Demo: Feature Extraction for Data Integration

This notebook implements a simplified version of Block Randomized Adaptive Iterative Lasso ("B-RAIL") from (Baker et al. 2020)[1]. We'll start with a regression implementation on continuous data. Then we'll apply the method for graphical model estimation on the NCI-60 dataset from `omicade4`[2]. We've made two important simplifications, relative to the original paper,

- We're only working with Gaussian likelihoods. In principle, we should be able to pass in different arguments to `glmnet`, but we haven't tested this.
- We've ignored the domain correction term from equation (3.2). This would require a more intricate Fisher information calculation, and the main idea of the algorithm can be communicated even without this term.

```
library(tidyverse)
library(glmnet)
library(glue)
library(omicade4) # BiocManager::install("omicade4")
source("brail.R")
theme_set(theme_bw())
set.seed(20230320)
```

0.1 Data Generation

Let's generate a toy dataset to test our implementation with. We'll have two blocks:

- Continuous block: i.i.d. standard normals.
- Binary block: 0/1 draws from $\text{Bern}(0.5)$.

Note that we've scaled all columns to mean zero and variance 1. Each data source is stored in a separate element of the list `x`.

```
n <- 500
p <- c(300, 300)
x <- list(
  rmat(rnorm)(n, p[1]),
  rmat(rbinom)(n, p[2], 1, 0.5)
) |>
  map(scale)
```

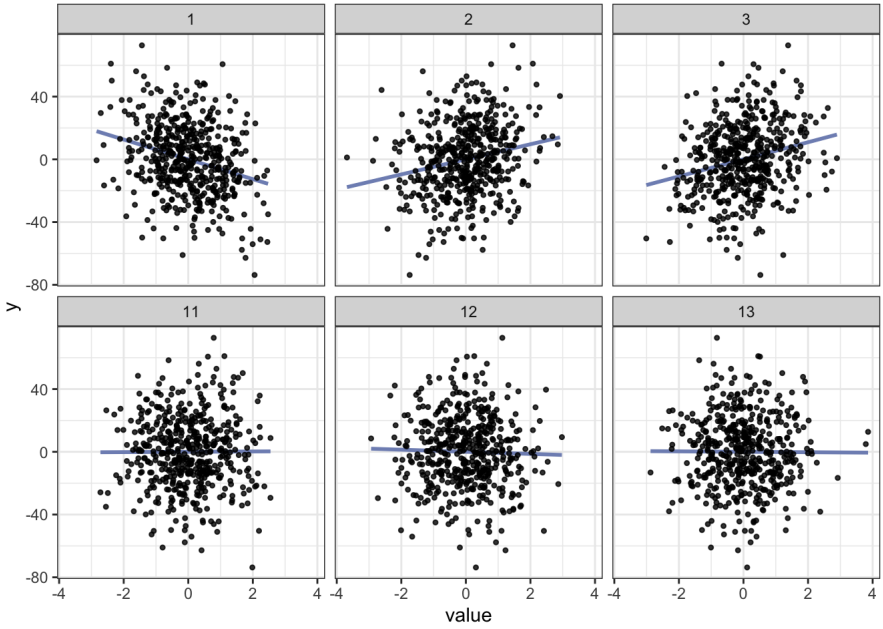
We generate a response using a linear model that ignores the difference in X domains,

$$y_i = \sum_k x_{ki}^T \beta_k + \epsilon_i.$$

for $i = 1, \ldots, n$. The first 10 entries of each β_k is drawn randomly from $\{-5, 5\}$, and the remaining entries are all 0. Don't be mystified by the `Reduce` call below: it just sums (coordinatewise) the vectors within the list defined by the second argument. The following figure gives examples of covariates with (top row) and without (bottom row) any real association with y.

```
p_signal <- c(10, 10)
signal_strength <- 5
```

```
beta <- map2(p, p_signal, ~ c(rep(signal_strength, .y),
         rep(0, .x - .y))) |>
  map(flip_sign)
y <- Reduce("+", map2(x, beta, ~ rnorm(n, mean = .x %*%
         .y, sd = 0.1)))
```



0.2 Regression Model

The B-RAIL algorithm runs the lasso sequentially across data domains, updating blocks $\hat{\beta}_k$ one at a time. There are two modifications, compared to naive sequential updating,

- The regularization parameter λ is adapted to each block.
- We only update coefficients $\hat{beta}_{kj}$ that pass a stability selection step.

The paper argues that these modifications are important, but it would be interesting to do an ablation study in our toy setting. The block below encapsulates the main logic of the two steps above. Perhaps the trickiest part of this implementation is the use of the `penalty.factor` argument. It's not often used, but it's a built-in `glmnet` option that allows for different regularization parameters for each feature – exactly what we need for adaptive regularization.

```
#' Logic for a single iteration of of B-RAIL
#'
#' @param x [list] The covariates. Each list element is
        assumed taken from a
#'   different data domain.
#' @param y [vector] The response. We're assuming its a
        continuous vector.
#' @param beta_hat [list] A list of the current estimate
        of the regression
#'   coefficients across each data domain.
#' @param k [int] Which domain's coefficient should we
        update?
#' @param B [int] Number of iterations of stability
        selection
#' @param tau [numeric] The stability selection threshold.
        A feature needs to
#'   appear in at least this many bootstraps in order to
        be selected.
#' @param epsilon [numeric] The ridge regression
        regularization penalty.
update_beta <- function(x, y, beta_hat, k = 1, B = 50, tau
        = 0.8, epsilon = 0.01) {

  # Residuals after fitting all other sources
  y_hat <- Reduce("+", map2(x[-k], beta_hat[-k], ~ .x %*%
        .y))
  r <- y - y_hat

  # Run many lasso models using different samples and
        regularizations
```

```
    indices <- sample(length(y), replace = TRUE)
    gammas <- runif(ncol(x[[k]]), 0.5, 1.5)
    lambda <- regularization(beta_hat[[k]], ncol(x[[k]]),
        length(y), x[[k]])
    fit <- glmnet(      x[[k]][indices, ],
      r[indices],       penalty.factor = gammas * lambda,
      lambda = rep(1, ncol(x[[k]])),      nlambda = 1
    )
    stability_betas[b, ] <- coef(fit)[-1, 1]  }

  # get the coefficients that are selected many times
  # if |s_hat| < 2, we randomly add two links - glmnet
        fails otherwise
  s_hat <- colMeans(stability_betas > 0) > tau
  if (sum(s_hat) < 2) {
    s_hat[sample(length(s_hat), 2)] <- TRUE   }

  beta_k <- coef(glmnet(x[[k]][, s_hat], r, alpha = 0,
        lambda = epsilon))[-1, 1]
  beta_hat[[k]][s_hat] <- beta_k  beta_hat}
```

This code called a function called `regularization` to update λ for each block (and iteration). I've cheated and ignored the Θ^{t-1} calculation, because keeping track of the data domains (continuous, count, binary, ...) for each source would require a bit of additional data management.

```
#' Compute the largest eigenvalue
#'
#' For very large matrices, the irlba package would be
        faster
max_eig <- function(x) {
  eigen(t(x) %*% x)$values[1]
}

#' Adaptive regularization, equation (3.2)
regularization <- function(beta_k, pk, n, xk) {
  # domain_correction <- max_eig(xk) / max_eig(theta) #
        ommitted - requires likelihood
  if (all(beta_k == 0)) {
```

```
          signal_correction <- sqrt(sum(beta_k ^ 2) / n)
  lasso_penalty <- sqrt(log(pk) / n * sum(beta_k != 0))
  signal_correction * lasso_penalty}
```

Given the `update_beta` function, it's easy to write the overall B-RAIL algorithm. We just cycle through blocks for however many iterations are deemed necessary. In the paper, they stop once $\hat{\beta}$ converges; here, we'll stop after a fixed number of iterations (this lets us test the code without having to wait so long).

```
#' Wrapper for B-RAIL regression algorithm
#'
#' @param x [list] The covariates. Each list element is
        assumed taken from a
#'   different data domain.
#' @param y [vector] The response. We're assuming its a
        continuous vector.
b_rail <- function(x, y, n_iter = 5, ...) {
  beta_hat <- map(x, ~ rep(0, ncol(.)))
  for (iter in seq_len(n_iter)) {
    message(glue("Running iteration {iter}..."))

    # cyclically update beta_hat, one block at a time
    for (k in seq_along(x)) {
      beta_hat <- update_beta(x, y, beta_hat, k, ...)
    }
  }

  beta_hat
}
```

The block below applies the algorithm to our simulated data and then computes a few metrics. Consistent with the observations from the paper, we have higher power for coefficients in the continuous domain. Of course, this is only one draw – we'd need to average estimates over many independent runs to draw more robust conclusions.

```
beta_hat <- b_rail(x, y)
fdps(beta, beta_hat)
```

```
## [1] 0.375 0.000
```

```
power(beta, beta_hat)
```

```
## [1] 0.5 0.4
```

0.3 Graphical Model Estimation

The real motivating example of this method comes from graphical model estimation. These methods are often used to try to understand the conditional independence relationships in a complex join distribution. In the multi-omics context, this is used for network inference across a collection of molecules. The interpretation hinges on the "guilt-by-association" principle: If two genomic features are always active together, then we usually assume they belong to the same functional pathway[3].

The mechanics of the graphical model estimation is: Loop over each column, treating it as the outcome in a regression model. Draw a directed edge between j and j' is the regression onto feature j had a nonzero coefficient coming from j'. After finishing the loop, draw an undirected edge if either (or both, depending on the merging criteria) link was found. This logic is implemented in the block below. The outer loop iterates across domains, and the inner loop iterates over features within a single domain.

```
        i <- 1   for (k in seq_along(x)) {
    pk <- ncol(x[[k]])    for (j in seq_len(pk)) {
      message(glue("Fitting column {j}/{pk} in source
        {k}/{length(x)}..."))

      # fit b-rail of current column onto all the rest
      y <- x[[k]][, j]      x_ <- x
      x_[[k]] <- x_[[k]][, -j]
      beta_hat <- b_rail(x_, y, ...)
      # get the column names to create the edgelist
      y_name <- str_c(names(x)[k], "_", colnames(x[[k]])
        [j])
      x_names <- map(x_, colnames)
      edges[[i]] <- map2_dfr(        beta_hat,
        x_names,
        ~ data.frame(to = .y[.x != 0], weight = .x[.x !=
        0], from = y_name),
        .id = "table"        )               i <- i + 1
    }     }               do.call(rbind, edges)}
```

We can apply this procedure to the `NCI60` arrays dataset. So that everything runs quickly, we'll subset to just the first 20 features.

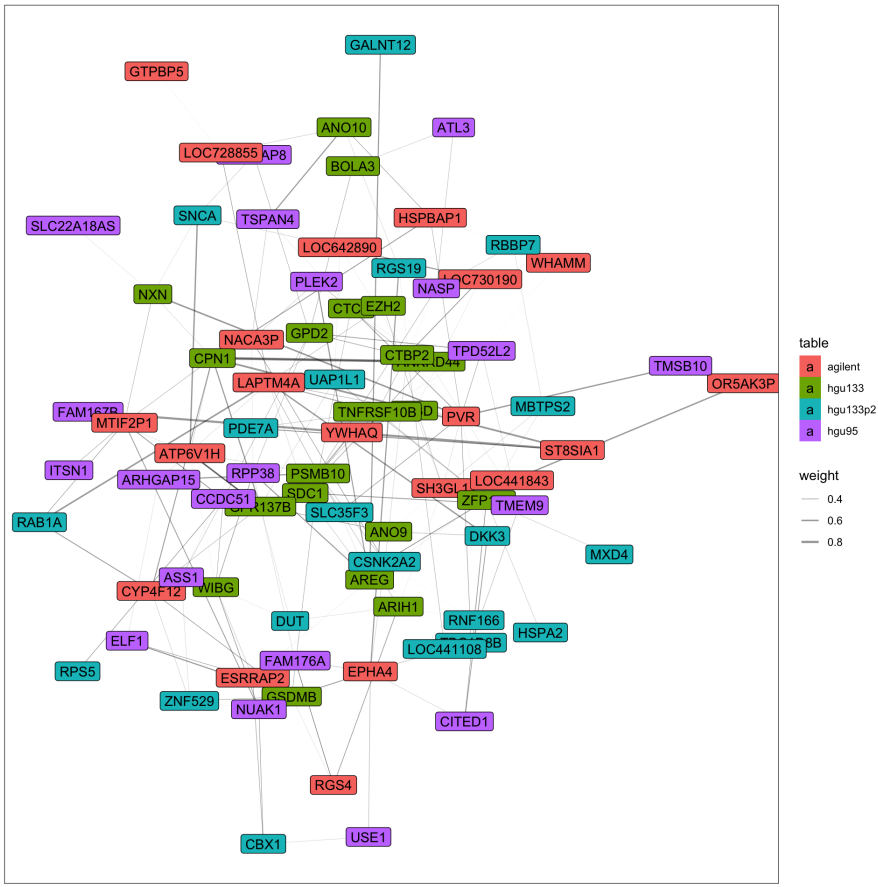
```
data(NCI60_4arrays)
x <- NCI60_4arrays |>
   map(\(u) scale(t(u[1:20, ]))) # only use first 20
        columns

edges <- b_rail_graph(x, tau = 0.8, n_iter = 2, B = 20)
```

In any real-world application, it's not enough to stop with the model coefficients – we need to be able to communicate the results clearly. We'll apply the `ggraph` package to visualize the estimated network. Our edge weights are given by the coefficient magnitudes $\left|\hat{\beta}_k\right|$ associated with each outcome feature.

```
        edges_ <- edges |>  mutate(
    to = str_c(table, "_", to),    raw_weight = weight,
    weight = abs(weight)  ) |>    select(-table) |>
  distinct()
G <- tbl_graph(edges = edges_, directed = FALSE) |>
  mutate(    table = str_extract(name, "[A-z0-9]+_"),
    table = str_remove(table, "_"),
    short_name = str_remove(name, "[A-z0-9]+_")  )
```

We first visualize the network as a node-link diagram. Each data domain is given a different color. As promised by the authors, we have a fair number of strong edges between domains.

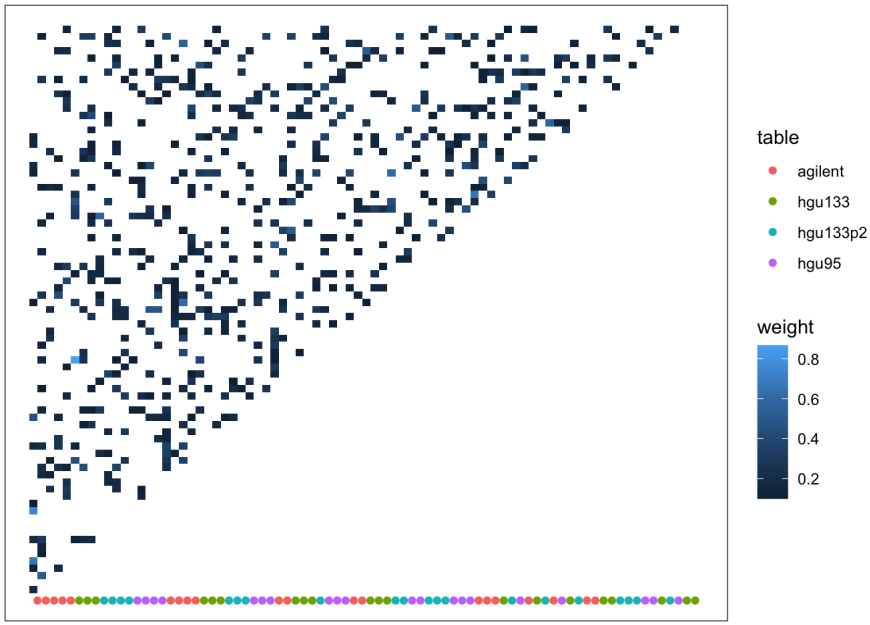
```
G %E>%
  filter(weight > 0.25) |>
  ggraph("kk") +
  geom_edge_link(aes(width = weight), alpha = 0.4) +
  scale_edge_width(range = c(0.01, 1)) +
  scale_edge_alpha(range = c(0.01, .2)) +
  geom_node_label(aes(label = short_name, fill = table))
```



Note that, though node-link diagrams are easily understood by many audiences, they don't scale well to larger graphs. In this setting, it is worth drawing an adjacency matrix. This would be a bit more useful if the matrix was interactive. As it is, we can't tell which two features any given edge corresponds to.

```
G %E>%
  filter(weight > 0.1) |>
  ggraph("matrix") +
```

```
  geom_edge_tile(aes(fill = weight)) +
  geom_node_point(aes(col = table), y = 1)
```

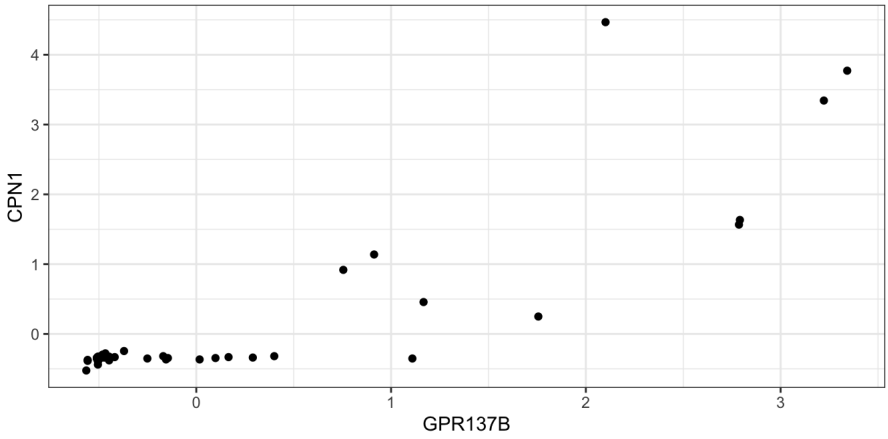


Let's confirm that B-RAIL is putting edges between correlated features. This marginal correlation is no guarantee that there is any partial correlation, but we would be concerned if we saw strong weights between features that seemed independent.

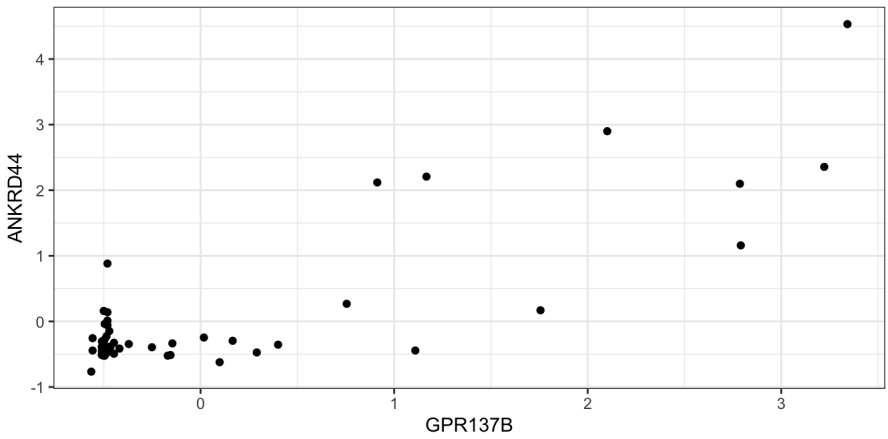
```
edges_ %>%
  arrange(-weight) |>
  head(4)
```

```
##                 to    weight            from raw_weight
## 1     hgu133_CPN1 0.8669381  hgu133_ANKRD44  0.8669381
## 2 agilent_ST8SIA1 0.7438074   hgu95_FAM167B  0.7438074
## 3     hgu133_CPN1 0.6569880 agilent_ST8SIA1  0.6569880
## 4 agilent_ATP6V1H 0.6433449  hgu133_GPR137B  0.6433449
```

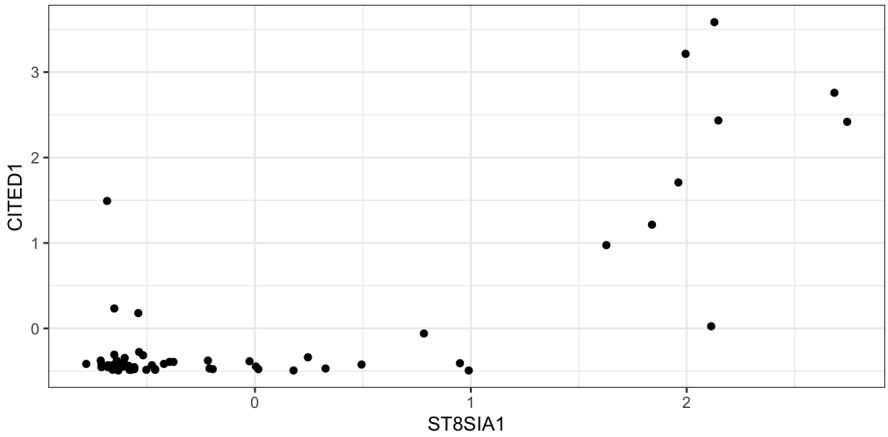
```
x_df <- x |>
  map(~ data.frame(.) |> rownames_to_column("sample")) |>
  map_dfr(~ pivot_longer(., -sample), .id = "table")

scatter(x_df, c("GPR137B", "CPN1"), c("hgu133", "hgu133"))
```




```
scatter(x_df, c("GPR137B", "ANKRD44"), c("hgu133",
        "hgu133"))
```



```
scatter(x_df, c("ST8SIA1", "CITED1"), c("agilent",
        "hgu95"))
```



Footnotes:

1. https://doi.org/10.1214/20-AOAS1389↩

2. https://bioconductor.org/packages/release/bioc/html/omicade4.html↩

3. True, it isn't causation. I wonder whether anyone has tried an HCI-style study, to see what it is about these graphs that actually aids scientific reasoning…↩