Efficient Approaches for Solving the Large-Scale k-medoids Problem

Alessio Martino, Antonello Rizzi and Fabio Massimo Frattale Mascioli
Department of Information Engineering, Electronics and Telecommunications, University of Rome "La Sapienza", Via Eudossiana 18, 00184 Rome, Italy

Keywords: Cluster Analysis, Parallel and Distributed Computing, Large-Scale Pattern Recognition, Unsupervised Learning, Big Data Mining.

Abstract: In this paper, we propose a novel implementation for solving the large-scale k-medoids clustering problem. Conversely to the most famous k-means, k-medoids suffers from a computationally intensive phase for medoids evaluation, whose complexity is quadratic in space and time; thus solving this task for large datasets and, specifically, for large clusters might be unfeasible. In order to overcome this problem, we propose two alternatives for medoids update, one exact method and one approximate method: the former based on solving, in a distributed fashion, the quadratic medoid update problem; the latter based on a scan and replacement procedure. We implemented and tested our approach using the Apache Spark framework for parallel and distributed processing on several datasets of increasing dimensions, both in terms of patterns and dimensionality, and computational results show that both approaches are efficient and effective, able to converge to the same solutions provided by state-of-the-art k-medoids implementations and, at the same time, able to scale very well as the dataset size and/or number of working units increase.

1 INTRODUCTION

The recent explosion of interest for Data Mining and Knowledge Discovery put *cluster analysis* on the spot again. Cluster analysis is the very basic approach in order to extract regularities from datasets. In plain terms, clustering a dataset consists in discovering clusters (groups) of patterns such that similar patterns will fall within the same cluster whereas dissimilar patterns will fall in different clusters.
Indeed, not only different algorithms, but different families of algorithms have been proposed in literature: partitional clustering (e.g. k-means (MacQueen, 1967; Lloyd, 1982), k-medians (Bradley et al., 1996), k-medoids (Kaufman and Rousseeuw, 1987)), which break the dataset into k non-overlapping clusters; density-based clustering (e.g. DBSCAN (Ester et al., 1996), OPTICS (Ankerst et al., 1999)) which detect clusters as the most dense regions of the dataset; hierarchical clustering (e.g. BIRCH (Zhang et al., 1996), CURE (Guha et al., 1998)) where clusters are found by (intrinsically) building a dendrogram in either a top-down or bottom-up approach.

Moreover, in the *Big Data* era, the need for clustering massive datasets emerged. Efficient and rather easy-to-use large-scale processing frameworks such as MapReduce (Dean and Ghemawat, 2008) or Apache Spark (Zaharia et al., 2010) have been proposed, gaining a lot of attention from Computer Science and Machine Learning researchers alike. As a matter of fact, several large-scale Machine Learning algorithms have been grouped in MLlib (Meng et al., 2016), the Machine Learning library built-in in Apache Spark.

In this work, we will focus on the k-medoids algorithm, a (hard) partitional clustering algorithm similar to the widely-known k-means. However, conversely to k-means, there is still some debate about a k-medoids parallel and distributed version due to the fact that the medoid evaluation complexity is quadratic in space and time and might therefore be unsuitable for large clusters. More into details, we propose a novel k-medoids implementation based on Apache Spark with two different procedures for medoids evaluation: an *exact* procedure and an *approximate* procedure. The former, albeit exact, requires the entire intra-cluster distance matrix evaluation which, as already introduced, might be unsuitable for large clusters. The latter overcomes this problem by adopting a scan-and-replace workflow, but returns a (sub)optimal medoid amongst the patterns in the cluster at hand.

The remainder of the paper is structured as follows: Section 2 will summarise the k-medoids prob-

Martino A., Rizzi A. and Frattale Mascioli F.
Efficient Approaches for Solving the Large-Scale k-medoids Problem.
DOI: 10.5220/0006515003380347
In *Proceedings of the 9th International Joint Conference on Computational Intelligence (IJCCI 2017)*, pages 338-347
ISBN: 978-989-758-274-5

lem; in Section 3 the proposed approaches and related implementations will be described; in Section 4 the efficiency and effectiveness of both the proposed medoid evaluation routines will be evaluated and discussed; Section 5 will draw some conclusions.

1.1 Contribution and State of the Art Review

As introduced in the previous section, parallel and distributed *k*-means implementations not only have been proposed in MapReduce (Zhao et al., 2009), but it is also included in MLlib. As far as *k*-medoids is concerned, to the best of our knowledge, there are very few parallel and distributed implementations proposed in literature. In (Yue et al., 2016) an implementation based on Hadoop/MapReduce has been proposed where, similarly to (Zhao et al., 2009), the Map phase evaluates the point-to-medoid assignment, whereas the Reduce phase re-evaluates the new medoids. Results show how the proposed implementation scales well with the dataset size, while it lacks of considerations regarding the effectiveness. In (Arbelaez and Quesada, 2013) the *k*-medoids problem is decomposed (space partitioning) in many local search problems which are solved in parallel. In (Jiang and Zhang, 2014) the *k*-medoids problem is again solved using Hadoop/MapReduce with a three-phases workflow (Map-Combiner-Reduce). Albeit it is unclear how the new medoids are evaluated, the Authors show both efficiency and effectiveness. Specifically, the latter is measured on a single labelled dataset, thus casting the clustering problem (unsupervised by definition) as a post-supervised learning problem. Both (Yue et al., 2016) and (Jiang and Zhang, 2014) start from Partitioning Around Medoids (PAM), the most famous algorithm for solving the *k*-medoids problem (Kaufman and Rousseeuw, 2009): a greedy algorithm which scans all patterns in a given cluster and, for each point, checks whether using that point as a medoid further minimises the objective function (§2): if true, that point becomes the new medoid.

In this work, a *k*-medoids implementation based on Apache Spark will be discussed which, due to its caching functionalities, is more efficient than MapReduce, especially when dealing with iterative algorithms (§3.1).Moreover, rather than using PAM, the (large-scale) *k*-medoids problem will be solved with the implementation proposed in (Park and Jun, 2009) which has a very *k*-means-like workflow and therefore it is possible to rely on a very easy-to-parallelise algorithm flowchart.

However, the medoid(s) re-evaluation/update is a critical task since it needs to evaluate pairwise distances amongst patterns in a given cluster in order to find the new medoid, namely the element which minimises the sum of distances. In order to overcome this problem, we propose two alternatives: an exact medoid evaluation based on cartesian product (suitable for small/medium clusters) and an approximate tracking method (suitable for large clusters as well), proposed in (Del Vescovo et al., 2014) but never tested on large-scale scenarios.

2 THE *k*-MEDOIDS CLUSTERING PROBLEM

k-medoids is a hard partitional clustering algorithm; it aims in partitioning the dataset $S = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$ into k non-overlapping groups (clusters), i.e. $S = \{S_1, ..., S_k\}$ such that $S_i \cap S_j = \emptyset$ if $i \neq j$ and $\cup_{i=1}^{k} S_i = S$.
In order to find the optimal partition, *k*-medoids tries to minimise the following objective function, namely the Within-Clusters Sum-of-Distances:

$$WCSoD = \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} d(\mathbf{x}, \mathbf{m}(i))^2 \tag{1}$$

where $d(\cdot,\cdot)$ is a suitable (dis)similarity measure, usually the standard Euclidean distance, and $\mathbf{m}(i)$ is the medoid for cluster i. In this work, a recent implementation based on the Voronoi iteration is adopted and its main steps can be summarised as follows:

1. select an initial set of k medoids
2. Expectation Step: assign each data point to closest medoid
3. Maximisation Step: update each clusters' medoid
4. Loop 2–3 until either medoids stop updating (or their update is negligible) or a maximum number of iterations is reached

It is worth stressing that minimising the sum of squares (1) is an NP-hard problem (Aloise et al., 2009) and therefore all methods (heuristics) proposed in literature (both in terms of efficiency and effectiveness) strictly depend on the initial medoids: this leaded to the implementation of initialisation heuristics such as *k*-means++ (Arthur and Vassilvitskii, 2007) or DBCRIMES (Bianchi et al., 2016).

It has already been discussed that evaluating the medoid is more complex than evaluating the mean, since it requires the complete pairwise distance matrix D between all points in a given cluster S_h, formally defined as an $|S_h| \times |S_h|$ real-valued matrix whose generic entry is given by

$$\mathbf{D}_{i,j}^{(S_h)} = d(\mathbf{x}_i, \mathbf{x}_j) \tag{2}$$

or, as far as the Euclidean distance is concerned, dually definable in terms of Gram matrix as

$$\mathbf{D}_{i,j}^{(S_h)} = \sqrt{\mathbf{G}_{i,j}^{(S_h)}} = \sqrt{\langle \mathbf{x}_i - \mathbf{x}_j, \mathbf{x}_i - \mathbf{x}_j \rangle} \quad (3)$$

for any two patterns $\mathbf{x}_i$, $\mathbf{x}_j \in S_h$.

Despite its complexity, by minimising the sum of pairwise distances rather than the squared Euclidean distance from the average point (i.e. k-means), k-medoids is more robust with respect to noise and outliers. Moreover, k-medoids is particularly suited if the cluster prototype must be an element of the dataset (the same is generally not true for k-means) and it is applicable to ideally any input space, not necessarily metric, since there is no need to define an algebraic structure in order to compute the representative element of a cluster.

3 PROPOSED APPROACH

3.1 A Quick Spark Introduction

In this work Apache Spark (hereinafter simply Spark) has been chosen over Hadoop/MapReduce due to its efficiency, as introduced in §1.1. Indeed, MapReduce forces any algorithm pipeline into a sequence of Map $\rightarrow$ Reduce steps, eventually with an intermediate Combiner phase, which strains the implementation of more complex operations such as *join*s or *filter*s as they have to be casted as well into Map $\rightarrow$ Reduce. In MapReduce, computing nodes do not have memory of past executions: this does not suit the implementation of iterative algorithms, as the master process must forward data to workers at each MapReduce job, which typically corresponds to a single iteration of a given algorithm.

Spark overcomes these problems as it includes highly efficient *map*, *reduce*, *join*, *filter* (and many others) operations which are not only natively distributed, but can be arbitrarily pipelined together. As far as iterative algorithms are concerned, Spark offers caching in memory and/or disk, therefore there is no need to forward data back and forth from/to workers at each iteration.

Atomic data structures in Spark are the so-called Resilient Distributed Datasets (RDDs): distributed (across workers[1]) collection of data with fault-tolerance mechanisms, which can be created starting from many sources (distributed file systems, databases, text files and the like) or by applying *transformations* on other RDDs.

Example of transformations which will turn useful in the following subsections are:

$map()$: $RDD2 = RDD1.map(f)$
creates $RDD2$ by applying function f to every element in $RDD1$

$filter()$: $RDD2 = RDD1.filter(pred)$
creates $RDD2$ by filtering elements from $RDD1$ which satisfy predicate $pred$ (i.e. if $pred$ is $True$)

$reduceByKey()$: $RDD2 = RDD1.reduceByKey(f)$
creates $RDD2$ by merging according to function f all values for each key in $RDD1$: $RDD2$ will have the same keys as $RDD1$ and a new value obtained by the merging function.

Similarly, examples of *actions* which can be applied to RDDs are:

$count()$: $RDD.count()$
count the number of elements in RDD

$collect()$: $RDD.collect()$
collects in-memory on the master node the entire RDD content.

3.2 Main Algorithm

The main parallel and distributed k-medoids flowchart is summarised in the Algorithm 1.
It is possible to figure `datasetRDD` to be a key-value pair RDD[2] or, in other words, an RDD where each record has the form $\langle$key ; value$\rangle$. Specifically, the key is a sequential integer ID and the value is (as a first approach) an n-dimensional real-valued vector.

At the beginning of each iteration, a new key-value pair RDD will be created (`distancesRDD`) where the key is still the pattern ID whereas, the value (`distanceVector`) is a k-dimensional real-valued vector where the i^{th} element contains the distance with the i^{th} medoid. The following two RDDs (`nearestDistRDD` and `nearestClusterRDD`) map the pattern ID with the nearest distance value and the nearest cluster ID (i.e. medoid), respectively. These evaluations end the Maximisation Step for the Voronoi iteration (§2).

`nearestClusterRDD` is the main RDD for the medoids update step (i.e. the Expectation Step for the Voronoi iteration): after gathering the list

[1] In Spark, *workers* are the elementary computational units, which can be either single cores on a CPU or entire computers, depending on the environment configuration (i.e. the *Spark Context*).

[2] In the following subsections and pseudocodes every variable whose name ends with `RDD` shall be considered as an RDD, therefore a data structure whose shards are distributed across the workers or, similarly, with no guarantee that a single worker has its entire content in memory.

of pattern IDs (`patternsInClusterIDs`) belonging to cluster ID `clID`, an additional RDD containing the patterns belonging to `clID` will be created (`patternsInClusterRDD`). This RDD, which still is a key-value pair RDD, is the main input for the medoids update step which we shall discuss in more details in the following subsection.

```
Begin main
Load datasetRDD from source (e.g. text-
file);
Cache datasetRDD on workers;
Set maxIterations, k,
approximateTrackingFlag;
medoids = initialSeeds();
for iter in range(1:maxIterations)
    previousMedoids = medoids;
    distancesRDD = datasetRDD.map(d(
    pattern,medoids));
    nearestDistRDD = distancesRDD.map(min(
    distanceVector));
    nearestClusterRDD = distancesRDD.map(
    argmin(distanceVector));
    for clID in range(1:k)
        patternsInClusterIDs =
        nearestClusterRDD.filter(clusterID
         == clID);
        patternsInClusterRDD =
        nearestClusterRDD.filter(clusterID
         in patternsInClusterIDs);
        if approximateTrackinFlag is True
            medoids[clID] =
            approximateMedoidTracking();
        else
            medoids[clID] =
            exactMedoidUpdate();
    WCSoD = nearestDistRDD.values().sum();
    if stopping criteria == True
        break;
End main
```

Algorithm 1: Pseudocode for the main *k*-medoids skeleton (namely, Voronoi iterations).

3.3 Updating Medoids

3.3.1 Exact Medoid Update

The Exact Medoid Update consists in evaluating the pairwise distance matrix amongst all the patterns in a given cluster and then finding the element which minimises the sum by rows/columns[3] of such matrix.

We define a-priori a parameter T, a threshold value which states the maximum allowed cluster cardinality in order to consider such cluster as "small". This parameter can be tuned considering both the expected clusters' sizes and the amount of memory available on the master node.

```
Begin function exactMedoidUpdate(
patternsInClusterRDD)
if patternsInClusterRDD.count() < T
    patterns = patternsInClusterRDD.
    collect().asmatrix();
    distanceMatrix = pdist(patterns,d);
    Sum distanceMatrix by rows or columns;
    medoidID = argmin of rows/columns sum;
    return patterns[medoidID];
else
    pairsRDD = patternsInClusterRDD.
    cartesian(patternsInClusterRDD);
    distancesRDD = pairsRDD.map(d(pattern1
    ,pattern2));
    distancesRDD = distancesRDD.
    reduceByKey(add);
    medoidID = distancesRDD.min(key=dist);
    return patternsInClusterRDD.filter(ID
    == medoidID).collect();
End function
```

Algorithm 2: Pseudocode for the Exact Medoid Update routine.

Algorithm 2 describes the Exact Medoid Update routine. Basically, if the cluster is "small" it is possible to collect on the master node the entire set of patterns and therefore use one of the many in-memory algorithms (`pdist`) for evaluating the pairwise distance matrix according to a given (dis)similarity measure `d`.

Conversely, if the cluster is "not-small", via Spark it is possible to evaluate the cartesian product of this cluster's RDD with itself, leading to a new RDD (`pairsRDD`) where each record is a pair of records from the original RDD, therefore it is possible to figure `pairsRDD` to have the form ⟨ID1 ; pattern1 ; ID2 ; pattern2⟩. Given these pairs, evaluating the (dis)similarity measure is straightforward and `distancesRDD` will have the form ⟨ID1 ; ID2 ; d(pattern1,pattern2)⟩. Given the analogy with the pairwise distance matrix, in order to perform the sum by rows or columns, `distancesRDD` will be reduced by using either the ID of the first pattern (by rows) or the ID of the second pattern (by columns) as key and using the addition operator to values. In other words, for ID1 (or ID2) we sum all the distances with other IDs (i.e. other patterns). Given these sums, the final step consists in evaluating the minimum of the resulting RDD considering the sum of distances rather than the ID leading to `medoidID`, the ID of the new medoid which will be filtered from `patternsInClusterRDD` and returned as the new, updated, medoid.

[3] In the following, we suppose that the (dis)similarity measure $d(\cdot,\cdot)$ is metric, therefore satisfies the symmetry property, i.e. $d(a,b) = d(b,a)$. For metric (dis)similarity measures the pairwise distance matrix is symmetric by definition and the sum by rows or columns leads to the same result.

3.3.2 Approximate Medoid Tracking

Whilst the (large scale) Exact Medoid Update (Algorithm 2, namely the *else* branch) relies on natively distributed and highly efficient Spark operations, the cartesian product might be unfeasible for very large clusters since it creates an RDD whose size is squared the size of the cluster (i.e. squared the size of the input RDD) according to Eqs. (2)-(3).

```
Begin function approximateMedoidTracking(
patternsInClusterRDD,medoids,clusterID)
if patternsInClusterRDD.count() < P
    patterns = patternsInClusterRDD.
    collect().asmatrix();
    distanceMatrix = pdist(patterns,d);
    Sum distanceMatrix by rows or columns;
    medoidID = argmin of rows/columns sum;
    return patterns[medoidID];
else
    pool = patternsInClusterRDD.filter(ID
    in range(1:P)).collect().asmatrix();
    patternsInClusterRDD =
    patternsInClusterRDD.filter(ID not in
    range(1:P));
    for pattern in patternsInClusterRDD
        Extract x1!=x2 from pool;
        if d(x1,medoids[clusterID])>=d(x2,
        medoids[clusterID])
            Remove x1 from pool;
        else
            Remove x2 from pool;
        pool = pool.append(pattern);
    distanceMatrix = pdist(pool,d);
    Sum distanceMatrix by rows or columns;
    medoidID = argmin of rows/columns sum;
    return pool[medoidID];
End function
```

Algorithm 3: Pseudocode for the Approximate Medoid Tracking routine.

To this end, a second, approximate, medoid evaluation based on (Del Vescovo et al., 2014) is proposed. For the sake of ease, their findings can be summarised as follows:

1. set a pool size P and fill the pool with the first P patterns from the cluster at hand
2. for every remaining pattern $\mathbf{x}$, select uniformly at random two items from the pool ($\mathbf{x}_1$ and $\mathbf{x}_2$) and check their distances with the current medoid $\mathbf{m}$. If $d(\mathbf{x}_1,\mathbf{m}) \geq d(\mathbf{x}_2,\mathbf{m})$, remove $\mathbf{x}_1$ from the pool; otherwise remove $\mathbf{x}_2$. Finally, insert $\mathbf{x}$ in the pool.
3. evaluate the new medoid with any standard in-memory routine using the patterns in the pool.

Algorithm 3 describes the Approximate Medoid Tracking routine. As in Algorithm 2, if the cluster is "small" there is not need to trigger any parallel and/or distributed medoid evaluation which can thus be done locally, in an exact manner. Conversely, if the cluster is "not-small" the first P items will be cached and removed from the RDD which contains the patterns in such cluster. Finally, the approximate procedure starts by iterating over the RDD and performing the steps from (Del Vescovo et al., 2014).

3.4 Selecting Initial Seeds

As discussed in §2, the solution returned by heuristics such as k-medoids is strictly dependent from the initial medoids (seeds) selection. One of the most widely-used approaches consists in random selection and multi-start: a given number (typically 5 or 10) of sets of k initial medoids are uniformly at random selected from the dataset and for each set of initial medoids the k-medoids algorithm will be run. The final clustering solution is determined by the run with minimum WCSoD. Other solutions have been proposed in literature such as k-means++ which we implemented for a more accurate seeds selection. In its standard form, k-means++ works as follows:

1. select the first centroid[4] uniformly at random from the dataset S
2. evaluate pairwise distances between patterns and already-selected centroid, mapping each pattern with its closest centroid; let us call $D(\mathbf{x})$ the closest distance between pattern $\mathbf{x} \in S$ and the already-selected centroids
3. a pattern from the dataset S is selected as new centroid with probability $\frac{D^2(\mathbf{x})}{\sum_{\mathbf{x}\in S} D^2(\mathbf{x})}$
4. Loop 2–3 until k initial centroids are selected

The Spark k-means++ distributed version is summarised in Algorithm 4.

Many of these RDDs have already been introduced in §3.2 and, since step #2 in k-means++ actually is the Maximisation step for the Voronoi iteration (§2), such RDDs will be created and evaluated in the same manner as described in Algorithm 1. One might object that collecting (i.e. loading into the master node memory) `nearestDistRDD` might not be suitable for large datasets. It is worth noticing that, in order to save space, only the distance values and not the IDs will be collected and, moreover, a modern computer can store several millions of elements within an in-memory array. Therefore (as a first approach) we do

[4] Historically, k-means++ was designed for k-means and therefore we will refer to the cluster representative as "centroid". However, since it returns k data points as initial seeds, the same procedure holds for "medoid".

not consider this operation as prohibitive.

```
Begin function initialSeeds(datasetRDD,k)
N = datasetRDD.count();
firstMedoidID = random.integer(1,N);
medoids = datasetRDD.filter(ID == firstMedoidID);
for iter in range(1:k-1)
    distancesRDD = datasetRDD.map(d(pattern,medoids));
    nearestDistRDD = distancesRDD.map(min(distanceVector));
    distances = nearestDistRDD.values().collect();
    distances = distances/sum(distances);
    newMedoidID = random.choice(range(1:N),prob=distances);
    medoids = medoids.append(datasetRDD.filter(ID == newMedoidID).collect());
End function
```

Algorithm 4: Pseudocode for k-means++.

4 COMPUTATIONAL RESULTS

4.1 Datasets and Environment Description

In order to show both the effectiveness and the efficiency of the proposed algorithm, 9 datasets have been considered. All of them are freely available from UCI Machine Learning Repository (Lichman, 2013) and their major characteristics are summarised in Table 1. Classes in labelled datasets (e.g. *Wine*, *Iris*) and eventual pattern IDs (e.g. *3D Road Network*) have been deliberately discarded. All datasets have been normalised before processing.

Table 1: List of datasets used for analysis.

Dataset Name	**# patterns**	**# features**
US Census 1990	2458285	68
3D Road Network	434874	4
KEGG Metabolic Relation Network (Directed)	53413	23
Bag of Words (ENRON)	39861	28102
Daily and Sport Activities	9120	5625
Bag of Words (KOS)	3430	6906
Bag of Words (NIPS)	1500	12419
Iris	178	13
Wine	150	4

Experiments shown in §4.2 and 4.3 have been conducted on a Linux Ubuntu 16.04LTS workstation with two Intel® Xeon® CPUs @2.60GHz for a total of 24 physical cores, 32GB RAM and 1TB HDD. Such 24 cores have been grouped in 6 groups of 4 cores each, in order to simulate 6 PCs working in parallel which will be the workers for these experiments. The latest Apache Spark version has been used (v2.2.0) and, specifically, its Python API (namely, *PySpark*) driven by Python v2.7.13 with NumPy v1.12.1 (van der Walt et al., 2011) for efficient numerical computations.

4.2 Effectiveness

In order to show the effectiveness of the proposed distributed implementation, the clustering problem has not been casted as a semi-supervised or post-supervised learning problem since, in our opinion, this is not a good way to address the effectiveness of a clustering algorithm (which solves unsupervised problems by definition) as one might tend to confuse *clusters* with *classes*. Rather, the solution (i.e. the objective function) returned by the distributed k-medoids implementation will be compared with the solution returned by a well-established and not distributed k-medoids implementation, specifically the one available in the MATLAB® R2015a.

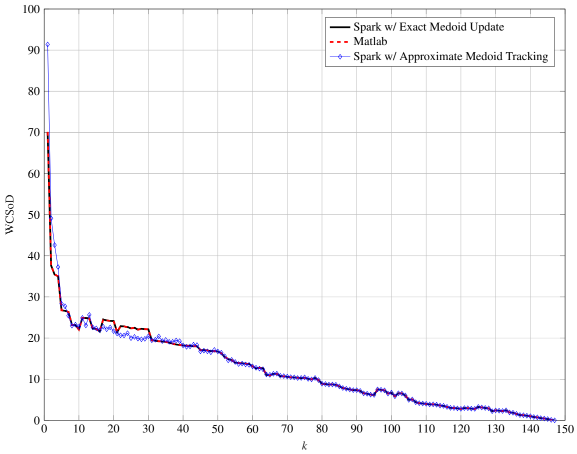
In Figure 1 the objective function returned by the distributed implementation is shown, considering both the Approximate Medoid Tracking and the Exact Medoid Update routines for medoids update, against the MATLAB k-medoids implementation. We considered the two smaller datasets (*Iris* and *Wine* in Figures 1(a) and 1(b), respectively) in order to be able to test every possible value for k; that is, from 1 up to the singleton (i.e. the total number of patterns). To ensure a fair comparison, the same (dis)similarity measure has been used (standard Euclidean distance) but, more importantly, the three competitors started from the very same initial seeds[5]. Since the Approximate Medoid Tracking has some randomness (i.e. its replacement policy), for every k the average objective function value across 5 runs is shown.

Clearly, Figure 1 shows that the proposed approach is effective. Specifically, when using the Exact Medoid Update, it returns the very same objective function value[6] when compared to the state-of-the-art (albeit local) MATLAB implementation. The Approximate Medoid Tracking has surprising results as well, especially for *Wine* (Figure 1(b)).

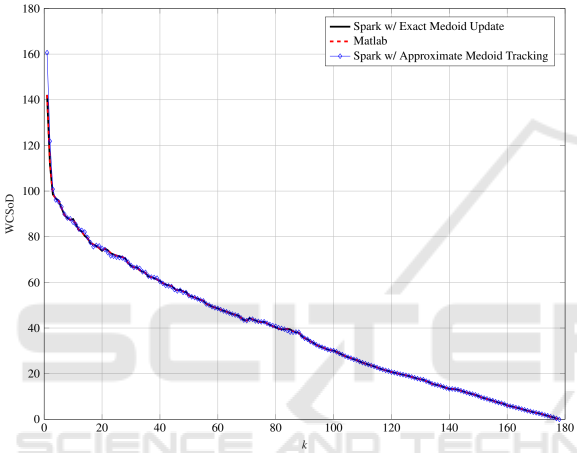
Similarly, in Figures 2(a) and 2(b) the results when using k-means++ rather than manual (i.e. deterministic) selection for initial medoids is shown. As in the previous case, the two figures regard *Iris* and *Wine* datasets, respectively.

[5] Specifically, for $k = i$ the first i patterns have been selected as initial seeds.

[6] Therefore returns the very same medoids and the very same point-to-medoid assignments.

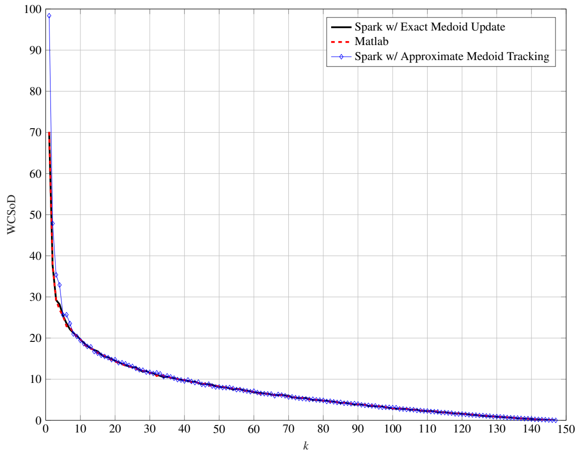


(a) Iris Dataset

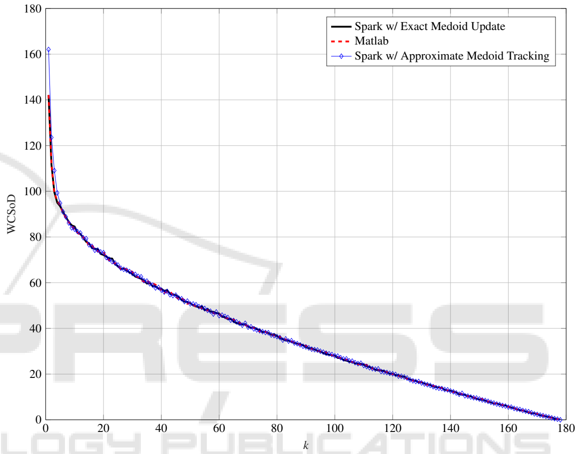


(b) Wine Dataset

Figure 1: Effectiveness tests with deterministic initial seeds selection.



(a) Iris Dataset



(b) Wine Dataset

Figure 2: Effectiveness tests with k-means++ initial seeds selection.

Figure 2 does not only confirm that the distributed k-medoids implementation is effective, but also that the distributed k-means++ implementation works very well. Obviously, since k-means++ has random behaviour, 5 runs have been performed for each competitor and the average objective function is shown.

Finally, it is worth stressing that a threshold of $T = 20$ patterns for both datasets has been used. Certainly the pool size for the Approximate Medoid Tracking has a major impact on the tracking quality itself: since the pool must fit in memory (as we expect from a "small cluster"), we dually used $T \equiv P$ to indicate the pool size as well. For the sake of ease and shorthand we omit any tests regarding how the medoid tracking quality and the procedure complexity changes as a function of the pool size: such results have been discussed in (Del Vescovo et al., 2014), to which we refer the interested reader.

4.3 Efficiency

In order to address the efficiency of the proposed implementation, the aim of this subsection is to show whether it respects the *speedup* and *sizeup* properties, defined as follows (Xu et al., 1999):

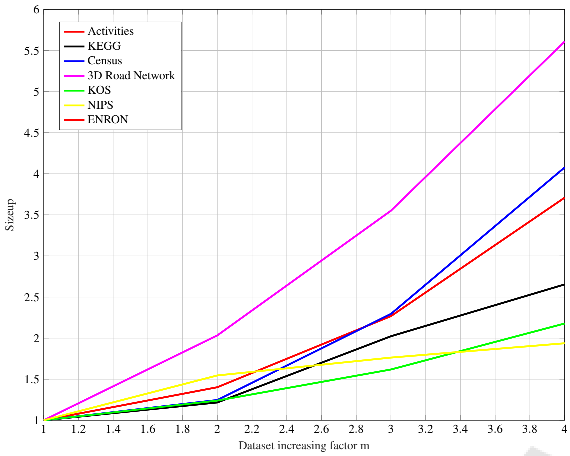
speedup measures the ability of the parallel and distributed algorithm to reduce the running time as more workers are considered. More formally, the dataset size is kept constant and the number of workers will increase from 1 to m. The speedup for an m-nodes cluster is defined as:

$$speedup(m) = \frac{\text{running time on 1 worker}}{\text{running time on } m \text{ workers}}$$

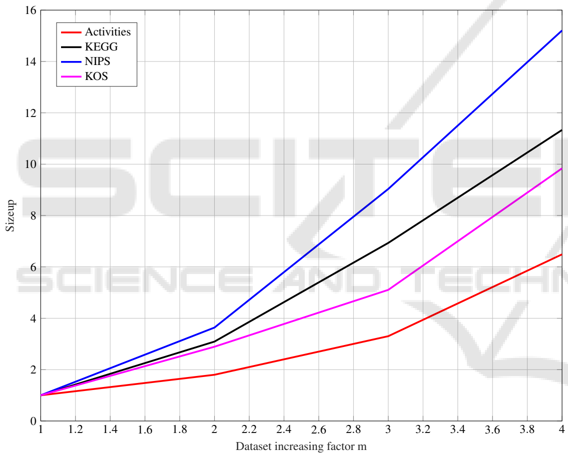
sizeup measures the ability of the parallel and distributed algorithm to manage an m times larger dataset while keeping the number of workers constant. The speedup for an m times larger dataset is defined as:

$$sizeup(S,m) = \frac{\text{running time for processing } m \times S}{\text{running time for processing } S}$$

Figure 3 shows the sizeup performances for both



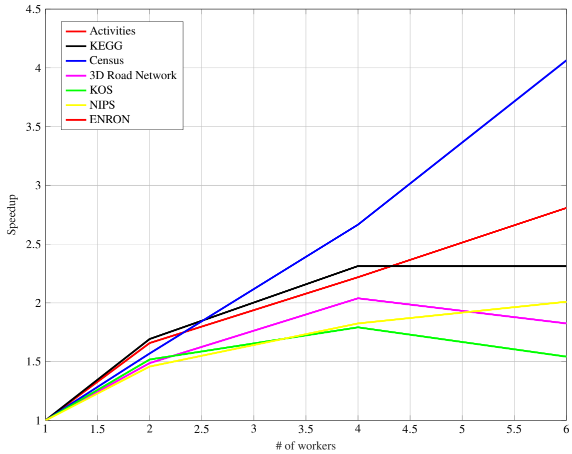
(a) Approximate Medoid Tracking



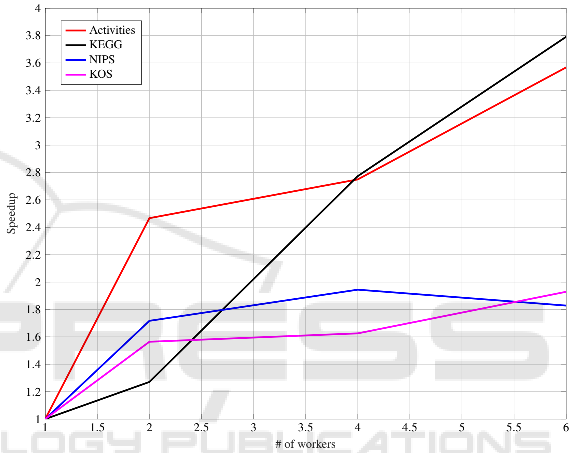
(b) Exact Medoid Update

Figure 3: Sizeup.

the medoid update routines. Specifically, the sizeup parameter m has been set to $m = 1,2,3,4$. All of the workers available and all but *Wine* and *Iris* datasets from Table 1 have been used. Figure 3(a) depicts the Approximate Medoid Tracking case which has a very good sizeup performances, especially for $m \geq 3$ (e.g. a 4-times larger dataset needs from 1.9 to 5.5 times more time). Figure 3(b) depicts the Exact Medoid Update case which has generally lower performances with respect to the former. However, as the dataset size increases, the performances tend to improve (e.g. for *KEGG* a 4-times larger problem needs 11-times more time, whereas for *Activities* it needs 6-times more time) but, on the other hand, this approach cannot be adopted for larger datasets under



(a) Approximate Medoid Tracking



(b) Exact Medoid Update

Figure 4: Speedup.

reasonable running times[7].

For the speedup, 1, 2, 4 and 6 workers and the same datasets as for the sizeup experiment have been used.

The algorithm has very good speedup performances when using the Approximate Medoid Update, Figure 4(a): the bigger the dataset, the better the speedup. For some datasets (typically the smallest ones) implying more than 4 workers does not improve the overall performances; indeed, since they are rather small and due to the Approximate Medoid Tracking routine being computationally lighter (§4.4), a massive parallelisation will unlikely be helpful as most of the time will be spent in scheduling/communication tasks rather than pure processing.

Figure 4(b), finally, shows the Exact Medoid Update case, which has very good speedup performances

[7]Whilst using all of the workers available, updating the medoid for a 100000-patterns cluster took more than 3 hours.

as well. Indeed, due to the intensive parallel evaluations required for solving the exact evaluation, more workers are crucial for speeding-up such phase. The above reasoning regarding smaller datasets still holds.

For the sake of completeness and comparison, in Table 2 the absolute running times for the *Activities* dataset have been reported.

Table 2: Absolute running times in minutes.

# workers	Approximate	Exact
1	2	30
2	1.3	13
4	0.97	11
6	0.76	8.5

4.4 On Computational Complexity

An additional reason for choosing the implementation proposed in (Park and Jun, 2009) over other implementations such as the most famous PAM for solving the k-medoids problem lies in its complexity. Indeed, since the chosen implementation is based on the Voronoi iterations as in k-means, the two algorithms have the same complexity, that is $O(nk)$ per-iteration, where n is the number of data points and k is the number of clusters. For the sake of comparison, PAM has a per-iteration complexity of $O(k(n-k)^2)$ (Ng and Han, 1994), making it unsuitable for large datasets.

In this work, patterns have been considered as d-dimensional real-valued vectors and, moreover, such input space has been equipped with the Euclidean distance as (dis)similarity measure, whose complexity is $O(d)$. Therefore, the overall per-iteration complexity can as well be defined as $O(nkd)$. By considering p computational units, the per-iteration complexity drops to $O(nk/p)$ (or $O(nkd/p)$ by explicitly taking into account the Euclidean distance) since the point-to-medoids distance will be computed in parallel.

As far as the medoid update is concerned, the Exact Medoid Update procedure has complexity $O(c^2/p)$ where c is the cluster size, whereas the Approximate Medoid Tracking has complexity $O(c-P) \simeq O(c)$ since usually $c \gg P$.

5 CONCLUSIONS

In this paper, we proposed a novel implementation for the large-scale k-medoids algorithm using the Apache Spark framework as parallel and distributed computing environment. Specifically, we proposed two procedures for the medoid evaluation: an exact (but computationally intensive) procedure and an approximate (but computationally lighter) procedure which have been proved to be very effective. As far as efficiency is concerned, we tested our algorithm in a single-machine multi-threaded environment with satisfactory results.

Starting from this implementation many further steps can be done. First, this implementation has been deliberately proposed to be as general as possible: indeed, by looking at the algorithm description in §3.2, 3.3 and 3.4, the end-user must just change the (dis)similarity measure $d(\cdot,\cdot)$ according to the nature of the input space at hand in order for the whole implementation to work. The same reasoning holds for k-means++: albeit it is true that there are efficient parallel and distributed implementations (e.g. k-means$||$, (Bahmani et al., 2012)), they namely work on the Euclidean space only. Moreover, as concerns initialisation techniques, a future work might focus on developing a DBCRIMES parallel and distributed implementation, since it is also suitable for dealing with non-metric spaces.

Further, we proposed two medoid evaluation procedures which are ideally robust to any cluster's size. This can be seen in Algorithm 1, where a for-loop scans one cluster at the time. This is robust because the entire processing power is dedicated to a single cluster which can therefore be massive as well, but if one is confident that clusters cannot be arbitrarily big it is possible to group the available workers[8] in such way that each group processes a given cluster. In this manner the aforementioned for-loop can be done in parallel as well.

Albeit the two medoids' update routines have been presented separately in order to discuss their respective strengths and weaknesses, they can be easily used together by adopting a double-threshold approach. Specifically, let T_1 and T_2 be two thresholds with $T_1 < T_2$: every cluster whose size is less than T_1 can be considered as "small" and processed fully in-memory; every cluster whose size is in range $[T_1, T_2]$ can be considered as "medium" and processed using the Exact Medoid Update and, finally, clusters greater than T_2, to be considered as "big", can be processed using the Approximate Medoid Tracking.

One of the most intriguing works is a direct extension of the former observation, as discussed in §2: ideally the k-medoids can work with any (dis)similarity measure and it does not need to define some operations for evaluating the new cluster representative (e.g. mean or median): whether it is possible to define a (dis)similarity measure between two objects, the k-medoids can be used. This opens a new horizon on the nature of the input space(s) which can

[8] As we did in order to simulate 4-core PCs.

be clustered, as the so-called *non-metric spaces* such as graphs or sequences, for which defining a mean or median is nonsensical.

REFERENCES

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

Ankerst, M., Breunig, M. M., Kriegel, H.-P., and Sander, J. (1999). Optics: Ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 49–60, New York, NY, USA. ACM.

Arbelaez, A. and Quesada, L. (2013). Parallelising the k-medoids clustering problem using space-partitioning. In *Sixth Annual Symposium on Combinatorial Search*.

Arthur, D. and Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics.

Bahmani, B., Moseley, B., Vattani, A., Kumar, R., and Vassilvitskii, S. (2012). Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633.

Bianchi, F. M., Livi, L., and Rizzi, A. (2016). Two density-based k-means initialization algorithms for non-metric data clustering. *Pattern Analysis and Applications*, 3(19):745–763.

Bradley, P. S., Mangasarian, O. L., and Street, W. N. (1996). Clustering via concave minimization. In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, NIPS'96, pages 368–374, Cambridge, MA, USA. MIT Press.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Del Vescovo, G., Livi, L., Frattale Mascioli, F. M., and Rizzi, A. (2014). On the problem of modeling structured data with the minsod representative. *International Journal of Computer Theory and Engineering*, 6(1):9.

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, volume 96, pages 226–231.

Guha, S., Rastogi, R., and Shim, K. (1998). Cure: An efficient clustering algorithm for large databases. *SIGMOD Rec.*, 27(2):73–84.

Jiang, Y. and Zhang, J. (2014). Parallel k-medoids clustering algorithm based on hadoop. In *Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on*, pages 649–652. IEEE.

Kaufman, L. and Rousseeuw, P. J. (1987). Clustering by means of medoids. *Statistical Data Analysis Based on the L1-Norm and Related Methods*, pages North–Holland.

Kaufman, L. and Rousseeuw, P. J. (2009). *Finding groups in data: an introduction to cluster analysis*, volume 344. John Wiley & Sons.

Lichman, M. (2013). UCI machine learning repository.

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137.

MacQueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In Cam, L. M. L. and Neyman, J., editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press.

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7.

Ng, R. T. and Han, J. (1994). Efficient and effective clustering methods for spatial data mining. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 144–155, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Park, H.-S. and Jun, C.-H. (2009). A simple and fast algorithm for k-medoids clustering. *Expert systems with applications*, 36(2):3336–3341.

van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30.

Xu, X., Jäger, J., and Kriegel, H.-P. (1999). A fast parallel clustering algorithm for large spatial databases. *Data Mining and Knowledge Discovery*, 3(3):263–290.

Yue, X., Man, W., Yue, J., and Liu, G. (2016). Parallel k-medoids++ spatial clustering algorithm based on mapreduce. *arXiv preprint arXiv:1608.06861*.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.

Zhang, T., Ramakrishnan, R., and Livny, M. (1996). Birch: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 103–114, New York, NY, USA. ACM.

Zhao, W., Ma, H., and He, Q. (2009). Parallel k-means clustering based on mapreduce. In *IEEE International Conference on Cloud Computing*, pages 674–679. Springer.