STA 663 Final Project

Affinity Propagation

Reza Momenifar*and Levin Zhu†

April 27, 2020

Abstract

We implement the affinity propagation clustering algorithm developed in Frey and Dueck (2007) using Python. Affinity propagation works by passing messages between data points to find exemplars amongst them. Advantages of affinity propagation over other clustering methods such as k-means or k-centers include the ability to easily alter the number of clusters and significant computational ease. Furthermore, affinity propagation does not require an exact number of clusters to be specified *a priori*. The algorithm is deterministic, without random initialization of parameters. Finally, affinity propagation can be applied with non-standard measures of similarity. We apply our algorithm to several simulated and real datasets and explore its performance.

1 Background

In this project, we implement the "affinity propagation" clustering algorithm introduced by Frey and Dueck (2007). A common technique in clustering is to find cluster centers that optimize some similarity metric (e.g. Euclidean distance) between the cluster center and the data points assigned to it. In the case that these cluster centers are actual data points, these centers are known as "exemplars." The central idea of "affinity propagation" is to initially consider each point as an exemplar, then let all the data points share "messages" with each other regarding how suitable they are to be one another's exemplars (i.e. an "affinity" to choosing the other data point as an exemplar), until the most suitable exemplars are selected.

Affinity propagation exhibits several advantages over some competing algorithms that identify cluster centers. First, affinity propagation is agnostic to the number and position of clusters. By considering each point as a potential exemplar, the algorithm avoids certain pitfalls of other cluster center algorithms such as k-centers clustering, which requires the user to specify the number of centers and is sensitive to where the randomly chosen initial centers are located. Affinity propagation, on the other hand, recursively selects the best exemplars among the whole data-set. The algorithm also takes as an input the "preference" of each

*Civil and Environmental Engineering PhD Student, mohammadreza.momenifar@duke.edu
†Marketing PhD Student, levin.zhu@duke.edu

individual data point to be an exemplar. This enables some prior model specification, as well as the ability to tune the number of final exemplars.

A second important advantage is computational speed. As we'll describe in more detail in Section 2.1, the computations in affinity propagation are all simple ones and local to pairs of data points. When dealing with large datasets, this computational advantage can make a significant difference, and we expand more on comparisons with other algorithms in sections 3 and 4.

Affinity propagation also has the advantage of operating on nonstandard similarity metrics. In particular. While metric-space clustering techniques such as k-means require data to lie in a continuous space, affinity propagation may be applied to non-continuous similarity metrics, and even similarity measures that are not symmetric and/or do not satisfy the triangle inequality.

Frey and Dueck (2007) implement the algorithm on several applications, including identifying exemplar faces among a set of images, identifying genes in DNA sequences, finding the most efficient air travel routes, and identifying exemplar sentences in the main text of their paper. Through these applications, affinity propagation is shown to have superior performance in terms of speed and accuracy than k-centers clustering. We explore further applications herein.

The affinity propagation algorithm may be used in our own research in a couple ways. First, the algorithm is related to using clustering methods as an approach for dimension reduction (lossy compression) as well as with dealing with missing data. We supply an example of this type of analysis in Section 4.3 with environmental data. Second, the algorithm can be applied to sparse datasets, which may be useful in marketing contexts where, for instance, online consumer purchases are rare among other online browsing activity. We leave analysis of affinity propagation with sparse datasets for future work.

The rest of this paper will proceed as follows. Section 2 will describe the algorithm and outline some of the steps we made to optimize the algorithm for performance. Sections 3 and 4 discuss several applications on both simulated and real data sets, some of which are from the original paper and some of which are our own. We conclude with a discussion in section 5.

1.1 Using Our Algorithm

Before proceeding with the rest of the paper, we first outline how to use our project code. Our GitHub repository for this project can be found at this link: https://github.com/MReza89/Stat663_Spring2020_FinalProject_RezaLevin/tree/master/Final_Report_Package, where our source code, test code, and example applications may be found. To install our package via `pip`, run

```
pip install -i https://test.pypi.org/simple/
> AffinityPropagation-RezaLevin2020-pkg-DukePhDs==0.0.3
```

in the command line (note this should be one line), then call

```
from AffinityPropagation_RezaLevin2020_pkg import Src_AP_V13
```

in Python to use our functions. Alternatively, if the file "Src_AP_V13.py" under the "Source Code" folder in the GitHub repository has already been downloaded to the local working directory, simply call `import Src_AP_V13` in Python directly.

2 Algorithm

2.1 Description of the Algorithm

Affinity propagation has three core components, similarity, responsibility, and availability. The similarity $s(i,k)$ between two points i and k is defined by some metric such as Euclidean distance. Each data point also has a "preference" $s(k,k)$ that is chosen beforehand, and influences the final outcome. For example, setting the preference to be the minimum of all the similarities yields the smallest number of clusters, while setting it to equal the median of all the similarities yields a moderate number of clusters. The closer the preferences are to zero, the larger the number of clusters. A unique aspect of affinity propagation lies with this similarity component. In particular, the non-continuous similarity measures may be used, including those that are not symmetric and/or do not satisfy the triangle inequality.

Next, the responsibility $r(i,k)$ represents "the accumulated evidence for how well-suited point k is to serve as the exemplar for point i, taking into account other potential exemplars for point i", and is a message that is sent from i to k. Finally, the availability $a(i,k)$ represents "the accumulated evidence for how appropriate it would be for point i to choose point k as its exemplar, taking into account the support from other points that point k should be an exemplar", and is a message sent from point k to point i. The responsibility and availability messages are sent between all data points at each iteration, until either a) a final iteration is reached, b) changes in the messages fall below a threshold, or c) after the exemplar decisions remain the same after some number of iterations. At every iteration, the exemplar for a point i is taken as the point k that maximizes $a(i,k) + r(i,k)$ at that iteration. In essence, affinity propagation does two things concurrently: first, it finds points that are most suitable to be exemplars, and second, it finds for each point which exemplar it should be linked with.

To be more mathematically precise, the responsibility and availability are updated as follows. First, responsibility is computed using the rule

$$r(i,k) \leftarrow s(i,k) - \max_{k' \text{s.t.} k' \neq k} \{a(i,k') + s(i,k')\}, \tag{1}$$

where in the first iteration, the availabilities $a(i,k)$ are all initialized to zero. The intuition here is that point i will assign a higher responsibility to point k if it is more similar, with a penalty equaling the maximum combined availability and similarity of some other point k'.

Note here that each point has a "self-responsibility" $r(k,k)$ which is computed as its preference $s(k,k)$ minus the maximum similarity from other points.

Next, the availability is computed using the rule

$$a(i,k) \leftarrow \min \left\{ 0, r(k,k) + \sum_{i' \text{s.t.} i' \notin \{i,k\}} \max\{0, r(i',k)\} \right\}. \tag{2}$$

As we can see, the availability is only non-negative if there is combined evidence of having a high self-responsibility $r(k,k)$ and enough other points assigning it positive responsibilities. The max is considered because only positive responsibilities matter - when choosing an exemplar, we only need to worry about how well a point does in explaining other points.

A "self-availability" is also computed, as follows:

$$a(k,k) \leftarrow \sum_{i' \text{s.t.} i' \neq k} \max\{0, r(i',k)\}. \tag{3}$$

At each iteration, the messages that are passed between points are damped in order to avoid numerical oscillations. Thus, in the algorithm, a damping value λ is assigned such that at each iteration, the new value is λ times the current value plus $1-\lambda$ times the update value.

Finally, at any iteration, the exemplars are chosen by finding, for each point i, the value of k that maximizes $a(i,k) + r(i,k)$.

The pseudocode of a plain (using loops) implementation of the algorithm is displayed in Algorithm 1.

Algorithm 1 Affinity Propagation (Plain)

Input: The $n \times n$ similarity matrix S, with diagonals $S_{i,i}$ are set to the preference $s(i,i)$ $\forall i \in \{1,\ldots,n\}$.
Input: The damping factor $\lambda \in [0.5, 1)$.
Initialize availabilities and responsibilities:
$A \leftarrow \mathbf{0}_{n,n}$ ▷ Notation: $A^t_{i,k} = a(i,k)$ at iteration t
$R \leftarrow \mathbf{0}_{n,n}$ ▷ Notation: $R^t_{i,k} = r(i,k)$ at iteration t
$E \leftarrow \mathbf{0}_{n,n}$ ▷ Notation: $E^t_{i,k} = e(i,k)$ at iteration t
for $t = 1,\ldots,T$ **do**
    for $i \in \{1,\ldots,n\}$ **do**
        for $k \in \{1,\ldots,n\}$ **do**
            Compute responsibilities (with damping):
            $r(i,k) \leftarrow \lambda(r(i,k)) + (1-\lambda)\left(s(i,k) - \max_{k' \text{s.t.} k' \neq k}\{a(i,k') + s(i,k')\}\right)$
    for $i \in \{1,\ldots,n\}$ **do**
        for $k \in \{1,\ldots,n\}$ **do**
            Compute availabilities (with damping): ▷ Using $r(i,k)$ from previous for-loop
            if $i \neq k$ **then**
                $a(i,k) \leftarrow \lambda(a(i,k)) + (1-\lambda)\left(\min\left\{0, r(k,k) + \sum_{i' \text{s.t.} i' \notin \{i,k\}} \max\{0, r(i',k)\}\right\}\right)$
            else
                $a(k,k) \leftarrow \lambda(a(k,k)) + (1-\lambda)\left(\sum_{i' \text{s.t.} i' \neq k} \max\{0, r(i',k)\}\right)$
    Find exemplars:
    $E^t \leftarrow A^t + R^t$
    for $i \in \{1,\ldots,n\}$ **do**
        Exemplar for point $i \leftarrow \arg\max_k E_{i,k \in \{1,\ldots,n\}^t}$
    if E^t is close enough to E^{t-1} **then** ▷ E.g. by using *allclose()* from *numpy*
        Break.
    else
        Continue.
Output: List of exemplars and exemplar labels for each point.

2.2 Optimization for Performance

After profiling the first version of the code, which was written based on the plain implementation of Affinity Propagation as described in the original paper, we found some bottlenecks that significantly increase the computation time. In what follows, we mention these bottlenecks and explain how our new implementations significantly enhanced the optimization of the algorithm.

2.2.1 Computing the Similarity Matrix

The first step in the affinity propagation implementation is to compute the similarity matrix. Given that in the original paper the negative Euclidean distance was used as the metric for computing the similarity, we need to have a function that takes an input 2d array M ($n * m$; features in columns and observations in rows) and returns a square matrix S that each element represents this metric between the rows of input data.

The plain implementation was based on using looping for M as a collection of row vectors. Using broadcasting, we were able to compute S around 35 times faster. While this is a significant improvement, we should note that this approach is memory inefficient because we converted a 2d array to a 3d one and therefore increasing the number of features (columns) could impair the broadcasting optimization. Indeed, our below example shows that by increasing the number of columns from 4 to 12 (keeping number of rows the same) the speed up reduces from 35 times to 16 times.

To make the broadcasting implementation memory efficient, we need to eliminate the need for constructing the 3d array. To this aim, we consider the Euclidean distance formula:

$$\sum_{i=0}^{D-1}(x_i - y_i)^2 = \sum_{i=0}^{D-1} x_i^2 + \sum_{i=0}^{D-1} y_i^2 - 2\sum_{i=0}^{D-1} x_i y_i. \tag{4}$$

In this formula, x and y represent the corresponding rows in M and so we need to run this formula for all the pairs. Furthermore, D indicates the number of columns in M. We want to compute this formula using broadcasting which means that x and y should represent the same 2d array, M. Computing the first two terms would be straightforward using broadcasting and we just need to reshape one of the computed vectors to construct a $n * n$ array when we sum them up. The third term is equivalent to computing matrix multiplication $-2\ (x \cdot y^T)$.[1]

This memory efficient implementation could make the computation of S around 110 times faster compared to the plain one while it is not affected by increasing the number of features.

In Table 1, we compare the computation time and standard deviation of these three implementations applied to both a 1000×4 and a 1000×12 random matrix where each element is drawn from a standard normal distribution. The table also shows the speed up ratio from the two broadcasting methods time versus the plain implementation time. All three implementations yield the same similarity matrix.

[1] See https://www.pythonlikeyoumeanit.com/Module3_IntroducingNumpy/Broadcasting.html

Table 1: Computation Time for Similarity Matrix

Implementation	1000×4			1000×12		
	Time	S.D.	Ratio	Time	S.D.	Ratio
Plain	3.14 s	84.1 ms	-	3.75 s	85.3 ms	-
Broadcasting	45 ms	428 μs	35.18	102 ms	12.1 ms	16.98
Broadcasting and Memory Efficient	11.9 ms	277 μs	108.29	14.7 ms	792 μs	110.05

2.2.2 Computing the Responsibility Matrix

The second step in the Affinity Propagation algorithm is to compute the responibility matrix, R, using the following formula:

$$r(i,k) \leftarrow s(i,k) - \max_{k's.t.k' \neq k} \{a(i,k') + s(i,k')\}. \tag{5}$$

Unlike S, the function for calculating R is called in each iteration and so it is critical to make it as efficient as possible. The plain calculation of R was based on using nested loops where for each pair of (i,k), a temporary vector holds the summation of $A(i,:)$ and $S(i,:)$ vectors, i and k indices of this vector are populated with negative infinity (because responsibility matrix does not send message to itself and also $k' \neq k$) and finally the maximum element of the temporary vector is subtracted from the corresponding component in S and $r(i,k)$ is updated. Such a implementation has a high complexity ($O(n^3)$) our goal is to improve the speed using vectorization.

Given that S matrix is already available, we just need to construct a matrix for the second term, $\max_{k's.t.k' \neq k} a(i,k') + s(i,k')$. To this aim, we first store the sum of A and S in a temporary matrix. For each component of this matrix, (i,k), we want to find the maximum element in the entire row excluding the i'th and k'th columns. To exclude the i'th columns for all such pairs, we just need to replace the diagonal components of the temporary matrix with negative infinity. At this stage, we can compute the maximum element and its corresponding index in each row of the temporary matrix (the first maximum value in each row). For each row in the temporary matrix, all the columns have the same maximum value except the column corresponding to the index of the maximum value. To take care of this point (ensure that the k'th columns are also excluded), we replace the maximum value in each row of the temporary matrix with negative infinity and again compute the maximum element in each row of the updated temporary matrix (they basically represent the second maximum value in each row). Now we can build a matrix for $\max_{k's.t.k' \neq k} a(i,k') + s(i,k')$. We first initialize this square matrix with zero and then add the first maximum values to it by broadcasting (so that all the columns would have the same values). Then in each row, the element corresponding to the first maximum value is replaced with the second maximum value. Finally we add the similarity matrix to this matrix to obtain the responsibility matrix.

Table 2 shows the results from our experiments using a 500×4 and 1000×4 random normal matrix. The vectorized implementation of updating R is around 180 times faster than the plain nested loops one.

Table 2: Computation Time for Responsibility Matrix

Implementation	400×4			1000×4		
	Time	S.D.	Ratio	Time	S.D.	Ratio
Plain	2.02 s	26.9 ms	-	9.65 s	1.19 s	-
Vectorized	3.39 ms	338 μs	178.34	21.8 ms	2.04 ms	183.97

2.2.3 Computing the Availability Matrix

The third step in the Affinity Propagation algorithm is to compute the Availability matrix, A, using the following formulas for the off-diagonal and diagonal components:

$$a(i,k) \leftarrow \min\{0, r(k,k) + \sum_{i' s.t. i' \notin \{i,k\}} \max\{0, r(i',k)\} \tag{6}$$

$$a(k,k) \leftarrow \sum_{i' \neq k} \max(0, r(i',k)) \tag{7}$$

Like R, the availability matrix is updated at every iteration and so it is critical to write its updating function as efficient as possible. For the plain implementation of this function, we employ nested loops to compute all the components of A. For each pair of i and k, we check whether it points to a diagonal or off-diagonal component and then use the corresponding formula. To compute $\sum_{i' s.t. i' \notin \{i,k\}} \max(0, r(i',k))$, which is common for both diagonal and off-diagonal components, we create a temporary vector holding the entire k'th column of R, set the i'th and k'th elements to negative infinity and then compute the summation of the remaining positive elements of this vector.

While a diagonal component of A is updated directly with this summation, an off-diagonal one is populated with the minimum between zero and the sum of the summation and the k'th diagonal component of R.

Although this implementation is pretty readable, it is not computationally efficient and so our goal is optimize it with vectorization. For this aim, we first construct a matrix for $\sum_{i' s.t. i' \notin \{i,k\}} \max(0, r(i',k))$ term. We initialize a temporary square matrix with R and then remove the negative value components. To accommodate the $i' \neq k$ condition, the diagonal components of this matrix are filled with zero. Then we compute the sum along columns of this matrix, which still includes $i' = k$ row in each column, and store them in a temporary vector. This vector itself will be used later for the diagonal components of A. In order to take care of $i' \neq k$ condition, we subtract the positive $i' = k$ components of R in each column from the temporary vector by broadcasting and store it in the temporary matrix. Afterwards, we add the diagonal components of R to this temporary matrix by broadcasting (which is again stored in the temporary matrix) and then remove its positive values. At this stage, the off-diagonal components of the resulting temporary matrix represent the corresponding components of A. Finally we replace the diagonal components of the temporary matrix with the elements of the temporary vector and now this temporary matrix fully represents A.

This vectorized implementation can provide at least 225 times speed up compared to the plain one. This speed up would become higher as the size of input data increases. as our experiments shows that increasing the number of data points from 500 to 1000 would increase the speed up from 225 times to 300 times.

This vectorized implementation can become even more efficient by removing the steps for replacing the diagonal components of the temporary matrix with zero (at the beginning) and later replacing it with the diagonal components of R. To this aim, after initializing the temporary square matrix with R removing its negative value components, we replace the diagonal components of the temporary matrix with the diagonal components of R. Then we compute the sum along the columns of the temporary matrix (get a vector) and subtract the temporary matrix from it and store the result in the temporary matrix. At this stage, the off-diagonal components of R are present in the off-diagonal components of the temporary matrix and also the diagonal components of R are absent in the diagonal components of the temporary matrix. Now we can store the diagonal components of the temporary matrix in a temporary vector which will be used later for the diagonal components of A. Finally we remove the positive values of the temporary matrix and replace its diagonal components with the temporary vector. The resulting temporary matrix fully represents A.

This improved vectorization can update A around 1.5 times faster than the earlier vectorization and around 500 times compared to the nested loops implementation. In our experiments with 500×3 and 1000×3 random normal matrices, we show that all the three implementations yield the same output and then compare their run time. Table 3 gives the computation time results as well as the speed-up ratios.

Table 3: Computation Time for Availability Matrix

Implementation	500×3			1000×3		
	Time	S.D.	Ratio	Time	S.D.	Ratio
Plain	3.24 s	221 ms	-	22.6 s	721 ms	-
Broadcasting	5.91 ms	585 μs	224.62	29.7 ms	1.76 ms	304.57
Efficient Broadcasting	2.89 ms	149 μs	328.59	17.1 ms	583 μs	495.87

2.2.4 Pseudocode after Optimization

Algorithm 2 displays the pseudocode of our affinity propagation algorithm after the optimization steps described above.

It is worth mentioning that we initialize only one temporary matrix throughout our implementation and avoid copying the matrices by modifying them in place. Furthermore, we could not benefit from distributed computing or concurrency due to the nature of this algorithm in which the messages are communicated simultaneously between all the data points and so such a structure leads to communication overhead.

Algorithm 2 Affinity Propagation (Optimized)

Input: The $n \times n$ similarity matrix S, with diagonals $S_{i,i}$ are set to the preference $s(i,i)$ $\forall i \in \{1, \ldots, n\}$, using steps described in Section 1.
Input: The damping factor $\lambda \in [0.5, 1)$.
Initialize availabilities and responsibilities:
$A, R, E, Z \leftarrow \mathbf{0}_{n,n}$ ▷ Z is the designated temporary matrix
for $t = 1, \ldots, T$ **do**
    $R^t \leftarrow Update\_R(S, R^{t-1}, A^{t-1}, Z, \lambda)$ ▷ Using steps described in Section 2.2.2
    $A^t \leftarrow Update\_A(R^t, A^{t-1}, Z, \lambda)$ ▷ Using steps described in Section 2.2.3
    $E^t \leftarrow A^t + R^t$
    for $i \in \{1, \ldots, n\}$ **do**
        Exemplar for point $i \leftarrow \arg\max_k E_{i,k \in \{1,\ldots,n\}^t}$
    if E^t is close enough to E^{t-1} **then** ▷ E.g. by using *allclose()* from *numpy*
        Break.
    else
        Continue.
Output: List of exemplars and exemplar labels for each point.

2.2.5 Other Python Implementations

Through profiling our code, we also noticed that the code can run even faster by using some alternative ways for replacing the diagonal components and removing the positive values in a matrix. The code for our experiments is shown in the accompanying Python notebook.

3 Applications on Simulated Datasets

We test the Affinity Propagation algorithm on two simulated datasets. We then apply other clustering algorithms to these datasets and compare their performance with our algorithm.

3.1 Gaussian Data

We simulate clusters in $\mathbb{R}^2$ by drawing the first and second coordinates of 5 cluster centers from two uniform $(0, 10)$ distributions, and drawing 20 data points around each cluster center with Gaussian noise in both dimensions, using a variance of 0.5.

We apply six clustering algorithms to this simulated dataset: the affinity propagation algorithm we developed for this project, the affinity propagation algorithm from *sci-kit learn*, k-means, spectral clustering, agglomerative clustering, and Gaussian mixture models.[2] It should be noted that the number of clusters was not specified for affinity propagation, and using the default preference (i.e. median similarity) yielded the correct number of clusters. The simulated dataset is shown in Figure 1, along with the clusters. Each algorithm found the exact same clusters, though with varying computation times which are displayed in Table

[2]Each of these algorithms, aside from our own affinity propagation algorithm, were implemented using *scikit-learn*.

4. For this simple dataset, only spectral clustering took significantly longer than the other algorithms.

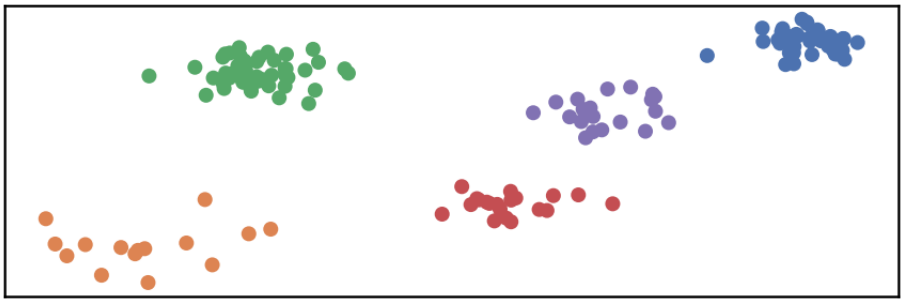

Figure 1: Gaussian Clusters

Algorithm	Computation Time
Affinity Propagation (our algorithm)	0.01 s
Affinity Propagation (*scikit-learn*)	0.02 s
K-Means	0.03 s
Spectral Clustering	0.06 s
Agglomerative Clustering	0.00 s
Gaussian Mixture Models	0.00 s

Table 4: Computation Time on Simulated Gaussian Clusters

3.2 Non-spherical Data

We also test the developed affinity propagation algorithm on non-spherical data. In particular, we sample two clusters in the shape of two different-sized rings. The simulated data and corresponding clusters using affinity propagation (with preferences set such that there are two resulting clusters), k-means, GMM, and DBSCAN (Density-Based Spatial Clustering of Applications with Noise) are shown in Figure 2. Only DBSCAN does well with this dataset.For such a dataset, we can transform the data from Cartesian coordinates to polar ones. This would help these algorithms to find the same clusters as found by DBSCAN algorithm.

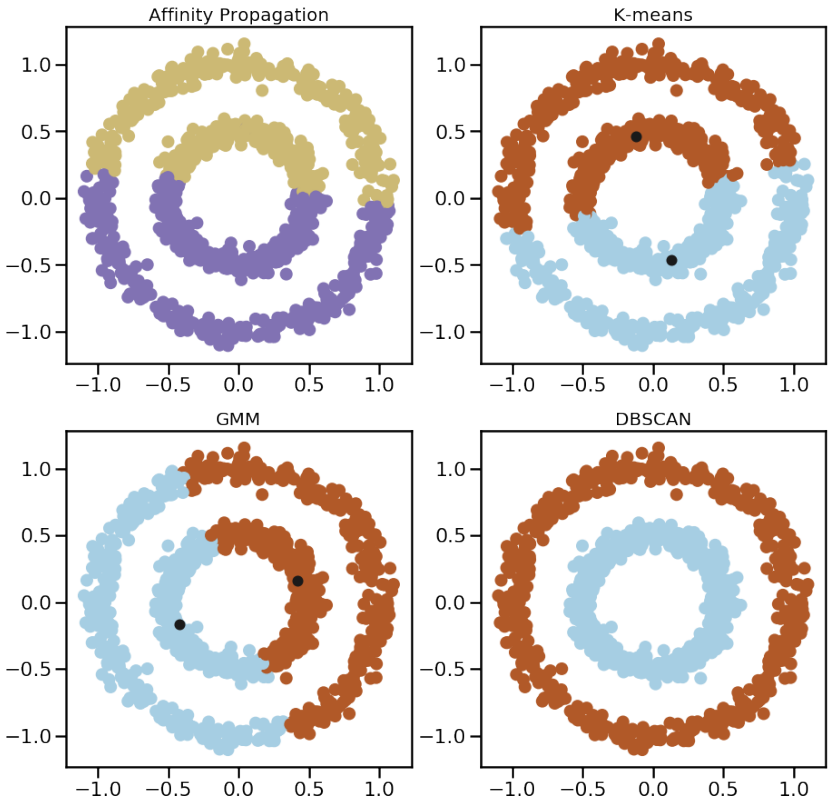

Figure 2: Clustering with Nonspherical Data

4 Applications on Real Datasets

We also apply affinity propagation on three different real-world datasets. The first two applications are from Frey and Dueck (2007), while the third application is an environmental application of affinity propagation that we introduce in this project.

4.1 Olivetti Faces Database

One of the applications of affinity propagation in Frey and Dueck (2007) is with "900 greyscale images extracted from the Olivetti face database." We download the Olivetti faces dataset from *scikit-learn*, which has a total of 400 samples from 40 distant subjects, ten slightly different pictures for each subject.

4.1.1 Fit Using Affinity Propagation

We apply our developed affinity propagation algorithm to the Olivetti dataset where each observation is a vectorized 64×64 image (i.e. the vector is of length 4096). In our application, we specify a uniform preference of -210 which yields exactly 40 exemplars, which matches the actual number of subjects in the dataset. The total wall time to run the algorithm was 736 ms.

Figure 3 displays half of the subjects in the dataset. Each row of ten images represents a single subject, and the red boxes show which faces were chosen as exemplars. As we can see, the algorithm wasn't able to select all 40 subjects as exemplars, as several subjects had two images selected as exemplars. Among the entire dataset, no subject had more than two images selected to be exemplar images. The match accuracy, defined as the number of images whose assigned exemplar matched the actual subject it belongs to, was 65%. The negative mean square error was -44.56.

Figure 3: Olivetti Faces Database, Select Subjects, Using Affinity Propagation

4.1.2 Comparison with K-Means Clustering

We also run k-means clustering on the Olivetti faces dataset. Over 100 runs, the average mean square error from each image to its assigned cluster center was -152.90, with a standard deviation of 6.27. This indicates that affinity propagation, with a MSE of -44.56, was more accurate in finding accurate cluster centers for the images.

k-means clustering was also significantly slower than affinity propagation. The average computation time from the 100 runs of k-means was 5.65 seconds with a standard deviation of 576 ms. Affinity propagation, on the other hand, only took 735 ms, a 7.69 times improvement in speed.

4.2 Text of Frey and Dueck (2007)

Following the original paper, we apply the affinity propagation algorithm to the main text of their manuscript to identify exemplar sentences. We encode sentence similarity according to the steps outlined in pages 5-6 of the Supporting Online Material of their paper. First, sentences are stripped of punctuation and case, and words with less than 5 characters are removed. Sentence similarity is then defined as the "negative sum of the information-theoretic costs of encoding every word in sentence i using the words in sentence k and a dictionary of all words in the manuscript." The cost of encoding a word in sentence i is equal to the negative logarithm of the number of words in sentence k if the word in sentence i matches a word in sentence k (where a match is satisfied as long as either word is a substring of the other). Otherwise, the cost is set to the negative logarithm of the total number of unique words in the manuscript.

The preferences for each sentence were set to the number of words in the sentence times the negative logarithm of the number of words in the manuscript dictionary, plus some constant. In our case, we set the constant to -80 to identify four exemplars.

In Table 5, we display the four sentences our algorithm found as the exemplars, as well as the four sentences identified in the original paper. Only one of the sentences match between our application and their application. It is difficult to pinpoint the exact reason, but it is possible that the exact manuscript draft or preferences used may have been different, or perhaps some sections (such as the abstract) were absent in their analysis while they were included in ours, or vice versa.

Exemplar Sentences Using Our Affinity Propagation Algorithm	
1.	Affinity propagation takes as input a collection of real-valued similarities between data points, where the similarity $s(i, k)$ indicates how well the data point with index k is suited to be the exemplar for data point i.
2.	If *a priori*, all data points are equally suitable as exemplars, the preferences should be set to a common value - this value can be varied to produce different numbers of clusters.
3.	The availability $a(i, k)$ is set to the self-responsibility $r(k, k)$ plus the sum of the positive responsibilities candidate exemplar k receives from other points.
4.	Affinity propagation found exemplars with much lower squared error than the best of 100 runs of k-centers clustering.
Exemplar Sentences in Original Paper	
1.	Affinity propagation identifies exemplars by recursively sending real-valued messages between pairs of data points.
2.	The number of identified exemplars (number of clusters) is influenced by the values of the input preferences, but also emerges from the message-passing procedure.
3.	The availability $a(i, k)$ is set to the self-responsibility $r(k, k)$ plus the sum of the positive responsibilities candidate exemplar k receives from other points.
4.	For different numbers of clusters, the reconstruction errors achieved by affinity propagation and k-centers clustering are compared.

Table 5: Exemplar Sentences Using Affinity Propagation

4.3 Environmental Data

Our final real-data application deals with environmental data, and is a new example we provide that is not included in the original paper.

It is common in environmental data that we do not have a feature value in all the pixels in a map (latitudes and longitudes) and so the goal is to estimate unknown pixels with the known nearby ones. Another scenario is that the resolution of map is high and we are interested to reduce the resolution (due to the limitation in the computational resources) and so the goal is to minimize lossy compression. To tackle such problems, one idea is to employ the clustering algorithms (such k-means) and then for each cluster assign the value of the cluster center to all the points that belong to that cluster. In what follows, we perform this approach on Bare fraction and Elevation maps using both the k-means and the (developed) affinity propagation algorithms. We train the clustering algorithms with a small random sample of original data and find the cluster centers and then predict which cluster each data point belongs to by finding the minimum Euclidean distance between them.

4.3.1 Data

Figure 4 shows some of the environmental data we might be interested in from a map.

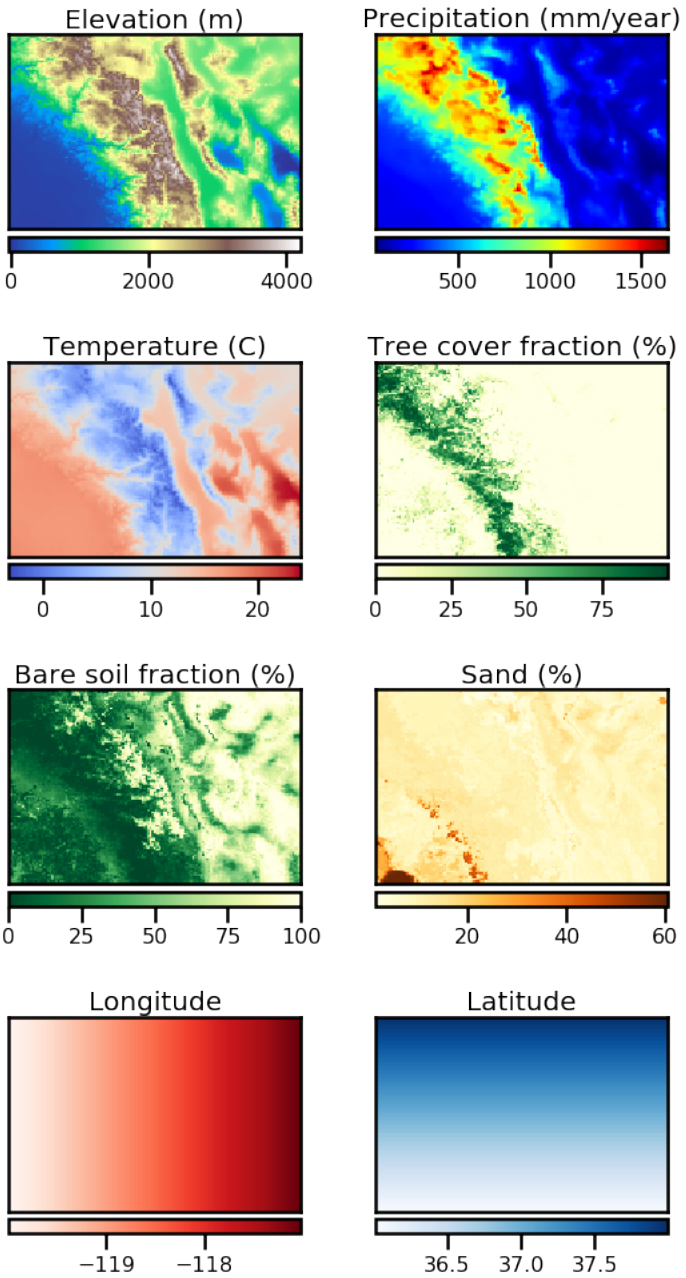

Figure 4: Examples of Environmental Data on a Map

In our application, we focus on bare cover fraction (or bare soil fraction) and elevation, of which the data is shown in Figure 5. We use only a small number of data to fit the cluster, in the hopes of using the resulting clusters to predict environmental elements on the map.

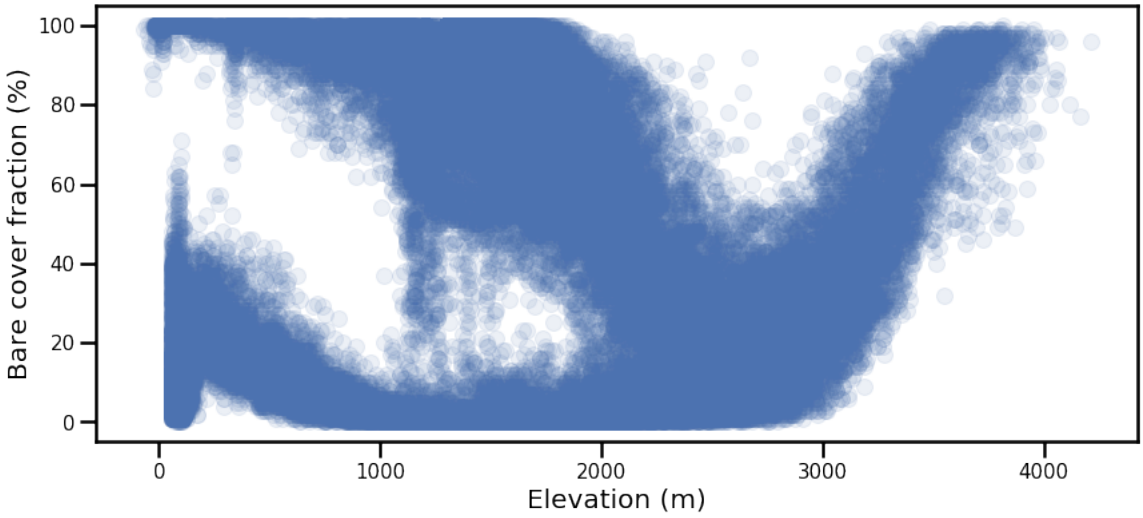

Figure 5: Bare Cover Fraction (%) versus Elevation (m)

4.3.2 Fit and Prediction using Affinity Propagation

First, we apply our developed affinity propagation algorithm to a random subset of 1500 data points from the original data. Setting the preferences to be the median similarity yields 24 clusters. The predicted clusters are shown in Figure 6.

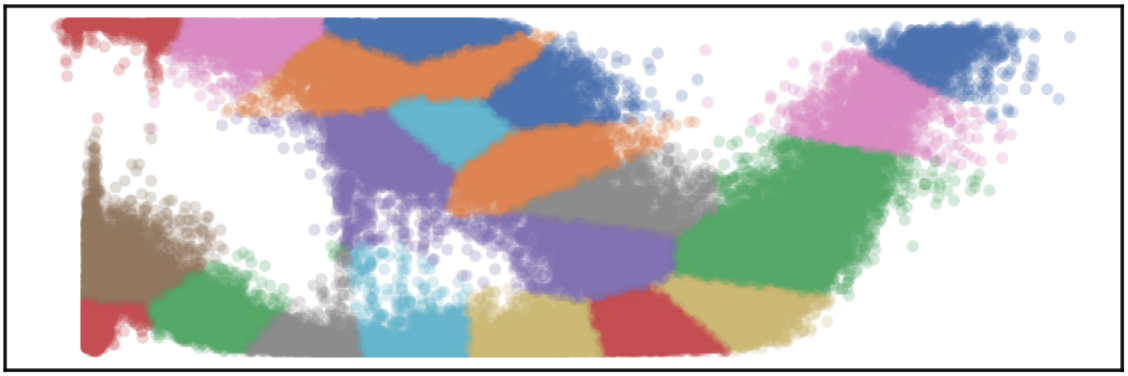

Figure 6: Predicted Clusters using Affinity Propagation

We can then use these clusters to reproduce the original image, as shown in Figure 7. In particular, for each pixel on the map, the cluster assignments determines the "color" of the pixel in both the elevation and bare cover fraction dimensions.

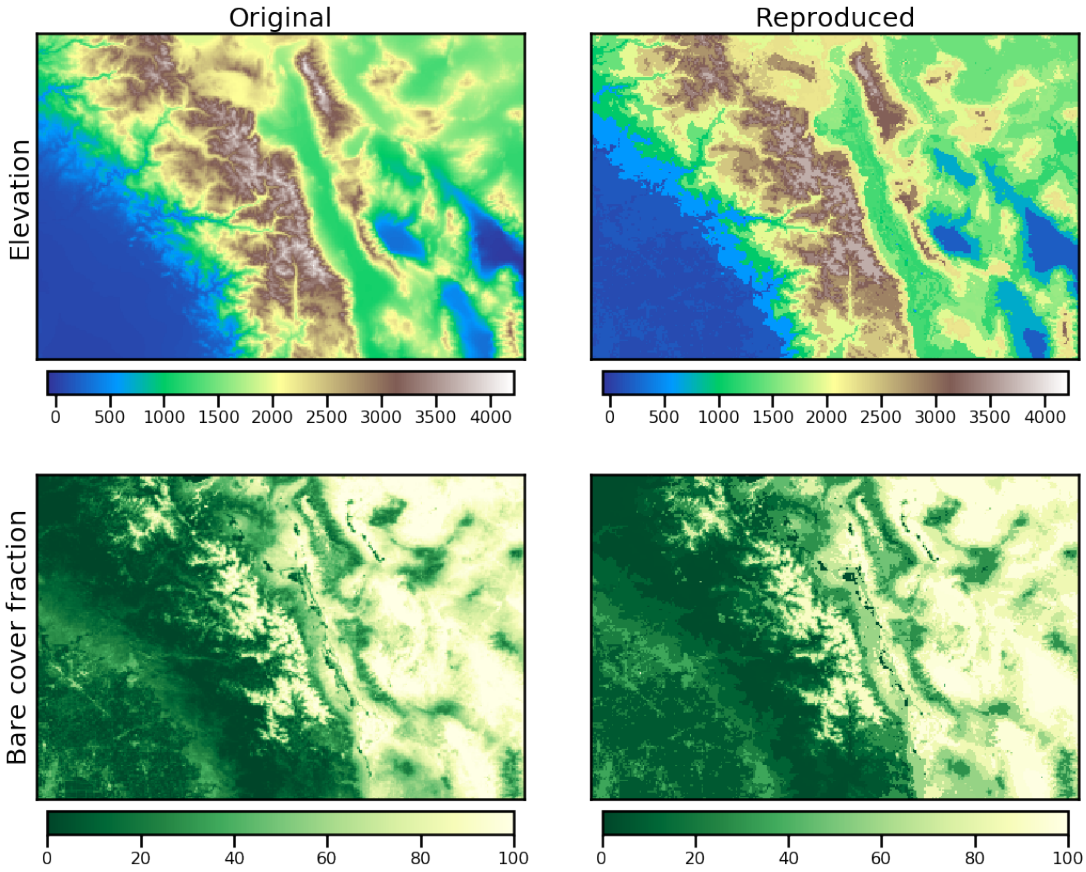

Figure 7: Original vs. Reproduced Maps, using Affinity Propagation

4.3.3 Fit and Prediction Using *k*-Means

Next, we fit a k-means model and set $k = 24$ to create 24 clusters. We then similarly predict the cluster assignments of all of the data. The resulting clusters are shown in Figure 8.

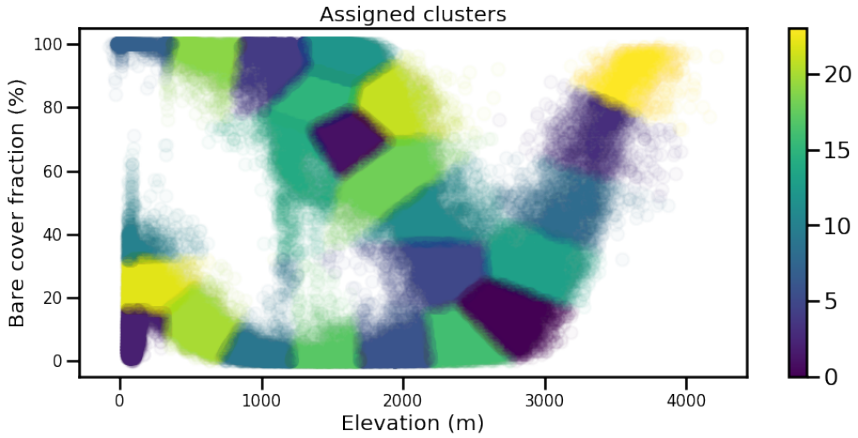

Figure 8: Predicted Clusters using k-Means

As with the affinity propagation clusters, we can then use the k-means clusters to reproduce the original image, as shown in Figure 9.

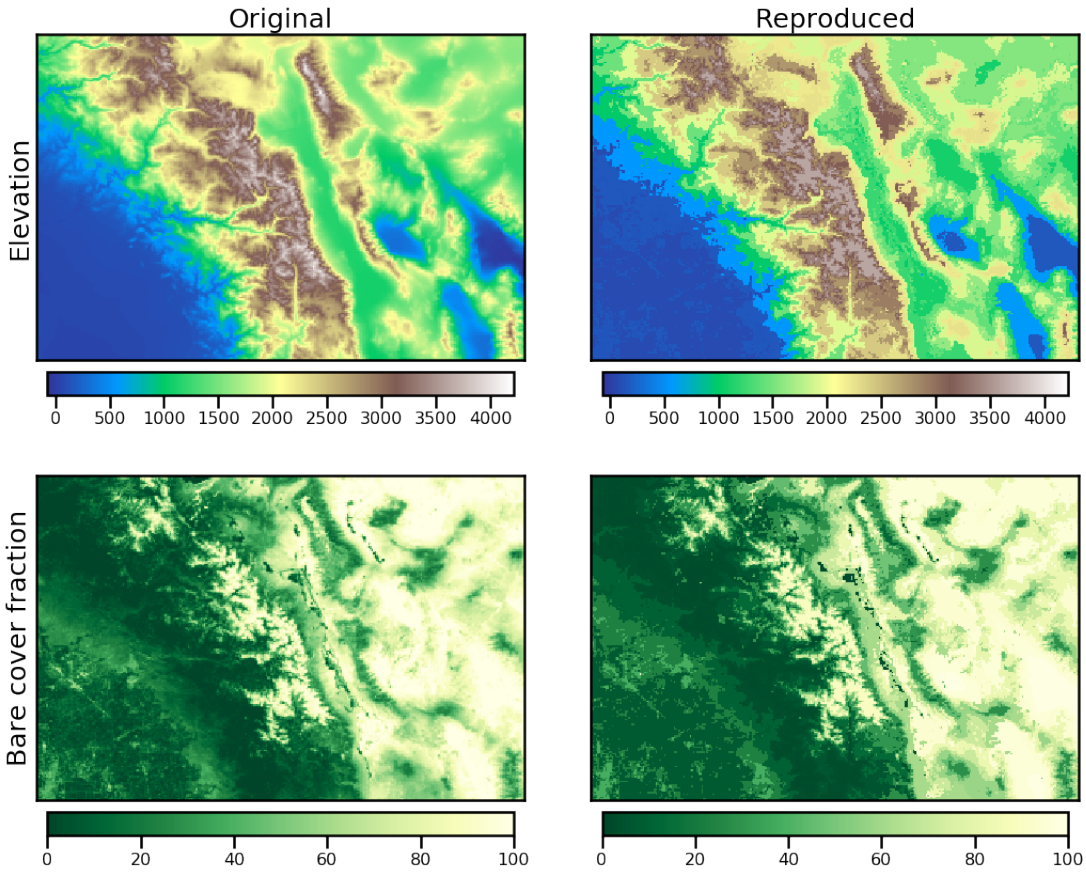

Figure 9: Original vs. Reproduced Maps, using k-Means

4.3.4 Accuracy of Predictions

In our research, we employ KGE, Kling Gupta Efficiency, as a metric to check the quality of the reduced map. This metric is defined as follows:

$$\text{KGE} = 1 - \sqrt{(\rho - 1)^2 + (\frac{\sigma_{org}}{\sigma_{sim}} - 1)^2 + (\frac{\mu_{org}}{\mu_{sim}} - 1)^2}$$

where ρ is the Pearson correlation between the original and simulated maps, σ_x is the standard deviation of a given map, and μ_x is the arithmetic mean of a given map. The KGE of the two algorithms, k-means and affinity propagation, are shown in Table 6.

Algorithm	KGE	
	Elevation Map	Bare Cover Fraction Map
Affinity Propagation	0.9979	0.9977
k-Means	0.9947	0.9950

Table 6: KGE (Kling Gupta Efficiency) of Reduced Maps

These results indicate that affinity propagation has almost the same performance as k-means when we train it with the same number of clusters. The advantage of Affinity Propagation is that the cluster centers (exemplars) are selected from the input data points and we do not need to specify number of clusters *a priori*.

5 Discussion

In this project, we developed our own version of the affinity propagation algorithm described in Frey and Dueck (2007) and tested the algorithm on both simulated and real datasets.

Affinity propagation has two main advantages as compared with similar centers-clustering algorithms. First, the algorithm is agnostic to the number of clusters, unlike k-means or k-centers where the number of clusters need to be specified beforehand. The preference feature in the algorithm allows the resulting clusters to be flexible to prior specifications of the model. Second, the algorithm is built on recursive iterations of simple, local computations, yielding significant computation improvements in certain cases. We see this in our first real-data application with the Olivetti faces database, in which our algorithm has almost an 8 times speed improvement.

A disadvantage of affinity propagation is that it doesn't do well with non-spherical datasets, as we showed in our second application to simulated data.

In the development of this algorithm, we built an inefficient ("plain") algorithm with primarily for-loops, as well as two more efficient algorithms mainly using broadcasting. The broadcasted algorithms yielded significant computation time improvements over the plain version, by a factor of several hundred.

Through various applications, we show the validity of our affinity propagation algorithm, and explore the differences between our algorithm and other similar algorithms such as k-means.

Future work with this project include investigating applications with other datasets. In the original paper, Frey and Dueck explore the algorithm's efficacy on a sparse DNA dataset. It would be interesting to find other sparse datasets to apply the algorithm to and see how it performs. Another area of research could be to test the exemplars that affinity propagation yields with exemplars that human subjects choose, to see how well affinity propagation actually does in reality. Finally, one of the core elements of the algorithm is the similarity matrix. One potential avenue of research would be to investigate how sensitive the algorithm is to various competing similarity metrics, such as Jaccard or Cosine similarity, in various applications.

References

Frey, B. J., & Dueck, D. (2007). Clustering by passing messages between data points. *Science*, *315*(5814), https://science.sciencemag.org/content/315/5814/972.full.pdf, 972–976. https://doi.org/10.1126/science.1136800