Time Series Clustering using K-Means Algorithm and Performance Optimizations with Rcpp and GPUTools

Aman Taxali

ataxali@umich.edu

Abstract

In this report we explore the clustering and computing performance of the K-Means algorithm. In the first section, we begin by discussing the algorithmic complexity and nuances of K-Means (Lloyd's Algorithm). We conclude this section by conducting a case study that employs clustering and Monte Carlo simulation on time series data.

In the second section, we implement one computationally complex subtask of the K-Means algorithm in 3 different programming archetypes. The performance of an iterative Rcpp implementation is compared against a vectorized R solution. Next, we introduce some facts about running R programs on umich's Flux GPU cluster, and summarize the performance of an R implementation that takes advantage of GPU acceleration.

We close with a short discussion on the types of problems that are best suited for each programming archetype.

Introduction to K-Means Algorithm

To get acquainted with K-Means (Lloyd's Algorithm), we breakdown its steps in pseudocode:

```
1. Randomly select initial cluster centers
2. Repeat (until convergence)
    a. for each (observation in dataset)
        • Compute Euclidean distance to each cluster center
    b. Assign each observation to nearest cluster
    c. Update cluster centers to mean of all observations in each cluster
```

We note that the steps highlighted in yellow above have algorithmic complexity O(k), where k is the number of iterations needed for convergence. The step highlighted in grey has complexity O(n×c), where n is the total number of observations and c is the number of clusters. In total, K-Means with Lloyd's Algorithm has complexity O(k×n×c), where typically $c \ll k \ll n$. It is also easy to see that this complexity has lower and upper bounds O(n) and O(n^2).

Monte Carlo Simulation of K-Means

Partitioning n observations into k clusters is a NP-hard problem. K-Means is a heuristic algorithm that converges to a local optimum. Furthermore, one of the nuances of Lloyd's Algorithm is that clustering results are sensitive to the choice of initial cluster centers.

One way to address these limitations is to conduct multiple independent runs of K-Means clustering, and to aggregate the statistics of interest across all runs at the end.

Case Study: Clustering Time Series Data

We use Monte Carlo simulation of K-Means to identify the average first and last days of the summer season for Chicago[1] and San Francisco[2], using historical time series weather data.

Since the weather datasets span multiple years, we first need to 'fold' the temperature recordings onto a continuous time range of 0 to 12, where 0 is January 1 of each year and 12 is December 31 of each year. Next, this data is partitioned into 3 clusters (corresponding to 'spring, 'summer' and 'winter' seasons). Our statistics of interest, the first and last days of summer, are the edge observations of the middle cluster (ignoring outliers).

We formulate the Monte Carlo analysis as follows:

```
1. for (step in 1:Number_of_MC_Iterations)
    a. Run K-Means clustering on (Date, Temperature) observations, with randomly selected initial cluster centers
2. For each observation, compute the most popular (mode) cluster allocation
3. Find first/last seven consecutive days that get allocated to middle cluster (corresponding to first/last full weeks of summer); and extract first/last days from these ranges respectively. This approach ignores outlier cluster edges.
```

Convergence in Monte Carlo Analysis

As expected, table below shows that our statistics of interest converge as number of Monte Carlo iterations increases (for both data sets). Further, we also see that the variance within cluster groups also decreases as the number of Monte Carlo iterations increases.

City	Statistic	Number of M.C. Iterations			
		1	5	10	20
Chicago	First Day of Summer	May 29	May 28	May 25	May 24
	Last Day of Summer	Oct 06	Oct 08	Oct 11	Oct 12
	Intra-Cluster Variance	167.51	166.25	164.83	163.61
San Fran.	First Day of Summer	Jun 07	May 27	Jun 02	Jun 01
	Last Day of Summer	Sep 13	Sep 13	Sep 15	Sep 15
	Intra-Cluster Variance	250.26	237.43	233.19	233.11

Table 1: Average summer in Chicago spans May 24-Oct 12, and in San Francisco from June 1-Sep 15

K-Means Runtime Performance

As seen in the introduction to this section, we confirm a multiplicative linear runtime cost between number of observations (n) and iterations for convergence (k).

Number of Observations (n)	Iterations to Convergence (k)			
	10	50	100	500
100	0.438	1.304	1.368	2.854
500	0.582	1.697	1.713	3.386
1000	1.824	1.982	2.138	4.061
5000	2.158	2.455	3.377	7.791

Table 2: Runtime of 10 K-Means simulations with 3 clusters (in seconds)

1. Chicago Beach Weather Station Data (2015-2017): https://catalog.data.gov/dataset/beach-weather-stations-automated-sensors
2. San Francisco Road Weather Info. Station Data (2014-2017): https://catalog.data.gov/dataset/road-weather-information-stations-788f8

Extending K-Means with Rcpp/GPUTools

In the previous section, we identified that a significant portion of the algorithmic complexity of K-Means is attributed to the repeated Euclidean Distance calculations between observations and cluster centers (total cost: O(n×c)). In this section, we implement this calculation three different ways and compare their performances. We use the SFPUC[1] time series temperature data for our performance profiles.

Euclidean Distance: Rcpp vs. Vectorized R

The Rcpp implementation of the 2-dimensional Euclidean Distance function is purely iterative (see Appendix 1.1). In contrast, the same function in R uses vector operations (see Appendix 1.2).

The table below displays the expected linear effect of the number of observations (n) and clusters (c) on runtime. The Rcpp implementation always outperformed the vectorized R function.

Number of Observations (n)	Number of Clusters (c)			
	5	10	20	50
100	3.99%	8.16%	2.26%	4.55%
500	11.64%	19.26%	9.46%	4.66%
1000	3.19%	14.16%	23.32%	32.77%
5000	1.61%	17.73%	27.95%	34.23%

Table 3: Relative improvement in runtime of Rcpp Euc. Dist. over R implimentation

Running R Programs on the GPU

Running computations on specialized hardware like GPUs typically requires low-level programming. However, with the growing popularity of computationally complex analyses, several R packages are being developed that allow high-level R programs to be accelerated using GPU hardware.

Accessing UMich's Flux GPU Cluster with R

Flux offers NVIDIA GPUs which are best accessed via CUDA. OpenCL is the other alternative, however NVIDIA GPUs only support OpenCL 1.x.

This greatly limits the GPU interfacing packages we are able to install in R on Flux. For example, even after several days of manually modifying the install and config scripts (and many helpful conversations with ARC support), the latest stable build of gpuR still could not be made to work on Flux.

As suggested by ARC support, the current easiest way for R programs to interface with GPU hardware is via the gputools module installed on Flux.

Euclidean Distance using GPUTools

The package gputools makes available a small selection of matrix operations that are accelerated on GPU hardware. One of these functions is the matrix cross-product. We note that the Euclidean Distances between two matrices of points can be found by:

1. Subtracting the two matrices
2. Taking the cross-product of the difference from step 1, with itself
3. Taking the square root of the diagonal entries of result from step 2

The algorithmic complexity of this approach is $O(n^2)$, which is clearly much slower than the Rcpp or vectorized R implementations previously discussed.

Euclidean Distance: GPU vs. CPU

We implement the matrix approach for Euclidean Distances using the gpuTcrossprod function from gputools and the tcrossprod function from base R.

The summary table below shows that the GPU implementation is slower than the CPU version when matrix sizes are small. Performance improvements from GPU hardware arise when matrix sizes are large.

Number of Observations (n)	Runtime by Implimentation (seconds)		
	GPU	CPU	**CPU/GPU**
100	1.1094	1.0831	**0.9763**
500	1.3527	1.3346	**0.9866**
1000	2.2811	2.3689	**1.0385**
5000	3.9127	4.2921	**1.0970**

Table 4: Matrix cross-prod runtime for gputools and base R (single processor for both)

Conclusions on Programming Archetypes

Rcpp offers the most versatile performance enhancements to base R. GPU programming is best done at low-level, and the gputools package is ideal only for extremely large matrix calculations.

1. SFPUC Wind Monitoring Data (2016): https://catalog.data.gov/dataset/san-francisco-wind-monitoring-data-prior-year-88c02

Appendix

1.1 Euclidean Distance in Rcpp

```
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector euc_dist_cpp(NumericMatrix x,
NumericMatrix y) {
  int xrow = x.nrow();
  NumericVector out(xrow);

  for (int i = 0; i < xrow; i++) {
    out[i] =  sqrt(pow(x(i,0)-y(i,0), 2) + pow(x(i,1)-y(i,1),
2));
  }
  return out;
}
```

1.2 Euclidean Distance in R

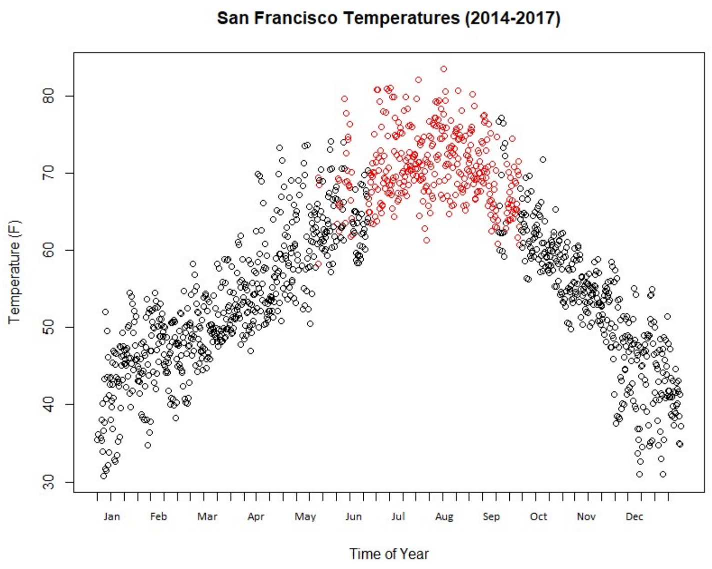
```
euc_dist_r <- function(dat, centers) {
  rowSums((dat - centers)^2)^(1/2)
}
```

1.3 Euclidean Distance Matrix Approach

```
#  argument tcrossprod_f is either gpuTcrossprod or
tcrossprod
euc_dist_crossprod <- function(dat, centers, tcrossprod_f)
{
    subdat = dat - centers
    return(diag(tcrossprod_f(subdat, subdat))^(1/2))
}
```

1.4 K-Means Clustering Visualization

The plot below displays the results of Monte Carlo simulation of 20 K-Means clustering runs on San Francisco weather data. Observations colored in red represent the "summer" cluster.



1.5 K-Means in R

```
# dat and init_centers should be matrixes with one
observation per row and 2 columns
# dist_f is the Euclidean Distance function to use
kmeans_custom <- function(dat, init_centers,
iter_max=10, dist_f, ...) {
  # centers should be randomly initialized by caller, if
following Lloyd's Algorithm
  centers = init_centers
  n_centers = dim(centers)[1]

  # function to calculate Euclidean Distance between
each point and cluster center i
  dists_cent_i <- function(i) {
    dist_f(dat, matrix(rep(centers[i, ], dim(dat)[1]),
              nrow=dim(dat)[1], byrow=TRUE), ...)
  }
  # calculate distance between each point and all cluster
centers
  dists = sapply(mclapply(1:(dim(centers)[1]),
              FUN=dists_cent_i, mc.cores=n_cores),
        function(x) {return(x)})
  row.names(dists) = rownames(dat)
```

```
    # assign each point to nearest cluster
    idx_center = apply(dists, 1, which.min)
    iter = 0

    repeat{
        # update clusters
        center_f = factor(idx_center, levels=1:n_centers)
        centervals = split(idx_center, center_f)

        # update cluster centers
        centers = foreach(i=1:(dim(centers)[1]), .combine = 
rbind) %do% {
            if (length(centervals[[i]]) == 1) {
                dat[names(centervals[[i]]), ]
            } else {
                colMeans(dat[names(centervals[[i]]), ])
            }
        }

        # recompute distances between each point and each 
new cluster center
        new_dists = sapply(mclapply(1:(dim(centers)[1]), 
                              FUN=dists_cent_i, 
mc.cores=n_cores), 
                        function(x) {return(x)})
        row.names(new_dists) = rownames(dat)

        # reassign each point to nearest cluster
        new_idx_center = apply(new_dists, 1, which.min)

        # check for convergence
        if(isTRUE(all.equal(new_idx_center, idx_center))) {
            break
        }

        # check against  iteration limit
        iter = iter + 1
        if(iter > iter_max) {
            warning("K-Means did not converge")
            break
        }

        dists = new_dists
        idx_center = new_idx_center
    }

    # return final cluster allocation for each point
    new_idx_center
}
```

1.6 Monte Carlo Wrapper for K-Means

```
    parallel.kmeans <- function(i) {
        dat = as.matrix(temperature_data)
        rownames(dat) = dat[, index_col]

        # randomly sample to init centers
        init_centers = dat[sample(nrow(dat), num_clusters), ]

        # pass the correct Euclidean Distance function and 
other additional arguments using script args
        if(euc_f == 'euc_dist_r') {
            result = kmeans_custom(as.matrix(dat), 
init_centers, 
                             iter_max=k_means_iter, euc_dist_r)
        } else if (euc_f == 'euc_dist_c') {
            result = kmeans_custom(as.matrix(dat), 
init_centers, 
                             iter_max=k_means_iter, euc_dist_c)
        } else {
            result = kmeans_custom(as.matrix(dat), 
init_centers, 
                             iter_max=k_means_iter, 
euc_dist_crossprod, 
                             get(cp_f), mat_prod_lim)
        }
        return(data.frame(idx=dat[, index_col], 
cluster=as.vector(result)))
    }

    # run each K-Means run in parallel
    results = mclapply(rep(num_clusters, num_iterations), 
FUN=parallel.kmeans, 
                  mc.cores=n_cores)
```

1.7 Data Cleaning for San Francisco Road Weather Data

```
# data source: https://catalog.data.gov/dataset/road-
weather-information-stations-788f8
sf_data = 
fread('./data/Road_Weather_Information_Stations.csv')
sf_data = sf_data[, .(AirTemperature, DateTime)]
sf_data = na.omit(sf_data)
sf_data[, "date" := as.Date(DateTime, "%m/%d/%Y")]
sf_data[, "year" := as.numeric(format(date, "%Y"))]
sf_data[, "month" := as.numeric(format(date, "%m"))]
sf_data[, "day" := as.numeric(format(date, "%d"))]
```

```
sf_data[, "idx" := year + (month-1) +
(day/days_in_months[month])]
sf_data = sf_data[, .(temp = mean(AirTemperature)),
by=.(idx, date, year, month, day)]
sf_data = sf_data[, .(temp, "idx"=(idx-year))]
save(sf_data, file = "sf_data.RData")
```

1.8 Data Cleaning for Chicago Beach Weather Data

```
# source https://catalog.data.gov/dataset/beach-
weather-stations-automated-sensors
ch_data = read_delim('./data/Beach_Weather_Stations_-
_Automated_Sensors.csv',
                delim=',',col_names=TRUE)
ch_data = ch_data %>%
  transmute(DateTime = ch_data$`Measurement
Timestamp`,
          temp = ch_data$`Wet Bulb Temperature`) %>%
  mutate(date = as.Date(DateTime, "%m/%d/%Y")) %>%
  select(-DateTime) %>%
  na.omit() %>%
  mutate(year = as.numeric(format(date, "%Y")),
        month = as.numeric(format(date, "%m")),
        day = as.numeric(format(date, "%d"))) %>%
  mutate(idx = year + (month-1) +
(day/days_in_months[month])) %>%
  group_by(idx, date, year, month, day) %>%
  summarize(temp=mean(temp)) %>%
  ungroup() %>%
  transmute(temp, idx = (idx-year))
save(ch_data, file = "ch_data.RData")
```

1.9 Data Cleaning for SFPUC Wind Montoring Data

```
data_files <- list.files(path = "./data/wdlarchive/2016/",
pattern = ".csv")
data_sets <- NULL
for(file in data_files) {
  dat = read.csv(file = paste0("./data/wdlarchive/2016/",
file), header = TRUE)
  data_sets = rbind(data_sets, dat)
}
data_sets = as.data.table(data_sets)
wd_data = data_sets[, .("date" = Interval_End_Time,
"temp" = Ambient_Temperature_Deg_C)]
wd_data = na.omit(wd_data)
wd_data[, "date" := as.Date(date, "%Y-%m-%d?")]
wd_data[, "year" := as.numeric(format(date, "%Y"))]
wd_data[, "month" := as.numeric(format(date, "%m"))]
wd_data[, "day" := as.numeric(format(date, "%d"))]
wd_data[, "idx" := year + (month-1) +
(day/days_in_months[month])]
wd_data = wd_data[, .(temp = mean(temp)), by=.(idx,
date, year, month, day)]
wd_data = wd_data[, .(temp, "idx"=(idx-year))]
save(wd_data, file = "wd_data.RData")
```