adjClust Package Vignette

Christopher Bolen

13 October, 2015

Contents

1 Introduction

This package implements the Adjacency Clustering algorithm designed by Chris Bolen.

2 Installation

Currently, installing AdjClust takes a bit of work. This will eventually be streamlined, but until the code is done, you'll need to download the source code from Bitbucket and compile it according to the instructions below.

2.1 Install required packages

```
> install.packages(c("gplots","hexbin","parallel","png"))
> source("https://bioconductor.org/biocLite.R")
> biocLite(c("cytofkit", "flowCore"))
```

2.2 Building adjClust

The build instructions assume that you have downloaded the most recent version of AdjClust, and have started an RStudio project in the download directory.

1. Install build dependencies:

   ```
   > install.packages(c("devtools", "roxygen2", "testthat", "knitr", "rmarkdown"))
   ```

2. Build with RStudio

   - Build -> Configure Build Tools
   - Check use devtools option
   - Check use roxygen option
   - Select configure roxygen options and check everything.
   - Build -> Build and Reload

2.3 Compiling tSNE

Before tSNE can be used, you must compile the source code. AdjClust contains a function, `compileTSNE`, which will automatically compile the code and make it usable within R. To use your system's default c++ compiler, simply call `compileTSNE` using the default parameters:

```
> compileTSNE()
```

2.3.1 Compiling tSNE with OpenMP support

If you are planning to run the adjClust algorithm with large datasets, it is almost a necessity that the tSNE algorithm be run in parallel. To do this, we will need to compile the code with OpenMP support. On Linux and Windows systems, this should be as simple as adding a single parameter to the `compileTSNE` function call:

```
> compileTSNE(use_omp=TRUE)
```

OpenMP support on OSX

As of October 2015, the default compiler for Mac operating systems (”clang++”) does not come with a working copy of OpenMP. That means that in order to enable OpenMP on a mac, you must install a different c++ compiler. There are many options available, but the Gnu compilers are one of the most popular. To install a recent version, follow the instructions below:

1. Install Homebrew (`http://brew.sh/`):

   ```
   ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
   ```

2. Install gcc version 4.9:

   Open up the terminal and run:

   ```
   brew install gcc49
   ```

3. Confirm GCC 4.9 is installed:

   ```
   g++-4.9 --version
   ```

Once gcc has been successfully installed, tSNE can be compiled with the newly installed compiler as follows:

```
> compileTSNE(compiler = "g++-4.9", use_omp = T)
```

3 The AdjClust workflow

3.1 Step 1: Reading in the FCS files

The first step in Adjacency Clustering is to perform dimensionality reduction with the tSNE algorithm. In the future we'll make this nice and streamlined, but for now it's gonna be a bit of a hassle. To do this, we need to read the data into R so that we can process it and get it into the format we want.

Let's say we have a folder filled with FCS files:

```
> fcsFolder = "~/fcsFiles"
> ##read all the filenames into a vector
> fcsFiles = dir(fcsFolder, full.names = T)
> fcsFiles

 [1] "/Users/chris/fcsFiles/set1.poolA.1172.wk104.ns.CD8T.fcs"
 [2] "/Users/chris/fcsFiles/set1.poolA.1172.wk104.pi.CD8T.fcs"
 [3] "/Users/chris/fcsFiles/set1.poolA.1172.wk18.ns.CD8T.fcs"
 [4] "/Users/chris/fcsFiles/set1.poolA.1172.wk18.pi.CD8T.fcs"
 [5] "/Users/chris/fcsFiles/set1.poolA.1172.wk40.ns.CD8T.fcs"
 [6] "/Users/chris/fcsFiles/set1.poolA.1172.wk40.pi.CD8T.fcs"
 [7] "/Users/chris/fcsFiles/set1.poolA.1172.wk53.ns.CD8T.fcs"
 [8] "/Users/chris/fcsFiles/set1.poolA.1172.wk53.pi.CD8T.fcs"
 [9] "/Users/chris/fcsFiles/set1.poolA.ctrl1.NA.ns.CD8T.fcs"
[10] "/Users/chris/fcsFiles/set1.poolA.ctrl1.NA.pi.CD8T.fcs"
```

The first thing we need to do is read in all of these fcs files. To do this, we need to use the `flowCore` package from Bioconductor.

```
> library(flowCore)
> fcsDat = lapply(fcsFiles,function(fn){read.FCS(fn)})
```

We're then going to take each one of these FCS objects and extract the marker expression data from them:

```
> fcsMats = lapply(fcsDat, function(fcs){
+   #get marker vals
+   eset = exprs(fcs)
+
+   ##replace channel names with marker names
+   channels = fcs@parameters@data$desc
+   names(channels) = colnames(eset)
+   channels[is.na(channels)] = colnames(eset)[is.na(channels)]
+   colnames(eset) = channels
+
+   return(eset)
+ })
> fcsMats[[1]][1:10,1:6]

          Time Event_length   BC1    BC2    BC3     BC4
 [1,]  2442.604           35 4.435 18.247  7.484  95.764
 [2,]  5441.979           43 4.255 26.072  7.239 126.849
 [3,]  5826.432           40 0.000 36.553  7.302  90.253
 [4,]  8387.188           38 0.000 17.719  7.852  95.326
 [5,]  8915.117           37 0.000 20.914  7.599  60.876
 [6,]  9489.388           30 0.000 29.219  9.105  26.002
 [7,] 10560.703           27 1.199 26.466  8.894 171.672
 [8,] 11107.930           61 0.000 29.771  9.983 112.359
 [9,] 11484.792           30 0.050 23.987  8.285  89.709
[10,] 12380.052           44 1.430 41.458 10.950  83.965
```

(side note: if you're working with lots of fcs files, you're gonna be using a lot of memory here. To mitigate this problem, remove objects as you go):

```
> rm(fcsDat)
```

Note that each one of these matrices are going to have different numbers of cells, but the column names of each of them should match:

```
> ##the number of cells
> sapply(fcsMats, nrow)

 [1]  1093  1612 11135  8688  2063  1018 19310 13363   861   464

> ##the columns in each matrix
> sapply(fcsMats, colnames)[1:10,1:4]

             [,1]           [,2]           [,3]           [,4]          
Time         "Time"         "Time"         "Time"         "Time"        
Event_length "Event_length" "Event_length" "Event_length" "Event_length"
Pd104Di      "BC1"          "BC1"          "BC1"          "BC1"         
Pd106Di      "BC2"          "BC2"          "BC2"          "BC2"         
Pd108Di      "BC3"          "BC3"          "BC3"          "BC3"         
In113Di      "BC4"          "BC4"          "BC4"          "BC4"         
In115Di      "CD57"         "CD57"         "CD57"         "CD57"        
La139Di      "CD16"         "CD16"         "CD16"         "CD16"        
Ce140Di      "beads"        "beads"        "beads"        "beads"       
Pr141Di      "IL8"          "IL8"          "IL8"          "IL8"         
```

Now, optionally: if we have lots of fcs files, each with lots of cells, we may want to subsample the files to make sure the tSNE algorithm doesn't take forever. To do this, we're going to set the maximum number of cells for each matrix to 10,000 and randomly subsample any fcs files that are larger than that number:

```
> ##maximum number of cells in each matrix
> maxCell = 10000
> ##subsample each matrix
> fcsSubsample = lapply(fcsMats, function(mat){
+   #pick which rows to keep
+   i = sample(1:nrow(mat), min(nrow(mat), maxCell))
+   #return those rows
+   mat[i,]
+ })
>
```

And now we can combine them together into a single matrix:

```
> combinedEset = do.call(rbind, fcsSubsample)
> dim(combinedEset)
```

```
[1] 45799    49
```

Ideally, your final dataset should have less than 1 million cells. More than that is okay, but the processing time increases exponentially with the number of cells, and even on a fast computer 2 million cells would take more than a day to run. Additionally, the quality of the map will decrease when the number of cells is above 2.5 million, so it is recommended that you never go above that.

In this case, we have a bit under 46,000 cells, which is a very reasonable number. However, before we can run tSNE, there's three more things we need to do. First, select the markers we want to include in the map:

```
> markers = c("CD57","IL8","CD45RO","perforin","TNF","CD25","MIP1b","CD103",
+             "CD107ab","CD27","CD45RA","CXCR5","CD28","CD38","CD69","TGFb",
+             "IL2","IFNg","IntB7","CCR7","HLA-DR","CD40L","PD-1","CD127")
> combinedEset = combinedEset[,markers]
```

This dataset only contains CD8+ T cells, so we are only going to include parameters relevant to that cell type.

Second, transform the data using the arcsinh transformation:

```
> combinedEset = apply(combinedEset, 2, arcsinhTransform())
```

And third, label which file each cell came from:

```
> ##calculate number of cells from each file
> cellCounts = sapply(fcsSubsample, nrow)
> ##make vector indicating source file for each cell
> sourceFile = rep(basename(fcsFiles), times=cellCounts)
> #store this in the rownames of combinedEset
> rownames(combinedEset) = sourceFile
> combinedEset[1:10, 1:4]
```

```
                                       CD57       IL8    CD45RO  perforin
set1.poolA.1172.wk104.ns.CD8T.fcs 0.8813736 2.1722397 1.4021823 0.8813736
set1.poolA.1172.wk104.ns.CD8T.fcs 2.6807824 2.3017927 0.8813736 0.8813736
set1.poolA.1172.wk104.ns.CD8T.fcs 1.8810560 0.8813736 3.5941699 4.3706957
set1.poolA.1172.wk104.ns.CD8T.fcs 0.8813736 1.3730408 3.6591353 4.1774186
set1.poolA.1172.wk104.ns.CD8T.fcs 1.6994229 6.7494688 1.4981793 0.8813736
set1.poolA.1172.wk104.ns.CD8T.fcs 2.1672794 3.1720323 0.9677238 0.8813736
set1.poolA.1172.wk104.ns.CD8T.fcs 5.4905238 0.8813736 1.8795652 5.7061175
set1.poolA.1172.wk104.ns.CD8T.fcs 5.0829696 0.8813736 2.6696749 4.0514614
set1.poolA.1172.wk104.ns.CD8T.fcs 5.5734680 0.8813736 3.3170278 4.4722263
set1.poolA.1172.wk104.ns.CD8T.fcs 4.7924485 1.8675624 3.6933417 5.4296543
```

Now we've finally finished processing the fcs files, and we can move on to running tSNE.

3.2 Step 2: Running tSNE

Again, I hope that in the future I can streamline this a lot more, but for now running tSNE takes a bit of work. Before running this section, make sure you have tSNE correctly installed and compiled according to the instructions in section 2.

Saving data for tSNE

Once we have a matrix of cells, such as the `combinedEset` matrix that we generated in section 3.1, we can finally start using the functions in `adjClust`. First, we need to save the data in a format that tSNE understands. To do this, we will use the `saveTSNEdat` function. This takes as input the `combinedEset` matrix and the folder where the data should be saved.

```
> outFolder = "~/tsneRun"
```

```
> saveTSNEdat(combinedEset, outFolder)
```

Optionally, the `saveTSNEdat` function also takes a number of other parameters, including:

- `num_proc` - The number of processors to use. Default is the maximum number for your machine.
- `theta` and `perplexity` - Run parameters for tSNE. It is generally fine to leave theta at the default, but it can be very useful to increase the perplexity when running tSNE with large datasets. ***When running tSNE with large datasets (≥500k cells), the perplexity should be increased to 100***. This will improve the clustering of cells into similar groups.

Running tSNE

Once the data has been saved, the tSNE algorithm can be run using the `runTSNE` function. Be warned: for large datasets, this can take more than 24 hours!

```
> runTSNE(outFolder)
```

```
Step 1: Computing input similarities...
Step 2: Building tree...
|===================| 100%
Step 3: Learning embedding...
|===================| 100%
Finished!
```

Alternatively, tSNE can be run directly on the command line. The `runTSNE` function provides an option for returning just the commands for running tSNE, which can be pasted directly into a terminal to run tSNE.

```
> runTSNE(outFolder, commandOnly=T)
```

```
cd ~/tsneRun
/Library/Frameworks/R.framework/Versions/3.2/Resources/library/adjClust/tSNE/pbh_tsne
```

(NOTE: This command will be different for each system. Don't just copy the command above.)

Reading in the tSNE results

Finally, once the tSNE map has finished running, the final map can be read into R using the `loadTSNEdat` function. This returns an object of class `tmap`, which contains both the newly generated tSNE map and the original marker values, as well as most of the clustering information that we are about to generate.

```
> tmap = loadTSNEdat(outFolder)
```

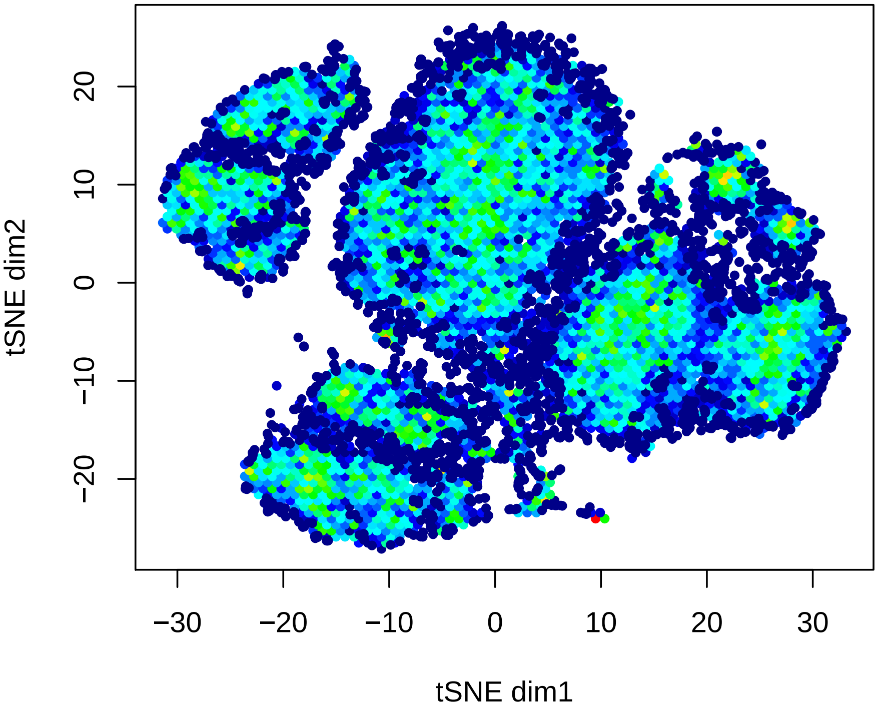
3.3 Analyzing the tSNE map

The adjClust package contains a number of functions for visualizing and analyzing a tSNE map. This section covers the basic plotting methods.

hexplot

Once the tSNE results has been loaded (via the loadTSNEdat function), the first step is to plot out the map.

```
> plot(tmap)
```



For tmap objects, the default plotting function calls hexplot, a highly customizable set of plotting functions specifically designed for visualizing high-density data. The default hexplot parameters split the plotting region up into a set of hexagonal bins, and everything in that bin is combined together into a single point, with the total number of points in each bin visualized using a color gradient. Low density bins are shown in dark blue, and higher density areas are represented using a color scale, going from cyan -> green -> yellow -> orange -> red.

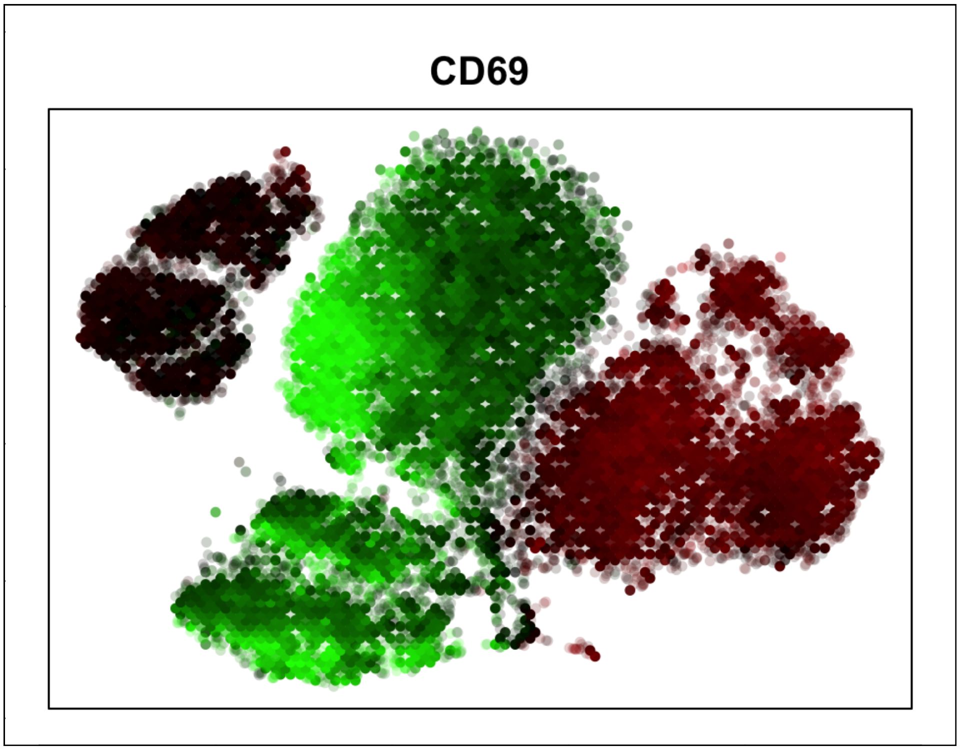
The hexplot function includes a number of options for making custom plots of the tSNE map, including parameters to control the colors, the number of bins, and how density information is displayed. Many of the most common customization options have been included in adjClust as separate functions, but it may eventually be useful to generate your own. Refer to the hexplot help file for more details.

plotMarkers

Now that we know what the map looks like, we can start to analyze where specific cell types have ended up on the map. We can do this using the `plotMarkers` function. This function uses parameter values from the original N-dimensional dataset to generate a custom-colored hexplot. Areas of the map with high expression of a specific parameter will be colored red, while areas with low expression will be colored green. Looking at these maps for each parameter individually makes it possible to find specific areas representing known cell types.

Below, we are plotting the expression of CD69, a marker of activation in T cells. We see that the map has three major groups: one large region in the center that is negative for CD69, and two smaller regions on the left and right sides of the map that are positive for CD69.

```
> plotMarkers(tmap, markerList=c("CD69"))
```



plotGroups

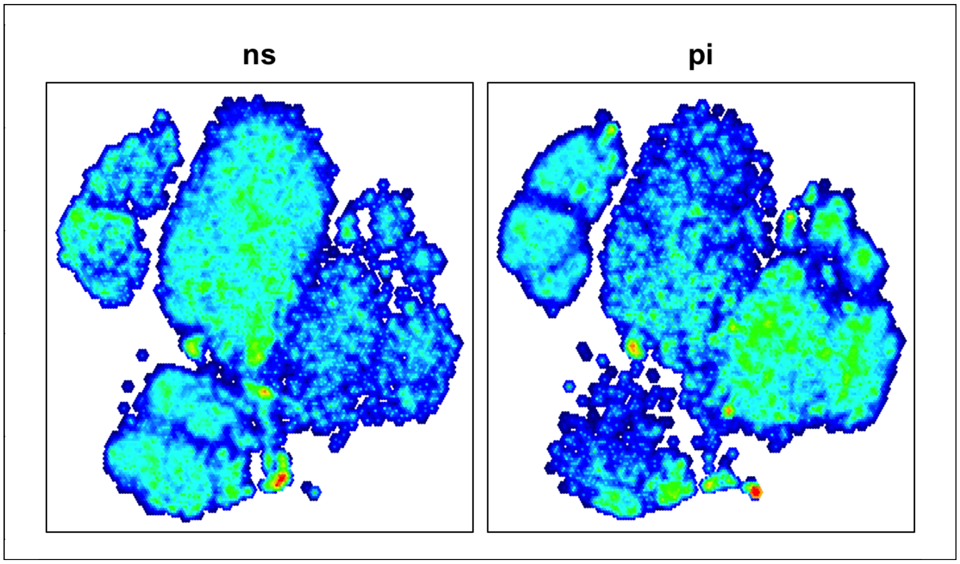
Another common task when examining tSNE maps is to compare sets of samples to each other. For example, our dataset contains ten total samples, half of which are unstimulated (ns), and the other half stimulated with PMA and Ionomycin (pi). We have information about each file stored in a data frame called fileInfo:

```
> fileInfo
```

```
                                  fn stim week donor
1  set1.poolA.1172.wk104.ns.CD8T.fcs   ns  104  1172
2  set1.poolA.1172.wk104.pi.CD8T.fcs   pi  104  1172
3   set1.poolA.1172.wk18.ns.CD8T.fcs   ns   18  1172
4   set1.poolA.1172.wk18.pi.CD8T.fcs   pi   18  1172
5   set1.poolA.1172.wk40.ns.CD8T.fcs   ns   40  1172
6   set1.poolA.1172.wk40.pi.CD8T.fcs   pi   40  1172
7   set1.poolA.1172.wk53.ns.CD8T.fcs   ns   53  1172
8   set1.poolA.1172.wk53.pi.CD8T.fcs   pi   53  1172
9    set1.poolA.ctrl1.NA.ns.CD8T.fcs   ns   NA  ctrl
10   set1.poolA.ctrl1.NA.pi.CD8T.fcs   pi   NA  ctrl
```

If we want to see where the cells from ”ns” and ”pi” samples are located in the map, we can use the `plotGroups` function. This function takes a data frame ”grp.info” which contains two columns. The first column is a list of filenames matching those in our dataset, and the second column is the group membership of that file. So if we want our two groups to be ”ns” and ”pi”, we would run `plotGroups` as follows:

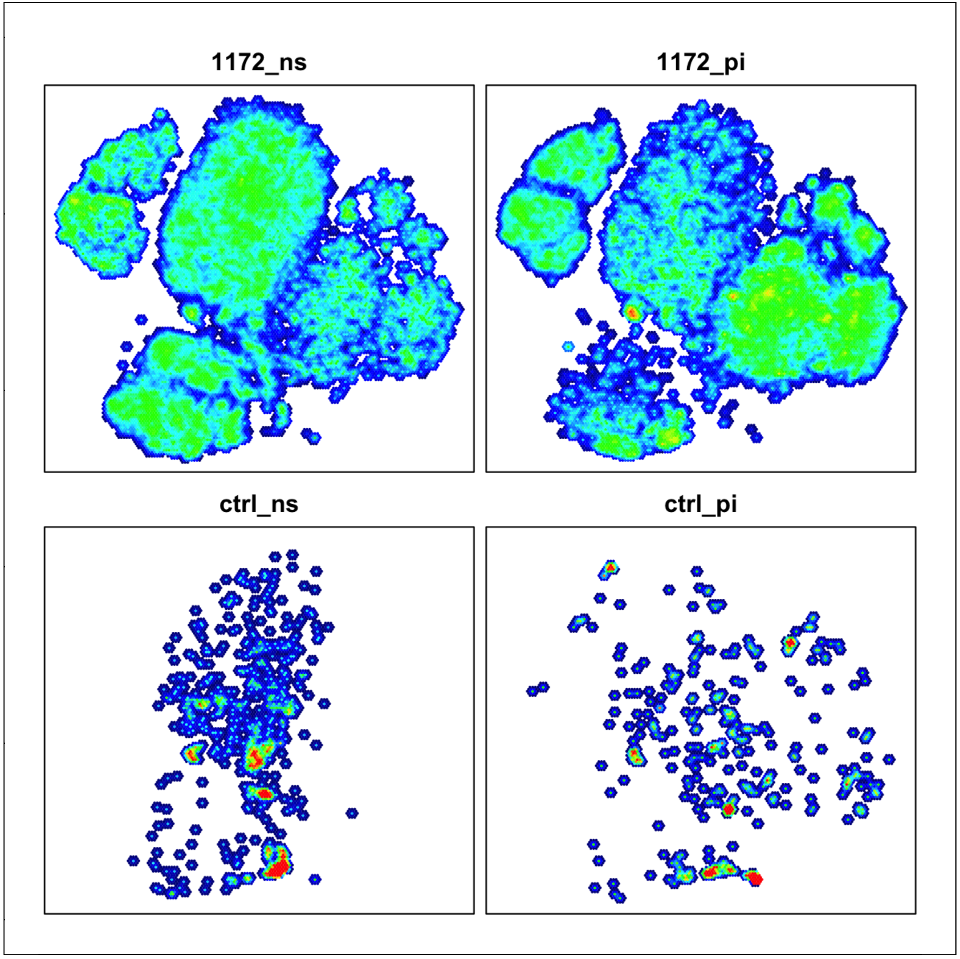
```
> grp.info = fileInfo[,c("fn","stim")]
> plotGroups(tmap, grp.info = grp.info)
```



It is also possible to define the groups using an R formula. For example, if we want to separate out the controls from the actual samples, we can run `plotGroups` as follows:

```
> plotGroups(fn~donor+stim, data = fileInfo, tmap = tmap)
```

”fn” is name of the column containing the filenames in `fileInfo`, and ”donor” & ”stim” are the columns that we want to use to separate groups.



3.4 Adjacency Clustering

The stuff I describe in the previous chapters is nice, but the Adjacency Clustering algorithm is the reason this package exists. The Adjacency Clustering method is an adapted version of the same hierarchical clustering method used in citrus. This method clusters groups of cells based off of similar marker expression, but adds the limitation that those sets of cells must be adjacent to each other in the 2-dimensional tSNE map as well. The result is a set of smooth-bounded clusters that are both biologically relevant and easy to interpret.

This chapter describes how to use the clustering algorithm, how to tweak the parameters for optimal performance, and how to determine the markers that define those clusters.

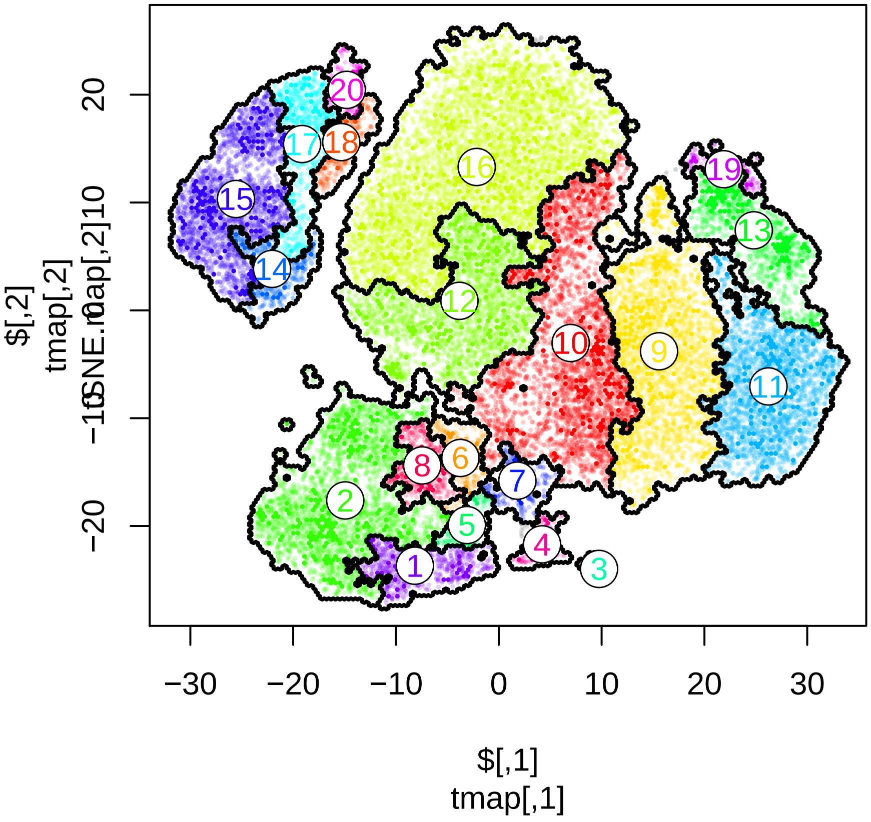
To start, we will attempt to cluster our tmap using the `cluster.tmap` function. In addition to taking the `tmap` object we loaded in using `loadTSNEdat`, this function also requires that you enter a number of clusters, `nclust`. There's no good rule for how many clusters to use, so we're going to start with a guess and change it as we see fit.

```
> tmap = cluster.tmap(tmap, nclust = 20)
```

(Note: The `cluster.tmap` function adds cluster information to our original `tmap` object, so we are overwriting the previous tmap object to save memory)

We can see how the clustering worked using the `plotClusters` function:

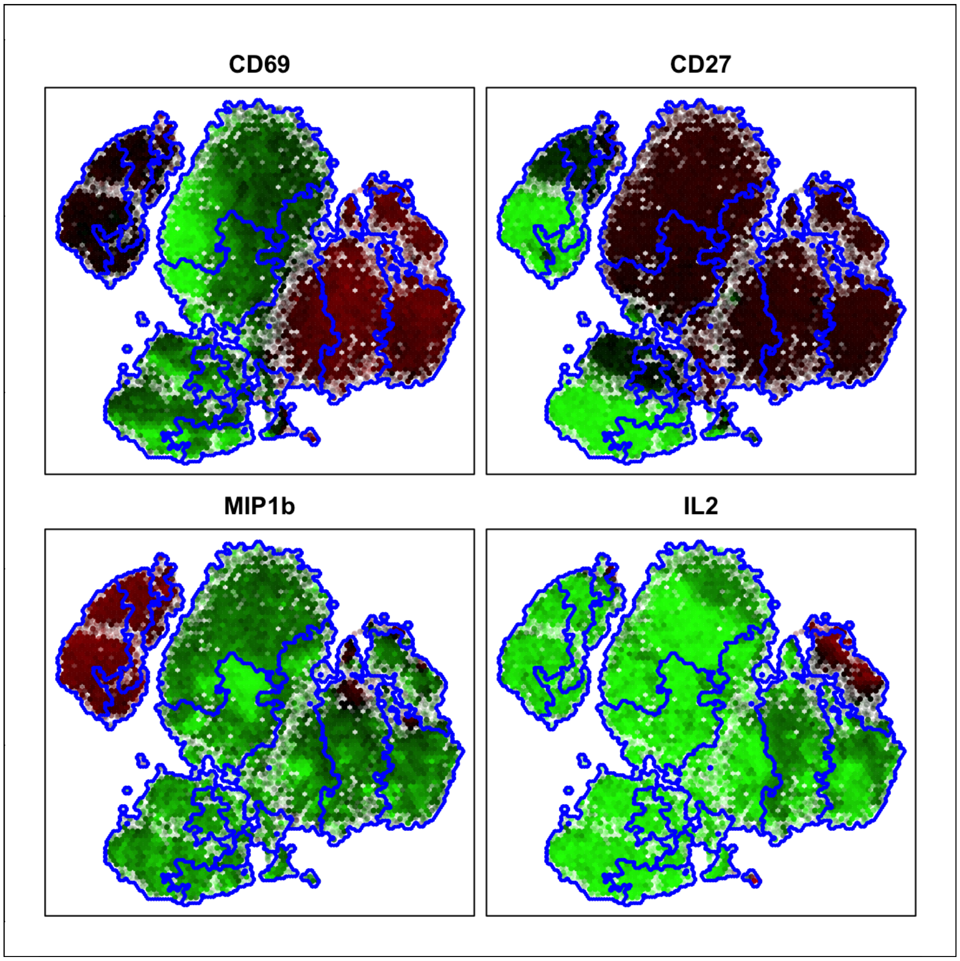
```
> plotClusters(tmap)
```



This function generates a `hexplot` with each cluster assigned a random color (if we run the function again, the colors of each cluster will change). Based on this map, we can see that the cluster edges generally do a good job at following the general topography of the tSNE map, but it's definitely not perfect.

To see how well these clusters match up against marker expression, we can again use the `plotMarkers` function.

```
> plotMarkers(tmap, markerList = c("CD69","CD27","MIP1b","IL2"))
```



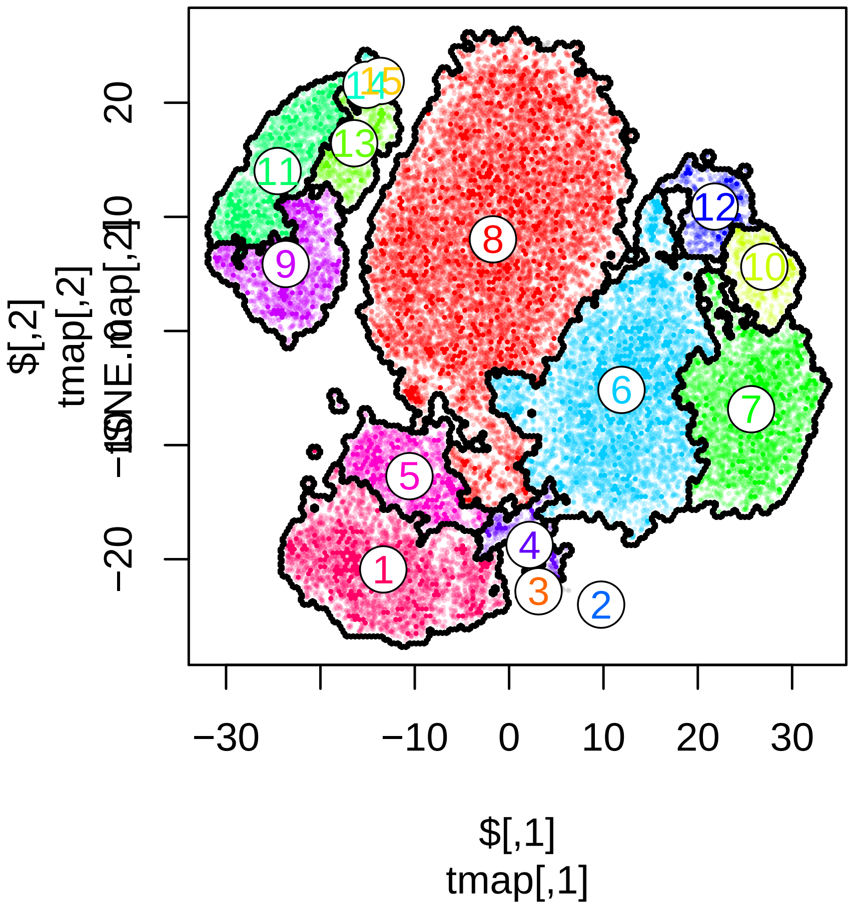
Above is just an example with four of the markers. Now that clustering has been performed, the cluster outlines have been automatically added to these plots (**Note: this will actually work for *all* the plotting methods we've used so far**). Adding the cluster outlines makes it much easier for us to see how the cells are clustering together, and whether or not the clusters accurately follow the marker expression.

Looking at these clusters, two things become apparent: First, that we chose to make too many clusters. There seems to be a large number of clusters that do not follow any marker, suggesting that boundaries have been defined based on very small changes in expression. Thus, lowering the number of clusters is probably appropriate. Second, some cluster edges seem to be following marker differences that are less important. For example, right in the middle of the map there is a cluster border separating cells with low (green to black) expression of MIP1b from those with zero (bright green) expression. While this difference certainly exists, it is most likely uninteresting from a biological perspective, as both groups would most likely be considered negative for MIP1b expression. To rectify this, we can provide weights for specific parameters, which will allow the clustering algorithm to favor differences in some parameters over others. Specifically, we'll give less weight to the MIP1b parameter while also increasing the weight of those parameters which the clusters don't adequately follow.

After some iteration and studying the clusters, I've come up with weightings for many of the markers in our map. Here's our second attempt at clustering:

```
> markerWt = c(MIP1b=0.75, CD27=1.5, IL2=1.5, IntB7=0.75,
+               CXCR5=0.5, CD25=0.5, CD107ab=0.5, CD69=1.25,
+               IFNg=1.3, CD28=0.75, IL8=1.25,perforin=1.2)
> tmap = cluster.tmap(tmap, nclust = 15, markerWt = markerWt)

> plotClusters(tmap)
```



These clusters looks pretty good. We could play with it more, but they appear to follow the general trends in the data quite well, and hopefully they are much easier to interpret than the default clusters.

3.4.1 Defining cluster phenotypes

Most of the time, playing around with the marker weights is enough to learn what markers define each cluster. However, sometimes it is useful to dig a little deeper into individual clusters to see what makes them unique. The `adjClust` package contains a method, `clusterReport`, which gives a summary of the major differences in marker expression between neighboring clusters. To generate the cluster report, run the function as follows:

```
> report = clusterReport(tmap)
```

The report is a list containing reports for every individual cluster. To view the report for a specific cluster, we can look at that entry in the list:

```
> report[[8]]

[1] "Compared to 4: Low in CD45RO; High in CD45RA;"
[2] "Compared to 5: Low in CD57, perforin; High in CD27, CD28, CCR7, CD127;"
[3] "Compared to 6: Low in CD69;"
```

This report implies that cluster 8 is CD45RA+CCR7+ (i.e. naive), and CD69 low (resting). This lines up pretty well with what we saw from the marker maps, and makes it easy to characterize the phenotype of those cells. However, some of the clusters are harder to figure out. Cluster 14, for example, is small and seems to be difficult to characterize. Even worse, the cluster report doesn't appear to tell us anything about this cluster.

```
> report[[14]]

[1] "Compared to 13:" "Compared to 15:"
```

That's because the differences in this cluster are smaller. The `clusterReport` works by only examining those markers that are most different. By default, it will only report the 5% of markers that are most differentially expressed across all clusters, but we can change this parameter to make the cutoffs more relaxed.

```
> report = clusterReport(tmap, alpha = 0.1)
> report[[14]]

[1] "Compared to 13: High in CD69;"
[2] "Compared to 15: Low in TGFb, IL2;"
[3] "Compared to total map: High in TNF;"
```

This allows us to see the smaller differences between these clusters. However, beware: now that the cutoffs are lower, we may see differences that are much less important. For example, the expression of CD69 is higher in cluster 14 than cluster 13, but looking back at the marker map in section 3.3, we see that all of the clusters in this area are high for CD69, and cluster 14 is simply the highest.

3.5 Cluster Analysis

One of the primary goals for cluster analysis is to compare the relative frequencies of a cluster between two (or more) groups of samples. We've already shown in section 3.3 how to use the plotGroups function to visually compare sets of samples. In this section, we'll detail how to make statistical comparisons of cluster membership between sample sets and how to extract the cluster data so that you can perform your own analysis outside of the adjClust package.

3.5.1 Comparing groups

The adjClust package contains a function, compareGroups, which is designed to perform a quick automated comparison of two groups of samples. This function works more or less the same as the plotGroups function that we used in section 3.3. The function takes three parameters as input, our tmap object, the grp.info data frame, and a contrast string, which describes the comparison that we want to make. We're going to use the same "grp.info" object that we created in that section for this analysis:

```
> head(grp.info)
```

```
                               fn stim
1 set1.poolA.1172.wk104.ns.CD8T.fcs   ns
2 set1.poolA.1172.wk104.pi.CD8T.fcs   pi
3  set1.poolA.1172.wk18.ns.CD8T.fcs   ns
4  set1.poolA.1172.wk18.pi.CD8T.fcs   pi
5  set1.poolA.1172.wk40.ns.CD8T.fcs   ns
6  set1.poolA.1172.wk40.pi.CD8T.fcs   pi
```

The comparison is then run as follows:

```
> cmpInfo = compareGroups(tmap, grp.info = grp.info, contrast = "pi-ns")
> head(cmpInfo)
```

```
   cluster statistic p.value   fdr
4        4         4  0.0947 0.444
6        6        21  0.0947 0.444
7        7        21  0.0947 0.444
5        5         6  0.2100 0.444
12      12        19  0.2100 0.444
10      10        18  0.2950 0.444
```

The table details the results of a two-sample test comparing the ”ns” and ”pi” groups, with info on p-values, test statistics, and fdr included for each cluster (In this case, the ”statistic” column refers to the W statistic from the Wilcoxon rank-sum test. More extreme values–i.e. close to either 0 or N1*N2–represent more significant tests, and thus will have more significant p-values. If we want to use a t-test instead, we can specify that parametric=TRUE, meaning that we should use a parametric test).

We see that although none of the p-values reach significance with an alpha of 0.05, the most significant clusters are numbers 4, 6 and 7; all of which are, based on the `clusterReport`, most likely activated cell types.

For more complicated group structures, it is possible to define groups using an R formula, just like we did for `plotGroups`. Below, we use this function to compare the ns and pi samples from a single donor, effectively dropping the controls from the analysis:

```
> cmpInfo = compareGroups(fn~donor+stim, data = fileInfo,
+                            contrast = "1172_pi-1172_ns", tmap = tmap)
> head(cmpInfo)
```

```
  cluster statistic p.value   fdr
4       4         2   0.112 0.421
5       5         2   0.112 0.421
6       6        14   0.112 0.421
7       7        14   0.112 0.421
3       3         4   0.186 0.558
1       1         4   0.312 0.642
```

Again, clusters 4, 6, and 7 come out as significant, although the p-value is slightly higher due to the reduced numbers of samples.

3.5.2 Cluster counts and percentages

There are far too many ways to analyze clustering data, and it would be silly to try and include them all in this package. However, the package provides two simple functions, `clusterCounts` and `clusterFract`, which return the cell counts and cell fractions, respectively, for each sample and cluster.

```
> cts = clusterCounts(tmap)
> cts[1:4,1:6]
```

```
                                    1  2 3   4    5    6
  set1.poolA.1172.wk104.ns.CD8T.fcs  138  0 0   9    8  257
  set1.poolA.1172.wk104.pi.CD8T.fcs  383  0 0  13   42   61
  set1.poolA.1172.wk18.ns.CD8T.fcs  1253 15 6 251 1172  201
  set1.poolA.1172.wk18.pi.CD8T.fcs   126  2 0  10   28 2523
```

```
> pct = clusterFract(tmap)
> signif(pct[1:4,1:4],3)
```

```
                                        1       2     3       4
  set1.poolA.1172.wk104.ns.CD8T.fcs 0.1260 0.00000 0e+00 0.00823
  set1.poolA.1172.wk104.pi.CD8T.fcs 0.2380 0.00000 0e+00 0.00806
  set1.poolA.1172.wk18.ns.CD8T.fcs  0.1250 0.00150 6e-04 0.02510
  set1.poolA.1172.wk18.pi.CD8T.fcs  0.0145 0.00023 0e+00 0.00115
```

These matrices can be used for whatever analyses you like, so do with them as you will.