Stat 218 Final Project Writeup

Terry Luongo

Homemade Splitting Algorithm for Regression Tree in 1D

My final project was building the splitting algorithm for a 1-dimensional regression tree. The splitting and pruning metric were built around minimizing the RMSE in each group up until a specified threshold. The splitting process uses gradient descent for algorithmic efficiency.

We will build this algorithm for the ground-up: first we will implement a primitive gradient descent method that can be used to the nearest local split. Then we will make an algorithm to calculate the best global split from the potential local splits given existing splits. Lastly, we repeat this process until our stopping criteria has been met, adding in a pruning step to minimize overfitting and improve interpretability.

Gradient Descent Approximation Algorithm

Calculate RMSE

```r
library(tidyverse)

# calculate rmse for given splits
calculate_rmse <- function(df,prev_splits,current) {
  splits <- prev_splits
  if (!is.null(current) && !(current %in% prev_splits)) {
    splits <- c(prev_splits,current)
  }
  splits <- c(min(df$x)-1,splits,max(df$x)+1)
  df <- df %>% mutate(group = cut(x, breaks = splits)) %>%
    group_by(group) %>%
    mutate(group_mean = mean(y)) %>%
    mutate(component_rmse = (y-group_mean)^2) %>%
    ungroup() %>%
    summarise(total_rmse = sum(component_rmse))
  df %>% select(total_rmse) %>% pull()
}
```

This is the lowest level of the algorithm. It calculates RMSE - the root mean squared error of each group's observations compared to their means. It takes a potential split and calculate what the total RMSE, given whatever current splits are factored in. We need it for a few levels of the algorithm - mainly to see what direction our gradient descent should step in to make a good split, as well as calculating which splits achieve a high reduction in RMSE during the tree pruning process.

Gradient Descent

```
# calculates average delta around radius of point, by index
find_avg_delta <- function(df, radius, prev_splits, point) {
  index <- which.min(abs(df$x - point))[[1]]

  rmse_vector <- sapply((index - radius):(index + radius),
                        function(i) calculate_rmse(df, prev_splits, df$x[i]))

  -mean(diff(rmse_vector))
}
```

We aim to use gradient descent on the RMSE function - the potential split value is the variable, and we are seeking to minimize the RMSE by optimally placing the split to achieve maximum separation between groups and minimum variability within groups. In traditional calculus, we would minimize the RMSE by taking the derivative and seeing where it goes from - to +, which would be the minimum of the function. However, we are unable to do this for two reasons. First, there is no true 'function' - all we have is a set of observations that we believe is related to an underlying true relationship between the variables. Additionally, as the size of the data increases, testing each value becomes significantly harder. Clearly we need some sort of approximation method to find the best way to split the dataset. This is where gradient descent comes in.

Gradient descent in a pure mathematical sense moves proportionally to the value of the derivative of a function. As the data will always be discrete (and not a function in the literal sense), however, we need some analogue to a derivative - which we can see more when we look at the definition of a derivative.

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

If we look more at the fraction component and disregard the limit that h is going to 0, we can treat the index of our dataset as h. So the difference between two successive data points can be thought of as the derivative - the rate of change. However I decided to extend this to an arbitrary number of successive data points - the 'radius' of points we are considering. This reduces overfitting and gives a wider picture of what is happening in the dataset. This is what the `find_avg_delta` function performs. It looks at `radius` points before and after our current value and calculates the average of the rate of change between these points. The gradient descent algorithm then steps through the dataset proportionally to this rate (by some factor `alpha` that the caller can specify).

```r
# finds local minima, should return in a tibble with RMSE and break point
# either stops when 0 move reached or after iterations has elapsed rounds last
# round_x_iterations -> if it is cycling around best point
step_gradient_descent <- function(df,init,alpha,radius,iterations,
                                  prev_splits = NULL,
                                  round_x_iterations=5) {
  zeroed <- FALSE
  iterations_to_round <- c()
  index <- which.min(abs(df$x - init))[[1]]

  for (i in 1:iterations) {

    value <- df$x[index]

    delta <- find_avg_delta(df,radius,prev_splits, value)
    move <- round(delta * alpha)
    index <- index + move

    if (move == 0) {
      zeroed <- TRUE
      break
    }

    if (iterations - i < round_x_iterations) {
      iterations_to_round <- c(iterations_to_round,df$x[index])
    }
  }

  split <- ifelse(zeroed, df$x[index], mean(iterations_to_round))
  rmse <- calculate_rmse(df, prev_splits, split)

  tibble(split = split, rmse = rmse)
  }
```

Now we can take a look at the `step_gradient_descent` function: it is moving proportionally to the rate of change in the RMSE function, seeking to minimize the total RMSE. We move the index of the dataset by `alpha * delta`, where delta is what the `find_avg_delta` function calculates. We move in this way until a stopping criteria is reached - either the move becomes 0, or we achieve some number of specified iterations. However, the latter has an interesting case. The function could potentially cycle around the desired stopping point. This is why I implemented the `round_x_iterations` parameter - if a local 0 isn't exactly found, we round the last `round_x_iterations` to return our best estimate of the split. This of course assumes that we have already found the neighborhood of the split and our cycling it - the iterations has to be high enough for this to work as intended.

Finding the best split - globally

```
# given array of numbers, calculates midpoints returning n-1 numbers
calculate_midpoints <- function(array) {
  midpoints <- numeric(0)
  # Iterate over pairs of consecutive numbers in the array
  for (i in seq(1, length(array)-1)) {
    start <- array[i]
    end <- array[i + 1]
    # Calculate the midpoint and append it to the result
    midpoint <- (start + end) / 2
    midpoints <- c(midpoints, midpoint)
  }

  return(midpoints)
}

# calculates midpoints given all previous splits, returns best split from ones starting at
find_global_split <- function(df,alpha,radius,iterations,
                              prev_splits = NULL,
                              round_x_iterations = 5) {
  splits <- sort(prev_splits)
  splits <- c(min(df$x)-1,splits,max(df$x)+1)
  midpoints <- calculate_midpoints(splits)
  minima <- tibble(split = numeric(), rmse = numeric())

  for(midpoint in midpoints) {
    local_min <- step_gradient_descent(df,midpoint,alpha,radius,iterations,
                                       prev_splits,
                                       round_x_iterations)
    minima <- bind_rows(minima, local_min)

  }

  minima %>% arrange(rmse) %>% head(1) %>% select(split) %>% pull()
}
```

This function returns the best split out of all the local splits. My local split function will essentially only find the local split within a region that has already been split - it can't ever cross a previous split as that will always increase RMSE. Therefore calculating just one split for each iteration of the algorithm will leave out increasingly more potential candidates as we get more splits in the dataset. To solve this problem, we calculate the midpoint of each region in the dataset and start the local algorithm from there. The algorithm returns the split and the RMSE achieved from splitting there. This function then returns the best split out of the potential candidates.

Starting at the midpoint was somewhat of an arbitrary choice, but I believe it makes sense as the gradient descent should have to travel the least on average to find a sensible value to split on.

Pruning, Putting it all Together

```
# recursively prune worst split
prune_splits <- function(df, splits, threshold) {
  init_rmse <- calculate_rmse(df,splits,NULL)
  rmse_without_each <- c()
  for (i in 1:length(splits)) {
    without_this <- calculate_rmse(df,splits[splits != splits[i]],NULL)
    rmse_without_each <- c(rmse_without_each, without_this - init_rmse)
  }
  sorted_indices <- order(rmse_without_each)
  worst_split <- splits[sorted_indices[1]]
  worst_rmse <- rmse_without_each[sorted_indices[1]]

  ratio <- worst_rmse / init_rmse

  if (length(splits) < 3 | ratio > threshold) {
    return(splits)
```

```r
  }
  else {
    # recursion
    prune_splits(df,splits[splits != worst_split],threshold)
  }
}

make_splits <- function(df, max_cuts, alpha, radius, iterations, threshold,
                        round_x_iterations = 5) {
  splits <- c()
  for (i in 1:max_cuts) {
    new <- find_global_split(df,alpha,radius,iterations,splits,round_x_iterations)
    splits <- c(splits,new)
  }

  splits <- prune_splits(df,splits,threshold)

  plot_parts(df,splits)
}
```

As each split is greedy - we are only choosing the best candidates from local minima of the gradient descent function - the pruning step significantly improves the quality of the splits. By making more splits than we anticipate needing, than removing the ones that achieve the least reduction in RMSE, we achieve a significant improvement in the quality of the tree. My pruning algorithm also goes by RMSE, where for each iteration we calculate the worst split in the tree, and then remove it if it achieves less than a specified reduction in RMSE.

`make_splits` puts the whole algorithm together - we keep track of each new split we add, then remove the bad ones.

We are now ready to visualize the algorithm.

Results and Visualization

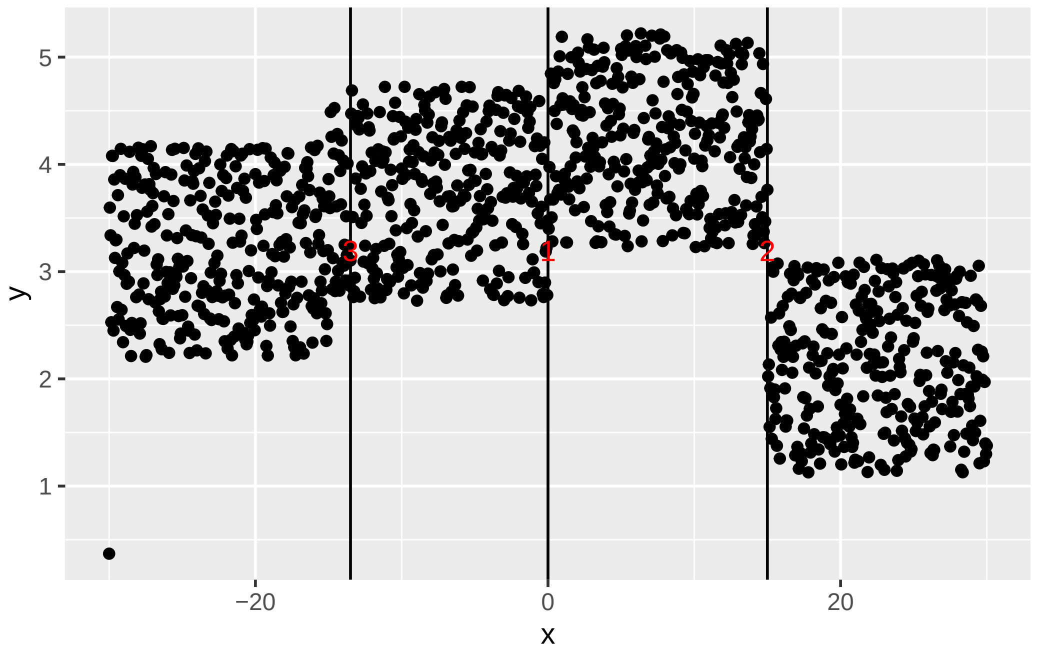
```r
# generate fake data in tibble

make_random_tibble <- function(range, distance, noise, sections, section_ratio) {
  x <- seq(from = -range, to = range, by = distance)
  y <- runif(length(x)) * noise

  chunk_length <- length(y) / sections
  for (i in 1:sections) {
    start_index <- ((i - 1) * chunk_length + 1)
    end_index <- (i * chunk_length)
    y[start_index:end_index+1] <- y[start_index:end_index+1] + runif(1) * noise * section_
  }
  tibble(x,y)
}

# plot what splits look like for dataset
plot_parts <- function(df,splits) {
  vlines <- data.frame(xintercept = splits, index = seq_along(splits))

  df %>% ggplot(mapping = aes(x = x, y = y)) +
    geom_point() +
    geom_vline(data = vlines, mapping = aes(xintercept = xintercept)) +
    geom_text(data = vlines, aes(x = xintercept, label = index, y = 3),
              vjust = -0.5, hjust = 0.5, color = "red")  # Adjust vjust and hjust as neede
}
```

These are the two functions I primarily used to test and develop my algorithm. `make_random_tibble` makes a tibble with uniformly-separated x-values, and a specified number of splits in the y direction where all of the values in a certain range are added some random amount. The ratio of regular noise to differences in the sections can be specified to see how sensitive the algorithm is to noise.

`plot_parts` simplify takes the dataset we generated and plots the splits we calculated against it.

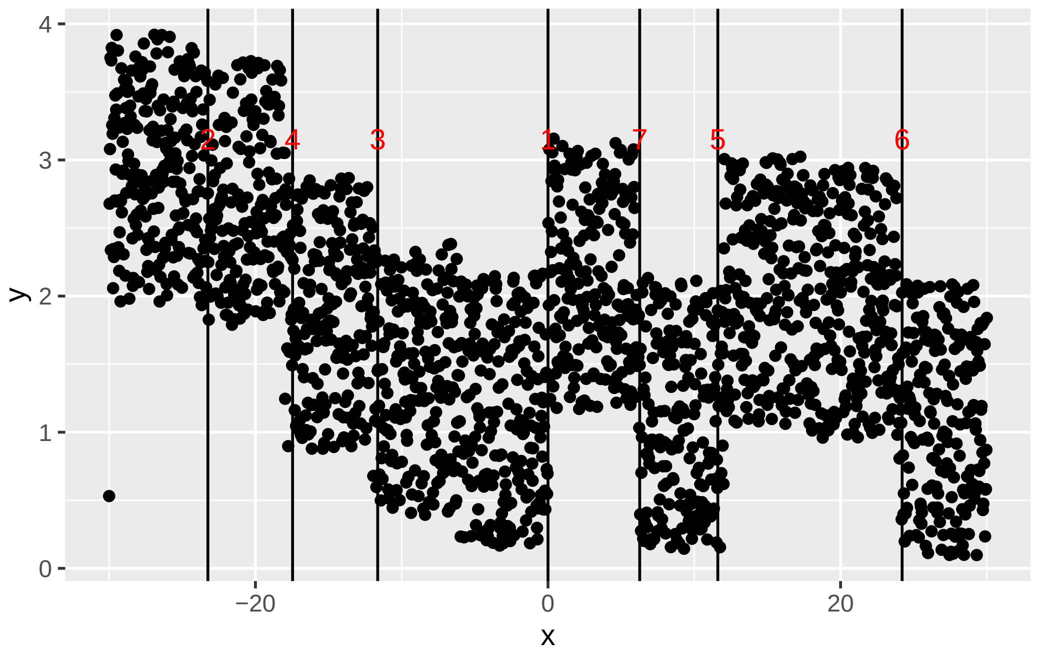
Demonstration

```
set.seed(2)

test <- make_random_tibble(range = 30,
                           distance = 0.05,
                           noise = 2,
                           sections = 4,
                           section_ratio = 3)

make_splits(test,
            max_cuts = 20,
            alpha = 0.5,
            radius = 2,
            iterations = 10,
            threshold = 0.01)
```

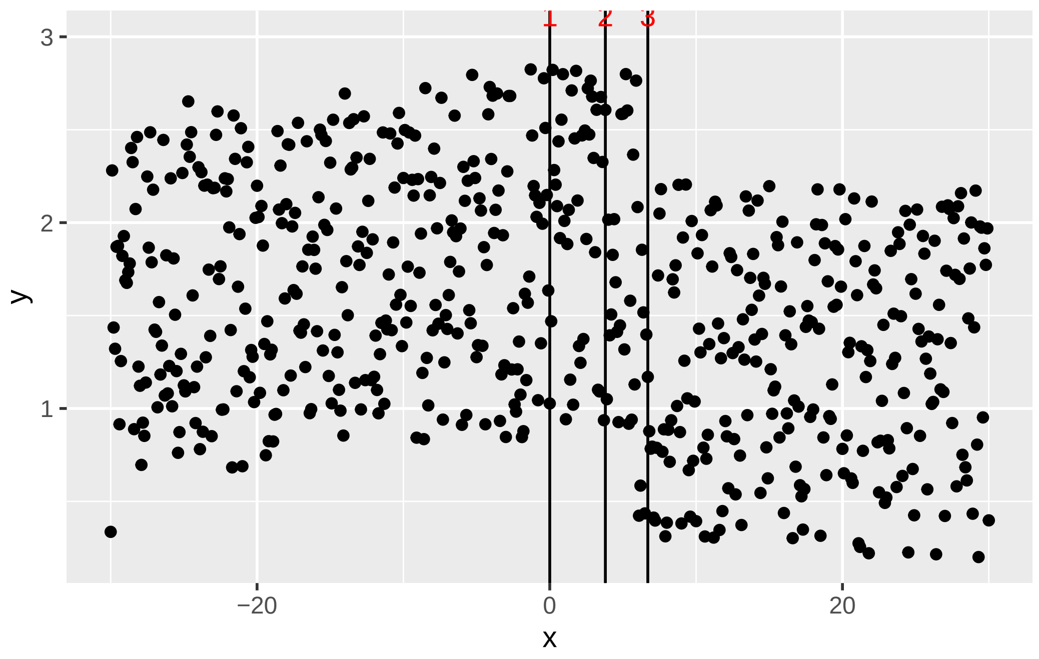


As you can see it correctly identified all of the splits here. We will decrease the split to noise ratio as well as increase the amount of splits to see how our algorithm competes.

```
set.seed(1)

test <- make_random_tibble(range = 30,
                           distance = 0.03,
                           noise = 2,
                           sections = 10,
                           section_ratio = 1)

make_splits(test,
            max_cuts = 20,
            alpha = 0.5,
            radius = 2,
            iterations = 25,
            threshold = 0.01)
```



```
set.seed(3)

test <- make_random_tibble(range = 30,
                           distance = 0.1,
                           noise = 2,
                           sections = 5,
                           section_ratio = 0.5)

make_splits(test,
            max_cuts = 20,
            alpha = 0.25,
            radius = 2,
            iterations = 25,
            threshold = 0.01)
```

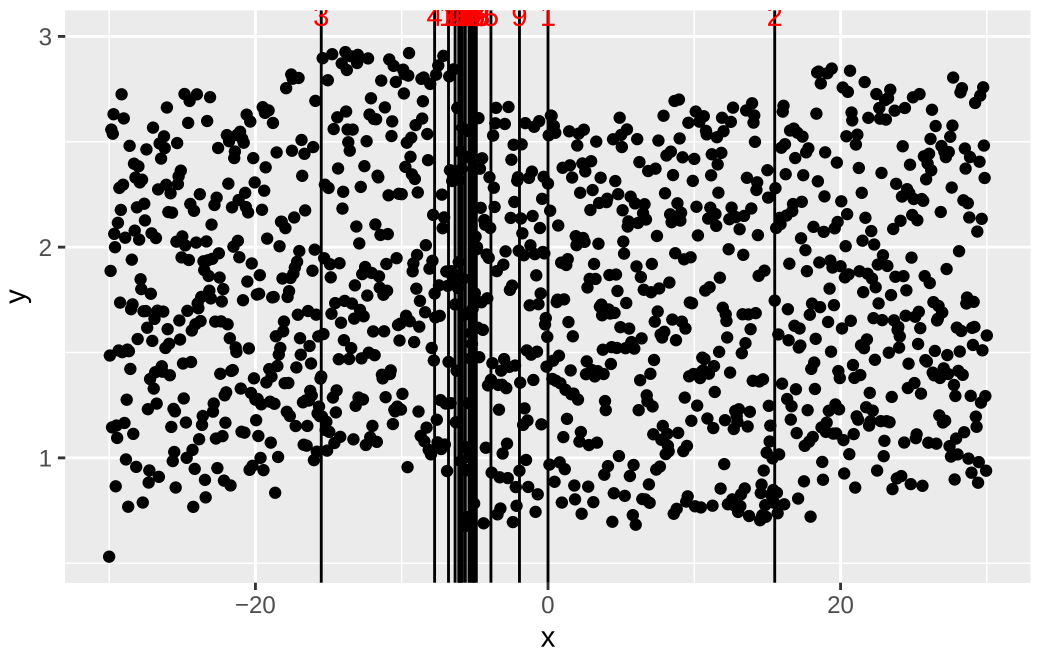


Clearly the algorithm struggles a little bit as the points get more sparse and the ratio of splits to noise decreases further, but it still does a decent job.

Limitations

1. I only developed and tested the algorithm on uniformly distributed x-values. Moves are made by index so might behave irrationally if we give it a more realistic x distribution.
2. Sometimes the path finding won't find a local split that will lead to much higher reductions in RMSE later down the line. It will get stuck optimizing over a certain area, neglecting entire portions of the distribution.

```r
set.seed(1)

test <- make_random_tibble(range = 30,
                           distance = 0.05,
                           noise = 2,
                           sections = 5,
                           section_ratio = 0.5)

make_splits(test,
            max_cuts = 20,
            alpha = 0.25,
            radius = 2,
            iterations = 25,
            threshold = 0.0)
```



You can clearly see that here when I turn off the pruning algorithm - splits are being very carefully considered over certain areas but not at all over others, even if a promising one could be found. Great example of high precision and low accuracy.

Future Work

If I had more time, I would extend the project in two main ways:

1. Multiple dimensions. Had a hesitancy to do this as visualization would become more difficult and would lose some intuition on what the algorithm is doing. Additionally, debugging the one dimensional case took far longer than expected.
2. Standardized interface - if you could call the algorithm in a format similar to the other stat learning algorithms we learned, eg. `mytree(y ~ x, data = test)` .
3. Improving efficiency - there are definitely a few optimizations that could be made in the `calculate_rmse` function. This is called thousands of times in constructing a single tree so improving how this function works would have large implications on performance.