Warning

This page is now outdated and we are in the process of bringing it up to speed with rmr 1.3. Please check the new draft.

Mapreduce in R

My first mapreduce job

Conceptually, mapreduce is not very different than a combination of lapplys and a tapply: transform elements of a list, compute an index — key in mapreduce jargon — and process the groups thus defined. Let's start with a simple lapply example:

```
small.ints = 1:10
out = lapply(small.ints, function(x) x^2)
```

The example is trivial, just computing the first 10 squares, but we just want to get the basics here, there are interesting examples later on. Now to the mapreduce equivalent:

```
small.ints = to.dfs(1:10)
out = mapreduce(input = small.ints, map = function(k,v) keyval(v, v^2))
```

This is all it takes to write your first mapreduce job in `rmr`. There are some difference that we will go through, but the first thing to notice is that it isn't all that different, and just two lines of code. There are some superficial differences and some more fundamental ones. The first line puts the data into HDFS, where the bulk of the data has to be for mapreduce to operate on. Of course, we are unlikely to write out big data with `to.dfs`, certainly not in a scalable way. `to.dfs` is nonetheless very useful for a variety of uses like writing test cases, REPL and HPC-type uses of mapreduce — that is, small data but big CPU demands. `to.dfs` can put the data in a file of your own choosing, but if you don't specify one it will create tempfiles and clean them up when done. The return value is a an object that you can use as a "big data" object. You can assign it to variables, pass it to functions, mapreduce jobs or read it back in. Now onto the second line. It has `mapreduce` replace `lapply`. We prefer named arguments with `mapreduce` because there's quite a few possible arguments, but one could do otherwise. The input is the variable `out` which contains the output of `to.dfs`, that is our small number data set in its HDFS version, but there are other choices as we will see. The function to apply, which is called a map function in contrast with the reduce function, which we are not using here, is a regular R function with a few constraints:

1. It's a function of two arguments, a key and a value
2. It returns a key value pair as returned by the helper function `keyval`, which takes any two R objects as arguments; you can also return a list of such objects, or `NULL`.

In this example, we are not using the key at all, only the value, but we still need both to support the general mapreduce case. The return value is an object, and you can pass it as input to other jobs or read it into memory (watch out, not good for big data) with `from.dfs`. `from.dfs` is complementary `to.dfs`. It returns a list of key-value pairs, which is the most general data type that mapreduce can handle. If you prefer data frames to lists, you can instruct `from.dfs` to perform a conversion to data frames, which will cover many many important use cases but is not as general as a list of pairs (structured vs. unstructured case). `from.dfs` is useful in defining map reduce algorithms whenever a mapreduce job produces something of reasonable size, like a summary, that can fit in memory and needs to be inspected to decide on the next steps, or to visualize it.

My second mapreduce job

We've just created a simple job that was logically equivalent to a lapply but can run on big data. That job had only a map. Now to the reduce part. The closest equivalent in R is arguably a tapply. So here is the example from the R docs:

```
groups = rbinom(32, n = 50, prob = 0.4)
tapply(groups, groups, length)
```

This creates a sample from the binomial and counts how many times each outcome occurred. Now onto the mapreduce equivalent:

```
groups = to.dfs(groups)
mapreduce(input = groups,
          map = function(k,v) keyval(v, 1),
reduce = function(k,vv) keyval(k, length(vv))))
```

First we move the data into HDFS with `to.dfs`. As we said earlier, this is not the normal way in which big data will enter HDFS; it is normally the responsibility of scalable data collection systems such as Flume or Sqoop. In that case we would just specify the HDFS path to the data as input to `mapreduce`. But in this case the input is the variable `groups` which points to where the data is temporarily stored, and the naming and clean up is taken care of for you. All you need to know is how you can use it. There isn't a map function, so it is set to default which is like an identity but consistent with the map requirements, that is `function(k,v) keyval(k,v)`. The reduce function takes two arguments, one is a key and the other is a list of all the values

associated with that key. Like in the map case, the reduce function can return `NULL`, a key value pair as generated by the function `keyval` or a list thereof. The default is somewhat equivalent to an identity function, under the constraints of a reduce function, that is `function(k, vv) lapply(vv, function(v) keyval(k,v))`. In this case they key is one possible outcome of the binomial and the values are all `NULL` and the only important thing is how many there are, so `length` gets the job done. Looking back at this second example, there are some small differences with `tapply` but the overall complexity is very similar.

Wordcount

The word count program has become a sort of "hello world" of the mapreduce world. For a review of how the same task can be accomplished in several languages, but always for map reduce, see this blog entry.

```
wordcount = function (input, output = NULL, pattern = " ") {
  mapreduce(input = input ,
            output = output,
            input.format = "text",
            map = function(k,v) {
                      lapply(
                         strsplit(
                                  x = v,
                                  split = pattern)[[1]],
                         function(w) keyval(w,1))},
                reduce = function(k,vv) {
                    keyval(k, sum(unlist(vv)))},
                combine = T)}
```

We are defining a function, `wordcount`, that encapsulates this job. This may not look like a big deal but it is important. Our main goal was not simply to make it easy to run a mapreduce job but to make mapreduce jobs first class citizens of the R environment and to make it easy to create abstractions based on them. For instance, we wanted to be able to assign the result of a mapreduce job to a variable — and I mean *the result*, not some error code or diagnostics — and to create complex expressions including mapreduce jobs. We take the first step here by creating a function that is itself a job, can be chained with other jobs, executed in a loop etc.

Let's now look at the signature. There is an input and optional output and a pattern that defines what a word is.

The implementation is just a single call to `mapreduce`. Therein, the input can be an HDFS path, the return value of `to.dfs` or another job or a list thereof — potentially, a mix of all three cases, as in `list("a/long/path", to.dfs(...), mapreduce(...), ...)`. The output can be an HDFS path but if it is `NULL` some temporary file will be generated and wrapped in a big data object like the ones generated by `to.dfs`. In either event, the job will return the information about the output, either the path or the big data object. So we simply pass along the input and output of the`wordcount` function to the `mapreduce` call and return whatever it returns. That way the new function also behaves like a proper mapreduce job — almost, more details here. The `input.format` argument allows us to specify the format of the input. The default is based on R's own serialization functions and supports all R data types. In this case we just want to read a text file, so the `"text"` format will create key value pairs with a `NULL`key and a line of text as value. You can easily specify your own input and output formats and even accept and produce binary formats with the functions `make.input.format` and `make.output.format`.

The map function, as we know already, takes two arguments, a key and a value. The key here is not important, indeed always `NULL`. The value contains one line of text, which gets split according to a pattern. Here you can see that `pattern` is accessible in the mapper without any particular work on the programmer side and according to normal R scope rules. This apparent simplicity hides the fact that the map function is executed in a different interpreter and on a different machine than the `mapreduce` function. Behind the scenes the R environment is serialized, broadcast to the cluster and restored on each interpreter running on the nodes. For each word, a key value pair (w, 1) is generated with `keyval` and the list of all of them is the return value of the mapper.

The reduce function takes a key and a list of values as input and simply sums up all the counts and returns the pair word, count using the same helper function, `keyval`. Finally, specifying the use of a combiner is necessary to guarantee the scalability of this algorithm.

Logistic Regression

Now onto an example from supervised learning, specifically logistic regression by gradient descent. Again we are going to create a function that encapsulates this algorithm.

```
logistic.regression = function(input, iterations, dims, alpha){
  plane = rep(0, dims)
  g = function(z) 1/(1 + exp(-z))
  for (i in 1:iterations) {
    gradient = from.dfs(mapreduce(input,
      map = function(k, v) keyval (1, v$y * v$x * g(-v$y * (plane %*% v$x))),
      reduce = function(k, vv) keyval(k, apply(do.call(rbind,vv),2,sum)),
      combine = T))
    plane = plane + alpha * gradient[[1]]$val }
```

```
  plane }
```

As you can see we have an input with the training data. For simplicity we ask to specify a fixed number of iterations, but it would be marginally more difficult to implement a convergence criterion. Then we need to specify the dimension of the problem, which is a bit redundant because it can be inferred after seeing the first line of input, but we didn't want to put additional logic in the map function, and then we have the learning rate `alpha`.

We start by initializing the separating plane and defining the logistic function. As before, those variables will be used inside the map function, that is they will travel across interpreter and processor and network barriers to be available where the developer needs them and where a traditional, meaning sequential, R developer expects them to be available according to scope rules — 0 fuss and familiar, powerful behavior.

Then we have the main loop where computing the gradient of the loss function is the duty of a map reduce job, whose output is brought straight into main memory with an from.dfs — there are temp files being created and destroyed behind the scenes but you don't need to know. The only important thing you need to know is that the gradient is going to fit in memory so we can do a `from.dfs` to get it without exceeding our resources.

This map reduce job has the input we specified in the first place, the training data.

The map function simply computes the contribution of an individual point to the gradient. Please note the variables `g` and `plane` making their necessary appearance here without any work on the developer's part. The access here is read only but you could even modify them if you wanted — the semantics is copy on assign, which is consistent with how R works and easily supported by hadoop. Since in the next step we just want to add everything together, we return a dummy, constant key for each record.

The reduce function, besides the usual R fidgeting to get numbers out of lists, is just a big sum. As far as the fidgeting, we decided to support the more general lists first and then add some API convenience for the cases where a data frame would suffice, which is a special case but a very important one. The conversion is not as general as we would like it to be but you can try it (`reduce.on.data.frame = T`).

Since we have only one key, all the work will fall on one reducer and that's not scalable, so in this example it's important to activate the combiner, in this case it's TRUE so it's the same as the reducer. Since sums are associative and commutative that's all we need. We also support a distinct combiner function.

After the map reduce job is complete and `from.dfs` has copied the only record that is supposedly produced by this job into the gradient variable, we just have to upgrade the separating plane and return it after the iterations are complete.

To make this example production-level there are several things one needs to do, like having a convergence criterion instead of a fixed iteration number an an adaptive learning rate, but probably gradient descent just requires too many iterations to be the right approach in a big data context. But this example should give you all the elements to be able to implement, say, conjugate gradient instead. In general, when each iteration requires I/O of a large data set, the number of iterations needs to be modest and algorithms with O(log(N)) number of iterations are natural candidates, even if the work in each iteration may be more substantial.

##K-means

We are now going to cover a simple but significant clustering algorithm and the complexity will go up just a little bit. To cheer yourself up, you can take a look at this alternative implementation which requires three languages, python, pig and java, to get the job done and is hailed as a model of simplicity.

We are talking about k-means and we are going to implement it in two parts: a function that controls the iterations and termination of the algorithm and one, essentially a mapreduce job, that does the grunt work of computing distances and electing centers. This is not a production ready implementation, but should be illustrative of the power of this package. You simply can not do this in pig or hive alone and it would take hundreds of lines of code in java.

```
kmeans =
  function(points, ncenters, iterations = 10, distfun = NULL) {
    if(is.null(distfun))
      distfun =
        function(a,b) norm(as.matrix(a-b), type = 'F')
    newCenters =
      kmeans.iter(
        points,
        distfun,
        ncenters = ncenters)
    for(i in 1:iterations) {
      newCenters = kmeans.iter(points, distfun, centers = newCenters)}
    newCenters}
```

This is the controlling program. Most of its lines are executed locally, meaning in the same interpreter where the `kmeans` function is called. We define a function that takes a set of points, a

desired number of centers, a number of iterations (a convergence test would be better, just for illustration purposes) and a distance function defaulting to euclidean distance.

This function uses another function, `kmeans.iter` which implements one iteration of kmeans. It can be called in two flavors. In the first call, only the number of centers is specified. The function returns a new set of centers.

Now let's look at the main loop. It gets executed a user-set number of times, for simplicity's sake, but implementing a convergence criterion should be easy. Its body is just a call to the other flavor of `kmeans.iter`, where a set of centers rather than just the number of centers is provided. and the return value is, as before, a new set of centers. Let's now look at `kmeans.iter`, the real workhorse.

```
kmeans.iter =
  function(points, distfun, ncenters = dim(centers)[1], centers = NULL) {
    from.dfs(mapreduce(input = points,
                       map =
                         if (is.null(centers)) {
                           function(k,v) keyval(sample(1:ncenters,1),v)}
                         else {
                           function(k,v) {
                             distances = apply(centers, 1, function(c) distfun(c,v))
                             keyval(centers[which.min(distances),], v)}},
                       reduce = function(k,vv) keyval(NULL, apply(do.call(rbind, vv), 2, mean))),
             to.data.frame = T)}
```

This is one iteration of kmeans, implemented as a map reduce job. We are already familiar with all the parameters to this function and we know what `from.dfs` does, in this case it moves new cluster centers computed by a mapreduce job and stored in HDFS into main memory. Here we are assuming we have a lot of points, as in big data, but a small number of centers, as in they fit in RAM. It's a common case but it might not cover all applications.

Next is the call actually performing a map reduce job. Its input is a set of points, as an HDFS path or big data object. We have two cases for the map function, one for the initialization, that is when there isn't a set of current cluster centers, only a desired number, and one for all the other calls from the main loop. In the former we just assign a random center to each point and generate a key value pair with the center as key and the point as value. That means, in the reduce stage, we are guaranteed that all points with the same center will end up together in the same reduce call. Of course that could mean that some reducers have an unacceptable amount of work to perform, but a simple modification of this program and the use of a combiner fixes that problem, so we are going to ignore the problem in this tutorial. In the latter we compute all the distances from a point to each center and return a key value pair that has the closest center as key and the point itself as value

To perform a sample run, we need some data. We can create is very easily from the R prompt:

```
clustdata = lapply(1:10000, function(i) keyval(NULL, c(rnorm(1, mean = i%%3, sd = 0.01),
                                                       rnorm(1, mean = i%%4, sd = 0.01))))
to.dfs(clustdata, "/tmp/clustdata")
```

That is, create some arbitrary data in the form of a list of key value pairs, the key here doesn't really matter and write it out to hdfs. And this is a simple test of what we've just implemented,

```
kmeans ("/tmp/clustdata", 12)
```

With a little extra work you can even get pretty visualizations like this one (code in source under `tests`)

[[kmeans.gif]]

Linear Least Squares

We are going to build another example, LLS, that illustrates how to build map reduce reusable abstractions and how to combine them to solve a larger task. We want to solve LLS under the assumption that we have too many data points to fit in memory but not such a huge number of variables that we need to implement the whole process as map reduce job. This is sort of a hybrid solution that is made particularly easy by the seamless integration of `rmr` with R and an example of a pragmatic approach to big data. If we have operations A, B, and C in a cascade and the data sizes decrease at each step and we have already an in-memory solution to it, than we might get away by replacing only the first step with a big data solution and then continuing with tried and true function and pacakges. To make this as easy as possible, we need the in memory and big data worlds to integrate easily.

This is the basic equation we want to solve in the least square sense:

X β = y

We are going to do it by using the function solve as in the following expression, that is solving the normal equations.

```
solve(t(X)%*%X, t(X)%*%y)
```

But let's assume that X is too big to fit in memory, so we have to compute the transpose and

matrix products using map reduce, then we can do the solve as usual on the results. This is our general plan.

We are going to adopt the following representation for matrices, here in data frame form (behind the scenes it's still lists).

```
   key1 key2        val

1     1    1  0.7595035
2     1    2  1.1731512
3     1    3  0.2112339
4     2    1  0.2305024
5     2    2 -0.5277821
6     2    3 -2.4413680
7     3    1 -0.8510856
```

The key is the pair row, col and the value is the matrix element. In practice this representation makes sense only for sparse matrices, so in a real world implementation we might want to use a representation with a submatrix for each record, but this is simpler to develop the ideas.

We start implementing the transpose with a tiny auxiliary function that swaps the elements of a two element list. You guessed right that this is going to be used to swap the raw index with the column index.

```
swap = function(x) list(x[[2]], x[[1]])
```

Then we define the map function for the transpose map reduce job. It uses a higher order function, `to.map`, to turn two ordinary functions into a map. This is possible because they act independently on the key and on the value. What this says is: swap the elements of the key and let the value through. We could have written it just as easily without `to.map`, but once you are familiar with it it is more readable this way.

```
transposeMap = to.map(swap, identity)
```

Then we have the map reduce transpose job which is abstracted into a function transpose, that we can use like any other job from now on.

```
transpose = function(input, output = NULL){
    mapreduce(input = input, output = output, map = transposeMap)}
```

It takes an input, an optional output and returns the return value of the map reduce job. It passes input and output to it and the map function we've just defined and that's all.

Detour: Relational Joins

Now we would like to tackle matrix multiplication but we need a short detour first. This takes one step further in hadoop mastery as we need to combine and process two files into one map reduce job. By default mapreduce supports merging two inputs the way hadoop does, that is once can specify multiple inputs and the only guarantee is that every record will go through one mapper. No order or grouping of any sort is guaranteed as the mappers are processing the input files.

What we need here is a very orderly merging so that we can multiply matrix elements that share an index and then sum them together. It actually looks like a join on one specific index. It turns out that joins are a very important subtask in many mapreduce algorithms and are more or less supported in a number of hadoop dialects. A generalized equijoin is part of the rmr API:

```
equijoin = function(
  left.input = NULL,
  right.input = NULL,
  input = NULL,
  output = NULL,
  outer = c("", "left", "right", "full"),
  map.left = to.map(identity),
  map.right = to.map(identity),
  reduce  = function(k, values.left, values.right)
    do.call(c,
            lapply(values.left,
                   function(vl) lapply(values.right,
                                       function(vr) reduce.all(k, vl, vr)))),
  reduce.all  = function(k, vl, vr) keyval(k, list(left = vl, right = vr)))
```

Instead of a single input, we have a `left.input` and `right.input`, as joins normally do, but in case we want to perform a self join, we can skip the first two arguments and specify only the third, `input`.

Then we have an output, optional as usual, and we can specify different flavors of join such as in left outer, right outer or full outer as usual.

Now to the interesting bits. This function is a bit relational join and a bit map reduce job. Instead of specifying join keys, we specify two separate map functions, one for the left input and one for the right input. Map functions, as usual, produce a key and a value. The join will be an equijoin on the keys. For each pair of matching records there will be a shared key and two values, one coming from each side. By default, we have simple pass-throguh or identity mappers.

The reduce can be specified as usual, but an alternate interface is offered with `reduce.all` which is more SQL-like in that it is a join without a group by on the join key, whereas the reduce form implies a group by. This is a little advanced in a number of ways and also very reusable, so we made it part of the library even if it is built on top of `mapreduce`. No we are going to see its application to perform a matrix multiplication.

Linear Least Squares (continued)

Back to our matrix multiplication task, which we will implement as an application of the equijoin just shown.

A = **BC**

$a_{ij} = \Sigma_k b_{ik} c_{kj}$

We first define the map function. It comes in two flavors, wether you want to join on the column index or on the row index, and in a matrix multiplication we need both. So here is a higher order function that generates both maps. It just produces a key-value pair with as key the desired index and as value a list with all the information, row, column and element, which we will need later on.

```
matMulMap = function(i) function(k,v) keyval(k[[i]], list(pos = k, elem = v))
```

And finally to the actual matrix multiplication. It is implemented as the composition of two jobs. One does the multiplications and the other the sums. There are other ways of implementing it but this is the most straightforward.

```
mat.mult = function(left, right, result = NULL) {
  mapreduce(
                input =
                equijoin(left.input = left, right.input = right,
                                 map.left = mat.mult.map(2),
                                 map.right = mat.mult.map(1),
                                 reduce = function(k, vvl, vvr)
                                   do.call(c, lapply(vvl, function(vl)
                                     lapply(vvr, function(vr) keyval(c(vl$pos[[1]], vr$pos[[2]]), vl$elem*vr$elem))))),
                output = result,
                reduce = to.reduce(identity, function(x) sum(unlist(x))))}
```

The first step is a join on the the column index for the left side and the row index from the right side, so that we bring together pairs of the form (b_{ik},c_{kj}). In the reduce we perform the multiplication and return a record with a key of (i,j) and a value equal to the multiplication.

The following or outer mapreduce doesn't have an explicit map, that means it defaults to the pass-through one. The interesting thing is that, by default, the grouping will happen on the (i,j) pair, therefore grouping all the right products that need to be summed together.

We now have all the elements in place to solve LLS: mapreduce transpose and matrix multiplication and old fashioned solve().

We need a simple function to turn matrices represented in list form and sparse format, that we use with mapreduce, into regular dense in memory R matrices. We rely on a feature of `from.dfs` that turns lists into data frames whenever possible so that this function just has to go from dataframe to dense matrix, using the `Matrix` package and `sparseMatrix` in particular.

```
to.matrix = function(df) as.matrix(sparseMatrix(i=df$V1, j=df$V2, x=df$V3))
```

Then our sought after semi-big-data LLS solution

```
linear.least.squares = function(X,y) {
  Xt = transpose(X)
  XtX = from.dfs(mat.mult(Xt, X), to.data.frame = TRUE)
  Xty = from.dfs(mat.mult(Xt, y), to.data.frame = TRUE)
  solve(to.matrix(XtX),to.matrix(Xty))}
```

We start with a transpose, compute the normal equations, left and right side, and call solve on the converted data.

##What we have learned

We will summarize here a few ways in which we have used the functions in the library.

Specify mapreduce jobs using familiar R functions:

```
mapreduce(..., map = function(k,v)..., reduce = function(k,vv)...)
```

Compose jobs like functions

```
mapreduce(mapreduce(...
```

store results where you want

```
mapreduce(..., output = "my-result-file", ...)
```

or in a variable

```
my.result = mapreduce(...)
```

create abstractions

```
my.job = function(x,y,z) { .... out = mapreduce(...);  ... out}
```

describe any data flow

```
out1 = my.job1(my.result); out2 = my.job2(my.result); merge.job(out1, out2)
```

move things in an out of memory and HDFS to create hybrid big-small-data algorithms, control flow and iteration, display results etc

```
if(length(from.dfs(my.job(...)))==0){...} else {...}
ggplot2(from.dfs(my.job(...)), ...)
```

Related Links

- [[Comparison of high level languages for mapreduce: k means]]
- [[Fast k means]]
- [[Changelog]]
- [[Philosophy]]
- [[Efficient rmr techniques]]
- [[Writing composable mapreduce jobs]]
- [[Use cases]]
- [[Getting data in and out]]
- [[FAQ]]