Principal Component Analysis and FastMap

Programming Assignment 3

Patrick Humphries (pvhumphr@usc.edu)

University of Southern California

INF 552 Machine Learning for Data Science (32458)

Spring 2020

Abstract The FastMap method is used to project a graph to an Euclidean space. This provides a heuristic method to determine the distance from one node to another in near-linear time. However, the Euclidean space could be more than two dimensions. Principal Component Analysis can reduce the number of dimensions to two dimensions to enable the user to visualize principal components and see clusters and patterns. This paper describes the implementation of both Principal Component Analysis and FastMap

Fast Map

Finding the shortest path in a graph using Dijkstra Algorithm is at best polynomial time. The *FastMap Algorithm [T.K. Satish Kumar et al., 2017]* finds the shortest path in near linear time using a heuristic method. This is accomplished by transforming the graph into an Euclidean space and using the Pythagorean Theorem to estimate the shortest path. This paper discusses the implementation of the FastMap Algorithm for Programming Assignment 3.

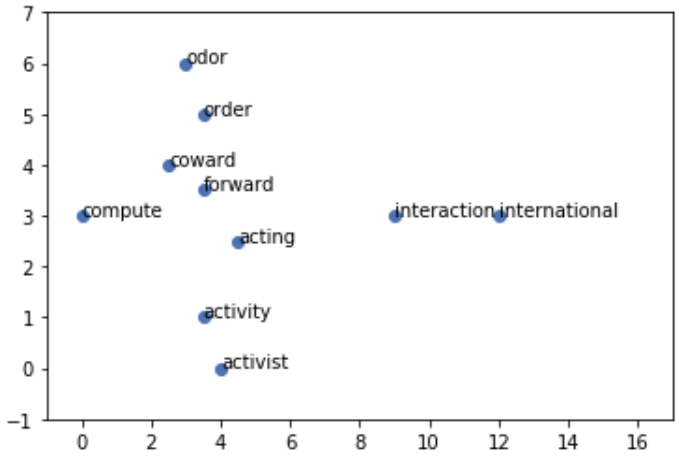
Requirement

Use FastMap to embed the objects in fastmap-data.txt into a 2D space. The first two columns in each line of the data file represent the IDs of the two objects; and the third column indicates the symmetric distance between them. The objects listed in fastmap-data.txt are actually the words in fastmapwordlist.txt (nth word in this list has an ID value of n). The distance between each pair of objects is the Damerau–Levenshtein distance between them. Plot the words on a 2D plane using your FastMap solution.

You can write your program in any programming language. However, you will have to implement the algorithms yourself instead of using high-level library functions - except for computing eigenvectors and eigenvalues. Provide a description of the data structures you use, any code-level optimizations you perform, and any challenges you face, and of course, the requested outputs.

Deliverable

vertex	spt	pu	pv	hypotenuse	difference
international	0	(12.0 , 3.0)	(12.0 , 3.0)	0.0	0.0
interaction	4	(12.0 , 3.0)	(9.0 , 3.0)	3.0	1.0
acting	10	(12.0 , 3.0)	(4.5 , 2.5)	7.5	2.5
activist	11	(12.0 , 3.0)	(4.0 , 0.0)	8.5	2.5
forward	11	(12.0 , 3.0)	(3.5 , 3.5)	8.5	2.5
activity	11	(12.0 , 3.0)	(3.5 , 1.0)	8.7	2.3
order	11	(12.0 , 3.0)	(3.5 , 5.0)	8.7	2.3
compute	12	(12.0 , 3.0)	(0.0 , 3.0)	12.0	0.0
coward	12	(12.0 , 3.0)	(2.5 , 4.0)	9.6	2.4
odor	12	(12.0 , 3.0)	(3.0 , 6.0)	9.5	2.5



The table lists the following columns.

- **vertex** This is the label of the data point.
- **spt** Shortest Path Tree(SPT): The number of edges from the start to goal.
- **pu** These are the coordinates of the start data point of the SPT that had the farthest separation from the goal.
- **pv** These are the coordinates of the goal data points.
- **hypotenuse** This is the heuristic for determining the distance from the start to the goal.
- **difference** This is the difference between the heuristic and the SPT.

The plot is the visualization of the data in the table. The label "international" identifies the start data point.

Algorithm

$Input:\ G = (V, E, w), K_{max}, and\ a\ threshold.$

$Output:\ K, and\ p_i \in \mathbb{R}^K\ for\ all\ v_i \in V.$

$w' = w\ and\ K = 0.$

While $K_{max} > 0$ do

Create a working graph: $G' = (V, E, w')$.

Run the Shortest Path Tree (SPT) algorithm with G' until the extreme vertices stabilize as well as the distance between these two vertices. This distance will become the base for the triangle for each vertex.

If the base is less than the threshold, quit.

For each vertex, do:

Run SPT from the left vertex of the base to the vertex. This will be the left side of the triangle for the vertex.

Run SPT from the right vertex of the base to the vertex. This will be the left side of the triangle for the vertex.

The value for the dimension, p_i, is calculated by $[p_v]_K = (left\ side + base - right\ side)/2.$

For each edge, decrement w'_K by $[p_v]$.

Increment K.

Decrement K_{max}.

This algorithm creates dimensions for each vertex, and applies a portion of the edge weight to the dimension. The process is repeated until all of the edge weights have been depleted by assigning portions to dimensions.

Data Structure

The graph G is composed of the following:

- V A dictionary of vertices key by the vertex number with the word as the vertex value.
- E A list of tuples that consist of adjacent vertex numbers.
- w A dictionary of edge weights keyed by edge tuples from E.

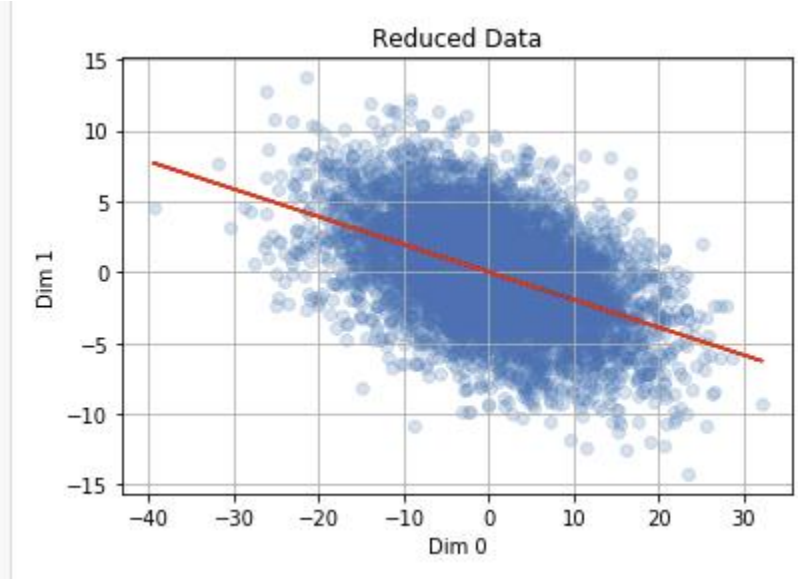
Principal Component Analysis

FastMap can project a graph into an Euclidean space. However this space could be more than two dimensions. To identify clusters and principal components, the data needs to be displayed in no more than two dimensions. This is done by dimension reduction: Find two dimensions that are closely correlated then ignore the dimension with the smaller variance. Data could still be visualized with little loss in accuracy.

Requirement

Use PCA to reduce the dimensionality of the data points in pca-data.txt from 3D to 2D. Each line of the data file represents the 3D coordinates of a single point. Please output the directions of the first two principal components.

Deliverable



The first Principal Component (PC1) is the red line. It has an angle of -11 degrees. PC2 would be perpendicular to it, so PC2 would be at 79 degrees.

Process

Instead of describing the algorithm, the step-by-step process is discussed. This is to demonstrate the author's understanding.

Load the Data

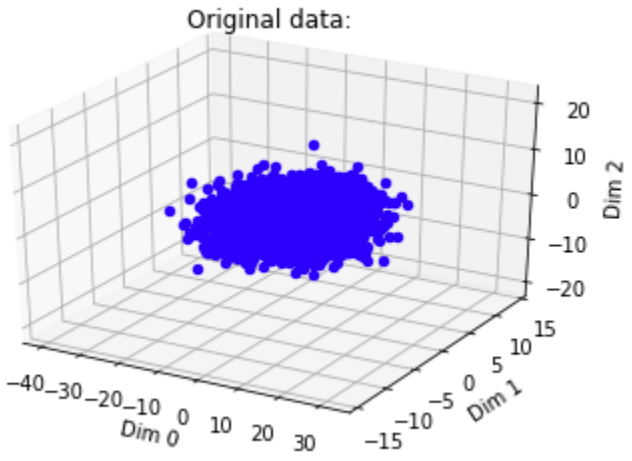
The data is loaded into two data structures. Each row of the file is loaded as a row into a Fact table. It is a dictionary with a tuple of dimension values as key and an arbitrary label of "Sample *" as value.

```
===== load_F() =====
Sample 0 attributes: (5.906262853951832, -7.729464584682111, 9.144944874608196)
Sample 1 attributes: (-8.640323106971573, 1.7242604350569888, -10.696805187953952)
Sample 2 attributes: (0.25854061492215674, 0.23062223968214718, 0.7674391638194876)
Sample 3 attributes: (-5.234353794193943, 3.194685075162285, -1.8943847407605263)
Sample 4 attributes: (12.62286294146571, -3.5078877906880543, 4.086258338102421)
Sample 5 attributes: (0.7855670569156168, 3.007478450451627, 0.001893147740198886)
Sample 6 attributes: (-13.845236700686861, 6.07010837676231, -11.57755017004028)
Sample 7 attributes: (6.9171360524662875, -0.2068945076277069, 4.9105774925259835)

===== load_D() ====

Dimensions:
Dim 0   values: 5.906262853951832       -8.640323106971573      0.25854061492215674
Dim 1   values: -7.729464584682111      1.7242604350569888      0.23062223968214718
Dim 2   values: 9.144944874608196       -10.696805187953952     0.7674391638194876
```

From the Fact Table, the Dimension table is created. It is a dictionary keyed by an arbitrary string representing each column of dimension values in the file. Think of the Dimension Table as the columns of the Fact Table. Above are just samples of the ~~excessive~~ data provided by the professor.

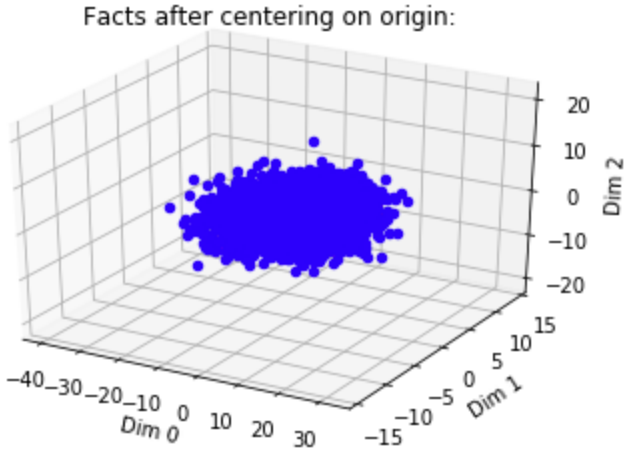
Plot the Data

The data with all three dimensions is plotted. This is just to provide an intuitive view of how the data is organized. It reminded me of the move titled "The Blob".



Center the Data

The data was centered on the origin. Each data point was adjusted by its mean. This is a convenience when calculating covariance since the mean will be zero in all of the calculations. With test data, the centering was meaningful. Not so with this blob.



Calculate Variances

The first step in dimension reduction is calculating variances. These variances become a part of the denominator when calculating correlation. The calculation was done without libraries. It was also done with *numpy* as a sanity check and to meet the software familiarization requirement.

```
==========
Variances:
        Calculated      numpy.var
Dim 0: 81.2              81.24
Dim 1: 13.7              13.698
Dim 2: 31.4              31.365
```

Calculate Covariances

The second step in dimension reduction is calculating covariances. The results of these calculations become the numerator in the correlation calculation. Again, the *numpy* function was used to verify the manufactured calculation.

```
==================
Covariances numpy:
        Dim 0   Dim 1   Dim 2
Dim 0   81.2    -15.8   31.7
Dim 1   -15.8   13.7    -15.3
Dim 2   31.7    -15.3   31.4

=======================
Covariances calculated:
        Dim 0   Dim 1   Dim 2
Dim 0   81.2    -15.8   31.7
Dim 1   -15.8   13.7    -15.3
Dim 2   31.7    -15.3   31.4
```

Calculate Correlation

The correlation between two dimensions is the covariance of the two dimensions divided by the square root of the product of the variances of the two dimensions:
$\rho_{x,y} = cov(x,y)/\sqrt{v_x v_y}$
where rho is the correlation between dimensions, covariance is $cov(x,y)$, and the variances are v_x and v_y. Again, the correlations were calculated with and without *numpy*.

```
==================
Correlations numpy:
        Dim 0   Dim 1   Dim 2
Dim 0   1.0     -0.47   0.63
Dim 1   -0.47   1.0     -0.74
Dim 2   0.63    -0.74   1.0

Largest Correlation (Dim 0, Dim 2): 0.63
Corresponding Variances: 81.2 31.4

========================
Correlations calculated:
        Dim 0   Dim 1   Dim 2
Dim 0   1.0     -0.47   0.63
Dim 1   -0.47   1.0     -0.74
Dim 2   0.63    -0.74   1.0

Largest Correlation (Dim 0, Dim 2): 0.63
Corresponding Variances: 81.2 31.4
Attribute to ignore based on variance: Dim 2
```

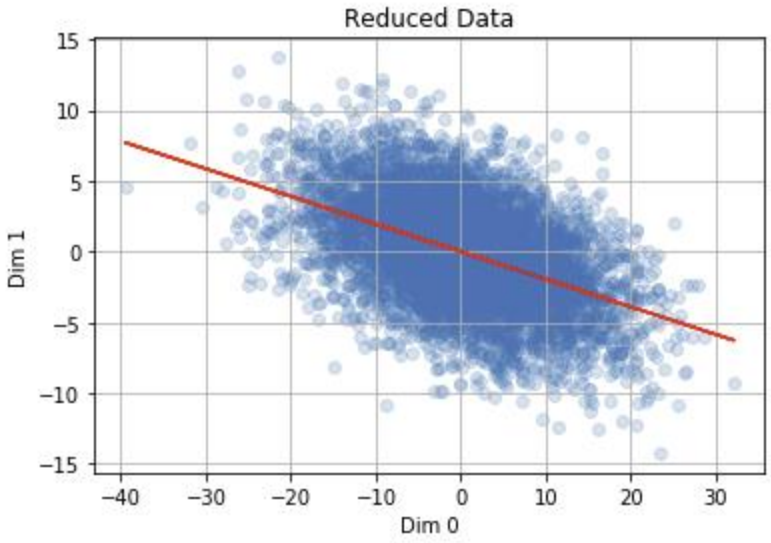
Identify Dimension to Ignore

As you can see from the output from correlation, *Dim 0* and *Dim 2* have the largest, albeit weak, correlation. Since *Dim 2* has the smaller variance, it was selected to be ignored. *Dim 0* with have more of an effect on the plot. Show here are the Fact Table before and after dimension reduction. Again, these are just sample rows.

```
Fact table prior to dimension reduction.
Number of samples: 5995
Sample 0 (5.9, -7.7, 9.0)
Sample 1 (-8.6, 1.7, -10.8)
Sample 2 (0.3, 0.2, 0.7)
Sample 3 (-5.2, 3.2, -2.0)
Sample 4 (12.6, -3.5, 4.0)
Sample 5 (0.8, 3.0, -0.1)
Sample 6 (-13.8, 6.1, -11.7)
Sample 7 (6.9, -0.2, 4.8)
Sample 8 (5.8, -3.2, 3.2)
Sample 9 (-10.5, 5.9, -13.5)
Sample 10 (-3.5, -3.4, 10.4)
Sample 11 (10.1, -0.7, 0.5)

Fact table after dimension reduction.
Number of samples: 5557
Sample 0 (5.9, -7.7)
Sample 1 (-8.6, 1.7)
Sample 2 (0.3, 0.2)
```

Plot the PC1

The plotting of a regression line going through the data and the origin yield an angle of -11 degrees. This is can bee seen in the visualization. The PC2 would be perpendicular at 79 degrees.



From this point, the data could be rotated left by 11 degrees. This would make PC1 horizonal and PC2 vertical. From the data in this position, eigenvalues and eigenvectors could be calculated.

Software Familiarization

For PCA, *sklearn* has a library to perform this function. However, during the building of the application, *numpy* was found to have functions that allow the creation of PC's in step-by-step fashion. This feature was applied to verify the calculated PC1.

A suitable library for FastMap was not found. Since the implementation was based on *FastMap Algorithm [T.K. Satish Kumar et al., 2017]*, it is suspected that all implementations are manufactured. As for finding the SPT, there are packages such as *Dijkstar* that find the shortest path, but likely not in near linear time.

Improvement

To improve performance, and visibility, a sample of data could be used for the PCA application. Using a random forest technique, a representative subset of the 5000+ records could be used, where individual data point could be identified.

Applications

Dijkstra's shortest path algorithm is used extensively in **Geographic Information Systems**. However, it is not well suited for large graphs *Algorithm for shortest path search in Geographic Information Systems by using reduced graphs [Rafael Rodiguez-Puete, et al, 2013]*.

Driving instructions offered by ***Google Maps*** is a candidate for FastMaps. When a user wants to know the best route, the application needs to find the destination and the relative direction.