NumPy / SciPy Recipes for Data Science: k-Medoids Clustering

Technical Report · February 2015

DOI: 10.13140/2.1.4453.2009

1 author:

Christian Bauckhage
University of Bonn
427 PUBLICATIONS **7,101** CITATIONS

SEE PROFILE

NumPy / SciPy Recipes for Data Science: k-Medoids Clustering

Christian Bauckhage
B-IT, University of Bonn, Germany
Fraunhofer IAIS, Sankt Augustin, Germany

***Abstract*—In this note, we study k-medoids clustering and show how to implement the algorithm using *NumPy*. To illustrate potential and practical use of this lesser known clustering method, we discuss an application example where we cluster a data set of strings based on bi-gram distances.**

I. Introduction

In a previous note [1], we looked at how to compute squared Euclidean distance matrices using *Python*. Here, we consider a data scientific application of distance matrices and discuss k-medoids clustering.

k-medoids clustering is a variant of k-means clustering that is also known as relational k-means [2], [3]. Similar to k-means, it minimizes distances between data points and cluster centroids; in contrast to k-means, it selects the centroids from among actual data points and works for any kind of data for which we can define a distance. k-medoids clustering therefore also applies to data that are not embedded in a vector space. Consider, for example, a problem in game analytics where we might want to cluster player names [4]. Even if names, i.e. strings of characters, do not allow for averaging, we may still compute a matrix of distances between strings and attempt to cluster based on this information only [5].

Our discussion below assumes that readers are passably familiar with *Python* and *NumPy* [6]. However, before we present a *NumPy* implementation of the k-medoids algorithm, we shall briefly review the underlying theory. Knowledgeable readers may safely skip to the next section.

II. Theory

To begin with, we recall the concept of the medoid of a set of data points and review the relation between k-means and k-medoids clustering. In what follows, we denote sets using calligraphic letters ($\mathcal{X}$), vectors using bold lowercase letters ($\boldsymbol{x}$), and matrices using bold uppercase letters ($\boldsymbol{D}$). To denote the i-th element of a set, we may write $\boldsymbol{x}_i \in \mathcal{X}$, $\mathcal{X}_i$, or $\mathcal{X}[i]$ interchangeably. Finally, if $\boldsymbol{D}$ is a matrix, we write $D_{i,j}$ to denote its (i,j) entry.

A. Means and Medoids

Without loss of generality, consider a sample

$$\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^m \tag{1}$$

of n data points where each $\boldsymbol{x}_i$ is an m-dimensional real valued vector. Being data scientists, we are of course familiar with the notion of the *sample mean*

$$\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{x}_i. \tag{2}$$

The *sample medoid*, on the other hand, appears to be less well known. It is given by

$$\boldsymbol{m} = \boldsymbol{x}_j = \underset{\boldsymbol{x}_l}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n} \left\|\boldsymbol{x}_l - \boldsymbol{x}_i\right\|^2 \tag{3}$$

and represents a rather intuitive idea. In a nutshell, the medoid $\boldsymbol{m}$ of $\mathcal{X}$ coincides with the data point $\boldsymbol{x}_j \in \mathcal{X}$ that is closest to the mean $\boldsymbol{\mu}$. To justify this statement, we state the following

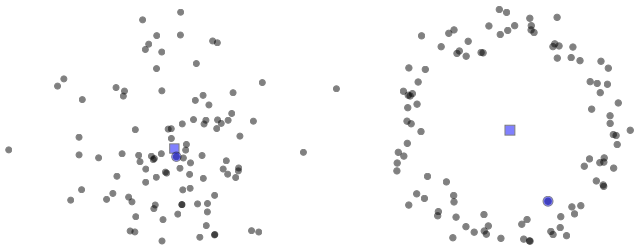
Lemma 1. *Given $\mathcal{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n\} \subset \mathbb{R}^m$, let $\boldsymbol{\mu} = \frac{1}{n}\sum_i \boldsymbol{x}_i$ be the sample mean and $\|\cdot\|$ be the Euclidean norm. Then*

$$\frac{1}{n}\sum_i \left\|\boldsymbol{x}_j - \boldsymbol{x}_i\right\|^2 \leq \frac{1}{n}\sum_i \left\|\boldsymbol{x}_k - \boldsymbol{x}_i\right\|^2 \tag{4}$$

implies that

$$\left\|\boldsymbol{x}_j - \boldsymbol{\mu}\right\|^2 \leq \left\|\boldsymbol{x}_k - \boldsymbol{\mu}\right\|^2. \tag{5}$$

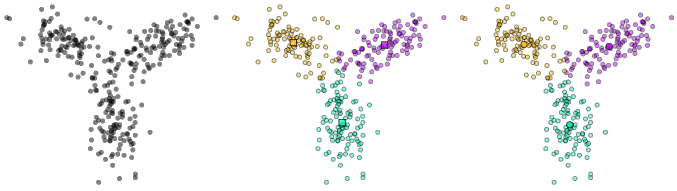
That is, the point $\boldsymbol{x}_j \in \mathcal{X}$ with the smallest average distance to all other points in $\mathcal{X}$ is closest to the sample mean $\boldsymbol{\mu}$.



(a) Gaussian data (b) non-Gaussian data

Fig. 1: Two data sets and their means (□) and medoids (○).

Given this result (which we prove in Section V), we now understand the behavior of the means and medoids in Fig. 1. Together, these examples point out a caveat for analysts who are working with centroid methods: for the Gaussian data in Fig. 1(a) as well as for the circular data in Fig. 1(b), the sample mean is located in the geometric center of the data. Yet, while in the Gaussian case it is close to the mode of the distribution, for the case where there is no clear mode, the mean is rather far



(a) set of 2D data points (b) k-means clusters (c) k-medoids clusters

Fig. 2: Simple data set consisting of three Gaussian blobs and results obtained from k-means and k-medoids clustering.

from most data. Accordingly, the medoid of the Gaussian data, too, is situated near the mode. However, in the non-Gaussian case, the medoid, i.e. the data point closest to the mean, is far from the mean and most data points. Whether or not these properties should be considered bugs or features will depend on the application context and is for the analyst to decide.

B. k-Means and k-Medoids Clustering

Having familiarized ourselves with means and medoids, we can look at k-means and k-medoids clustering. In the simplest setting, both approaches consider a set of n data points

$$\mathcal{X} = \{\boldsymbol{x}_i\}_{i=1}^{n},\ \boldsymbol{x}_i \in \mathbb{R}^m \tag{6}$$

and attempt to determine a set of k clusters

$$\mathcal{C} = \{\mathcal{C}_\kappa\}_{\kappa=1}^{k},\ \mathcal{C}_\kappa \subset \mathcal{X} \tag{7}$$

such that data points within a cluster are similar.

The popular k-means algorithm represents each cluster by its mean $\boldsymbol{\mu}_\kappa$ and assigns point $\boldsymbol{x}_i$ to cluster $\mathcal{C}_\kappa$ if $\boldsymbol{\mu}_\kappa$ is the closest mean. This idea reduces clustering to the problem of finding appropriate means. Since this may prove surprisingly difficult [7], k-means clustering is typically realized using the following greedy optimization procedure:

1) initialize cluster means $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \ldots, \boldsymbol{\mu}_k$
2) repeat until convergence
   a) determine clusters

$$\mathcal{C}_\kappa = \left\{ \boldsymbol{x}_i \;\middle|\; \left\|\boldsymbol{x}_i - \boldsymbol{\mu}_\kappa\right\|^2 \leq \left\|\boldsymbol{x}_i - \boldsymbol{\mu}_\lambda\right\|^2 \right\} \tag{8}$$

   b) update cluster means

$$\boldsymbol{\mu}_\kappa = \frac{1}{n_\kappa} \sum_{\boldsymbol{x}_i \in \mathcal{C}_\kappa} \boldsymbol{x}_i \tag{9}$$

Looking at this procedure, it appears obvious that k-medoids clustering can be realized analogously; all we need to do is to replace means by medoids:

1) initialize cluster medoids $\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_k$
2) repeat until convergence
   a) determine clusters

$$\mathcal{C}_\kappa = \left\{ \boldsymbol{x}_i \;\middle|\; \left\|\boldsymbol{x}_i - \boldsymbol{m}_\kappa\right\|^2 \leq \left\|\boldsymbol{x}_i - \boldsymbol{m}_\lambda\right\|^2 \right\} \tag{10}$$

   b) update cluster medoids

$$\boldsymbol{m}_\kappa = \boldsymbol{x}_j = \operatorname*{argmin}_{\boldsymbol{x}_l \in \mathcal{C}_\kappa} \frac{1}{n_\kappa} \sum_{\boldsymbol{x}_i \in \mathcal{C}_\kappa} \left\|\boldsymbol{x}_l - \boldsymbol{x}_i\right\|^2 \tag{11}$$

Figure 2 shows how both algorithms perform on a data set consisting of three blob-like components. Setting $k = 3$, both reliably identify these clusters; yet, while the means in Fig. 2(b) do not correspond to any data points, the medoids in Fig. 2(c) coincide with given data. Since cluster means and medoids will generally differ, both algorithms typically produce slightly different clusters as can be seen in the figure.

C. k-Medoids Clustering with Distance Matrices

To conclude the theoretical part of this note, we now recast the k-medoids clustering procedure in a more abstract manner.

We begin by noting that, contrary to the $\boldsymbol{\mu}_\kappa$ in k-means, the $\boldsymbol{m}_\kappa$ in k-medoids are guaranteed to coincide with data points so that k-medoids clustering exclusively relies on distances between data points. While this is obvious for (11), it also holds for (10), because if, say, $\boldsymbol{m}_\kappa$ coincides with $\boldsymbol{x}_j$, then

$$\left\|\boldsymbol{x}_i - \boldsymbol{m}_\kappa\right\|^2 \equiv \left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|^2.$$

Again in contrast to k-means, all distances evaluated during k-medoids clustering can therefore be precomputed and stored in a distance matrix $\boldsymbol{D}$ where

$$D_{i,j} = \left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|^2. \tag{12}$$

Hence, in order to implement k-medoids clustering, we may work with a distance matrix and sets of indices instead of sets of data points.

Let us thus assume an index set $\mathcal{I} = \{1, \ldots, n\}$ to refer to data points in $\mathcal{X}$ and an index set $\mathcal{K} = \{1, \ldots, k\}$ to refer to clusters in $\mathcal{C}$. Also, instead of $\mathcal{C}_\kappa \in \mathcal{C}$, we henceforth write $\mathcal{C}[\kappa], \kappa \in \mathcal{K}$ to emphasize the difference between sets of data points and sets of indices of data points. Finally, we introduce a set

$$\mathcal{M} \subset \mathcal{I},\ |\mathcal{M}| = k \tag{13}$$

containing the indices of those data points that coincide with medoids. Thus, if for cluster $\kappa \in \mathcal{K}$ we have $\mathcal{M}[\kappa] = j \in \mathcal{I}$, this is to say that $\boldsymbol{m}_\kappa = \boldsymbol{x}_j$.

Then, given a distance matrix $\boldsymbol{D}$, the update step in (11) of the k-medoids algorithm can be recast as

$$\mathcal{M}[\kappa] = j = \operatorname*{argmin}_{l \in \mathcal{C}[\kappa]} \frac{1}{n_\kappa} \sum_{i \in \mathcal{C}[\kappa]} D_{l,i} \tag{14}$$

and the clustering step in (10) can be reformulated as

$$\mathcal{C}[\kappa] = \left\{ i \;\middle|\; D_{i,\mathcal{M}[\kappa]} \leq D_{i,\mathcal{M}[\lambda]} \right\} \tag{15}$$

where we applied indirect indexing. Consider, for instance, cluster $\mathcal{C}[\kappa]$; if its medoid $\boldsymbol{m}_\kappa = \boldsymbol{x}_j$, then $\mathcal{M}[\kappa] = j$ and

$$\left\|\boldsymbol{x}_i - \boldsymbol{m}_\kappa\right\|^2 = D_{i,\mathcal{M}[\kappa]} = D_{i,j} = \left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|^2.$$

At first sight, all these ideas and modification may seem confusing and complex. However, we shall see next that they actually allow for a seamless *Python* / *NumPy* implementation of k-medoids clustering.

Listing 1: k-medoids clustering

```
1  def kMedoids(D, k, tmax=100):
2      # determine dimensions of distance matrix D
3      m, n = D.shape
4
5      # randomly initialize an array of k medoid indices
6      M = np.sort(np.random.choice(n, k))
7
8      # create a copy of the array of medoid indices
9      Mnew = np.copy(M)
10
11     # initialize a dictionary to represent clusters
12     C = {}
13
14     for t in xrange(tmax):
15         # determine clusters, i.e. arrays of data indices
16         J = np.argmin(D[:,M], axis=1)
17         for kappa in range(k):
18             C[kappa] = np.where(J==kappa)[0]
19
20         # update cluster medoids
21         for kappa in range(k):
22             J = np.mean(D[np.ix_(C[kappa],C[kappa])],axis=1)
23             j = np.argmin(J)
24             Mnew[kappa] = C[kappa][j]
25         np.sort(Mnew)
26
27         # check for convergence
28         if np.array_equal(M, Mnew):
29             break
30
31         M = np.copy(Mnew)
32     else:
33         # final update of cluster memberships
34         J = np.argmin(D[:,M], axis=1)
35         for kappa in range(k):
36             C[kappa] = np.where(J==kappa)[0]
37
38     # return results
39     return M, C
```

III. Practice

In this section, we discuss a simple and straightforward *Python* / *NumPy* implementation of the k-medoids clustering algorithm. First, however, we briefly familiarize ourselves with the practical computation of the mean $\boldsymbol{\mu}$ and medoid $\boldsymbol{m}$ of a data sample $\mathcal{X}$.

In order for all our code snippets to work, we require the following imports:

```
import numpy as np
import numpy.random as rnd
import matplotlib.pyplot as plt
```

To produce 2D Gaussian data as shown in Fig. 1(a), we may resort to the function `multivariate_normal` and invoke it as follows

```
mean = np.array([0., 0.])
Cov = np.array([[1., 0.], [0., 1.]])
X = rnd.multivariate_normal(mean, Cov, 100).T
```

This creates a data matrix $\boldsymbol{X} \in \mathbb{R}^{2\times 100}$ consisting of Gaussian distributed column vectors whose ample mean $\boldsymbol{\mu}$ results from

```
mu = np.mean(X, axis=1)
```

In order to determine the sample medoid $\boldsymbol{m} = \boldsymbol{x}_j$, we first compute a squared Euclidean distance matrix $\boldsymbol{D}$

```
D = squaredEDM(X)
```

using any of the methods discussed in [1] and then execute

```
j = np.argmin(np.mean(D, axis=1))
me = X[:,j]
```

for which we encourage the reader to verify that it implements equation (3) using the *NumPy* function `mean` and `argmin`. Finally, for visual inspection of our results, we may use

```
plt.scatter(mu[0], mu[1], marker='s', s=60)
plt.scatter(me[0], me[1], marker='o', s=60)
plt.scatter(X[0,:], X[1,:], s=30)
plt.show()
```

to obtain a plot of mean, medoid, and data similar to Fig. 1(a).

Having seen how to compute medoids based on distance matrices, we can now elaborate on the implementation of the k-medoids clustering algorithm shown in Listing 1. To better appreciate this code, we shall discuss it line by line:

- arguments passed to `kMedoids` are an $n \times n$ distance matrix $\boldsymbol{D}$, the number k of clusters to be identified, and the maximum number $t_{\max}$ of iterations to be performed
- in line 6, we create an array to represent the set $\mathcal{M}$ of medoid indices introduced in (13); to initialize the array, we apply `choice` to randomly select k integers between 0 and $n-1$; we also sort the array using `sort` to facilitate later tests for convergence
- in line 9, we create a copy $\mathcal{M}_{\text{new}}$ of $\mathcal{M}$ which will be used in the update steps of the algorithm
- in line 12, we initialize a *Python* dictionary to represent the set of clusters $\mathcal{C}$ to be determined
- the `for` loop in line 14 is used to realize the iterative updates of clusters and medoids and we note our use of `xrange` which may buy efficiency should the loop be terminated early
- in line 16, we consider k columns of $\boldsymbol{D}$ indexed by the numbers in $\mathcal{M}$; that is, we look at an $n \times k$ submatrix of $\boldsymbol{D}$ where $D_{i,\mathcal{M}[\kappa]}$ indicates the distance between data point $\boldsymbol{x}_i$ and medoid $\boldsymbol{m}_\kappa$; for each row of this submatrix, we determine the column with the smallest entry using `argmin` and store the results in an array J
- in line 17, we iterate over the clusters in $\mathcal{C}$ and, in line 18, use `where` to represent cluster $\mathcal{C}[\kappa]$ as an array whose elemenets contain the indices of those elements in J that equal κ
- in line 21, we iterate over the clusters in $\mathcal{C}$ and invoke `ix_` in line 22 to compute a submatrix of $\boldsymbol{D}$ containing only rows and columns indexed by the elements in $\mathcal{C}[\kappa]$; from this submatrix, we determine the medoid of $\mathcal{C}[\kappa]$ and store it in $\mathcal{M}_{\text{new}}[\kappa]$
- in line 25, wet sort $\mathcal{M}_{\text{new}}$ so as to test for convergence; to this end, we compare arrays $\mathcal{M}$ and $\mathcal{M}_{\text{new}}$ in line 28 and terminate the loop if they are equal; otherwise, we set $\mathcal{M}$ to $\mathcal{M}_{\text{new}}$ and continue with the iterative updates
- finally, we note our use of *Python*'s `for ... else` construct in line 32; if the procedure did not converge within $t_{\max}$ steps, we need to compute one final update of the clusters in order to take into account the latest update of the medoids.

Listing 2: a data set of names

```
X = np.array(['abe simpson','apu nahasapeemapetilon',
   'barney gumbel', 'bart simpson','carl carlson',
   'charles montgomery burns','clancy wiggum',
   'comic book guy','disco stu','dr. julius hibbert',
   'dr. nick riveria','edna krabappel','fat tony',
   'gary chalmers','groundskeeper willie',
   'hans moleman','homer simpson','kent brockman',
   'krusty the clown','lenny leonard','lisa simpson',
   'maggie simpson','marge simpson','martin prince',
   'mayor quimby','milhouse van houten','moe syslak',
   'ned flanders','nelson muntz','otto mann',
   'patty bouvier','prof. john frink','ralph wiggum',
   'reverend lovejoy','rod flanders','selma bouvier',
   'seymour skinner','sideshow bob','snake jailbird',
   'todd flanders','waylon smithers'])
```

Listing 3: n-grams of a string

```
def nGrams(text, n):
    return map(''.join, zip(*[text[i:] for i in range(n)]))
```

IV. Application Example

Given our implementation of the k-medoids clustering algorithm, we will now consider a practical application of the method. To begin with, we recall that k-medoids clustering is a relational clustering approach because it solely depends on distances between data points and can thus be computed from a given distance matrix. We also note that k-medoids clustering is not restricted to Euclidean distances but works with arbitrary distances.

This allows for applications in contexts where we are dealing with data for which the notion of a mean is ill defined. Consider for instance, the data in Listing 2. It consists of an array of strings, each of which corresponds to a likely familiar name. If we are interested in clustering these names, we need to remind ourselves that strings do not form a Euclidean vector space and do not allow for the computation of means. However, there are numerous ideas as to how to define string distances so that relational clustering becomes applicable.

In the following, we resort to an idea discussed in a blog by Simon White[1] who proposes to consider *bi-gram similarities*.

A bi-gram is an n-gram where $n = 2$ and constitutes a popular concept in text- and speech processing. In order for this note to be self contained, we provide Listing 3 which shows how to compute n-grams of a string. It realizes a variant of a beautiful *Python* recipe by Scott Triglia[2] and when invoked as follows

```
nGrams('homer simpson', n=2)
```

produces the following list of bi-grams

```
['ho', 'om', 'me', 'er', 'r ', ' s', 'si',
 'im', 'mp', 'ps', 'so', 'on']
```

[1] http://www.catalysoft.com/articles/strikeamatch.html

[2] http://locallyoptimal.com/blog/2013/01/20/elegant-n-gram-generation-in-python

Listing 4: n-gram distance matrix

```
def nGramDistMatrix(X, n):
    m = len(X)
    D = np.zeros((m,m))
    for i in range(m):
        ngi = set(nGrams(X[i], n))
        lngi = len(ngi)
        for j in range(i+1,m):
            ngj = set(nGrams(X[j], n))
            lngj = len(ngj)
            lintersect = len(set.intersection(ngi,ngj))
            d = 1. - 2. * lintersect / (lngi + lngj)
            D[i,j] = d
            D[j,i] = d
    return D
```

TABLE I: Clustering results according to bi-gram distance

rod flanders	marge simpson	prof. john frink
clancy wiggum	abe simpson	dr. julius hibbert
gary chalmers	apu nahasapeemapetilon	dr. nick riveria
groundskeeper willie	barney gumbel	martin prince
hans moleman	bart simpson	
kent brockman	carl carlson	
milhouse van houten	charles montgomery burns	
ned flanders	comic book guy	
otto mann	disco stu	
patty bouvier	edna krabappel	
reverend lovejoy	fat tony	
selma bouvier	homer simpson	
seymour skinner	krusty the clown	
sideshow bob	lenny leonard	
todd flanders	lisa simpson	
waylon smithers	maggie simpson	
	mayor quimby	
	moe syslak	
	nelson muntz	
	ralph wiggum	
	snake jailbird	

Now, given two strings and their corresponding sets S_1 and S_2 of bi-grams, White proposes to consider the following bi-gram similarity

$$s(S_1, S_2) = \frac{2 \cdot |S_1 \cap S_2|}{|S_1| + |S_2|} \tag{16}$$

for which we note that $0 \leq s \leq 1$. We also observe that $s = 1$ if S_1 and S_2 are equal and that $s = 0$ if S_1 and S_2 are disjoint, i.e. do not share any bi-gram.

In order to turn this similarity measure into a distance, we simply define

$$d(S_1, S_2) = 1 - s(S_1, S_2) \tag{17}$$

such that $d = 0$ if S_1 and S_2 are equal and $d > 0$ otherwise.

Since *Python* readily provides the data type `set`, it is indeed trivial to compute a bi-gram distance matrix from an array of strings (see Listing 4) and we may attempt to cluster the names in `X` into, say, $k = 3$ clusters by means of

```
M, C = kMedoids(nGramDistMatrix(X, n=2), k=3)
```

Once `M` and `C` have been computed, we can look the resulting medoids using

```
print X[M]
```

and the corresponding clusters can be inspected by means of

```
for c in range(k):
    print X[C[c]]
```

Table I summarizes a result we obtained this way. Its three columns represent the three clusters that were identified and the names in the first row represent the corresponding cluster medoids. Looking at the table, we note that all the members of the inner Simpson family have been grouped together. This is comforting, because their names indeed share several bi-grams and should thus be considered similar. What is arguably pleasant about this particular outcome is that most of the (aspiring) academics of Springfield have been assigned to the third cluster. This can be attributed to the observation that their names share share bi-grams such as `'dr'`, `'. '`, or `'in'`.

V. Proof of Lemma 1

This section is for our mathematically inclined readers, who are interested in the proof of the lemma we briefly considered in section II.

Proof: To begin our proof, we note that we may subtract and add the sample mean $\boldsymbol{\mu}$ from and to each term on the left hand side (LHS) of (4) so that its overall value remains unchanged. That is, we may consider

$$\begin{aligned}\frac{1}{n}\sum_i \left\|\boldsymbol{x}_j-\boldsymbol{x}_i\right\|^2 &= \frac{1}{n}\sum_i \left\|\boldsymbol{x}_j-\boldsymbol{\mu}-\boldsymbol{x}_i+\boldsymbol{\mu}\right\|^2\\ &= \frac{1}{n}\sum_i \left\|(\boldsymbol{x}_j-\boldsymbol{\mu})-(\boldsymbol{x}_i-\boldsymbol{\mu})\right\|^2.\end{aligned}$$

Using this result and expanding each of the squared Euclidean distances in the sum, the LHS of (4) can then be written as

$$\begin{aligned}&\frac{1}{n}\sum_i \left(\left\|\boldsymbol{x}_j-\boldsymbol{\mu}\right\|^2+\left\|\boldsymbol{x}_i-\boldsymbol{\mu}\right\|^2-2(\boldsymbol{x}_j-\boldsymbol{\mu})^T(\boldsymbol{x}_i-\boldsymbol{\mu})\right)\\ =&\left\|\boldsymbol{x}_j-\boldsymbol{\mu}\right\|^2+\frac{1}{n}\sum_i \left\|\boldsymbol{x}_i-\boldsymbol{\mu}\right\|^2-2(\boldsymbol{x}_j-\boldsymbol{\mu})^T\frac{1}{n}\sum_i(\boldsymbol{x}_i-\boldsymbol{\mu})\\ =&\left\|\boldsymbol{x}_j-\boldsymbol{\mu}\right\|^2+\frac{1}{n}\sum_i \left\|\boldsymbol{x}_i-\boldsymbol{\mu}\right\|^2-2(\boldsymbol{x}_j-\boldsymbol{\mu})^T(\boldsymbol{\mu}-\boldsymbol{\mu})\\ =&\left\|\boldsymbol{x}_j-\boldsymbol{\mu}\right\|^2+\frac{1}{n}\sum_i \left\|\boldsymbol{x}_i-\boldsymbol{\mu}\right\|^2\end{aligned}$$

Since these arguments and algebraic manipulations also apply to the right hand side (RHS) of (4), we find that the inequality in (4) can be cast as

$$\left\|\boldsymbol{x}_j-\boldsymbol{\mu}\right\|^2+\frac{1}{n}\sum_i \left\|\boldsymbol{x}_i-\boldsymbol{\mu}\right\|^2 \leq \left\|\boldsymbol{x}_k-\boldsymbol{\mu}\right\|^2+\frac{1}{n}\sum_i \left\|\boldsymbol{x}_i-\boldsymbol{\mu}\right\|^2$$

which immediately implies that $\left\|\boldsymbol{x}_j-\boldsymbol{\mu}\right\|^2 \leq \left\|\boldsymbol{x}_k-\boldsymbol{\mu}\right\|^2$ as claimed. ■

VI. Summary and Caveats

Our goals with this note were to discuss k-medoids clustering, to present a *NumPy* implementation of the algorithm, and to show how it can be used to cluster data for which we cannot compute means. Since it appears as if k-medoids clustering is less well known among data scientists, we therefore hope to have demonstrated that it provides an intuitive and conceptually simple approach towards relational clustering, that is towards clustering based on distance information only.

However, before we conclude, we need to mention potential pitfalls regarding the *NumPy* implementation in Listing 1:

- the k-medoids implementation presented in this note is mainly of didactic rather than of practical value
- we do not recommend its use for problems that involve large amounts of data (e.g. $n > 1000$), because it has not been optimized for efficiency
- in particular, our implementation heavily relies on *fancy indexing* (e.g. in lines 16 or 22) which provides *NumPy* programmers with enormous flexibility but is known to be slow as it creates copies rather than views of the arrays that are indexed into; hence, for growing amounts of data, the distance matrix $\boldsymbol{D}$ may be too large to allow for timely execution of this code; we will address this issue in an upcoming note where we discuss more efficient implementations
- the initial medoids are determined in a random fashion (line 6); however, random initializations are typically a bad idea in centroid-based clustering [5], [7] as they may considerably delay the convergence of the procedure and may also impact the quality of the final clustering result; a video illustrating these negative effects of naïve initializations on k-means or k-medoids clustering can be found on YouTube[3]; this issue, too, will be addressed in an upcoming note.

References

[1] C. Bauckhage, "NumPy / SciPy Recipes for Data Science: Squared Euclidean Distance Matrices," researchgate.net, Oct. 2014, https://dx.doi.org/10.13140/2.1.4426.1127.

[2] L. Kaufman and P. Rousseeuv, "Clustering by Means of Medoids," in *Statistical Data Analysis Based on the L1 Norm and Related Methods*, Y. Dodge, Ed. Elsevier, 1987, pp. 405–416.

[3] B. Szalkai, "An Implementation of the Relational k-means Algorithm," *arXiv:1304.6899 [cs.LG]*, 2013.

[4] A. Drachen, C. Thurau, J. Togelius, G. Yannakakis, and C. Bauckhage, "Game Data Mining," in *Game Analytics – Maximizing the Value of Player Data*, M. Seif El-Nasr, A. Drachen, and A. Canossa, Eds. Springer, 2013, ch. 12.

[5] C. Bauckhage, A. Drachen, and R. Sifa, "Clustering Game Behavior Data," *IEEE Trans. on Computational Intelligence and AI in Games*, 2015, in press; https://dx.doi.org/10.1109/TCIAIG.2014.2376982.

[6] T. Oliphant, "Python for Scientific Computing," *Computing in Science & Engineering*, vol. 9, no. 3, 2007.

[7] D. Aloise, A. Deshapande, P. Hansen, and P. Popat, "NP-Hardness of Euclidean Sum-of-Squares Clustering," *Machine Learning*, vol. 75, no. 2, 2009.

[3] https://www.youtube.com/watch?v=9nKfViAfajY