HistogramTools for Distributions of Large Data Sets

Murray Stokely

Version 0.1 as of September 13, 2013

Abstract

Histograms are a common graphical representation of the distribution of a data set. They are particularly useful for collecting very large data sets into a binned form for easier data storage and analysis. The **HistogramTools** R package augments the built-in support for histograms with a number of methods that are useful for analyzing large data sets. Specifically, methods are included for serializing histograms into a compact Protocol Buffer representation for sharing between distributed tasks, functions for manipulating the resulting aggregate histograms, and functions for measuring and visualizing the information loss associated with histogram representations of a data set.

1 Introduction

Histograms have been used for over a century (**?**) as a compact graphical representation of a distribution of data. Support for generating histograms is included in almost all statistical software, including R. However, the growth in large-scale data analysis in R with the MapReduce (**?**) paradigm has highlighted the need for some extensions to the histogram functionality in R(**?**) for the collection, transmission, and manipulation of larged binned data sets.

Much previous work on histograms (**??**) involves identifying ideal breakpoints for visualizing a data set. Our context is different; we use histograms as compact representations in a large MapReduce environment, and need to merge histograms from subsets of the data to obtain a histogram for the whole data set. In this case, the challenges are engineering challenges, rather than visualization challenges. For example, how do we efficiently store large numbers of histograms generated frequently by real-time monitoring systems of many thousands of computers? How can we aggregate histograms generated by different tasks in a large scale computation? If we chose very granular bucket boundaries for our initial data collection pass, how can we then reshape the bucket boundaries to be more relevant to the analysis questions at hand?

2 Parallel Data Collection Patterns

Many large data sets in fields such as particle physics and information processing are stored in binned or histogram form in order to reduce the data storage requirements (**?**).

There are two common patterns for generating histograms of large data sets with MapReduce. In the first method, each mapper task can generate a histogram over a subset of the data that is has been assigned, and then the histograms of each mapper are sent to one or more reducer tasks to merge.

In the second method, each mapper rounds a data point to a bucket width and outputs that bucket as a key and '1' as a value. Reducers then sum up all of the values with the same key and output to a data store.

In both methods, the mapper tasks must choose identical bucket boundaries even though they are analyzing disjoint parts of the input set that may cover different ranges, or we must implement multiple phases.

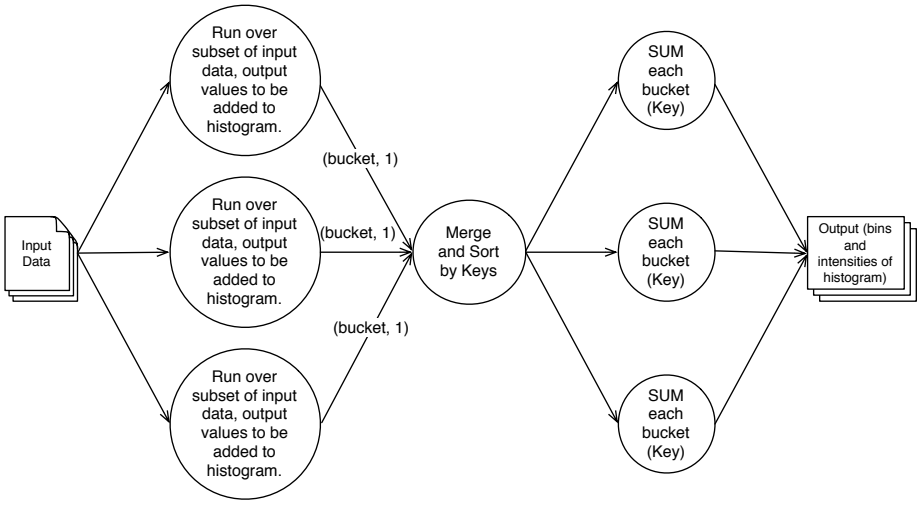


Figure 1: Diagram of MapReduce Histogram Generation Pattern

Figure 1 illustrates the second method described above for histogram generation of large data sets with MapReduce.

This package is designed to be helpful if some of the Map or Reduce tasks are written in R, or if those components are written in other languages and only the resulting output histograms need to be manipulated in R.

3 Efficient Representations of Histograms

Consider an example histogram of 100 random data points. Figure 2 shows the graphical representation and list structure of R histogram objects.

```
> myhist <- hist(runif(100))
```

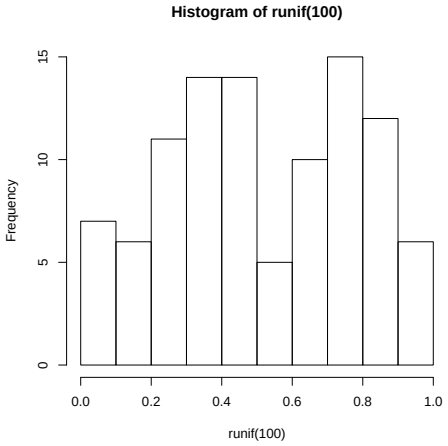
This histogram compactly represents the full distribution. Histogram objects in R are lists with 7 components: breaks, counts, density, mids, name, and equidist.

If we are working in a parallel environment and need to distribute such a histogram to other tasks running in a compute cluster, we need to serialize this histogram object to a binary format that can be transferred over the network.

3.1 Native R Serialization

R includes a built-in serialization framework that allows one to serialize any R object to an Rdata file.

```
> length(serialize(myhist, NULL))
```




```
$breaks
 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

$counts
 [1]  7  6 11 14 14  5 10 15 12  6

$intensities
 [1] 0.7 0.6 1.1 1.4 1.4 0.5 1.0 1.5 1.2 0.6

$density
 [1] 0.7 0.6 1.1 1.4 1.4 0.5 1.0 1.5 1.2 0.6

$mids
 [1] 0.05 0.15 0.25 0.35 0.45 0.55 0.65 0.75 0.85 0.95

$xname
[1] "runif(100)"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"
```

Figure 2: Example Histogram

```
[1] 650
```

This works and is quite convenient if the histogram must only be shared between tasks running the R interpreter, but it is not a very portable format.

3.2 Protocol Buffers

Protocol Buffers are a flexible, efficient, automated, cross-platform mechanism for serializing structured data (**??**). The RProtoBuf package (**?**) provides an interface for manipulating protocol buffer objects directly within R.

Of the 7 elements stored in an R histogram object, we only need to store three in our serialization format since the others can be re-computed. This leads to the following simple protocol buffer definition of the breaks, counts, and name of a histogram:

```
syntax = "proto2";

 package HistogramTools;

 message HistogramState {
   repeated double breaks = 1;
   repeated int32 counts = 2;
   optional string name = 3;
 }
```

The package provides `as.Message` and `as.histogram` methods for converting between R histogram objects and this protocol buffer representation.

In addition to the added portability, the protocol buffer representation is significantly more compact.

```
> hist.msg <- as.Message(myhist)
```

Our histogram protocol buffer has a human-readable ASCII representation:

```
> cat(as.character(hist.msg))
```

```
RProtoBuf library not available
```

But it is most useful when serialized to a compact binary representation:

```
> length(hist.msg$serialize(NULL))
```

```
RProtoBuf not available.NULL
```

This protocol buffer representation is not compressed by default, however, so we can do better:

```
> raw.bytes <- memCompress(hist.msg$serialize(NULL), "gzip")
> print(length(raw.bytes))
```

```
RProtoBuf not availableNULL
```

We can then send this compressed binary representation of the histogram over a network or store it to a cross-platform data store for later analysis by other tools. To recreate the original R histogram object from the serialized protocol buffer we can use the `as.histogram` method.

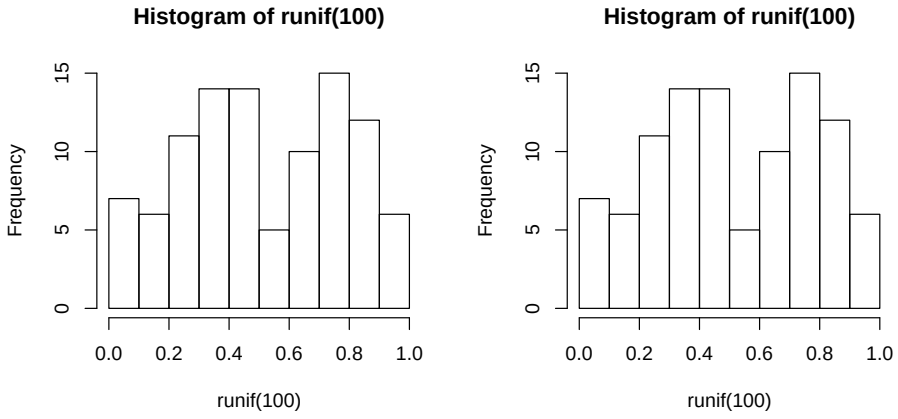
```
> uncompressed.bytes <- memDecompress(raw.bytes, "gzip")
```

```
> if (require(RProtoBuf)) {
+ length(uncompressed.bytes)
+ new.hist.proto <- HistogramTools.HistogramState$read(uncompressed.bytes)
+ }
```

```
NULL
```

The resulting histogram is the same as the original; it was converted to a protocol buffer, serialized, compressed, then uncompressed, parsed, and converted back to a histogram.

```
> par(mfrow=c(1,2))
> plot(myhist)
> if (require(RProtoBuf)) {
+ plot(as.histogram(new.hist.proto))
+ } else {
+ plot(myhist)
+ }
```



4 Quantiles and Cumulative Distribution Functions

When histograms are used as a binned data storage mechanism to reduce data storage cost, information about the underlying distribution is lost. We can however approximate the quantiles and cumulative distribution function for the underying distribution from the histogram.

The `Count`, `ApproxMean`, and `ApproxQuantile` functions are meant to help with this, but note that they will only be accurate with very granular histogram buckets. They would rarely be appropriate with histogram buckets chosen by the default algorithm in R.

```
> hist <- hist(c(1,2,3), breaks=c(0,1,2,3,4,5,6,7,8,9), plot=FALSE)

> Count(hist)

[1] 3

> ApproxMean(hist)

[1] 1.5

> ApproxQuantile(hist, .5)

50%
1.5

> ApproxQuantile(hist, c(.05, .95))

 5% 95%
0.6 5.1
```

The `HistToEcdf` function takes a histogram and returns an empirical distribution function similar to what is returned by the `ecdf` function on a distribution.

```
> h <- hist(runif(100), plot=FALSE)
> e <- HistToEcdf(h)
> e(.5)

[1] 0.46

> par(mfrow=c(1,2))
> plot(h)
> plot(HistToEcdf(h))
> par(mfrow=c(1,1))
```

5 Error estimates in CDFs approximated from histograms

When constructing cumulative distribution functions from binned histogram data sets there will usually be some amount of information loss. We can however come up with an upper bound for the error by looking at two extreme cases. For a given histogram h with bucket counts C_i for $1 \leq i \leq n$ and left-closed bucket boundaries B_i for $1 \leq i \leq n+1$, we construct two data sets. Let X be the (unknown) underlying data set for which we now only have the binned representation h. Let $X_{\min}$ be the data set constructed by assuming the data points in each bucket of the histogram are all equal to the left bucket boundary. Let $X_{\max}$ be the data set constructed by assuming the data points in each bucket of the histogram are at the right bucket boundary. Then F_X is the true empirical CDF of the underlying data, and $F_{X_{\min}}(x) \geq F_X(x) \geq F_{X_{\max}}(x)$.

$$X_{\min} = \left((B_i)_{k=1}^{C_i} : 1 \leq i \leq n\right) X_{\max} = \left((B_{i+1})_{k=1}^{C_i} : 1 \leq i \leq n\right)$$

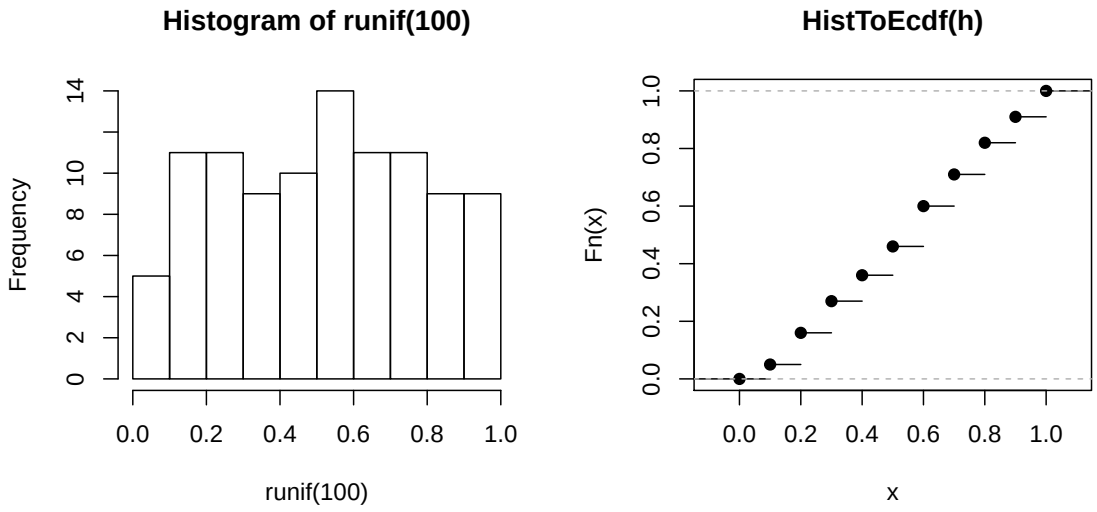


Figure 3: Histogram and CDF Created with HistToEcdf

The package provides two different distance metrics to measure the difference between empirical cumulative distribution functions $F_{X_{\min}}$ and $F_{X_{\max}}$. These distances serve as upper-bounds for the amount of error between the true empirical distribution function F_X of the unbinned data and an ECDF calculated from the binned data.

5.1 Kolmogorov-Smirnov Distance of the Cumulative Curves

The first metric provided by the package is the two-sample Kolmogorov-Smirnov distance between $X_{\min}$ and $X_{\max}$. In other words, it the largest possible distance between cumulative distribution functions that could be represented by the binned data. This metric is more formally defined as

$$\sup_x |F_{X_{\max}}(x) - F_{X_{\min}}(x)|$$

This function is also occasionally called the maximum displacement of the cumulative curves (MDCC).

```
> x <- rexp(1000)
> h <- hist(x, breaks=c(0,1,2,3,4,8,16,32), plot=FALSE)
> x.min <- rep(head(h$breaks, -1), h$counts)
> x.max <- rep(tail(h$breaks, -1), h$counts)
> ks.test(x.min, x.max, exact=F)

        Two-sample Kolmogorov-Smirnov test

data:  x.min and x.max
D = 0.608, p-value < 2.2e-16
alternative hypothesis: two-sided
```

```
> KSDCC(h)
[1] 0.608
```

The `KSDCC` function accepts a histogram, generates the largest and smallest empirical cumulative distribution functions that could be represented by that histogram, and then calculates the Kolmogorov-Smirnov distance between the two CDFs. This measure can be used in cases where expanding the binned data into `x.min` and `x.max` explicitly to calculate distances using e.g. `ks.test` directly would not be feasible. Figure 4 illustrates geometrically what value is being returned.

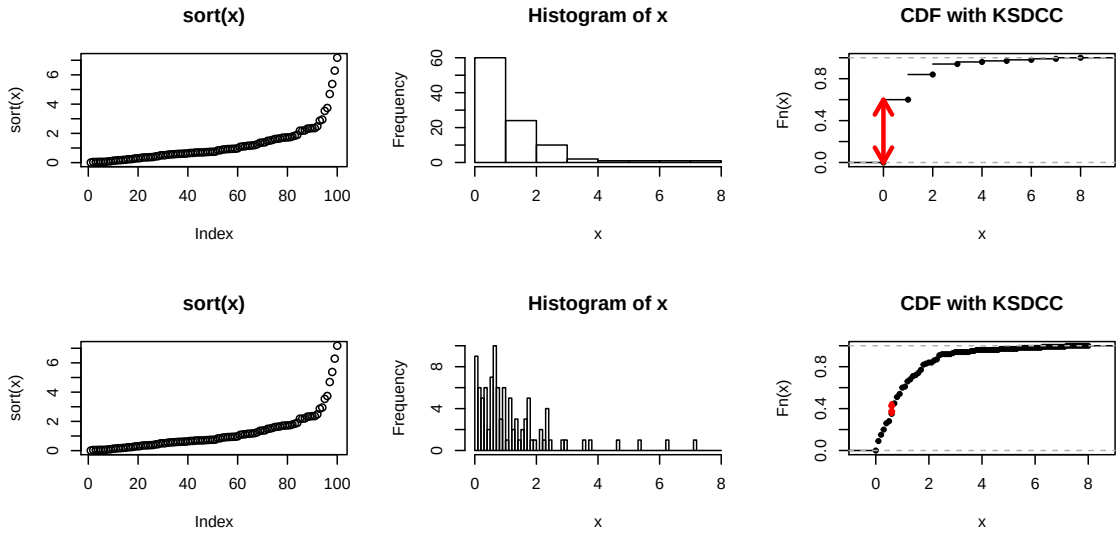


Figure 4: Sorted raw data (column 1), Histograms (column 2), CDF with KSDCC (column 3)

5.2 Earth Mover's Distance of the Cumulative Curves

The "Earth Mover's Distance" is like the Kolmogorov-Smirnof statistic, but uses an integral to capture the difference across all points of the curve rather than just the maximum difference. This is also known as Mallows distance, or Wasserstein distance with $p = 1$.

The value of this metric for our histogram h is

$$\int_{\mathbb{R}} |F_{X_{\max}}(x) - F_{X_{\min}}(x)|dx,$$

Figure 5 shows the same e.c.d.f of the binned data represented in the previous section, but with the yellow boxes representing the range of possible values for any real e.c.d.f. having the same binned representation. For any distribution X with the binned representation in h, the e.c.d.f $F(X)$ will pass only through the yellow boxes. The area of the yellow boxes is the Earth Mover's Distance.

```
> x <- rexp(100)
> h1 <- hist(x, plot=FALSE)
```

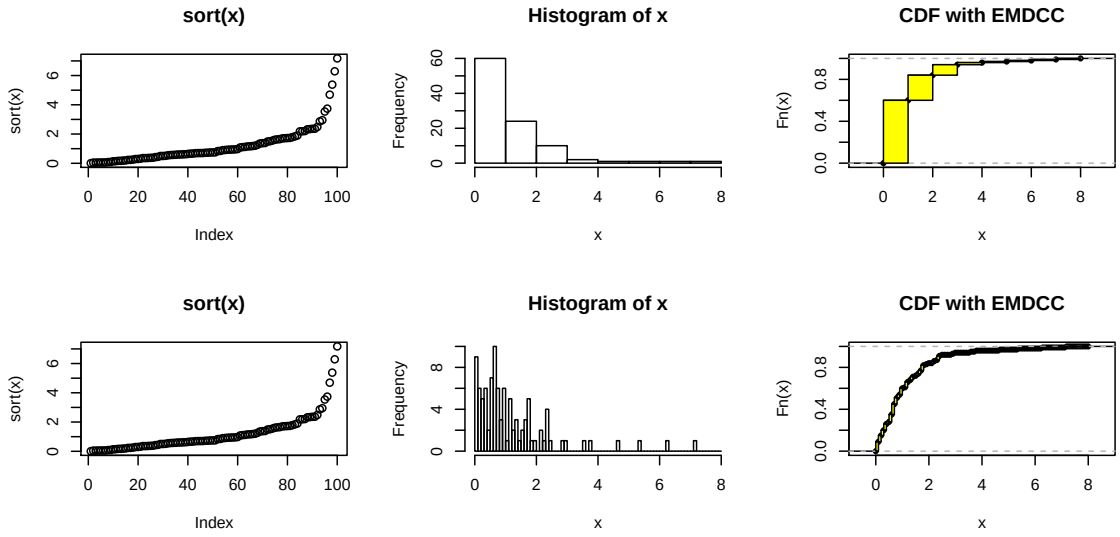


Figure 5: Sorted raw data (column 1), Histograms (column 2), CDF with EMDCC (column 3)

```
> h2 <- hist(x, breaks=seq(0,round(max(x) + 1),by=0.1), plot=FALSE)
> KSDCC(h1)
[1] 0.34
> KSDCC(h2)
[1] 0.12
> EMDCC(h1)
[1] 0.1
> EMDCC(h2)
[1] 0.01666667
```

So, using this metric, the second lower example with more granular histogram bins produces an ECDF with reduced worst case error bounds compared to the one with default buckets, as expected.

6 Histogram Bin Manipulation

6.1 Trimming Empty Buckets from the Tails

When generating histograms with a large number of fine-grained bucket boundaries, the resulting histograms may have a large number of empty consecutive buckets on the left or right side of the histogram. The `TrimHistogram` function can be used to remove them as illustrated in Figure 6.

```
> hist.1 <- hist(runif(100,min=2,max=4), breaks=seq(0,6,by=.2), plot=FALSE)
> hist.trimmed <- TrimHistogram(hist.1)
```

```
> length(hist.1$counts)
[1] 30
> sum(hist.1$counts)
[1] 100
> length(hist.trimmed$counts)
[1] 10
> sum(hist.trimmed$counts)
[1] 100
> par(mfrow=c(1,2))
> plot(hist.1)
> plot(TrimHistogram(hist.1), main="Trimmed Histogram")
```

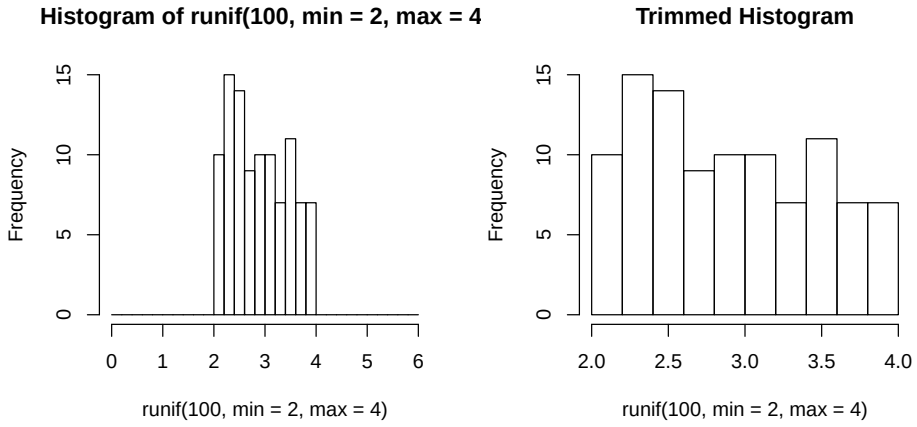


Figure 6: Effect of the `TrimHistogram` function.

6.2 Adding/Merging Histograms

If histograms (for different data sets) have the same bucket boundaries, it is possible to add them together to obtain the histogram for the combined data set by aggregating the counts values with the `AddHistograms` function illustrated in Figure 7.

```
> hist.1 <- hist(c(1,2,3,4), plot=FALSE)
> hist.2 <- hist(c(1,2,2,4), plot=FALSE)
> hist.sum <- AddHistograms(hist.1, hist.2)
```

`AddHistograms` accepts an arbitrary number of histogram objects to aggregate as long as they have the same bucket boundaries.

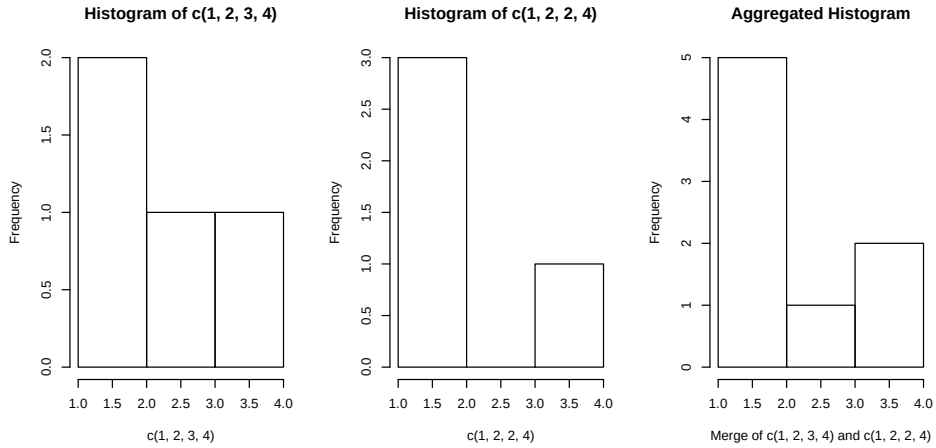


Figure 7: Effect of the `AddHistograms` function.

```
>   hist.1 <- hist(c(1,2,3), breaks=0:9, plot=FALSE)
>   hist.2 <- hist(c(1,2,3), breaks=0:9, plot=FALSE)
>   hist.3 <- hist(c(4,5,6), breaks=0:9, plot=FALSE)
>   hist.sum <- AddHistograms(hist.1, hist.2, hist.3)
>   hist.sum
$breaks
 [1] 0 1 2 3 4 5 6 7 8 9

$counts
[1] 2 2 2 1 1 1 0 0 0

$density
[1] 0.2222222 0.2222222 0.2222222 0.1111111 0.1111111 0.1111111
[7] 0.0000000 0.0000000 0.0000000

$mids
[1] 0.5 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5

$xname
[1] "Merge of 3 histograms"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"
```

6.3 Merging Buckets

We may want a version of a histogram with fewer buckets, to save storage or to produce better plots. The `MergeBuckets` function takes a histogram and a parameter for the number of adjacent buckets to merge together and returns a new histogram with different bucket boundaries as illustrated in Figure 8.

```
> overbinned <- hist(c(rexp(100), 1+rexp(100)), breaks=seq(0, 10, by=.01), plot=FALSE)
> better.hist <- MergeBuckets(overbinned, adj=30)
```

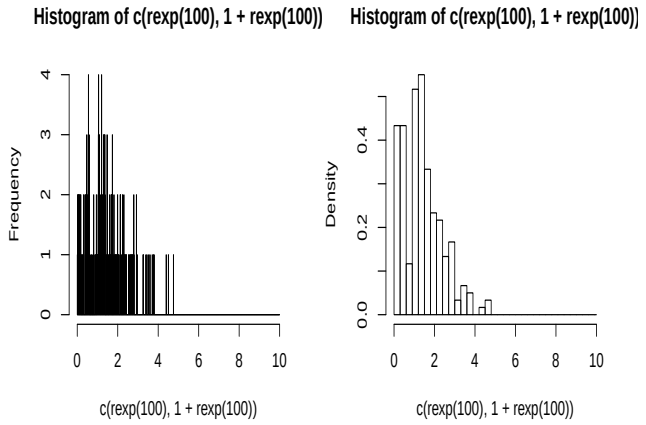


Figure 8: Effect of the `MergeBuckets` function.

6.4 Subsetting Histograms

The `SubsetHistogram` function takes a histogram and a new minimum and maximum bucket boundary and returns a new histogram with a subset of the buckets as illustrated in Figure 9.

```
> hist.1 <- hist(runif(100, min=0, max=10), breaks=seq(from=0, to=10, by=.5), plot=FALSE)
> hist.2 <- SubsetHistogram(hist.1, minbreak=2, maxbreak=6)
```

7 Summary

The HistogramTools package presented in this paper has been in wide use for the last several years to allow engineers to read distribution data from internal data stores written in other languages. Internal production monitoring tools and databases, as well as the output of many MapReduce jobs have been used.

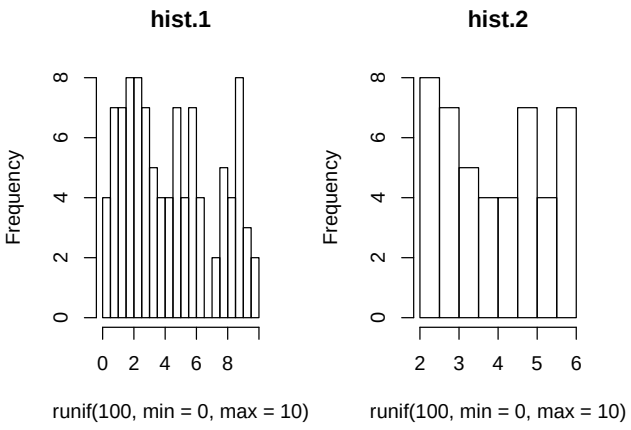


Figure 9: Effect of the `SubsetHistogram` function.