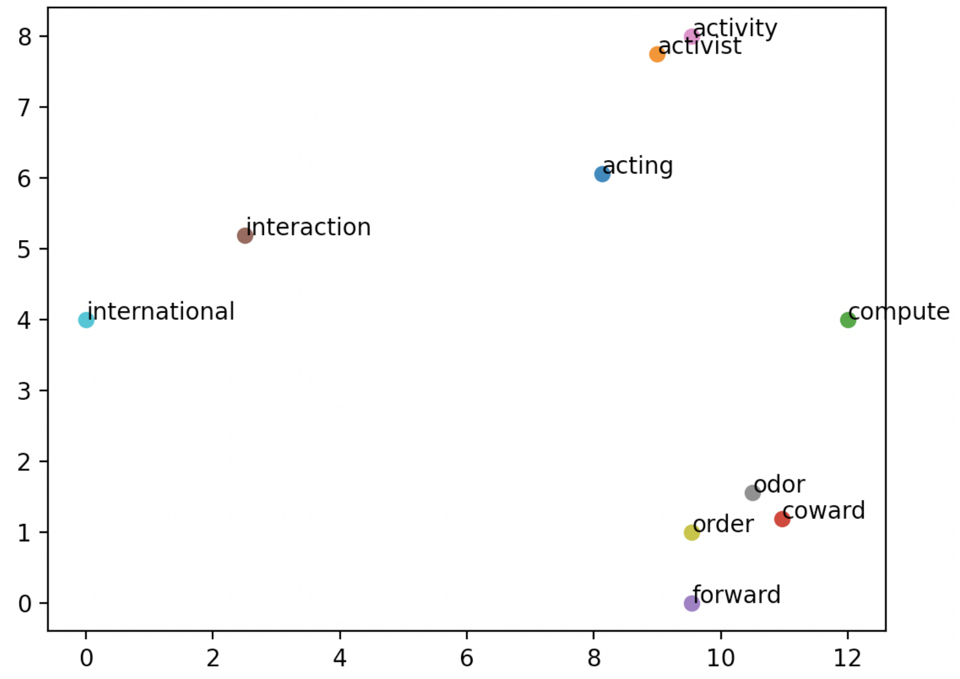
Pick the farthest pair of objects [`pickFarthestPair(self)`]:

First of all, we start from a random pivot. Next, we need to find out which object is the farthest from the current pivot. Repeat, until (1) the "farthest object" and the "current pivot" keep finding each other, which means it converged, and this pair is the farthest pair of objects; **OR** (2) we just allow it to change the pivot for 5 times, and return the last pair as the result even if it doesn't converge.

Run FastMap algorithm [`runFastMap(self)`]:

Main function implementing the FastMap algorithm.

Plot out the result [`plot(self)`]:



We plotted out objects according to coordinates learned by the algorithm, and also labeled them with the word according to the file *fastmap-wordlist.txt*. We found that the plot may vary with each time running the code, which is reasonable considering that there are some random factors in FastMap. Also, the plot will only flip horizontally or vertically each time running the code, and the relative distance between words will remain constant.

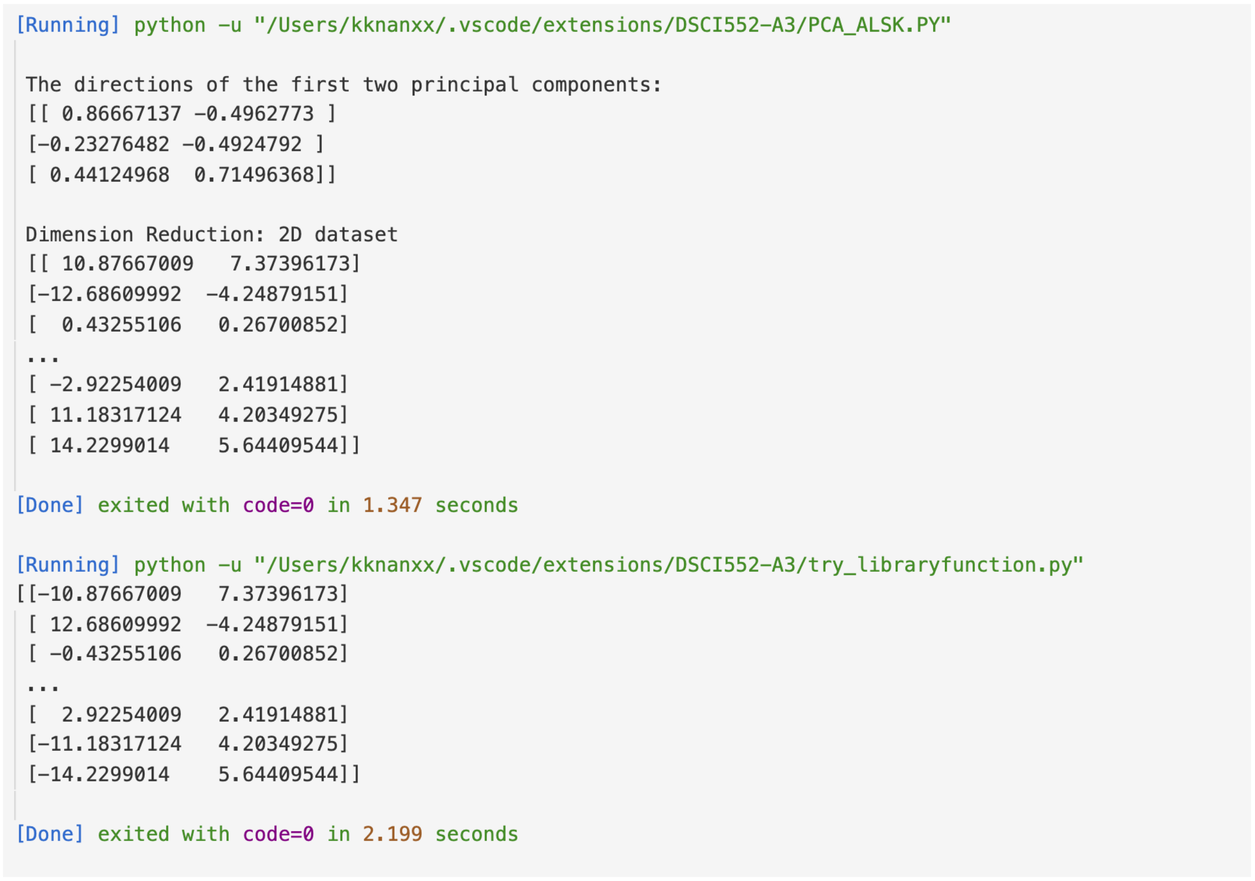
■ Challenges

1. It's a little bit difficult for us to understand what's the meaning of each record in the FastMap data file. But soon we figured out that it represents the domain-specific distance between words in the file *fastmap-wordlist.txt*.

2. It took us a while to come to the solution of the new distance function after each iteration. Then we found the pattern of change: the distance function of two objects is subtracting all the differences on coordinates created by previous iterations from the domain-specific distance.

Part 2: Software Familiarization

Sklearn is always the first place to go for machine learning algorithms library function.
In this assignment, to confirm our output, we also implement PCA with the sklearn library function. [`from sklearn.decomposition import PCA`]



```
[Running] python -u "/Users/kknanxx/.vscode/extensions/DSCI552-A3/PCA_ALSK.PY"

 The directions of the first two principal components:
 [[ 0.86667137 -0.4962773 ]
 [-0.23276482 -0.4924792 ]
 [ 0.44124968  0.71496368]]

 Dimension Reduction: 2D dataset
 [[ 10.87667009   7.37396173]
 [-12.68609992  -4.24879151]
 [  0.43255106   0.26700852]
 ...
 [ -2.92254009   2.41914881]
 [ 11.18317124   4.20349275]
 [ 14.2299014    5.64409544]]

[Done] exited with code=0 in 1.347 seconds

[Running] python -u "/Users/kknanxx/.vscode/extensions/DSCI552-A3/try_libraryfunction.py"
[[-10.87667009   7.37396173]
 [ 12.68609992  -4.24879151]
 [ -0.43255106   0.26700852]
 ...
 [  2.92254009   2.41914881]
 [-11.18317124   4.20349275]
 [-14.2299014    5.64409544]]

[Done] exited with code=0 in 2.199 seconds
```

As you can see, except for some minor deviations in the positivity and negativity of the coordinates, everything else is the same. By using the library function, we were able to get the same output with one line code. Compared to doing PCA implementation by ourselves which is much more convenient.

However, for fast-map, we were unable to find any library functions to work with.

Part 3: Applications

■ PCA

PCA is a popular tool for reducing the number of variables which is helpful for building machine learning models with a large number of variables.

■ FastMap

FastMap embeds a given non-negative edge-weighted undirected graph in a Euclidean space and approximately preserves the pairwise shortest path distances between vertices. This technique can be used for solving path-finding and multi-agent meeting problems.