BaCoN - Tutorial

Thomas Rohde

This vignette describes how to predict functional buffering between gene pairs, using DepMap data.

Install BaCoN package

```r
if(!require(devtools)) {install.packages("devtools")}

devtools::install_github("billmannlab/BaCoN")
```

Import packages

```r
library(BaCoN)
library(data.table)
library(stringr)
library(tidyverse)
theme_set(theme_light())
```

Data preparation

This script depends on DepMap gene expression as well as Chronos scores from the 23Q2 version.

Please download them manually and place the CSV files in the "data" directory.

- Chronos scores: "CRISPRGeneEffect.csv"
- Gene expression: "OmicsExpressionProteinCodingGenesTPMLogp1.csv"

Import DepMap gene expression and fitness effect data

```r

import_depmap <- \(filepath) {
  object <- as.matrix(data.table::fread(filepath), rownames = 1)
  colnames(object) <- stringr::str_split_i(colnames(object), " \\(", 1)
  return(object)
}

gene_expression <- import_depmap(
  file.path("..", "data", "OmicsExpressionProteinCodingGenesTPMLogp1.csv"))

chronos <- import_depmap(
  file.path("..", "data", "CRISPRGeneEffect.csv"))

message(str_c(round(sum(is.na(chronos)) / length(chronos) * 100, 2),
              " percent of values are imputed."))
#> 0.94 percent of values are imputed.
```

```
# Impute missing Chronos scores as gene-wise means

for (i in 1:ncol(chronos)) {
  chronos[,i][is.na(chronos[,i])] <- mean(chronos[,i], na.rm = T)
}
```

Subset expression and fitness effect matrix

```
exp_th <- 1000
chr_th <- 1
```

The gene expression and the Chronos score matrix are subsetted to the cell lines that are represented in both datasets.

Conceptually, we expect that buffering predictions require the buffering partner to be expressed in a sufficient number of cell lines and the knockout of the buffered partner to have a certain impact on cell fitness.

We therefore apply filter criteria to reduce the gene space and remove genes with low signal.

We restrict the gene space to genes that do not show an expression (log2 TPM+1) of ≥ 3 in at least 1000 of the cell lines. Chronos genes were selected by a required essentiality level (1) shown in a minimum of 30 cell lines. This way, genes with low essentiality across most of the cell lines were removed.

Note: To keep a low runtime, we apply very strict thresholds in this vignette. In a comprehensive analysis, it is strongly recommended to cover a larger fraction of the genome, by defining less strict thresholds.

```
#To correlate fitness effect with expression data, it is necessary to equalize cell lines.
intersecting_cell_lines <- intersect(rownames(gene_expression),
                                     rownames(chronos))

expression_genes <- names(which(apply(
  gene_expression[intersecting_cell_lines,] >= 3,
  2, sum, na.rm = T) >= exp_th))
chronos_genes <- names(which(apply(
  abs(chronos[intersecting_cell_lines,]) > chr_th, 2, sum) >= 30))

gene_expression_subset <- gene_expression[intersecting_cell_lines,expression_genes]
chronos_subset <- chronos[intersecting_cell_lines,chronos_genes]
```

The resulting universe is reduced to 3423 expression genes and 1787 fitness genes.

Compute PCC correlation matrix:

```
pcc_matrix <- cor(gene_expression_subset, chronos_subset,
                  use = "pairwise.complete.obs")
```

Compute BaCoN matrix:

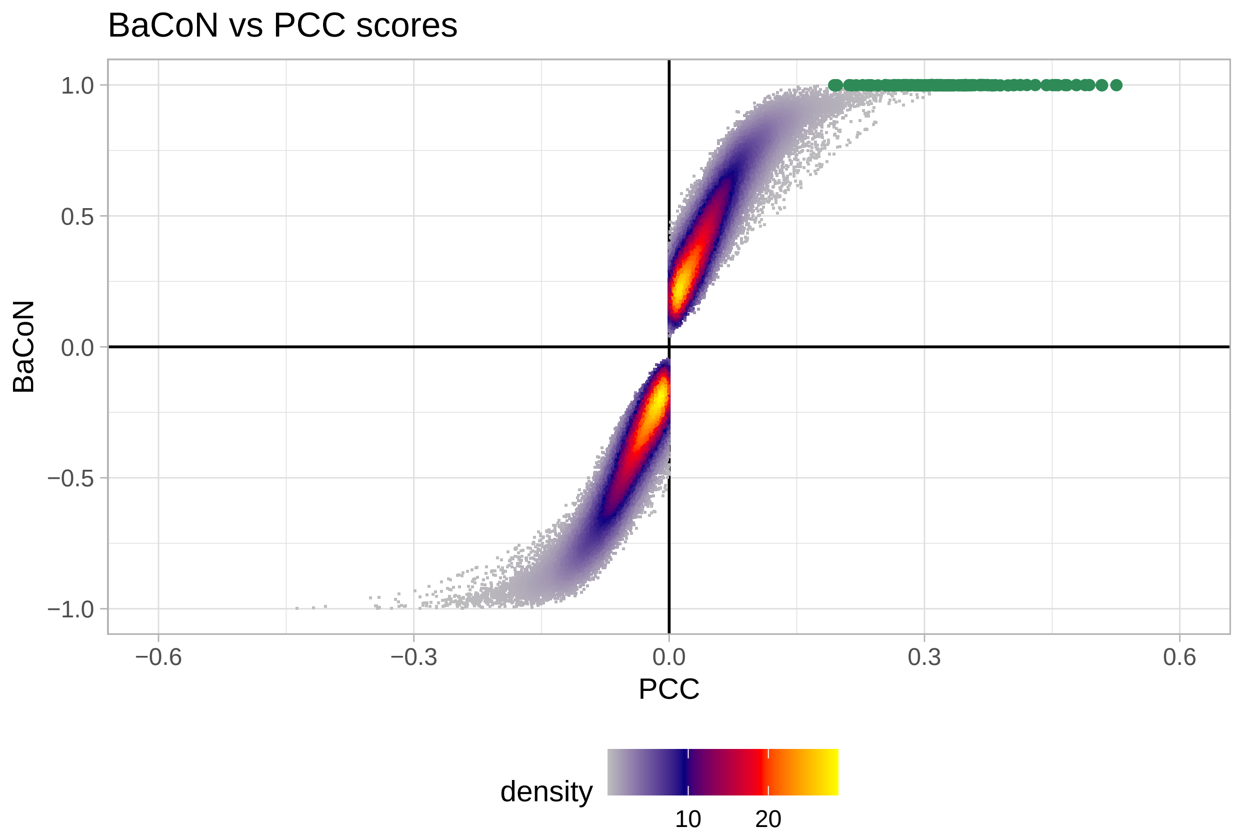
```
bacon_matrix <- BaCoN(pcc_matrix)
#>
#> Chosen threshold: none,
#>                      chosen correction factor: 0.05.
#> Ready to run (19:14:30).
#>
#> Completed after 1.84 minutes.
```

Collect top 100 predictions:

We use the lowest BaCoN score of the top 100 predictions as cutoff. This can lead to more than 100 pairs in the prediction set, as BaCoN-scored pairs are tied.

```
cutoff <- sort(bacon_matrix, decreasing = T)[100]

predictions <- data.table(which(bacon_matrix >= cutoff, arr.ind = T),
                          BaCoN = bacon_matrix[which(bacon_matrix >= cutoff)],
                          PCC = pcc_matrix[which(bacon_matrix >= cutoff)])

predictions[, `:=`(expression_gene = rownames(bacon_matrix)[row],
                   fitness_gene = colnames(bacon_matrix)[col])]

predictions <- predictions[,
                           .(expression_gene, fitness_gene, BaCoN, PCC)][
                             order(BaCoN, PCC, decreasing = T)]

predictions
#>      expression_gene fitness_gene     BaCoN       PCC
#>               <char>       <char>     <num>     <num>
#>   1:          DDX19B       DDX19A 0.9996161 0.4936106
#>   2:          BCL2L1         MCL1 0.9996161 0.4783933
#>   3:          SYNGR2       CHMP4B 0.9996161 0.4301568
#>   4:          PPP2CB       PPP2CA 0.9996161 0.4203088
#>   5:         TIMM17B      TIMM17A 0.9996161 0.4125872
#>  ---
#> 105:           SAAL1       PTPN11 0.9982726 0.2625796
#> 106:           UQCRH        HNF1B 0.9982726 0.2582083
#> 107:         ATP5F1C        MCM10 0.9982726 0.2452835
#> 108:            SQLE         BRD2 0.9982726 0.2142160
#> 109:         LAMTOR3         TBCA 0.9982726 0.1964991
```

Show score distribution and predictions



Runtime

On smaller correlation matrices (~ 5000 x 5000 genes), a BaCoN matrix can be computed in less than 5 minutes. However, in its current R implementation, the runtime of BaCoN scales exponentially in case of large matrices (> 12000 x 12000), the function can have a runtime of many hours. There are ways to reduce this problem:

1. Thresholding:

We observed that the top BaCoN predictions all showed an initial PCC z-score > 2. By limiting the computation of BaCoN scores to this small fraction of pairs, we can drastically reduce runtime while keeping the predictions of interest. The function `suggest_BaCoN_threshold` identifies the z-score threshold that ensures that BaCoN scores are computed for the top 1% of PCC values:

```
suggest_BaCoN_threshold(pcc_matrix)
#> Possible thresholds:
#>                         none     1    2    3
#> numeric threshold          0  0.06 0.13 0.19
#> values to compute [%]   100 28.69 5.07 0.87
#>
#> Suggesting threshold 2.
#> [1] "2"
```

```r
thresholded_bacon_matrix <- BaCoN(input_matrix = pcc_matrix, threshold = "2")
#>
#> Chosen threshold: 2,
#>                     chosen correction factor: 0.05.
#> Ready to run (19:16:56).
#>
#> Completed after 0.24 minutes.
```

We can show that all BaCoN scores above the z-score threshold are equal to the ones we computed before:

```r
!all(is.na(thresholded_bacon_matrix)) &
  cor(as.vector(bacon_matrix), as.vector(thresholded_bacon_matrix),
    use = "pairwise.complete.obs") == 1
#> [1] TRUE
```

2. Parallelization

A second solution to reduce runtime is the parallelization of `BaCoN`. In its default implementation, the function relies on the `apply` function. When executed with multiple threads (`n_cores = ...`), the function instead executes `future_apply` from the `future.apply` package. Using this alternative, runtime can be reduced, especially when no threshold is set.

However, a few caveats need to be mentioned:

- parallelization architectures in R are often OS-dependent, and at this point the parallelization was only tested on Windows machines
- when using multiple threads and large input matrices, RAM usage can scale drastically, which can cause the function to collapse

```r
bacon_mat_par <- BaCoN(pcc_matrix, n_cores = 8)
#> BaCoN is set up to run with 8 cores.
#>
#> Chosen threshold: none,
#>                     chosen correction factor: 0.05.
#> Ready to run (19:17:12).
#>
#> Completed after 0.86 minutes.
```

3. Thresholding and parallelization can be combined:

```r
thresholded_bacon_mat_par <- BaCoN(pcc_matrix, threshold = "2", n_cores = 8)
#> BaCoN is set up to run with 8 cores.
#>
#> Chosen threshold: 2,
#>                     chosen correction factor: 0.05.
#> Ready to run (19:18:04).
#>
#> Completed after 0.54 minutes.
```

Summary

We were able to compute `BaCoN` matrices smaller than 4000 x 4000 genes in less than 10 minutes using one core of an AMD Ryzen 9 5900X processor. Using 8 threads, matrices up to ~ 6500 x 6500 become feasible in under 10 minutes. However, the biggest runtime improvement is achieved by thresholding based on PCC z-scores. For a 12000 x 12000 correlation matrix, most reliable results were achieved using `BaCoN` with 4-8 threads.

Experimental: The BaCoN_data.table function

The exponentially increasing runtime of `BaCoN` is due to the greater number of required computations, but also the handling of very large objects. The `data.table` environment is designed to optimize speed and RAM usage when dealing with large datasets. We are experimenting with a version of `BaCoN` that converts the input correlation matrix into a `data.table`. The function is designed to minimize runtime increase caused by data handling, as well as RAM usage. The runtime improvement on large matrices compared to the default `BaCoN` function remains to be tested.

```
bacon_dt_matrix <- BaCoN_datatable(pcc_matrix)
#> BaCoN_datatable started... (19:18:37).
#> Halfway done... (19:19:03).
#> Done. (19:19:54).
```

The output scores of `BaCoN` and `BaCoN_data.table` are equal:

```
cor(data.frame(BaCoN = as.vector(bacon_matrix),
               BaCoN_parallelized = as.vector(bacon_mat_par),
               BaCoN_datatable = as.vector(bacon_dt_matrix)))
#>                    BaCoN BaCoN_parallelized BaCoN_datatable
#> BaCoN                  1                  1               1
#> BaCoN_parallelized     1                  1               1
#> BaCoN_datatable        1                  1               1
```