1 Vignette

The following vignette will provide a brief introduction to the `mpmi` package, showing the use of the two main functions (`cmi()` and `mmi()`) and well as explicit parallelisation of their pairwise versions (`cmi.pw()` and `mmi.pw()`). Note that all these functions give MI estimates, jackknife bias-corrected MI estimates and z-scores for $H_0 : MI = 0$. Faster versions that don't perform the jackknife and just return the MI are `mminjk()`, `cminjk()`, `cminjk.pw()` and `mminjk.pw()` (here the additional 'njk' in the function names stands for 'no jackknife').

First we load relevant libraries

```
library(mpmi)
library(MASS)
```

1.1 Continuous vs continuous comparisons

First we calculate MI for all pairs of a group of continuous variables. We simulate 100 variables on 50 subjects where the variables follow 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 100 variables have increasing means:

```
mu <- 1:100
```

We use a trick from the R help for `rWishart()` to guarantee a positive definite covariance matrix and simulate multivariate normal observations.

```
S <- toeplitz((100:1)/100)
set.seed(123456789)
dat <- mvrnorm(50, mu, S)
```

The `cmi()` function is then applied.

```
system.time(ctsresult <- cmi(dat))

##    user  system elapsed
##   1.165   0.000   0.687
```

This shows 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] 1.06 0.951 0.87 0.799 0.76 ...
##  $ bcmi   : num [1:100, 1:100] 1.104 0.984 0.899 0.823 0.754 ...
##  $ zvalues: num [1:100, 1:100] 9.06 8.13 7.18 6.41 6.46 ...
```

The raw MI values:

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

##      [,1] [,2] [,3] [,4] [,5]
## [1,] 1.06 0.95 0.87 0.80 0.76
## [2,] 0.95 1.11 1.02 0.93 0.86
## [3,] 0.87 1.02 1.06 0.99 0.93
## [4,] 0.80 0.93 0.99 1.04 0.98
## [5,] 0.76 0.86 0.93 0.98 1.12
```

Jackknife bias corrected MI values:

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

##      [,1] [,2] [,3] [,4] [,5]
## [1,] 1.10 0.98 0.90 0.82 0.75
## [2,] 0.98 1.15 1.06 0.96 0.86
## [3,] 0.90 1.06 1.11 1.02 0.94
## [4,] 0.82 0.96 1.02 1.08 1.01
## [5,] 0.75 0.86 0.94 1.01 1.15
```

And jackknife z-scores:

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

##      [,1] [,2] [,3] [,4]  [,5]
## [1,] 9.06 8.13 7.18 6.41  6.46
## [2,] 8.13 9.36 8.48 7.71  7.78
## [3,] 7.18 8.48 9.18 8.45  8.63
## [4,] 6.41 7.71 8.45 8.85  9.54
## [5,] 6.46 7.78 8.63 9.54 10.22
```

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(dat[, 1], dat[, 1])

## $mi
## [1] 1.06
##
## $bcmi
## [1] 1.104
##
## $zvalue
## [1] 9.061
##
```

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

1.2 Discrete vs continuous comparisons

To demonstrate MI for mixed comparisons we generate 100 random SNP values then shift some continuous variables accordingly.

```
set.seed(987654321)
disc <- rep(c("A", "H", "B"), 50 * 100)
disc <- matrix(disc, nrow = 50, ncol = 100)
disc <- apply(disc, 2, sample)
```

This just shuffles a fairly even set of A, H, and B for each variable. Now we introduce a fairly strong U-shaped shift for variables. For continuous variable i and discrete variable j, we apply a shift for comparisons such that $i = j$. We scale the continuous variables first (to mean zero with unit variance) for simplicity.

```
cts <- scale(dat)
for (i in 1:100)
{
    for (k in 1:50)
    {
        if (disc[k, i] == "A")
            cts[k, i] <- cts[k, i] - 2
        if (disc[k, i] == "B")
            cts[k, i] <- cts[k, i] - 2
    }
}
```

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

```
system.time(mixedresult <- mmi(cts, disc))

##    user  system elapsed
##   2.432   0.008   1.351
```

The results object for mixed comparison has the same form as for continuous comparisons. The only difference is instead of square symmetric matrices the results are $n_c \times n_d$ 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)

## List of 3
##  $ mi     : num [1:100, 1:100] 0.3275 0.0883 0.0373 0.0516 0.2066 ...
##  $ bcmi   : num [1:100, 1:100] 0.26616 0.00791 -0.01663 -0.01343 0.10405 ...
##  $ zvalues: num [1:100, 1:100] 3.015 0.173 -0.427 -0.357 2.074 ...
```

As before we have the raw MI values:

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

##      [,1] [,2] [,3] [,4] [,5]
## [1,] 0.33 0.11 0.08 0.16 0.06
## [2,] 0.09 0.35 0.07 0.09 0.07
## [3,] 0.04 0.19 0.42 0.04 0.06
## [4,] 0.05 0.08 0.11 0.37 0.09
## [5,] 0.21 0.15 0.13 0.27 0.36
```

Jackknife bias corrected MI values:

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

##       [,1]  [,2]  [,3]  [,4]  [,5]
## [1,]  0.27  0.02 -0.01  0.09 -0.05
## [2,]  0.01  0.29 -0.02  0.00 -0.03
## [3,] -0.02  0.14  0.39 -0.02  0.00
## [4,] -0.01  0.02  0.05  0.34  0.03
## [5,]  0.10 -0.01 -0.03  0.16  0.29
```

And jackknife z-scores (note the larger z-scores along the diagonal, corresponding to the variables for which we induced an association):

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

##       [,1]  [,2]  [,3]  [,4]  [,5]
## [1,]  3.02  0.25 -0.29  1.38 -0.83
## [2,]  0.17  2.95 -0.40  0.01 -0.46
## [3,] -0.43  1.58  4.10 -0.48 -0.08
## [4,] -0.36  0.56  1.14  3.81  0.71
## [5,]  2.07 -0.12 -0.52  2.15  3.27
```

Using the pairwise function:

```
mmi.pw(cts[, 1], disc[, 1])

## $mi
```

```
## [1] 0.3275
##
## $bcmi
## [1] 0.2662
##
## $zvalue
## [1] 3.015
##
```

1.3 Explicit parallelisation

The pairwise functions are provided to allow the researcher to explicitly control parallelisation. Here we demonstrate how to parallelise in R using the `parallel` package (based on the now-obsolete `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).

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.

1.3.1 Mixed comparisons

We first show how to parallelise the mixed comparisons as this is more straightforward than the continuous comparisons. This is also a good opportunity to try the R bytecode compiler. First we load the libraries:

```
library(parallel)
library(compiler)
```

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)
{
    zs <- rep(NaN, 100)
    for (j in 1:100)
    {
        zs[j] <- mmi.pw(cts[, j], disc[, i], h = hs[j])$z
    }
    return(zs)
}
fi <- cmpfun(fi)
```

This returns a vector containing the z-scores for SNP i. Extending this to also keep the MI scores is straightforward.

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(cts, 2, dpik, level = 3L, kernel = "epanech")
```

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

```
system.time(parmmi <- mcmapply(fi, 1:100))

##    user  system elapsed
##   0.612   0.036   2.226
```

We can check that the results are equal:

```
sum(abs(mixedresult$z - parmmi))

## [1] 0
```

1.3.2 Continuous comparisons

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)
{
    zs <- rep(NaN, 100)
    for (j in i:100)
    {
        zs[j] <- cmi.pw(dat[, i], dat[, j], h = hs2[c(i, j)])$z
    }
    return(zs)
}
fi <- cmpfun(fi)
```

Once again we pre-compute the smoothing parameters:

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

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:

```
system.time(parcmi <- mcmapply(fi, 1:100))

##    user  system elapsed
##   4.497   0.124   0.695
```

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). A simple approach for this check is to define a convenience function to extract the lower triangle of a matrix:

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

## [1] 0
```

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.