Open Trade Statistics

By Pachá (aka Mauricio Vargas Sepúlveda)

Introduction

Open Trade Statistics (OTS) was created with the intention to lower the barrier to working with international economic trade data. It includes a public API, a dashboard, and an R package for data retrieval.

The project started when I was affected by the fact that many Latin American Universities have limited or no access to the United Nations Commodity Trade Statistics Database (UN COMTRADE).

There are alternatives to COMTRADE, for example the Base Pour L'Analyse du Commerce International (BACI) constitutes an improvement over COMTRADE as it is constructed using the raw data and a method that reconciles the declarations of the exporter and the importer. The main problem with BACI is that you need UN COMTRADE institutional access to download their datasets.

After contacting UN COMTRADE, and suggesting to them my idea of doing something similar to BACI but available for anyone but keeping commercial purposes out of the scope of the project, I got an authorization to share curated versions of their datasets.

Different projects such as The Atlas of Economic complexity and The Obervatory of Economic complexity use UN COMTRADE data and focus on data visualization to answer questions like:

- What did Germany export in 2016?
- Who imported Electronics in 1980?
- Who exported Refined Copper in 1990?
- Where did Chile export Wine to in 2016?

Unlike existing visualization projects, I wanted to focus on data retrieval and reproducibility, and the starting point was to study the existing trade data APIs to create something more flexible and easier to use than those tools.

Making the code (always) work

I started organizing code I wrote during the last four years at https://github.com/tradestatistics/. There was code there that I haven't touched in more than two years, and I wrote almost no comments indicating what the parts of the code actually do, so it was not understandable for others.

Reproducibility can be explained as: *"Work in a smart way so that your future-self won't ask 'Why does the code return an error?', 'What does this code do?' or 'Why did the result change if I haven't touched the script?'"*. My data cleaning process was not reproducible, and it was tragic to discover! I decided to start using RStudio Server to test the code line by line, in a fresh environment, and then dividing the code into smaller pieces and commenting what the different sections actually do.

Once I had reproducible results I took a snapshot of my packages by using packrat. To ensure reproducibility over time, I decided to build R from source, isolated from the system package manager and therefore avoiding accidental updates that might break the code.

Is it worth mentioning that I'm using DigitalOcean virtual machines to store the datasets and run all the services required to run an API. Under their Open Source Sponsorships the server cost is subsidized.

As a way to contribute back to the community, here you can find a ready to use RStudio image to work with databases and build R packages.

The power of Open Source

With a reproducible data pipeline I had the power to do more, and to do it in a sustainable way. Finally I was able to create the R package that I wanted to submit for rOpenSci software peer review, but that package was the final step.

The base for the project is Ubuntu, the database of choice is PostgreSQL, and R constitutes 95% of the project.

The datasets were cleaned by using data.table, jsonlite, dplyr, tidyr, stringr and janitor. The database was created by using RPostgreSQL. The documentation is R markdown and bookdown. The dashboard was made with Shiny. To adhere to a coding style guide I used styler. In addition to all of that I used doParallel, purrr, Rcpp and Matrix packages in order to use the largest possible share of available resources in the server and in an efficient way, so there is a fraction of code involving sparse matrices and C++.

Even our API was made with R. I used the Plumber package, and I used it to combine RPostgreSQL, dplyr, glue and other R packages. With some input sanitization, and to avoid situations like this XKCD vignette, I was ready to start working on a new R package for rOpenSci and a dashboard that I wanted to visualize the data.

The web service is nginx enhanced with a secured connection by using Let's Encrypt. The landing page is a modified HTML5UP template, and Atom and yui-compressor were the tools to personalize the CSS behind the landing, documentation and dashboard with Fira Sans as the typeface of choice.

Even our email service is a stack of Open Source tools. We use mail-in-a-box with some very simple tweaks such as email forwarding and integration with Thunderbird.

rOpenSci contributions

Thanks to Maëlle Salmon, Amanda Dobbyn, Jorge Cimentada, Emily Riederer, Mark Padgham the overall result can be said to be top quality!

After a long reviewing process (more than six month considering initial submission!), what started as an individual process mutated into something that I consider a collective result. Thanks to the amazing team behind rOpenSci, to their constructive feedback, exhaustive software reviewing and the confidence to propose ideas that that I had never gotten, what you have now is not just a solid R package.

The hours spent as a part of the reviewing process translated into changes to the database and the API. According to the reviewers comments, there are limited opportunities to implement server-side changes and then updating the R code. With the inclusion of different API parameters that I initially didn't consider, the current API/package state provides an efficient solution way better than post-filtering. You'll always extract exactly the data you require and no more than that.

One useful contributed idea was to create aliases for groups of countries. Both in the API and package, you can request an ISO code such as "usa" (United States) or an alias such as "c-am" (all countries in America) that returns condensed queries.

Our API vs the Atlas and the OEC

Our project covers a large documentation with different examples for both API and R package. The package example is reserved for the next section, so you'll probably like to skip this part.

As a simple example, I shall compare three APIs by extracting what did Chile export to Argentina, Bolivia and Perú in 2016 using just common use R packages (jsonlite, dplyr and purrr).

What I am going to do now is to obtain the same information from three different sources, showing how easy or hard is to use each source, and commenting some of the problems that emerge from different APIs.

Packages

```
library(jsonlite)
library(dplyr)
library(purrr)
```

Open Trade Statistics

In case of not knowing the ISO codes for the country of origin or destination, I can check the countries data and inspect it from the browser.

With the code above, it is quite clear that this API is easy to use:

```
# Function to read each combination reporter-partners
read_from_ots <- function(p) {
  fromJSON(sprintf("https://api.tradestatistics.io/yrpc?y=2016&r=chl&p=%s", p))
}

# The ISO codes are here: https://api.tradestatistics.io/countries
partners <- c("arg", "bol", "per")

# Now with purrr I can combine the three resulting datasets
# Chile-Argentina, Chile-Bolivia, and Chile-Perú
ots_data <- map_df(partners, read_from_ots)

# Preview the data
as_tibble(ots_data)
```

```
# A tibble: 2,788 x 15
    year reporter_iso partner_iso product_code product_code_le… export_value_usd import_value_usd
   <int> <chr>        <chr>       <chr>                   <int>            <int>            <int>
 1  2016 chl          arg         0101                        4           593659          1074372
 2  2016 chl          arg         0106                        4               NA            36588
 3  2016 chl          arg         0201                        4               NA        138990325
 4  2016 chl          arg         0202                        4               NA           501203
 5  2016 chl          arg         0204                        4               NA           213358
 6  2016 chl          arg         0206                        4               NA           202296
 7  2016 chl          arg         0207                        4               NA         21218283
 8  2016 chl          arg         0302                        4         35271947               NA
 9  2016 chl          arg         0303                        4           249011          1180820
10  2016 chl          arg         0304                        4         15603048            28658
# … with 2,778 more rows, and 8 more variables: export_value_usd_change_1_year <int>,
#   export_value_usd_change_5_years <int>, export_value_usd_percentage_change_1_year <dbl>,
#   export_value_usd_percentage_change_5_years <dbl>, import_value_usd_change_1_year <int>,
#   import_value_usd_change_5_years <int>, import_value_usd_percentage_change_1_year <dbl>,
#   import_value_usd_percentage_change_5_years <dbl>
```

The resulting data is tidy and, in my opinion, it involved few and simple steps. The codes from the product_code column are official Harmonized System (HS) codes, and those are used both by UN COMTRADE and all over the world.

To answer the original question, with this data as is, is not possible to tell, but I can use the API again to join two tables. I'll obtain the product information and then I'll group the data by groups of products:

```
# Product information
products <- fromJSON("https://api.tradestatistics.io/products")

# Join the two tables and then summarise by product group
# This will condense the original table into something more compact
# and even probably more informative
ots_data %>%
  left_join(products, by = "product_code") %>%
  group_by(group_name) %>%
  summarise(export_value_usd = sum(export_value_usd, na.rm = T)) %>%
  arrange(-export_value_usd)
```

```
# A tibble: 97 x 2
   group_name                                              export_value_usd
   <chr>                                                              <int>
 1 Vehicles; other than railway or tramway rolling stock, and…        444052393
 2 Nuclear reactors, boilers, machinery and mechanical applia…        328008667
 3 Mineral fuels, mineral oils and products of their distilla…        221487719
 4 Electrical machinery and equipment and parts thereof; soun…        179309083
 5 Plastics and articles thereof                                      172385449
 6 Iron or steel articles                                             153072803
 7 Miscellaneous edible preparations                                  149936537
 8 Paper and paperboard; articles of paper pulp, of paper or …        149405846
 9 Fruit and nuts, edible; peel of citrus fruit or melons             139800139
10 Wood and articles of wood; wood charcoal                           113034494
# … with 87 more rows
```

Now we can say that in 2016, Chile exported primarily vehicles to Argentina, Bolivia and Perú.

The Observatory of Economic Complexity

This API is documented here. I'll try to replicate the result from OTS API:

```
# Function to read each combination reporter-partners
read_from_oec <- function(p) {
  fromJSON(sprintf("https://atlas.media.mit.edu/hs07/export/2016/chl/%s/show/", p))
}

# From their documentation I can infer their links use ISO codes for countries,
# so I'll use the same codes from the previous example
destination <- c("arg", "bol", "per")

# One problem here is that the API returns a nested JSON that doesn't work with map_df
# I can obtain the same result with map and bind_rows
oec_data <- map(destination, read_from_oec)
oec_data <- bind_rows(oec_data[[1]]$data, oec_data[[2]]$data, oec_data[[3]]$data)

# Preview the data
as_tibble(oec_data)
```

```
# A tibble: 9,933 x 15
   dest_id export_val export_val_grow… export_val_grow… export_val_grow… export_val_grow… hs07_id hs07_id_len
   <chr>        <dbl>            <dbl>            <dbl>            <dbl>            <dbl> <chr>         <dbl>
 1 saarg      455453.             6.97            0.108          398317.          182558. 010101            6
 2 saarg      100653.             1.79           -0.064           64634.          -39290. 010101…           8
 3 saarg      354799.            15.8             0.217          333682.          221847. 010101…           8
 4 saarg          NA             NA              NA                NA               NA    010106            6
 5 saarg          NA             NA              NA                NA               NA    010106…           8
 6 saarg          NA             NA              NA                NA               NA    010201            6
 7 saarg          NA             NA              NA                NA               NA    010201…           8
 8 saarg          NA             NA              NA                NA               NA    010202            6
 9 saarg          NA             NA              NA                NA               NA    010202…           8
10 saarg          NA             NA              NA                NA               NA    010204            6
# … with 9,923 more rows, and 7 more variables: import_val <dbl>, import_val_growth_pct <dbl>,
#   import_val_growth_pct_5 <dbl>, import_val_growth_val <dbl>, import_val_growth_val_5 <dbl>, origin_id <chr>,
#   year <dbl>
```

At first sight the API returned many more rows than in the previous example. To obtain the exact same result I'll need post-filtering at product code. One curious column in the table above is hs07_id_len, and it reflects length of the HS code. For example, the first row the HS code is 010101 and its length is 6. This can be a huge problem as that column contains values 6 and 8, because the HS does not contain 8 digits codes and those 6 digits codes are not official HS codes.

If you need to join that table with official HS tables, for example, in case of having to append a column with product names, exactly zero of the codes above shall have match. Among all HS codes, "7325" means "Iron or steel; cast articles" and "732510" means "Iron; articles of non-malleable cast iron", and those are official codes used by all customs in the world. In the OEC case, their "157325" code is actually "7325" from the HS, because they append a "15" that stands for "product community #15, metals".

Let's filter with this consideration in mind:

```
# Remember that this is a "false 6", and is a "4" actually
as_tibble(oec_data) %>%
  filter(hs07_id_len == 6)
```

```
# A tibble: 2,558 x 15
   dest_id export_val export_val_grow… export_val_grow… export_val_grow… export_val_grow… hs07_id hs07_id_len
   <chr>        <dbl>            <dbl>            <dbl>            <dbl>            <dbl> <chr>         <dbl>
 1 saarg      558931.            0.223            0.277          101763.          394357. 010101            6
 2 saarg           NA               NA               NA               NA               NA  010106            6
 3 saarg           NA               NA               NA               NA               NA  010201            6
 4 saarg           NA               NA               NA               NA               NA  010202            6
 5 saarg           NA               NA               NA               NA               NA  010204            6
 6 saarg           NA               NA               NA               NA               NA  010206            6
 7 saarg           NA               NA               NA               NA               NA  010207            6
 8 saarg    41842074.             0.14            0.163         5146236.        22203666. 010302            6
 9 saarg      621080.             1.93           -0.135          409185.         -661807. 010303            6
10 saarg    20324918.            0.287            0.231         4534606.        13148256. 010304            6
# … with 2,548 more rows, and 7 more variables: import_val <dbl>, import_val_growth_pct <dbl>,
#   import_val_growth_pct_5 <dbl>, import_val_growth_val <dbl>, import_val_growth_val_5 <dbl>, origin_id <chr>,
#   year <dbl>
```

Finally I can get something closer to what can be obtained with OTS API.

The Atlas of Economic Complexity

I couldn't find documentation for this API but still I'll try to replicate the result from OTS API (I obtained the URL by using Firefox inspector at their website):

```
# Function to read each combination reporter-partners
read_from_atlas <- function(p) {
  fromJSON(sprintf("http://atlas.cid.harvard.edu/api/data/location/42/hs_products_by_partner/%s/?level=4digit", p))
}

# Getting to know these codes required web scraping from http://atlas.cid.harvard.edu/explore
# These codes don't follow UN COMTRADE numeric codes with are an alternative to ISO codes
destination <- c("8", "31", "173")

# The resulting JSON doesn't work with map_df either
# This can still be combined without much hassle
atlas_data <- map(destination, read_from_atlas)
atlas_data <- bind_rows(atlas_data[[1]]$data, atlas_data[[2]]$data, atlas_data[[3]]$data)

# Preview the data
as_tibble(atlas_data)
```

```
## # A tibble: 59,518 x 6
##    export_value import_value location_id partner_id product_id  year
##           <int>        <int>       <int>      <int>      <int> <int>
##  1        23838       413061          42          8        650  1995
##  2       172477       368650          42          8        650  1996
##  3       146238       310383          42          8        650  1997
##  4        69139       141525          42          8        650  1998
##  5        79711        97951          42          8        650  1999
##  6        85042       392098          42          8        650  2000
##  7       463361       252611          42          8        650  2001
##  8       191069       186278          42          8        650  2002
##  9        88566       106782          42          8        650  2003
## 10       234638       113184          42          8        650  2004
## # … with 59,508 more rows
```

Post-filtering is required at year as there are more years than what was requested:

```
as_tibble(atlas_data) %>%
  filter(year == 2016)
```

```
## # A tibble: 2,718 x 6
##    export_value import_value location_id partner_id product_id  year
##           <int>        <int>       <int>      <int>      <int> <int>
##  1       463809      1074354          42          8        650  2016
##  2            0        17189          42          8        655  2016
##  3            0    139638464          42          8        656  2016
##  4            0       507301          42          8        657  2016
##  5            0       212049          42          8        659  2016
##  6            0       124921          42          8        661  2016
##  7            0     20601067          42          8        662  2016
##  8     34454500            0          42          8        667  2016
##  9       211614       724851          42          8        668  2016
## 10     14944704        25975          42          8        669  2016
## # … with 2,708 more rows
```

Finally I can get something closer to what can be obtained with OTS API. Here is a major drawback that is the product id consists in numbers, and this is totally against HS codes which are always used as character provided some codes start with zero.

R package

Even when the package connects to the API, it required a dedicated site with documentation and examples. Please check the documentation here.

Now that I've compared the APIs I'll dig a bit in the R package we have prepared. If I want to obtain the same data as with the examples above, I can do this:

```
# install.packages("tradestatistics")
library(tradestatistics)

ots_create_tidy_data(
  years = 2016,
  reporters = "chl",
  partners = c("arg", "bol", "per")
)
```

```
# A tibble: 2,788 x 20
    year reporter_iso partner_iso reporter_fullna… partner_fullnam… product_code product_code_le… product_fullnam…
   <int> <chr>        <chr>       <chr>            <chr>            <chr>                   <int> <chr>
 1  2016 chl          arg         Chile            Argentina        0101                        4 Horses, asses, …
 2  2016 chl          arg         Chile            Argentina        0106                        4 Animals, n.e.c.…
 3  2016 chl          arg         Chile            Argentina        0201                        4 Meat of bovine …
 4  2016 chl          arg         Chile            Argentina        0202                        4 Meat of bovine …
 5  2016 chl          arg         Chile            Argentina        0204                        4 Meat of sheep o…
 6  2016 chl          arg         Chile            Argentina        0206                        4 Edible offal of…
 7  2016 chl          arg         Chile            Argentina        0207                        4 Meat and edible…
 8  2016 chl          arg         Chile            Argentina        0302                        4 Fish; fresh or …
 9  2016 chl          arg         Chile            Argentina        0303                        4 Fish; frozen (e…
10  2016 chl          arg         Chile            Argentina        0304                        4 Fish fillets an…
# … with 2,778 more rows, and 12 more variables: group_code <chr>, group_name <chr>, export_value_usd <int>,
#   import_value_usd <int>, export_value_usd_change_1_year <int>, export_value_usd_change_5_years <int>,
#   export_value_usd_percentage_change_1_year <dbl>, export_value_usd_percentage_change_5_years <dbl>,
#   import_value_usd_change_1_year <int>, import_value_usd_change_5_years <int>,
#   import_value_usd_percentage_change_1_year <dbl>, import_value_usd_percentage_change_5_years <dbl>
```

Here the added value is that the package does all the work of combining the data, and it does some joins for you to add country names, product names and full product category/community description.

There are several cases where the functions within this package remain simple. For example, if I require different years, and instead of product level data I just need aggregated bilateral flows from all countries in America to all countries in Asia, this is how to obtain that data:

```
ots_create_tidy_data(
  years = 2010:2017,
  reporters = "c-am",
  partners = "c-as",
  table = "yr"
)
```

```
# A tibble: 386 x 21
    year reporter_iso reporter_fullna… export_value_usd import_value_usd top_export_prod… top_export_trad…
   <int> <chr>        <chr>                       <dbl>            <dbl> <chr>                       <dbl>
 1  2010 aia          Anguilla                 12165731         64287919 8514                      3274981
 2  2010 ant          Neth. Antilles         1631080123       2966955978 2710                    1229297847
 3  2010 arg          Argentina             76056875101      64416501373 2304                    9352050413
 4  2010 atg          Antigua and Bar…       2464746725       2573456652 8703                     263196190
 5  2010 bhs          Bahamas                3139761427      12310398156 2710                    1473528434
 6  2010 blz          Belize                  459835990       1053385171 2709                     130371691
 7  2010 bmu          Bermuda                 718819987       5021642429 8903                     531493968
 8  2010 bol          Bolivia                7754773449       7197859978 2711                    2797774138
 9  2010 bra          Brazil               241714684212     225037224410 2601                   37576257058
10  2010 brb          Barbados               1097175359       2655154217 8481                     110615870
# … with 376 more rows, and 14 more variables: top_import_product_code <chr>,
#   top_import_trade_value_usd <dbl>, export_value_usd_change_1_year <dbl>,
#   export_value_usd_change_5_years <dbl>, export_value_usd_percentage_change_1_year <dbl>,
#   export_value_usd_percentage_change_5_years <dbl>, import_value_usd_change_1_year <dbl>,
#   import_value_usd_change_5_years <dbl>, import_value_usd_percentage_change_1_year <dbl>,
#   import_value_usd_percentage_change_5_years <dbl>, eci_4_digits_product_code <dbl>,
#   eci_rank_4_digits_commodity_code <int>, eci_rank_4_digits_commodity_code_delta_1_year <int>,
#   eci_rank_4_digits_commodity_code_delta_5_years <int>
```

How to contribute?

If you are interested in contributing to this project send us a tweet or an email. We'd also like to read ideas not listed here.

Here's a list of ideas for future work:

- *Crops data*: I got suggestions to include crops data from The Food and Agriculture Organization of the United Nations (FAO) to be easily be able to compare volumes and exported values of crop/commodity groups. The problem is that UN COMTRADE and FAO group their data in different categories.
- *R package for Economic Complexity*: We have a set of both R and Rcpp functions such as Revealed comparative advantage, Economic Complexity Index, and other related functions that might lead to a new package.
- *D3plus htmlwidget*: D3plus is an open source D3 based library which is released under MIT license. It can be a good alternative to Highcharts based on the license type. I've been working on my spare time in a D3plus htmlwidget that integrates with Shiny.
- *Network layouts*: We are in the middle of creating the Product Space for the HS rev. 2007. The idea is to provide a visualization that accounts for products not included in the networks used both by the Atlas and the Observatory, which use HS rev. 1992 and therefore do not reflect the huge changes, especially in electronics, in the last two decades.