Histogramming in map-reduce

Jim Pivarski

Princeton University – DIANA

April 25, 2016

Motivation

I'm working with an analysis group (Oliver Gutsche, Matteo Cremonesi, and Cristina Suárez) to do a CMS dark matter search using Apache Spark.

All steps are to be distributed across the Spark cluster:

(1) skim and pull out relevant features;
(2) exploratory data analysis (EDA);
(3) final plots.

Motivation

I'm working with an analysis group (Oliver Gutsche, Matteo Cremonesi, and Cristina Suárez) to do a CMS dark matter search using Apache Spark.

All steps are to be distributed across the Spark cluster:

(1) skim and pull out relevant features;

(2) exploratory data analysis (EDA);

(3) final plots.

Someday, it may be possible for analysis groups to do all work on a central server, rather than downloaded skims.

Motivation

I'm working with an analysis group (Oliver Gutsche, Matteo Cremonesi, and Cristina Suárez) to do a CMS dark matter search using Apache Spark.

All steps are to be distributed across the Spark cluster:

(1) skim and pull out relevant features;
(2) exploratory data analysis (EDA);
(3) final plots.

Someday, it may be possible for analysis groups to do all work on a central server, rather than downloaded skims.

In some industries, this is already common, especially with SQL. (Replace the word "skim" with "table.")

Step (1) is underway

You may have seen my previous talks on accessing ROOT in Scala (Spark's native language).

Three different solutions are now in place for testing:

- bulk convert ROOT $\rightarrow$ Avro $\rightarrow$ any industry format;
- read ROOT directly into Scala in a single process with JNI;
- stream ROOT data through a UNIX pipe into Scala.

They each have different strengths, and we'll pick a favorite in the course of the analysis.

Steps (2) and (3): making plots

As I said, we want to avoid downloading the whole dataset to a laptop for a traditional ntuple-analysis.

Spark has a functional for reducing data in a distributed way:

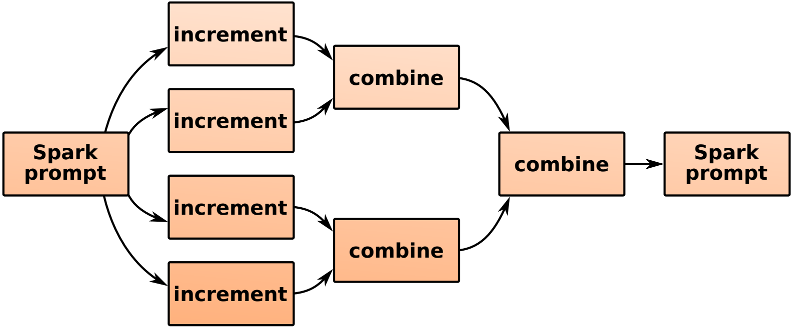
```
RDD.aggregate(initialize)(increment, combine)
```

where

- `RDD` is a collection of data of type $\mathcal{D}$ (end of skimming chain)
- `initialize` creates a counter of type $\mathcal{C}$
- `increment` is a function from $(\mathcal{C}, \mathcal{D}) \rightarrow \mathcal{C}$
- `combine` is a function from $(\mathcal{C}, \mathcal{C}) \rightarrow \mathcal{C}$

Aggregate functional

```
RDD.aggregate(initialize)(increment, combine)
```



(Hadoop equivalent: reduce; SQL equivalent: "`GROUP BY`")

Histograms fit naturally into aggregate

```
// hypothetical import ROOT
import org.dianahep.scaroot.classes.TH1F

val finalHist = RDD.aggregate(
    // "booking"
    new TH1F("pt", "pt", 100, 0, 20))(
    // "filling"
    {(h, d) => h.Fill(sqrt(d.px**2 + d.py**2)); h},
    // "merging"
    {(h1, h2) => h1.Add(h2); h1})
```

Histograms fit naturally into aggregate

```
// hypothetical import ROOT
import org.dianahep.scaroot.classes.TH1F

val finalHist = RDD.aggregate(
    // "booking"
    new TH1F("pt", "pt", 100, 0, 20))(
    // "filling"
    {(h, d) => h.Fill(sqrt(d.px**2 + d.py**2)); h},
    // "merging"
    {(h1, h2) => h1.Add(h2); h1})
```

Not bad, but what if you want to fill more than one histogram?

- These are *functions:* all histograms must be passed in as arguments and collected as return values (`h` and `h1` above).
- No global state because Spark is distributed.
- Data analyst has to maintain histogram code in three places.

First idea:

Move the logic of histogram-filling into the booking stage.

```
val h = Histogram("pt", 100, 0, 20,
                  {d => sqrt(d.px**2 + d.py**2)})
```

“fill rule” $f : \mathcal{D} \to \mathbb{R}$

This functional design allows the filling and merging to be automatic: no user input required.

```
RDD.aggregate(h)(auto_increment(), auto_combine())
```

Second idea:

Collect histograms into a container that also has automated filling and merging.

```
val pack_o_histograms = Label(
      "pt" -> Histogram(100, 0, 20, fill_pt),
      "Emiss" -> Histogram(100, 0, 50, fill_Emiss),
      ...)

RDD.aggregate(pack_o_histograms)(auto_increment(),
                                 auto_combine())
```

(`Label` and `Histogram` share a superclass; `auto_increment()` and `auto_combine()` call them the same way.)

Third idea:

Let all of these pieces be composable.

```
val directories =
      Label("dir1" ->
                Label("pt" -> Histogram(...),
                      "Emiss" -> Histogram(...)),
            "dir2" ->
                Label("pass" -> Count(...),
                      "maxpt" -> Maximize(...)))
```

(Combining directories of histograms is similar to ROOT's `hadd`.)

Notice that histograms themselves can be decomposed into smaller pieces:

```
val histogram = Histogram(100, 0, 20, fill_rule)
val histogram = Bin(100, 0, 20, fill_rule, Count())
```

where

- `Count` is an aggregator that counts events;
- `Bin` is an aggregator that makes 100 sub-aggregators and uses `fill_rule` to decide which one to pass the data on to, just as `Label` passes the data on to all of its contents.

Notice that histograms themselves can be decomposed into smaller pieces:

```
val histogram = Histogram(100, 0, 20, fill_rule)
val histogram = Bin(100, 0, 20, fill_rule, Count())
```

where

- `Count` is an aggregator that counts events;
- `Bin` is an aggregator that makes 100 sub-aggregators and uses `fill_rule` to decide which one to pass the data on to, just as `Label` passes the data on to all of its contents.

We get two-dimensional histograms for free:

```
val hist2d = Bin(binsX, lowX, highX, fillX,
                    Bin(binsY, lowY, highY, fillY,
                        Count()))
```

With the right sub-aggregators, we can get profile plots:

```
val profile = Bin(binsX, lowX, highX, fillX,
                                 Deviate(fillY))
  // "Deviate" accumulates mean & std deviation
```

Box-and-whisker plots:

```
val box_whiskers = Bin(binsX, lowX, highX, fillX,
                         Branch(Quantile(fillY),
                                Minimize(fillY),
                                Maximize(fillY)))
  // "Quantile" accumulates median & quartiles
  // "Branch" makes a tree of subaggregators
```

Heatmaps (average per bin, not a two-dimensional histogram):

```
val heatmap = Bin(binsX, lowX, highX, fillX,
                    Bin(binsY, lowY, fillY,
                         Average()))
  // "Average" accumulates a mean only
```

Mix and match with alternate binning schemes:

Fill a hashmap instead of an array:

```
val unknown_support =
      SparselyBin(binWidth, fillX, Count())
  // "SparselyBin" creates subaggregators as needed
```

Non-uniform bins:

```
val partitions_like_clustering =
      CentrallyBin(binCenters, fillX, Count())
val completely_arbitrary_bins =
      IrregularlyBin(binRanges, fillX, Count())
```

Use a clustering algorithm to find bin centers:

```
val first_look = AdaptivelyBin(fillX, Count())

val violin_plot = Bin(binsX, lowX, highX, fillX,
                      AdaptivelyBin(fillY, Count())
```

Similarly for super-histogram structures:

```
val efficiency = Fraction(cut, Histogram(...))
```

where `cut` is a function from $\mathcal{D} \to$ bool; two identical histograms are booked, one (denominator) is filled with all events, the other (numerator) only if it passes the cut.

```
val stack = Stack(q, cuts, Histogram(...))
```

where `q` is a function from $\mathcal{D} \to \mathbb{R}$ and `cuts` are successively tighter thresholds; $N_{\text{cuts}} + 1$ histograms are created.

```
val partition = Partition(q, cuts, Histogram(...))
```

Histograms now represent data *between* cuts (think of centrality bins in heavy ion plots).

Categorical features, too:

Fill rule maps from $\mathcal{D} \to$ string:

```
val bar_chart = Categorize(fillType, Count())
```

Order of categories on the axis can be imposed *after* aggregation.
The data are accumulated in a hashmap.

```
val backgrounds =
     Categorize({d => d.eventType},
       Histogram(120, 0, 120, {d => d.dimuonMass}))
```

Stacking order can also be imposed *after* aggregation.

A whole analysis can be a tree of nested histogram primitives with lambda functions at each level.

```
Label
 ├── "trigger thresholds"
 │    └── Stack → Fraction → Bin → Count
 ├── "cut scan"
 │    └── Stack → Label
 │         ├── "a vs b" → Bin → Deviate
 │         └── "b vs c" → Bin → Deviate
 └── "pT windows"
      └── Partition → Label
           ├── "a" → Bin → Count
           └── "b" → Bin → Count
```

Can answer questions like, "which cuts were applied in this plot?" by walking the tree, rather than scanning a `for` loop for `break` statements by eye.

Implementation

I started implementing this grammar to see if it makes sense.

http://github.com/diana-hep/histogrammar/

(with an “a,” get it?)

Primitive	Scala	Python	Primitive	Scala	Python
`Count`	done	done	`CentrallyBin`	done	
`Sum`	done	done	`AdaptivelyBin`	done	
`Average`	done	done	`IrregularlyBin`		
`Deviate`	done	done	`Fraction`	done	
`AbsoluteErr`	done	done	`Stack`	done	
`Minimize`	done	done	`Partition`	done	
`Maximize`	done	done	`Categorize`	done	
`Quantile`	done		`Label`	done	
`Bag`	done	done	`UntypedLabel`	done	
`Bin`	done	done	`Index`	done	
`SparselyBin`	done	done	`Branch`	done	

Shared JSON representation so primitives can be freely exchanged.

Other languages: C++, SQL, R, Javascript (for d3), CUDA?, ...

Example Spark session

```
import org.dianahep.histogrammar._
import org.dianahep.histogrammar.histogram._

// declare histograms
val px_histogram = Histogram(100, -5, 5,
  {mu: Muon => mu.px})
val pt_histogram = Histogram(80, 0, 8,
  {mu: Muon => sqrt(mu.px**2 + mu.py**2)})
val cut_histogram = Histogram(100, -5, 5,
  {mu: Muon => mu.px}, {mu: Muon => mu.py < 0})

// wrap them up in a collection
val all_histograms = Label("px" -> px_histogram,
  "pt" -> pt_histogram, "cut" -> cut_histogram)

// fill them in Spark
val final_result = rdd.aggregate(all_histograms)
      (new Increment, new Combine)
```

Example Spark session

```
all_histograms("pt").entries        // 0
final_result("pt").entries          // 100000
```

Example Spark session

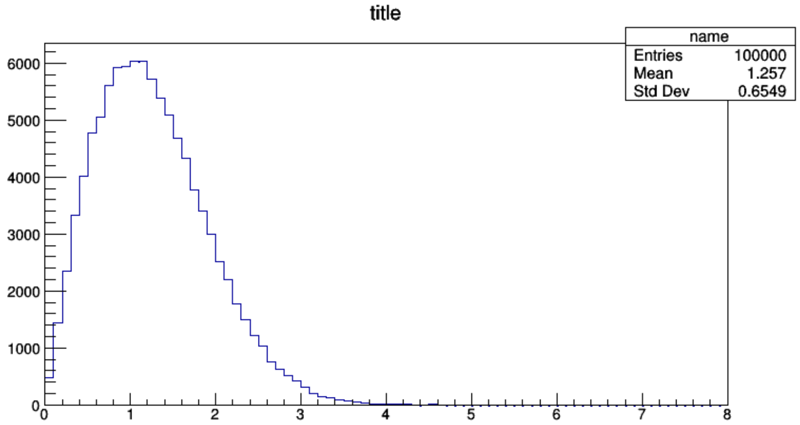
```
all_histograms("pt").entries        // 0
final_result("pt").entries          // 100000
```

```
println(final_result("pt").ascii)
```

```
                        0                                                   6616.50
                        +---------------------------------------------------------+
underflow          0    |                                                         |
[  0    ,  0.100)  506  |****                                                     |
[  0.100,  0.200)  1420 |************                                             |
[  0.200,  0.300)  2424 |*********************                                    |
[  0.300,  0.400)  3356 |*****************************                            |
[  0.400,  0.5  )  4258 |*************************************                    |
[  0.5  ,  0.600)  4688 |****************************************                 |
[  0.600,  0.700)  5262 |*********************************************            |
[  0.700,  0.800)  5805 |**************************************************       |
[  0.800,  0.900)  5855 |**************************************************       |
[  0.900,  1    )  6015 |****************************************************     |
[  1    ,  1.10 )  5977 |***************************************************      |
[  1.10 ,  1.20 )  5940 |***************************************************      |
[  1.20 ,  1.30 )  5763 |**************************************************       |
[  1.30 ,  1.40 )  5463 |***********************************************          |
[  1.40 ,  1.5  )  5009 |*******************************************              |
[  1.5  ,  1.60 )  4676 |****************************************                 |
[  1.60 ,  1.70 )  4226 |************************************                     |
[  1.70 ,  1.80 )  3743 |********************************                         |
[  1.80 ,  1.90 )  3226 |****************************                             |
[  1.90 ,  2    )  2911 |*************************                                |
[  2    ,  2.10 )  2449 |*********************                                    |
...
```

In Python...

```
th1f = final_result("pt").TH1F("name", "title")
th1f.Draw()
# because "import ROOT" didn't raise an ImportError
```



Histogrammar does not produce graphics

(The ASCII art histogram is a placeholder/debugging/fun.)

Although any combination of the primitives can be aggregated and used in an analysis, special combinations like `Bin(Count)` are recognized as plottable.

Histogrammar should link to external packages, such as Matplotlib and ROOT, to do the actual plotting.

- Minimal codebase to reimplement in a variety of languages;
- More of a clearinghouse than a software product, connecting systems that iterate over data to systems that plot data.

Count	Count data, ignoring their content. (Actually a sum of weights.)
Sum	Accumulate the sum of a given quantity.
Average	Accumulate the weighted mean of a given quantity.
Deviate	Accumulate a weighted variance, mean, and total weight of a given quantity (using an algorithm that is stable for large numbers).
AbsoluteErr	Accumulate the weighted Mean Absolute Error (MAE) of a quantity whose nominal value is zero.
Minimize	Find the minimum value of a given quantity. If no data are observed, the result is NaN.
Maximize	Find the maximum value of a given quantity. If no data are observed, the result is NaN.
Quantile	Accumulate an adaptively binned histogram to compute approximate quantiles, such as the median.
Bag	Accumulate raw data up to an optional limit, at which point only the total number is preserved.
Bin	Split a given quantity into equally spaced bins between specified limits and fill only one bin per datum.
SparselyBin	Split a quantity into equally spaced bins, filling only one bin per datum and creating new bins as necessary.
CentrallyBin	Split a quantity into bins defined by a set of bin centers, filling only one datum per bin with no overflows or underflows.
AdaptivelyBin	Split a quanity into bins dynamically with a clustering algorithm, filling only one datum per bin with no overflows or underflows.
Fraction	Accumulate two containers, one with all data (denominator), and one with data that pass a given selection (numerator).
Stack	Accumulate a suite containers, filling all that are above a given cut on a given expression.
Partition	Accumulate a suite containers, filling the one that is between a pair of given cuts on a given expression.
Categorize	Split a given quantity by its categorical (string-based) value and fill only one category per datum.
Label	Accumulate any number of containers of the SAME type and label them with strings. Every one is filled with every input datum.
UntypedLabel	Accumulate containers of any type except Count and label them with strings. Every one is filled with every input datum.
Index	Accumulate any number of containers of the SAME type anonymously in a list. Every one is filled with every input datum.
Branch	Accumulate containers of DIFFERENT types, indexed by i0 through i9. Every one is filled with every input datum.