Generating a Splicing Index Using xmapcore

Chris Wirth & Tim Yates

August 5, 2011

Contents

1 Introduction

With our (now deprecated) package exonmap, it was possible to generate the splicing index for the probesets in one or more genes, as defined in the Affymetrix white paper "Alternative Transcript Analysis Methods for Exon Arrays".

This function does not exist for xmapcore (as it only applies to Exon Arrays, and xmapcore contains different array types), but this document describes how this si method may be implemented to perform the same function as before.

2 The code

2.1 Package Requirements

As well as xmapcore, we will require Biobase for its function pData, and genefilter for the rowtests function.

```
> library(Biobase)
> library(genefilter)
```

2.2 Group Indices

First we will need a method which, when passed an `ExpressionSet` (from the `Biobase` package), a column name for the group data and a set of arrays, will return the indices of interest:

```
> get.group.index = function( x, group.column, members ) {
+   # extract the meta-data from the eSet object
+   pd  = pData( x )
+   # Find the column of interest
+   grp = pd[, colnames( pd ) == group.column ]
+   # Return the indices of interest
+   seq_along( grp )[ is.element( grp, members ) ]
+ }
```

2.3 The `si` method itself

So now, we come to the actual `si` method. This has been lifted pretty much verbatim from `exonmap`, but commented so that each step in the algorithm is explained.

```
> si = function( x, ids, group, gps, median.gene=FALSE, median.probeset=FALSE, unlogged=TRUE ) {
+   tr1 = if( missing( group ) ) { gps[[1]] } else { get.group.index(x, group, gps[1]) }
+   tr2 = if( missing( group ) ) { gps[[2]] } else { get.group.index(x, group, gps[2]) }
+   r = gene.to.exon.probeset.expr( x, ids )
+   if( dim( r )[ 1 ] == 0 ) {
+     # No results
+     return( r )
+   }
+   # Split based on gene id
+   gene.list = split( r, r$IN1 )
+   lapply( gene.list, function( a ) {
+     a   = a[ !duplicated( a$probeset ), , drop = FALSE ]
+     idx = c( tr1, tr2 )
+     tmp = as.matrix( a[, idx, drop = FALSE ] )
+     tr1 = seq_along( tr1 )
+     tr2 = length( tr1 ) + seq_along( tr2 )
+     idx = c( tr1, tr2 )
+     # Get our gene and probeset average function dependant on params
+     avfun.g = if( median.gene     ) { median } else { mean }
+     avfun.p = if( median.probeset ) { median } else { mean }
+     # Generate the SI depending on whether the expression data was logged or not
+     tmp = if( unlogged ) { 2^tmp } else { tmp }
+     avs = apply( tmp, 2, avfun.g )
+     tmp = sweep( tmp, 2, avs, if( unlogged ) { '/' } else { '-' } )
+     s1  = apply( tmp[, tr1, drop = FALSE], 1, avfun.p )
+     s2  = apply( tmp[, tr2, drop = FALSE], 1, avfun.p )
+     si  = if( unlogged ) { log2( s1 / s2 ) } else { s1 - s2 }
+     fac = as.factor( idx %in% tr2 )
+     r   = rowttests( tmp, fac )
+     val.and.stats = r[, c( 'p.value', 'statistic' ),drop = FALSE]
+     repeated.avg  = rep( avfun.g( log2( avs[ tr1 ] ) - log2( avs[ tr2 ] ) ), length( si ) )
+     r             = cbind( si, val.and.stats, repeated.avg )
+     # Set the row an column names for the returned object
+     rownames( r ) = a$probeset
+     colnames( r ) = c( 'si', 'p.score', 't.statistic', 'gene.av' )
+     r
+   } )
+ }
```

2.4 Calling the function

For this test, we have an eSet object stored locally which contains expression data for an MCF7/MCF10 dataset around the gene 'tp53'.

```
> load( '../unitTests/HuEx-1_0.tp53.expr.RData' )
> x.rma

ExpressionSet (storageMode: lockedEnvironment)
assayData: 240 features, 6 samples
  element names: exprs
protocolData: none
phenoData
  rowNames: ex1MCF7_r1.CEL ex1MCF7_r2.CEL ... ex2MCF10A_r3.CEL (6
    total)
  varLabels: sample group
  varMetadata: labelDescription
featureData: none
experimentData: use 'experimentData(object)'
Annotation: huex10stv1
```

So, we can call our `si` method, passing this data, the gene id for 'tp53' and the fact that the first 3 columns are one sample type (mcf7) and the second 3 are the other sample.

```
> si( x.rma, symbol.to.gene( 'tp53' ), gps=list(1:3,4:6) )

$ENSG00000141510
                  si    p.score t.statistic    gene.av
3743908  0.112899062 0.22479731  1.43429922 0.02815739
3743909 -0.157154547 0.51591129 -0.71178196 0.02815739
3743912 -0.044769083 0.70715317 -0.40359763 0.02815739
3743913 -0.186660284 0.33324256 -1.09958985 0.02815739
3743915 -0.162931473 0.08576344 -2.26962208 0.02815739
3743917  0.126866306 0.17884429  1.62805226 0.02815739
3743918  0.056591953 0.32816552  1.11278250 0.02815739
3743919  0.062286225 0.46526044  0.80632965 0.02815739
3743920 -0.156910026 0.05743946 -2.64234332 0.02815739
3743922  0.048387256 0.57010143  0.61787261 0.02815739
3743923 -0.102274940 0.36081354 -1.03096096 0.02815739
3743924  0.008979375 0.93770685  0.08317721 0.02815739
3743925  0.006989562 0.95775111  0.05636913 0.02815739
3743928  0.074234272 0.89284542  0.14348489 0.02815739
3743936 -0.865412389 0.11450250 -2.01226730 0.02815739
```

As can be seen, this call returns a list, with one `data.frame` per gene. In this example, we only have passed a single gene for brevity.

3 Code in one go for copying and pasting

Here is the code for the function in a format you can copy and paste (the code above is run through R, so has annoying symbols embedded in it):

```
library(Biobase)
library(genefilter)

get.group.index = function( x, group.column, members ) {
  # extract the meta-data from the eSet object
  pd  = pData( x )
  # Find the column of interest
  grp = pd[, colnames( pd ) == group.column ]
  # Return the indices of interest
  seq_along( grp )[ is.element( grp, members ) ]
}

si = function( x, ids, group, gps, median.gene=FALSE, median.probeset=FALSE, unlogged=TRUE ) {
  tr1 = if( missing( group ) ) { gps[[1]] } else { get.group.index(x, group, gps[1]) }
  tr2 = if( missing( group ) ) { gps[[2]] } else { get.group.index(x, group, gps[2]) }
  r = gene.to.exon.probeset.expr( x, ids )
  if( dim( r )[ 1 ] == 0 ) {
    # No results
    return( r )
  }
  # Split based on gene id
  gene.list = split( r, r$IN1 )
  lapply( gene.list, function( a ) {
    a   = a[ !duplicated( a$probeset ), , drop = FALSE ]
    idx = c( tr1, tr2 )
    tmp = as.matrix( a[, idx, drop = FALSE ] )
    tr1 = seq_along( tr1 )
    tr2 = length( tr1 ) + seq_along( tr2 )
    idx = c( tr1, tr2 )
    # Get our gene and probeset average function dependant on params
    avfun.g = if( median.gene     ) { median } else { mean }
    avfun.p = if( median.probeset ) { median } else { mean }
    # Generate the SI depending on whether the expression data was logged or not
    tmp = if( unlogged ) { 2^tmp } else { tmp }
    avs = apply( tmp, 2, avfun.g )
    tmp = sweep( tmp, 2, avs, if( unlogged ) { '/' } else { '-' } )
    s1  = apply( tmp[, tr1, drop = FALSE], 1, avfun.p )
    s2  = apply( tmp[, tr2, drop = FALSE], 1, avfun.p )
    si  = if( unlogged ) { log2( s1 / s2 ) } else { s1 - s2 }
    fac = as.factor( idx %in% tr2 )
    r   = rowttests( tmp, fac )
    val.and.stats = r[, c( 'p.value', 'statistic' ),drop = FALSE]
    repeated.avg  = rep( avfun.g( log2( avs[ tr1 ] ) - log2( avs[ tr2 ] ) ), length( si ) )
    r             = cbind( si, val.and.stats, repeated.avg )
    # Set the row an column names for the returned object
    rownames( r ) = a$probeset
    colnames( r ) = c( 'si', 'p.score', 't.statistic', 'gene.av' )
    r
  } )
}
```