Interacting with The Demographic and Health Surveys (DHS) Program data sets

2019-01-31

There seem to be a lot of ways to write about your R package, and rather than have to decide on what to focus on I thought I'd write a little bit about everything. To begin with I thought it best to describe what problem `rdhs` tries to solve, why it was developed and how I came to be involved in this project. I then give a really brief overview of what the package can do, before continuing to describe how writing my first proper package and the rOpenSci review process was. Lastly I wanted to share a couple of things that I learnt along the way. These are not very clever or difficult things, but rather things that were difficult to Google, which now I think about it should probably be the best metric for a difficult problem.

Motivation

What is the DHS Program

The Demographic and Health Survey (DHS) Program has collected and disseminated population survey data from over 90 countries for over 30 years. This amounts to over 400 surveys that give representative data on health indicators, which in many countries provides the key data that mark progress towards targets such as the Sustainable Development Goals (SDGs) and inform health policy such as detailing trends in child mortality [Silva et al. 2012 and characterising the distribution of malaria control interventions in Africa in order to map the burden of malaria since the year 2000 Bhatt et al. 2015.

This is all to the say that the DHS provides really useful data. However, although standard health indicators are routinely published in the survey final reports that are published by the DHS programme, much of the value of the DHS data is derived from the ability to download and analyse the raw datasets for subgroup analysis, pooled multi-country analysis, and extended research studies.

This where I got involved, in trying to create a tool that helped enable researchers to quickly gain access to the raw data sets.

How did I get involved

I am fortunate enough to be a PhD student in a really large department at Imperial College London, which means that I get the opportunity to be involved in many projects that are outside the scope of my actual PhD. The "downside" of that is sometimes you get given "code monkey" jobs as the bottom rung of the monkey ladder. And so, a few months into my PhD (Nov 2016), I was given the job of downloading data on malaria test results from the DHS programme that was going to be used by some collaborators. At the time I was very happy to be involved, however, I was apprehensive to spend too long on the job as I didn't know how much time to be spending on side projects vs my PhD (something I still don't know with 6 months to go). This combined with only having a year or so's experience writing R meant that the code I wrote to do the job was a bunch of scrappy scripts that required manually downloading the datasets before parsing them with these R scripts. Dirty but it got the job done.

Some time passed, and another collaborator wanted some different data collated from the DHS programme. At this point, I had 6 more months familiarity with R and knew a bit more so I started writing it as an R package. However, it was still messy and it required manually downloading the datasets first, but I was happy with it and again it wasn't a major project of mine. This would have been probably where the project ended if I hadn't had a conversation (Sept 2017) in the tea room (prompted solely by the presence of free biscuits) with the other main author of `rdhs`, Jeff Eaton.

We got chatting, and realised we both had a bunch of scripts for doing bits of the analysis pipeline. We also realised that we had both had numerous requests for data sets from the DHS programme at which point we thought it would be best to do something properly. I had also at this point been keen to start using `testhat` within my work as I had been told it would save me time in the future, and up till that point I hadn't found a good case to get to grips with it (mainly writing code on my own, that was never very big and was only used by myself). And so we started writing `rdhs`, which was accepted by `ropensci` and `CRAN` in December 2018.

Package overview

> Disclaimer: A lot of the following section (the **API** and **Dataset Downloads** headings) is very similar to the Introduction Vignette. So if you want just an overview of the package then head there. However, more of my ramblings about the package development process can be found after.

Most of the functionality of `rdhs` can be roughly summarised in the 5 main steps that are involved from wanting to get data on x to having a curated data set created from survey data from multiple surveys. These steps involve:

1. Accessing standard survey indicators through the DHS API.
2. Using the API to identifying the surveys and datasets relevant to your particular analysis, i.e. the ones that ask questions related to your topic of interest.
3. Downloading survey datasets from the DHS website.
4. Loading the datasets and associated metadata into R.
5. Extracting variables and combining datasets for pooled multi-survey analyses.

We will explore these 5 main steps, with the first 2 showing how `rdhs` functions as an API client and the last 3 points showing how `rdhs` can be used to download raw data sets from the DHS website. Before we have a look at these, let's first load `rdhs`:

```
library(rdhs)
```

API

1. Access standard indicator data via the API

The DHS programme has published an API that gives access to a number of different data sets, which each represent one of the DHS API endpoints (e.g. https://api.dhsprogram.com/rest/dhs/tags, or https://api.dhsprogram.com/rest/dhs/surveys). Each of these data sets are described within the DHS API website, and there are currently 12 different data sets available from the API. Each of these data sets can be accessed using anyone of `dhs_<>()` functions.
All exported functions within `rdhs` that start *dhs_* interact with a different data set of the DHS API.

One of those functions, `dhs_data()`, interacts with the the published set of standard health indicator data calculated by the DHS.

```
## what are the indicators
indicators <- dhs_indicators()
str(indicators[1,])
```

```
## 'data.frame':    1 obs. of  22 variables:
##  $ Definition            : chr "Age-specific fertility rate for the three years preceding the survey
##  $ NumberScale           : int 0
##  $ IndicatorType         : chr "I"
##  $ MeasurementType       : chr "Rate"
##  $ IsQuickStat           : int 0
##  $ ShortName             : chr "ASFR 10-14"
##  $ IndicatorId           : chr "FE_FRTR_W_A10"
```

```
##  $ Level1                 : chr "Fertility"
##  $ IndicatorTotalId       : chr ""
##  $ Level2                 : chr "Fertility rates"
##  $ Level3                 : chr "Women"
##  $ SDRID                  : chr "FEFRTRWA10"
##  $ IndicatorOldId         : chr ""
##  $ TagIds                 : chr ""
##  $ DenominatorWeightedId  : chr ""
##  $ Label                  : chr "Age specific fertility rate: 10-14"
##  $ IndicatorOrder         : int 11763005
##  $ Denominator            : chr "Per thousand women years exposed in the period 1-36 months prior to
##  $ QuickStatOrder         : chr ""
##  $ IndicatorSpecial1Id    : chr ""
##  $ DenominatorUnweightedId: chr ""
##  $ IndicatorSpecial2Id    : chr ""
```

Each call to the DHS API returns a `data.frame` by default with all the results available by default.

Since there are quite a lot of indicators, it might be easier to first query by tags. The DHS tags their indicators by what areas of demography and health they relate to, e.g. anaemia, literacy, malaria parasitaemia are all specific tags.

```
# What are the tags
tags <- dhs_tags()

# Let's say we want to view the tags that relate to malaria
tags[grepl("Malaria", tags$TagName), ]
```

```
##    TagType                   TagName TagID TagOrder
## 31       0       Malaria Parasitemia    36      540
## 43       2 Select Malaria Indicators    79     1000
```

Depending on your analysis this maybe more than enough detail, and we can now quickly access data from the API. For example, to find out the trends in antimalarial use in Africa, and see if perhaps antimalarial prescription has decreased after RDTs were introduced (assumed 2010).

```
# Make an api request
resp <- dhs_data(indicatorIds = "ML_FEVT_C_AML", surveyYearStart = 2010,breakdown = "subnational")

# filter it to 12 countries for space
countries  <- c("Angola","Ghana","Kenya","Liberia",
                "Madagascar","Mali","Malawi","Nigeria",
                "Rwanda","Sierra Leone","Senegal","Tanzania")

# and plot the results
library(ggplot2)
ggplot(resp[resp$CountryName %in% countries,],
       aes(x=SurveyYear,y=Value,colour=CountryName)) +
  geom_point() +
  geom_smooth(method = "glm") +
  theme(axis.text.x = element_text(angle = 90, vjust = .5)) +
  ylab(resp$Indicator[1]) +
  facet_wrap(~CountryName,ncol = 6)
```

2. Identify surveys relevant for further analysis

You may, however, wish to do more nuanced analysis than the API allows. The following 4 sections detail a very basic example of how to quickly identify, download and extract datasets you are interested in.

Let's say we want to get all DHS survey data from the Democratic Republic of Congo and Tanzania in the last 5 years (since 2013), which covers the use of rapid diagnostic tests (RDTs) for malaria. To begin we'll interact with the DHS API to identify our datasets.

To start our extraction we'll query the *surveyCharacteristics* data set using `dhs_survey_characteristics()` function:

```
## make a call with no arguments
sc <- dhs_survey_characteristics()
sc[grepl("Malaria", sc$SurveyCharacteristicName), ]
```

```
##    SurveyCharacteristicID SurveyCharacteristicName
## 57                     96            Malaria - DBS
## 58                     90     Malaria - Microscopy
## 59                     89            Malaria - RDT
## 60                     57          Malaria module
## 61                      8 Malaria/bednet questions
```

There are 87 different survey characteristics, with one specific survey characteristic for Malaria RDTs. We'll use this to then find the surveys that include this characteristic.

```
# lets find all the surveys that fit our search criteria
survs <- dhs_surveys(surveyCharacteristicIds = 89,
                     countryIds = c("CD","TZ"),
                     surveyType = "DHS",
                     surveyYearStart = 2013)

# and lastly use this to find the datasets we will want to download
# and let's download the flat files (.dat) datasets
datasets <- dhs_datasets(surveyIds = survs$SurveyId,
                         fileFormat = "flat",
                         fileType = "PR")
str(datasets)
```

```
## 'data.frame':    2 obs. of  13 variables:
##  $ FileFormat          : chr  "Flat ASCII data (.dat)" "Flat ASCII data (.dat)"
##  $ FileSize            : int  6595349 6622102
##  $ DatasetType         : chr  "Survey Datasets" "Survey Datasets"
##  $ SurveyNum           : int  421 485
##  $ SurveyId            : chr  "CD2013DHS" "TZ2015DHS"
##  $ FileType            : chr  "Household Member Recode" "Household Member Recode"
##  $ FileDateLastModified: chr  "September, 19 2016 09:58:23" "August, 07 2018 17:36:25"
##  $ SurveyYearLabel     : chr  "2013-14" "2015-16"
##  $ SurveyType          : chr  "DHS" "DHS"
##  $ SurveyYear          : int  2013 2015
##  $ DHS_CountryCode     : chr  "CD" "TZ"
##  $ FileName            : chr  "CDPR61FL.ZIP" "TZPR7AFL.ZIP"
##  $ CountryName         : chr  "Congo Democratic Republic" "Tanzania"
```

We can now use this to download our datasets for further analysis.

Dataset Downloads

3. Download survey datasets

We can now go ahead and download our datasets. To be able to download survey datasets from the DHS website, you will need to set up an account with them to enable you to request access to the datasets. Instructions on how to do this can be found here. The email, password, and project name that were used to create the account will then need to be provided to `rdhs` when attempting to download datasets.

Once we have created an account, we need to set up our credentials using the function `set_rdhs_config()`. You can also specify a directory for datasets and API calls to be cached to using `cache_path`. You can also change where your
configuration is saved with `config_path` and `global`, and also change how API requests are returned with `data_frame`. See the Introduction Vignette for more clarity about these options.

```
## set up your credentials
set_rdhs_config(email = "rdhs.tester@gmail.com",
                project = "Testing Malaria Investigations",
                cache_path = "project_one",
                config_path = "~/.rdhs.json",
                data_frame = "data.table::as.data.table",
                global = TRUE)
```

```
## Writing your configuration to:
##    -> ~/.rdhs.json
```

To see what config that is being used by `rdhs` at any point, then use `get_rdhs_config()` to view the config settings.

If we have a look back at our datasets object, we'll see there are 19 datasets listed. However, not all of them will be relevant to our malaria RDT questions. `rdhs` provides tools for helping search quickly within downloaded data sets, so it is often best to download all of them to begin with:

```
# download datasets
downloads <- get_datasets(datasets$FileName)
```

4. Load datasets and associated metadata into R.

We can now examine what it is we have actually downloaded, by reading in one of these datasets:

```
# read in our dataset
cdpr <- readRDS(downloads$CDPR61FL)
```

The dataset returned here contains all the survey questions within the dataset.
The dataset is by default stored as a *labelled* class from the haven package. This class preserves the original semantics and can easily be coerced to factors with `haven::as_factor()`. Special missing values are also preserved. For more info on the *labelled* class have a look at their github.

So if we have a look at what is returned for the variable *hv024*:

```
head(cdpr$hv024)
```

```
## [1] 4 4 4 4 4 4
```

```
# and then the dataset
class(cdpr$hv024)
```

```
## [1] "haven_labelled"
```

If we want to get the data dictionary for this dataset, we can use the function `get_variable_labels`, which will return what question each of the variables in our dataset refer to:

```
# let's look at the variable_names
head(get_variable_labels(cdpr))
```

```
##   variable                                                    description
## 1     hhid                                              Case Identification
## 2    hvidx                                                      Line number
## 3    hv000                                            Country code and phase
## 4    hv001                                                    Cluster number
## 5    hv002                                                   Household number
## 6    hv003 Respondent's line number (answering Household questionnaire)
```

The default behaviour for the function `get_datasets` was to download the datasets, read them in, and save the resultant `data.frame` as a `.rds` object within the cache directory. You can control this behaviour using the `download_option` argument as such:

- `get_datasets(download_option = "zip")` - Just the downloaded zip will be saved
- `get_datasets(download_option = "rds")` - Just the read in rds will be saved
- `get_datasets(download_option = "both")` - The zip is downloaded and saved as well as the read in rds

The other main reason for reading the dataset in straight away as the default option is that `rdhs` will also create a table of all the survey variables and their labels (definitions) and cache them for you, which then allows us to quickly query for particular search terms or survey variables:

```
# rapid diagnostic test search
questions <- search_variable_labels(datasets$FileName, search_terms = "malaria rapid test")
```

We can also query our datasets for the survey question variables. (In the example above the survey variable label was *Result of malaria rapid test* and the variable was *hml35*), using `search_variables()`:

```
# and grab the questions from this now utilising the survey variables
questions <- search_variables(datasets$FileName, variables = c("hv024","hml35"))
head(questions)
```

```
##   variable                  description dataset_filename
## 1    hv024                     Province         CDPR61FL
## 2    hml35 Result of malaria rapid test         CDPR61FL
## 3    hv024                       Region         TZPR7AFL
## 4    hml35 Result of malaria rapid test         TZPR7AFL
##                                                                                  dataset_path
## 1 /home/oj/GoogleDrive/AcademicWork/Imperial/git/rdhs/paper/project_one/datasets/CDPR61FL.rds
## 2 /home/oj/GoogleDrive/AcademicWork/Imperial/git/rdhs/paper/project_one/datasets/CDPR61FL.rds
## 3 /home/oj/GoogleDrive/AcademicWork/Imperial/git/rdhs/paper/project_one/datasets/TZPR7AFL.rds
## 4 /home/oj/GoogleDrive/AcademicWork/Imperial/git/rdhs/paper/project_one/datasets/TZPR7AFL.rds
##   survey_id
## 1 CD2013DHS
## 2 CD2013DHS
## 3 TZ2015DHS
## 4 TZ2015DHS
```

5. Extract variables and combining datasets for pooled multi-survey analyses.

To extract our data we pass our questions object to the function `extract_dhs`, which will create a list with each dataset and its extracted data as a `data.frame`. We also have the option to add any geographic data available, which will download the geographic data files for you and add this data to you resultant extract:

```
# let's just use the PR files thus
datasets <- dhs_datasets(surveyIds = survs$SurveyId, fileFormat = "FL", fileType = "PR")
downloads <- get_datasets(datasets$FileName)
```

```
# and grab the questions from this again along with also questions detailing the province
questions <- search_variables(datasets$FileName, variables = c("hv024","hml35"))

# and now extract the data
extract <- extract_dhs(questions, add_geo = FALSE)

# what does our extract look like
str(extract)
```

```
## List of 2
##  $ CDPR61FL:Classes 'dhs_dataset' and 'data.frame':  95949 obs. of  3 variables:
##   ..$ hv024   : 'haven_labelled' int [1:95949] 4 4 4 4 4 4 4 4 4 4 ...
##   .. ..- attr(*, "label")= chr "Province"
##   .. ..- attr(*, "labels")= Named int [1:11] 1 2 3 4 5 6 7 8 9 10 ...
##   .. .. ..- attr(*, "names")= chr [1:11] "kinshasa" "bandundu" "bas-congo" "equateur" ...
##   ..$ hml35   : 'haven_labelled' int [1:95949] NA NA NA NA NA NA NA 1 0 NA ...
##   .. ..- attr(*, "label")= chr "Result of malaria rapid test"
##   .. ..- attr(*, "labels")= Named int [1:3] 0 1 9
##   .. .. ..- attr(*, "names")= chr [1:3] "negative" "positive" "missing"
##   ..$ SurveyId: chr [1:95949] "CD2013DHS" "CD2013DHS" "CD2013DHS" "CD2013DHS" ...
##  $ TZPR7AFL:Classes 'dhs_dataset' and 'data.frame':  64880 obs. of  3 variables:
##   ..$ hv024   : 'haven_labelled' int [1:64880] 1 1 1 1 1 1 1 1 1 1 ...
##   .. ..- attr(*, "label")= chr "Region"
##   .. ..- attr(*, "labels")= Named int [1:30] 1 2 3 4 5 6 7 8 9 10 ...
##   .. .. ..- attr(*, "names")= chr [1:30] "dodoma" "arusha" "kilimanjaro" "tanga" ...
##   ..$ hml35   : 'haven_labelled' int [1:64880] NA NA NA NA NA NA NA 0 NA NA ...
##   .. ..- attr(*, "label")= chr "Result of malaria rapid test"
##   .. ..- attr(*, "labels")= Named int [1:3] 0 1 9
##   .. .. ..- attr(*, "names")= chr [1:3] "negative" "positive" "missing"
##   ..$ SurveyId: chr [1:64880] "TZ2015DHS" "TZ2015DHS" "TZ2015DHS" "TZ2015DHS" ...
```

The resultant extract is a list, with a new element for each different dataset that you have extracted. The responses from the dataset are by default stored as a *labelled* class from the haven package.

We can now combine our two data frames for further analysis using the `rdhs` package function `rbind_labelled()`. This function works specifically with our lists of labelled data.frames:

```
# first let's bind our first extraction, without the hv024
extract_bound <- rbind_labelled(extract)
```

```
## Warning in rbind_labelled(extract): Some variables have non-matching value labels: hv024.
## Inheriting labels from first data frame with labels.
```

```
head(extract_bound)
```

```
##            hv024 hml35  SurveyId  DATASET
## CDPR61FL.1     4    NA CD2013DHS CDPR61FL
## CDPR61FL.2     4    NA CD2013DHS CDPR61FL
## CDPR61FL.3     4    NA CD2013DHS CDPR61FL
## CDPR61FL.4     4    NA CD2013DHS CDPR61FL
## CDPR61FL.5     4    NA CD2013DHS CDPR61FL
## CDPR61FL.6     4    NA CD2013DHS CDPR61FL
```

This hasn't quite done what we might want though. The *hv024* variable stores the regions for these 2 countries, which will not be the same and thus the labels will be different between the two of them. Without specifying any additional arguments `rbind_labelled()` will simply use the first data.frames labelling as the default, which will mean that some of the Tanzanian provinces will have been encoded as DRC provinces -

not good!

There are a few work arounds. Firstly, we can specify a *labels* argument to the function which will detail how we should handle different variables. *labels* is a names list that specifies how to handle each variable. If we simply want to keep all the labels then we use the string "concatenate":

```
# lets try concatenating the hv024
better_bound <- rbind_labelled(extract, labels = list("hv024"="concatenate"))

head(attr(better_bound$hv024,"labels"))
```

```
##        arusha      bandundu     bas-congo dar es salaam        dodoma
##             1             2             3             4             5
##      equateur
##             6
```

We could also specify new labels for a variable. For example, imagine the two datasets encoded their RDT responses differently, with the first one as `c("No","Yes")` and the other as `c("Negative","Positive")`. These would be for our purposes the same response, and so we could either leave it and all our results would use the `c("No","Yes")` labelling. But we may want to use the latter as it's more informative/correct, or we may want to be crystal clear and use `c("NegativeTest","PositiveTest")`. we can do that like this:

```
# lets try concatenating the hv024 and providing new labels
better_bound <- rbind_labelled(
  extract,
  labels = list("hv024"="concatenate",
                "hml35"=c("NegativeTest"=0, "PositiveTest"=1))
)

# and our new label
head(attr(better_bound$hml35,"labels"))
```

```
## NegativeTest PositiveTest
##            0            1
```

The other option is to not use the labelled class at all. We can control this when we download our datasets, using the argument `reformat=TRUE`, which will ensure that no factors or labels are used and it is just the raw data.

```
# download the datasets with the reformat arguments
downloads <- get_datasets(datasets$FileName, reformat=TRUE)

# grab the questions but specifying the reformat argument
questions <- search_variables(datasets$FileName, variables = c("hv024", "hml35"),
                                      reformat=TRUE)

# and now extract the data
extract <- extract_dhs(questions, add_geo = FALSE)
```

We now have managed to go from our opening question to a finalised data set that we can use going forwards for any downstream analysis (and hopefully it didn't take that long to do it!)

Ramblings from a first time package development process

Clichéd but the process of actually writing a package, and all that entailed, was a real highlight. In particularly, actually having the opportunity to work on a code base with someone else in a collaborative way. I work in a

large collaborative group, however, this has not translated as much to working on the same set of code with someone. As a result I've never had to properly learn how to use git outside of `commit` and `push`, nor had I made use of much of the useful aspects of GitHub. So learning how to correctly use branches in git and realising that helpful comments are actually helpful (eventually) was really great. With this in mind I wanted to thank Jeff Eaton again for taking on this project. He definitely helped drive it over the finish line, and it was nice to have a glimpse at what working as a developer would look like if I decide to leave pure academia.

There were also a few things that before I started writing `rdhs` I knew I would have to figure out but I didn't have a clue where to start, and for which repeated googling didn't eventually help with. Fortunately, I work in the same department as Rich FitzJohn, so it was great having someone to point me in the right direction. The following, are 3 of the things that I genuinely had no idea how to do before, so I thought I'd share them here (and so I can remind myself in the future):

1. Logging into a website from R

The DHS website has a download manager that you can use to select surveys you want to download, and it will auto generate a list of URLs in a text file. When I saw this, I thought this would be great for creating a database of what data sets and the URLs a user's login details can give them, which can then be cached so that `rdhs` knows whether you can download a data set or not. The only problem is, that to download those data sets you need to be logged in, and you also need to be logged in to get to the download manager. For me, I didn't know how to translate being "logged in" into R code, or even what that looked like. But turns out it wasn't too bad after being shown where to start looking by Rich.

To know where to look I opened up Chrome and went to developer tools. From there I opened up the **Network Tab**, which then records the information being sent to the webpage. So to know what information is required to login I simply logged in as normal, and then inspected what appeared in the network tab's **Headers Tab**. This then showed me what the needed **Request URL** was, and what information was being submitted in the **Form Data** at the bottom of this tab.

I could then use this information to *log in* from with R using an `httr::POST` request:

```
# authentication page
terms <- "https://dhsprogram.com/data/dataset_admin/login_main.cfm"

# create a temporary file
tf <- tempfile(fileext = ".txt")

# set the username and password
values <- list(
  UserName = your_email,
  UserPass = your_password,
  Submitted = 1,
  UserType = 2
)

# log in.
message("Logging into DHS website...")
z <- httr::POST(terms, body = values) %>% handle_api_response(to_json = FALSE)
```

To me, this seemed really cool, and then meant I could do the same style of steps to get to the Download Manager webpage and then tick all the check boxes in the page to generate the URL with all the download links in.

2. Caching API results and responding to changes in the API

We wanted to be able to cache a user's API request for them locally when designing `rdhs`. We felt this was important as it would reduce the burden on the API itself, as well as enable researchers who were without internet (e.g. currently working in the field), the ability to still access previous API requests. However, designing something neat that would be easy to respond to changes in the API version would I thought would be outside my skill set.

Again, enter Rich and this time with his package storr. This was a lifesaver, and created an easy infrastructure for storing API responses in a key-value store. I could then use the specific API URL as the key and the response as the value. Initially I thought I would have to keep saving the response with explicit names (e.g. the URL), but `storr` handles all this for you, and also then helps get around having too long file names if your API request is very long for example.

To respond to changes in the API, my solution was perhaps not the neatest, but I simply kept a record of the date you last made an API request and compared it to the API's data updates endpoint. If I could see any recent changes, I then could clear all the API requests cached. This would made a lot simpler using the `namespaces` options in `storr`, which meant that I was able to keep all API cached data in one place, which could then be easily deleted on mass.

3. Tests requiring authentication and Travis environment variables

The last thing caused me the most amount of headaches. How do I write tests that require authentication and can use `travis` for continuous integration. Initially, I made a dummy account with the DHS website for this, but realised that sharing the credentials of an account with access to just dummy data sets would not enable me to test the weird edge cases that started popping up related to certain data sets. The first solution that I used for a few months was to set up environment variables within `travis` itself, which could then be used to create a valid set of credentials.

This worked, however, it meant that I would have to write a lot of the `rdhs` functionality to use environment variables that were the user's email and password, which felt wrong and quite clunky. All I wanted was to pass to Travis was a valid set of login credentials that would then be used within the tests, much in the same way that a user would. To do this I had to learn a bit more about what the `.travis.yml` document could actually be used for, because to begin with I had only been using it to specify the software language.

Again, Rich pointed me to using `sodium` to create an encrypted version of a valid login credentials:

```
# read in a key from a local file
key <- sodium::hash(charToRaw(readLines("scripts/key.txt")))

# create a tat with all the necessary login credentials
zip("rdhs.json.tar",files=c("rdhs.json", "tests/testthat/rdhs.json"))

# read this tar in as binary data
dat <- readBin("rdhs.json.tar",raw(),file.size("rdhs.json.tar"))

# encrypt the data using sodium and our key before saving it
enc <- sodium::data_encrypt(msg = dat,key = key)
saveRDS(enc,"rdhs.json.tar.enc")
```

This could then be included in the GitHub repository, and I could then set up the key as a Travis environment variable to decrypt it. This decryption step could then be written within my `.travis.yml` file, and would mean that all my tests had access to my login credentials in a secure way.

Acknowledgements and Final Thoughts

Firstly, I want to thank Anna Krystalli for handling the review, and for being incredibly patient throughout, especially at the end as we were fixing the last authentication bug. Also many thanks to Lucy McGowan

and Duncan Gillespie for taking the time to review the package and for their input, which led to lots of improvements (and also linking the `add_line` function from `httr` was seriously helpful, and I've used that function in lots of other my other work now). I also wanted to more broadly thank the review process as a whole. Having the option to discuss the package and needed solutions with the reviewers within a GitHub issues system is fantastic. It made the process personal and was substantially improved over review processes I have had at academic journals. Lastly, another big thank you Jeff Eaton and Rich FitzJohn, and also to the infectious disease epidemiology department at Imperial for providing a lot of really helpful guinea pig testing of the numerous iterations of `rdhs`.

Options to Contribute

There are a few things that would be great to add in the future.

The first would be to add a suite of tools for doing spatial mapping. A lot of the time, people want to know what the prevalence of x is either at a fine spatial scale, or grouped at administrative/county/state levels. `rdhs` helps provide the tools to get geolocated measures of x, and I think it would be a great next step to add a suite of mapping tools. It would be great if they could be used to either create a mesh through these points (probably using `INLA`), or calculate survey weighted means at requested spatial scales or match them to a provided `SpatialPolygons` object.

The second is less related to specific issues, but continues how best to integrate downstream analysis pipelines. One example is a package in development by Jeff Eaton called demogsurv, which is used to calculate common demographic indicators from household survey data, including child mortality, adult mortality, and fertility. This is just one example, but over time there will be a number of bespoke analysis tools down the line, and so it would be nice to begin a collection/grouping of these tools.

Lastly, it would be nice to have a way to manually add sources of survey data. At the moment the pipeline for downloading raw data sets used the DHS API a lot, however, what if you had some survey data (either locally or shared at a URL) that you wanted to bring into your analysis pipeline. Something similar to this is done for the `model_datasets` within `rdhs`, which is a set of dummy data sets that the DHS hosts online but are not included in their API.