



Tax Microsimulation at The Budget Lab

Published: April 12, 2024

Model documentation [↗](#)

Introduction

The Budget Lab produces projections of the budgetary, distributional, and economic impacts of policy changes, including changes to tax policy. Our main approach to building these projections is *policy microsimulation*, a kind of model that simulates individual members of a population and calculates their policy-dependent outcomes, like tax liability or health insurance status, under different policy scenarios. We then aggregate these outcomes to project variables like tax revenues or the uninsured rate.

Why microsimulation? There are two key reasons why it is a natural fit for tax policy analysis:

- **Microsimulation is an ideal tool for analyzing policies with heterogeneous effects.** The tax code affects different people differently. Tax liability depends on individual- and family-level characteristics and includes countless interactions. Accurate tax calculation requires knowing the joint distribution of attributes like marital status, number of children, wage income, and homeownership, to name a few. Furthermore, we're interested in the distributional impact of policy changes per se. By focusing on the micro-level, our modeling lets us look at how policy impacts subsets of the population.
- **Microsimulation is an ideal tool for scenario analysis.** The analytical framework for revenue estimation is scenario analysis, where the *what-if* can be about different topics like tax policy ("how much would TCJA extension cost?"), economic projections ("how much will payroll taxes fall if unemployment rises by 1 percentage point?"), or behavioral assumptions ("how does voluntary compliance affect tax revenues?"). A policy microsimulation allows users to examine model results under different assumptions for policy rules and economic scenarios.

Policy microsimulation models consist of two components. The first is a *policy calculator* – a deterministic function representing the law, where inputs are individual characteristics (such as marital status and income), and the output is some policy-dependent variable (e.g. tax liability). The second component is a *population simulator*. This portion projects the population into the future, including its behavioral responses to policy changes. The output of the population simulator is the input to the policy calculator.

In this report, we describe all major facets of our tax microsimulation model including data cleaning procedures, projection assumptions, tax calculator functions, and behavioral responses. Each section provides links to the corresponding model code on our [GitHub page](#).

Data processing and population projections

This section describes how we clean, augment, and project the underlying tax data. The code which implements these calculations and algorithms can be found in the [Tax-Data](#) repository.

IRS PUF

The basis of the tax model is a representative and anonymized sample of tax returns purchased from the IRS. This public-use file, or “PUF”, contains more than 150,000 records (about a 1 percent sample of returns) with information on most variables required to calculate taxes: filing status, number of dependents, income composition, deductions, and more. The IRS also offers a supplemental file with additional demographic information, like age and gender, for a subset of taxpayers. However, for our purposes, the PUF has two major shortcomings:

- **Timing.** For a specific tax year, the PUF only becomes available many years later. For example, as of early 2024, the most recently available PUF is for tax year 2015. This lag means that the underlying data is out of date. Major statistical adjustments are required to “age” the PUF forward to reflect population characteristics of a more recent tax year – our jump-off point for making projections into the future.
- **Limited universe.** By definition, the PUF is limited to the tax-filing universe. But our work requires knowing the characteristics of the non-filing population, who may be affected by counterfactual policy reforms. We also encounter truncation and censoring issues for some variables. For example, we do not observe donations for those who do not elect to itemize deductions. To overcome these limitations, we build statistical models to impute these records and attributes.

Note that our choice to use the PUF (a data source that cannot be exposed to unauthorized users) as our core dataset means that we cannot offer the model as a fully open-source project. One potential alternative approach would be to begin with a publicly available survey microdata source like the Current Population Survey (CPS) and adjust its attributes such that it becomes statistical representative of tax data.¹ We feel that, [at this point in time](#), direct use of the PUF best allows us to accurately model the broadest array of policy reforms of interest to the Budget Lab.

The remainder of this section describes the steps we take to convert the raw PUF into a usable series of projected tax data files for use in our tax model.

Historical aging

It is 2024 at the time of this writing, but the most recently available PUF is from tax year 2015. Our projections require a more recent jump-off point. We overcome this problem by utilizing additional data sources which give us some hints about what the population of tax filers looked like in the years from 2015 through 2023. The first step, then, is *historical aging* – the process by which the PUF is updated (“aged”) to be statistically representative of a more recent tax year. Due to varying data quality at different intervals during this period 2015, there are three discrete steps within the historical aging process.

The first period is 2015-2017, years for which we have supplemental IRS statistics and consistent variable definitions throughout.² To generate a historically aged PUF through 2017, we follow the method described in Section 2.2 of [Ricco \(2020\)](#), which itself adapts an approach described [O’Hare \(2009\)](#). The idea is that record weights are adjusted until the resulting data matches some specified “target” statistical characteristics, which allows us to extrapolate cross-sectional data in year t to some future year $t+n$ without modeling the underlying longitudinal dynamics. In this case, the IRS provides [summary tabulations](#) of income and deduction categories

(both returns counts and dollar values) broken out by AGI group, and the joint distribution of filing status, age range, and AGI; we choose key record count statistics from these tabulations to as our reweighting targets.³ Then, adjustments for each record weight are determined as the result of a linear programming optimization problem, a full mathematical description of which can be found in Ricco (2022). This step updates the data to account for *extensive margin* changes in variables over the period (for example, the share of records between \$40,000 and \$50,000 in AGI with nonzero wage income). Finally, we rescale dollar-amount variable values to close any remaining gap – the intensive margin component of change over the period. The result is a historically aged 2017 PUF.

The second discrete phase of historical aging is 2018-2019, years for which we have supplemental IRS tabulations but the introduction of a major tax reform creates inconsistencies in variable definitions.⁴ Among other changes, the Tax Cuts and Jobs Act (TCJA) increased the standard deduction and limited certain itemized deductions, which led tens of millions of filers to stop itemizing deductions. This change created a truncation problem wherein certain deductible expenses, like charitable deductions, are observable only if the filer has a sufficiently large amount of itemized deductions.⁵ Thus, for some key variables, the IRS tabulations after 2017 are measuring a different universe of filers and are not usable as a direct target. For these years, we take a different approach. First, we update record weights to reflect the composition of growth in the number of tax returns filed by marital status and age. The weights are scaled such that the growth rate of a tax unit reflects the underlying growth composition of the ages of its members. Next, after mapping all dollar-amount variables to one of ten income categories unaffected by TCJA changes, we scale variables by their respective category's per-filer income growth.

The final phase of historical aging is 2020-2023.⁶ The first two years of this period are difficult to model because of the unique recession as well as policy changes which temporarily expanded the universe of filing tax units; for 2022 and 2023, detailed IRS tabulations do not yet exist. For this period, we rely on a combination of historical data sources to age the data. Record weights are first scaled by age- and marital status-specific population growth, as measured by the Social Security Administration (SSA). For income and deduction variables, we again assign each variable to one of 12 categories and scale by per-filer growth.⁷ For income variables, the data source is CBO's supplemental revenue projections file, which provides actual and projected values for 1040 aggregates; for mortgage interest, we use historical mortgage interest paid as measured by the National Income and Product Accounts; for charitable contributions, we use [Giving USA's](#) estimate of total charitable contributions by household.

Imputation: non-filing population

Some Americans are not required to file a tax return, usually because their income is too low. The PUF only contains records for those who file a tax return, but we are often interested in modeling the impact of reforms which would affect current-law non-filers (for example, a fully refundable CTC design). Therefore, we need to simulate the attributes of the non-filing population and add the resulting synthetic records to our dataset. Among existing tax microsimulation models, a common approach is to build tax units in the CPS microdata, determine which simulated tax units do not file returns, then append these records to the PUF; see Ricco (2020), [Perese \(2017\)](#), [Rohaly et al. \(2005\)](#) for descriptions of this general approach. We rely on the work of [Piketty et al. \(2018\)](#), who also use the CPS for this step in their construction of microdata underlying a system of distributional national accounts. We append imputed non-filers from their 2017 public-use microdata file.⁸

Imputation: other variables

Some variables required to accurately calculate taxes under current law or counterfactual policy reform scenarios are incomplete or missing entirely from the PUF. It also lacks key demographic information useful for projecting

the composition of taxpayers into the future. We add several additional variables via statistical imputation.⁹

Major variables include:

- **Age.** Taxpayer age is critical for both accurate tax calculation (some tax provision explicitly condition on age) and population projections (expected shifts in the age distribution will affect total revenues over time). The demographic supplement to the PUF file provides age ranges for primary filers for about two thirds of records; no information is available for the remaining records, nor is any information available on the age of secondary filers in the case of joint returns. To impute age, we begin by comparing the observed number of filers in a given age range/filing status group to that of administrative totals from IRS tabulations. The residual amount is the target for the imputation; we randomly assign age ranges until those are totals are met.¹⁰ Next, we impute age actual ages conditional on range. For each range, we estimate the distribution of age ranges using CPS data and draw from these distributions. Finally, we impute ages for secondary filers by estimating the distribution of age gaps in marriages and drawing from this distribution for joint returns.
- **Dependent ages.** We follow a similar process for imputing dependent ages, ranges for which are available only for a subset of records and sometimes available only for a subset of dependents on a given return. We begin by imputing age ranges for each dependent who does not have an associated range. Probability distributions are broken out by type of dependent (child at home, child away from home, parent, and other) and are estimated on returns with one dependent only, which allows us to precisely map dependent type to age range.¹¹ Then, we estimate probability distributions for within-range ages on CPS data and draw from those distributions to impute specific dependent ages.
- **Gender.** As with age ranges, we observe gender only for a subset of tax returns. [Gender is an important mediator of labor market outcomes and as such is a critical input when modeling behavioral responses involving labor supply and earnings.](#) We impute gender by estimating the target male share of adults by filing status, presence of dependents, and presence of wage income on the microdata files underlying Piketty et al. (2018), then randomly drawing from these distributions for records without gender information.
- **Earnings split.** Payroll taxes are applied at the individual level, but our unit of analysis in the PUF is the tax return. For joint returns in the PUF, individual characteristics like wages and business income are aggregated to the tax return level, though we do have some information on earnings split through the demographic file. To impute earnings split for the remaining filers, we draw from implied distributions from tabulations of the wage and self-employment earnings split for joint returns by earnings percentile from [Saez \(2016\)](#).
- **Pass-through characteristics.** The value of the Qualified Business Income (QBI) deduction depends on a business's industry characteristics and its wage and capital structure. Because the most recent PUF tax year is 2015 but this provision was introduced in 2018, we have none of this information at the micro level and must impute it. First, we assume that 20 percent of pass-through business owners derive their income from a Specified Service Trade or Business (SSTB), a designation that makes the taxpayer ineligible for the QBI deduction if their taxable income is sufficiently high. Next, we impute employer status for each pass-through owner: did your business pay wages to an employee? Probability estimates for this step are taken from [Prisinzano et al. \(2016\)](#). Then, we collect IRS data on aggregate wages paid by legal pass-through entity type (sole proprietorship, partnership, S corporation). We allocate these wages to wage-paying business owners, with 50 percent being allocated equally and the rest being allocated in proportion to positive net income. The final step is to calibrate our imputations such that we broadly match the observed QBI deduction values for 2018 using employer probability as a free parameter. While our estimates match historical aggregates and CBO projections, this closeness is largely by construction. Our methods may not be robust to reforms which dramatically alter the nature of the QBI deduction.

Future aging

To project the tax data beyond the most recent year of historical data (2023 at the time of this writing), we generally rely on CBO's projections.¹² Our method is identical to that of historical aging during the period 2020-2023 as described above, only we use projected values rather than historical data. For demographics, we rely on CBO's projections of SSA-concept population subgroups. For income and deduction variables, we begin with CBO's projections of 1040 variables in the Budget Outlook's supplemental revenue projections. For years beyond the budget window, we rely on CBO's Long-Term Budget Outlook, again mapping PUF variables to broad economic aggregates.

Note that we are not bound to CBO's economic projections. If, for example, we wish to understand the revenue impacts of population aging over the coming decade, we can supply an alternative projection of demographic trends and run the tax model on the resulting microdata. In that sense, economic projections are a parameterizable feature of the model.

Tax calculator

The result of the previous section is a time series of simulated tax microdata for future years. This data is the input to the *tax calculator* – the model component which calculates policy-dependent tax variables for all records. Code for the functions described in this section can be found [here](#).

The tax calculator is not unlike tax preparation software: given some information about a household's characteristics (such as wages, mortgage interest, and marital status), the tax calculator returns tax liability. Crucially, the tax calculator is parameterized. Every major tax parameter is exposed to the user as an input variable. For example, rather than hard-code the CTC value as \$2,000, the tax calculator includes a parameter called "ctc_value", which can take on any value.) This way, users can define a policy scenario by supplying the calculator with the full set of parameters needed to characterize tax law. Current law, the notion of baseline used in scorekeeping convention, is just one instance of a policy scenario configuration. Users can also write new code to characterize policy reforms for which no current-law parameter exists, such as a progressive surcharge on AGI or a charitable credit.

In addition to a payroll tax module, we incorporate all major elements of individual income taxes. Calculator functions are split into one of four categories:

1. Income determination: taxable capital gains, taxable Social Security benefits, AGI, taxable income
2. Deductions: standard deduction, itemized deductions, QBI deduction, personal exemptions
3. Tax liability: ordinary rates, preferred rates, Alternative Minimum Tax (AMT), Net Investment Income Tax (NIIT)
4. Credits against tax: Saver's Credit, Child and Dependent Care Credit (CDCTC), education credits, Child Tax Credit (CTC), Earned Income Tax Credit (EITC), rebate/UBI/stimulus check

These functions rely on two types of input information. The first is the input attributes of a tax unit(s); for example, in the case of the CTC calculator function, this would be number of dependents, earnings, AGI, and more. The second kind of required information is tax law parameters; in the CTC example, this information would include the maximum credit value per child, the phase-in rate, the maximum age, and more. Again, tax law information is supplied as arguments to the function (rather than be hard-coded into the function) so that we can flexibly supply different tax law scenarios to the code to examine the impact of policy changes. In the model, tax

law information is conveyed via a structured set of configuration files. More information on how these files are structured can be found in the Technical Implementation section below.

The tax calculator also contains functionality to estimate different kinds of tax rates. The effective marginal tax rate (EMTR) is defined as the additional amount of tax paid on the next dollar of some type of income. For example, if earning one more dollar of wages increases tax liability by 25 cents, the EMTR is 25 percent. EMTRs often depart from statutory marginal tax rates due to phase-in and phase-out structures within the tax code. The tax calculator measures EMTRs by adding one dollar to a given income (or expense) variable for all records then recalculating taxes; the change in taxes is the EMTR. We can also measure other types of tax rates, for example first-dollar EMTRs (by calculating the difference between taxes when some variable is set to \$0 and \$1) and average effective tax rates (by setting some income variable of initial value \$Y to \$0, recalculating taxes, then dividing the change in taxes by -\$Y).

Behavioral feedback

To this point, we've discussed two of the three dimensions which define a scenario in the tax model – economic projections and tax law. The third is behavioral feedback.

Depending on the [type of revenue estimate we're modeling](#), the tax model incorporates certain types of expected behavioral feedback in response to a policy change. For example, investors may wait to realize capital gains if tax rates rise; owners of pass-through businesses might change legal form of organization when the corporate tax rate falls; low-income parents may drop out of the labor force if EMTRs rise. These kinds of responses are distinct from the baseline economic projections described above – these are *changes* to the baseline projections caused by changes in tax policy. For this reason, behavioral responses are incorporated within the simulation (where tax law is parameterized) rather than the tax data pre-processing stage.

At runtime, the tax model admits a list of user-written behavioral feedback modules. These modules are functions expressing input attributes (e.g. capital gains) as a function of tax policy (e.g. the marginal effective tax rate on capital gains). When tax policy changes, these modules are called and update the input attributes of tax microdata to reflect behavioral feedback. While these functions are standardized in the structure of their input and output, the content of these functions can take any form. The idea is that model users can include behavioral feedback of any functional form in their simulation.

When running a counterfactual policy scenario, the tax simulator runs twice. First, it runs in “static” mode, wherein all tax unit attributes are held fixed at their baseline levels when calculating taxes under the policy reform. This step allows us to assess the first-order effects of a policy — for example EMTRs under the new policy regime. Then, the model runs the policy reform scenario again, this time in non-static mode. Behavioral feedback modules are executed at the beginning of non-static mode. These modules modify input attributes according to their user-specific logic, then Tax-Simulator re-calculates taxes. The result is a policy reform simulation which accounts for behavioral feedback.

Post-processing

Our typical workflow involves configuring one or more counterfactual policy scenarios then running the model for those scenarios (plus the baseline). The core model output comprises of scenario-specific microdata files for each year – in other words, the tax data files supplied as input are appended with derived variables the written as output. This microdata, in turn, is subject to further post-processing to produce the metrics seen in our analyses.¹³ Revenue estimates are one such example. Micro-level estimated tax payments are aggregated by year, scenario,

and budget category to generate time series projections of fiscal receipts. The difference between this series for a reform scenario and the baseline series forms a revenue estimate. We can also decompose a policy reform scenario into its provision-level components. When this feature is activated, *stacked* revenue estimates, wherein provisions are given an order and each provision is scored relative to a baseline that includes all provisions “stacked” before it, are produced during the post-processing step.

We also generate distribution metrics during post-processing. Using the microdata output files, we measure the tax change relative to baseline at the tax unit level for a given scenario. Then, we categorize tax units by some dimension – currently income or age – and calculate category-level averages for various measures of tax change.

Other Budget Lab models also rely on the microdata files produced by the tax model. For example, we characterize tax policy scenarios in our use of FRB/US in part by measuring certain average and marginal tax rates based on tax model output.

Technical implementation

Due to the data privacy constraints described above, we cannot allow readers to run the code directly. But our model code is freely available to view online at our [GitHub page](#). This section offers an overview of the architecture of our model codebase, helping interested readers understand where to find certain model features and how each piece fits together.

The calculations described in this report are split into two codebases, [Tax-Data](#) and [Tax-Simulator](#). Both are written in R, a statistical programming language. Tax-Data processes and projects the tax microdata, which is then used as input to Tax-Simulator. Tax-Data is comparatively simple: it is a linear set of instructions, organized into thematic standalone scripts, each called sequentially in *main.R*.

Tax-Simulator is the broader and larger codebase – it is “the model”. It is designed as a collection of functions which operate on simulated tax microdata (files contained in the */src* folder) and configuration files which parameterize those functions (files contained in the */config* folder). The remainder of this section describes major components in Tax-Simulator, mapping concepts to those described earlier in this document.

Scenario configuration: runscripts

A configuration of Tax-Simulator parameters is called a *scenario*. As described above, three dimensions characterize a scenario: economic and demographic projections, tax law, and behavioral response assumptions. In English, a scenario might be: “TCJA extension against CBO’s economic baseline while assuming a labor supply elasticity of 0.2” or “assume immigration is doubled over the next decade but tax law is unchanged”.

To run Tax-Simulator, we supply a list of scenarios. This list is called a *runscript* – a CSV file where records are scenarios and columns are configuration options. Runscripts can be found in */config/runscripts*. Columns include:

- **id:** name of scenario (“baseline” being a reserved word).
- **tax_law:** file path to tax law configuration folder.
- **behavior:** space-separated file paths to behavioral assumption configuration files.
- Optional “**dep.**” arguments: alternative specifications for *dependencies*, which are data inputs characterizing economic projections. These dependencies are upstream in the Budget Lab data pipeline. Examples include projected tax microdata (*Tax-Data*) and economic and demographic aggregates (*Macro-Projections*). If left blank, defaults from *interface_versions.yaml* are used.

- Other runtime configuration variables controlling forecast horizon, tax rate calculations, and more

Because we are generally interested in comparing counterfactual reform scenarios to the baseline, most runscripts include the baseline. By definition, the baseline scenario is characterized by CBO's economic baseline, current-law tax policy, and no behavioral feedback.

Scenario configuration: tax law

Tax law comprises of all rules which govern calculation of tax liability, such as rates, brackets, phase-out rules, definitions of income, and more. These rules might be scheduled to vary over time and may be indexed to inflation. We represent tax law as an instantiation of *tax parameters* – collections of thematically related tax provisions called *subparameters*. For example, the Child Tax Credit is a tax parameter, made of subparameters like the credit's per-child value, the phase-out threshold, the phase-out rate, the maximum child age, whether a Social Security Number is required, and so forth.

Subparameters are associated with two additional attributes: (1) the time series of its values, expressed as the set of year-value pairs for years when policy changes, and (2) and an optional collection of inflation indexation rules. The latter comprises of four parameters: the measure of inflation, the base year of indexation, the rounding step (e.g. nearest \$100), and the rounding direction.

Each tax parameter is stored in its own YAML file. Top-level elements correspond to subparameters. There are two additional, optional top-level elements. The first is “indexation_defaults”, which lets the user specify default values for indexation attributes when many subparameter share the same indexation rules. The second is “filing_status_mapper” – a dictionary which operates on and aggregates subparameters into a single subparameter which varies by filing status. For example, rather than having to specify individual subparameters for single vs joint returns, we can instead specify that, e.g., the married value is twice the value of the single parameter.

An instance of tax law is represented by a folder containing subparameter YAML files, the filepath to which is supplied in a runscript. For baseline (current law), YAML files for all subparameters are specified. For counterfactual policy scenarios, only changes from baseline need to be specified. Folders can be found in the */config/scenarios/tax_law*.

Scenario configuration: behavioral assumptions

The final dimension of scenario configuration is behavioral feedback: functions mapping changes in tax policy to change in some variable, like wages, capital gains, or pass-through income. Behavioral feedback is configured through the submission of behavior modules – R scripts which contains custom functions which operate on the model's underlying tax data. In this sense, behavior modules provide a standardized interface through which users can impose behavioral feedback assumptions with no restriction on functional form.

Behavioral feedback modules are stored in */config/scenarios/tax_law*. File paths to modules are supplied to runscripts under the “behavior” column argument. [This how-to guide](#) gives further detail on behavior modules: interface requirements, how they integrate with the rest of Tax-Simulator, and some examples of existing modules.

Model code: organization

Model code is organized into four subfolders of */src*:

- **/data.** This folder contains functions which define and build core data structures, including tax units and tax law. It also contains functions for post-processing operations like the construction of revenue estimates and distribution tables.
- **/calc.** This folder contains tax calculation functions, including helper functions to calculate tax rates.
- **/sim.** This folder contains functions governing the flow of scenario execution from reading input data all the way through writing output data. It also includes the functions which parse and execute behavioral feedback modules.
- **/misc.** This folder contains helper functions which parse configuration files and generate lists of global variables.

Model code: data structures

Tax-Simulator is largely written using functions and data structures from the *tidyverse*, a collection of R packages designed for data science applications. At the core of Tax-Simulator is the dataframe of tax units: a tabular data structure where rows represent tax units and columns are attributes of those tax units. For a given year, tax unit microdata is loaded into memory as a dataframe. Over the course of a model run, three kinds of operations occur on the tax unit dataframe:

1. **Merging of tax law attributes.** Tax parameters are represented as attributes of individual tax units – that is, columns in the tax units dataframe. For example, the column “ctc.po_thresh” contains values for the CTC phaseout threshold: under current law in 2024, the value will be \$400K for joint return records, \$200K for single returns, and so on. Tax parameters are merged onto the tax units dataframe at the beginning of a simulation year. This structure lets us efficiently calculate tax variables within calculator functions (more on this below).
2. **Updating of values to reflect behavioral feedback.** Behavior modules directly operate on variables in the tax units dataframe.
3. **Addition of derived tax calculation columns.** Each tax calculator function adds new variables to the tax unit dataframe. For example, the “calc_agi()” function takes as input the tax units dataframe and adds a new variable (“agi”) to it. The final result of tax calculation is a tax units dataframe with all derived 1040 variables as additional columns

Model code: tax calculator

Tax calculation – the process by which we derive all tax variables for a given scenario-year – is implemented as a sequence of function calls where each function handles a specific element of individual income (or payroll) taxes.

Each calculator function takes as input the tax units dataframe, which, at the time of execution, contains all tax law attributes as well as. In For example, calculating itemized deductions requires knowing AGI, which itself is a calculated value. Therefore “calc_agi()” is called prior to “calc_item()”, and the latter function contains calculated values for AGI at the time of its execution. The return value of a calculator function is dataframe containing only the newly derived variables.

Every calculator function lists its required input variables, which are categorized as either being attributes of the tax unit (such as wages, number of children, or itemized deductions) or attributes of tax law (such as). The function will check for the presence of these variables in the columns of the tax units dataframe. Then, custom tax calculation logic is carried out in the body of the function.

To allow for the ability to model the broadest array of reforms in Tax-Simulator, functions are written with an eye towards parameterization and generalizability. For example, the system of preferred-rate brackets for investment income is expressed as a general function [found in Schedule D](#). Nonetheless, some reforms will require the addition of new parameters or even entire functions.

Writing Behavioral Feedback Modules in Tax-Simulator

Footnotes

1. See [PolicyEngine](#) for an example of this kind of approach
2. Corresponding code is found in [process_targets.R](#), [reweight.R](#), and [create_2017_puf.R](#).
3. A full list of our baseline targets can be found [here](#).
4. Code for aging these years can be found in [project_puf.R](#).
5. Of course, this truncation problem existed prior to TCJA as well, albeit to a much smaller degree. It's only a problem for our purposes if we wish to score proposals which expand the universe of itemizers. Because TCJA is temporary and the pre-TCJA itemizing rules are current law starting in 2026, we need to project the pre-TCJA-concept for each itemized deduction variable.
6. Code for aging these years can be found in [project_puf.R](#).
7. Specific variable mappings can be found [here](#).
8. See [impute_nonfilers.R](#) for corresponding model code.
9. The code for these imputations can be found [here](#).
10. The code for this portion of the imputation process can be found [here](#).
11. See [this script](#) for model code.
12. Code for aging these years can be found in [project_puf.R](#).
13. Model code for post-processing can be found [here](#).