Minh Tran – HW3: FastMap

1. **Part 1: Implementation:**

a. Write a program to implement FastMap Algorithm:

- Pycharm IDE is used with Python 3.6 as the programming language. Basic libraries such as numpy and random are used to facilitate the program
- Algorithm: (based on the original paper by Faloutsos et al. 1995)

Algorithm 2 *FastMap*
begin
Global variables:
$N \times k$ array **X**[ ] /* At the end of the algorithm, the i-th row is the image of the i-th object. */
$2 \times k$ pivot array **PA**[] /* stores the ids of the pivot objects - one pair per recursive call */
int col# =0; /* points to the column of the **X**[] array currently being updated */

Algorithm *FastMap*(k, $\mathcal{D}()$, $\mathcal{O}$)
1) if ($k \leq 0$)
{ return; }
else
{col# ++;}
2) /* choose pivot objects */
let O_a and O_b be the result of *choose-distant-objects*($\mathcal{O}$, $\mathcal{D}()$);
3) /* record the ids of the pivot objects */
PA[1, col#] = a; **PA**[2, col#]= b;
4) if ($\mathcal{D}(O_a, O_b) = 0$)
set **X**[i, col#] =0 for every i and return
/* since all inter-object distances are 0 */
5) /* project objects on line (O_a, O_b) */
for each object O_i,
compute x_i using Eq. 3 and update the global array: **X**[i, *col#*] = x_i
6) /* consider the projections of the objects on a hyper-plane perpendicular to the line (O_a, O_b); the distance function $\mathcal{D}'()$ between two projections is given by Eq. 4 */
call *FastMap*($k-1$, $\mathcal{D}'()$, $\mathcal{O}$)
end

Figure 5: Algorithm '*FastMap*'

- Data structure:

i. Fastmap-data.txt is converted to a 2D numpy array to represent a distance matrix between 2 object Oi and Oj with distancesMat[i][j] = distancesMat[j][i] = D(Oi, Oj)

ii. Fastmap algorithm is performed, which consists of three sub-segments of code:

- ✓ getFarthestPoints: given the distance matrix, find 2 points that are farthest from each other

Algorithm 1 *choose-distant-objects* ($\mathcal{O}$, *dist*())
begin
1) Choose arbitrarily an object, and let it be the second pivot object O_b
2) let O_a = (the object that is farthest apart from O_b) (according to the distance function *dist*())
3) let O_b = (the object that is farthest apart from O_a)
4) report the objects O_a and O_b as the desired pair of objects.

end

Figure 4: Heuristic to choose two distant objects.

- ✓ getProjectCoord: given the distance matrix, the line to project (consisting of two points) and the outer point; find the projecting coordinate based on the following formula:

$$x_i = \frac{{d_{a,i}}^2 + {d_{a,b}}^2 - {d_{b,i}}^2}{2d_{a,b}}$$

- ✓ recalcDistanceMat: recompute the distance matrix by the following formula after each dimension:

$$(\mathcal{D}'(O_i', O_j'))^2 = (\mathcal{D}(O_i, O_j))^2 - (x_i - x_j)^2 \quad i, j = 1, \ldots, N$$

iii. Visualization is performed by importing the word list and plotting the corresponding coordinates with the word itself as the annotation

iv. Files included:

- ✓ fastMapMT.py - Contains the implementation of fastMap from scratch
- ✓ fastmap-data.txt – Input distance data.
- ✓ fastmap-wordlist.txt – Input word list
- ✓ Hw3MinhTranReport.pdf – pdf report file

- Code-level optimization:
    i. The code is currently fast with only 10 data points given and 2 dimensions to embed.
    ii. Numpy array is used for faster access and computation.
    iii. Copy.deepcopy is used to store the original distance matrix
- Challenges:
    i. If there are a lot of data points given, we may have to maintain 2 large 2d array distancesMat along the way.
    ii. The code is only functional to plot 2-d target space.

b. Run fastmap program on data file:

- Distance Matrix: [10 by 10]

  [[0. 4. 7. 6. 7. 7. 4. 6. 6. 10.]
  [4. 0. 7. 7. 8. 9. 2. 8. 8. 11.]
  [7. 7. 0. 5. 6. 10. 6. 6. 6. 12.]
  [6. 7. 5. 0. 2. 10. 7. 4. 5. 12.]
  [7. 8. 6. 2. 0. 10. 8. 5. 4. 11.]
  [7. 9. 10. 10. 10. 0. 9. 10. 9. 4.]
  [4. 2. 6. 7. 8. 9. 0. 8. 8. 11.]
  [6. 8. 6. 4. 5. 10. 8. 0. 2. 12.]
  [6. 8. 6. 5. 4. 9. 8. 2. 0. 11.]
  [10. 11. 12. 12. 11. 4. 11. 12. 11. 0.]]

- Embedding coordinates: [10 by 2 or 10 words, each with 2 dimensions]

  [[3.875, 6.0625],
  [3.0, 7.749999999999999],
  [0, 4.0],
  [1.0416666666666667, 1.1875],
  [2.4583333333333335, 0],
  [9.5, 5.1875],
  [2.4583333333333335, 8.0],
  [1.5, 1.5624999999999996],
  [2.4583333333333335, 1.0],

[12.0, 4.0]]

- Final embedding coordinate of each given word:

  Embedding word "acting" to 2-D target space: [3.875, 6.0625]

  Embedding word "activist" to 2-D target space: [3.0, 7.749999999999999]

  Embedding word "compute" to 2-D target space: [0, 4.0]

  Embedding word "coward" to 2-D target space: [1.0416666666666667, 1.1875]

  Embedding word "forward" to 2-D target space: [2.4583333333333335, 0]

  Embedding word "interaction" to 2-D target space: [9.5, 5.1875]

  Embedding word "activity" to 2-D target space: [2.4583333333333335, 8.0]

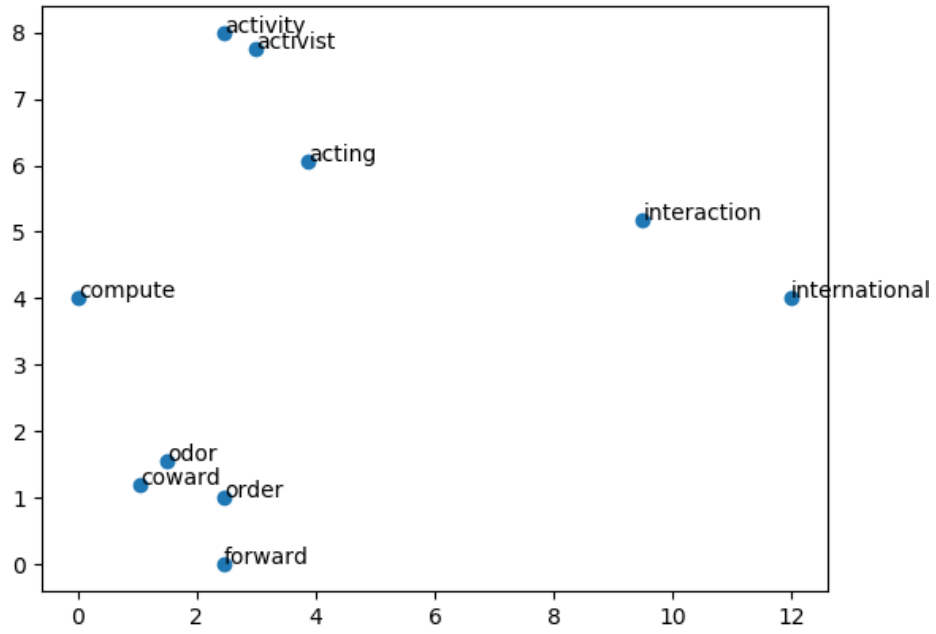
  Embedding word "odor" to 2-D target space: [1.5, 1.5624999999999996]

  Embedding word "order" to 2-D target space: [2.4583333333333335, 1.0]

  Embedding word "international" to 2-D target space: [12.0, 4.0]

- Plot in 2D space:

  The results are satisfactory as words with similar letters are grouped close to each other such as “activity-activist-acting”, “interaction-international”, “coward-forward”. However, word pair such as odor-coward are also grouped near each other, which might not be ideal.



c. Implement PCA:

- Pycharm IDE is used with Python 3.6 as the programming language. Basic libraries such as numpy and random are used to facilitate the program
- Algorithm:

Input: a dataset represented as $\boldsymbol{X}$, #components p we want

Step 1 Center the data by subtracting the mean

Step 2 Find the top p (unit norm) eigenvectors of the covariance matrix $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$, denote it by $\boldsymbol{V} \in \mathbb{R}^{\mathrm{D}\times p}$ (each column is an eigenvector).

Step 3 Construct the new compressed dataset $\boldsymbol{X}\boldsymbol{V} \in \mathbb{R}^{N\times p}$

- Implementation:

With $\boldsymbol{X} \in \mathbb{R}^{N\times \mathrm{D}}$ being the data matrix, we want

$$\max_{v:\|v\|_2=1} \boldsymbol{v}^{\mathrm{T}} \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right) \boldsymbol{v}$$

Let the max be $\lambda > 0$, then

$$\boldsymbol{v}^{\mathrm{T}} \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right) \boldsymbol{v} = \lambda = \lambda\, \boldsymbol{v}^{\mathrm{T}}\boldsymbol{v} = \boldsymbol{v}^{\mathrm{T}}(\lambda\, \boldsymbol{v})$$

It follows,

$$\boldsymbol{v}^{\mathrm{T}} \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{v} - \lambda\, \boldsymbol{v}\right) = 0$$

Conclusion: the first PC is the top eigenvector of the covariance matrix.

- Data structure:
  i. A tuple is used to represent a list of principal components: ('PC', ['idx', 'eigVal', 'eigVec']) with their corresponding index number, eigenvalue and eigenvector.
  ii. Run "python pcaMT.py pca-data.txt 2 pca-mt.txt
    - ✓ Pca(x, targetDims): takes the input data and desired target dimension to return the desired principal component vectors
    - ✓ PCAtransform(old, PCs): take the old data and the principal components to transform the old data to new data with PCA-based components.

iii. Files included:

- ✓ pcaMT.py - Contains the implementation of PCA from scratch
- ✓ pca-data.txt – 3D Input data.
- ✓ Pca-mt.txt – output file that recorded the old components and new PCA components of each data point

- Code-level optimization:
  i. Often, we have a large number of dimensions (say, 10,000) but a relatively small number of observations (say, 75). In that case, instead of directly computing the eigenvectors of x^T x (a 10,000 x 10,000 matrix), it's more efficient to compute the eigenvectors of x x^T (a 75 x 75 matrix) and translate these into the eigenvectors of x^T x by using the transpose trick. The transpose trick says that if v is an eigenvector of M^T M, then M^T v is an eigenvector of MM^T. I arbitrarily select "100" as the switching threshold.
  ii. Subtract the mean to center the data at each dimension and simplify the math during finding eigenvalues and eigenvectors of the covariance matrix.
  iii. Svd and eigh are used to find eigenvalues and eigenvectors
  iv. The rule of thumb is to choose the eigenvalues that can capture 90% the variation of the original data.
- Challenges:
  i. PCA is a linear method, it does not do much when every direction has similar variance. Therefore, kernel method needs to be combined with PCA if needed.

d. Run PCA program on data file:

- Principal Components: these vectors represent the directions in the original space with the most variations.
  i. Principal Component Vector 0:
  [-0.86667137 0.23276482 -0.44124968]
  with eigenvalue: 780.7798553351226
  ii. Principal Component Vector 1:

[-0.4962773 -0.4924792   0.71496368]

with eigenvalue: 345.5074412366038

- Transformed original data to PCA-compressed data

```
PCA: [ 5.90626285 -7.72946458  9.14494487] -> [-10.95314032   7.41375984]
PCA: [ -8.64032311   1.72426044 -10.69680519] -> [12.60962969 -4.2089934 ]
PCA: [0.25854061 0.23062224 0.76743916] -> [-0.50902129  0.30680664]
PCA: [-5.23435379  3.19468508 -1.89438474] -> [ 6.11597151 -0.33004127]
PCA: [12.62286294 -3.50788779  4.08625834] -> [-13.55944693  -1.61535229]
PCA: [7.85567057e-01 3.00747845e+00 1.89314774e-03] -> [ 0.01837134 -1.86962616]
PCA: [-13.8452367    6.07010838 -11.57755017] -> [18.52076821 -4.39585332]
PCA: [ 6.91713605 -0.20689451  4.91057749] -> [-8.20983228  0.17995818]
PCA: [ 5.8360168  -3.1783129   3.30339243] -> [-7.25532893  1.03077594]
PCA: [-10.47894405   5.9253877  -13.35735376] -> [16.35495064 -7.26769097]
PCA: [-3.53922532 -3.40528598 10.49223588] -> [-2.35498126 10.93503735]
PCA: [10.1288691  -0.69868362  0.55246716] -> [-9.18480575 -4.28764675]
PCA: [ 4.23111393  4.15919396 -1.72329486] -> [-1.93846797 -5.38021559]
PCA: [ 4.32399856 -4.22701531  5.13749368] -> [-6.99830364  3.6089362 ]
PCA: [-3.39913605  3.19737546 -1.65891444] -> [ 4.42216587 -1.07379043]
PCA: [-2.90884681  5.28542611 -2.69432678] -> [ 4.94014631 -3.0857136 ]
PCA: [-9.11787439 -3.12879894  1.5743598 ] -> [6.4792406  7.19147262]
PCA: [ 9.43054031 -6.03971993  5.43590229] -> [-11.97760373   2.18074607]
```

2. **Part 2: Software familiarization**

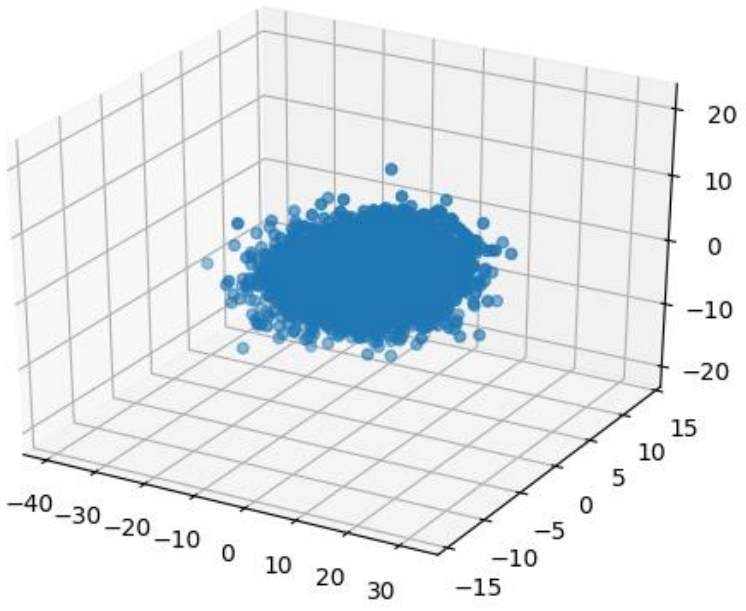
a. **Sklearn for PCA – pcaSklearnMT.py:**

- ✓ Scikit-Learn offers useful built-in methods to implement PCA.
- ✓ 3 simple steps are needed:
  - Import data
  - Create model and fit to data:
  - Run command: python pcaSKLearnMT.py pca-data.txt 2 pca-sklearn-mt.txt
- ✓ Output:

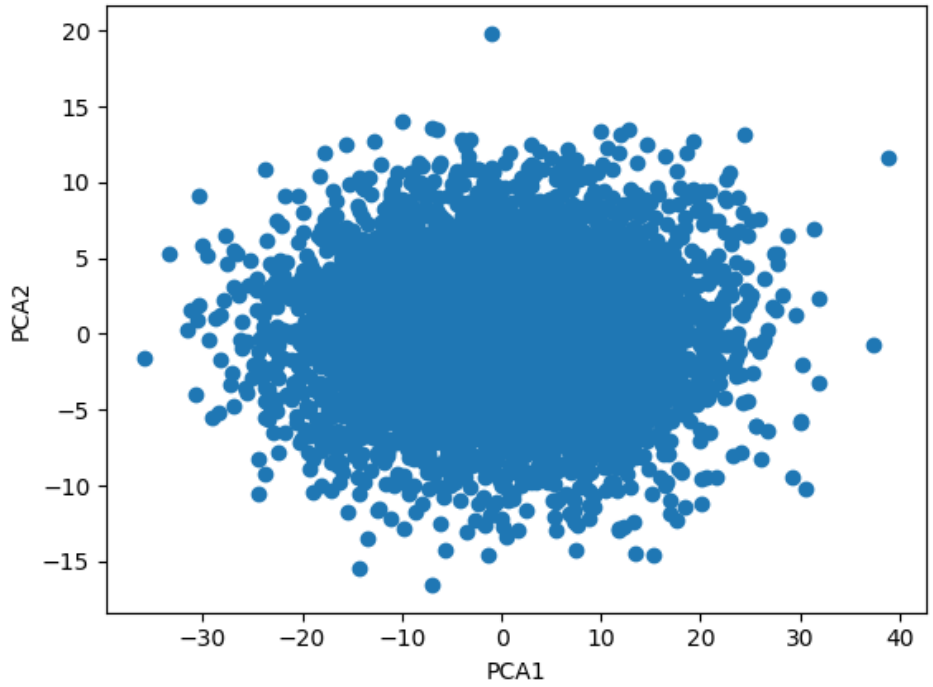
SKLearn Principal Component 0: [-0.86667137 0.23276482 -0.44124968]

SKLearn Principal Component 1: [-0.4962773 -0.4924792 0.71496368]

3D data is shown below:



2D compressed PCA-based data is shown below:



✓ The results are the same to my implementation of PCA

```
PCA: [ 5.90626285 -7.72946458  9.14494487] -> [-10.87667009   7.37396173]
PCA: [ -8.64032311   1.72426044 -10.69680519] -> [12.68609992 -4.24879151]
PCA: [0.25854061 0.23062224 0.76743916] -> [-0.43255106  0.26700852]
PCA: [-5.23435379  3.19468508 -1.89438474] -> [ 6.19244174 -0.36983938]
PCA: [12.62286294 -3.50788779  4.08625834] -> [-13.4829767   -1.65515041]
PCA: [7.85567057e-01 3.00747845e+00 1.89314774e-03] -> [ 0.09484158 -1.90942428]
PCA: [-13.8452367    6.07010838 -11.57755017] -> [18.59723844 -4.43565144]
PCA: [ 6.91713605 -0.20689451  4.91057749] -> [-8.13336204  0.14016007]
PCA: [ 5.8360168  -3.1783129   3.30339243] -> [-7.1788587   0.99097782]
PCA: [-10.47894405   5.9253877  -13.35735376] -> [16.43142087 -7.30748909]
PCA: [-3.53922532 -3.40528598 10.49223588] -> [-2.27851103 10.89523924]
PCA: [10.1288691  -0.69868362  0.55246716] -> [-9.10833551 -4.32744487]
PCA: [ 4.23111393  4.15919396 -1.72329486] -> [-1.86199773 -5.42001371]
PCA: [ 4.32399856 -4.22701531  5.13749368] -> [-6.9218334   3.56913808]
PCA: [-3.39913605  3.19737546 -1.65891444] -> [ 4.49863611 -1.11358855]
PCA: [-2.90884681  5.28542611 -2.69432678] -> [ 5.01661655 -3.12551171]
PCA: [-9.11787439 -3.12879894  1.5743598 ] -> [6.55571083 7.1516745 ]
PCA: [ 9.43054031 -6.03971993  5.43590229] -> [-11.90113349   2.14094796]
PCA: [-1.19842949 -5.34016013 14.31896657] -> [-6.44612608 13.42241415]
PCA: [-14.28944949   7.1896171   -6.25182177] -> [16.89283123 -0.95883108]
PCA: [-1.27818764  0.38588744  0.55920895] -> [1.02730911 0.80430995]
PCA: [ 5.07499587 -1.26387773  3.01388713] -> [-5.94594637  0.21884996]
```

3. Applications

FastMap is a powerful algorithm to maintain the given distance function while projecting objects into points in k-d space, using k feature-extraction functions. A domain expert is responsible for designing the feature extraction function to assess the similarity/distance of two objects. Thus objects can be plotted as points in 2-d or 3-d space, revealing potential clusters, correlations among attributes and other regularities that data mining is looking for. Tasks that can be run are finding objects similar to a given query object, finding pairs of objects that are most similar to each other, visualizing distribution of objects in some appropriately chose space. Application of fastMap can be found in multimedia databases where search-by-content is a highly desirable demand with selected feature vectors. It can also be used in medical database (brain scan, Xray, DNA matching) , time series (financial data, stock price, sale number), similarity searching in string database (spelling, typing, OCR error correction).

In quantitative finance, principal component analysis can be directly applied to the risk management of interest rate derivative portfolios. PCA has also been applied to equity portfolios in a similar fashion, both to portfolio risk and to risk return. One application is to reduce portfolio risk, where allocation strategies are applied to the "principal portfolios" instead of the underlying stocks. A variant of principal components analysis is used in neuroscience to identify the specific properties of a stimulus that increase a neuron's probability of generating an action potential.