Stratified Subsample Function

Vincent Knight-Schrijver

10 September 2021

Introduction

This function was developed to evenly sample cells from an unique ID vector across two different factors or stratifications. For example, 1, "cluster ID" and 2) "donor ID"), and produce a new evenly distributed but smaller unique ID vector. This document describes the function and goes on to show results and discusses some uses and things to consider.

TL:DR

f1 and f2 are factors while cell_IDs is a unique character vector. meanwhile N is the threshold to stop sampling cells from factor 1. N can be determined arbitrarily to equalise to a particular cluster or by some objective function.

```
  new.sample = stratified_subsample(
    f1 = clusters,
    f2 = donors,
    cell_IDs = cell_IDs,
    N = n
  )
```

Otherwise, stick through the code and see what was considered.

Randomly sampled dataset for the vignette here.

We'll generate some random data classes to simulate our sampling strategy. For example, a dataset with 10,000 - 20,000 cells, maximum of 7 donors and a bunch of clusters, say, 13.

```
# colours for plotting
# Discrete but up to 13
  Discrete13 = c(
  '#a6cee3',
  '#1f78b4',
  '#b2df8a',
  '#33a02c',
  '#fb9a99',
  '#e31a1c',
  '#fdbf6f',
  '#ff7f00',
  '#cab2d6',
  '#6a3d9a',
  '#ffff99',
  '#b15928',
  'black'
  )

# SEED for the random sample
  set.seed(42)

  cell_n = 15000
  donor_n = 7
  clus_n = 13

  cell_IDs = unique(sapply(1:cell_n, function(i){paste(sample(LETTERS,25), collapse="")}))
  donors = as.factor(
    sort(
      sample(
        rep(LETTERS[1:donor_n], abs(round(rnorm(donor_n, (cell_n/donor_n*2), cell_n/donor_n))))
      )[1:cell_n]
      )
    )
  clusters = unlist(
      sapply(levels(donors), function(i) {
        sample(as.factor(
          rep(
            1:clus_n,
            abs(round(rnorm(clus_n, sum(donors==i)/(clus_n*2), 4*(sum(donors==i)/clus_n))))
          )
        ))[1:sum(donors==i)]
      }
    )
  )


# name these factors
  names(donors) = cell_IDs
  names(clusters)= cell_IDs
```

We can see that the classes are unevely distributed

```
table(donors)
```

```
## donors
##    A    B    C    D    E    F    G
## 1561 1734 1456  966 2157 3618 3508
```

```
table(clusters)
```

```
## clusters
##    1    2    3    4    5    6    7    8    9   10   11   12   13
##  843 1417  759 1021 1475  699 1594  640 1661 1095  861 1864 1071
```

```
#######################################################
```

```
# function for plots in this vignette
  plot.pops = function(var1, var2, col=Discrete13){

      cell.id.matrix <- matrix(0, length(levels(var1)), length(levels(var2)))
      rownames(cell.id.matrix) <- levels(var1)
      colnames(cell.id.matrix) <- levels(var2)
      # table format
      for(i in levels(var2)){
        cell.id.matrix[names(table(var1[var2==i])),i] <- table(var1[var2==i])
      }

      barplot(cell.id.matrix,
          col=Discrete13,
          las=1, xaxt="n", border=NA, cex.axis=0.7,
          horiz=T, names=gsub(".*- ", "", colnames(cell.id.matrix))
          )
      mtext("# of cells",3, padj=-3)
      axis(3)
    }

# pre-subsample barplot of distributions
    plot.pops(donors, clusters)
```

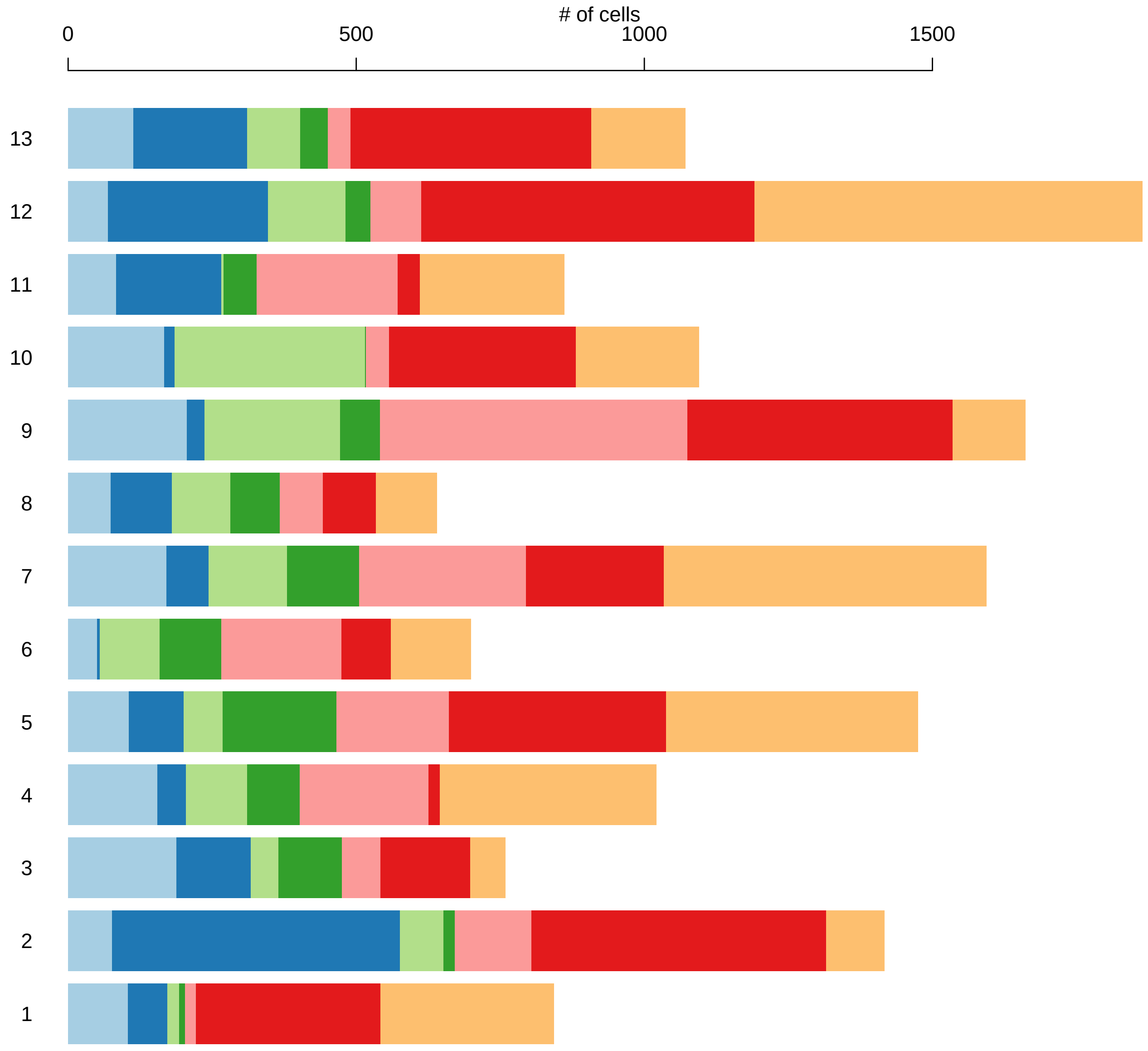


Figure 1: Non-subsampled dataset shows large cell type and donor variance.

We then describe the stratified sampling function here, we can use multiple methods. Firstly we will consider the most simple option and collapse the factor levels into one unified factor.

```
# Simply paste the factors together:
  combined_factor = as.factor(paste(clusters, donors, sep="###"))
# Re-apply the names to the new factor
  names(combined_factor) = cell_IDs
```

Use a simple sampling function to evenly sample from every level across this factor

```
# Function
# ID_vector is unique cells
# class_factor is the classes to sample from
# n is the number from each level to sample
# override means that samples from levels will be taken even if the minimum level size is less than n.
#    otherwise this function will sample evenly at the minimum level size
```

```r
sample.equal = function(ID_vector, class_factor, n, override = F){
  n2sub <-  min(table(class_factor))
  if(n < n2sub){a = n} else {a = n2sub}

  equal.sample.vector <- unlist(lapply(levels(class_factor), function(k) {
      if(override==T){sample(ID_vector[class_factor==k], min(c(length(ID_vector[class_factor==k]), n)
        sample(ID_vector[class_factor==k], a, replace=F)
      }
    }))

  return(equal.sample.vector)
}
```

Run the function on our newly combined factor vector

```r
sample1 = sample.equal(cell_IDs, combined_factor, 10, override=T)
sample2 = sample.equal(cell_IDs, combined_factor, 50, override=T)
sample3 = sample.equal(cell_IDs, combined_factor, 100, override=T)

table(donors[sample1])
```

```
##
##   A   B   C   D   E   F   G
## 130 125 124 122 130 130 130
```

```r
table(clusters[sample1])
```

```
##
##  1  2  3  4  5  6  7  8  9 10 11 12 13
## 70 70 70 70 70 65 70 70 70 62 64 70 70
```

```r
######################################################
  par(mfrow=c(1,3), mar=c(4,4,4,0), oma=c(0,6,0,6))
# post-subsample barplot of distributions
    plot.pops(donors[sample1], clusters[sample1])
      mtext("Cluster ID",2, padj=-3)

# post-subsample n = 50
    plot.pops(donors[sample2], clusters[sample2])

# post-subsample n = 100
    plot.pops(donors[sample3], clusters[sample3])
```

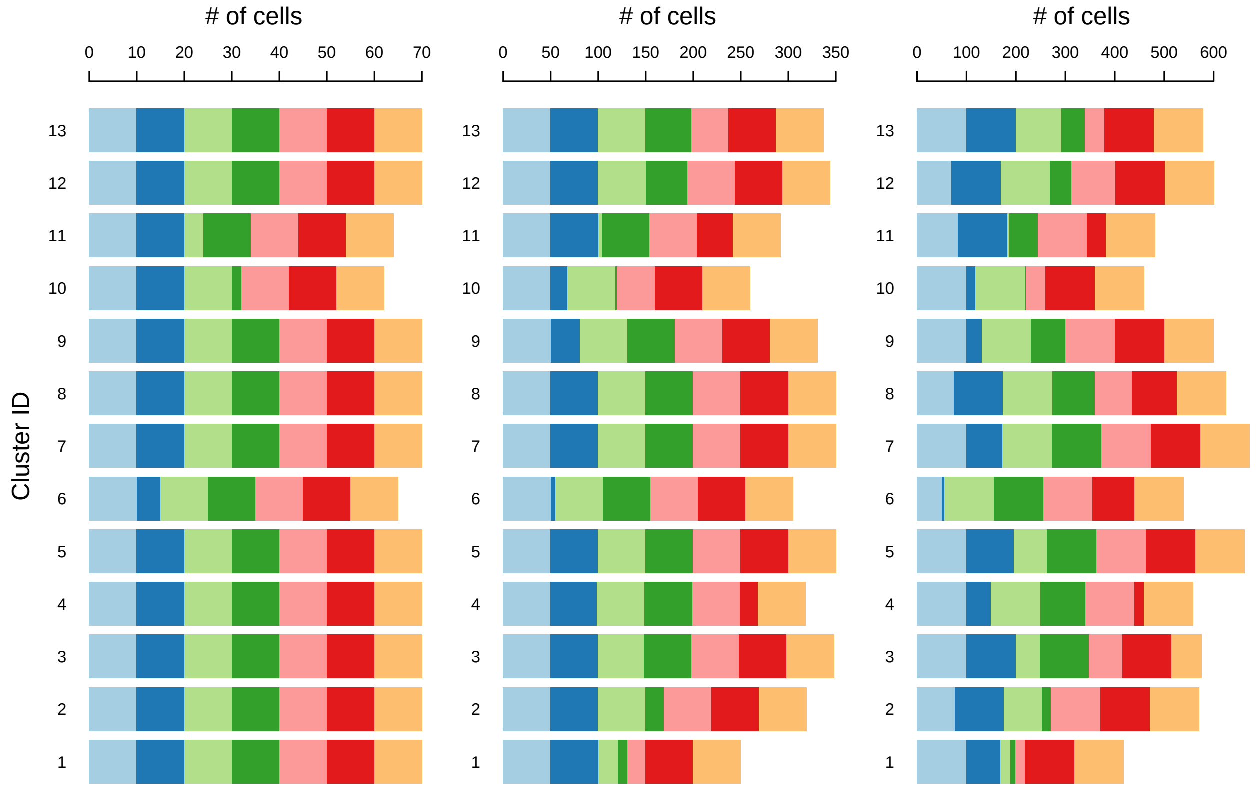


Figure 2: Dataset sampled using a simple function is effective at low sampling sizes but loses cluster size homogeneity at larger sampling sizes.

However, we notice above that higher sampling value show greater differences between clusters or donors, and the even composition of this new sample drops off. At this point we may decide to preserve the cluster size or donor size across samples (or stick to a smaller dataset, which might work if you have enough power for the analysis). This next function is a long-winded interative sampling approach that samples evenly from all donors until the cluster size reaches n. If it fails to do so, it is because there are no more cells to sample from this donor or cluster. I appreciate that this is a rather complex and poorly annotated function.

```r
# Arguments:

# clusters - the cluster labels - with the groups shared within each cluster k
# donors - the vector of class / sample labels (not the levels), or groups shared within clusters
# cell_IDs - the vector of sample names, essentially column names of a [genes, cells] matrix
# N - the number of samples to take from each cluster in total... typically it could be the smallest cl
# for generalisation, I call clusters and donors f1 and f2 (factor 1 and factor 2)

  stratified_subsample <- function(f1, f2, cell_IDs, N, seed=1){

  ### For scRNAseq dataset, the principle was to reduce clusters to equal sizes, equally distributed
  #  by donors...
  # f1 == clusters
    clusters = f1
  # f2 = donors
    donors = f2

    class.cluster.table = sapply(levels(clusters), function(K){table(donors[clusters==K])})
```

```
# re-seed (resetting the Seurat global seed fix in analysis)
   set.seed(1)
   rm(.Random.seed, envir=globalenv())
   set.seed(seed)

  # smallest cluster = n
    samples.clusters = lapply(levels(clusters), function(K) {
  #  for each cluster...
   # Sample the minimum of N / class samples
     sample.loop = sample.equal(# SAMPLE FROM the cells in that cluster K
         ID_vector = cell_IDs[clusters == K],
       # donors associated with cells in cluster == K
         class_factor = donors[clusters == K],
       # Sample the minimum of the minimum cluster size / number of donors in each K cluster
         n = min(round(N/length(levels(donors[clusters == K]))), min(class.cluster.table[,K]))
     )
      # OMIT the names already sampled (a hacky "without replacement")
       sample.omit = !cell_IDs[clusters == K] %in%  sample.loop

     # RESAMPLE to make up to ~ N
     while(
       # Total clusters  >  # Clusters remaining after omitting already sampled samples
       length(unique(donors[clusters == K])) >= length(unique(donors[clusters == K][sample.omit])) &
       length(sample.loop) < N &
       sum(sample.omit) > 0   # FINISH LOOP AND RETURN IF CLUSTER K HAS ALL BEEN SAMPLED
      ){
       # New TOTAL number to sample...
        new.target = N - length(sample.loop)
       # Sample the minimum length, adding onto our sample.loop vector
        sample.loop = c(sample.loop,

          sample.equal(
       # cell_IDs, omitting the already sampled samples
         ID_vector = cell_IDs[clusters == K][sample.omit],
       # donors associated with cells in cluster == K
         class_factor = as.factor(as.character(donors[clusters == K][sample.omit])),
       # Sample the minimum of the minimum cluster size / number of donors in each K cluster
         n = min(
            ceiling(new.target/length(unique(donors[clusters == K][sample.omit]))),
            min(table(as.factor(as.character(donors[clusters == K][sample.omit]))))
            )
          )
        )
       # discard sampled items again, and repeat loop if conditions are still met...
         sample.omit = !(cell_IDs[clusters == K] %in%  sample.loop)
     }

   # Remove duplicated items - This occurs if sample number chosen is above the
   # number of samples in cluster and the class does not round into N
    return(sample.loop)
  }
  )
    return(unlist(samples.clusters))
  }
```

```
  sample4 = stratified_subsample(clusters, donors, cell_IDs, 100)

  table(donors[sample4])
```

```
##
##   A   B   C   D   E   F   G
## 199 188 187 179 199 199 199
```

```
  table(clusters[sample4])
```

```
##
##   1   2   3   4   5   6   7   8   9  10  11  12  13
## 100 105 105 105 105 101 105 105 105 104 100 105 105
```

In demonstrating this function's output we can view it in comparison with the output of the simpler function across the different stratification sizes, 10, 50, 100 (of each cluster-donor combination) compared with ∼ 70, 350, 700 (of final cluster size).

```
# Use complex function to generate new samples
  sample5 = stratified_subsample(clusters, donors, cell_IDs, donor_n*10)
  sample6 = stratified_subsample(clusters, donors, cell_IDs, donor_n*50)
  sample7 = stratified_subsample(clusters, donors, cell_IDs, donor_n*100)

######################################################
par(mfrow=c(2,3), mar=c(4,4,4,0), oma=c(0,6,0,6))
# post-subsample barplot of distributions
   plot.pops(donors[sample1], clusters[sample1])
   mtext("Simple function \n Cluster ID",2, padj=-3)

# post-subsample n = 50
   plot.pops(donors[sample2], clusters[sample2])

# post-subsample n = 100
   plot.pops(donors[sample3], clusters[sample3])

## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
# post-subsample n = 70
   plot.pops(donors[sample5], clusters[sample5])
      mtext("Complex function \n Cluster ID",2, padj=-3)
      mtext("small",1, padj=0)

# post-subsample n = 350
   plot.pops(donors[sample6], clusters[sample6])
      mtext("medium",1, padj=0)

# post-subsample n = 700
   plot.pops(donors[sample7], clusters[sample7])
      mtext("large",1, padj=0)
```

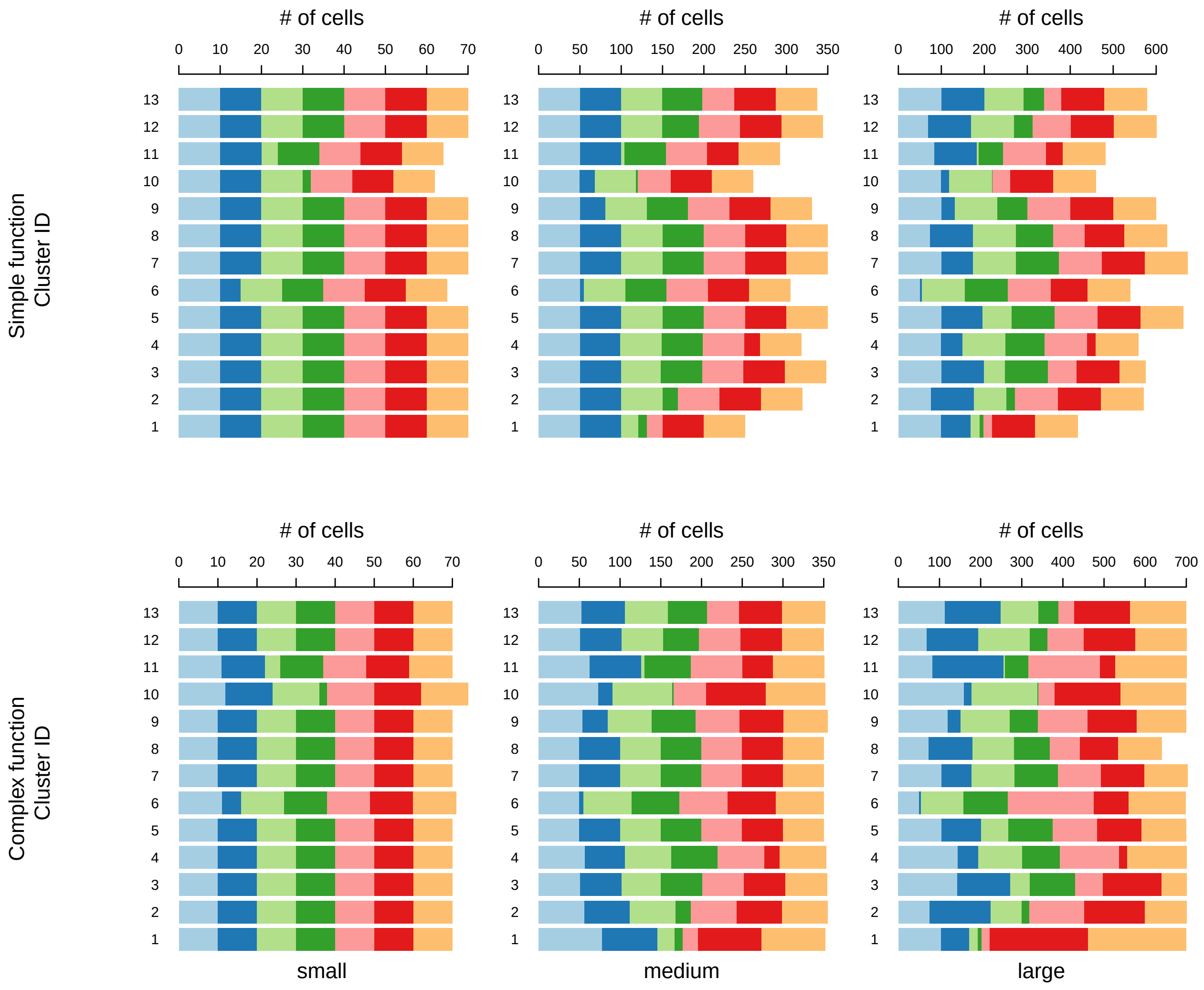


Figure 3: Complex and simple stratified sampling functions compared show that the complex function maintains cluster size homogeneity at larger sampling sizes.

We see that donor sampling is also different between the two methods.

```
#############################################################
par(mfrow=c(2,4), mar=c(4,4,4,0), oma=c(0,6,0,6))
# post-subsample barplot of distributions
  plot.pops(clusters[sample1], donors[sample1])
    mtext("Simple function \n Donor ID",2, padj=-3)

# post-subsample n = 50
   plot.pops(clusters[sample2], donors[sample2])

# post-subsample n = 100
   plot.pops(clusters[sample3], donors[sample3])

# NO SAMPLING
   plot.pops(clusters[sample3], donors[sample3])
     mtext("NO SAMPLING - all cells",1, padj=0)
```

```
## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
# post-subsample n = 70
   plot.pops(clusters[sample5], donors[sample5])
      mtext("Complex function \n Donor ID",2, padj=-3)
      mtext("small",1, padj=0)

# post-subsample n = 350
   plot.pops(clusters[sample6], donors[sample6])
      mtext("medium",1, padj=0)

# post-subsample n = 700
   plot.pops(clusters[sample7], donors[sample7])
      mtext("large",1, padj=0)
```

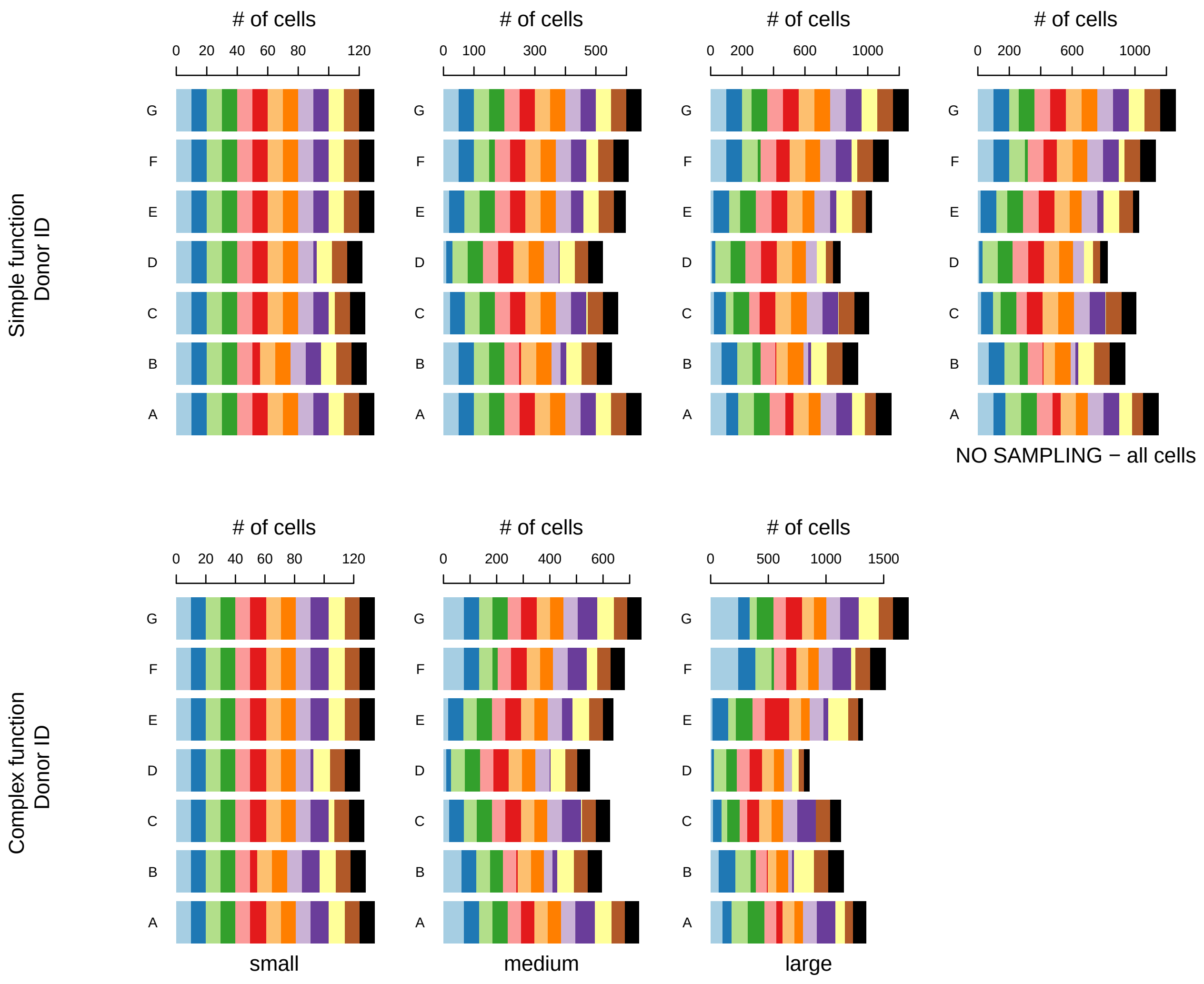
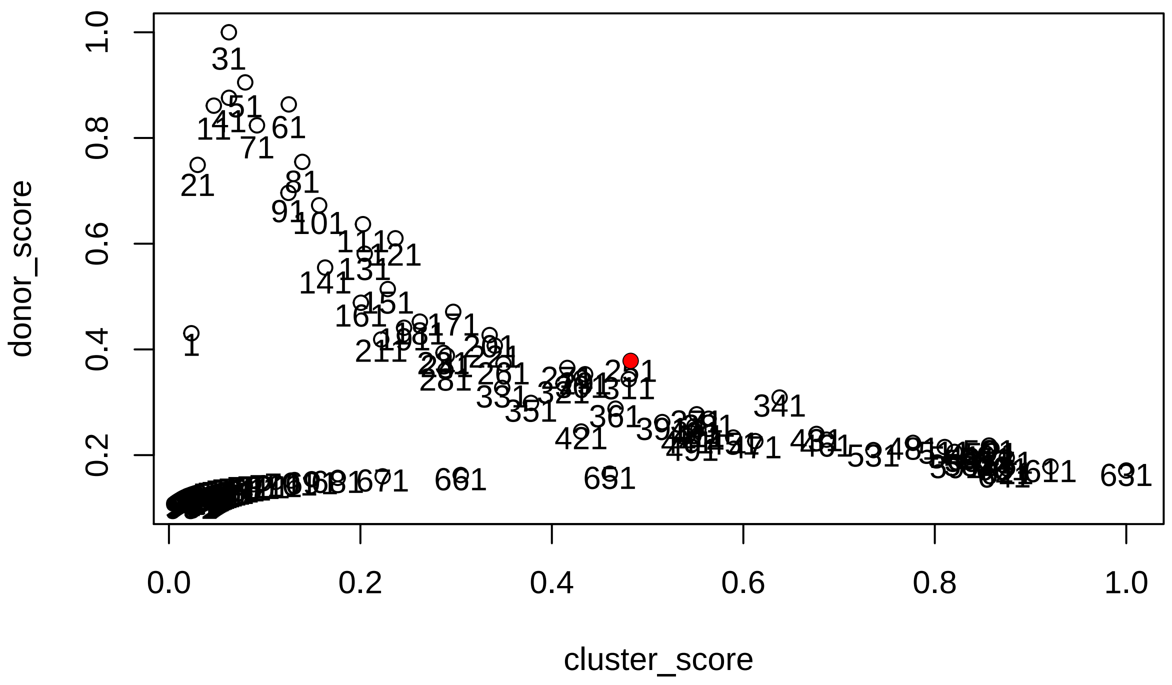


Figure 4: Switching the visualisation from clusters to donors shows however, that larger sampling numbers can make uneven donor sampling with the complex function

Discussion of use

Well that covers the sampling function and results. It also suggests that the larger clusters you try to sample (across uneven group sizes) the less equally you can sample, so it's up to your subjective discretion to find the sweet spot, or focus on maximising a partcular cell group cluster (such as me trying to retrieve all epicardial cells and having the dataset align with the number of epicardial cells). Alternatively an objective function could be set up between donor and cluster tables, such as using the standard deviation between the composition of donors and clusters and the number of cells sampled. Here, I see if we can calculate an objective of minimising standard deviation while maximising mean cluster sizes.

```
# Very heuristic funcion optimisation... harmonic mean of cluster and donor means / variance of each
  sds_cluster = numeric()
  sds_donor = numeric()
  mean_cluster = numeric()
  mean_donor = numeric()
  for(i in seq(1, 1000, 10)){
    sample8 = stratified_subsample(clusters, donors, cell_IDs, i)
    sds_cluster = c(sds_cluster,sd(table(clusters[sample8])))
    sds_donor = c(sds_donor,sd(table(donors[sample8])))

    mean_cluster = c(mean_cluster,mean(table(clusters[sample8])))
    mean_donor = c(mean_donor,mean(table(donors[sample8])))
  }

  cluster_score = (mean_cluster/(sds_cluster+1)) / max(mean_cluster/(sds_cluster+1))
  donor_score = (mean_donor/(sds_donor+1)) / max(mean_donor/(sds_donor+1))

# harmonic mean of (mean / sd)
  optimal.i = which.max(
              2 * (donor_score*cluster_score) /
                  (donor_score+cluster_score)
              )

## multiplication of (mean / sd)
#  optimal.n = seq(1, 1000, 10)[which.max((mean_donor/sds_donor)*(mean_cluster/sds_cluster))]

# Plot these data points
  plot(cluster_score, donor_score)
    text(cluster_score, donor_score-(donor_score)*0.05, seq(1, 1000, 10))
# One objective? - optimal sampling number
  points(cluster_score[optimal.i], donor_score[optimal.i],
    col="red", pch=16)
```



```
# optimal number to sample
  optimal.n = seq(1, 1000, 10)[optimal.i]

# run function:
  sample8 = stratified_subsample(clusters, donors, cell_IDs, optimal.n)
```

And have a look at the results:

```
par(mfrow=c(2,2), mar=c(4,4,4,0), oma=c(0,6,0,6))
########################################################
  plot.pops(donors, clusters)
    mtext("BEFORE\n\nCluster ID",2, padj=-2)

# pre-subsample barplot of distributions
  plot.pops(clusters, donors)
    mtext("Donor ID",2, padj=-2)

########################################################
# post-subsample barplot of distributions
  plot.pops(donors[sample8], clusters[sample8])
    mtext("AFTER\n\nCluster ID",2, padj=-2)

# post-subsample barplot of distributions
  plot.pops(clusters[sample8], donors[sample8])
    mtext("Donor ID",2, padj=-2)
```

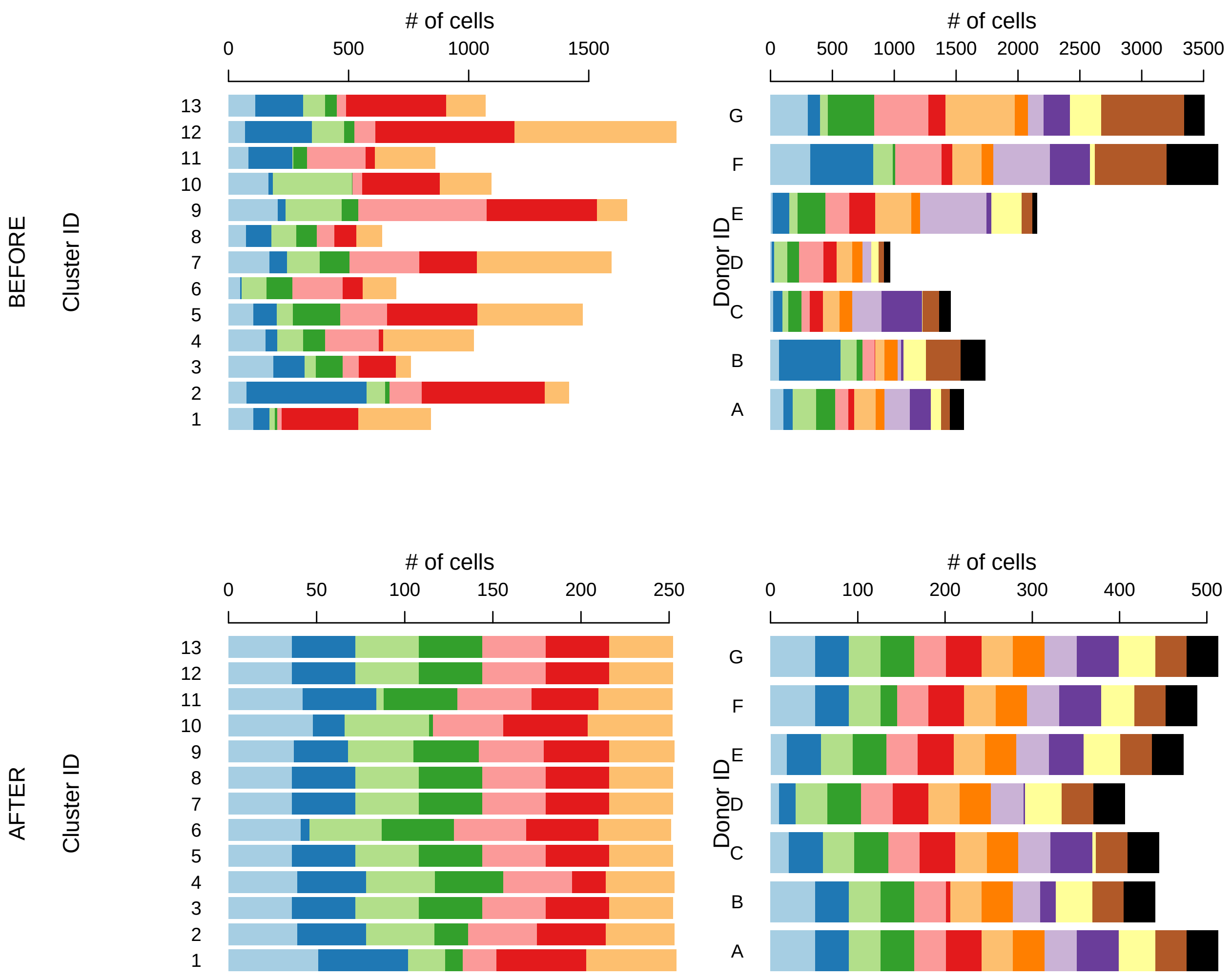


Figure 5: Objectively optimal sampling number and results of sampling both donor and cell clusters

```
########################################################
```

TLDR

f1 and f2 are factors while cell_IDs is a unique character vector. meanwhile N is the threshold to stop sampling cells from factor 1. N can be determined arbitrarily to equalise to a particular cluster or by some objective function.

```
new.sample = stratified_subsample(
   f1 = clusters,
   f2 = donors,
   cell_IDs = cell_IDs,
   N = n
 )
```