Advertising Effects

Group 4 - Drew Ficken / Jace Herrmann / Krishna Kadiyala / Nuraddin Samadzade / Nao Azuma

May 8, 2019

Contents

1 Overview **2**

2 Data **2**
2.1 Brands and product modules 2

3 Data preparation **3**
3.1 RMS scanner data (`move`) 3
3.2 Ad Intel advertising data (`adv_DT`) 4
3.3 Calculate adstock/goodwill 7
3.4 Merge scanner and advertising data 9
3.5 Reshape the data 9
3.6 Time trend 10

4 Data inspection **11**
4.1 Time-series of advertising levels 11
4.2 Overall advertising variation 11

5 Advertising effect estimation **13**
5.1 Main models 13
5.2 Estimation using border strategy 14
5.3 Optional extension: Estimate (calibrate) the carry-over parameter 18

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

1 Overview

Our goal is to estimate own and competitive brand-level advertising effects. 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. The advertising data is one of the newest additions to set of data available for academic research and teaching 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 of brands.

2 Data

2.1 Brands and product modules

Data location:

```
data_folder = "/classes/3710501_spr2019/Data/Assignment-4"
```

The table `brands_DT` in the file `Brands.RData` provides information on the available product categories (product modules) and brands, including the "focal" brands for which we may estimate advertising effects.

product_module_code	product_module_desc	brand_code_uc	brand_descr	focal_brand
1484	SOFT DRINKS - CARBONATED	531429	COCA-COLA R	TRUE
8412	ANTACIDS	621727	PRILOSEC	TRUE
1553	SOFT DRINKS - LOW CALORIE	531433	COCA-COLA ZERO DT	TRUE

Choose Prilosec in the Antacids category for your analysis. Later, you can optionally repeat your analysis for the other brands.

```
selected_module = 8412
selected_brand  = 621727
```

3 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 (movement) data, and the Nielsen Ad Intel DMA-level TV advertising data. The scanner data and advertising data are named according to the product module, such as `move_8412.RData` and `adv_8412.RData`.

```
load(paste0(data_folder, "/", "Brands.RData"))
load(paste0(data_folder, "/", "Stores-DMA.RData"))
load(paste0(data_folder, "/", "move_8412.RData"))
load(paste0(data_folder, "/", "adv_8412.RData"))

### get a feel for brand meta-data ###

#str(brand_DT)

#brand_DT

### get a feel for store meta-data ###

#str(stores)

#head(stores)

### get a feel for RMS store-level scanner (movement) data ###

#str(move)

#head(move)

### get a feel for Nielsen Ad Intel DMA-level TV advertising data ###

#str(adv_DT)

#head(adv_DT)
```

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.

3.1 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.

```
setnames(move, "units", "quantity")
setnames(move, "promo_percentage", "promotion")
names(move)
```

```
[1] "brand_code_uc" "store_code_uc" "week_end"      "quantity"
[5] "price"         "promotion"
```

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

corresponds to the aggregate competitor brand.

```
move[, brand_name := ifelse(brand_code_uc == selected_brand, "own","comp")]

table(move$brand_name) ### 3,495,721 observations of Prilosec | 10,291,669 observations of Competitor b
```

```
    comp      own
10291669  3495721
```

We need to aggregate the data for each store/week observation, separately for 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.

```
move = move[, .(price = mean(price), promotion = mean(promotion), quantity = sum(quantity)),
                  keyby = .(store_code_uc, week_end, brand_name)]

### key will be DMA_code and date when merging with Ad Intel data
```

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 will proactively alert us of a likely 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` with the movement data.

```
#move = merge(move, stores[,.(dma_code, store_code_uc)], by = "store_code_uc")    Doesn't work!

move = merge(move, stores_dma)

#head(move)
```

3.2 Ad Intel advertising data (`adv_DT`)

The table `adv_DT` contains information on brand-level GRP's (gross rating points) for each DMA/week combination. 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. We aggregated these data at the DMA/week level. Weeks are indicated by `week_end`, where the corresponding date is always a Saturday. We use Saturdays so that the `week_end` variable in the advertising data corresponds to the date convention in the RMS scanner data, where `week_end` also corresponds to a Saturday.

The data contain two variables to measure brand-level GRP's, `grp_direct` and `grp_indirect`. `grp_direct` records GRP's for which we can create a direct, unambiguous match between the brand name in the scanner

data and the name of the advertised brand. Sometimes, however, it is not entirely clear if we should associate an ad in the Ad Intel data with the brand in the RMS data. For example, should we count ads for BUD LIGHT BEER LIME when measuring the GRP's that might affect sales of BUD LIGHT BEER? As such matches are somewhat debatable, we record the corresponding GRP's in the variable `grp_indirect`.

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 capture that the number of GRP's 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$week_end), to = max(adv_DT$week_end), by = "week")
```

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

```
setkey(adv_DT, brand_code_uc, dma_code, week_end)

adv_DT = adv_DT[CJ(brands,dma_codes,weeks)]

adv_DT[is.na(adv_DT)] = 0

#head(adv_DT)
```

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 GRP's (separately for `grp_direct` and `grp_indirect`).

```
adv_DT[, brand_name := ifelse(brand_code_uc == selected_brand, "own","comp")]

adv_DT = adv_DT[, .(grp_direct = sum(grp_direct), grp_indirect = sum(grp_indirect)),
                  keyby = .(dma_code, week_end, brand_name)]

#head(adv_DT)
```

At this stage we need to decide if we want to measure GRP's using only `grp_direct` or also including `grp_indirect`. I propose to take the broader measure, and sum the GRP's from the two variables to create a combined `grp` measure. You can later check if your results are robust if you use `grp_direct` only (this robustness analysis is optional).

```
adv_DT[, grp := grp_direct+grp_indirect]

#head(adv_DT)
```

3.3 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 δ.

```
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_end)
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 `log(1+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 data table to calculate 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 large. 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]
```

3.4 Merge scanner and advertising data

Merge (join) the advertising data with the scanner data based on brand name, DMA code, and week.

```
### first remove stock_a and stock_b from adv_DT

#adv_DT = adv_DT[,-c("stock_a","stock_b")]

setkey(adv_DT, brand_name, dma_code, week_end)

setkey(move, brand_name, dma_code, week_end)

move = merge(move, adv_DT)

head(move)
```

```
   brand_name dma_code   week_end store_code_uc     price  promotion
1:       comp      500 2010-01-02         88153 0.5957322 0.00000000
2:       comp      500 2010-01-02         95752 0.5644275 0.08261606
3:       comp      500 2010-01-02        123214 0.6173410 0.00000000
4:       comp      500 2010-01-02        129685 0.5777817 0.20306278
5:       comp      500 2010-01-02        189538 0.5948342 0.00000000
6:       comp      500 2010-01-02        366762 0.4063640 0.23904801
   quantity grp_direct grp_indirect      grp adstock
1:     1432   211.2821            0 211.2821      NA
2:     1946   211.2821            0 211.2821      NA
3:     1592   211.2821            0 211.2821      NA
4:     1146   211.2821            0 211.2821      NA
5:     1698   211.2821            0 211.2821      NA
6:     4388   211.2821            0 211.2821      NA
```

3.5 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.

```
#### don't need grp_direct and grp_indirect as we are just going to use grp
move = move[,-c("grp_direct", "grp_indirect")]

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

move = move[complete.cases(move)]

#head(move)
```

If you inspect the data you will 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`).

3.6 Time trend

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

```
move[, `:=`(year = year(week_end), month = month(week_end))]
move[, month_index := 12*(year - min(year)) + month]

move = move[,-c("year","month")]

#head(move)

#tail(move)
```

4 Data inspection

4.1 Time-series of advertising levels

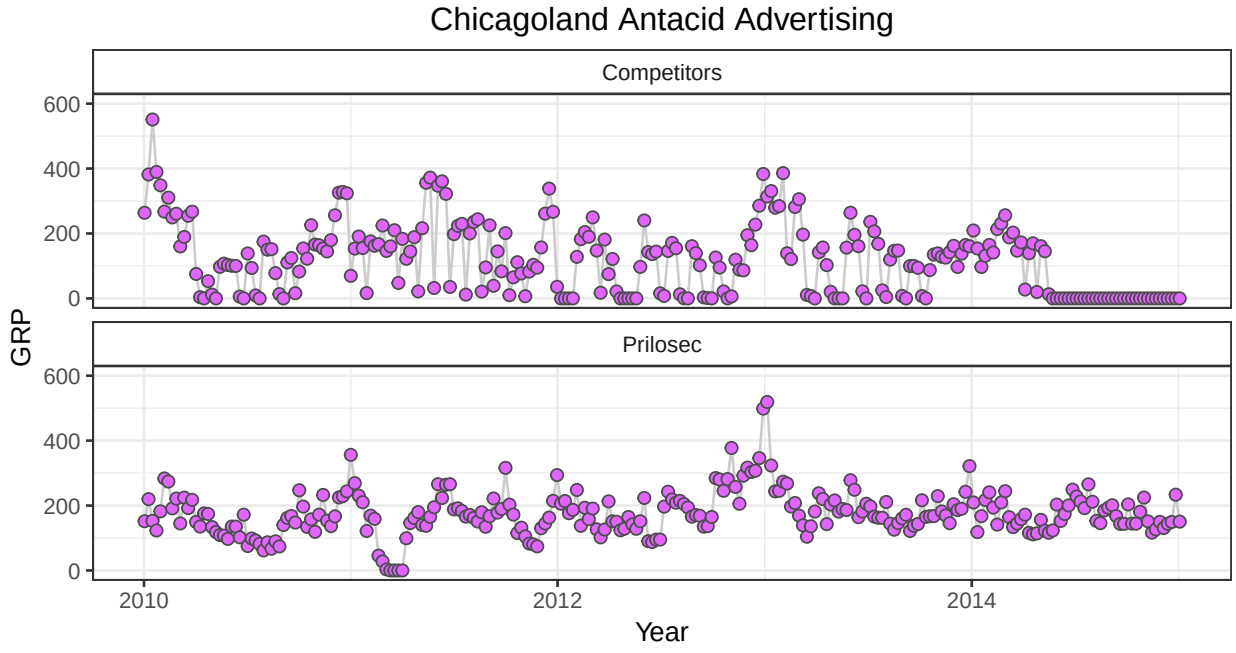
Advertising data are awesome to look at. First, pick a DMA. You can easily get a list of all DMA names and codes from the `stores` table. I picked `"CHICAGO IL"`, which corresponds to `dma_code` 602. Then plot the time-series of weekly GRP's for your chosen market, separately for the own and competitor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

Note: I suggest to create a facet plot to display the time-series of GRP's for the two brands. Use the `facet_grid` or `facet_wrap` layer as explained in the ggplot2 guide (see "More on facetting"). If you want to reorder and possibly rename the facet labels, create a new group variable (e.g. `brand_description`) using the `factor()` function in R. The details are explained at the end of the "More on facetting" section.

```
adv_chi = adv_DT[dma_code == 602]

labels <- c(comp = "Competitors", own = "Prilosec")

ggplot(adv_chi, aes(x=week_end, y = grp)) + geom_line(color = "gray80", size = 0.5) +
geom_point(shape = 21, color = "gray30", fill = "mediumorchid1", size = 2, stroke = 0.5) +
scale_y_continuous(limits = c(0, 600)) +
facet_wrap(~ brand_name, labeller=labeller(brand_name = labels), ncol = 1) +
theme_bw() + theme(strip.background=element_rect(fill="white"), plot.title = element_text(hjust = 0.5))
labs(x = "Year", y = "GRP", title = "Chicagoland Antacid Advertising")
```



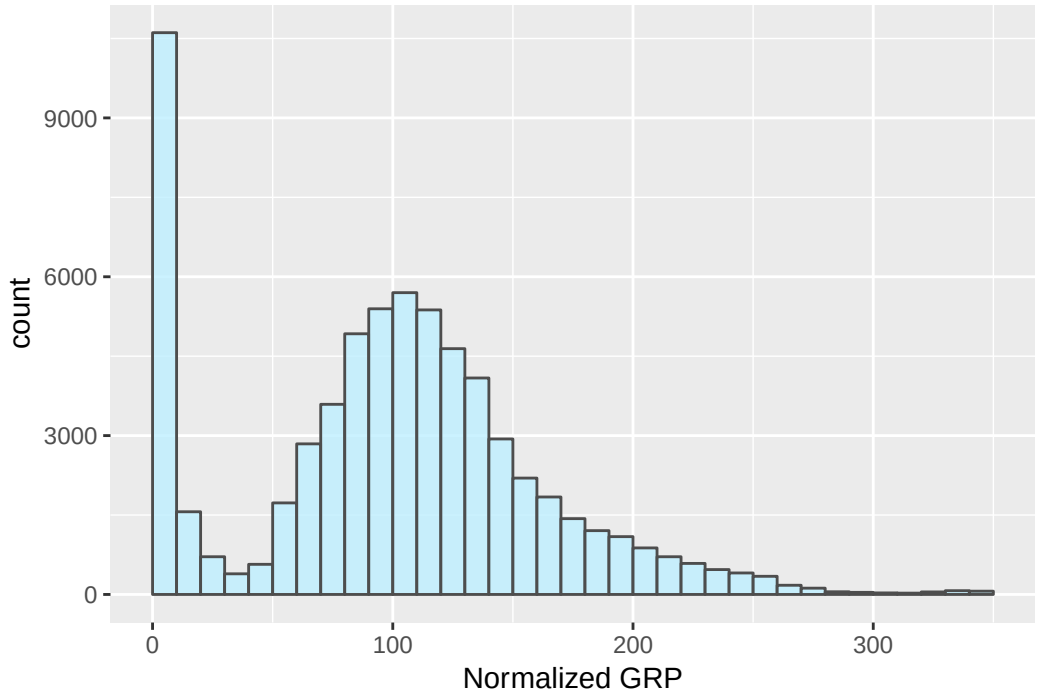
4.2 Overall advertising variation

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`.

```
adv_DT[, normalized_grp := 100*grp/mean(grp), by = dma_code]

ggplot(adv_DT, aes(x = normalized_grp)) +
 geom_histogram(binwidth = 10, center = 5,
color = "gray30", fill = "lightblue1", alpha = 0.8) +
  scale_x_continuous("Normalized GRP",limits=c(0,350))
```

```
Warning: Removed 223 rows containing non-finite values (stat_bin).
```



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.

5 Advertising effect estimation

5.1 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) to specification 1.
3. Like specification 2., but not controlling for time fixed effects.

Combine the results, for example using the `stargazer` package.

```
fit_base = felm(log(1+quantity_own)~log(price_own) + log(price_comp) + promotion_comp +
promotion_own | store_code_uc + month_index, data = move)

fit_adstock = felm(log(1+quantity_own)~log(price_own) + log(price_comp) + promotion_comp +
promotion_own + adstock_comp + adstock_own | store_code_uc + month_index, data = move)

fit_ad_no_time = felm(log(1+quantity_own)~log(price_own) + log(price_comp) + promotion_comp +
promotion_own + adstock_comp + adstock_own | store_code_uc, data = move)


stargazer(fit_base, fit_adstock, fit_ad_no_time,
          type  = "text",
          column.labels  = c("Base", "+ Adstock", "+ Adstock - Time Effects"),
          dep.var.labels.include = FALSE,
          header = FALSE,
          single.row = FALSE)
```

```
======================================================================================
                                        Dependent variable:
                    ------------------------------------------------------------------
                            Base               + Adstock       + Adstock - Time Effects
                            (1)                   (2)                    (3)
--------------------------------------------------------------------------------------
log(price_own)           -2.091***             -2.089***              -1.842***
                          (0.014)               (0.014)                (0.014)

log(price_comp)          -0.099***             -0.098***              -0.394***
                          (0.013)               (0.013)                (0.013)

promotion_comp            0.050***              0.051***               0.018***
                          (0.006)               (0.006)                (0.006)

promotion_own             0.913***              0.913***               0.981***
                          (0.003)               (0.003)                (0.003)

adstock_comp                                    0.002***               0.009***
                                                (0.0003)              (0.00004)

adstock_own                                     0.010***              -0.003***
                                                (0.001)                (0.0001)
```

```
-------------------------------------------------------------------------------------
Observations              2,800,307            2,800,307              2,800,307
R2                          0.632                0.632                  0.627
Adjusted R2                 0.630                0.631                  0.625
Residual Std. Error 0.935 (df = 2786823) 0.935 (df = 2786821)   0.942 (df = 2786868)
=====================================================================================
Note:                                                    *p<0.1; **p<0.05; ***p<0.01
```

5.2 Estimation using 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)]
```

Only merge those 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 situations where we would actually like to allow data.table to perform a many-to-many match, using the added option `allow.cartesian = TRUE`.

```
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 (or rather: *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/time fixed effects.
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).

```
fit_border = felm(log(1+quantity_own)~log(price_own) + log(price_comp) + adstock_comp +
adstock_own + promotion_comp + promotion_own | store_code_uc +
border_name:month_index, data = move)
```

```
fit_bord_clust = felm(log(1+quantity_own)~log(price_own) + log(price_comp) + adstock_comp +
adstock_own +  promotion_comp + promotion_own| store_code_uc +
border_name:month_index | 0 | dma_code, data = move)

stargazer(fit_border, fit_bord_clust,
          type  = "text",
          column.labels  = c("Ad model + border/time FE", "+ Cluster dma's"),
          dep.var.labels.include = FALSE,
          header = FALSE,
          single.row = FALSE)
```

```
================================================================================
                                          Dependent variable:
                                    -----------------------------------
                                    Ad model + border/time FE + Cluster dma's
                                              (1)                   (2)
--------------------------------------------------------------------------------
log(price_own)                          -2.051***             -2.051***
                                         (0.020)               (0.142)

log(price_comp)                         -0.242***             -0.242***
                                         (0.017)               (0.074)

adstock_comp                            0.004***              0.004***
                                        (0.0001)              (0.0003)

adstock_own                             0.007***              0.007***
                                        (0.0002)              (0.0004)

promotion_comp                           0.021**               0.021
                                         (0.008)               (0.015)

promotion_own                           1.000***              1.000***
                                         (0.004)               (0.038)

--------------------------------------------------------------------------------
Observations                            1,714,507             1,714,507
R2                                        0.605                 0.605
Adjusted R2                               0.603                 0.603
Residual Std. Error (df = 1708010)        0.972                 0.972
================================================================================
Note:                                          *p<0.1; **p<0.05; ***p<0.01
```

Summarize and describe all your main estimation results.

```
avg_adstock_own = mean(move$adstock_own)

avg_adstock_comp = mean(move$adstock_comp)

avg_promotion_own = mean(move$promotion_own)

avg_promotion_comp = mean(move$promotion_comp)
```

```r
avg_price_own = mean(move$price_own)

avg_price_comp = mean(move$price_comp)

avg_log_price_own = mean(log(move$price_own))

avg_log_price_comp = mean(log(move$price_comp))

avg_table = cbind.data.frame(avg_adstock_own, avg_adstock_comp, avg_promotion_own, avg_promotion_comp, a

names(avg_table)=c("Adstock - O", "Adstock - C", "Promo - O", "Promo - C", "Price - O", "Price - C", "lo

kable(avg_table, caption = "Averages for Move Dataset ('O' = own, 'C' = competitor)")
```

Table 1: Averages for Move Dataset ('O' = own, 'C' = competitor)

Adstock - O	Adstock - C	Promo - O	Promo - C	Price - O	Price - C	log(Price) - O	log(Price) - C
49.95773	36.66697	0.16397	0.0933814	0.716218	0.5528012	-0.3384973	-0.6026293

Advertising Models: Base, + Adstock, + Adstock - Time Effects

The results of these panel regression summaries are actually quite unexpected.

For the base model, which includes own/competitor prices and promotions, controlling for store and time effects, results in an own-price-elasticity of -2.091. It makes sense that this is a large negative number, implying a downward sloping demand curve. The confusing part of the results is the fact that the competitor-price-elasticity is also negative, meaning that as competing brands raise their prices, consumers purchase less prilosec, and when competing brands lower their prices, consumers purchase more prilosec. This is very counterintuitive and may show that we haven't adequately accounted for confounding factors.

Once we take adstock for own/competition into account in the second model, most of the overlapping covariates' coefficients don't really change at all. Rather, we just arrive at estimates for the adstock variables that are statistically significant, yet very small in magnitude (.01 for own adstock, .002 for competitor adstock). Looking at the average adstock for own and competing brands we are able to get a sense of how these added advertising effects influence the log(1+quantity_own). We see that for an average observation, the AVERAGE adstock of prilosec will add to the log(1+quantity_own) by 50.07*.010, or 0.5, vs. an observation that has no adstock. It's also important to note that the fact that the competitor advertising effect is positive (though very small), which is also somewhat counterintuitive. This could be because advertising can remind consumers of the category (antacids) in general, and then when going to the store they choose prilosec over the advertised brand. We see similar trends with the promotion covariates as well, though the coefficients are much larger (yet the promotion data in the dataset has much smaller magnitude than the adstock does). Price still has a larger impact than advertising or promotion.

The third model removes the time effects that we controlled for in the first two models. This causes our adstock coefficients to shift such that the adstock_own coefficient is now negative (from 0.010 to -0.003) and the adstock_comp to increase (from 0.002 to 0.009). When coefficients are this close to zero, they become very sensitive to changes in the model. The fact that the adstock_own coefficient is negative doesn't really make much sense (more advertising should not lead to fewer quantities purchased, unless the viewer is now annoyed with the advertisements). This is why it is so important to remove confounds when trying to estimate causal advertising effects.

As mentioned in one of the discussions on canvas, it is common to arrive at counterintuitive results when trying to determine causal effects in the real world. We can either try to gather more data as a means of removing false positives, or we can try to account for confounding variables that may be leading to these

counterintuitive results.

For the next two models we included the interactions between the month variable and counties straddling the DMA border. This caused the log(price_own) coefficient to increase slightly from -2.089 to -2.051, and yet caused the log(price_comp) coefficient to decrease a significant amount from -0.098 to -0.242. Again, this result goes against the typical substitution effects we learned about in microeconomics, which means we still may not be accounting for the confounding variables in a sufficient way. The adstock coefficients are now positive again (own = .007, comp = .004), implying that all advertising for the category is good for prilosec. Clustering the standard errors at the DMA level didn't change the model in any noticeable way.

There were a couple of assumptions we made when creating the data that could have led to these surprising results. First, the N_lags = 52 and the delta = 0.9 were set arbitrarily in the equation to calculate adstock. Typically, we would want to estimate the models with different carry-over values (delta), and then select final model based on goodness-of-fit measure. We would then repeat for L using a similar approach. Second, to simplify the model we aggregated all of the competitors into one variable rather than leaving them to standalone. This is most likely the cause for the negative coefficients we saw for the log(competitor price) coefficient. Ultimately, we don't feel that the final model that we arrived at offered conclusive evidence to show causal advertising effects in that we'd be able to predict lift in quantity sold for individual DMA's that were treated with advertising, yet some further manipulation/granularity to the data/model could help us get there.

5.3 Optional extension: Estimate (calibrate) the carry-over parameter

Note: *This part of the assignment is entirely optional*!

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

- 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 MSE (mean squared error) between the observed output used in the regression and the fitted values

Report the carry-over factor δ that yields the best overall fit, based on the MSE.