UNIVERSITÀ DEGLI STUDI DI TRENTO
Facoltà di Scienze Matematiche, Fisiche e Naturali

Master of Science in Computer Science

Final thesis

Decentralized clustering with estimation of the number of clusters

Advisor:
Prof. Alberto Montresor

Student:
Alessio Guerrieri

Accademic year 2009 - 2010
Graduation session: 15 December 2010

CONTENTS

LIST OF FIGURES

1

CHAPTER 1:INTRODUCTION

1.1 CLUSTERING IN BRIEF

One natural question we may ask when given a set of items, is how are these items related. Can we divide them into homogenous groups? How can we even define what does homogenous mean? This problem is tackled by the area of cluster analysis: given a set of items we want to group them in clusters such that all items in the same clusters are "similar". The problem becomes more difficult when the items are not located in the same machine but are distributed across a network. This thesis presents a decentralized algorithm for clustering that works even when we do not know in advance how many clusters we need to create.

Decentralized data clustering

Clustering has a huge amount of applications in many different fields. In bioinformatics, for example, clustering can be used to group genes with similar expression patterns. In market analysis we can group customers with similar needs and desires to get a better understanding of the market. It can be used to find communities in social networks or to automatically divide documents into fields.

Some applications of clustering

The most commonly known clustering algorithm is the classical K-Means algorithm based on work done by LLoyd [16]. K-Means follows this procedure: it starts from K centroids (points representing the center of a cluster), repeatedly assigns each point to the closest centroid and recomputes the position of the centroids according to the points assigned to it. This algorithm is guaranteed to converge to a local minima of the average distance between each point and the center of its cluster.

While this algorithm has many positive characteristics, it still needs as input the value of K, the number of clusters we want to find. If we give K-Means the wrong K (for example because we do not know which is the correct number of clusters in the data) the algorithm fails to answer precisely. One algorithm that is able to compute a good clustering even without knowing the number of clusters is X-Means [20]. This algorithm uses K-Means as a subroutine and continues to divide the data set in smaller subsets using a statistical measure to decide where does the data need more clusters. Our algorithm tries to obtain the same results as X-Means while working in a decentralized and possibly online setting.

Limitations of the K-Means algorithm

1.2 MOTIVATIONS FOR A DECENTRALIZED CLUSTERING ALGORITHM

In some application fields, new problems can arise when we try to cluster a set of data. Consider for example the problem of BotNet detection, where we may want to group subsets of the nodes in the network that share similar (malicious) behaviour and communication patterns [11]. There are many issues that arise when we try to use a centralized approach to clustering in such a setting:

Motivations for decentralized clustering

- The data is not available in a single machine and it may be impracticable to keep it in only one. This happens in many fields when the amount of data is extremely large.
- Sending all the data to one machine may be too costly or may take too much time. If we want fast results then having to wait for the data to be collected in a single machine is less than desirable.
- The data might be sensitive and sending it on a network might cause a loss of privacy. If the data is collected by different organizations it is unlikely that they will agree on collecting the data in a single machine owned by a third party.
- We are not working on static data but on streams of data. When new data is constantly arriving it is impossible to follow the approach of waiting until all data is collected in a single machine.

1.3 MAIN CONTRIBUTIONS

The main contribution of this thesis is the combination of various known techniques to obtain a decentralized algorithm able to cluster data distributed in a network, without the need of previous knowledge about the number of clusters. Our algorithm first partitions the data into chunks at each node, then uses a decentralized version of K-Means on each of the chunks of data. Each of this instances of the distributed K-Means algorithm returns K centroids (points equal to the average of a cluster) and each node uses the list of all centroids generated by the instances as a compact representation of the data of the entire network. Each node will then run X-Means, a clustering algorithm able to choose the number of clusters, on this representation. All nodes then agree on the number of clusters using an aggregation algorithm and will each run a centralized instance of K-Means with the right number of clusters to get the final clustering.

Our algorithm in brief

This algorithm has two important properties. By using the fact that the algorithm works independently on each chunk of data, we can adapt it to work in an online setting (in which we want to look only at the last N chunks of data arrived) without the need of restarting the algorithm by scratch each time a new chunk of data arrives. Also, the nodes do not directly exchange information about the single points they are trying to cluster, thus achieving some kind of privacy if the data is sensitive.

Online and privacy properties

Using some simulation we show how does this decentralized algorithm behave using synthetically-generated data. The algorithm is able to reach high precision even when it is not given the exact number of clusters for its first step and does it with a global amount of computation comparable with the centralized version. The algorithm is robust in the case of failures and is able to reach the same precision (although the computation time increases).

Experimental results

BACKGROUND: This thesis is part of the Autonomic Security project, a 2-year project financed by the Italian Ministry of Education, University and Research (MIUR) under the PRIN 2008 programme. This project aims at creating a decentralized distributed system for security, focusing on these two tasks:

1. Research on the "self-*" properties of distributed systems and how can we use local interactions between nodes in a large scale to make these properties emerge.

2. Applying the results of the first step on self-protection against botnets.

The development of a decentralized clustering algorithm is part of the first task in this project. The next step will be to use our algorithm on real data obtained by the other research groups involved in the project and use a decentralized version of the BotMiner framework [11] to try to effectively detect botnets.

ORGANIZATION: Chapter 2 explains the background and some of the work already done in this area. The clustering problem is introduced, together with two different algorithms to solve clustering in the centralized setting and a few decentralized clustering algorithms. Chapter 3 illustrates the decentralized version of K-Means and our decentralized clustering algorithm. Chapter 4 shows all the experimental results obtained by running our decentralized clustering algorithm on synthetic data. It will also show how well does our algorithm react in case of failures of the nodes. Finally Chapter 5 explains some possible future directions of this study.

2

CHAPTER 2: RELATED WORK

This chapter introduces some of the terminology used in the rest of this work and provides an overview of the algorithms related to the problem we are trying to solve. The first section describes the clustering problem in general and the different types of clusters we may want to obtain. Section 2.2 describes the K-Means problem, explains the standard algorithm to solve it and, in Section 2.3 how it can be used as a subroutine to overcome some of its limitations. Section 2.4 shows the statistical criterions used to measure the precision of a clustering algorithm while Section 2.5 describes some other decentralized algorithms that use very different approaches to cluster data distributed in a network.

2.1 CLUSTERING

The clustering problem involves grouping items together in a way that puts "similar items" in the same groups. The definition 1 presents the problem in its most generic form:

Definition 1 (Clustering problem):
Given a set of items, partitions them in subsets as to minimize a certain metric.

Important characteristic of the clustering problem is that, at least in its most recurrent variety, it is a form of *unsupervised learning*. While in *supervised learning* we have a subset of the data that was already processed (by a learned entity like a human), in unsupervised learning algorithms we do not have any additional information outside of the data itself. Doing supervised learning on the clustering problem would mean having access to a subset of points already grouped in clusters. It is easy to see that this type of information can help solve the problem on the entire data, at least if we assume that the source of the data is homogeneous. Thus, when we talk about clustering, we assume that we do not have any additional information about the data outside of the actual points.

Unsupervised vs supervised learning

The items that we want to cluster have to be represented in a way that is easy to process. The standard way to represent an item is to select a collection of numerical features that are capable to describe an item and allow to decide whether it belongs to a set or not. Feature selection is a very complex field, which is outside the scope of this thesis. We just assume here that features are selected by an human researcher, according to the application field from which comes the data. For example, if we want to use clustering to find nodes that communicate in a similar way, the characteristics that we are interested in may be the number of packets sent per second, the average size of packets or similar metrics. Each feature is mapped to a dimension and for each item in the set we obtain a point in $\mathbb{R}^n$ with n equal to the number of "interesting" features we have chosen. To decide how many features we want to use, we have to strike a balance between too few features (not having enough information to solve the

Mapping items to $\mathbb{R}^n$

clustering problem) and too many features (incurring in efficiency issues and risk of overfitting the data).

Once we have defined how to map the items in the vectorial space $\mathbb{R}^n$ we need to define the notion of "similarity". How to do this is entirely dependent on the type of data. For example, if the features are discrete we might want to count the number of features in common and use it as the measure of similarity. Even if we want to directly use the coordinate of the points we have to choose between many measures of distance:

Options for the similarity/distance function

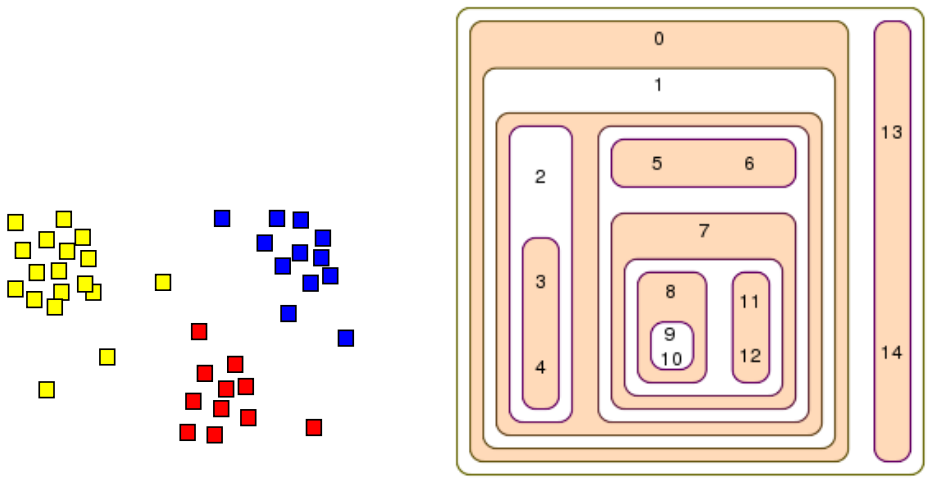
- Euclidean distance: this is the standard distance between points in a vectorial space, it's equal to the square root of the sum over all dimensions of the square of the differences.
- Manhattan distance: named after the streets of the borough of Manhattan, it is computed by summing the difference over each dimension.
- Chebyshev distance: it is equal to the biggest between the differences over any dimension.
- Mahalanobis distance: more complex than the euclidian distance, takes into account correlation and scale of the different dimensions [18].

The Euclidean distance is the most commonly used between the ones on the list but it first needs a normalization on the dimensions: if one of the features is qualitately different and is mapped incorrectly it might become too much important and might eclipse the other features.

Another decision we have to make when we choose which clustering algorithm we want to use is the following: how strict do we want to make the groups? Are the items allowed to exists in different clusters? If we make the clusters strict, meaning that each item can be mapped to one and only one cluster, then we are talking about *hard clustering*. If we allow the items to be simultaneously mapped to different clusters then we are talking about *soft clustering*. This thesis is about hard clustering; we will just provide a brief introduction to soft clustering. Each item is not mapped to a single cluster but gives to each cluster a weight between 0 and 1, with the sum over all clusters being equal to 1. The weights represent how close is the item to each cluster and help in capturing cases in which clusters might overlap in the data. There are many algorithms on this version of the clustering problem, the most famous being *Fuzzy C-Means* [7], an algorithm similar to the K-Means algorithm presented in the next section.

The last aspect we describe is the difference between *flat clustering* and *hierarchical clustering*. As the name suggest, in flat clustering all clusters are considered at the same level. In hierarchical clustering the items are clustered in an hierarchy, so items that are together in one of the smallest clusters are more similar than items that are together in the biggest clusters. Figure 1a and 1b show the main differences between the two approaches. The algorithms for hierarchical clustering are themselves divided between top-down algorithms, in which items start all in the same cluster and are then divided, and bottom-up algorithms, in which at the beginning each item is a single cluster and they are then joined until we get one cluster containing all items.

flat vs hierarchical clustering



(a) (b)

Figure 1: Flat clustering and hierarchical clustering

2.2 K-MEANS CLUSTERING

The classical example of flat clustering is *K-Means clustering*. The K in the name refers to the number of clusters we want to obtain while the suffix "Means" refers to the fact that each cluster is represented by a *centroid* equal to the mean of all points in that cluster. The K-Means clustering problem is part of the hard clustering family, so each item can be mapped to only a single cluster. The problem is properly defined in the following:

Definition of K-Means

DEFINITION 2 (K-Means Problem):
Given a set S *of points in* $\mathbb{R}^n$ *and a number* K*, create* K *subsets* $C_1 ... C_K$ *such that:*

1. $\bigcup_i C_i = S$
2. $\forall_{i,j}, C_i \bigcap C_j = \varnothing$
3. *minimizes* $\sum_i \sum_{x \in C_i} \|x - \mu_i\|^2$

Looking at the definition: we can see that the first two requirements force each point to be in exactly one cluster, thus ensuring that the resulting clustering is an hard clustering. The third requirement defines which is the correct clustering: the one that minimizes the square distance between the points and the centroid of its cluster. Note that by defining μ_i as the weighted mean of the points in C_i and normalizing the distance measure by the weight we can easily adapt this problem to a soft clustering setting.

The natural question we can ask is what is the complexity of this problem. The results are somewhat off-putting. If both k (the number of clusters) and d (the number of dimensions of the points) are fixed, then the problem is solvable in $O(n^{dk+1} \log n)$ [13]; if any one of these parameter is generic and unbounded then the problem becomes *NP-Complete* [2, 17]. Since no exact polynomial time algorithms are known to solve this problem we use heuristic algorithms.

Time complexity of the K-Means problem

Description of the K-Means Algorithm

The classical algorithm proposed to solve the K-Means clustering problem is usually called *K-Means* itself, or *LLoyd's Algorithm* [16] from the name of the proposer. The idea of the algorithm is quite simple. We take K random points as the starting centroids. We then assign each point in the dataset to the closest centroid, thus creating a clustering. We recompute the position of each centroid to be equal to the average of the points assigned to it. We then repeat the process, assigning each point to the closest centroid and recomputing the new position of each centroid until we reach an equilibrium (a local optima of the data). We show the pseudocode in algorithm 1, while in figure 2 we show a successful run of the algorithm.

Algorithm 1 K-Means (LLoyd) Algorithm

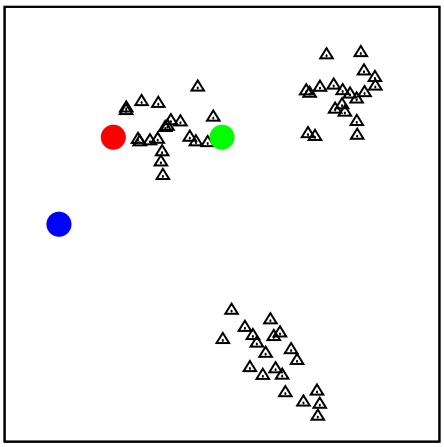
Require: $K \geqslant 1$
  for all $i \in [1..K]$ **do**
    $c_i \Leftarrow$ GetRandomPoint()
  end for
  repeat
    Set all C_i equal to $\varnothing$
    for all $\mathtt{item} \in S$ **do**
      $j \Leftarrow \mathtt{argmin}_i(\|c_i - \mathtt{item}\|)$
      $C_j = C_j \bigcup \{\mathtt{item}\}$
    end for
    for all $i \in [1..K]$ **do**
      $\mathtt{old}c_i = c_i$
      $c_i = \frac{1}{|C_i|} \cdot \sum_{j \in C_i} j$
    end for
  until $\forall_i, \mathtt{old}c_i = c_i$
  return all C_i

Time complexity of the K-Means Algorithm

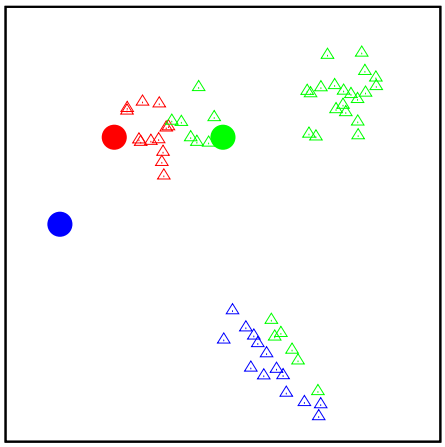
Studying the time complexity of this algorithm is not an easy task. It is easy to see that the time complexity is $O(n \cdot k \cdot d \cdot \#\text{iter})$, with d equal to the number of dimensions of the data, but bounding the number of iterations is a much more difficult problem. While in practice it has been observed that the number of iterations is usually much smaller than n [10], it has been proven that, in the worst case, the number of iterations can be superpolynomial. More precisely, it has been proven that exists a lower bound for the worst case running time equal to $2^{\Omega(\sqrt{n})}$ [4]. There have been some studies on this discrepancy between the actual speed and the theoretical bounds and, in the last few years, it has been proven that the smoothed time complexity of the K-Means algorithm is polynomial [3]. This means that, even for the worst case input, if we are allowed to randomly perturb the data set the algorithm runs in expected polynomial time. Since real world data sets are naturally perturbed we have some guarantees on the speed of this algorithm.

Aside from the running time of the algorithm, that as we have seen can be quite fast on real data, we have a bigger concern. This algorithm is, as we have said, an heuristic algorithm. When does it fail? Figure 3 shows an example run in which K-Means fails to cluster correctly all points. What can happen is that a centroid takes control of

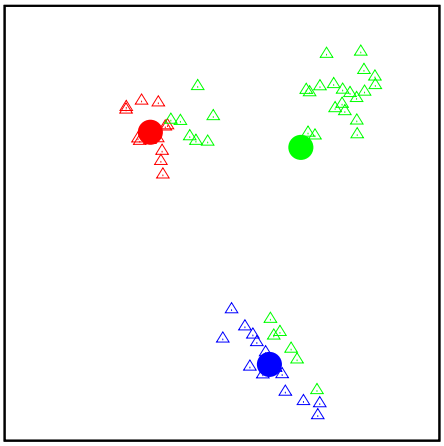
Figure 2: Successful run of K-Means with K=3



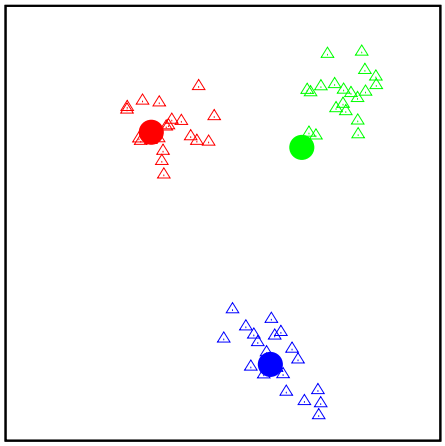
(a) The three centroids are chosen at random



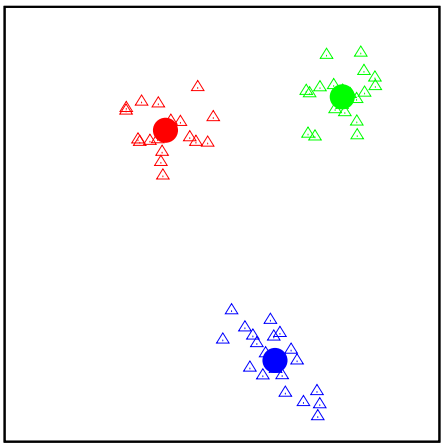
(b) Each point is assigned to the closest centroid



(c) The position of each centroid is recomputed



(d) Each point is reassigned to the closest centroid



(e) The algorithm has converged to the optimal solution

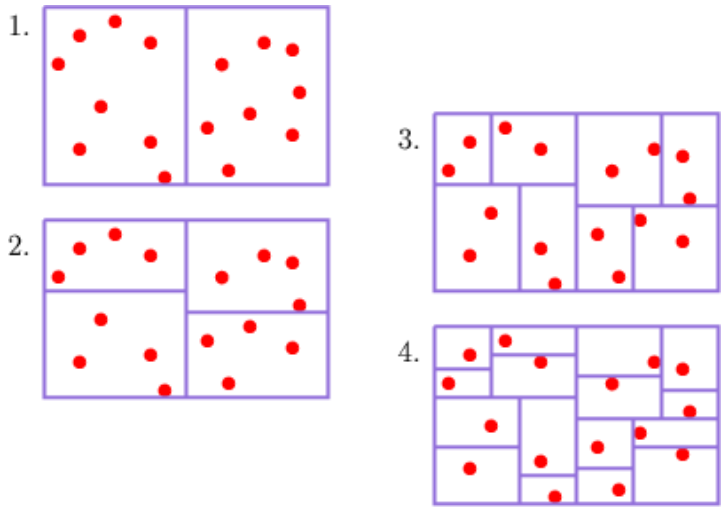
points from different clusters and there is no other centroid close enough to get those points back and start the process of dividing the points in different centroids. In the example showed in figure 3 the red centroid captures all points of both the top-left and the top-right clusters and the other two centroids are too far to have any effect on it.

In general the K-Means Algorithm is strongly affected by its starting position and a bad choice of the initial centroids can influence not only the running time but also the precision of the clustering. It has been shown that adding a new pass before K-Means itself to choose the starting centroids in a more smart way can increase both precision and efficiency. The K-Means++ algorithm [5] first chooses one centroid at random and then chooses iteratively one data point as the next centroid with probability proportional to the squared distance from the closest centroid, until we have reached the desired number of them. The idea of this algorithm is having starting centroids that are as far from each other as possible to cover most of the points. The authors claim that this additional step creates a dramatical increase in precision without any costs in term of running time.

Choosing the starting centroids

There are also many ways to speed up the K-Means algorithm itself, avoiding unnecessary computations and using smart data structures. One crucial observation is that if we can find the closest centroid for a whole subset of data points then we do not need to access each data point as long as we have the sum of the subset. Also, by geometric reasoning, given two centroids we can separate the space in two half spaces, and exclude that data points in one half space can choose the centroid in the other half space as their closest centroid.

Using geometric properties to speed up the algorithm

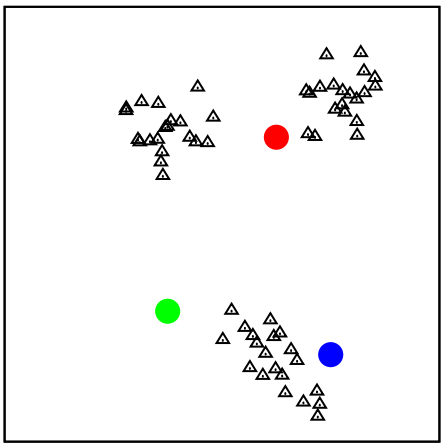


All these informations can be captured by a data structure called kd-tree. This tree is built by recursively dividing the dataset in two, creating an hyperplane parallel to one of the axis. Using this data structure it is possible to determine which centroid is the closest to an entire subset of data points and to exclude some centroids from consideration altogether. The hierarchical structure makes it easy to compute the necessary information for each subset.

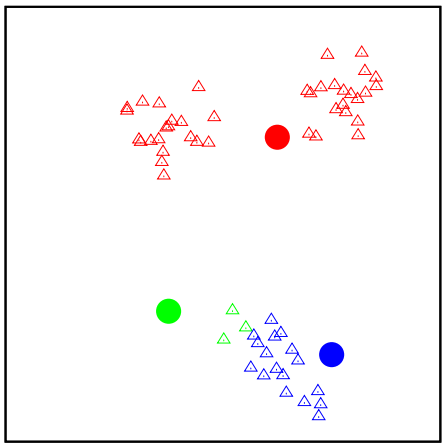
2.3 X-MEANS

In the previous section we talked about the K-Means problem as the most common way to look at clustering. In this algorithm we make one crucial assumption on the

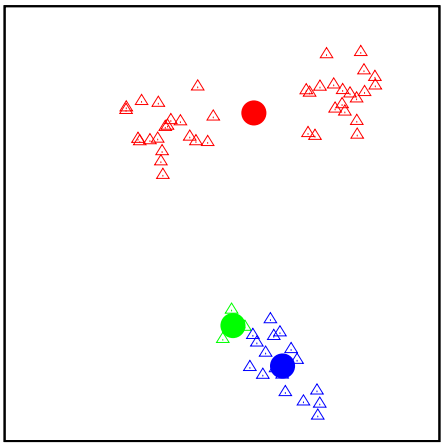
Figure 3: Unsuccessful run of K-Means with K=3



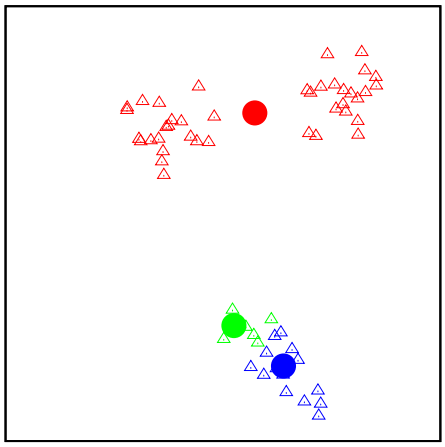
(a) The three centroids are chosen at random



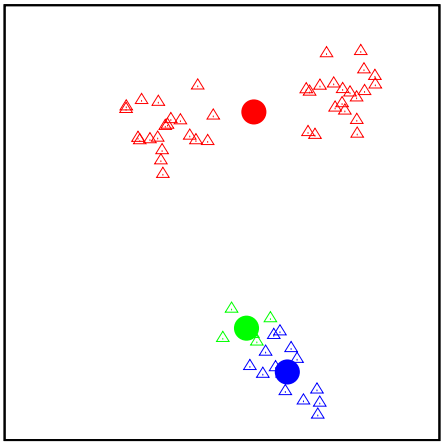
(b) Each point is assigned to the closest centroid. Note that the red centroid is the closest to the points in the two top clusters



(c) The position of each centroid is recomputed



(d) Few points change clusters



(e) The algorithm was not able to overcome a bad starting position

Limitation of K-Means: the number of clusters

information that we may have before running the algorithm: the fact that we already know the number of clusters in the real data. In many settings this is not a valid assumption and maybe the number of clusters itself is the variable that we want to obtain from the entire analysis. The only way to complete this type of analysis with K-Means is to repeatedly run the algorithm with different K and then choose the one that better represents the data.

How do we compare different clusterings?

Another question we might ask is how do we decide which clustering is better when the number of clusters is different. If we simply look at variables like variance or the maximum distance between any point and its centroid then we would obtain that the best clustering is the one that assigns each data point to its own singleton cluster. This clearly is not a very good answer, so we need to use some more sophisticated measures to compare clustering with a different number of clusters. In section 2.4 we show a number of measures that are commonly used.

In the clustering algorithm presented in this thesis we will make use of the X-Means algorithm developed by Pelleg [20]. This algorithm uses 2-Means (K-Means with K=2) as a subroutine and a statistical measure (usually the Bayesian Information Criterion (BIC), see Section 2.4) to obtain a clustering with the number of clusters chosen between 2 and a maximum number given as input. In algorithm 2 we show the pseudocode of X-Means without going into all the details.

Algorithm 2 XMeans algorithm (simplified)

```
Require: Data set S, Maximum number of clusters MAX
Require: Function 2Means(S) returns two clusters
  Clustering ← 2Means(S)
  BestScore ← −∞
  while |Clustering| < MAX do
    NewClustering = {}
    for all Cl ∈ Clustering do
      Cl2 ← 2Means(Cl)
      if Measure({Cl}) > Measure(Cl2) then
        NewClustering ← NewClustering ⋃ {Cl}
      else
        NewClustering ← NewClustering ⋃ Cl2
      end if
    end for
    Clustering ← NewClustering
    if Measure(Clustering) > BestScore then
      BestScore ← Measure(Clustering)
      BestClustering ← Clustering
    end if
  end while
  return BestClustering
```

The X-Means algorithm works as follows: we first cluster the entire data using 2-Means thus obtaining two clusters. Now the following process is repeated until

the number of clusters has exceeded the maximum number given as input to the algorithm:

Description of the X-Means algorithm

1. Run 2-Means on each cluster of the current clustering. The two starting centroids for this run are chosen by starting from the centroid of the original cluster and moving along a random vector by a distance proportional to the size of the region covered by the cluster.

2. If any of the clusters of the current clustering is a worse representation of the data (according to an appropriate metric) than the two clusters obtained by dividing it in two, replace the cluster with its two "sons".

3. If no clusters is worse than its sons then choose a constant fraction of the clusters to be replaced by their sons using the usual metric.

When the number of clusters in the current clustering has exceeded the maximum number given in input, we return the best clustering that we have found at the end of any iteration. We show precisely which metrics can be use in the next section.

In Figure 4 we can see an example iteration of X-Means. We can see that in the bottom clusters the splitting decreased the BIC score, since the measure penalized the additional number of clusters. Only in the top cluster the splitting results in an improvement over the single cluster.

The authors explain the reasoning behind this algorithm: instead of increasing the number of clusters by one at each iteration X-Means concentrates on the areas of space that are not well represented by the current clustering without having to account for interferences from the other clusters.

The advantages of this algorithm are many:

Advantages of the X-Means algorithm

- It is known that 2-Means is less likely to incur into a local minima with respect to K-Means ran with a bigger number of clusters.

- While the algorithm goes on there is an increase of the number of clusters and the number of instances of 2-Means that we need to execute, but each 2-Means has to work on a smaller number of data points.

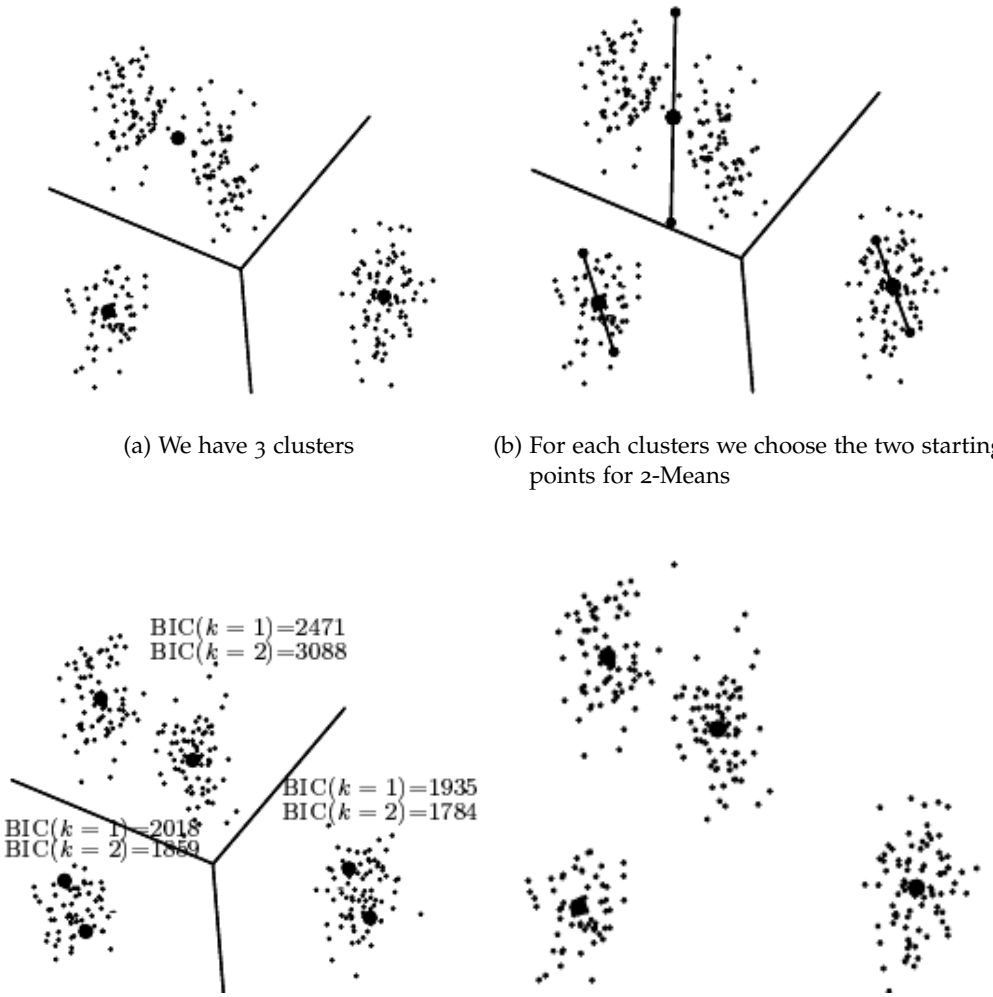
- The fact that the computations are mostly local creates many opportunities to cache values computed in previous iterations of the X-Means algorithm to speed up the running time.

- The resulting clustering can also be seen as a tree of clusters, thus creating an hierarchical structure on top of the clusters.

2.4 STATISTICAL MEASURES TO EVALUATE CLUSTERING

As we have seen in the previous section, X-Means needs some kind of statistical criterion to decide which between two different clustering better represents the underlying distribution of the data. We also need a criterion to compare the quality of the clustering obtained by our decentralized framework against a centralized clustering algorithm. This does not need to be the same criterion since the two

Figure 4: Example iteration of X-Means (from the original paper by Pelleg and Moore [20])



(a) We have 3 clusters

(b) For each clusters we choose the two starting points for 2-Means

(c) The results of 2-Means (with the BIC measure)

(d) The surviving clusters after this iteration

settings are very different. While the X-Means algorithm itself does not have access to labelled data (it cannot already know which points should be in the same cluster or otherwise the whole algorithm would be pointless) when we compare the centralized and decentralized algorithm we do have precise information on where do the single points come from. Since we have generated the data on which we run our algorithm we are able to label it and use a different measure to compare the clustering with the optimal theoretical clustering.

2.4.1 *Unlabelled data*

In the general case we cannot assume the presence of labelled data so we need to directly use the resulting clustering to get a measure of how good it is. The most basic way to measure how close are the points inside a cluster is to simply look at the statistic that the K-Means algorithm is trying to optimize. Given a set of points S given as a list of k clusters $C_1 \dots C_k$, each with average μ_i we can compute the following value:

A simple metric: inner-cluster sum of squares

$$\text{sum of squares} = \frac{1}{|S|} \sum_{i \in [1..k]} \sum_{x \in C_i} ||x - \mu_i||^2$$

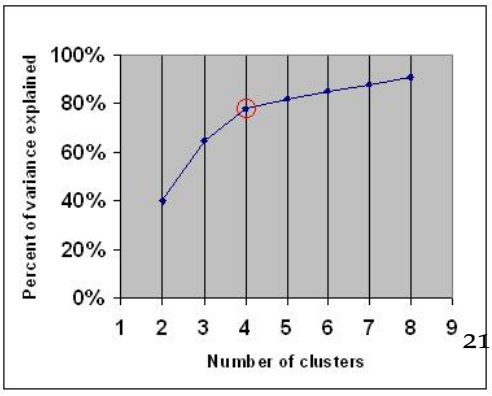
This is a very good metric to evaluate how close is each point to the centroids of its own cluster but it has one big shortcoming: we cannot compare clusterings with a different number of clusters. If we think about the idea behind this measure it is easy to see that the greater the number of clusters the better the clustering will be. Given any data, the optimal clustering will be the one in which each item is in its own singleton cluster since the measure will be equal to zero. If we do not know the exact number of clusters and we want to see which between two clustering of different size is the best then we need to add something that penalizes the bigger number of clusters.

If we look more closely to this problem, we can realize that the main issue with the naive approach of using directly the sum of squares metric is that we are actually overfitting the model to the data. What we want is to avoid using models that are too complex when simple models are able to capture the data to a similar degree. As Occam wrote during the middle ages:

> Numquam ponenda est pluralitas sine necessitate (Plurality must never be posited without necessity)

What we want to do is penalize complex model to advantage of simpler models.

One way to use the variance or similar measures to choose the number of clusters is using the *Elbow criterion*. On the right we can see a figure that shows how the quality of clustering (measured using the variance) grows with the number of clusters. At one point, marked by the circle, the rate of growth of the quality starts to decrease and creates



The elbow criterion

an "elbow" in the figure. This approach does not help us in comparing two different clusters but it has other applications. For example running K-Means with different Ks and then choosing the clustering using the elbow criterion is sometimes a simple but viable approach.

If we want to compare two clusterings of different dimension then one valid metric, used also by X-Means, is the *Bayesian Information Criterion* [21]. This measure uses the log-likelihood of the dataset according to the model and its number of parameters (in our case the number of centroids multiplied by the number of dimensions) to compute the following metric (from the original Schwarz paper [21]):

The Bayesian Information Criterion

$$\mathrm{BIC}(j) = \log M_j(S) - \frac{1}{2} k_j \log n$$

In this formula j is a model, $M_j(S)$ is the maximal likelihood of dataset S using model j, k_j is the number of parameters of the model and $n = |S|$. This is closely related to the *Akaike Information Criterion* [1] in which the negative component of the formula (the one that penalizes the complexity of the model) is simply k_j, without taking into account the number of points in the dataset. This does tell us that the BIC metric penalizes more the complexity of models then the AIC metric.

In the X-Means paper the authors apply the Bayesian Information Criterion to the clustering setting using the identical spherical gaussian assumption. To compute the log-likelihood of the data they simply sum the log-likelihood of the data of each cluster before subtracting the penalty for the complexity of the model.

BIC under the identical spherical gaussian assumption

Being M_j model (clustering) j, S the entire dataset, C_i the set of points in cluster i, K the number of clusters, D the number of dimensions we can compute the BIC score of a clustering as follows:

$$\begin{aligned}
\hat{l}(C_n) &= -\frac{|C_n|}{2}\log 2\pi - \frac{|C_n|\cdot D}{2}\log \hat{\sigma}^2 - \frac{|C_n| - K}{2} \\
&+ |C_n|\cdot\log|C_n| - |C_n|\cdot\log|S| \\
\hat{l}(S) &= \sum_{i\in[1..K[} \hat{l}(C_i) \\
p_j &= (K+1)\cdot D \\
\mathrm{BIC}(M_j) &= \hat{l}_j(S) - \frac{p_j}{2}\cdot\log|S|
\end{aligned}$$

This formula is computed and used at each step of the X-Means algorithm to make all local decisions on how many clusters we need to use and, globally, to decide which clustering we return.

2.4.2 *Labelled data*

If we have generated the data or if the data has already been labelled by another entity or algorithm then we have additional informations that we might use to get a

measurement of how good the clustering is, not againts other clustering but against the underlying distribution. The metric that we use in the rest of this thesis and that is computed on the resulting clustering of the algorithms is the F-Score (known also as F-measure). This is a global measure that has maximum value equal to 1 (perfect clustering) and is computed starting from two different metrics.

We want to see how good cluster C_i is when representing a given distribution. Call D_j the set of points generated by that distribution. Clearly, the optimal case happens when the two sets are exactly equal. In the worst case scenario the two sets have no point in common and they are completely unrelated to each other. What we want is to get a value between 0 (no points in common) and 1 (exactly the same) that measures how related are two sets of points. Traditionally we have two values: *precision* and *recall*.

The precision measure captures how many of the points in C_i come from the correct distribution over the number of points in C_i. If all points in C_i are also in D_j then we have perfect precision, while if no point in C_i is in D_j then the precision is equal to 0. The recall measure instead captures how many of the points that are in D_j (and that should be included in C_i) are actually included in C_i. If all points that have been generated by the distribution are included in C_i then we have perfect recall, while, as usual, if no points are in both sets we obtain recall zero. The formulas are very simple: *Precision and recall*

$$\text{Precision}(D_j, C_i) = \frac{|C_i \bigcup D_j|}{|C_i|}$$
$$\text{Recall}(D_j, C_i) = \frac{|C_i \bigcup D_j|}{|D_j|}$$

Note that each of the two metric taken by itself is not always a good indication of how related are two sets. For example, if C_i contains not only all points in D_j but also all the points of the dataset the recall is still equal to 1 (with very low precision), while if C_i is a singleton set containing a single point from D_j the precision is 1 (with recall very small). If we want a single metric that measures the relations between these two sets we need some kind of average of precision and recall.

The most common way to aggregate precision and recall in a single value is the *F-Score*. This value is equal to the harmonic mean of the two values, giving them exactly the same weight. The generalized version of this measure is known as F_β, with β being equal to 1 if we want to give the same weight to precision and recall. If $\beta = 2$ then we give recall twice the importance of precision and if $\beta = 0.5$ we give precision twice the importance than recall. The two formulas are listed below. *F-Score*

$$\begin{aligned} F_\beta &= (1+\beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \cdot \text{precision} + \text{recall}} \\ F &= \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \end{aligned}$$

In the rest of the thesis we use the simple F-Score, giving the same weight to both precision and recall.

To compute the F-Score of a clustering C with respect to the correct clustering D (created by simply grouping all points that come from the same distribution) we

follow this approach: for each cluster in D we compute the F-Score with all cluster in C and take the maximum. During the last step we simply average all this values (with weights proportional to the size of the clusters).

F-Score for clustering

$$
\begin{aligned}
F(D_i, C) &= max_{c \in C} F(D_i, c) \\
F(D, C) &= \sum_{d \in D} \frac{F(d, C) \cdot |d|}{\sum_{d \in D} |d|}
\end{aligned}
$$

This measure works correctly also when the number of clusters of the two clustering is different. If C has too many clusters then the recall is lower while if C has too few clusters then the precision is lower. By combining the two in a single value we are able to capture both cases.

2.5 DECENTRALIZED CLUSTERING

In this section we briefly describe various different decentralized clustering algorithms. All these algorithms make different assumptions on the distribution of the data in the decentralized setting and may have different aims than the one presented in this thesis.

2.5.1 *SODAS: Dynamic decentralized any-time hierarchical clustering*

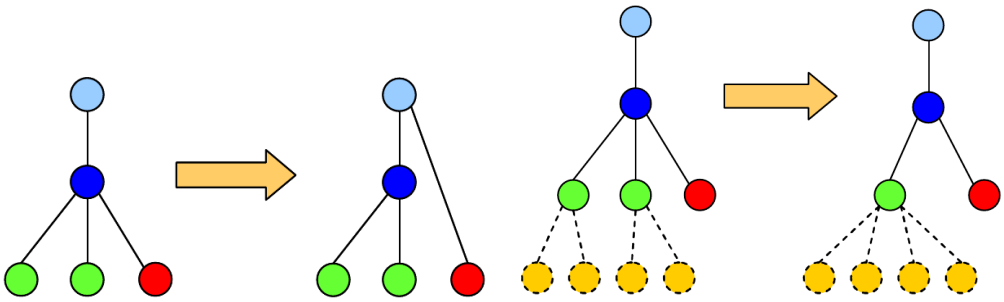
Van Dyke Parunak et al. [22] present a decentralized algorithm for hierarchical clustering called *SODAS* (Self-Organizing Data and Search) paradigm. This algorithm uses ideas borrowed from the biological behaviour of ants. These creatures are able to effectively cluster the items they keep into their nest (being those items food, eggs, larvae, etc...) so that each item is close to items of the same type. This clustering is mantained even when new items arrive (new food is collected), items disappear (food is eaten) or items change properties (an egg hatches and a larva is born). If we have a group of ants moving randomly in the nest we can obtain this type of results using few simple rules. Each ant does the following:

SODAS: using ants for clustering

- continues to move randomly
- keeps track of the type of the last few items it has met
- if it is not carrying anything, it picks up the object with probability that decreases if it has recently seen objects of the same type
- if it is carrying an object it drops it with probability that increases if it has recently seen objects of the same type

The paper uses these ideas to create an algorithm that is able to create an hierarchical clustering by making small, local decisions.

The algorithm first creates a tree, with each leaf corrisponding to one item that we want to cluster. Then, by using different metrics to see how close are the items in a subtree, we can apply one of these two operations: *promotion* and *merge* (see Figure 5).



(a) Promote operation: a subtree is moved up in the tree

(b) Merge operation: two subtrees are joined

Figure 5: The two operations of the SODAS algorithm

When we a node decides to apply the promotion operation on one of its sons the node removes the son from the list of its childrens and makes it the son of its father. A node decides to promote a son only if this operation increases the homogeneity of its children (and thus this son was very different from its siblings). The merge operation is applied when a node finds that two of its childrens are very similar. The two sons are then merged into one node having as childrens the childrens of both nodes.

The algorithm itself

The combination of these two operations is able to support the characteristics needed by this algorithm: when new data arrives, it is positioned in a random position in the tree. The promote operation is invoked repeatedly and the subtree containing the new data moves up in the tree. When it reaches another subtree containing similar data the merge operation is invoked as needed and the subtree finds its place in the tree. This setting is easy to adapt in a distributed setting since each computing node can work with a single subtree, make all the decision locally and exchange data with the neighbours when needed. When the algorithm reaches an equilibrium each computing node contains a subset of the items.

2.5.2 *Group formation among decentralized autonomous agents*

This work [19] uses a different approach than the algorithm we have just seen. Each node in the network is given a single item and a small number of neighbours in the network. The main idea is to rearrange these links so that nodes that contain similar items are also close in a network sense. At the end of the process each connected component in the network will represent a cluster in the data.

Each item is a node

Each node is given an item (for example a point in $\mathbb{R}^n$) and a set of objectives. These objectives are points, not necessary from the same feature space of the item, that are chosen from a function of the item itself. The node tries to create edges with nodes that have similar objectives, thus creating clusters of similar nodes. For example, if the items are points in $\mathbb{R}^2$ then the objectives might be close points that surround the item.

The objectives of a node

The reason for the use of these objectives instead of directly comparing the data is twofold: the objectives may have much smaller dimensionality than the items and, as

we will see, we can enable the mixing of the objectives between nodes of the same clusters. This last property will greatly speed up the algorithm in later stages.

Overview of the algorithm

The algorithm runs iteratively, each node keeps a list of unmatched edges (objectives that are not met), matched edges (objectives that are matched) and connected links (edges that will create the proper clusters). At the initial stage there are no connected links and so each node is a single cluster. Each turn is composed by the following four steps.

- Connecting: each cluster chooses some of its outgoing links to be connected according to a probabilistic rule.
- Mixing: each cluster takes all unmatched links and uses a random permutation on the unmet objectives of its nodes.
- Matching: each node tests its objectives and, when met, creates a new matched edge
- Breaking: each cluster chooses a number of its own matched edges and break them

By applying these four rules and using an appropriate decision algorithm to decide when to create and remove edges, the algorithm is able to create clusters of nodes with similar items.

2.5.3 *K-Means in MapReduce*

The MapReduce framework

MapReduce [9] is a framework created by Google to give an easy to understand abstraction for the creation of distributed algorithms. The programmer has only to define two functions, Map and Reduce, and then the data and the workload is spread between the nodes of the systems. The MapReduce framework has been implemented in the open source Apache Hadoop project [8] and now it is possible to write programs in the MapReduce framework also outside of the Google headquarters.

The Map function and the Reduce function

As the name says MapReduce is defined by two functions. The Map function takes a pair $(key, value)$ and returns a set of pairs $(intkey, val)$. The Reduce function receives a key and a set of values and returns a set of values. Given a set of data, seen as pairs $(key, value)$, the framework can execute the Map function on each of these pair independently. The results of the calls to the Map function are then grouped according to the key, creating pairs $(key, listofvalues)$ that are given to the Reduce function.

Counting the frequencies of each word in a set of documents

The most common example to show how does the MapReduce framework works is the wordcount problem: given a set of documents we need to count the frequency of each word in the documents. The Map function receives a pair $(key, value)$ with key being the name of the document and $value$ the contents of the document. For each word in the document the Map function emits a pair $(word, 1)$. The reducer receives the list of values corresponding to each word and simply has to sum all values to get the frequency for a specific word. Note that in this example is not only the Map function that can be parallelized, but also the Reduce function can be executed separately in each node and then combined with another Reduce call.

The classical K-Means algorithm has been implemented in Mahout, a machine learning library built on top of Hadoop. The idea is quite simple: the act of finding the closest centroid for each point is done independently for each point. Finding the average of the point assigned to a certain cluster is an operation that can be easily implemented by a Reducer. This algorithm works as follows: the Map function receives a pair $(\mathtt{listcentroids}, \mathtt{point})$ and outputs a new pair $(\mathtt{centroid}, \mathtt{point})$ with $\mathtt{centroid}$ being equal to the current centroid that is closest to the point. The Reduce function thus receives the list of points that have been mapped to a certain centroid and computes the average to get the new position of the centroids. This process is repeated as many times as needed accordingly to the K-Means algorithm.

K-Means in MapReduce

2.5.4 *Comparison between these algorithms and our approach*

Our clustering algorithm uses different assumptions than the algorithms we presented in this section. What we want is an algorithm with the following characteristics:

1. Works in a fully decentralized mode.
2. Can work in an online setting.
3. Does not exchange or communicate the input data between different nodes.
4. Does not need the number of clusters as input.

As we have seen the SODAS algorithm moves subtrees of data between nodes of the different network and does not satisfy the third of the requirements we just listed. The Group formation algorithm instead sees each input point as a node in the network and sends only the objectives in the network. While this algorithm also does not need the number of clusters to work the fact that each point is a single node makes its implementation difficult when the number of points is much bigger than the number of nodes.

The implementation of the K-Means algorithm in the Map-Reduce framework is quite similar to the distributed K-Means algorithm that we will use as a subroutine in our decentralized clustering algorithm. The main differences are that the Reduce function (finding out the new position of each centroid by computing the average of all points mapped to it) is not executed on the entire network but each node computes the new centroids using only the data in its neighbourhood. Using this approach we obtain an algorithm that is more decentralized and does not need any hierarchy on the nodes of the network.

3 CHAPTER 3: A DECENTRALIZED CLUSTERING ALGORITHM

In this chapter we describe our decentralized clustering algorithm. We define how the data arrives into the network and which assumptions can we make on the level of cooperation of the nodes. We show a distributed version of K-Means, how do we use it as a step for our algorithm and how this algorithm can be easily adapted to work in an online setting.

3.1 INTRODUCTION

Our aim was to create a decentralized clustering algorithm that did not need the number of clusters as a parameter and in which nodes do not have to exchange possibly private data to each other. Our initial approach was to take the X-Means algorithm and try to develop a decentralized algorithm based on the same idea of repeatedly splitting the clusters. This algorithm was very challenging to obtain, since the X-Means algorithm needs an huge amount of syncronization and exchange of data: if the data is distributed in the network then, as the clusters get smaller, we will need more communication to find which nodes contain data from the different clusters. We would also need a decentralized algorithm to compute the Bayesian Information Criterion and execute it at each step of the algorithm.

First approach: a decentralized X-Means

To avoid having to adapt X-Means in a decentralized setting we decided to use a different approach. We divide the data in N chunks as it arrives to each node and we run a different instance of a distributed K-Means algorithm (chosen from the literature) on each of these chunks. Each of these instances will return K centroids, with each node obtaining roughly the same results. Each node will put all the $N \cdot K$ centroids in a single list and will use it as a compact representation of the data. Each node will run X-Means locally on the $N \cdot K$ centroids and, after some rounds of communication to agree on the number of clusters, will be finally able to generate the final clustering.

The main idea of our algorithm

The assumptions we make on the setting are the following:

Assumptions on the data and the network

- Each node in the network receives a stream of points
- The amount of points each node receives might be different
- The data is homogenous, meaning that each node receives data from the same set of distributions
- The nodes do not want to share the points they received but only aggregate values
- We do not know the number of clusters that better represent the data

We will first describe the decentralized K-Means algorithm we chose to use. Once the behaviour of this algorithm will be clear it will also be easier to understand how does it fits inside our decentralized clustering algorithm.

3.2 DISTRIBUTED K-MEANS

As a building block for our algorithm we used the distributed K-Means algorithm developed by Bandyopadhyay et al. [6]. This algorithm was created to be able to cluster streams of data in peer-to-peer environments, is quite simple and easy to implement and shows many interesting properties.

Distributed K-Means algorithm idea

The idea is quite simple: each iteration of the K-Means Algorithm is executed at the same time by all nodes in the network. Each node starts with the same centroid, updates them using its own data and computes an average between its own centroids and the centroids computed by some of its neighbours. This process is repeated until we reach a steady state.

If we look more closely to the algorithm we can see that what it does is a simple relaxation of the averaging step of K-Means. All nodes start from the same centroid and are able to map each of its point to the closest of them. In the centralized version of K-Means we would take all the points that have been mapped to a single centroid and compute their mean to get the new position of that centroid. Trying to follow exactly this approach in the distributed version of K-Means would require each node to send to all other nodes its list of new centroids. This approach would be similar to the MapReduce algorithm we have seen in the previous chapter. The distributed K-Means algorithm instead uses an approximation by asking the centroids only to a certain number of neighbours. This choice makes the algorithm more robust and, since we do not actually need to wait for all other nodes to finish their computation step, it also allows more flexibility in term of syncronization between each iteration of K-Means. In our case we assume that there actually is syncronization between the nodes in the network but we still want to see what happens when some of the nodes go down and stop answering queries. This algorithm is able to adapt and survive even with high failure rate, as it is showed in the next chapter.

Comparison with the centralized K-Means algorithm

Going more deeply in our implementation of this algorithm we need to explain some details. The first question is how do we get the starting centroids. In the traditional centralized K-Means the starting centroids are tipically chosen between the points we are trying to cluster while in a decentralized setting it is not possible to choose points that are present in the data of all nodes. We still do not want to choose completely random points since they may end in parts of the $\mathbb{R}^n$ space that are completely empty, thus decreasing both the efficiency and the accuracy of the clustering algorithm. In our implementation we assume that the nodes have already decided which are the starting centroids by some mechanism and we give to all of them the same set of starting centroids. This set is generated using the same routine executed to generate all of the data points. If we wanted to implement this mechanism in a fully distributed way we could have used the following strategy: each node chooses K points from its data, computes a random value between 0 and 1 and sends the centroids paired with the value to all neighbours. Each node keeps the centroids with associated the biggest value and sends them to its neighbours. In a logarithmic number of communication steps we should reach the situation in which all nodes have the same set of centroids chosen from the data set of a random node. This has not been implemented in our simulations to avoid complicating the system.

Choosing the starting centroids

Another important part of this algorithm that we need to define is the distributed termination condition. In the centralized K-Means the algorithm terminates when there are no more updates to the centroids and we have reached a steady state. This approach is not viable in a decentralized setting so we need a condition based on the local view of each node without taking into account the entire network. Each node keeps track of the last set of centroids it has generated and, at each iteration, checks if they have been changed by measuring the distance between the centroids and checking if the average change is more than a given threshold. If $(w_1, \ldots, w_K)$ are the old centroids and $(w'_1, \ldots, w'_K)$ are the new ones, the formula used is the following:

termination condition

$$\mathrm{Change} = \frac{1}{K} \cdot \sum_{i \in [1..K]} \|w'_i - w_i\|$$

Using this procedure, a node can check if its own centroids have changed in the last iteration. As a termination condition, a node termines the execution and outputs its centroids when both the local centroids and the ones of the contacted neighbors have not changed in the last COUNT iterations, with COUNT being a parameter of to the algorithm. When a node has terminated the algorithm continues to answer queries from other nodes but it does not update its centroids anymore.

Another note we need to make regards how do nodes average their centroids with their neighbours' centroids. This is not a difficult task but it must be made with caution. Since different nodes in the network may receive more data than others we want to give more importance to the centroids of nodes with a big amount of data. The idea is quite simple: the nodes not only send their centroids but also the number of points associated to them. Each node, once it has received the centroids, simply computes a weighted average. If a node has received centroids $(w^j_1, \ldots, w^j_K)$ from neighbour j, each of them with the number of points assigned $(n^j_1, \ldots, n^j_K)$, it adds its own centroids to the list and then computes the new position as following:

Averaging the centroids

$$w'_i = \frac{1}{\sum_j n^j_i} n^j_i \cdot w^j_i$$

This approach ensures good behaviour when nodes receive a different number of data points: the nodes that receive a lot of points will be more important in the computation and they will naturally help the nodes with less points.

Another interesting property we want to emphasize is the fact that each node only communicates with the other using aggregate data. In the communication rounds the nodes only send their centroids to some of their neighbours. The actual points are not shared in the network and each node can avoid sharing information with the other nodes. This is highly desirable when the data is distributed between nodes representing different companies and they want to collaborate to get a good clustering without having to directly share their points with the competition.

Privacy of the data

The algorithm receives a number of parameters to precisely define its behaviour.

The parameters of the distributed K-Means algorithm

- K: the number of clusters we are trying to find. As in the centralized version of K-Means this algorithm needs the number of clusters and reacts poorly when the value given to it is different from the number of clusters in the underlying distribution

- QU: the number of queries the algorithm performs in each iteration. Each node asks their centroids to QU of its neighbours, choosing them at random during each iteration. In our simulations we assume that each node has a fixed number of links chosen at random at startup.

- DELTA: used for the termination condition. Since each iteration the node communicates to a possibly different set of neighbours we need to allow for very small movements of the centroids. While the centroids have changed less then DELTA we consider them as they did not change at all.

- COUNT: for how many consecutive iterations must a node not see any changes in his centroids and in the centroids of its neighbours before considering the algorithm terminated. The addition of this parameter causes the nodes to wait more and makes sure that they check that different neighbours have reached a steady state.

Algorithm 3 Simple pseudocode for the distributed K-Means algorithm

```
Each node:
Get/choose starting centroids w_1 ··· w_K
c ← 0
repeat
  Map each point to the closest centroid
  Compute position of new centroids w'_1 ··· w'_K
  Check if new centroids are different than centroids computed during the last
  iteration at this step
  Ask QU random neighbours for their centroids
  Answer any request with w'_1 ··· w'_K
  Compute new w_1 ··· w_K using weighted average between own centroids and
  centroids received
  if not changed and neighbours not changed then
    c ← c + 1
  else
    c ← 0
  end if
until c > COUNT
```

3.3 OUR DECENTRALIZED CLUSTERING ALGORITHM

Limitations of the distributed K-Means algorithm

The distributed K-Means algorithm we have just described has the same problem as the standard K-Means algorithm in that it strongly depends on the parameter K. If we do not know in advance which is the correct value of K and we guess incorrectly then the algorithm fails to produce a good clustering. Our choice was to still use the decentralized K-Means algorithm, but only as a subroutine for our clustering algorithm. Figure 6 gives an overview of the entire algorithm.

Description of our algorithm

Our clustering algorithm can be summarised in the following steps:

- Each node receives a stream of data points and divides them into chunks using the same criterion as the other nodes. For example, a node can group the points arrived in each hour as a single chunk. It is not important that each chunk be of the same size but each node must have the same number of them and must be able to start at the same time the computation on each chunk.
- The network executes the distributed K-Means algorithm on each chunk, using a K chosen arbitrarily. We show that even choosing K different from the right number of cluster does not decrease the precision of the whole algorithm in the same way it affects the distributed K-Means algorithm.
- When the computation has finished each node has computed a list of centroids, K for each chunk. This list should be similar (even if not exactly the same) in each node.
- Each node executes the X-Means algorithm on its list of centroids using the list as a compact representation of the data given to the entire network.
- To avoid computing clusterings with different number of clusters, we run a pairwise averaging algorithm on the values computed at each node. This algorithm is run until we obtain the same number in all nodes.
- Finally, each node runs K-Means on the list of centroids, using K equal to the number of clusters resulting from the previous step and reusing some computation done by the X-Means algorithm.

We now go into more detail in the implementation and specification of each step of the algorithm.

3.3.1 *Choosing chunks*

Ideas from streaming algorithms

The idea of dividing data in chunks and working separately on each of them comes from a paper ("Clustering data streams" [12]) on how to execute clustering on a stream of data. Our algorithm is not a streaming algorithm but the usage of techniques from data stream analysis helps us add some online capabilites to the algorithm.

How do we divide the data in chunks

In our simulations, the nodes receive the data already in chunks and do not have to divide it. In a real application we would need a way to make sure that each node has the same number of chunks and that is able to start at the same time. The easiest way to obtain this result would be dividing the data according to the timestamp (creating a chunk for each interval of time). Even if the nodes receive data at different rates and therefore create chunks of different size the distributed K-Means algorithm makes sure that each node has similar results.

3.3.2 *Distributed K-Means*

The nodes in the network need to execute the distributed K-Means algorithm on each chunk of data. The implementation of this algorithm is the one we explained before with the simple addition of an id to each message. This id is used to make sure that

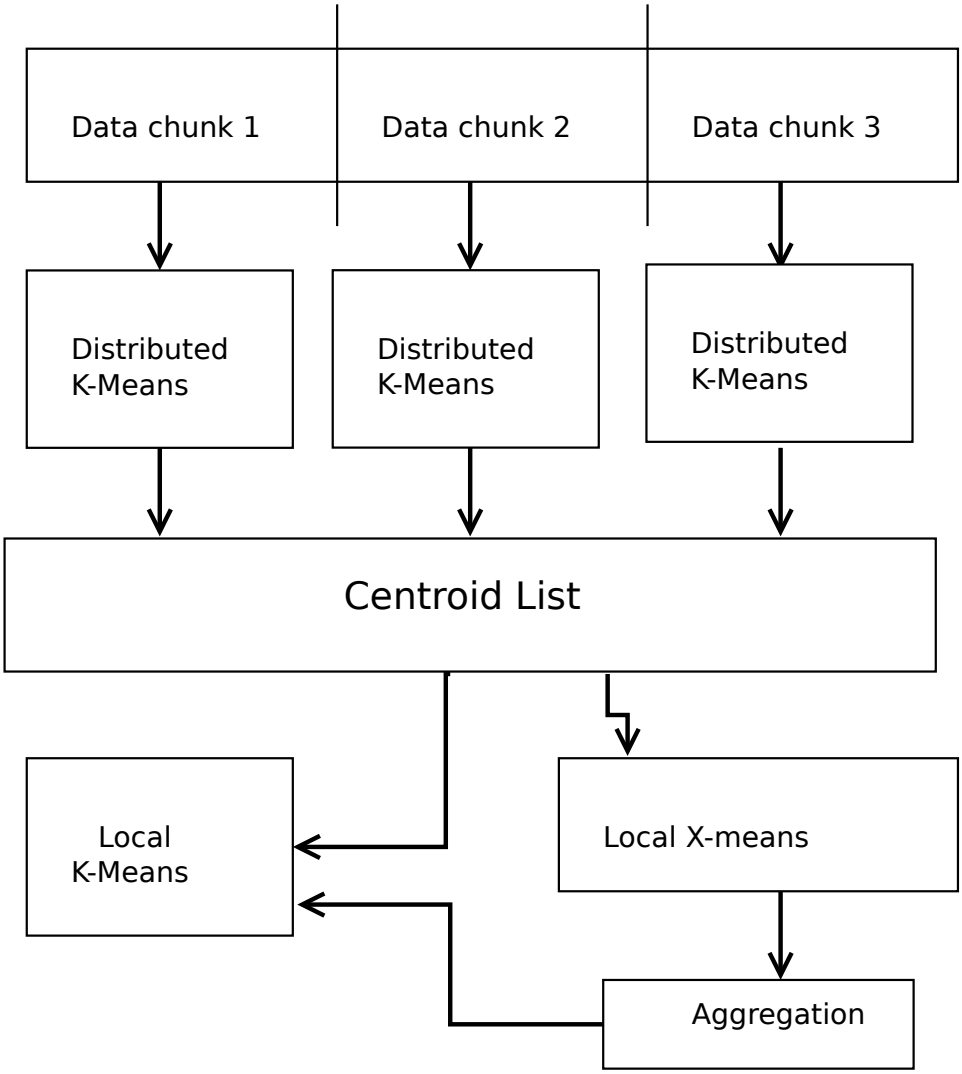


Figure 6: An overview of our clustering algorithm. This procedure is executed on each node in the network

the node knows which instance of the distributed K-Means algorithm has to receive the message.

We start an instance of the distributed K-Means algorithm any time a chunk of data items has arrived. Each instance is completely independent and it chooses its own starting centroids using the same procedure we explained in section 3.2. The termination condition is kept the same and once all instances have finished (we explain in section 3.4 what can we do in the cases in which the streams of data are unbounded) we have obtained a set of centroids, K for each chunk. Since the distributed K-Means algorithm converges to roughly the same set of centroids in all nodes we can say that the list of centroids resulting from this step is similar in all nodes, regardless of the amount of data points each node received. We use this list to represent all the data in the network.

The distributed K-Means instances

3.3.3 *X-Means*

In this step each node in the network executes their own instance of X-Means on the list of centroids. It may seem like a waste of computation making each node execute X-Means, since the list of centroids of each node should be similar, but it is not exactly the case. The X-Means algorithm still depends from the results of all the different runs of 2-Means and this algorithm, being a special case of K-Means, depends from the starting centroids, chosen completely at random. Different runs of X-Means result in different clustering and, in unlucky cases, in a number of clusters far from the correct answer. By running X-Means separately on each node in the network we use the power of the system to make sure that unlucky instances of the X-Means algorithm are ignored.

Each node executes X-Means

One problem we found with this approach is that the X-Means algorithm was created to work on points, while we now give them as input a set of centroids. Each centroid is seen as a weighted point, with the weight being equal to the number of points that have been mapped to it in the distributed K-Means algorithm. We had a problem with our initial implementation of the Bayesian Information Criterion: a cluster containing a single centroid from the list is seen as a cluster with big size (all the points mapped to the centroid in the distributed K-Means algorithm) and variance equal to zero. Using each cluster as a single point (getting rid of the weights) did not give the X-Means algorithm enough data to correctly determine the number of clusters. Our solution was to make each node compute the variance of each centroid in the list using their own data and using this value as a "lower bound" on the cost of making a cluster containing that centroid.

Using the variance of each centroid

3.3.4 *Pairwise averaging*

We could directly use the clustering obtained by X-Means as the final results of our algorithm but we decided to add two additional steps. The result of the X-Means algorithm and the number of clusters obtained can vary significantly in different instances and we would like each node to create a clustering as similar as possible

and with the same number of clusters. To obtain this we used the simple pairwise averaging algorithm from the aggregation paradigm [14].

Pairwise averaging aggregation protocol

Each node chooses a neighbour and sends to it the number of clusters it has found. The neighbour answer the request with its own number of clusters and then both nodes compute the average between the two values. In the next iteration they will both use the average as their estimation of the number of clusters. If node i and j execute a round of this algorithm, with n_i being the current estimate of the number of clusters of node i the new estimate is computed as follows:

$$n'_i = n'_j = \frac{n_i + n_j}{2}$$

Termination condition

This protocol is guaranteed to converge quite quickly since after each communication round the variance decreases of a constant factor. To allow each node to take part in this protocol the nodes do not terminate until all nodes have joined this protocol, using a simple bit-vector exchanged between the active nodes. The nodes terminate this protocol and stop updating their value when the bit-vector is full, meaning that all nodes have been able to contribute, and their own value did not change more than a given amount in the last step.

3.3.5 *Local K-Means*

When the pairwise averaging step is over, all nodes have the same estimate of the number of clusters in the underlying data. Now each node can run the centralized K-Means algorithm on the list of centroids using the value K that we have computed in the previous steps. Here we can consider each centroid in the data as a weighted point without having to worry about the inner variance.

Reusing the centroids of X-Means as the starting centroids

In our first implementation of this step all instances of K-Means were started using random points in the data set, but it seemed than on average we would get clustering with worse precision than the one computed by the local X-Means step. This behaviour was caused by the usual issue of K-Means: the convergence to a local minima. To solve this problem we just decided to reuse the centroids found by the local X-Means as the starting centroids for the local K-Means. Since both algorithms are run on exactly the same data, starting from the results of X-Means helps the K-Means algorithm to reach the same precision of X-Means while having exactly the desired number of clusters.

A more efficient implementation of this algorithm could share additional information between the two algorithms. If we wanted to compute some of the data structures commonly used to speed up clustering algorithms (like the kd-trees), we could reuse them in both algorithm, thus avoiding unnecessary computations.

3.3.6 *Remarks*

A few remarks on the algorithm as a whole:

Data exchanged during the algorithm

- The nodes in the network communicate only during two steps: the distributed K-Means step and the pairwise averaging step. In the first one they send only their centroids and only aggregate data is sent on the network. In the second

step the only value that is shared is the number of centroids. The actual points given to the nodes are not shared across the network.

- At the end of the algorithm the nodes in the network still obtain different clusterings since we use the centroids computed by each local X-Means instance as the starting centroids for the local K-Means. If we wanted to obtain a much more similar clustering we might want to make all nodes start from the same centroids. We could choose these centroids from the results of the local X-Means of a random node by using a procedure similar to the one described for choosing the starting centroid of the distributed K-Means.

Getting the same clustering in each node

3.4 ONLINE VERSION

While the simulations shown in the next chapter are set in a static setting the algorithm itself was created to be easily adapted in an online setting. It works using the following assumptions:

- Each node receives data from an unbounded, possibly neverending data stream.
- The data can be divided into chunks as usual.
- We are interested in clusters resulting from the last N chunks of data.

Assumptions in the online setting

This model is often called the "sliding window" model, meaning that while new data arrives old data becomes useless and must be ignored in the execution of the algorithm.

To show how our algorithm can work in this setting we explain what happens when a new chunk of data arrives in a network. All nodes have already reached the last step of the decentralized clustering algorithm on the previous N chunks of data and have obtained a clustering of K_0 clusters

Steps of the online version of our algorithm

- All nodes in the network start an instance of the distributed K-Means algorithm on the new chunk, using $K = K_0$.
- When the distributed K-Means algorithm has terminated the nodes update their list of centroids by throwing away the ones resulting from the oldest, now outdated chunk and adding the new centroids computed from the new chunk.
- We run the local X-Means algorithm giving it an upper bound on the number of clusters based on K_0. Since most of the data is the same the underlying data cannot have changed by much.
- The pairwise aggregation step and the local K-Means step is repeated as usual thus obtaining a new clustering and a new estimate of the number of clusters.

Note that the fact that the data is separated in chunks helps us in avoiding repeating unnecessary computations in the distributed K-Means step. Reusing the old number of clusters as K helps in making even less important the initial value of K. The

The algorithm becomes self-adjusting

algorithm will self-adjust automatically and if there will be changes in the sources of the data the estimate will change accordingly.

Another interesting point is that we do not need to wait for the last chunk to be processed to start working on it. The distributed K-Means algorithm can be started the moment the data is ready, using the last estimate of the number of clusters computed as its own K.

4 CHAPTER 4:EXPERIMENTAL RESULTS

We decided to implement the whole algorithm using PeerSim [15], a P2P Simulator written in Java. The whole system is comprised by three algorithms: a Generator Algorithm that generates the data and sends it to the clustering algorithm; the clustering Algorithm itself with its own implementation of the distributed K-Means algorithm; an Evaluator algorithm that, given the correct assignment computed by the generator algorithm, analizes the precision of the clustering computed the decentralized clustering algorithm using the F-Score measure.

After explaining the implementation of both the generator and the evaluator algorithm, we show some results on the precision and running time of both the distributed K-Means algorithm and our decentralized clustering algorithm.

4.1 GENERATOR

The data has been generated using a simple algorithm: the generator class receives as parameters:

Parameters of the generator algorithm

- The number of dimensions: unless stated otherwise our experiment are executed with two dimensional data.
- The size of the world: we give the algorithm the size of the space in which it has to generate the points.
- The number of clusters it has to generate: as we have said this information is only given to the generator algorithm and the evaluator algorithm that checks the precision of the clustering algorithm.
- The desired variance of the clusters of data: in our simulation all clusters have the same variance.
- The number of chunks of data: as we described in the previous chapter the generator automatically creates the chunks of data. In our simulation the chunks are then sent to the clustering sequentially, one every constant number of iterations.
- The number of points per chunk: this value can be constant in all nodes or can be chosen randomly in a defined interval

The generator first chooses the means for each of the K distribution it must generate. This is done independently so there is a possibility that two clusters overlap, thus possibly creating errors when we apply our clustering algorithm on the data. Since we compare our decentralized algorithm against two centralized algorithms we do not worry about this issue since also the centralized algorithms would suffer in the same way.

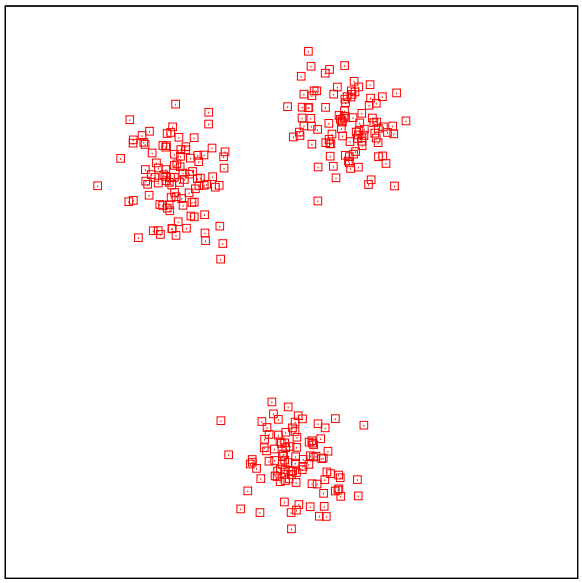


Figure 7: Data generated by the generator algorithm using three clusters

For each point, we first choose a distribution at random with the same probability. As the second step we get values from a standard gaussian distribution, one for each dimension, and we add them to the mean of the chosen distribution to obtain the new point. Note that, while each point has the same probability of being generated from all clusters, we do not ensure that a chunk will contain data from all distributions. If the number of points in a chunk is small then it is likely that some distributions will not be represented in it. Figure 7 shows some data generated by the algorithm, with three clusters.

4.2 EVALUATOR

The evaluator algorithm is run once all nodes in the network have finished working. This algorithm gets the labelled data of the entire network from the generator and extract the clustering obtained by the decentralized clustering algorithm in each node. To have something to compare our algorithm against we then run the centralized K-Means (with the correct number of clusters) and the centralized X-Means on the data of the entire network. On each of these clustering we measure the F-Score according to the formulas explained in section 2.4.2.

Comparing precision of our algorithm against centralized algorithms

To measure how good is our decentralized clustering we measure the F-Score of the clustering computed by each node in the network and we calculate an average score. The clustering computed by each node is tested against the data of the entire network. Note that if the algorithm returns a centroid that is very far from any of the real data points, then we do not introduce any errors since this centroid will never be used and no points will be mapped incorrectly.

Precision of the decentralized algorithms

The Evaluator also tries to compare how much computational work is done in the decentralized setting against the centralized setting. Since the time spent in

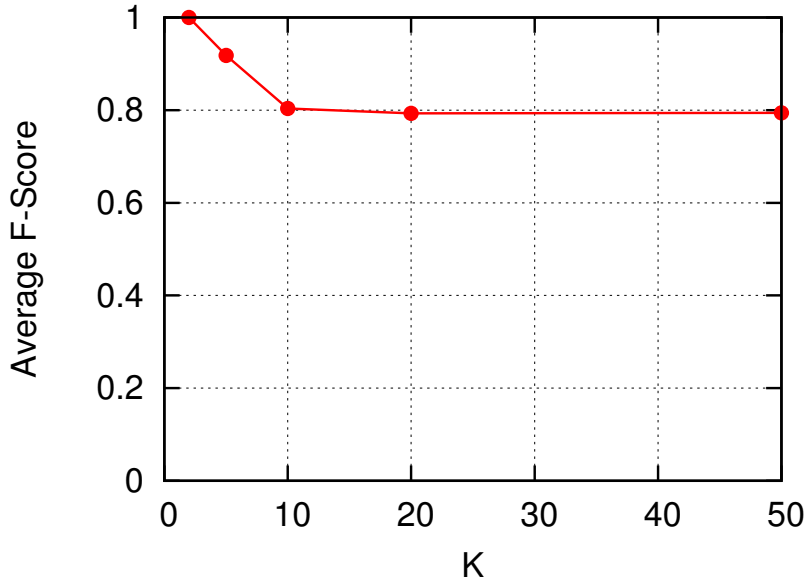


Figure 8: Distributed K-Means algorithm on a network of 100 nodes and data coming from K distributions. The algorithm is extremely good with K small but it still gets high F-Score when K grows.

the decentralized algorithm is dominated by communication time it is not possible to simply compare the number of seconds used by the simulation to execute the decentralized algorithm to get a good measure of the efficiency. What we do is to count the number of times the distance function is called across the entire execution. The distance function is used very often in both the distributed and centralized algorithm and it is a good estimate for the amount of computation made by the algorithm.

Estimating the amount of work done by the algorithms

4.3 DISTRIBUTED K-MEANS

We first studied the behaviour of the distributed K-Means algorithm. Here we simulate a run of the algorithm when giving a single chunk of data to each node.

Figure 8 shows the F-Score of the K-Means algorithm when it is given the exact number of clusters used by the generator algorithm to create the data. The distributed K-Means algorithm is always precise but it is especially good when the clusters are few. This is consistent with the observation that 2-Means (K-Means with K=2) is very precise and can avoid more easily a local minima [20].

F-Score of the distributed K-Means Algorithm

When the distributed K-Means algorithm is not given the correct number of clusters the precision decreases. In figure 9 we show what happens when we give different values of K to the algorithm while keeping the number of clusters constant (20). When the algorithm is given the right value the precision is high, but the further is the parameter from the correct value the worse are the results. The distributed K-Means algorithm is expecially bad when it is given a lower number of clusters.

F-Score of the distributed K-Means Algorithm with wrong K

Completion time

Another interesting property we may study is how long does the algorithm takes as a function of K. Figure 10 shows that the completion time needed by the algorithm (when given the correct value of K), increases with the complexity of the data.

Completion time with wrong K

When the distributed K-Means algorithm is given data with a fixed complexity (the number of clusters do not change) then the behaviour is different. In figure 11 we show what happens when, as in figure 9, the data comes from 20 different distributions and the distributed K-Means algorithm gets different K. The algorithm is slower to complete when it is given a smaller K than when it is given a bigger K. It seems that greater the number of clusters it tries to create the quicker it reaches a local optima and completes the execution.

Percentage of nodes that have completed

In figure 12 we show how does the percentage of nodes that have completed the algorithm grows with respect to the time. This simulation has been done by giving the right parameters to the distributed K-Means algorithm with data generated from 20 distributions. We can see that the nodes do not terminate together but a few of them might have to wait much longer than the fastest nodes. This causes some problems in our decentralized clustering algorithm because we need all nodes to have terminated the algorithm before we are able to start the last few steps. As we explained in section 3.3.4 we will use a bit-vector to make sure that all nodes are ready to continue before starting the last few steps of our algorithm.

Scalability of the algorithm

The last property we want to study is the scalability of this algorithm. What happens when the number of nodes grow? In figure 13 we show that the F-Score of the distributed K-Means algorithm, when given the right amount of clusters, remains mostly the same.

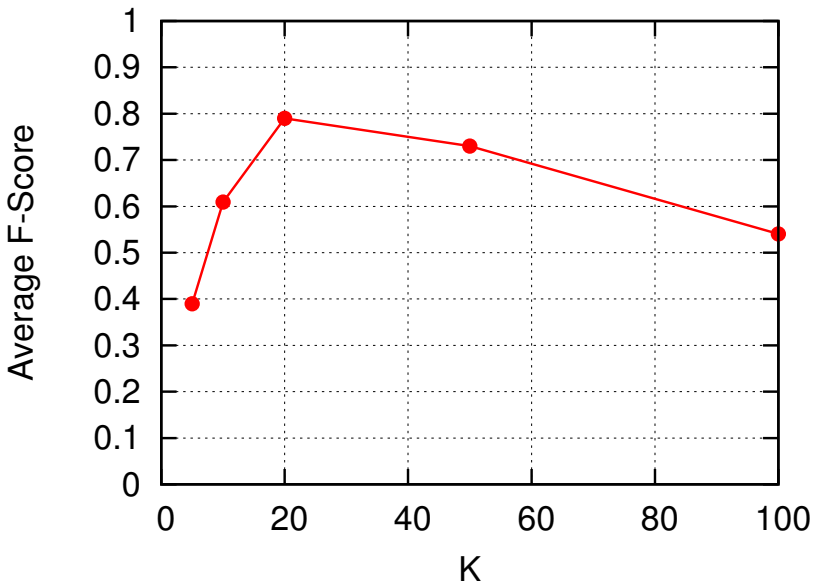


Figure 9: Distributed K-Means algorithm on a network of 100 nodes and data coming from exactly 20 distributions. The results are worse when given the wrong number of clusters

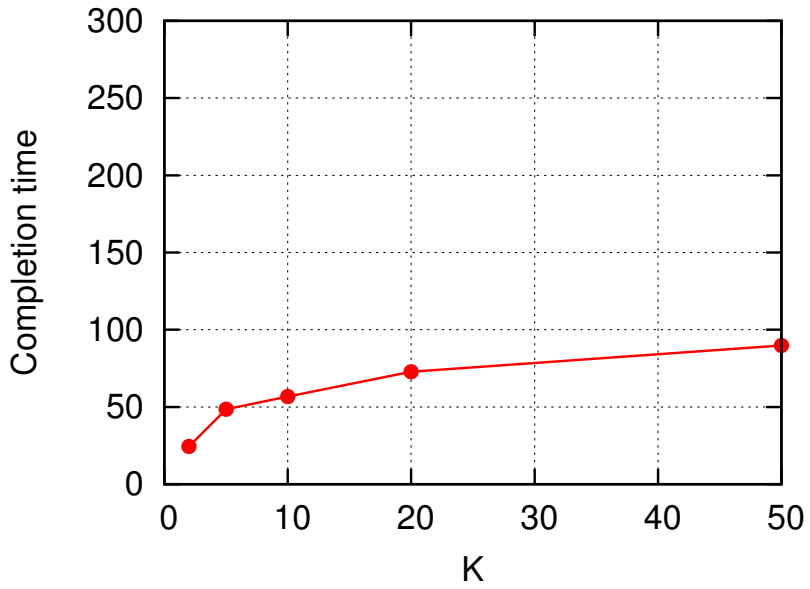


Figure 10: Completion time of the distributed K-Means algorithm on a network of 100 nodes and data coming from K distributions.

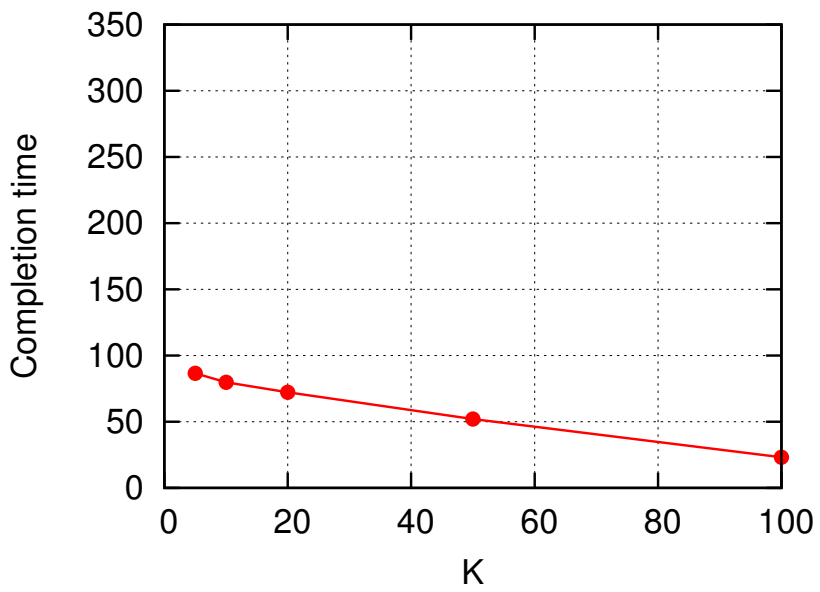


Figure 11: Completion time of the distributed K-Means algorithm on a network of 100 nodes and data coming from 20 distributions.

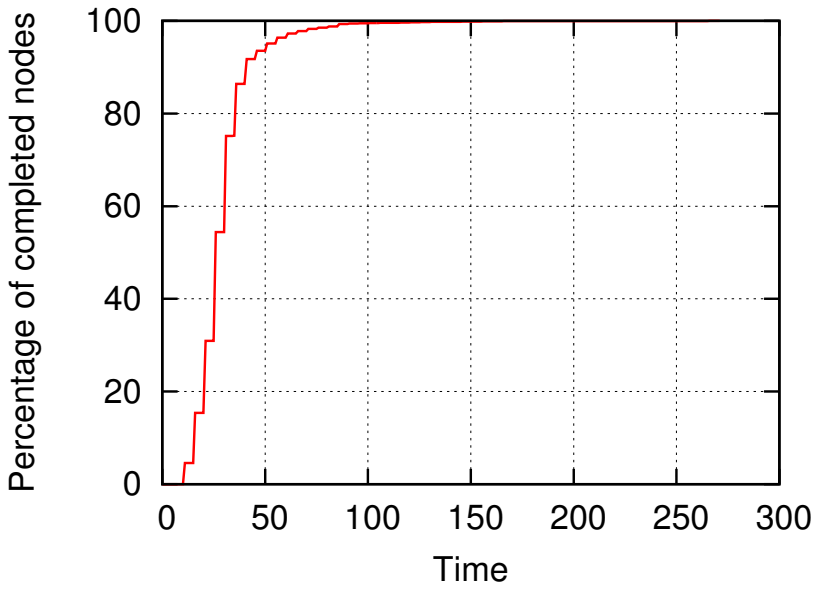


Figure 12: Percentage of nodes that have completed the distributed K-Means algorithm, 100 nodes, K=20

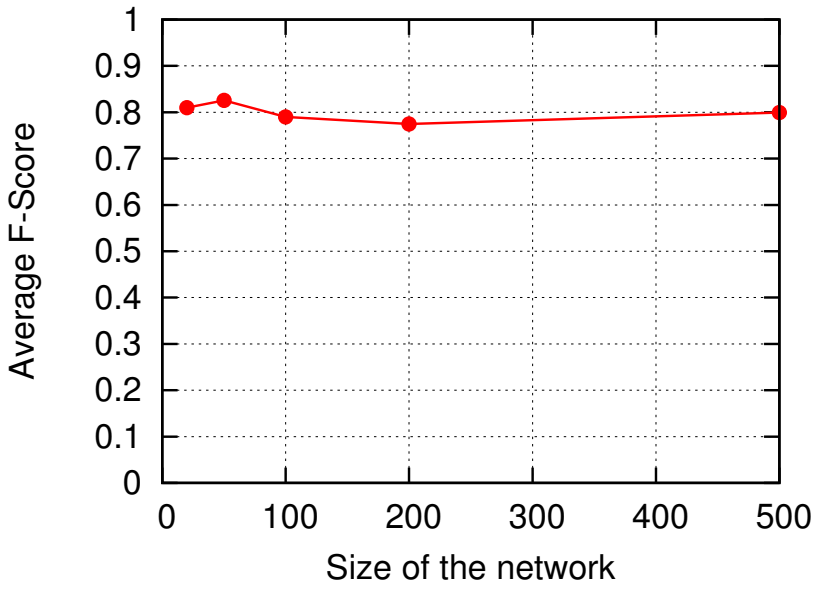


Figure 13: Scalability of the system: F-Score of the distributed K-Means algorithm when the size of the network grows

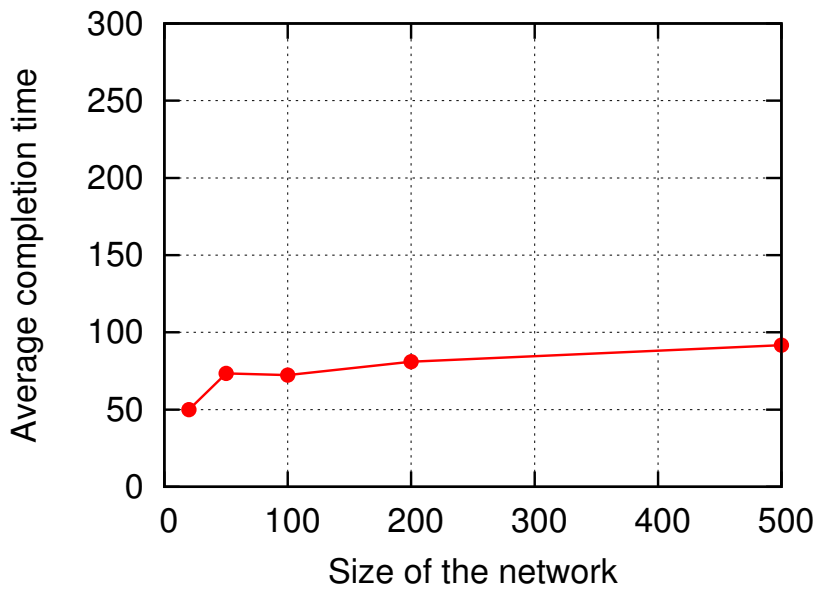


Figure 14: Scalability of the system: completion time of the distributed K-Means algorithm when the size of the network grows

Looking at the completion time of the algorithm we can see in figure 14 that when the size of the network increases the completion time grows quite slowly. When the number of nodes increases from 100 to 500 we only see a 25% increase in the completion time of the algorithm.

Summarizing the characteristics of this distributed version of K-Means we can say that, when given the correct number of clusters, it is able to find a clustering with very high F-Score, while when it is given the wrong number of clusters the algorithm suffers greatly. When we use it in our more complex clustering algorithm we will need to add a barrier to make sure that all nodes have terminated. We also saw that we should not worry too much about the size of the network since the algorithm has shown good scalability in our simulations.

4.4 OUR DECENTRALIZED CLUSTERING ALGORITHM

When we run simulations to test our decentralized clustering algorithm we generate the data using the same parameters as before and we deliver it in ten chunks. As before we compute the average F-Score of the clustering obtained in the network and we compare it against the clustering obtained by the centralized K-Means and the centralized X-Means.

4.4.1 *F-Score*

In figure 15 we compare the F-Score of our decentralized algorithm and the two centralized clustering algorithm. The algorithm is run in a network of 100 nodes with chunks of different sizes on data coming from 20 different distributions. We can see

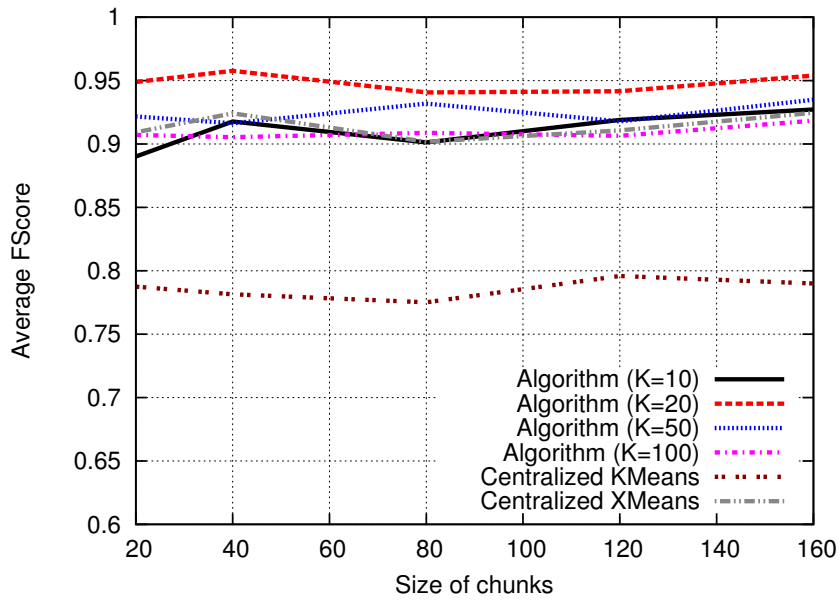


Figure 15: F-Score of our decentralized clustering algorithm given different Ks against two centralized algorithms, network size equal to 100, data from 20 distributions

F-Score of our algorithm

that the centralized K-Means algorithm obtains a clustering of lower quality, even when it is given the correct number of clusters. This is caused by the usual problem of the presence of local minima. The centralized X-Means gives a much better F-Score because of its ability to avoid local minima. Our distributed clustering algorithm obtains a very good F-Score even when it is not given the correct number of clusters. We can see that while our clustering algorithm gets better results when it is given the exact number of clusters, when we give it different Ks we still obtain clustering of quality comparable to the one computed by X-Means.

Amount of computation

Another interesting property of our algorithm is the amount of computation (measured by the number of calls to the distance function) that is needed to complete the execution. In figure 16 we can see that the quantity calls needed depends from the K given to the algorithm. The bigger the K, the greater the number of calls to the function. The global amount of computation needed by our clustering algorithm is always comparable to the centralized algorithm.

Completion time

In figure 17 we can see that, even if the amount of computation is bigger, the completion time (in which we only take into account the communication costs) seems lower when the algorithm is given a bigger K. To measure the actual running time of the algorithm we would need a way to compare the computation time (as measured by the number of distance function calls) against the communication time.

While we have seen that the precision of our algorithm is high even when given the incorrect number of clusters we may want to see which is the actual number of clusters computed by our algorithm. In figure 18 we show the number of clusters found by our decentralized clustering algorithm when given different values of K, while the data comes from 20 different distributions. The results are quite interesting:

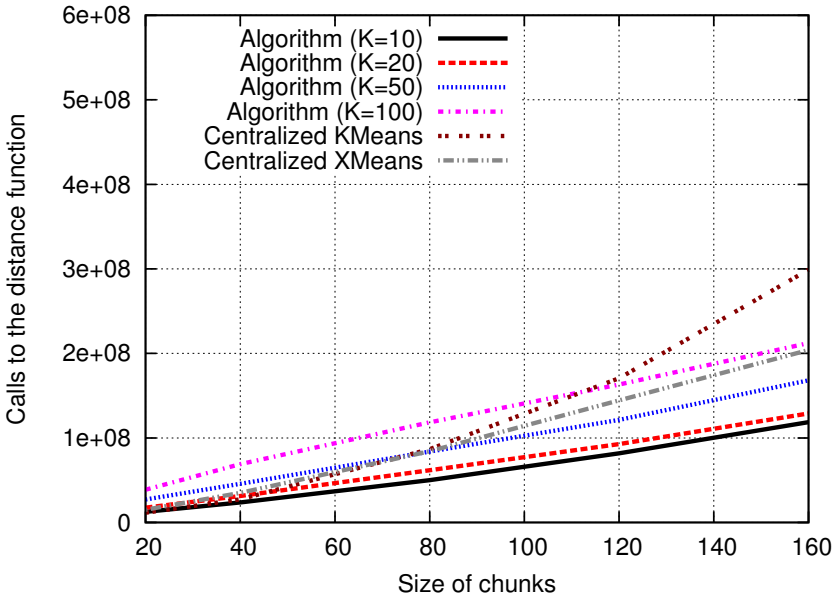


Figure 16: Number of global calls to the distance function by the decentralized clustering algorithm against two centralized algorithms, 100 nodes, 20 distributions

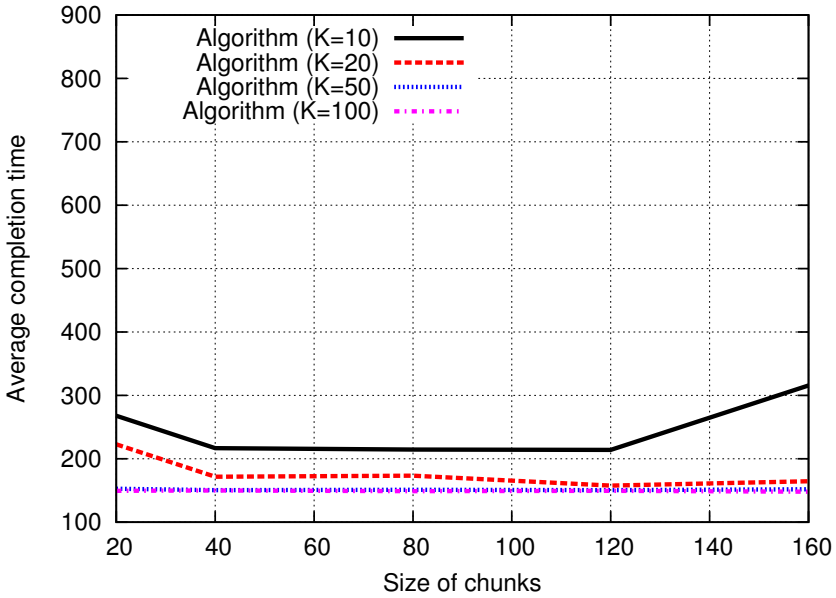


Figure 17: Completion time of the decentralized clustering algorithm with different K, 100 nodes, 20 distributions

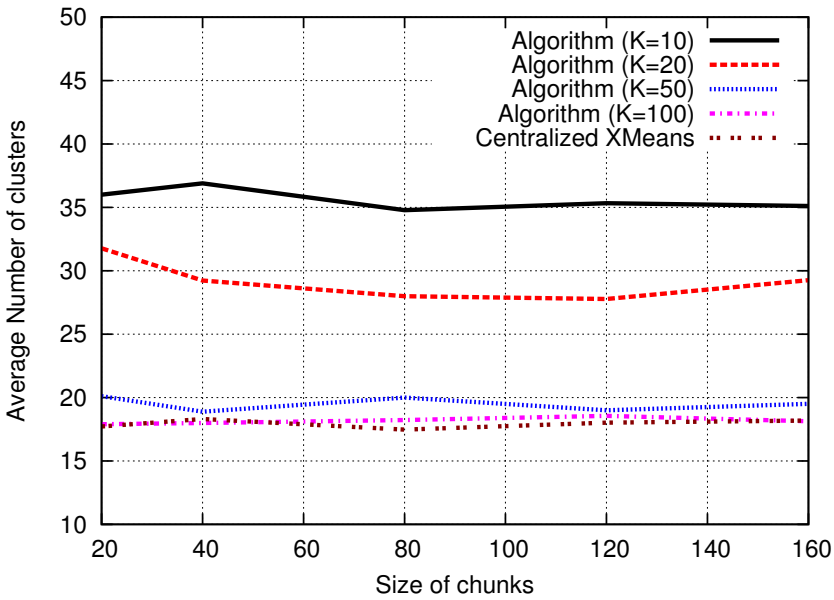


Figure 18: Average number of clusters of the decentralized clustering algorithm with different K, 100 nodes, 20 distributions

Number of clusters found

the algorithm is closer to the correct answer when it is given K bigger than necessary while when it is given the correct K the number of clusters it creates is bigger than necessary. This behaviour is easily explained: giving a bigger K to the distributed K-Means algorithm results in a bigger number of centroids generated and thus a better representation of the data. When K is small then it is more likely that in the distributed K-Means algorithm more than one cluster is mapped to a single centroid, thus creating a new point in the list of centroids that does not corrispond to any of the 20 original clusters. The reason the algorithm is still very precise is that these centroids, being quite far from the real distributions, will not be used when mapping the data of the entire network to the clusters.

4.4.2 *Scalability*

As the next step we want to see the behaviour of our distributed clustering algorithm when the network grows in size. In figure 19 we see the F-Score of the algorithm when ran on data from 20 distributions with K equal to 40. We can see that the F-Score remains constant even when the number of nodes in the network grows.

Scalability of the algorithm

In figure 20 we see, for the same experiment, the completion time of our clustering algorithm against the number of nodes in the network. We see that the differences are very small and that we need a similar amount of communication rounds to complete the algorithm. This fact was expected since the nodes in the network only look and communicate to their local neighbours and the data is distributed homogeneously in the network.

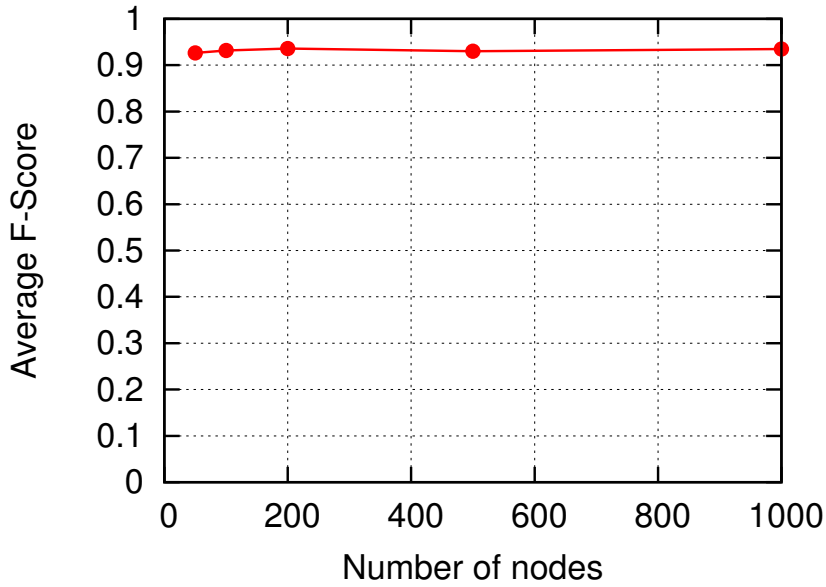


Figure 19: Scalability: F-Score of our decentralized clustering algorithm given K=40 and data from 20 distribution

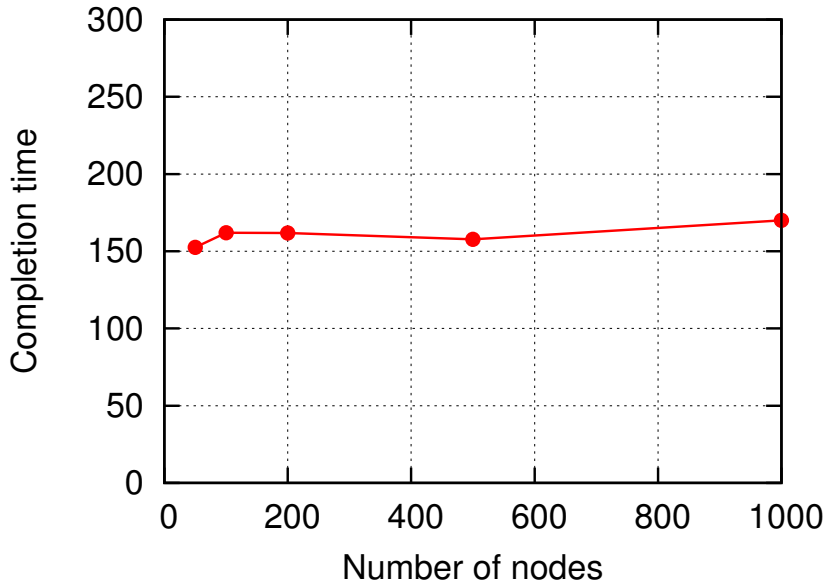


Figure 20: Scalability of the system: completion time of our decentralized clustering algorithm given K=40 and data from 20 distribution

4.4.3 *Robustness*

As the last step we did some analysis on the robustness of our algorithm. In our model the nodes may go down for a period of time before getting repaired and rejoining the distributed algorithm. Nodes that are down do not communicate with the rest of the network and do not work on their data. They are still able to receive a new chunk and store it until the node goes up. This model has been chosen to make sure that each node has the data on which it has to work.

Definition of the model to test robustness of our algorithm

Our model for robustness uses two parameters: the probability of failure (PF) and the probability of recovery (PR). All nodes start the computation in the up state and then, every iteration of the simulation, we use the following procedure:

- All nodes in the up state go down with probability PF.
- All nodes in the down state go up with probability PR.

This simple model is able to capture many different behaviour, as shown in figure 21. When this process has reached a stable state it will keep active a fraction p of the nodes in the network according to following formula:

$$p \cdot PF = (1 - p) \cdot PR$$

We obtain a steady state when the average number of up nodes that fail is equal to the average number of down nodes that recover. Solving for p gives the following:

$$p = \frac{PR}{PF + PR}$$

In figure 21 we can see that if we set $PR = PF = 0.1$ or $PR = PF = 0.2$ we reach the same steady state. The difference is that the greater are the two parameters the quicker the system is to reach it from the initial state.

In figure 22 we show the F-Score of our algorithm with different values of the two parameters. This simulation has been done using a network of 100 nodes with data from 20 different distribution and $K = 40$. Even with very high failure percentage and low recovery rate the algorithm is able to reach roughly the same F-Score. Note that in the case in which the probability of recovery is 0.1 and the probability of failure is 0.9 on average 90% of the nodes in the network is down at all times. The algorithm is still able to reach a good clustering even in these extreme conditions.

Our algorithm is precise even with high failure rate

These positive results come with a steep price in the number of communication rounds needed to complete. As we can see in figure 23, when the recovery probability is low the time needed to complete the execution of the algorithm grows very quickly with the failure probability. Still, with the recovery probability equal to 0.5, meaning that nodes do not stay down for long, the computation time does not grow too much. We can conclude that for the algorithm to really suffer we need that nodes go down with high frequency and for prolonged periods of time. In the other cases the completion time is not too strongly affected by the leaving and joining of nodes.

The completion time might suffer from high failure rate

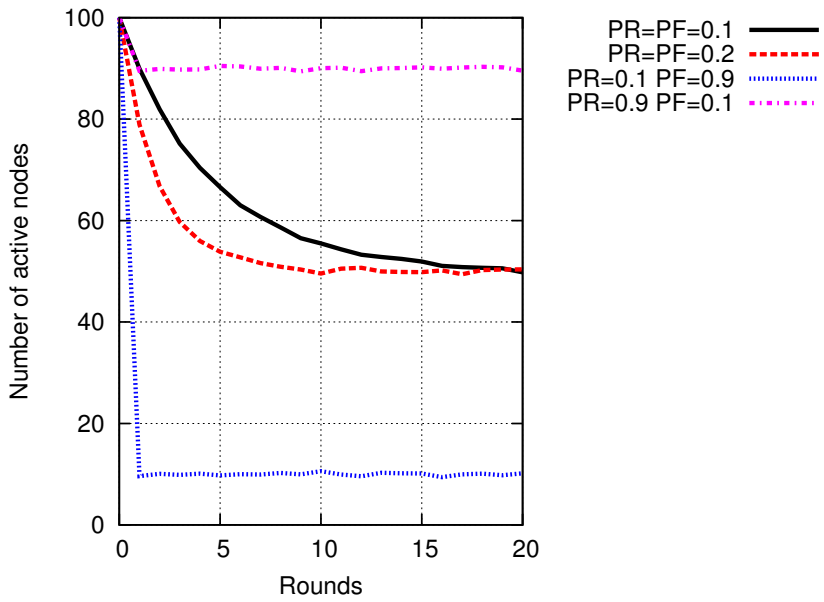


Figure 21: Robustness example: percentage of active nodes with different parameters

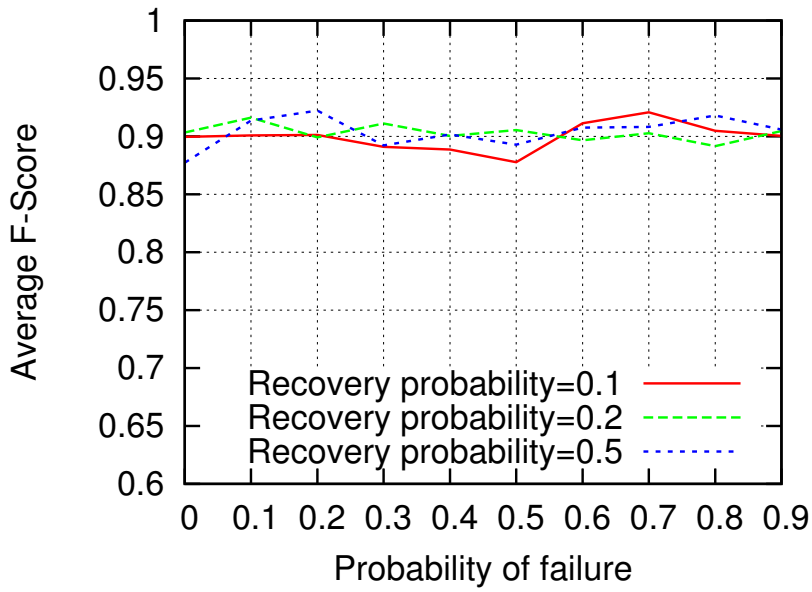


Figure 22: Robustness of our algorithm: F-Score of our decentralized clustering algorithm given K=40 and data from 20 distribution in a network of 100 nodes. We plot the results using three different values for the recovery probability against the failure probability

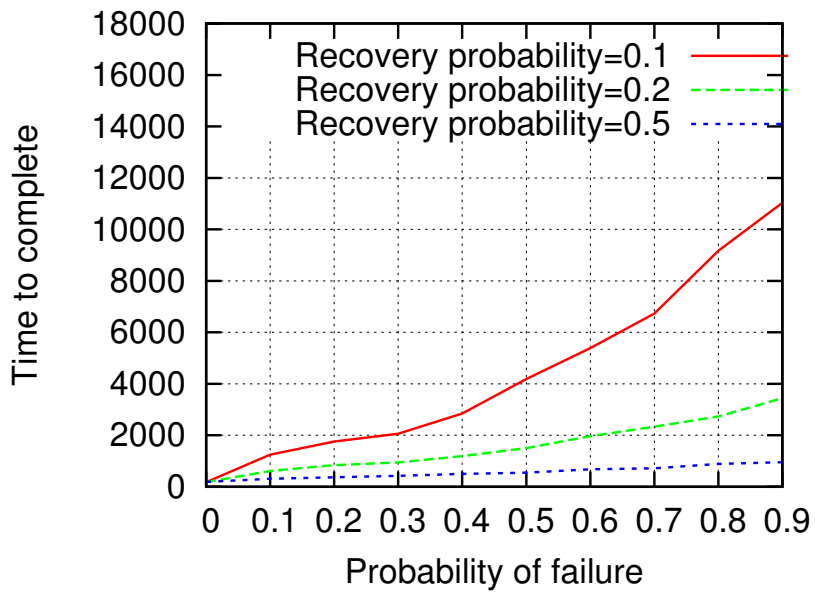


Figure 23: Robustness of the algorithm: F-Score of our decentralized clustering algorithm given K=40 and data from 20 distribution in a network of 100 nodes. We plot the results using three different values for the recovery probability against the failure probability

4.4.4 *Conclusions*

As we have seen in this chapter our decentralized clustering algorithm shows many good properties. It is very precise, with a F-Score comparable to the F-Score of the best centralized algorithm we implemented. The computation is well shared between the nodes in the network and the total number of calls to the distance function in the network is equivalent to the number of calls made by the centralized algorithm. Our algorithm does overestimate the number of cluster when given K equal or smaller than the correct value but the precision still remains high. The algorithm scales well with the dimension of the network and keeps the convergence time mostly constant. If we allow nodes to go down for a small period of time the algorithm still returns a good clustering while not paying too much in terms of completion time.

5 CHAPTER 5:CONCLUSIONS

Summary

This thesis introduced a decentralized clustering algorithm able to work even without a precise estimate for the number of clusters. The data is given independently to each node in the network and they collaborate to obtain a common clustering without directly sharing the points and only sending aggregate data. The nodes execute a distributed version of K-Means on each chunk of data they receive and use the results of all instances of K-Means as a compact representation of the data of the entire network. An algorithm called X-Means is then executed on this representation to obtain an estimate of the number of clusters. Simulations on synthetic data show that our algorithm is precise, scalable and robust.

Future work

There are many directions in which this study may continue. The online aspect of the distributed clustering algorithm has still not being studied in detail. It would be potentially interesting to implement the online algorithm in PeerSim and run some simulations not only on data with a constant number of cluster but also on data with changing number of cluster. We could model a setting in which distribution are born, change during time and finally disappear. It would be interesting to see if our algorithm is able to react to the change in the underlying distribution and detect the fact that the number of clusters may have changed.

We also need to test this algorithm on real data, not on data simply generated from a gaussian distribution. We plan to apply our algorithm with the aim of detecting botnets using the example of BotMiner [11] when we will be able to obtain usable data. Before applying our algorithm on real data, we may need to make more tests on data of higher dimensionality.

Finally, in our algorithm we use the X-Means algorithm on the list of centroids to obtain an estimate on the number of clusters in the data. This is not the only known algorithm to get these type of results and it would be interesting to see what happens when we change the algorithm at this step.

THANKS

I can't thank half of you half as well as I would like; and I have less than half the time I need to thank you half as well as you deserve. So I will be very brief.

Thanks to my parents, my sister, my uncles and aunts, all my cousins, my grandparents, my parents's cousins, my second cousins, my cousins' cousins and all the people I know that are related to me in some obscure way. Thanks to professor Montresor, who read a badly written thesis while in a different continent. Also thanks to professor James Naismith, who did not read this thesis but invented the game of basketball and that's enough for me. Thanks to all the professors who teached me something during my studies in Trento, Atlanta and München. Finally, thanks to all my friends from far and near, life would be very boring without all of you.

BIBLIOGRAPHY

[1] H. Akaike. A new look at the statistical identification model. *IEEE transactions on Automatic Control*, 19(6):716–723, 1974. (Cited on page 22.)

[2] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, 2009. (Cited on page 13.)

[3] D. Arthur, B. Manthey, and H. Roglin. k-means has polynomial smoothed complexity. pages 405–414, 2009. (Cited on page 14.)

[4] D. Arthur and S. Vassilvitskii. How slow is the k-means method? pages 144–153, 2006. (Cited on page 14.)

[5] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. pages 1027–1035, 2007. (Cited on page 16.)

[6] Sanghamitra Bandyopadhyay, Chris Giannella, Ujjwal Maulik, Hillol Kargupta, Kun Liu, and Souptik Datta. Clustering distributed data streams in peer-to-peer environments. *Information Sciences*, 176(14):1952 – 1985, 2006. Streaming Data Mining. (Cited on page 30.)

[7] J.C. Bezdek. Pattern recognition with fuzzy objective function algorithms. 1981. (Cited on page 12.)

[8] D. Borthakur. The hadoop distributed file system: Architecture and design. *retrieved fromlucene. apache. org/hadoop*, 2007. (Cited on page 26.)

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. (Cited on page 26.)

[10] R.O. Duda, P.E. Hart, and D.G. Stork. Pattern classification. 2, 2001. (Cited on page 14.)

[11] G. Gu, R. Perdisci, J. Zhang, and W. Lee. Botminer: Clustering analysis of network traffic for protocol-and structure-independent botnet detection. pages 139–154, 2008. (Cited on pages 7, 9, and 53.)

[12] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering*, pages 515–528, 2003. (Cited on page 33.)

[13] M. Inaba, N. Katoh, and H. Imai. Applications of weighted voronoi diagrams and randomization to variance-based k-clustering:(extended abstract). page 339, 1994. (Cited on page 13.)

[14] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252, 2005. (Cited on page 36.)

[15] M. Jelasity, A. Montresor, G.P. Jesi, and S. Voulgaris. Peersim: A peer-to-peer simulator. (Cited on page 39.)

[16] S.P. Lloyd. Least squares quantization in pcm. *IEEE Transactions On Information Theory*, 28(2), 1982. (Cited on pages 7 and 14.)

[17] M. Mahajan, P. Nimbhorkar, and K. Varadarajan. The planar k-means problem is np-hard. *Theoretical Computer Science*, 2010. (Cited on page 13.)

[18] P.C. Mahalanobis. On the generalized distance in statistics. 12:49, 1936. (Cited on page 12.)

[19] E. Ogston, M.V.A.N. STEEN, and F. Brazier. Group formation among decentralized autonomous agents. *Applied Artificial Intelligence*, 18(9):953–970, 2004. (Cited on page 25.)

[20] D. Pelleg and A. Moore. X-means: Extending k-means with e cient estimation of the number of clusters. page 727, 2000. (Cited on pages 7, 18, 20, and 41.)

[21] G. Schwarz. Estimating the dimension of a model. *The annals of statistics*, 6(2):461–464, 1978. (Cited on page 22.)

[22] H. Van Dyke Parunak, R. Rohwer, T. Belding, and S. Brueckner. Dynamic decentralized any-time hierarchical clustering. *Engineering Self-Organising Systems*, pages 66–81, 2007. (Cited on page 24.)