Optimized K-Means Clustering Algorithm in Haskell

Arnau Abella
Universitat Politècnica de Catalunya

April 14, 2020

1 Introduction

The *k-means* clustering algorithm aims to partition n observations into k clusters in which each observation belongs to the cluster with the *nearest mean* (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into *Voronoi cells.* It is popular for cluster analysis in data mining.

This paper describes an *optimized* implementation in Haskell, the purely funcitonal programming language, of the *standard k-means clustering* algorithm, see algorithm 1. This project also provides a *library* with a very simple API to use. Furthemore, it include a couple of utils to plot the result of the clustering, see in figure 2.

1.1 The algorithm

Given a set of observations $(x_1, \ldots, x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into $k \leq n$ sets, $S = (S_1, \ldots, S_k)$, so as to minimize the *within-cluster sum of squares (WCSS)*. Formally, the objective is to find:

$$\arg\min_S \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2 \tag{1}$$

The algorithm implementation is *straightforward* and it can be easily implemented in any turing-complete programming language [ZJ14].

Algorithm 1: Standard K-Means algorithm

```
1. Choose k data objects representing the cluster centroids
2. Assign each data objet of the entire data set to the cluster having the
   closest centroid
3. Compute new centroid for each cluster, by averaging the data objects
   belonging to the cluster
4. If at least one of the centroids has changed, go to step 2, otherwise go
   to step 5
5. Output the clusters
```

The number of clusters k can be found by drawing an Elbow plot of the $WSCC$ error, see figure 1.

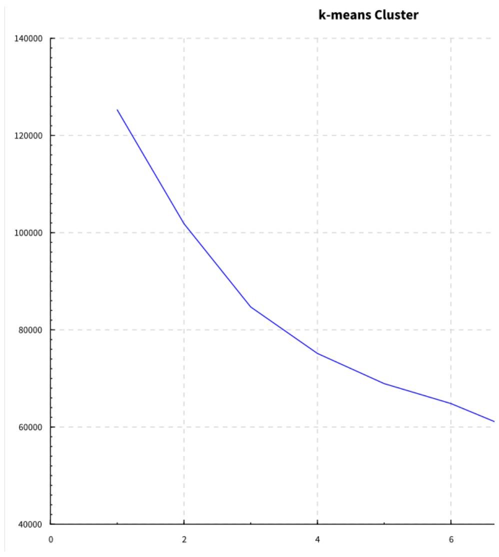


Figure 1: Elbow Plot of the *WCSS* dependening on k.

1.2 Complexity

The complexity of the standard algorith in d-dimensions is $\mathcal{O}(t \cdot n \cdot k \cdot d)$, where t is the number of iterations until local optimum, n is the number of observations, k is the number of clusters, and d is the number of dimensions.

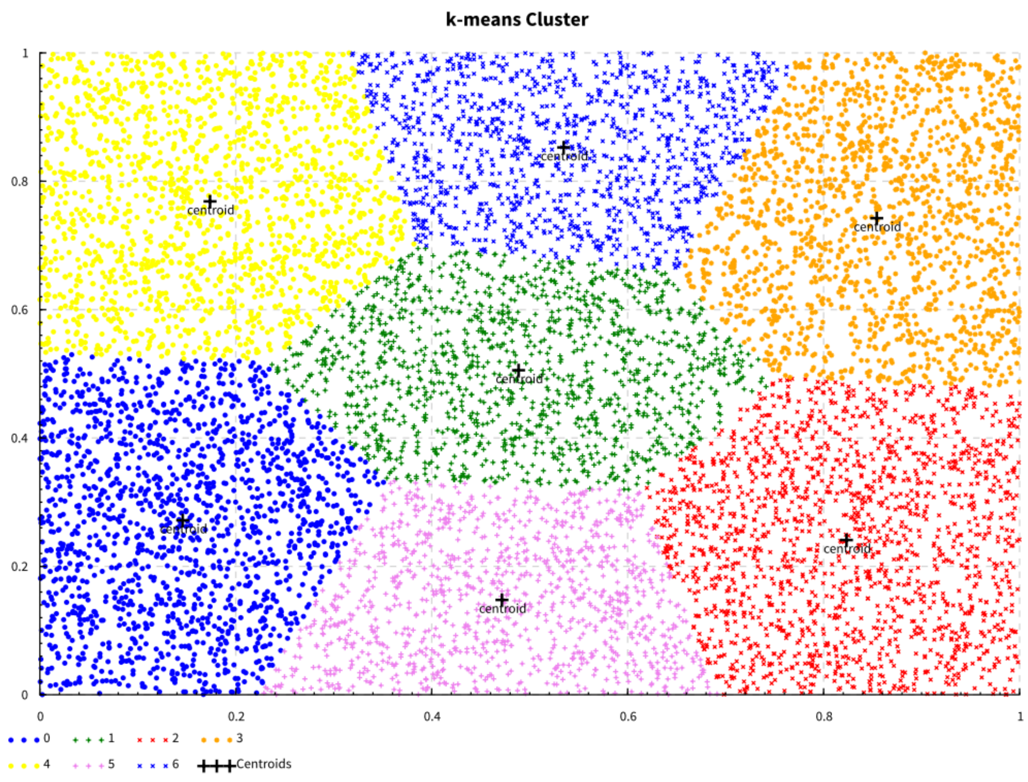


Figure 2: Example of k-means clustering for 100,000 observations.

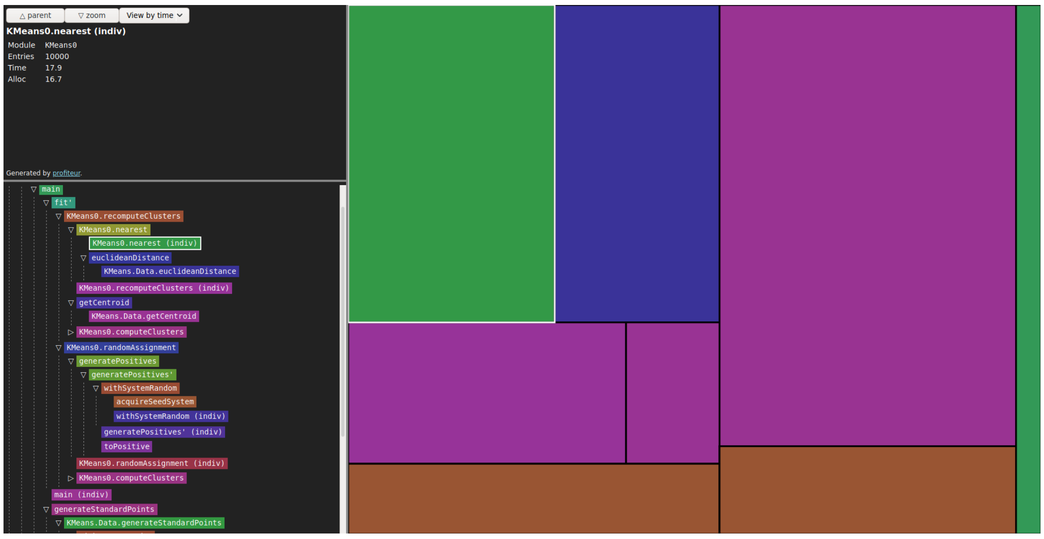
2 The code

2.1 Implementation

The first implementation of the algorithm from *pseudo-code* to real code took performance into consideration by using an array-like data structure (from the library `vector`) and points was represented in an efficient way by using *unboxed* types (types that are represented by words in registers instead of pointers). The code can be found at `KMeans0.hs`.

After finishing the first implementation, we wanted to measure its performance using a consistent and objective method. We used a well-known benchmark framework called `criterion` [O'S]. We wrote a couple of benchmarks, see `Benchmark.hs`. The initial results were not good as it took more than one minute to process less than $5,000$ 2-dimensional points.

In order to see what was making the implementation so slow, we wrote a simple program (see `./profile/Main0.hs`) and we used GHC's profiling capabilities [Tea] to generate a *prof* file which contained detailed statistics of the execution time, garbage collection pauses, execution trace, etc. `profiteur` [dJ] is a nice tool for visualizing the *prof* files which are usually long and hard to read.

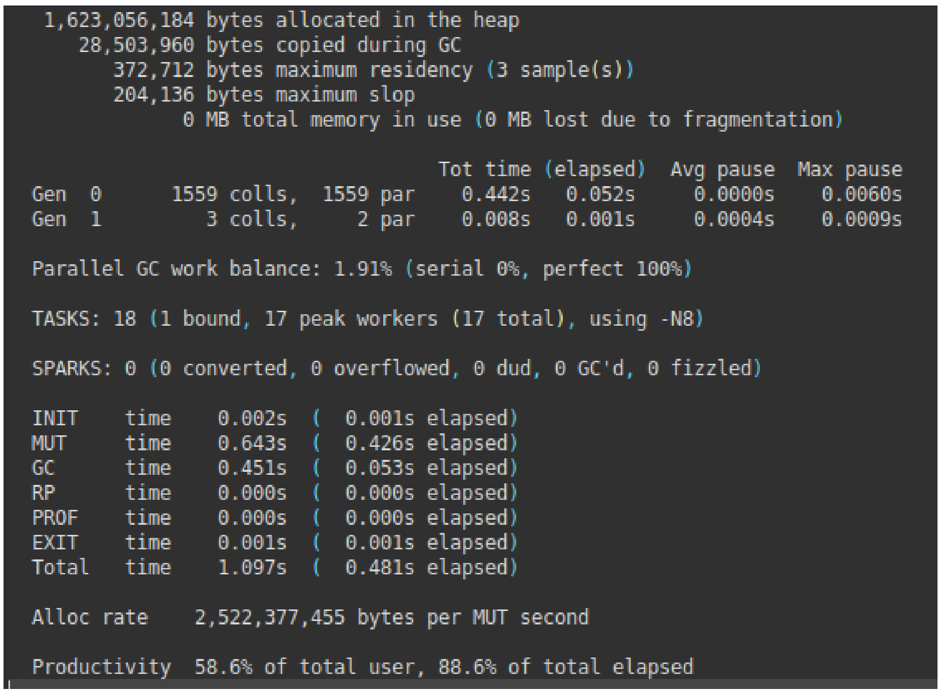


```
KMeans0.nearest (indiv)
Module   KMeans0
Entries  10000
Time     17.9
Alloc    16.7

Generated by profiteur.

▽ main
  ▽ fit'
    ▽ KMeans0.recomputeClusters
      ▽ KMeans0.nearest
          KMeans0.nearest (indiv)
        ▽ euclideanDistance
            KMeans.Data.euclideanDistance
        KMeans0.recomputeClusters (indiv)
      ▽ getCentroid
          KMeans.Data.getCentroid
      ▷ KMeans0.computeClusters
    ▽ KMeans0.randomAssignment
      ▽ generatePositives
        ▽ generatePositives'
          ▽ withSystemRandom
              acquireSeedSystem
              withSystemRandom (indiv)
            generatePositives' (indiv)
            toPositive
        KMeans0.randomAssignment (indiv)
      ▷ KMeans0.computeClusters
    main (indiv)
  ▽ generateStandardPoints
    ▽ KMeans.Data.generateStandardPoints
```

Figure 3: profiteur, graphic visualization of *prof* files.

2.2 Optimizations

The profiling gave us some hints about were was the bottleneck (see Figure 4). As you may see, the garbage collector (GC) is taking up to half of the execution time so this is probably due to a spaceleak which is a known problem of lazy evaluation. Spaceleaks are hard to find but easy to solve. We followed a novel technique to find where the memory was being accumulated by restricting the maximum amount of stack. After finding where the memory was leaking, we forced the evaluation of the big thunk of memory to avoid the garbage collector going ill. To apply strictness in some points of the code, we used the great haskell extension `BangPatterns` which gives a nice sugar-syntax to call `seq and deepseq` primitives to force evaluation.

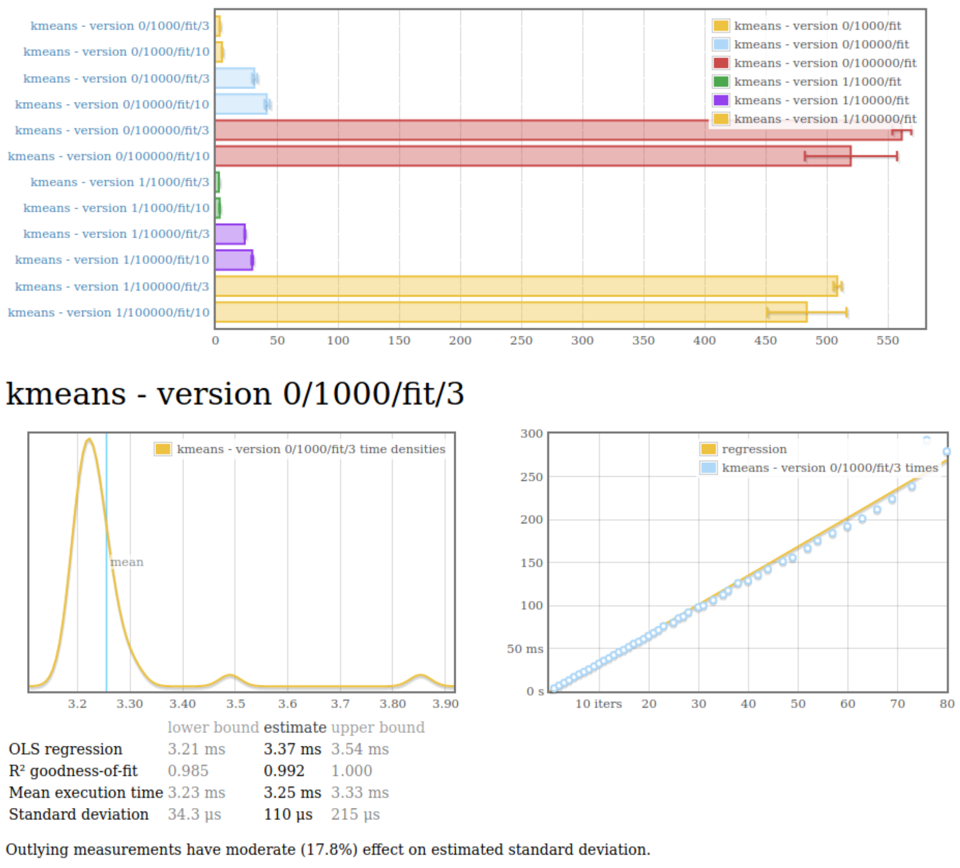


```
  1,623,056,184 bytes allocated in the heap
     28,503,960 bytes copied during GC
        372,712 bytes maximum residency (3 sample(s))
        204,136 bytes maximum slop
              0 MB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      1559 colls,  1559 par    0.442s   0.052s     0.0000s    0.0060s
  Gen  1         3 colls,     2 par    0.008s   0.001s     0.0004s    0.0009s

  Parallel GC work balance: 1.91% (serial 0%, perfect 100%)

  TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.002s  (  0.001s elapsed)
  MUT     time    0.643s  (  0.426s elapsed)
  GC      time    0.451s  (  0.053s elapsed)
  RP      time    0.000s  (  0.000s elapsed)
  PROF    time    0.000s  (  0.000s elapsed)
  EXIT    time    0.001s  (  0.001s elapsed)
  Total   time    1.097s  (  0.481s elapsed)

  Alloc rate    2,522,377,455 bytes per MUT second

  Productivity  58.6% of total user, 88.6% of total elapsed
```

Figure 4: Profile statistics from GHC's profiling.

Apart from fixing the spaceleak that gave us a huge speed-up, we applied some microoptimizations to the hot spots of our algorithm that are the `euclideanDistance` function and the `constructCluster` function.

Figure 5 displays the final results for different values of n (observations) and k (clusters). For a better visualization, open `./bench/bench.html` in your favourite browser and play with the interactive plots and nice statistics produced by `criterion`.



	lower bound	estimate	upper bound
OLS regression	3.21 ms	3.37 ms	3.54 ms
R² goodness-of-fit	0.985	0.992	1.000
Mean execution time	3.23 ms	3.25 ms	3.33 ms
Standard deviation	34.3 μs	110 μs	215 μs

Outlying measurements have moderate (17.8%) effect on estimated standard deviation.

Figure 5: Benchmark of `KMeans0` and `KMeans1`.

3 Conclusions & Future Work

This project described and implemented an optimized *standard k-means clustering algorithm* in Haskell by applying low-level optimitizations, also known as *micro-optimizations*, such as strict annotations and inlineable code. This process required to understand Haskell evaluation and performance issues. We got assistance from really great tools to profile and benchmark our application and the result is quite good compared to the first implementation.

A major speed-up can be achieved by implementing a more efficient version of your algorithm. For example, C.M. Poteras et al. proposed an optimized version of the K-Means clustering algorithm which achieves an observable speed up of 1.7x on large data sets [PMM14]. Implementing the optimized algorithm 2 in Haskell is feasible and an interesting task for future work.

Algorithm 2: Optimized K-Means algorithm

```
1. Defined constant $WIDTH$
2. Define intervals $I_i = [i * WIDTH, (i+1) * WIDTH]$ and tag them with
   value $i * WIDTH$
3. Mark the entire data set to be visited
4. For each point to be visited
5. Compute $e = min(d_{PC_l} - d_{PC_w})$ where $C_w$ is the center of the winner
   (closest) cluster and $C_l, l = 1..k, l \neq w$ stands for all other centroids.
6. Map all points with $i * WIDTH < e < (i+1) * WIDTH$ to interval
   $i * WIDTH$ where $i$ is a positive integer.
7. Compute new centroids $C_j$, where $j = 1..k$ and their maximum
   deviation $D = max(|C_j C_{j'}|)$
8. Update $I_j$'s tag by substracting $2 * D$ (points owned by this interval
   go closer to the edge by $2 * D$)
9. Pick up all points inside intervals whose tag is less or equal to 0, and
   go to 4 to revisit them.
```

References

[dJ] Jasper Van der Jeug. Profiteur: Visualiser for ghc's *.prof.

[O'S] Brian O'Sullivan. Criterion: robust, reliable performance measurement and analysis.

[PMM14] C.M. Poteras, M.C. Mihaescu, and M. Mocanu. An optimized version of te k-means clustering algorithm. Technical Report ACSIS, Vol 2, University of Craiova, 2014.

[Tea] GHC Team. Ghc manual: Profiling.

[ZJ14] Mohammed J. Zaki and Wagner Meira Jr. *Data Mining and Analysis: Fundamental Concepts and Algorithms.* Cambridge University Press, 2014.