Determining the k in k-means with MapReduce

Thibault Debatty, Pietro Michiardi, Wim Mees, Olivier Thonnard

Determining the k in k-means with MapReduce

Thibault Debatty
Royal Military Academy, Brussels, Belgium
thibault.debatty@rma.ac.be

Pietro Michiardi
EURECOM, Campus SophiaTech, France
pietro.michiardi@eurecom.fr

Wim Mees
Royal Military Academy, Brussels, Belgium
wim.mees@rma.ac.be

Olivier Thonnard
Symantec Research Labs, Sophia Antipolis, France
olivier_thonnard@symantec.com

Abstract

In this paper we propose a MapReduce implementation of G-means, a variant of k-means that is able to automatically determine k, the number of clusters. We show that our implementation scales to very large datasets and very large values of k, as the computation cost is proportional to nk. Other techniques that run a clustering algorithm with different values of k and choose the value of k that provides the "best" results have a computation cost that is proportional to nk^2.

We run experiments that confirm that the processing time is proportional to k. These experiments also show that, because G-means adds new centers progressively, if and where they are needed, it reduces the probability to fall into a local minimum, and finally finds better centers than classical k-means processing.

1. INTRODUCTION

Discovering groups of similar objects in a dataset, also known as clustering, is one of the most fundamental techniques of data analysis [12]. Clustering algorithms are used in many fields including machine learning, pattern recognition, image analysis, information retrieval, market segmentation and bioinformatics.

A lot of different algorithms exist, mainly depending on their definition of a cluster. Density based algorithms, like DBSCAN [8] and OPTICS [2] for example, define a cluster as a high density region in the feature space. Other algorithms assume that the data is generated from a mixture of statistical distributions. Finally, centroid models, like k-means, represent each cluster by a single center point. This algorithm thus implicitly assumes that the points in each cluster are spherically distributed around the center [9].

The most known algorithm for computing k-means clustering is Lloyd's algorithm [13], also known as "the k-means algorithm". Although, it was published more than 30 year ago, it is still widely used today as it is at the same time simple and effective [12]. However, it also has a number of drawbacks:

1. It may converge to a local minimum, producing counterintuitive or even inconsistent results.
2. It is not really efficient, and may converge very slowly.
3. It prefers clusters of approximately similar size, as it always assigns an object to the nearest center. This often leads to incorrect borders between clusters.
4. Finally, like a lot of other clustering algorithms, it requires the number of clusters – k – to be specified in advance, which is considered as one of the most difficult problems to solve in data clustering [12].

In this paper we tackle this last drawback. We present and analyze the performance of a MapReduce implementation of G-means[9], an efficient algorithm to determine k. We also compare our algorithm to a common MapReduce implementation of k-means.

More specifically, we first show that a MapReduce implementation of G-means requires some modifications of the original algorithm to reduce I/O operations, as these are very costly in MapReduce, and to reduce the number of chained MR jobs. We also show that an efficient implementation that maximizes processing parallelism requires a hybrid design that takes into account the number of nodes running the algorithm and the quantity of heap memory available.

We then study the performance aspects of the proposed algorithm implementation by modeling the communication and computational cost. We show that our algorithm is able determine k and find clusters with a computation cost proportional to nk. Other techniques that run a clustering algorithm with different values of k and choose the value of k that provides the "best" results have a computation cost that is proportional to nk^2.

Finally, we evaluate both solutions experimentally. Our results confirm that the proposed MR implementation of G-means has linear complexity with respect to k. The algorithm also takes full advantage of additional computing nodes, which makes it scalable to very large

datasets. Moreover, our experiments show that our implementation clearly outperforms the classical iterative k-means solution as it reduces the probability to fall into a local minimum and provides better clustering results.

The rest of the paper is organized as follows : In section 2 we present G-means[9] and other existing methods to determine k, as well as other optimizations of k-means. In section 3 we present and justify our MapReduce implementation of G-means. In section 4 we estimate and compare the computation and communication costs of the MapReduce implementations of G-means and k-means. In section 5 we present our experimental results, and finally we present our conclusions.

2. RELATED WORK

When clustering a dataset, the right number of clusters to use – k – is often a parameter of the algorithm.

Even when analyzing data visually, the correct choice of k is often ambiguous. It largely depends on the shape and scale of the distribution of points in the data set and on the desired clustering resolution of the user.

In addition, arbitrarily increasing k will always reduce the amount of error in the resulting clustering, to the extreme case of zero error if each data point is considered its own cluster.

If an appropriate value of k is not apparent from prior knowledge of the properties of the data set, it must be chosen somehow. There are several methods for making this decision. Lots of them rely on cluster evaluation metrics. They run a clustering algorithm with different values of k, and choose the value of k that provides the "best" results according to some criterion.

For example, Dunn's index (DI) [7] can be used to determine the number of clusters. The k for which the DI is the highest can be chosen as the number of clusters.

The elbow method [20] is another possible criterion. It chooses a number of clusters so that adding another cluster doesn't give much better modeling of the data. Therefore, it computes the percentage of variance explained (the ratio of the between-group variance to the total variance, also known as an F-test) for different values of k. In the graph of the percentage of variance explained by the clusters against the number of clusters, the first clusters will add much information (explain a lot of variance), but at some point the marginal gain will drop, giving an angle in the graph. The number of clusters is chosen at this point, hence the "elbow criterion". As it is a visual method, this "elbow" cannot always be unambiguously identified.

The average silhouette of the data [18] is another useful criterion for assessing the natural number of clusters. The silhouette of a point is a measure of how close it is to other points within its cluster and how loosely it is matched to points of the neighboring cluster, i.e. the cluster whose average distance from the point is lowest. A silhouette close to 1 implies the point is in an appropriate cluster, while a silhouette close to -1 implies the point is in the wrong cluster. If there are too many or too few clusters, as it may occur when a wrong value of k is used with k-means algorithm, some of the clusters will typically display much narrower silhouettes than the rest. Thus silhouette plots and averages may also be used to determine the natural number of clusters within a dataset.

Sugar and James [19] used information theory to propose a new index of cluster quality, called the "Jump method". The method is based on the notion of "distortion", which is a measure of within-cluster dispersion. For each possible value of k, the method calculates the "jump" of distortion compared with previous value of k. The Estimated number of clusters is the value of k with the largest jump.

Tibshirani and al. [21] proposed anoter method based on dispersion, called the "Gap statistic" for estimating the number of clusters in a data set. The idea is to compare the change in within-cluster dispersion to that expected under an appropriate null distribution as reference. The number of clusters is then the value for which the observed dispersion falls the farthest below the expected dispersion obtained under a null distribution.

Finally, two other studies presented iterative techniques to determine the number of clusters when performing k-means clustering, which do not require to run k-means for every possible value of k: X-means [17] and G-means [9].

X-means iteratively uses k-means to optimize the position of centers and increases the number of clusters if needed to optimize the Bayesian Information Criterion (BIC) or the Akaike Information Criterion (AIC) measure. The main advantage of the algorithm is the efficiency of the test used to select the most promising centers for refinement. This leads to a fast algorithm that outputs both the number of clusters and their position. Experiments showed this technique revealed the true number of clusters in the underlying distribution, and that it was much faster than repeatedly using k-means for different values of k.

G-means is also an iterative algorithm but it uses Anderson-Darling test to verify whether a subset of data follows a Gaussian distribution. G-means runs k-means with increasing values of k in a hierarchical fashion until the test accepts the hypothesis that the points assigned to each center follow a Gaussian distribution. Experimental results showed that the algorithm seems to outperform X-means.

The G-means algorithm starts with a small number of clusters, and increases the number of centers. At each iteration, the algorithm runs k-means to refine the current centers. The clusters whose data appears not

to come from a Gaussian distribution are then split.

For each cluster X (being a subset of data) of center c, the algorithm works as follows:

1. Find two new centers c_1 and c_2.
2. Run k-means to refine c_1 and c_2.
3. Let $v = c_1 - c_2$ be the vector that connects the two centers. This is the direction that k-means believes is important for clustering.
4. Let X' be the projection of X on v. X' is a one-dimensional representation of the data projected on v.
5. Normalize X' so that it has zero mean and variance equal to 1.
6. Use Anderson-Darling to test X':
   - If X' follows a normal distribution, keep the original center, and discard c_1 and c_2.
   - Otherwise, split the cluster in two, use c_1 and c_2 as new centers and run the algorithm on each sub-cluster.

The main advantage of this algorithm is that it simplifies the test for Gaussian fit by projecting the data to one dimension where the test is simple to apply. Moreover it only creates new centers where needed, improving clustering quality.

In this paper, we present a MapReduce implementation of the G-means algorithm. Some key challenges to be addressed are the various design choices for parallelizing the algorithm, as these may have a significant impact on final results quality, but also on communication and computational cost.

While the choice of k is a critical question, many other optimizations have been proposed in the literature to improve or speed up k-means processing.

A first optimization consists in selecting better initial centers, which allows the algorithm to converge quicker, reduces the probability to fall into a local minimum and reduces the number trials needed. In k-means++ [3], the starting centers are chosen randomly, but with a probability proportional the distance to the nearest already chosen center. Bahmani [4] also proposed a MapReduce version of k-means++ initialization algorithm. Another common possibility is to use canopy clustering [15] to compute the initial centers. Algorithms also exist to avoid local minimums, for example by swapping points between clusters [11].

Other optimizations deal with nearest neighbor (NN) search. In k-means, a NN search is required to decide to which cluster a point belongs. It is thus one of the basic operations of k-means processing, but also of a lot of other clustering algorithms. One type of efficient NN search algorithm uses tree-based structures, like the mrkd-tree algorithm proposed by Pelleg et al. [16]. The algorithm uses a multi-resolution k-d tree to represent groups of points and efficiently identify the nearest cluster centers for those points. Vrahatis et al. [22] proposed a version that uses a windowing technique based on range trees. A range tree on a set of points in d-dimensions is a recursively defined multi-level binary search tree. Each level of the range tree is a binary search tree on one of the d-dimensions, which allows fast range searches. Another category of algorithms uses random projection, like Locality Sensitive Hash used by Buhler [5].

Other algorithms improve the clustering efficiency by first summarizing a large data set, and then applying the clustering algorithm. Different approaches exist:

- Replace a small tight group of objects (but not the whole cluster) by a single object [6] or by a coreset [10];
- Pre-process data to reduce dimensionality, dropping unnecessary features (dimensions) [1];
- Partition data into overlapping subsets [15] for high dimensional data.

While all these different optimizations of k-means are definitively valuable, it is outside the scope of this paper to implement and evaluate all of them. However, some of these optimizations could be easily integrated in the MapReduce implementation proposed in this paper, and we are considering them as part of our future work.

3. MAPREDUCE IMPLEMENTATION OF G-MEANS

Our implementation of G-means for MapReduce is presented in Algorithm 1.

Algorithm 1 MapReduce G-means pseudo-code

```
PickInitialCenters
while Not ClusteringCompleted do
    KMeans
    KMeansAndFindNewCenters
    TestClusters
end while
```

The first step, `PickInitialCenters`, is a classical step of any k-means algorithm. The main difference with respect to classical k-means implementations is that it picks pairs of centers (c_1 and c_2). We use a serial implementation, that picks initial centers at random, but other distributed or more efficient algorithms can be found in the literature and can perfectly be used instead.

The algorithm then enters a `while` loop that will continue as long as there are clusters that must be split.

The first operation of the loop is a classical MapReduce implementation of k-means with combiners[1], to refine to position of current centers.

The last iteration of k-means is implemented in a separate MapReduce job called `KMeansAndFindNewCenters` in Algorithm 2. It will also, for each cluster, pick the two new centers (c_1 and c_2) that will be used at next iteration. This job is specific to our implementation and is further explained below.

Finally, the clusters are tested using the MapReduce job referred to as `TestClusters` in Algorithm 1. For each point, the job searches the cluster it belongs to (using the centers from previous iteration), then projects it on the vector formed by the two corresponding centers (of current iteration). Finally, for each cluster it tests if the projections form a normal distribution. This job, also specific to the proposed implementation, is explained in more details here below.

As can be noticed, our MapReduce implementation of G-means differs from the sequential version in three main aspects.

First, the original G-means algorithm works locally, by analyzing each cluster separately. It thus requires that each point is "linked" in some way to the cluster it belongs to at each iteration of the algorithm. Implementing this in MapReduce would require a write operation at each iteration, to save this information in the distributed file system.

This membership information can of course be used to reduce computations at some steps of the algorithm:

- When running k-means, for each point, the algorithm does not have to compute the distance to each center, but only to c_1 and c_2, the 2 children centers of the cluster the point currently belongs to;
- When testing the clusters, the cluster to which a point belongs is directly identified, and the algorithm does not have to compute the distance from this point to each cluster.

However, binding the points to their cluster would require a write operation at each iteration, and could at best spare $O(2nk)$ distance computations. Given the very high cost of I/O operations in MapReduce, we do not recommend using this solution. Moreover, as mentioned above, other techniques already exist to optimize nearest neighbor search that can perfectly be added to our implementation.

Next, in the original G-means algorithm, new centers are picked at the beginning of each iteration. Implementing this directly in MapReduce would require an additional MapReduce job. To minimize the number of jobs executed at each iteration and the number of dataset reads, we merge this operation with the last iteration of k-means. Thus, the `KMeansAndFindNewCenters` operation will perform classical k-means and at the same time find 2 new centers (c_1 and c_2) for each cluster, which will be used at next iteration of G-means.

Finally, while the sequential algorithm analyzes clusters individually, and thus adds new centers sequentially, the MapReduce version analyzes all clusters in parallel and will thus try to double the number of centers at each iteration. As a result, it may eventually overestimate the value of k. Future versions of the algorithm will thus add a post-processing step to merge close centers.

One of the subtleties of the MapReduce version of G-means, as proposed in Algorithm 1, is that each iteration has to deal with centers from previous, current and next iteration:

- `KMeans` refines the centers of current iteration;
- `KMeansAndFindNewCenters` picks centers that will be used at next iteration;
- `TestClusters` assigns each point to its cluster (a center from previous iteration), then projects it on the vector joining the 2 corresponding centers of current iteration.

3.1 KMeans and Find New Centers

`KMeansAndFindNewCenters` is a MapReduce job with combiners that performs two operations at the same time:

1. Run k-means to refine current centers;
2. For each current center, pick two new centers (c_1 and c_2) that will possibly be used at next iteration.

In our implementation, the new centers are chosen randomly. More sophisticated algorithms can be used to select the new points, but they may require an additional MapReduce job.

Algorithm 2 KMeansAndFindNewCenters Mapper

Input: point (text)
Output:
centerid (long) ⇒ coordinates (float[]), 1 (int)
centerid + *OFFSET* (long) ⇒ coordinates (float[]), 1 (int)

procedure MAP(*key*, *point*)
  Find nearest *center*
  Emit(*centerid*, *point*)
  Emit(*centerid* + *OFFSET*, *point*)
end procedure

[1]A combiner is a well-known pre-aggregation optimization available in MapReduce.

The Map step of the job is presented in Algorithm 2. The coordinates of each point are emitted twice. This doubles the quantity of data to be shuffled and transmitted over the network. However, this effect is largely mitigated by the use of a combiner. The efficiency of the combiner is of course very dependent of the dataset. There are recent execution engines (such as SPARK[2]) that allow to specify "partition-preserving" operations. Preserving partitions would help the combiners to perform more efficiently at next iteration. It is however outside the scope of this paper to consider such optimizations.

To make the distinction between coordinates that correspond to new centers to be used at next iteration of the algorithm and current centers that we want to refine with k-means, we use an arbitrary high offset value. More precisely, as the type of `center id` is a Java Long, we use an offset value equal to half the largest possible value of a Java Long. The value of `OFFSET` is thus 2^{62} (approximatively $4E18$). This also limits our algorithm to datasets with at most 2^{62} centers.

We could also use a text prefix, but although simpler to interpret, this choice would hurt performance due to the requirement of an additional parsing phase. Moreover, during the shuffle phase, sorting text keys requires more processing than simple integer values.

The combiner and reducer test the value of the key. If it is larger than the predefined offset, they keep only 2 new centers per cluster. Otherwise they perform classical k-means reduction and compute the new position of each cluster center.

3.2 Test Clusters

The `TestClusters` procedure is is the last MapReduce job of our distributed G-means implementation (Algorithm 1). The mapper projects the points of a cluster on the line joining the two centers (c_1 and c_2) and the reducer then tests if these values follow a normal distribution.

At the first steps of G-means, when k is low, this algorithm performs poorly as the parallelism of the reduce phase is bounded by k.

To achieve higher parallelism, the algorithm adopts another strategy when k is low, called `TestFewClusters` (Algorithm 5. The test for normality is directly performed by the mapper, thus on subsets of data. This of course only delivers correct results if the number of samples for each subset is sufficient, which we can suppose is verified for low values of k. Anderson-Darling is a powerful statistical test, which has proved being reliable even with small samples (as a rule of thumb, a minimum size of 8 is considered to be sufficient). In our implementation we use a threshold of 20, to stay on the safe side. The number of reduce tasks is still equal to k (which is low), but as their task is only to combine the decisions taken by mappers, this will not limit the performance of the algorithm.

[2] `http://www.spark-project.org/`

Algorithm 3 TestClusters Mapper

Input: *point* (text)
Output: *vectorid* (int) ⇒ *projection* (double)

procedure SETUP
  Build vectors from center pairs
  Read centers from previous iteration
end procedure

procedure MAP(*key*, *point*)
  Find nearest *center*
  Find corresponding *vector*
  Compute *projection* of *point* on *vector*
  Emit(*vectorid*, *projection*)
end procedure

Algorithm 4 TestClusters Reducer

Input: *vectorid* (int) ⇒< *projection* (double) >

procedure REDUCE(*vectorid*, *projections*)
  Read *projections* to build a *vector*
  Normalize *vector* (mean 0, stddev 1)
  ADTEST(*vector*)
  if normal **then**
    Mark cluster as found
  end if
end procedure

Moreover, `TestFewClusters` limits the size of the vector of projections to a level that fits into RAM memory: If we assume that the value of a point in each dimension is stored as a string of approximatively 15 characters (the number of significant decimal digits of IEEE 754 double-precision floating-point format), and each character is encoded using 1 Byte, the number of points in a dataset is $O(\frac{S}{15D})$, where S is the size of the dataset (in Bytes) and D is the number of dimensions.

For each point, the algorithm will compute a projection, encoded as a double (8 Bytes). The total memory space needed to store all projections is thus $O(8\frac{S}{10D})$ and thus $O(\frac{S}{D})$ Bytes, which can be very large. In the worst case scenario, if all points of the dataset belong to the same cluster, as a result of the `TestClusters` procedure, the amount of memory required by a single combiner will be prohibitive.

When `TestFewClusters` is used, the quantity of memory required by each mapper to store the projections will be $O(\frac{Ss}{D})$, where Ss is the size of a single split (64MB on a default Hadoop installation), which is now completely reasonable.

Choosing when to switch from one strategy to the

Algorithm 5 TestFewClusters Mapper

Input: *point* (text)
Output: *vectorid* (int) $\Rightarrow A^{*2}$ (double)

procedure SETUP
  Build vectors from center pairs
  Read centers from previous iteration
end procedure

procedure MAP(*key*, *point*)
  Find nearest *center*
  Find corresponding *vector*
  Compute *projection* of *point* on *vector*
  Add *projection* to *list vectorid*
end procedure

procedure CLOSE
  for Each *list* **do**
    Read projections to build a *vector*
    Normalize *vector* (mean 0 , stddev 1)
    Compute $A^{*2} = adtest(vector)$
    Emit($vectorid \Rightarrow A^{*2}$)
  end for
end procedure

other is, as often, a matter of compromise.

If the algorithm switches too late (i.e., when k is large), the algorithm will keep using the `TestFewClusters` strategy, even for a large number of clusters. As the test for normality is performed by the mapper, there is a risk that the number of points in some clusters is smaller than the threshold. The mapper is then not able to compute a decision.

If the algorithm switches too early (i.e., when k is small), the test is performed by the reducers even for a small number of clusters. There is a risk that the number of projections received by a single reducer becomes too large and exhausts the heap: in the worst case, the maximum amount of memory required by a single reducer will be $O(\frac{S}{D})$ Bytes (if the complete dataset belongs to a single cluster), and in the best case it will be $O(\frac{S}{kD})$ Bytes (if all k clusters have the same number of points).

In our MapReduce implementation of G-means, at each iteration the algorithm counts the number of points that belong to each cluster. By doing so, the algorithm can estimate the maximum amount of heap memory that will be required as the number of points belonging to the biggest cluster multiplied by the average quantity of heap memory required per point (that we determined experimentally).

When an algorithm uses almost all heap memory available, the Java Virtual Machine (JVM) has to regularly trigger the garbage collector to make room for new objects and variables, which seriously degrades performance. To avoid this, we use a maximum heap usage coefficient.

The algorithm will thus first use the `TestFewClusters` strategy, and switch to the other strategy only when the following two conditions are met: the number of clusters to test is larger than the total reduce capacity, and the estimated maximum amount of required heap memory is less than 66% of the heap memory of the JVM.

As illustration, Figure 1 shows the centers found by successive iterations of our final MapReduce G-means algorithm for a subset of data, consisting of 10 clusters in $\mathbb{R}^2$. At each iteration the algorithm splits clusters in 2, except clusters that pass the test, and optimizes centers position using k-means. The algorithm finally finds 14 centers, as shown in Figure 4.

4. COST MODELIZATION

We now estimate the cost of MapReduce G-means clustering. More precisely, we estimate the number of dataset reads, the number of computations and the quantity of data that is shuffled.

Each iteration of G-means consists of three steps: `KMeans`, `KMeansAndFindNewCenters`, and `TestClusters`.

Each iteration of `KMeans` requires one dataset read[3], $O(kn)$ distance computations, and shuffles $O(n)$ coordinates in worst case (if no combiner is used). As the new centers are placed in an efficient way, where they are really needed, we found experimentally that only two k-means iterations are sufficient.

`KMeansAndFindNewCenters` consists of a single k-means iteration, but the mapper will emit each point a second time to pick two new centers for each current center. It also requires one dataset read, $O(kn)$ distance computations and, without combiner, shuffles $O(2n)$ coordinates.

`TestClusters` requires one dataset read. It computes $O(kn)$ distances, $O(n)$ projections and performs $O(k)$ anderson-darling tests. For large values of k (thus for a large number of clusters), this will be dominated by distances computation and anderson-darling tests. The step also shuffles $O(n)$ projections.

If the algorithm starts with a single cluster at iteration 0, at iteration i it is updating $2^{i+1} = 2k$ centers to test $2^i = k$ possible clusters. At iteration i, the total number of clusters that have been tested is

$$1 + 2 + 4 + ... + 2^i = \sum_{j=0}^{i} 2^j = 2^{i+1} - 1$$

The number of iterations required to test values of k

[3]Depending on the underlying execution engine, it may be possible to avoid subsequent dataset reads. This is the case for example with SPARK, where you can cache the dataset in memory and make sure to preserve the data partitioning.

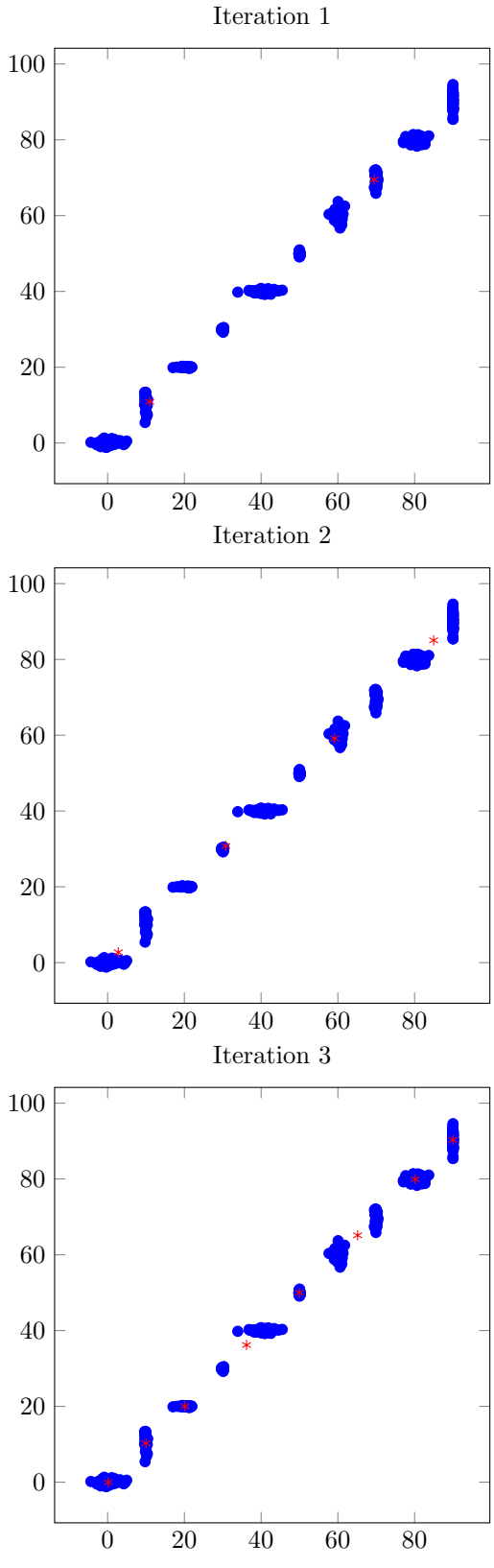

Figure 1: Evolution of centers positioned by G-means in a dataset containing 10 clusters in $\mathbb{R}^2$

between 1 and k_{real} is theoretically

$$n = \log_2 k_{real}$$

In practice a few additional iterations are required because MapReduce G-means tends to overestimate the number of clusters, and because some clusters are discovered before others.

The $\sum k$ for all iterations of G-means is:

$$\sum_{j=0}^{n} k = \sum_{j=0}^{n} 2^j = 2^{n+1} - 1 \simeq O(2^{\log_2 k_{real}+1} - 1) = O(2k_{real})$$

In total, G-means algorithm requires $O(4 \log_2 k_{real})$ dataset reads, computation of $O(4n \sum k) = O(8nk_{real})$ distances and $O(\sum k) = 2k_{real}$ anderson-darling tests.

The algorithm is thus able to find k with a number of computations that remains proportional to k_{real}! The price to pay is an iterative processing, that requires $O(\log_2 k_{real})$ iterations, and thus $O(\log_2 k_{real})$ dataset reads.

At the other side, the classical way to find k is to use a MapReduce implementation of k-means, to let it run for different values of k, and to use one of the criteria described above to find the bet value of k. However, this is not efficient.

To compare MapReduce versions of k-means and G-means in a fair way, we used another implementation, multi-k-means, that computes the centers for all possible values of k at each iteration. The mapper step is presented by algorithm 6. The combiner and reducer are classical.

Algorithm 6 Multi-k-means Mapper

Input: *point* (text)
Output: *k_centerid* (text) $\Rightarrow$ *coordinates* (float[]), 1 (int)

```
procedure MAP(key, point)
    for k = k_min; k <= k_max; k+ = k_step do
        Find nearest center
        Emit(k_centerid ⇒ point)
    end for
end procedure
```

The main drawback is of course that number of distances computed and the quantity of data that is shuffled and transmitted over the network at each iteration of k-means are much bigger. But the quantity of data to shuffle is largely reduced by using the combiner. So this drawback is largely outbalanced by the advantage that all possible values of k can be tested in a single round, thus vastly reducing the number of iterations and dataset reads!

To test all values of k between 1 and k_{max}, the total number of centers computed by multi-k-means is:

$$\sum_{j=1}^{k_{max}} j = \frac{k(k+1)}{2} \simeq O(k^2)$$

At each iteration, multi-k-means requires 1 dataset read, $O(nk_{max}^2)$ distance computations and shuffles $O(nk_{max})$ coordinates if no combiner is used.

Clearly, from a theoretical point of view G-means has a huge advantage over multi-k-means, as the number of computations remains proportional with k_{real} instead of k_{max}^2. It does, however need $O(\log_2 k_{real})$ iterations, and thus $O(\log_2 k_{real})$ dataset reads.

For example, for a dataset containing 100 clusters, G-means theoretically requires 7 iterations, and thus $O(800n)$ distance computations, $O(200)$ anderson-darling tests and 28 dataset reads. At the other side, for such a small value, multi-k-means already requires $O(10000n)$ distance computations at each iteration!

Moreover, G-means stops processing when k is found, while multi-k-means has to process all possible values of k before taking a decision. As G-means adds new centers progressively, where they are required, it reduces the probability to get stuck in a local minimum, while this can be the case for multi-k-means if initial centers are poorly chosen. A production version of multi-k-means thus requires multiple runs with different starting points, or an additional job to select initial centers, for example using canopy clustering[15], or an algorithm that avoids local optima [11]. Finally, once the centers have been computed for different values of k, multi-k-means requires at least one additional job to find the correct value of k.

In any way, there is a risk that because of skewed data, some reducers will have a higher workload, thus reducing the global efficiency of the algorithm. Handling skewed data in MapReduce is a whole subject by itself and is left as future work.

5. EXPERIMENTAL RESULTS

To test the algorithms we propose in this paper, we use a Hadoop implementation and run tests on a cluster consisting of 4 nodes. Each node is equipped with 2 quad-core Xeon processors and 32GB of RAM.

As a first experiment, we want to estimate the quantity of heap memory required by the reducer of the `TestClusters` step of Algorithm 1. Therefore we run the algorithm with different datasets that consist of a variable number of points. During the first iteration of the algorithm, all points belong to a single cluster, and will thus be tested by a single reducer. We iteratively run the algorithm, and let the amount of heap memory vary. When the quantity of available heap memory becomes to small, the job crashes with an error ("Java heap space"). The results are shown on Figure 2.

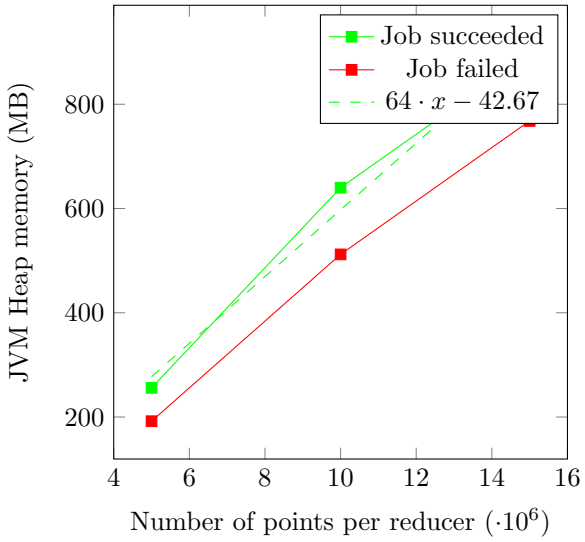

Figure 2: Estimation of the amount of heap memory required by the reducer of the step `TestClusters`

Linear regression shows our reducer requires approximatively 64 Bytes (8 doubles) per point. This value can certainly be further optimized, but it is not the goal of this paper to create a production level version of the algorithm. So for all other tests, the algorithm uses that value of 64 to estimate the quantity of heap memory required by the reducer of the `TestClusters` step, and to decide when to switch from one strategy to the other.

We now turn to the experiments performed in order to test our G-means algorithm on different synthetic datasets. We used five datasets of 10M points (in $\mathbb{R}^{10}$) generated using a Gaussian distribution, and using a variable number of clusters ranging from 100 up to 1600. Table 1 shows for each dataset the real number of clusters, the number of clusters discovered by G-means, as well as the number of iterations and time required.

Table 1: Results of G-means clustering

	d100	d200	d400	d800	d1600
Clusters	100	200	400	800	1600
Discovered	134	305	626	1264	2455
Time (sec)	1286	1667	2291	4208	5593
Iterations	9	10	11	13	13

As expected, the algorithm overestimates the number of clusters. The proportion of discovered clusters to the real number of clusters seems to be quite constant (1.5). The algorithm thus requires a post-processing step to merge clusters, the development of which is left as future work.

The number of iterations is also slightly greater than the theoretical value $(1 + \log_2 k)$. As some centers are discovered before the last iteration, the algorithm will not create new centers for them. It may thus require 1 ore more additional iterations to discover all centers.

Finally, as expected, the execution time scales linearly with k.

We then compare G-means with a hadoop implementation of multi-k-means. For each dataset, multi-k-means computes the position of centers for all values of k between one and the real number of clusters in the dataset. Table 2 shows the average computing time for a single iteration.

Table 2: Average time of a single iteration of multi-k-means

	d50	d100	d141	d200	d400
Clusters	50	100	141	200	400
Time (sec)	237	751	1356	2637	10252

The execution time of both G-means and multi-k-means are graphed in Figure 3.

We observe now that the execution time of multi-k-means rises exponentially. Moreover, for a single iteration of multi-k-means, and for a value of k as low as 100, G-means already outperforms multi-k-means.

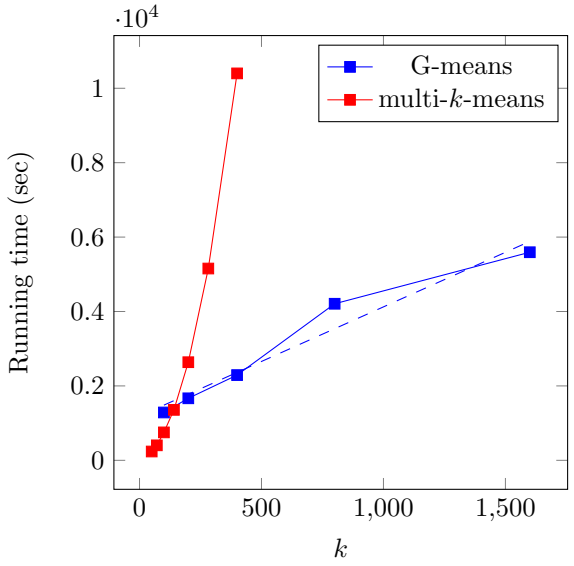

Figure 3: Running time of G-means and multi-k-means

We also want to evaluate the consistency of the clustering results provided by both algorithms. The final goal of k-means clustering is to partition the n data points into k sets $S = S_1, S_2, ..., S_k$ so as to minimize the within-cluster sum of squares (WCSS):

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

where $\boldsymbol{\mu}_i$ is the mean of points in $\mathbf{S_i}$. Therefore we use WCSS as a measure of the quality of clustering.

For different datasets, Table 3 shows the real number of clusters in the dataset, the number of clusters discovered by G-means, the average distance between points and their centers discovered by G-means, and the average distance between points and centers when using multi-k-means for the same value of k. A smaller value means the position of the centers is better. For multi-k-means, we let the algorithm run 10 iterations, which is enough to find a stable solution.

Table 3: Real number of clusters in each dataset, number of clusters discovered by G-means, and average distance between points and centers found by G-means and multi-k-means

	d100	d200	d400
k_{real}	100	200	400
k_{found}	150	279	639
G-means	3.34	3.33	3.23
multi-k-means	3.71	3.60	3.39

As can be seen in Table 3, G-means consistently outperforms multi-k-means, by approximatively 10%. This is mainly due to the fact G-means progressively adds new centers, if and where they are needed. This reduces the probability to fall into a local minimum.

This effect is illustrated on Figure 4. Both plots show a small dataset consisting of 10 clusters. The upper plot shows the 14 centers found by G-means. This is more than the real number of clusters, but the clusters are correctly detected. The lower plot shows the centers found by multi-k-means after 10 iterations, for $k = 10$. Two centers are located in the penultimate cluster, around $(80, 80)$. Although the multi-k-means searches the position of the correct number of centers, it falls into a local minimum and does not detect the correct clusters, which results in a larger average distance between point and center.

Finally, to test the scalability of the algorithm, we generate a dataset consisting of 100M points (in $\mathbb{R}^10$) distributed in 1000 clusters using a Gaussian distribution. We then run our MR G-means algorithm on 4, 8 and 12 nodes. All tests completed after 13 iterations of G-means. The respective running times are shown in Table 4 and on Figure 5.

We can observe that the running time decreases roughly linearly with the number of nodes, which shows that our algorithm and Hadoop can take advantage of additional

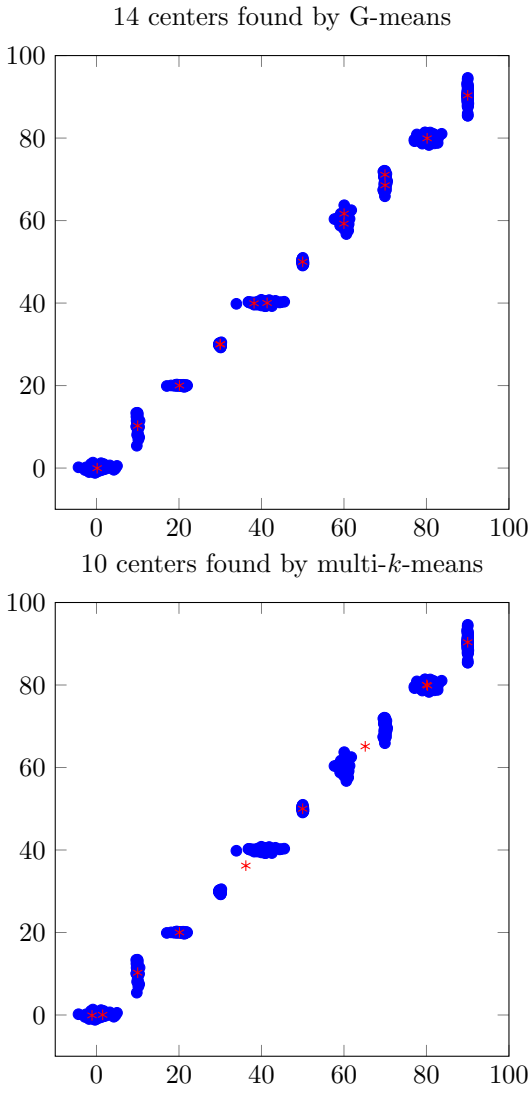

Figure 4: Results of MR G-means and multi-k-means

Table 4: Running time of MR G-means to cluster a dataset of 100M points

	T4	T8	T12
Nodes	4	8	12
Time (min)	798	447	323

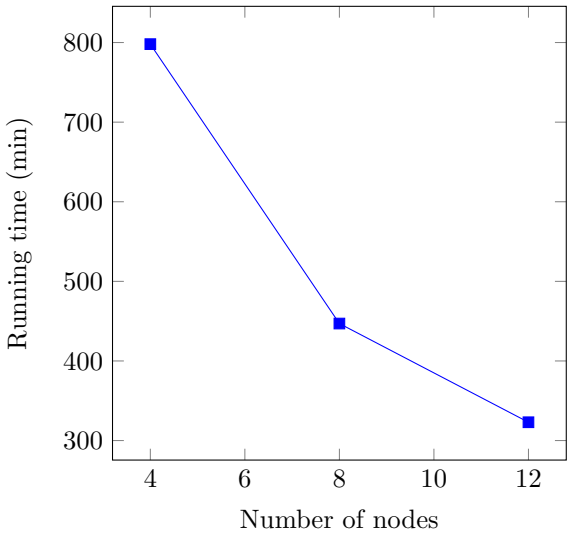

Figure 5: Running time of MR G-means to cluster a dataset of 100M points

nodes and thus scale to very large datasets.

6. CONCLUSIONS AND FUTURE WORK

Despite its known drawbacks and alternative techniques, k-means [14] is still a largely used clustering algorithm. It also has a lot of variants and optimizations. One of these variants, G-means, is able to automatically determine k, the number of clusters. In this paper we proposed a MapReduce implementation of G-means that is able to estimate k with a computation cost that is proportional to k.

We ran experiments that confirm that the processing time scales linearly with k. These experiments also show that, because G-means adds new centers progressively, if and where they are needed, it reduces the probability to fall into a local minimum, and eventually finds better centers than classical k-means processing.

The algorithm still has some caveats, as it tends to constantly overestimate the number of clusters, but it definitely deserves interest when it comes to clustering large datasets consisting of an unknown number of clusters.

As future work, we plan to explore ways to extend our MapReduce implementation of G-means by leveraging more advanced batch execution engine (e.g. SPARK) which can provide advanced configuration options at

run-time in order to save unnecessary disk I/O operations via in-memory caching and partition-preserving operations. We also plan to run additional experiments on a larger cluster to evaluate further the performance and scalability of the algorithm.

Acknowledgement

This work has been partially supported by the EU project BigFoot (FP7-ICT-317858).

7. REFERENCES

[1] Charu C. Aggarwal, Joel L. Wolf, Philip S. Yu, Cecilia Procopiuc, and Jong Soo Park. Fast algorithms for projected clustering. *SIGMOD Rec.*, 28(2):61–72, June 1999.

[2] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. *SIGMOD Rec.*, 28(2):49–60, June 1999.

[3] David Arthur and S Vassilvitskii. k-means++: The advantages of careful seeding. *Proceedings of the eighteenth annual ACM–SIAM symposium on Discrete algorithms*, 2007.

[4] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. *Proc. VLDB Endow.*, 5(7):622–633, March 2012.

[5] Jeremy Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419–428, 2001.

[6] Rudi Cilibrasi and Paul M. B. Vitnyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51:1523–1545, 2005.

[7] J. C. Dunn. A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. *Journal of Cybernetics*, 3(3):32–57, 1973.

[8] Martin Ester, Hans-Peter Kriegel, Jörg S, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996.

[9] Greg Hamerly and Charles Elkan. Learning the k in k -means. In *Neural Information Processing Systems*. MIT Press, 2003.

[10] S. Har-Peled and S. Mazumdar. Coresets for k-means and k-median clustering and their applications. pages 291–300, 2004.

[11] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A k-means clustering algorithm. *Applied Statistics*, 28(1), 1979.

[12] Anil K Jain. Data Clustering : 50 Years Beyond K-Means. *Pattern Recognition Letters*, 2009.

[13] Stuart P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[14] J. MacQueen. Some methods for classification and analysis of multivariate observations. Proc. 5th Berkeley Symp. Math. Stat. Probab., Univ. Calif. 1965/66, 1, 281-297 (1967)., 1967.

[15] Andrew McCallum, Kamal Nigam, and Lyle H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 169–178, New York, NY, USA, 2000. ACM.

[16] Dan Pelleg and Andrew Moore. Accelerating exact k-means algorithms with geometric reasoning. *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, 1999.

[17] Dan Pelleg and AW Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. *ICML*, 2000.

[18] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20(0):53 – 65, 1987.

[19] Catherine A Sugar and Gareth M James. Finding the number of clusters in a dataset. *Journal of the American Statistical Association*, 98(463):750–763, 2003.

[20] RobertL. Thorndike. Who belongs in the family? *Psychometrika*, 18(4):267–276, 1953.

[21] Robert Tibshirani, Guenther Walther, and Trevor Hastie. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2):411–423, 2001.

[22] M.N. Vrahatis, B. Boutsinas, P. Alevizos, and G. Pavlides. The New k-Windows Algorithm for Improving the k-Means Clustering Algorithm. *Journal of Complexity*, 18(1):375–391, March 2002.