Fractribution Model - Quick start

Daniel Booth

2019-01-14

Overview

The Fractribution framework enables you to derive user-level fractional attribution values from your marketing and event touchpoints data. This package, **fractribution.model**, fits a customisable user-level fractional attribution model using a simplified Shapley Value method. Additionally, and for comparison purposes, a last-touch attribution function is also implemented in the package.

In this vignette we will explore the core functions in the package so you are able to get started quickly.

Examples

Load package

First load the package. We will also load `dplyr` to use in some of the examples:

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

Paths to conversion data

The `attribution_fit()` function requires a `path_summary` input which needs to be created from your marketing data (such as Google Analytics sessions). Additionally, if you want to produce the attribution at the customer-level, you need to also include a `path_customer_map` input.

You can produce this data either through your own data engineering, or through the **fractribution.data** package (see `?fractribution.data::run_attribution_report` for more details).

For now, for your reference, we have included examples of these two input files in the package and we explain these below.

Path summary

The `path_summary` input is a dataframe with columns `path`, `total_conversions`, and `total_non_conversions`. The last two columns are the `path`'s aggregated conversion and non-conversion counts.

For your reference there is an `example_path_summary` dataframe loaded with the package:

```
example_path_summary
#> # A tibble: 758 x 3
#>    path      total_conversions total_non_conversions
#>    <chr>                 <dbl>                 <int>
#>  1 B                       596                 36462
#>  2 A                       477                135201
#>  3 C                       250                 40727
#>  4 D                       182                  6355
#>  5 A > A                    87                 13648
#>  6 B > B                    46                  4515
#>  7 C > C                    42                  3586
```

```
#>  8 D > D                    26                  533
#>  9 A > A > A                22                 3187
#> 10 H                        19                  585
#> # ... with 748 more rows
```

Path customer map

The `path_customer_map` input is is a dataframe mapping from `path` to `customer_id`.

Again there is a reference, `example_path_customer_map`, loaded with the package:

```
example_path_customer_map
#> # A tibble: 1,983 x 2
#>    path                                          customer_id
#>    <chr>                                         <chr>
#>  1 F                                             cid_1
#>  2 F                                             cid_2
#>  3 F                                             cid_3
#>  4 A > A > A > A > A > A > A > A > A > A > A > A  cid_4
#>  5 A > A > A > A > A > A > A > A > A > A > A > A  cid_5
#>  6 B                                             cid_6
#>  7 B                                             cid_7
#>  8 B                                             cid_8
#>  9 B                                             cid_9
#> 10 B                                             cid_10
#> # ... with 1,973 more rows
```

Attribution fit

The `attribution_fit()` function runs through the counterfactual shapley value algorithm to fit an attribution model for the paths and their conversion probabilities.

As mentioned above, you can produce the attribution at the `customer_id` or `path` level. To control this use the `path_level_only` argument, which will default to `FALSE` (i.e. the customer-level report is produced).

To run the attribution report, use the `attribution_fit()` function, for example:

```
fractional_attribution <- attribution_fit(example_path_summary,
                                          example_path_customer_map)

# Inspect some customers
fractional_attribution %>%
  filter(customer_id %in% c('cid_1644', 'cid_1683', 'cid_1755'))
#> # A tibble: 3 x 13
#>   path  customer_id     b     a     c     d     h     g     i     f     e
#>   <chr> <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 A > C cid_1755    0     0.462 0.538     0     0     0     0     0     0
#> 2 B > A cid_1683    0.553 0.447 0         0     0     0     0     0     0
#> 3 B > C cid_1644    0.585 0     0.415     0     0     0     0     0     0
#> # ... with 2 more variables: j <dbl>, k <dbl>
```

See that for the customer with **cid_1683**, who had the **B > A** path to conversion, fractribution has attributed **55.3%** of the conversion to **channel B** and **44.7%** to **channel A**.

Path level only

If you just want to fractional values at the path- (not customer-) level, set `path_level_only = TRUE`:

```
path_fracs <- attribution_fit(example_path_summary,
                              path_level_only = TRUE)

# Inspect some paths
path_fracs %>%
  filter(path %in% c('B', 'A', 'B > C', 'B > D', 'B > D > G', 'D > C > A')) %>%
  select(path:g)
#> # A tibble: 6 x 7
#>   path          b     a     c     d     h     g
#>   <chr>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 B         1     0     0     0         0 0
#> 2 A         0     1     0     0         0 0
#> 3 B > C     0.585 0     0.415 0         0 0
#> 4 B > D     0.405 0     0     0.595     0 0
#> 5 B > D > G 0.393 0     0     0.262     0 0.345
#> 6 D > C > A 0     0.305 0.330 0.365     0 0
```

Notice how the value of **D** in the path **B > D** (**59.5%**) reduces significantly when a **G** is added to the path (i.e. **B > D > G**). That is, **D**'s value drops to **26.2%**, whereas **B**'s value remains very stable (**40.5%** to **39.3%**).

Path transform method

Hidden in the two examples above was the default `path_transform_method = 'exposure'` argument. Raw paths to conversion can be messy, especially if your lookback period is long or you have many different channels defined. Thus it is effective to transform paths before conducting the counterfactual search during the attribution fit. We have five options for this: **unique**, **exposure**, **first**, **recency**, and **frequency**, each with various benefits specific to the use case. See `?attribution_fit` for details on each.

To demonstrate how the `path_transform_method` will change the attribution fit, here's the same input data from above, but using the **first** path transform instead (which could be more appropriate for a brand awareness type marketing strategy):

```
# 'first' path transform
first_fit <- attribution_fit(example_path_summary,
                             example_path_customer_map,
                             path_transform_method = 'first')

# Inspect some customers
first_fit %>%
  filter(customer_id %in% c('cid_1644', 'cid_1683', 'cid_1755'))
#> # A tibble: 3 x 13
#>   path  customer_id     b     a     c     d     h     g     i     f     e
#>   <chr> <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 A > C cid_1755    0     0.472 0.528     0     0     0     0     0     0
#> 2 B > A cid_1683    0.554 0.446 0         0     0     0     0     0     0
#> 3 B > C cid_1644    0.586 0     0.414     0     0     0     0     0     0
#> # ... with 2 more variables: j <dbl>, k <dbl>
```

Notice how the attribution fractions have changed slightly. This change could be more dramatic in your own data.

Reporting

Now that we can get the attribution fractions at the customer-level, we will typically want to join in revenue information and roll up to a higher-level channel report.

Fractribution supports this with the `channel_attribution_report()` and `channel_revenue_attribution_report()` functions, as well as a `last_touch_attribution()` function for comparison purposes. We'll explore these below.

Channel Attribution

`channel_attribution_report()` will take the path-level attribution fractions fit in `attribution_fit()` and roll up to a channel level report. For example if we use our `fractional_attribution` fit from before, we get the following:

```
fractribution <- channel_attribution_report(fractional_attribution)

# Inspect
fractribution
#> # A tibble: 11 x 2
#>    channel attributed_conversions
#>    <chr>                    <dbl>
#>  1 b                          683
#>  2 a                          644
#>  3 c                          344
#>  4 d                          255
#>  5 h                           25
#>  6 g                           17
#>  7 i                            8
#>  8 f                            5
#>  9 e                            2
#> 10 j                            0
#> 11 k                            0
```

So here we can say channel **b** was attributed **683** conversions, etc.

It's also a good sanity check to confirm that the fractribution `attribution_conversions` and `path_summary$total_conversions` column sums are equal:

```
# Attributed conversions
sum(fractribution$attributed_conversions)
#> [1] 1983

# Original paths to conversion
sum(example_path_summary$total_conversions)
#> [1] 1983
```

Combining Last touch attribution

For comparison we include a `last_touch_attribution()` function. We can aggregate last-touch attribution with the following:

```
last_touch <- last_touch_attribution(example_path_summary)

# Inspect
last_touch
```

```
#> # A tibble: 11 x 2
#>    last_channel last_touch_conversions
#>    <chr>                         <dbl>
#>  1 B                               655
#>  2 A                               648
#>  3 C                               361
#>  4 D                               258
#>  5 H                                28
#>  6 G                                17
#>  7 I                                 8
#>  8 F                                 6
#>  9 E                                 2
#> 10 J                                 0
#> 11 K                                 0
```

Note this also includes the same `path_transform_method` argument which you can tweak depending on your use case. It defaults to **unique** by which we get the actual last touch.

We can join this to the fractribution channel fractions using the following (note there is a little processing needed on channel first):

```
# Clean up last-touch channel name
library(stringr)
last_touch <- last_touch %>%
    mutate(last_channel = str_to_lower(last_channel) %>%
      str_replace_all(' - ', ' ') %>%
      str_replace_all('-', ' ') %>%
      str_replace_all(' ', '_'))

# Join to fractribution
channel_report <- fractribution %>%
  inner_join(last_touch, by = c('channel' = 'last_channel')) %>%
  mutate(difference = attributed_conversions - last_touch_conversions)

# Inspect
channel_report
#> # A tibble: 11 x 4
#>    channel attributed_conversions last_touch_conversions difference
#>    <chr>                   <dbl>                  <dbl>      <dbl>
#>  1 b                         683                    655         28
#>  2 a                         644                    648         -4
#>  3 c                         344                    361        -17
#>  4 d                         255                    258         -3
#>  5 h                          25                     28         -3
#>  6 g                          17                     17          0
#>  7 i                           8                      8          0
#>  8 f                           5                      6         -1
#>  9 e                           2                      2          0
#> 10 j                           0                      0          0
#> 11 k                           0                      0          0
```

You can now see which channels are shown to improve in value in fractribution over last-touch attribution. For example **28** additional conversions were granted to **b** (last-touch undervaluing). Also **17** conversions were removed from **c** (last-touch overvaluing), etc.

Adding a default conversion value

Next, you might have some default conversion value you can assign to estimate an attributed revenue:

```
# Set default conversion value
conversion_val <- 70

# Distribute
channel_report_default_rev <- channel_report %>%
  mutate(attributed_revenue = attributed_conversions * conversion_val,
         last_touch_revenue = last_touch_conversions * conversion_val) %>%
  select(attributed_conversions, attributed_revenue,
         last_touch_conversions, last_touch_revenue)

# Inspect
channel_report_default_rev
#> Warning in seq.default(along = x): partial argument match of 'along' to
#> 'along.with'

#> Warning in seq.default(along = x): partial argument match of 'along' to
#> 'along.with'
#> # A tibble: 11 x 4
#>    attributed_conver~ attributed_reven~ last_touch_conver~ last_touch_reve~
#>                 <dbl>             <dbl>              <dbl>            <dbl>
#>  1                683             47810                655            45850
#>  2                644             45080                648            45360
#>  3                344             24080                361            25270
#>  4                255             17850                258            18060
#>  5                 25              1750                 28             1960
#>  6                 17              1190                 17             1190
#>  7                  8               560                  8              560
#>  8                  5               350                  6              420
#>  9                  2               140                  2              140
#> 10                  0                 0                  0                0
#> 11                  0                 0                  0                0
```

Note we don't recommend this approach but *instead* suggest using the actual revenue if possible (see the next section).

Channel report with revenue and ROAS

Attributing a default, single, conversion value to all customers is suboptimal if you know—from your own transaction systems—what the actual conversion was for each customer.

If you do know this revenue, you can get the most leverage from fractribution's customer-level output.

The `channel_revenue_attribution_report()` function will facilitate the process for you.

It takes two additional inputs: `conversion_revenue` and (optionally) `channel_spend`. Including `channel_spend` means you will also get a **ROAS** calculation in the report.

Like with `attribution_fit()` there are example datasets loaded with the package. We'll explore these below and then show the report function.

Conversion revenue

The `conversion_revenue` input is a dataframe with columns `customer_id` and `conversion_value`. The `customer_id`s are from `path_customer_map` and the `conversion_value`s are the currency value for each customer's conversion.

For your reference there is an `example_conversion_revenue` dataframe loaded with the package:

```
example_conversion_revenue
#> # A tibble: 1,983 x 2
#>    customer_id conversion_value
#>    <chr>                  <dbl>
#>  1 cid_1                  151.
#>  2 cid_2                  102.
#>  3 cid_3                  234.
#>  4 cid_4                   35.1
#>  5 cid_5                   72.6
#>  6 cid_6                  194.
#>  7 cid_7                  159.
#>  8 cid_8                  261.
#>  9 cid_9                  218.
#> 10 cid_10                 187.
#> # ... with 1,973 more rows
```

Channel spend

The `channel_spend` input is a dataframe with columns `channel` and `total_spend` to use for ROAS calculations. Each channel that appears in a path should have a record, with the `total_spend` being the amount spent on the channel during the reporting period. If the channel is non-marketing (e.g. organic search) set the `total_spend` to `NA`.

Again there is a reference, `example_channel_spend`, loaded with the package:

```
example_channel_spend
#> # A tibble: 9 x 2
#>   channel total_spend
#>   <chr>         <dbl>
#> 1 A            27916.
#> 2 B               NA
#> 3 C               NA
#> 4 D            10649.
#> 5 H              900
#> 6 G              800
#> 7 I              988.
#> 8 F             1649.
#> 9 E              200
```

Attributed Revenue and ROAS

Finally we can calculate attributed revenue and ROAS. To do this use the `channel_revenue_attribution_report()` function:

```
fractribution_and_roas <-
  channel_revenue_attribution_report(fractional_attribution,
                                     example_conversion_revenue,
                                     example_channel_spend)

# Inspect
```

```
fractribution_and_roas
#> # A tibble: 9 x 5
#>   channel attributed_conversions attributed_revenue total_spend   roas
#>   <chr>                   <dbl>              <dbl>       <dbl>  <dbl>
#> 1 b                         683             102637          NA  NA
#> 2 a                         644              99309       27916.  3.56
#> 3 c                         344              50897          NA  NA
#> 4 d                         255              38269       10649.  3.59
#> 5 h                          25               3595         900   3.99
#> 6 g                          17               2489         800   3.11
#> 7 i                           8               1134         988.  1.15
#> 8 f                           5                881        1649.  0.534
#> 9 e                           2                397         200   1.98
```

Run all path transform methods for comparison

If you want to run all path transform methods and compare them you can do something like the following code examples.

Compare attribution fits

Map through each `path_transform_method` and collect `attribution_fit()`s in a list:

```
# Comparing the different methods
path_transform_methods <- c("unique", "exposure", "first", "frequency")

attribution_models <- purrr::map(
  path_transform_methods,
  ~ attribution_fit(example_path_summary,
                    example_path_customer_map,
                    .x))

names(attribution_models) <- path_transform_methods

# Inspect
attribution_models
```

Comparing channel reports

Continuing from above, pass the list of fits (`attribution_models`) into the `channel_attribution_report()` function, again collecting the results in a list:

```
# See the final reports
channel_reports <- purrr::map(attribution_models,
                              channel_attribution_report)

# Inspect
channel_reports
```

Comparing last-touch attribution reports

Similarly to above, we can also map through `last_touch_attribution()` for each `path_transform_method`:

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

last_touch_reports <- purrr::map(
  path_transform_methods,
  ~ last_touch_attribution(
    example_path_summary,
    .x))

names(last_touch_reports) <- path_transform_methods

# Inspect
last_touch_reports
```

Additional details

If you would like to discover more about the actual fractribution algorithm, there is an **Attribution fit - Detailed overview** vignette included in the package which will go into more detail, as well as explore more on the differences between the `path_transform_methods`. To launch run:

```
vignette('attribution_fit_details')
```

Bugs and features

fractribution.model is a work in progress and so you might find some issues. If you do, please let me know at dbooth@google.com and I'll try fix it asap!

Also if you find you need an additional feature please reach out to me and I can scope and, if I think it's suitable, add into a future release.