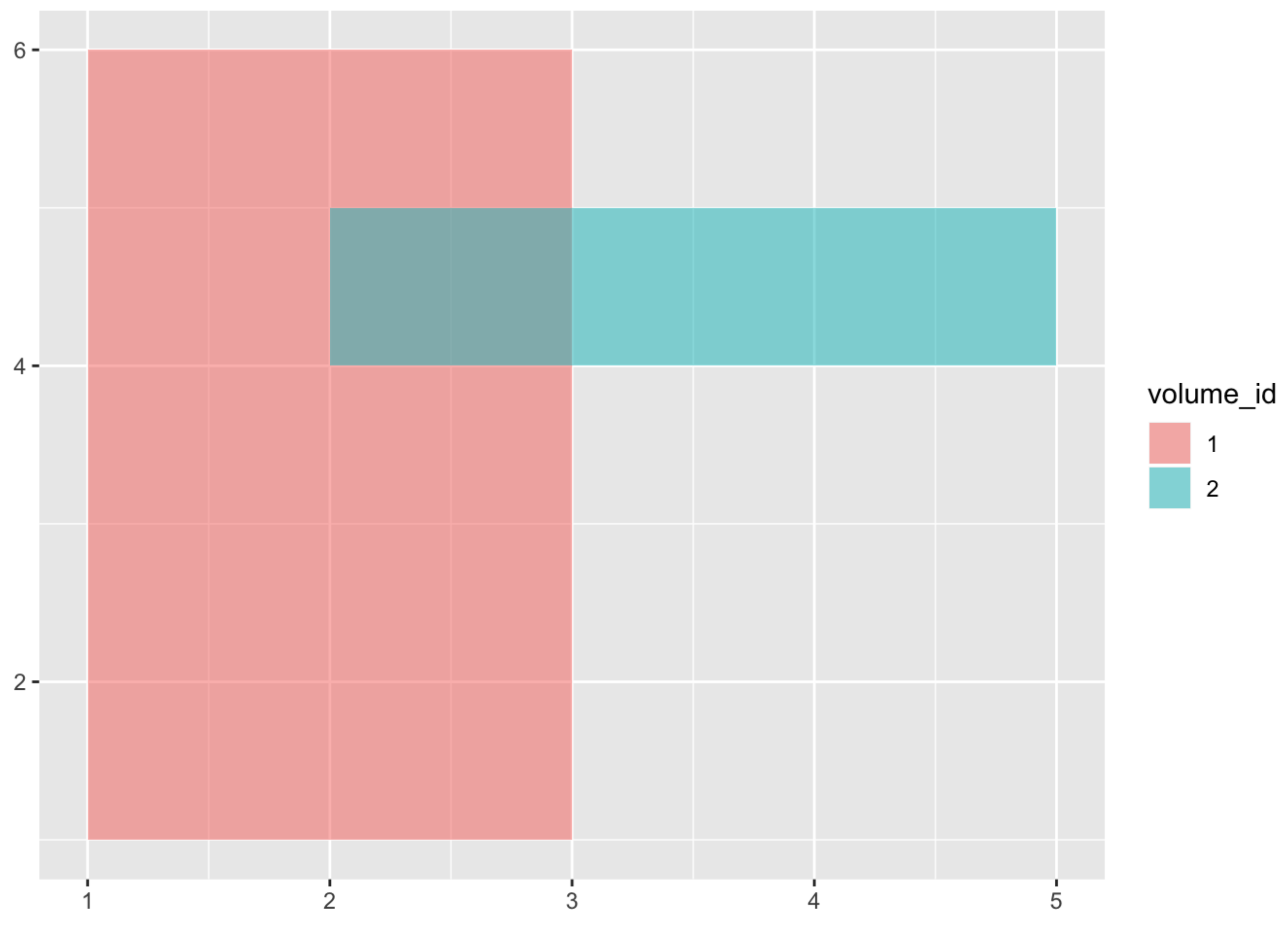
The Overlapped Hyperrectangle Problem

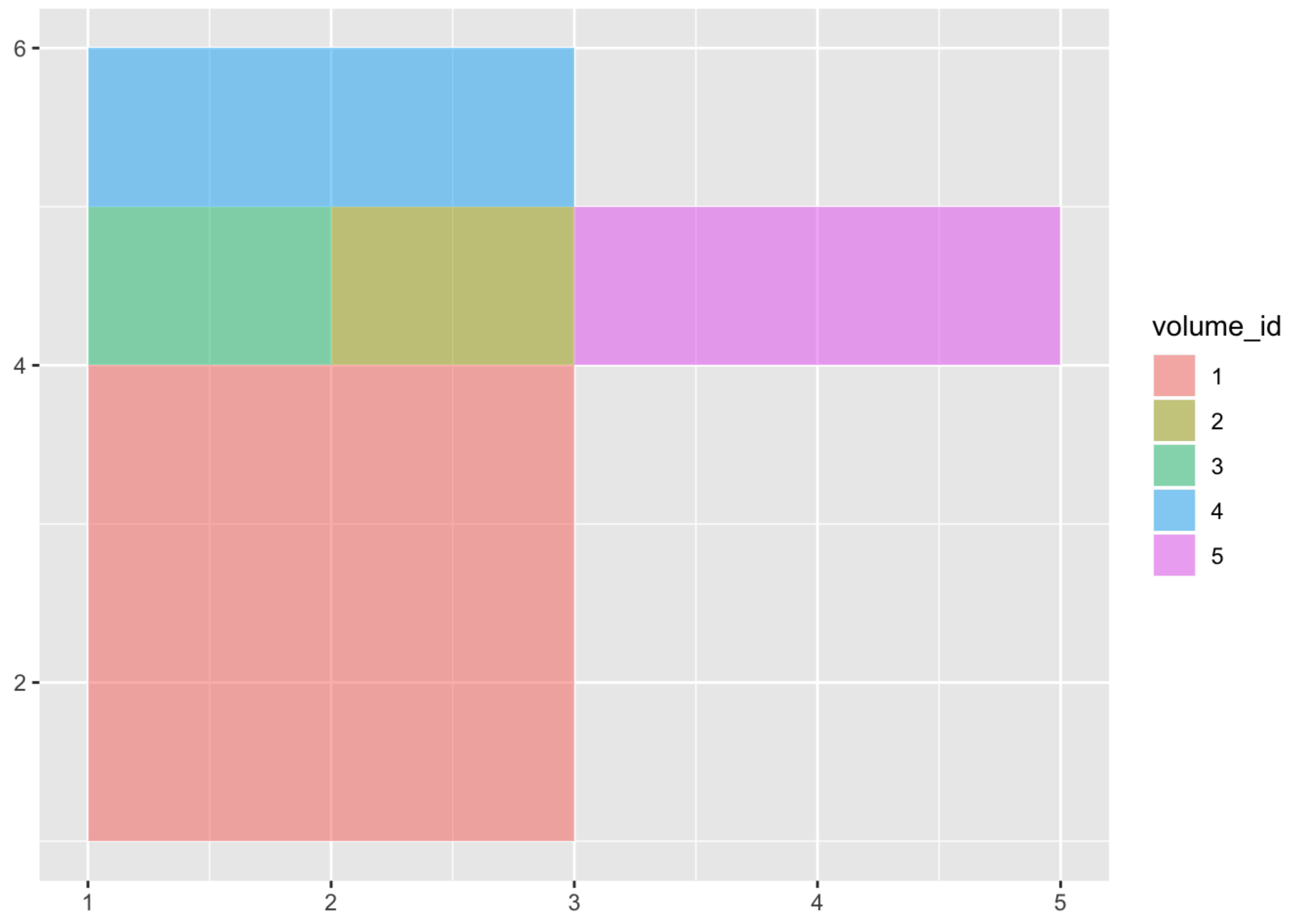
The Problem

First, a concise definition: Given a set of hyperrectangles, find any set of hyperrectangles which occupy the same space with no overlap & contain at least all the original boundaries. A hyperrectangle is the space defined by the cartesian product of ranges. With less jargon, axis-aligned rectangles/prisms/volumes in arbitrary dimensions.

Take an example - 2 dimension rectangles with some overlap:



Here’s a solution:



This is a valid solution because:

- the orginal space is exactly covered
- all the covering volumes are rectangles
- there is no overlap between rectangles
- none of the original boundaries are crossed in the solution.

Motivation

Before describing a solution to this (possibly vacuous seeming) problem, I'd like to motivate it with an example.

The RuleFit algorithm (my implementation here) is a predictive modeling method. At a very high level, its purpose is to produce a set of conjunctive ranges on the predictor set (imagine them as a set of real valued vectors) which explain variation in the response vector. Take for example a predictor set of {age, weight, height} with a response of jumping height (i.e. we're predicting how high someone can jump). One might imagine that lower age, lower weight, and larger height produce larger jumping height, and vice versa; RuleFit may pick up on such a pattern & produce, for example, the following set of "rules":

- {height:[130-200] & weight:[50-60] & age:[10-25]}
- {height:[175-225] & weight:[55-75] & age:[20-35]}
- {height:[100-250] & weight:[70-100] & age:[45-55]}
- etc.

with the idea that a person qualifying for one or more of these rules provides useful information for predicting jumping height. In reality, each rule is assigned a real value (“effect”) which are summed up to produce the jumping height prediction.

Notice that many of these ranges are overlapped, which means a person may qualify for many rules. That makes interpretation & inference somewhat challenging, since we cannot longer can glance at a single rule as a single “nugget” of information. The practitioner is forced to contextualize rules & compute the high-dimensional intersections, which a human mind (or at least mine) isn’t good at. Wouldn’t it be great if these rules were disjoint, so that a person can only qualify for one rule? Solving the overlapped hyperrectangle problem does just that.

(Note: I’ve made some over-simplifications of RuleFit and encourage you to check out the above link if you’re interested.)

Solution

A conceptually simple solution is to:

- Extend all boundaries to infinity (treating the boundary hyperplanes as fixed and allowing all other dimensions to be free)
- Re-create hyperrectangles according to the new boundaries
- Prune hyperrectangles outside the original covered space

To prove it’s a solution:

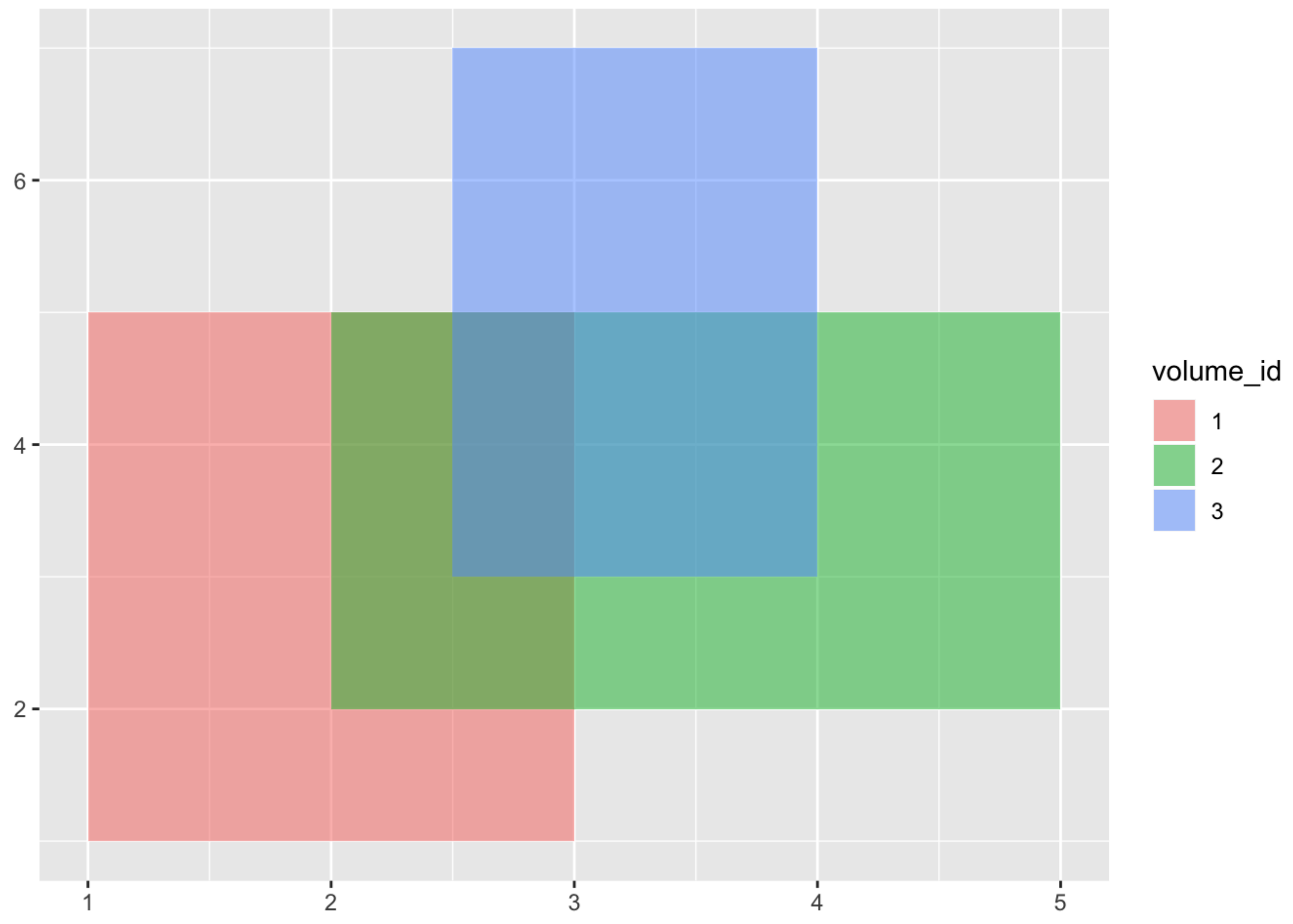
- It creates hyperrectangles, as all original boundaries are orthogonal by problem definition.
- Clearly all the original boundaries are preserved because they are simply scaled in size.
- It is capable of capturying a superset of the original space because it is partitioning the entire n-dimensional space.
- The non-covered hyperrectangles are able to be cleanly removed, as in without eating into the covered space, since, as argued above, all the original boundaries are preserved.

Without ado, we step into some code.

Representation

Hyperrectangles will be represented in dimension-range format. In this representation, an n-dimensional hyperrectangle consists of n dimension-ranges. For this exercise, hyperrectangles are given an id and all placed into one data frame. Here we define and visualize a 2 dimensional example:

```
example_2d <- data.frame(
    dimension = rep(c('x','y'), 3),
  volume_id = rep(1:3, each=2),
  min = c(1,1,
          2,2,
          2.5,3),
  max = c(3,5,
          5,5,
          4,7)
)
```

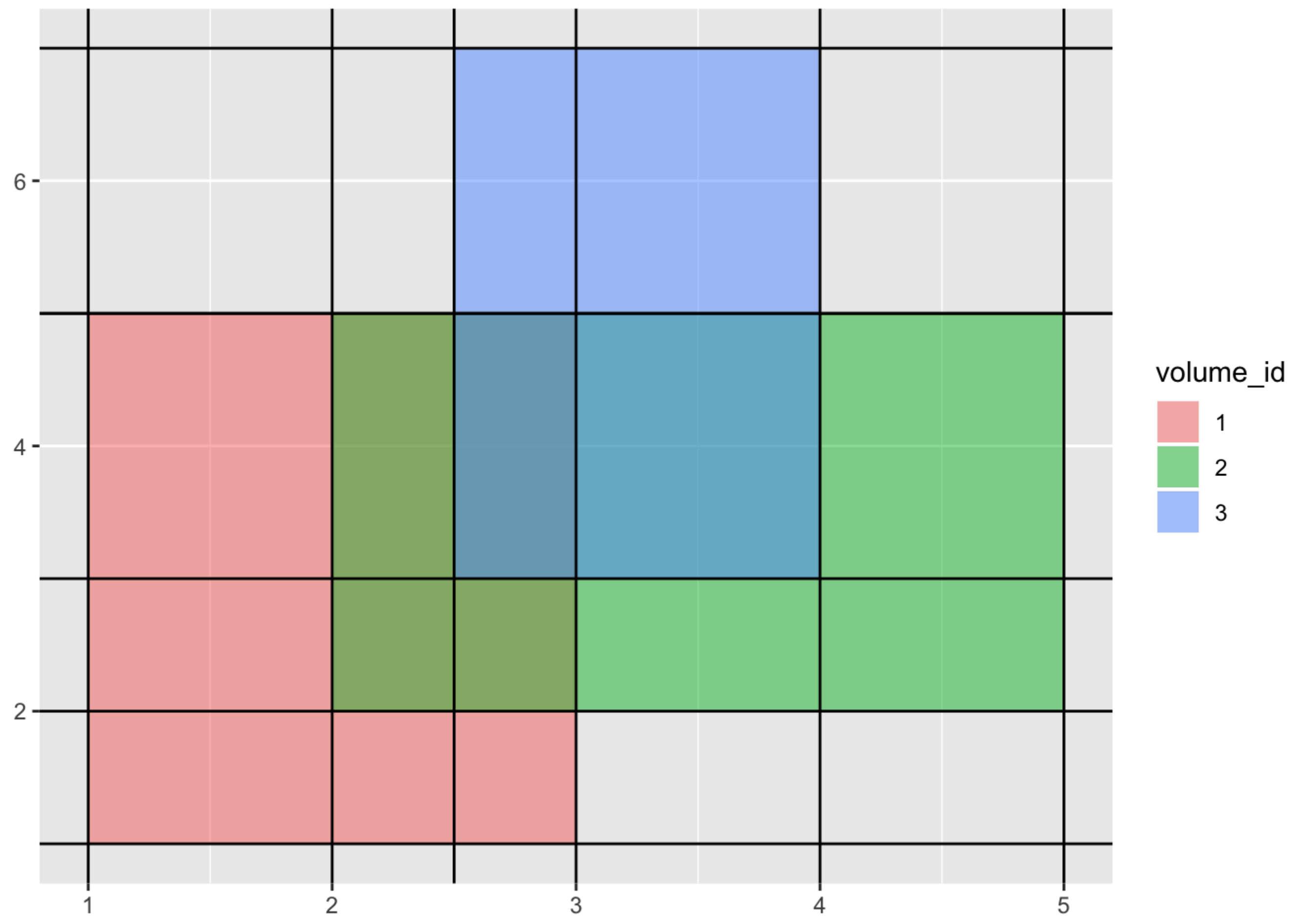


Extending Boundaries

This is as simple as picking “dimension=value” as a fixed point (while letting all other dimensions be free) for all range bounds.

```
build_fully_partitioned_space <- function(volumes) {
  volumes %>%
    mutate(bound = min) %>%
    select(dimension, bound) %>%
    rbind(
      volumes %>%
        mutate(bound = max) %>%
        select(dimension, bound),
      stringsAsFactors = FALSE
    )
}
```

Visualizing our example:

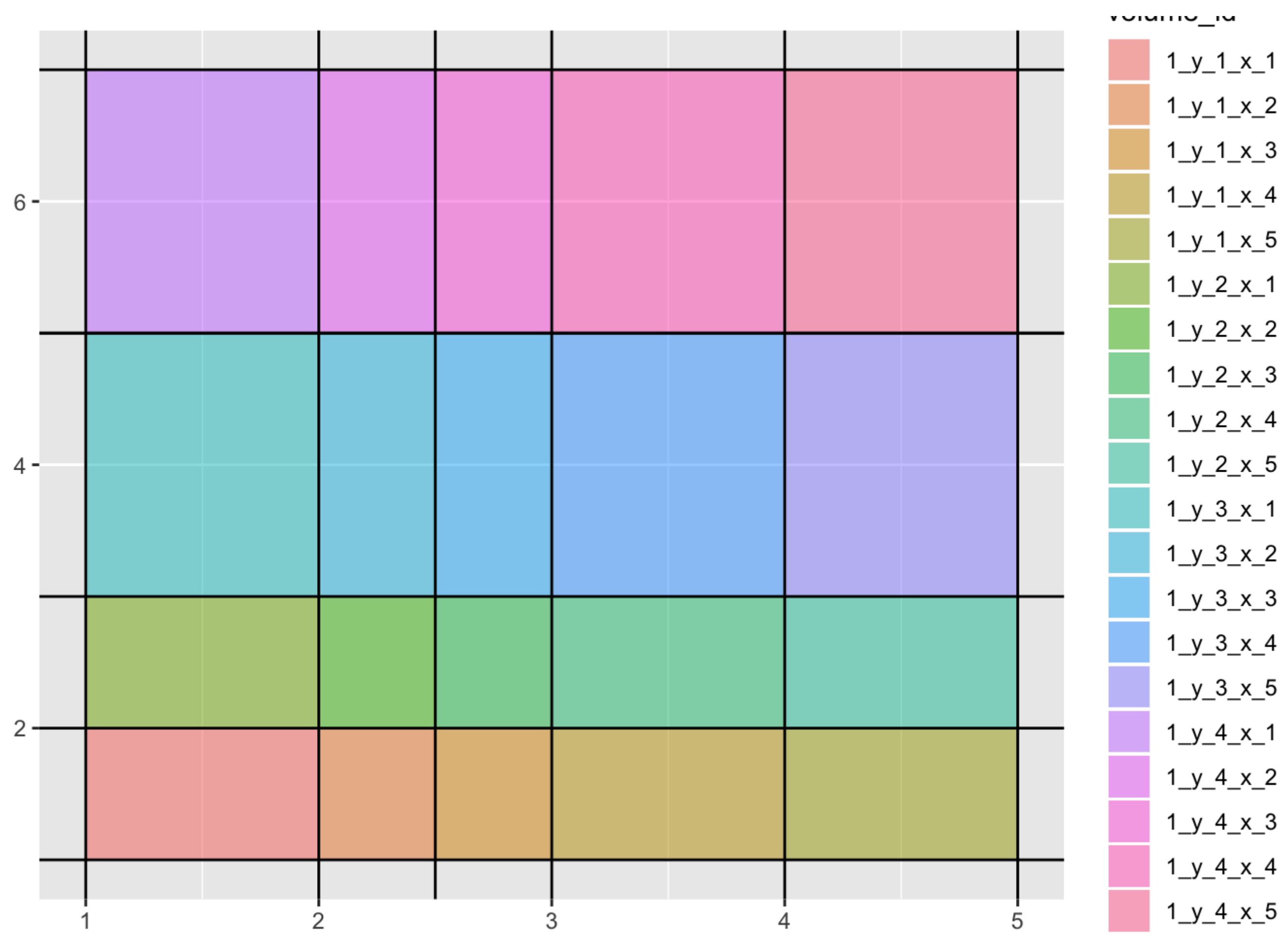


Recreating Hyperrectangles

The next step is to take the orthogonal hyperplanes from the above step and form then into hyperrectangles abutted against one another.

```
generate_volumes_from_partitioned_space <- function(partitioned_space,
id_starter = 1) {
  if (nrow(partitioned_space) == 0) {
    return(data.frame())
  }

  # pick an arbtirary first dimension
  dimension_of_interest <- partitioned_space$dimension[1]
  dimension_bounds <- partitioned_space %>%
    filter(dimension == dimension_of_interest) %>%
    # this is a small optimization - equal bounds are redundant
    distinct() %>%
    arrange(bound)

  # there should always be 2 or more, since each bound corresponds to
hyperrectangle edge
  stopifnot(nrow(dimension_bounds) > 1)
  # subspace meaning everything outside the dimension of interest
  partitioned_subspace <- partitioned_space %>% filter(dimension !=
dimension_of_interest)
  # recursively build ranges from the subspace before tacking on ranges for
the dimension of interest in this stack frame
  subspace_volumes <-
generate_volumes_from_partitioned_space(partitioned_subspace, id_starter =
id_starter)

  # "expanded" by the dimension of interest, that is
  expanded_volumes <- data.frame()
  for (bound_ix in 1:(nrow(dimension_bounds) - 1)) {
    # note that we are iterating on the sorted bounds
    lower_bound <- dimension_bounds$bound[bound_ix]
    upper_bound <- dimension_bounds$bound[bound_ix + 1]

    if (nrow(subspace_volumes) == 0) {
      # case this is the first dimension - there's nothing to add onto
      volume_id <- paste0(id_starter, '_', dimension_of_interest, '_',
bound_ix)
      new_dimension_bounds <- list(volume_id = volume_id,
                                   min = lower_bound,
                                   max = upper_bound,
                                   dimension = dimension_of_interest)
    }
    else {
      # case this is after the first dimension - create a new volume for
each subspace volume with the new bounds added (cartesian product)
      new_dimension_bounds <- lapply(unique(subspace_volumes$volume_id),
function(volume_id) {
        list(volume_id = paste0(volume_id, '_', dimension_of_interest, '_',
bound_ix), # TODO this form of creating an ID could get costly in higher
dimensions
             min = lower_bound,
             max = upper_bound,
             dimension = dimension_of_interest)
      }) %>% bind_rows() %>%
        rbind(subspace_volumes %>%
                mutate(volume_id = paste0(volume_id, '_',
dimension_of_interest, '_', bound_ix)),
              stringsAsFactors= FALSE)
    }

    expanded_volumes <- rbind(expanded_volumes, new_dimension_bounds,
                                                stringsAsFactors = FALSE)
  }

  return(expanded_volumes)
}
```

Visualizing our results (with some difficulty, there’s a lot of colored rectangles):

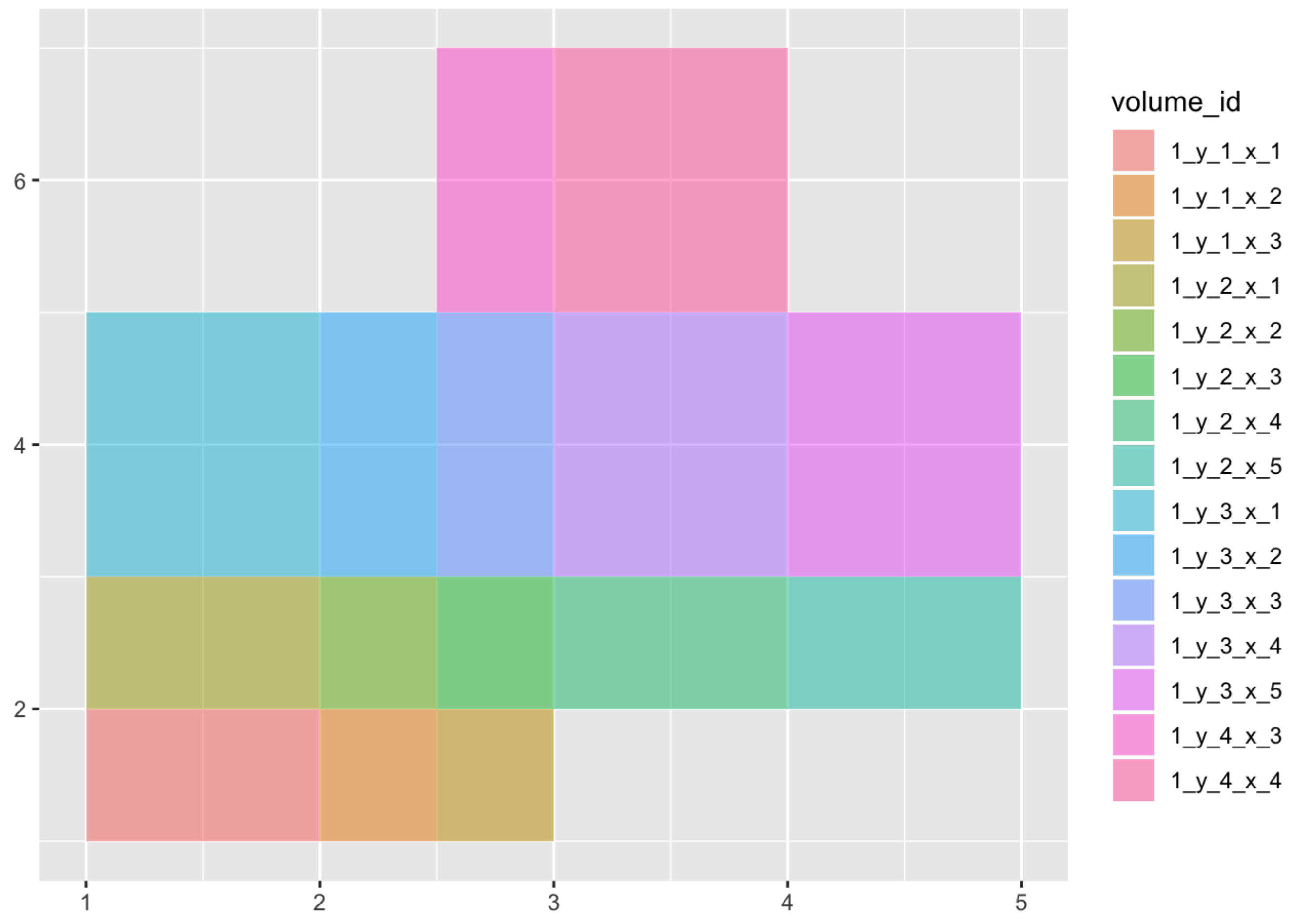


Pruning

You’ll notice that the above is not actually our original space- actually it’s the minimal bounding hyperrectangle. Here we prune away any new hyperrectangles not in the original covering space:

```
prune_uncovering_volumes <- function(new_volumes, original_volumes) {
    # we left join because not all new volumes belong to all old volumes
    # the range join prescribes that the original volumes contains the new
volume
    original_to_new_volumes <- fuzzy_left_join(original_volumes,
new_volumes,
                                           by = c('min' = 'min',
                                                  'max' = 'max',
                                                  'dimension' = 'dimension'),
                                           match_fun = c(`<=`, `>=`, `==`))
%>%
      # renaming some things in a reasonable way
      mutate(dimension = dimension.x) %>%
      select(-dimension.x, -dimension.y)

  covering_volumes <- data.frame()
  for (new_volume_id_to_check in unique(new_volumes$volume_id)) {
    volume <- new_volumes %>%
      filter(volume_id == new_volume_id_to_check)

    in_covering_space <- FALSE
    for (original_volume_id_to_check in unique(original_volumes$volume_id))
{
      original_volume_to_check <- original_to_new_volumes %>%
        filter(volume_id.x == original_volume_id_to_check)
      # here we make sure all dimensions are contained
      volume_dimensions_contained <- original_to_new_volumes %>%
        filter(volume_id.x == original_volume_id_to_check &
               volume_id.y == new_volume_id_to_check) %>%
        pull(dimension) %>%
        setequal(original_volume_to_check$dimension)

      if (volume_dimensions_contained) {
        in_covering_space <- TRUE
        break
      }
    }

    if (in_covering_space) {
      covering_volumes <- rbind(covering_volumes,
                                volume,
                                stringsAsFactors = FALSE)
    }
  }

  covering_volumes
}
```

And visualizing:



Putting It All Together

Here’s out complete algorithm:

```
unoverlap_hyperrectangles <- function(volumes) {
  partitioned_space <- build_fully_partitioned_space(volumes)
  new_volumes <- generate_volumes_from_partitioned_space(partitioned_space)
  prune_uncovering_volumes(new_volumes, volumes)
}
```

A couple nice features to note:

- Although our example was 2 dimensional, everything extends nicely to many dimensions
- In the case of unbounded hyperrectangles (i.e. one or more bounds are infinite), this algorithm still works

And less nice features:

- The solution is frequently suboptimal. In the above example, many rectangles are superfluous, suggesting that there could be a “fusing” step after pruning.

Runtime

Take `V` as the number of hyperrectangles and `D` as their dimensionality. Looking at each step of the algorithm:

- Partitioning the space: `O(VD)` since every range is run through
- Generating volumes: Each level of the recursion, considering a d dimension space, does `dV^(2^(d-1) + 1)` work. The `V^(2^d)` term represents the recursively expanded boundaries, where `O(V)` bounds the number of bounds produced along any dimension, and the `2^d` term represents the cartesian product of the current dimension against all prior subspaces. Solving this recurrence across `1` to `D` recursions is beyond my current time limitations, so I'll give a non-tight bound of `O((DV^(2^(D-1) + 1))^D)`
- Pruning: At this point, `O(DV^(2^(d-1) + 1))` hyperrectangles exist. All original hyperrectangles, `V` are crossed with the new hyperrectangles across all dimensions, `D`, giving `O(D^2 * V^(2^(d-1) + 1))` performance.

As these steps are additive, the total runtime is dominated by the runtime of generating all volumes.

A Cheeky Extension

Ok, so this approach is pretty straight forward. Why did I spend the time writing about it? Besides to the fun application to RuleFit, it solves what is, at face value, a seemingly difficult problem: Given a set of potentially overlapped hyperrectangles, compute the hyper-volume they occupy.

If approached with the problem, you might stumble around with recursive inclusion/exclusion applications (as I did when I failed a simpler version of this problem interviewing at google). But, with overlaps between rectangles removed, the solution is exceedingly simple! Just compute the hyper-volume of all hyperrectangles and sum up.

```
compute_hypervolume <- function(volumes) {
  unoverlap_hyperrectangles(volumes) %>%
    group_by(volume_id) %>%
    summarize(
      hypervolume = Reduce(`*`, max - min)
    ) %>%
    summarize(
      total_hypervolume = sum(hypervolume)
    ) %>%
    pull(total_hypervolume)
}

compute_hypervolume(example_2d)
```

```
## [1] 17
```

Fusing

Several months after writing this (Nov. 2018), I revisited this topic with the desire to optimize the solution. In the context of RuleFit, it is desirable to have as few volumes as possible; it creates fewer model features, meaning fewer model parameters to the benefit of:

- Improved interpretability - fewer parameters means less information for a human observer to store in memory
- Improved LASSO runtime (even the best LASSO algorithms are super-linear in the number of parameters being fit)

Here I aim to extend the algorithm to "fuse" abutted hyperrectangles that preserve the valid solution. The algorithm in English:

- For each dimension, find all pairs of volumes that have equal (min,max) or (max,min) range boundaries and are subspaces of the same original hyperrectangles
- For each abutted pair, iterate over all other dimensions over which they do not abutt.
    - Fusable hyperrectangles have equal ranges on all others dimensions
    - The fused hyperrectangle is computed as the set of all other dimension ranges, and the sequential joining of the ranges of the abutted dimension
- Repeat this process until no fusings are made.

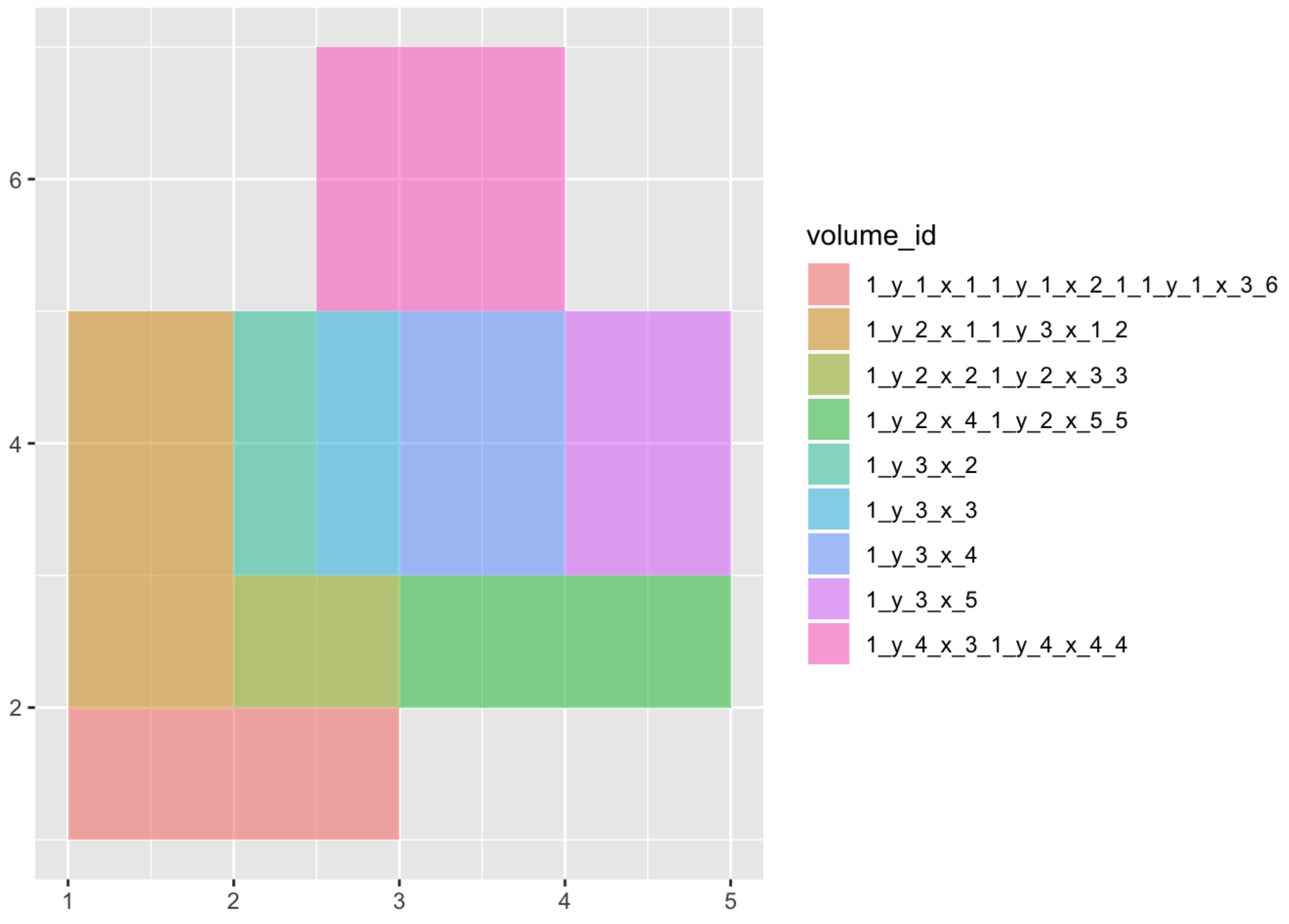
The key to understanding the correctness of this approach is that abutting on more than 1 dimension is not possible, since the space has been unoverlapped; So, what of the remaining dimensions? If the hyperrectangles can be fused into a valid hyperrectangle, the other dimension ranges must all be equal! If they weren't the space we're describing is no longer a cartesian product of orthogonal ranges (since two unmatched ranges lie parallel). The step of checking that candidate hyperrectangles for fusing belong to the same hypervolumes (there may be many) guarantees that the original boundaries are preserved; this is a critical component of our problem statement.

In terms of runtime, I am almost certain this algorithm is suboptimal; in the worst case, we could repeat the inner process equal to the number of hypervolumes being fused. There is almost certainly a deeper structure to the problem that is being ignored in doing this. In terms of final solution, I am certain this algorithm is suboptimal. Since the order of selected subregions to fuse is arbitrary, it's possible that unfusable shapes are created. For a quick example proving this, consider a 5x5 grid. If we follow the above algorithm with adversarial fuse selection ordering (which is valid, since the selection is arbitrary), it is simple to construct one more unfusable rectangles. Clearly the optimal solution is 1 rectangle, so this algorithm is suboptimal.

```
fuse_abutted_hyperrectangles <- function(volumes, original_volumes) {
  dimensionality <- n_distinct(volumes$dimension)

  fused_volumes <- volumes
  fuses_possible <- TRUE
  fused_volume_unique_id <- 1

  while (fuses_possible) {
    fuses_possible <- FALSE
    candidate_fuses <- fused_volumes %>%
      inner_join(fused_volumes, by = c('dimension' = 'dimension',
                                       'max' = 'min'),
                 suffix = c('.left', '.right')) %>%
      filter(volume_id.left != volume_id.right) %>%  # this should only
happen if a range is of size 0
      mutate(
        max = max.right # since the left max (where the abuttment happens
on the right min) must be less than the right max
      ) %>%
      distinct(dimension, volume_id.left, volume_id.right, min, max)

    # note this is a one to many maping, since the originals are overlapped
    current_volumes_to_original <- fused_volumes %>%
      fuzzy_inner_join(original_volumes, by = c('min' = 'min',
                                                'max' = 'max',
                                                'dimension' = 'dimension'),
                       match_fun = c(`>=`, `<=`, `==`)) %>%
      group_by(volume_id.x, volume_id.y) %>%
      filter(n_distinct(dimension.x) == dimensionality) %>%
      summarize(
        volume_id = volume_id.x[1],
        original_volume_id = volume_id.y[1]
      ) %>%
      ungroup() %>%
      select(volume_id, original_volume_id)

    for (candidate_fuse_ix in seq_len(nrow(candidate_fuses))) {
      candidate_fuse <- candidate_fuses[candidate_fuse_ix, ]
      # subvolume because we ignore the dimension of the fuse
      subvolume_left <- fused_volumes %>%
        filter(volume_id == candidate_fuse$volume_id.left & dimension !=
candidate_fuse$dimension)
      subvolume_right <- fused_volumes %>%
        filter(volume_id == candidate_fuse$volume_id.right & dimension !=
candidate_fuse$dimension)

      # this case implies the volume has already been joined
      # meaning the candidate fuse may not be valid any longer - catch it
next iteration
      if (nrow(subvolume_left) == 0 || nrow(subvolume_right) == 0) {
        next()
      }
      stopifnot(nrow(subvolume_left) == nrow(subvolume_right) &&
                  n_distinct(subvolume_left$dimension) ==
n_distinct(subvolume_right$dimension))

      dimension_matches <- subvolume_left %>%
        inner_join(subvolume_right, by = c('dimension' = 'dimension',
                                           'min' = 'min',
                                           'max' = 'max'))

      original_volume_counts <- current_volumes_to_original %>%
        filter(volume_id %in% c(candidate_fuse$volume_id.left,
candidate_fuse$volume_id.right)) %>%
        group_by(original_volume_id) %>%
        count() %>%
        pull(n)

      if (nrow(dimension_matches) == dimensionality - 1 &&
          all(original_volume_counts == 2)) { #
        fuses_possible <- TRUE

        # add in the new volume
        fused_volume <- rbind(
          dimension_matches %>% select(min, max, dimension),
          candidate_fuse %>% select(min, max, dimension),
          stringsAsFactors = FALSE)
        fused_volume$volume_id <- paste0(candidate_fuse$volume_id.left,
'_',
                                          candidate_fuse$volume_id.right,
'_',

as.character(fused_volume_unique_id))
        fused_volume_unique_id <- fused_volume_unique_id + 1
        fused_volumes <- rbind(fused_volumes,
                               fused_volume,
                               stringsAsFactors = FALSE)

        # clean up the old volumes
        fused_volumes <- fused_volumes %>%
          filter(volume_id != candidate_fuse$volume_id.left & volume_id !=
candidate_fuse$volume_id.right)
      }
    }
  }
  return(fused_volumes)
}
```

Running & visualizing on our prior example:

```
fused_unoverlapped_hyperrectangles <-
fuse_abutted_hyperrectangles(unoverlapped_hyperrectangles, example_2d)
```



And just like that, we’ve gone from 15 to 9 volumes! Although not generally the case, this solution is optimal; no further fusings can be made without violating the original boundaries.

Our final algorithm is amended as follows:

```
unoverlap_hyperrectangles <- function(volumes) {
  partitioned_space <- build_fully_partitioned_space(volumes)
  new_volumes <- generate_volumes_from_partitioned_space(partitioned_space)
  solution <- prune_uncovering_volumes(new_volumes, volumes)
  fuse_abutted_hyperrectangles(solution, volumes)
}
```