# Advertising Effects

*Günter J. Hitsch*

*January 26, 2017*

```r
library(bit64)
library(data.table)
library(RcppRoll)
library(ggplot2)
library(lfe)
library(stargazer)
```

## Overview

Our goal is to estimate own and competitive advertising effects for two brands, Charmin and Coca-Cola. We combine store-level sales data from the Nielsen RMS scanner data set with DMA-level advertising exposure data from the Nielsen Ad Intel advertising data. We are currently in the process of building this advertising data set for academic research purposes at the Kilts Center of Marketing, Booth School of Business.

We will compare estimates based on a within-market strategy that controls for cross-sectional heterogeneity across markets with a border strategy that exploits the discontinuity in advertising at the common border between two neighboring DMA's. The border strategy is based on research by Professor Brad Shapiro. This assignment replicates some results that are part of an ongoing research project with Brad Shapiro that focuses on providing a comprehensive and general overview of advertising effectiveness across a large number (1,000 or more) brands.

## Data

When you download the data, please note that the RMS scanner data and Ad Intel advertising data are both named according to their `product_module_code` and hence have identical file names. Therefore, make sure that the files are stored in separate, appropriately named directories.

### Brand and product module codes

```
 product_module_code brand_code_uc    brand_descr           product_module_descr
                1484        531429    COCA-COLA R        SOFT DRINKS - CARBONATED
                7260        526996        CHARMIN                   TOILET TISSUE
```

Choose one of the two brands before you start the analysis:

```r
selected_module = 7260
selected_brand  = 526996
model_name      = "Toilet-Tissue-Charmin"
```

## Data preparation

To prepare and build the data for the main analysis, load the brand and store meta-data in `Brands.RData` and `Stores-DMA.RData`, the RMS store-level scanner data (named by product module), and the Nielsen Ad Intel DMA-level TV advertising data (also named by product module).

Both the RMS scanner data and the Ad Intel advertising data include information for the top four brands in the category (product module). To make our analysis computationally more manageable we will not distinguish among all individual competing brands, but instead we will aggregate all competitors into one single brand.

### RMS scanner data (`move`)

For consistency, rename the `units` to `quantity and`promo_percentage`to`promotion`. The promotion variable captures promotional activity as a continuous variable with values between 0 and 1.

Create the variable `brand_name` that we will use to distinguish between the own and aggregate competitor variables. `own` corresponds to the focal brand, and `comp` (or any other name that you prefer) corresponds to all the competitor data.

We need to aggregate the data for each store/week observation, separately by the `own` and `comp` data. To aggregate prices and promotions we can take the simple arithmetic *mean* over all competitor brands (a weighted mean may be preferable but is not necessary in this analysis where prices and promotions largely serve as controls, not as the main marketing mix variables of interest). Aggregate quantities can be obtained as the *sum* over brand-level quantities.

Later, when we merge the RMS scanner data with the Ad Intel advertising data, we need a common key between the two data sets. This key will be provided by the DMA code and the date. Hence, we need to merge the `dma_code` found in the `stores` table with the RMS movement data.

For reasons that will be explained further below, the `stores` table contains more than one row for each `store_code_uc` observation. Therefore, if we attempt a merge, data.table will throw an error message (try it!). The reason is that data.table recognizes that there is frequently more than one observation in the `stores` table that matches the observations in `move` given the key `store_code_uc`. data.table can (using the `allow.cartesian` option) match all those observations by increasing the size of the overall matched table, but not by default. Indeed, in many situations when we attempt a many-to-many match we probably made a mistake, and data.table catches this mistake.

Here, we recognize that there is one unique `dma_code` for each store. Hence, to perform the merge, we simply need to extract the DMA and store code variables and retain only *unique* rows using the `unique` function.

```
stores_dma = unique(stores[, .(store_code_uc, dma_code)])
```

Now merge the `dma_code`.

### Ad Intel advertising data (`adv_DT`)

The Ad Intel data contain information on national and local GRP's (gross rating points) and also on the number of occurrences of an ad during a week. The original data are more disaggregated, and include individual occurrences on a specific date and at a specific time and the corresponding number of impressions. Using the same methodology used by Nielsen, we aggregated these data at the DMA/week level. Weeks are indicated by `week_start`, the first day (Monday) of a given week.

The data do not contain observations for all DMA/week combinations during the observation period. In particular, no DMA/week record is included if there was no corresponding advertising activity. For our purposes, however, it is important to record that the number of GRP's (or occurrences) was 0 for such observations. Hence, we need to "fill the holes" in the data set.

As always, data.table makes it easy to achieve this goal. Let's illustrate using a simple example:

```
set.seed(444)
DT = data.table(dma  = rep(LETTERS[1:2], each = 5),
                week = 1:5,
                 x     = round(runif(10, min = 0, max =20)))
DT = DT[-c(2, 5, 9)]
DT
```

```
   dma week  x
1:   A    1  3
2:   A    3  8
3:   A    4  7
4:   B    1 12
5:   B    2 11
6:   B    3  1
7:   B    5  6
```

In `DT`, the observations for weeks 2 and 5 in market A and week 4 in market B are missing.

To fill the holes, we need to key the data.table to specify the dimensions—here the `dma` and `week`. Then we perform a *cross join* using `CJ` (see `?CJ`). In particular, for each of the variables along which `DT` is keyed we specify the full set of values that the final data.table should contain. In this example, we want to include the markets A and B and all week, 1-5.

```
setkey(DT, dma, week)
DT = DT[CJ(c("A", "B"), 1:5)]
DT
```

```
    dma week  x
 1:   A    1  3
 2:   A    2 NA
 3:   A    3  8
 4:   A    4  7
 5:   A    5 NA
 6:   B    1 12
 7:   B    2 11
 8:   B    3  1
 9:   B    4 NA
10:   B    5  6
```

We can replace all missing values (`NA`) with another value, say -111, like this:

```
DT[is.na(DT)] = -111
DT
```

```
    dma week    x
 1:   A    1    3
 2:   A    2 -111
 3:   A    3    8
 4:   A    4    7
 5:   A    5 -111
 6:   B    1   12
 7:   B    2   11
 8:   B    3    1
 9:   B    4 -111
10:   B    5    6
```

Use this technique to expand the advertising data in `adv_DT`, using a cross join along along all `brands`, `dma_codes`, and `weeks`:

```
brands    = unique(adv_DT$brand_code_uc)
dma_codes = unique(adv_DT$dma_code)
weeks     = seq(from = min(adv_DT$week_start), to = max(adv_DT$week_start), by = "week")
```

Now perform the cross join and set missing values to 0.

Now create own and competitor names, and then aggregate the data at the DMA/week level, similar to what we did with the RMS scanner data. In particular, aggregate based on the sum of national and local GRP's. As we will not work with the occurrence data, you can discard them.

Finally, create a total `grp` variable, defined as the sum of national and local GRP's.

### Calculate adstock/goodwill

Advertising is likely to have long-run effects on demand. Hence, we will calculate adstock or goodwill variables for own and competitor advertising. We will use the following, widely-used adstock specification ($a_t$ is advertising in period $t$):

$$g_t = \sum_{l=0}^{L} \delta^l \log(1 + a_{t-l}) = \log(1 + a_t) + \delta \log(1 + a_{t-1}) + \cdots + \delta^L \log(1 + a_{t-L})$$

We add 1 to the advertising levels (GRP's) before taking the log because of the large number of zeros in the GRP data.

Here is a particularly easy and fast approach to calculate adstocks. First, define the adstock parameters—the number of lags and the carry-over factor $\delta$.

```
N_lags = 52
delta  = 0.9
```

Then calculate the geometric weights based on the carry-over factor.

```
geom_weights = cumprod(c(1.0, rep(delta, times = N_lags)))
geom_weights = sort(geom_weights)
tail(geom_weights)
```

```
[1] 0.59049 0.65610 0.72900 0.81000 0.90000 1.00000
```

Now we can calculate the adstock variable using the `roll_sum` function in the `RcppRoll` package.

```
setkey(adv_DT, brand_name, dma_code, week_start)
adv_DT[, adstock := roll_sum(log(1+grp), n = N_lags+1, weights = geom_weights,
                             normalize = FALSE, align = "right",  fill = NA),
        by = .(brand_name, dma_code)]
```

Explanations:

1. Key the table along the cross-sectional units (brand name and DMA), then along the time variable. This step is *crucial*! If the table is not correctly sorted, the time-series order of the advertising data will be incorrect.
2. Use the `roll_sum` function based on `grp`. `n` indicates the total number of elements in the rolling sum, and `weights` indicates the weights for each element in the sum. `normalize = FALSE` tells the function to leave the `weights` untouched, `align = "right"` indicates to use all data above the current row in the sum, and `fill = NA` indicates to fill in missing values for the first rows for which there are not enough elements to take the sum.

Alternatively, you could code your own weighted sum function:

```
weightedSum <- function(x, w) {
   T = length(x)
   L = length(w) - 1
   y = rep_len(NA, T)
   for (i in (L+1):T) y[i] = sum(x[(i-L):i]*w)
   return(y)
}
```

Let's compare the execution speed:

```
time_a = system.time(adv_DT[, stock_a := weightedSum(log(1+grp), geom_weights),
                              by = .(brand_name, dma_code)])
```

```
time_b = system.time(adv_DT[, stock_b := roll_sum(log(1+grp), n = N_lags+1,
                                                  weights = geom_weights,
                                                  normalize = FALSE,
                                                  align = "right",  fill = NA),
                              by = .(brand_name, dma_code)])
```

Although even the `weightedSum` function is fast, the speed difference with respect to the optimized code in `RcppRoll` is fast. Lesson: Spend a few minutes searching the Internet to see if someone has already written a package that solves your coding problems.

```
(time_a/time_b)[3]
```

### Merge scanner and advertising data

We can merge (join) the advertising data with the scanner data based on the `brand_name`, `dma_code`, and week. However, in the RMS data the `week_end` corresponds to the last day of a Nielsen RMS week (a Saturday), while in the advertising data the `week_start` corresponds to the first day of a week (a Monday). For a merge, we need the time periods to conform. An easy way to achieve this is to add a `week_end` variable to the advertising data:

```
adv_DT[, week_end := week_start + 5]
```

This works because Date objects in R are days. Adding the number 5 to any Data variable advances the date by 5 days. Hence, the first Monday of a wweek will be mapped to the following Saturday.

Now we can merge the scanner data and the corresponding DMA-level GRP data (`grp`).

### Reshape the data

Use `dcast` to reshape the data from long to wide format. The store code and week variable are the main row identifiers, and we will also add the `dma_code` to the row variables, because we will check the robustness of our results to clustered standard errors at the DMA level. Quantity, price, promotion, and adstock are the column variables.

```
move = dcast(move, dma_code + store_code_uc + week_end ~ brand_name,
             value.var = c("quantity", "price", "promotion", "adstock"))
```

When you inspect the data you see many missing `adstock` values, because the adstock variable is not defined for the first `N_lags` weeks in the data. To free memory, remove all missing values from `move` (`complete.cases`).

### Time trend

Create a time trend or index for each month/year combination in the data.

## Data inspection

### Time-series of advertising levels

Advertising data are awesome to look at. First, pick a DMA. You can easily get a list of all DMA names and codes from the `stores` table. I picked `"CHICAGO IL"`, which corresponds to `dma_code` 602. Then plot the time-series of weekly GRP's for your chosen market.

### Overall advertising variation

Create a new variable **at the DMA-level**, `normalized_grp`, defined as `100*grp/mean(grp)`. This variable captures the percentage deviation of the GRP observations relative to the DMA-level mean of advertising. Plot a histogram of `normalized_grp`.

Note: To visualize the data you should use the `scale_x_continuous` layer to set the axis `limits`. This data set is one of many examples where some extreme outliers distort the graph. See the ggplot2 Introduction for details.

## Advertising effect estimation

We are now ready to estimate the main models.

Estimate the following specifications:

1. Base specification that uses the log of 1+`quantity` as output and the log of prices (own and competitor) and promotions as inputs. Control for store and month/year fixed effects.
2. Add the `adstock` (own and competitor).
3. Like specification 2., but not controlling for time fixed effects.

Combine the results, for example using the `stargazer` package. Discuss your findings.

## Border strategy

Now we employ the border strategy that we discussed in class to estimate the advertising effects.

### Merge border names

The `stores` table contains two variables that we will use for the border strategy. First, `on_border` indicates if a store is located in a county at a DMA border (`TRUE`) or not (`FALSE`). Second, for all border stores the table contains the `border_name`.

Before merging, we convert the `border_name` variable to a factor representation. This saves memory and (further below) helps the `lfe` package to create fixed effects.

```
stores[, border_name := as.factor(border_name)]
```

Then use only observations in the store table for which `on_border == TRUE`, and merge those observations with the `move` table. This will reduce the size of the `move` table because stores that are not at a border will be dropped.

**Warning**: As we already discussed before, there are multiple observations in `stores` for each store code. This is because some stores are adjacent to more than one border. This is one of the rare stations where we would actually like to allow data.table to perform a many-to-many match, using the added option `allow.cartesian = TRUE`.

```r
move = merge(move, stores[on_border == TRUE, .(store_code_uc, border_name)],
             allow.cartesian = TRUE)
```

The central idea of the border strategy is to estimate the advertising effects based on differences in advertising exposure across two counties on one and the other side of a DMA Border. In particular, we want to allow for a *common* time trend in these two adjacent DMA's that controls for any organically occurring variation in demand that may be correlated with the overall advertising levels.

To allow for time trends (time *fixed effects*) that are specific to each border, we need to use an interaction between the `border_name` variable and the `month_index`. In an R formula, interactions between two variables are `var_1:var_2`, hence the border name/time interaction will be `border_name:month_index`.

Estimate:

4. Advertising model with both store fixed effects and border name/time interactions.
5. Model 4. with standard errors that are clustered at the DMA level.

To estimate clustered standard errors, use `... | 0 | dma_code` at the end of the formula (i.e. placed after the fixed effect variables).

## Optional extension: Estimate the carry-over parameter

Search over a range of values for the carry-over factor $\delta$. For each trial value of $\delta$:

- Re-calculate the adstocks
- Estimate the corresponding demand model and store the output in an object (variable), say `fit`
- The fitted (predicted) values from the regression are contained in `fit`: `fit$fitted.values`
- Calculate the RMSD (root-mean-square deviation) between the observed output used in the regression and the fitted values

Report the carry-over factor $\delta$ that yields the best overall fit, based on RMSD.