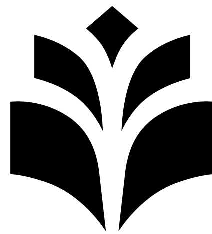
# Split $k$-means

Olli Rantinoja

Master's thesis



UNIVERSITY OF
EASTERN FINLAND

School of Computing
Computer Science

November 2023

UNIVERSITY OF EASTERN FINLAND, Faculty of Science, Forestry and Technology
Joensuu
School of Computing
Computer Science

Rantinoja, Olli: Split $k$-means
Master's thesis, 58 p.
Supervisor: Professor Pasi Fränti
4th November, 2023

**Abstract:** Clustering is a form of unsupervised machine learning that seeks to find groups similar to each other in the targeted dataset. $k$-means is a well-known clustering algorithm that searches iteratively for $k$ groups in a dataset by selecting $k$ centroids and moving them to the correct places. However, $k$-means has weaknesses, such as the fact that it is a greedy algorithm. This means that it finds a locally optimal result rather than a global optimum. This work introduces the split $k$-means algorithm, which aims to solve the above problem by dividing the generated clusters until $k$ clusters are found. The algorithm can be implemented in three different variations. The three variations of the algorithm presented in the work are compared with each other and performance is tested against the random swap algorithm. The conclusion is that it is possible to enhance the $k$-means algorithm by splitting clusters iteratively using a heuristic method.

**Keywords:** Clustering algorithms; Unsupervised learning; Machine learning

**ACM CCS (2012)**
• **Computing methodologies → Cluster analysis**.

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden, metsätieteiden ja tekniikan tiedekunta
Joensuu
Tietojenkäsittelytieteen laitos
Tietojenkäsittelytiede

Rantinoja, Olli: Split $k$-means
Pro gradu -tutkielma, 58 s.
Ohjaaja: Professori Pasi Fränti
4. marraskuuta 2023

**Tiivistelmä:** Ryhmittely eli klusterointi on ohjaamattoman koneoppimisen muoto, jossa pyritään löytämään kohteena olevasta datajoukosta keskenään samankaltaisia ryhmiä. $k$-means on tunnettu ryhmittelyalgoritmi, joka etsii iteratiivisesti $k$ ryhmää datajoukosta valiten $k$ sentroidia ja siirtämällä ne oikeisiin paikkoihin. $k$-meansilla on kuitenkin heikkouksia, kuten se, että se on ahne algoritmi. Tämä tarkoittaa sitä, että se löytää paikallisesti optimaalisen tuloksen globaalin optimin sijaan. Tässä työssä esitellään split $k$-means -algoritmi, joka pyrkii ratkaisemaan edellämainitun ongelman jakamalla syntyneitä klustereita, kunnes $k$ klusteria on löytynyt. Algoritmi voidaan toteuttaa kolmena eri variaationa. Työssä esitellyn algoritmin kolmea variaatiota vertaillaan keskenään, ja niiden suorituskykyä verrataan random swap -algoritmiin. Johtopäätöksenä on, että $k$-means-algoritmia voidaan parantaa jakamalla klustereita iteratiivisesti käyttäen heuristista menetelmää.

**Avainsanat:** Klusterointialgoritmit; Ohjaamaton oppiminen; koneoppiminen

**ACM CCS (2012)**
• **Computing methodologies** → **Cluster analysis**;

# Acronyms

***k*-NN** *k*-nearest neighbours

**BF** Brute force

**CB** code-vector-based

**CI** Centroid index

**DBA** Davies-Bouldin index

**GLA** generalized Lloyd algorithm

**MSE** Mean Squared Error

**PB** partitioning-based

**PCA** principal-component analysis

**SSE** Sum of Squared Errors

# Contents

**1 Introduction** **1**

**2 *k*-means** **4**

2.1 Initialization methods . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2 Objective function . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Comparing results with the centroid index . . . . . . . . . . . . . . 13
2.4 Random swap . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.5 How many clusters? . . . . . . . . . . . . . . . . . . . . . . . . . . 16

**3 Divisive clustering** **19**

3.1 Iterative splitting algorithm . . . . . . . . . . . . . . . . . . . . . . 19
3.2 Bisecting *k*-means . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**4 Split *k*-means** **25**

4.1 How to split? . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.2 Which cluster to split next? . . . . . . . . . . . . . . . . . . . . . . 28
4.3 Fine-tuning the result . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.4 Split *k*-means variants . . . . . . . . . . . . . . . . . . . . . . . . . 34

**5 Experiments** **39**

5.1 Implementations . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.2 Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.3 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
5.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

**6 Conclusions** **55**

**References** **57**

# 1. Introduction

Clustering is an unsupervised machine learning form where the goal is to discover the natural groups in data (Theodoridis, 2008). In clustering a set of objects are classified in groups where more or less similar objects are in the same group and each group contains at least one data object. The formed groups reveal the differences and similarities between the objects and can be used to make conclusions about the data.

The concept of clustering has been referred to with different names in different contexts. Theodoridis (2008) states that in pattern recognition it is often referred to as *learning without a teacher* and *unsupervised learning*, while in graph theory it is referred to as *partition*. On the other hand, in biology and ecology it is often referred to as *numerical taxonomy* and in social sciences as *topology*. Despite the different names, clustering can be used for many purposes, pretty much regardless of the field of science.

Theodoridis (2008) presents the actual clustering problem with a taxonomy example from biology. Consider the following animals:

**Mammals** sheep, cat, dog

**Birds** seagull, sparrow

**Reptiles** lizard, viper

**Fish** blue shark, red mullet, gold fish

**Amphibians** frog

When clustering the animals in the list above, the actual clustering result depends on the criterion of a cluster. Figure 1.1 is an example by Theodoridis (2008) and it demonstrates the significant differences in clustering results when the clustering criterion changes. Even though the frog belongs to the class of amphibians in biological taxonomy, the cluster where the frog belongs to varies depending on the context since the clustering criterion is different.

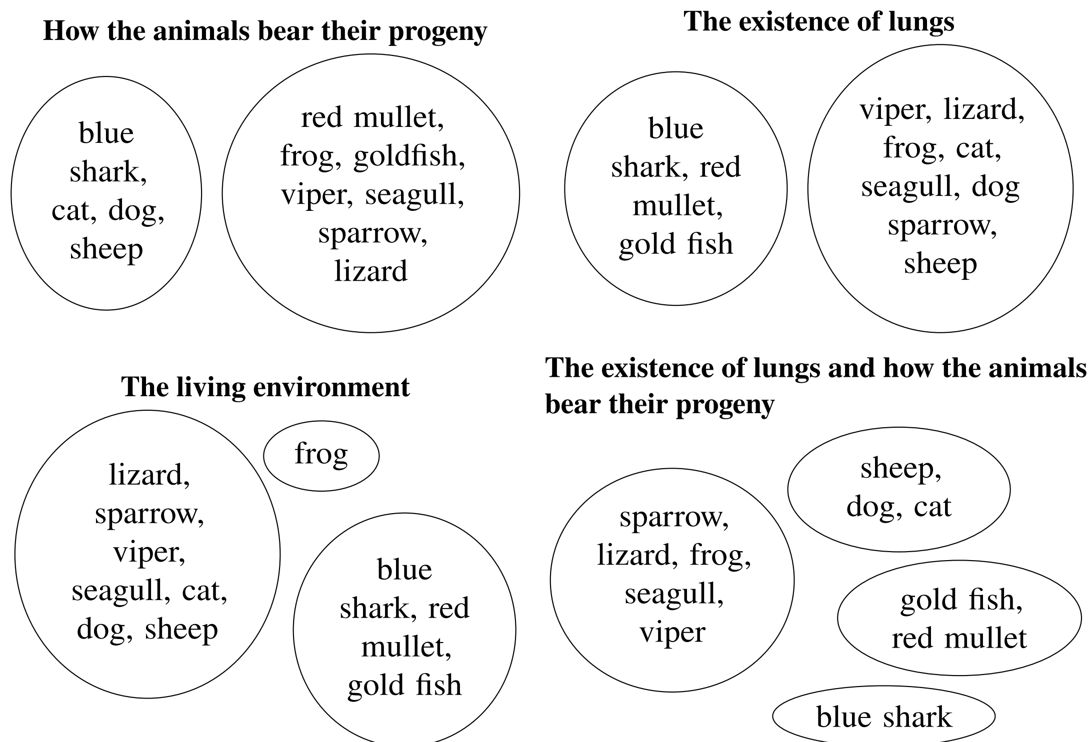



**Figure 1.1:** Resulting clusters based on different clustering criterion (Theodoridis, 2008)

As seen in the example in Figure 1.1, there is no exact definition of clustering. The exact definition of clustering depends on the definition of a single cluster (Theodoridis, 2008). However, a universally accepted definition for a *cluster* does not exist and most of the proposed definitions have been based on loosely defined terms, or they have oriented on specific kinds of clusters (Theodoridis, 2008).

There are two fundamental ways to approach the clustering problem: the *agglomerative* (top-down) approach defines clustering as separating more homogeneous groups from heterogeneous population and the *divisive* (bottom-up) approach defines clustering as finding groups in data by some natural criterion of similarity (Estivill-Castro, 2002).

Besides answering the question of where the clusters are located in the data, the number of clusters has to be determined as well. According to Kärkkäinen and Fränti (2002a), if the number of clusters is known, the clustering is called *static*. In static clustering, determining the number of clusters is not a part of the clustering algorithm and the clustering algorithm only solves the locations of clusters. In contrast, if the number of clusters is not known, the clustering is called *dynamic* and solving the number of clusters is part of the algorithm.

Fahad et al. (2014) proposed a classification framework from the standpoint of the algorithm designer. The framework shown in Figure 1.2 is a tool for understanding the

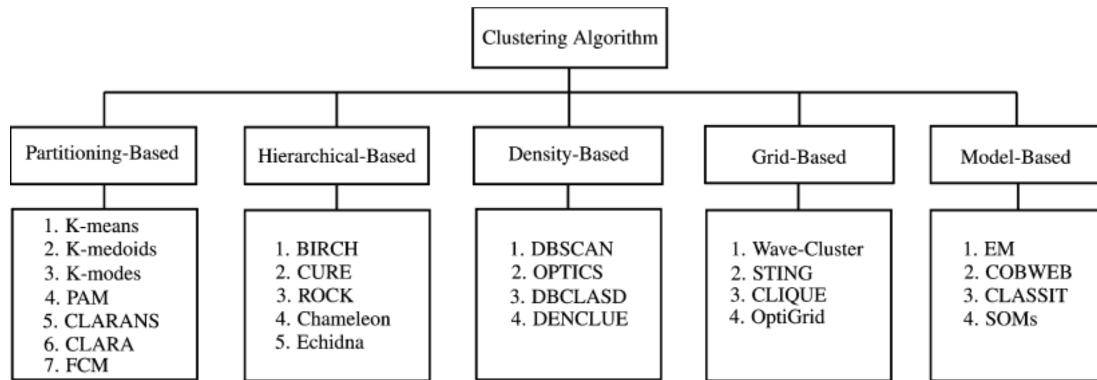



**Figure 1.2:** Clustering taxonomy proposed by Fahad et al. (2014)

big picture of clustering by broadly classifying the taxonomy of clustering algorithms into groups. The framework does not separate the algorithms solving the dynamic or static clustering problem.

The framework proposed by Fahad et al. (2014) is only indicative and the best ideas can be combined from different categories when designing new algorithms. The use of well-researched algorithms, whose strengths and weaknesses are known in advance, as soil for the design of new algorithms emerges when analyzing the new algorithms. In this thesis a new *k*-means based divisive clustering algorithm is designed and analyzed.

First, this thesis delves into the function of the *k*-means algorithm and its weaknesses in Chapter 2. By deconstructing the algorithm, its operating conditions can be better understood. The gained knowledge can then be used when designing a new algorithm. After introducing the *k*-means algorithm, divisive clustering is discussed in Chapter 3. Two different divisive algorithms are introduced which are bisecting *k*-means and iterative splitting algorithm.

Chapter 4 contains the core section of the thesis. First, the building blocks of the split *k*-means algorithm are introduced. Once the theory of the algorithm has been discussed, three variations of the split *k*-means algorithm are introduced. Finally, in Chapter 5 the practical experiments and their results are presented and analyzed.

While split *k*-means is a new algorithm, it is highly influenced by the iterative splitting algorithm. The main idea in implementing a new algorithm is to explore the idea of splitting clusters in simpler way. Iterative splitting algorithm is fast and provides good clustering results, but it may be difficult to implement. Split *k*-means aims to be an alternative for it, that is easier to implement.

# 2. $k$-means

The $k$-means problem can be described as follows: given a finite set $S = \{p_1, p_2, \ldots, p_n\}$ of points and an integer $k \geq 1$, find $k$ centres so that it minimizes the sum of the squared euclidean distances between every point in the set $S$ and their corresponding nearest centre point (Mahajan et al., 2012). It has been shown by Drineas et al. (2004), Dasgupta (2008) and Aloise et al. (2009) that the $k$-means problem remains NP-hard even if the value of $k$ is fixed to two.

$k$-means can be used to refer to both the problem and an algorithm solving it. Although literature often refers to this problem as $k$-means, the term will henceforth be used in this thesis to refer to the naive $k$-means algorithm originally proposed by Stuart P. Lloyd and later published by MacQueen (1967). Lloyd's paper on the original version of $k$-means was published later in 1982 (Lloyd, 1982).

The naive $k$-means algorithm works as follows. First $k$ points are selected as the initial cluster centres. The selection can be done in multiple ways which are presented later in this thesis. A common way, which was proposed by MacQueen (1967) as well, is to select them randomly.

After selecting the initial centroid points, the solution is then enhanced by two subsequent steps: *assignment step* and *update step*, which together form one iteration of $k$-means (Fränti & Sieranoja, 2019). In assignment step every data point is assigned to the nearest centroid point by calculating the euclidean distance to every centroid and by choosing the closest one. The mathematical notation for the assignment step is presented by Malinen and Fränti (2014) as follows:

$$P_j^{(t)} = \{X_i \; : \; ||X_i - C_j^{(t)}|| \; \leq \; ||X_i - C_{j^*}^{(t)}|| \; \forall \; j^* = 1, \ldots, k\}$$

Since a single centroid is the mean of every data point belonging to the cluster the centroid represents, it has to be recalculated if the partition is modified. The recalculation

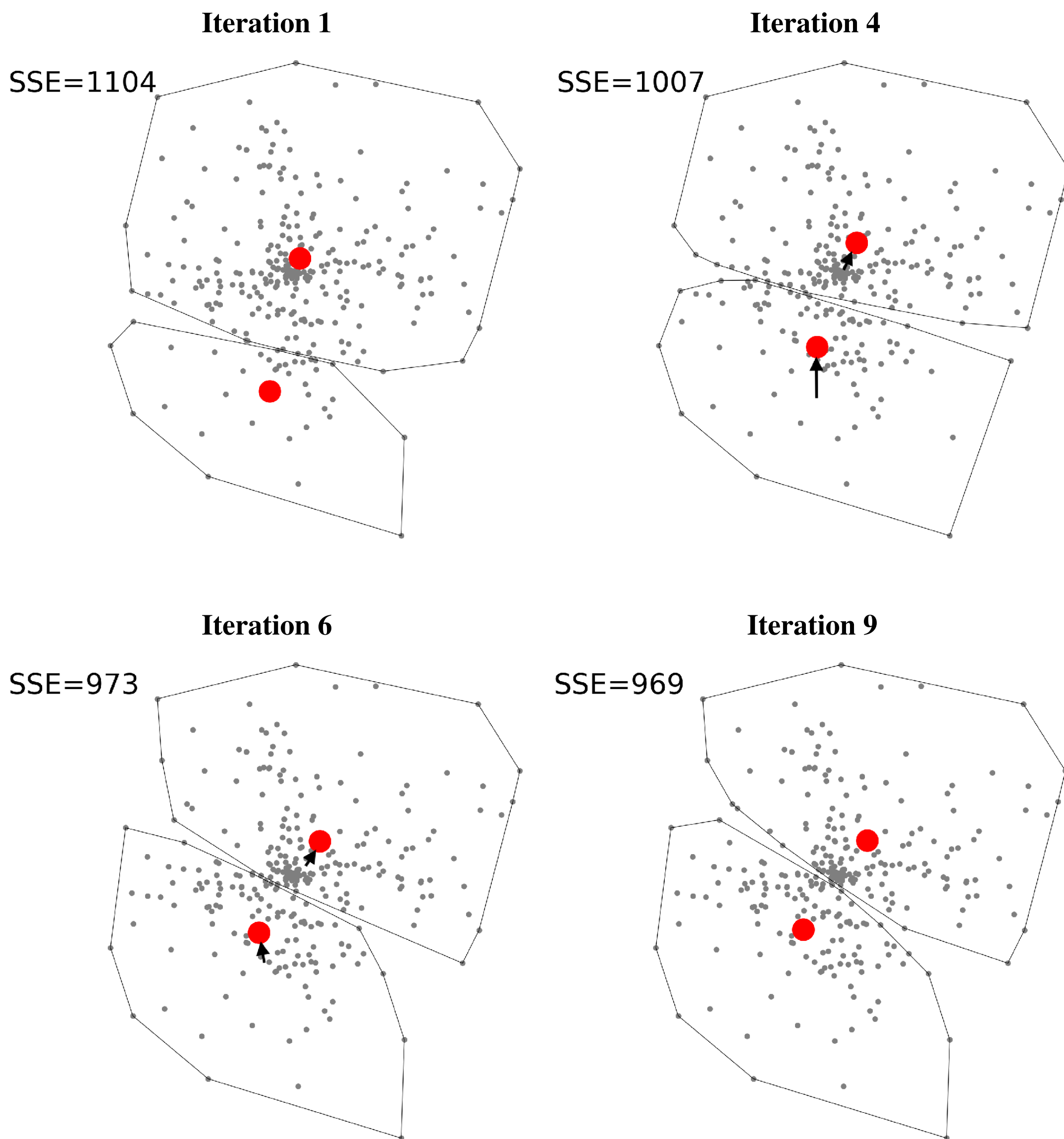



**Figure 2.1:** *k*-means iterations move the centroids until their locations are stabilized.

is done in update step. Malinen and Fränti (2014) present the mathematical notation as follows:

$$C_j^{(t+1)} = \frac{1}{|P_j^{(t)}|} \sum_{X_i \in P_j^{(t)}} X_i$$

The algorithm continues iterating until the centroids are stabilized. The stabilization means that the centroids are in the same locations after the iteration as they were before it. The *k*-means clustering process is presented in Figure 2.1 in where the centroids are moved until stable positions are found after ninth iteration.

A fixed number of iterations can also be used and it is common to define a maximum iteration count in publicly distributed $k$-means implementations such as the $k$-means algorithm in the scikit-learn package (Pedregosa et al., 2011).

Figure 2.2 represents the $k$-means algorithm. First it preforms the assignment step by finding the nearest centroid for every data point. The UpdateCentroids function used in the assignment step is presented in Figure 2.3. The assignment step produces a partition data structure ($P$) where the index of the data point contains the index of the centroid to which it is assigned. For example, if the data point in index 10 ($X_{10}$) has the centroid in index 2 ($C_2$) as the closest centroid, then it is assigned to it by placing the value of 2 in partition data structure index 10 ($P_{10} = 2$).

After assigning the data points into clusters, the centroid locations are updated with the UpdateCentroids function that is presented in Figure 2.4.

```
KMeans(X, C, P)
    repeat
        C_previous ← C
        for i ← 1; N do
            P_i ← FindNearestCentroid(X_i, C)
        end for
        C = UpdateCentroids(X, C, P)
    until C = C_previous
    return C, P
end
```

**Figure 2.2:** The standard $k$-means algorithm

```
FindNearestCentroid(x, C)
    d_min ← ∞
    for i ← 1; K do
        d ← CalculateEuclideanDistance(x, C_i)
        if d < d_min then
            d_min ← d
            i_min ← i
        end if
    end for
    return i_min
end
```

**Figure 2.3:** Algorithm for finding the nearest centroid

```
UpdateCentroids(X, C, P)
    sums ← [0_1, ..., 0_K]
    counts ← [0_1, ..., 0_K]
    for i ← 1; N do
        sums_{P_i} ← sums_{P_i} + X_i
        counts_{P_i} ← counts_{P_i} + 1
    end for
    for k ← 1; K do
        C_k ← sums_k / counts_k
    end for
    return C
end
```

**Figure 2.4:** Algorithm for updating the centroids of the clusters

## 2.1 Initialization methods

The $k$-means algorithm needs an initial solution. The initial solution is a set of centroids that the algorithm starts to move into better locations with the assign and update steps. By moving the centroids, $k$-means tries to minimize the result of the objective function that is usually Sum of Squared Errors (SSE). Since $k$-means is a greedy algorithm, it tends to find the local optimum solution instead of the global one. This has been noted to be the main limitation of $k$-means by Fränti and Sieranoja (2019).

The initialization of centroids is important when seeking the global minimum solution. Depending on the dataset, it might be even impossible for $k$-means to find the global minimum if the centroids happen to be in unfavourable locations. The initial solution may be random, based on some heuristic method or the initialization can be done by some other clustering algorithm as well.

#### Random centroids

```
InitializeRandomCentroids(X, K)
    C ← [0_1, ..., 0_K]
    for i ← 1; K do
        C_i ← PickRandom(X)
    end for
    return C
end
```

**Figure 2.5:** Algorithm for initializing $k$ centroids randomly

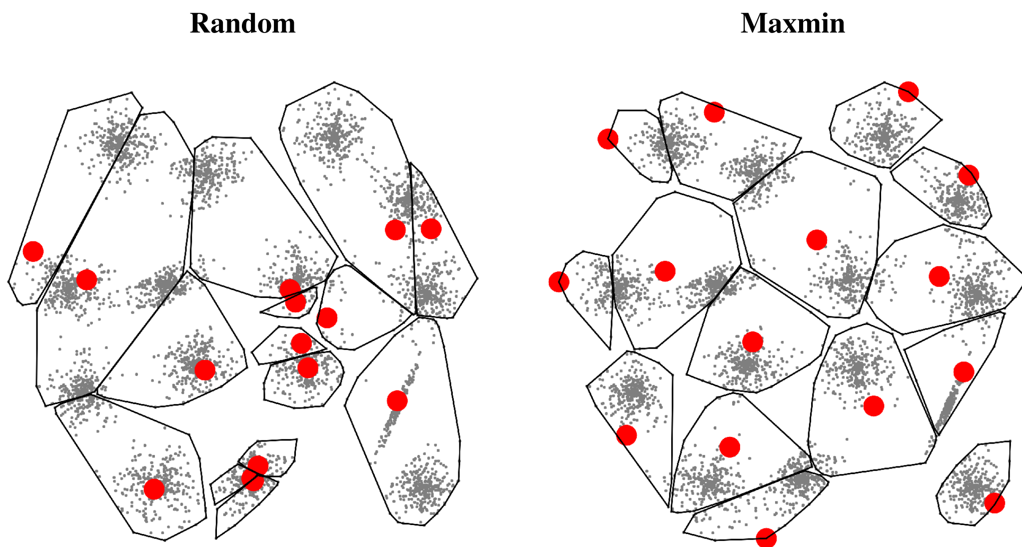



**Figure 2.6:** Random and Maxmin initialization methods. Random initialization is a fast way to initialize the centroids, but the resulting clusters may be located in a narrow area which in turn results to locally optimal clustering. Maxmin is good at avoiding this scenario when all centroids are located narrowly since it spreads the centroids widely.

The most popular method of selecting the initial centroids is selecting $k$ number of randomly picked data points as the initial centroids (Fränti & Sieranoja, 2019). This ensures that every cluster has at least one data point in it. The selection can be done, for example, by just selecting $k$ first data points in the dataset. That would guarantee that the selection is always the same. However, if the data is not in random order the centroids will be just next to each other which is often unfavourable because it tends to result in locally optimal solution. An algorithm for picking data points randomly as the initial solution is presented in Figure 2.5.

Another way to perform the random centroids' selection is to shuffle the data by swapping every point for a randomly chosen data point and choosing the first $k$ points after that (Fränti & Sieranoja, 2019). This takes $O(n)$ time and guarantees that the points are independent of the order of data.

Both of these methods guarantee that one data point cannot be chosen multiple times as an initial centroid. Also, since randomness is a required feature for some $k$-means variants, such as repeated $k$-means, shuffling the data points before selection provides a good way to generate different initial solutions for the algorithm (Fränti & Sieranoja, 2019).

### Random partitions

Selecting random centroids is not the only way to perform a random selection. In random partition selection, every data point is assigned into a random partition and after this assignment phase the centroids are calculated as the means of these partitions (Fränti & Sieranoja, 2019). The centroids of these partitions are typically in the centre of the data since they are just average points of randomly distributed data points.

This method provides a more deterministic approach selecting the initial centroids than the *random centroids* method. However, this method suffers from the same weakness as the *k*-means algorithm itself. If there is low or no overlap in the data, the algorithm does not perform well and is not able to find the global minimum (Fränti & Sieranoja, 2019).

### Furthest point heuristic

The furthest point heuristic selects the first centroid arbitrarily and continues to select the next centroids one by one. Every new selected centroid is the furthest point from its nearest centroid (Fränti & Sieranoja, 2019). The algorithm for calculating the furthest point heuristic is presented in Figure 2.7. Since this method is also known as the *maxmin* method and more often referred with that name in the source literature, it is called as the maxmin method in this thesis as well.

Basically, the algorithm in Figure 2.7 calculates the distances between every data point and every selected centroid so far. All data points are assigned to their closest centroid and the next centroid that is chosen is the data point with the longest distance to its closest centroid. Selecting new centroids is continued until *k* centroids are found. Examples of the resulting initializations of random centroids and the maxmin method can be seen in Figure 2.6.

### Sorting heuristics

The selection can be done by first sorting all data points according to some criterion and then selecting points by some heuristics (Fränti & Sieranoja, 2019). Possible sorting criteria may be at least distance to the centre point, density, centrality and attribute with the greatest variance.

```
Maxmin(X, K)
    C ← [0_1, ..., 0_K]
    C_1 ← PickRandom(X)
    for k ← 2; K do
        furthestPoint ← null
        maximumDistance ← 0
        for i ← 1; X do
            closestCentroid ← null
            minimumDistance ← ∞
            for j ← 1; C do
                distance ← CalculateEuclideanDistance(X_i, C_j)
                if distance < minimumDistance then
                    closestCentroid ← X_i
                    minimumDistance ← distance
                end if
            end for
            if minimumDistance > maximumDistance then
                furthestPoint ← X_i
                maximumDistance ← minimumDistance
            end if
        end for
        C_k ← furthestPoint
    end for
    return C
end
```

**Figure 2.7:** Maxmin algorithm for choosing $K$ initial centroids

After sorting, the selection can be done at least by selecting the first $k$ points, selecting the $k$ first points while avoiding points within a distance of $\epsilon$ to the chosen centroids or by selecting every $(N/k)$th point.

No single superior method rise above others in the literature when comparing the sorting heuristic methods. According to Fränti and Sieranoja (2019), the sorting heuristic would be effective if the clusters were distinctly separated and had varying criterion values. This happens with some datasets but more often the resulting centroids tend to be just random data points in the dataset.

### Projection-based heuristics

Several projection-based heuristics can be found in the literature. In projection-based heuristics, the projection is done by some method and after that the data points are partitioned into clusters of equal size, each containing $k$ data points (Fränti & Sieranoja,

2019). The projection can be done with several methods, but a common way, that is used in several algorithms, is calculating the data's principal axis. The projection based on the principal axis maximizes the variance (Fränti & Sieranoja, 2019).

Another method for calculating the axis is choosing two random data points as the reference points of the projection line. This provides randomness for the algorithm but does not necessarily lead to a good result without repeating the initialization several times (Fränti & Sieranoja, 2019).

A more deterministic alternative to two random points is to select only one reference point randomly and selecting the furthest point of it as the second reference point of the axis. This method still includes randomness since the first point is chosen randomly. However, the results may be better and require fewer repeats of the algorithm.

**Density-based heuristics**

Calculating the density is not a trivial task. Fränti and Sieranoja (2019) mention three different ways to initialize centroids by using the density and the first one is by using buckets. When using buckets, the dataset is divided with a grid to buckets and the frequencies of the resulting buckets are calculated. The number of buckets varies, but the value of $10 \times k$ is often used. The bucket technique is not trivial to apply to high dimensional datasets while it works for low-dimensional data.

Another two methods mentioned by Fränti and Sieranoja (2019) process every data point separately and calculate the densities for each of them. $\epsilon-$radius method uses $\epsilon$ as the distance threshold and counts the number of data points inside the resulting area. The second method uses the $k$-nearest neighbours ($k$-NN) method for finding the nearest neighbours and then calculate the average distance to them.

The last two ways to initialize the centroids are computationally heavy since the neighbourhood has to be calculated for every single data point. A straightforward implementation leads to $O(n^2)$ time complexity, but the time complexity can be reduced to $O(n^{1.5})$ if the calculations are done only for a reduced $\sqrt{n}$ sized subset of the data (Fränti & Sieranoja, 2019).

## 2.2 Objective function

While $k$-means is a *clustering algorithm* that tries to minimize the objective function, the objective function can be seen as a *clustering method*. According to Fränti and Sieranoja (2019) the selection of the objective function is even more significant than the selection of the actual clustering algorithm since in real life applications, the clustering results primarily depend on the clustering method and secondarily on the clustering algorithm.

The Sum of Squared Errors (SSE) is a common objective function found in the literature. As an error, the euclidean distance is calculated between every point and their corresponding nearest centroid point. The euclidean distance and the sum of squared error can be defined as follows.

$$d(x, c) = \sqrt{(x_1 - c_1)^2 + (x_2 - c_2)^2 + \ldots + (x_n - c_n)^2}$$

$x$ represents a data point in the dataset, $c$ represents a centroid and $p$ represents the partition set.

$$\text{SSE} = \sum_{i=1}^{N} d(x_i, c_{p_i})^2$$

With SSE value, the Mean Squared Error (MSE) can be calculated with the following formula where $N$ is the length of the dataset and $d$ is the number of dimensions in the dataset. The algorithm for calculating MSE is presented in Figure 2.8.

$$\text{MSE} = \frac{\text{SSE}}{N \cdot d}$$

CalculateMSE($X, C, P$)
  sum $\leftarrow$ 0
  **for** $i \leftarrow 1; N$ **do**
    sum $\leftarrow$ sum + distance$(X_i, C_{P_i})^2$
  **end for**
  **return** $\frac{\text{sum}}{N \cdot d}$
**end**

**Figure 2.8:** Algorithm for calculating the MSE value of a clustering result

## 2.3 Comparing results with the centroid index

In order to compare clustering results with each other and with the ground truth as well, a comparison algorithm is needed. The objective function itself is not able to compare the structure of the clustering and the raw value of most external indexes and neither SSE nor MSE tells how significant the result is (Fränti & Sieranoja, 2019). Centroid index (CI) is a measurement method proposed by Fränti et al. (2014) for calculating the cluster level similarity between two clustering results.

First, the centroid index creates a map of the nearest neighbours for two sets of cluster prototypes and then counts the prototypes that did not have any neighbours. The pseudo code of the method can be seen in Figures 2.9 and 2.10 and the mathematical notation of the method is presented by Fränti et al. (2014) as follows:

$$C = c_1, c_2, c_3, \ldots, c_{K1}$$

$$C' = c'_1, c'_2, c'_3, \ldots, c'_{K2}$$

$$q_i \leftarrow \underset{1 \le j \le K2}{\arg\min} ||c_i - c'_j||^2 \ \forall \ i \in [1, K1]$$

The prototypes that did not have any neighbors are called *orphans*.

$$\text{orphan}(c'_j) = \begin{cases} 1 & q_i \neq j \forall i \\ 0 & \text{otherwise} \end{cases}$$

$$CI_1(C, C') = \sum_{j=1}^{K2} \text{orphan}(c'_j)$$

The mapping is not symmetric ($C \rightarrow C' \neq C' \rightarrow C$). It means that the result may differ when calculating the orphans in another way. According to Fränti et al. (2014) the symmetrical version maps the sets in both ways ($C_1 \rightarrow C_2$ and $C_2 \rightarrow C_1$) and chooses the maximum of those results.

$$CI_2(C, C') = \max[CI_1(C, C'), CI_1(C', C)]$$

The result is an integer that clearly tells how many centroids are located differently in the two results that were mapped against each other. A common way to measure clustering structure quality is to map the resulting centroids of a clustering algorithm to the ground truth centroids, but it can also be used to compare the results of two

clustering algorithms. It can be useful for example in situations when there is no ground truth available at all for the target dataset. An example of interpreting the result of the Centroid Index method is presented in Figure 2.11.

```
CalculateOrphans(C_1, C_2)
    M ← [0_1, ..., 0_K]
    for i ← 1; C_1 do
        c ← null
        c_dist ← ∞
        for j ← 1; C_2 do
            d ← EuclideanDistance(C1_i, C2_j)
            if d < c_dist then
                c ← j
                c_dist ← d
            end if
        end for
        M_c ← M_c + 1
    end for
    x ← 0
    for i ← 1; M do
        if M_i = 0 then
            x ← x + 1
        end if
    end for
    return x
end
```

**Figure 2.9:** The orphan prototypes are found by calculating the closest prototype from the second set, keeping a record of the number of neighbors and finally counting the ones that have no neighbors.

## 2.4 Random swap

Random swap is a clustering algorithm proposed by Fränti and Kivijärvi (2000). It is an iterative algorithm that aims to find the global minimum by randomly switching the locations of centroids. Random swap is presented here since it is used later in Chapter 5 as a benchmark algorithm.

The pseudo code of the random swap algorithm is presented in Figure 2.12. In every iteration of the random swap algorithm one *trial swap* is made. Trial swap means that one centroid is removed and one new random data point is selected as a new centroid. After the swap phase, a *local repartition* is optionally run. The local repartition is done

```
CentroidIndex(C1, C2)
    a ← CalculateOrphans(C1, C2)
    b ← CalculateOrphans(C2, C1)
    return max(a, b)
end
```

**Figure 2.10:** The centroid index calculation is symmetric when the mapping is done in both ways

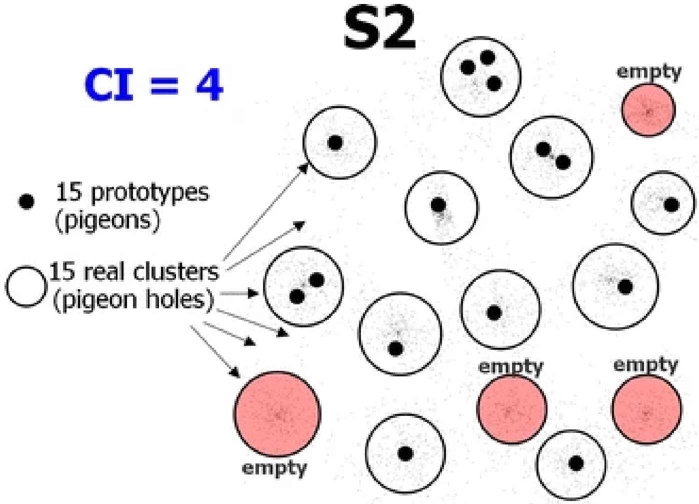



**Figure 2.11:** Fränti (2018) illustrated the CI algorithm result interpretation with the S2 dataset. The smaller black dots (*pigeons*) represent the clustering algorithm result prototypes and the large circles (*pigeon holes*) represent the ground truth prototypes. In CI = 0 result there would be only one pigeon in every pigeon hole. However, in this case the CI value equals 4 and according to that there are four orphan prototypes. Those are represented in this picture as empty pigeon holes.

in order to speed up the algorithm and it is not necessary especially if the $k$-means algorithm is run multiple times (Fränti, 2018).

After the local repartitioning, an ordinary $k$-means is run. Most commonly two iterations of $k$-means are used and after that the objective function $f$ evaluates the solution. If the trial swap provides a better solution that manages to decrease the value of the objective function, then the centroids and partition are replaced with the new ones.

The swaps can be categorized in three categories demonstrated in Figure 2.13: *trial*, *accepted* and *successful* (Fränti, 2018). Every iteration performs one trial swap, but only the ones that improve the objective function are accepted. However, an accepted swap does not improve the structure of the clustering. A successful swap improves the overall structure of the clustering reducing the CI value. It means that the swap managed to pick up a centroid from a congested area and drop it into a sparse area.

```
RandomSwap(X)
    C ← Select random representatives(X)
    P ← Optimal partition(X, C)
    for T times do
        C_new, j ← Random swap(X, C)
        P_new ← Local repartition(X, C_new, P, j)
        C_new, P_new ← KMeans(X, C_new, P_new)
        if f(C_new, P_new) < f(C, P) then
            C, P ← C_new, P_new
        end if
    end for
    return C, P
end
```

**Figure 2.12:** Random swap iterates $T$ number of times performing a single trial swap in each iteration. In the pseudo code $f$ represents the objective function that evaluates the clustering result (Fränti, 2018).

The recommended number of trial swaps is 5 000 which is commonly used as the default iteration value in public random swap implementations. Also, $T = N$, where $T$ is the number of iterations and $N$ is the number of data points in the dataset, can be used as a general rule (Fränti, 2018). However, since the number of iterations can be easily changed, random swap is an excellent algorithm when a high clustering quality is desired. In these cases, the algorithm can be continuously iterated for a fixed number of time (Fränti, 2018).

## 2.5 How many clusters?

$k$-means solves the static clustering problem. It means that it needs the number of clusters pre-determined. The number of clusters can be manually determined or an algorithm can be used to estimate it. In the literature, various approaches are discussed for solving the number of clusters.

#### Brute force

According to Kärkkäinen and Fränti (2002a) the simplest way to find out the number of clusters is to brute force it. *Brute force* (BF) algorithm in this context means that a selected static clustering algorithm is applied to the target data with different values of $M$ in a decided range $[M_{min}, M_{\max}]$ and the results of each clustering are evaluated with

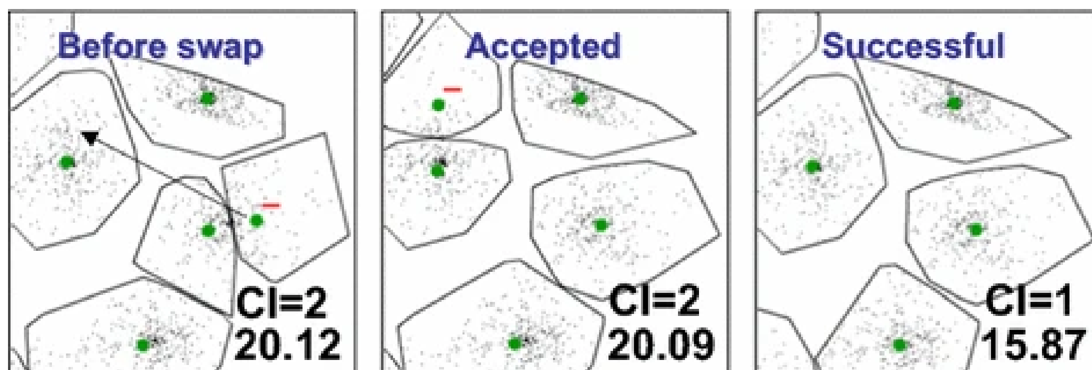



**Figure 2.13:** The random swap algorithm swaps centroids by taking a random centroid and placing it in a randomly chosen location (left). An accepted swap is a swap that decreases the total SSE (middle). In addition to decreasing the total SSE, a successful swap decreases the CI as well (right). The swaps are illustrated by Fränti (2018) with the S2 dataset.

an evaluation function $f$. With the correct evaluation function and clustering algorithm, the correct number of clusters can be determined using the brute force technique even though the process will be slow.

Since MSE does not take the number of clusters into account, it cannot be used as the evaluation function. Kärkkäinen and Fränti (2002b) considered two different measurement methods as the evaluation functions: *Davies-Bouldin index* (DBA) and *variance ratio F-test*.

#### Stepwise clustering algorithm

Another algorithm for finding the number of clusters is the stepwise clustering algorithm proposed by Kärkkäinen and Fränti (2002b). It is similar to the brute force algorithm, but it is designed to be faster. The basic idea is that since the centroids should be in almost the same places when the number of centroids decreases by one, there is no need to start clustering completely from scratch. The centroids from the previous iteration can be kept when a random centroid is removed. The algorithm has two variants, $Step^+$ and $Step^-$, the first of which increases the number of centroids by one on each iteration and the latter decreases them by one. The logic by which a centroid is removed or which a point is chosen as a new centroid is derived from the random swap algorithm. The centroids are chosen randomly. Pseudocode of the $Step^-$ algorithm is presented in Figure 2.14.

```
Step⁻(X, M_min, M_max)
    C, P ← InitializeRandomSolution(X, M_max)
    C_best, P_best ← RandomSwap(X, C, P, M_max)
    for m ← M_max − 1; M_min do
        C ← RemoveRandomCentroid(C)
        C, P ← RandomSwap(X, C, P, m)
        if f(X, C, P) < f(X, C_best, P_best) then
            C_best, P_best ← C, P
        end if
    end for
    return C_best, P_best
end
```

**Figure 2.14:** Stepwise clustering algorithm's Step$^{-}$ variant decreases the number of centroids by one on each iteration (Kärkkäinen & Fränti, 2002b).

# 3. Divisive clustering

Hierarchical clustering algorithms construct a tree-like hierarchy of the clusters. They can start with $n$ data points that belong to a single cluster or start from the opposite direction where there are $n$ clusters containing only a single data point. *Agglomerative methods* start with $n$ clusters and merge them together creating new ones. *Divisive methods*, discussed in this chapter, start with only a single cluster and start by dividing it and the resulting clusters into new clusters (Kaufman, 1990).

Divisive, also known as split based, algorithms have been proposed in literature before. One of them is an iterative splitting algorithm proposed by Fränti et al. (1997) and the second one is the bisecting $k$-means presented by Steinbach et al. (2000). These two algorithms are presented in this thesis since they have influenced the split $k$-means algorithm the most.

## 3.1 Iterative splitting algorithm

The iterative splitting algorithm shown in Figure 3.1 starts with a single cluster and continues to split until the desired number of clusters have been found, just like other hierarchical divisive clustering algorithms. The algorithm can be implemented in such a way that it splits only one cluster in each round or in such a way that it splits all clusters in each round. The variant that splits all clusters performs a binary split for the clusters (Fränti et al., 1997).

Choosing to split only one cluster at a time results in $M$ steps in total when running the algorithm. When all clusters are split on each iteration, $\log M$ steps are executed. In this case, the cluster selection step in Figure 3.1 is naturally skipped since all clusters are selected.

```
Iterative splitting algorithm
    m ← 1
    Calculate the training set centroid
    repeat
        Select cluster(s) to be split
        Split the cluster(s)
        m ← m + 1
        Refine the partitions and code vectors
    until m = M
    return M code vectors
end
```

**Figure 3.1:** Iterative splitting algorithm (Fränti et al., 1997)

Fränti et al. (1997) propose four different variants of the algorithm which perform the partition refining phase differently. Some variants use the *generalized Lloyd algorithm* (GLA) when tuning the intermediate or final solutions while others do not use it.

**Split** Iterative splitting algorithm

**S+GLA** Iterative splitting as an initial codebook to GLA

**SGLA** Iterative splitting using GLA at partition refining phase

**SLR** Iterative splitting using local repartitioning at partition refining phase

### Choosing the cluster to split

Considering the variant that splits only one cluster at a time, an important design choice is to select the cluster to be split. Fränti et al. (1997) propose four methods for the selection: *the highest variance*, *the skewest cluster*, *the widest cluster* and *the local optimization strategy*.

The highest variance method chooses the cluster with the highest individual variance. The method does not result in an optimal solution even though it would be a natural choice when minimizing the total squared error (Fränti et al., 1997). The method is not able to detect the multimodality of the cluster nor is it able to choose the cluster that yields the greatest improvement in the total distortion.

The method of choosing the widest cluster is to calculate the maximal distance between two furthest data points in a cluster and compare these results when choosing the cluster.

This method is not capable of detecting the multimodality of the cluster or the greatest improvement after the split (Fränti et al., 1997).

Besides, calculating the widest cluster is a heavy operation since it has an $O(n^2K)$ time complexity when calculating it with the algorithm presented in Figure 3.2. This becomes a bottleneck especially with large clusters. A more convenient way to *approximate* the cluster width is to calculate the furthest data point from the centroid, then calculate the furthest data point from that point and use that distance as the width of the cluster. The algorithm for approximating the cluster width is presented in Figure 3.3. It takes only $O(nK)$ time to approximate the width of the cluster, and the algorithm gives a good enough result to compare the clusters with each other.

```
NaiveClusterWidth(Cluster)
    MaxWidth ← 0
    for i ← 1; N do
        for j ← i; N do
            w ← CalculateEuclideanDistance(Cluster_i, Cluster_j)
            if w > MaxWidth then
                MaxWidth ← w
            end if
        end for
    end for
    return MaxWidth
end
```

**Figure 3.2:** A naive algorithm for calculating the width of a cluster. The *cluster* represents all data points in the target cluster.

The skewest cluster can be selected as well. Measuring the multimodality is difficult, but the skewness of the cluster can be calculated by calculating the third moment. Fränti et al. (1997) approximate the skewness with the following formula. A high value of $w$ indicates a skew distribution of vectors inside the cluster.

$$w = \left| \sum_i \left| x_i - \bar{x} \right| (x_i - \bar{x}) \right|$$

The fourth method is to select the cluster which decreases the distortion the most. This is called local optimization strategy. After every iteration, the value is known for all clusters except two new clusters from previous split. The number of splits is therefore doubled when comparing to the previously discussed methods.

```
ApproximateClusterWidth(Cluster, Centroid)
    MaxDistance ← 0
    for i ← 1; N do
        d ← CalculateEuclideanDistance(Cluster_i, Centroid)
        if d > MaxDistance then
            MaxDistance ← d
            FirstPoint ← Cluster_i
        end if
    end for
    MaxDistance ← 0
    for i ← 1; N do
        d ← CalculateEuclideanDistance(Cluster_i, FirstPoint)
        if d > MaxDistance then
            MaxDistance ← d
        end if
    end for
    return MaxDistance
end
```

**Figure 3.3:** An algorithm for approximating the width of a cluster. *Cluster* represents all data points in the target cluster and *Centroid* represents the centroid of the target cluster.

### Splitting methods

Fränti et al. (1997) categorize the splitting methods in two categories: *code-vector-based* (CB) and *partitioning-based* (PB). Code-vector-based variants choose two new data points in a cluster by some heuristic approach. Those vectors become the new centroids and the old one is removed. Partitioning-based methods are based on the *principal-component analysis* (PCA).

A method proposed by Linde et al. (1980) is to choose the new vectors by calculating $C - \epsilon$ and $C + \epsilon$ and using those as the new centroids. $\epsilon$ represents a fixed perturbation vector. Fränti et al. (1997) fixed the perturbation vector to the standard deviation ($\epsilon = \sigma$). However, if the direction of the perturbation vector is not considered, it does not have much use. A better approach is just to choose two random vectors as the new centroids (Fränti et al., 1997).

The third heuristic method proposed by Fränti et al. (1997) is called the *two-furthest-strategy method* and it is a modification of the widest cluster calculation. It works by calculating the furthest vector $C_1$ of the centroid $C$ and finally calculating the furthest vector $C_2$ of the vector $C_1$. The method works in $O(nK)$ time.

The second category, partitioning-based methods, is not based on heuristics, but instead on the principal-component analysis. The rough algorithm for splitting using the PCA is described in Figure 3.4.

```
Calculate the principal axis using the power method
Select the dividing point P on the principal axis
Partition the training vectors with a hyperplane
Calculate two new code vectors as the centroids of the two subclusters
```

**Figure 3.4:** A rough principal-component based split algorithm proposed by Fränti et al. (1997)

### Refinement of partitions

Fränti et al. (1997) state that a natural choice to refine the partitions and code vectors is to apply a couple GLA iterations to the intermediate solutions. However, since applying a full global GLA iteration is expensive, a new method is proposed. The method is called *local repartitioning* and it works by comparing each data point with their currently closest centroid and the two new centroids that were produced by the last split.

The local repartitioning algorithm is presented in a simplified way in Figure 3.5. Since the data points of the split cluster no longer have the original centroid left, the comparisons are performed only between the new centroids ($C_{\text{new}_1}$ and $C_{\text{new}_2}$). Respectively for vectors outside of the split cluster, the comparisons are performed between their originally closest centroid ($C_i$) and the new centroids.

```
LocalRepartition(X, C, P, C_new1, C_new2)
    for i ← 1; N do
        C_old ← C_{P_i}
        P_i ← SelectNearestCentroid(X_i, C_old, C_new1, C_new2)
    end for
    return P
end
```

**Figure 3.5:** Local repartitioning algorithm described by Fränti et al. (1997)

## 3.2 Bisecting *k*-means

In addition to the iterative splitting algorithm, another algorithm that uses splitting to form hierarchical clustering is bisecting *k*-means. The algorithm is presented in the

literature in various ways. However, a common variation of the algorithm is shown in Figure 3.6. Similarly to the iterative splitting algorithm, an important design choice when using bisecting $k$-means is to choose which cluster to split.

```
Bisecting k-means(X, K)
    repeat
        Select a cluster c to split
        repeat
            Use k-means to split c to c1 and c2
            Calculate inter-cluster dissimilarity for c1 and c2
        until Fixed number of iterations
        Select the split that produces the highest overall similarity
    until K clusters are formed
    return P, C
end
```

**Figure 3.6:** Bisecting $k$-means algorithm proposed by Steinbach et al. (2000)

Steinbach et al. (2000) state that various methods exist for selecting the cluster to split. The method can be, for example, choosing the biggest cluster, choosing the one that has the least overall similarity to others or to use some kind of criterion. A criterion can be, for example, based on both size and similarity. Steinbach et al. (2000) decided to split the largest cluster after experimenting with the methods and finding that the differences were small.

Steinbach et al. (2000) believe that bisecting $k$-means outperforms the naive $k$-means because it produces clusters that are relatively uniform-sized. Naive $k$-means tends to produce different-sized clusters. In their experiments, where they clustered documents, they could measure the cluster qualities. Smaller clusters tended to have better quality than bigger ones. However, small high-quality clusters did not contribute much to the overall quality. Bigger clusters had lower quality and often made negative contribution to the overall quality.

In this thesis, bisecting $k$-means implementations always choose the cluster with the greatest SSE value. This approach has also been used by Chen et al. (2021). In addition, the division that is selected is the one that yields the smallest overall SSE.

# 4. Split *k*-means

```
Split k-means
    Initialize cluster
    repeat
        Select cluster
        Split cluster
        Run k-means
    until found clusters = K
    Fine tune the result by k-means
end
```

**Figure 4.1:** Pseudo code of the split *k*-means algorithm

The previous chapters discussed clustering in general, *k*-means and divisive clustering algorithms and their details. In this chapter, split *k*-means is introduced and its details are discussed. First, the building blocks of the algorithm are discussed and after that, the selected variations of it are discussed.

Split *k*-means is an iterative clustering algorithm that solves the initial centroid positions in addition to optimizing them. It starts by putting all data points into a single cluster and splits it. It continues splitting the resulting clusters until *k* clusters are found. However, the value of *k* remains to be predefined.

Similarly to the iterative splitting algorithm that was introduced in Section 3.1, the split *k*-means depends on important design choices such as determining which cluster to split next and how to perform the split. Split *k*-means has taken considerable influence from the iterative splitting algorithm, but its biggest difference is that it uses either intra-cluster *k*-means or global *k*-means for splitting the clusters. In addition, split *k*-means always fine-tunes the final result, while in the iterative splitting algorithm this phase remains to be optional.

In this thesis three variants of the algorithm are presented. The rough high-level pseudo code of the algorithm can be seen in Figure 4.1. The algorithm contains three major

moving parts: *choosing the cluster to split*, *choosing how to split it* and finally *how to use k-means*. The static parts are initialization and finalization since every variant starts with a single randomly chosen centroid and every variant uses *k*-means for fine-tuning the result. Since choosing which cluster to split next often includes splitting as well, splitting is addressed in this thesis first.

## 4.1 How to split?

Splitting a cluster means taking one cluster and producing several new clusters from it by introducing new centroids. The split can be done in multiple ways, but in this thesis the split always produces two new individual clusters that both contain at least one data point. Since the basic idea of split *k*-means is to be a simple algorithm, the *k*-means algorithm is always used when splitting a cluster, in one way or another.

#### Local split

The local split is done by choosing a target cluster, removing the existing centroid of that cluster and by choosing two random points in the target cluster as the new centroids. After initializing the new centroids, the intra-cluster *k*-means algorithm is run. This means that *k*-means is run within the target cluster itself and data points outside of that cluster are not affected. The local split process is illustrated in Figure 4.2. In the first picture the original target cluster is shown. In the second picture, the original centroid is removed and two randomly selected data points are selected as the new centroids. In the third picture the data points have been repartitioned and lastly, in the fourth picture the intra-cluster *k*-means has moved the centroids to their stable locations.

Intra-cluster *k*-means in split *k*-means algorithm is not limited by iterations in local split since in general *k*-means can perform the clustering of two clusters fast enough to not substantially affect the performance of the algorithm.

#### Global split

In the same way as local split, global split removes the centroid from the target cluster and chooses two random data points as the new centroids. However, after choosing the centroids, it runs *k*-means *globally*. As seen in Figure 4.3, this affects other clusters besides the target cluster. In the first picture, the target cluster has been chosen and in

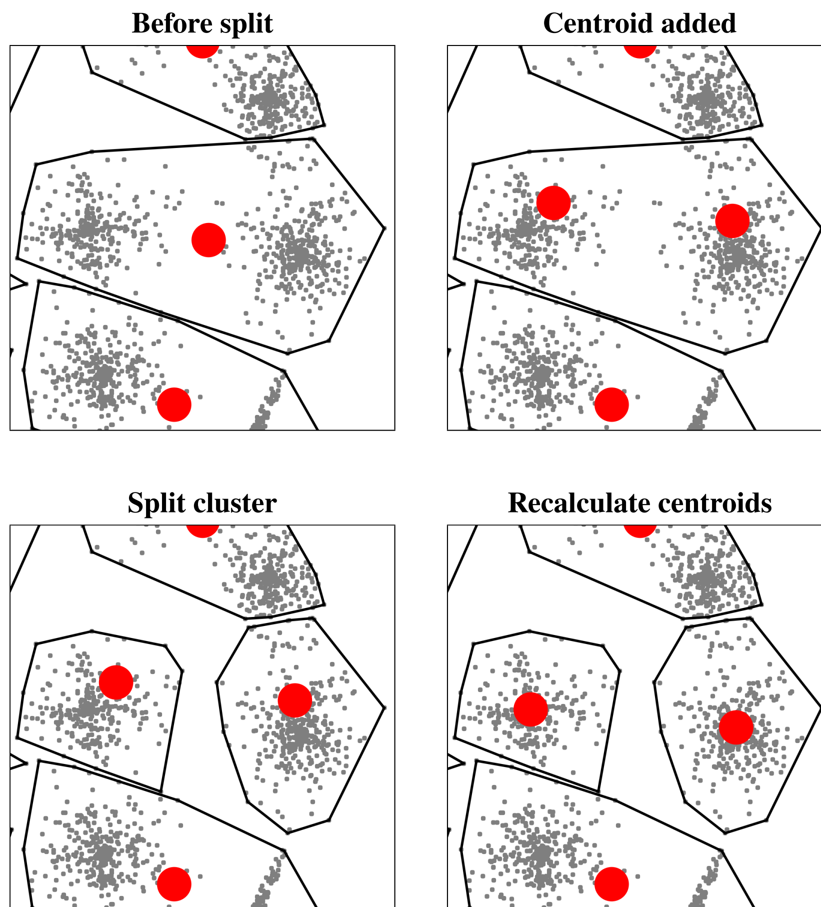



**Figure 4.2:** Local split process removes the original centroid, chooses two new data points as the new centroids and performs intra-cluster $k$-means for the the original cluster.

the second picture the original centroid is removed and two new randomly selected data points are selected as the new centroids. In this case, luck did not favour the algorithm since both of the new centroids are located within the area of the left density peak. The third picture shows that the other centroid has begun its journey towards the right-hand density peak. At the same time, the neighbour clusters' shapes have changed. It can be seen in the last picture that only two global iterations of $k$-means were sufficient to move the new centroids to correct locations. Also, the neighbour clusters and the clusters outside of the picture are fine-tuned.

Since the $k$-means is run globally, it performs slower than the intra-cluster $k$-means, especially with large datasets and with the large values of $k$. For this reason, an iteration limit of two global $k$-means iterations is applied to the global split. The limit number is derived from the random swap algorithm discussed in Section 2.4.

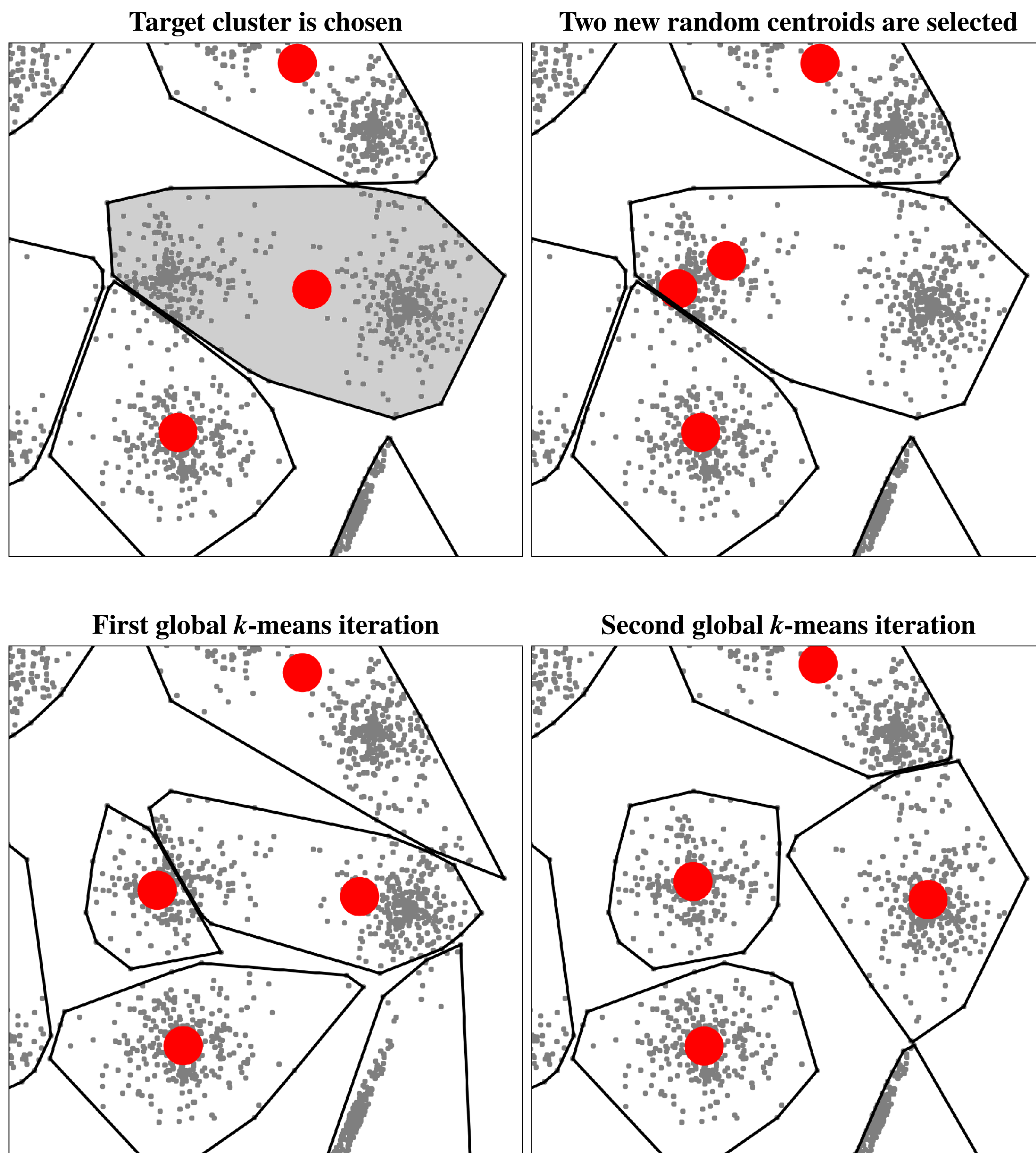



**Figure 4.3:** Global split process affects all clusters, not only the target cluster. Each global $k$-means iteration also fine-tunes the boundaries of the other clusters as well.

## 4.2 Which cluster to split next?

After the split $k$-means has initialized the first cluster, it splits it. Since there is only one cluster to choose from, the choice is obvious. After the first split there are two clusters to choose from and a decision has to be made by some criterion. There are multiple possible criteria and perhaps the most obvious would be to choose a random cluster.

As seen in Figure 4.4, the random split does not usually yield a good clustering structure. Just like plain $k$-means, it can find a locally optimal solution but struggles in finding

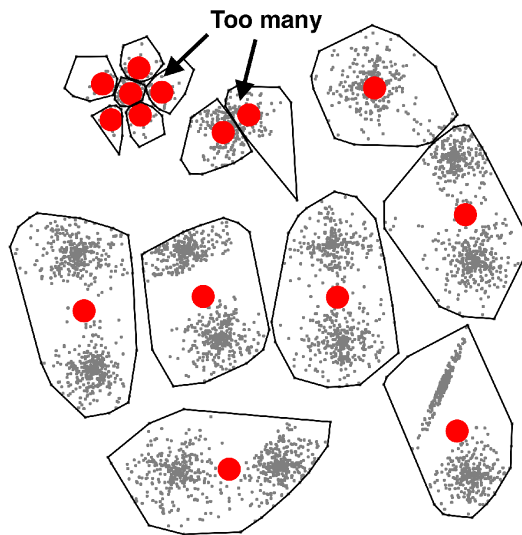



**Figure 4.4:** Local split applied to randomly selected clusters in S1. When the centroids are located in wrong areas, even the fine-tuning after the split process cannot fix the clustering. The final result is a local minimum instead of the global one.

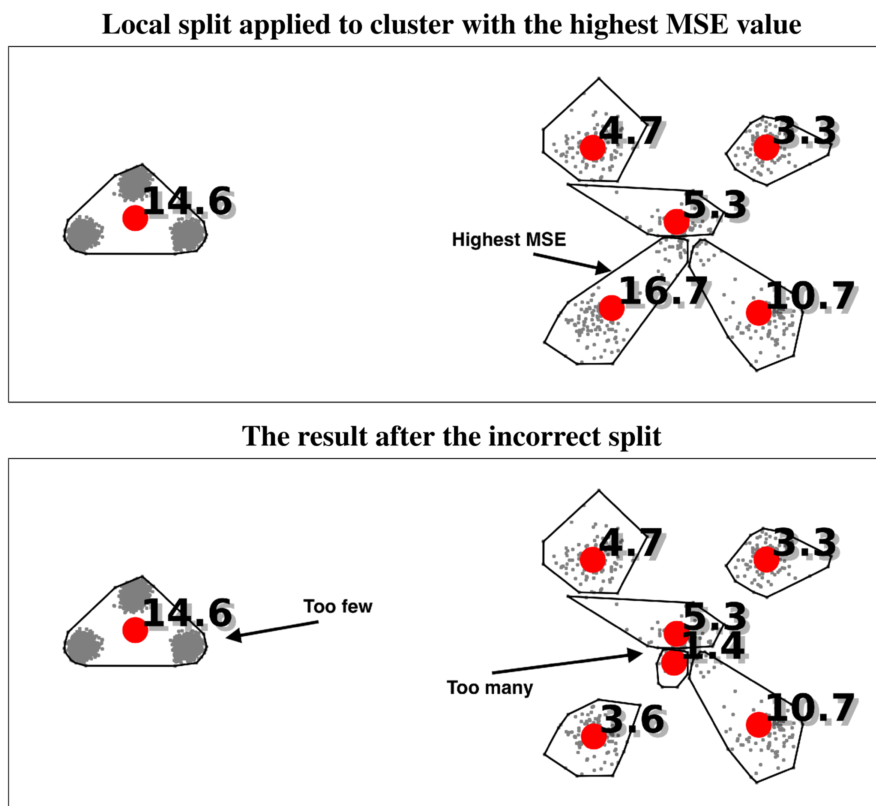



**Figure 4.5:** Choosing the cluster with the highest MSE does not result in good clustering when using local split. Fine tuning with $k$-means is not able to move the redundant centroid from the right side to the left side.

the global optimum. Choosing the cluster randomly does not let the algorithm benefit from the information it could already calculate from the clusters, such as the SSE and

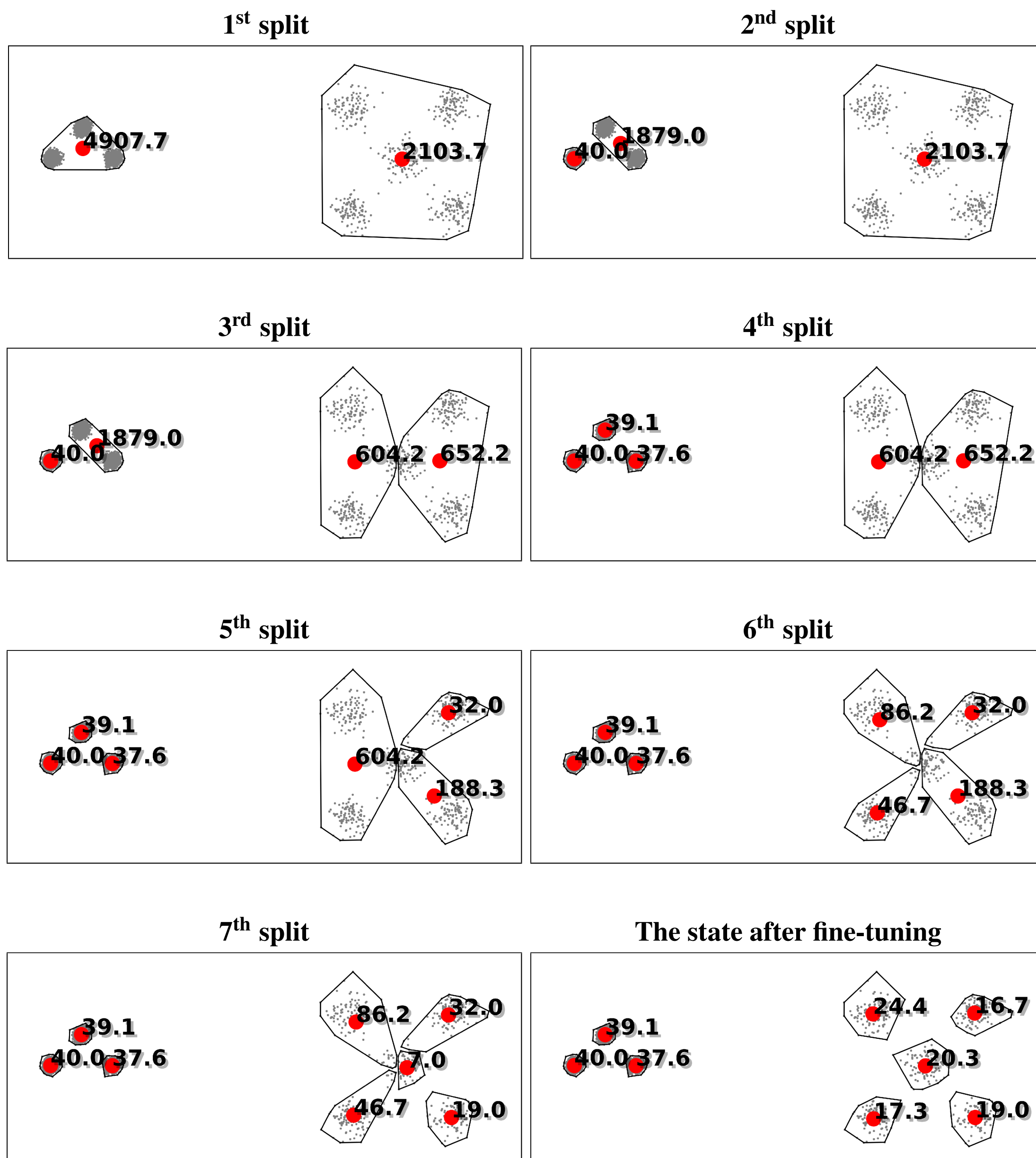



**Figure 4.6:** Unbalance dataset can be correctly clustered by splitting the cluster with the highest SSE.

MSE. Naturally a better way to choose the cluster to split next is to try to collect some information from the clusters and use it when determining the next cluster to split.

A better way to choose the cluster is to choose the one with the greatest MSE value. This method works better than the random method, but what if the cluster with the greatest MSE value is just a larger one with greater variance and some other cluster with smaller MSE actually contains several clusters with smaller variances? An example of this is demonstrated with the *Unbalance* dataset in Figure 4.5.

Since the MSE values between clusters are not comparable, they cannot be used. As seen in Figure 4.5, the algorithm chooses the wrong cluster to be split and the end result is wrong. For this reason only SSE can be used when comparing the clusters with each other. The clustering process when locally splitting the cluster with the greatest SSE value is presented in Figure 4.6.

### Tentative choice

```
Tentative Choice(X, C, P, cache)
    for all clusters do
        if cluster ∈ cache then
            SSE_diff ← cache_cluster
        else
            Calculate SSE_before
            Split cluster
            Calculate SSE_after
            SSE_diff ← SSE_before − SSE_after
            cache_cluster ← SSE_diff
        end if
        if SSE_diff > SSE_maxdiff then
            Target cluster ← cluster
        end if
    end for
    Remove cache_cluster
    return Target cluster, cache
end
```

**Figure 4.7:** Pseudo code of the tentative cluster choice. The cache data structure is simply a lookup table and it can be implemented with an array for example.

The cluster with the greatest value of SSE may not be multimodal and therefore splitting it might result in an error. The idea of tentative choice is to perform trial splits, measure their quality and naturally to select the best split. In the tentative choice, every single cluster will be split by choosing two random points in a cluster and by running the intra-cluster k-means algorithm for that particular cluster. This is called a *local split* which was presented in Section 4.1. The key concept is to calculate the SSE value of the whole clustering result before the tentative split and compare the SSE value after each tentative split with the value before the split. The cluster that is selected is chosen by the split that produces the greatest difference between the SSE values before and after the split. The pseudo code of the algorithm is presented in Figure 4.7 and the method is illustrated in Figure 4.8.

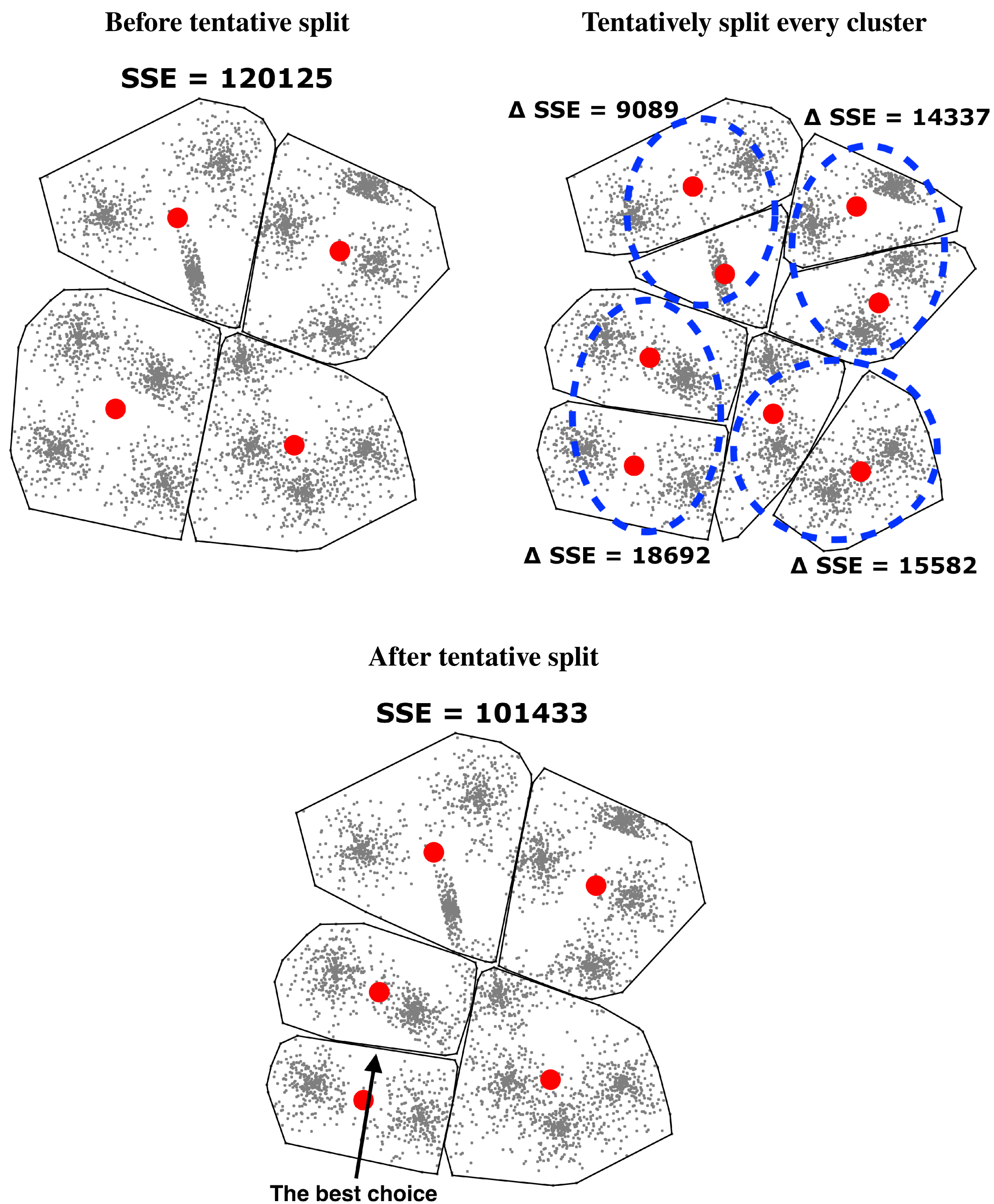



**Figure 4.8:** Tentative choice process when $k4 \rightarrow 5$. In here every tentative split is performed. The SSE values are calculated before every tentative split and after every tentative split. The SSE differences are stored and finally the tentative split with the greatest SSE difference is chosen. After the tentative iteration the intra-cluster split is performed again for the chosen cluster as seen in the last picture. The SSE differences of other splits cached for future use.

Split $k$-means works by iteratively splitting a new cluster in every iteration. Naive tentative splitting ends up processing the same splits over and over again for the same clusters that have been tentatively split in past iterations. Since local split does not affect other clusters, the solution to this performance issue is to keep a record of the SSE difference values of every cluster and to calculate the values only for the new clusters

in every iteration. All splits can be done for the same data structure without a need to start over. In this way in the tentative choice process, the cache data structure can be checked if it contains the SSE difference for a specific cluster already and use that value if it does. The pseudo code seen in Figure 4.7 takes advantage of this method.

Another possibility to tentatively split the cluster is to run $k$-means globally for the whole dataset after choosing the points within the target cluster. This method is called a *global split*. Global split is laborious in comparison with the local split and it affects other clusters as well. Since the global split affects other clusters, every split has to be done individually and started from the same state. On top of that, the algorithm cannot keep a record of SSE difference values since the clusters are not necessary the same on each iteration.

Running $k$-means globally for each cluster split is a heavy process in itself, but copying data structures over and over again and the lack of ability to use a cache for clusters' SSE differences makes it unpractical and too tedious to run for large datasets. For this reason, this method was rejected when designing the algorithm. Only local split is done when tentatively splitting clusters.

## 4.3 Fine-tuning the result

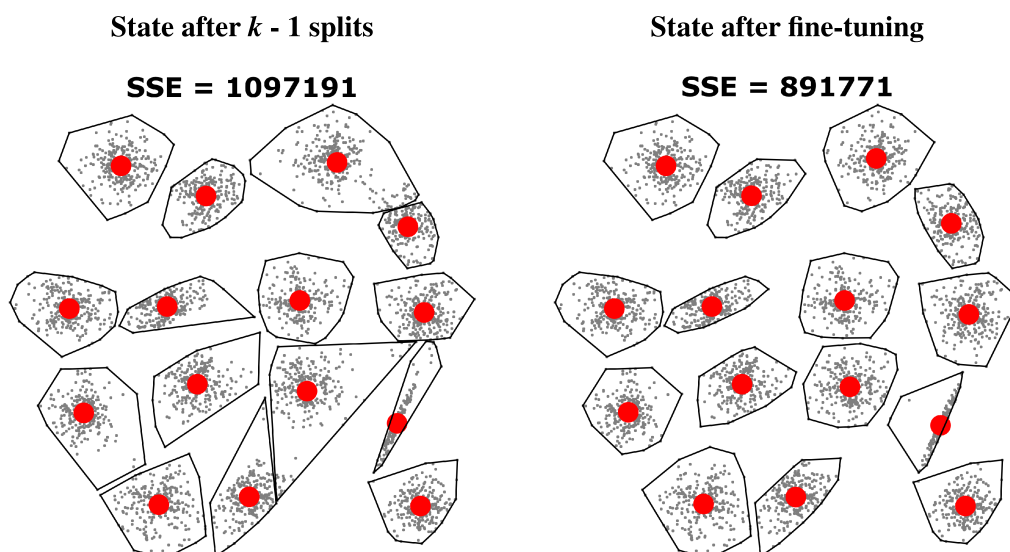



**Figure 4.9:** After the tentative split $k$-means algorithm has performed $k - 1$ splits, the centroids are approximately in correct positions. However, the result on the left is still not optimal and it has to be fine-tuned with the $k$-means algorithm. In the picture on the right, the boundaries of the partitions are fine-tuned.

After the split $k$-means algorithm has reached the desired $k$ number of clusters, it fine-tunes the clustering result with the $k$-means algorithm. Literature shows that $k$-means is an effective algorithm for fine-tuning and it is used similarly for example in the random swap algorithm (Fränti & Kivijärvi, 2000). An example of using $k$-means when fine-tuning the final result of the algorithm can be seen in Figure 4.9.

Fine-tuning can be done for the intermediate results as well. A couple of global $k$-means iterations can be used but if the tentative choice is used when choosing the clusters, moving centroids invalidates the SSE difference cache.

A local repartition can be used to fine-tune the intermediate results. Local repartition does not affect all clusters, it only affects the target clusters and their neighbour clusters. When using it, fine-tuning can be performed without completely invalidating the SSE cache. The local repartition process can be seen in Figure 4.13.

## 4.4 Split *k*-means variants

After recognizing the moving parts of the algorithm and comparing their strengths and weaknesses, three main variants were chosen for final implementation: *full force split*, *tentative split* and *hybrid split*. The variants have the same main structure presented in Figure 4.1, but they differ in how the next cluster is chosen and how it is split. Also, the intermediate results are treated differently. A brief comparison of the design choices is presented in Table 4.1.

**Table 4.1:** Summary of the design choices of the split $k$-means variants and bisecting $k$-means.

| Design parameter | bisecting $k$-means | tentative split | hybrid split | full force split |
|---|---|---|---|---|
| Which cluster to split next? | highest SSE | tentative choice | tentative choice | highest SSE |
| How to split? | local split | local split | local split | global split |
| Intermediate fine-tuning | no | no | local repartition | no |
| Result fine-tuning | no | $k$-means | $k$-means | $k$-means |

### Tentative split

The tentative split variant uses a tentative choice in cluster selection. It tentatively splits every cluster and chooses the one that decreases SSE the most. The selected cluster is split by local split and the intermediate results are not processed at all. After $k$ clusters

```
Tentative Split k-means
    Initialize cluster
    repeat
        Tentatively select cluster that decreases SSE the most
        Locally split the cluster
    until found clusters = K
    Fine tune the result by k-means
end
```

**Figure 4.10:** Pseudo code of the tentative split *k*-means algorithm

are found, the result is fine-tuned with *k*-means. The high-level pseudocode of the tentative split variant is presented in Figure 4.10.

However, the pseudo code does not tell the implementation details of the algorithm. Since the local split used by the variant affects only the target cluster, the SSE differences of the previous iteration can be used in the next iteration. The tentative process can be accelerated by saving the SSE differences in a cache data structure. On the next iteration, the tentative splitting process does not have to split every cluster, but only the new clusters. The entry of the selected cluster can be invalidated from the cache when it is split. This way the tentative splits have to be performed only on the new clusters from the previous iteration and the rest of the SSE differences can be found in the cache.

Similarly to the iterative splitting algorithm by Fränti et al. (1997), the time complexity of the tentative split variant's splitting phase depends on the sizes of the clusters. If it can be assumed that the division will proceed evenly, then the time complexity is as follows:

$$n + \frac{n}{2} + \frac{n}{3} + \ldots + \frac{n}{k} + \ldots + 1 = \sum_{n=1}^{n} \frac{n}{i} = O(n \log n)$$

### Full force split

Tentatively choosing the next cluster to split can be used with global split as well. However, since global split affects other clusters in addition to the target cluster, the acceleration method described in the previous section cannot be used with it. Likewise, tentatively choosing the target cluster by using local split could be used as well, but globally splitting the target cluster would again invalidate the cache used in the tentative split variant.

```
Full Force Split k-means
    Initialize cluster
    repeat
        Choose the cluster with the greatest SSE
        Globally split the cluster
    until found clusters = K
    Fine tune the result by k-means
end
```

**Figure 4.11:** Pseudo code of the full force split *k*-means algorithm

Using the tentative approach with any kind of global repartitioning or moving the centroids requires copying data structures between every split in order to restore the previous state. This introduces a substantial amount of unnecessary work.

Since the tentative split done with the global *k*-means is too heavy to process whether the tentative splitting is done locally or globally, the design choice was to choose the target cluster non-tentatively. The algorithm simply chooses the cluster that has the greatest SSE value.

The actual split is done globally by removing a centroid from the target cluster, choosing two new data points in that cluster randomly and finally running two global *k*-means iterations. The pseudo code of the algorithm is presented in Figure 4.11.

The time complexity of the full force variant depends on the number of the global *k*-means iterations ($g$):

$$kgnk = O(gnk^2)$$

**Hybrid split**

When comparing the early stage results of tentatively splitting clusters by using the local split and global split, an idea of a hybrid version came out. Can the worst excesses of local split be filtered out with a couple global *k*-means iterations for intermediate results with the algorithm still performing better than tentatively splitting with the global split?

The main idea of hybrid split is to combine the best features from the tentative split and full force split variants. The initial idea was to modify the tentative split variant to include a global fine-tuning of intermediate results between every split. A good clustering result was achieved with a couple of global *k*-means iterations. However,

```
Hybrid Split k-means
    Initialize cluster
    repeat
        Tentatively select cluster that decreases SSE the most
        Locally split cluster
        Fine tune the intermediate result by local repartition
    until found clusters = K
    Fine tune the result by k-means
end
```

**Figure 4.12:** Pseudo code of the hybrid split *k*-means algorithm

since the tentative variant keeps the SSE differences of previously tried splits in the cache, and the global *k*-means moved every single centroid in many cases, this way completely invalidated the cached SSE values.

The next design idea was to find a way to perform repartition only for the neighbour clusters of the target cluster that was split. The final design solution was to perform a *local repartition* only and the idea for that came from Fränti and Kivijärvi (2000).

The local repartition algorithm is presented in Figure 3.5 and the process is demonstrated in Figure 4.13. The local repartition method is the same as in the random swap algorithm for speeding up the process and in the iterative splitting algorithm. When used with the split *k*-means algorithm, it does not affect other than the surrounding clusters of the split cluster. Most of the cached SSE results are still valid on the next iteration and only the newly created clusters and their neighbour clusters have to be tentatively split on the next iteration.

The pseudo code of the hybrid split algorithm is presented in Figure 4.12. The implementation of local repartition is followed by an invalidation of neighbour cluster entries in the cache used by the tentative cluster selection.

Local repartition increases the time complexity of the algorithm. According to Fränti et al. (1997) the local repartitioning has a time complexity of $O(NMK)$, where $M$ is the number of iterations.

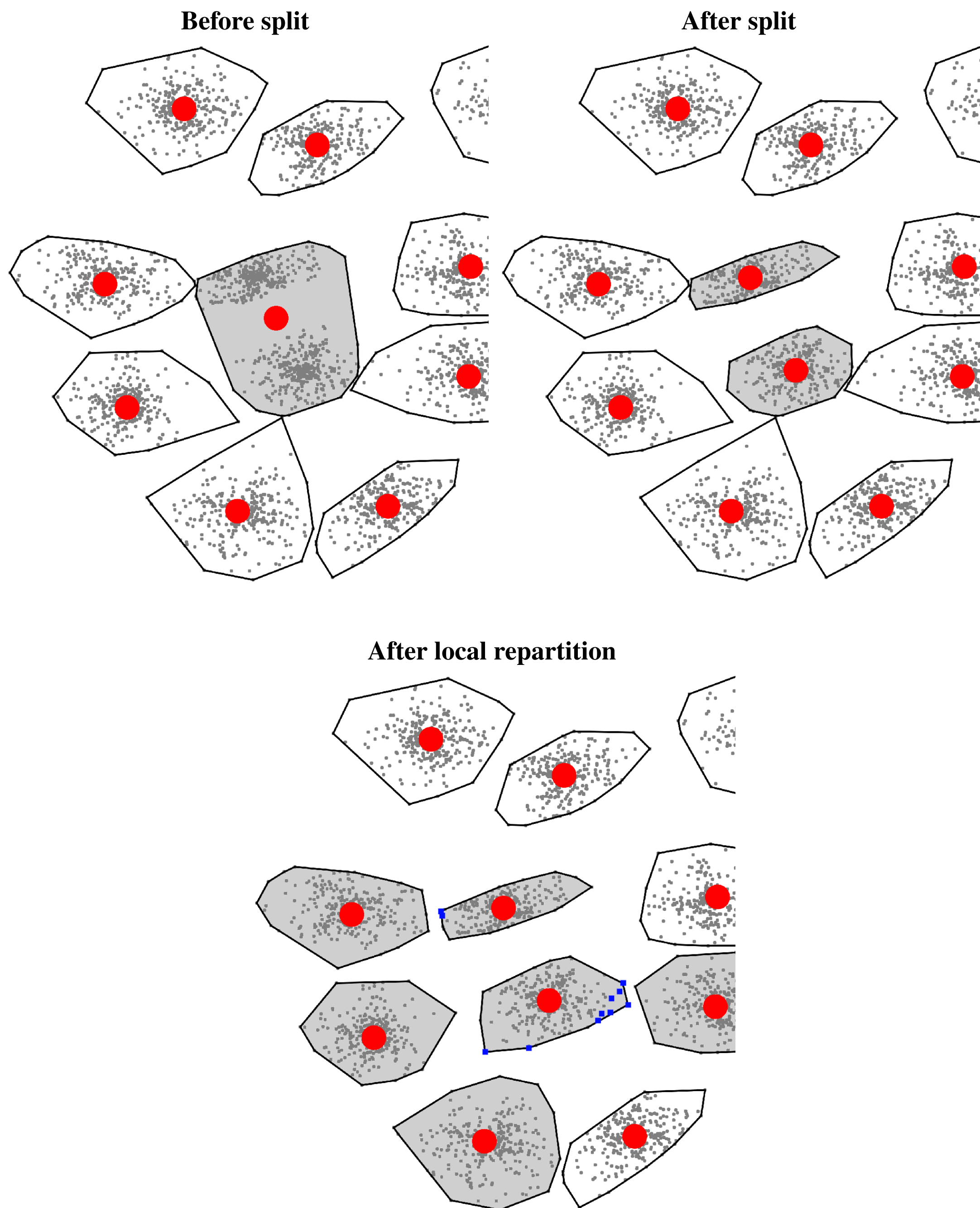



**Figure 4.13:** Hybrid split $k$-means performs local repartition for intermediate results. First it splits the target cluster and after the split the clusters are repartitioned against their original centroid and the new centroids. Affected clusters are highlighted in gray. In the last picture some data points have moved from surrounding clusters to the new clusters. These individual data points are highlighted with blue.

# 5. Experiments

The purpose of experiments was to find out if split $k$-means provides better clustering quality and speed than naive $k$-means, bisecting $k$-means or random swap. Also, the three variants of split $k$-means were compared with each other and with the iterative splitting algorithm. When comparing the results of split $k$-means with the results of the iterative splitting algorithm, the idea was to see if split $k$-means can achieve similar or better results than the iterative splitting algorithm that uses PCA for splitting the clusters.

## 5.1 Implementations

The split $k$-means algorithms were originally implemented in Python 3. Since the performance of Python quickly became a bottleneck when processing large amounts of data, the algorithm was rewritten by using NumPy Python library. NumPy is an open source package that provides pre-compiled functions for mathematical operations and data structures for replacing, for example, the ordinary Python lists (Marowka, 2018). However, in addition to the implementation that uses NumPy, another implementation was written that uses the scikit-learn (Pedregosa et al., 2011) $k$-means implementation. In the scikit-learn implementation of the algorithms, all calls to both global and intra-cluster $k$-means were replaced with scikit-learn package's implementation of the $k$-means algorithm. All experiments were run on a single core.

The experiments contain other algorithms as well. The random swap and the iterative splitting algorithm implementations are written in C. They can be found publicly on https://cs.uef.fi/ml/software/ in the CBModules package. However, when comparing the results of the NumPy version of the split $k$-means with random swap, the random swap implementation was implemented with NumPy as well. Similarly, when comparing the results of the scikit-learn version of the split $k$-means with random swap, the random swap implementation was implemented with scikit-learn.

The naive $k$-means and bisecting $k$-means implementations are written either with Numpy or provided by the scikit-learn package (Pedregosa et al., 2011).

**Table 5.1:** Run times (ms) of the random swap implementations. All implementations reach CI = 0 clustering quality on all datasets and all implementations were run with the recommended 5 000 iterations.

| Dataset | Random swap CBModules | Random swap NumPy | Random swap scikit-learn |
|---|---|---|---|
| S1 | 2986 | 9467 | 9429 |
| S2 | 3612 | 9355 | 9479 |
| S3 | 4154 | 9329 | 9487 |
| S4 | 4583 | 9303 | 9454 |
| A1 | 1768 | 7238 | 7324 |
| A2 | 3318 | 12935 | 12921 |
| A3 | 4991 | 19432 | 19356 |
| Unb | 9821 | 9787 | 9909 |
| Dim32 | 1700 | 6198 | 6217 |
| Birch1 | 159842 | 378966 | 395324 |
| Birch2 | 55071 | 397190 | 394083 |
| **Average** | **22895** | **79018** | **80271** |

## 5.2 Data

The data used in experiments consists of several datasets in the clustering basic benchmark dataset (Fränti & Sieranoja, 2018). All used datasets can be seen in Figure 5.1.

Each S set contain 15 Gaussian clusters and their overlap varies between S1 and S4 from 9% to 44%. In S1, the clusters are clearly separated and in S4 they overlap heavily, while still being clearly visible.

The overlap is estimated by calculating the distance of a point to its closest centroid ($d1$) and to the closest point in another cluster ($d2$). If the distance to the closest point in another cluster is smaller than the distance to the closest centroid ($d2 < d1$), then the point is considered as overlapped (Fränti & Sieranoja, 2018).

$$\text{overlapped}(d_1, d_2) = \begin{cases} 1, & d_1 > d_2 \\ 0, & \text{otherwise} \end{cases}$$

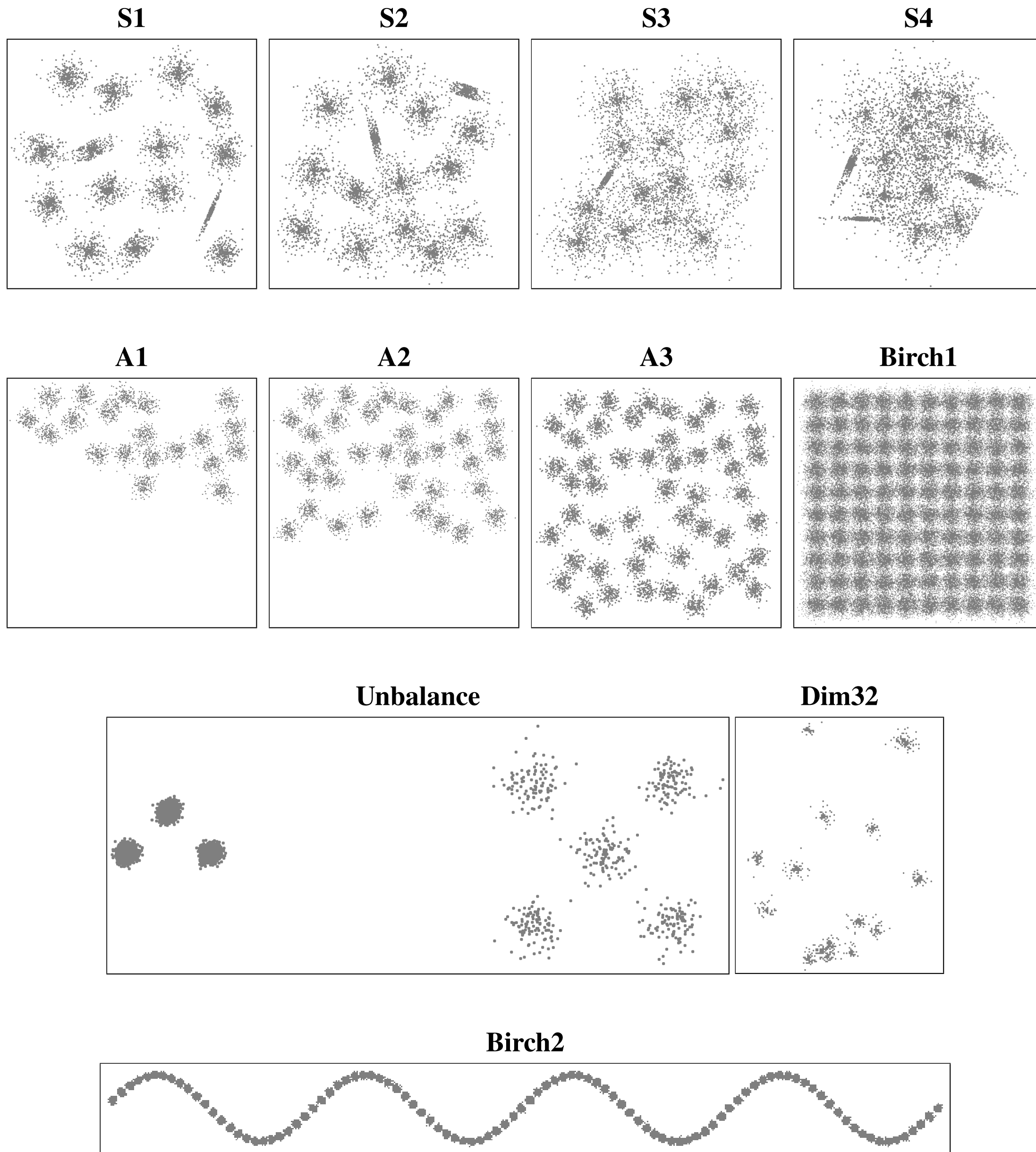



**Figure 5.1:** Clustering basic benchmark datasets used in the experiments (Fränti & Sieranoja, 2018). Since Dim32 dataset contains 32 dimensions, only the first two dimensions are plotted.

Fränti and Sieranoja (2018) define the overall overlap as the overlapped points in relation to the total number of data points:

$$\text{overlap} = \frac{1}{N} \sum \text{overlapped}(d_1, d_2)$$

A sets consist of spherical clusters, which are subsets ($A1 \subset A2 \subset A3$) of each other. A1 dataset contains 20 clusters, A2 contains 35 clusters and A3 contains 50 clusters.

Unbalance is a dataset of eight clusters that are grouped into two well-separated groups. The left-hand group consists of three dense clusters and the right-hand group consists of five sparse clusters.

Birch datasets are larger ones. Birch1 consists of clusters that form a $10 \times 10$ grid. Birch2 dataset contains clusters whose ground truth centroids form a sine curve (Fränti & Sieranoja, 2018). Both Birch datasets contain 100 clusters.

## 5.3 Evaluation

When comparing the clustering results, internal and external measurements were used. Internal measurements depend only on the data points themselves. Respectively, external measures use the ground truth as well. The time spent on clustering is shown in milliseconds and it is measured from the start of running the algorithm to the completion of the final result.

### Internal measures

The internal measures for clustering quality are SSE and MSE. Both methods were introduced in Section 2.2. Since the objective function of *k*-means and split *k*-means as well is SSE, the most natural way to compare them is to compare the SSE values. MSE is the normalized version of the SSE value and it is included in results as well.

Because the datasets are not normalized in any way before clustering, the SSE and MSE values vary greatly between them. The datasets are used in the experiments as they are and the resulting MSE and SSE values are scaled in the results according to the following coefficients.

**S1, S2, S3, S4, Birch1** MSE = MSE $\times 10^{-8}$, SSE = SSE $\times 10^{-12}$

**A1, A2, A3, Unbalance, Birch2** MSE = MSE $\times 10^{-6}$, SSE = SSE $\times 10^{-10}$

**Dim32** MSE = MSE, SSE = SSE $\times 10^{-4}$

### External measures

An external method for measuring the clustering structure is CI which was introduced in Section 2.3. Since the ground truth for all datasets was available, CI values were calculated for all results. When evaluating the CI, a value of 0 means a correct clustering structure and a value of 1 means that one centroid is incorrectly located. Respectively, higher values mean that more centroids are incorrectly located.

In the results, CI values are averaged over the test runs. The CI values are the primary way to see the quality of the clustering structure. However, in order for them to be comparable between different datasets, a relative CI is calculated as well. A relative CI (rel-CI) is calculated with the formula rel-CI = CI / $k$.

When the CI is known, a success rate can be calculated as well. The success rate tells how often the algorithm successfully achieves a CI = 0 value. For example, if the algorithm is run 1 000 times of which 980 times it manages to reach the CI value of 0, then the success rate is 98%. While success rate measures the performance of the algorithm, it does not tell anything about the quality of clustering when it does not reach the perfect structure. An algorithm might be good even if it does not manage to reach the correct clustering often since perfect clustering results may not be necessary in all situations.

## 5.4 Results

All implementations of the three variants of the split $k$-means, two implementations of naive $k$-means and bisecting $k$-means, three implementations of random swap and three different variants of the iterative splitting algorithm were run 100 times over the S1, S2, S3, S4, A1, A2, A3, Unbalance, Dim32, Birch1 and Birch2 datasets. The results contain comparison between three split $k$-means variations in Tables 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7, and time and quality comparisons between $k$-means, bisecting $k$-means, split $k$-means and random swap in Tables 5.10. 5.11, 5.12 and 5.13. Also, Tables 5.8 and 5.9 contain the results of the SPLIT, SLR and SGLA variants of the iterative splitting algorithm for reference.

**Table 5.2:** Summary of the tentative split (NumPy) implementation results averaged over 100 runs

| Dataset | Success | MSE | SSE | Time (ms) | CI | rel-CI |
|---|---|---|---|---|---|---|
| S1 | 97% | 9.06 | 9.06 | 112 | 0.03 | 0.2% |
| S2 | 100% | 13.28 | 13.28 | 136 | 0.00 | 0.0% |
| S3 | 75% | 17.39 | 17.39 | 213 | 0.25 | 1.7% |
| S4 | 100% | 15.71 | 15.71 | 206 | 0.00 | 0.0% |
| A1 | 29% | 2.26 | 1.35 | 127 | 0.71 | 3.6% |
| A2 | 83% | 1.97 | 2.07 | 283 | 0.17 | 0.5% |
| A3 | 80% | 1.96 | 2.94 | 466 | 0.20 | 0.4% |
| Unb | 100% | 16.50 | 21.45 | 73 | 0.00 | 0.0% |
| Dim32 | 99% | 8.16 | 26.72 | 35 | 0.01 | 0.1% |
| Birch1 | 14% | 4.77 | 95.44 | 27254 | 1.12 | 1.1% |
| Birch2 | 100% | 2.28 | 45.67 | 7654 | 0.00 | 0.0% |

**Table 5.3:** Summary of the tentative split (scikit-learn) implementation results averaged over 100 runs

| Dataset | Success | MSE | SSE | Time (ms) | CI | rel-CI |
|---|---|---|---|---|---|---|
| S1 | 98% | 9.01 | 9.01 | 54 | 0.02 | 0.1% |
| S2 | 100% | 13.28 | 13.28 | 56 | 0.00 | 0.0% |
| S3 | 80% | 17.28 | 17.28 | 60 | 0.20 | 1.3% |
| S4 | 78% | 15.96 | 15.96 | 60 | 0.22 | 1.5% |
| A1 | 45% | 2.21 | 1.32 | 63 | 0.55 | 2.8% |
| A2 | 86% | 1.96 | 2.06 | 126 | 0.14 | 0.4% |
| A3 | 71% | 1.98 | 2.97 | 197 | 0.36 | 0.7% |
| Unb | 100% | 16.50 | 21.45 | 31 | 0.00 | 0.0% |
| Dim32 | 100% | 7.10 | 23.25 | 43 | 0.00 | 0.0% |
| Birch1 | 8% | 4.82 | 96.33 | 2561 | 1.47 | 1.5% |
| Birch2 | 95% | 2.29 | 45.88 | 1875 | 0.05 | 0.1% |

**Table 5.4:** Summary of the hybrid split (NumPy) implementation results averaged over 100 runs

| Dataset | Success | MSE | SSE | Time (ms) | CI | rel-CI |
|---|---|---|---|---|---|---|
| S1 | 100% | 8.92 | 8.92 | 228 | 0.00 | 0.0% |
| S2 | 100% | 13.28 | 13.28 | 297 | 0.00 | 0.0% |
| S3 | 100% | 16.89 | 16.89 | 331 | 0.00 | 0.0% |
| S4 | 100% | 15.71 | 15.71 | 374 | 0.00 | 0.0% |
| A1 | 100% | 2.02 | 1.21 | 222 | 0.00 | 0.0% |
| A2 | 100% | 1.93 | 2.03 | 553 | 0.00 | 0.0% |
| A3 | 99% | 1.93 | 2.90 | 1027 | 0.01 | $< 0.1\%$ |
| Unb | 100% | 16.50 | 21.45 | 89 | 0.00 | 0.0% |
| Dim32 | 100% | 7.10 | 23.25 | 50 | 0.00 | 0.0% |
| Birch1 | 99% | 4.64 | 92.80 | 44161 | 0.01 | $< 0.1$ |
| Birch2 | 100% | 2.28 | 45.67 | 15538 | 0.00 | 0.0% |

**Table 5.5:** Summary of the hybrid split (scikit-learn) implementation results averaged over 100 runs

| Dataset | Success | MSE | SSE | Time (ms) | CI | rel-CI |
|---|---|---|---|---|---|---|
| S1 | 100% | 8.92 | 8.92 | 115 | 0.00 | 0.0% |
| S2 | 100% | 13.28 | 13.28 | 128 | 0.00 | 0.0% |
| S3 | 100% | 16.89 | 16.89 | 130 | 0.00 | 0.0% |
| S4 | 100% | 15.71 | 15.71 | 138 | 0.00 | 0.0% |
| A1 | 99% | 2.03 | 1.22 | 128 | 0.01 | < 0.1% |
| A2 | 100% | 1.93 | 2.03 | 312 | 0.00 | 0.0% |
| A3 | 97% | 1.93 | 2.90 | 569 | 0.03 | < 0.1% |
| Unb | 100% | 16.50 | 21.45 | 46 | 0.00 | 0.0% |
| Dim32 | 99% | 8.16 | 26.72 | 58 | 0.01 | < 0.1% |
| Birch1 | 99% | 4.64 | 92.80 | 10742 | 0.01 | < 0.1% |
| Birch2 | 100% | 2.28 | 45.67 | 7107 | 0.00 | 0.0% |

**Table 5.6:** Summary of the full force split (NumPy) implementation results averaged over 100 runs

| Dataset | Success | MSE | SSE | Time (ms) | CI | rel-CI |
|---|---|---|---|---|---|---|
| S1 | 90% | 9.35 | 9.35 | 67 | 0.10 | 0.6% |
| S2 | 84% | 13.76 | 13.76 | 83 | 0.16 | 1.1% |
| S3 | 84% | 17.20 | 17.20 | 93 | 0.16 | 1.1% |
| S4 | 67% | 16.02 | 16.02 | 128 | 0.33 | 2.2% |
| A1 | 80% | 2.10 | 1.26 | 73 | 0.20 | 1.0% |
| A2 | 86% | 1.96 | 2.06 | 319 | 0.15 | 0.4% |
| A3 | 79% | 1.96 | 2.94 | 815 | 0.21 | 0.4% |
| Unb | 96% | 17.25 | 22.42 | 26 | 0.04 | 0.5% |
| Dim32 | 38% | 121.00 | 396.47 | 37 | 1.05 | 6.6% |
| Birch1 | 77% | 4.67 | 93.42 | 40383 | 0.27 | 0.3% |
| Birch2 | 100% | 2.28 | 45.67 | 37201 | 0.00 | 0.0% |

**Table 5.7:** Summary of the full force split (scikit-learn) implementation results averaged over 100 runs

| Dataset | Success | MSE | SSE | Time (ms) | CI | rel-CI |
|---|---|---|---|---|---|---|
| S1 | 93% | 9.23 | 9.23 | 30 | 0.07 | 0.5% |
| S2 | 93% | 13.48 | 13.48 | 27 | 0.07 | 0.5% |
| S3 | 95% | 16.99 | 16.99 | 29 | 0.05 | 0.3% |
| S4 | 76% | 15.96 | 15.96 | 29 | 0.24 | 1.6% |
| A1 | 97% | 2.03 | 1.22 | 28 | 0.03 | 0.2% |
| A2 | 91% | 1.95 | 2.05 | 77 | 0.09 | 0.3% |
| A3 | 90% | 1.94 | 2.91 | 180 | 0.10 | 0.2% |
| Unb | 94% | 1.76 | 22.85 | 16 | 0.06 | 0.8% |
| Dim32 | 28% | 130.21 | 426.66 | 20 | 1.15 | 7.2% |
| Birch1 | 90% | 4.65 | 93.04 | 4926 | 0.11 | 0.1% |
| Birch2 | 97% | 2.29 | 45.79 | 4807 | 0.03 | < 0.1% |

### Split *k*-means variants

In general, the hybrid version seemed to be the most successful when comparing the clustering quality. This can be seen by examining the CI and SSE values.

The hybrid algorithm managed to correctly cluster all datasets most of the time. However, the tentative split's scikit-learn implementation performed worse with the more overlapped S3 and S4 sets. S datasets contain Gaussian clusters and the overlap varies between them. S1 is the least overlapped dataset and S4 is the most overlapped one. Similarly, the full force variant struggled with the S4 dataset as seen in Tables 5.6 and 5.7.

Unlike the *k*-means algorithm, when looking at the Tables 5.6 and 5.7, full force split *k*-means seems to perform worse the more the dataset is overlapped. In Table 5.2 can be seen that the trend is the same even though the scikit-learn implementation of the tentative split *k*-means manages to correctly split S4 more times than S3. The result may be explained with the divisive nature of the split *k*-means algorithm. With clearly separated clusters, it manages to place the centroids more often in separate clusters while naive *k*-means struggles to move centroids to the correct place when the clusters have no overlap.

Since A sets are subsets of each other and contain no other variation except the size of the dataset, the quality results between the different variations of split *k*-means should have no other clear differences except the clustering time. This holds true for hybrid split and full force split variants, but the tentative split variant has a serious performance issue with the A1 dataset of which it manages to cluster correctly only 29% of the time when using the NumPy implementation and 45% when using the scikit-learn implementation.

This particular issue is demonstrated in Figure 5.2. Since the tentative split variant does not perform any fine-tuning of intermediate results, choosing the split that yields the greatest SSE difference is not always the best choice. With A1 dataset, the upper left density peak is often improperly clustered which is seen in the results in Table 5.2.

A closer look into the issue in Figure 5.3 shows that one cluster is often split right in the middle. After the fourth local split, a cluster with two clear density peaks has appeared. According to the end result, this cluster decreases the overall SSE more than the correct split.

According to Fränti and Sieranoja (2018), one weakness of *k*-means is demonstrated by the Unbalance dataset. When initializing the initial centroids for the naive *k*-means algorithm, if no centroids are chosen in the area of small clusters, *k*-means fails to move

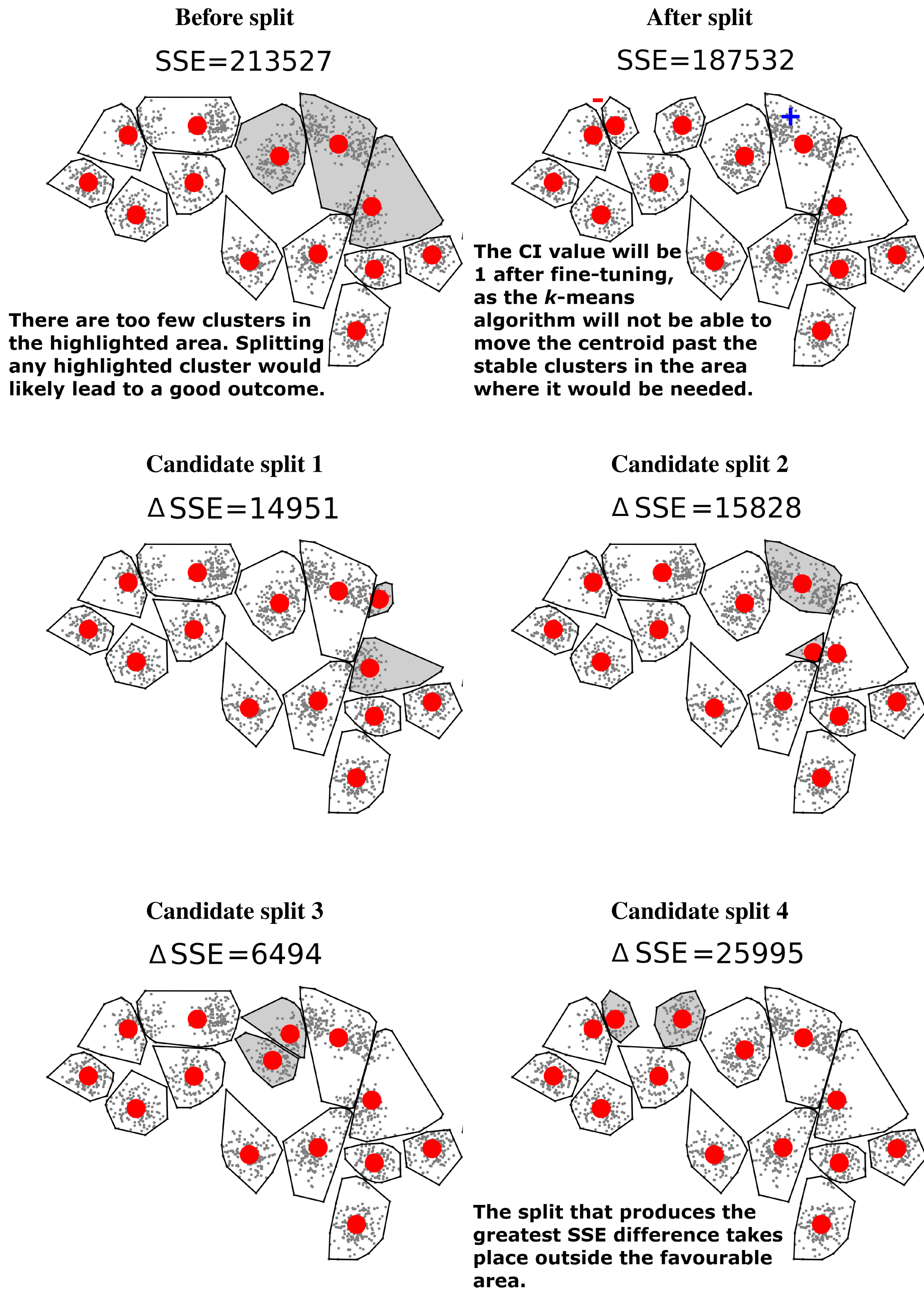



**Figure 5.2:** Tentative split variant splits the wrong cluster in A1 dataset. In this case choosing the split with the greatest SSE difference does not yield the best outcome when measuring the quality of the clustering structure.

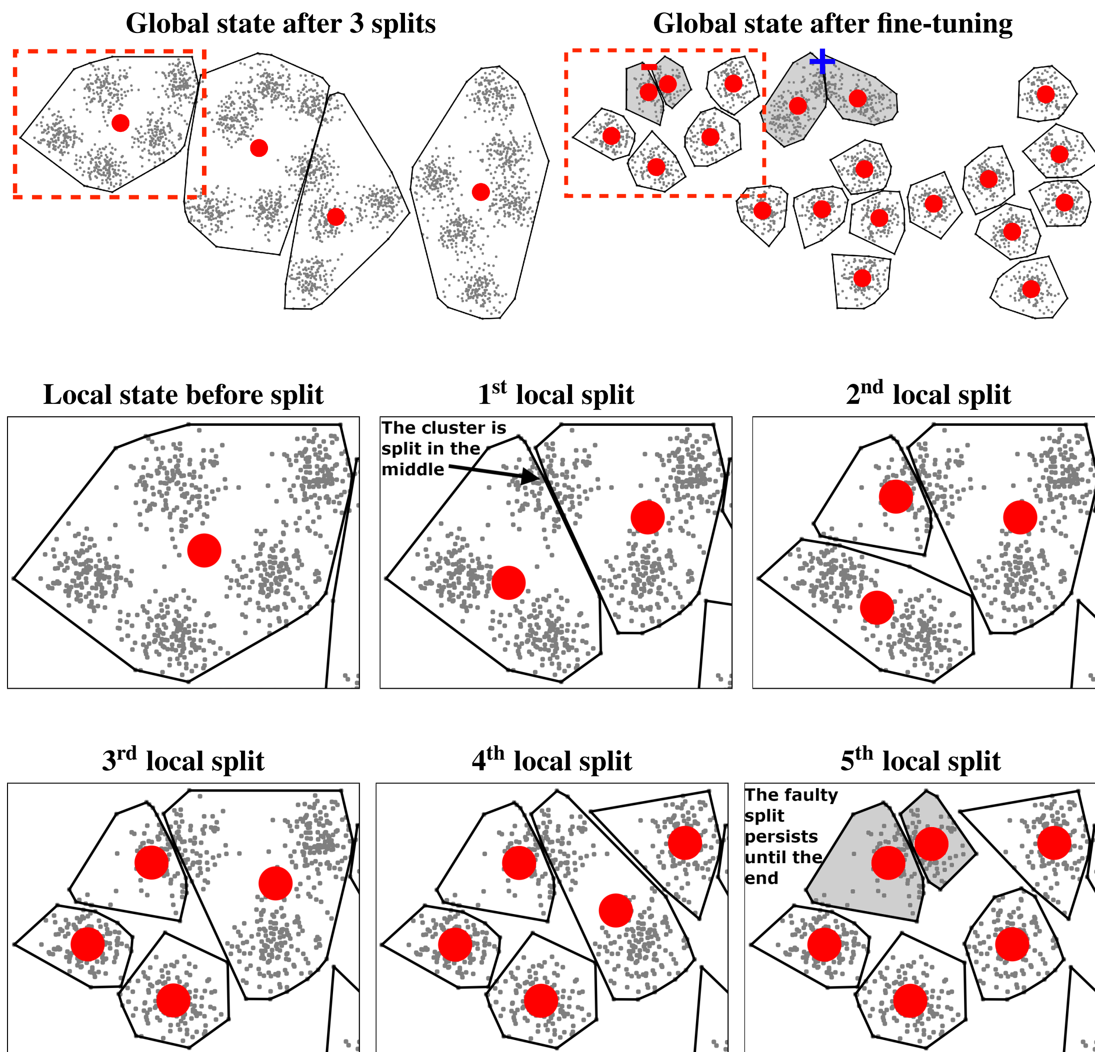



**Figure 5.3:** A closer look to the A1 dataset shows the weakness that affects the tentative split variant. If a cluster is split in the middle, then both sides might get their own centroid. Since tentative split does not perform any fine-tuning for intermediate results, the error persists until the end.

the centroids into this area and the final result will be incorrect. According to the results in Tables 5.2 and 5.4, tentative or hybrid variations succeed in splitting the clusters correctly every time. Even the full force variation that does not perform any tentative splits succeeds in this task in over 94% of the cases as seen in Tables 5.6 and 5.7.

The only dataset that had over two dimensions was Dim32 with 32 dimensions. The clusters in Dim32 are well separated as in the A sets as well. While the global splitting seems to have trouble finding the correct clustering, the tentative approach seems to work very well for high dimensional data. However, the CI value of the full force variant was only 1 on average as seen on Table 5.6 and 5.7.

Birch1 is the most difficult dataset for the tentative split variant since the algorithm man-

ages to cluster it correctly only 14% of the time when using the NumPy implementation and only 8% when using the scikit-learn implementation. The averaged CI values are 1.12 and 1.47. Based on the empirical examination, it usually varies between 1 and 2 between individual executions. A typical clustering result can be seen in Figure 5.4.

Similarly to the results by Fränti et al. (1997) with the iterative splitting algorithm, the results of the split *k*-means follow the same trend. The nearest corresponding algorithms for split *k*-means variants are S+GLA, SLR+GLA and SGLA. S+GLA is similar to the tentative split variant since it does not perform any refinement for intermediate results. The variant that performs it is SLR+GLA. It performs the same local repartition as hybrid split. The heavyweight equivalent to full force split is SGLA that performs GLA iterations for intermediate results. When looking at the averages of execution times in Table 5.12, it can be seen that full force split takes the most time. Tentative split is the fastest and hybrid split is the second fastest.

However, the quality of clustering does not follow the same order of the results by Fränti et al. (1997). The hybrid split variant is capable of the best quality on all meters when looking at Tables 5.4, 5.5, 5.10 and 5.11.

#### Algorithm benchmark

**Table 5.8:** Iterative splitting algorithm CI values averaged over 100 runs

| Dataset | SPLIT | SLR | SGLA |
|---|---|---|---|
| S1 | 0.00 | 0.00 | 0.00 |
| S2 | 0.00 | 0.00 | 0.00 |
| S3 | 1.00 | 0.00 | 0.00 |
| S4 | 1.00 | 0.00 | 0.00 |
| A1 | 1.00 | 0.00 | 0.00 |
| A2 | 0.00 | 0.00 | 0.00 |
| A3 | 1.00 | 0.00 | 0.00 |
| Unb | 0.00 | 0.00 | 0.00 |
| Dim32 | 0.00 | 0.00 | 0.00 |
| Birch1 | 63.00 | 0.00 | 0.00 |
| Birch2 | 49.00 | 0.00 | 0.00 |
| **Average** | **10.55** | **0.00** | **0.00** |

In general, every split *k*-means variant can cluster all datasets in better than CI = 1 quality. The only exceptions for this are Dim32 dataset with full force split and Birch1 dataset with tentative split. When comparing tentative split with bisecting *k*-means, the trends of their performance are very similar. The biggest challenges occur with the same datasets and similarly both of the algorithms seem to work well with the same

**Table 5.9:** Iterative splitting algorithm run times (ms) averaged over 100 runs

| Dataset | SPLIT | SLR | SGLA |
|---|---|---|---|
| S1 | 50 | 60 | 69 |
| S2 | 50 | 55 | 68 |
| S3 | 44 | 55 | 69 |
| S4 | 53 | 59 | 67 |
| A1 | 51 | 55 | 58 |
| A2 | 53 | 64 | 86 |
| A3 | 49 | 76 | 146 |
| Unb | 48 | 54 | 64 |
| Dim32 | 42 | 58 | 71 |
| Birch1 | 247 | 846 | 6451 |
| Birch2 | 228 | 522 | 4768 |
| **Average** | **83.18** | **173.09** | **1083.36** |

**Table 5.10:** CI-values averaged over 100 runs (NumPy)

| Dataset | *k*-means | bisecting *k*-means | tentative split | hybrid split | full force split | random swap (NumPy) |
|---|---|---|---|---|---|---|
| S1 | 1.87 | 0.00 | 0.03 | 0.00 | 0.10 | 0.00 |
| S2 | 1.44 | 0.00 | 0.00 | 0.00 | 0.16 | 0.00 |
| S3 | 1.19 | 1.00 | 0.25 | 0.00 | 0.16 | 0.00 |
| S4 | 0.87 | 0.00 | 0.00 | 0.00 | 0.33 | 0.00 |
| A1 | 2.34 | 0.14 | 0.71 | 0.00 | 0.20 | 0.00 |
| A2 | 4.58 | 0.00 | 0.17 | 0.00 | 0.15 | 0.00 |
| A3 | 6.58 | 0.03 | 0.20 | 0.01 | 0.21 | 0.00 |
| Unb | 3.87 | 0.00 | 0.00 | 0.00 | 0.04 | 0.00 |
| Dim32 | 3.50 | 0.00 | 0.01 | 0.00 | 1.05 | 0.00 |
| Birch1 | 6.64 | 4.66 | 1.12 | 0.01 | 0.27 | 0.00 |
| Birch2 | 16.90 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **Average** | **4.53** | **0.53** | **0.23** | **0.00** | **0.24** | **0.00** |

datasets. The biggest challenge for both algorithms is the Birch1 dataset whereby the CI value increases over 1.12 with tentative split and over 4.6 with bisecting *k*-means.

As seen in Table 5.10, random swap successfully manages to cluster every dataset every time. The cost of the good quality is the run time of the algorithm. Compared with the hybrid split *k*-means, the run time of random swap is usually 5-20 times longer. With only one exception with A3 dataset, hybrid split manages to keep the CI value under 0.01.

Figure 5.4 presents the typical clustering results of the algorithms. The issue with *k*-means is that it finds the local optimum but it will not be able to move the centroids to correct locations if there are stable centroids in between. Bisecting *k*-means does not perform any fine-tuning for the bisected clusters and for that reason the clusters

**Table 5.11:** CI-values averaged over 100 runs (scikit-learn)

| Dataset | $k$-means | bisecting $k$-means | tentative split | hybrid split | full force split | random swap (scikit-learn) |
|---|---|---|---|---|---|---|
| S1 | 1.94 | 0.02 | 0.02 | 0.00 | 0.07 | 0.00 |
| S2 | 1.46 | 0.00 | 0.00 | 0.00 | 0.07 | 0.00 |
| S3 | 1.31 | 1.00 | 0.20 | 0.00 | 0.05 | 0.00 |
| S4 | 0.93 | 0.00 | 0.22 | 0.00 | 0.24 | 0.00 |
| A1 | 2.58 | 0.50 | 0.55 | 0.01 | 0.03 | 0.00 |
| A2 | 4.71 | 0.06 | 0.14 | 0.00 | 0.09 | 0.00 |
| A3 | 6.52 | 0.11 | 0.36 | 0.03 | 0.10 | 0.00 |
| Unb | 3.87 | 0.00 | 0.00 | 0.0 | 0.06 | 0.00 |
| Dim32 | 3.28 | 0.00 | 0.00 | 0.01 | 1.15 | 0.00 |
| Birch1 | 7.36 | 7.54 | 1.47 | 0.01 | 0.11 | 0.00 |
| Birch2 | 17.62 | 0.01 | 0.05 | 0.00 | 0.03 | 0.00 |
| **Average** | **4.69** | **0.84** | **0.27** | **0.01** | **0.18** | **0.00** |

**Table 5.12:** Run times (ms) averaged over 100 runs (NumPy)

| Dataset | $k$-means | bisecting $k$-means | tentative split | hybrid split | full force split | random swap (NumPy) |
|---|---|---|---|---|---|---|
| S1 | 72 | 123 | 112 | 228 | 67 | 9467 |
| S2 | 84 | 169 | 136 | 297 | 83 | 9355 |
| S3 | 98 | 189 | 213 | 331 | 93 | 9329 |
| S4 | 157 | 207 | 206 | 374 | 128 | 9303 |
| A1 | 66 | 132 | 127 | 222 | 73 | 7238 |
| A2 | 202 | 275 | 283 | 553 | 319 | 12935 |
| A3 | 400 | 451 | 466 | 1027 | 815 | 19432 |
| Unb | 64 | 52 | 73 | 89 | 26 | 9787 |
| Dim32 | 9 | 39 | 35 | 50 | 37 | 6198 |
| Birch1 | 41824 | 15637 | 27254 | 44161 | 40383 | 378966 |
| Birch2 | 16394 | 7267 | 7654 | 15538 | 37201 | 404124 |
| **Average** | **5397** | **2231** | **3324** | **5715** | **7202** | **79649** |

look angular. Random swap, hybrid split and full force split have found the correct clustering, but tentative split has two centroids in incorrect locations.

The performance profiles of three split $k$-means variants are shown in Figures 5.5 and 5.6. In general, full force split seems to be the fastest variant when the value of $k$ is low. When the value increases, the full force variant gets significantly slower. For large datasets, tentative split is the fastest variant. Naturally, hybrid split is always slower than tentative split since it is basically the same algorithm with only local repartition added for intermediate results.

**Table 5.13:** Run times (ms) averaged over 100 runs (scikit-learn)

| Dataset | *k*-means | bisecting *k*-means | tentative split | hybrid split | full force split | random swap (scikit-learn) |
|---|---|---|---|---|---|---|
| S1 | 4 | 8 | 54 | 115 | 30 | 9429 |
| S2 | 4 | 9 | 56 | 128 | 27 | 9479 |
| S3 | 5 | 11 | 60 | 130 | 29 | 9487 |
| S4 | 7 | 12 | 60 | 138 | 29 | 9454 |
| A1 | 4 | 10 | 63 | 128 | 28 | 7324 |
| A2 | 9 | 18 | 126 | 312 | 77 | 12921 |
| A3 | 16 | 26 | 197 | 569 | 180 | 19356 |
| Unb | 2 | 5 | 31 | 46 | 16 | 9909 |
| Dim32 | 1 | 6 | 42 | 58 | 20 | 6217 |
| Birch1 | 881 | 237 | 2561 | 10742 | 4926 | 395324 |
| Birch2 | 235 | 144 | 1875 | 7107 | 4807 | 393228 |
| **Average** | **106** | **44** | **466** | **1770** | **924** | **80193** |

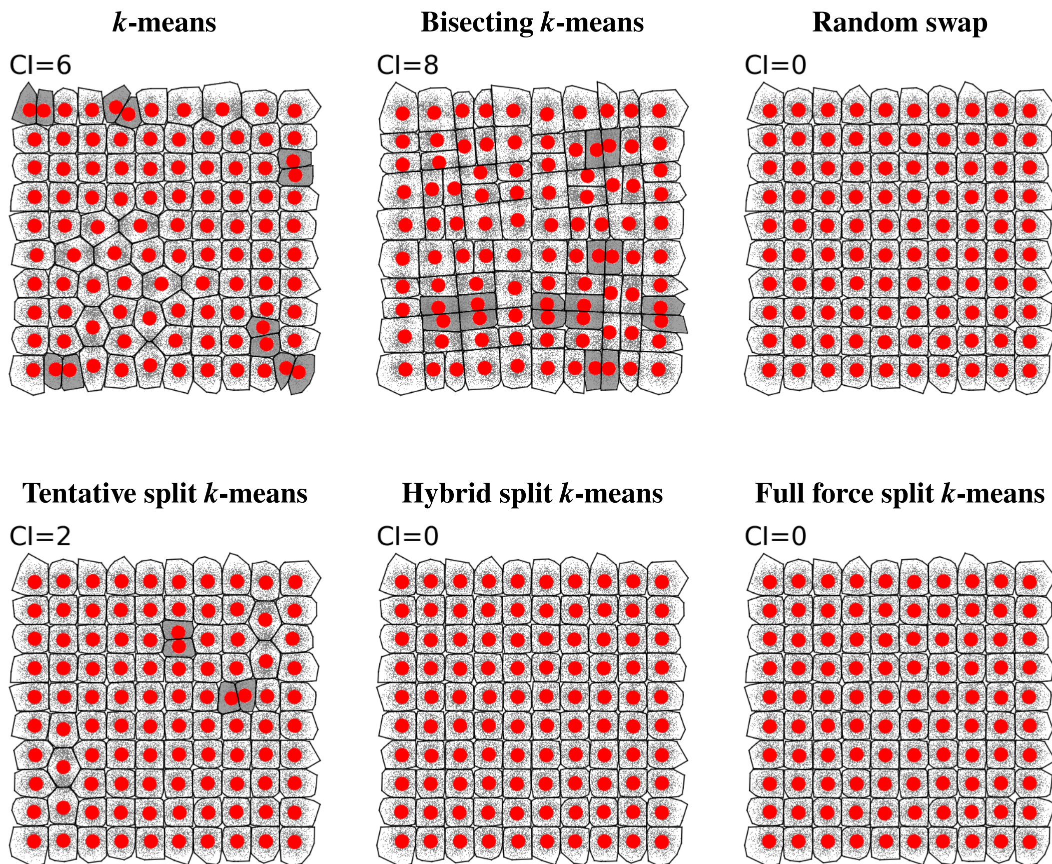



**Figure 5.4:** Birch1 dataset clustered with different algorithms. Bisecting *k*-means does not perform any fine-tuning. With proper fine-tuning the CI value would likely be lower than 8.

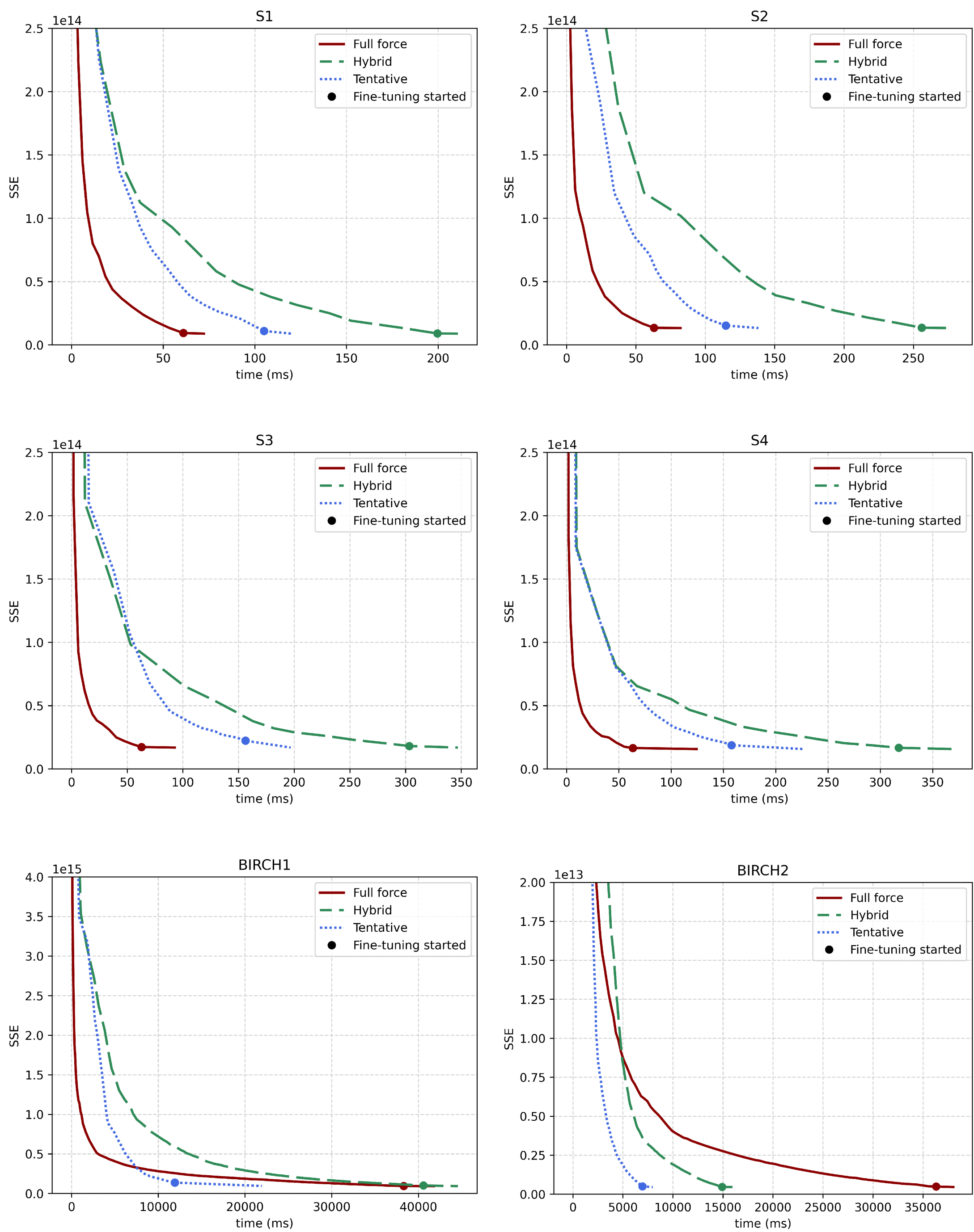



**Figure 5.5:** The full force split implementation is the fastest one when the value of $k$ is lower. It gets slower when $k$ increases. The S datasets all have 15 clusters, while Birch datasets have 100 clusters.

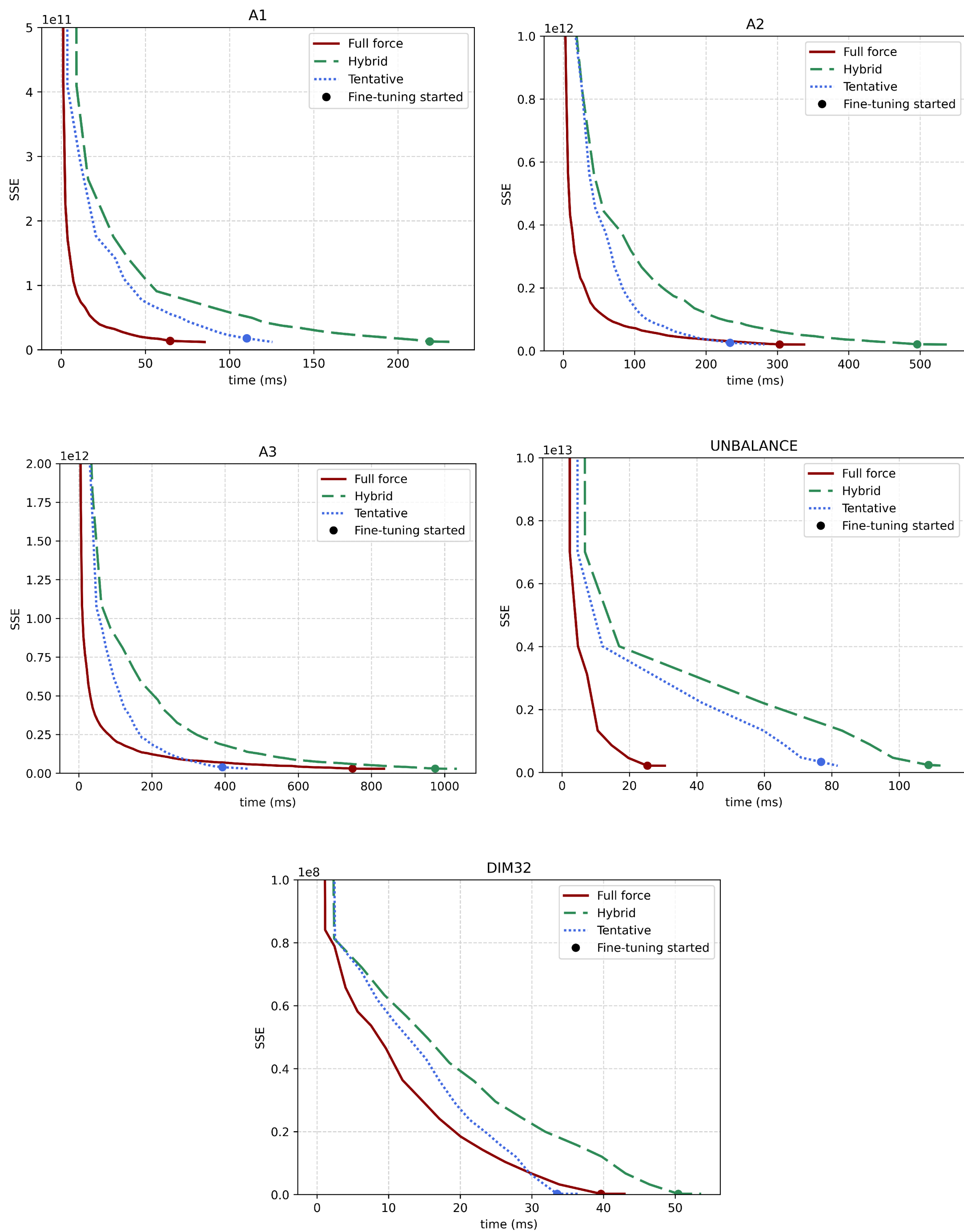



**Figure 5.6:** The same phenomenon, as in the case of S and Birch datasets, can be seen with A datasets as well. While the value of $k$ increases, full force split gets slower. Naturally hybrid split is always slower than the tentative split due to the fine-tuning of intermediate results.

# 6. Conclusions

In this thesis a new clustering algorithm and three variations of it were introduced. The theory of clustering, clustering algorithms, divisive clustering, *k*-means and *k*-means based algorithms were discussed, and they were used when designing the new algorithm.

A new split *k*-means algorithm was introduced and the details and design ideas were discussed. Since the algorithm had many moving parts, there was no single clear design to choose when designing the final algorithm. For this reason, the algorithm was implemented in three variations: *tentative split*, *hybrid split* and *full force split*. The basic idea was to solve the initial centroid position generation along with the clustering itself and for this reason the splitting was done until *k* clusters were found.

In addition, the idea was to develop the iterative splitting algorithm forward and explore the idea of not using PCA for splitting. Instead of PCA a simpler way to split a cluster was to just choose two random data points in it as the new centroids and split the cluster by running either intra-cluster *k*-means or global *k*-means.

The initial design idea was to tentatively split every cluster with *k*-means and always choose the split that produces the greatest decrease in the SSE value. The tentative split variation was based on this approach and it used intra-cluster *k*-means for splitting. Similarly, the second idea was to try global *k*-means for tentative splitting and build another algorithm variant using that. However, using *k*-means globally for splitting turned to be a computationally heavy task since it could not re-use old tentative split results. Also, a global repartition is generally heavier than an intra-cluster repartition. For this reason the full force split variant did not perform the tentative part. Instead, it split clusters globally and chose the cluster with the greatest SSE value.

The best ideas of both of these variations were combined in the hybrid split variation. It was designed to tentatively split clusters but to also fine-tune the intermediate results between split iterations. The original idea of running a couple of global *k*-means iterations for the intermediate results was abandoned since the global *k*-means tended to move all centroids instead of only the relevant ones. Moving non-relevant centroids

caused the algorithm to perform poorly since the split results from the previous iterations could not be re-used. This showed that it is better to implement a local repartition strategy where only the split cluster and its neighbour clusters were repartitioned.

The variations were implemented in Python 3 and they were compared against *k*-means, bisecting *k*-means and random swap clustering algorithms. All algorithms were run over 11 different datasets in the basic clustering benchmark and the results were analysed. The results showed that divisive clustering based on *k*-means can be used to achieve better clustering results than with the naive *k*-means algorithm or divisive bisecting *k*-means algorithm. The results also showed that it is difficult to outperform random swap clustering quality, but a good clustering result can be obtained in less time.

Future development ideas for split *k*-means could include implementing it in a more efficient programming language such as C or Fortran and parallelize it with modern technologies such as OpenMP or Open MPI. Also, the naive *k*-means could be replaced with fast *k*-means in order to speed up the algorithm. Research ideas for the future include choosing the cluster based on some other information than the differences between SSE values. Furthermore, better initial centroid positions for the random swap algorithm could perhaps be generated with the tentative split algorithm.

# References


Aloise, D., Deshpande, A., Hansen, P., & Popat, P. (2009). Np-hardness of euclidean sum-of-squares clustering. *Machine learning*, *75*(2), 245–248.

Chen, Y., Liu, W., Zhao, H., Cao, S., Fu, S., & Jiang, D. (2021). Bisecting k-means based fingerprint indoor localization. *Wireless Networks*, *27*, 3497–3506.

Dasgupta, S. (2008). *The hardness of k-means clustering*. Department of Computer Science; Engineering, University of California . . .

Drineas, P., Frieze, A., Kannan, R., Vempala, S., & Vinay, V. (2004). Clustering large graphs via the singular value decomposition. *Machine learning*, *56*(1-3), 9–33.

Estivill-Castro, V. (2002). Why so many clustering algorithms: A position paper. *SIGKDD Explor. Newsl.*, *4*(1), 65–75. https://doi.org/10.1145/568574.568575

Fahad, A., Alshatri, N., Tari, Z., Alamri, A., Khalil, I., Zomaya, A. Y., Foufou, S., & Bouras, A. (2014). A survey of clustering algorithms for big data: Taxonomy and empirical analysis. *IEEE transactions on emerging topics in computing*, *2*(3), 267–279.

Fränti, P., & Kivijärvi, J. (2000). Randomised local search algorithm for the clustering problem. *Pattern analysis and applications : PAA*, *3*(4), 358–369.

Fränti, P. (2018). Efficiency of random swap clustering. *Journal of big data*, *5*(1), 1–29.

Fränti, P., Kaukoranta, T., & Nevalainen, O. (1997). On the splitting method for vector quantization codebook generation. *Optical Engineering*, *36*(11), 3043–3051. https://doi.org/10.1117/1.601531

Fränti, P., Rezaei, M., & Zhao, Q. (2014). Centroid index: Cluster level similarity measure. *Pattern Recognition*, *47*(9), 3034–3045. https://doi.org/https://doi.org/10.1016/j.patcog.2014.03.017

Fränti, P., & Sieranoja, S. (2018). K-means properties on six clustering benchmark datasets. *Applied intelligence (Dordrecht, Netherlands)*, *48*(12), 4743–4759.

Fränti, P., & Sieranoja, S. (2019). How much can k-means be improved by using better initialization and repeats? *Pattern Recognition*, *93*, 95–112. https://doi.org/https://doi.org/10.1016/j.patcog.2019.04.014

Kärkkäinen, I., & Fränti, P. (2002a). Dynamic local search for clustering with unknown number of clusters. *2002 International Conference on Pattern Recognition*, *2*, 240–243.

Kärkkäinen, I., & Fränti, P. (2002b). Stepwise algorithm for finding unknown number of clusters. *Advanced Concepts for Intelligent Vision Systems (ACIVS'2002), Gent, Belgium*, 136–143.

Kaufman, L. (1990). *Finding groups in data : An introduction to cluster analysis*. John Wiley & Sons.

Linde, Y., Buzo, A., & Gray, R. (1980). An algorithm for vector quantizer design. *IEEE Transactions on Communications*, *28*(1), 84–95. https://doi.org/10.1109/TCOM.1980.1094577

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE Transactions on Information Theory*, *28*(2), 129–137. https://doi.org/10.1109/TIT.1982.1056489

MacQueen, J. (1967). Classification and analysis of multivariate observations. *5th Berkeley Symp. Math. Statist. Probability*, 281–297.

Mahajan, M., Nimbhorkar, P., & Varadarajan, K. (2012). The planar k-means problem is np-hard. *Theoretical computer science*, *442*, 13–21.

Malinen, M. I., & Fränti, P. (2014). Balanced k-means for clustering. In P. Fränti, G. Brown, M. Loog, F. Escolano, & M. Pelillo (Eds.), *Structural, syntactic, and statistical pattern recognition* (pp. 32–41). Springer Berlin Heidelberg.

Marowka, A. (2018). Python accelerators for high-performance computing. *The Journal of supercomputing*, *74*(4), 1449–1460.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Steinbach, M., Karypis, G., & Kumar, V. (2000). A comparison of document clustering techniques.

Theodoridis, S. (2008). *Pattern recognition.* (4th ed.). Elsevier Science.