Introduction to mpmi

Chris Pardy

August 27, 2013

1 Using the mpmi package

The following vignette will provide a brief introduction to the `mpmi` package, showing the use of the two main functions (`cmi()` and `mmi()`) as well as explicit parallelisation of their pairwise versions(`cmi.pw()` and `mmi.pw()`).

First we load the library

```
> library(mpmi)
```

1.1 Continuous vs continuous comparisons

We demonstrate the calculation of MI and BCMI for all pairs of a group of continuous variables using a simulated dataset included in the `mpmi` package. The dataset, `mpmidata` contains a matrix of continuous data `cts` and a matrix of categorical data `disc`. The continuous data consists of 50 subjects with 100 variables following a multivariate normal distribution (note that this is done for simplicity as our approach is designed to work for a much wider class of distributions). The continuous data were simulated to have an association that decays linearly as the distance between each pair of variables' indices increases. For reference this was created as follows (note that this requires the `MASS` library to be loaded):

```
> # library(MASS)
> # mu <- 1:100
> # S <- toeplitz((100:1)/100)
> # set.seed(123456789)
> # dat <- mvrnorm(50, mu, S)
> # cts <- scale(dat)
```

The data are loaded and the `cmi()` function is then applied:

```
> data(mpmidata)
> ctsresult <- cmi(cts)
```

Below we show the structure of the results object. It is a list containing 3 matrices. For a set of continuous variables these are square symmetric matrices of a similar form to a correlation matrix.

```
> str(ctsresult)
```

```
List of 3
 $ mi     : num [1:100, 1:100] 0.863 0.759 0.737 0.684 0.673 ...
 $ bcmi   : num [1:100, 1:100] 0.88 0.772 0.748 0.694 0.682 ...
 $ zvalues: num [1:100, 1:100] 11.85 10.28 9.74 8.93 8.39 ...
```

The raw MI values:

```
> round(ctsresult$mi[1:5,1:5], 2)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] 0.86 0.76 0.74 0.68 0.67
[2,] 0.76 0.74 0.72 0.68 0.67
[3,] 0.74 0.72 0.76 0.71 0.68
[4,] 0.68 0.68 0.71 0.71 0.69
[5,] 0.67 0.67 0.68 0.69 0.73
```

Jackknife bias corrected MI values:

```
> round(ctsresult$bcmi[1:5,1:5], 2)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] 0.88 0.77 0.75 0.69 0.68
[2,] 0.77 0.75 0.74 0.69 0.68
[3,] 0.75 0.74 0.78 0.72 0.70
[4,] 0.69 0.69 0.72 0.73 0.70
[5,] 0.68 0.68 0.70 0.70 0.74
```

We can check the results against the pairwise function. In this case we calculate the MI between the first variable and itself, which estimates its entropy.

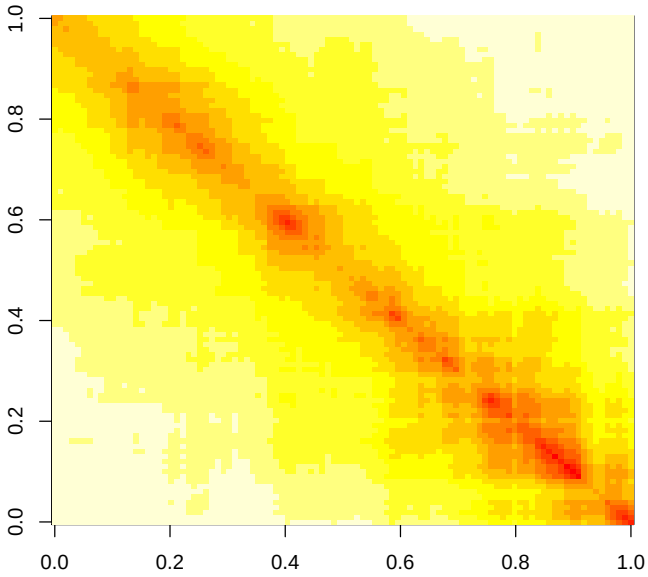
```
> cmi.pw(cts[,1], cts[,1])
```

```
$mi
[1] 0.862821

$bcmi
[1] 0.8795766

$zvalue
[1] 11.85451
```

This agrees with the results above (i.e., the `[1,1]` element of each results matrix).

We can use the `mp()` function to plot an MI (or correlation) matrix. This plots the matrix with points corresponding to the same order that they are displayed in a numerical matrix (i.e, the usual mathematical way). It is scaled so that red is the largest value and white is the smallest. When applied to the results above we can see the larger values along the diagonal of the BCMI matrix, decaying as the difference between i and j increases.

```
> mp(ctsresult$bcmi)
```



1.2 Discrete vs continuous comparisons

To demonstrate MI for mixed comparisons we generate 75 random SNP variables and create a new set of continuous data where some of the values have been shifted according to the categories.

```
> # set.seed(987654321)
> # disc <- rep(c("A", "H", "B"), ceiling(50 * 75 / 3))
> # disc <- matrix(disc, nrow = 50, ncol = 75)
> # disc <- apply(disc, 2, sample)
```

This shuffles a fairly even set of A, H, and B for each variable. We then introduce a fairly strong U-shaped shift to continuous variable i based on the value of discrete variable k, but only for cases where $i = k$.

```
> cts2 <- cts
> for (variable in 1:75)
+ {
+     for (subject in 1:50)
+     {
+         if (disc[subject, variable] == "A")
+         {
+             cts2[subject, variable] <- cts[subject, variable] - 2
+         }
+         if (disc[subject, variable] == "B")
+         {
+             cts2[subject, variable] <- cts[subject, variable] - 2
```

```
+         }
+     }
+ }
```

We run the `mmi()` function on the discrete and continuous data:

```
> mixedresult <- mmi(cts2, disc)
```

The results object for mixed comparisons have the same form as the results object for continuous comparisons. The only difference is that now instead of square symmetric matrices (for continuous data) the results are $n_c \times n_d$ matrices where n_c is the number of continuous variables and n_d is the number of discrete variables. The row index refers to continuous variables and the column index refers to discrete variables.

```
> str(mixedresult, width = 60, strict.width = "cut")
```

```
List of 3
 $ mi     : num [1:100, 1:75] 0.35235 0.09735 0.00692 0.09..
 $ bcmi   : num [1:100, 1:75] 0.3246 0.03786 -0.05755 0.00..
 $ zvalues: num [1:100, 1:75] 4.353 0.623 -2.112 0.159 0.1..
```

As before we have the raw MI values:

```
> round(mixedresult$mi[1:5,1:5], 2)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,] 0.35 0.07 0.03 0.07 0.18
[2,] 0.10 0.28 0.00 0.07 0.05
[3,] 0.01 0.04 0.32 0.02 0.05
[4,] 0.10 0.11 0.07 0.35 0.16
[5,] 0.07 0.04 0.02 0.09 0.26
```

And jackknife bias corrected MI values:

```
> round(mixedresult$bcmi[1:5,1:5], 2)
```

```
      [,1]  [,2]  [,3]  [,4]  [,5]
[1,]  0.32  0.00 -0.04  0.01  0.12
[2,]  0.04  0.25 -0.06  0.01 -0.01
[3,] -0.06 -0.03  0.29 -0.04 -0.01
[4,]  0.01  0.03 -0.04  0.32  0.08
[5,]  0.01 -0.01 -0.04  0.04  0.23
```

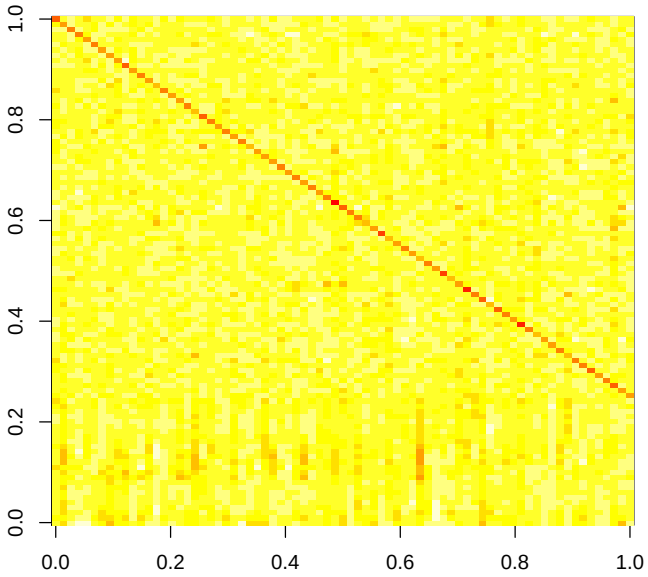
Once again we can check by using the pairwise function:

```
> mmi.pw(cts2[,1], disc[,1])
```

```
$mi
[1] 0.3523501

$bcmi
[1] 0.3245981

$zvalue
[1] 4.353068
```

We can use `mp()` to plot the BCMI values and see the strong associations we've induced for cases where $i = j$ (note that the BCMI matrix is not square):

```
> mp(mixedresult$bcmi)
```



1.3 Explicit parallelisation

The pairwise functions are provided to allow the user to explicitly control parallelisation. Here we demonstrate how to parallelise in R using the `parallel` package (based on the older `multicore`) package. As this package makes use of the POSIX `fork()` system function it can only be run on POSIX systems (i.e., Linux and MacOS; note that the implicit OpenMP parallelisation works on all three platforms, Linux, MacOS and Windows). For portability we will not actually run the code in this section, although it should work fine on Linux and Mac.

To apply this approach we need to create a function that will be run in parallel. Each application of this function will be sent to a processor core, so we must decide on 'packaging' groups of MI calculations such that this is done in an efficient way. Details are given below.

The pairwise functions `mmi.pw()`, `cmi.pw()` and `dmi.pw()` are provided to facilitate explicit parallelisation. Each of these functions calculates MI and BCMI values for comparisons between two variables with appropriate types.

1.3.1 Mixed comparisons

We first show how to parallelise the mixed comparisons as this is more straightforward than the continuous comparisons. Performance may be further im-

proved by using the R bytecode compiler. First we must load the `parallel` and `compiler` libraries:

```
> # library(parallel) # Commented for portability
> library(compiler)
```

The `mmi.pw()` function will calculate appropriate smoothing bandwidths as required. This will result in a lot of unnecessary computational repetition, so it is much faster to pre-compute the bandwidths before running the comparisons in parallel:

```
> hs <- apply(cts2, 2, dpik, level = 3L, kernel = "epanech")
```

Now we must choose how to parallelise. The simplest approach is to write a function that calculates all comparisons between continuous variables and a single discrete variable (or vice versa). This is the same approach implemented by OpenMP in `mmi()`. For each SNP i we apply the following function:

```
> fi <- function(i)
+ {
+     bcmis <- rep(NaN, 100)
+     for (j in 1:100)
+     {
+         bcmis[j] <- mmi.pw(cts2[,j], disc[,i], h = hs[j])$bcmi
+     }
+     return(bcmis)
+ }
> fi <- cmpfun(fi)
```

This returns a vector containing the BCMI values for SNP i. Modifying `fi()` to also keep the raw MI scores is straightforward.

We now use the `mcmapply()` function from the `parallel` package (which is now a part of base R). This will calculate the vectors returned by the `fi()` and bind them as columns in a matrix.

```
> # parmmi <- mcmapply(fi, 1:75)
```

We can check that the results are equal to those calculated using implicit parallelisation:

```
> # sum(abs(mixedresult$bcmi - parmmi))
```

1.3.2 Continuous comparisons

Once again we pre-compute the smoothing parameters:

```
> hs2 <- apply(cts, 2, dpik, level = 3L, kernel = "epanech")
```

For the continuous comparisons we only need to calculate each comparison once to fill the lower (or upper) triangle of the results matrix. This requires a slight modification to the range of the loop in `fi()`:

```
> fi <- function(i)
+ {
```

```
+     bcmis <- rep(NaN, 100)
+     for (j in i:100)
+     {
+         bcmis[j] <- cmi.pw(cts[,i], cts[,j], h = hs2[c(i,j)])$bcmi
+     }
+     return(bcmis)
+ }
> fi <- cmpfun(fi)
```

We smooth each of the two continuous variables by a different amount, so the `cmi.pw()` function requires two additional parameters which are input as a vector. These will be automatically calculated if not explicitly given. We run this in parallel in the same way as above:

```
> # parcmi <- mcmapply(fi, 1:100)
```

Now we check the results. The `parcmi` matrix contains an upper triangle full of missing values which would usually need to be symmetrised (the `cmi()` wrapper function takes care of this). In general, an MI matrix for continuous variables is symmetric (much like a correlation matrix) and has entropy estimates along the diagonal. So to check these results we simply need to check that the lower triangle of `parcmi` is equal to the lower triangle of `ctsresult$bcmi`. A simple approach for this check is to define a convenience function `lt()` to extract the lower triangle of a matrix, and observe that the sum of the absolute differences is computationally zero:

```
> lt <- function(x) x[lower.tri(x, diag = TRUE)]
> # sum(abs(lt(ctsresult$bcmi) - lt(parcmi)))
```

1.4 Parallelisation across multiple machines

The parallel version can be run across multiple machines in a cluster in a similar manner, by using the `snowfall` R package. This requires helper functions to be written that are identical to the `fi()` above.

1.5 A note about z-values

The functions in this package also return z-scores from the jackknife test for the hypothesis of no association (i.e., zero MI). We have found p-values and confidence intervals based on these z-scores to be highly variable and often quite wrong. Do not use these for statistical inference. The jackknife bias correction however does work quite well to reduce error in estimation of MI values (which we report as BCMI).

Since we essentially get the z-scores for free after calculating the bias correction we have decided to report them. They are useful for giving some idea of the strength of an observed association and can be considered as a heuristic transformation of the BCMI values that may aid interpretation. A permutation test is a much better choice for inference.