PIP Manual

DECIS Poverty GP

2021-06-22

Contents

Prerequisites

You need to make sure the `bookdown` package is installed in your computer

```
install.packages("bookdown")

# or the development version
 devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

Part I

Introduction

Chapter 1

Introduction

PIP workflow is ...

Chapter 2

Internal Workflow

This chapter explains the internal technical workflow of the PIP project. Many of the different steps of the workflow are explained in more detail in other chapters of this book. This chapter, however, presents an overview of the workflow and its components.

2.1 The Github group

The technical code of the PIP project is organized in Git repositories in the Github group /PIP-Technical-Team. You need to be granted collaborator status in order to contribute to any of the repositories in the group. Also, many of the repositories do not play a direct role in the PIP technical workflow. Some of them are intended for documenting parts of the workflow or for testing purposes. For example, the repository of this book–/PIPmanual–is not part of the workflow of PIP, since it is not necessary for any estimation. Yet, you need get familiar with all the repositories in case you need to make a contribution in any of them. In this chapter, however, we will focus on understanding the repositories that affect directly the PIP workflow.

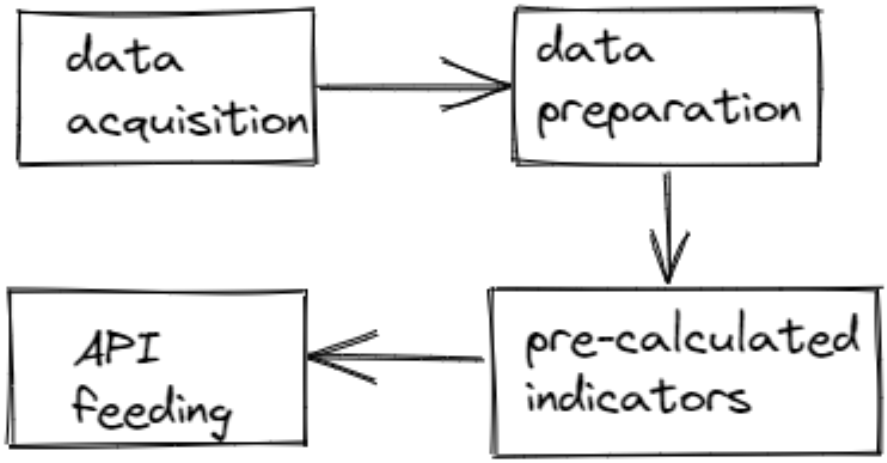
First, we will see the overview of the workflow. It is an overview because each bucket of its buckets is a workflow in itself. Yet, it is important to have the overview clear in order to understand how and where all the pieces fall together. Then, we will unpack each of the workflow buckets to understand them in more detail.

2.2 Overview

The workflow overview is mainly composed of four steps.

1. Data acquisition

2. Data preparation
3. pre-computed indicators
4. API feeding



Each of the steps (or buckets) is prerequisite of the next one, so if something changes in one of the them, it is necessary to execute the subsequent steps.

2.3 Data acquisition

Before understanding how the input data of PIP is acquired, we need to understand PIP data itself. The PIP is fed by two kinds of data: welfare data and auxiliary data.

Welfare data refers to the data files that contain at least one welfare vector and one population expansion factor (i.e., weights) vector. These two variables are the minimum data necessary to estimate poverty and inequality measures.[1] These files come in four varieties; microdata, group data, bin data, and synthetic data. The details of welfare can be found in Section 3. Regardless of the variety of the data, all welfare data in PIP are gathered from the Global Monitoring Database, GMD. For a comprehensive explanation of how the household data are selected and obtained, you may check chapter *Acquiring household survey data*of the methodological PIP manual.

Microdata is uploaded into the PRIMUS system by the regional teams of the Poverty Global Practice. To do so, each regional team has to follow the GMD guidelines, which are verified by the Stata command `{primus}`. Rarely. the `{primus}` command does NOT capture some potential errors in the data. More details in Section 5

[1] Population data is also necessary when working with group data.

As of now (June 22, 2021), *Group data* is divided two: historical group data and new group data. Historical group data is organized and provided by the PovcalNet team, who sends it to the Poverty GP to be included in the datalibweb system. New group data is collected by the poverty economist of the corresponding country, who shares it with her regional focal team, who shares it with PovcalNet team. This new group data is organized and tested by the PovcalNet team and then send back to the poverty GP to be included in the datalibweb system.

Bin data refers to welfare data from countries for which there is no poverty economist. Most of these countries are developed countries such as Canada or Japan. The main characteristic of this data is that it is only available through the LISSY system of the LIS data center, which does not allow access to the entire microdata. Thus, we need to contract the microdata into 400 bins or quantiles. The code that gathers LIS data is available in the Github repository PovcalNet-Team/LIS_data.

Finally, *synthetic data*, refers to simulated microdata from an statistical procedure. As of now (June 22, 2021), the these data are estimated using multiple imputation techniques, so that any calculation must take into account the imputation-id variable. The data is calculated by the poverty economist of the country and the organizes by the global team in the Poverty GP.

Auxiliary data refers to all data necessary to temporally deflate and line up welfare data, with the objective of getting poverty estimates comparable over time, across countries, and, more importantly, being able to estimate regional and global estimates. Some of these data are national population, GDP, consumer price index, purchasing parity power, etc. Auxiliary data also include metadata such as time comparability or type of welfare aggregate. Since each measure of auxiliary data is acquired differently, all the details are explain in Section 4.

2.4 Data preparation

This step assumes that all welfare data is properly organized in the datalibweb system and vetted in PRIMUS. In contrast to the previous global-poverty calculator system, PovcalNet, the PIP system only gathers welfare from the datalibweb server.

The welfare data preparation is done using the repository /pipdp. As of now (June 22, 2021), this part of the process has been coded in Stata, given that the there is no an R version of `{dataliweb}`. As for the auxiliary data preparation, it is done with package `{pipaux}`, available in the repository /pipaux. Right now the automation of updating the auxiliary has not been implemented. Thus, it has to be done manually by typing `pipaux::pip_update_all_aux()` to update all measures, or use function `pipaux::update_aux()` to update a particular measure.

2.5 Pre-computed indicators

All measures in PIP that do not depend on the value of the poverty line are pre-computed in order to make the API more efficient and responsive. Some other indicators that not depend on the poverty lines but do depend on other parameters, like the societal poverty, are not included as part of the pre-computed indicators.

This step is executed in the repository /pip_ingestion_pipeline, which is a pipeline powered by the `{targets}` package. The process to estimate the pre-computed indicators is explained in detail in Chapter 7. The pipeline makes use of two R packages, `{wbpip}` and `{pipdm}`. The former is publicly available and contains all the technical and methodological procedures to estimate poverty and inequality measures at the country, regional, and global level. The latter, makes use of `{wbpip}` to execute the calculations and put the resulting data in order, ready to be ingested by the PIP API.

2.6 API feeding

We need to finish this part.

Part II

PIP data

Chapter 3

Welfare data

blah blah blah

Chapter 4

Auxiliary data

Chapter 5

PRIMUS

According to the description in the Stata repository worldbank/primus,

> The **PRIMUS** system is designed to facilitate this process of generating internal estimates of the World Bank's poverty indicators and reduce the time needed for resolving discrepancies. It is a workflow management platform for the submission, review and approval of poverty estimates and a tracking facility to capture the inputs and results of the estimation process for future reference and audits.

As such, **PRIMUS** is the platform used by the PovcalNet team to approve the adoption of new survey data into the PovcalNet system.

5.1 Interacting with PRIMUS

The interaction with **PRIMUS** is done through different systems, so it is best to begin by clarifying terms.

Website platform

PRIMUS can be accessed by typing primus/ in your browser. As long as you're connected to the intranet it should work fine. However, if you have any issues connecting to the platform, please send an email to Minh Cong Nguyen, requesting access.

Each database uploaded into PRIMUS gets a unique transaction ID. This ID is important because it is not unique to a dataset but unique to the transaction (or vintage of the data). That is, if one particular dataset is uploaded more than once, it will get two different transaction IDs. When talking to the Poverty GP, you better refer to the transaction ID rather than the survey (or at least both) because, though you may be talking about the same country/year, you

are actually talking about two different transactions. See for instance Brazil 2013.

Stata command

The Poverty GP maintains the Stata repository worldbank/primus from which you can download the command `primus`. Right now, this is the official place from which you can access this command. From now on, each time we refer to the command, we use `primus`, whereas when we refer to the website, we use PRIMUS.

Please, make sure you have it properly installed in your computer, by following the instruction section **??**. Basically, you need to install first the github Stata command by E. F. Haghish

```
net install github, from("https://haghish.github.io/github/")
```

Now, you can install `primus` by just typing the following in Stata

```
github install worldbank/primus
```

In case this does not work, follow instructions in section **??** for alternative methods.

Corrections to `primus` Stata command

The `primus` command is maintained by the Poverty GP, so we have no control over modifications or improvements. The best you can do in case you need to fix or modify something in this command is to fork the repository, clone the forked repo into your computer, check out a new branch, make any modification, and generate a pull request to the master branch of the original repository. Once you have done that, make sure to send an email with your suggestions for improvement to Ani Rudra Silwal, copying to the D4G Central Team (Nobuo Yoshida and Minh Cong Nguyen).

5.2 Understanding PRIMUS

Each time a database is uploaded into PRIMUS, it is assigned a transaction ID. During the uploading process (or right after it has finished), the three parties– DECDG, DECRG, or the Poverty GP–evaluate the quality of the new or corrected data and approve them or reject them in the system. Depending on the decision of all the parties, each transaction will take one of three possible status, *pending*, *approved,* or *rejected*.

As of today (2020-11-20), there is no one who represents DECRG. So, the approving process might be different and it will need to be changed in the PRIMUS system. Please check.

The transaction ID is *pending* when at least one of the three parties (DECDG, DECRG, or the Poverty GP) has not approved it in the system. You can click on the check box *PENDING* in the PRIMUS website to see which surveys have such a status, or you can use the `primus` command list this,

```
qui primus query, overallstatus(PENDING)
list transaction_id country year date_modified in 1/`=min(10, _N)'
     +-----------------------------------------------+
     |              transaction_id   country   year |
     |-----------------------------------------------|
  1. | TRN-000327173-EAP-IDN-QR48Q       IDN   2017 |
  2. | TRN-000327173-ECA-DEU-YJYVZ       DEU   1995 |
  3. | TRN-000327173-ECA-DEU-2P4DR       DEU   2002 |
  4. | TRN-000327173-ECA-DEU-LJN8R       DEU   2003 |
  5. | TRN-000327173-ECA-DEU-ZSN9J       DEU   2005 |
     |-----------------------------------------------|
  6. | TRN-000327173-ECA-DEU-UBS7M       DEU   2008 |
  7. | TRN-000327173-ECA-DEU-41TOU       DEU   2009 |
  8. | TRN-000327173-EAP-AUS-KKZ2E       AUS   2004 |
     +-----------------------------------------------+
```

Notice that the overall status of a transaction is independent from survey ID. Thus, it is possible to find several transactions for the same country and year. Indonesia 2017, for instance, has three transactions, two of them rejected and one of them pending.

```
qui primus query, country(IDN) year(2017)
list transaction_id overall_status date_modified in 1/`=min(10, _N)'
     +-------------------------------------------------+
     |              transaction_id        date_modified |
     |-------------------------------------------------|
  1. | TRN-000104674-EAP-IDN-8R9IF   23may2018 15:28:47 |
  2. | TRN-000327173-EAP-IDN-TYA1A   23may2018 23:57:27 |
  3. | TRN-000327173-EAP-IDN-QR48Q   24may2018 00:27:33 |
     +-------------------------------------------------+
```

A transaction is *rejected* when at least one of the three parties rejected the database. Finally, a transaction is *approved* only when all three parties have approved it into the system.

We recommend you understand the basic functionality of the `primus` command by reading the help file (type `help primus` in Stata).

5.3 Checking PRIMUS estimates

The real first step to check the quality of the recently uploaded data into PRIMUS is to download the basic estimates of each data and compare them

with our own. There is no need to calculate and compare all the estimates available in PRIMUS but the mean in PPP, the poverty headcount, and the Gini index.

The `primus` command allows us to download the estimates of each transaction, but it has to be done one by one. Fortunately, the `pcn` command downloads all the estimates of pending transactions for us and properly stores them in the folder `p:\01.PovcalNet\03.QA\02.PRIMUS\pending\`. You only need to type,

```
pcn primus pending, down(estimates)
```

In addition, `pcn` checks the date for which you're downloading the estimates and keeps only those transactions that have been uploaded for the next spring or annual-meetings release. For instance, assume that today, 2020-11-20, you want to see the estimates of pending transactions in PRIMUS. Since annual meetings take place around September, `pcn` assumes you are interested in the estimates for the Spring-meetings release, around March next year. Thus, it will filter the results from `primus`, keeping only those transactions that were uploaded from November 2020. Now it is likely that the PRIMUS system has not been opened for uploading new data in November, as it usually opens around December and July. Thus, it is likely that you will find and error saying `There is no pending data in PRIMUS for the combination of country/years selected`.

You can load the recently-downloaded estimates by typing,

```
pcn primus pending, load(estimates)
```

Now, you have to check whether the new estimates make sense. Once way to that is to follow this do-file, `p:\01.PovcalNet\03.QA\02.PRIMUS\pending\2020_SM\estimates\checks\com`

You do NOT need to check the estimates with the working data (wrk) as it is suggested in the do-file above. The PovcalNet System is now fully integrated with the `datalibweb` system, so the CPI, PPP, and microdata will be always the same. The best you can do at this stage is to make sure the estimates in PRIMUS make sense at the country level.

5.4 Confirming and approving data in PRIMUS

Once you have checked that the estimates of pending transactions make sense, you need to approve them. As explained in section 5.2, the approval on PRIMUS requires the consent of three parties. The PovcalNet team had the responsibility to approve on behalf or two of them, DECDG and DECRG. This process can easily done with the code below, which can be found in this file, `p:\01.PovcalNet\03.QA\02.PRIMUS\pending\2020_SM\approve\primus_approve.do`.

```
/*================================================
0: Program set up
```

```
==================================================*/
version 14
drop _all

*---------- Modify this
local excl = "BRA SOM SSD" // countries to exclude
local excl = "" // countries to exclude

/*==================================================
Load data
==================================================*/

primus query, overalls(pending)
//------------Cut off date
local filtdate = "2019-12-01" // filter date (december last year)
local filtdate = "2020-02-18" // filter date (surveys uploaded by Minh)
keep if  date_modified >= clock("`filtdate'", "YMD")

//------------Select username
if (lower("`c(username)'") == "wb424681") {
  local dep = "povcalnet"
}
else if (lower("`c(username)'") == "wb384996") {
  local dep = "decdg"
}
else {
  noi disp in red "you don't have rights to run this code"
  break
}

tab `dep'
keep if `dep' == "PENDING"

if ("`excl'" != "") {
  local excl: subinstr local excl " " "|", all
  drop if regexm("`country'", "`excl'")
}

/*==================================================
Approve (Do NOT modify)
==================================================*/

local n = _N
preserve
qui foreach i of numlist 1/`n' {
```

```
  restore, preserve
  local country = country[`i']
  local year    = year[`i']
  local id      = transaction_id[`i']

  noi disp in y "primus action, tranxid(`id') decision(approved)"
  cap noi primus action, tranxid(`id') decision(approved)
  if (_rc) noi disp "problem with `id'"
}
```

Basically, this is what you need to do with this file.

1. Modify `local excl` in case you do **not** want to approve one or several countries.
2. Modify `local filtdate` in which you select the date from which you want to approve transactions.
3. Make sure at least two people approve. One on behalf of "povcalnet" (which is the alias used for DECRG) and another on behalf of "decdg."
4. PRIMUS has a double-confirmation process, so you need to "confirm" and then "approve" the transaction. For that, you only need to change the option `decision()` from `approved` to `confirmed`.

For some unknown reason, the PRIMUS system did not accept the approval of some transactions. If this happens again, you need to talk to Minh Cong Nguyen, so he can do the approval manually.

Part III

Technical Considerations

Chapter 6

Load microdata and Auxiliary data

Make sure you have all the packages installed and loaded into memory. Given that they are hosted in Github, the code below makes sure that any package in the PIP workflow can be installed correctly.

```
## First specify the packages of interest
packages = c("pipaux", "pipload")

## Now load or install&load all
package.check <- lapply(
  packages,
  FUN = function(x) {
    if (!require(x, character.only = TRUE)) {
      pck_name <- paste0("PIP-Technical-Team/", x)
      devtools::install_github(pck_name)
      library(x, character.only = TRUE)
    }
  }
)
#> Loading required package: pipaux
#> Loading required package: pipload
```

6.1 Auxiilary data

Even though `pipaux` has more than 25 functions, most of its features can be executed by using only the `pipaux::load_aux` and `pipaux::update_aux` functions.

6.1.1 udpate data

the main function of the `pipaux` package is `udpate_aux`. The first argument of this function is `measure` and it refers to the measure data to be loaded. The measures available are **cpi, gdm, gdp, pce, pfw, pop, and ppp**.

```
pipaux::update_aux(measure = "cpi")
#> no labels available for measure `country_list`
#> v CPI data is up to date
```

6.1.2 Load data

Loading auxiliary data is the job of the package `pipload` through the function `pipload::pip_load_aux()`, though `pipaux` also provides `pipaux::load_aux()` for the same purpose. Notice that, though both function do exactly the same, the loading function from `pipload` has the prefix `pip_` to distinguish it from the one in `pipaux`. However, we are going to limit the work of `pipaux` to update auxiliary data and the work of `pipload` to load data. Thus, all the examples below use `pipload` for loading either microdata or auxiliary data.

```
df <- pipload::pip_load_aux(measure = "cpi")
#> v Most recent version of data loaded
#> '//w1wbgencifs01/pip/PIP-Data_QA/_aux/cpi/cpi.fst'
head(df)
#>    country_code cpi_year survey_year        cpi ccf survey_acronym
#> 1:          AGO     2000     2000.21 0.03385145   1            HBS
#> 2:          AGO     2008     2008.50 0.72328920   1      IBEP-MICS
#> 3:          AGO     2018     2018.17 2.93543023   1          IDREA
#> 4:          ALB     1996     1996.00 0.44446184   1            EWS
#> 5:          ALB     2002     2002.00 0.78033877   1           LSMS
#> 6:          ALB     2005     2005.00 0.83852839   1           LSMS
#>    change_cpi2011    cpi2011 cpi_domain cpi_domain_value cpi2011_unadj
#> 1:              0 0.03385145          1                1    0.03385145
#> 2:              1 0.72328920          1                1    0.72328920
#> 3:              1 2.93543023          1                1    2.93543023
#> 4:              1 0.44446184          1                1    0.44446184
#> 5:              1 0.78033877          1                1    0.78033877
#> 6:              1 0.83852839          1                1    0.83852839
#>    cpi2011_AM20 cpi2011_unadj_AM20 cpi2005_AM20 cpi_final_2019 cpi_data_level
#> 1:  0.033848061        0.033848061  0.071889997             NA       national
#> 2:  0.723337197        0.723337197  1.528669953             NA       national
#> 3:  3.060594983        3.060594983           NA             NA       national
#> 4:  0.444432734        0.444432734  0.530049980             NA       national
#> 5:  0.780287716        0.780287716  0.950504005             NA       national
#> 6:  0.838473458        0.838473458  1.000000000             NA       national
#>              cpi_id
#> 1: CPI_v05_M_v01_A
```

```
#> 2: CPI_v05_M_v01_A
#> 3: CPI_v05_M_v01_A
#> 4: CPI_v05_M_v01_A
#> 5: CPI_v05_M_v01_A
#> 6: CPI_v05_M_v01_A
```

6.2 Microdata

Loading PIP microdata is the most practical action in the `pipload` package. However, it is important to understand the logic of microdata.

PIP microdata has several characteristics,

- There could be more than once survey for each Country/Year. This happens when there are more than one welfare variable available such as income and consumption.
- Some countries, like Mexico, have the two different welfare types in the same survey for the same country/year. This add a layer of complexity when the objective is to known which is default one.
- There are multiple version of the same harmonized survey. These version are organized in a two-type vintage control. It is possible to have a new version of the data because the Raw data–the one provided by the official NSO–has been updated, or because there has been un update in the harmonization process.
- Each survey could be use for more than one analytic tool in PIP (e.g., Poverty Calculator, Table Maker, or SOL). Thus, the data to be loaded depends on the tool in which it is going to be used.

Thus, in order to make the process of finding and loading data efficiently, `pipload` is a three-step process.

6.2.1 Inventory file

The inventory file resides in `y:/PIP-Data/_inventory/inventory.fst`. This file is a data frame with all the microdata available in the PIP structure. It has two main variables, `orig` and `filename`. The former refers to the full directory path of the database, whereas the latter is only the file name. the other variables in this data frame are derived from these two.

The inventory file is used to speed up the file searching process in `pipload`. In previous packages, each time the user wanted to find a particular data base, it was necessary to look into the folder structure and extract the name of all the file that meet a particular criteria. This is time-consuming and inefficient. The advantage of this method though, is that, by construction, it finds all the the data available. By contrast, the inventory file method is much faster than the "searching" method, as it only requires to load a light file with all the data

available, filter the data, and return the required information. The drawback, however, is that it needs to be kept up to date as data changes constantly.

To update the inventory file, you need to use the function `pip_update_inventory`. If you don't provide any argument, it will update the whole inventory, which may take around 10 to 15 min–the function will warn you about it. By provide the country/ies you want to update, the process is way faster.

```
# update one country
pip_update_inventory("MEX")
#> i file 'inventory.fst' is up to date.
#> No update performed

# Load inventory file
df <- pip_load_inventory()
head(df[, "filename"])
#>                                          filename
#> 1:       AGO_2000_HBS_V01_M_V01_A_PIP_PC-GPWG.dta
#> 2: AGO_2008_IBEP-MICS_V02_M_V02_A_PIP_PC-GPWG.dta
#> 3:  AGO_2008_IBEP-MICS_V02_M_V02_A_PIP_TB-ALL.dta
#> 4:     AGO_2018_IDREA_V01_M_V01_A_PIP_PC-GPWG.dta
#> 5:      AGO_2018_IDREA_V01_M_V01_A_PIP_TB-ALL.dta
#> 6:       ALB_1996_EWS_V01_M_V01_A_PIP_PC-HIST.dta
```

6.2.2 Finding data

Every dataset in the PIP microdata repository is identified by seven variables! Country code, survey year, survey acronym, master version, alternative version, tool, and source. So giving the user the responsibility to know all the different combinations of each file is a heavy burden. Thus, the data finder, `pip_find_data()`, will provide the names of all the files available that meet the criteria in the arguments provided by the user. For instance, if the use wants to know the all the file available for Paraguay, we could type,

```
pip_find_data(country = "PRY")[["filename"]]
#>  [1] "PRY_1990_EH_V01_M_V02_A_PIP_PC-GPWG.dta"
#>  [2] "PRY_1995_EH_V01_M_V02_A_PIP_PC-GPWG.dta"
#>  [3] "PRY_1997_EIH_V01_M_V03_A_PIP_PC-GPWG.dta"
#>  [4] "PRY_1999_EPH_V01_M_V03_A_PIP_PC-GPWG.dta"
#>  [5] "PRY_2001_EIH_V01_M_V05_A_PIP_PC-GPWG.dta"
#>  [6] "PRY_2001_EIH_V01_M_V05_A_PIP_TB-ALL.dta"
#>  [7] "PRY_2002_EPH_V01_M_V05_A_PIP_PC-GPWG.dta"
#>  [8] "PRY_2002_EPH_V01_M_V05_A_PIP_TB-ALL.dta"
#>  [9] "PRY_2003_EPH_V01_M_V05_A_PIP_PC-GPWG.dta"
#> [10] "PRY_2003_EPH_V01_M_V05_A_PIP_TB-ALL.dta"
#> [11] "PRY_2004_EPH_V01_M_V05_A_PIP_PC-GPWG.dta"
#> [12] "PRY_2004_EPH_V01_M_V05_A_PIP_TB-ALL.dta"
```

```
#> [13] "PRY_2005_EPH_V01_M_V05_A_PIP_PC-GPWG.dta"
#> [14] "PRY_2005_EPH_V01_M_V05_A_PIP_TB-ALL.dta"
#> [15] "PRY_2006_EPH_V01_M_V05_A_PIP_PC-GPWG.dta"
#> [16] "PRY_2006_EPH_V01_M_V05_A_PIP_TB-ALL.dta"
#> [17] "PRY_2007_EPH_V01_M_V05_A_PIP_PC-GPWG.dta"
#> [18] "PRY_2007_EPH_V01_M_V05_A_PIP_TB-ALL.dta"
#> [19] "PRY_2008_EPH_V01_M_V05_A_PIP_PC-GPWG.dta"
#> [20] "PRY_2008_EPH_V01_M_V05_A_PIP_TB-ALL.dta"
#> [21] "PRY_2009_EPH_V01_M_V06_A_PIP_PC-GPWG.dta"
#> [22] "PRY_2009_EPH_V01_M_V06_A_PIP_TB-ALL.dta"
#> [23] "PRY_2010_EPH_V01_M_V06_A_PIP_PC-GPWG.dta"
#> [24] "PRY_2010_EPH_V01_M_V06_A_PIP_TB-ALL.dta"
#> [25] "PRY_2011_EPH_V01_M_V07_A_PIP_PC-GPWG.dta"
#> [26] "PRY_2011_EPH_V01_M_V07_A_PIP_TB-ALL.dta"
#> [27] "PRY_2012_EPH_V01_M_V04_A_PIP_PC-GPWG.dta"
#> [28] "PRY_2012_EPH_V01_M_V04_A_PIP_TB-ALL.dta"
#> [29] "PRY_2013_EPH_V01_M_V03_A_PIP_PC-GPWG.dta"
#> [30] "PRY_2013_EPH_V01_M_V03_A_PIP_TB-ALL.dta"
#> [31] "PRY_2014_EPH_V01_M_V03_A_PIP_PC-GPWG.dta"
#> [32] "PRY_2014_EPH_V01_M_V03_A_PIP_TB-ALL.dta"
#> [33] "PRY_2015_EPH_V01_M_V03_A_PIP_PC-GPWG.dta"
#> [34] "PRY_2015_EPH_V01_M_V03_A_PIP_TB-ALL.dta"
#> [35] "PRY_2016_EPH_V01_M_V02_A_PIP_PC-GPWG.dta"
#> [36] "PRY_2016_EPH_V01_M_V02_A_PIP_TB-ALL.dta"
#> [37] "PRY_2017_EPH_V01_M_V02_A_PIP_PC-GPWG.dta"
#> [38] "PRY_2017_EPH_V01_M_V02_A_PIP_TB-ALL.dta"
#> [39] "PRY_2018_EPH_V01_M_V03_A_PIP_PC-GPWG.dta"
#> [40] "PRY_2018_EPH_V01_M_V03_A_PIP_TB-ALL.dta"
#> [41] "PRY_2019_EPH_V01_M_V01_A_PIP_PC-GPWG.dta"
#> [42] "PRY_2019_EPH_V01_M_V01_A_PIP_TB-ALL.dta"
```

Yet, if the user need to be more precise in its request, she can add information to the different arguments of the function. For example, this is data available in 2012,

```
pip_find_data(country = "PRY",
              year = 2012)[["filename"]]
#> [1] "PRY_2012_EPH_V01_M_V04_A_PIP_PC-GPWG.dta"
#> [2] "PRY_2012_EPH_V01_M_V04_A_PIP_TB-ALL.dta"
```

6.2.3 Loading data

Function `pip_load_data` takes care of loading the data. The very first instruction within `pip_load_data` is to find the data avialable in the repository by using `pip_load_inventory()`. The difference however is two-fold. First, `pip_load_data` will load the default and/or most recent version of the coun-

try/year combination available. Second, it gives the user the possibility to load different datasets in either list or dataframe form. For instance, if the user wants to load the Paraguay data in 2014 and 2015 used in the Poverty Calculator tool, she may type,

```
df <- pip_load_data(country = "PRY",
                    year    = c(2014, 2015),
                    tool    = "PC")
#> i Loading data and creating a dataframe
#> v Loading data and creating a dataframe ... done
#>

janitor::tabyl(df, survey_id)
#>                               survey_id     n    percent
#>  PRY_2014_EPH_V01_M_V03_A_PIP_PC-GPWG 20109 0.39548834
#>  PRY_2015_EPH_V01_M_V03_A_PIP_PC-GPWG 30737 0.60451166
```

Chapter 7

Poverty Calculator Pipeline (pre-computed estimations)

The Poverty Calculator Pipeline–hereafter, and only for the rest of this chapter, pipeline–is the technical procedure to calculate the pre-computed estimations of the PIP project. These estimations have two main purposes:

1. Provide the user with instantaneous information about distributive measures of all the household surveys in the PIP repository that do not depend on the value of the poverty line. Avoiding thus the need for re-computation as it was the case in PovcalNet for some of these measures.
2. Provide the necessary inputs to the PIP API.

This chapter walks you through the folder structure of the folder, the main R script, `_targets.R`, and the complete and partial execution of the script. Also, it provides some tips for debugging.

7.1 Folder structure

The pipeline is hosted in the Github repository PIP-Technical-Team/pip_ingestion_pipeline. At the root of the repo you will find a series of files and folders.

```
#> +-- batch
#> +-- pip_ingestion_pipeline.Rproj
#> +-- R
#> +-- README.md
#> +-- renv
#> +-- renv.lock
#> +-- run.R
#> +-- _packages.R
```

```
#> \-- _targets
#> \-- _targets.R
```

Folders

- `R` Contains long R functions used during the pipeline
- `batch` Script for timing the execution of the pipeline. This folder should probably be removed
- `_targets` Folder for all objects created during the pipeline. You don't need to look inside as it content is managed by the `targets` package.
- `renv` Folder for reproducible environment.

Files

- `_packages.R` Created by `targets::tar_renv()`. Do not modify manually.
- `_targets.R` Contains the pipeline. This is the most important file.

7.2 Prerequisites

Before you start working on the pipeline, you need to make sure to have the following PIP packages.

Note: Notice that directives below have suffixes like `@development`, which specify the branch of the particular package that you need to use. Ideally, the master branch of all packages should be used, but that will only happen until the end of the development process.

Note2: If you update any of the packages developed by the PIP team, make sure you always increased the version of the package using the function `usethis::use_version()`. Even if the change in the package is small, you need on increased the version of the package. Otherwise, `{targets}` won't execute the sections of the pipeline that run the functions you changed.

```
remotes::install_github("PIP-Technical-Team/pipdm@development")
remotes::install_github("PIP-Technical-Team/pipload@development")
remotes::install_github("PIP-Technical-Team/wbpip@halfmedian_spl")
install.packages("joyn")
```

In case `renv` is not working for you, you may need to install all the packages mentioned in the `_packages.R` script at the root of the folder. Also, make sure to install the most recent version of `targets` and `tarchetypes` packages.

7.3 Structure of the _targets.R file

Even thought the pipeline script looks like a regular R Script, it is structured in a specific way in order to make it work with {[targets](https://docs.ropensci.org/targets/)} package, starting by the fact that it must be called _targets.R in the root of the project. It highly recommended you read the entire targets manual to fully understand how it works. Also, during this chapter, we will referring to the manual constantly to expand in any particular targets' concept.

Start up

The first part of the pipeline sets up the environment. It,

1. loads the {targets} and {tarchetypes} packages,
2. creates default values like directories, time stamps, survey and reference years boundaries, compression level of .fst files, among other things.
3. executes tar_option_set() to set up some option in {targets}. Two particular options are important, packages and imports for tracking changes in package dependencies. You can read more about it in sections Loading and configuring R packages and Packages-based invalidation of the targets manual.
4. Attach all the packages and functions of the project by running source('_packages.R') and source('R/_common.R')

Step 1: small functions

According to the section Functions in pipelines of the targets manual, it is recommend to only use functions rather than expressions during the executions. Presumably, the reason for this is that targets track changes in functions but not in expressions. Thus, this section of the scripts defines small functions that are executed along the pipeline. In the section above, the scripts source('R/_common.R') loads longer functions. Yet, keep in mind that the 'R/_common.R' was used in a previous version of the pipeline before {targets} was implemented. Now, most of the function in 'R/_common.R' are included in the {pipdm} package.

Step 2: prepare data

This section used to longer in previous versions of the pipeline because identified the auxiliary data, loaded he PIP microdata inventory, and created the cache files. Now, it only identifies the auxiliary data. Not much to be said here.

7.3.1 Step 3: The actual pipeline

This part of the pipeline is long and it is explained in detail in the next section. Suffice is to say that the order of the pipeline is the following,

1. Load all necessary data. That is, auxiliary data and inventories, and then create any cache fie that has not been created yet.
2. Calculate means in LCU
3. Crate deflated survey mean (DSM) table
4. Calculate reference year table (aka., interpolated means table)
5. Calculate distributional stats
6. Create output tables
    1. join survey mean table with dist table
    2. join reference year table with dist table
    3. coverage table aggregate population at the regional level table
7. Clean and save.

7.4 Understanding the pipeline

One thing is to understand the how the `{targets}` package works and something else is to understand how the targets of the Poverty Calculator Pipeline are created. For the former, you can read the targets manual. For the latter, we should start by making a distinction between the different types of targets.

In `{targets}` terminology, there are two kinds of targets, **stems** and **branches**. **Stems** are unitary targets. That is, for each target there is only one single R object. **Branches**, on the other hand, are targets that contain several objects or *subtargets* inside (You can learn more about them in the chapter Dynamic branching of the targets manual). We will see the use of this type of targets when we talk about the use of cache files.

Stem targets

There are two ways to create **stem** targets: either using `tar_target()` or using `tar_map()` from the `{tarchetypes}` package. The `tar_map()` function allows to create **stem** targets iteratively. See for instance the creation of targets for each auxiliary data,

```
tar_map(
  values = aux_tb,
  names  = "auxname",

  # create dynamic name
  tar_target(
    aux_dir,
    auxfiles,
    format = "file"
```

```
  ),
  tar_target(
    aux,
    pipload::pip_load_aux(file_to_load = aux_dir,
                          apply_label = FALSE)
  )

)
```

`tar_map()` takes the values in the data frame `aux_tb` created in Step 2: prepare data and creates two type of targets. First, it creates the target `aux_dir` that contains the paths of the auxiliary files, which are available in the column `auxfiles` of `aux_tb`. This is done by creating an internal target within `tar_map()` and using the argument `format = "file"`. This process lets `{targets}` know that we will have objects that are loaded from a file and are not created inside the pc pipeline.

Then, `tar_map()` uses the the column `auxname` of `aux_tb` to name the targets that contain the auxiliary files. Each target will be prefixed by the word "aux". This is why we had to add the argument `file_to_load` to `pipload::pip_load_aux`, so we can let `{targets}` know that the file paths defined in target `aux_dir` are used to create the targets prefixed with "aux", which are the actual targets. For example, if I need to use the population data frame inside the pc pipeline, I'd use the target `aux_pop`, which had a corresponding file path in `aux_dir`. In this way, if the original file referenced in `aux_dir` changes, all the targets that depend on `aux_pop` will be run again.

Branches targets

Let's think of a branch target like

As explained above, branch targets are targets made of many "subtargets" that follow a particular pattern. the Most of the targets created in the pc pipeline are **branch** targets because we need to execute the same procedure in every cache file. This could have been done internally in one single, but then we would lose the tracking features of `{targets}`. Also, we could have created a **stem** target for every cache file, result, and output file, but that would have been not only impossible to visualize, but also more difficult to code. Thus branch targets is the best option.

The following example illustrates how it works,

```
# step A
tar_target(
  cache_inventory_dir,
  cache_inventory_path(),
  format = "file"
```

```
),

# step B
tar_target(
  cache_inventory,
  {
    x <- fst::read_fst(cache_inventory_dir,
                       as.data.table = TRUE)
  },
),

# step C
tar_target(cache_files,
           get_cache_files(cache_inventory)),

# step D
tar_files(cache_dir, cache_files),

# step E
tar_target(cache,
           fst::read_fst(path = cache_dir,
                         as.data.table = TRUE),
           pattern = map(cache_dir),
           iteration = "list")
```

The code above illustrates several things. It is divided in steps, being the last step– step E–the part of the code in which we create the **branch** target. Yet, it is important to understand the steps before.

In step A we create target `cache_inventory_dir`, which is merely the path of the file than contains the cache inventory. Notice that it is returned by a function and not entered directly into the target. Since it is a file path, we need to add the argument `format = "file"` to let `{targets}` know that it is input data. In step B we load the cache inventory file into target `cache_inventory` by providing the target "path" that we created in step A. This file has several column. One of them contains the file path of every single cache file in the PIP network drive. That single column is extracted from the cache inventory in step C. Now, in step D, each file path is declared as input, using the convenient function `tar_files()`, creating thus a new target, `cache_dir`. FInally, we create **branch** target `cache` with all the cache files by loading each file. To do this iteratively, we parse the `cache_dir` target to the `path` argument of the function `fst::read_fst()` and to the `pattern = map()` argument of the `tar_target()` function. Finally, we need to specify that the output of the iteration is stored as a list, using the argument `iteration = "list"`.

The basic logic of **branch** targets is that the vector or list to iterate through

should be parsed to the argument of the function and to the `pattern = map()` argument of the `tar_target()` function. it is very similar to `purrr::map()`

Note: If we are iterating through *more than one* vector or list, you need to (1) separate each of them by commas in the `map()` part of the argument (See example code below). (2) make sure all the vectors or lists **have the same length**. This is why we cannot remove NULL or NA values from any target. (3) make sure you do **NOT** sort any of the output targets as it will loose its correspondence with other targets.

```
# Example of creating branch target using several lists to iterate through.
tar_target(
  name      = dl_dist_stats,
  command   = db_compute_dist_stats(dt       = cache,
                                    mean     = dl_mean,
                                    pop      = aux_pop,
                                    cache_id = cache_ids),
  pattern   =  map(cache, dl_mean, cache_ids),
  iteration = "list"
)
```

Creating the cache files

The creation of the cache files is done in the following code,

```
tar_target(pipeline_inventory, {
  x <- pipdm::db_filter_inventory(
    dt = pip_inventory,
    pfw_table = aux_pfw)

  # Uncomment for specific countries
  # x <- x[country_code == 'IDN' & surveyid_year == 2015]
}
),
tar_target(status_cache_files_creation,
           pipdm::create_cache_file(
             pipeline_inventory = pipeline_inventory,
             pip_data_dir       = PIP_DATA_DIR,
             tool               = "PC",
             cache_svy_dir      = CACHE_SVY_DIR,
             compress           = FST_COMP_LVL,
             force              = TRUE,
             verbose            = FALSE,
             cpi_dt             = aux_cpi,
             ppp_dt             = aux_ppp)
)
```

It is important to understand this part of the pc pipeline thoroughly because the cache files used to be created in Step 2: prepare data rather than here. Now, it has not only been integrated in the pc pipeline, but it is also possible to execute the creation of cache files independently from the rest of the pipeline, by following the instructions in [Executing the _targets.R file].

The first target, `pipeline_inventory` is just the inner join of the pip inventory dataset and the price framework (pfw) file to make sure we only include what the **pfw** says. This data set also contains a lot of information useful for creating the cache files. Notice that the commented line in this target would filter the pipeline inventory to have only the information for IDN, 2015. In case you need to update specific cache files, you have to do add the proper filtering condition in there.

In the second target, `status_cache_files_creation`, you will create the cache files but notice that the returning value of the function `pipdm::create_cache_file()` is not the cache file per-se but a list with the status of the process of creation. If the creation of a particular file fails, it does not stop the iteration that creates the all cache files. At the end of the process, it returns a list with the creation status of each cache file. Notice that function `pipdm::create_cache_file()` requires the CPI and the PPP auxiliary data. This is so because the variable `welfare_ppp`, which is the welfare aggregate in 2011 PPP values, is added to the cache files. FInally, and more importantly, argument `force = TRUE` ensures that even if the cache file exists already, it should be modified. This is important when you require additional features in the cache file from the then ones it has now. If set to `TRUE`, it will replace any file in the network drive that is listed in `pipeline_inventory`. If set to `FALSE`, only the files that are in `pipeline_inventory` but not in the cache folder will be created. Use this option only when you need to add new features to all cache data or when you are testing and only need a few surveys with the new features.

7.5 Understanding {pipdm} functions

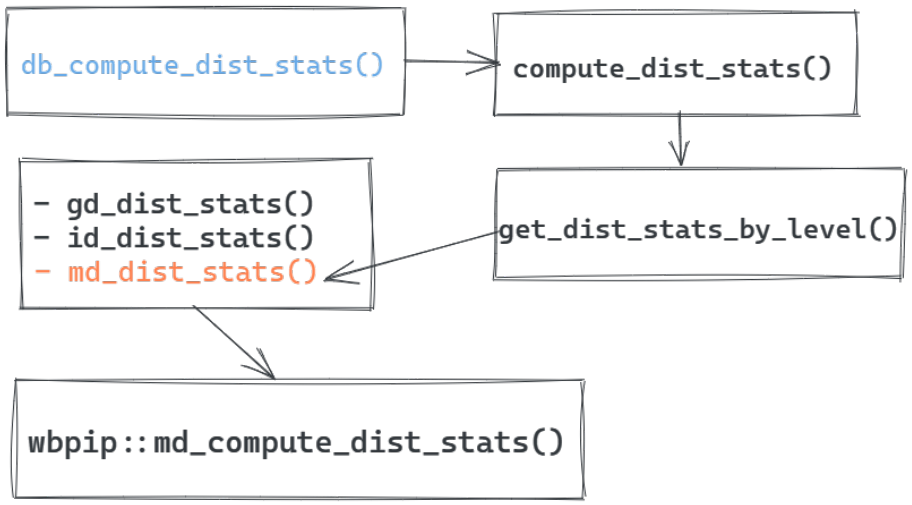
The `{pipdm}` package is the backbone of the pc pipeline. It is in charge of executing the functions in `{wbpip}` and consolidate the new DataBases. This is why, many of the functions in `{pipdm}` are prefixed with "db_".

7.5.1 Internal structure of {pipdm} functions

The main objective of `{pipdm}` is to execute the functions in `{wbpip}` to do the calculations and then build the data frames. As of today, 2021-06-22, the process is a little intricate.

Let's take the example of estimating distributive measures in the pipeline. The image below shows that there are at least three intermediate function levels between the `db_compute_dist_stats()` function, which executed directly in

the pc pipeline, and the `wbpip::md_compute_dist_stats()`, which makes the calculations. Also, notice that the functions are very general in regards to the output. No higher level function is specific enough to retrieve only one measure such as the Gini coefficient, or the median, or the quantiles of the distribution. If you need to add or modify one particular distributive measure, you must do it in functions inside `wbpip::md_compute_dist_stats()`, making sure the new output does not mess up the execution of any of the intermediate functions before the results get to `db_compute_dist_stats()`.



This long chain of functions is inflexible and makes debugging very difficult. So, if you need to make any modification, identify first the chain of execution in each `pipdm` function you modify and then make sure your changes do not affect the output format as it may break the chain of execution. Also, this is a good example to show why this structure needs to be improved.

7.5.2 Updating {pipdm} (or any other PIP package)

As explained above, if you need to modify any function in `pipdm` or in `wbpip` you need to make sure that the output does not conflict with the chain of execution. Additionally, If you update any of the packages developed by the PIP team, make sure you always increased the version of the package using the function `usethis::use_version()`. Even if the change in the package is small, you need to increase the version of the package. Otherwise, `{targets}` won't execute the sections of the pipeline that run the functions you changed. Finally as explained in the Prerequisites, if you are working on a branch different than master, make sure you install that version of the package before running the pipeline.

7.6 Executing the `_targets.R` file

The `.Rprofile` in the root of the directory makes sure that both `{targets}` and `{tarchetypes}` are loaded when the project is started. The execution of the whole pipeline might be very time consuming because it still needs to load all the data in the network drive. If you use a desktop remote connection the execution might be faster than running locally, but it is still very time consuming. So, my advise is that you only execute the targets that are directly affected by your changes and manually check that everything looks ok. After that, you can execute the whole thing confidently and leave it running overnight.

In order to execute the whole pipeline, you only need to type the directive `tar_make()` in the console. If you want to execute only one target, then type the name of the target in the same directive, e.g., `tar_make(dl_dist_stats)`. Keep in mind that if the inputs of prior targets to the objective target have changed, those targets will be executed first.

7.7 Debugging

Debugging in targets is not easy. Yet, there are two ways to do it. The first way is provided in the chapter Debugging of the Targets Manual. It provides clear instruction on how to debug *while still being in the pipeline*, but it could be the case, as it happened to me, that you don't find this method flexible enough to dig deep enough into the problem. Alternatively, you could debug by stepping out of the pipeline a little bit and gain more flexibility. This is how I do it.

You need to debug in one of two case. One, because you got an error when running the pipeline with `tar_make()` or, two, because your results are weird. In either case, you should probably have an idea–though not always–of where the problem is. If the problem is an error in the execution of the pipeline, `{targets}` printed messages are usually informative.

Debugging stem targets

Let's see a simple example. Assume the problem is in the target `dt_dist_stats`, which is created by executing the function `db_create_dist_table` of the `{pipdm}` package. Since the problem is in there, all the targets and inputs necessary to create `dt_dist_stats` should be available in the `_targets/` data store. So, you can load them using `tar_load()` and execute the function in debugging mode. Like this,

```
tar_load(dl_dist_stats)
tar_load(svy_mean_ppp_table)
tar_load(cache_inventory)

debugonce(pipdm::db_create_dist_table)
```

```
pipdm::db_create_dist_table(
  dl        = dl_dist_stats,
  dsm_table = svy_mean_ppp_table,
  crr_inv   = cache_inventory
  )
```

Notice that you need to use the `::` because the environment in which `{targets}` runs is different from your Global environment in which you might not have attached all the libraries.

Debugging branch targets

The problem debugging branch targets is that if the problem is in a specific survey, you can't access the "subtarget" using the survey ID, or something like that, because the name of the subtarget is created by `{targets}` using a random number. This requires a little more of work.

Imagine now that the distributive measures of IDN 2015 are wrong. You see the pipeline and notice that these calculation are executed in target `dl_dist_stats`, which is branch target created over **all the cache files!** It looks something like this,

```
tar_target(
  name      = dl_dist_stats,
  command   = db_compute_dist_stats(dt       = cache,
                                    mean     = dl_mean,
                                    pop      = aux_pop,
                                    cache_id = cache_ids),
  pattern   =  map(cache, dl_mean, cache_ids),
  iteration = "list"
)
```

In order to find the problem in IDN 2015, this what you could do,

```
# Load data
dt <- pipload::pip_load_cache("IDN", 2015, "PC")
tar_load(dl_mean)
tar_load(cache_ids)
tar_load(aux_pop)

# Extract corresponding mean and cache ID
idt      <- which(cache_ids == unique(dt$cache_id))
cache_id <- cache_ids[idt]
mean_i   <- dl_mean[[idt]]

# Esecute the function of interest
debugonce(pipdm:::compute_dist_stats)
```

```
ds <- pipdm::db_compute_dist_stats(dt       = dt,
                                   mean     = mean_i,
                                   pop      = aux_pop,
                                   cache_id = cache_id)
```

First, you load all the inputs. Since target `dl_mean` is a relatively light object, we load it directly from the `_targets/` data store. Targets `cache_ids` and `aux_pop` are data frames, not lists, so we also load them from memory. The microdata, however, is problematic because target `cache`, which is the one that is parsed to create the **actual** `dl_dist_stata` target, is a huge list with all the micro, grouped, and imputed data. The solution is then to load the data frame of interest, using either `pipload` or `fst`.

Secondly, we need to filter the list `dl_mean` and the data frame `cache_ids` to parse only the information accepted by `pipdm::db_compute_dist_stats()` function. This has to be done when debugging because in the actual target this is done iteratively in `pattern = map(cache, dl_mean, cache_ids)`.

Finally, you execute the function of interest. Notice something else. The target `aux_pop` is parsed as a single data frame because `pipdm::db_compute_dist_stats()` requires it that way. Note: This is also one of the reasons these functions in `{pipdm}` need some fixing and consistency in the format of the their inputs.