AMSS version 1.0

Google, Inc.

2017-06-06

Contents

Aggregate Marketing System Simulator: an introduction

What is AMSS?

The Aggregate Marketing System Simulator (AMSS) is a simulation tool capable of generating aggregate-level time series data related to marketing measurement (e.g., channel-level marketing spend, website visits, competitor spend, pricing, sales volume). It is flexible enough to model a wide variety of marketing situations that include different mixes of advertising spend, levels of ad effectiveness, types of ad targeting, sales seasonality, competitor activity, and much more.

Additionally, AMSS generates ground truth for marketing performance metrics, including return on advertising spend (ROAS) and marginal ROAS (mROAS). The capabilities provided by AMSS create a foundation for evaluating and improving measurement methods, including media mix models (MMMs), campaign optimization, and geo experiments, across complex modeling scenarios.

AMSS for media mix modeling

MMMs are a typical area of application for AMSS. MMMs are statistical models built on aggregate historical marketing data to

- Estimate the impact of various media strategies (the media mix) on revenue or some other key performance indicator (KPI).
- Forecast the impact of future media strategies on the KPI.
- Recommend media strategies to optimize future KPIs.

Because they are built on aggregate historical data, making causal conclusions based on MMM analyses is fraught with difficulties. Sources of validation for MMM methodologies are scarce; experimental validation is often too expensive or impractical to run. AMSS-based model evaluation allows analysts to cheaply test MMM methodology performance in an environment where the ground truth is known.

Vignette overview

This vignette shows briefly how to use the R package `amss`

- to generate simulated data
- to calculate ground truth for marketing metrics

For details on specific functions, please refer to the package manual.

A detailed discussion of AMSS's design is available in our paper at https://research.google.com (see reference). The code below is based on the example scenario family from the paper.

Background: AMSS's simulation model

AMSS operates by tracking a population of consumers' relationships to the product category and the brand over time. The state of the population at any time is summarized by segmenting the population into groups composed of individuals with the same mindset. It is important to understand how AMSS conceptualizes the consumer mindset before using the simulator.

Consumer relationships with the category and the brand are summarized as discrete latent variables in six dimensions. The first three describe the consumer's relationship with the category:

- **Market state**: whether the consumer has interest in the product category. Consumers may be either 'in-market' or 'out-of-market.'
- **Satiation state**: whether the consumer's interest in the product category has been (temporarily) satiated by a recent purchase. Consumers may be either 'satiated' or 'unsatiated.'
- **Activity state**: where the consumer is along the path to making a purchase in the category. Consumers may be in the 'inactive,' 'exploratory,' or 'purchase' activity state. Individuals in either of the latter two states must be both 'in-market' and 'unsatiated.'

The last three describe the consumer's brand relationships:

- **Brand favorability state**: the consumer's opinion and awareness of the advertiser's brand. Consumers may be 'unaware,' 'negative,' 'neutral,' 'somewhat favorable,' or 'favorable.'
- **Brand loyalty state**: the consumer's brand loyalty. The consumer may be a 'switcher,' 'loyal' to the advertiser's brand, or 'competitor-loyal.' Only 'favorable' consumers can be 'loyal.'
- **Brand availability state**: The phyical or mental availability of the advertiser's brand, from the perspective of the consumer. For example, if 25% of consumers shop in grocery stores that do not carry the advertiser's brand, then the 25% of the population should be in the 'low' brand availability state (the other 75% would be either 'average' or 'high').

The package constants `kMarketStates`, `kSatiationStates`, `kActivityStates`, `kFavorabilityStates`, `kLoyaltyStates`, and `kAvailabilityStates` store the potential values users may take in any particular dimension of the population segmentation. The `data.table` `kAllStates` lists all possible combinations of the six dimensions, for a total of 198 consumer states.

Simulating data from a single scenario

Attaching the package

```
library(amss)
```

Natural behavior, in the absence of marketing interventions

To begin with, we specify the number of time intervals in the simulation. Here, we ask for 4 years of weekly data.

```
n.years <- 4
time.n <- n.years * 52
```

In every simulation, the user must specify the natural behavior of the consumer population in the absence of marketing interventions.

For example, consumer mindset will naturally change over time. We specify below the natural transitions in activity state and brand favorability state that occur at the beginning of each time interval. These transitions occur marginally. When transition matrices in a dimension (e.g., brand availability) are unspecified, the default specifies no migration.

Individuals from all activity states have a 60%/30%/10% chance of entering the inactive/exploratory/purchase states at the beginning of each time interval. Note that because the transition matrices below are row-identical, the population segmentation at the end of the previous time interval has no effect on the population segmentation after these transition matrices have been applied. This means there is no 'lagged effect,' i.e., deviations from the stable distribution caused by marketing interventions will not persist beyond the time interval during which the intervention occurs.

```
activity.transition <- matrix(
    c(0.60, 0.30, 0.10,  # migration originating from inactive state
      0.60, 0.30, 0.10,  # exploratory state
      0.60, 0.30, 0.10),  # purchase state
    nrow = length(kActivityStates), byrow = TRUE)
favorability.transition <- matrix(
    c(0.03, 0.07, 0.65, 0.20, 0.05,  # migration from the unaware state
      0.03, 0.07, 0.65, 0.20, 0.05,  # negative state
      0.03, 0.07, 0.65, 0.20, 0.05,  # neutral state
      0.03, 0.07, 0.65, 0.20, 0.05,  # somewhat favorable state
      0.03, 0.07, 0.65, 0.20, 0.05),  # favorable state
    nrow = length(kFavorabilityStates), byrow = TRUE)
```

The user also specifies seasonal changes in the size of the market for the relevant product category.

```
# a sinusoidal pattern
market.rate.nonoise <-
    SimulateSinusoidal(n.years * 52, 52,
                       vert.trans = 0.6, amplitude = 0.25)
# with some added noise
```

```
market.rate.seas <- pmax(
    0, pmin(1,
            market.rate.nonoise *
                SimulateAR1(length(market.rate.nonoise), 1, 0.1, 0.3)))
```

The functions `SimulateSinusoidal()`, `SimulateDummy()`, `SimualteAR1()`, and `SimulateCorrelated()` are provided for the user's convenience. They are useful in situations such as the one above, where the user wishes to generate a particular type of time series as part of the simulation setup.

The complete set of parameters controlling consumer mindset in the absence of marketing interventions is specified as a list below.

```
nat.mig.params <- list(
    population = 2.4e8,
    market.rate.trend = 0.68,
    market.rate.seas = market.rate.seas,
    # activity states for newly responsive (in-market & un-satiated)
    prop.activity = c(0.375, 0.425, 0.2),
    # brand favorability, initial proportions.
    prop.favorability = c(0.03, 0.07, 0.65, 0.20, 0.05),
    # everyone is a switcher
    prop.loyalty = c(1, 0, 0),
    transition.matrices = list(
        activity = activity.transition,
        favorability = favorability.transition))
```

Media channels

The next step is to specify the behavior of any marketing interventions included in the simulation. Currently, AMSS focuses on modeling the effects of media advertising through the functions `DefaultTraditionalMediaModule()` and `DefaultSearchMediaModule()`. Marketing interventions whose behavior cannot be modeled accurately through either of these functions can be added to a simulation through custom functions written by the user.

Before specifying the behavior of any media channel in particular, we set a common budget cycle for all media channels. Here, budgets are determined on a yearly basis, so each week within a budget period of 52 consecutive weeks is assigned the same identifier `budget.index`.

```
budget.index <- rep(1:n.years, each = 52)
```

Functions such as `CalculateROAS()` determine the effect of changing the budget for a specified set of budget periods, so `budget.index` should be carefully specified according to the needs of the user. The (quarterly, yearly) 'budget' is considered the target quantity the advertiser is trying to optimize or dertermine the value of.

A traditional media channel: TV

Below, we specify the behavior of television advertising. TV is modeled as a traditional media format using `DefaultTraditionalMediaModule()`.

The user should specify a weekly flighting pattern for TV spend.

```
tv.flighting <-
    pmax(0,
         market.rate.seas +
```

```
        SimulateAR1(length(market.rate.seas), -0.7, 0.7, -0.7))
tv.flighting <- tv.flighting[c(6:length(tv.flighting), 1:5)]
```

TV causes changes in consumer mindset by increasing brand awareness and brand favorability.

```
tv.activity.trans.mat <- matrix(
    c(1.00, 0.00, 0.00,  # migration originating from the inactive state
      0.00, 1.00, 0.00,  # exploratory state
      0.00, 0.00, 1.00),  # purchase state
    nrow = length(kActivityStates), byrow = TRUE)
tv.favorability.trans.mat <- matrix(
    c(0.4,  0.0,  0.4, 0.2, 0.0,  # migration from the unaware state
      0.0,  0.9,  0.1, 0.0, 0.0,  # negative state
      0.0,  0.0,  0.6, 0.4, 0.0,  # neutral state
      0.0,  0.0,  0.0, 0.8, 0.2,  # somewhat favorable state
      0.0,  0.0,  0.0, 0.0, 1.0),  # favorable state
    nrow = length(kFavorabilityStates), byrow = TRUE)
```

The complete list of arguments for the traditional media module includes arguments controlling audience membership (i.e., tv viewership), the budget, the cost per exposure, and the relationship between ad frequency and efficacy. See `DefaultTraditionalMediaModule()` for more details.

```
params.tv <- list(
    audience.membership = list(activity = rep(0.4, 3)),
    budget = rep(c(545e5, 475e5, 420e5, 455e5), length = n.years),
    budget.index = budget.index,
    flighting = tv.flighting,
    unit.cost = 0.005,
    hill.ec = 1.56,
    hill.slope = 1,
    transition.matrices = list(
        activity = tv.activity.trans.mat,
        favorability = tv.favorability.trans.mat))
```

這邊的意思是 activity multiplier for (“inactive”, “exploratory”, “purchase”)，所以(0.4, 0.4, 0.4)意思是0.4 * all segments with activity status “purchase”

Paid search

Paid search is modeled using the function `DefaultSearchMediaModule()`. The complete list of parameters for the search media module includes arguments controlling

- who makes queries and how many queries they make
- the relationship between the budget assigned to paid search and the advertiser's auction settings (spend cap, bid, keyword list)
- the cost per click
- the click through rate
- the effectiveness of search in changing consumer mindset

The full specification can be found in the manual entry for `DefaultSearchMediaModule()`.

Some settings are relatively simple to specify. For example, paid search runs in an auction environment. The competitive landscape determines the minimum and maximum cost per click (CPC) for the advertiser. This may be specified as a vector if it varies over time, or as a single numeric value.

```
cpc.min <- 0.8
cpc.max <- 1.1
```

The settings an advertiser uses to control a paid search campaign are more complicated. An advertiser controls a paid search campaign in three ways:

- By adjusting a 'spend cap' that puts a hard limit on the amount to be spent during a time interval.
- By adjusting the keyword list, and thus, the set of queries the advertiser is bidding on.
- By adjusting the bid submitted to the auction.

Like `DefaultTraditionalMediaModule()`, the function `DefaultSearchMediaModule()` allows specification of budget periods and their associated budgets. The settings for paid search are then specified as a function of the budget. For example, in this scenario we specify the spend cap as follows:

```
# uncapped spend, shut off the first 2 of every 13 weeks
spend.cap.fn <- function(time.index, budget, budget.index) {
  if ((time.index %% 13) > 1) {
    return(Inf)
  } else {
    return(0)
  }
}
```

In this case, the spend cap function specifies that the paid search campaign is on during 11 out of every 13 weeks, and when it is on the spend is uncapped. This remains true no matter the budget assigned to the media channel. The bid also doesn't depend on the budget, and is fixed at 1.1 (the maximum CPC).

```
bid.fn <- function(time.index, per.capita.budget, budget.index) {
  return(1.1)
}
```

The relationship between budget and spend is driven by the function `kwl.fn()`. Like `bid.fn()`, it is written as a function of the per capita budget, i.e., the budget divided by the size of the population, so that the behavior of the paid search module scales to the size of the population.

```
kwl.fn <- function(time.index, per.capita.budget, budget.index) {
    return(4.5 * per.capita.budget)
}
```

The function returns the proportion of queries that match the advertiser's keyword list. When it returns a vector, the proportion is specified for each population segment individually. This allows more complex scenarios where the quality of the targeting of the keyword list varies as it changes size.

As the budget increases, the keyword list expands and the proportion of queries that match also grows. This eventually results in more spend in paid search. Generally, a precise match between budget and spend is not necessary. Budget is a tool that allows the user to specify the mechanism an advertiser uses to increase its spend in paid search. Rather than specifying campaign settings directly, which would let the simulator answer questions such as "What would be the effect of changing my bid?," the function `DefaultSearchMediaModule()` specifies the relationship between budget and campaign settings, so that the simulator will help advertisers answer questions such as "What would be the effect of changing my spend, given that my method of changing my spend is X?" When mimicking different types of advertiser behavior, users should consider:

- Does the advertiser increase spend by uncapping the weekly spend?
- By expanding the keyword list?
- By increasing its bids?
- A combination of the above?

The answer to these questions impacts the relationship between sales and ad dollars spent in paid search

Just as in the traditional media module, the search media module specifies the effect of paid search through marginal transition matrices. Below, the effect of paid search is limited to changes in activity state.

```
search.activity.trans.mat <- matrix(
    c(0.05, 0.95, 0.00,  # starting state: inactive
      0.00, 0.85, 0.15,  # starting state: exploratory
      0.00, 0.00, 1.00),  # starting: purchase
    nrow = length(kActivityStates), byrow = TRUE)
search.favorability.trans.mat <- matrix(
    c(1.0, 0.0, 0.0, 0.0, 0.0,  # unaware
      0.0, 1.0, 0.0, 0.0, 0.0,  # negative
      0.0, 0.0, 1.0, 0.0, 0.0,  # neutral
      0.0, 0.0, 0.0, 1.0, 0.0,  # favorable
      0.0, 0.0, 0.0, 0.0, 1.0),  # loyal
    nrow = length(kFavorabilityStates), byrow = TRUE)
```

Scenarios exploring the complexities of search can be created by adjusting different aspects of the specification. For example, a user of AMSS may add complexity with a non-zero effectiveness for organic search results in the `relative effectiveness` parameter; `relative effectiveness` scales the effects of organic search results, paid impression(s) with no paid click, and paid impression(s) with paid click(s) against the maximal effect of search.

```
params.search <- list(
    audience.membership = list(activity = c(0.01, 0.3, 0.4)),
    budget = (2.4e7 / n.years) * (1:n.years),
    budget.index = budget.index,
    spend.cap.fn = spend.cap.fn,
    bid.fn = bid.fn,
    kwl.fn = kwl.fn,
    query.rate = 1,
    cpc.min = cpc.min,
    cpc.max = cpc.max,
    ctr = list(activity = c(0.005, 0.08, 0.10)),
    relative.effectiveness = c(0, 0.1, 1),
    transition.matrices = list(
        activity = search.activity.trans.mat,
        favorability = search.favorability.trans.mat))
```

Sales

The sales parameters below specify

- The price of the advertiser's product at each time point. A constant price can be written as a single number.
- The linear relationship between price and demand for consumers who have reached the 'purchase' activity state. The demand curve differs by brand state (brand favorability, brand loyalty, and brand availability).
- The strength of competitor effects is also specified here. Demand for the competitor's product differs by brand loyalty state.
  - As specified by `competitor.demand.max`, a maximum of 80% of purchasing consumers who are 'switchers' (first entry) or 'competitor-loyal' (third entry) will choose a competitor's brand given high pricing of the advertiser's brand. Since the second entry is 0, consumers who are loyal to the advertiser's brand will never buy a competitor brand.
  - By default, the parameter `competitor.demand.replacement` is defined as `list(loyalty = c(0.5, 0, 1))`. The first entry refers to consumers with loyalty state `switcher`, and specifies that switchers conflicted between the advertiser and its competitors will choose the advertiser's

brand at half the rate they would have in the absence of competitive pressure. When this parameter increases, the presence of competitive pressure has a stronger negative effect on advertiser sales. Suppose that there are 100 purchasers labeled as `switcher`. Also suppose, first, that at the current price point, in the absence of competition, all these switchers would have purchased the advertiser's product; second, that in the advertiser's absence 80 would have bought a competitor product. Then, the replacement rate specifies that of the 80 (minimum of 100 and 80) people who potentially could purchase either brand, half (40) choose the advertiser's brand. Since there were 20 consumers who weren't interested in any competitor products at the current pricing levels, a total of 60 consumers choose the advertiser's brand and 40 choose a competitor's.

```
sales.params <- list(
    competitor.demand.max = list(loyalty = c(0.8, 0, 0.8)),
    advertiser.demand.slope = list(favorability = rep(0, 5)),
    advertiser.demand.intercept = list(
        favorability = c(0.014, 0, 0.2, 0.3, 0.9)),
    price = 80)
```

Simulate data

Call the package's main function, `SimulateAMSS()`, to generate the data.

```
sim.data <- SimulateAMSS(
    time.n = time.n,
    nat.mig.params = nat.mig.params,
    media.names = c("tv", "search"),
    media.modules = c(
        `DefaultTraditionalMediaModule`,
        `DefaultSearchMediaModule`),
    media.params = list(params.tv, params.search),
    sales.params = sales.params)
```

Simulation output: S3 class `amss.sim`

The output of a call to `SimulateAMSS()` is an object of S3 class `amss.sim`. Objects with this class are lists with three elements:

- `data`: the observed data generated by the simulator, as a `data.table` with one row per time interval.
- `data.full`: the full data specifying the values of all hidden and observed variables by both time interval and population segment. It is formatted as a list of `data.table`'s, one for each time interval, with rows corresponding to the population segments.

  Each `data.table` in the `data.full` element of `sim.data` includes the following variables.

```
##  [1] "market"                "satiation"
##  [3] "activity"              "favorability"
##  [5] "loyalty"               "availability"
##  [7] "time.index"            "pop"
##  [9] "pop.out"               "pop.in"
## [11] "tv.budget.index"       "tv.budget"
## [13] "tv.audience"           "tv.volume"
## [15] "tv.spend"              "tv.absolute.reach"
## [17] "tv.frequency"          "search.budget.index"
## [19] "search.budget"         "search.audience"
```

```
## [21] "search.ctr"                  "search.clicks"
## [23] "search.imps"                 "search.matching.query.volume"
## [25] "search.query.volume"         "search.spend"
## [27] "total.spend"                 "brand.sales"
## [29] "competitor.sales"            "revenue"
## [31] "profit"
```

- `params`: the parameter list used to generate the data. It is important that the object retains this information in order to be able to calculate ground truth for various marketing metrics in the future.

Observed data

Generally, it is good idea to throw away some of the initial weeks of data, to make sure the simulation has reached a stable state. Here, we modify the observed data in `sim.data$data` by applying a burn-in period of 52 weeks.

```
burn.in.length <- 52
final.year.end <- n.years * 52
final.year.start <- final.year.end - 51
observed.data <- sim.data$data[(burn.in.length + 1):final.year.end, ]
```

It is possible to add additional variables to the observed data. For example, although this would not usually be the case in reality, we can provide the modeler with perfect knowledge of the seasonal variation in market size with the following code.

```
observed.data[,
              market.rate :=
                  market.rate.seas[(burn.in.length + 1):final.year.end]]
```

Functions such as `SimulateCorrelated()` are useful for tuning the accuracy of such a variable, so the user can test model performance under various levels of imperfect knowledge of the seasonality.

This leaves the modeler with a 156x18 `data.table` of observed.data. The observed variables include weekly media volume, media spend, and sales data.

```
names(observed.data)
```

```
##  [1] "time.index"                "tv.volume"
##  [3] "tv.spend"                  "search.clicks"
##  [5] "search.imps"               "search.matching.query.volume"
##  [7] "search.query.volume"       "search.spend"
##  [9] "total.spend"               "brand.sales"
## [11] "competitor.sales"          "revenue"
## [13] "profit"                    "tv.budget.index"
## [15] "tv.budget"                 "search.budget.index"
## [17] "search.budget"             "market.rate"
```

A scenario family with varying levels of lag

The above scenario can be modified so that the effect of TV on brand favorability persists beyond the initial time interval of exposure. The function below allows the user to specify a `lag.alpha` between 0 and 1 that controls the amount of lag by modifying the natural transition in brand favorability state over time. By mixing the original transition matrix with the identity matrix, the user can specify some non-zero tendency of consumers to retain the brand favorability state they ended the previous time interval in.

Note that the transition matrix controlling the maximum *initial* impact of TV on brand favorability is scaled by `lag.alpha` in order to keep the *total* impact of TV constant.

```
GetLagArgs <- function (
    time.n, nat.mig.params, params.tv, params.search, sales.params,
    lag.alpha = 0) {
  # add lag by mixing no-lag transition matrix w/ identity matrix
  nat.mig.params$transition.matrices$favorability <-
      (1 - lag.alpha) * nat.mig.params$transition.matrices$favorability +
      lag.alpha * diag(nrow(nat.mig.params$transition.matrices$favorability))

  # adjust initial media impact to keep total impact constant
  params.tv$transition.matrices$favorability <-
      (1 - lag.alpha) * params.tv$transition.matrices$favorability +
      lag.alpha * diag(nrow(params.tv$transition.matrices$favorability))

  return(list(
      time.n = time.n,
      nat.mig.params = nat.mig.params,
      media.names = c("tv", "search"),
      media.modules = c(
          `DefaultTraditionalMediaModule`,
          `DefaultSearchMediaModule`),
      media.params = list(params.tv, params.search),
      sales.params = sales.params))
}
```

We can generate the arguments for `SimulateAMSS()` for a scenario where the strength of the lagged effect is 0.5 by calling `GetArgs()` with `lag.alpha = 0.5`.

```
args <- GetLagArgs(
    time.n = time.n,
    nat.mig.params = nat.mig.params,
    params.tv = params.tv,
    params.search = params.search,
    sales.params = sales.params,
    lag.alpha = 0.5)
```

To evaluate a modeling method for its robustness to increasing amounts of lag, the user may wish to generate multiple argument lists that set `lag.alpha` to gradually increasing values from 0 to 1.

Calculating the ground truth: return on advertising spend

AMSS provides the function `CalculateROAS()` for calculating the true return on advertising spend (ROAS) for any simulation scenario. `CalculateROAS()` compares revenue under two simulation settings (the original, and a counterfactual of what would have happened if the advertiser had set a different budget for the relevant media channel) and reports the expected ratio of the difference in revenue to the difference in advertising spend. The expectation is calculated empirically; multiple replicates are drawn to reduce any possible noise introduced by randomness in the simulator.

By default, `CalculateROAS()` calculates the average ROAS over the entire spend in a media channel.

```
tv.roas <- CalculateROAS(
    sim.data,
    media.names = "tv",
```

```
    t.start = final.year.start,
    t.end = final.year.end,
    min.reps = 10,
    max.time = 30,
    verbose = TRUE)
```

By default, `verbose = FALSE`, and `CalculateROAS()` simply outputs a value for the ROAS. When `verbose = TRUE`, the function outputs the full sample of ROAS values it drew during the estimation process and the precision of the estimate.

The user can also calculate the marginal ROAS (mROAS) by specifying the `budget.proportion` argument to the function. The code below would calculate the mROAS as the ROAS over a 5% decrease in TV spend.

```
tv.mroas <- CalculateROAS(
    sim.data,
    media.names = "tv",
    budget.proportion = 0.95,
    t.start = final.year.start,
    t.end = final.year.end,
    min.reps = 10,
    max.time = 30,
    verbose = TRUE)
```

The `min.reps` argument specifies that `CalculateROAS()` should calculate the ROAS by sampling from the true distribution a minimum of 10 times. The function will continue sampling until the desired precision is achieved, or the time limit of 30 minutes (specifed as the `max.time`) has passed. Note that precision is specified as both a 95% confidence margin of error and a coefficient of variation; achieving either of these will halt the sampling. When the function runs out of time without achieving either the desired margin of error (default of 0.01) or the desired coefficient of variation (default of 0.01), `CalculateROAS()` prints a warning message.

Generally, calculation of the mROAS will take more time to achieve a requested level of precision than calculation of ROAS. This is due to the small difference in spend between the original scenario and the counterfactual, which forms the denominator of the ROAS formula. Precision can be increased by extending the time allotted for the calculation, or by choosing `budget.proportion` further from 1.

References

[1] Zhang, S. and Vaver, J. (2017). The Aggregate Marketing System Simulator. *https://research.google.com/pubs/pub45996.html*.

Disclaimer