University of Pisa

Artificial Intelligence and Data Engineering
Cloud Computing

K-means on MapReduce

The **Don Matteo** group:

Dario Pagani	585281
Ricky Marinsalda	585094
Giulio Bello	603078

June 27, 2023

Contents

Chapter 1

Algorithm to select initial centroids

1.1 Idea

To run the *K-means* algorithm is necessary to select k initial centroids, they can either be selected statically by the user — that is as an algorithm's parameter — or stochastically from the data; in the latter case they could either be drawn randomly with uniform probability or by employing a probability function that changes the likelihood to draw an element accordingly to certain metrics. For simplicity's sake our implementation chooses the k initial point with equal probability.

We used a *Map-Reduce* procedure to draw those k points.

1.2 Implementation

Key This *MapReduce* procedure doesn't use a key, so only one reducer will be spawned by the framework, in our *Hadoop* implementation we used the `NullWritable` data-type as output key to reduce traffic, as such object serializes to zero bytes.

Mapper The mapper assigns a random label to each data point, this value is not be confused with the *MapReduce*'s key; finally it emits the tuple made of the label and the data point.

Combiner and Reducer Then, the combiner sorts the tuples by their label and emits the first k smallest labels and their samples. Finally, the reducer performs the same operations as the combiner, emitting k values.

Probability If the data are split equally among the nodes and the random numbers generator generates all numbers with equal probability, then all dataset's samples have circa equal probability to be drawn.

Complexity Let $n = |D|$ be the number of samples, N the number of mapper tasks and k the number of samples to draw; then the time complexity is

$$T \in \mathcal{O}\left(\frac{n}{N} \cdot (1 + \log(k)) + N \cdot k\right)$$

and the space complexity is

$$S \in \Theta\left(k\right)$$

if a sorted data structure is used to store only the first k smallest labels and their associated data. The I/O complexity is optimal as we perform only linear scans of the file.

Algorithm 1 Random select

Require: $k \in \mathbb{N}^+$

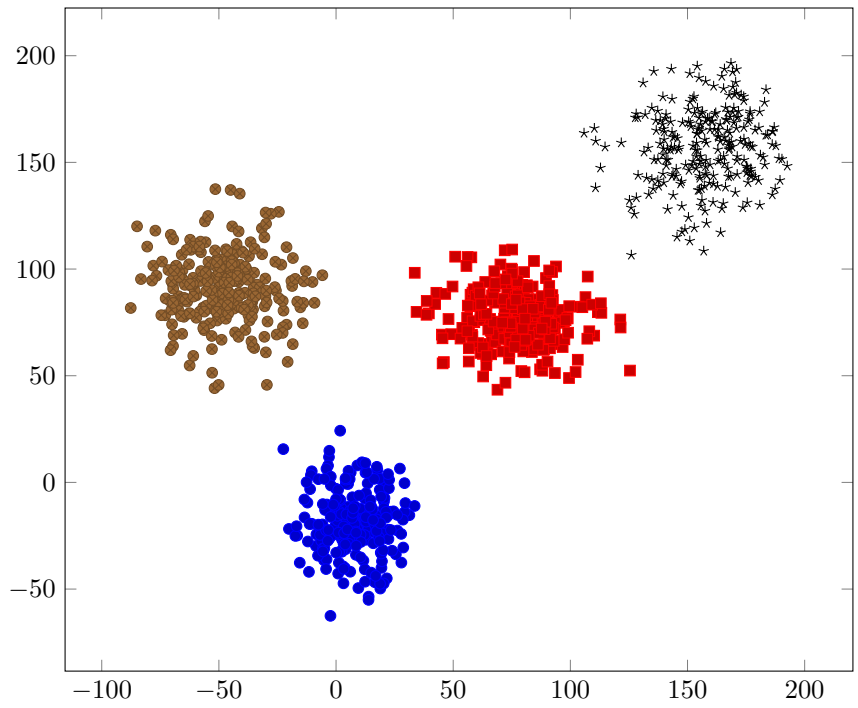
```
procedure MAPPER(nLine, r)
    I ← RAND( )                          ▷ Assign a random number to each file's row
    return ⟨null, ⟨I, r⟩⟩                ▷ Constant key for all lines
end procedure

procedure COMBINER(S)
    L is a data structure ordered by key I   ▷ E.g. a binary search tree
    L ← ∅
    while S ≠ ∅ do
        ⟨null, ⟨I, r⟩⟩ ← READ(S)         ▷ Read data from the mapper (or combiners)
        INSERT(L, ⟨I, r⟩)
        if |L| > k then                  ▷ We store at most k + 1 elements
            POP_LAST(L)                  ▷ We remove the last element (sorted by I)
        end if
        NEXT(S)
    end while
    for all ⟨I, r⟩ ∈ L do
        EMITT(⟨null, ⟨I, r⟩⟩)
    end for
end procedure

procedure REDUCER(S)
    COMBINER(S)                          ▷ Same as the combiner
end procedure
```

Figure 1.1: Example of generation with $k = 4$

Example of syntethic clusters ($k = 4$), we generate also noise but it was ommited



Chapter 2

K-means

2.1 General idea

Algorithm 2 describes how an iteration of K-means is performed, several iterations are performed until the exit condition is reached. An iteration starts with a set of clusters' centroids C and at its end produces a new set C', the procedure is said to have converged when:

$$\forall i \in \{1, \cdots, k\} \qquad |c_i - {c'}_i| < \epsilon \tag{2.1}$$

since — if bad start centroids were selected — K-means may take a large number of iterations to converge, a maximum number of iterations is set as parameter of the problem.

2.1.1 Average builder

Let us define an **AverageBuilder** as a tuple that contains information necessary to compute an average over a set of points $p_1, \ldots, p_s$:

$$B = \left\langle \sum_{i=1}^{s} p_i, s \right\rangle = \langle \Sigma, s \rangle \qquad\qquad p_i \in \mathbb{R}^d \wedge s \in \mathbb{N} \tag{2.2}$$

in this way we can define a function to compute the average:

$$\textsc{ComputeAverage}(B) = \frac{\Sigma}{s} = \frac{1}{s} \cdot \sum_{i=1}^{s} p_i \tag{2.3}$$

and another to merge two objects:

$$\textsc{MergeBuilder}(B_a, B_b) = \langle \Sigma_a + \Sigma_b, s_a + s_b \rangle \tag{2.4}$$

the latter will be useful to gradually *build* an average from a stream of data points, like in combiner and reducer.

2.1.2 Mapper

The mapper's role is to assign each point $p \in D$ to the closest centroid $c_t \in C$, its output is a key-value pair, where the key represents the index of the nearest centroid t, and the value is an *AverageBuilder* with just one point stored in it.

Input The KMeansMapper algorithm takes three parameters as input:

- k: The key represents the identifier of the data point
- $p \in \mathbb{R}^d$: The data point to be assigned to a centroid
- $C = \{c_1, \ldots, c_k\}$: A list of points that are clusters' centroids

Output The mapper's output is in the form $\langle t, \langle p, 1 \rangle \rangle$, where t is the closest centroid's index.

Example Usage Suppose we have a K-means clustering problem with three clusters and the following data point and centroids:

- Data point: $p = (1, 2, 3)$
- Centroids: $C = \{(4, 5, 6), (7, 8, 9), (10, 11, 12)\}$

Applying the KMeansMapper algorithm to the data point and centroids, we get the following result:

$$\textsc{Mapper}(k, p, C) \longrightarrow \langle 1, \langle (1, 2, 3), 1 \rangle \rangle$$

This means that the data point p is closer to the centroid of index 1 $(4, 5, 6)$ and such as it belong to that centroid's cluster.

Complexity Analysis The time complexity of the KMeansMapper algorithm depends on the number of centroids in the *centroids*' list, let's denote this number as k. A call to the procedure requires a complete visit of the C set, thus performing $\Theta(k)$ iterations, each iteration requires the computation of the expression $|p - c_i|$ that it's done with $\Theta(d)$ operations; thus achieving a total time complexity of

$$T_{\textsc{Mapper}} \in \Theta\left(k \cdot d\right)$$

The space complexity of the algorithm is $\Theta(d)$ since it needs to store an additional vector to keep track of the nearest centroid while iterating through C.

$$S_{\textsc{Mapper}} \in \Theta\left(d\right)$$

2.1.3 Combiner and reducer

The Reducer's role is to compute a new set of clusters C', this is done by spawning a Reducer for each cluster $1, \ldots, k$ and each Reducer task computes the centroid's cluster's new central point by averaging all point within. In order to reduce the I/O complexity of the MapReduce program, we have to introduce a Combiner procedure that reduces the number of element to transmit over the network.

As stated before both procedures operate on AverageBuilder objects.

Fundamentally the reducer's and combiner's implementations are almost the same, the only difference is that the reducer computes the average of the points by performing the operation ComputeAverage as described in Formula 2.3.

Input The **KMeansReducer** algorithm takes two parameters as input:

- k: The cluster's index
- S: A stream of AverageBuilders that contains information regarding the cluster's points

Output The Combiner's output is the same as the mapper's. The Reducer's output is a tuple $\langle k, {c'}_k \rangle$, the second component represents the new cluster's centroid for the next iteration of *K-mean*, if the exit condition is not met.

Complexity Analysis

$$T_{\text{Combiner}} \in \mathcal{O}\left(\frac{n}{N} \cdot d\right)$$

$$S_{\text{Combiner}} \in \Theta(d)$$

The same figure in space complexity applies to the Reducer; while the Reducer's time complexity is:

$$T_{\text{Reducer}} \in \mathcal{O}\left(N \cdot d + d\right) = \mathcal{O}\left(N \cdot d\right)$$

as an instance of Reducer will have to merge the averages coming from N Combiner tasks for a certain cluster c_i and as the ComputeAverage's time complexity is of class $\Theta(d)$.

Algorithm 2 KMeans' iteration for MapReduce

Require: $B = \langle \Sigma, s \rangle$ to be an AverageBuilder, as described in Section 2.1.1
Require: $k \in \mathbb{N}^+$ to be the number of clusters to generate
Require: $C = \{c_1, \ldots, c_k\}$ to be the centroids' set

procedure ComputeAverage(B) ▷ The same as Equation 2.3
  return $\frac{1}{s} \sum_{i=1}^{s} p_i$
end procedure

procedure MergeBuilder(B_a, B_b) ▷ The same as Equation 2.4
  return $\langle \Sigma_a + \Sigma_b, s_a + s_b \rangle$
end procedure

procedure Mapper(offset, p, C)
  closestCentroid $\leftarrow \underset{c \in C}{\text{argmin}} \{\text{Distance}(c, p)\}$ ▷ k iterations
  $B \leftarrow \langle p, 1 \rangle$ ▷ An AverageBuilder object with just p inside
  return $\langle$closestCentroid, $B\rangle$
end procedure

procedure Combiner(centroid's index, S)
  $B \leftarrow \langle 0, 0 \rangle$ ▷ Empty average builder
  for all $B' \in S$ **do**
    $B \leftarrow$ MergeBuilder(B, B')
  end for
  return $\langle k, B \rangle$ ▷ Returns an `AverageBuilder`
end procedure

procedure Reducer(centroid's index, S)
  $B \leftarrow$ Combiner(centroid's index, S) ▷ To merge the builders we re-use the combiner
  return ComputeAverage(B) ▷ New cluster's centroid
end procedure

2.1.4 General complexity of a single iteration

Hadoop splits the n tuples over multiple tasks on N cluster's nodes; for each tuple, each node has to perform a call to the Mapper and each Mapper's result is streamed to the Combiner, notice that since the results are streamed to the combiner, it'll have to only merge them in time $\Theta(d)$ and since $d \in \mathcal{O}(k \cdot d)$ we can omit the term; finally all Hadoop's nodes stream their Combiners' results to k Reducers. Those operations have a complexity of:

$$T_{\text{Kmeans}} \in \mathcal{O}\left(\frac{n}{N} \cdot k \cdot d \;\; + \;\; k \cdot (N \cdot d)\right) \simeq \mathcal{O}\left(\frac{n}{N} \cdot k \cdot d\right) \tag{2.5}$$

and

$$S_{\text{Kmeans}} \in \Theta\left(k + d\right)$$

I/O complexity In principle, if no combiner is used, the number of bytes O_{KMEANS} to be transmitted over the network would be of class $\Theta(n \cdot d)$, since the mappers would transmit all of their data tuples to the Reducers; however, if the combiner is added to the job, we can reduce greatly this figure, saving a lot of I/O over the network:

$$O_{\text{KMEANS}} \in \mathcal{O}(N \cdot k \cdot d)$$

2.2 Implementation

Let's take a look at the main classes and data types we implemented to realize our *MapReduce* application on Hadoop.

`Point` Class The `Point` class represents a point in a d-dimensional space:

$$p = (p_1, \ldots, p_d)$$

It provides the following functionalities:

- Computation of the norm
- Computation of sum between two points and scalar product with a number
- toString function
- Implementation of the `Writable` interface to support serialization and de-serialization

I/O It takes $(4 + 8d)$ bytes to serialize a `Point` object.

`AverageBuilder` Class It provides the following functionalities:

- Storage of the sum of points and their cardinality (number of points)
- Methods to add points to the computation, either individually or by combining with another `AverageBuilder` object
- Computation and retrieval of the average point
- Implementation of the `Writable` interface for serialization and de-serialization

The `Point` class handles point-related operations in the Mapper and Reduce classes, while the `AverageBuilder` class facilitates efficient computation of the average of multiple points used in the combiner and reducer.

I/O It takes $(4 + 8d + 8)$ bytes to serialize an `AverageBuilder` object.

Total I/O If no combiner is employed, the application would have to transmit $(4 + 8d + 8) \cdot n$ bytes over the network, for example a dataset of $n = 4 \times 10^6$ samples with $d = 6$ features and $k = 4$ would generate 240MB [1] of traffic; on the other hand, if the combiner is used, it would have to transmit $(4 + 8d + 8) \cdot N \cdot k$ bytes over the network, using the same example and assuming $N = 2$, it'd have to send just 480 bytes, shaving off **two orders of magnitude**!

We can reduce even further those figures by refactoring the implementation a little: it's possible to make the d field from `Point` static — since it's a constant parameter of the algorithm — saving 4 bytes; and it's possible to use a variable length serializer for the `cardinality` field of `AverageBuilder` reducing its size to just a couple of bytes.

[1] Not MiB

Chapter 3

Results

3.1 Synthetic data

Centroids To test our program we had to generate synthetic data. We wrote a program that generates k points $c_1, \ldots, c_k$, the centroids, by drawing them from a random variable $C \in [-B, B]^d$ that has a continuous uniform distribution of probability in its domain.

Cluster's data For each generated centroid c_i, we generate a *Gaussian standard vector* $X_i \sim \mathcal{N}_d(\mu, \Sigma)$ with mean $\mu = c_i$ and the variance of each component σ_{jj} with $j = 1, \ldots, d$ is drawn from a random variable with uniform distribution of probability, this allows the program to generate ellipsoidal clusters. Finally, the program iterates through all centroids' random vectors $X_0, ..., X_k$ until it generates n points, those points are streamed to `stdout` and written to a file.

Noise On top of the data we added some noise from a random uniform variable $N \in [-B - \epsilon, B + \epsilon]^d$.

Comparisons We had to run the program several times to generate different files with different sizes, for comparison's sake it is necessary for the clusters' shapes and centroids to be the same every time the program is ran; to achieve this it uses the same *seed* for its pseudo-random number generator used to initialize its random variables' parameters.

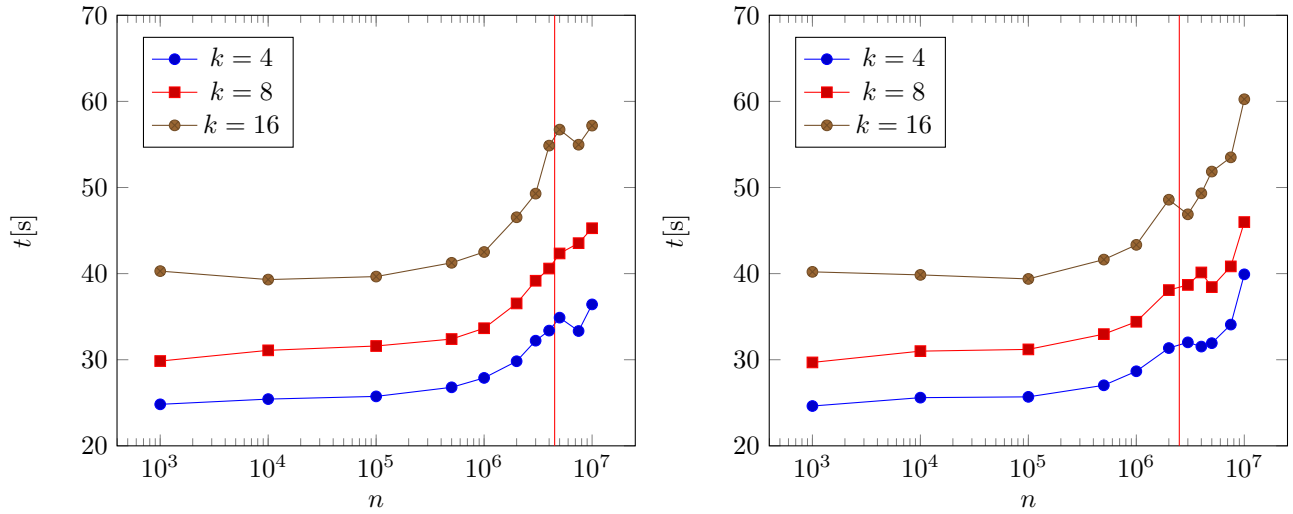
We can see an example of synthetic clusters in Figure 1.1

3.2 Benchmark

Methodology We ran our application on synthetic datasets of various combinations of n, d, k, we used a Bash script to log information about each execution's exit code, execution time, number of iterations and total time to run all iterations' jobs; this procedure was done a couple of times for each dataset. Finally, we computed the average time to execute a K-means iteration for each execution and the average for all runs of the same dataset (n, k, d).

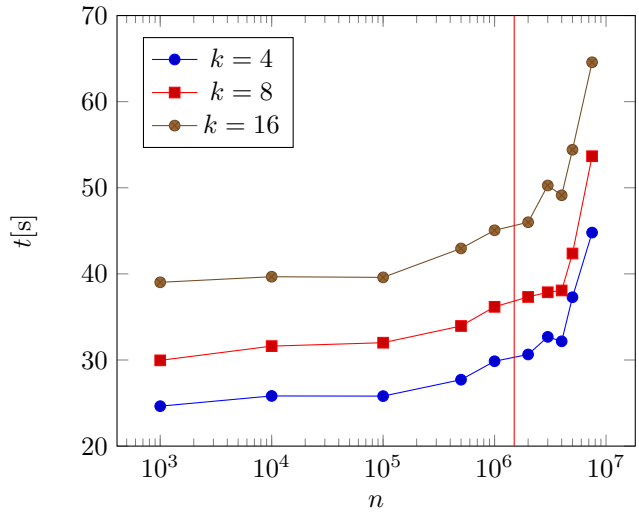
Results We can observe that, after a certain $\bar{n}$, all execution times start to show a linear dependency in n; a similar observation holds true with k, however — strangely enough — we observe that when d is increased, the execution time slightly drops with respect to k, this might be due to CPUs' pipelining capabilities, since most of the d-dependent operations occur in loops of such length.

Execution time We can observe that, independently from the dataset's dimension n and dimensionality d, it takes a certain time to bootstrap the Hadoop job; in particular we have to read and write k centroids to HDFS. We've observed that such time is $1.25k + 20$ seconds, if we remove such time from our figures, we see that application's complexity (Equation 2.5) holds true for $n > 10^6$ with a constant factor of circa 100×10^{-9} seconds.

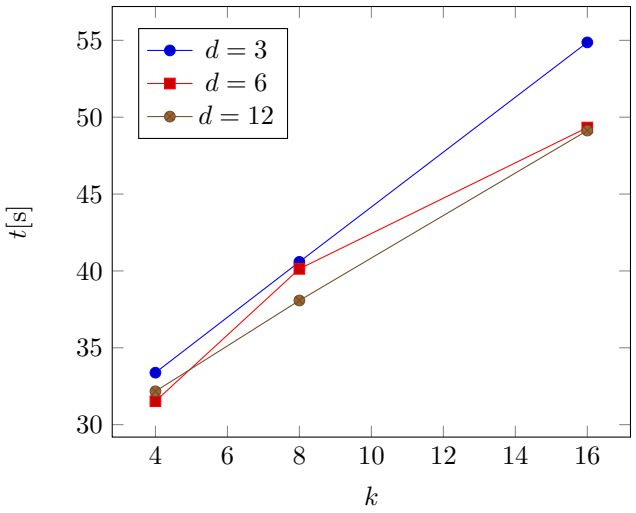


(a) Execution times with $d = 3$

(b) Execution times with $d = 6$



(c) Execution times with $d = 12$



(d) Execution times with $n = 4 \times 10^6$

Figure 3.1: Execution time of a single iteration of K-means. Right of red line $|F| > 128$ MiB

Parallelism Notice how there's a drop in time when the file's size passes 128 MiB — that's when n passes the red line — that's because its contents are spanning over two HDFS's chunks, so Hadoop is very likely to spawn two tasks and assign one chunk to each; in spite of this parallelism, we're not able to observe a halving of in execution time.

Considerations We see that the MapReduce framework begins to make sense when we work with large amount of data that can be split over multiple computing nodes; otherwise the system's overhead is so large that the execution time remains — in our experiments — constant for smaller datasets. Of course, we benchmarked a system made only of three servers and **very** limited resources, in a real scenario with more powerful machines the execution time may be better.

Improvements Finally, we have to state that it might be possible to improve the procedure's speed by replacing the double precision floating point numbers with just single precision ones or even, with appropriate considerations, with integers as the ALU is orders of magnitude faster than the FPU; however the latter solution is difficult to implement as we'd have to properly scale values and we'd have to compute roots of integral numbers.

3.3 Comparison

Initial centroids We used Skitlearn's *k-means++* procedure to select good initial centroids, then we used them in both Skitlearn's k-means and ours.

Results We obtained similar results in most cases, an example is shown in Figure 3.2; however is some cases we obtained different results, for instance, in a dataset in which two cluster were close together, Skitlearn's implementation was able to properly separate them, while ours was not.

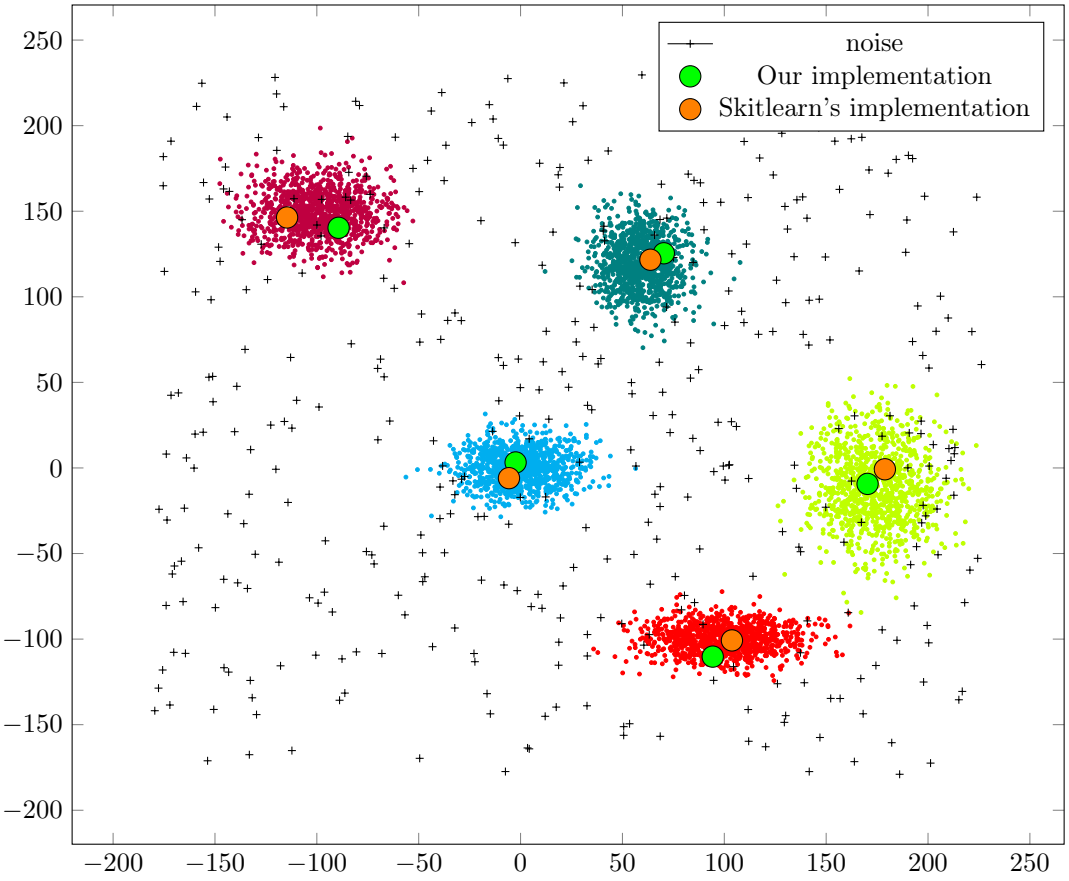


Figure 3.2: Comparison between two K-means implementations