Parallelizing Distance Computations in Unsupervised Clustering Techniques

Antonio Javier Samaniego Jurado
samaniegojurado.samaniegojurado@student.uva.nl
University of Amsterdam

ABSTRACT

The increasing amount (terabytes) of data generated in modern applications has resulted in a need for faster, more efficient ways to process information. Traditional clustering algorithms have also been challenged in classification tasks when dealing with huge repositories of data, facing computational efficiency challenges. K-means clustering is among the most popular methods used for this kind of unsupervised problems in all kinds of fields. In this project, a feasible, reliable MapReduce-based approach to parallelize distance computations in this technique is presented, showing the computational cost of scaling to high amounts of data and that such cost can be exponentially decreased with a linear increase in the number of parallel nodes.

1 INTRODUCTION

Due to the growing amount of data generated in modern applications (cloud, web services, mobile apps, etc), tasks such as classification or anomaly detection have become too computationally expensive or unfeasible. In particular, unsupervised clustering approaches that have proven their potential when tackling those problems with unlabeled data start facing performance challenges as the number of data points gets large. For instance, scopes such as fraud detection in finance, where millions of transactions are made every day and are stored at a faster rate than they can be labeled as fraud or not fraud are specially benefited by the power of clustering methods. The MapReduce paradigm [1] represents a powerful, widely versatile way to distribute independent computations that can easily be combined together once finished, significantly improving performance.

The challenge of scaling computations to large amounts of data is common across a variety of data-processing algorithms. In particular, when it comes to clustering techniques the main problem resides in the need of multiple data epochs (i.e. loading the entire dataset several times) to ensure a significant level of convergence. This is a big constraint when 1) the amount of memory available is limited and 2) time complexity is high, both having an impact on performance. Both problems get increasingly bigger as the dataset becomes larger, which motivates the optimization of underlying operations. In this project, a MapReduce-based approach of parallelizing k-means is presented, allowing practitioners to scale the effectiveness and simplicity of this algorithm up to vast amounts of data, where standard, serialized runs struggle to maintain a high performance. The following two datasets have been used for this purpose:

- *Artificially generated data*: Generated through a random normal distribution with pre-defined upper/lower that strategically form 3 clusters out of N 2-dimensional points, leaving N as an input parameter.
- *ULB Machine Learning Group credit-card transaction dataset* [7]: Contains 285k credit card transactions made by anonymous european cardholders in September 2013 over the course of 2 days. Includes 30 numerical features plus a fraud/not-fraud label.

Related Work

While previous solutions have focused on the limited memory issue (more specifically on compressible, in-memory, and discardable data sub-sets) limiting the number of data scans[5], as well as on time complexity, by aggregating local results to obtain global models while still keeping a sampling-based approach[4], this project focuses on making the underlying operations more efficient. In particular, optimizing distance computations between each data point and corresponding cluster center, based on a MapReduce approach. In particular, a comparative study presented by Dweepna et al.[6], where premises on the theory behind using MapReduce in Hadoop for distributed computing are suggested, lacking experimental results. In the case of k-means, novel approaches have aimed to make changes to the base algorithm in order to better make the initial guess of centroids, achieving faster convergence before parallelization (*k-means++*); and suggested variations in parallelization to improve performance by better distributing chunks of data across nodes [9]. However, no experiments with different data sizes and number of distributed nodes were performed. This project presents a quantitative approach to study the feasibility, reliability, and computational performance impact of parallelizing k-means clustering through a comprehensive set of quantitative experiments.

Research Questions

This project aims to provide practitioners with a way to scale the effectiveness and simplicity of k-means to large amounts of data by parallelizing the underlying distance computations using MapReduce. In particular, it aims to address the following research questions:

(1) How can MapReduce be applied to unsupervised clustering techniques to parallelize distance computations?
(2) What impact does the parallelization have on performance against a serialized approach?

This study serves as a tangible benchmark against serialized results from the base algorithm. One limitation is the lack of access to a public dataset of up to terabytes in size, as well as the hypothetical, corresponding computing resources (e.g. multiple computing clusters to handle more tests). However, a reliable, single-machine implementation with the same level of evaluation rigor and accuracy of a higher-scale distributed cluster has been developed, which allows to address larger-scale behaviour expectations.

2 IMPLEMENTATION

As previously described, the implementation and performance evaluation benchmark will be performed on k-means, serving as the baseline unsupervised clustering algorithm. Such baseline, serialized version of k-means is defined in Algorithm 1.

Algorithm 1 K-means baseline algorithm

```
Input (1) Set of data points $X = \{x_1, x_2, ..., x_n\}$,
      (2) Number of clusters k
Output $M = \{m_{c_1}, m_{c_2}, ..., m_{c_k}\}$ cluster means/centroids
1: $M \longleftarrow$ Initialize random centroids
2: while not converged do
3:     for $x \subset X$ do
4:         Assign $x$ to cluster $c_x$ with closest mean $m_{c_x}$
5:     end for
6:     for $c \subset C$ do
7:         $M \longleftarrow$ Update mean $m_c$
8:     end for
```

Taking a set of data points X and a number of clusters k as input, the baseline algorithm starts by randomly choosing k points within X space as intial means/centroids $M = \{m_{c_1}, m_{c_2}, ..., m_{c_k}\}$, defining the initial clusters $\{c_1, c_2, ..., c_k\}$. Once initialized, the distance between every data point $x \subset X$ and mean $m \subset M$ is computed, assigning each x to the cluster c_x with closest mean m_{c_x}. This updates every cluster $c \subset C$. Euclidean distance will be applied, defined as:

$$d(x, m) = \sqrt{(x - m)^2} \tag{1}$$

with W-dimensional x and m ($W \subset \mathbb{Z}$). Finally, the mean m_i of the new cluster c_i is updated, which yields a new centroid. Both the cluster assignment and mean update steps are repeated until convergence is achieved based on a certain criteria. In this case, the convergence criteria will be satisfying any of the following conditions:

- Means $m \subset M$ of updated clusters $c \subset C$ remain unchanged. Taking a minimum distance threshold, the current means from iteration i, and new/updated means from iteration $i+1$ as D_{min}, M_i and M_{i+1} respectively, convergence is met if:
$$D_{min} = \alpha d(M_1, M_2) \tag{2}$$
$$d(M_i, M_{i+1}) < D_{min} \tag{3}$$
Where $d(M_1, M_2)$ is the means change in distance from iteration 1 to 2, assumed to be the maximum change of all iterations; and α denotes the change coefficient, which determines what proportion of $d(M_1, M_2)$ is taken as the minimum threshold D_{min} ($0 < \alpha < 1$ and default $\alpha = 0.05$).
- Maximum number of iterations I is reached (default $I = 10$ was chosen).

Note that the better the intial centroid random choice the faster the convergence. Since one of the limitations of k-means (among other clustering algorithms) is data dependency, that is, the need of previous data to compute current, which does not allow parallelization to simultaneously perform operations in one single run, in the proposed methods the following assumption is made: distance computations between a set of data points X and a cluster mean m_1 are independent from the same computations between X and another cluster mean m_2.

Algorithm 2 MapReduce K-means algorithm

```
Input (1) Set of data points $X = \{x_1, x_2, ..., x_n\}$,
      (2) Number of clusters k
      (3) Number of parallel nodes D
Output $M = \{m_{c_1}, m_{c_2}, ..., m_{c_k}\}$ cluster means/centroids
1: $M \longleftarrow$ Initialize random centroids
2: Broadcast $M$ to every node $d \subset D$
3: while not converged do
4:     $[(c_{x_1}, x_1), ..., (c_{x_N}, x_N)] \longleftarrow$ Map x $\subset X$ and emit <$c_x, x$> with $c_x$ the cluster assigned to $x$ with closest mean $m_{c_x}$
5:     $M \longleftarrow$ Group mapped data by cluster, and for each cluster recompute centroid $m_{c_{x_1,x_2,...,x_T}} = AVG(\{x_1, x_2, ..., x_T\})$, with $\{x_1, x_2, ..., x_T\}$ the subset of $X$ assigned to $c_{x_1,x_2,...,x_T}$
6:     Broadcast new $M$ to all nodes
```

Based on such assumption, the parallel approach consists in mapping the input X to D nodes, D points at a time (i.e. process one point $x \subset X$ per parallel node $d \subset D$). A node d

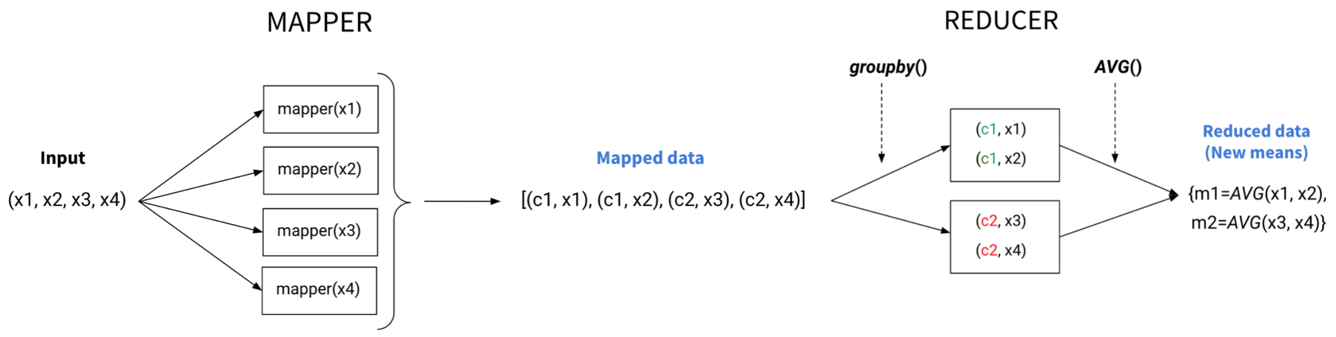


Figure 1: Proposed MapReduce approach - Flow from input to mapped and reduced data.

assigns x to the cluster c_x with closest mean m_{c_x}, emitting a key-value tuple in the form of (c_i, x). Once all N input points are mapped, this step (mapper) yields the mapped data, a list of tuples $[(c_{x_1}, x_1), ..., (c_{x_N}, x_N)]$ of cluster-point pairs. The tuples are then grouped by cluster, and for each cluster the centroid/mean is recomputed. That is, $m_{c_{x_1,x_2,...,x_T}} = AVG(\{x_1, x_2, ..., x_T\})$ with $\{x_1, x_2, ..., x_T\}$ the subset of X assigned to $c_{x_1,x_2,...,x_T}$. Note that this final step (reducer) performs both the sometimes referred to in MapReduce as *shuffling* (grouping by key, i.e. cluster in this case) and pure *reducing* (updating centroids by recomputing means M). In this scenario, each node has to compute the euclidean distance $d(x, m)$ between a data point x and every cluster mean $m \subset M$. Therefore, each mean m_i must be present in all D nodes. Both map and reduce phases are graphically shown in Fig. 1.

The presented methods require a system of D processors working in parallel. Given the lack of direct access to a cluster of distributed computers and for the purpose of this project, a choice for Python's multiprocessing API [2] has been made. As a process-based "threading" interface, it allows to execute multiple processes (instead of threads) on a single computer, leveraging D processors on a First In, First Out (FIFO) basis. In particular, a *Pool* object has been used, which starts an independent process for every parallel job and speeds up computations by taking advantage of the fast data input of the proposed approach (one point mapped per parallel process d).

3 EVALUATION

Two main advantages of the presented implementation makes it a good fit for the proposed parallel k-means approach:

- The pool object distributes the distance computation task to the worker processes, having the execution wait until all processes are finished before collecting/returning values in the form of list to the parent process. This allows to gather the emitted tuples (list of tuples/mapped data) in the map phase.
- On the other hand, since all processes are able to access the same (common) variables stored in memory, the step of *broadcasting* means to all nodes after every iteration can be achieved by creating a variable that stores the current means and is updated when an iteration finishes.

The use of this tool ensured a reliable, single-machine implementation with the same level of evaluation rigor and accuracy of a higher-scale distributed cluster. In order to evaluate the methods and develop a solid answer to the research questions, a set of experiments has been run, testing the algorithm's computational performance. This has allowed to measure the impact of parallelization and study its variability as parameters are tuned. Specifically, **total and iteration elapsed times** have been measured under the following 2 scenarios:

1. Elapsed Times on Variable Input Data Size and Number of Clusters

In this first experiment, the number of parallel working nodes was fixed to $D = 10$. The choice of 10 is mostly arbitrary, although it was tested to ensure it represents a safe minimum threshold that adequately sets a baseline where the effects of parallelization are significant.

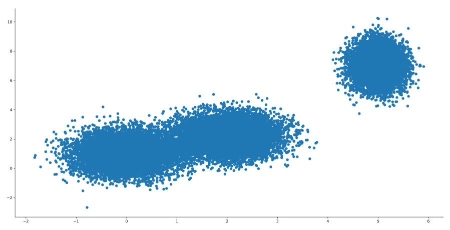
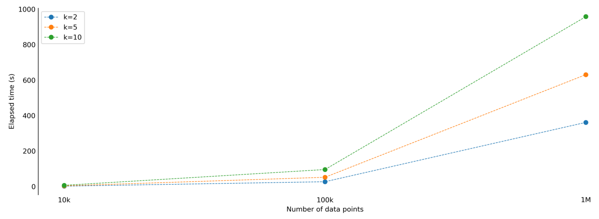
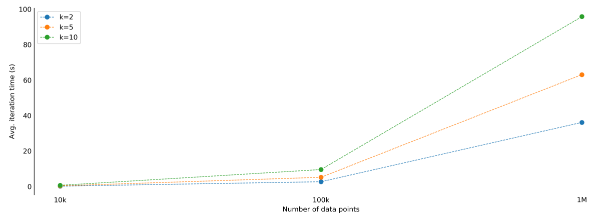


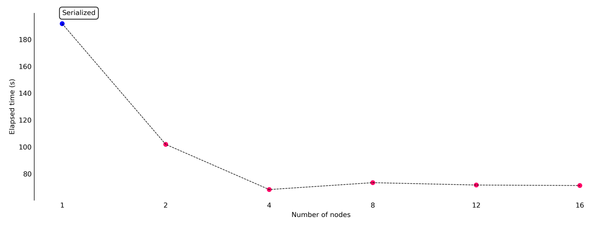
Figure 3: Generated dataset for $N = 10k$ points.



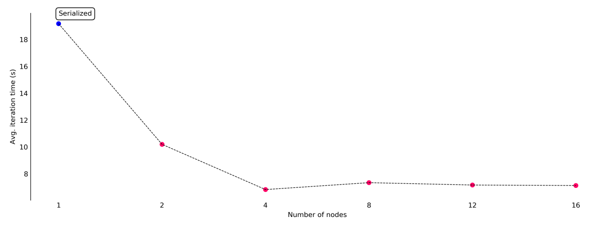
(a) Experiment 1: Total elapsed time.



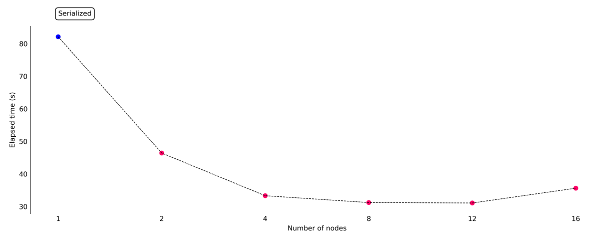
(b) Experiment 1: Average iteration time.



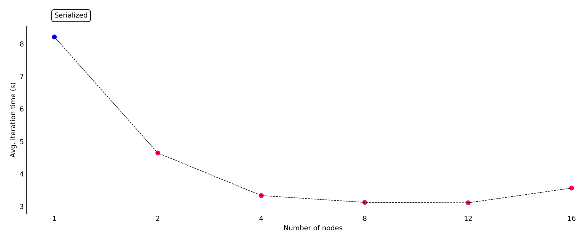
(c) Experiment 2: Total elapsed time (generated dataset).



(d) Experiment 2: Average iteration time (generated dataset).



(e) Experiment 2: Total elapsed time (fraud dataset)



(f) Experiment 2: Average iteration time (fraud dataset).

Figure 2: Experimental results

Based on that, the aim of this experiment is to study the impact of different input data sizes N and number of clusters k on the distributed k-means algorithm, which also tests its reliability on changing conditions. Since the data size N is a variable, this was only run on the randomly generated dataset. A choice for a random normal distribution with pre-defined upper/lower limits was made to generate the data and strategically form 3 clusters out of N 2-dimensional points, leaving N as a parameter. Fig. 3 shows an example of the resulting generated dataset for $N = 10k$ points.

2. Elapsed Times on Variable Number of Parallel Nodes

In this second experiment, the input data size was set to $N = 100k$ points and number of clusters to $k = 8$. While the choice of N was tested as a result of experiment 1, choosing 8 clusters is a replication of the default value of k in the k-means method of scikit-learn [3], a widely used machine learning library. This experiment aims to purely test the computational performance impact of the distributed algorithm by measuring total/iteration elapsed time for different number of parallel nodes D. Since D is independent from the input data, this was run on both the generated and fraud dataset. As previously mentioned, the fraud dataset contains 285k credit card transactions made by anonymous european cardholders in September 2013 over the course of 2 days. While this size can also be achieved on the generated dataset, the main quantitative difference between the two is that each transaction in the fraud dataset has 30 PCA-transformed numerical features (encoded for confidentiality purposes) plus the fraud/not fraud label. This not only tests the parallel algorithm convergence on the number of input points but also their dimension (30 vs. 2 in the generated data). Moreover, as it is not artificially generated, it allows the algorithm to show its performance on real data under an actual industry problem scenario. Note that for the case of the the fraud dataset, the number of clusters was set to $k = 2$ (fraud/not-fraud).

A shared assumption in both experiments is the maximum number of iterations for a sigle run of the algorihtm was set to $I = 10$. Fig. 2 shows the results of both experiments, run on a 2.2 GHz Quad-Core Intel Core i7 processor and 16 GB 1600 MHz DDR3 memory (MacBook Pro Retina, 15-inch, Mid 2014). The resulting source code is available at GitHub [8].

4 DISCUSSION

Following the presented evaluation strategy, both experiments were successfully run on the two datasets. The following insights were subsequently taken from the results.

Feasibility, Reliability, and Usability of the Distributed Algorithm

The first thing that was observed from running the experiments was its feasibility and reliability, both from a technical implementation and expected behaviour perspective. Firstly, the use of the optimized python's multiprocessing API allowed it to scale from 10k to 1M input data points and 1 to 16 parallel nodes (processes) without memory bottleneck issues. Secondly, it is shown that setting a baseline scenario where D nodes work in parallel to study other changing factors (data size and number of clusters in this case) was possible without interfering or creating experimental bias. Lastly, developing an API-oriented, reusable, and extensible implementation allowed for effortless parameter tuning and overall experimental process, which can easily be replicated by future practitioners.

Impact of Parallelization on Computational Performance

In terms of the impact of parallelization on computational performance, the following observations were made:

(1) Total elapsed time grows exponentially with an exponential increase of the input data size and number of clusters.
(2) Total elapsed time is exponentially reduced with a linear increase of number of parallel nodes.
(3) Average iteration times behavior stays consistent with total elapsed times across both experiments.
(4) The above behaviour remains consistent when the dimension of the input data points grows.

The first observation shows that there is a direct mapping between the growth rate of total elapsed times and input data sizes N and number of clusters k. That is, one grows with the other at the same rate. As shown in Figs. 2a and 2b, total elapsed time grows exponentially as N and k grow also exponentially ($N = [10k, 100k, 1M]$ points and $k = [2, 5, 10]$ clusters). Recalling that the number of nodes for this experiment was fixed to $D = 10$, this suggests that scaling the clustering algorithm to higher amounts of data (N) or increasing the number of clusters to separate the data on (k) is still an increasingly (exponential) expensive task in terms of performance, despite having a number of workers processing distance computations in parallel. An additional factor to take into account is that this experiment was run on generated data designed to form 3 clusters. As N increases, the clusters start to progressively overlap, resulting in a higher amount of less separable data. This makes it harder for the algorithm to converge (i.e. higher total elapsed time) on bigger data sizes and number of clusters to separate the data on.

The second observation shows that even though the computational cost of processing N points with k clusters grows exponentially with N and k, for a given N and k this cost is reduced also exponentially with a linear increase of parallel working nodes ($D = [1, 2, 4, 8, 12, 16]$), as shown in Figs. 2c, 2d, 2e and 2f. This represents a powerful performance improvement due to parallelization, since a linear increase in the number of nodes is a feasible and relatively cheap task (e.g. upgrading a computer cluster from 2 to 4 nodes), and is still able to decrease the elapsed time of a very computationally expensive problem, exponentially. While this experiment was run on $N = 100k$ points and $k = [8, 2]$ clusters(8 and 2 for the generated and fraud data, respectively), this behavior applies to other fixed values of N and k.

The third and fourth observations show that the described behaviors remain consistent on an iteration level (i.e. average iteration times behave the same way as total elapsed times) and when the dimension of the input data points gets bigger, as shown in the fraud dataset results (Figs. 2e and 2f), where performance behaves the same way for 30-dimensional points (PCA-transformed numerical features) vs. 2-dimensional in the generated dataset.

Classification Accuracy of the Distributed Algorithm

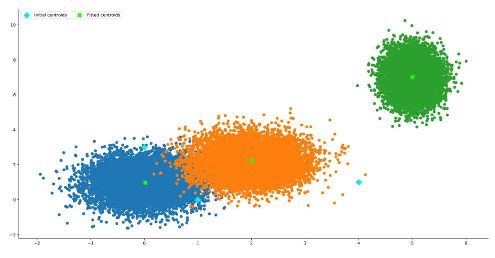


Figure 4: Fitted data for $N = 10k$ points, $k = 3$ clusters and $D = 10$ nodes

Additionally, even though the purpose of this project was not to achieve a certain level of accuracy when classifying/separating points into the different clusters, Fig. 4 shows that the distributed algorithm is able to fit the data shown in Fig. 3 with a significant level of accuracy (run for $N = 10k$ points, $k = 3$ clusters and $D = 10$ nodes).

Future Work

The following suggestions could contribute positively to this project as future work.

- Optimize the way input data points are mapped to D nodes at a time. For instance, map t points at a time per node instead of only one.
- Implement the proposed algorithm in a production, environment to show it can scale to cloud-level applications.
- If data access allows it, replicate the proposed evaluation strategy and associated experiments on a bigger dataset (TB of data).

REFERENCES

[1] [n.d.]. *MapReduce.* Retrieved February 12, 2020 from https://en.wikipedia.org/wiki/MapReduce

[2] [n.d.]. *Multiprocessing: Process-based threading interface.* Retrieved April 8, 2020 from https://docs.python.org/2/library/multiprocessing.html

[3] [n.d.]. *Sckit-learn, machine learning in python.* Retrieved April 8, 2020 from https://scikit-learn.org/stable/index.html

[4] Malika Bendechache. [n.d.]. *Efficient Large Scale Clustering based on Data Partitioning.* Retrieved February 22, 2020 from https://arxiv.org/pdf/1704.03421.pdf

[5] P.S. Bradley. [n.d.]. *Scaling Clustering Algorithms to Large Databases.* Retrieved February 22, 2020 from https://aaai.org/Papers/KDD/1998/KDD98-002.pdf

[6] B.B.Panchal Dweepna Garg, Khushboo Trivedi. [n.d.]. *A Comparative study of Clustering Algorithms using MapReduce in Hadoop.* Retrieved February 12, 2020 from https://ijert.org/research/a-comparative-study-of-clustering-algorithms-using-mapreduce-in-hadoop-IJERTV2IS101148.pdf

[7] ULB ML Group. [n.d.]. *Credit Card Fraud Detection Dataset.* Retrieved February 12, 2020 from https://kaggle.com/mlg-ulb/creditcardfraud

[8] Antonio Javier Samaniego Jurado. [n.d.]. *MapReduce K-means.* Retrieved April 8, 2020 from https://github.com/samaxtech/mapreduce-k-means

[9] Stanford Max Bodoia. [n.d.]. *MapReduce Algorithms for k-means Clustering.* Retrieved February 12, 2020 from https://stanford.edu/~rezab/classes/cme323/S16/projects_reports/bodoia.pdf