An Introduction to SingleCellAssay

Andrew McDavid and Greg Finak

September 21, 2014

1 Philosophy

SingleCellAssayis an R/Bioconductor package for managing and analyzing Fluidigm single–cell gene expression data as well as data from other types of single–cell assays. Our goal is to support assays that have multiple *features* (genes, markers, etc) per *well* (cell, etc) in a flexible manner. Assays are assumed to be mostly *complete* in the sense that most *wells* contain measurements for all features.

1.1 Internals

A SingleCellAssay object can be manipulated as a matrix, with rows giving wells and columns giving features.

1.2 Statistical Testing

Apart from reading and storing single–cell assay data, the package also provides functionality for significance testing of differential expression using a combined binomial and normal–theory likelihood ratio test, as well as filtering of individual outlier wells. These methods are described our papers.

2 Examples

With the cursory background out of the way, we'll proceed with some examples to help understand how the package is used.

2.1 Reading Data

Data can be imported in a Fluidigm instrument-specific format (the details of which are undocumented, and likely subject-to-change) or some derived, annotated format, or in "long" (melted) format, in which each row is a measurement, so if there are N wells and M cells, then the data.frame should contain $N \times M$ rows. The use of key–value mappings makes the reading of various input formats very flexible, provided that they contain the minimal required information expected by the package.

For example, the following data set was provided in as a comma-separated value file. It has the cycle threshold (ct) recorded. Non-detected genes are recorded as NAs. For the Fluidigm/qPCR single cell expression functions to work as expected, we must use the *expression threshold*, defined as $et = c_{\max} - ct$, which is proportional to the log-expression.

Below, we load the package and the data, then compute the expression threshold from the ct, and construct a FluidigmAssay.

```
## Loading SingleCellAssay
## Creating a new generic function for 'copy' in package 'SingleCellAssay'
## Creating a generic function for 'split' from package 'base' in package 'SingleCellAssay'
## Creating a generic function for 'melt' from package 'reshape' in package 'SingleCellAssay'
## Creating a generic function for 'logLik' from package 'stats' in package 'SingleCellAssay'

#library(SingleCellAssay)
require(plyr)
data(vbeta)
colnames(vbeta)
```

```
##  [1] "Sample.ID"         "Subject.ID"        "Experiment.Number"
##  [4] "Chip.Number"       "Stim.Condition"    "Time"
##  [7] "Population"        "Number.of.Cells"   "Well"
## [10] "Gene"              "Ct"
```

```
vbeta <- computeEtFromCt(vbeta)
vbeta.fa <- FluidigmAssay(vbeta, idvars=c("Subject.ID", "Chip.Number", "Well"), primerid='Gene', measurement=
show(vbeta.fa)
```

```
## FluidigmAssay  on layer  Et
##  1  Layers;  456  wells;  75  features
##  id:  vbeta all
```

We see that the variable `vbeta` is a `data.frame` from which we construct the `FluidigmAssay` object. The `idvars` is the set of column(s) in `vbeta` that uniquely identify a well (globally), the `primerid` is a column(s) that specify the feature measured at this well. The `measurement` gives the column name containing the log-expression measurement, `ncells` contains the number of cells (or other normalizing factor) for the well. `geneid`, `cellvars`, `phenovars` all specify additional columns to be included in the `featureData`, `phenoData` and `cellData` (TODO: wellData). The output is a `FluidigmAssay` object with 456 wells and 75 features.

We can access the feature–level metadata and the cell–level metadata using the `fData` and `cData` accessors.

```
head(fData(vbeta.fa),3)
```

```
##        primerid   Gene
## B3GAT1   B3GAT1 B3GAT1
## BAX         BAX    BAX
## BCL2       BCL2   BCL2
```

```
head(cData(vbeta.fa),3)
```

```
##             Number.of.Cells            Population     wellKey Subject.ID
## Sub01 1 A01               1 CD154+VbetaResponsive Sub01 1 A01      Sub01
## Sub01 1 A02               1 CD154+VbetaResponsive Sub01 1 A02      Sub01
## Sub01 1 A03               1 CD154+VbetaResponsive Sub01 1 A03      Sub01
##             Chip.Number Well Stim.Condition Time ncells
## Sub01 1 A01           1  A01      Stim(SEB)   12      1
## Sub01 1 A02           1  A02      Stim(SEB)   12      1
## Sub01 1 A03           1  A03      Stim(SEB)   12      1
```

We see this gives us the set of genes measured in the assay, or the cell-level metadata (i.e. the number of cells measured in the well, the population this cell belongs to, the subject it came from, the chip it was run on, the well id, the stimulation it was subjected to, and the timepoint for the experiment this cell was part of). The wellKey is a hash of idvars columns, helping to ensure consistency when splitting and merging `SingleCellAssay`objects. TODO: Some of this "cell–level" information could arguably be part of the `@phenoData` slot of the object. This functionality is forthcoming but doesn't limit what can be done with the package at this stage.

2.2 Subsetting, splitting, combining

It's possible to subset `SingleCellAssay`objects by wells and features. Square brackets ("[") will index on the first index and by features on the second index. Integer and boolean and indices may be used, as well as character vectors naming the cellKey or the feature (via the primerid). There is also a `subset` method, which will evaluate its argument in the frame of the `cData`, hence will subset by wells.

```
sub1 <- vbeta.fa[1:10,]
show(sub1)
```

```
## FluidigmAssay  on layer  Et
##  1  Layers;  10  wells;  75  features
##  id:  vbeta all
```

```
sub2 <- subset(vbeta.fa, Well=='A01')
show(sub2)
```

```
## FluidigmAssay  on layer  Et
##  1  Layers;  5  wells;  75  features
##  id:  vbeta all
```

```
sub3 <- vbeta.fa[1:10,6:10]
show(sub3)
```

```
## FluidigmAssay  on layer  Et
##  1  Layers;  10  wells;  5  features
##  id:  vbeta all
```

```
cellData(sub3)
```

```
## An object of class 'AnnotatedDataFrame'
##   rowNames: Sub01 1 A01 Sub01 1 A02 ... Sub01 1 A10 (10 total)
##   varLabels: Number.of.Cells Population ... ncells (9 total)
##   varMetadata: labelDescription
```

```
featureData(sub3)
```

```
## An object of class 'AnnotatedDataFrame'
##   rowNames: CCL4 CCL5 ... CCR5 (5 total)
##   varLabels: primerid Gene
##   varMetadata: labelDescription
```

The cellData and featureData `AnnotatedDataFrames` are subset accordingly as well.

A `SingleCellAssay`may be split into a list of `SingleCellAssay`, which is known as an `SCASet`. The split method takes an argument which names the column (factor) on which to split the data. Each level of the factor will be placed in its own `SingleCellAssay`within the SCASet.

```
sp1 <- split(vbeta.fa, 'Subject.ID')
show(sp1)
```

```
## SCASet of size  2
## Samples  Sub01, Sub02
```

The splitting variable can either be a character vector naming column(s) of the `SingleCellAssay`, or may be a `factor` or `list` of `factors`.

It's possible to combine `SingleCellAssay`objects or an `SCASet` with the `combine` method.

```
combine(x=sp1[[1]],y=sp1[[2]])
```

```
## Note:  method with signature 'DataLayer#DataLayer' chosen for function 'combine',
##  target signature 'SingleCellAssay#SingleCellAssay'.
##  "SingleCellAssay#ANY" would also be valid
```

```
## FluidigmAssay  on layer  Et
##  1  Layers;  456  wells;  75  features
##  id:  Sub01
```

```
combine(sp1)
```

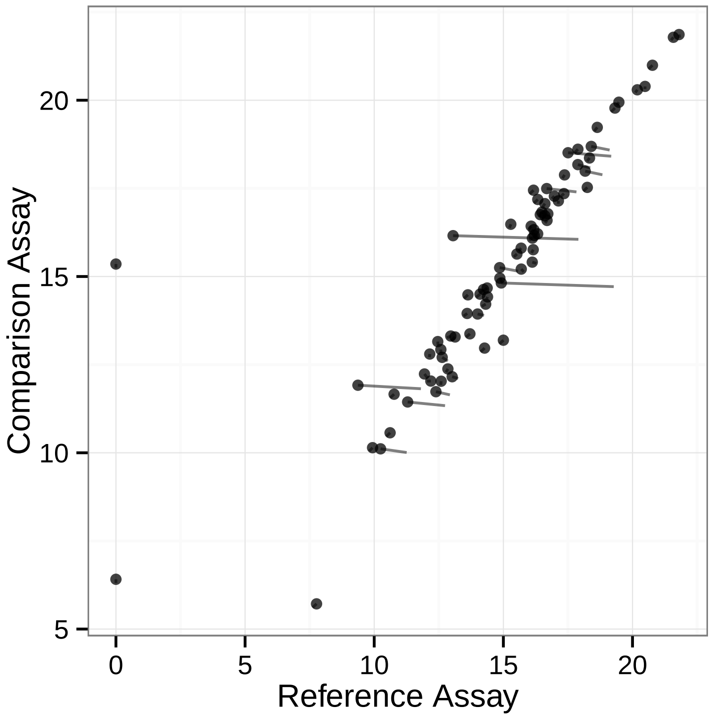
```
## FluidigmAssay  on layer  Et
##  1  Layers;  456  wells;  75  features
##  id:  Sub01
```

2.3 Filtering

We can filter and perform some significance tests on the `SingleCellAssay`. We may want to filter any wells with at least two outlier cells where the discrete and continuous parts of the signal are at least 9 standard deviations from the mean. This is a very conservative filtering criteria. We'll group the filtering by the number of cells.

We'll split the assay by the number of cells and look at the concordance plot after filtering.

```
vbeta.split<-split(vbeta.fa,"Number.of.Cells")
#see default parameters for plotSCAConcordance
plotSCAConcordance(vbeta.split[[1]],vbeta.split[[2]],filterCriteria=list(nOutlier = 1, sigmaContinuous = 9,si

## Sum of Squares before Filtering:  14.95
##  After filtering:  12.4
##  Difference:  2.54
```

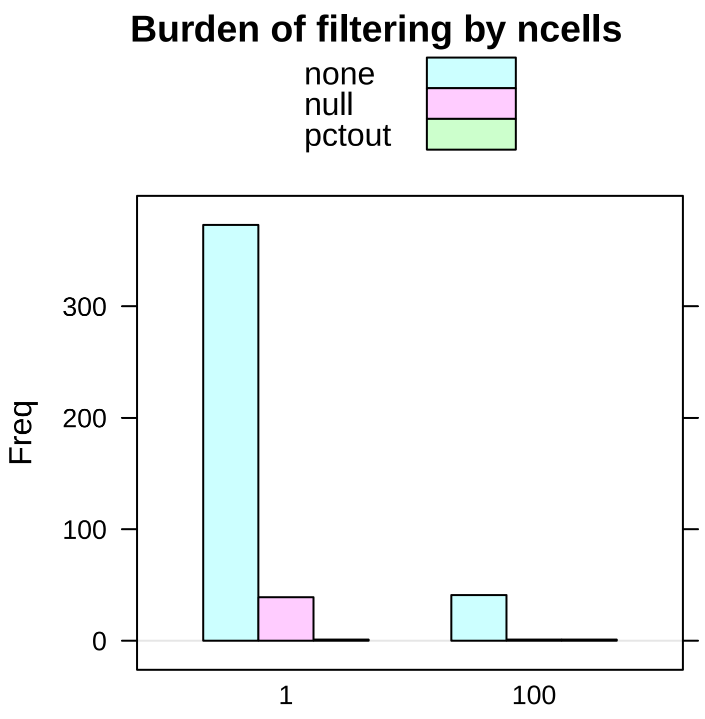


The filtering function has several other options, including whether the filter shuld be applied (thus returning a new SingleCellAssay object) or returned as a matrix of boolean values.

```
vbeta.fa

## FluidigmAssay  on layer  Et
##  1  Layers;  456  wells;  75  features
##  id:  vbeta all

## Split by 'ncells', apply to each component, then recombine
vbeta.filtered <- filter(vbeta.fa, groups='ncells')
## Returned as boolean matrix
was.filtered <- filter(vbeta.fa, apply_filter=FALSE)
## Wells filtered for being discrete outliers
head(subset(was.filtered, pctout))

##              intout null pctout
## Sub01 1 D05  FALSE TRUE   TRUE
## Sub01 1 D06  FALSE TRUE   TRUE
## Sub01 1 D07  FALSE TRUE   TRUE
## Sub01 1 D08  FALSE TRUE   TRUE
## Sub01 1 D10  FALSE TRUE   TRUE
## Sub01 1 D11  FALSE TRUE   TRUE
```

There's also some functionality for visualizing the filtering.

```
burdenOfFiltering(vbeta.fa, 'ncells', byGroup=TRUE)
```



2.4 Significance testing under the Hurdle model

There are two frameworks available in the package. The first framework `zlm` offers a full linear model to allow arbitrary comparisons and adjustment for covariates. The second framework `LRT` can be considered essentially performing t-tests (respecting the discrete/continuous nature of the data) between pairs of groups. `LRT` is subsumed by the first framework, but might be simpler for some users, so has been kept in the package.

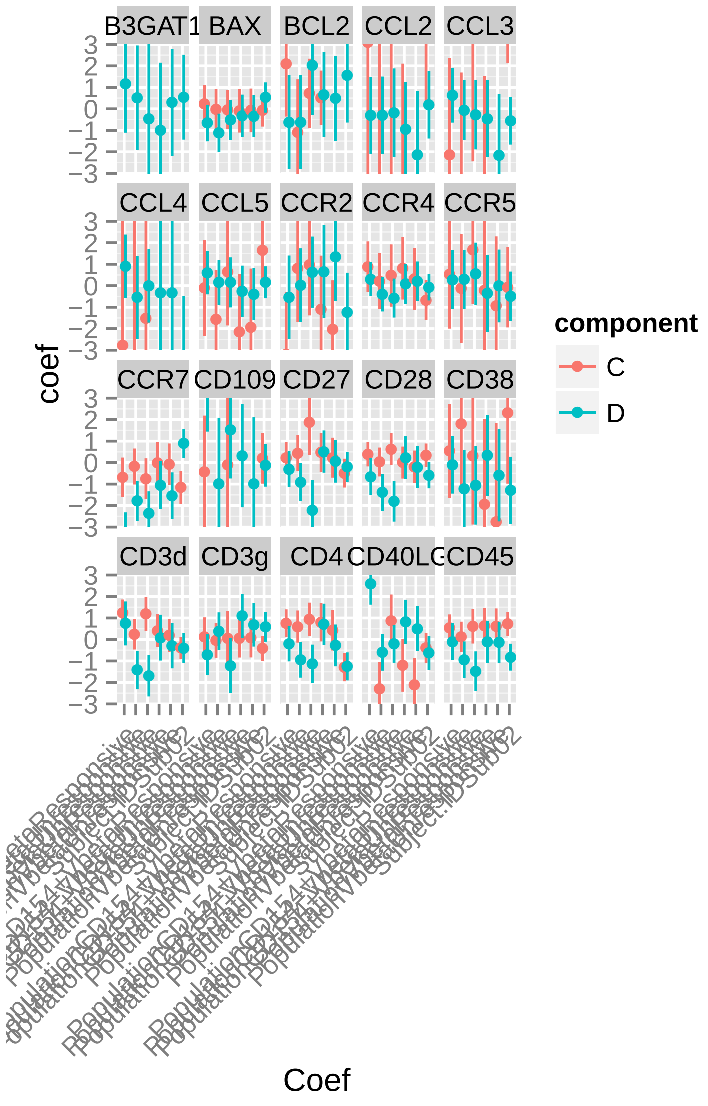
We'll describe `zlm`. Models are specified in terms of the variable used as the measure and covariates present in the `cellData` using symbolic notation, just as the `lm` function in R.

```
vbeta.1 <- subset(vbeta.fa, ncells==1)
## Consider the first 20 genes
vbeta.1 <- vbeta.1[,1:20]
layername(vbeta.1)

## [1] "Et"

head(cData(vbeta.1))

##              Number.of.Cells            Population     wellKey Subject.ID
## Sub01 1 A01                1 CD154+VbetaResponsive Sub01 1 A01      Sub01
## Sub01 1 A02                1 CD154+VbetaResponsive Sub01 1 A02      Sub01
## Sub01 1 A03                1 CD154+VbetaResponsive Sub01 1 A03      Sub01
## Sub01 1 A04                1 CD154+VbetaResponsive Sub01 1 A04      Sub01
## Sub01 1 A05               1 CD154+VbetaResponsive Sub01 1 A05      Sub01
## Sub01 1 A06               1 CD154+VbetaResponsive Sub01 1 A06      Sub01
##             Chip.Number Well Stim.Condition Time ncells
## Sub01 1 A01           1  A01      Stim(SEB)   12      1
## Sub01 1 A02           1  A02      Stim(SEB)   12      1
## Sub01 1 A03           1  A03      Stim(SEB)   12      1
## Sub01 1 A04           1  A04      Stim(SEB)   12      1
## Sub01 1 A05           1  A05      Stim(SEB)   12      1
## Sub01 1 A06           1  A06      Stim(SEB)   12      1
```

Now, for each gene, we can regress on `Et` the factors `Population` and `Subject.ID`.

In each gene, we'll fit a Hurdle model with a separate intercept for each population and subject. A an S4 object of class "ZlmFit" is returned, containing slots with the genewise coefficients, variance-covariance matrices, etc.

```
library(ggplot2)
library(reshape)
library(abind)
zlm.output <- zlm.SingleCellAssay(~ Population + Subject.ID, vbeta.1, method='bayesglm', ebayes=TRUE)
show(zlm.output)

## Fitted zlm on  20  genes and  413  cells.
##  Using  BayesGLMlike  to fit.

## returns an array of both discrete and continuous
coefAndCI <- aaply( c(C='C', D='D'), 1, function(component){
    ## coefficients for each gene
    coefs <- coef(zlm.output, which=component)
    ## standard errors for each gene
    se2 <- se.coef(zlm.output, which=component)*2
    ci.lo <- coefs-se2
    ci.hi <- coefs+se2
    abind(coef=coefs, ci.lo=ci.lo, ci.hi=ci.hi, rev.along=0)
}, .inform=TRUE)

mCoefAndCI <- setNames(melt(coefAndCI), c('component', 'gene', 'Coef', 'variable', 'value'))
CoefAndCI <- cast(mCoefAndCI, ... ~ variable, subset=Coef != '(Intercept)')
ggplot(CoefAndCI, aes(x=Coef, y=coef, ymin=ci.lo, ymax=ci.hi, col=component))+geom_pointrange(position=positi
```

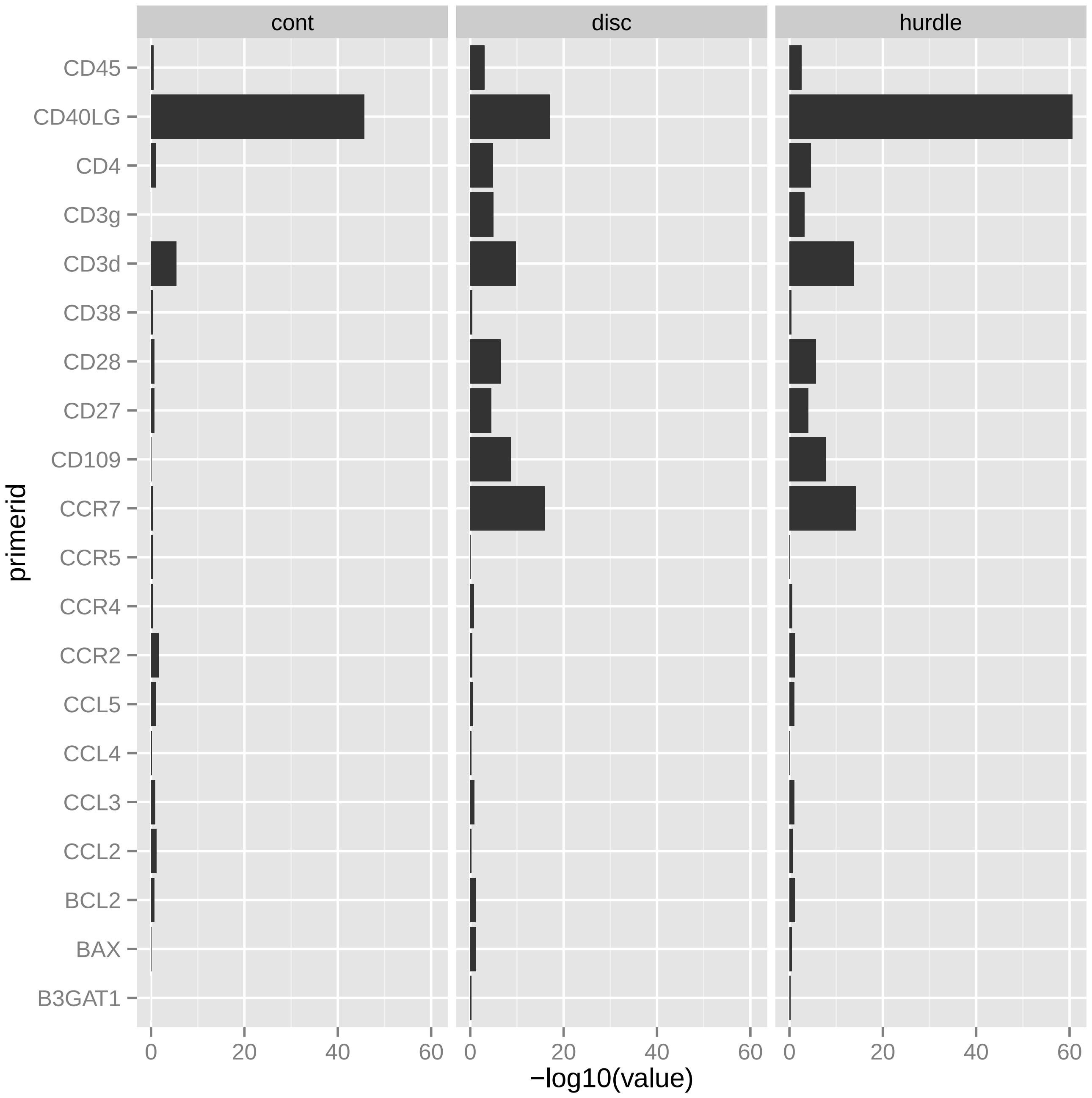


Try `showMethods(classes='ZlmFit')` to see a full list of methods.

The combined test for differences in proportion expression/average expression is found by calling a likelihood ratio test on the fitted object. An array of genes, metrics and test types is returned. We'll plot the -log10 P values by gene and test type.

```
zlm.lr <- lrTest(zlm.output, 'Population')

## Refitting on reduced model...
## .
## Done!

dimnames(zlm.lr)

## $primerid
##  [1] "B3GAT1" "BAX"    "BCL2"   "CCL2"   "CCL3"   "CCL4"   "CCL5"
##  [8] "CCR2"   "CCR4"   "CCR5"   "CCR7"   "CD109"  "CD27"   "CD28"
## [15] "CD38"   "CD3d"   "CD3g"   "CD4"    "CD40LG" "CD45"
##
## $test.type
## [1] "cont"   "disc"   "hurdle"
##
```

```
## $metric
## [1] "lambda"     "df"          "Pr(>Chisq)"

pvalue <- ggplot(melt(zlm.lr[,,'Pr(>Chisq)']), aes(x=primerid, y=-log10(value)))+geom_bar(stat='identity')+fa
print(pvalue)
```



In fact, the `zlm` framework is quite general, and has wrappers for a variety of modeling functions that accept `glm`-like arguments to be used, such as mixed models (using `lme4`) and Bayesian regression models (using `arm`). Multicore support is offered by setting `options(mc.cores=4)`, or however many cores your system has.

```
library(lme4)
lmer.output <- zlm.SingleCellAssay(~ Stim.Condition +(1|Subject.ID), vbeta.1, method='glmer')
```

2.5 Two-sample Likelihood Ratio Test

Another way to test for differential expression is available through the `LRT` function, which is analogous to two-sample T tests.

```
two.sample <- LRT(vbeta.1, 'Population', referent='CD154+VbetaResponsive')
car::some(two.sample)

##                  Population test.type primerid direction  lrstat   p.value
## 4    CD154-VbetaResponsive      comb     CCL2        -1  3.3431 1.880e-01
## 16   CD154-VbetaResponsive      comb     CD3d        -1 10.0112 6.700e-03
## 43 CD154+VbetaUnresponsive      comb     BCL2        -1 10.6098 4.967e-03
## 54 CD154+VbetaUnresponsive      comb     CD28        -1  4.9349 8.480e-02
## 55 CD154+VbetaUnresponsive      comb     CD38        -1  0.5766 7.495e-01
## 58 CD154+VbetaUnresponsive      comb      CD4         1  3.0131 2.217e-01
## 59 CD154+VbetaUnresponsive      comb   CD40LG        -1 68.6846 1.217e-15
## 63         VbetaResponsive      comb     BCL2         1  8.1617 1.689e-02
## 74         VbetaResponsive      comb     CD28         1  4.2629 1.187e-01
## 85       VbetaUnresponsive      comb     CCL3        -1 11.8960 2.611e-03
```

Here we compare each population (CD154-VbetaResponsive, CD154+VbetaUnresponsive, CD154-VbetaUnresponsive, VbetaResponsive, VbetaUnresponsive) to CD154+VbetaResponsive. The `Population` column shows which population is being compared, while `test.type` is `comb` for the combined normal theory/binomial test. Column `primerid` gives the gene being tested, `direction` shows if the comparison group mean is greater (1) or less (-1) than the referent group, and `lrtstat` and `p.value` give the test statistic and χ^2 p-value (two degrees of freedom).

Other options are whether additional information about the tests are returned (`returnall=TRUE`) and if the testing should be stratified by a character vector naming columns in `cData` containing grouping variables (`groups`).

Note that these tests have been subsumed by `zlm.SingleCellAssay` but remain in the package for user convenience.

3 Implementation Details

Here we provide some background on the implementation of the package.

There are several fundamental new object types provided by the package. `DataLayer` is the base class, which is provides an array-like object to store tabular data that might have multiple derived representations. A `SingleCellAssay` object contains a `DataLayer`, plus cell and feature data. New types of single cell assays can be incorportated by extending `SingleCellAssay`.

Different derived classes of `SingleCellAssay`require different fields to be present in the `cellData` and `featureData` These requirements are set for each class by the slots `cmap` and `fmap`, giving required columns in cell and feature data, respectively.

We have found it useful to enforce naming conventions to reduce confusion when combining data across projects, so the constructor will rename the fields the user provides to match the values specifed in `cmap` and `fmap`.

Sets of single cell assays are stored in the `SCASet` class. A constructor for SCASet is provided to construct an SCASet directly from a data frame. Alternatively, a SingleCellAssay or derived class can be `split` on an arbitray variable to produce an SCASet.

On construction of a `SingleCellAssay` object, the package tests for completeness, and will fill in the missing data (with NA) if it is not, so assays with lots of missing data can make reading marginally slower.