R + REDCap Example Data Cleaning Workflow

Jennifer Thompson, MPH

June 4, 2018

Chapter 1

Introduction

Summary and Goals

This document demonstrates the ongoing process of data cleaning used by the Vanderbilt CIBS Center. Most of our data is stored in REDCap databases and is cleaned at multiple points throughout data collection, with the goal of the highest quality data possible in the least amount of time once enrollment is complete. This example will demonstrate the R code we use to accomplish this goal, and briefly describe the rest of our process.

Notes

- All code assumes that the user has rights to use the REDCap API for data export, and that a working API token is stored in the .Renviron file in the working directory, in the format

  `RCTOKEN=manylettersandnumbers`

  For more information on the REDCap API, please see `Project Setup -> Other Functionality` within an existing REDCap project. For general information on working with the API, the Github wiki of the redcapAPI package has a good overview. (This example includes basic API usage and will not use the package, but if you are interested in using more of the API's functionality, it would be a great one to investigate.)
- The code will use several helper functions, sourced from `dataclean_helpers.R` in the same working directory/Github repository. The code will also be copied in an appendix to this document.

Motivating Example

This example uses a sample REDCap database for a three-month longitudinal study of adult patients taking a dietary supplement and measuring creatinine, HDL and LDL cholesterol, and weight over time. (Sample database is adapted with thanks from REDCap's project templates.) The study codebook is available here.

Contents

Chapter 2

Process Overview

Chapter 3

Step 1: Use REDCap API to Export Raw Data

We use REDCap's API capabilities to export the data automatically every time the script is run, reducing the potential for error and saving time compared to manually exporting every time data is cleaned.

Requirements:

1. Working API token

You must have appropriate user rights for your database in order to request an API token. Once you have the correct user rights, log into the REDCap project. On the lefthand side under `Applications`, you will see a line for `API and API Playground`. Click here, then on the button titled `Generate API token`.

Once your token is generated, **never share it with anyone**. It gives you permission and ability to access research data, and should be kept protected at all times. If you share code with other people, one way to do this safely is to store your API token in a hidden `.Renviron` file in the appropriate working directory, like this:

```
RCTOKEN=manylettersandnumbers
```

You can then access the token using the function `Sys.getenv()`.

2. R's `httr` package, built for working with APIs

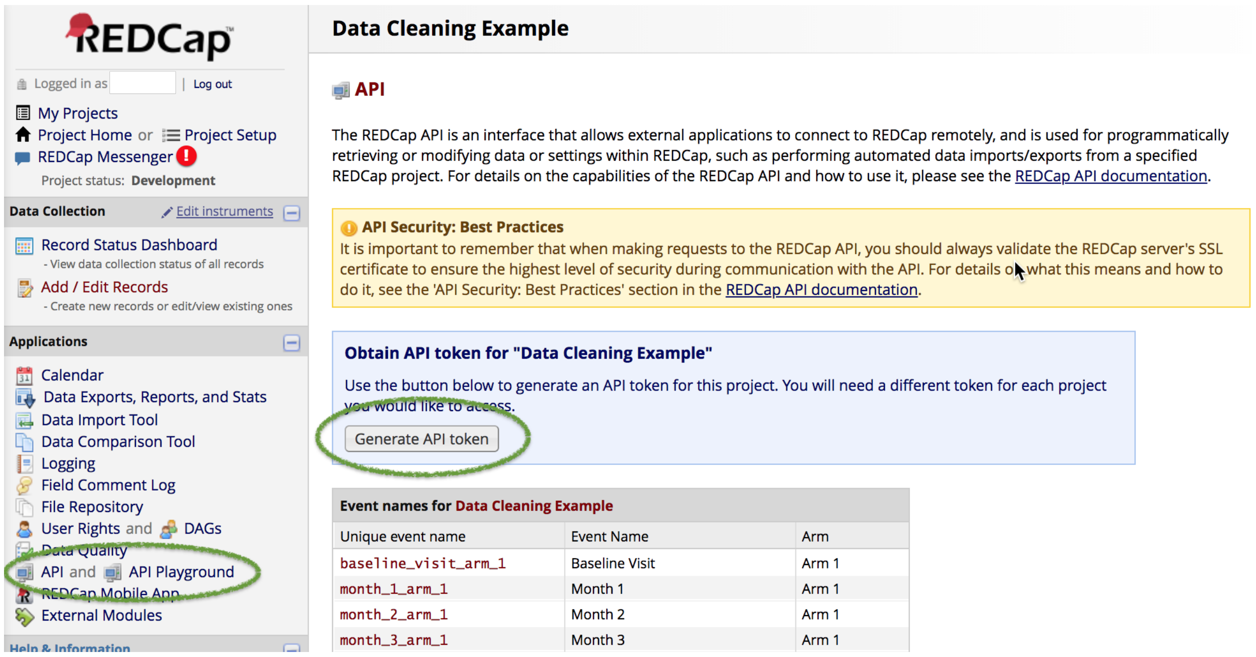
More information on `httr` can be found in the documentation and vignettes, linked from CRAN.

Approaches

There are (at least) two approaches to exporting REDCap data:

1. Read in the entire database in a single `httr::POST` call, then create subsets in R as needed
2. Read in specific subsets in separate calls

Approach #1 is fine if your database is not complex or very large, and/or if you are not yet comfortable working with the API. Approach #2 is valuable in more complicated situations.



Data Cleaning Example

API

The REDCap API is an interface that allows external applications to connect to REDCap remotely, and is used for programmatically retrieving or modifying data or settings within REDCap, such as performing automated data imports/exports from a specified REDCap project. For details on the capabilities of the REDCap API and how to use it, please see the REDCap API documentation.

API Security: Best Practices
It is important to remember that when making requests to the REDCap API, you should always validate the REDCap server's SSL certificate to ensure the highest level of security during communication with the API. For details on what this means and how to do it, see the 'API Security: Best Practices' section in the REDCap API documentation.

Obtain API token for "Data Cleaning Example"
Use the button below to generate an API token for this project. You will need a different token for each project you would like to access.

Event names for Data Cleaning Example

Unique event name	Event Name	Arm
baseline_visit_arm_1	Baseline Visit	Arm 1
month_1_arm_1	Month 1	Arm 1
month_2_arm_1	Month 2	Arm 1
month_3_arm_1	Month 3	Arm 1

Figure 3.1: Getting an API token

For example, here, our database is longitudinal, and different data is collected at different time points (for example, date of birth is only collected at baseline). If we read in the entire database at once, we will have a lot of missing values and extra columns.

I will show an example of Approach 1 for reference, but will primarily use Approach 2. Either way, we want to create the following datasets:

- Baseline data
- Monthly data
- Study completion data

For both approaches, we need this setup:

```r
## Load httr
library(httr)

## Source helper functions (script should be stored in this working directory)
source("dataclean_helpers.R")

## Set URL for REDCap instance (yours may be different)
rc_url <- "https://redcap.vanderbilt.edu/api/"
```

Approach 1: Example

This approach uses the simplest API call, but needs some R work after exporting the data.

```r
## Use API + httr::POST to get all data at once
main_post <- httr::POST(
  url = rc_url,
  body = list(
    token = Sys.getenv("RCTOKEN"),
      ## API token gives you permission to get data
    content = "record",         ## export *records*
    format = "csv",             ## export as *CSV*
    rawOrLabel = "label",       ## export factor *labels* vs numeric codes
    exportCheckboxLabel = TRUE ## export ckbox *labels* vs Unchecked/Checked
  )
)

## main_post has class "response"; read it as a CSV to create a data.frame
main_df <- post_to_df(main_post)

## Create subsets with data collected at various time points
baseline_df <- subset(main_df, redcap_event_name == "Baseline Visit")
monthly_df <- subset(main_df, redcap_event_name %in% paste("Month", 1:3))
completion_df <- subset(main_df, redcap_event_name == "Study Completion")
```

Note that unless we spend the time to manually subset them, each of those three data.frames will have many columns with blank values. For example, `baseline_df` will have a column for `compliance`, even though `compliance` is only collected at monthly visits. This is not a major problem if your project is small, but can be a problem if you have a large project.

Approach 2

This approach uses three separate `httr::POST` calls to create separate datasets that are exactly what we need.

REDCap's API Playground can be useful in figuring out which options to include in the `body` argument of `httr::POST`. (Do note that as of the time of this writing, the example R code from the Playground uses `RCurl`; `httr` is currently more commonly used and thus it is easier to find documentation and assistance for it.)

(The `rawOrLabel = "label"` and `exportCheckboxLabel = TRUE` elements in the `body` argument of `POST` are personal preference. I set these to export labels because I usually find that it is more clear - ie, it is easier to figure out what `sex == Male` is doing than `sex == 1`. However, depending on your database and your purposes, you may want to change these to use the raw numeric codes - for example, if you have fields with very long labels.)

Differences from Approach 1 in the `body` of `httr::POST`:

1. We specify **forms**, using their raw names (eg, `baseline_data` instead of `Baseline Data`).
2. We specify **events**, again using their raw names (eg, `baseline_visit_arm_1` instead of `Baseline Visit`).
3. We always specify `study_id` as a **field** in addition to the other forms. REDCap does not always export the ID by default.

If you are exporting >1 form or event, separate them with commas. You can find raw event names by going to `Project Setup -> Define My Events` within your REDCap project, and raw form names by looking at the data dictionary. (They can also be exported as project metadata using the API; there is an example in `dataclean_helpers.R`.)

Note: The function `post_to_df()`, which creates a data.frame from the result of `httr::POST`, is created in `dataclean_helpers.R`.

Baseline and Demographic Data

This code chunk exports all the fields collected at the baseline visit (the Demographic and Baseline Visit forms), as well as study ID. It only exports the Baseline Visit event, because all other events would have `NA` values for these fields. Therefore, each study ID will have at most one record in this dataset.

```r
## Data from baseline visit only: Demographics and Baseline Data forms
baseline_post <- httr::POST(
  url = rc_url,
  body = list(
    token = Sys.getenv("RCTOKEN"), ## API token gives you permission
    content = "record",            ## export *records*
    format = "csv",                ## export as *CSV*
    forms = "demographics,baseline_data", ## forms
    fields = c("study_id"),               ## additional fields
    events = "baseline_visit_arm_1",      ## baseline visit event only
    rawOrLabel = "label",      ## export factor *labels* vs numeric codes
    exportCheckboxLabel = TRUE ## export ckbox *labels* vs Unchecked/Checked
  )
)

## baseline_post has class "response"; read it as a CSV to create a data.frame
baseline_df <- post_to_df(baseline_post)

## Double-check if you like! Commented out to save space
## baseline_df
```

Beautiful! Keep going for the monthly and study completion forms.

Monthly Visit Data

This code chunk exports all the fields collected at each monthly visit (the Monthly Visit form), as well as study ID. It exports all three monthly visit events; all other events will have `NA` values for these fields. Each study ID will have up to three records in this dataset.

```r
## Data from all monthly visits
monthly_post <- httr::POST(
  url = rc_url,
  body = list(
    token = Sys.getenv("RCTOKEN"),           ## API token gives you permission
    content = "record",                      ## export *records*
    format = "csv",                          ## export as *CSV*
    forms = "monthly_data",                  ## forms
    fields = c("study_id"),                  ## additional fields
    events = paste(sprintf("month_%s_arm_1", 1:3), collapse = ","),
      ## all 3 monthly visit events
    rawOrLabel = "label",      ## export factor *labels* vs numeric codes
    exportCheckboxLabel = TRUE ## export ckbox *labels* vs Unchecked/Checked
  )
)
monthly_df <- post_to_df(monthly_post)

## Double-check if you like! Commented out to save space
## monthly_df
```

Study Completion Data

This code chunk exports all the fields collected at study completion, as well as study ID. It exports only the study completion event; therefore, each study ID will have at most one record.

```r
## Data from study completion visits
completion_post <- httr::POST(
  url = rc_url,
  body = list(
    token = Sys.getenv("RCTOKEN"),     ## API token gives you permission
    content = "record",                ## export *records*
    format = "csv",                    ## export as *CSV*
    forms = "completion_data",         ## form
    fields = c("study_id"),            ## additional fields
    events = "study_completion_arm_1", ## study completion event
    rawOrLabel = "label",      ## export factor *labels* vs numeric codes
    exportCheckboxLabel = TRUE ## export ckbox *labels* vs Unchecked/Checked
  )
)
completion_df <- post_to_df(completion_post)

## Double-check if you like! Commented out to save space
## completion_df
```

Chapter 4

Step 2: Download Resolved Queries

Sometimes, issues with the data are known but cannot be resolved. For example, if a patient was not weighed at a monthly visit, that value can never be entered. Having it be brought up as a problem repeatedly forces study staff to re-investigate problems they have already investigated, which is both irritating and a waste of time and effort.

Therefore, an important step in our process is the **documentation** of each issue. In addition to serving as a record of why the original data may have changed, this marks issues which cannot be resolved (or are confirmed correct, in the case of an extreme value) as unfixable or correct, and they are then able to be removed from future data cleans. We document these issues in a separate REDCap project, which contains at minimum the following fields:

- Created by the data clean script:
  - Unique identifier for every "query" (issue with the data)
    Typically a combination of patient ID, the date of the data clean, and a number between 1 and the total number of queries found during that data clean.
  - Patient identifier (study ID)
  - Date of data clean
  - Form in the data entry database where the problem is located *(eg, Monthly Visit)*
  - Event in the data entry database where the problem is located *(eg, Monthly Visit 2)*
  - Text describing the problem *(eg, "Sex is Male, but patient is marked as pregnant")*
- Fields filled out by study staff:
  - Was the issue corrected? Options:
    * No
    * Yes
    * Value confirmed correct *(for accuracy queries only - eg, "weight really was xxxx kg")*
  - If the issue was not corrected, why?
  - Electronic signature for the person who corrected the issue
- Fields filled out by the coordinating center/staff member in charge:
  - Date query was reviewed by coordinating center
  - Was the issue closed? Options:
    * Yes, it is permanently unfixable *(eg, patient was never weighed)*
    * Yes, the query was the result of a programming error or miscommunication
    * No (project manager should contact site to reconcile)
  - Electronic signature field for coordinating center member who reviewed query

Depending on the study, it might also be helpful to have additional fields, such as whether a Note to File was recorded.

The example documentation codebook for this project can be found here.

We download the data in our documentation database in the same way as the raw data. However, note that since it is a separate REDCap project, you will need a separate API token. Mine is saved in my `.Renviron` file as the object `DOCTOKEN`.

```r
## Documentation of queries already checked
doc_post <- httr::POST(
  url = rc_url,
  body = list(
    token = Sys.getenv("DOCTOKEN"),
    content = "record",
    format = "csv",
    rawOrLabel = "label",
    exportCheckboxLabel = TRUE
  )
)
## This won't work till I enter some data!
# doc_df <- post_to_df(doc_post)

## Double-check if you like! Commented out to save space
## doc_df
```

Chapter 5

Step 3: Create a data.frame of All Issues

Chapter 6

Step 4: Remove Unfixable Queries

Chapter 7

Step 5: Disseminate Issues to Study Staff

Chapter 8

Step 6: Iterate!

I mean iteration in two ways.

Repeat This Process, Early and Often

The more frequently you clean your data, the more prepared you will be for things like interim analyses and DSMB or progress reports, and the less time you'll have to spend at the end of the study (when everyone is very excited about getting the final results!). This is especially important for multicenter studies or studies that enroll over years, where sites or staff members may join and leave the group; once a site is closed or a coordinator has retired, it is (understandably!) a challenge to get effort from that site to clean data. Besides, no one wants to get a large, overwhelming number of queries at the end of the study!

How frequently you choose to clean your data will depend on enrollment rates and how many staff members are available to do the cleaning, but we recommend repeating this process as often as is reasonable.

Always Be Improving

Much like your study protocols, your cleaning script will rarely be perfect on the first try. As the study goes on, you will always find more ways that data can be "wrong," or have more questions that are inspired by unexpected data or discussions with study staff. The vast majority of time spent on this data cleaning script is during its initial development, but there will always be things that need to be changed or added.

This is one reason that **communication** between the statistician/database manager and study staff is hugely important! We work together not only at the beginning of this process, to design the database and come up with lists of data points that need to be checked, but throughout study enrollment and data collection to make sure that protocol changes are adequately accounted for, misunderstandings are cleared up quickly, etc. Typically, as study staff are working through one round of data cleaning, I keep a list of things that need to be investigated - queries they believe shouldn't be there or aren't clear, queries that need to be added due to a protocol change, etc. Then when it's time for the next round of data cleaning, I block off some time to investigate anything that has come up, fix or add what needs attention, and *then* rerun the next round.

In addition to improving the data itself and the data cleaning script, we use this process as a way to improve our study documents and staff education: If there is a piece of data that is systematically showing up as an issue, perhaps it is due to something that was not clearly addressed at the study startup visit and needs to be revisited, or should be written out fully in an SOP.

Chapter 9

Appendix: `dataclean_helpers.R`

This file contains several "helper functions" which make the processes of downloading data and checking problems of similar types easier to do and debug. When using the workflow described here, this file should be stored in the same working directory and sourced within the script, as seen above. It is copied here for convenience.

```r
################################################################################
## Helper functions for dataclean.R
################################################################################

## -- Helper function to create data.frames from `response` objects created ----
## -- by httr::POST -------------------------------------------------------------
post_to_df <- function(post_obj){
  ## Use read.csv to create a data.frame from the response object
  tmp <- read.csv(
    text = as.character(post_obj),
    stringsAsFactors = FALSE,
    na.strings = ""
  )

  ## REDCap exports many underscores in checkbox variable names; cut down to 1
  names(tmp) <- gsub("_+", "_", names(tmp))

  return(tmp)
}

## -- Read in data dictionary --------------------------------------------------
## -- We will use this for variable labels, limits -----------------------------
library(httr)

ddict_post <- httr::POST(
  url = "https://redcap.vanderbilt.edu/api/",
  body = list(
    token = Sys.getenv("RCTOKEN"), ## API token gives you permission to get data
    content = "metadata",          ## export *metadata* (data dictionary)
    format = "csv"                 ## export as *CSV*
  )
)
```

```r
datadict <- read.csv(
  text = as.character(ddict_post),
  na.strings = "",
  stringsAsFactors = FALSE
)

## -- Return a field label given the field name --------------------------------
## Data entry staff may not know what "sga_b" is, but they know what
## "Subject Global Assessment" is.
get_label <- function(
  variable,         ## character; must be in one row of ddict$field_name
  ddict = datadict, ## data.frame
  ## Default column names are based on REDCap data dictionary exports;
  ## you can supply your own data.frame with your desired column names
  cname_vname = "field_name", ## colname that contains variable ("sga_b")
  cname_label = "field_label" ## colname that contains label ("Subj Global Asmt")
){

  ## Checks: ddict must be a data.frame with fields field_name and field_label,
  ## at minimum; field_name must include specified variable
  if(!inherits(ddict, "data.frame")){
    stop("'ddict' must be a data.frame", call. = FALSE)
  }
  if(!all(c(cname_vname, cname_label) %in% names(ddict))){
    stop(
      "Columns of 'ddict' must include `cname_vname`, `cname_label`",
      call. = FALSE
    )
  }
  if(sum(ddict[, cname_vname] == variable, na.rm = TRUE) != 1){
    stop(
      "'variable' must be represented in exactly one row in 'ddict'",
      call. = FALSE
    )
  }

  ## With all those checks out of the way, getting the label is simple:
  return(
    as.character(ddict[ddict[, cname_vname] == variable, cname_label])
  )
}

## -- Data checking functions --------------------------------------------------
## Each function takes as arguments a data.frame and a set of variable names,
## and returns a data.frame with two columns: ID (study ID + REDCap event) and
## msg (error message)

## -- Helper function for all check_xxxxx() functions: -------------------------
## Takes as arguments:
## - Matrix of T/F or 1/0 (problem/not), one column per issue
## - data.frame of issue names (column 1) and error messages (column 2)
## Returns data.frame with one row per error and two columns:
## - id: "record ID;REDCap event name"
## - msg: error message (eg, "Missing age", "Height lower than suggested limit")
## If no errors, returns a data.frame w/ 0 rows
create_error_df <- function(
  error_matrix,
  error_codes
){
  ## All values in error_matrix should be logical or 1/0
  if(!(is.logical(error_matrix) | all(error_matrix %in% 0:1))){
    stop("`error_matrix` should be logical or all 0/1", call. = FALSE)
  }

  ## error_codes should be a matrix or data.frame with two columns; all columns
  ## in error_matrix should be represented in error_codes[, 1]
  if(ncol(error_codes) != 2){
    stop(
      "`error_codes` should have two columns: `variables` and `msgs`",
      call. = FALSE
    )
  }
  if(!all(colnames(error_matrix) %in% error_codes[, 1])){
    stop(
      "All columns in `error_matrix` must be represented by a row in `error_codes`",
      call. = FALSE
    )
  }

  error_data <-
    do.call(
      rbind,
      lapply(
        1:ncol(error_matrix),
        FUN = function(i){
          data.frame(
            id = rownames(error_matrix)[which(error_matrix[, i])],
            msg = rep(
              error_codes[match(colnames(error_matrix)[i], error_codes[, 1]), 2],
              sum(error_matrix[, i])
            )
          )
        }
      )
    )

  return(error_data)
}

## -- Check for basic missingness -----------------------------------------------
## (when a variable should be present for all records in df)
check_missing <- function(df, variables, ddict = datadict){
  if(nrow(df) > 0){
    ## Create error messages ("Missing" + database label)
    missing_msgs <- unlist(
      lapply(
        variables,
        FUN = function(x){
          paste("Missing", get_label(x, ddict))
        }
      )
    )

    ## Create data frame of column names, error messages
    error_codes <- as.data.frame(cbind(variables, missing_msgs))

    ## Matrix for whether each variable is missing:
    ## column names = variables, row names = ID + REDCap event
    missing_matrix <- do.call(
      cbind,
      lapply(variables, FUN = function(x){ is.na(df[,x]) })
    )
    colnames(missing_matrix) <- variables
    rownames(missing_matrix) <-
      paste0(df[,"study_id"], ';', df[,"redcap_event_name"])

    ## Create final data set: One row per column per missing value, with error
    ## message that matches that column name
    ## id = ID + REDCap event; msg = error message ("Missing ...")
    missing_data <- create_error_df(
      error_matrix = missing_matrix, error_codes = error_codes
    )
  } else{
    missing_data <- NULL
  }

  return(missing_data)
}

## -- Check whether numeric variables fall within specified limits -------------
## If using a REDCap data dictionary, these can be the text_validation_min/max
## fields; you can also specify your own limits that were not built into the
## REDCap database design.
## df_limits is a data.frame that has, at minimum, one column for the variable
## name (column 1) and one column each for minimum and maximum values.
## Everything in `variables` should be represented by a row in df_limits.

check_limits_numeric <- function(
  df,                                ## data.frame to check
  variables,                         ## character vector of variables to check
  ddict = datadict,                  ## data.frame containing variable labels
  ## These defaults assume you are using a REDCap data dictionary, but you can
  ## pass your own dataset too.
  df_limits = datadict,              ## data.frame containing limits
  cname_min = "text_validation_min", ## Column name for minimum limit
  cname_max = "text_validation_max"  ## Column name for maximum limit
){
  ## Checks
  if(!inherits(df, "data.frame")){
```

```
  stop("`df` must be a data.frame", call. = FALSE)
}
if(!all(variables %in% names(df))){
  stop("All elements of `variables` must be columns in `df`", call. = FALSE)
}
if(!inherits(df_limits, "data.frame") |
   !all(c(cname_min, cname_max) %in% names(df_limits))){
  stop("`df_limits` must be a data.frame with columns `cname_min`, `cname_max`",
       call. = FALSE)
}
if(!all(variables %in% df_limits[, 1])){
  stop(
    "All elements of `variables` must be represented by a row in `df_limits`",
    call. = FALSE
  )
}
## Warning if anything listed in `variables` is not a numeric field
not_numeric <- setdiff(
  variables,
  subset(datadict, text_validation_type_or_show_slider_number == "number")$field_name
)
variables <- setdiff(variables, not_numeric)

if(length(not_numeric) > 0){
  warning(
    sprintf(
      "The following `variables` are not numeric: %s",
      paste(not_numeric, collapse = "; ")
    ),
    call. = FALSE
  )
}

if(nrow(df) > 0){
  ## Create components for error messages in a data.frame:
  error_codes <- data.frame(
    variables = variables,
    var_label = unlist(lapply(variables, FUN = get_label)),
    min_value = unlist(lapply(
      variables, FUN = function(x){
        as.numeric(df_limits[df_limits[, 1] == x, cname_min])
      }
    )),
    max_value = unlist(lapply(
      variables, FUN = function(x){
        as.numeric(df_limits[df_limits[, 1] == x, cname_max])
      }
    ))
  )

  ## If both min and max are missing, there are no limits to check
  no_limits <-
    subset(error_codes, is.na(min_value) & is.na(max_value))$variables
```

```r
  variables <- setdiff(variables, no_limits)

  if(length(no_limits) > 0){
    warning(
      sprintf(
        "The following `variables` have no limits in `df_limits`: %s",
        paste(no_limits, collapse = "; ")
      ),
      call. = FALSE
    )
  }

  error_codes <- subset(error_codes, !(variables %in% no_limits))

  ## Combine pieces to create error messages
  error_codes$msgs <- with(error_codes, {
    ifelse(
      !is.na(min_value) & !is.na(max_value),
      sprintf(
        "%s is not between recommended limits of %s and %s; please correct or confirm accuracy",
        var_label, min_value, max_value
      ),
    ifelse(
      !is.na(min_value),
      sprintf(
        "%s is lower than recommended limit of %s; please correct or confirm accuracy",
        var_label, min_value
      ),
    ifelse(
      !is.na(max_value),
      sprintf(
        "%s is higher than recommended limit of %s; please correct or confirm accuracy",
        var_label, max_value
      )
    )))
  })

  ## Replace NA min/max with -/+Inf
  error_codes$min_value <- with(error_codes, {
    ifelse(is.na(min_value), -Inf, min_value)
  })
  error_codes$max_value <- with(error_codes, {
    ifelse(is.na(max_value), Inf, max_value)
  })

  ## Matrix for whether each variable is outside limits:
  ## column names = variables, row names = ID + REDCap event

  ## Function to check values for one variable
  outside_limits <- function(df, variable, min, max){
    ifelse(is.na(df[, variable]), FALSE,
           df[, variable] < min | df[, variable] > max)
  }
```

```
    ## Apply that function over all variables
    limits_matrix <- mapply(
      FUN = outside_limits,
      variable = variables,
      min = error_codes$min_value,
      max = error_codes$max_value,
      MoreArgs = list(df = df)
    )
    colnames(limits_matrix) <- variables
    rownames(limits_matrix) <-
      paste0(df[,"study_id"], ';', df[,"redcap_event_name"])

    ## Create final data set: One row per column per missing value, with error
    ## message that matches that column name
    ## id = ID + REDCap event; msg = error message
    limits_data <- create_error_df(
      error_matrix = limits_matrix,
      error_codes = subset(error_codes, select = c(variables, msgs))
    )
  } else{
    limits_data <- NULL
  }

  return(limits_data)
}
```