Interface, Design, and Computational Considerations for Divide and Recombine

Ryan Hafen @hafenstats
Hafen Consulting, Purdue University

Interface Symposium
June 12, 2015
GOAL: **DEEP ANALYSIS OF LARGE COMPLEX DATA**

» Data most often does not come with a model

» If we already (think we) know the algorithm / model to apply and simply apply it to the data and we’re done, we are not doing *analysis*, we are *processing*

» Deep analysis means detailed, comprehensive analysis that does not lose important information in the data

» It means trial and error; an iterative process of hypothesizing, fitting, validating, learning

» It means a lot of visualization

» It means access to 1000s of statistical, machine learning, and visualization methods, regardless of data size

» It means access to a high-level language to make iteration efficient in analyst time
Example: Power Grid

- 2 TB data set of high-frequency power grid measurements at several locations on the grid
- Identified, validated, and built precise statistical algorithms to filter out several types of bad data that had gone unnoticed in several prior analyses (~20% bad data!)
Often the goal is pure classification / prediction (tech, kaggle, etc.)

Even in these cases, it is good to explore, visualize

How do you know you are not choosing from a bunch of poor performers?

What if the data quality is terrible and you don't know it?

What about interpretability?
“If [you have no] concern about error bars, about heterogeneity, about noisy data, about the sampling pattern, about all the kinds of things that you have to be serious about if you’re a statistician – then … there’s a good chance that you will occasionally solve some real interesting problems. But you will occasionally have some disastrously bad decisions. And you won’t know the difference a priori. You will just produce these outputs and hope for the best.”

– Michael Jordan
GOAL: DEEP ANALYSIS OF LARGE COMPLEX DATA

» Large complex data has any or all of the following:

– Large number of records

– Many variables

– Complex data structures not readily put into tabular form of cases by variables

– Intricate patterns and dependencies that require complex models and methods of analysis

– Not i.i.d.!
What we want to be able to do (with large complex data)

» Work completely in R

» Have access to R's 1000s of statistical, ML, and vis methods ideally with no need to rewrite scalable versions

» Be able to apply any ad-hoc R code to any type of distributed data object

» Minimize time thinking about code or distributed systems

» Maximize time thinking about the data

» Be able to analyze large complex data with nearly as much flexibility and ease as small data
Divide and Recombine (D&R)

» Simple idea:
  – specify a meaningful division of the data
  – apply an analytic or visual method independently to each subset of the divided data in embarrassingly parallel fashion
  – recombine the results to yield a statistically valid D&R result for the analytic method

» D&R is not the same as MapReduce (but makes heavy use of it)
How to Divide the Data?

» Typically “big data” is big because it is made up of collections of smaller data from many subjects, sensors, locations, time periods, etc.

» It is therefore natural to break the data up based on these dimensions and apply visual or analytical methods to the subsets individually

» We call this “conditioning variable” division

» It is in practice by far the most common thing we do (and it’s nothing new)

» Another option is “random replicate” division
Analytic Recombination

» Analytic recombination begins with applying an analytic method independently to each subset
  
  – The beauty of this is that we can use any of the small-data methods we have available (think of the 1000s of methods in R)

» For conditioning-variable division:
  
  – Typically the recombination depends on the subject matter
  
  – Example: apply the same model to each subset and combine the subset estimated coefficients and build a statistical model or visually study the resulting collection of coefficients
Analytic Recombination

For random replicate division:

- Observations are seen as exchangeable, with no conditioning variables considered
- Division methods are based on statistical matters, not the subject matter as in conditioning-variable division
- Results are often approximations

Approaches that fit this paradigm

- Coefficient averaging
- Subset likelihood modeling
- Bag of little bootstraps
- Consensus MCMC
- Alternating direction method of multipliers (ADMM)
Visual Recombination

» Data split into meaningful subsets, usually conditioning on variables of the dataset

» For each subset:
  – A visualization method is applied
  – A set of cognostics, metrics that identify attributes of interest in the subset, are computed

» Recombine visually by sampling, sorting, or filtering subsets based on the cognostics

» Implemented in the trelliscope package
Data structures for D&R

» Must be able to break data down into pieces for independent storage / computation

» Recall the potential for: “Complex data structures not readily put into tabular form of cases by variables”

» **Key-value pairs**: a flexible storage paradigm for divided data
  
  – each subset is an R list with two elements: key, value
  
  – keys and values can be any R object
### Setosa

<table>
<thead>
<tr>
<th></th>
<th>Sepal.Length</th>
<th>Sepal.Width</th>
<th>Petal.Length</th>
<th>Petal.Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.1</td>
<td>3.5</td>
<td>1.4</td>
<td>0.2</td>
</tr>
<tr>
<td>2</td>
<td>4.9</td>
<td>3.0</td>
<td>1.4</td>
<td>0.2</td>
</tr>
<tr>
<td>3</td>
<td>4.7</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
</tr>
<tr>
<td>4</td>
<td>4.6</td>
<td>3.1</td>
<td>1.5</td>
<td>0.2</td>
</tr>
<tr>
<td>5</td>
<td>5.0</td>
<td>3.6</td>
<td>1.4</td>
<td>0.2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Versicolor

<table>
<thead>
<tr>
<th></th>
<th>Sepal.Length</th>
<th>Sepal.Width</th>
<th>Petal.Length</th>
<th>Petal.Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>7.0</td>
<td>3.2</td>
<td>4.7</td>
<td>1.4</td>
</tr>
<tr>
<td>52</td>
<td>6.4</td>
<td>3.2</td>
<td>4.5</td>
<td>1.5</td>
</tr>
<tr>
<td>53</td>
<td>6.9</td>
<td>3.1</td>
<td>4.9</td>
<td>1.5</td>
</tr>
<tr>
<td>54</td>
<td>5.5</td>
<td>2.3</td>
<td>4.0</td>
<td>1.3</td>
</tr>
<tr>
<td>55</td>
<td>6.5</td>
<td>2.8</td>
<td>4.6</td>
<td>1.5</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distributed data objects (ddo)

- A collection of k/v pairs that constitutes a set of data
- Arbitrary data structure (but same structure across subsets)

> irisDdo

Distributed data object backed by 'kvMemory' connection

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size (stored)</td>
<td>12.67 KB</td>
</tr>
<tr>
<td>size (object)</td>
<td>12.67 KB</td>
</tr>
<tr>
<td># subsets</td>
<td>3</td>
</tr>
</tbody>
</table>

- Other attributes: getKeys()
- Missing attributes: splitSizeDistn
Distributed data frames (ddf)

» A distributed data object where the value of each key-value pair is a data frame

» Now we have more meaningful attributes (names, number of rows & columns, summary statistics, etc.)

> irisDdf

Distributed data frame backed by 'kvMemory' connection

| attribute      | value                                                                 |
|----------------+----------------------------------------------------------------------------|
| names          | Sepal.Length(num), Sepal.Width(num), and 3 more                              |
| nrow           | 150                                                                        |
| size (stored)  | 12.67 KB                                                                   |
| size (object)  | 12.67 KB                                                                   |
| # subsets      | 3                                                                           |

* Other attrs: getKeys(), splitSizeDistn(), splitRowDistn(), summary()
D&R computation

» MapReduce is sufficient for all D&R operations
  – Everything uses MapReduce under the hood
  – Division, recombination, summaries, etc.
TESSERA
Software for Divide and Recombine

» D&R Interface
  – **datadr** R package: R implementation of D&R that ties to scalable back ends
  – **Trelliscope** R package: scalable detailed visualization

» Back-end agnostic design

```
<table>
<thead>
<tr>
<th>Interface</th>
<th>datadr / trelliscope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>MapReduce</td>
</tr>
<tr>
<td>Storage</td>
<td>Key/Value Store</td>
</tr>
</tbody>
</table>
```

http://tessera.io
Supported back ends (currently)

- datadr / trelliscope
- R
- Multicore R
- RHIPE / Hadoop
- Memory
- Local Disk
- HDFS

Small
Medium
Large

And more… (like Spark)
What does a candidate back end need?

» MapReduce that can run R in the map and reduce
» Distributed key-value store
» Fast random access by key
» Ability to broadcast auxiliary data to nodes
» A control mechanism to handle backend-specific settings (Hadoop parameters, etc.)

» To plug in a back end, implement methods that tie to generic MapReduce and data connection classes
datadr

» Distributed data types / backend connections
  – `localDiskConn()`, `hdfsConn()`, `sparkDataConn()`
    connections to ddo / ddf objects persisted on a backend storage system
  – `ddo()`: instantiate a ddo from a backend connection
  – `ddf()`: instantiate a ddf from a backend connection

» Conversion methods between data stored on different backends
datadr: division-independent methods

» drQuantile(): estimate all-data quantiles, optionally by a grouping variable
» drAggregate(): all-data tabulation
» drHexbin(): all-data hexagonal binning aggregation

» summary() method computes numerically stable moments, other summary stats (freq table, range, #NA, etc.)
datadr: division and recombination

» A `divide` function takes a ddf and splits it by columns in the data or randomly
» Division of ddos with arbitrary data structures must typically be done with custom MapReduce code (unless data can be temporarily transformed into a ddf)
» Analytic methods are applied to a ddo/ddf with the `addTransform` function
» Recombinations are specified with `recombine`, which provides some standard combiner methods, such as `combrbind`, which binds transformed results into single data frame
datadr: data operations

» **drLapply()**: apply a function to each subset of a ddo/ddf and obtain a new ddo/ddf

» **drJoin()**: join multiple ddo/ddf objects by key

» **drSample()**: take a random sample of subsets of a ddo/ddf

» **drFilter()**: filter out subsets of a ddo/ddf that do not meet a specified criteria

» **drSubset()**: return a subset data frame of a ddf

» **drRead.table()** and friends

» **mrExec()**: run a traditional MapReduce job on a ddo/ddf
maxMap <- expression({
  for(curMapVal in map.values)
    collect("max", max(curMapVal$Petal.Length))
})

maxReduce <- expression(
  pre = {
    globalMax <- NULL
  },
  reduce = {
    globalMax <- max(c(globalMax, unlist(reduce.values)))
  },
  post = {
    collect(reduce.key, globalMax)
  }
)

maxRes <- mrExec(hdfsConn("path_to_data"),
  map = maxMap,
  reduce = maxReduce
  control =
)
maxMap <- expression({
  for(curMapVal in map.values)
    collect("max", max(curMapVal$Petal.Length))
})

maxReduce <- expression(
  pre = {
    globalMax <- NULL
  },
  reduce = {
    globalMax <- max(c(globalMax, unlist(reduce.values)))
  },
  post = {
    collect(reduce.key, globalMax)
  }
)

maxRes <- mrExec(sparkDataConn("path_to_data"),
  map = maxMap,
  reduce = maxReduce
  control =
)
maxMap <- expression({
  for(curMapVal in map.values)
    collect("max", max(curMapVal$Petal.Length))
})

maxReduce <- expression(
  pre = {
    globalMax <- NULL
  },
  reduce = {
    globalMax <- max(c(globalMax, unlist(reduce.values)))
  },
  post = {
    collect(reduce.key, globalMax)
  }
)

maxRes <- mrExec(localDiskConn("path_to_data"),
  map = maxMap,
  reduce = maxReduce
  control =
)
maxMap <- expression({
  for(curMapVal in map.values)
    collect("max", max(curMapVal$Petal.Length))
})

maxReduce <- expression(
  pre = {
    globalMax <- NULL
  },
  reduce = {
    globalMax <- max(c(globalMax, unlist(reduce.values)))
  },
  post = {
    collect(reduce.key, globalMax)
  }
)

maxRes <- mrExec(data,
  map = maxMap,
  reduce = maxReduce
  control =
)
D&R Example

» Zillow home price data:

```r
> head(housing)

   fips  county state      time nSold medListPriceSqft medSoldPriceSqft
1 06001 Alameda County  CA 2008-10-01   NA        307.9787       325.8118
2 06001 Alameda County  CA 2008-11-01   NA        299.1667           NA
3 06001 Alameda County  CA 2008-11-01   NA           NA        318.1150
4 06001 Alameda County  CA 2008-12-01   NA        289.8815       305.7878
5 06001 Alameda County  CA 2009-01-01   NA        288.5000       291.5977
6 06001 Alameda County  CA 2009-02-01   NA        287.0370           NA
```
D&R Example

» Divide by county and state

```r
> byCounty <- divide(housing,
>     by = c("county", "state"), update = TRUE)
> byCounty

Distributed data frame backed by 'kvMemory' connection

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>names</td>
<td>fips(cha), time(Dat), nSold(num), and 2 more</td>
</tr>
<tr>
<td>nrow</td>
<td>224369</td>
</tr>
<tr>
<td>size (stored)</td>
<td>16.45 MB</td>
</tr>
<tr>
<td>size (object)</td>
<td>16.45 MB</td>
</tr>
<tr>
<td># subsets</td>
<td>2883</td>
</tr>
</tbody>
</table>

* Other attributes: getKeys(), splitSizeDistn(), splitRowDistn(), summary()
* Conditioning variables: county, state
```
D&R Example

» Look at a subset

```r
> byCounty[[1]]

$key
[1] "county=Abbeville County|state=SC"

$value
     fips   time   nSold medListPriceSqft medSoldPriceSqft
1  45001 2008-10-01    NA       73.06226            NA
2  45001 2008-11-01    NA       70.71429            NA
3  45001 2008-12-01    NA       70.71429            NA
4  45001 2009-01-01    NA       73.43750            NA
5  45001 2009-02-01    NA       78.69565            NA
...
```
D&R Example

» Look at a subset by key

```r
> byCounty[["county=Monongalia County|state= WV"]]

$key
[1] "county=Monongalia County|state= WV"

$value
fips     time   nSold medListPriceSqft medSoldPriceSqft
1 54061 2008-10-01   NA       120.4167          NA
2 54061 2008-11-01   NA       121.7949          NA
3 54061 2008-11-01   NA          NA          NA
4 54061 2008-12-01   NA       121.3571          NA
5 54061 2009-01-01   NA       121.3571          NA
...
Apply a transformation to get slope of fitted line of list price vs. time

```r
> lmCoef <- function(x)
> coef(lm(medListPriceSqft ~ time, data = x))[2]
>
> byCountySlope <- addTransform(byCounty, lmCoef)
>
> byCountySlope[[1]]

$key
[1] "county=Abbeville County|state=SC"

$value
    time
-0.0002323686
```
Recombine the slope coefficients into a data frame

```r
> countySlopes <- recombine(byCountySlope, combRbind)
> head(countySlopes)

   county state    val
time  Abbeville County  SC -0.0002323686
time1 Acadia Parish  LA  0.0019518441
time2 Accomack County  VA -0.0092717711
time3    Ada County  ID  -0.0030197554
time4    Adair County  IA  -0.0308381951
time5    Adair County  KY   0.0034399585
```
A Note About Lazy Evaluation

» Systems like Spark provide lazy evaluation
  – Specify a series of computation steps but don’t execute until a result is asked for
  – The idea is that the resulting computation graph can be optimized

» In D&R, we (mostly) don’t do this
  – Any *divide, recombine*, or function beginning with *dr* immediately kicks off a MapReduce job
  – This is a deliberate choice made for good reason
Why Not Lazy Evaluation in D&R?

» Divisions can typically be accomplished with one MapReduce job and they are to be persistent, so why not compute right away?

» Applying an analytic method and recombining is also one MapReduce job, and we want the result right away in this case as well

» So really, we don’t need to do lazy evaluation

» Ok, there are a few cases where we string data operations together, e.g. divide followed by drFilter, etc.
  – You could argue we should have lazy evaluation here
  – Why not? Debugging!
Debugging in Distributed Computing

» Distributed debugging is very difficult
  – Which subset did the error come from?
  – What was the environment like in the R instance running on the node where the error occurred?
  – etc.

» One of the most common causes of bugs is specifying operations on data that we have not yet seen and therefore do not know exactly what its structure is (and we get it wrong)

» This is a major reason we don’t lazy evaluate sequences of commands
The One Lazy Evaluation Exception

» Applying transformations to ddo/ddf objects with `addTransform` is a lazily evaluated operation
  – The transformation is made note of and applied when the transformed object is computed on
  – We can do this and still keep things simple
  – A transformed object behaves in every way as if it has already been transformed
Lazy Evaluation of `addTransform`

```r
> lmCoef <- function(x)
>   coef(lm(medListPriceSqft ~ time, data = x))[2]
>
> byCountySlope <- addTransform(byCounty, lmCoef)

Transformed distributed data object backed by 'kvMemory' connection

<table>
<thead>
<tr>
<th>attribute</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size (stored)</td>
<td>16.45 MB (before transformation)</td>
</tr>
<tr>
<td>size (object)</td>
<td>16.45 MB (before transformation)</td>
</tr>
<tr>
<td># subsets</td>
<td>2883</td>
</tr>
</tbody>
</table>

* Other attributes: `getKeys()`
* Conditioning variables: county, state

> byCountySlope[[1]]

$key
[1] "county=Abbeville County|state=SC"

$value
  time
-0.0002323686
Recap of Some Key Points

- D&R is a simple but powerful and scalable paradigm.
- Think of D&R as turning a big data problem into many small data problems, which we can attack with the full arsenal of R.
- MapReduce is sufficient for D&R, but not the same thing.
- We strive to use methods that do not require iterative application of MapReduce.
- Key/Value pairs for storage – provide the flexibility we need to deal with large complex data.
- Divisions are persistent (and expensive to compute) and should be well thought out.
- A single data set can (and usually does) have multiple divisions.
- Typically there are many recombinations applied to a given division – recombinations are much faster to compute.
Learning More About Tessera

» [tessera.io](https://tessera.io)
  – Scripts to get an environment set up
    • Workstation
    • Vagrant
    • AWS Elastic MapReduce
  – Links to tutorials, papers
  – Blog

» [github.com/tesseradata](https://github.com/tesseradata)

» [@TesseralO](https://twitter.com/TesseralO)
How to Help & Contribute

» Open source BSD / Apache license
» Google user group
» Start using it!
  – If you have some applications in mind, give it a try!
  – You don’t need big data or a cluster to use Tessera
  – Ask us for help, let us help you showcase your work
  – Give us feedback
» See resources page in tessera.io
» Theoretical / methodological research
  – There’s plenty of fertile ground
Acknowledgements

» U.S. Department of Defense Advanced Research Projects Agency, XDATA program

» U.S. Department of Homeland Security, Science and Technology Directorate

» Division of Math Sciences CDS&E Grant, National Science Foundation

» PNNL, operated by Battelle for the U.S. Department of Energy, LDRD Program, Signature Discovery and Future Power Grid Initiatives