iNZightTools: Tools for **iNZight**

Tom Elliott

Te Rourou Tātaritanga, Victoria University of Wellington
Department of Statistics, University of Auckland

Abstract

A package to do things.

1 Introduction

iNZight (Elliott et al., inproc) is a graphical user interface (GUI) for easy data exploration and visualisation. As part of its development, we needed wrapper packages to translate between the user interface and R functions.

- GUI has input fields user can population, equivalent to function arguments
- generally, one window talks to one function (but not always)
- often some complicated manipulation to go from user input values to valid function arguments
- doing this in a GUI is bad practice
- instead developed intermediary package that takes simple inputs and produces a `data.frame` output
- overtime this developed to also attach the **tidyverse** (Wickham et al., 2019) code used to perform the action
- simple function/argument interface makes it easier for beginners

There are a range of different classes of methods in **iNZightTools**. Many are data manipulation functions linking to **dplyr** (Wickham et al., 2020) or similar. However there are others that are 'clever' helper functions for choosing the correct function for a given situation.

- import data from a range of file formats with `smart_read()`
- import a survey design from a specification file format with `import_survey()`

2 Designing code-writing functions

Unlike the majority of R packages, **iNZightTools** provides little new functionality to users, but instead wraps existing functionality from a range of other packages in an attempt to provide a simple, stable application programming interface (API) for both users new to R, but more importantly the GUI software package **iNZight**. These *wrapper functions* need to simplify the inputs to many more complex methods in such as way as to provide arguments that can be connected to graphical inputs such as dropdown boxes and sliders. Second, they must compile and execute the necessary code to perform the required action(s). A side-effect of this step is that the function generates code—in our case typically creating calls to **tidyverse** packages—which can be attached to the result and allow users to inspect the code required to perform the chosen action. This produces a system by which users new to R can begin exploring more complicated functions by modifying existing code.

2.1 Choosing function arguments

Arguably the most important part of designing a wrapper function is the choice of arguments. By design, these methods are not supposed to provide the full set of functionality available from the underlying packages, but instead should provide a simple subset of features that are easy to access with minimal effort. An important part of **iNZight** is the use of *smart defaults*, and this starts off by specifying good defaults for as many arguments as possible. Take, for example, a function for creating class intervals from a numeric variable. The minimum information required would be the variable name, and everything else can provide "good defaults", for example the number of intervals. In this way, R users can call the function on a variable in a dataset and get a result instantly, and build up from there by specifying additional arguments.

Within **iNZightTools**, most of the argument choice has been decided by the requirements of the GUI, though in most cases this is a reasonable set of features for beginners to familiarize themselves with. However, not all arguments are equal, and often some are dependent on others, while in other scenarios some

arguments are needed only when another is ignored. In these situations, we rely on R's lazy-evaluation to ignore unused arguments (which may depend on variables not defined).

2.2 Constructing calls

The basic framework **iNZightTools** uses to construct calls using R's *expression* syntax, prefixing the call with `~`. The basic form of the call is then written out using placeholder variables in the parts that will be modified by the users. In simple expressions, one single call is required, while in more complex methods several steps are often required, including context-specific steps (within `if-else` or `switch` statements).

Once the main structure of the call is complete, the individual arguments must be put together. In some situations this is as simple as passing the argument from the function call, while in others it requires preparing data structures (such as a named vector or list), as required by the underlying function. Details for some specific cases are given in the following sections.

2.3 Evaluating calls and returning results

Once the components are ready, we pass the function through two methods: `replaceVars()` and `interpolate`. The first, `replaceVars()`, substitutes placeholder variables with the names of created structures (named vectors and lists) which we wish to appear in the final code statement. Secondly, `interpolate()` evaluates the expression in the current environment, but can accept additional arguments specifying the values of additional arguments (notably character values). Not only does `interpolate()` evaluate the expression and return the result, but additionally attaches the expression to the objected, stored as a `"code"` attribute. In all(?) cases, the result returned is a `data.frame` or `tibble`.

To extract the code from the returned object, there is the `iNZightTools::code()` function. GUIs can use this to extract code from returned objects and stored it in code history, while users can examine the code used and modify it to access more advanced aspects of the underlying methods.

In the following sections, I will describe some specific implementations for three important components of the **iNZightTools** package: *data import*, *data wrangling*, and *variable manipulation*.

Table 1: File types supported by smart_read().

Extension	Format	Function(s)	Package
txt	Tab Delimited File	read_text/read_delim	readr
csv	Comma Separated Values File	read_csv/read_delim	readr
sav	SPSS Save File	read_sav	haven
sas7bdat, xpt	SAS Data Files, SAS XPORT Files	read_sas, read_xpt	haven
xls, xlsx	Excel Files	read_excel	readxl
dta	STATA Files	read_dta	haven
json	JSON Data	fromJSON	jsonlite
rds	Serialized R Object	readRDS	base
svydesign	Survey Design File	import_survey	surveyspec

3 Importing Data

- import data with the `smart_read()` function
- uses file extension to guess best package and function to use, e.g., `.xlsx` uses `haven::read_excel()` (Wickham and Miller, 2020)
- also handles metadata parsing for comma separated values (CSV)
- subsection about datatypes (numeric, factor/categorical, and date-time), with links to subsections with details

Before any kind of analysis can be undertaken, users mustfirst import their dataset into R. This can be an arduos process for novices, especially if the data is not in a CSV format and thus requires use of an additional R package to import. Similarly, trying to design an R GUI module to import data in a wide range of formats would become quite complicated as the number of formats increase. Our solution was to create a single function, `smart_read()`, that takes a file path or URL as the only required argument and imports the data using the file extension (.csv, .xlsx, etc) to figure out the appropriate package and function to use to import it. At the time of writing, `smart_read()` can read 11 file extensions, listed in table 1.

When combined with the code writing functionality from section 2, this provides a powerful way for beginners and non-beginners alike to quickly import a dataset without first figuring out what function to use. Once the initial read has been performed, it can be examined and if changes are needed, these can either be made to the call to `smart_read()`, or by checking the code and modifying it as needed.

3.1 Parsing Metadata for Easier Data Distribution and Import

- often data coded (factors as numbers instead of labels)
- users need to refer to information (often external) to first set-up the variables correctly before they can get started with visualisation
- this is hard/not feasible for novice users
- metadata can be included in/distributed with the raw data
- `smart_read()` will parse the metadata and apply transformations
- here are some examples

On top of the standard data import possibilities, **iNZightTools** also includes its own "metadata" specification, allowing data distributers to specify information about the variables in the dataset. This format will be familiar to users acquainted with **roxygen2** . Some key features include automatically coding integer values as factors, which can greatly reduce the size of a CSV file but skip a painful step (particularly for GUI users).

The `smart_read()` function automatically checks for metadata at the top of a CSV file, and parses it if present. This triggers modification of function arguments, for example `readr::read_csv()`, where possible (for example when parsing integers as a factor variable), and in other cases uses `dplyr::mutate()` to modify the data object returned, for example if factor levels are renamed or combined.

3.2 Data types

There are many data types understood by R, however for most data analysis situations these can be condensed into numeric, factors/categorical, and date-times. **iNZightTools** automatically coerces all variables to one of these (characters to factors for historic reasons, and restricts the values a variable can take, particularly useful for novice users interfacing through a GUI). Of course, the code used to do this is all generated using the principles of section 2, so can be viewed by the user and modified as needed.

Date-time variables are read automatically if the package supports it (e.g., `readr::read_csv()`). The format depends on whether the value is a date, a

datetime, or a time ("duration"). Table X shows which value types are used, and the package used to support this. Working with these types of variables is facilitated by some variable manipulation methods, which are described in section 5.

4 Data wrangling with iNZightTools

- a bunch of methods: filter, sort, aggregate, join
- here are some examples
- and accessing the code
- dataset validation

5 Variable manipulation methods

- renaming, releveling, reordering, transforming
- some date-time specific operations

6 Conclusion

- summarize
- what's next (plots, GUI, etc? maybe?)

Acknowledgements

iNZight is free software developed by students of the Department of Statistics, University of Auckland. Recent work has been made possible by Te Rourou Tātaritanga (`https://terourou.org`), a research group at Victoria University of Wellington funded by an MBIE Endeavour Grant, ref 62506 ENDRP, with additional funding from iNZight Analytics.

References

T. Elliott, C. Wild, D. Barnett, and A. Sporle. **iNZight**: A graphical user inferface for data visualisation and analysis through R. inproc.

H. Wickham and E. Miller. ***haven**: Import and Export SPSS, Stata and SAS Files*, 2020. URL `https://CRAN.R-project.org/package=haven`. R package version 2.3.1.

H. Wickham, M. Averick, J. Bryan, W. Chang, L. D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T. L. Pedersen, E. Miller, S. M. Bache, K. Müller, J. Ooms, D. Robinson, D. P. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, and H. Yutani. Welcome to the **tidyverse**. *Journal of Open Source Software*, 4(43):1686, 2019. doi: 10.21105/joss.01686.

H. Wickham, R. François, L. Henry, and K. Müller. ***dplyr**: A Grammar of Data Manipulation*, 2020. URL `https://CRAN.R-project.org/package=dplyr`. R package version 1.0.2.