censusprofiler_intro

Welcome to `censusprofiler`, a package designed to simplify regionalized census data capture. While `censusprofiler` can perform several functions, at its core, it takes a geographic point, draws a radius, and makes calls to the census api for geographical units around within that radius, and provides output that is more suited for presentation.

All census calls are made via the US Census API. In order to use the Census API (and `censusprofiler`), you will need an API key, which you can obtain here (https://api.census.gov/data/key_signup.html), To save the API key as an environmental variable, run `set_api_key()`.

This package was designed to interface with the American Community Survey in mind. However, the US Census API provides multiple datasets. Implementation of these additional data sources remains a TODO for the package.

Workflows

The basic censusprofile workflow revolves around `profiler()`, which makes data calls and parses the data into a census profile object. Users may select a variety of static geographies for calls (us, state, county, tract, block group), or use two additional methods for selecting a collection of smaller geographies (see below).

Censusprofiler logic is as follows:

profiler -> geographic processing -> census API call -> data processing and formatting -> data return (profile object)

The profile object is a `list` with four data-types, pragmatically named `type1data`, `type2data`, `type3data`, and `type4data`. Type 1 data is a simple dataframe containing every entry by variable and geography. Type 2 data replicates Type 1 data, but removes the summary variable from the dataframe (typically with a _001 suffix). Type 3 data provides aggregate values for the entire geographic area requested. Type 4 data is aggregate, with the summary variable removed.

It is also possible to get an unformatted dataset by using the `simpleReturn` parameter set to `TRUE`.

Single Profile Creation

A profile object is a geographically-bounded data call on provided variables. This geographical boundary may be obtained in three ways: as a static geographical unit (e.g., one or more counties), by supplying an address and radius in miles, or by capturing smaller geographies inside a larger one (e.g., all tracts inside a metropolitan area).

Profile objects are lists with two elements: `info` and `data`. `Info` captures basic supplied information about the profile object, and `data` contains a list with the four datatypes described above.

Let's build a simple profile object to evaluate racial composition of some census tracts in Chicago. We'll take as our address "60 W Walton St, Chicago, IL 60610" and run the following code:

```r
profile <- profiler(name="Chicago Neighborhoods",
                    year=2022,
                    tableID = "B02001",
                    geography="tract",
                    filterAddress = "60 W Walton St, Chicago, IL 60610",
                    filterRadius = 0.5,
                    geosObject = geos)
```

This returns a list object:

```r
print(names(profile))
#> [1] "info" "data"
```

The first list item (`info`) contains supplied and deduced geographic details about the entity around which the profile is built.

```r
print(names(profile$info))
#> [1] "name"        "address"     "radius"      "year"        "coordinates"
#> [6] "buffer"      "states"      "counties"    "tracts"
print(profile$info)
#> $name
#> [1] "Chicago Neighborhoods"
#>
#> $address
#> [1] "60 W Walton St, Chicago, IL 60610"
#>
#> $radius
#> [1] 0.5
#>
#> $year
#> [1] 2022
#>
#> $coordinates
#> Simple feature collection with 1 feature and 0 fields
#> Geometry type: POINT
#> Dimension:     XY
#> Bounding box:  xmin: -87.63039 ymin: 41.89986 xmax: -87.63039 ymax: 41.89986
#> Geodetic CRS:  WGS 84
#>                     geometry
#> 1 POINT (-87.63039 41.89986)
#>
#> $buffer
#> Simple feature collection with 1 feature and 0 fields
#> Geometry type: POLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -87.64015 ymin: 41.89256 xmax: -87.62064 ymax: 41.90715
#> Geodetic CRS:  WGS 84
#>                         geometry
#> 1 POLYGON ((-87.62612 41.9063...
#>
#> $states
#> [1] "17"
#>
```

```
#> $counties
#> [1] "031"
#>
#> $tracts
#>  [1] "081500" "081300" "081700" "081800" "081201" "081000" "080300" "080202"
#>  [9] "081100" "081600" "081401" "838300" "081202" "080100"
```

The second list item (`data`) contains the four data types described above.

```
print(names(profile$data))
#> [1] "type1data" "type2data" "type3data" "type4data"
```

These dataframes contain more information than supplied by the raw API call, in an effort to simplify the data acquisition process, to allow you to focus more on using the data.

```
print(names(profile$data$type1data))
#>  [1] "table_id"          "year"              "variable"          "concept"
#>  [5] "labels"            "estimate"          "subtotal"          "pct"
#>  [9] "subtotal_by_type" "pct_by_type"       "moe"               "name"
#> [13] "geography"         "state"             "county"            "tract"
#> [17] "geoid"             "calculation"       "type"              "type_base"
#> [21] "varID"             "dt"
```

Column names are described below:

- `table_id`: This is a grouping variable. On the Census website, it is referred to as "group". However, I have adopted the language of table. A table_id of B02001 holds all variables with that prefix (i.e., B02001_001, B02001_002, B02001_003, etc).
- `year`: The year fetched. For ACS data, it may be any year between 2009 and current year.
- `variable`: Dataframes list each individual variable with its associated values.
- `concept`: Concepts are short descriptions of their respective table_id.
- `label`: Labels are short descriptions of the individual variables, within that concept.
- `estimate`: Because we are primarily focused on the ACS, we use "estimate" rather than "value" or "total." The ACS uses sampling procedures to render estimates, based on the decennial census for each geographical area. (See `moe` below)
- `subtotal`: Subtotals are calculated on the basis of geographical units (or geoids). Because profile objects often include multiple tracts, or counties, it is helpful to see subtotals of table_ids x geographic unit. These are clearest in type 1/2 data. Total combined area tallies are found in type 3/4 data.
- `pct`: Calculated estimate percentages of subtotals, per geographic unit.
- `subtotal_by_type`: Variables sometimes include subvariables. For example, in table_id B02001, one option is "Two or More Races", consisting of two subvalues. The hierarchy is captured in column `type`, and `subtotals_by_type` and `pct_by_type` both calculate tallies and percentages with respect to these hierarchies, rather than on a flat basis.
- `pct_by_type`: See `subtotal_by_type`. Note: `pct` will show percent of estimates relative to subtotal, while pct_by_type will show percent of estimates relative to its parent variable.
- `moe`: Because ACS data is comprised of estimates, the API also returns the margin of error for each variable/geography value.
- `name`: The geographical unit captured by name.
- `geography`: The depth of geographical "resolution" (e.g., state, county, tract, block group).
- `state` / `county` / `tract` / `block group`: The FIPS code for each the larger geographic units containing the primary geographic unit.
- `geoid`: A unique identifier for the geographic unit, comprised of nested FIPS codes (i.e., stateFIPS+countyFIPS+tractFIPS).

- `calculation`: Whether a simple count, or median.
- `type`: A derived value identifying the variable type, or level. In the data returned by the Census API, this is denoted by label patterns. For example, in the ACS, the subtotal variable B02001_001 has the label "Estimate!!Total:", while the variable B02001_008 ("Two or more races") has the label "Estimate!!Total:!!Two or more races:". The subvariable B02001_009 ("Two races including Some other race") has the label "Estimate!!Total:!!Two or more races:!!Two races including Some other race". To ease in identification and manipulation of variables, these have been tagged as "root", "summary", and "level_1" respectively.
- `type_base`: Similar to `type`, `type_base` identifies type, but in a broader fashion: it simply identifies the root variable, and all other subordinate variables.
- `varID`: Sometimes it can be helpful to indicate the suffix of the variable. This is provided in simple numerical format.
- `dt`: Identifier indicating what type of data this is (1-4).

Optimization

The `censusprofiler` logic may be optimized by preloading several data objects:

- census variables
- geography variable object

These can be obtained by running `load_data(load_censusVariables=TRUE, load_geos=TRUE)`. The census variables object provides access to all variables available for the dataset queried (e.g., "acs5"), and is used for error-checking and dataframe formatting. The geography variable object reduces data calls for geographical functions by preloading the relevant shapefiles for the area queried. Both of these objects may be passed to `profiler()` and other functions and speed up processing time immensely.

Standalone Functions

`censusprofiler` provides two additional functions which can be implemented as standalone utilities, or in a rmarkdown report.

tabler()

To display data in a simple table, call tabler(). The function may be used with an existing profile object, or in a standalone manner. If standalone, tabler() uses profiler() to capture data in the background. Output is generated using the `flextable`. This is a convenience function with minimal customization, designed primarily to give quick output for reviewing results.

```r
tabler(profile,
       tableID = "B02001",
       dispPerc = TRUE)
```

Label	Est. (n)	Est. (%)
White alone	54,582	74%
Black or African American alone	3,349	5%
American Indian and Alaska Native alone	468	1%

Label	Est. (n)	Est. (%)
Asian alone	8,989	12%
Native Hawaiian and Other Pacific Islander alone	8	0%
Some Other Race alone	1,624	2%
Two or More Races:	4,422	6%
Two races including Some Other Race	*2,195*	*50%*
Two races excluding Some Other Race, and three or more races	*2,227*	*50%*

One interesting benefit of tabler() is the ability to present comparison statistics. If you create comparison tables (using create_comparison_data()), you can pass these into tabler() to view how your geographic area statistics compare to statewide, or nationwide statistics.

```r
tabler(profile,
       tableID = "B02001",
       dispPerc = TRUE,
       stateCompare = statesCompare,
       state = 17,
       usCompare = usCompare)
```

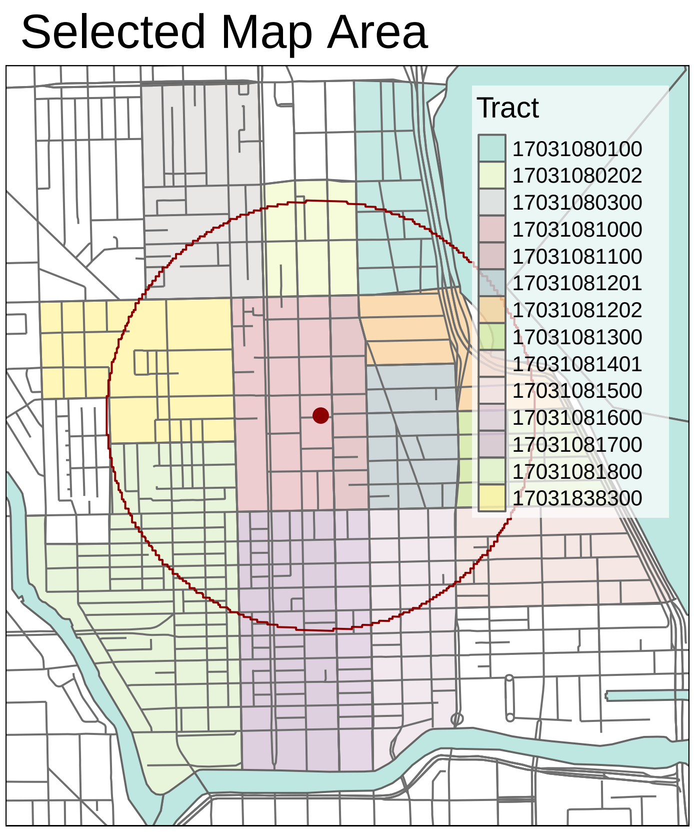
Label	Est. (n)	Est. (%)	Tot. US %	Tot State
White alone	54,582	74%	64%	61
Black or African American alone	3,349	5%	12%	13
American Indian and Alaska Native alone	468	1%	1%	0
Asian alone	8,989	12%	5%	5
Native Hawaiian and Other Pacific Islander alone	8	0%	0%	0
Some Other Race alone	1,624	2%	5%	6
Two or More Races:	4,422	6%	7%	7
Two races including Some Other Race	*2,195*	*50%*	*3%*	*4*
Two races excluding Some Other Race, and three or more races	*2,227*	*50%*	*3%*	*3*

mapper()

Additionally, we have employed the `tmap` package to provide mapping capacities to the data collected. This function, like tabler(), can be utilized with existing data, or in a standalone fashion. It provides several options for output, including a simple map of selected geographical units with or without a radius overlay, density maps, and proportion maps when profile object data is supplied. Additionally, because we use `tmap`, both static and interactive maps are available for output.

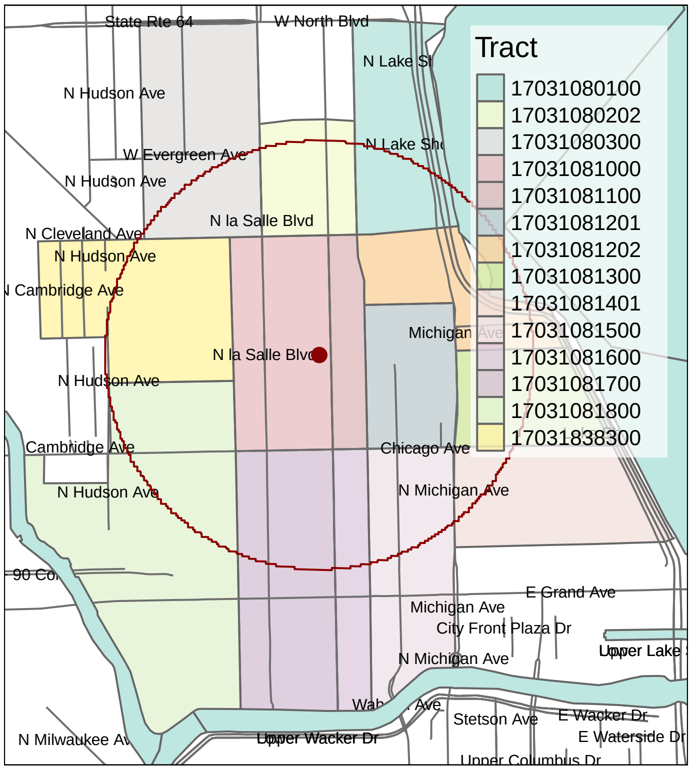
Street labeling is less than ideal in static tmap renderings. The interactive mode is far more interesting. However, the interactive mode is not suitable for printing (i.e., this document). However, you can exercise some control over what is displayed by setting the parameter `road_resolution` between 1-5.

```r
mapper(mapDF = profile,
       geography = "tract",
       MapTitle = "Selected Tracts by Radius",
       geosObject = geos,
       radiusOnly = TRUE,
       road_resolution = 3)
#>   |                                                                                 |
```



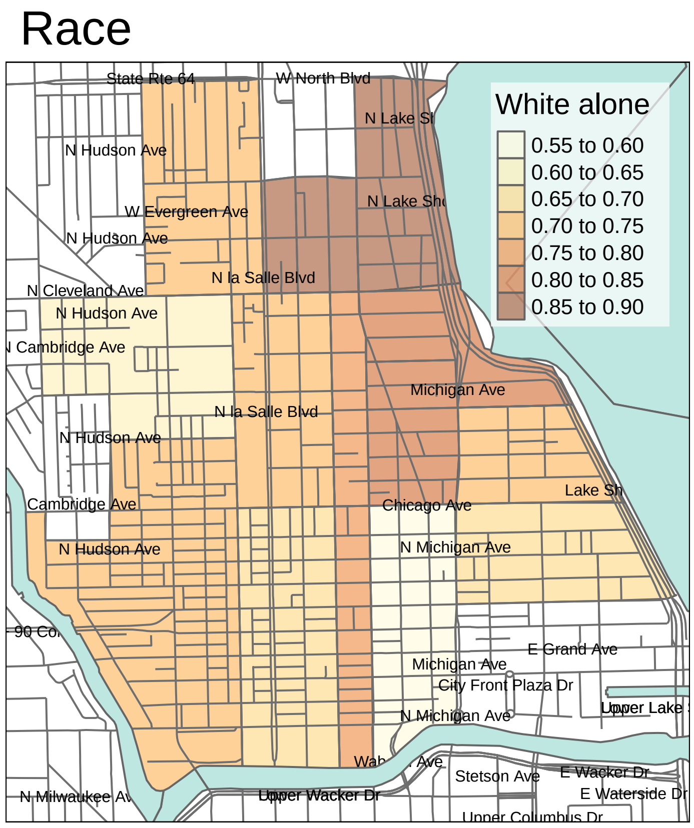
```r
mapper(mapDF = profile,
       MapTitle = "Selected Tracts by Radius",
       geography = "tract",
       geosObject = geos,
       radiusOnly = TRUE,
       road_resolution = 4)
```

Selected Map Area



Mapper() also allows us to provide a variable for proportional visual analysis. Variables can be given by their explicit name (e.g., "B02001_002), or by simple numerical value (e.g. 2). If `dispPerc` is set to FALSE, mapes will be colored by relative estimate values. If set to TRUE, maps will be colored by relative proportion values.

```
mapper(mapDF = profile,
       tableID = "B02001",
       variable = 2,
       geography = "tract",
       dispPerc = TRUE,
       geosObject = geos,
       road_resolution = 2,
       censusVars = CV)
```



Conclusion

This short introduction covers the basics, but there's more to discover with `censusprofiler`. We encourage you to dig into the functionality, report any issues, and have fun exploring the US Census!