ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

Fakulta strojní – Ústav přístrojové a řídicí techniky

BACHELOR THESIS

CLUSTERING METHODS FOR DATA ANALYSIS

KLASTROVACÍ METODY PRO DATOVOU ANALÝZU

Kryštof Bystřický

2018/2019

Prohlašuji, že jsem tuto práci vypracoval(a) samostatně s použitím literárních pramenů a informací, které cituji a uvádím v seznamu použité literatury a zdrojů informací.

Datum: ....

podpis

BACHELOR‘S THESIS ASSIGNMENT

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

I. Personal and study details

Student's name:	**Bystřický Kryštof**	Personal ID number:	**465337**
Faculty / Institute:	**Faculty of Mechanical Engineering**		
Department / Institute:	**Department of Instrumentation and Control Engineering**		
Study program:	**Theoretical Fundamentals of Mechanical Engineering**		
Branch of study:	**No Special Fields od Study**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Clustering methods for data analysis

Bachelor’s thesis title in Czech:

Klastrovací metody pro datovou analýzu

Guidelines:

1) Create a summary of clustering methods
2) Describe important clustering methods and apply them using Python and appropriate libraries
3) Describe chosen algorithm/algorithms
4) Suggest an application of chosen methods and apply them on a suitable dataset
5) Describe a way to interpret clustering outputs

Bibliography / sources:

[1] J. C. Bezdek, R. Ehrlich, a W. Full, „FCM: The fuzzy c-means clustering algorithm', Comput. Geosci., roč. 10, č. 2–3, s. 191–203, led. 1984.
[2] Tan, Pang-Ning, et al. Introduction to Data Mining. 1st ed, Pearson Addison Wesley, 2006.
[3] Leskovec, Jurij, et al. Mining of Massive Datasets / Jure Leskovec, Standford University, Anand Rajaraman, Milliways Labs, Jeffrey David Ullman, Standford University. Second edition, Cambridge University Press, 2014.

Name and workplace of bachelor’s thesis supervisor:

Ing. Mgr. Jakub Jura, Ph.D., U12110.3

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **26.04.2019** Deadline for bachelor thesis submission: **12.06.2019**

Assignment valid until: ____________

[illegible] [illegible] [illegible]

Ing. Mgr. Jakub Jura, Ph.D.
Supervisor’s signature

Head of department’s signature

prof. Ing. Michael Valášek, DrSc.
Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Bystřický

Student’s signature

Abstract

In this paper, basic data mining techniques will be described. A closer look is then taken at cluster analysis-the task of grouping objects based on their similarity. A significant portion of the theoretical part deals with data preprocessing and visualization. Clustering methods are applied with Python scripts utilizing appropriate data mining Python modules. The analyzed data set is an array of composite tube vibration measurements data. In the practical part, the data is first transformed into frequency characteristics, yielding a frequency characteristic for each tube. The data is then preprocessed and clustered, with the goal of sorting the tubes into 4 groups based on the structure of the tube's composite material.

Abstrakt

V této práci jsou popsány základní techniky data miningu. Podrobně je popsána především klastrová(shluková) analýza, tedy seskupování objektů na základě jejich podobností. Značná část teoretické části se zabývá předpřípravou a vizualizací dat. Klastrovací metody jsou aplikovány pomocí Python skriptů za použití příslušných Python modulů pro data mining. Analyzovaná datová sada je soubor dat z měření vibrací kompozitových trubek. V praktické části tyto data nejdříve transformuji na frekvenční charakteristiky-každé trubce náleží jedna frekvenční charakteristika. Tyto data jsou poté předzpracovány a klastrovány za účelem roztřídění trubek do 4 skupin podle struktury kompozitu ze kterého je trubka vyrobena.

Keywords

Clustering analysis, clusters, data analysis, data mining, k-means, fuzzy c-means, hierarchical clustering, centroid, image segmentation, partitional clustering, composite materials, composite structures, Python, sklearn, skfuzzy, scikit

Klíčové slova

Klastrovací analýza, klastery, datová analýza, data mining, k-means, fuzzy c-means, hierarchické klastrování, centroid, segmentace obrazu, rozdělovací klusterování, kompozitní materiály, kompozitní struktury, Python, sklearn, skfuzzy, scikit

Contents

Part I

Theoretical part

1 Data Mining

The rapid growth of information technology enabled the collection, accumulation and analysis of massive volumes of data. Data mining refers to the analysis step of KDD(Knowledge Discovery in Databases). KDD was defined by Usama Fayyad as a "field concerned with the development of methods and techniques for making sense of data."[11] KDD is a series of steps that turns high volumes of low-level data, which is practically impossible to interpret directly, into higher-level data. Higher-level data can be thought of as a reduced version of the original data that is suitable for interpretation.[2][11]

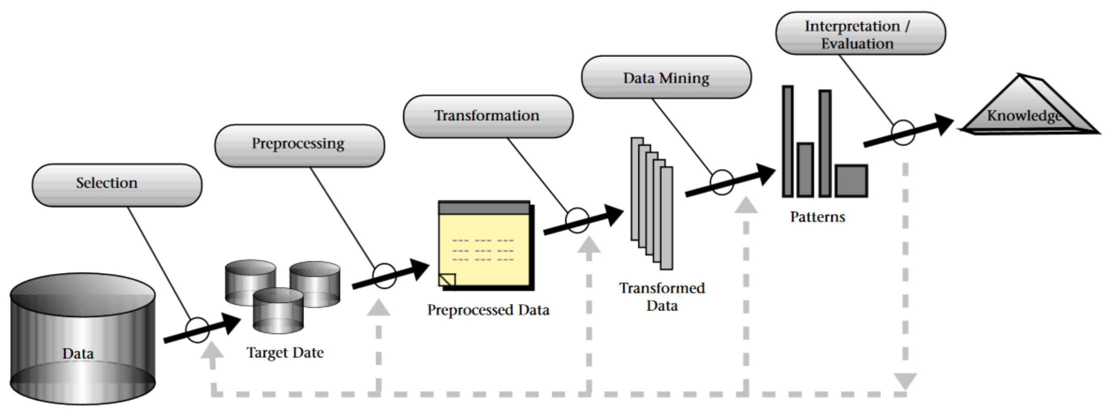


Fig. 1: "An overview of the steps that compose the KDD process."[11]

1.1 Supervised and unsupervised tasks

1.1.1 Supervised tasks

Supervised tasks work with a set of objects with labels(results) created by a "supervisor". The goal of supervised(predictive) tasks is to find relationships between independent input variables and resultant labels. The discovered relationships are then used for automatic classification or estimation of newly received, unlabeled data objects.[2][1]

1.1.2 Unsupervised tasks

Unsupervised(descriptive) tasks reduce raw, unlabeled data into patterns that best describe the measured system. Descriptive analysis seeks to uncover previously hidden patterns within the data.
Unsurprisingly, unsupervised algorithms aren't provided with any labeled data. These algorithms generally assume that observations with similar features result in identical labels, so they label object elements based on their relative proximity.[2][1]

1.2 Data mining techniques

1.2.1 Classification and regression

Both techniques fall under supervised tasks. Their goal is identical: find relationships between the features(predictor variables) $x_1, x_2, ..., x_N$ and the assigned label(response variable) y in the training data. These relationships are then used to build a statistical, predictive model that can assign a label to every new, unlabeled object element based on its features. Before classification/regression itself, we should make that all the features of the training data actually have an effect on the response variable.[6] Redundant features won't make these tasks impossible, but they slow down the learning process and they cause overfitting of the model. An overfitted model is a model that is too closely tied to the training data. Such a model misunderstands the inherent noise as a predictor, resulting in poor prediction ability when evaluating new data. The relevance of features is quantified by relevance and redundancy analysis.[12] [6]

Classification Classification predicts and assigns a categorical(discrete, unordered) labels to unlabeled objects. Since classification is a supervised task, it necessarily consists of two steps:

- Learning step

  Various algorithms are used to find relationships between labels of training(labeled) data and its features. A statistical predictive model(classifier) is built based on these discovered relationships.

- Classification step

  New, unlabeled objects are given to the classifier. The objects are then classified(labeled)

Predictive models can be represented by many different forms, such as decision trees, neural networks or IF-THEN rules.[6]

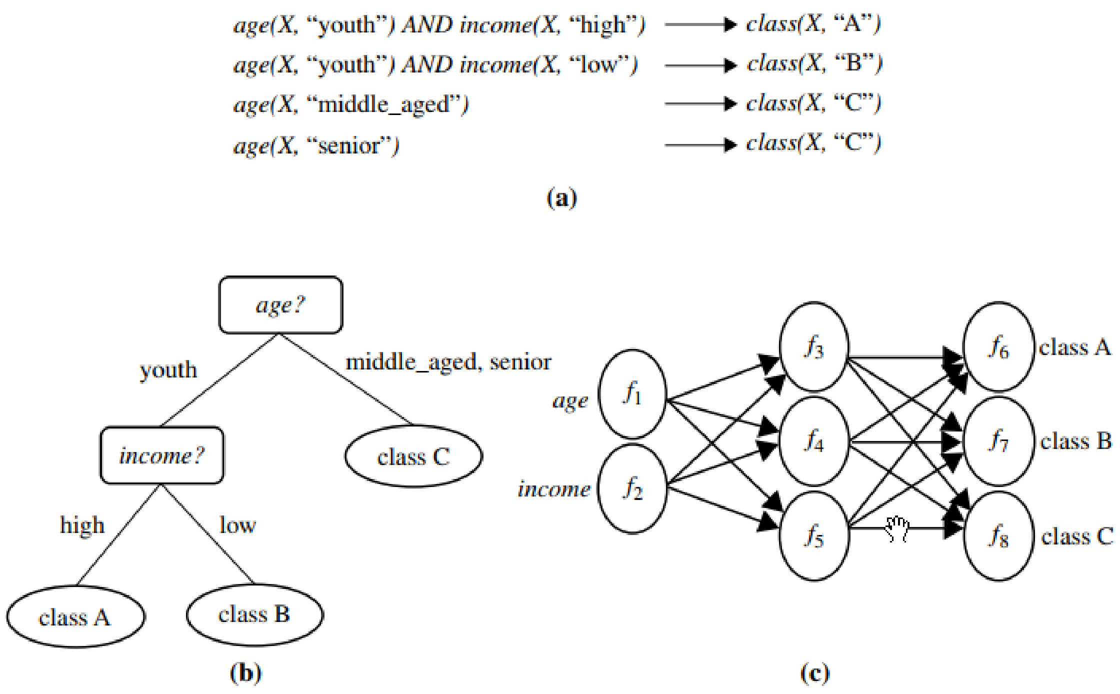

age(X, "youth") AND income(X, "high") → class(X, "A")
age(X, "youth") AND income(X, "low") → class(X, "B")
age(X, "middle_aged") → class(X, "C")
age(X, "senior") → class(X, "C")
(a)
age?
youth
middle_aged, senior
income?
class C
high
low
class A
class B
(b)
age
income
f1
f2
f3
f4
f5
f6
f7
f8
class A
class B
class C
(c)

Fig. 2: (a) IF-THEN rules, (b) decision tree, (c) neural network. Taken from [6]

Regression Regression models are used to estimate numerical, continuous response values. Similarly to classification, its function can be described in two steps:

- Fitting step

  Find a function that is an optimal representation of the relationships between predictor variables and response variables.

- Prediction step

  Use the fitted function to predict the output value of each new data element.

1.2.2 Clustering

Clustering is an unsupervised method that attempts to group objects into classes–clusters such that objects within one cluster are similar and objects in differing clusters are dissimilar. As an unsupervised method, clustering algorithms work with unlabeled data sets. The goal of clustering analysis is to uncover the natural structure of the data sets and return appropriate cluster assignations for each data object. Clustering is convenient when working with large data sets. These data sets generally are nearly always unlabeled, as labeling would be very time consuming and expensive.

A good example of such data sets is recorded speech, as there is little difficulty in obtaining the data, but labeling the data – specifying what word is actually being said at every instant, is the difficult part. This data can, however, be automatically clustered by a clustering algorithm. In this case, every cluster would represent each word uttered in the recording. Clustered data can then be interpreted by a human classifier, who assigns an appropriate label to each cluster.[10].

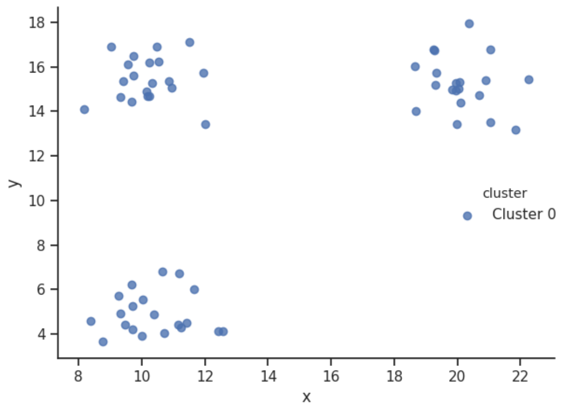


Fig. 3: Before clustering.

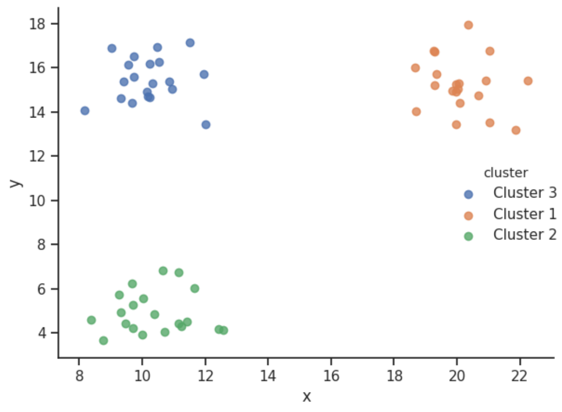


Fig. 4: After clustering.

Clustering methods can be exploited for different purposes. Its most common applications are in analysis or processing of multivariate data[5]. Practical applications of clustering span a wide variety of fields. In engineering, examples of clustering applications are pattern recognition or signal analysis. For example, it can be used for finding different intraday household electricity demand profiles[13]. It can also be used for data compression for calculations with massive data sets[2]. Clustering analysis is used in the pharmaceutical industry, where gene expression data sets with high dimensionality(10 000) aren't uncommon[9]. It is also widely used for analysis of social networks, where clustering enables targeted advertising by separating people by their interests or other characteristics. Clustering analysis is also used in outlier detection, where it helps find objects with a high dissimilarity to others. Outlier detection can then be used for fraud detection or fault detection. [6]

2 Data Representation

2.1 Data matrix

A data set $\mathbf{X}$ consists of $\mathbf{N}$ objects $\mathbf{x_i}$ $(\mathbf{i = 1, 2, ..., N})$. Each object can be described by a multidimensional vector with $\mathbf{d}$ dimensions(features) which are denoted by $\mathbf{x_{id}}$ $(\mathbf{d = 1, 2, ..., d})$. We can think of a data set as a $\mathbf{N \times d}$ matrix, called the data matrix, where each row is a separate object(data vector) and each column is a separate feature. The dimensionality $\mathbf{d}$ of a data set is a measure of how many features(variables) represent each data object[1][6].

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1d} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & x_{N3} & \dots & x_{Nd} \end{bmatrix}$$

2.2 Dissimilarity matrix

A dissimilarity matrix is an $\mathbf{N \times N}$ matrix, whose elements represent the dissimilarity indexes, or the dissimilarities(distances) between pairs of data vectors. More specifically, a dissimilarity matrix element $\mathbf{d(i, j)}$ evaluates the dissimilarity between the data objects $\mathbf{x_i}$ and $\mathbf{x_j}$. The matrix is symmetrical, as the dissimilarity of $\mathbf{x_i}$ to $\mathbf{x_j}$ is assumed to be equivalent to the dissimilarity of $\mathbf{x_j}$ to $\mathbf{x_i}$. Diagonal elements are always equal to 0, since there is no dissimilarity between a data object and itself. We can calculate the dissimilarity matrix using the data matrix and a chosen dissimilarity measure. [3][1][6]

$$\begin{bmatrix} 0 & d(1,2) & d(1,3) & \dots & d(1,N) \\ d(2,1) & 0 & d(2,3) & \dots & d(2,N) \\ d(3,1) & d(3,2) & 0 & \vdots & d(3,N) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ d(N,1) & d(N,2) & d(N,3) & \dots & 0 \end{bmatrix}$$

The total number of dissimilarity calculations to create a dissimilarity matrix is n^2. Since it is symmetrical and the values on its diagonal are always 0, we can calculate only the stricly lower triangular matrix(lower triangular minus the diagonal) and optimize the number of calculations to $\frac{n^2-n}{2}$. The computational time complexity of a dissimilarity matrix calculation is $\mathcal{O}(n^2)$.

2.3 Feature types

Features can be either continuous, discrete or binary. By official definition, continuous values can take on an infinite amount of values even within a restricted range. In other words, there is no inherent elemental step for continuous values. Examples of continuous variables are time, temperature, length etc. Since real measurements can never be truly continuous and computers inherently work only with discrete values, we usually understand continuous values as values with a high number of possible values("high number" is arbitrary, but we can assume it's $> 10^3$). Discrete values can take on a limited number of values in a restricted range. They have an inherent elementary step. Discrete features can be for example the number of people, number of occasions or even non-numerical values like colors or names. A special type of discrete values are binary values, which can only take on two possible values. For example good/bad, on/off, yes/no. [1][3].

Another important distinction between features is their measurement level, which distinguishes features based on the relative significance of their values. There are four scales of measurement, nominal, ordinal, interval, and ratio[3]:

- Nominal[3]. These features are represented by string labels. Any mathematical calculation is nonsensical, as there is no implied order and the differences between the labels have no mathematical meaning. In practice, a data set with nominal values uses numbers as placeholders for each label. When represented by numbers, a value between two elementary steps is meaningless. Examples of such features are names, colors, facial expressions, materials etc. More specifically, a column with materials might take on values 1-4, where 1 = *steel*, 2 = *copper*,

$3 = brass$, $4 = aluminium$.

- Ordinal[3][1]. Represented similarly as nominal features. They have an implied order, but only in relation to one another. The separation between values remains meaningless, as does their ratio. Examples of ordinal values are school grades, rating scales or shirt sizes.
- Interval[3]. These features are represented by numbers. The separation between numbers becomes meaningful, the ratio remains meaningless. A common example is the Celsius scale, as ratios are meaningless for scales with no natural zero.
- Ratio[3]. Represented similarly is interval features. They have a natural zero, so both separation and ratios between its values make sense. A good example of a ratio scale is the Kelvin scale, or the frequency scale. We can say that a frequency of $20Hz$ is $2\times$ larger than a frequency of $10Hz$.

Nominal and ordinal features are often referred to as qualitative or categorical, while interval and ratio features are known as quantitative[1].

2.4 Time series

A time series $\mathbf{x_i}$ is an ordered sequence of T values $\mathbf{x_{it}(t = 1, 2, \ldots, T)}$, where T is the number of measurements in a certain range. Time series describe the changes of the observed value in either the time domain or frequency domain. Each time series is a separate data object, that can be represented by a single vector with dimensionality $d = T$. Examples of time series are the daily closing prices of gold, air temperatures during the day or the displacement of a pendulum. Time series are often only one-dimensional(meaning they show only one variable as a function of time), and can easily be visualized with a simple 2-D plot. The term time series is often used for sequences that aren't ordered chronologically, such as frequency characteristics, where values are instead ordered by frequency.

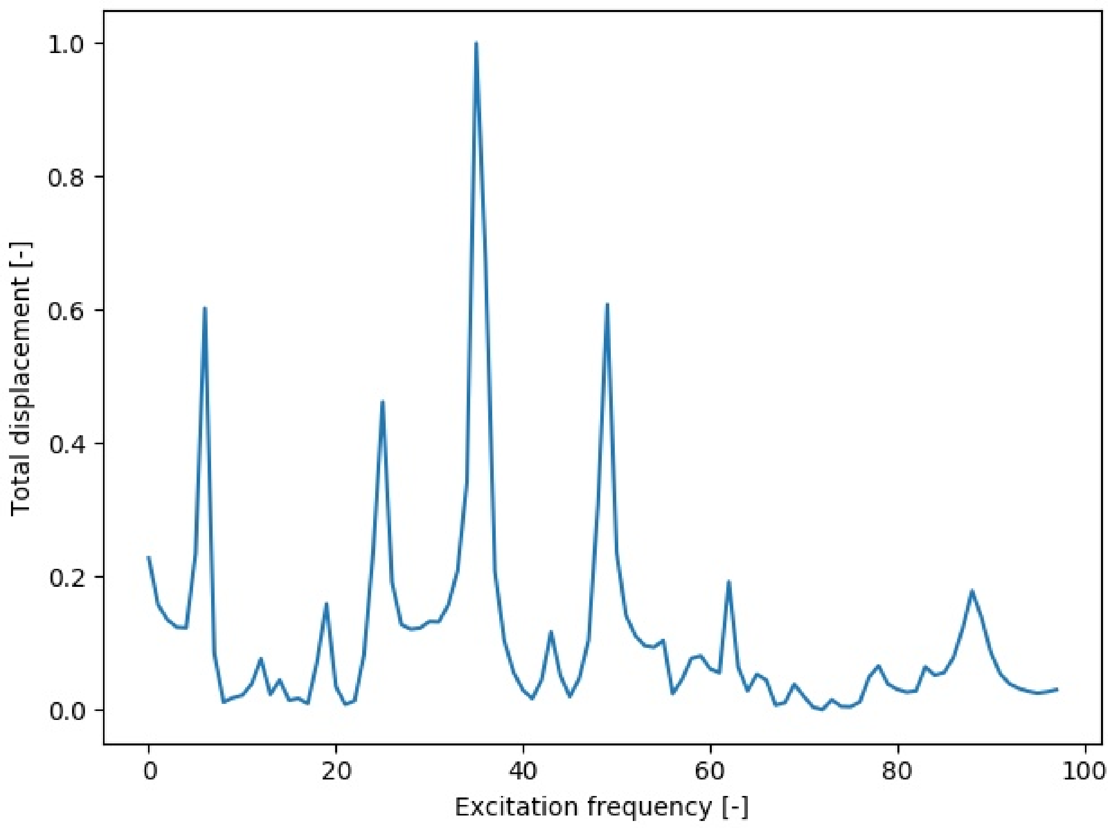


Fig. 5: Example of a single time(frequency) series object.

2.5 Data preprocessing

Unfortunately, raw data is rarely suitable for analysis. A cricital preprocessing step is dealing with the missing values; without it, analysis wouldn't be possible at all. Other steps like normalization ensure that all features are given an equal weight for the analysis. When working with huge data sets, we might want to reduce the number of objects in our data set(numerosity reduction). Data sets with huge dimensionality are often required to go through a dimensionality reduction step, as analysis in high-dimensional spaces becomes very difficult(the curse of dimensionality[2]). As the number of steps in data processing can be large, and it might not be obvious which preprocessing methods are even suitable for the given task, data preprocessing is often the most time consuming task of a data scientist.

2.5.1 Dealing with missing values

In practice, data sets often contain missing values. If the number of data objects(vectors) with missing features is way smaller than the total number of data objects within the data set, we can simply discard the incomplete objects. Since this is rarely the case, many different strategies were developed to either replace the missing features or to calculate the dissimilarity indexes between incomplete objects.

- We can calculate the dissimilarity of two data vectors by neglecting the contributions of missing features on the total similarity. The dissimilarity index $D[x_i, x_j]$ is then calculated followingly:

$$D(x_i, x_j) = \frac{d}{d - \sum_{l=1}^{d} \delta_{ijl}} \sum_{\substack{all\ l \\ \delta_l = 0}} d_l(x_{il}, x_{jl}) \tag{1}$$

  [3][1] where $d_l(x_{il}, x_{jl})$ is the dissimilarity(distance) between data vectors $\mathbf{x_i}, \mathbf{x_j}$ in the l-th dimension and δ_{ijl} is a binary value that takes on a value of 1 if a feature l is missing in at least one of the data vectors $\mathbf{x_i}, \mathbf{x_j}$, otherwise it is 0.

- Suppose we have a data vector with a missing feature. If we can calculate a dissimilarity matrix, we can find K nearest neighbors of our incomplete data vector, while requiring the neighbors to have a value for the missing feature. Then we estimate the missing feature as an average of the data vector's K nearest neighbors. The value of K is arbitrary, but should be proportional to N, the size of our dataset.[1]

Estimating feature values imposes a bias on the data, as objects with all features available have an influence on objects whose feature values had to be estimated. If a certain feature is missing a value in too many objects, we should consider ignoring this feature completely, as the quality of the estimation would likely be poor and the bias would be too high.

2.5.2 Data normalization

Data mining methods that work with the notion of dissimilarity often perform better when the data set's features have been normalized. Most common dissimilarity measures, for example the euclidean distance, generally assign more weight to features with higher ranges. To illustrate the unsuitability of unnormalized data for clustering purposes, assume a data matrix of 2-dimensional objects, where the range of values in the 1st feature x_{i1} is $1000\times$ larger than the range of the values in the 2nd feature x_{i2}. When evaluating the relative distance of the object pairs, the dissimilarity contributions of the 1st feature will completely outweigh the contributions of the 2nd feature, and the objects will appear 1-dimensional. As a result, the 2nd feature's dissimilarity contributions are effectively neglected and any information they brings us is lost.

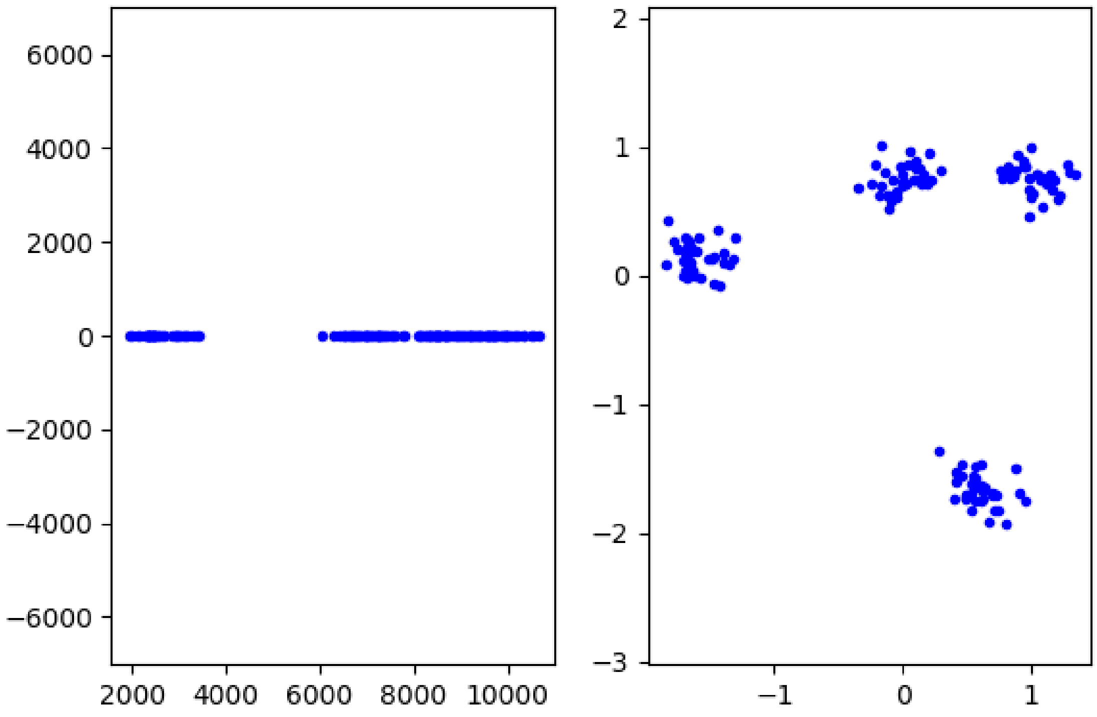


Fig. 6: A generated data set with upscaled variances and centroids in the 1st(x) dimension. Unnormalized on the left, normalized on the right. In the unnormalized data, we can see 2 or 3 clusters. After normalization, we can clearly see 4 prominent clusters.

Normalizing the data set transforms it into a form where all features have an equal range of values. Assume an $N \times d$ data matrix $\mathbf{X}^* = [x_1^*,\ x_2^*,\ \ldots,\ x_N^*]$, where each data vector $x_i^* = [x_{i1}^*,\ x_{i2}^*,\ \ldots,\ x_{id}^*]$. The asterisk denotes that the data matrix(and the data vectors) is not normalized. The most common normalization technique is to translate and scale the feature axes so that each feature has zero mean and unit variance[1]. This operation is known as standardization. The jth feature mean m_j and the jth feature variance σ_j^2 are defined as follows[1]:

$$m_j = \frac{1}{N}\sum_{i=1}^{N} x_{ij}^* \tag{2}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} (x_{ij}^* - m_j)^2 \tag{3}$$

feature values are then transformed according to this formula [1]:

$$x_{ij} = \frac{x^*_{ij} - m_j}{\sigma_j} \tag{4}$$

Unfortunately, normalizing isn't universally desirable, and in some cases it might even worsen our results. For example when a larger range of a certain feature is caused by a large distance between clusters in that feature. The global feature variance for this feature will then be far larger than the inherent intercluster variance, and rescaling the feature will cause the cluster separation in the feature to be less prominent. In this case, rescaling the feature will also significantly change the shape of the cluster, which might cause problems for some centroid based clustering methods such as k-means, which works best when the clusters are globular – have equal inter-cluster variances for each feature. Notice the change in the relative distances between the clusters. The two

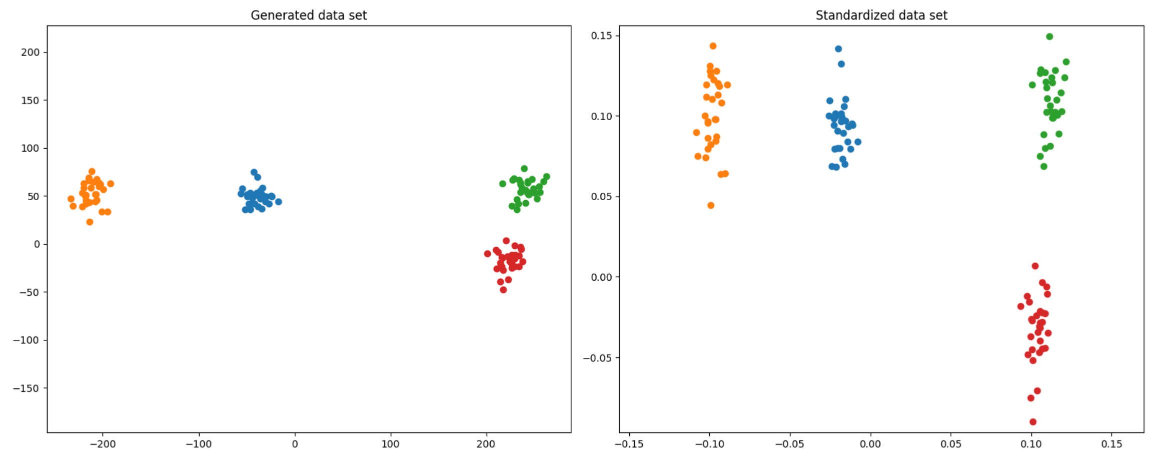


Fig. 7: A generated data set with 4 prominent, globular clusters. Unnormalized on the left, standardized on the right.

clusters on the right are initially relatively close together, but after standardization, the upper cluster(green) is as close to the cluster to its left(blue) as it is to the cluster below it(red). Clusters also changed their shape from globular into elliptical.

2.6 Data visualization

It is often useful to be able to visualize our data set, either to gauge the existence of clusters, or to quickly verify final clustering results. Most obvious way to visualize numeric data vectors is to simply plot them in Cartesian coordinates as a scatter plot. This can be easily done for data sets with 3 or less features(dimensions).

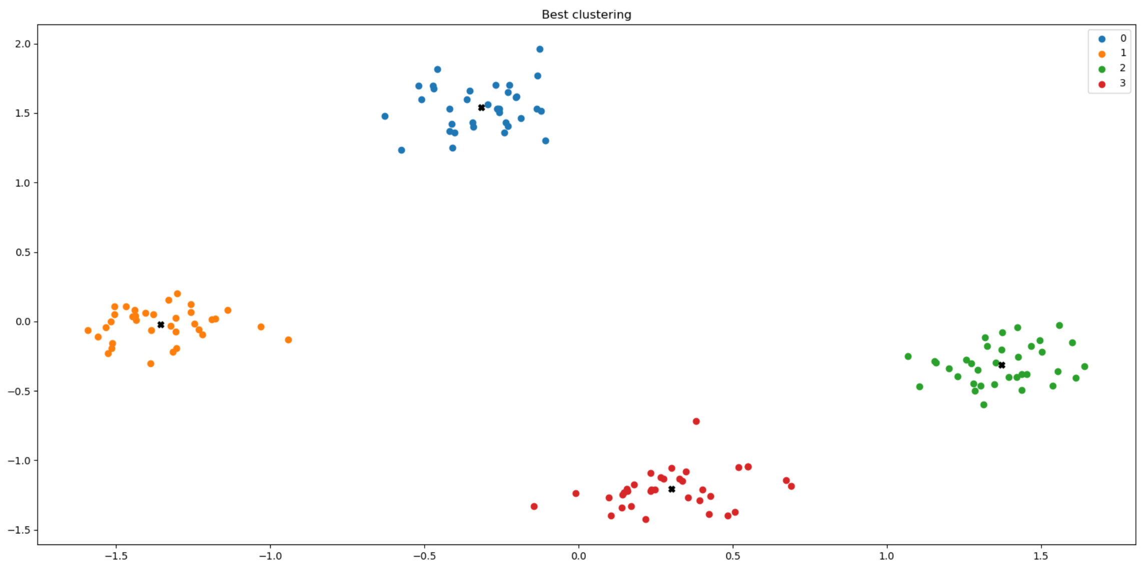


Fig. 8: Visualizing a 2-dimensional clustered data set using a scatter plot.

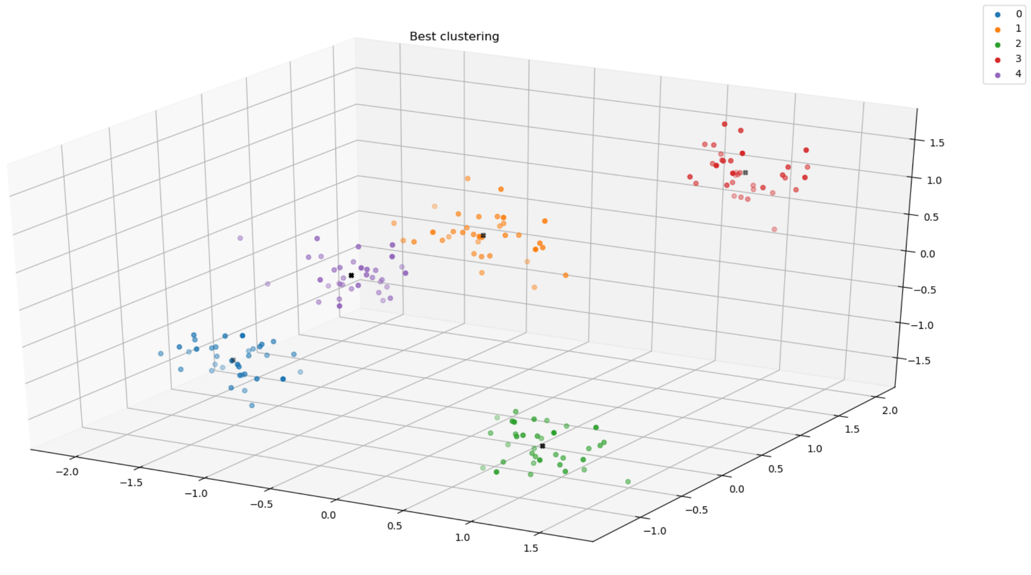


Fig. 9: Visualizing a 3-dimensional clustered data set using a scatter plot.

In a forced, static view, 3-dimensional projection can be confusing. It is often convenient to be able to move the view to get a better sense of the spacial differences.

Scatter plots can be utilized to visualize high-dimensional(> 3) data too. Of course, our previous, naive approach isn't feasible in this case. We can visualize these data sets using a scatter-plot matrix[6]. For a d-dimensional data set, this means creating a $d \times d$ matrix of 2-D scatter plots that visualize data vectors in all the possible 2-D views. An $(i - j)$th element of the scatter-plot matrix shows the data vectors in the dimensions i and j. The concept can be intuitively understood for a 3-D case; a scatter-plot matrix would visualize the top view(x-y axes) , the front view(x-z axes) and the side view(y-z axes). The plots on the diagonal $(i-i)$ of the scatter-plot matrix show the histogram(or an estimated probability density function) of data points in the i-th dimension.

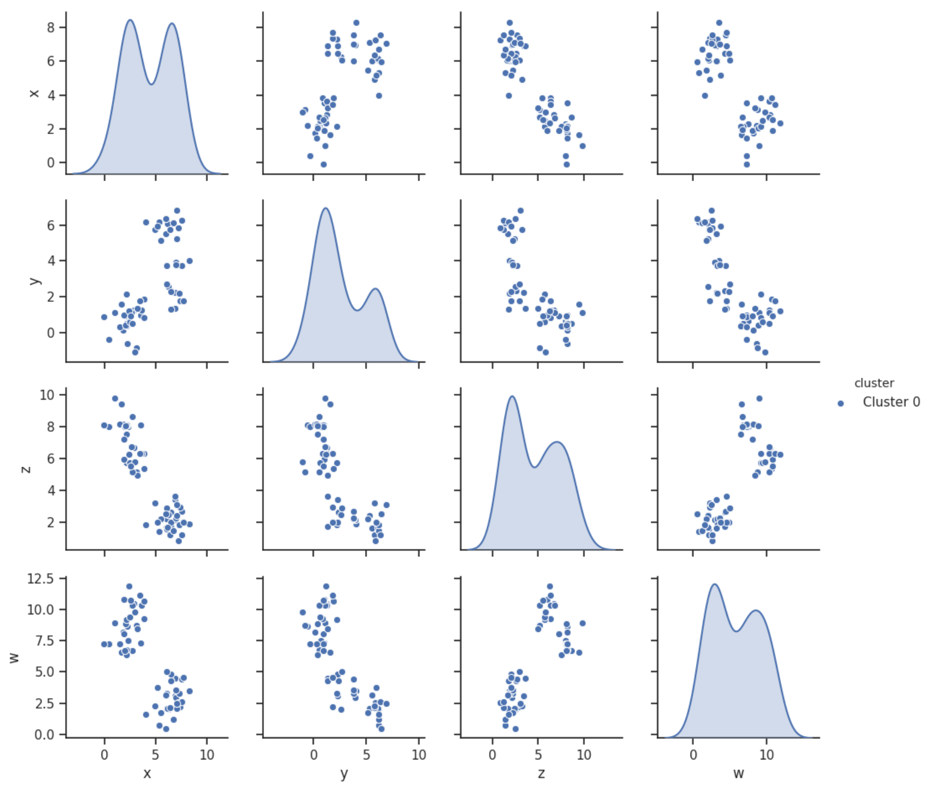


Fig. 10: Visualizing a 4-dimensional data set using a scatter plot matrix.

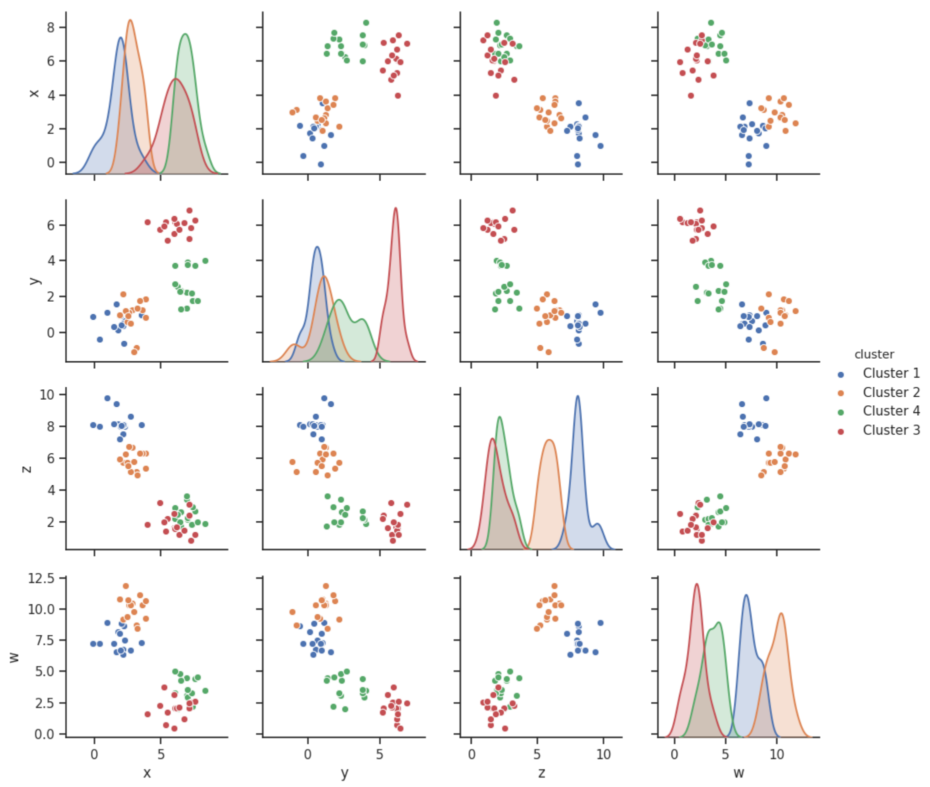


Fig. 11: Visualizing a clustered, 4-dimensional data set using a scatter plot matrix.

2.7 Dissimilarity measures

To create a partition that groups similar objects, we first need to define how we want to actually evaluate similarity. In practice, similarity is usually evaluated by its polar opposite–dissimilarity. Similarity and dissimilarity are related, and they're often generalized as proximity. A dissimilarity index for a pair of objects will be low if the object pair is similar(for two identical objects, the dissimilarity will be 0), and high if they're dissimilar. The most common dissimilarity measure for ratio and interval data is the Euclidean distance, which is a special case of the Minkowski distance, defined as[6]:

$$d(x_i, x_j) = \sqrt[h]{\sum_{f=1}^{d} |x_{if} - x_{jf}|^h} \tag{5}$$

$$h \geq 1$$

where **d** is the number of features and **h** is a real number. The Minkowski distance is also often called the L_h norm. The Euclidean distance L_2 is the Minkowski distance with $\mathbf{h} = \mathbf{2}$[6]:

$$d(x_i, x_j) = \sqrt[2]{\sum_{f=1}^{d} (x_{if} - x_{jf})^2} \tag{6}$$

Another common metric is the Manhattan distance L_1. The Manhattan distance is a simple sum of deviations in each dimension[6]:

$$d(x_i, x_j) = d(x_i, x_j) = \sum_{f=1}^{d} |x_{if} - x_{jf}| \tag{7}$$

The last special Minkowski metric is the supremum distance L_∞ ($\mathbf{h} \rightarrow \infty$), which is obtained by calculating deviations in each dimension and picking the deviation with maximum value[6]:

$$d(x_i, x_j) = \lim_{h \to \infty} \left(\sum_{f=1}^{d} |x_{if} - x_{jf}|^h) \right)^{\frac{1}{h}} = \max_f^d |x_{if} - x_{jf}| \tag{8}$$

As was explained in the subsection Data preprocessing 2.5.1, the dissimilarity index can be computed(equation 1) even if some of the values in the data object pair are missing.
We can evaluate the dissimilarity between two time series as a sum of the distances between vectors in each step of the sequence:

$$d(y_i, y_j) = \sum_{t=1}^{T} d(y_{it}, y_{jt}) \tag{9}$$

where y_i, y_j are time series and T is the number of vectors in the time series.

3 Clustering Methods

Clustering is a task of grouping objects into clusters in a way that maximizes similarity between objects in a given cluster while minimizing the similarity between them in differing clusters. In special cases, this can be done manually by visualizing the vectors in space and finding clusters by eye. Manual clustering has many downsides: An apparent limitation is that visualizing data points in a multidimensional ($d > 3$) space is not feasible, and while multidimensional data can be visualized using a scatter plot matrix, picking out clusters in them is difficult even for just 4-dimensional data. The results of manual clustering are also not objective, as different humans might see different clusters. It is also relatively time consuming and repetitive, so its usage in high frequency applications is very inconvenient. In summary, manual clustering is slow, unreliable and only possible in a very limited scope of applications. For these reasons, clustering is done by algorithms.[1]

3.1 Hierarchical clustering

Hierarchical clustering methods generate an entire hierarchical structure of the data set. The resulting structure is a sequence of nested partitions, and can be visualized with a binary tree called a dendrogram. Each level of the dendrogram is generated by agglomerating the two data objects with the highest relative similarity. A dendrogram starts with N branches(clusters), and is progressively agglomerated into a single branch(trunk).A dendrogram therefore has $N-1$ levels. We get separate clusters by cutting the dendrogram at any level, the number of cut branches is equal to the number of clusters. Methods for determining optimal number of clusters(at which level to cut) are discussed later. Hierarchical clustering works in steps[3]:

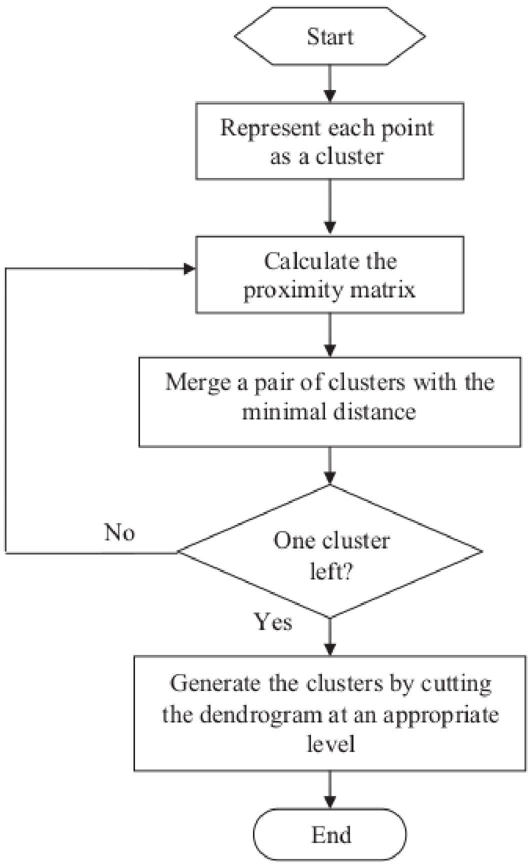
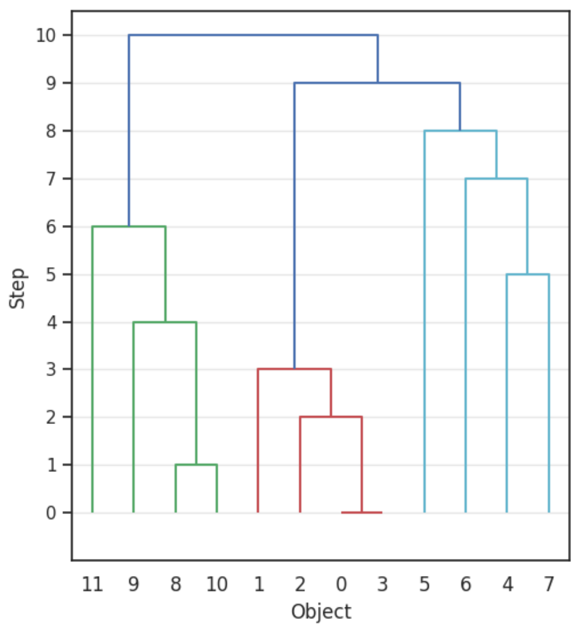

Start
Represent each point as a cluster
Calculate the proximity matrix
Merge a pair of clusters with the minimal distance
One cluster left?
No
Yes
Generate the clusters by cutting the dendrogram at an appropriate level
End

Fig. 12: Flowchart of hierarchical clustering algorithm. Taken from [3]

1. Start with N single-member clusters C_i
2. Calculate the $N \times N$ dissimilarity matrix D
3. In the dissimilarity matrix, find the minimal value $d(C_i, C_j)$, combine the two most similar clusters into one cluster.
4. Recalculate the dissimilarity matrix
5. Repeat steps 3 and 4 until only 1 cluster remains



A dendrogram generated from a hierarchical clustering of an artificial 4-d data set with 3 clusters. In the first(0-th) step, objects 0 and 3 were the most similar and were merged. In the second step, objects 8 and 10 were the most similar, and so on. Notice the objects being ordered based on their relative similarity, so the dendrogram lines don't intersect each other. In this case, the number of clusters can be seen by looking at the relatively long horizontal lines that have to be drawn to merge the clusters, as this implies that the clusters are relatively dissimilar.

Fig. 13: A dendrogram.

To get more than 1 cluster from the dendrogram, we need to cut the dendrogram at some level. If the total number of clusters K is known beforehand, we can simply cut the dendrogram so that we cut K branches. If we don't have any a priori information about the number of clusters, we can specify a maximal dissimilarity for merging clusters. The dendrogram will then be cut at the level on which this maximum distace threshold is reached.

Hierarchical methods are deterministic, since they always yield identical results on different runs of the algorithm.

Since the computational complexity of dissimilarity matrix calculations rises rapidly with the number of vectors ($\mathcal{O}(n^2)$), this basic method of hierarchical clustering isn't suitable for clustering of large data sets. It is also quite sensitive to noise and outliers. For these reasons, many more advanced hierarchical methods have been developed, such as BIRCH(Balanced Iterative Reducing and Clustering using Hierarchies) and CURE (Clustering Using REpresentatives)[3].

3.1.1 Linkage criteria

When calculating the distance between two clusters, we must first define how exactly we want to evaluate this distance. For example, assume a cluster C_i with one member x_i and a cluster C_{jk} with 2 members x_j, x_k. It is clear that there is no single way of determining the distance between these clusters. Most common ways of evaluating the distance function are[3]:

- Single-linkage(Minimum distance), where the distance between clusters is evaluated as the distance between the two closest points between the clusters. Using our example from before, the distance between clusters would be:

$$d(C_i, C_{jk}) = \min[d(x_i, x_j), d(x_i, x_k)] \tag{10}$$

  Single-linkage is appropriate if the inherent clusters have more complex shapes. However, it can give misleading results if the separation between clusters is small or if the clusters intersect each other. If one object from C_j is misclassified as belonging into C_i, a chain reaction will cause merging of both clusters into one.

- Complete-linkage(Maximum distance), where the distance is evaluated as the distance between the two furthest points between the clusters:

$$d(C_i, C_{jk}) = \max[d(x_i, x_j), d(x_i, x_k)] \tag{11}$$

- Centroid-linkage(Mean distance), where the distance is determined as the distance between cluster means(centroids).

$$d(C_i, C_{jk}) = d(x_i, \frac{x_j + x_k}{2}) \tag{12}$$

3.2 Partitional clustering

Partitional clustering algorithms seek to partition a set of objects $\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_N}\}$ into $\mathbf{K}$ clusters $\{\mathbf{C_1}, \mathbf{C_2}, ..., \mathbf{C_K}\}$ in a way that minimizes or maximizes the clustering criterion function $\mathbf{J}$. Unlike hierarchical clustering, partitional clustering returns no information about the similarity hierarchy of objects. Partitional clustering also has a constant number of clusters $\mathbf{K}$, which needs to be specified at the initialization of the clustering algorithm. At the algorithm's initialization, the centroids of these clusters are scattered across the feature space. This scattering(seeding) can be done randomly, although more advanced seeding approaches are recommended for quicker convergence and/or better reliability.[3]

3.2.1 Criterion functions

A desired clustering result is a partition of objects that maximalizes the similarity of objects within the same cluster while minimalizing the similarity of objects in differing clusters. A criterion function is a quantitative measure of the quality of the partition. [1][3]

The most common criterion is the sum-of-squared-error criterion. The squared-error is the squared euclidean distance between a vector in a cluster and the cluster's centroid. Squared-error is also known as within-cluster variation or intra-cluster variation.[1] In a particular cluster, the squared-error is calculated as:[3]

$$e_i^2 = \sum_{j=1}^{N} \gamma_{ij}^q ||x_j - m_i||^2 \tag{13}$$

The sum-of-squared-error criterion is then calculated as a sum of squared-errors for each cluster:

$$J(\Gamma, M) = \sum_{i=1}^{K}\sum_{j=1}^{N} \gamma_{ij}^q ||x_j - m_i||^2 = \sum_{i=1}^{K}\sum_{j=1}^{N} \gamma_{ij}^q (x_j - m_i)^T (x_j - m_i) \tag{14}$$

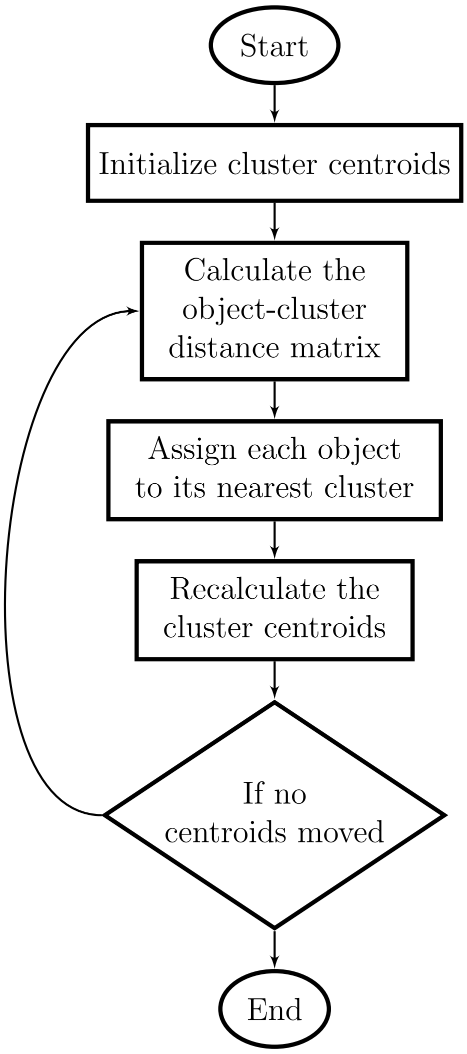
where $\mathbf{\Gamma} = \{\gamma_{\mathbf{ij}}\}$ is a partition matrix, where $\gamma_{\mathbf{ij}} = \mathbf{1}$ if the vector $\mathbf{x_j}$ belongs to the cluster $\mathbf{C_i}$ and $\gamma_{\mathbf{ij}} = \mathbf{0}$ otherwise. In fuzzy logic methods, γ_{ij} can take on any value from the interval $(0, 1)$. The parameter $\mathbf{q}$ is a partition fuzziness parameter, we get a hard partition for $q = 1$ and fuzzy partition for $q > 1$.

$\mathbf{M} = [\mathbf{m_1}, \mathbf{m_2}, ..., \mathbf{m_K}]$ is the matrix of cluster centroids, where each centroid is calculated as the mean of all objects within a cluster:

$$\mathbf{m_i} = \sum_{\mathbf{j=i}}^{\mathbf{N}} \frac{\gamma_{\mathbf{ij}}^{\mathbf{m}} \mathbf{x_j}}{\gamma_{\mathbf{ij}}^{\mathbf{m}}} \tag{15}$$

3.2.2 K-means clustering

The K-means algorithm is one of the most common in practical applications, particularly due to its simplicity and relatively low computation times. The most basic K-means method is described by the following flowchart:



Start
Initialize cluster centroids
Calculate the object-cluster distance matrix
Assign each object to its nearest cluster
Recalculate the cluster centroids
If no centroids moved
End

1. Initialize **K** cluster centroids according to the seeding method
2. Calculate an $\mathbf{N} \times \mathbf{K}$ matrix of distances **D**, which contains the distances between each data object and each cluster
3. Assign each object to its nearest cluster
4. Move the cluster centroids to the mean of all objects within the cluster
5. Repeat steps 2, 3, 4 until centroids stop moving
6. Output the clusters

Each element of the distance matrix **D** is calculated as:

$$D_{ij} = d(x_i, c_j) \tag{16}$$

An $\mathbf{N} \times \mathbf{K}$ partition matrix $\boldsymbol{\Gamma} = \{\gamma_{\mathbf{ij}}\}$ is then created. A binary membership function $\gamma_{\mathbf{ij}}$ returns 1 if the object $\mathbf{x_i}$ is closer to $\mathbf{c_j}$ than to any other centroid or 0 otherwise.

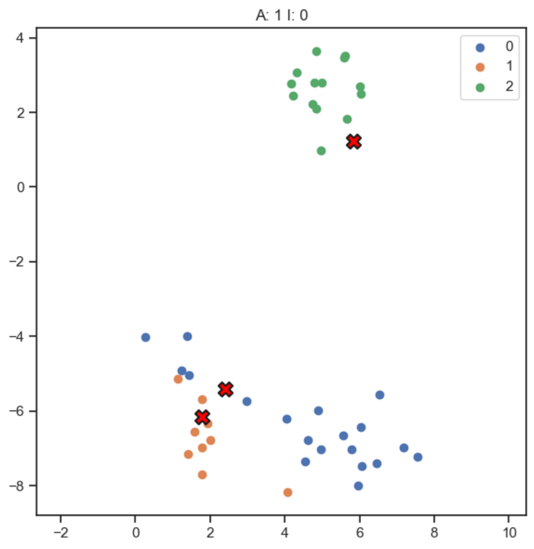


Fig. 14: K-means algorithm, first iteration.

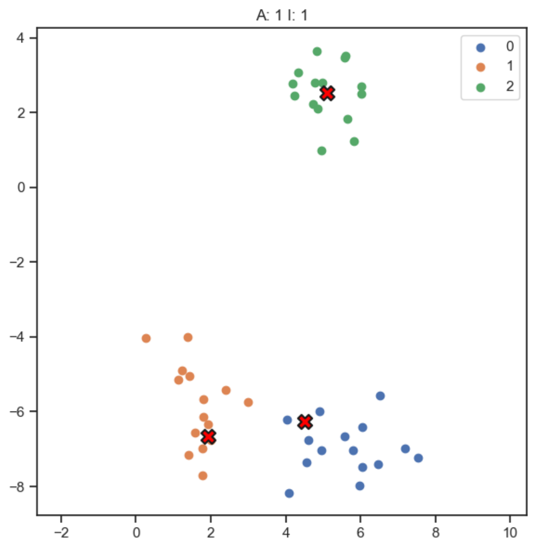


Fig. 15: K-means algorithm, second iteration.

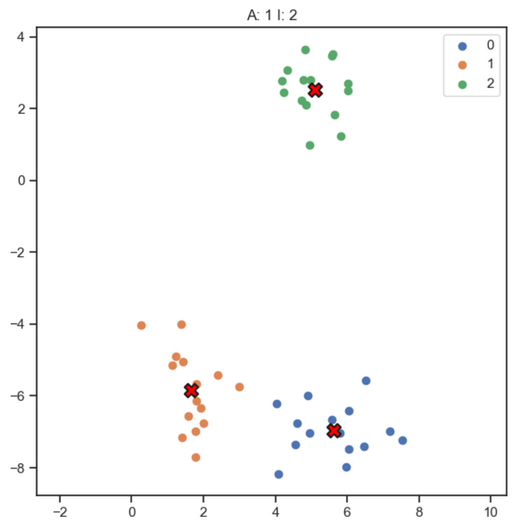


Fig. 16: K-means algorithm, third iteration.

Visualization of k-means algorithm iterations on an artificial data set with 3 clusters. Red crosses are the cluster centroids. The number of clusters **K** was specified as 3. The scatter plots show the positions of centroids at the start of each iteration. In the first iteration, 3 random objects are picked as the cluster centroids. Centroids are then moved to the mean of the resulting clusters. These steps are repeated until centroids stop moving.

Partition algorithms can converge to a local minimum of the squared-error function. Since such partition is obviously undesirable, the K-means algorithm is usually run multiple times and the attempt with the lowest sum-of-squared-error value is chosen as the best partition. K-means is a considered a stochastic method, as different runs of the algorithm can yield different results. This uncertainty is caused by different centroid initializations. For an identical centroid initialization, K-means always yields identical results.

Optimization options The calculation in step 2 is by far the most time intensive part of the algorithm. However, recalculating the distances for all objects in every iteration is unnecessary. When an object is significantly closer to one centroid than to the other centroids, a small centroid move won't affect its cluster assignation in the next iteration. Following this logic, we can significantly speed up the algorithm by limiting the algorithm to only calculate distances for the objects which are at risk of being reassigned.[14]

A very common optimization of the algorithm is the k-means++ seeding method. For each vector $\mathbf{x} \in \mathbf{X}$, let $\mathbf{D}(\mathbf{x})$ be its distance from the nearest cluster centroid. Then initialize centroids according to this algorithm[15]:

1. From an uniform distribution, choose a random object $\mathbf{x}$ as the centroid $\mathbf{c_1}$.
2. Pick another random object x as the centroid $\mathbf{c_i}$, choosing $\mathbf{x} \in \mathbf{X}$ with the probability $\frac{\mathbf{D(x)^2}}{\sum_{\mathbf{x} \in \mathbf{X}} \mathbf{D(x)^2}}$.
3. Repeat step 2 until all $\mathbf{K}$ centroids are initialized.

In step 2, the probability function gives objects that are far from the nearest cluster a higher chance of being picked as the next cluster centroid. This means the initial centroids are more likely to be well separated from each other and located in high density areas. With k-means++ initialization, the computation time requirements are usually lowered significantly. The algorithm is also more likely to converge to the global minimum, reducing the need for repeated runs of the algorithm.[15]

Determining the number of clusters K If the number of clusters **K** is unknown, we'll have to find a way to estimate the optimal number of clusters. One of the most common methods for determining **K** is to run the k-means algorithm multiple times with different number of centroids **K** at the initialization. The sum-of-squared-error criterion will generally decrease with rising **K**, since for $\mathbf{K} = \mathbf{N}$ the criterion is equal to 0. The decreases in the criterion will be significant if **K** is less than the inherent number of clusters within the data set. These sum-of-squared-error changes will however become relatively small once **K** is higher than the inherent number of clusters, as the algorithm will have to start cutting up separate clusters, which doesn't affect the errors as intensely. This observation can be utilized for estimating the inherent number of clusters, as this discontinuity will create an "elbow" in the $\mathbf{J} - \mathbf{K}$(sum-of-squared-errors as a function of number of clusters) plot.

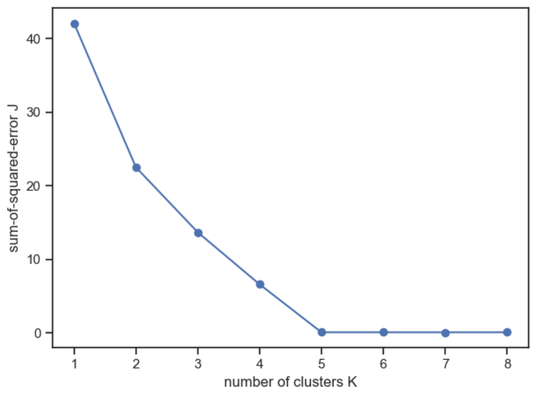


Fig. 17: Plot of the Sum-of-squared-error as a function of number of clusters.

For a data set with 5 prominent clusters, we will get a shape like this. The number of clusters **K** is then determined by the position of the elbow.
If there are no prominent, globular clusters in a data set, the criterion will steadily decrease with **K** until reaching **N** without creating an elbow. Non-globular clusters could still exist within the data set, even if no elbow forms.

3.2.3 Fuzzy C-means clustering

Fuzzy c-means clustering works similarly to k-means, the only exception is the membership function isn't binary. The membership function γ_{ij}, or the probability of object x_i belonging to a cluster c_j, is defined as[17]:

$$\gamma_{ij} = \frac{1}{\sum_{k=1}^{C} \left(\frac{d(x_i, c_j)}{d(x_i, c_k)} \right)^{2/(q-1)}} \tag{17}$$

and clusters are calculated as[17]:

$$c_j = \frac{\sum_{i=1}^{N} \gamma_{ij}^{q} x_i}{\sum_{i=1}^{N} \gamma_{ij}^{q}} \tag{18}$$

where q is a parameter that defines the fuzziness of the partitioning. The parameter q is usually $q = 2$, all applications of the algorithm in this paper will therefore also use $q = 2$.

Cluster validity In c-means, we can conveniently use the membership functions γ_{ij} to evaluate the quality of the partition. A good fuzzy partitioning would be a partitioning where every object has a high higher membership value for one cluster and low membership value for other clusters. In other words, a good partitioning is a partitioning with low fuzziness. A common way to evaluate the partitioning is therefore to sum the squares of every element in the membership matrix U and normalizing it with the number of objects N, or[17]:

$$V_{FPC} = \frac{\sum_{i}^{N} \sum_{j}^{C} \gamma_{ij}^{2}}{N} \tag{19}$$

This value is referred to as the fuzzy partitioning coefficient, or FPC. The closer the FPC is to 1 the better the partitioning.

4 Application Examples

4.1 Image segmentation

One of the more common applications of clustering is in image segmentation. The task is to separate the pixels into **C** clusters in a color space(for example the RGB color space, with 3 dimensions R,G,B). Clustering pixels in the color space basically reduces the number of colors in the image to **C**, the number of clusters. Pixels with similar colors will be assigned to identical clusters. The clusters in a color space might not be of good quality–they usually aren't well separed at all. This, however, isn't a problem in this particular application, as our goal isn't drawing conclusions from the cluster characteristics, but simplifying the image. Once the pixels are assigned clusters, the clusters can then be assigned labels by a human classifier.
For example, we can use c-means clustering with 8 clusters on a Mercator projection of the Earth. Each found cluster is then a separate world biome.

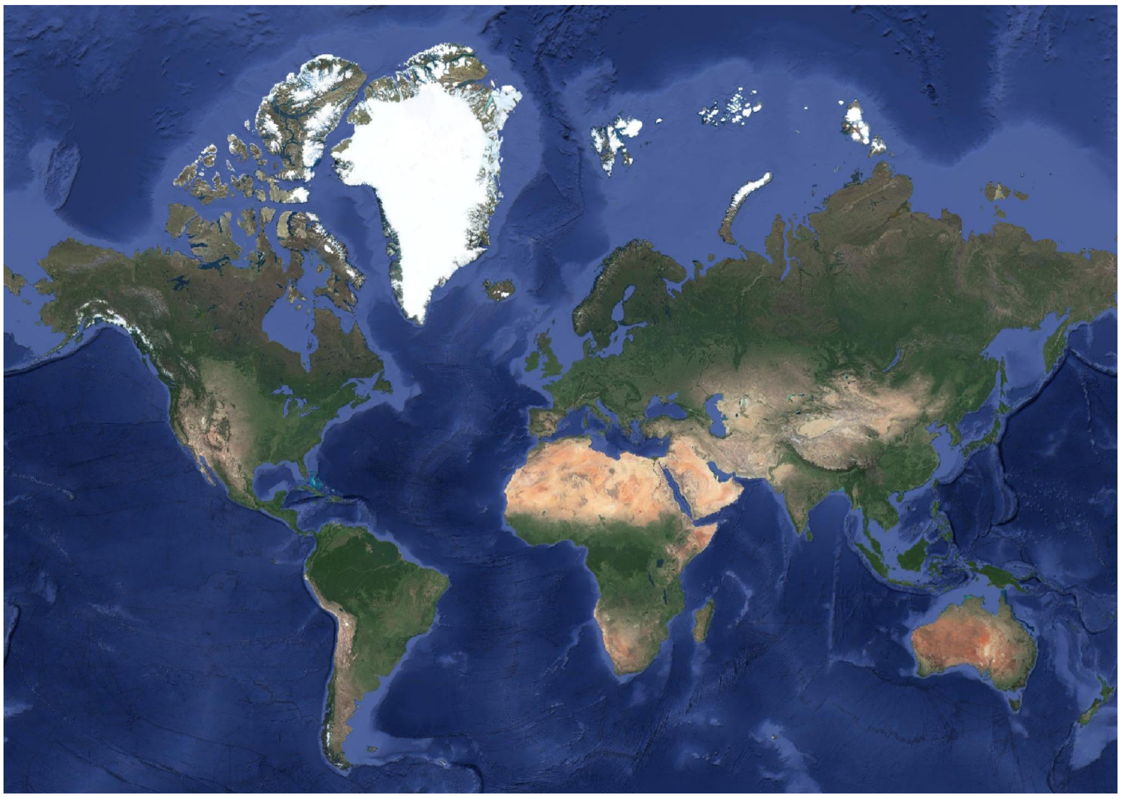


Fig. 18: Mercator projection of the Earth before processing. Taken from Google Maps.

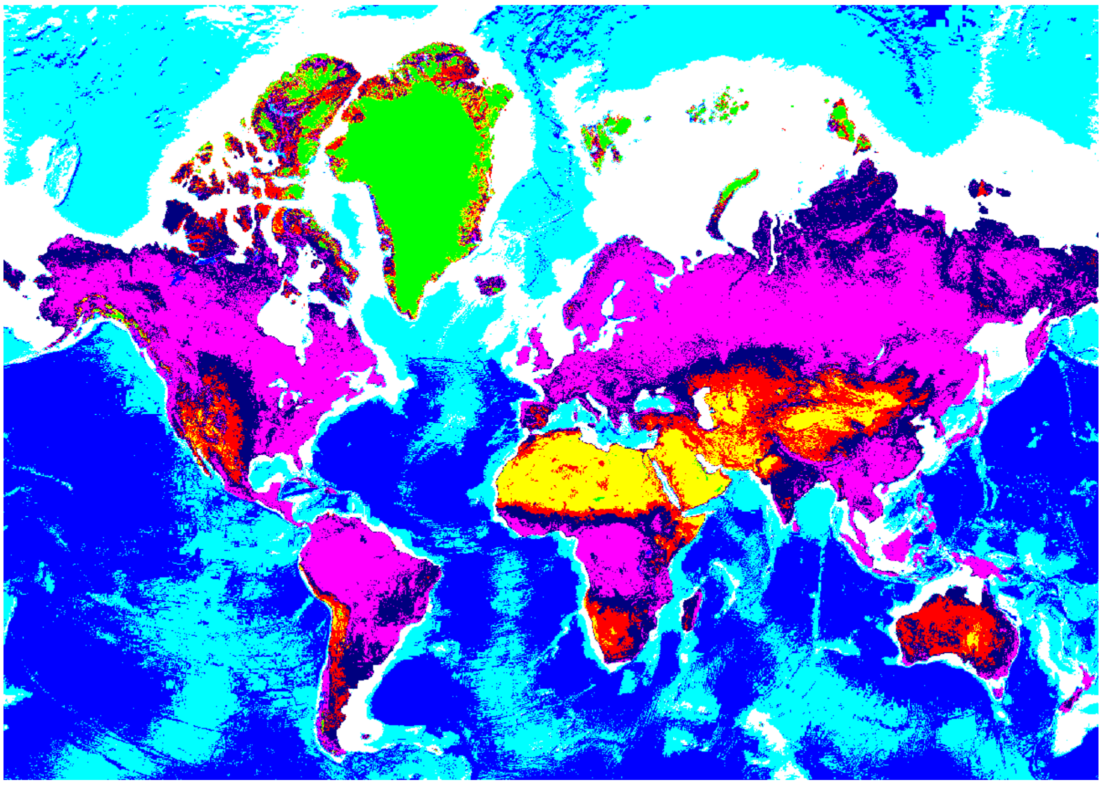


Fig. 19: Mercator projection of the Earth after fuzzy c-means clustering with 8 clusters, using random colors for centroids for better contrast.

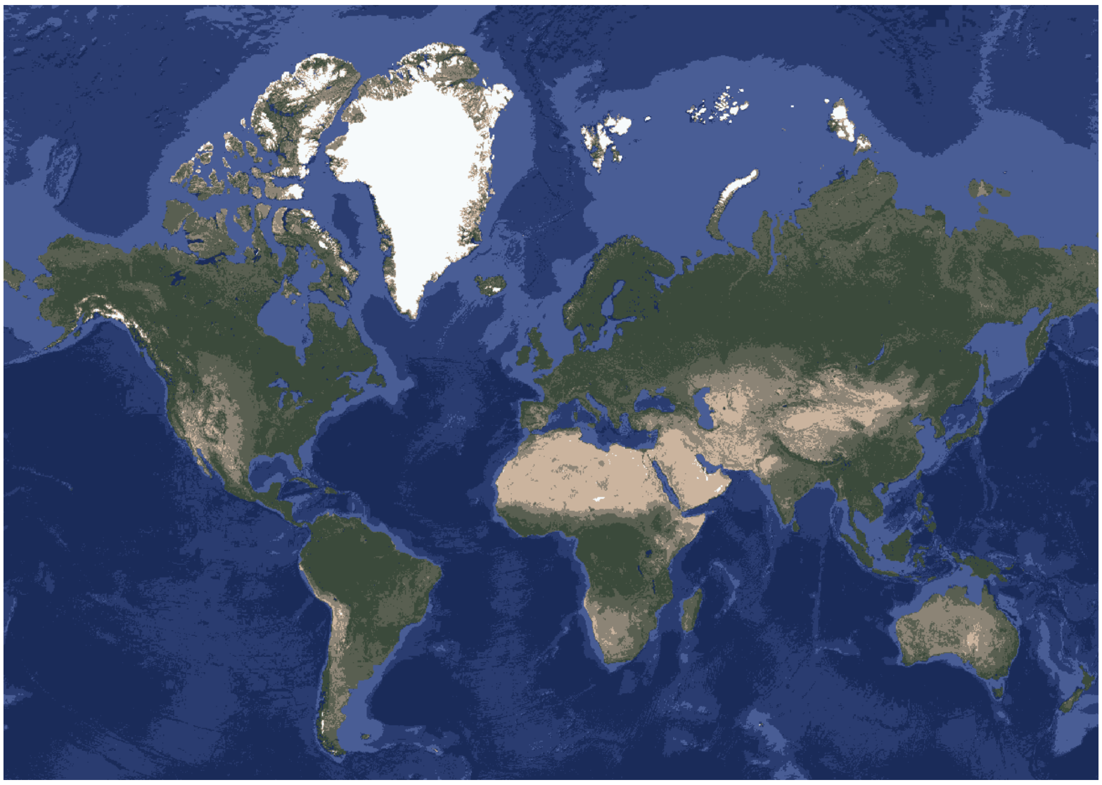


Fig. 20: Mercator projection of the Earth after fuzzy c-means clustering with 8 clusters, using the cluster centroids as colors.

The clustering algorithm doesn't utilize any information about the positions of each pixel. Simply using the pixel coordinates as features generally works quite poorly. One way to utilize the information is described in [17]. The paper describes a modification to the fuzzy c-means algorithm for image segmentation utilizing spatial information. Basically, the c-means clustering is first done is color space. Then we use a spatial function $\mathbf{h_{ij}}$ that, just as the membership function $\gamma_{\mathbf{ij}}$, defines the probability of pixel $\mathbf{x_j}$ to belong to a cluster $\mathbf{c_i}$ based on the membership values of pixels in a square window around the pixel $\mathbf{x_j}$. The spatial function $\gamma_{\mathbf{ij}}$ of a pixel $\mathbf{x_j}$ is large if many clusters in the square window belong to cluster $\mathbf{c_i}$. A new membership function is then calculated, using both the old membership function and the spatial function. This modification of the fuzzy c-means reduces the noise in the image and produces more homogeneous clusters.

4.2 Electricity demand profiles

Time series clustering can be used for finding separate time patterns in data. One such application is in [13], where time series clustering is used to find different intra-day electricity demand profiles. Two significant clusters were found in all seasons, implying that there are two different demand profiles. The intra-day electricity demand profiles were measured in 100 households. Clustering in this application separated the objects(households) into two groups. Each household was surveyed, and survey data such as number of household members, number of children, education levels of household members was collected. A prediction model was then built from the data that predicts the demand profiles of households based on more easily obtainable survey data.

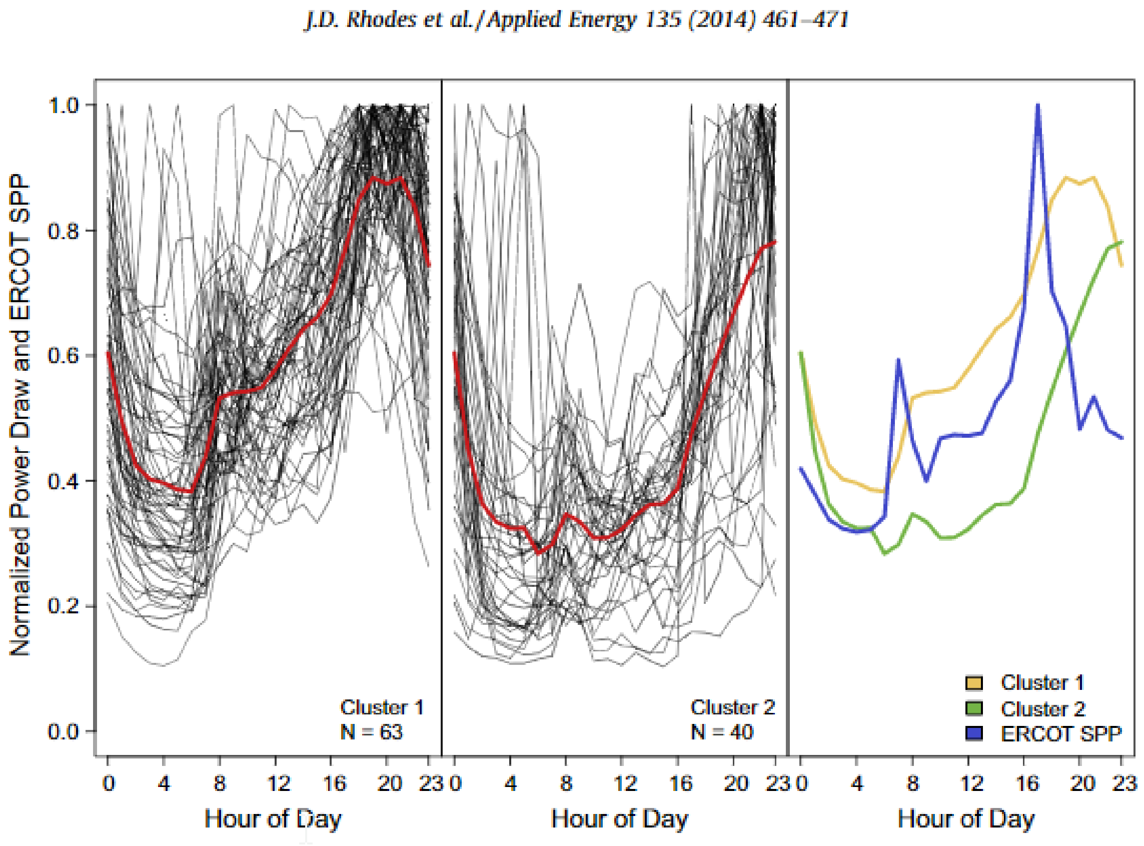


Fig. 21: Clustering of intra-day electricity demand profiles in spring. Two plots on the left show the cluster objects and the cluster centroid in red. The right plot compares the two cluster centroids. Taken from [13].

Part II

Practical Part

5 Composite tube data

Clustering can be thought of as automatic labeling of objects, so that objects that are similar have identical labels. Let's try using cluster analysis for labeling composite tube measurements, so that composite tubes with an identical composite structure are assigned identical labels. We have 3 data sets of vibration measurements: X1, X2 and X3. Each datasets contains measurements of 8 tubes. Every 2 tubes then have a different composite structure (N1, N2, P, T). Our task is to partition each data set into 4 clusters, identifying the 4 different structures. Measurement has been done on 8 tubes in total, and each subtype is represented by 2 tubes. We have apriori information about the number of clusters(4–total number of composite structures) and the number of tubes in each cluster(2).

The tubes were being excited by a mechanical exciter TIRA S51144-M. The excitation frequency started at $0.125Hz$ and kept slowly increasing up to $3200Hz$. At each $0.125Hz$ step, a measurement was taken–a total number of 25600 measurements $\mathbf{N}$. The measurements were taken with a vibrometer Polytec PSV 400 D4063. Each data set is an $\mathbf{N} \times \mathbf{d} \times \mathbf{t}$ matrix, where $\mathbf{N}$ is the total number of measurements for different excitation frequencies and $\mathbf{t}$ is the number of tubes(8). The dimensionality of the measurements $\mathbf{d}$ is 3, one for each spacial axis.[18]

6 Analyzing the data set by frequency slices

6.1 Clustering in single frequency slices

We can think of the data set as a list of N different data matrices. The data set shape is basically $frequency \times measurement \times tube$. If we specify a frequency(or rather its index), we can get a slice of the data set with shape $measurement \times tube$. This gives us a simple data matrix, with x, y, z values for each tube for the specified excitation frequency. My first clustering attempts were on these simple slices. Unfortunately, clustering didn't consistently yield 4 equally sized clusters as desired.

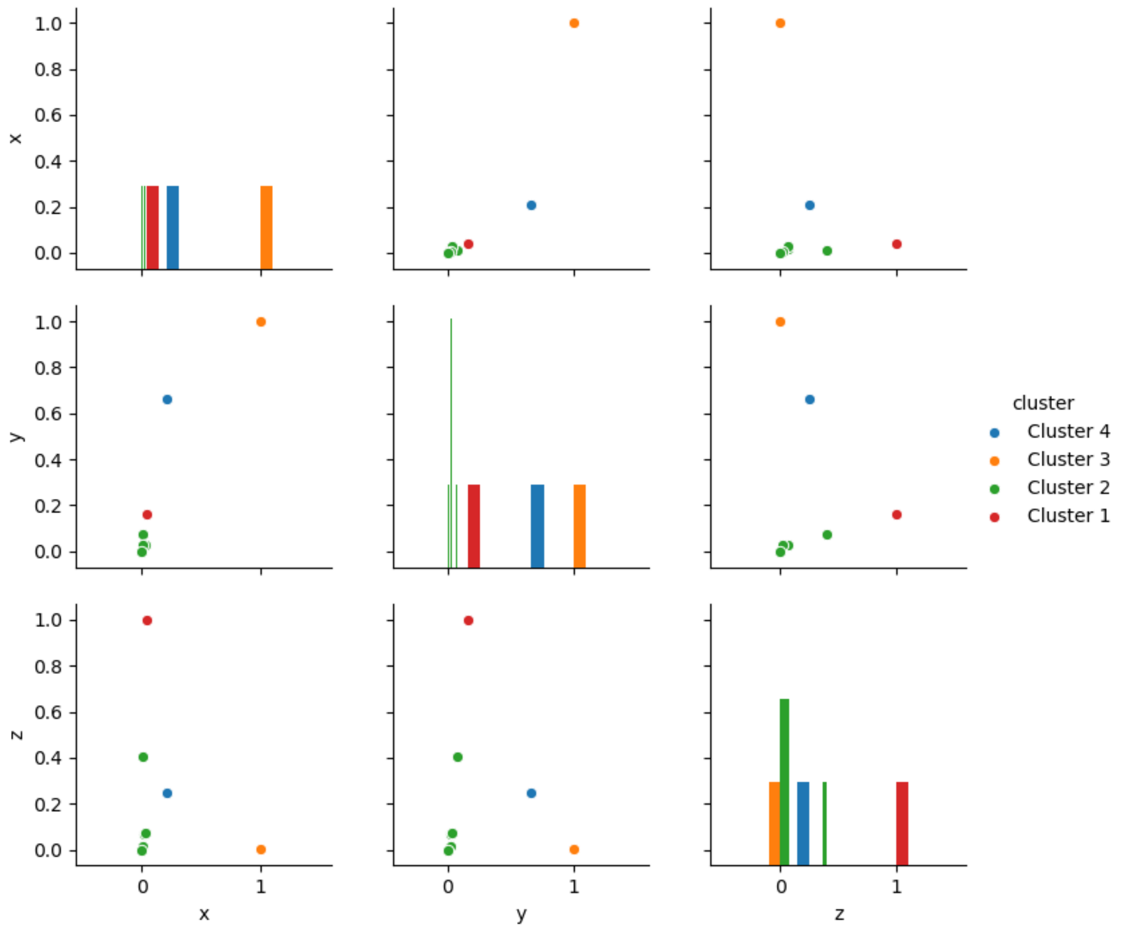


Fig. 22: Attempt at clustering composites with x-y-z features for excitation frequency at index 4001(500Hz). Preprocessed data set X1.

The clustering yielded results, but they didn't match the apriori information. 5 out of 8 tubes were assigned to one cluster, and their features were around the 0 point. There clearly are no two similar objects, as all the points that are separated from the main clusters are by themselves. The FPC measure was pretty high at 0.95, particularly due to the fact that single object clusters automatically have an inter-cluster FPC of 1, and all the objects in the big clusters are basically identical and close to their cluster's centroid, so their membership to the cluster was very high as well. The results were similar across most frequencies. A clustering with 4 equally sized clusters was found in less than 2% of all frequencies.

6.2 Clustering consolidated frequency slices

Another attempt was to consolidate a number of frequency slices into one to get more objects and decrease the influence of possible outliers on the clustering results. Resultant clusters generally looked like this:

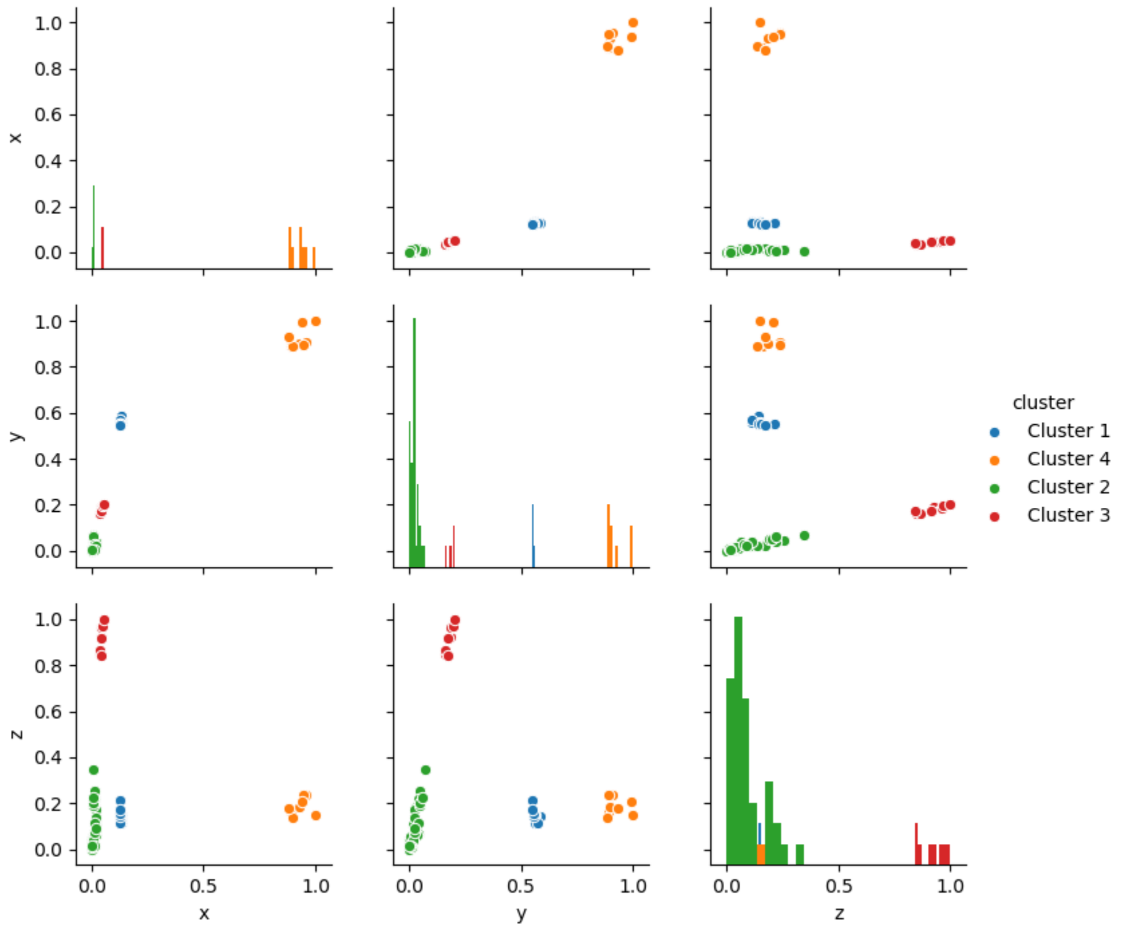


Fig. 23: Attempt at clustering composites with x-y-z features for excitation frequencies at indexes [4001-4008](501 Hz). Features were rescaled to a $[0, 1]$ range in each single frequency slice. Data set X1.

These consolidates slices actually contained 4 larger clusters as desired. However, the clusters weren't of equal size, as 40 out of 64 objects were again assigned to one cluster. The other clusters had 8 objects each, equal to the number of consolidated frequency slices. It later became apparent to me that the similarity of objects in these clusters was due to the fact that the measurements for each frequency were taken shortly after each other and the clusters were just measurements for a single tube. This also explains why each cluster had 8 objects, as that was the number of consolidated frequency slices. While these attempts didn't yield the desired results, they uncovered the sequential nature of the data and pointed me towards time series analysis(in frequency domain). I also realized that using the full x, y, z measurements is misleading, since the specific tube oscillations are quite chaotic. Two identical tubes might oscillate in different planes at identical excitation frequencies due to slightly different conditions, but their total amplitudes remain identical.

7 Time series preprocessing

7.1 Loading the data

Before we can start preprocessing the data, we need to load the data set. The data sets are saved as .npy files. The code snippet for importing required modules, creating directories for saving results and loading the data set:

```
import BClustering as c #custom module for clustering
import BTools as bt #custom module with other tools
import numpy as np
import matplotlib.pyplot as plt
from sklearn import preprocessing
import tkinter as tk
from tkinter import filedialog
import os
import pandas as pd
#Function for creating directories
def createDir(path, dir):
    if not os.path.exists(path + dir):
        os.mkdir(path + dir)
        print("Directory ", path + dir, " created ")
    else:
        print("Directory ", path + dir, " already exists")
    return path + dir + "/"

#initializing the tk interpreter
#simple GUI for choosing file paths
root = tk.Tk()
root.withdraw()
```

```
#Loading the data set as a numpy array
filepath = filedialog.askopenfilename() #path to the .npy file
array = np.load(filepath)

#Results will be saved to this directory
filedir = filedialog.askdirectory()
filedir += '/'

#stopping the tk interpreter
root.destroy()

#Creating directories in the filedir path
datadir = createDir(filedir, "fuzzy_c_means")
rawSeries_dir = createDir(datadir, "0Raw_series")
cutAveragedSeries_dir = createDir(datadir, "1Cut_averaged_series")
preprocessedSeries_dir = createDir(datadir, "2Preprocessed_series")
```

7.2 Condensing features

Since using the x, y, z data proved misleading, it needs to be somehow condensed. I decided to do this by simply transforming it into a single feature–the Euclidean norm of the 3-dimensional x, y, z vector. I recalculated the measurements for each tube in every frequency slice accordingly:

$$s = \sqrt{x^2 + y^2 + z^2} \tag{20}$$

Using this condensed feature, we now have exactly one value for each tube in each frequency slice. This is quite convenient, as the entire data set is now of the shape $N \times 8$, where 8 is the number of tubes. When we slice this matrix by specifying a tube, we get a single time series with N values. The code snippet for condensing the features is shown in the next subsection.

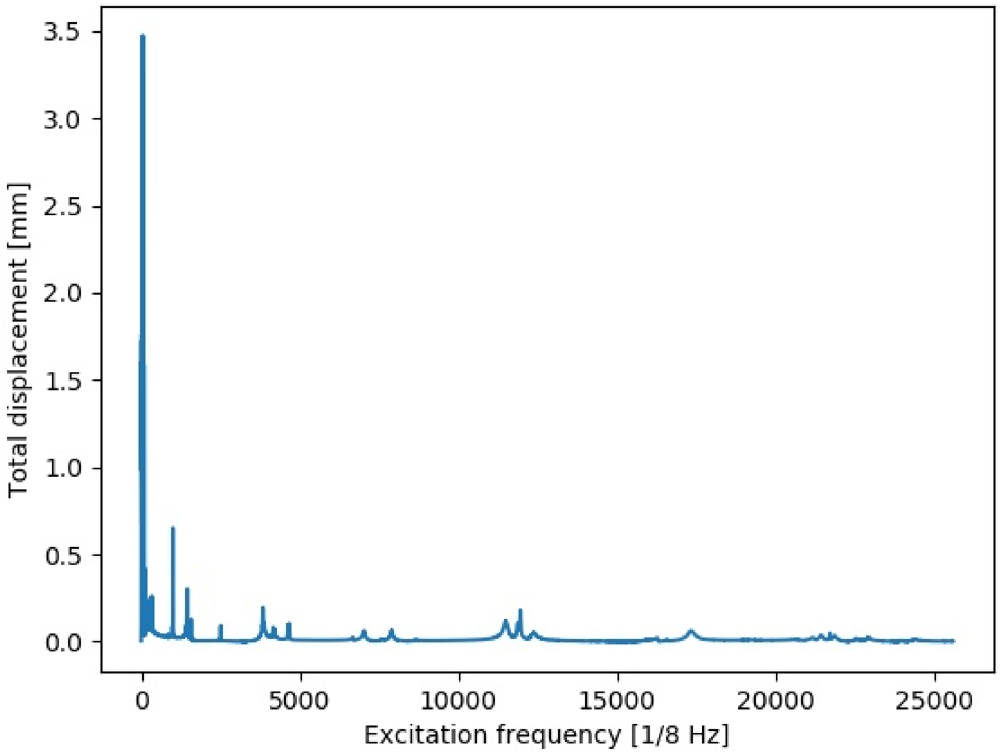


Fig. 24: Tube with index 0 in the X1 dataset. Raw time series data.

7.3 Cutting off inconvenient data

Values on the x-axis are actually the frequency indexes, although they can be interpreted as units of $1/8Hz$. We can see a large spike at the start of the measurement. Judging by the size of the spike in relation to the rest of the measurements, it seems artificial; meaning it is a reaction to an external stimulus. The spike is there for every tube in the data set, it doesn't really bring us any information about the differences between the tube composite structures. For these reasons, I decided to cut the first 1500 values from the time series to get rid of the spike, as it completely dwarfs the values at the later values and it could hinder our clustering results.
The code snippet for condensing features and cutting off the spike:

```
Xarr_tube = [] #main data matrix, expected shape 'tubes' x 'N'
cutoff = 1500
tubes = np.shape(array)[2] #number of tubes
dim = np.shape(array)[1] #dimensionality
N = np.shape(array)[0] - cutoff #number of measurements
#Create the series of total amplitudes for each tube
for tube in range(tubes):
    Xarr = [] #time series for 'tube'
    #Calculation of the total amplitude for each frequency
    #Iterating starts at the "cutoff" index
    for index, f in enumerate(array[cutoff:]):
        x_coord = f[0][tube]
        y_coord = f[1][tube]
        z_coord = f[2][tube]
        # Calculate the absolute distance
        distance = np.sqrt(pow(x_coord, 2) +
                           pow(y_coord, 2) +
                           pow(z_coord, 2))
        Xarr.append(distance)
    #Add the series of amplitudes for
    #'tube' to the data matrix
    Xarr_tube.append(Xarr)
```

7.4 Smoothing the time series

My next step in preprocessing the data was smoothing the series using a weighted moving average(WMA) with a period of 500. Weights are set up so that more recent values have bigger impact on the average. Moving averages are practically low-pass filters, so they get rid of the noise in the data. The code snippet performing this function:

```
period = 500
Xarr_tube = bt.weighted_moving_average(Xarr_tube, period)
```

The WMA function from my module takes in the entire data matrix as an input and returns a data matrix with averaged rows:

```
def weighted_mean(Values):
    sum = 0
    length = len(Values)
    for i, value in enumerate(Values):
        sum += pow((i/length), 2) * value #weight (i/length)^2
    mean = sum / length
    return mean
```

```
def weighted_moving_average(X, period):
    Xarr_tube_MA = [] #List with the averaged time series
    tubes = len(X) #number of objects(tubes)
   N = len(X)[0] #number of values in each series
    for tube in range(tubes):
        xarr_avg = [] #List for the 'tube' time series
        for i in np.arange(period, N - period, 1):
            #The averaging window
            period_window = X[tube][i - period:i]
            #calculate weighted mean of the avg window
            mean_ = weighted_mean(period_window)
            xarr_avg.append(mean_)
        Xarr_tube_MA.append(xarr_avg)
        print("Tube␣" + str(tube) + "␣data␣smoothed.")
    return Xarr_tube_MA
```

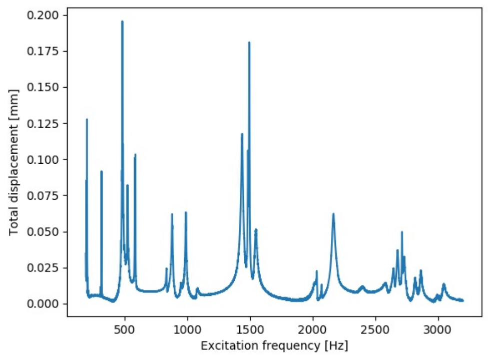


Fig. 25: Tube with index 0 in the X1 dataset. Time series after cutting off first 1500 values.

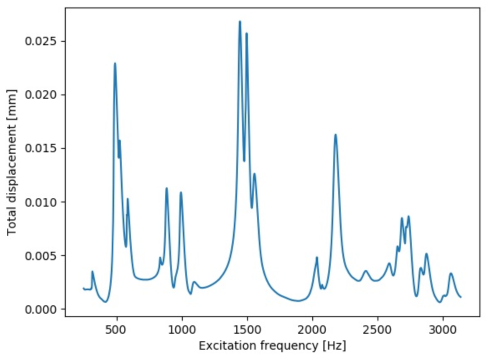


Fig. 26: Tube with index 0 in the X1 dataset. Time series after cutting off first 1500 values and smoothing it with a WMA500.

Since moving average cannot calculate values for the first 500(=period) values, the smoothed time series is 500 values shorter. The 500th value is a weighted average of the first 500 values, so they are still somewhat represented in the series.

7.5 Rescaling the time series

The last step is rescaling the data. I used min-max method, which scales the time series into a $[0, 1]$ interval, with the lowest value in the time series being at 0 and the highest value at 1.

```
#Rescale each time series
for i, Xarr in enumerate(Xarr_tube_MA):
    #Rescaling the time series
    Xarr_rescaled = preprocessing.minmax_scale(Xarr)
    #Rewriting the time series
    Xarr_tube[i] = Xarr_rescaled
```

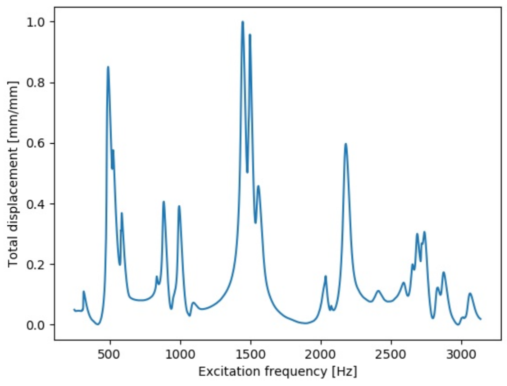


Fig. 27: Tube with index 0 in the X1 dataset. Fully preprocessed time series.

8 Time series clustering

8.1 Partitional clustering

Since partitional clustering algorithms require a list of vectors as their input. A time series with **N** values can be, rather counter-intuitively, considered as a vector with N dimensions. After a little thought, this however makes sense, as dissimilarity calculations calculate the deviation in each dimension. The Manhattan distance between two time series vectors is therefore just a sum of distances between the time series for all indexes. The usual Euclidean distance doesn't make much sense for this application.

8.1.1 K-means clustering

For k-means clustering I used the Python package sklearn.cluster. The code snippet performing the K-means clustering and plotting the results:

```
#Clustering
km = cluster.k_means(Xarr_tube, n_clusters=4)
clusters = km[1] #cluster assignations
centroids = km[0] #centroid coordinates

# Convert the time series into my custom vector classes
# Each time series has
X_tube = []
for tube in range(len(Xarr_tube)):
    x = vector(clusters[tube], tube, *Xarr_tube[tube])
    X_tube.append(x)

#Create figure
fig1 = plt.figure(figsize=(12, 8))
```

```
#Create subplots, add axis labels and subplot titles
#Each subplot is a separate cluster
ax = [fig1.add_subplot(2, 2, i + 1) for i in range(4)]
[ax[i].set_title("Cluster " + str(i)) for i in range(4)]
[ax[i].set_xlabel("Excitation frequency [Hz]") for i in range(4)]
[ax[i].set_ylabel("Total displacement [-]") for i in range(4)]
#Plot the time series for each tube into the according subplot
for i, X in enumerate(X_tube):
    cluster = X.cluster
    #x axis indexes
    x_data = range(len(X))
    #shift x axis by (cutoff + period)
    x_data = [x + (cutoff + period) for x in x_data]
    #divide the x axis by 8 to convert units to Hz
    x_data = [x / 8 for x in x_data]
    y_data = X
    ax[cluster].plot(x_data, y_data)

#Create handles for the plot legends
handles = [[], [], [], []]
for x in X_tube:
    handles[x.cluster].append("Tube " + str(x.counter))
for handle in handles:
    handle.append("Centroid")
```

```
#Plot the time series for each cluster centroid
for cluster, centroid in enumerate(centroids):
    x_data = range(len(centroid))
    x_data = [x + (cutoff + period) for x in x_data]
    x_data = [x / 8 for x in x_data]
    y_data = centroid
    ax[cluster].plot(x_data, y_data, '-.k')

#add the legends to each subplot
for i, axis in enumerate(ax):
    axis.legend(handles[i])

plt.tight_layout()
plt.savefig(datadir + "/Results_kMeans" + ".jpg")
plt.close()
```

```
#Calculate distances between centroids and objects
distance_from_cluster = np.empty([tubes, K])
for i, x in enumerate(X_tube):
    for i2, centroid in enumerate(centroids):
        #Compute the distance between 'i'th object and 'i2'th cluster
        #subtract centroid coordinates from the object coordinatess
        tmp = np.add(x.coordinates, np.multiply(-1, centroid))
        #convert to absolute distances
        tmp = np.abs(tmp)
        #sum the deviations to get total distance
        tmp = np.sum(tmp)
        distance_from_cluster[i][i2] = tmp

#Create a DataFrame from the distance matrix
df = pd.DataFrame(distance_from_cluster)
#Rename the columns
column_names = ['Cluster␣' + str(c) for c in range(K)]
df.columns = column_names
#Rename the rows
row_names = ['Object␣' +str(t) for t in range(tubes)]
df['Objects'] = row_names
df.set_index('Objects', inplace=True)
df.to_excel(datadir + 'distances.xlsx')

#Save the final centroids and preprocessed series
np.save(datadir + "clusterAssignations", clusters)
np.save(datadir + "centroids", centroids)
np.save(datadir + "preprocessedData", Xarr_tube)
```

Clustering the preprocessed X1 data set with the K-means method with number of clusters $K = 4$ yielded the following results:

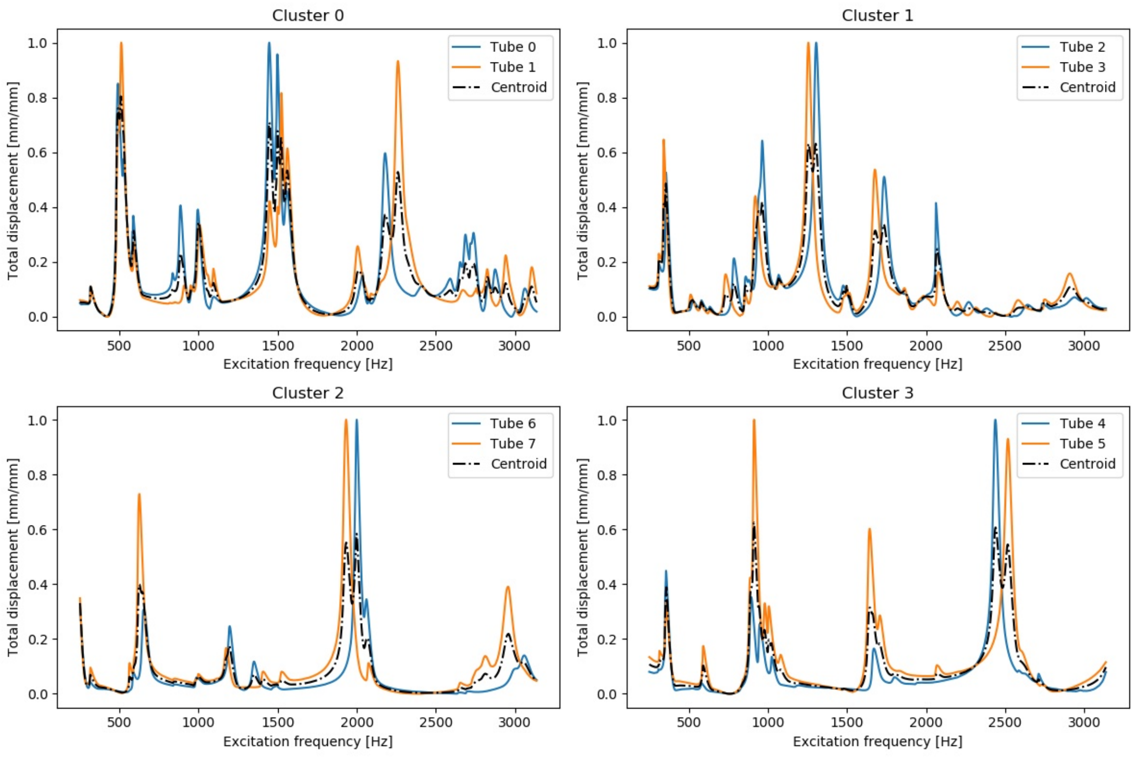
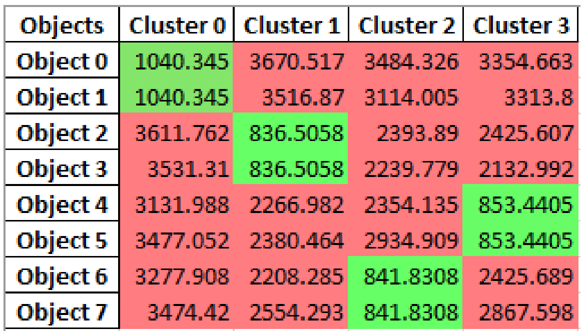


Fig. 28: K-means clustering of the X1 dataset.



Objects	Cluster 0	Cluster 1	Cluster 2	Cluster 3
Object 0	1040.345	3670.517	3484.326	3354.663
Object 1	1040.345	3516.87	3114.005	3313.8
Object 2	3611.762	836.5058	2393.89	2425.607
Object 3	3531.31	836.5058	2239.779	2132.992
Object 4	3131.988	2266.982	2354.135	853.4405
Object 5	3477.052	2380.464	2934.909	853.4405
Object 6	3277.908	2208.285	841.8308	2425.689
Object 7	3474.42	2554.293	841.8308	2867.598

Fig. 29: Distance matrix with distances between tube time series and cluster centroids for the X1 dataset.

8.1.2 C-means clustering

The code for C-means clustering is similar, although there's no clustering function in sklearn, so I had to use the package *skfuzzy*. Since the skfuzzy c-means algoritm accepts the data matrix in a different shape and also doesn't return defuzzified cluster assignations, I made my own function for reshaping the data set, running the algorithm, defuzzyfying the results and outputting them:

```
def c_means_clustering(Xarr, number_of_clusters, show=False):
    #making sure the Xarr is a numpy array and
    #transposing it for the clustering function
    Xarr = np.array(Xarr)
    Xarr = np.transpose(Xarr)

    #clustering, metric set as cityblock aka. Manhattan
    cm = fuzzy.cmeans(Xarr, number_of_clusters, 2,
                      error=0.0005, maxiter=1000,
                      metric="cityblock")
    #membership values matrix
    u = cm[1]
    #FPC index
    fpc = cm[6]

    #transposing the membership values matrix and the
    #original data matrix back into the standard shape
    u = np.transpose(u)
    Xarr = np.transpose(Xarr)
```

```
#Transforming the array into a list of vectors
#and assigning clusters based on membership values
X = [] #creating an empty list for vectors
N = len(Xarr) #number of objects
for i in range(N):
    #assigning a cluster based on the index of the max
    #membership value
    cluster = np.argmax(u[i])
    #Create the vector and append it to X
    x = vector(cluster, i, *Xarr[i])
    X.append(x)

#Create a list of clusters from my cluster class
Cluster = []
for c in range(number_of_clusters):
    #add an empty cluster to Clusters
    Cluster.append(clusterclass(c, []))
    #search for vectors assigned to the cluster
    #and add them to the cluster
    for x in X:
        if x.cluster == c:
            Cluster[c].add_vector(x)
    #Calculate the variance and centroid of the cluster
    Cluster[c].recalculate()

print("Clustering complete!\n")
print("FPC: " + str(fpc))
return X, Cluster, fpc, u
```

The script using this function for clustering, visualizing and saving the results is then very similar to the previously shown K-means script. Due to issues with clustering of the X2 data set with the c-means algorithm, I chose a different scaling method; *robust_scale* was used instead of *minmax_scale*. This scaling method scales the data series according to its interquartile range rather than its whole range. Clustering the preprocessed data set X2 into a C-means algorithm yields these results:

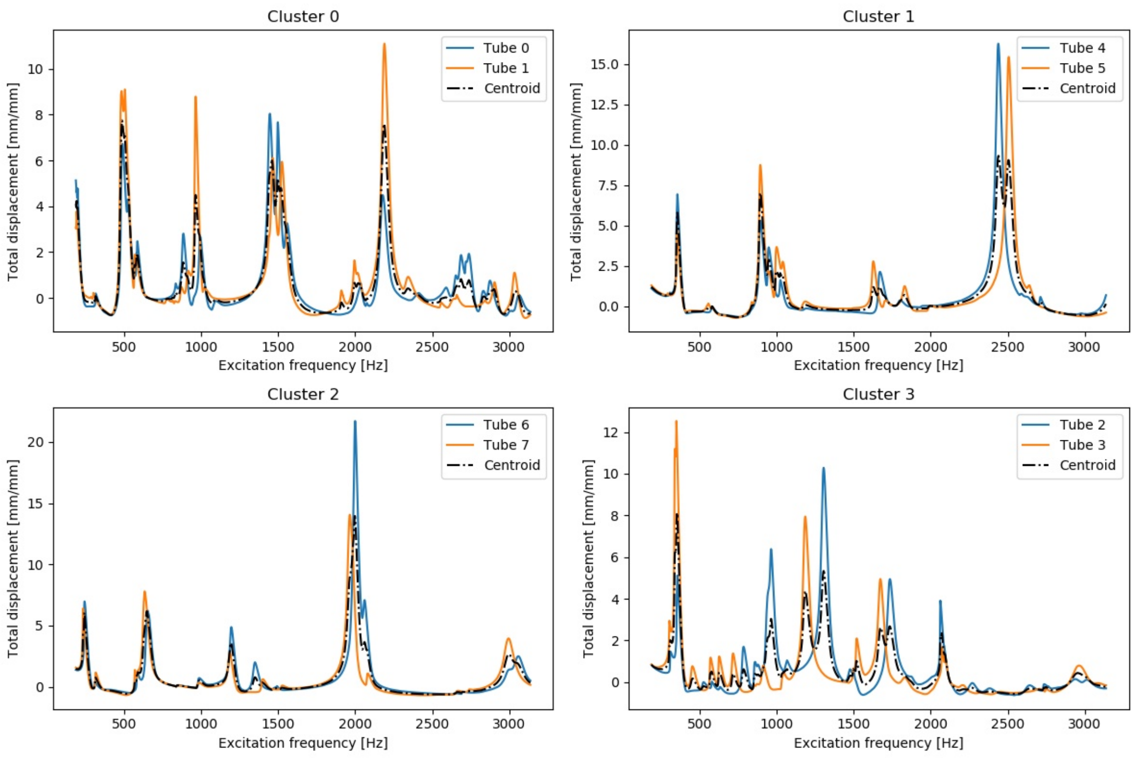
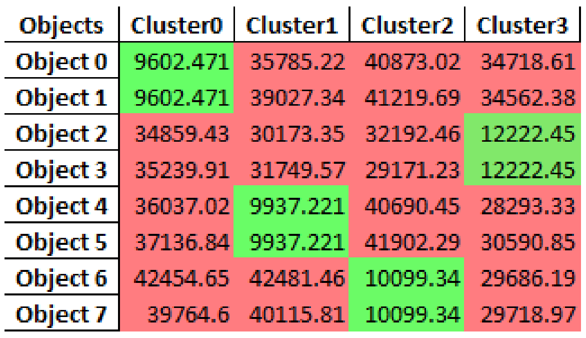


Fig. 30: C-means clustering of the X2 dataset.



Objects	Cluster0	Cluster1	Cluster2	Cluster3
Object 0	9602.471	35785.22	40873.02	34718.61
Object 1	9602.471	39027.34	41219.69	34562.38
Object 2	34859.43	30173.35	32192.46	12222.45
Object 3	35239.91	31749.57	29171.23	12222.45
Object 4	36037.02	9937.221	40690.45	28293.33
Object 5	37136.84	9937.221	41902.29	30590.85
Object 6	42454.65	42481.46	10099.34	29686.19
Object 7	39764.6	40115.81	10099.34	29718.97

Fig. 31: Distance matrix with distances between tube time series and cluster centroids for the X2 dataset.

8.1.3 Discussion

The clustering yielded believable results, each cluster has 2 objects, which fits our prior information about the number of tubes representing each composite structure. The time series in each cluster also appear similar visually. The resonant frequencies are often slightly shifted for each tube in the same cluster. This can be explained by the tubes being slightly different in some ways, for example slightly defective or damaged. From the final centroids, we can estimate the resonant frequencies for each composite tube. If the centroids were more established–meaning we had more data and more tubes in each cluster, we could even assess the level of damage for each tube, by calculating its dissimilarity from the cluster centroid(assuming the centroid is similar to undamaged tubes, which would be true if most of the measured tubes were undamaged.)

If we had more solid centroids(i.e more measured tubes), we could determine the phase shift between each time series and its cluster's centroid. This can be done by calculating the distance from its cluster's centroid multiple times, while shifting the time series in the frequency axis for every calculation. The phase shift from the centroid would then be estimated as the phase shift at which the similarity was maximal. The function *correlate* from the *scipy.signal* module performs this task. A script implementing this function for finding phase shifts between the time series in each cluster is included in the attachments of this paper.

8.2 Hierarchical clustering

Hierarchical clustering requires a dissimilarity matrix as its input, so before starting the clustering itself, the $N \times N$ dissimilarity matrix needs to be calculated. I chose the Manhattan distance as the dissimilarity measure. The code snippet performing this function:

```
#Dissimilarity matrix calculation
dissimilarity_matrix = []
dim = len(Xarr_tube[0]) #dimensionality
N = len(Xarr_tube) #number of tubes
#Calculating each row of the matrix
for row in range(N):
    dissimilarity_matrix_row = []
    #Calculate the dissimilarity between
    # the "row"th and the "col"th tube
    for col in range(N):
        sum = 0 #the sum of deviations in every dimension
        for d in range(dim):
            #add the deviation in "d"th dimension
            sum += np.abs(Xarr_tube[row][d] - Xarr_tube[col][d])
        #add the element to the "row"th row
        dissimilarity_matrix_row.append(sum)
    #add the "row"th row to the matrix
    dissimilarity_matrix.append(dissimilarity_matrix_row)

#Save the dissimilarity matrix to an excel file
df = pd.DataFrame(dissimilarity_matrix,
                        index=range(8), columns=range(8))
df.to_excel(datadir + "/dissimilarity_matrix.xlsx")
```

With the dissimilarity matrix calculated, we can move on to clustering itself. I used the function AgglomerativeClustering from the sklearn package. The code snippet for clustering and plotting the dendrogram:

```
#Function to plot a dendrogram
#taken from:
#https://github.com/scikit-learn/scikit-learn/blob/
# 70cf4a676caa2d2dad2e3f6e4478d64bcb0506f7/examples/
# cluster/plot_hierarchical_clustering_dendrogram.py
def plot_dendrogram(model, **kwargs):
    # Children of hierarchical clustering
    children = model.children_

    # Distances between each pair of children
    # Since we don't have this information,
    # we can use a uniform one for plotting
    distance = np.arange(children.shape[0])

    # The number of observations contained in each cluster level
    no_of_observations = np.arange(2, children.shape[0] + 2)

    # Create linkage matrix and then plot the dendrogram
    linkage_matrix = np.column_stack(
                         [children,
                         distance,
                         no_of_observations]).astype(float)

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, color_threshold=4,
                    show_contracted=True, **kwargs)
```

```
#Perform the clustering chosing complete linkage
hc = AgglomerativeClustering(affinity='precomputed',
                                                n_clusters=4,
                    linkage='complete').fit(dissimilarity_matrix)


#Plot and save the dendrogram
plot_dendrogram(hc)
plt.ylim(−1) #set the yaxis lower limit to −1
plt.savefig(datadir + "/Dendrogram" + ".jpg")
plt.close()
```

Hierarchical clustering of the preprocessed X3 dataset yields these results:

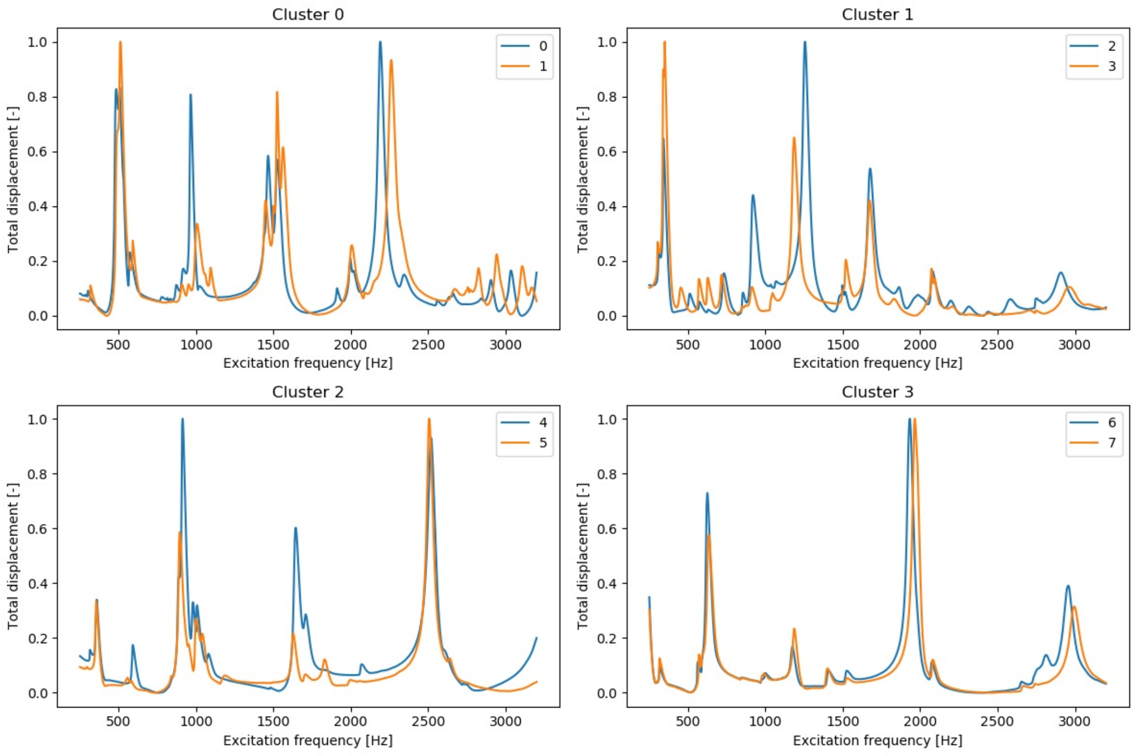


Fig. 32: Cluster assignations for each tube from the X3 dataset from a hierarchical clustering algorithm with complete linkage.

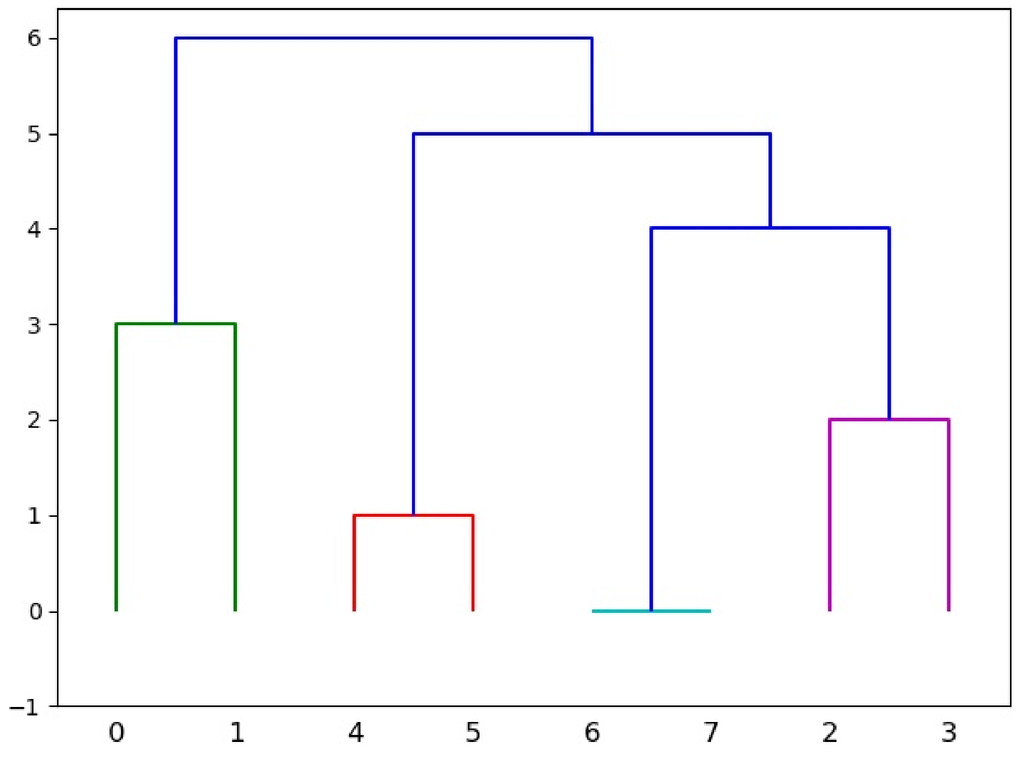


Fig. 33: Dendrogram generated from the hierarchical clustering of the X3 data set.

Discussion As there is no notion of a centroid for hierarchical clusters, there's no 'average' time series for each cluster. We can however still determine resonant frequencies of each composite structure tube from the time series of any tube within the cluster. The generated dendrogram shows the similarities between the clusters(structures). The two most similar clusters are cluster 3 (tubes 6,7) and the cluster 1 (tubes 2,3). When these two clusters are merged and compared to the other clusters, they're the most similar to the cluster 2 (tubes 4,5). The cluster 0 (tubes 0,1) is the most unique, as it has been merged in the last level of the dendrogram.

9 Conclusion

Using appropriate Python modules(sklearn, skfuzzy, scikit), I wrote scripts for transforming composite tube vibration measurement data into frequency characteristics, preprocessing them and clustering them with different clustering methods. Three dimensional measurements in each frequency were first condensed into a single feature. This transformed the data set into a set of frequency characteristics, which could be easily compared. Frequency characteristics were smoothed with a weighted moving average with a period of 500, which also made the dissimilarity between the frequency characteristics more pronounced. Smoothed frequency characteristics were rescaled to a predetermined range. These processed frequency characteristics were afterwards partitioned into 4 clusters, with the goal of sorting tubes into 4 classes based on the structure of the composite material. I had apriori information about the number of tubes representing each composite structure in the data set, and the clustering results were in accordance with the expectations–each cluster contained 2 tubes. The script managed to yield these correct results for each of the 3 datasets, confirming that clustering is a viable method for identifying composite structures.

I also tried clustering for image segmentation. Using the Python PIL module, I loaded the image, transformed it into an RGB pixel array and clustered the pixels based on their RGB values. This essentially segments the image based on pixel colors. When a Mercator projection map of the Earth was loaded and segmented into clusters, the clusters represented different Earth biomes. I tried using clustering for identifying roads in satellite images, but the attempts were mostly unsuccessful. The clustering only partitioned pixels based on their colors, so modifying the clustering algorithm to utilize spatial information(such as in [17]) might produce better results. The unsuccessful attempts are included in the attachments to this paper.

The attachments also contain a script with my own k-means implementation, however it wasn't used in the practical part as the algorithm is poorly optimized compared to the functions from specialized modules such as sklearn. It was mostly used for constructing figures for the theoretical part.

References

[1] A. K. Jain and R. C. Dubes, Algorithms for clustering data. Englewood Cliffs, NJ: Prentice Hall, 1988.

[2] J. Leskovec, A. Rajaraman, and J. D. Ullman, Mining of Massive Datasets. Cambridge University Press, 2014.

[3] R. Xu and D. Wunsch, Clustering. John Wiley & Sons, 2008.

[4] A. K. Jain and M. H. C. Law, "Data Clustering: A User's Dilemma," in Pattern Recognition and Machine Intelligence, vol. 3776, S. K. Pal, S. Bandyopadhyay, and S. Biswas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1–10.

[5] J. A. Hartigan, Clustering algorithms. New York: John Wiley & Sons, 1996.

[6] J. Han, J. Pei, and M. Kamber, Data Mining: Concepts and Techniques. Elsevier, 2011.

[7] "'Deduction' vs. 'Induction' vs. 'Abduction.'" [Online]. Available: https://www.merriam-webster.com/words-at-play/deduction-vs-induction-vs-abduction. [Accessed: 28-Jun-2019].

[8] P. Berkhin, "A Survey of Clustering Data Mining Techniques," in Grouping Multidimensional Data: Recent Advances in Clustering, J. Kogan, C. Nicholas, and M. Teboulle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 25–71.

[9] G. Kerr, H. J. Ruskin, M. Crane, and P. Doolan, "Techniques for clustering gene expression data," Computers in Biology and Medicine, vol. 38, no. 3, pp. 283–293, Mar. 2008.

[10] R. O. Duda, P. E. Hart, and D. G. Stork, Pattern Classification (2Nd Edition). New York, NY, USA: Wiley-Interscience, 2000.

[11] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, “From Data Mining to Knowledge Discovery in Databases,” p. 18.

[12] L. Yu and H. Liu, “Efficient Feature Selection via Analysis of Relevance and Redundancy,” Journal of Machine Learning Research, vol. 5, pp. 1205–1224, 2004.

[13] J. D. Rhodes, W. J. Cole, C. R. Upshaw, T. F. Edgar, and M. E. Webber, “Clustering analysis of residential electricity demand profiles,” Applied Energy, vol. 135, pp. 461–471, Dec. 2014.

[14] C. M. Poteraş, C. Mihăescu, and M. Mocanu, “An Optimized Version of the K-Means Clustering Algorithm,” presented at the 2014 Federated Conference on Computer Science and Information Systems, 2014, pp. 695–699.

[15] D. Arthur and S. Vassilvitskii, “k-means++: The Advantages of Careful Seeding,” p. 11.

[16] J. C. Bezdek, R. Ehrlich, and W. Full, “FCM: The fuzzy c-means clustering algorithm,” Computers & Geosciences, vol. 10, no. 2–3, pp. 191–203, Jan. 1984.

[17] K.-S. Chuang, H.-L. Tzeng, S. Chen, J. Wu, and T.-J. Chen, “Fuzzy c-means clustering with spatial information for image segmentation,” Computerized Medical Imaging and Graphics, vol. 30, no. 1, pp. 9–15, Jan. 2006.

[18] Jan Němec. ”Detection of composite destruction using laser vibrometer” Praha, 2017. Bachelor thesis. Czech Technical University in Prague. Faculty of Mechanical Engineering.

List of Figures

List of used software

- Texmaker, MiKTeX (LaTeX)
- PyCharm
- Zotero
- SAS Planet

List of attachments

1. 'Clustering' – PyCharm project

    - /venv/ArtificialClustering - a folder with a script for creating artificial clusters, clustering the artificial data and visualizing results. The script was mostly used for testing my own k-means algorithm and for creating figures for this paper.
    - /venv/composites - a folder with 5 scripts for analyzing the composite tube time series data. There is one script for clustering by frequency slices, 3 scripts for preprocessing and clustering(k-means, c-means and hierarchical), and 1 script for calculating the phase shifts between time series and their centroids.
    - /venv/imgProcessing - contains a script for clustering RGB images. The script loads an image, resizes it to a specified width, transforms the image into a pixel array and performs pixel clustering. Results are then saved as .png files into a specified directory. The script saves the same result twice with different color coding, one where each cluster's color is its cluster centroid and one where the cluster color is chosen from a list with contrasting colors.
    - /venv/BClustering.py – a custom module with functions for clustering. Contains a function for c-means clustering which uses a function from skfuzzy to perform the clustering and my own function for k-means clustering. The k-means function worked well for artificial data and image processing, however it was quite slow for large images as it is not optimized. The module also contains a function for creating the J-K plot for finding the number of clusters(elbow method).
    - /venv/BTools.py - a custom module with other useful tools. Contains functions for the custom k-means algorithm, generating artificial clustered data sets and a moving average function for smoothing time series data.

- /venv/customClasses.py - a module with custom vector and cluster classes.

2. 'CompositesResults' – a folder with clustering results for each data set. Contains 3 folders X1, X2 and X3. Each folder then contains 3 folders with results of k-means, c-means and hierarchical clustering for given data set.
3. 'Data' – a folder with composite tube data sets.
4. 'Other' – a folder with test results.
    - 'ArtificialData' – A folder with results of the custom k-means algorithm for artificial data with different dimensionality and cluster count
    - 'ImageProcessing' – A folder with results of image processing. Contains results of processing satellite images with different pre-specified number of clusters.
    - 'MovingAverages' – A folder with k-means clustering results for the X1 dataset preprocessed with different moving average periods or types(Simple MA or Weighted MA).