Attribution fit - Detailed overview

Daniel Booth

2019-01-14

Overview

This vignette will go into detail about how the attribution algorithm works, and the differences between the `path_transform_methods`.

For a general introduction to the package please, if you haven't already, see the **Fractribution Model - Quick start** vignette included in the package. To launch run:

```
vignette('fractribution_model_quick_start')
```

The fractribution algorithm

Motivation

Fractribution uses a simplified counterfactual shapley value algorithm to assign fractions to each touchpoint in a path to conversion. The idea is that if you have a path to conversion that is **B > C > A** you want to be able to use a data-driven approach to appropriately assign a **fractional credit** to each touchpoint in the path. This might be say:

- **A**: 35.8%
- **B**: 43.4%
- **C**: 20.8%

These fractional values—at the **path-level**—are particularly useful if you additionally know the revenue value of the conversion(s) that this path resulted in. For example, if one converting customer on this path had created **$100** in revenue we'd want to distribute that revenue across the fractional values to yield an **attributed revenue** figure. Here it would be:

- **A**: $35.80
- **B**: $43.40
- **C**: $20.80

where $35.80 + $43.40 + $20.80 = $100.00.

Ultimately, we sum up these attributed revenue fractions across every customer to get a **channel-level attribution** report. This is what the `channel_revenue_attribution_report()` function does.

A worked example

***NOTE**: Everything shown in this vignette happens under the hood of the `attribution_fit()` function. You should NEVER have to do this manually. We are exploring the internals here just for demonstration purposes.*

Let's explore this in more detail. First load the package. We will also load `dplyr` to use in some of the examples:

```
library(fractribution.model)
library(dplyr)
```

Prepare paths and conversion probabilities

Now we will unravel the `attribution_fit()` function. Using the **path_summary** input provided, we first transform paths as per the **path_transform_method** (this is explained further in the path transform methods section below). Next we calculate conversion probabilities for each path.

For the **exposure** path transform, this yields the following:

```
# Transform paths and calculate conversion probs
path_summary <- fractribution.model:::exposure_path_summary_transform(example_path_summary)

# Filter down our example
demo_path_summary <- path_summary %>%
  filter(path %in% c('B > C > A',
                     'B > A',
                     'B > C',
                     'C > A',
                     'A',
                     'B',
                     'C'))

# Inspect
demo_path_summary
#> # A tibble: 7 x 4
#>   path      total_conversions total_non_conversions conversion_prob
#>   <chr>                 <dbl>                 <int>           <dbl>
#> 1 B                       655                 46020          0.0140
#> 2 A                       612                154849          0.00394
#> 3 C                       310                 45768          0.00673
#> 4 B > C                    18                   549          0.0317
#> 5 B > A                    15                   250          0.0566
#> 6 C > A                     7                   357          0.0192
#> 7 B > C > A                 1                    10          0.0909
```

See that for path **B > C > A** the conversion probability is **0.0909**.

Counterfactuals and fractions

The fractribution algorithm works as follows:

1. Start with a **baseline path** and its **conversion_probability**
2. Calculate the baseline path's **leave-one-out counterfactuals**
3. Calculate the attribution fraction for each event in the baseline path as the **marginal contribution** that the event adds to the **conversion_probability** over its respective counterfactual. That is: `conversion_prob(path with event) - conversion_prob(path without event)`
4. If required (and this is the default), **normalize** the attribution fractions
5. Repeat steps 1. through 4. for all paths

Now let's do this with real data. Here our baseline path will be **B > C > A**.

Step 1: First let's define the **baseline_path** and its **baseline_conversion_probability**:

```
baseline_path <- 'B > C > A'
baseline_conversion_prob <- demo_path_summary %>%
  filter(path == baseline_path) %>%
  pull(conversion_prob)
```

```
# Inspect
baseline_conversion_prob
#> [1] 0.09090909
```

Step 2: Now we want to calculate each **leave-one-out counterfactual** and its respective **conversion_probability**:

```
# The path length, is the number of counterfactuals
drop_indicies <- fractribution.model:::path_length(baseline_path)

# Counterfactuals (drop an event at each index)
counterfactuals <- purrr::map_chr(
  1:drop_indicies,
  ~ fractribution.model:::drop_event(baseline_path, .))

# Get conversion probability
counterfactuals <- demo_path_summary %>%
      filter(path %in% counterfactuals)

# For demonstration purposes, manually add in the dropped event
counterfactuals <- counterfactuals %>%
  mutate(dropped_event = c('A', 'C', 'B')) %>%
  select(dropped_event, counterfactual = path, conversion_prob)

# Inspect
counterfactuals
#> # A tibble: 3 x 3
#>   dropped_event counterfactual conversion_prob
#>   <chr>         <chr>                   <dbl>
#> 1 A             B > C                  0.0317
#> 2 C             B > A                  0.0566
#> 3 B             C > A                  0.0192
```

See that for path **B > C > A** the counterfactuals are **B > C**, **B > A**, and **C > A**.

Step 3: Calculate the marginal contribution of each event in the **baseline_path**:

To determine the fractional value for event **A** in the baseline path, we will take the **baseline_conversion_prob** and subtract the **conversion_prob** for its leave-one-out counterfactual, which here is **0.0317**. We replicate the same for events **B** and **C**. To run all we can do:

```
attribution_fractions <- counterfactuals %>%
  mutate(marginal_contribution = baseline_conversion_prob - conversion_prob) %>%
  select(event = dropped_event, attribution_fraction = marginal_contribution)

# Inspect
attribution_fractions
#> # A tibble: 3 x 2
#>   event attribution_fraction
#>   <chr>                <dbl>
#> 1 A                   0.0592
#> 2 C                   0.0343
#> 3 B                   0.0717
```

So here **A**'s attribution_fraction of **0.0592** is **0.0909 - 0.0317**, etc.

Step 4: Normalize the **attribution_fraction**s:

Finally we want to make the **attribution_fraction**s sum to 1 so we are able to interpret them as percentages of a conversion. This makes it simple to distribute revenue across each event.

To do this we normalize, that is divide by the sum:

```
attribution_fractions <- attribution_fractions %>%
  mutate(normalized_attribution_fraction =
           attribution_fraction / sum(attribution_fraction)) %>%
  select(event, normalized_attribution_fraction)

# Inspect
attribution_fractions
#> # A tibble: 3 x 2
#>   event normalized_attribution_fraction
#>   <chr>                           <dbl>
#> 1 A                               0.358
#> 2 C                               0.208
#> 3 B                               0.434
```

Which yields a fractional attribution of 35.8% for A, 43.4% for B, and 20.8% for C.

Another example, more succinct

The actual implementation isn't as drawn out as what is shown above. There is a `fractional_values()` function that handles the counterfactuals and the marginal contributions for us. Let's leverage it this time for simplicity.

Again start with a **baseline_path**, this time **B > C**, and its **baseline_conversion_probability**:

```
baseline_path <- 'B > C'
baseline_conversion_prob <- demo_path_summary %>%
  filter(path == baseline_path) %>%
  pull(conversion_prob)
```

Calculate the marginal contributions (here we are NOT yet normalizing):

```
fractribution.model:::fractional_values(
  baseline_path,
  baseline_conversion_prob,
  all_paths = demo_path_summary,
  normalize = FALSE,
  path_transform_method = 'exposure')
#> # A tibble: 1 x 3
#>   path       B      C
#>   <chr>  <dbl>  <dbl>
#> 1 B > C 0.0250 0.0177
```

We actually can normalize directly in the previous step:

```
fractribution.model:::fractional_values(
  baseline_path,
  baseline_conversion_prob,
  all_paths = demo_path_summary,
```

```
  normalize = TRUE,
  path_transform_method = 'exposure')
#> # A tibble: 1 x 3
#>   path      B     C
#>   <chr> <dbl> <dbl>
#> 1 B > C 0.585 0.415
```

Which yields a fractional attribution of 58.5% for B and 41.5% for C.

Map across all paths

The implementation in the package maps across all paths like so:

```
fracs <-
  purrr::map2_df(demo_path_summary$path,
                 demo_path_summary$conversion_prob,
                 ~ fractribution.model:::fractional_values(
                   .x, .y,
                   all_paths = demo_path_summary,
                   path_transform_method = 'exposure'))

# We also clean up the channel names so they make good column names
# Replace all NAs with 0, and rename columns without spaces or dashes
fracs <- fracs %>%
  mutate_all(funs(replace(., is.na(.), 0))) %>%
  rename_all(funs(paste0(stringr::str_to_lower(.) %>%
                         stringr::str_replace_all(' - ', ' ') %>%
                         stringr::str_replace_all('-', ' ') %>%
                         stringr::str_replace_all(' ', '_'))))

# Inspect
fracs
#> # A tibble: 7 x 4
#>   path          b     a     c
#>   <chr>     <dbl> <dbl> <dbl>
#> 1 B         1     0     0
#> 2 A         0     1     0
#> 3 C         0     0     1
#> 4 B > C     0.585 0     0.415
#> 5 B > A     0.553 0.447 0
#> 6 C > A     0     0.450 0.550
#> 7 B > C > A 0.434 0.358 0.208
```

You can see the results of the two examples before in here.

Special cases

There are a few special cases that arise. These are listed and shown with examples below.

The counterfactual doesn't exist

For baseline path **B > D > G**, the counterfactual for **B**, **D > G**, doesn't exist. This is approached as meaning that the **conversion_probability** for the path **D > G** is just equal to **0** and so the attribution fraction for **B** is just the **baseline_conversion_prob**.

Concretely, here is that **baseline_conversion_prob** for path **B > D > G** :

```
baseline_path <- 'B > D > G'
baseline_conversion_prob <- path_summary %>%
  filter(path == baseline_path) %>%
  pull(conversion_prob)

# Inspect
baseline_conversion_prob
#> [1] 0.5
```

Now look at the non-normalized attribution fractions:

```
fractribution.model:::fractional_values(
  baseline_path,
  baseline_conversion_prob,
  all_paths = path_summary,
  normalize = FALSE,
  path_transform_method = 'exposure')
#> # A tibble: 1 x 4
#>   path          B     D     G
#>   <chr>     <dbl> <dbl> <dbl>
#> 1 B > D > G   0.5 0.333 0.440
```

See that **B** gets **0.5** (which is the **baseline_conversion_prob**).

A marginal contribution is negative

There will be cases where a counterfactual conversion probability will actually be *greater than* the **baseline_conversion_prob**. Hence the marginal contribution will actually be **negative**.

In these cases, we do NOT assign a negative fraction to the event, as this can cause the other events in the path to have a fractional attribution *greater than* 1. This is a problem because it can, in turn, cause the channel-level report to attribute more conversions to that channel than actually existed (that is, the total number of conversions for paths containing that event).

To mitigate this issue, in these cases, we just floor negative marginal contributions to **0**.

Concretely, here's a path with the issue:

```
baseline_path <- 'C > A > C'
baseline_conversion_prob <- path_summary %>%
  filter(path == baseline_path) %>%
  pull(conversion_prob)

# Inspect
baseline_conversion_prob
#> [1] 0.01724138
```

Get the counterfactuals:

```
# The path length, is the number of counterfactuals
drop_indicies <- fractribution.model:::path_length(baseline_path)

# Counterfactuals (drop an event at each index)
counterfactuals <- purrr::map_chr(
  1:drop_indicies,
  ~ fractribution.model:::drop_event(baseline_path, .))
```

```r
# Get conversion probability
counterfactuals <- path_summary %>%
  filter(path %in% counterfactuals) %>%
  select(path, conversion_prob)

# Inspect
counterfactuals
#> # A tibble: 2 x 2
#>   path  conversion_prob
#>   <chr>           <dbl>
#> 1 A > C          0.0239
#> 2 C > A          0.0192
```

See that `conversion_prob('A > C') > baseline_conversion_prob` and `conversion_prob('C > A') > baseline_conversion_prob` so the marginal contribution of both **C**s in the path **C** > **A** > **C** is negative. Thus we set the attribution fraction for **C** to **0**.

As explained, the non-normalized fractions are such:

```r
fractribution.model:::fractional_values(
  baseline_path,
  baseline_conversion_prob,
  all_paths = path_summary,
  normalize = FALSE,
  path_transform_method = 'exposure')
#> # A tibble: 1 x 3
#>   path           A     C
#>   <chr>      <dbl> <dbl>
#> 1 C > A > C 0.0172     0
```

When this is normalized, **A** in fact gets 100% of the credit:

```r
fractribution.model:::fractional_values(
  baseline_path,
  baseline_conversion_prob,
  all_paths = path_summary,
  normalize = TRUE,
  path_transform_method = 'exposure')
#> # A tibble: 1 x 3
#>   path           A     C
#>   <chr>      <dbl> <dbl>
#> 1 C > A > C     1     0
```

All marginal contributions are negative

Extending from above, there can be cases where ALL the marginal contributions for the **baseline_path** will be negative, and so all events are given **0** fractional attribution.

To mitigate this issue, in these cases, we just give the last touch in the path the full contribution (last-touch attribution).

This issue is most likely to occur when `path_transform_method = 'unique'` so let's set this up:

```r
# Transform paths and calculate conversion probs
path_summary <- fractribution.model:::unique_path_summary_transform(example_path_summary)
```

For example the path **D** > **D** > **D** has **baseline_conversion_prob** equal to **0.03592814**:

```
baseline_path <- 'D > D > D'
baseline_conversion_prob <- path_summary %>%
  filter(path == baseline_path) %>%
  pull(conversion_prob)

# Inspect
baseline_conversion_prob
#> [1] 0.03592814
```

Which is less than all the counterfactuals (note here both counterfactuals are **D > D**):

```
# The path length, is the number of counterfactuals
drop_indicies <- fractribution.model:::path_length(baseline_path)

# Counterfactuals (drop an event at each index)
counterfactuals <- purrr::map_chr(
  1:drop_indicies,
  ~ fractribution.model:::drop_event(baseline_path, .))

# Get conversion probability
counterfactuals <- path_summary %>%
  filter(path %in% counterfactuals) %>%
  select(path, conversion_prob)

# Inspect
counterfactuals
#> # A tibble: 1 x 2
#>   path  conversion_prob
#>   <chr>           <dbl>
#> 1 D > D          0.0465
```

So the attribution goes all to the last-event, here **D**:

```
fractribution.model:::fractional_values(
  baseline_path,
  baseline_conversion_prob,
  all_paths = path_summary,
  normalize = TRUE,
  path_transform_method = 'exposure')
#> # A tibble: 1 x 2
#>   path          D
#>   <chr>     <dbl>
#> 1 D > D > D     1
```

Path transform methods

The `attribution_fit()` function employs a **path_transform_method** argument to instruct how to transform paths before conducting the counterfactual search during the attribution fit (as shown above).

This is helpful because raw paths to conversion can be messy, especially if your lookback period is long or you have many different channels defined.

We have five path transforms available:

- **unique**: treat all events in a path as unique.
- **exposure**: collapse repeat events that are immediately in sequence.

- **first**: take only the first occurance of any given event.
- **recency**: look at where the event occured in the timeline before conversion and: treat the same events differently if they occur in different time buckets; whereas collapse events if they are within the same bucket. The buckets are (in days), `{1, 2, 3-4, 5-7, 8-14, 15-30}`.
- **frequency**: count events from their first occurance.

Depending on your marketing strategy you might favour one or the other. Additionally, you can run all methods and average the fractions across some weightings you specify, it's up to you.

Same path, different transforms

To give a concrete example, let's consider the following path $\mathbf{A > A > B > C > B}$. Let's put it in a dataframe that we can work with (note the difference in recency notation):

```
# Create a tbl to display results nicely
transforms <- tibble(path_transform = c("unique", "exposure", "first",
                                        "recency", "frequency"),
                     initial_path = c(rep('A > A > B > C > B', 3),
                                      'A(15-30) > A(5-7) > B(1) > C(1) > B(1)',
                                      'A > A > B > C > B'))

# Inspect
transforms
#> # A tibble: 5 x 2
#>   path_transform initial_path
#>   <chr>          <chr>
#> 1 unique         A > A > B > C > B
#> 2 exposure       A > A > B > C > B
#> 3 first          A > A > B > C > B
#> 4 recency        A(15-30) > A(5-7) > B(1) > C(1) > B(1)
#> 5 frequency      A > A > B > C > B
```

Now each transform yields as follows:

```
# Define a quick transform routing function
transform_path <- function(path, path_transform_method) {
  transformed_path <- switch(path_transform_method,
    unique = path,
    exposure = fractribution.model:::collapse_sequential_repeats(path),
    first = fractribution.model:::collapse_all_repeats(path),
    recency = fractribution.model:::collapse_all_repeats(path),
    frequency = fractribution.model:::collapse_by_count(path)
  )

  return(transformed_path)
}

# Add to our tbl
transforms <- transforms %>%
  mutate(transformed_path = purrr::map2_chr(initial_path, path_transform,
                                            ~ transform_path(.x, .y)))
# Inspect
transforms
#> Warning in seq.default(along = x): partial argument match of 'along' to
#> 'along.with'
```

```
#> # A tibble: 5 x 3
#>   path_transform initial_path                    transformed_path
#>   <chr>          <chr>                           <chr>
#> 1 unique         A > A > B > C > B                A > A > B > C > B
#> 2 exposure       A > A > B > C > B                A > B > C > B
#> 3 first          A > A > B > C > B                A > B > C
#> 4 recency        A(15-30) > A(5-7) > B(1) > C(1~ A(15-30) > A(5-7) > B(1) ~
#> 5 frequency      A > A > B > C > B                A(2) > B(2) > C(1)
```

Make note of the slight differences in the **path_transform_method**s.

The recency row is cut off above, so here it is in its entirety:

```
transforms %>% filter(path_transform == 'recency') %>% glimpse()
#> Observations: 1
#> Variables: 3
#> $ path_transform   <chr> "recency"
#> $ initial_path     <chr> "A(15-30) > A(5-7) > B(1) > C(1) > B(1)"
#> $ transformed_path <chr> "A(15-30) > A(5-7) > B(1) > C(1)"
```

Different transforms, same result

There are many cases where you will get the **same resulting path** despite using **different transforms**. This occurs across the **unique**, **exposure**, and **first** methods.

Here are some examples—

Exposure and first the same:

```
path <- 'A > A > B > C'

# Exposure method
fractribution.model:::collapse_sequential_repeats(path)
#> [1] "A > B > C"

# First method
fractribution.model:::collapse_all_repeats(path)
#> [1] "A > B > C"
```

Unique, exposure, and first the same:

```
path <- 'A > B > C'

# Unique method
path
#> [1] "A > B > C"

# Exposure method
fractribution.model:::collapse_sequential_repeats(path)
#> [1] "A > B > C"

# First method
fractribution.model:::collapse_all_repeats(path)
#> [1] "A > B > C"
```

Different input, same transform, same result

There are also many cases where, using a **single path transform** method, you will get the **same resulting path** despite having different input paths. This occurs across the **exposure**, **first**, **recency**, and **frequency** methods.

Here are some examples—

Frequency and first method:

```
# Inputs
input_1 <- 'A > B > A'
input_2 <- 'A > A > B'

# Same via frequency method
fractribution.model:::collapse_by_count(input_1)
#> [1] "A(2) > B(1)"

fractribution.model:::collapse_by_count(input_2)
#> [1] "A(2) > B(1)"

# Same via first method
fractribution.model:::collapse_all_repeats(input_1)
#> [1] "A > B"

fractribution.model:::collapse_all_repeats(input_2)
#> [1] "A > B"
```

Exposure method:

```
# Inputs
input_1 <- 'A > B > C'
input_2 <- 'A > A > B > C > C'

# Same via exposure method
fractribution.model:::collapse_sequential_repeats(input_1)
#> [1] "A > B > C"

fractribution.model:::collapse_sequential_repeats(input_2)
#> [1] "A > B > C"
```

Recency method:

```
# Inputs
input_1 <- 'A(15-30) > A(5-7) > B(1) > C(1) > B(1)'
input_2 <- 'A(15-30) > A(15-30) > A(5-7) > B(1) > C(1) > B(1) > C(1)'

# Same via recency method
fractribution.model:::collapse_all_repeats(input_1)
#> [1] "A(15-30) > A(5-7) > B(1) > C(1)"

fractribution.model:::collapse_all_repeats(input_2)
#> [1] "A(15-30) > A(5-7) > B(1) > C(1)"
```

Comparing fractions across path transforms

Your choice of **path_transform_method** will affect the final results. It's hard to anticipate the effects so we suggest trying all methods and comparing the results. Ultimately you should choose one (or a combination) that best suits your marketing strategy.

To give you insight into the differences, here's an example where we map across all the **path_transform_method**'s (not recency here as data input structure is different).

So we can fit all the output within the code blocks we filter to just consider paths starting with **B** or **A**; ending with **B** or **A**; and only containing **B** or **A**:

```
ab_paths <- example_path_summary %>%
  filter(stringr::str_detect(path, '^(A > |B > )(A > |B > )*(A|B)$'))
```

Map and fit fractribution for all methods:

```
path_transform_methods <- c("unique", "exposure", "first", "frequency")

attribution_models <- purrr::map(
  path_transform_methods,
  ~ attribution_fit(ab_paths,
                    path_transform_method = .x,
                    path_level_only = TRUE))

names(attribution_models) <- path_transform_methods
```

Now let's inspect the results.

Unique will not reduce any paths, so let's just look at **A > B** and **B > A**:

```
attribution_models$unique %>%
  filter(path %in% c('A > B', 'B > A')) %>%
  select(path, a, b)
#> # A tibble: 2 x 3
#>   path      a     b
#>   <chr> <dbl> <dbl>
#> 1 B > A   0.5   0.5
#> 2 A > B   0     1
```

Exposure collapses down to just **9** paths:

```
attribution_models$exposure %>%
  select(path, a, b)
#> # A tibble: 9 x 3
#>   path                  a     b
#>   <chr>             <dbl> <dbl>
#> 1 A                 1     0
#> 2 B                 0     1
#> 3 B > A             0.503 0.497
#> 4 A > B             0     1
#> 5 A > B > A         1     0
#> 6 A > B > A > B     0     1
#> 7 B > A > B         0     1
#> 8 B > A > B > A     1     0
#> 9 B > A > B > A > B 0     1
```

First collapses down further to just **4** paths:

```
attribution_models$first %>%
  select(path, a, b)
#> # A tibble: 4 x 3
#>   path      a     b
#>   <chr> <dbl> <dbl>
#> 1 A     1     0
#> 2 B     0     1
#> 3 B > A 0.504 0.496
#> 4 A > B 0     1
```

Frequency has many paths, so let's just consider forms of **B > A**:

```
attribution_models$frequency %>%
  filter(stringr::str_detect(path, '^B\\([0-9]*\\) > A\\(1\\)$')) %>%
  select(path, a, b)
#> # A tibble: 6 x 3
#>   path            a     b
#>   <chr>       <dbl> <dbl>
#> 1 B(1) > A(1) 0.5   0.5
#> 2 B(2) > A(1) 0.468 0.532
#> 3 B(3) > A(1) 0.495 0.505
#> 4 B(4) > A(1) 1     0
#> 5 B(5) > A(1) 1     0
#> 6 B(6) > A(1) 1     0
```

Make note of the slight differences across all the methods.