We have covered a basic k-means implementation with rmr in the Tutorial. If you tried it out, though, you probably have noticed that its performance leaves to be desired and wonder if anything can be done about it. Or your have read Efficient rmr techniques and would like to see those suggestions put to work beyond the toy “large sums” example used therein. Then this document should be of interest to you since we will cover an implementation that is modestly more complex and is two orders of magnitude faster. To make the most of it, it's recommended that you read the other two documents first.

First we need to reorganize our data representation a little, creating bulkier records that contain a sizeable subset of the data set each, as opposed to a single point. To this end, instead of storing one point per record we will store a matrix, with one data point per row of this matrix. We'll set the number of rows to 1000 which is enough to reap the benefits of using “vectorised” functions in R but not big enough to hit memory limits in most cases.

```
recsize = 1000
input = to.dfs(lapply(1:100,
           function(i) keyval(NULL, cbind(sample(0:2, recsize, replace = T) + rnorm(recsize, sd = .1),
                                sample(0:3, recsize, replace = T) + rnorm(recsize, sd = .1)))))
```

This is how a sample call would look like, with the first argument being a sample dataset.

```
  kmeans(input, 12, iterations = 5, fast = T)}
```

This creates and processes a dataset with 100,000 data points, organized in 100 records. For a larger data set you would need to increase the number of records only, the size of each record can stay the same. As you may recall, the implementation of kmeans we described in the tutorial was organized in two functions, one containing the main iteration loop and the other computing distances and new centers. The good news is the first function can stay largely the same but for the addition of a flag that tells whether to use the optimized version of the “inner” function, so we don't need to cover it here (the code is in the source under tests, only in the dev branch for now) and a different default for the distance function — more on this soon. The important changes are in the kmeans.iter.fast function, which provides an alternative to the kmeans.iter function in the original implementation. Let's first discuss why we need a different default distance function, and in general why the distance function has a different signature in the fast version. One of the most CPU intensive tasks in this algorithm is computing distances between a candidate center and each data point. If we don't implement this one in an efficient way, we can't hope for an overall efficient implementation. Since it takes about a microsecond to call the simplest function in R (vs. ~10 nanoseconds in C), we need to get a significant amount of work done for each call. Therefore, instead of specifying the distance function as a function of two points, we will switch to a function of one point and and a set thereof that returns that distance between the first argument and each element of the second. In this implementation we will us a matrix instead of a set, since there are powerful primitives available to operate on matrices. The following is the default distance function with this new signature, where we can see that we avoided any explicit loops over the rows of the matrix yy. There are two implicit loops, Reduce and lapply, but internally they used vectorized operators, that is the overhead of those explicit loops is small compared to the time taken by the vectorised operators.

```
fast.dist = function(yy, x) { #compute all the distances between x and rows of yy
   squared.diffs = (t(t(yy) - x))^2
   ##sum the columns, take the root, loop on dimension
   sqrt(Reduce(`+`, lapply(1:dim(yy)[2], function(d) squared.diffs[,d])))}
```

With fast distance computation taken care of, at least for the euclidean case, let's look at the fast implementation of the kmeans iteration.

```
kmeans.iter.fast =
  function(points, distfun, ncenters = dim(centers)[1], centers = NULL) {
```

There is no news here as far as the signature but for a different distance default, so we can move on to the body. The following function is a conversion function that allows us to work around a limitation in the RJSONIO library we are using to serialize R objects. Unserializing a deserialized matrix returns a list of vectors, which we can easily turn into a matrix again. Whenever you have doubts whether the R object you intend to use as an argument or return value of a mapper or reducer will be encoded and decoded correctly, an option is to try RJSONIO::fromJSON(RJSONIO::toJSON(x)) where x is the object of interest. This a price to pay for using a language agnostic serialization scheme.

```
  list.to.matrix = function(l) do.call(rbind,l)
```

The next is the main mapreduce call, which, as in the Tutorial, can have two different map functions: let's look at each in detail.

```
  newCenters = from.dfs(
   mapreduce(
    input = points,
```

The first of the two map functions is used only for the first iteration, when no set of cluster centers is available, only a number, and randomly assigns each point to a center, just as in the Tutorial, but here the matrix argument v represents multiple data points and we need to assign each of them to a center efficiently. Moreover, we are going to switch from computing the means of data points in a cluster to computing their sums and counts, delaying taking the ratio of the two as much as possible. This way we can apply early data reduction as will be clear soon. To achieve this, the first step in the mapper is to extend the matrix of points with a column of counts, all initialized to one. The next line assigns points to clusters using sample. This assignment is then supplied to the function by which applies a column sum to each group of rows in the matrix v of data points, as defined by being closest to the same center. This is where we apply the sum operation at the earliest possible stage — you can see it as an in-map combiner. Also, since the first column of the matrix is now devoted to counts, we are calculating those as well. In the last line, the only thing left is to generate a list of keyvalue pairs, one per center, and return it.

For all iterations after the first, the assignment of points to centers follows a min distance criterion. The first line back-converts v to a matrix whereas the second uses the aforementioned fast.dist function in combination with apply to generate a data points x centers matrix of distances. The next assignment, which takes a few lines, aims to compute the row by row min of this distance matrix and return the index of a column containing the minimum for each row. We can not use the customary function min to accomplish this as it returns only one number, hence we would need to call it for each data point. So we need to use its parallel, less known relative pmin and apply it to the columns of the distance matrix using the combination do.call and lapply. The output of this is a two column matrix where each row contains the index of a row and the column index of the min for that row. The following assignment sorts this matrix so that the results are in the same order as the v matrix. The last few steps are the same as for the first type of map and implement the early reduction we talked about.

In the reduce function, we simply sum over the colums of the matrix of points associated with the same cluster center. Actually, since we have started adding over subgroups of points in the mapper itself, what we are adding here are already partial sums and partial counts (which we store in the first column, as you may recall). Since this is an associative and commutative operation, it can only help to also switch the combiner on. There is on subtle change necessary to do so successfully, which required some debugging even for the author of this document, allegedly a mapreduce expert. In a first version, the reducer returned key value pairs with a NULL key. After all, the reduce phase happens after the shuffle phase, so what use are keys? Not so if the combiner is turned on, as records are first shuffled into the combiner and then re-shuffled into the reducer. So the reducer has to set a key, usually keeping the one set by the mapper (the reducer has to be key-idempotent for the combiner to work)

```
     reduce = function(k, vv) {
          keyval(k, apply(list.to.matrix(vv), 2, sum))},
```

The last few lines are an optional argument to from.dfs that operates a conversion from list to dataframe when possible, the selection of centers with at least a count of one associated point and, at the very last step, converting sums into averages.

```
   newCenters = cbind(newCenters$key, newCenters$val)
   newCenters = newCenters[newCenters[,2] > 0, -1]
   (newCenters/newCenters[,1])[,-1]}
```

To recap, we started by using a slightly different representation with a performance-tunable record size. We have re-implemented essentially the same mapper and reducer functions as in the plain-vanilla version, but using only vectorised, efficient library function that acted on all the data points in one record within a single call. Finally, we have delayed the computation of means and replaced it with the computation of pairs (sum, count), which, thanks to the associativity and commutativity of sums, can be performed on arbitrary subgroups as soon as they are formed, effecting an early data reduction. These transformations exemplify only a subset of the topics covered in [[Efficient rmr techniques]] but are enough to produce a dramatic speedup in this case.