Automatic Data Standardization

May 3rd, 2024

Title Automatic Data Standardization – Developer Facing Documentation

Version 0.7.1

Description This Automatic Data Standardization documentation is meant to provide an overview of the files/scripts to ensure that maintainability and potential future maintenance of a script is easy to understand and make changes to. Each script/file will have a descriptive breakdown of any functions, libraries, and logic for code.

Depends R (>= 4.0.0)

Built On R 4.3.2

Encoding UTF-8

Language en-CA

Author Cole Chuchmach [aut/cre]
Barret Monchka [aut/ctb]

Maintainer Cole Chuchmach

Date/Publication 2024-05-03 09:54:10 CST

R topics documented:

data_standardization_script.R

Description

data_standardization_script.R is the script that will take in a unclean data file, pre-processing it and outputting a file that is cleaned and standardized. The script contains **four** main functions that handle the preprocessing and error handling.

Details

The available specifications are:

- standardize_data()
- pre_process_chunks()
- pre_process_data()
- error_handle()

The packages used to aid in preprocessing the data are:

- DBI
- dplyr
- stringi
- stringr
- data.table
- readr
- haven
- openxlsx
- writexl
- tidyverse
- splitstackshape
- shiny
- tools

standardize_data()

Description

standardize_data() will call the **flag_standardizing_script.R** using source so that we can access its helper preprocessing functions, as well as establish a connection to the metadata information. Using it to determine what dataset_id is being preprocessed, passing the arguments to the pre_process_chunks() function.

Usage

standardize_data(file_path, input_dataset_code, input_flags, output_folder, standardization_rules_metadata)

Arguments

input_file_path	A path to a file.
input_dataset_code	A dataset_code, used in the query statement within the function which will get the desired dataset code. File names meant to be pre-processed should contain a prefix containing the dataset code kept in the metadata, separated by something like a hyphen, underscore, etc.
input_flags	A lookup table containing manually set or default values used by the flag script which will use the values of the flags to determine how to pre-process and output the health data.
output_folder	An output folder that the successfully cleaned data ready for linkage will be placed in once pre-processing is complete.
standardization_rules_metadata	A path to a metadata file.

Examples

standardize_data("input_data/data.txt", "samin", user_flags, "output_fold", "metadata.sqlite")

pre_process_chunks()

Description

pre_process_chunks() will use the metadata connection to send various queries to the database, grabbing the source fields of the dataset using the dataset_id, getting whether the input data has a header row, and what file extension the input file is.

This is followed by grabbing the standardizing module name for the fields we're interested in pre-processing, as well as the destination field/output names for what the pre-processed file column names will be.

The file extension is then used to determine how we will read the file, reading the file in chunks to avoid loading the entire dataset into memory, passing the chunked portion to the pre_process_data() function.

After a successful pre-process of the chunk, a note is made of how many rows have been read and continue reading the file from the point we left off last, after finishing processing, the file is moved to the output folder using the input argument.

Usage

```
pre_process_chunks(file_path, dataset_metadata_conn, dataset_id,
flag_lookup_table, output_folder, dataset_code)
```

Arguments

file_path	A path to a file.
dataset_metadata_conn	A database connection that is used in addition to the dataset_id, passed from the standardize_data() function. Is also used in the pre_process_data() function.
dataset_id	A dataset_id found and passed from the standardize_data() function, is used to make various calls to the database connection argument to obtain information required to pre-process the health data.
flag_lookup_table	A lookup table containing manually set or default values used by the flag script which will use the values of the flags to determine how to pre-process and output the health data.
output_folder	An output folder that the successfully cleaned data ready for linkage will be placed in once pre-processing is complete.
dataset_code	A dataset code used as prefix for the successful output and error files.

Details

There is an if-statement/branch for each file extension which attempts to use the fastest way of reading a specific type of file extension.

Files with the extensions, **.txt** and **.csv** make use of **fread** from the **data.table** package. Files with the extension **.sas7bdat** use **read_sas** from the **haven** package, and files that use a form of **fwf** use **read_fwf** from the **readr** package.

Each branch creates a variable called **chunk_size** which is the max number of rows read in a single iteration. The user can provide an integer **10,000 <= X <= 1,000,000** which is how many rows are read at a time. The read chunks can also be read using **shell commands** to aid in memory usage.

Once a chunk is read, an error check is made by comparing how many columns were found in the chunk, compared to the number of source fields in the metadata, if a differ is found, the program throws an exception and stops the program, writing an error file to the user explaining the error.

Once a chunked portion has been converted into a data table, the **dplyr** select function takes a subset of only the desired rows for processing using the field orders to determine which specific columns to grab.

After pre-processing, any **NA** values in the dataset are replaced with an empty record/string, and an additional column is placed in the data frame, containing a unique primary key which relates to a single entry/row in the cleaned data frame.

The program then writes the clean data table to a clean database connection, and based on what output the user decided they wanted the output format to be, it can also be written to a **csv** or **rds** file and makes note of how many rows were read on this iteration of chunking.

The function finishes off by returning the path to the sqlite file such that the **standardize_data()** function can determine if it will returning the data frame to the user.

pre_process_data()

Description

pre_process_data() is called from the pre_process_chunks() function, and pre-processes the unclean dataset by slowly building up a fully cleaned dataset using the pre-processing modules defined by the standardizing module ids defined in the metadata.

If the user wishes to design their own standardizing module, they may create a function that accepts **four parameters** (*field names, source field ids, current cleaned data frame, standardized column name*) and provide any rules or code to process the data how they want so long as it creates a new column using the standardized column names, and ends with returning the clean dataset with the new column.

To add a new function, the user should first access the **metadata_ui.R** shiny app and create a new standardizing module following the instructions in the app such that it can be used for future fields that may need to be standardized that way.

Usage

```
pre_process_data(source_data_frame, split_source_fields, db_conn,
dataset_to_standardize, standardized_name_lookup_table,
standardized_function_lookup_table, flag_lookup)
```

Arguments

source_data_frame
A source data frame, the chunk read and passed from the pre_process_chunks() function.

split_source_fields
A vector of split source fields that have a standardizing module assigned to them in the metadata, grouped by those standardizing modules.

standardization_rules_db
A database connection that is used in addition to the dataset_id, passed from the standardize_data() function. Is also used in the pre_process_data() function.

dataset_to_standardize
A dataset_id found and passed from the standardize_data() function, is used to make various calls to the database connection argument to obtain information required to pre-process the data.

standardized_name_lookup_table
A vector containing all standardized column names/destination fields for a standardizing module.

standardized_function_lookup_table
A vector containing all standardizing module names, used by concatenating/pasting the name onto the prefix "pre_process_" which will automatically call the correct module.

flag_lookup
A lookup table containing manually set or default values used by the flag script which will use the values of the flags to determine how to pre-process and output the health data.

Details

The **pre_process_** specifications are:

record_primary_key
If multiple source fields are used to form a primary key during linkage, the module will pass the values within the field to the clean data frame without modification but will prefix the string "**record_primary_key_**" onto the original column name.

individual_id
Normalizes the column in the source data frame that keeps track of whether a single person can have multiple records, doesn't modify the specific ID, is instead lifted straight from the source.

phin
Removes punctuation from the person's PHIN and replaces all non-numerical characters with a blank character. As well as replace any phin value consisting of solely 0's with an empty record.

registration_no
Replaces any punctuation from the source registration number with a blank space, and removes any all 0 records.

primary_given_name

Grabs unprocessed primary given names from the source data frame, passes the names to the **standardize_names()** function in the flag standardizing script, getting processed primary given names back.

secondary_given_name

Grabs unprocessed secondary given names from the source data frame, passes the names to the **standardize_names()** function in the flag standardizing script, getting processed secondary given names back. A unique flag option is used here for extracting middle initials if the user requests so.

primary_surname

Grabs unprocessed primary surnames from the source data frame, passes the names to the **standardize_names()** function in the flag standardizing script, getting processed primary surnames back.

prior_surname

Grabs unprocessed prior surnames from the source data frame, passes the names to the **standardize_names()** function in the flag standardizing script, getting processed prior surnames back.

secondary_surname

Grabs unprocessed secondary surnames from the source data frame, passes the names to the **standardize_names()** function in the flag standardizing script, getting processed secondary surnames back.

birth_date

First grabs the current birthdate field, and source field id.

Sends a query statement to the metadata connection, getting compound field information, specifically, the **compound_field_format_id**.

Grabs the unprocessed birthdates and makes two additional queries which will attempt to grab either the separators or indexes used to parse the birthdate, making a call to **split_compound_field()** in the flag standardizing script which will parse based on input split objects and whether its indexes or separators.

Lastly, another query is made which grabs the destination fields from the metadata and goes through each split vector of the split birthdate, putting the day, month, and year into the desired column using a standardized column name.

birth_year

Lifts the birth year straight from the source data frame, replacing any punctuation with a blank character.

birth_month

Lifts the birth month straight from the source data frame, replacing any punctuation with a blank character.

Further processing is done by converting any character-based months to their numerical counterparts (*i.e., Jan = 1, Feb = 2, etc…*)

birth_day

Lifts the birthday straight from the source data frame, replacing any punctuation with a blank character.

gender

Grabs the unprocessed genders from the source data frame, along with its source field ID.

It then runs a query on the metadata, grabbing all source values found in the source dataset, and what standardized values they map to in the metadata.

A gender lookup table is constructed, and the standardized genders are grabbed by passing the unprocessed genders through the lookup table, replacing any values that didn’t map with an empty string.

address1

Grabs the unprocessed addresses from the source data frame, runs the **extract_postal_codes()** function from the flag standardizing script to take out any postal codes that may be part of the record.

Next, runs it through the **standardize_addresses()** function in the flag standardizing script, getting processed addresses back.

Lastly, groups the potentially extracted postal codes into its own column, making sure to not add duplicates.

address2

Grabs the unprocessed alternative addresses from the source data frame, runs the **extract_postal_codes()** function from the flag standardizing script to take out any postal codes that may be part of the record.

Next, runs it through the **standardize_addresses()** function in the flag standardizing script, getting processed addresses back.

Lastly, groups the potentially extracted postal codes into its own column, making sure to not add duplicates.

city

Grabs the unprocessed addresses from the source data frame, runs the **extract_postal_codes()** function from the flag standardizing script to take out any postal codes that may be part of the record.

Next, runs it through the **standardize_addresses()** function in the flag standardizing script, getting processed addresses back.

Lastly, groups the potentially extracted postal codes into its own column, making sure to not add duplicates.

province

Uses the look up values in the metadata to standardize the province values lifted from the source data to a common format.

country

Lifts the country straight from the source data frame.

postal_code

Grabs the unprocessed postal codes from the field by using an extract function paired with a regular expression to pick out any postal codes that fit the format **CNCNCN** or **CNC NCN.**

Additional processing is done by replacing instances of O with 0, and I with 1, before adding it the processed data frame.

acquisition_date

First grabs the current acquisition date field, and source field id.

Sends a query statement to the metadata connection, getting compound field information, specifically, the **compound_field_format_id**.

Grabs the unprocessed acquisition dates and makes two additional queries which will attempt to grab either the separators or indexes used to parse the acquisition date, making a call to **split_compound_field()** in the flag standardizing script which will parse based on input split objects and whether its indexes or separators.

Lastly, another query is made which grabs the destination fields from the metadata and goes through each split vector of the split acquisition date, putting the day, month, and year into the desired column using a standardized column name.

acquisition_year

Lifts the acquisition year straight from the source data frame, replacing any punctuation with a blank character.

acquisition_month

Lifts the acquisition month straight from the source data frame, replacing any punctuation with a blank character.

Further processing is done by converting any character-based months to their numerical counterparts (*i.e., Jan = 1, Feb = 2, etc…*)

acquisition_day

Lifts the acquisition day straight from the source data frame, replacing any punctuation with a blank character.

compound_name

Function first grabs the unprocessed compound names and sends a query statement to the metadata connection, getting the **compound_field_format_id**.

Function makes an additional query to grab the separators used to split the compound names and uses the **split_compound_field()** function from the flag standardizing script.

Lastly, another query is made to get the destination fields, where the split vector of names is assigned to the destination fields in the destination mapping order after being standardized and having any initials extracted in the case of secondary given names.

numeric_date

Function grabs the unprocessed date fields, and sends a query statement to the metadata connection, getting the **numeric_date_format_id** and **destination_field_type** (*birthdate or acquisition date*).

Function makes an additional query using the format_id to get the **time_measurement** (*Days, Seconds, Hours, etc.*), as well as the **origin_date**.

Then uses the numerical format information along with the **as.POSIXct** function to convert a numerical date into a common YYYY MM DD format. Further splitting the date into three parts, year, month, and day.

Lastly, makes another query statement to get the destination fields using the destination field type, and assigns the split date accordingly.

pass_through_field

If a field is used in the linkage process using the values straight from the source field, its passed through to the clean data frame using the same column name and field values.

record_priority

Function handles record priority by passing through the original column similar to the pass_through_field() function, but holds metadata information on the backend with information on priority of the values used for breaking ties during the linkage process.

Each **pre_process_** function ends by trimming any whitespace in the records and adds a column to an empty data frame with the column name equal to the **standardized_col_name** input argument. Some functions use additional functions, these are found in the **flag_standardizing_script.R** file and make use of the user defined flags.

Pre-processing is handled by a loop that goes through each group of fields that belong to a standardizing module, passing these split source fields, their IDs, the current built processed data frame and what the standardized column name should be. We finish this function by returning a fully constructed clean data frame.

Before returning the clean data frame, a standardizing option is pulled which will summarize the imputation values obtained from **impute_sex** and **extract_postal_codes** if the user has requested. This will accept a path to a directory where the imputation metadata will go.

The metadata information consists of **missing values**, **missing %, values assigned, missing % assigned, and total % assigned**.

The sex imputation metadata will group by **male**, **female**, and **overall**, informing how many sex values were assigned to each group.

The postal code metadata consists of one row and will identify how many postal code records were missing, and then record how many were extracted from location-based fields, using the same fields as the sex metadata.

error_handle()

Description

The **error_handle()** function is an exception/error handling function that will log error messages and print these messages to an output log, allowing the user to read and see what went wrong in the program, allowing them to hopefully understand and make corrections to a specific datasets metadata.

Usage

error_handle(metadata_conn, err_msg, cleaned_file)

Arguments

standardization_rules_metadata	A database connection that is used in addition to the dataset_id, passed from the standardize_data() function. Is also used in the pre_process_data(), and pre_process_chunks() functions.
err_msg	An error message indicated what went wrong in the program, this message will be written to an output **.txt** file and placed in a folder containing error messages.
cleaned file	A database connection to the sqlite file holding the currently processed dataset that was being written to at the time of error.
error_file_path	A path to the input file.
error_folder_path	A path to the output folder where the error **.txt** file will be created.

Details

The function will print out the error message to the console and use the output folder path passed as parameter as the location for the error message file.

Then create a text file with the file path name being used as a prefix for the **.txt** file called **FILE_PATH_ERRORS_README.txt** which will then write the **error** argument to the file, then placing the file in the folder.

Finally, closes the database and any clean file connections before running an additional **stop()** command to signal that we're closing the program.

flag_standardizing_script.R

Description

flag_standardizing_script.R is the script that houses many pre-processing functions that are abstract enough so that multiple standardizing modules may access and call the functions for use.

Details

The available specifications are:

- standardize_names()
- standardize_locations()
- split_compound_field()
- list_curr_names()
- list_curr_given_names()
- list_curr_surnames()
- impute_sex()
- standardize_output_file()
- extract_postal_codes()
- concat_postal_codes()
- compile_non_linkage_data()
- create_standardizing_options_lookup()

standardize_names()

Description

standardize_names() will take in a vector of names, along with the user flag look up table, standardizing people's names based on the user defined settings.

Usage

```
standardize_names(input_names, flag_lookup_table)
```

Arguments

input_names	A vector of names, can be primary or secondary given names, primary or secondary surnames, and prior surnames.
flag_lookup_table	A lookup table containing manually set or default values that will determine how persons names are handled specifically by the function.

Examples

standardize_names(unprocessed_primary_given_names, user_flags)

Details

The function first assigns the unprocessed names to a value called **curr_names** before pre-processing begins.

The first name pre-processing option is removing any punctuation from a persons name, including hyphens, apostrophes, commas, etc.

The second name pre-processing option is compressing name white space for given names or surnames that may be two or more words.

The third name pre-processing option is converting names to Latin-ASCII, removing any accents or diacritics in a person's name.

The fourth/last name pre-processing option is converting name case to either uppercase, lowercase, or keeping default values.

The processed names are then returned.

standardize_locations()

Description

standardize_locations() will take in a vector of location values, along with the user flag look up table, standardizing people's location fields based on the user defined settings.

Usage

standardize_locations(input_addresses, flag_lookup_table)

Arguments

input_locations	A vector of addresses, can be primary or alternative addresses, countries, cities, or any location-based field.
flag_lookup_table	A lookup table containing manually set or default values that will determine how persons addresses are handled specifically by the function.

Example

standardize_locations(unprocessed_addresses, user_flags)

Details

The function first assigns the unprocessed locations to a value called **curr_locations** before pre-processing begins.

The first pre-processing option is removing any punctuation from the location, including hyphens, apostrophes, commas, etc.

The second pre-processing option is compressing white space for given locations, compressing house numbers, names, apartment numbers all together.

The third pre-processing option is converting locations to Latin-ASCII, removing any accents or diacritics.

The fourth/last pre-processing option is converting the location values case to either uppercase, lowercase, or keeping its default values.

The processed locations are then returned.

split_compound_field()

Description

split_compound_field() will take in a vector of single fields that are to be split into separate fields, it will either use separators or indexes based on the source fields metadata.

Usage

```
split_compound_field(compound_field, split_objects, split_type)
```

Arguments

compound_Field	A vector of compound data meant to be split using the objects in “split_objects”, the other argument “split_type” will determine the algorithm/logic for splitting a compound field.
split_objects	A vector of split objects that will either use **split** logic to separate a compound field based on a specific character or string. Or we will use **substr** logic to separate a compound field based on integer indexes.
split_Type	Splitting a compound field will either be of type **separators**, or of type **indexes**.

Example

```
split_compound_field(birthdates, c(“/”, “/”), “separators”)
```

Details

If the **split_type** is chosen to be “separators”, a vector is constructed called **name_split**, equal to the length of **number of separators + 1**.

The algorithm runs a for loop through each separator in the list of separator objects, first trimming any leading or trailing whitespace, followed by using the **str_split_fixed** function from the **stringr** package, splitting on the first instance of the current separator.

The current split is then put into its own vector, the first half which is ready to be mapped to a destination field, and the other half which may still need to be split further.

The algorithm continues until all separators have been gone through, the complete split is then returned.

If the **split_type** is chosen to be “indexes”, a vector is constructed once again called **name_split**, this time equal to the **number of separators**.

The algorithm runs a loop through the indexes, beginning with the start index at position **1**, and the end index equal to the value in the metadata. The **substr** function is used to substring the compound field apart, and the substring is placed in the name_split variable.

This continues until all index values have been gone through, then the complete split is returned.

list_curr_names()

Description

list_curr_names() will take in a vector of names, the current built up data frame, and append all current names (primary/secondary given names, primary/secondary surnames, and prior surnames) if the user flag is defined as yes before pre-processing.

Usage

```
list_curr_names(names_to_append, df)
```

Arguments

names_to_append	A vector of names to be concatenated together, separated by a space. Contains all current processed names of the dataset.
df	The current built up and processed data frame, a new column called “**all_curr_names**” is defined and the processed names are appended to it.

Example

list_curr_names(names, processed_data_frame)

list_curr_given_names()

Description

list_curr_given_names() will take in a vector of primary of secondary given names, the current built up data frame, and append all given names if the user flag is defined as yes before pre-processing.

Usage

list_curr_given_names(names_to_append, df)

Arguments

names_to_append	A vector of given names to be concatenated together, separated by a space. Contains all current processed given names of the dataset.
df	The current built up and processed data frame, a new column called "**all_curr_given_names**" is defined and the processed names are appended to it.

Example

list_curr_given_names(names, processed_data_frame)

list_curr_surnames()

Description

list_curr_surnames() will take in a vector of primary of secondary surnames, the current built up data frame, and append all surnames if the user flag is defined as yes before pre-processing.

Usage

list_curr_surnames(names_to_append, df)

Arguments

names_to_append	A vector of surnames to be concatenated together, separated by a space. Contains all current processed given names of the dataset.
df	The current built up and processed data frame, a new column called "**all_curr_surnames**" is defined and the processed surnames are appended to it.

Example

list_curr_surnames(names, processed_data_frame)

impute_sex()

Description

After processing a chunk of data, the user may ask for missing sex values to be imputed, one option for imputing these missing values is the **gender** and **genderdata** packages, this will require that the packages are installed on their own as some systems may need them to be installed locally, so the package can't depend on them.

Another available option is to submit a custom file to help attempt to infer sexes. The file should contain at least two columns, one containing first names with the other being the sexes for that name. The file would be read and then compressed such that each name will only appear once with what the majority sex for that name would be.

The last option is to internally impute sex, by way of using the processed first names and genders from the source file to try and impute missing sexes, which also takes the majority sex of each name.

Usage

impute_sex(source_df, processed_names, processed_sexes, flag_lookup_table)

Arguments

source_df	The current processed/cleaned data frame of the input file.
processed_names	The current processed primary given names on this chunk iteration or after processing finishes.
processed_sexes	The current processed sexes on this chunk iteration.
flag_lookup_table	Lookup table used to determine what output file format is to be used. Output format can be of type **rdata**, **csv**, **xlsx** or just **sqlite**.

Example

impute_sex(clean_df, first_names, processed_sexes, flags)

standardize_file_output()

Description

Takes in a connection to the database, the output path folder, and the flag lookup table. Uses the value from the flag lookup table when searching for "**file_output**" to determine what format the output file will be.

Usage

```
standardize_file_output(db_conn, output_folder_path, flag_lookup_table,
input_dataset_code)
```

Arguments

db_conn	A connection to the metadata database, used to read the cleaned table after processing has finished.
output_folder_path	The output_folder_path will be used to determine where the standardized file be placed.
flag_lookup_table	Lookup table used to determine what output file format is to be used. Output format can be of type **rds**, **csv**, or just **sqlite**.
input_dataset_code	Used as prefix for the cleaned file to help differentiate amongst other cleaned files.

Example

```
standardize_file_output (metadata_conn, path_to_folder, user_flags, "samin")
```

extract_postal_codes()

Description

Takes in a vector of values that may potentially have postal codes embedded in them, regular expressions are used to extract the values and append them to a new column in the current processed data frame called "**alt_postal_code**".

Usage

```
extract_postal_codes(to_extract, df)
```

Arguments

to_extract	A vector of data which may contain concatenated postal codes which should be extracted.
df	The current built up and processed data frame, a new column called "**alt_postal_code**" is defined and the extracted postal codes are appended to it. The postal codes are stored here before

we return the column which acts as a vector of all the extracted postal codes.

Example

extract_postal_codes (addresses, processed_data_frame)

Details

First try to extract postal codes from the values we're interested in extracting from, this is done first using the form CNCNCN where **C = character** and **N = number**. Extracted postal codes are then added to the **alt_postal_code** column of the data frame.

If nothing was extracted after the first try, the regular expression for extracting changes to include a space in between the first and second half of the postal code, and the extracted postal codes are also added to the **alt_postal_code** column of the data frame.

A list is returned containing the original unprocessed values we extracted from, as well as the extracted postal codes we currently have.

concat_postal_codes()

Description

Modifies the **alt_postal_code** column of the data frame, by splitting the data and removing any duplicate postal codes that it may contain in a single row.

Usage

concat_postal_codes(df, alt_postal_codes)

Arguments

df	The current built up and processed data frame containing the column **alt_postal_code.**
alt_postal_codes	A vector of alternative postal codes that will first be added to the data frame column before de-duplication takes place.

Example

concat_postal_codes (processed_data_frame, postal_codes_to_add)

Details

The processed data frame is modified by first splitting each row in a vector using a space character, it is then processed using the **unique()** function along with the **map_chr** function keep only unique postal codes/postal codes appearing only once.

The modified data frame is then returned.

compile_non_linkage_data()

Description

Fields that were not included in the processed data frame that is to be used during linkage, as well as some additional fields that may be included such as postal code, gender, birthday, are also included in the output program and health data.

Usage

```
compile_non_linkage_data(source_data_frame, db_conn, dataset_id)
```

Arguments

source_data_frame	The source data frame, containing both the health and program data we're interested in extracting, as well as the valid linkage fields found by checking the metadata database with the dataset_id.
db_conn	A connection to the metadata database, used with the **dataset_id** argument to determine what fields are valid to also include in the health and program data output.
dataset_id	The dataset_id of the dataset we're currently cleaning.

Example

```
compile_health_and_program_data (df, metadata_conn, 10)
```

Details

From the read chunk of data on the current iteration, a subset of the desired columns are selected and passed to this function,

create_standardizing_options_lookup()

Description

Using this function the user can supply any number of flag options they want (not all are required, missing entries will be given a default value) which will be used as additional standardization rules.

A lookup table consisting of chosen flags and the assigned default options for bad inputs or undefined choices is returned to the user.

Arguments

convert_name_case
Convert the capitalization of a person's name. **(Options: "upper", "lower", "default")**

convert_name_to_ascii
Remove diacritics of a person's name. **(Options: "yes", "no")**

remove_name_punctuation
Remove any symbols or punctuation from a person's name. **(Options: "yes", "no")**

compress_name_whitespace
Replace name white space with an empty string symbol. **(Options: "yes", "no")**

list_all_curr_given_names
Combine all given names of a person into an additional column. **(Options: "yes", "no")**

list_all_curr_surnames
Combine all surnames of a person into an additional column. **(Options: "yes", "no")**

list_all_curr_names
Combine all names of a person into an additional column. **(Options: "yes", "no")**

impute_sex
Impute missing values in sex fields. **(Options: "yes", "no")**

impute_sex_type
How should the sex imputation take place? **(Options: "default" - Must have the gender and genderdata packages installed, "custom" - Must be a .csv file with two columns [primary_given_name] & [sex], "internal" - Use sex values from the source data set)**

chosen_sex_file
If the custom type is chosen, supply an input file.

compress_location_whitespace
Replace location-based fields white space with an empty string symbol. **(Options: "yes", "no")**

remove_location_punctuation
Remove any symbols or punctuation from location-based fields. **(Options: "yes", "no")**

convert_location_case
Convert the capitalization of a location-based field. **(Options: "upper", "lower", "default")**

convert_location_to_ascii
Remove diacritics of a location based field. **(Options: "yes", "no")**

extract_postal_code	Attempt to extract postal codes from location based fields and place an additional column. **(Options: "yes", "no")**
file_output	What file output format is desired. **(Options: "csv", "rds", "sqlite")**
output_health_and_program_data	Output non-linkage fields in a separate file? **(Options: "yes", "no")**
chunk_size	How many rows should be read at a time when reading the source file in chunks? **(Options: 10000-1000000)**
debug_mode	Print additional information to the console in case of potential bugs? **(Options: "on", "off")**
max_file_size_output	What is the max file size that a data frame can be before it isn't returned? **(Options: integer in Mega-bytes)**
read_mode	Read the input file using normal file reading methods (***path***) or use shell commands to lessen the load on memory (***cmd***).

Example

```
flags <- create_standardizing_options_lookup(convert_name_case = "upper",
impute_sex = "yes", chunk_size = 15000, max_file_size_output = 200)
```

metadata_ui.R

Description

metadata_ui.R is the shiny app that will allow the user to make various changes to, or view their standardization rules in an easy way.

Details

The available specifications are:

- Home
- View Standardization Rules
- Add and Update Databases
- Manage Database Standardization Rules
- Enable and Disable Databases
- Create New Standardizing Module
- Create New Destination Field
- startMetadataUI()

The packages used to aid in the app are:

- shiny
- DBI
- DT
- shinyjs
- shinyBS

Home

Description

The home page provides a brief and descriptive overview of how each modification page works and allows the user to flow between these pages using buttons.

Details

For each specification of the UI and its inner pages, a textbox provides the user with a rundown of each page and buttons that will link the user to these pages directly with no input required.

The server side logic has a function written for each button which will transition the user to the alternative pages by modifying the **modification_form** value to first determine what page we should be on, the **update_type** to determine whether that page will be used for adding or updating, and the **table_to_add_to** value which will determine what type of standardizing rules we will be adding or updating.

View Standardization Rules

Description

View standardization rules allow users to view all current existing tables along with their standardization rules and limit searches by specific datasets.

Details

view_dataset_table_ui

Creates a select input for the user, containing every table in the metadata, this can be accessed using the **view_dataset_table** variable when accessing input variables.

view_limited_dataset

Runs a query on the metadata, getting every dataset name from the **datasets** table, using it to construct a select input field for the user, which can be accessed via the **view_dataset_selection_sf** variable.

Observe (changes in viewed dataset)

Looks for changes in **view_dataset_table** and **view_dataset_selection_sf.** If a table contains multiple different values based on datasets, it will use the specific limited source field to limit the search to a specific dataset. Otherwise, the entire table will be printed.

Add and Update Databases

Description

Allows users to add a new database to the datasets metadata, involves the user entering a dataset code, name, whether or not it has a header row, and what file extension it has.

Alternatively, selecting a row from the table containing all current enabled datasets will allow users to update the database information.

Details

created_new_dataset

A hidden select input value that will determine if the user had just recently created a new dataset, this will be used to provide the user with an additional button after creating a dataset entry which links them directly to the **Add Source Fields** page.

updated_dataset

A hidden select input value that will determine if the user had just recently updated an existing dataset, this will be used to provide the user with an additional button after updating a dataset entry which links them directly to the **Update Source Fields** page.

view_current_created_datasets

Renders a data table showcasing all the current datasets in the metadata that are enabled.

observe (rows_selected)

Selecting a row will pre-populate the input fields with the information stored in the metadata, allowing for quick updates.

observeEvent (submit_new_dataset_record)

Using input values from the UI section of the script, we use the **new_record_ dataset_code**, **dataset_name**, **has_header**, and **file_extension** to insert into datasets.

Afterwards, a green button is provided to the user which links them to add new source fields to the database.

observeEvent (update_dataset_record)

Updates the selected entry in the metadata using the new/updated UI input values supplied by the user. Afterwards, provides a green link button to start updating the source fields of the database.

observe (dataset_code)

Shortens the dataset code provided in the input field if it exceeds **20 characters**.

go_to_manage_database

Updates the **modification_form**, **update_type**, and **table_to_add_to** values to transition to the **Add Source Fields** page.

go_to_manage_database_update

Updates the **modification_form**, **update_type**, and **table_to_add_to** values to transition to the **Update Source Fields** page.

Add to Source Fields

dataset_to_update

Renders a data table for the user containing all datasets that a user can add a new source field to. Can be

	accessed using the **dataset_to_update_rows_selected** variable.
select_standardizing_module	Renders a select input field for the user, containing all standardizing modules available when adding a new source field. Can be accessed using **selected_standardizing_module**. This will additionally provide the user with a printed description of what the current selected module does/how it handles data.
standardizing_module_description	Submits a query using the selected **standardizing module** which will fill in a small textbox below the input fields with a description of what the chosen **standardizing module** performs on the source data.
standardizing_module_type_create	Renders a hidden select input field that determines what standardizing module type is being selected when adding a new record, this is used to bring up additional fields when creating data, such as compound or categorical fields. Can be accessed using **standardizing_type_create_source_field**.
observeEvent (selected_standardizing_module)	Observes changes in what standardizing module the user selects, using that module id to make a query and determine what module type it is in the metadata, updating the module type for creating a new record.
compound_format_types	If the new record being created is a compound field, a data table is rendered showing all the compound fields that exist for the user to choose from, OR they can add a new compound format if they require one that doesn't already exist. Can be accessed using **compound_format_types_rows_selected**.
numeric_format_types	If the new record being created is a numerical date field, a data table is rendered showing all the numerical date fields that exist for the user to choose from, OR they can add a new numerical date format if they require one that doesn't already exist. Can be accessed using **numeric_format_types_rows_selected**.

categorical_inputs

If the user is adding a source field that behaves like a categorical variable, the function will use the UI variable **number_of_categorical_variables** to print out a list of side-by-side input fields.

There will be **n** text inputs which is the input/source value, which can be accessed by the variable **categorical_input_value_n**.

And there will be **n** select inputs which is the standardized categorical value, this can be accessed by the variable **categorical_output_value_n**.

record_priority_inputs

If the user is adding a source field that would be used as a tie breaker field in data linkages, they are provided with the option to choose how many values are in the field. Then, they may enter each value and its corresponding priority where **1** is the highest priority, **2** second highest, and so on.

observe (field_order)

Observe for changes in what dataset the user selects, this will modify the maximum value the user can select when choosing a field order, this affected the **new_field_order_max_add** variable.

observeEvent (submit_new_dataset_field)

Uses the general field inputs along with the **standardizing_module_type_create** value to determine what specific insert statements to run as handling compound formats, categorical, numerical, and priority formats are all a little different.

observeEvent (add_compound_format_for_new_sf)

Transitions the users to the **Add Compound Format** page so that they may add a new format if an existing format does not suit their dataset.

observeEvent (add_numeric_format_for_new_sf)

Transitions the users to the **Add Numeric Date Format** page so that they may add a new format if an existing format does not suit their dataset.

Add to Compound Formats

created_new_compound_format_new_sf

A hidden select input that will return the user to the **Add Source Fields** page if they transitioned

	here via the button on that page when adding a new source field.
created_new_compound_format_update_sf	A hidden select input that will return the user to the **Update and Delete Source Fields** page if they transitioned here via the button on that page when updating a source field.
add_compound_field_formats	Renders a data table containing all current compound field formats that are viewable. Provided below the table are also **two examples** of how to add new formats.
separator_inputs_to_add	The field will contain inputs equal to the **number_of_separators_to_add** input multiplied by 2 plus 1. Allowing for **n** separators and **n+1** destination fields. The values can be accessed using the prefixes of **separator_destination_field_to_add_n** and **separator_to_add_n**.
observeEvent (submit_new_compound_format)	When user submits a compound format, it will use the variables of **number_of_separators_to_add** or **number_of_indexes_to_add**, **new_compound_format_description_to_add** and **new_compound_format_to_add**. It will insert the variables and create a new compound format, using that format id to also insert the separators and destination fields.

Add to Categorical Fields	
categorical_fields_datasets	Renders a data table containing all current datasets that exist in the metadata, allowing users to select one to modify. Can be accessed using **categorical_fields_datasets_rows_selected**.

categorical_source_fields_to_add

Using the previously defined variable, it will select the dataset_id that the user selected and store it in a global variable for later.

While also rendering a data table which shows all current categorical fields of the chosen dataset.

categorical_source_fields_to_add_curr

Renders a table which presents the user with all the current categorical fields of the selected source field.

categorical_fields_to_add

Renders rows of fields equal to the **number_of_categorical_fields_to_add** variable, this creates **n** text inputs and **n** select inputs for the user to enter source categorical values and map them to the standardized values.

The variables can be accessed by using **categorical_field_input_value_n** and **categorical_field_output_value_n**.

observeEvent
(add_new_categorical_fields)

When the user submits their categorical fields a ***for loop*** is run which runs a query for each row the user entered values for.

Each loop is one insert statement using the input and output variables previously defined.

Add to Categorical Values

curr_categorical_values_add

Renders a data table showing all the currently created standardized categorical values.

observeEvent
(add_new_categorical_values)

When user submits a new categorical value, it uses the variable **new_categorical_value** and uses an INSERT query to add the new value to the metadata.

Add to Numeric Date Formats

created_new_numeric_format_new

A hidden select input that will return the user to the **Add Source Fields** page if they transitioned here via the button on that page when adding a new source field.

created_new_numeric_format_update

A hidden select input that will return the user to the **Add Source Fields** page if they transitioned here via the button on that page when adding a new source field.

add_numeric_date_formats

Renders a data table showing all the currently created numeric data formats.

observeEvent
(submit_new_numeric_date_format)

When user submits a new numeric date format, the inputs consisting of the label, origin date (*date to start counting from*), and time measurement (*secs, mins, hrs*) to use to count and uses an insert statement to create a new format.

Add to Record Priorities

record_priority_datasets

Renders a data table containing all current datasets that exist in the metadata, allowing users to select one to modify. Can be accessed using **record_priority_datasets_rows_selected**.

record_priority_fields_to_add

Using the previously defined variable, it will select the dataset_id that the user selected and store it in a global variable for later.

While also rendering a data table which shows all current record priority fields of the chosen dataset.

record_priority_fields_to_add_curr

Renders a table which presents the user with all the current record priorities and associated values of the selected source field.

record_priorities_to_add

Renders rows of fields equal to the **number_of_record_priorities_to_add** variable, this creates **n** text inputs and **n** numeric inputs for the user to enter source field values and map them to what priority it has over the other values.

The variables can be accessed by using **record_priority_field_input_value_n** and **record_priority_field_output_value_n**.

observeEvent
(add_new_record_priorities)

When the user submits their record priority fields a ***for loop*** is run which submits a query for each row the user entered values for.

Each loop is one insert statement using the input and output priorities previously defined.

Update and Delete Source Fields	
source_field_to_update_datasets	Renders a data table of all current data sets, allowing the user to select the one they want to update.
source_field_to_update	Renders a data table containing all current source fields in the dataset the user previously chose.
updated_compound_format_type	Renders a data table containing all current compound formats for a user to select if the source field they are updating is to be a compound field.
numeric_format_types_update	Renders a data table containing all current numeric date formats with a button to create a new format if an existing one is not suitable.
select_updated_standardizing_module	Renders a select input field containing all standardizing modules for a user to select. This will also generate a description below it of how fields of this type would be standardized.
standardizing_module_description_update	Submits a query using the selected **standardizing module** which will fill in a small textbox below the input fields with a description of what the chosen **standardizing module** performs on the source data.
standardizing_module_type_update	Renders a hidden select input field that determines what standardizing module type is being selected when updating an existing record, this is used to bring up additional fields when updating data, such as compound or categorical fields. Can be accessed using **standardizing_type_update_source_field**.

observeEvent (select_updated_standardizing_module)

Observes changes in what standardizing module the user selects, using that module id to make a query and determine what module type it is in the metadata, updating the module type for updating an existing record.

observeEvent (source_field_to_update_rows_selected)

Observe event for when user selects a source field to update, pull the source field id and field order before updating. This will also prepopulate the UI inputs with the current metadata entry.

updated_record_priority_inputs

If the user is updating a source field so that it would be used as a tie breaker field in data linkages, they are provided with the option to choose how many values are in the field. Then, they may enter each value and its corresponding priority where **1** is the highest priority, **2** second highest, and so on.

updated_categorical_inputs

There will be **n** text inputs which is the input/source value, which can be accessed by the variable **categorical_input_value_n**.

And there will be **n** select inputs which is the standardized categorical value, this can be accessed by the variable **categorical_output_value_n**.

observeEvent (update_source_field)

Compared to the add_source_field function, updating makes use of deleting instances where source field ids appear in the compound fields, categorical fields, numerical date fields, and record priority fields tables.

While also using UPDATE queries to change the field name, order, and what standardizing module it uses.

This still will run INSERT queries if the user changes a field to a new standardizing module like compound or categorical fields.

observeEvent (add_compound_format_for_updated_sf)	Transitions the users to the **Add Compound Format** page so that they may add a new format if an existing format does not suit their dataset.
observeEvent (add_numeric_format_for_updated_sf)	Transitions the users to the **Add Numeric Date Format** page so that they may add a new format if an existing format does not suit their dataset.
observeEvent (delete_source_field)	A small window appears with the option to confirm deleting the selected source field, otherwise the user may click cancel or off the window to return without deleting.
observeEvent (delete_source_field_confirm)	If the user confirms to deleting a source field then the function runs and the source field along with all instances of its source field id being deleted.

Update Compound Formats	
updatable_compound_formats	Renders a data table containing all compound formats that are able to be updated, this means it only presents the user with formats used by zero source fields. Below the table are also **two** examples of how to update to a new formats of both types.
observeEvent (updatable_compound_fomats_rows_selected)	Observes what row the user selected and grabs the format id of the compound format that the user will update.
updatable_separator_inputs	The field will contain inputs equal to the **new_updatable_number_of_separators** input multiplied by 2 plus 1. Allowing for **n** separators and **n+1** destination fields. The values can be accessed using the prefixes of

	updatable_separator_destination_field_n and **updatable_separator_n**.
observeEvent (update_existing_compound_format)	When user submits an updated compound format, the previous separators and destination mappings are deleted. Then an UPDATE query is run on the chosen compound format, updating the format and its description, followed by the INSERT statements for the new separators and field destinations.

Update and Delete Categorical Fields

dataset_to_update_cfs	Renders a data table allowing the user to select a dataset to update the categorical fields of.
observeEvent (dataset_to_update_cfs_rows_selected)	Renders a data table containing all categorical fields of the selected dataset, and makes note of the selected dataset_id.
updated_standardized_value	Renders a select input containing all standardized categorical values. Can be accessed by the variable **standardized_categorical_value_updated**.
observeEvent (categorical_field_to_update_rows_selected)	Observes for when a categorical field value is selected and makes note of the selected source field id, and source categorical value. This will prepopulate the UI inputs with the selected metadata record.
observeEvent (submit_updated_categorical_field)	When user submits an updated categorical field, the source value and standardized value are updated using a single UPDATE query.
observeEvent (delete_categorical_field)	A small window appears with the option to confirm deleting the selected categorical field, otherwise the user may click cancel or off the window to return without deleting.

observeEvent (delete_categorical_field_confirm)	When user confirms to delete a categorical field value both times, the lookup value that would be used in processing is deleted from the database.

Update Numeric Date Formats

updatable_numeric_formats	Renders a data table showing all the currently created numeric data formats that are updatable, meaning that they are used **zero** times by source fields.
observeEvent (updatable_numeric_formats_rows_selected)	Based on the row the user selected, the UI inputs are prepopulated with the metadata record values.
observeEvent (update_exisiting_numeric_format)	When user submits a new numeric date format to replace the current selected one to update, the inputs consisting of the label, origin date (*date to start counting from*), and time measurement (*secs, mins, hrs*) to use to count and uses an insert statement to create a new format and remove the old one.

Update and Delete Record Priority

dataset_to_update_record_priority	Renders a data table allowing the user to select a dataset to update the record priority fields of.
observeEvent (dataset_to_update_record_priority_rows_selected)	Renders a data table containing all record priority fields of the selected dataset and makes note of the selected dataset_id.
observeEvent (record_priorities_to_update_rows_selected)	Observes for when a record priority field value is selected and makes note of the selected source field id, source field value, and its priority number. This will prepopulate the UI inputs with the selected metadata record.

observeEvent (submit_updated_record_priority)

When user submits an updated record priority field, the source value and priority value are updated using a single UPDATE query.

observeEvent (delete_record_priority)

A small window appears with the option to confirm deleting the selected source field, otherwise the user may click cancel or off the window to return without deleting.

observeEvent (delete_record_priority_confirm)

When user confirms to delete a record priority field value both times, the lookup value that would be used in processing is deleted from the database.

Enable and Disable Databases

Description

Allows the user to disable databases such that versioning of database metadata entries can be kept, so the user has the option to create a new metadata entry when a modification to an existing dataset comes in.

Or they may disable a current database entry and enable a previous version in the event that the database layout and field order revert back. So long as there is no currently enabled database that shares the same dataset code, in that case, the database the existing dataset code should be changed or disabled before the other is enabled.

Details

databases_disable

Renders a data table of all datasets from the "datasets" table in the metadata that are currently enabled. The user may select one and press the button below the table to disable it.

databases_enable

Renders a data table of all datasets from the "datasets" table in the metadata that are currently disabled. The user may select one and press the button below the table to enable it, so long as no enabled database shares the same dataset code.

observeEvent (disable_selected_database)

Updates the metadata of the current selected dataset by setting its enabled setting to 0, this means this versioning of the database will not be considered when standardizing a database with the same code and name.

observeEvent (enable_selected_database)
Updates the metadata of the current selected dataset by setting its enabled setting to 1, this means this versioning of the database will be considered when standardizing a database with the same code and name. Like before, this will only work if there is no enabled database sharing the same dataset code.

Create New Standardizing Module

Description

If the user would like to create their own standardizing module such that they would like to handle fields in a certain way, they may do this by adding a new standardizing module so that it may be made available to future or current source fields.

The user will be asked to input a standardizing module name, a short description of what the module accomplishes, what destination field the module would output to and whether the field using the module should also be included alongside the non-linkage fields in the other output file that is not standardized.

Once a user's standardizing modules is in the metadata, they may add a new function/module to the **process_data()** function in the **data_standardization_script.R** file by prefixing the function with **pre_process_** and then adding their function name to the end. For example, adding a new module called "**new_data**", the function would be **pre_process_new_data <- function()**.

Additionally, the function should intake **four function parameters**. This includes field names, source field ids, current cleaned data frame, and standardized column name. This should look like (**pre_process_new_data <- function(new_data_fields, source_field_ids, df, standardized_col_name){}**

Details

curr_standardizing_modules
Renders a data table of all current standardizing modules.

new_module_dest
Creates a UI render in the form of a select input which contains all current destination field mappings, with the

	additional option for the standardizing module being added not having or needing a destination field.
observeEvent (create_module)	Takes the inputs from the UI text boxes, and select inputs, running a singular query statement to add the standardizing module to the list of all modules, allowing the user to now select this module on pages like “Add Source Fields” and “Update Source Fields”.

Create New Destination Field

Description

A user may create a new destination field if they would like a compound format to output data into a new column name, or if they would like a standardizing module, they are in the process of making output to a destination field specific to that module.

The destination field takes in an input of a destination field name, and a short description of what this destination field captures in terms of data.

Details

curr_destination_fields	Renders a data table of all current destination fields.
observeEvent (create_dest_field)	Takes the inputs from the UI text boxes, running a singular query statement to add the destination field to the list of all destination fields, allowing the user to now select this destination fields on pages like “Add Compound Formats” and “Create New Standardizing Modules”.

startMetadataUI()

Description

The function used to start the metadata UI application, it takes in a file which should be a **standardizing rules metadata file**, it will confirm the passed file is valid by confirming it is first an **.sqlite** file, then by verifying the tables of the dataset.

Usage

```
startMetadataUI(“path/to/metadata.sqlite”)
```

Arguments

metadata_file_path	A path to a metadata containing the standardizing rules.

data_standardization_ui.R

Description

data_standardization_ui.R is the shiny app that will allow the user to standardize their input files in a clean and accessible way by the use of a GUI front end, this will allow users to specify an input file, output folder and modify/change any processing flags to fit their need.

Details

The available specifications are:

- Home Page
- Standardize Data
- startDataStandardizationUI()

The packages used to aid in the app are:

- shiny
- DBI
- DT
- shinyjs
- shinyBS
- datastan

Home Page

Description

Like the metadata user interface, the first and only page serves as an easy/accessible way for users to alter the processing flags to handle specific fields, as well as enter their input file and folder.

The input flags are grouped by their specific flag standardizing script functions and are provided alongside a help button to provide the users with extra clarification on what certain flags achieve.

Details

The available specifications and input options are:

- Source File
- Output Folder
- Standardizing Rules Metadata
- Name Case Conversion

- Name ASCII Conversion
- Name Punctuation Removal
- Name Whitespace Compression
- Given Names Listing
- Surnames Listing
- All Names Listing
- Impute Sex
- Sex Imputation Type
- Sex Imputation Custom File
- Address Whitespace Compression
- Address Punctuation Removal
- Address Case Conversion
- Address ASCII Conversion
- Postal Code Extraction from Location Based Fields
- Output File Type
- Output Non-Linkage Variables
- Chunking Size
- Reading Mode

Standardize Data

Description

When user chooses to press the **Standardize Data** at the bottom of the UI, the specifications dictated above have their currently selected input values pulled and converted into a flag look up table that will be passed to the standardizing script.

Next, the input file the user provided undergoes brief error handling by way of ensuring that both a file and output folder have been submitted, and if the user has chosen to impute sex using a custom file, a check has been made for that.

Afterwards, the input file is broken down to obtain its base name such that we can identify what its dataset code is by splitting the name and taking the first split (*the prefix*). We then confirm that the extension is a valid file for reading (*either .txt, .csv or .sas7bdat*) or we will throw an error.

Once everything is ready, we will then run the standardize_data function from the standardizing script, while periodically updating the user with notification messages throughout the process, sharing how many rows have currently been read from the input file, with a success message following completion.

If an error happened to occur, an error notification message is pushed to the UI informing the user to check the error output folder and see what errors occurred, with instructions on how to fix it within the metadata, or the provided source files.

Additional Notes

Details

The SQLite database that stores standardizing rules metadata allows for concurrent access to the file so that multiple users may make SELECT, INSERT, UPDATE, or DELETE queries with the file at the same time.

Upon starting the Metadata GUI and using the data standardization function either programmatically or through the Data Standardization GUI will begin by making a query to the database and setting the busy timeout to 10 seconds, meaning if the database is locked due to a transaction, the query will attempt to rerun many times during that locked period so that the query executes successfully.

If the query is unable to execute before the database is unlocked, then a try/catch block will handle the error by rolling back the transaction, helping keep the database in a valid state where the part of the database that was either updated or deleted returned to its previous state.

As both the UPDATE and DELETE pages of the metadata GUI are the only page to involve executing multiple queries, they use transactions more compared to the singular INSERT and SELECT statements from other pages.