4-10-2018

Clustering biological data with self-adjusting high-dimensional sieve

Josselyn Gonzalez
Illinois State University, jgonz16@ilstu.edu

CLUSTERING BIOLOGICAL DATA WITH SELF-ADJUSTING HIGH-DIMENSIONAL SIEVE

JOSSELYN GONZALEZ

67 Pages

Data classification as a preprocessing technique is a crucial step in the analysis and understanding of numerical data. Cluster analysis, in particular, provides insight into the inherent patterns found in data which makes the interpretation of any follow-up analyses more meaningful. A clustering algorithm groups together data points according to a predefined similarity criterion. This allows the data set to be broken up into segments which, in turn, gives way for a more targeted statistical analysis. Cluster analysis has applications in numerous fields of study and, as a result, countless algorithms have been developed. However, the quantity of options makes it difficult to find an appropriate algorithm to use. Additionally, the more commonly used algorithms, while precise, require a familiarity with the data structure that may be resource-consuming to attain. Here, we address this concern by developing a novel clustering algorithm, the sieve method, for the preliminary cluster analysis of high-dimensional data. We evaluate its performance by comparing it to three well-known clustering algorithms for numerical data: the k-means, single-linkage hierarchical, and self-organizing maps. To compare the algorithms, we measure accuracy by using the misclassification or error rate of each algorithm. Additionally, we compare the within- and between-cluster variation of each clustering result through multivariate analysis of variance. We use each algorithm to cluster Fisher's Iris Flower data set, which consists of 3 "true" clusters and 150 total observations, each made up of four numerical measurements. When the optimal clustering structure is known, we found that the k-means and self-organizing maps are the more efficient algorithms in terms of speed and accuracy. When this structure is not known, we found that the sieve algorithm, despite higher misclassification rates, was able to obtain the optimal clustering structure through a truly blind clustering. Thus, the sieving algorithm functions as an informative and blind preliminary clustering method that can then be followed-up by a more refined algorithm. The existence of reliably efficient clustering process for numerical data means that more time, effort, and computational resources can be spent on a more rigorous and targeted statistical analysis.

KEYWORDS: cluster analysis, k-means, self-organizing maps, hierarchical, sieve

CLUSTERING BIOLOGICAL DATA WITH SELF-ADJUSTING HIGH-DIMENSIONAL SIEVE

JOSSELYN GONZALEZ

A Thesis Submitted in Partial
Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Mathematics

ILLINOIS STATE UNIVERSITY

2018

CLUSTERING BIOLOGICAL DATA WITH SELF-ADJUSTING HIGH-DIMENSIONAL SIEVE

JOSSELYN GONZALEZ

ACKNOWLEDGMENTS

I would like to express my sincerest gratitude to the members of my thesis committee, Drs. Olcay Akman, Epaminondas Rosa, and Ranee Thiagarajah, for their support, motivation, and invaluable advise. In particular, I want to thank Dr. Olcay Akman for his guidance and endless patience throughout my two years at Illinois State University. His passion for mathematical and statistical biology has been a great source of motivation for me. I am also genuinely grateful to Drs. Epaminondas Rosa and Rosangela Follmann for their constant encouragement and assistance.

Also, thank you to my wonderful friends for always being willing to humor my stress-addled ramblings and to my family for their relentless and unconditional support. Finally: les quiero agradecer especialmente a mis papás por todo el amor y apoyo que siempre me han dado. Les quiero dar las gracias por tener confianza en mí y mis metas. Sería imposible realizar mis sueños sin ellos y no tuviera el valor de confrontar mis debilidades. Los quiero mucho.

J. G.

CONTENTS

TABLES

FIGURES

CHAPTER I: CLUSTERING: AN INTRODUCTION

Charu C. Aggarwal, a researcher for IBM and author of a number of data mining books, describes the problem of data clustering with the following statement: “Given a set of data points, partition them into a set of groups which are as similar as possible.” [1] That is, the purpose of clustering is to separate a data set according to its natural data structures. The resulting separations or partitions are called clusters.

The definition of a cluster can be stated in multiple ways. Here, we provide a general definition: a cluster is a collection of objects which are similar to each other. Objects belonging to different clusters are not as similar. A more rigorous definition of a cluster requires a proximity measure and a similarity criterion. Examples of common proximity measures used for numerical data include the Euclidean distance, the Manhattan distance, and the discrete metric [6]. The similarity criterion then determines how similar two objects must be to be clustered together. The results of a clustering algorithm depend almost entirely on the chosen proximity measure and similarity criterion.

Clustering algorithms are considered unsupervised classification, which are a type of algorithm that classifies data objects into groups based completely on the natural features or patterns present in the data. In contrast, supervised classification, which includes regression analysis and analysis of variance, classifies data objects based on external information, such as labels or characteristics predefined by the user. That is, while supervised classification requires *a priori* knowledge about the structure of the data, unsupervised classification does not and is actually implemented to find structure in the data [9].

In a 1936, biologist and statistician Ronald Fisher published what later became a benchmark data set of Iris flower measurements for the purposes of discussing methods to discriminate between groups present in numerical data [5]. Figure 1 depicts the scatter plots for each of the four measurements. While Fisher’s work showed that a discriminant analysis of data is invaluable to the development of strong predictive models or classification rules, a cluster analysis is a necessary tool to confirm the existence of patterns or discriminating features in the data. As a result, cluster analysis usually serves as a preliminary step for other statistical algorithms and helps researchers gain meaningful insight into the distribution of the data with which they are working. It has countless applications in fields of study that require the analysis of large data sets such as morphology, ecology, medical sciences, and many others.

Cluster analysis algorithms provide quick, reliable, and consistent information about the data at hand [9]. Their usefulness has further led to the development of many types of clustering algorithms. For numerical data, the most well-known algorithms are k-means clustering, hierarchical (single-linkage) clustering, and self-organizing maps. Although they are commonly used, these clustering algorithms typically require a familiarity with the structure of the data in order to obtain optimal clustering results. In this work, we propose the sieve method, a novel clustering algorithm we have developed as an informative preliminary tool for data analysis or to implement as a starting point for other clustering processes. Additionally, we compare the performances of the four algorithms by applying them to the Iris data set. We use resulting misclassification (error) rates, test statistics, and computing times to determine the efficiency and accuracy of the algorithms.

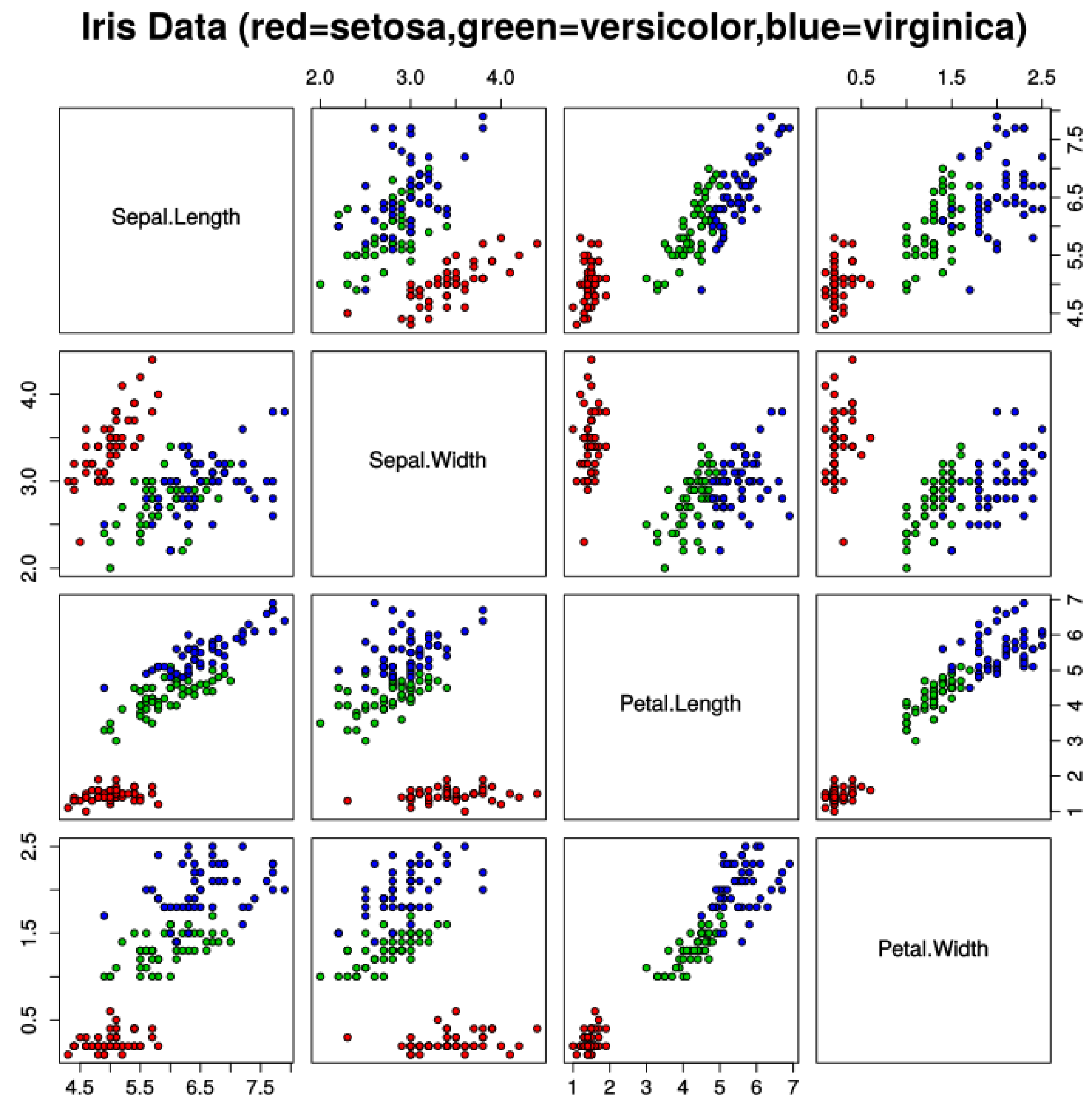


Figure 1: A scatterplot matrix of the Iris Data Set. Pairwise components are sufficient to distinguish the *I. setosa* from the *I. versicolor* and *I. virginica*, but all four components are necessary to discriminate between the three species groups. Image credit: Nicoguaro, CC-BY-4.0.

CHAPTER II: A BEGINNER'S GUIDE TO CLUSTERING

The process of data clustering varies according to the particular research problem being addressed. Each field of study may also have its own conventions. Jain [14] and Xu [23] each provide a basic outline of the procedure for performing a typical cluster analysis. In most cases, the best clustering of the data comes from iteratively performing these steps, a summary of which are given below:

1. Data Representation,
2. Clustering Algorithm Selection,
3. Cluster Validation, and
4. Result Interpretation.

Data Representation

While it's possible to perform a blind clustering of the data, the process is more efficient and the results more valuable when the user initially identifies important patterns and features of the data that are relevant to the research question. This step is referred to as feature selection and it is especially crucial when clustering high-dimensional data. Feature selection may involve graphically representing the data in the form of a scatterplot, scatterplot matrix, or histogram. For example, the scatterplot matrix of the Iris data set in Figure 1 makes clear the differences that exist between the three Iris species. While sepal length and width are enough to separate the *setosa* species from the other two, the petal measurements are necessary to discriminate the three species. Feature selection may also involve the use of proximity matrices, matrices whose ij-th entry represents the distance or similarity between the i-th and j-th data points. Viewing the data in this way helps make clearer which features or dimensions provide important insight. Other feature selection and extraction methods are Principal Component Analysis and Singular Value Decomposition, both of which are used in dimensionality reduction [6].

Another factor that must be taken into consideration when selecting the features on which to focus is the type of data making up the data set. Whether the data is numerical, categorical, binary, string, or something else entirely determines what kind of algorithms will be used. Many types of data exist and may be specific to a particular research area or field of study. Hence, it is sometimes necessary to standardize the data, i.e. transform the data so that it is dimensionless and easier to use and interpret. Such transformations, however, may result in a loss of original information, so it is important that the chosen transformation technique preserve as much of the information contained in the data as possible. Again, each area of study tends to have its own conventions for the standardization of data to address this very concern. See [6] for a thorough discussion of standardization and transformation techniques.

A simple transformation technique that can be used to transform high-dimensional vectors into 2-dimensional vectors is projection using an orthonormal basis. Suppose we have a set of p-dimensional vectors $\{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$. We begin the projection by choosing two vectors $\mathbf{w}_1$ and $\mathbf{w}_2$ that are linearly independent, i.e. there does not exist a scalar c such that $\mathbf{w}_2 = c\mathbf{w}_1$. These two vectors span the 2-D plane onto which we wish to orthogonally project the data set. We first orthonormalize the basis by computing the following vectors:

$$\begin{aligned}
\hat{\mathbf{w}}_1 &= \frac{\mathbf{w}_1}{\|\mathbf{w}_1\|} \\
\mathbf{w}_2^{\intercal} &= \mathbf{w}_2 - \langle \mathbf{w}_2, \hat{\mathbf{w}}_1 \rangle \hat{\mathbf{w}}_1 \\
\hat{\mathbf{w}}_2 &= \frac{\mathbf{w}_2^{\intercal}}{\|\mathbf{w}_2^{\intercal}\|},
\end{aligned}$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product and $\| \cdot \|$ is the vector norm. The new basis satisfies the normality condition $\|\hat{\mathbf{w}}_1\| = \|\hat{\mathbf{w}}_2\| = 1$ and the orthogonality condition $\langle \hat{\mathbf{w}}_1, \hat{\mathbf{w}}_2 \rangle = 0$. Each vector $\mathbf{v}_i \in \{\mathbf{v}_1, \cdots, \mathbf{v}_n\}$ is now represented in 2-dimensions by the point $(\langle \mathbf{v}_i, \hat{\mathbf{w}}_1 \rangle, \langle \mathbf{v}_i, \hat{\mathbf{w}}_2 \rangle)$.

Clustering Algorithm Selection

Once the important characteristics of the data are determined, the natural next step is to design or select a clustering algorithm that works best with the data. Designing a clustering algorithm mainly involves determining a distance or similarity measure and a similarity criterion. The resulting clusters are entirely dependent upon these choices.

There is no universal clustering algorithm. Once again, many fields of study have their own conventions when it comes to clustering. However, certain parameters must be met in order for the distance or similarity measure to provide any meaningful information. It is important to note the inverse relationship between distance and similarity despite the interchangeable use of the terms. To state the relationship more explicitly, the smaller the distance, the more similar two objects are and vice versa.

In order for d to be a distance measure applied to a data set $\{x_1, x_2, ..., x_n\}$, the following conditions must hold for indices $1 \leq i, j, k \leq n$:

(i) $d(x_i, x_j) = d(x_j, x_i)$,

(ii) $d(x_i, x_j) \geq 0$,

(iii) $d(x_i, x_k) \leq d(x_i, x_j) + d(x_j, x_k)$, and

(iv) $d(x_i, x_j) = 0$ if and only if $x_i = x_j$.

Condition (i) is symmetry, i.e. the distance between any two objects will remain the same no matter in what order the measurement is taken. Condition (ii) requires that all distances be non-negative for there would be no applicable meaning otherwise. Condition (iii) describes the triangle inequality, which essentially states that the shortest path between two objects is always the most direct path. Finally, condition (iv) states that the distance between an object and itself is always zero, making this the only time distance is non-positive.

The most well-known and commonly used distance measure for numerical data is the Euclidean distance in 2-dimensional space. With data objects $\mathbf{x}_i = (x_{i1}, x_{i2})$ and

$\mathbf{x}_j = (x_{j1}, x_{j2})$, the Euclidean distance d between any two points $\mathbf{x}_i$ and $\mathbf{x}_j$ is given by

$$d(\mathbf{x}_i, \mathbf{x}_j) = \sqrt{(x_{i1} - x_{j1})^2 + (x_{i2} - x_{j2})^2}.$$

This definition can be extended to p-dimensional space where $p \geq 1$. The generalized Euclidean distance in p-dimensions between two points $\mathbf{x}_i = (x_{i1}, x_{i2}, ..., x_{ip})$ and $\mathbf{x}_j = (x_{j1}, x_{j2}, ..., x_{jp})$ is given by

$$d(\mathbf{x}_i, \mathbf{x}_j) = \left(\sum_{k=1}^{p} (x_{ik} - x_{jk})^2 \right)^{\frac{1}{2}}.$$

For numerical data, other examples of distance functions are the Manhattan distance, maximum distance, and average distance. For more information or for more examples of proximity measures for non-numerical data, see [4], [6], [15], and [23].

Once a distance or proximity measure is selected, a similarity criterion is then stipulated. The purpose of the similarity criterion is to define how similar two objects must be to be clustered together. This criterion may involve the optimization of a function or it may be a numerical threshold.

Many clustering algorithms exist, even when considering only numerical data sets. The most well-known are the k-means, hierarchical (single-linkage), and self-organizing maps, all three of which will be discussed in detail in chapter 4.

Cluster Validation

Cluster validation is the procedure of evaluating the goodness of the results of a clustering algorithm. It is an important step that helps the user avoid the trap of finding patterns in random data and can be used in situations that call for a comparison of the efficacy of clustering algorithms. This step is arguably one of the most challenging in the clustering process. As mentioned before, the resulting clusters of any algorithm are dependent almost entirely on the similarity measure and similarity that are chosen and are, therefore, subjective. Hence, an objective validation process is required to prove that the number of clusters is optimal and that the clusters themselves are meaningful.

Clustering validation statistics can be categorized into three classes: internal, external, and relative criteria. In internal criteria, only the internal information of the clustering process is used to evaluate the accuracy and efficiency of the clustering results. That is, no external information is referenced. Internal criteria can also be used to determine the optimal number of clusters. External criteria, on the other hand, uses externally provided information about the data set, such as class or group labels. It is typically used to compare the results of a clustering to a known result. For instance, in the Iris data set, each vector of observations is labeled with the name of the Iris species to which it belongs. Since we know the "true" cluster number in advance, this approach is mainly used for selecting the appropriate clustering algorithm for a specific data set. In relative criteria, the clustering results of an algorithm are analyzed by running the algorithm with different parameter values and is generally used for determining the optimal number of clusters. See [3], [10], and [22] for more information about clustering validation statistics.

One example of internal cluster validation uses a one-way multivariate analysis of variance (MANOVA). A MANOVA as performed on a clustering structure considers each cluster as a treatment. Similar to its univariate analog, a MANOVA is used to test the null hypothesis that there is no treatment, or cluster, effect. That is, a rejection of the null hypothesis suggests that the separation of the data into clusters is valid. An efficient clustering maximizes the distance between cluster groups while minimizing the distance between points within each cluster group. In the univariate ANOVA, the test statistic that is used to quantify the contribution of each type of variation is an F_{ratio}. In the multivariate case, a similar test can be performed. Define k as the total number of clusters and n_j as the number of elements in the j-th cluster. Let x_{ji} be the i-th object in the j-th cluster, $\overline{\mathbf{x}}_j$ the arithmetic mean of the j-th cluster, and $\overline{\mathbf{x}}$ the grand mean of the entire data set. Then Wilk's lambda statistic is defined as

$$\Lambda^* = \frac{|\mathbf{W}|}{|\mathbf{B} + \mathbf{W}|},$$

where

$$\mathbf{W} = \sum_{j=1}^{k} \sum_{i=1}^{n_j} (\mathbf{x}_{ji} - \overline{\mathbf{x}})(\mathbf{x}_{ji} - \overline{\mathbf{x}})^\intercal$$

is the matrix of the within-group sum of squares and cross products and

$$\mathbf{B} = \sum_{j=1}^{k} (\overline{\mathbf{x}}_j - \overline{\mathbf{x}})(\overline{\mathbf{x}}_j - \overline{\mathbf{x}})^\intercal$$

is the corresponding between-groups matrix. As Λ^* quantifies the amount of variation that is contributed by the within-groups variation relative to the corrected total variation $\mathbf{B} + \mathbf{W}$, a small (near zero) value of Λ^* supports a rejection of the null hypothesis that there is no treatment effect.

Another common test statistic that is used in a MANOVA is Pillai's Trace statistic, which is defined as

$$\text{tr}[\mathbf{B}(\mathbf{B} + \mathbf{W})^{-1}].$$

Pillai's Trace considers the variance contribution of $\mathbf{B}$ relative to the total variation and ranges from 0 to 1. As a result, large values of Pillai's trace are necessary for the rejection of the null hypothesis. Other examples of test statistics used in a MANOVA test are the Hotelling-Lawley Trace and Roy's Maximum Root [21].

For any MANOVA test, several assumptions must be satisfied: the j-th group has common mean vector μ_j for $j = 1, 2, \cdots, k$, the entire data set has a common variance-covariance matrix, each group is multivariate normal, and each group is independently sampled. In a general MANOVA test, deviance from any of these assumptions may require a transformation of the data or can inform the use of a particular test statistic. For example, Pillai's trace tends to be more forgiving of deviance from normality while Roy's Maximum Root is a powerful test to use when large differences exist between a group and all the others with respect to a single characteristic [15]. It is important to emphasize that we use the MANOVA test statistic only to detect the differences between and similarities within clusters. We do not test for particular treatments, but merely use a ratio of sum of squares such as the F_{ratio} or Pillai's Trace as a well-established tool to detect cluster structure presence in the data.

For an in-depth discussion about one-way MANOVAs, see [15] and [21]. For a discussion about other validity indices that are commonly used in cluster analysis, see [6] and [9].

Interpretation of Results

Recall that the purpose of clustering a data set is to separate the data objects in a way that is reflective of the natural structure of the data set. By doing so, one can gain an understanding of the data that otherwise wouldn't have been clear. It is important to note, however, that data clustering doesn't automatically provide solutions to whatever research problem one is trying to solve. In fact, it is important to resist over-interpretation of the clustering solutions. In many cases, the first attempt at clustering a data set results in a clustering that may not be the most effective, so multiple clusterings must be done.

Visualization of the clustered data plays a large role in the interpretation of the clustering. If the data is low-dimensional, it is easy to create scatterplots or dendograms to compare the clustering results. Several visualization techniques have been developed for higher-dimensional data, but interpretation may not be as straightforward. One example of a 2-dimensional representation of a high-dimensional data set involves the use of parallel coordinates [13]. In a parallel coordinate system, each dimension or feature of the data objects is represented by a vertical axis that is parallel to the other dimension axes. A single data object, then, is represented by a line intersecting each axes at its respective dimension value. Figure 2 shows a clustered Iris data set plotted in parallel coordinates. As useful as parallel coordinates have proven to be, for exceptionally large data sets, they may lead to hard-to-read data. Further, the scaling of each axis may have an effect on the perceived distance between data points. Another 2-dimensional coordinate system is the Star coordinate system, as developed by Kandogan [17], in which each dimension axis extends from a common origin and is initially placed at equal angles from each of the other dimensions. The placement of each point, then, is the result of a spiral-shaped path corresponding to each dimension component of the data. The angle and scaling of each dimension, which can be adjusted to reflect the correlation between different dimensions, again, may have a major effect on interpretation.

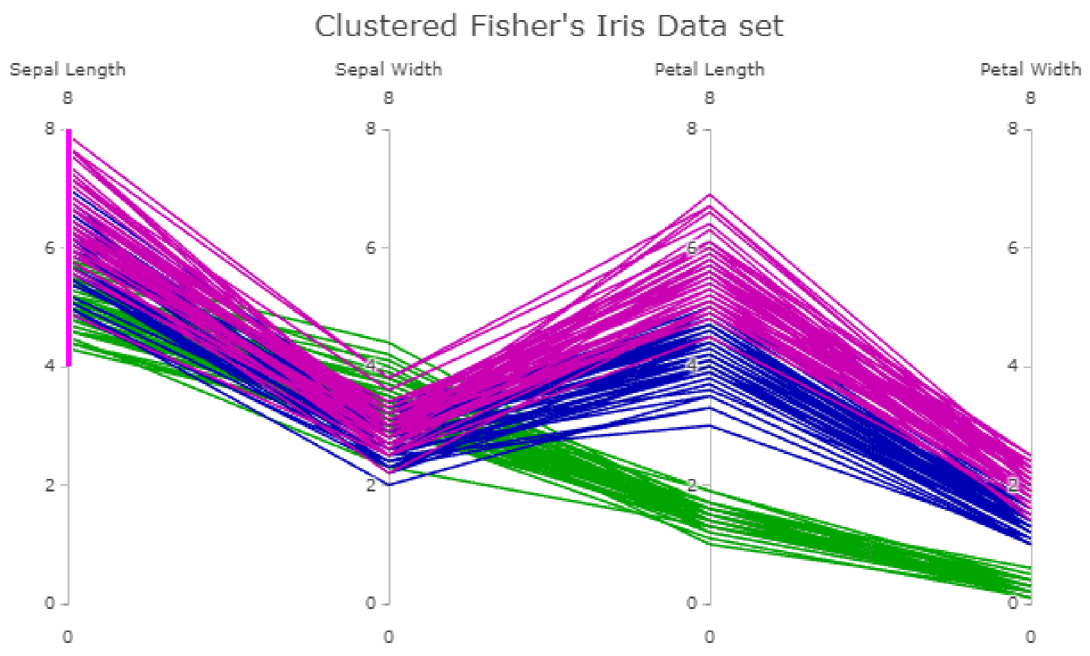


Figure 2: The Iris data set in parallel coordinates. The species *setosa*, *versicolor*, and *virginica* are shown in green, blue, and purple, respectively. Each line runs through the four component axes and represents an individual data object.

CHAPTER III: TYPES OF CLUSTERING

The two main types of clustering are known as hard and fuzzy clustering. Hard clustering requires that each data object in a data set to be clustered into one and only one cluster. This type of clustering includes partitional and hierarchical clustering, which further breaks down into divisive and agglomerative clustering. In fuzzy clustering, each data object may belong to one or more cluster and its presence in a cluster corresponds to some probability or membership value.

Hard Clustering

A clustered data set in both hard and fuzzy clustering may be represented by a $k \times n$ matrix U. Borrowing the notation from Gan [6], the matrix looks as follows:

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ u_{21} & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ u_{k1} & u_{k2} & \cdots & u_{kn} \end{bmatrix}, \tag{III.1}$$

where k is the resulting number of clusters and n is the number of data points in the original data set. Each entry in U is denoted by u_{ji} where $j \in \{1, \ldots, k\}$ and $i \in \{1, \ldots, n\}$. In hard clustering, each entry u_{ji} may only take a value of either 0 or a 1. If the data point i is in the cluster j, then $u_{ji} = 1$. Otherwise, $u_{ji} = 0$.

There are two conditions that U must satisfy in hard clustering:

(1) $\sum_{j=1}^{k} u_{ji} = 1$, and

(2) $\sum_{i=1}^{n} u_{ji} > 0$.

Condition (1) states that a data object i maybe only belong to one cluster. Simply put, only one entry may take the value of 1 within any particular column. To meet condition (2), there must be no empty clusters, i.e. each column must contain an entry of value 1.

As mentioned, partitional and hierarchical clustering algorithms fall under hard clustering. The main difference between these two types of clustering is the resulting structure of the clustered data set. Whereas partitional clustering results in a structure consisting of discrete partitions, hierarchical clustering results in a tree-like, nested structure.

In partitional clustering, the goal is to find the optimal partitioning of a data set according to some criterion function. Partitional clustering algorithms tend to be very efficient (relative to other clustering algorithms) when applied to big data sets [6]. One of the most common algorithms that is used in partitional clustering is the k-means algorithm, which will be discussed in detail in the next chapter. Another partitional clustering approach, self-organizing maps, will also be discussed later.

There are two approaches to hierarchical clustering: agglomerative and divisive. In agglomerative hierarchical clustering, the process begins with every data object placed in its own cluster. At each time step, the most similar cluster pairs are combined according to the chosen similarity measure. Divisive clustering adopts the opposite approach: the clustering begins with one large cluster containing all points that is iteratively broken up into smaller clusters according to some optimization criterion.

The linkage of cluster pairs in agglomerative clustering can be done in a number of ways. Examples of well-known methods are single-linkage, complete-linkage, and centroid-linkage clustering. In single-linkage clustering, the distance between two clusters is defined as the distance between the closest pair of points, each belonging to either cluster. That is, at each time step, the two clusters with the closest nearest neighbors are combined. In contrast, in complete-linkage, the two clusters containing the closest furthest neighbors are combined. In centroid-linkage, the centroid of each cluster is calculated and the two clusters with the closest centroids are combined. We will revisit single-linkage hierarchical clustering in the next chapter.

No matter the approach to hierarchical clustering, the data set X is broken up into Q partitions $\{H_1, H_2, ..., H_Q\}$ such that if subsets $C_i \in H_m$, $C_j \in H_l$, and $m > l$, then either $C_i \subset C_j$ or $C_i \cap C_j = \emptyset$ for all $i \neq j, m, l = 1, ..., Q$. That is, for two subsets located at different levels of the hierarchy, one is entirely contained in the other or both are mutually exclusive. Dendrograms or binary trees provide a straight-forward visualization of the nested structure of a hierarchically clustered data set.

Fuzzy Clustering

Fuzzy clustering makes use of fuzzy sets, which were defined by Zadeh [24]. These are sets whose elements have degrees of membership within the set. Suppose X is a data set. A fuzzy set A is formed if there exists a function $f_A : X \to [0, 1]$ such that each element $a \in A$ is of the form $a = f_A(x)$, for some $x \in X$. That is, each point in X is assigned a value between 0 and 1 which describes its degree of membership or the probability of its placement in the set A. Fuzzy clustering, then, results in data objects belonging to one or more clusters with their memberships in a particular cluster corresponding to some probability.

The results of fuzzy clustering can be represented by the same matrix U defined in equation (3.1.1). The conditions for fuzzy clustering are similar to the those for hard clustering. Recall that for hard clustering, each entry of the $k \times n$ matrix U is of the form $u_{ji} \in \{0, 1\}$ where $j \in \{1, \ldots, k\}$ and $i \in \{1, \ldots, n\}$ index the clusters and data points, respectively. For fuzzy clustering, we loosen the condition on u_{ji} such that $u_{ji} \in [0, 1]$. Then the following conditions must hold [6]:

(1) $\sum_{j=1}^{k} u_{ji} = 1$, and

(2) $\sum_{i=1}^{n} u_{ji} > 0$.

Condition (i) now requires that for each data object, the sum of its degrees of membership across all clusters be equal to 1. Condition (ii), as before, requires there to be no empty clusters.

One example of a fuzzy clustering algorithm is the fuzzy k-means algorithm, sometimes referred to as the c-means algorithm in the literature. Similar to its hard clustering counterpart, the goal of a fuzzy k-means algorithm is to minimize an objective function. Suppose we have a data set $D = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n\}$ and let $q \in [0, 1]$. Here, q is known as the fuzzifier and determines the "fuzziness" of the resulting clusters. Large q values result in small membership values u_{ji} and, thus, a fuzzier clustering. The objective function is defined as

$$E_q = \sum_{i=1}^{n} \sum_{j=1}^{k} u_{ji}^q d^2(\mathbf{x}_i, V_j),$$

where $d(\cdot, \cdot)$ is an inner product metric function and the V_j are the centroids, or means, of the initial clustering of the data. This initial clustering of D is, of course, allowed to overlap as long as all points are included in at least one cluster. At any iteration, the degree of membership of the data point $\mathbf{x}_i$ in the cluster j is

$$u_{ji} = \left(\frac{d^2(\mathbf{x}_i, V_j)}{\sum_{l=1}^{k} d^2(\mathbf{x}_i, V_l)} \right)^{\frac{1}{1-q}}.$$

Each centroid V_j is recalculated at each time step in the following way:

$$V_j = \frac{\sum_{i=1}^{n} u_{ji}^q \mathbf{x}_i}{\sum_{i=1}^{n} u_{ji}^q}.$$

After each iteration, the membership matrices of consecutive times steps are compared. If

$$\max_{ji} |u_{ji}^{old} - u_{ji}^{new}| < \varepsilon$$

where $\varepsilon > 0$ is some predefined criterion for stability, then the fuzzy k-means algorithm is complete. Otherwise, the membership matrix using new centroids is calculated.

Fuzzy k-means algorithms tend to be more time-consuming and complex than their hard clustering counterparts. Nevertheless, fuzzy clustering has important applications due to its flexibility in grouping data that has its basis in uncertain parameters, such as in studies of gene expression [7]. Other non-biological applications include consumer behavior and market segmentation. See [2] for further reading on fuzzy clustering in pattern recognition.

CHAPTER IV: SIEVE CLUSTERING

With this work, we introduce a novel clustering method which takes its inspiration from the simple act of using a sieve to separate objects of different sizes. We developed the sieve method as a simplistic approach to clustering. Just as a sieve is a mesh tool that is used to separate finer objects from coarser objects, in our algorithm, a sieving surface is used to separate the "finer" data objects, or the data objects satisfying our similarity criterion, from the "coarser", or unclustered, data set. The similarity criterion of the algorithm is the size of the openings in the mesh, referred to here as the sieve size. As such, a larger sieve size results in fewer clusters. The sieve method is used with numerical data as the similarity measure that we use is the dot product.

Clustering algorithms, by definition, do not require prior knowledge about the structure of data. However, for an efficient clustering, it is recommended that the user familiarize themselves with the data first by using other preprocessing techniques. For example, a k-means clustering algorithm, while efficient, requires the user to determine the optimal number of clusters k. Clustering by self-organizing maps requires similar knowledge. In a hierarchical algorithm, the user must know the distances between clusters at each linkage in order to determine the optimal threshold. By contrast, our sieving algorithm performs a truly blind clustering. The initial clustering is naive, but the algorithm runs through multiple clusterings so as to aid the user in determining the optimal k. Thus, the sieve algorithms works best as a preprocessing technique which can then be followed-up by a more refined algorithm, such as a k-means or SOM.

The sieving algorithm, as it is currently defined, takes a 2-dimensional data set X as input. A sieving size $s \in (0, 1)$ is chosen by the user. In each iteration, a sieve surface is created by randomly choosing an angle $\theta \in \{1, 2, \cdots, 180\}$ to create the line $y = \tan(\theta)$. For a data object to pass through the sieve, its corresponding vector must be near-perpendicular to the line y. That is, let the sieve surface vector be defined as $\mathbf{y} = \langle \cos\theta, \sin\theta \rangle$. Then all vectors $\mathbf{x} = \langle x_1, x_2 \rangle \in X$ that satisfy $|\mathbf{y} \cdot \mathbf{x}| \leq s$, pass through the sieve. In particular, if $0 \leq \mathbf{y} \cdot \mathbf{x} \leq s$, then $\mathbf{x}$ is placed in the "positively" perpendicular cluster, while the points $\mathbf{x}$ satisfying $-s \leq \mathbf{y} \cdot \mathbf{x} < 0$ are placed in the "negatively" perpendicular cluster. At the end of each iteration, all points that have passed through the sieve are removed from the data set, a new θ is chosen to create another sieve surface, and points are clustered into new "positively" and "negatively" perpendicular clusters. This process is repeated until all points in X belong to a cluster. Figure 3 is an illustration of a sieve iteration in 2-dimensions.

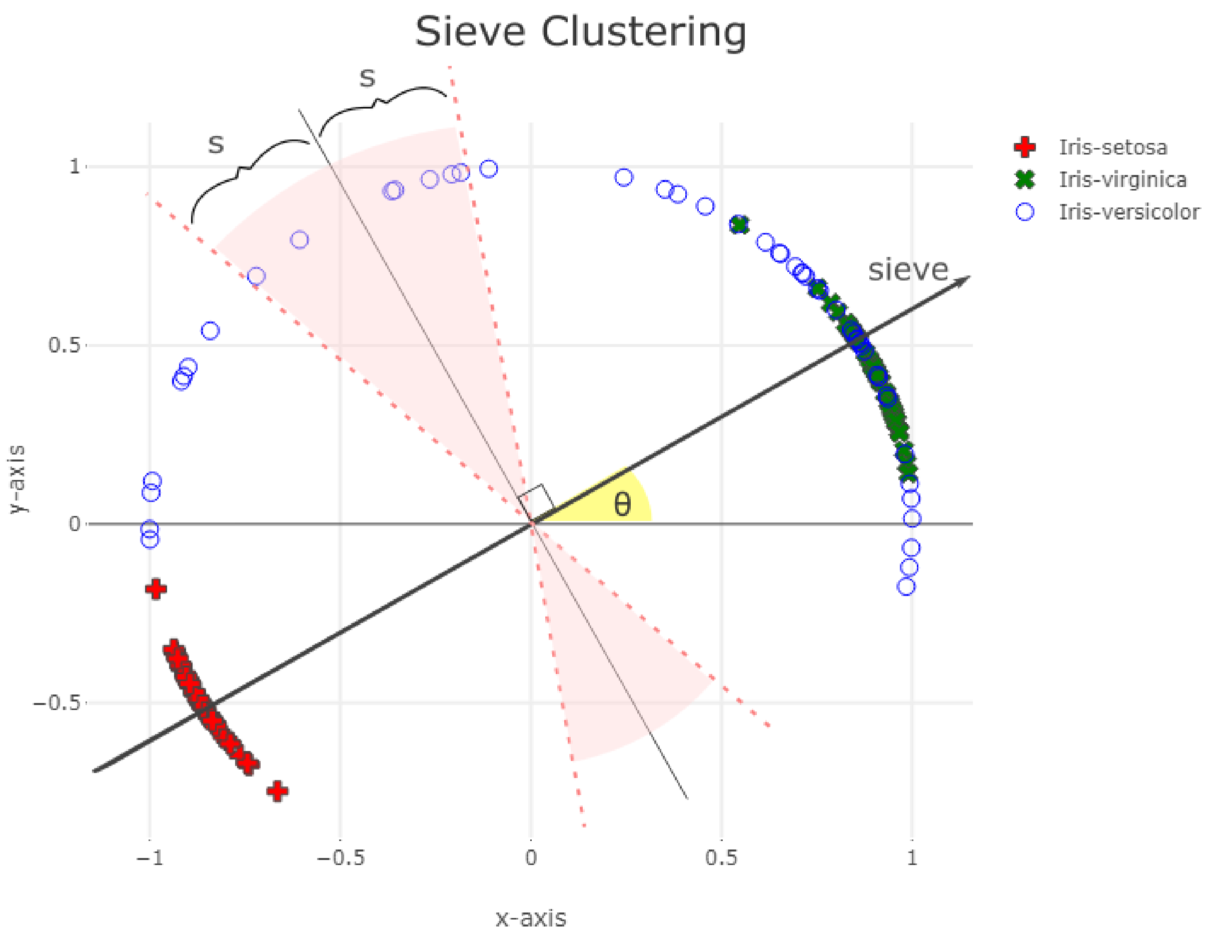
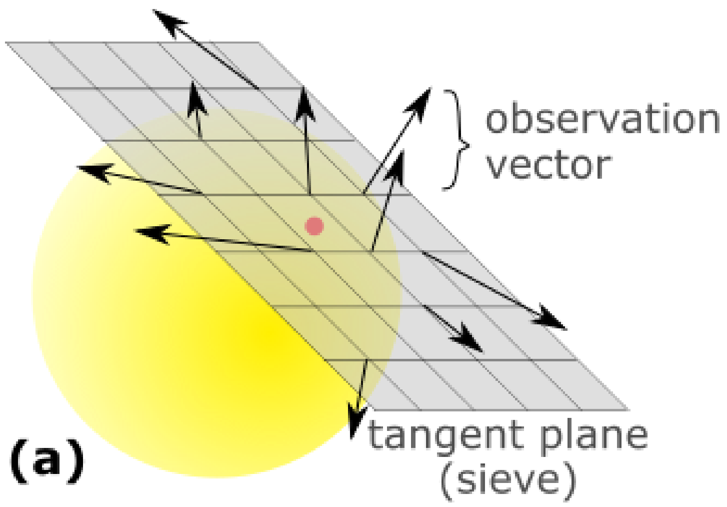


Figure 3: An iteration of the sieve algorithm in 2-D. The figure shows the Iris data set projected into 2-dimensions, shifted center around the origin, and normalized. An iteration of the sieve process begins with a randomly chosen angle θ (in yellow), which is used to create a sieving surface (black line segment). All points that fall within the pink region are perpendicular or near-perpendicular to the sieve, i.e. the corresponding dot product has a magnitude less than or equal to the sieve size s. These points are placed in a cluster according to An the sign of the dot product and are then removed from the coarse data set.

The sieve methods tends to produce a relatively large number of clusters dependent on the sieve size used, so a process similar to that of a agglomerative hierarchical algorithm is enacted. Like in an agglomerative algorithm, the number of clusters is reduced through a combination of cluster pairs according to a criterion. In sieve clustering, we make use of centroid-linkage, i.e. we define the distance between two clusters as the distance between their centroids. The clusters that share the smallest centroid distance are combined if the MANOVA test statistic of the potential clustering is more optimal than that of the current clustering. The process is over when the clustering optimizes this test statistic.

Although the sieve algorithm that we provide in Appendix A works with only 2-D data, transformation of a data set using an orthogonal basis (as explained in Chapter 2) allows the algorithm to be applicable to higher-dimensional data. An equivalent but more involved approach is to extend the sieve algorithm to handle higher-dimensional data. Figure 4 illustrates a 3-D representation of a sieving iteration. In the 3-D case, the sieve surface is a plane that is tangent to a sphere centered at the origin. This tangent plane can be created by randomly selecting a point on the surface of the sphere. Data objects that are perpendicular and near perpendicular to the sieve surface pass through the sieve and are clustered. A sieving algorithm can be developed for higher dimensions using a higher-dimensional sphere.



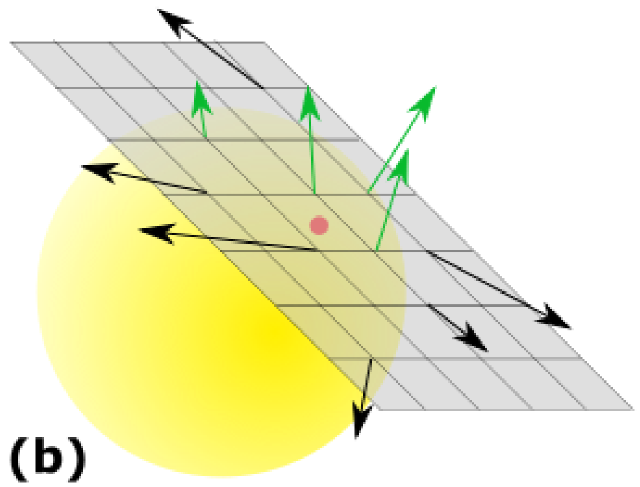


Figure 4: An iteration of the sieve algorithm in 3-D. In a 3-dimensional sieving algorithm, a sphere (yellow) can be used to create a sieve surface. **(a)** A point (red) on the surface of the sphere is randomly chosen and the corresponding tangent plane functions as the sieve. **(b)** As in the 2-D case, the data objects that are near-perpendicular to the sieve surface are clustered. At the next iteration, a new point on the sphere is randomly selected and another sieving surface is created.

CHAPTER V: COMMON CLUSTERING METHODS

We now provide an overview of some commonly used clustering algorithms which we used to compare and assess the performance of the sieve algorithm.

k-means Clustering Algorithm

The k-means algorithm is a partitional clustering algorithm, so it involves an objective function that quantifies the quality of the clustering. The optimal partitioning of the data set is the partitioning that minimizes this objective function. The k-means algorithm is only for numerical data sets and typically uses the Euclidean distance.

Suppose the data set to be clustered has n elements and we wish to cluster them into k clusters. Let C_j denote the j-th cluster for $j \in \{1, \cdots, k\}$. The standard k-means algorithm uses an error function defined as

$$E = \sum_{j=1}^{k} \sum_{\mathbf{x} \in C_j} d(\mathbf{x}, \mu(C_j))$$

where $\mu(C_j)$ is the centroid, or arithmetic mean, of the j-th cluster and $d(\cdot, \cdot)$ is the distance measure. That is, the error function is the sum of squared differences between each observation and its corresponding centroid. By iteratively taking the distances between each data point $\mathbf{x}$ and the centroid of the cluster to which it belongs, the objective of this algorithm is to find a partitioning of the data set that minimizes E.

There are two parts to a k-means clustering algorithm: initialization and iteration. In the initialization phase, the number of desired clusters k is determined. The data set is split into k groups. In the iteration phase, the distance between each data point $\mathbf{x}$ and the k centroids are calculated, and the minimum of these distances is chosen. That is, for each $\mathbf{x}$, we find the cluster J in which the minimum is achieved,

$$J = \underset{1 \leq j \leq k}{\arg\min}\, d(\mathbf{x}, \mu(C_j)),$$

and then place $\mathbf{x}$ into the J-th cluster. After all points are placed in a cluster, the k

centroids are recalculated to reflect the current state of the clusters. The value of E is determined and compared to the value at the previous step. The clustering is complete when E is minimized or, equivalently, when there is no more significant change in the centroid values or in the cluster membership of each data point from one iteration to the next.

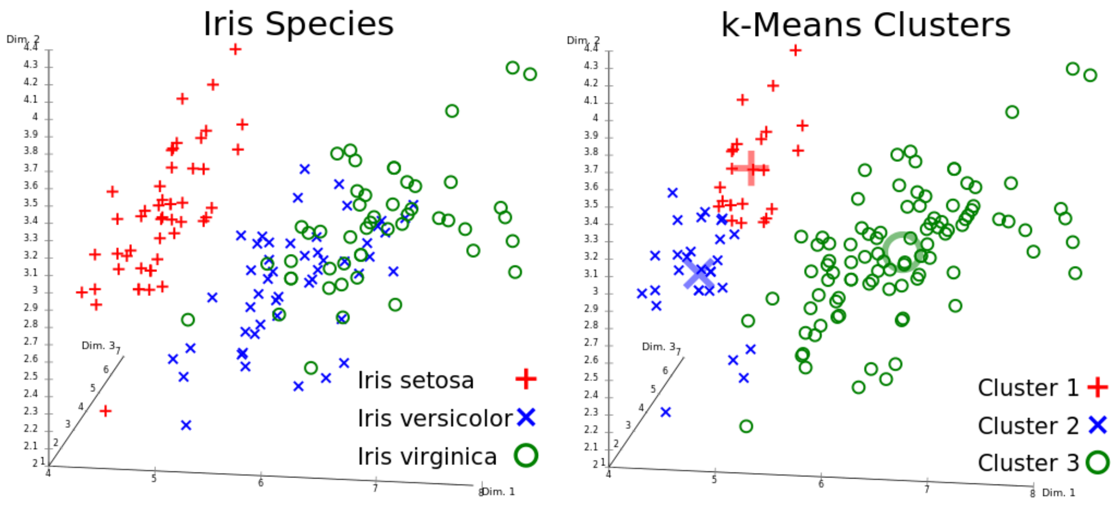


Figure 5: An example of a k-means clustering of the Iris data set. On the left, a 3-D plot of the Iris data set, each species shown in a different color and symbol. On the right, the results of a k-means clustering of the same data set. Three initial cluster centroids were chosen, represented by larger $+$, $\circ$, and $\times$ symbols. The centroids and the cluster contents were then iteratively adjusted to minimize average distance from the centroid. Image credit: Chire, Public Domain.

Ultimately, the clustering result of a k-means algorithm depends on the desired number of clusters k and on the choice of initial k groups. One way to select the optimal k value is to try multiple values. Typically, as k increases, the average distance of all the points in the data set X to their respective centroid decreases. At some value of k, however, this average distance from the centroid no longer changes significantly. This is the optimal value of k to use. To initialize the k clusters, k data points can be randomly chosen as centroids, and the rest of the points would then be placed into one of these clusters. A more computationally efficient way of initializing the clusters requires some *a* *priori* knowledge of the data set either through observation or by performing a separate clustering analysis. One could also choose the initial k centroids by choosing points that are approximately equally dispersed throughout the data set. In this way, the danger of mistakenly separating points that should be clustered together is minimized.

There are a number of variants of k-means algorithms. They are relatively simple to run and their complexity depends on the number of iterations, number of clusters k, the number of data points in X, and the dimension of these data points [11]. These algorithms work relatively well with big or high-dimensional data sets.

Hierarchical Single-Linkage Clustering Algorithm

As previously discussed, single-linkage is considered an agglomerative hierarchical method. The clustering of a data set X begins with each data object considered as a singleton cluster. In single-linkage, the distance between two clusters is defined as the minimum distance between a pair of data objects where one data object is in one cluster and the other is in the other cluster [4]. That is, the distance between clusters C_i and C_j is

$$d(C_i, C_j) = \min\{d(x, y) \text{ such that } x \in C_i, y \in C_j\}.$$

This distance is referred to as the nearest-neighbor distance. At each time step in the algorithm, the two clusters with the minimum nearest-neighbor distance are combined. These steps are repeated until there is one large cluster containing all the data objects in X. The user can then observe the resulting dendogram and decide on an appropriate threshold. It is this threshold that determines the number of clusters.

The time efficiency of the algorithm depends almost entirely on the size of the data set, so it works best with smaller data sets. As discussed in [4], the order of cluster linkage is important in agglomerative clustering, and single-linkage in particular can result in unbalanced cluster sizes.

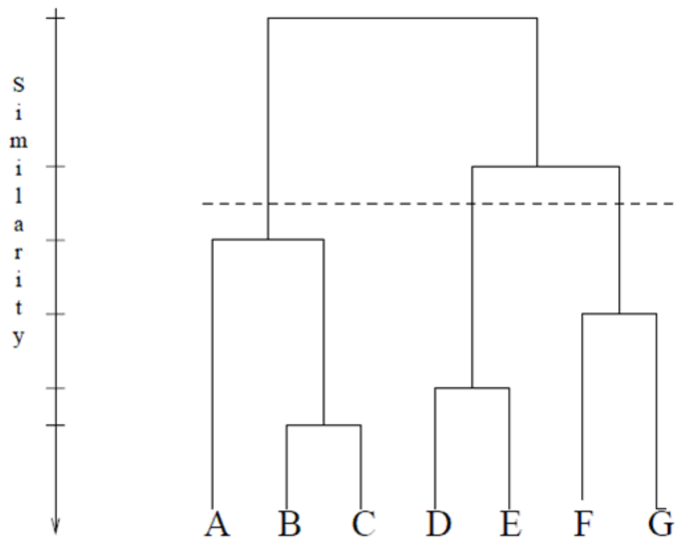


Figure 6: A dendogram representation of hierarchical clustering results. The process began with singleton clusters containing points A, B, C, D, E, F, and G. The closest pair of clusters was combined at each time step.The dashed line represents the clustering threshold. At this threshold, the final clusters are $\{A, B, C\}$, $\{D, E\}$, and $\{F, G\}$. Image credit: Henriquerocha, Public Domain.

Self-Organizing Maps

Self-organizing maps (SOMs), also known as Kohonen maps, form a class of partitional clustering algorithms that use artificial neural networks to generate a low-dimensional representation of a high-dimensional data set while simultaneously reflecting the structure of the data set in a visual way [19]. SOMs require the input of a "training" numerical data set and, through an iterative competitive learning algorithm, outputs a one- or two-dimensional map representing the data. This map eases the visualization of the underlying structures of the data set as the training allows it to preserve the similarity relationships between the original data objects and results in a clustering of the data.

In any competitive learning system, there are input nodes and output nodes. The input nodes are the input data objects and the output nodes are a set of units which are each assigned a weight vector either randomly or using *a priori* knowledge of the data set. More clearly, let the input nodes be denoted by $i \in \{1, \ldots, n\}$ and the output nodes by

$j \in \{1, \ldots, k\}$, where n is the dimension of the data set (i.e. the number of features of each data object) and k corresponds to the number of clusters. Then each output node j is weighted by a vector $\mathbf{w_j} \in \mathbb{R}^\mathbf{n}$ whose components w_{ij} are each connected to the i-th component of the input vector $\mathbf{x} \in \mathbb{R}^\mathbf{n}$. That is, for an input vector $\mathbf{x}$, each of its components x_i is connected to every output node j by some weighted connection denoted by w_{ij}. An iterative training algorithm then compares the input vector $\mathbf{x}$ to every weight vector $\mathbf{w_j}$ and seeks out the index J for which the similarity between $\mathbf{x}$ and $\mathbf{w_j}$ is maximized or, equivalently, for which the distance between the two is minimized. For node J and for some neighborhood around node J, an update, or activation, takes place where the weights of each node are updated in a way that makes them more similar to $\mathbf{x}$. For the next time step, a new input vector is presented to the nodes and the weight vectors of the nodes adjust accordingly. As the algorithm progresses, the size of the updating neighborhood around J decreases until the neighborhood contains only the node J. The algorithm is complete when the positions of the nodes satisfy some predetermined condition of stability.

In an SOM, the output nodes are called neurons and the weighted connections between the components of the input vector and the neurons are called synapses. That is, for an input vector $\mathbf{x}$, each of its components x_i is connected to every neuron j by some weighted synapse denoted by w_{ij}. For each input vector $\mathbf{x}$, the goal is to identify the "winning" weight vector, i.e., the weight vector that is most similar to the input vector. If $d(\cdot, \cdot)$ is a distance measure, then the winning vector is $\mathbf{w}_J$ whose index is defined as

$$J := \arg\min_j \{d(\mathbf{x}, \mathbf{w}_j)\}.$$

Once a winning vector is determined, an activation takes place for all weight vectors within some predetermined neighborhood of J. Let the neighborhood around J be denoted by $N(J)$. The activation function of a weight vector at iteration $t+1$ is

$$\mathbf{w_j}(t+1) = \begin{cases} \mathbf{w}_j(t) + \eta(t)(\mathbf{x}(t) - \mathbf{w}_j(t)) & \text{for} \quad j \in N(J) \\ \mathbf{w}_j(t) & \text{for} \quad j \notin N(J), \end{cases}$$

where $\eta(t)$ is called the learning rate. It is a monotonically decreasing function selected to ensure that a more fine-tuned learning is taking place as the algorithm proceeds. Commonly used learning functions are linear, inversely related to the total number of iterations performed by the algorithm, exponential, or a power series. Figure 7 shows an example SOM iteration and Figure 8 is a closer look at the training process.

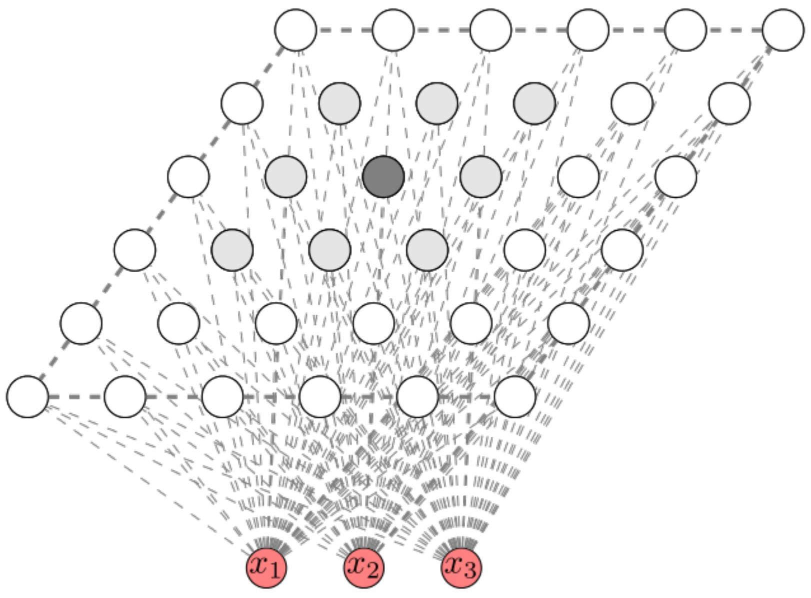


Figure 7: An example of an SOM iteration. On this6 $\times$ 6-neuron grid, the dashed lines represent the weighted connections, or synapses, between neurons and data objects $\mathbf{x}_1, \mathbf{x}_2$, and $\mathbf{x}_3$. After comparing a data object to the neurons, the algorithm determines that the dark gray neuron is the “winner”. The winning neuron undergoes an activation that makes it more similar to the presented data object. The light gray neurons fall within some neighborhood of the winning neuron, so they undergo an activation as well, albeit at a smaller scale. Image credit: MartinThoma, CC0-1.0.

After the update of the appropriate weights is complete, the next iteration of the algorithm begins with a new input vector $\mathbf{x}$ presented to the updated neurons.With each iteration of the algorithm, the size of the neighborhood around the winning neuron decreases until, finally, the only neuron affected by the update is the winning neuron itself. Additionally, the learning rate $\eta(t)$ also decreases for each time step, which means the effect the update has on the winning neuron and its neighbors lessens over time. The algorithm is considered complete when the change in the position of the neurons occurring between time steps is below some predetermined positive number.

There are several decisions that must be made before performing an SOM. To begin with, the number of desired clusters must be determined. This can be done, for example, by using a separate clustering algorithm to preprocess the data. The weight vectors of each neuron must then be initialized either by randomly assigning each entry of the vector a value between 0 and 1 or by assigning values using prior knowledge of the structure of the data set. The latter, if done correctly, can increase the efficiency of the SOM. For example, if it is obvious that some data objects are significantly different from each other, it may be more beneficial to initialize the weight vectors in a way that reflects this structure.

Next we determine the best type of neighborhood or neighborhood function to use. Common neighborhoods are circular, square, or hexagonal in shape. For these types of neighborhoods, every neuron within them is updated in the same exact way. The use of neighborhood functions allows for a more customizable update. If we define a neighborhood function by

$$h_{Jj} = \begin{cases} \eta(t) & \text{for} \quad j \in N(J) \\ 0 & \text{for} \quad j \notin N(J) \end{cases},$$

then it specifies that each neuron within $N(J)$ is affected in the same way by the learning function $\eta(t)$. The updating function can then be written as

$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + h_{Jj}(\mathbf{x}(t) - \mathbf{w}_j(t)).$$

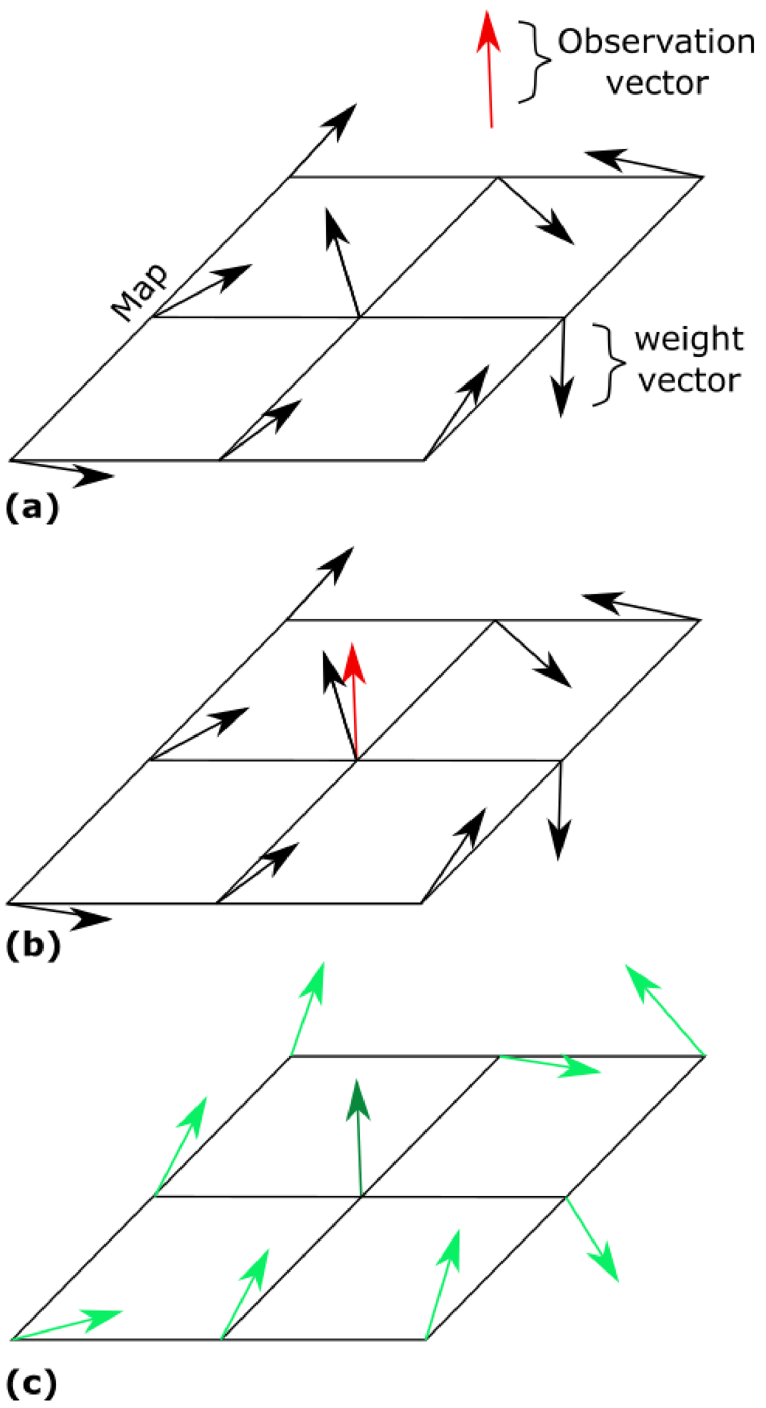


Figure 8: Another example of an SOM algorithm iteration. **(a)** A 3 × 3-neuron grid is presented with an input vector (in red). **(b)** The input vector is compared to each weight vector in the map, and the winning weight vector is selected. **(c)** The winning vector is updated (dark green) so that it is more similar to the input vector. All sufficiently nearby vectors are updated as well (light green).

If, within the neighborhood, we want the neurons closest to the winning neuron J to be affected more than those near the border of $N(J)$, we can define the neighborhood function as the product of the learning rate and another function bounded by a range of $[0, 1]$ which takes its maximum value only when $j = J$. A commonly used neighborhood function of this type is the Gaussian neighborhood function

$$h_{Jj} = \eta(t) \exp\left(\frac{-d(r_J, r_j)^2}{2\sigma^2(t)}\right)$$

where r_J and r_j are the positions of the the J-th and j-th neuron, respectively, and $\sigma(t)$ is some decreasing kernel width function which provides the radius of the neighborhood at time step t. For a more detailed discussion of kernel width functions, see [20].

Finally, the last item to be set is the condition for stability of the SOM. The condition requires that the change in position of the neurons from one time step to the next remain less than some small, positive number ε. The value of ε determines the stopping point of the algorithm.

Once an SOM is complete, the resulting map allows for an easy-to-understand visual of the structure of the data set. Furthermore, the map also tells us that each input vector $\mathbf{x}$ belongs to the Jth cluster.

Similar to k-means algorithms, an SOM algorithm requires knowledge of the number of clusters in advanced. This means that some preprocessing technique or multiple SOMs must be performed to find the optimal number of clusters. According to [23], SOMs are also inefficient at handling outliers.

CHAPTER VI: COMPARISON OF PERFORMANCE OF CLUSTERING ALGORITHMS

Simulations and Results

To compare the performances of the k-means, single-linkage hierarchical, SOM, and sieve algorithms, we used a combination of internal, external, and relative criteria. In particular, we used each algorithm to cluster the 4-dimensional Iris data set [5] for several values of k and then compared misclassification rates, MANOVA test statistics and corresponding p-values, and computational times.

All algorithms were run using Python version 3.4. The images in this section were also created using Python. The k-means and hierarchical algorithms are packages available in SciPy, a Python-based collection of mathematical and statistical algorithms and tools. We ran the k-means and hierarchical algorithms using [8] and [16] as guides, respectively. We wrote the SOM algorithm so as to focus more on the clustering than on the visualization. Finally, we developed and wrote the sieve algorithm. All algorithms are included in their entirety in Appendix A. In all algorithms, we used the same function to return the cluster misclassifcation rate (code also included in Appendix A). R was used to run the MANOVAs on the multivariate clustering results of the k-means, hierarchical, and SOM algorithms. The test statistic that R returns is Pillai's Trace. The sieve algorithm works with 2-D data and requires the value of the test statistic to end the clustering process, so we performed a MANOVA within the code using the relationship between the Wilks' Λ^* test statistic for 2-D data and the F-distribution as discussed in [15] and summarized later.

The Iris data set contains 150 observations, each made up of four measurements in centimeters: sepal length, sepal width, petal length, and petal width. Three Iris flower species are included in this data set (*setosa*, *virginica*, and *versicolor*) and serve as the data labels. There are 50 observations for each species. There are two cases for which our misclassification function returns a 0% misclassification rate with the Iris data set: the clustering results in exactly 3 clusters, each containing all 50 observations pertaining to a particular species, or the clustering results in 150 singleton clusters. A clustering consisting of only 1 cluster has an approximate misclassifcation rate of 66.67% since the algorithm assumes that only one of the three species has been placed in the cluster correctly; the algorithm considers the other 100 observations misclassified.

First, we ran the k-means algorithm. The initial k centroids are randomly chosen from the data set. By default, the algorithm runs a k-means 20 times, each time iteratively adjusting the centroids until the change in the error function E since the previous iteration falls below a threshold, defined in this algorithm as 1×10^{-5}. The algorithm returns the first clustering whose change falls below the threshold or the clustering that results in the smallest change. The error statistics of the k-means clustering for $k \in \{2, 3, 4, 5\}$ are shown in Table 1, where MR is misclassification rate.

Table 1: *Error Statistics of the k-means Algorithm.*

k	2	3	4	5
MR (%)	33.33	11.33	12.0	9.33
Pillai	0.876	0.903	0.722	0.581
p-Value	2.2×10^{-16}	2.2×10^{-16}	2.2×10^{-16}	2.2×10^{-16}
Time (s)	0.0156	0.015	0.0156	0.0156

For all values of k shown in the Table 1, the results are significant. Recall that Pillai's Trace ranges from 0 to 1, where its value corresponds to the contribution of the between-cluster variance to the total variance. Thus, a good clustering is indicated by a large Pillai's Trace test statistic since it indicates that the between-cluster variation is larger than that of the within-clusters. Although the smallest misclassification rate occured at $k = 5$, the k-means algorithm obtained a maximum Pillai's Trace value at $k = 3$ at the same level of statistical significance. It's clear also that even as k increased, the computational time needed to complete the clustering remained constant.

Next, we ran the single-linkage hierarchical clustering algorithm. Euclidean distance was used as the distance measure. The algorithm returns a $(n-1) \times 4$ linkage matrix, where n is the number of elements in the data set. Every row of this matrix represents a particular time step. The first two columns describe which two clusters were combined at a that time step, the third column is the distance between those two clusters, and the fourth provides the total number of data objects in the combined cluster. This linkage matrix can then be used to form a dendogram. The resulting number of clusters is determined by a user-defined threshold. Table 2 shows the single-linkage clustering results of the Iris data set. Figures 9 and 10 show the corresponding screeplot and dendogram, respectively.

Table 2: *Error Statistics of the Single-Linkage Hierarchical Algorithm.*

k	2	3	4	5
MR (%)	33.33	32.0	31.33	30.67
Pillai	0.907	0.892	0.890	0.882
p-Value	2.2×10^{-16}	2.2×10^{-16}	2.2×10^{-16}	2.2×10^{-16}
Time (s)	0	0.0156	0.0	0.0

The value of Pillai's Trace test statistic is large and statistically significant for all number of clusters k tested. The largest test statistic corresponds to $k = 2$. The misclassification rate at $k = 2$, however, is the largest at 33.33%. In fact, all the clustering results shown in the table show a misclassification rate of greater than 30%. In Figure 9, the screeplot was graphed using the distance between linked clusters at each time step. A possible elbow occurs at approximately $k = 2$ while another occurs at $k = 4$. This aligns with the results in Table 2. The dendogram in Figure 10 shows the order of cluster-linkage and how the final clustering results were obtained. The coloring of the linkages in the dendogram are determined by a color threshold, which has a default value of 70% of the maximum distance between linked clusters. The red and green indicate two groups that fall below this threshold. The computational times for each clustering are not entirely different since the same linkage matrix is used to obtain each clustering result.

For each value of $k \in \{2, 3, 4, 5\}$, we initialized the SOM algorithm with a

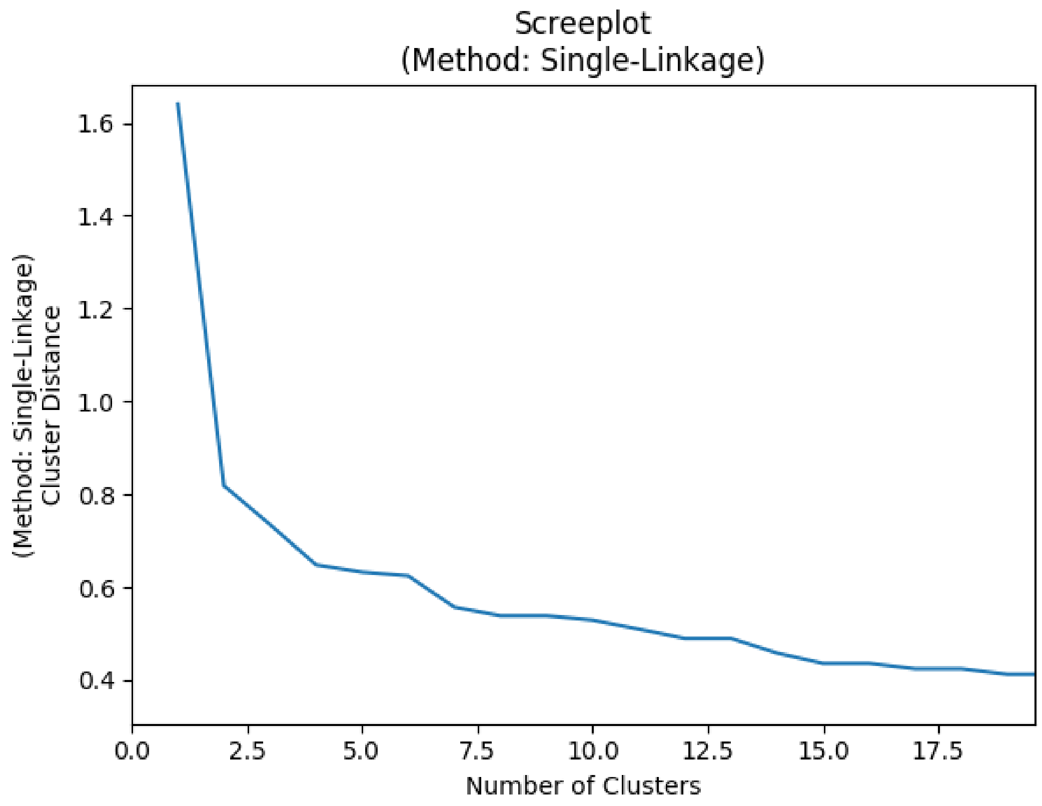


Figure 9: The screeplot of the single-linkage clustering of the Iris data set.

$4 \times k$-neuron grid. We defined the learning function $\eta(t)$ and the kernel width function $\sigma(t)$ as the same exponential decreasing function dependent on the total number of iterations. That is, we set the total number of iterations at 400, and defined

$$\eta(t) = \sigma(t) = \exp\left(\frac{-t}{400}\right).$$

A Gaussian neighborhood function was used for a soft competitive learning. Because randomness is involved in the initialization of the weight vectors and in the order of the clustering in an SOM algorithm, separate runs even under the same parameters show different results. The results of a single instance of each k are summarized in Table 3.

With the SOM algorithm, the minimum misclassification rate and the most significant clustering results were achieved at $k = 4$. The largest Pillai's trace value

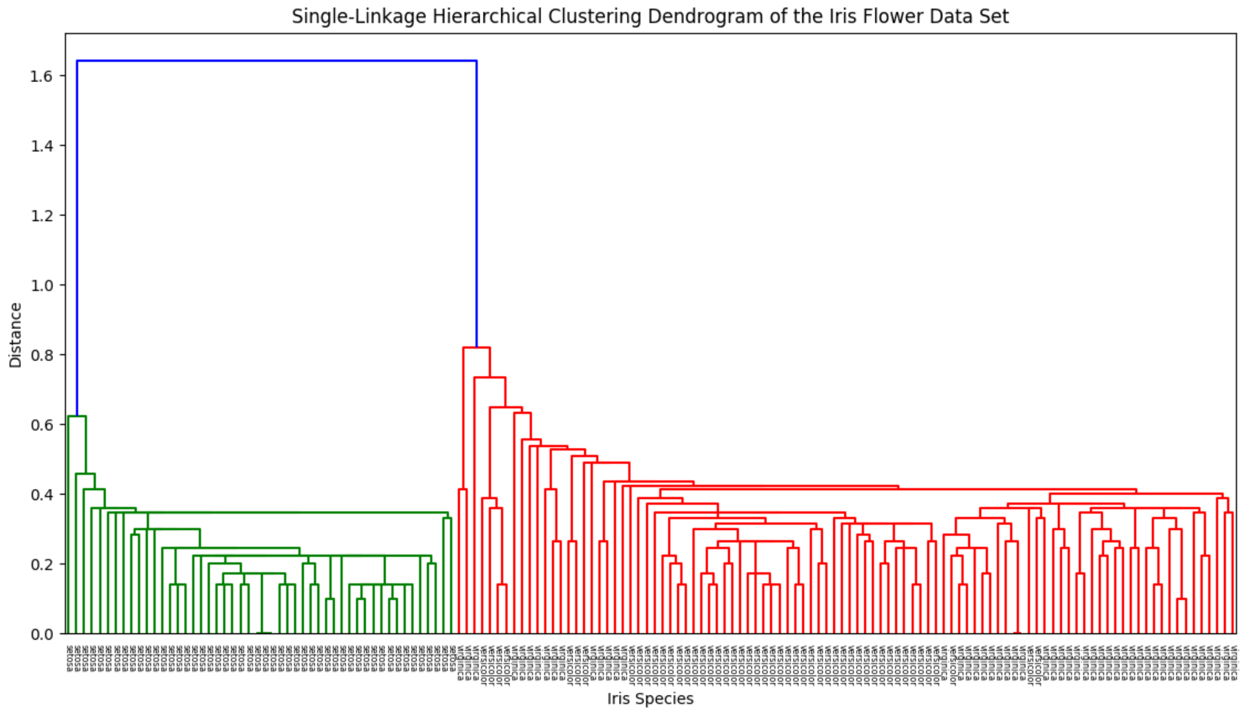


Figure 10: The Iris data set single-linkage dendogram. The coloring of the dendogram is dependent on the linkage-distance threshold, set here by default at approximately 1.15, which is 70% of the maximum distance. The red and green are the two clusters that the particular threshold returns.

Table 3: *Error Statistics of the SOM Algorithm.*

k	2	3	4	5
MR (%)	33.33	8.67	6.67	12.0
Pillai	0.825	0.312	0.359	0.252
p-Value	2.2×10^{-6}	4.03×10^{-11}	2.58×10^{-13}	1.33×10^{-8}
Time (s)	0.0625	0.0781	0.1094	0.125

occurred at $k = 2$, but showed the least significant results of the clusterings. We see also that the computational time increased steadily as k increased.

The test statistic that is used in the sieve algorithm and that is presented in Tables 4, 5, and 6 is a ratio involving Wilks' $\Lambda^* = \dfrac{|W|}{|W+B|}$. This ratio is referred to here as an F_{ratio} because it has an exact F-distribution when the data set is 2-dimensional. More explicitly, for 2-D data, our test statistic is a value

$$F_{ratio} := \left(\frac{n-k-1}{k-1}\right)\left(\frac{1-\sqrt{\Lambda^*}}{\sqrt{\Lambda^*}}\right) \sim F_{2(k-1),2(n-k-1)},$$

where k is the number of clusters at the particular time step and n is the total number of elements in the data set. A good clustering is indicated by a small (near zero) Λ^*, which corresponds to a large F_{ratio} value. The computational time included in the tables under each k is the amount of time the algorithm took to combine two clusters to obtain k clusters.

In order to take advantage of the relationship between Λ^* and the easy-to-calculate F-distribution, we first had to transform the data. After defining the data mean as the new origin, we projected the entire 4-D data set into 2 dimensions via orthogonal projection, as per the technique described in Chapter 2. We then normalized the data so that all points were at an equal distance from the origin and allowed for a better interpretation of the dot product as used in the algorithm.

The clustering of the sieve algorithm is dependent on the sieve size, i.e. a large sieve size s results in fewer initial and final clusters. For this reason, we ran the sieve algorithm for $s = \{0.1, 0.5, 0.9\}$. Tables 4-6 show the single instance results for these values of s, respectively.

As shown in Table 4, a sieve size $s = 0.1$ resulted in 42 initial clusters with a misclassification rate of 26.67%. The algorithm stopped at $k = 7$ where it achieved its largest F_{ratio}. The p-value of the F-test at $k = 7$ supports the significance of the clustering despite the misclassification rate of 28.67%. The computational time of each linkage is included in Table 4, and the total computational time was 4.08 seconds. Figure 11 is a scree plot of the clustering process that shows the relationship between the number of clusters and the corresponding F_{ratio} test statistic. As clusters were linked, the F_{ratio} value increased as well until the algorithm terminated at $k = 7$.

Table 4: *Error Statistics of the Sieve Algorithm with $s = 0.1$.*

k	7	8	9	10	$\cdots$	42
MR (%)	28.67	28.67	28.67	28.67	$\cdots$	26.67
F_{ratio}	46.448	39.662	35.762	32.272	$\cdots$	6.64
p-Value	4.45×10^{-5}	1.73×10^{-5}	6.46×10^{-6}	2.67×10^{-6}	$\cdots$	5.43×10^{-10}
Time (s)	0.0938	0.0938	0.0781	0.0937	$\cdots$	0.0781
Total Computational Time: 4.08 s						

We then ran the sieve algorithm for sieve size $s = 0.5$ (Table 5). The larger s value resulted in fewer initial clusters with a misclassification rate of 27.33%. The algorithm obtained the largest F_{ratio} value at $k = 2$. However, the clustering at $k = 2$ would not be considered statistically significant for significance levels of $\alpha < 0.05$. Since $k = 2$ is also associated with the largest misclassification rate, this may not be the best clustering of the data. The error statistics support that a better clustering occurred at $k = 3$ with a misclassification rate of 32.67% and statistically significant results for $\alpha < 0.01$. The total computational time was 1.76 seconds. Figure 12 is the scree plot for this clustering process. The first possible elbow point occurs at $k = 3$, which further supports a better clustering.

With sieve size $s = 0.9$, the algorithm resulted in 16 initial clusters and terminated at 2 clusters. The final clustering had a 33.33% misclassification rate and significant results

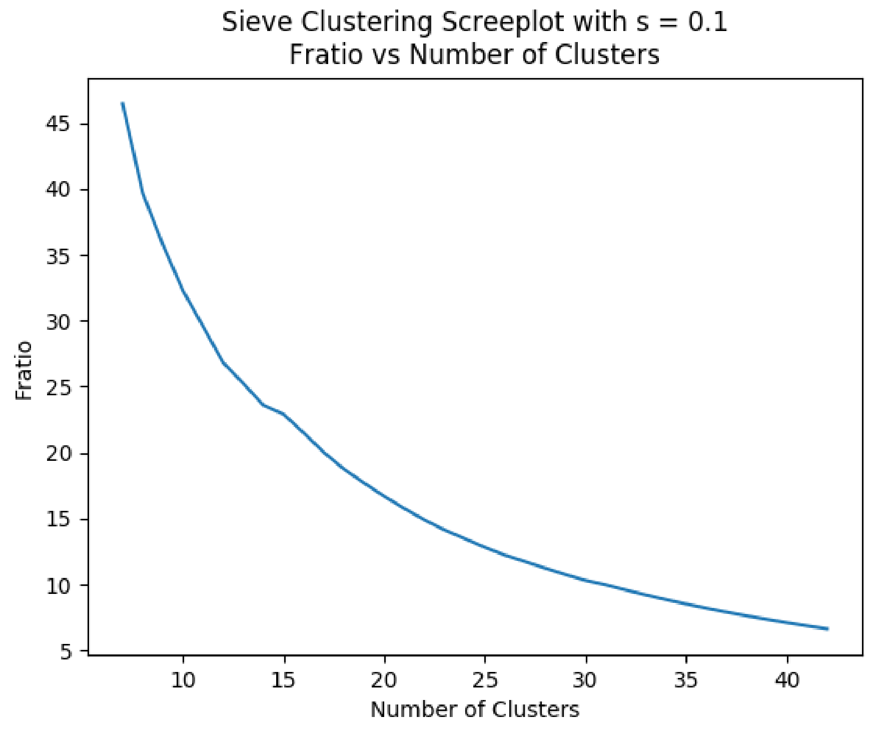


Figure 11: The screeplot of the sieve clustering of the Iris data set with s = 0.1.

Table 5: *Error Statistics of the Sieve Algorithm with s = 0.5.*

k	2	3	4	5	$\cdots$	23
MR (%)	38.0	32.67	32.67	32.0	$\cdots$	27.33
F_{ratio}	139.046	95.877	70.722	54.684	$\cdots$	12.866
p-Value	0.0675	0.0104	0.0023	0.001	$\cdots$	4.11×10^{-9}
Time (s)	0.0781	0.0781	0.0937	0.0938	$\cdots$	0.0312
Total Computational Time: 1.76 s						

for $\alpha = 0.1$. Again, we may consider the clustering at $k = 3$ a better fit since the same misclassification rate is maintained and the results are significant for $\alpha \leq 0.05$. The total computation time was 1.25 seconds. The screeplot of the clustering process in Figure 13 shows a possible elbow points at $k = 3$,.

Discussion

The smallest misclassification rate we recorded was achieved by the SOM algorithm for $k = 4$ despite the low value of the corresponding Pillai's Trace test statistic. Likewise, the largest value of Pillai's Trace occurred at $k = 2$ under the single-linkage hierarchical

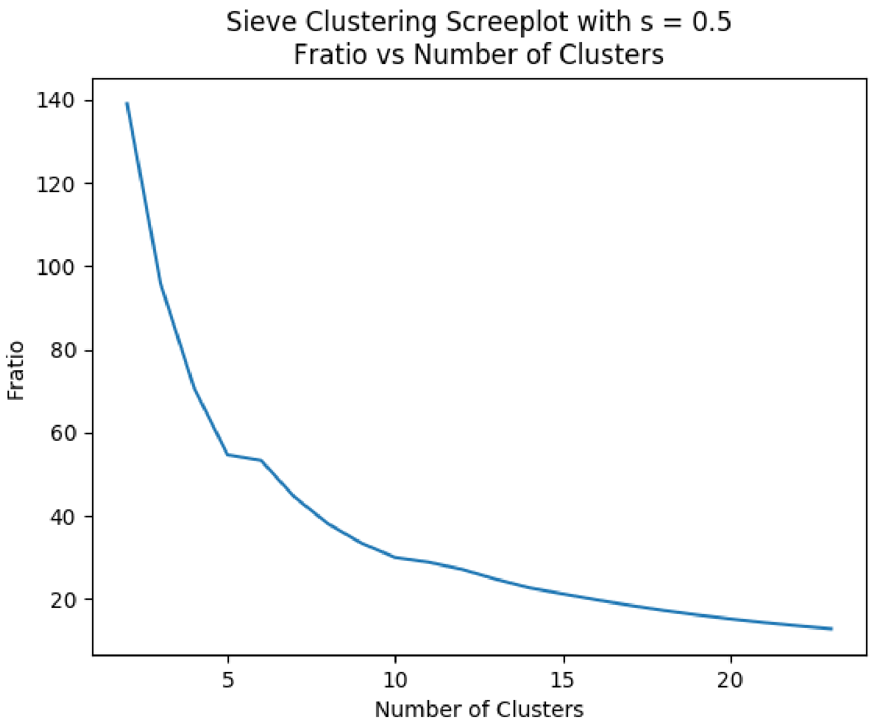


Figure 12: The screeplot of the sieve clustering of the Iris data set with s = 0.5.

Table 6: *Error Statistics of the Sieve Algorithm with s = 0.9.*

k	2	3	4	5	$\cdots$	16
MR (%)	33.33	33.33	32.0	32.0	$\cdots$	32.0
F_{ratio}	235.228	118.911	85.141	66.859	$\cdots$	17.593
p-Value	0.0519	0.0084	0.0017	0.0004	$\cdots$	1.12×10^{-7}
Time (s)	0.0815	0.0781	0.0937	0.0781	$\cdots$	0.0312
Total Computational Time: 1.25 s						

algorithm, but with a misclassification rate of 33.33%. One reason that these cluster validation criteria seemingly provide different information is that the calculation of the test statistic used only the internal information obtained from the clustering via a MANOVA while the calculation of the misclassification rate required the use of externally placed data labels (i.e. species names). The scatter plots of the Iris data set in Figure 1 show that there is a fair amount of overlap between *I. versicolor* and *I. virginica* while *I. setosa* remain mostly separate. Indeed, further analysis of the cluster structure of each algorithm result shows that while the algorithms were relatively successful at sorting the *setosa*

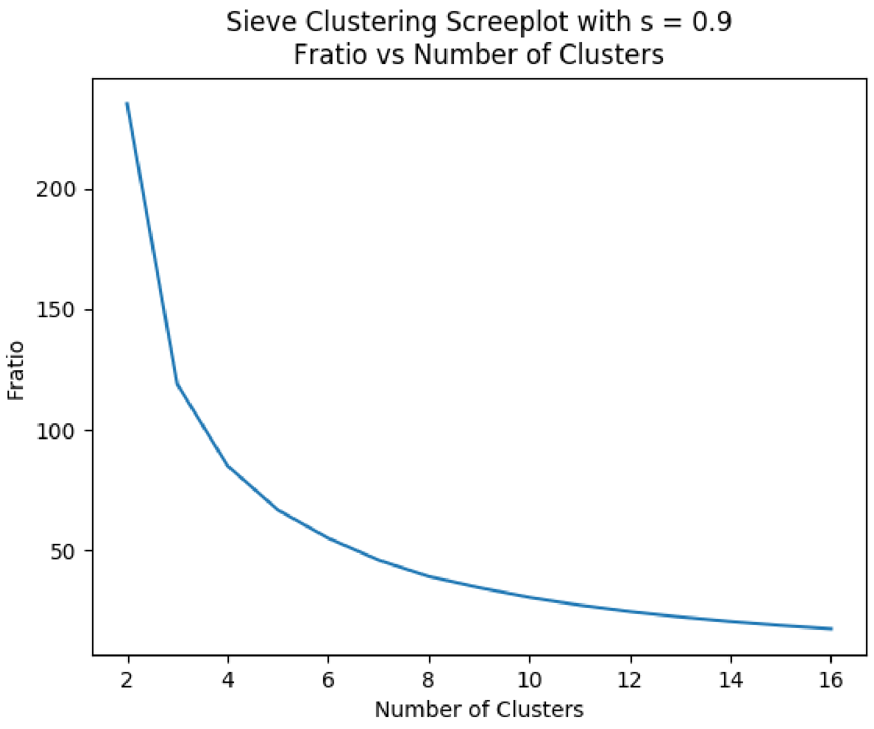


Figure 13: The screeplot of the sieve clustering of the Iris data set with s = 0.9.

species into its own cluster(s), the algorithms tended to sort the *versicolor* and *virginica* together. This would contribute to the high misclassification rates shown in Tables 1-6.

The "true" number of clusters in the Iris data set is 3. It can be argued that the error statistics of the k-means algorithm agree with this as the highest Pillai's Trace statistic was achieved at $k = 3$ even though it is associated with only the second lowest misclassification rate. The results of the hierarchical algorithm support a clustering of $k = 2$ despite the relatively high misclassification rate. While the SOM algorithm obtained a large Pillai's Trace at $k = 2$, it also reached a high misclassification rate. A clustering of $k = 4$ returned the lowest misclassification rate of all four algorithms, but also a low Pillai's Trace. The sieve method, despite the higher misclassification rates, supported a clustering of $k = 7$ at a sieve size $s = 0.1$, but supported $k = 3$ for $s = 0.5$ and $s = 0.9$.

As is made clear by Tables 4-6, the clustering outcome of the sieve algorithm is heavily dependent on the initial clustering. While the misclassification rate experiences a steady increase as the divisive procedure progresses, the test statistic that we use to measure the the goodness of the clustering becomes more favorable as well. As was the case with the results shown Tables 5 and 6, although the program may terminate at a particular value of k that obtains the optimal test statistic value, this termination may require further analysis. Of course, the outcome of the algorithm also depends on the sieve size. Small sieve sizes s produce a large number of initial clusters. They also result in a large number of final clusters when compared to the results of larger s values. Contrary to our expectations, a smaller value of s did not result in significantly better misclassification rates than the larger values. The clusterings of $s = 0.5$ and $s = 0.9$ took notably less time and actually supported the clustering at $k = 3$, unlike the clustering of $s = 0.1$, which terminated before reaching $k = 3$.

Although the k-means and SOM return lower misclassification rates than the single-linkage hierarchical and sieving algorithms, they require the user to know ahead of time the range of optimal k values to test. If the data is low-dimensional, it is easy to determine k based on scatter plots or histograms. Parallel or star coordinates could be used for slightly higher-dimensional data. For very high-dimensional data, however, it may be necessary to run other preprocessing tests or to enact dimensionality-reduction techniques.

Any hierarchical algorithm can be graphically represented as a dendogram. Even for higher-dimensional data, if the data set is relatively small, the dendogram is invaluable in determining a clustering threshold and, thus, the ideal number of clusters. For larger data sets, the linkage matrix and corresponding dendogram tend to become convoluted and difficult to read, as is nearly the case in Figure 10. The same is true for data sets of widely-varying data objects.

The sieving algorithm returned misclassification rates similar to those of the hierarchical algorithm. The high misclassification rates may have been due to the initial transformations of the data. For certain values of sieve size s, however, the algorithm was able to determine possibly optimal values of k. The sieve algorithm, then, can be used as a preliminary clustering technique to help determine a range of potentially viable values of k. A more precise clustering algorithm, such as the k-means or SOM, could then be implemented to obtain optimal clustering results.

Conclusion

Because clustering has limitless applications in fields that rely on data analysis, it is important for there to be reliable algorithms for quick and easy clustering. With this study, we introduced a new clustering method, a sieving algorithm based on the idea of using a mesh to separate finer from coarser objects. In our algorithm, these objects are high-dimensional data vectors that are common in ecological and biological studies and the mesh is a high-dimensional sieve of varying mesh sizes. Using the sieving algorithm to gain familiarity with data structure and the k-means or SOM algorithm as a follow-up method for a more accurate analysis would be an efficient approach to tackling research questions. As this is the preliminary attempt at sieve clustering, we studied its performance for 2-D data. However, with the orthogonal vector transformation given in Chapter 2, it is possible to transform any p-dimensional ($p > 2$) vector into a 2-D vector. It is also possible to extend the algorithm to work with higher-dimensional data or with other data types. Our hope is to provide researchers with tried-and-true methods to not only further their understanding of numerical data, but to ease the process of statistical analysis.

REFERENCES

[1] C. C. Aggarwal, *Data Clustering: Algorithms and Applications.* CRC Press, Boca Raton, FLorida, 2014.

[2] J. C. Bezdek. *Pattern Recognition with Fuzzy Objective Function Algorithms.* Plennum Press, New York, 1981.

[3] G. Brock, V. Pihur, S. Datta, and S. Datta, “clValid: An R Package for Cluster Validation,” *Journal of Statistical Software* **25** (2008), 1–22.

[4] B. S. Everitt, *Cluster Analysis.* John Wiley & Sons Ltd, Chichester, West Sussex, United Kingdom, 2011.

[5] R. A. Fisher, “The Use of Multiple Measurements in Taxonomic Problems,” *Annals of Eugenics 7.7* (1936), 179–188.

[6] G. Gan, C. Ma, and J. Wu, *Data Clustering: Theory, Algorithms, and Applications.* SIAM, Philadelphia, Pennsylvania, 2007.

[7] S. Ghosh, and S. Mitra, “Gene selection using biological knowledge and fuzzy clustering,” *2012 IEEE International Conference on Fuzzy Systems* (2012), 1–9.

[8] The Glowing Python, `https://glowingpython.blogspot.com/2012/04/k-means-clustering-with-scipy.html`, 2012.

[9] A. Jain, and R. C. Dubes, *Algorithms for Clustering Data.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[10] M. Charrad, N. Ghazzali, V. Boiteau and A. Niknafs, “NbClust: An R Package for Determining the Relevant Number of Clusters in a Data Set,” *Journal of Statistical Software* **61** (2014), 1–36.

[11] J. A. Hartigan and M. A. Wong, “Algorithm AS 136: A K-Means Clustering Algorithm,” *Journal of the Royal Statistical Society. Series C (Applied Statistics)* **28**, 100–108.

[12] T. J. Hastie, R. J. Tibshirani, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction.* Springer, New York, 2009.

[13] A. Inselberg, and T. Avidan “The automated multidimensional detective,” *Proceedings 1999 IEEE Symposium on Information Visualization*(1999), 112–119, 151.

[14] A. K. Jain, M. N. Murty, and P. J. Flynn, “Data Clustering: A Review,” *ACM Computing Surveys*(1999), 264–323.

[15] R. A. Johnson and D. W. Wichern, *Applied Multivariate Statistical Analysis.* Prentice Hall, 1998.

[16] Jorn's Blog, `https://joernhees.de/blog/2015/08/26/scipy-hierarchical-clustering-and-dendrogram-tutorial/`, 2015.

[17] E. Kandogan, "Star Coordinates: A Multi-dimensional Visualization Technique with Uniform Treatment of Dimensions," (2000).

[18] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley and Sons, 1990.

[19] T. Kohonen, *Self-organizing maps*. Springer, 2001.

[20] J. Lampinen and T. Kostiainen, "Self-Organizing Map in Data-Analysis - Notes on Overfitting and Overinterpretation," *In Proceedings of ESANN*(2000), 239–244.

[21] STAT 505 - Applied Multivariate Statistical Analysis, `https://onlinecourses.science.psu.edu/stat505/`, 2018.

[22] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*. Elsevier Inc., 2009.

[23] R. Xu and D. C. Wunsh, *Clustering*. John Wiley & Sons Inc, 2009.

[24] L. A. Zadeh, "Fuzzy Sets," *Information and Control*(1965), 338–353.

A portion of this work was published in *Algebraic and Combinatorial Computational Biology*, edited by R. Robeva and M. Macauley, Copyright Elsevier (2018).

APPENDIX A: PYTHON CODE

The Python code for all algorithms are included in this appendix. The Python packages NumPy and SciPy must be installed to run the code. For the graphs, Plotly and Matplotlib must be installed. The input file “irisdata1.csv” contains 150 rows and 5 columns. Each row represents a single Iris flower. The columns correspond to sepal length, sepal width, petal length, petal width, and species name. The data set can be found at `https://archive.ics.uci.edu/ml/machine-learning-databases/iris/`.

Misclassification Rate Function

The following is the code for the misclassification rate as it appears in the k-means, hierarchical, and SOM algorithms. That is, the following code is for 5-D data, where the first 4 components are numerical and the final is the data label. The misclassification rate function for the sieve algorithm is included in the sieve code.

```
def misclass(data):
    # data: numerical, clustered data set
    misstot = 0r
    for i in range(len(data)):
        cdict = {}
        maxitem = 0
        for j in range(len(data[i])):
            if data[i][j][4] not in cdict:
                cdict[data[i][j][4]] = 1
            else:
                cdict[data[i][j][4]] += 1
        # uncomment the following for printout of cluster contents
        #print("Cluster", i+1, "||", cdict)
        #print()
        # determine within each cluster which items were
```

```
            # most likely misclassified , i.e. which
            # appear less frequently
        for k in cdict.items():
            if k[1] > maxitem:
                maxitem = k[1]
                maxkey = k[0]
        cmiss = 0
        for k in cdict.items():
            if k[0] != maxkey:
                cmiss += k[1]
        misstot += cmiss
    rate = misstot / len(xdata) * 100
    print("------------------------------------------------")
    print("SUMMARY")
    print()
    print("sieve clusters = ", len(data))
    print()
    print("Misclassification rate: ", round(rate,3), "%")
    print()
```

k-means

```
import csv
from numpy import vstack,array
import numpy as np
from scipy.cluster.vq import kmeans,vq
from matplotlib import pyplot as plt
#-------------------------------------------------------------------
# import data from csv file
```

```
xdata = [] # without labels
namedata = [] # with labels
with open('irisdata1.csv') as csvfile:
    read = csv.reader(csvfile)
    next(read, None)
    for row in read:
        for i in range(4):
            row[i] = float(row[i]) # convert to numerical values
        xdata.append(row[:4])
        namedata.append(row[:5])
#--------------------------------------------------------------------------
def main():
    for k in range(1,6):
        # computing k-means
        centroids,_ = kmeans(xdata,k)
        # assign each sample to a cluster
        idx,_ = vq(xdata,centroids)
        # retrive clusters
        clusterList = [[] for i in range(max(idx)+1)]
        for j in range(len(idx)):
            clusterList[idx[j]].append(namedata[j])
        # save file in comma-delimited txt file
        outfile = open("kmeans_out_k"+str(k)+".txt", "w")
        for i in range(len(clusterList)):
            for j in range(len(clusterList[i])):
                for l in range(len(clusterList[i][j])):
                    outfile.write(str(clusterList[i][j][l])+ ",")
                outfile.write(str(i)+"\n")
        outfile.close()
        misclass(clusterList)
main()
```

Single-Linkage Hierarchical

```
from matplotlib import pyplot as plt
import scipy.cluster.hierarchy as hac
import numpy as np
import csv
#------------------------------------------------------------------------
# input vectors - import from csv file
# each row in irisdata1.csv is made up of 5 components:
#   4 numerical and 1 string value (the label of the object)
xdata = [] # without labels
namedata = [] # with labels
with open('irisdata1.csv') as csvfile:
    read = csv.reader(csvfile)
    next(read, None)
    for row in read:
        for i in range(4):
            row[i] = float(row[i]) # convert to numerical values
        xdata.append(row[:4])
        namedata.append(row[:5])
#------------------------------------------------------------------------
def main():
    # perform the clustering
    a = np.array(namedata)
    z = hac.linkage(a[:,[0,1,2,3]], "single")
    # determine a threshold to retrieve clusters
    max_d = 0.64
    idx = hac.fcluster(z, max_d, criterion = "distance")
    # create clustered data set
    clusterList = [[] for i in range(max(idx))]
    for i in range(len(idx)):
        clusterList[idx[i]-1].append(namedata[i])
    misclass(clusterList)
    # save file in comma-delimited txt file
    outfile = open("singlink_"+str(len(clusterList))+".txt", "w")
    for i in range(len(clusterList)):
        for j in range(len(clusterList[i])):
            for l in range(len(clusterList[i][j])):
                outfile.write(str(clusterList[i][j][l])+ ",")
            outfile.write(str(i)+"\n")
    outfile.close()
#----------------------------------------------------------------------
    # uncomment the following to create a dendogram
    '''
    # change label format for neater dendogram
    for i in a:
        i[4] = i[4][5:]
    plt.figure(figsize=(30, 10))
    plt.title('Single-Linkage Hierarchical Clustering Dendrogram of the Iris
    plt.xlabel('Iris Species')
    plt.ylabel('Distance')
    hac.dendrogram(
        z,
        leaf_rotation=-90.,  # rotates the x axis labels
        leaf_font_size=5.5,  # font size for the x axis labels
        labels = a[:,4]      # x labels
    )
    #plt.axhline(y=max_d,c="k") # plots the threshold
    plt.show()
    '''
#---------------------------------------------------------------
        # uncomment the following to create a scree plot
        '''
    fig, axes = plt.subplots(1,1)
    axes.plot(range(1, len(z)+1), z[::-1, 2])
    # determine possible elbow/knee points
    knee = np.diff(z[::-1, 2], 2)
    axes.plot(range(2, len(z)), knee)
    num_clust1 = knee.argmax() + 2
    axes.text(num_clust1, z[::-1, 2][num_clust1-1], 'possible\n<- knee point'
    part1 = hac.fcluster(z, num_clust1, 'maxclust')
    m = '\n(Method: {})'.format("Single-Linkage")
    plt.setp(axes, title='Screeplot{}'.format(m),
             xlabel='Number of Clusters',
             ylabel='{}\nCluster Distance'.format(m))
    plt.tight_layout()
    plt.show()
    '''
```

```
main()
```

SOM

```
import math
from decimal import *
import random
import csv
import numpy as np
import plotly as py
#----------------------------------------------------------------------
# distance function - euclidean in R^4
def dist(list_a, list_b):
    if len(list_a) == len(list_b):
        d = 0
        for i in range(len(list_a)):
            d += (list_a[i] - list_b[i])**2
        return d**(0.5)
    else:
        print("Error: Vectors are not the same dimension!")
# learning rate - exp decay
def learn(t):
    if t == 0:
        return 1
    else:
        return 1 * math.exp(-t/400)
# radius of neighborhood function - exp decay
def radius(t):
    if t == 0:
        return 1
    else:
        return 1 * math.exp(-t/400)
#------------------------------------------------------------------
def centroidvector(data):
    centroidlist = []
    for i in range(len(data)):
        centroid = []
        for k in range(2):
            sum_clust = 0
            for j in range(len(data[i])):
                sum_clust += data[i][j][k]
            centroid.append(round(sum_clust/len(data[i]),4))
        centroidlist.append(centroid)
    return centroidlist
##################################################################
#    #    #    #    #    SOM CLUSTERING    #    #    #    #    #    #
##################################################################
# randomly initialize the weight vectors
# determine number of neurons (i.e. how many clusters?)
dim1 = 1
# determine length of weigth vectors
# (i.e. length of numerical input vectors)
dim2 = 4
w = []
for i in range(dim1):
    vect = []
    for j in range(int(dim2)):
        vect.append(random.random())
    w.append(vect)
#-----------------------------------------------------------------------
# input vectors - import from csv file
xdata = []
namedata = []
with open('irisdata1.csv') as csvfile:
    read = csv.reader(csvfile)
    next(read, None)
    for row in read:
        for i in range(4):
            row[i] = float(row[i])  # convert to numerical values
        xdata.append(row[:4])       # numerical, no labels
        namedata.append(row[:5])    # numerical, with labels
#-----------------------------------------------------------------------
def main():
    # begin iterations
    for t in range(400):
        # present a random input vector to neurons
        xin = random.choice(xdata)
        # initialize min_dist by some ridiculous number
        min_dist = 50
        # determine winning neuron
        # this is the neuron closest to input vector
        for k in range(dim1):
            if min_dist > dist(xin, w[k]):
                min_dist = dist(xin, w[k])
                min_wt = k # store index of winning vector
        # update all neurons - using neighbor criteria - soft competitive
        for i in range(dim1):
            for j in range(dim2):
                diff = xin[j] - w[i][j]
                nhood = math.exp(-(dist(w[i],w[min_wt])**2)/(2*(radius(t)**2)
                w[i][j] = w[i][j] + learn(t) * nhood * diff
#------------------------------------------------------------------
    # return clusters
    somClusters = [[] for j in range(dim1)]
    for i in range(len(xdata)):
        min_dist = 100
        for j in range(dim1):
            if min_dist > dist(xdata[i],w[j]):
                min_dist = dist(xdata[i], w[j])
                min_index = j
        somClusters[min_index].append(xdata[i])
    # label the data objects
    for i in range(dim1):
        for j in range(len(somClusters[i])):
            for k in range(len(xdata)):
                if somClusters[i][j] == namedata[k][:4]:
                    somClusters[i][j].append(namedata[k][4])
#------------------------------------------------------------------
    # get summary statistics of clustering results
    misclass(somClusters)
    # save file in comma-delimited txt file
    outfile = open("som_k"+str(dim1)+".txt", "w")
    for i in range(len(somClusters)):
        for j in range(len(somClusters[i])):
            for l in range(len(somClusters[i][j])):
                outfile.write(str(somClusters[i][j][l])+ ",")
            outfile.write(str(i)+"\n")
    outfile.close()
main()
```

Sieve

```
import math
import random
import csv
import numpy as np
from scipy.stats import f
import plotly as py
import plotly.graph_objs as go
from matplotlib import pyplot as plt
#------------------------------------------------------------------------
# distance function - euclidean in R^2
def dist(list_a, list_b):
    if len(list_a) == len(list_b):
        d = 0
        for i in range(len(list_a)):
            d += (list_a[i] - list_b[i])**2
        return d**(.5)
    else:
        print("Error: Vectors are not the same dimension!")
#———————————————————————————————————————————————————————————————
# vector projection into 2D
def proj(data):
    # data: dataset as list where final component is label
    # define independent vectors - can be random
    w1 = [1,2,3,4]
    w2 = [3,1,5,7]
    # orthonormal basis time
    w1_hat = w1 / np.linalg.norm(w1)
    x2 = w2 - np.dot(w2, w1_hat) * w1_hat
    w2_hat = x2 / np.linalg.norm(x2)
    w1_hat = w1_hat.tolist()
    w2_hat = w2_hat.tolist()
    # 2-D data
    data_2d = []
    for i in range(len(data)):
        v = data[i][:4] # without label
        x_var = np.dot(v, w1_hat)
        y_var = np.dot(v, w2_hat)
        data_2d.append([round(x_var,3),
                   round(y_var,3),data[i][4]]) # with label
    return data_2d
#———————————————————————————————————————————————————————————————
def normalize(data,dim):
    # data: dataset as list, final component is label
    # dim: length of numerical vector, i.e. without label
    normdata = []
    for i in range(len(data)):
        v = data[i][:dim] # without label
        normv = v / np.linalg.norm(v)
        normv = normv.tolist()
        for j in range(len(normv)):
            normv[j] = round(normv[j],4) # round to 4 dec places
        normv.append(data[i][dim]) # with label
        normdata.append(normv)
    return normdata
#------------------------------------------------------------------
# misclassification rate for 2D clusters
def misclass(data):
    # data: numerical, clustered data set
    misstot = 0
    for i in range(len(data)):
        cdict = {}
        maxitem = 0
        for j in range(len(data[i])):
            if data[i][j][2] not in cdict:
                cdict[data[i][j][2]] = 1
            else:
                cdict[data[i][j][2]] += 1
        # uncomment for printout of cluster content
        #print("Cluster", i+1, "||", cdict)
        #print()
        # determine within each cluster which items were
```

```
            # most likely misclassified, i.e. which
            # appear less frequently
        for k in cdict.items():
            if k[1] > maxitem:
                maxitem = k[1]
                maxkey = k[0]
        cmiss = 0
        for k in cdict.items():
            if k[0] != maxkey:
                cmiss += k[1]
        misstot += cmiss
    rate = misstot / len(xdata) * 100
    print("--------------")
    print("SUMMARY")
    print()
    print("sieve clusters = ", len(data))
    print()
    print("Misclassification rate: ", round(rate,3), "%")
    print()
#------------------------------------------------------------
# returns a vector of cluster means
def centroidvector(data):
    centroidlist = []
    for i in range(len(data)):
        centroid = []
        for k in range(2):
            sum_clust = 0
```

```
                for j in range(len(data[i])):
                    sum_clust += data[i][j][k]
                centroid.append(round(sum_clust/len(data[i]),4))
            centroidlist.append(centroid)
        return centroidlist
#-----------------------------------------------------------------------
# calculate the within-cluster sum of squares and cross products
def SSW(data, centroidvector):
    sswComp = []
    for k in range(2):
        ssw = 0
        quant = 0
        for i in range(len(data)):
            for j in range(len(data[i])):
                quant += (data[i][j][k]-centroidvector[i][k])**2
        sswComp.append(quant)
    return sswComp
#-----------------------------------------------------------------------
# calculate the between-clusters sum of squares
def SSB(data, centroidvector):
    # find the mean of the centroids
    grandMean = []
    for k in range(2):
        sum_comp = 0
        num_elem = 0
        for i in range(len(data)):
            sum_comp += len(data[i])*centroidvector[i][k]
            num_elem += len(data[i])
        grandMean.append(round(sum_comp/num_elem,4))
    # find the SSB
    ssbComp = []
    for k in range(2):
        ssb = 0
        quant = 0
        for i in range(len(data)):
            diff = centroidvector[i][k]-grandMean[k]
            quant = len(data[i])*(diff)**2
            ssb += quant
        ssbComp.append(ssb)
    return [grandMean, ssbComp, num_elem]
#----------------------------------------------------------------------
# sum of squares and cross-products of a MANOVA
# used to calculate the Wilks lambda test statistic
# only for 2-d data
def SSCP(data):
    # component SSB and SSW
    centroidList = centroidvector(data)
    grandMean, ssb, dataNum = SSB(data, centroidList)
    ssw = SSW(data, centroidList)
    # cross-product contributions
    # between
    cpb = 0
    for i in range(len(data)):
        b1 = centroidList[i][0] - grandMean[0]
```

```
            b2 = centroidList[i][1] - grandMean[1]
            cpb += len(data[i])*b1*b2
        # within
        cpw = 0
        for i in range(len(data)):
            for j in range(len(data[i])):
                w1 = data[i][j][0]-centroidList[i][0]
                w2 = data[i][j][1]-centroidList[i][1]
                cpw += w1*w2
        # sum of squares & cross-product matrices
        B = np.array([[ssb[0],cpb],[cpb, ssb[1]]])
        W = np.array([[ssw[0],cpw],[cpw,ssw[1]]])
        # calculate the Wilks lambda
        Lambda = np.linalg.det(W)/np.linalg.det(W+B)
        # calculate the test statistic
        if len(data) >= 2:
            df1 = dataNum-len(data)-1
            df2 = len(data)-1
            Fratio = (df1/df2)*((1-math.sqrt(Lambda))/math.sqrt(Lambda))
        elif len(data) == 1:
            Fratio = 0
        return Fratio
#--------------------------------------------------------------------
# input vectors - import from csv file
# each row in irisdata1.csv is made up of 5 components:
#   4 numerical and 1 string value (the label of the object)
xdata = [] # without labels
```

```
namedata = [] # with labels
with open('irisdata1.csv') as csvfile:
    read = csv.reader(csvfile)
    next(read, None)
    for row in read:
        for i in range(4):
            row[i] = float(row[i]) # convert to numerical values
        xdata.append(row[:4])
        namedata.append(row[:5])
####################################################################
#    #    #    #    Sieve Clustering    #    #    #    #    #    #    #
####################################################################
def main():
    # prepare the data for sieving - transformations
    # shift the origin to the mean so that it is the
    #  "center" of our data cloud
    # get data mean (4-D)
    datamean = []
    for j in range(len(xdata[0])):
        compsum = 0
        for i in range(len(xdata)):
            compsum += xdata[i][j]
        datamean.append(round(compsum / len(xdata),4))
    # shift the rest of the data accordingingly
    # to do this, subtract mean from each data object
    trnamedata = []
    for i in range(len(xdata)):
        trx = []
        for j in range(len(xdata[0])):
            trx.append(round(xdata[i][j]−datamean[j],4))
        trx.append(namedata[i][4])
        trnamedata.append(trx)
    # projection of data into 2−D and normalization
    trdata_2d = normalize(proj(trnamedata),2)
    #trdata_2d = proj(trnamedata)
#−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    # SIEVE TIME
    # create a copy of the data
    coarse = trdata_2d[:]
    # set the size of the sieve, ss \in (0,1)
    ss = 0.9
    print("Sieve size = ", ss, end = "\n\n")
    # initialize list to store sieving results
    sieveClust = []
    # initialize a set of all possible theta to choose from
    thetalist = list(range(0,180))
    # set a counter for number of iterations
    t = 0
    # will run iterations until either all points have gone
    #   through the sieve or until we reach 150 iterations
    while len(coarse) > 0 and t < len(xdata):
        # when there is only one element left to be clustered, no point
        #   in running through iterations
        if len(coarse) == 1:
```

```
        sieveClust.append([coarse[0]])
        coarse.remove(coarse[0])
    # introduce a sieve
    # remove from thetalist so that it cannot be chosen again
    theta = random.choice(thetalist)
    thetalist.remove(theta)
    # convert from degrees to radians
    theta = math.radians(theta)
    # create a sieve surface vector
    # on each side of sieve surface, all points perpendicular
    #   or near-perpendicular run through the sieve and
    #   are removed from the data cloud
    posperp = []
    negperp = []
    # every element in coarse data set is treated as a vector and
    #   is compared to the sieve surface vector
    # if perpendicular (plus/minus sieve size), then they are
    #   placed into posperp/negperp
    for i in coarse:
        if math.fabs(np.dot([math.cos(theta),math.sin(theta)],
                                i[:2])) <= ss:
            # vector is posperp if dot product is nonnegative
            if np.dot([math.cos(theta),math.sin(theta)],
                                i[:2]) >= 0:
                posperp.append(i)
            else:
                negperp.append(i)
                coarse.remove(i)
        # to ensure there are no empty clusters, check the lengths
        #   of posperp/negperp before appending to sieve-clustered list
        if len(posperp) > 0:
            sieveClust.append(posperp)
        if len(negperp) > 0:
            sieveClust.append(negperp)
        # one iteration has been performed, add to counter
        t += 1
    misclass(sieveClust)
#-------------------------------------------------------------------
    # caculate the error stats of the initial clustering
    Fratio = SSCP(sieveClust)
    FList = [Fratio]
    p_val = 1-f.cdf(Fratio,(len(xdata)-len(sieveClust)-1),
                len(sieveClust)-1)
    print("Initial Fratio = ", Fratio)
    print("p-Value = ", p_val)
    print("\n Time Elapsed = ", end1-start)
    centroidList = centroidvector(sieveClust)
    # initialize boolean - will terminate loop when
    #   Fratio is maximized
    proceed = True
    while (proceed == True) & (len(sieveClust) > 1):
        minDist = 50
        # identify the closest pair of clusters
        # uses centroid-linkage
```

```
for i in range(len(centroidList)):
    for j in range(len(centroidList)):
        if (i != j) & (dist(centroidList[i],
                        centroidList[j]) <= minDist):
            minDist = dist(centroidList[i],
                       centroidList[j])
            idx1 = i
            idx2 = j
# combine the closest clusters
newClust = sieveClust[idx1]+sieveClust[idx2]
sieveClustNew = sieveClust[:]
sieveClustNew.remove(sieveClust[idx1])
sieveClustNew.remove(sieveClust[idx2])
sieveClustNew.append(newClust)
# calculate new Fratio
FratioNew = SSCP(sieveClustNew)
# if the new Fratio is larger than the current, the
#   combining process will continue
if FratioNew >= Fratio:
    sieveClust = sieveClustNew
    centroidList = centroidvector(sieveClustNew)
    Fratio = FratioNew
    FList.append(Fratio)
    end2 = time.time()
    misclass(sieveClust)
    p_val = 1-f.cdf(Fratio,(len(xdata)-len(sieveClust)-1),
        len(sieveClust)-1)
```

```
            print("Fratio = ", Fratio)
            print("p-Value = ", p_val)
        else:
            proceed = False
#--------------------------------------------------------------
    # scree plot!
    n = len(sieveClust)
    rFList = FList.sort(reverse = True)
    plt.plot(range(n,n+len(FList)), FList)
    plt.title("Sieve Clustering Screeplot with s = " + str(ss) +
              "\nFratio vs Number of Clusters")
    plt.xlabel("Number of Clusters")
    plt.ylabel("Fratio")
    plt.show()
main()
```