CrossVA Documentation

Release 0.5a

MITRE

Oct 19, 2022

Getting Started

CHAPTER 1

Configuration Files

CrossVA runs by applying the specified mappings in its configuration files to the raw data provided. The package comes with some default configurations which map common inputs to common outputs, but it is possible to create your own customized version.

These files must have the following columns:

- New Column Name
- New Column Documentation
- Source Column ID
- Source Column Documentation
- Relationship
- Condition
- Prerequisite

Of those, only *New Column Name*, *Source Column ID*, *Relationship*, and *Condition* must be filled out in every row.

Each row in the configuration mapping gives instructions to map information from a single column in the raw input data to a new column in the final transformed data.

Each row can be read as roughly: The new column *New Column Name* will get the value True in rows where the column *Source Column ID* in the input data is *Relationship* to *Condition*, and, optionally, the *Prerequisite* column in the output data is also True.

When the input data is NA, the NAs will be preserved through the transformation. That is, no matter the *Relationship* and *Condition*, if the value in the source column is NA, then the result will be NA, instead of a boolean.

Note: If the same column name has multiple conditions specified, then each operation will only update the pre-existing column. False and NA values in the transformed data will be overwritten if a prior condition was false, but a new condition was true.

This creates an implicit *OR* relationship between the different conditions listed.

Structure of Mapping Configuration table

New Column Name	New Column Documentation	Source Column ID	Source Column Documentation	Relationship	Condition	Prerequisite
DEL_ELSE	Did she give birth elsewhere, e.g. on the way to a facility?	Id10337	(Id10337) Where did she give birth?	eq	other	
DEL_ELSE	Did she give birth elsewhere, e.g. on the way to a facility?	Id10337	(Id10337) Where did she give birth?	eq	on_route_to_hospital_or_facility	

- The first row indicates that where the column 'Id10337' is equal to other in the input data, then 'DEL_ELSE' is true in the output data.
- The first row indicates that where the column 'Id10337' is equal to 'on_route_to_hospital_or_facility' in the input data, then DEL_ELSE is true in the output data.

In this sense, if 'Id10337' is 'on_route_to_hospital_or_facility', 'DEL_ELSE' will be set to False on the first condition, and then updated to True on the second condition. However, if 'Id10337' is *other*, then 'DEL_ELSE' will be set to True on the first condition, and remain True (not updated) on the second condition, regardless of the value in the input data, since a condition for the new column has already been satisfied.

New Column Name

The *New Column Name* column should contain the name of the new column to be created in the final data. All of the columns required by the intended algorithm should be listed, with corresponding documentation in the New Column Documentation column if possible.

New Column Documentation

The *New Column Documentation* column should contain a brief statement explaining what the new column is meant to represent.

Source Column ID

The *Source Column ID* column should contain the unique identifier at the end of the column name in the input data. It should only be left blank in cases where the New Column being created (and required by the intended algorithm) depends on information that is unavailable in the source data, and thus there is no relevant source column.

Source Column Documentation

The *Source Column Documentation* column should contain a brief statement explaining what information the source column contains. This, along with the *New Column Documentation* column, makes it much easier to at-a-glance check the logic behind these mappings.

Relationship

The *Relationship* column should contain one of 8 valid relationships, which use the value in the *Condition* column to return a boolean value for the output data. The currently supported relationships are:

- eq: is equal to
- gt: is greater than
- ge: is greater than or equal to
- lt: is less than
- le: is less than or equal to
- ne: is not equal to
- contains: contains the substring
- between: is between 2 numbers, inclusive

Condition

The *Condition* column should contain the condition being applied to the source column. For example, *yes*, *5* or *15 to 30*.

Note: Conditions in the form ## to ## should only be used when the relationship is between, in order to give the two numbers that make up the low and high end of the acceptable range.

Prerequisite

The *Prerequisite* column is optional. It should be left blank if there is no prerequisite. If there is a prerequisite condition, then this column should contain the name of the column in the final data to reference.

For example, the new column *MAGEGP1* is created based on the condition of if the source column *ageInYears* is *between 12 to 19*. It also lists a prerequisite of *FEMALE*, which is a previously created column in the output data, containing its own boolean, which checks to see if Id10019 is equal to “female”.

CHAPTER 2

transform module

Defines main CrossVA function, *transform* which maps raw VA data into data for use with a VA algorithm in OpenVA.

transform.**transform**(*mapping*, *raw_data*, *raw_data_id=None*, *verbose=2*, *preserve_na=True*, *result_values={'Absent': 'n', 'NA': '.', 'Present': 'y'}*)

transforms raw VA data (*raw_data*) into data suitable for use with a VA algorithm, according to the specified transformations given in *mapping*.

Parameters

- **mapping** (*string, tuple or Pandas DataFrame*) – Should be either a tuple in form (input, output), a path to csv containing a configuration data file, or a Pandas DataFrame containing configuration data
- **raw_data** (*string or Pandas DataFrame*) – raw verbal autopsy data to process
- **raw_data_id** (*string*) – column name with record ID
- **verbose** (*int*) – integer from 0 to 5, controlling how much status detail is printed to console. Silent if 0. Defaults to 2, which will print only errors and warnings.
- **preserve_na** (*bool*) – whether to preserve NAs in data, or to count them as FALSE. Overridden with True for InSilicoVA, False for InterVA4 when mapping is given as a tuple. Defaults to TRUE, which allows NA values to perpetuate through the data.
- **result_values** (*dict*) – available as a simple customization option if user would like values indicating presence, absence, and NAs to be mapped to certain values.

Returns the raw data transformed according to specifications given in mapping data. Default values are y where symptom is present, n where symptom is absent, and if . are preserved, they are represented in the data as NaNs. If NAs are not preserved, they are considered to be false / absent / 0.

Return type Pandas DataFrame

Examples

You can specify the mapping as ('input', 'output') and the path to csv as a string:

```
>>> transform(("2016WHOv151", "InterVA4"), "resources/sample_data/2016WHO_mock_
↪data_1.csv").loc[range(5),["ACUTE","CHRONIC","TUBER"]]
   ACUTE  CHRONIC  TUBER
0      y        n      .
1      y        n      .
2      n        y      .
3      n        y      .
4      y        n      .
```

You can also give the data and mapping as Pandas DataFrames:

```
>>> my_special_data = pd.read_csv("resources/sample_data/2016WHO_mock_data_1.csv")
>>> my_special_mapping = pd.read_csv("resources/mapping_configuration_files/
↪2016WHOv151_to_InSilicoVA.csv")
>>> transform(my_special_mapping, my_special_data).loc[range(5),["ACUTE","CHRONIC
↪","TUBER"]]
   ACUTE  CHRONIC  TUBER
0      y        n      .
1      y        n      .
2      n        y      .
3      n        y      .
4      y        n      .
```

Note that by default, *preserve_na* is *True* and NA values will be left in. If *preserve_na* is *False*, or if the algorithm does not preserve NAs, then NA values will be filled in as 0's, as they are in the first InterVA4 example above.

The user can also pass in a different mapping dictionary for result_values to change the values from their defaults of 0 (False / Absent), 1 (True / Present), and NaN (No data / missing), if they need their results in a different format.

```
>>> transform(("2016WHOv151", "InterVA4"), "resources/sample_data/2016WHO_mock_
↪data_1.csv", result_values={"Absent":"A","Present":"P","NA":"Missing"}).
↪loc[range(5),["ACUTE","CHRONIC","TUBER"]]
  ACUTE CHRONIC    TUBER
0     P       A  Missing
1     P       A  Missing
2     A       P  Missing
3     A       P  Missing
4     P       A  Missing
```

The mapping-data relationship is designed to be as flexible as possible, while still emphasizing tracebility and alerting the user to data integrity issues.

Not every source column in the mapping needs to be represented in the data. If source columns are missing in the source data, then those columns will be created and filled with NA values.

```
>>> transform(("2016WHOv151", "InSilicoVA"), "resources/sample_data/2016WHO_mock_
↪data_2.csv").loc[range(5),["ACUTE","FEMALE","MARRIED"]]
Validating Mapping-Data Relationship . . .
<BLANKLINE>
 WARNINGS
[?]          3 (1.3%) expected source column IDs listed in mapping file ('-
↪ageInDaysNeonate', '-Id10019', and '-Id10059') were not found in the input data␣
↪columns. Their values will be NA.
```

(continues on next page)

(continued from previous page)

```
[?]          '-Id10019' is missing, which affects the creation of  column(s)
↪'FEMALE', and 'MALE'
[?]          '-Id10059' is missing, which affects the creation of  column(s)
↪'MARRIED'
[?]          '-ageInDaysNeonate' is missing, which affects the creation of␣
↪column(s) 'DIED_D1', 'DIED_D23', 'DIED_D36', 'DIED_W1', and 'NEONATE'
   ACUTE  FEMALE  MARRIED
0      y       .        .
1      y       .        .
2      y       .        .
3      y       .        .
4      y       .        .
```

transform will also accept mapping configurations with missing values, with new columns that are specified but missing source columns. These new columns will be created so that the final result has the correct expeted columns for the algorithm, but filled with NA values to indicate the lack of information. If *preserve_na* is set to *False*, then the NA values will also be *False*.

This situation is common between certain questionnaire sources and algorithms. For example, in the mapping between the PHRMC Short questionnaire to InterVA5 mapping, there are 107 InterVA5 variables that are listed in the mapping configuration to be created, but have no corresponding question in PHRMC short.

For example, variables i004a and i004b have no specifications in the mapping below. They are still listed under "New Column Name" so CrossVA knows that they should be created in the final result, but because they have no logic defined, they will be left as their default value of NA.

```
>>> phrmc_to_interva5 = pd.read_csv('resources/mapping_configuration_files/
↪PHRMCShort_to_InterVA5.csv')
>>> phrmc_to_interva5.iloc[:5,[0,2,4,-1]]
  New Column Name Source Column ID Relationship Meta: Notes
0           i004a               NaN          NaN   Not asked
1           i004b               NaN          NaN   Not asked
2           i019a          gen_5_2            eq         NaN
3           i019b          gen_5_2            eq         NaN
4           i022a         gen_5_4h            ge         NaN
```

The *transform* function will warn the user of this behavior.

```
>>> transform(phrmc_to_interva5, "resources/sample_data/PHRMC_mock_data_1.csv").
↪iloc[:5,:5]
Validating Mapping Configuration . . .
<BLANKLINE>
 WARNINGS
[?]      124 new column(s) listed but not defined in Mapping Configuration␣
↪detected. These ('i004a', 'i004b', 'i059o', 'i082o', 'i087o', 'i091o', 'i092o',
↪'i093o', 'i094o', 'i095o', etc) will be treated as NA.
Validating Mapping-Data Relationship . . .
<BLANKLINE>
 WARNINGS
[?]      9 (5.7%) expected source column IDs listed in mapping file ('child_6_2',
↪'child_4_4', 'child_4_20', 'child_4_7a', 'child_4_40', 'child_4_28', 'child_4_30
↪', 'child_1_5a', and 'child_5_1') were not found in the input data columns.␣
↪Their values will be NA.
[?]      'child_1_5a' is missing, which affects the creation of  column(s) 'i358a'
[?]      'child_4_20' is missing, which affects the creation of  column(s) 'i171o'
[?]      'child_4_28' is missing, which affects the creation of  column(s) 'i208o'
[?]      'child_4_30' is missing, which affects the creation of  column(s) 'i233o'
```

(continues on next page)

(continued from previous page)

```
[?]      'child_4_4' is missing, which affects the creation of  column(s) 'i150a'
[?]      'child_4_40' is missing, which affects the creation of  column(s) 'i200o'
[?]      'child_4_7a' is missing, which affects the creation of  column(s) 'i183o'
[?]      'child_5_1' is missing, which affects the creation of  column(s) 'i418o'
[?]      'child_6_2' is missing, which affects the creation of  column(s) 'i130o'
   ID  i004a  i004b  i019a  i019b
0   1      .      .      y      n
1   2      .      .      n      n
2   3      .      .      n      n
3   4      .      .      y      n
4   5      .      .      n      n
```

However, the mapping-data relationship must be valid. For example, if the source column IDs are not unique for the input data - that is, if multiple columns in the input data contain the same source ID - then validation will fail.

For example, *bad_data* contains columns named *A-Id10004* and *B-Id10004*, but the 2016 WHO mapping is looking for just *-Id10004* as a source ID. CrossVA cannot tell which column should be used, so validation fails.

```
>>> bad_data = pd.read_csv("resources/sample_data/2016WHO_bad_data_1.csv")
>>> transform(("2016WHOv151", "InSilicoVA"), bad_data)
Validating Mapping-Data Relationship . . .
<BLANKLINE>
 ERRORS
[!]      1 source column IDs ('-Id10004') were found multiple times in the input␣
↪data. Each source column ID should only occur once as part of an input data␣
↪column name. It should be a unique identifier at the end of an input data␣
↪column name. Source column IDs are case sensitive. Please revise your mapping␣
↪configuration or your input data so that this condition is satisfied.
```

CHAPTER 3

configuration module

Structure for Configuration class

class configuration.**Configuration**(*config_data*, *verbose=1*, *process_strings=True*)

Bases: object

Configuration class details the relationship between a set of input data and output data. It is composed of MapConditions that transform an input data source (2012 WHO, 2016 WHO 141, 2016 WHO 151, PHRMC SHORT) into a different data form (PHRMC SHORT, InSilicoVA, InterVA4, InterVA5, or Tarrif2) for verbal autopsy.

Variables

- **given_columns** (*Pandas Series*) – columns of mapping dataframe.
- ***required_columns*** (*Pandas Series*) – required columns in mapping data.
- ***main_columns*** (*list*) – the four main columns required in config_data.
- ***valid_relationships*** (*Pandas Series*) – contains list of valid relationships to use in comparisons. Relationships should be an attr of Pandas Series object, or be defined as a subclass of MapCondition.
- **config_data** (*Pandas DataFrame*) – dataframe containing mapping relationships written out.
- **given_prereq** (*Pandas Series*) – lists pre-requisites referenced in config data.
- **new_columns** (*Pandas Series*) – lists the new columns to be created with config data.
- **source_columns** (*Pandas Series*) – lists the source columns required in the raw input data.
- **verbose** (*int*) – controls default verbosity of printing to console.
- **process_strings** (*boolean*) – whether or not to remove whitespace and non-alphanumeric characters from strings in condition field and in raw_data during mapping.
- ***validation*** (Validation) – a validation object containing the validation checks made

describe()
Prints the mapping relationships in the Configuration object to console.

Parameters **None** –

Returns None

Examples

```
>>> MAP_PATH = "resources/mapping_configuration_files/"
>>> EX_MAP_1 = pd.read_csv(MAP_PATH + "example_config_1.csv")
>>> Configuration(EX_MAP_1).describe()
MAPPING STATS
<BLANKLINE>
 -   16 new columns produced ('AB_POSIT', 'AB_SIZE', 'AC_BRL', 'AC_CONV', 'AC_
↪COUGH', etc)
 -   12 source columns required ('Id10403', 'Id10362', 'Id10169', 'Id10221',
↪'Id10154', etc)
 -   7 relationships invoked ('eq', 'lt', 'between', 'ge', 'contains', etc)
 -   13 conditions listed ('yes', '14', '10', '21', '15 to 49', etc)
 -   1 prerequisites checked ('FEMALE')
```

list_conditions()
Lists the final mapping conditions contained in Configuration object

Returns list of MapConditions, where each MapConditions is created from a row of processed mapping data.

Return type list

Examples

```
>>> MAP_PATH = "resources/mapping_configuration_files/"
>>> EX_MAP_1 = pd.read_csv(MAP_PATH + "example_config_1.csv")
>>> c = Configuration(EX_MAP_1)
>>> c.list_conditions()[:5]
[<StrMapCondition:     AB_POSIT = [column Id10403].eq(yes)>,
 <StrMapCondition:     AB_SIZE = [column Id10362].eq(yes)>,
 <NumMapCondition:     AC_BRL = [column Id10169].lt(14.0)>,
 <NumMapCondition:     AC_CONV = [column Id10221].lt(10.0)>,
 <NumMapCondition:     AC_COUGH = [column Id10154].lt(21.0)>]
```

main_columns = ['New Column Name', 'Source Column ID', 'Relationship', 'Condition']

required_columns = 0 New Column Name 1 New Column Documentation 2 Source Column ID 3 S

valid_relationships = 0 gt 1 ge 2 lt 3 le 4 between 5 eq 6 ne 7 contains Name: valid

validate(*verbose=None*)
Prepares and validates the Configuration object's mapping conditions. Validation fails if there are any inoperable errors. Problems that can be fixed in place are processed and flagged as warnings.

Parameters **verbose** (*int*) – controls print output, should be in range 0-5, each higher level includes the messages of each level below it. Where verbose = 0, nothing will be printed to console. Where verbose = 1, print only errors to console, where verbose = 2, also print warnings, where verbose = 3, also print suggestions and status checks, where verbose = 4, also print passing validation checks, where verbose = 5, also print description of configuration conditions. Defaults to None; if none, replace with self.verbose attribute

Returns

boolean representing whether there are any errors that prevent validation

Return type Boolean

Examples

```
>>> MAP_PATH = "resources/mapping_configuration_files/"
>>> EX_MAP_2 = pd.read_csv(MAP_PATH + "example_config_2.csv")
>>> c = Configuration(EX_MAP_2)
>>> c.validate(verbose=4)
Validating Mapping Configuration . . .
<BLANKLINE>
 CHECKS PASSED
[X]          All expected columns ('New Column Name', 'New Column␣
↪Documentation', 'Source Column ID', 'Source Column Documentation',
↪'Relationship', 'Condition', and 'Prerequisite') accounted for in␣
↪configuration file.
[X]          No leading/trailing spaces column New Column Name detected.
[X]          No leading/trailing spaces column Relationship detected.
[X]          No leading/trailing spaces column Prerequisite detected.
[X]          No leading/trailing spaces column Condition detected.
[X]          No whitespace in column Condition detected.
[X]          No upper case value(s) in column Relationship detected.
[X]          No upper case value(s) in column Condition detected.
[X]          No non-alphanumeric value(s) in column Source Column ID detected.
[X]          No non-alphanumeric value(s) in column Relationship detected.
[X]          No non-alphanumeric value(s) in column Condition detected.
[X]          No new column(s) listed but not defined in Mapping Configuration␣
↪detected.
[X]          No NA's in column New Column Name detected.
[X]          No NA's in column Source Column ID detected.
<BLANKLINE>
 ERRORS
[!]          3 values in Relationship column were invalid ('eqqqq', 'another␣
↪fake', and 'gee'). These must be a valid method of pd.Series, e.g. ('gt',
↪'ge', 'lt', 'le', 'between', 'eq', 'ne', and 'contains') to be valid.
[!]          2 row(s) containing a numerical relationship with non-number␣
↪condition detected in row(s) #8, and #9.
[!]          2 values in Prerequisite column were invalid ('ABDOMM', and
↪'Placeholder here'). These must be defined in the 'new column name' column␣
↪of the config file to be valid.
<BLANKLINE>
 WARNINGS
[?]          2 whitespace in column New Column Name detected in row(s) #6,␣
↪and #8. Whitespace will be converted to '_'
[?]          1 whitespace in column Relationship detected in row(s) #4.␣
↪Whitespace will be converted to '_'
[?]          1 whitespace in column Prerequisite detected in row(s) #9.␣
↪Whitespace will be converted to '_'
[?]          1 non-alphanumeric value(s) in column New Column Name detected␣
↪in row(s) #6. This text should be alphanumeric. Non-alphanumeric characters␣
↪will be removed.
[?]          2 duplicate row(s) detected in row(s) #1, and #14. Duplicates␣
↪will be dropped.
[?]          1 NA's in column Relationship detected in row(s) #3.
```

(continues on next page)

(continued from previous page)

```
[?]          1 NA's in column Condition detected in row(s) #6.
False
```

class configuration.**CrossVA**(*raw_data, mapping_config, na_values=['dk', 'ref', ''], verbose=2*)

Bases: object

Class representing raw VA data, and how to map it to an algorithm

Variables

- **mapping** (*type*) – a validated Configuration object that details how to transform the type of data in *raw_data* to the desired output.
- **data** (*Pandas DataFrame*) – a Pandas DataFrame containing the raw VA data
- **prepared_data** (*Pandas DataFrame*) – a Pandas DataFrame containing a prepared form of the VA data to use with the Configuration object.
- ***validation*** (Validation) – Validation object containing the validation checks that have been made on the raw data and between the raw data and mapping Configuration.
- **verbose** (*int*) – Controls verbosity of printing to console, 0-5 where 0 is silent.

process()

Applies the given configuration object's mappings to the given raw data.

Args: None

Returns a dataframe where the transformations specified have been applied to the raw data, resulting

Return type Pandas DataFrame

validate(*verbose=None*)

Validates that RawVAData's raw input data and its mapping configuration object are compatible and prepares input data for use.

Parameters **verbose** (*int*) – int from 0 to 5, representing verbosity of printing to console. Defaults to None; if None, replaced with self.verbose attribute.

Returns True if valid, False if not.

Return type boolean

Examples

```
>>> MAP_PATH = "resources/mapping_configuration_files/"
>>> EX_MAP_1 = pd.read_csv(MAP_PATH + "example_config_1.csv")
>>> EX_DATA_1 = pd.read_csv("resources/sample_data/mock_data_2016WHO151.csv")
>>> CrossVA(EX_DATA_1, Configuration(EX_MAP_1)).validate(verbose=0)
True
```

CHAPTER 4

validation module

Module containing Validation class, and Vcheck class and its subclasses

class validation.**Err**(*message*)
Bases: *validation.VCheck*

VCheck subclass representing a serious problem in data validation that prevents validation.

Examples

```
>>> Err("This is a data validation error").expand()
Tier                                  Error
Bullet                                  [!]
Level                                     1
Title                                ERRORS
Message    This is a data validation error
dtype: object
```

bullet()
abstract property, must be overriden. Should be a str, representing a bullet point

level()
abstract property, must be overriden. Should be int ,representing VCheck tier

tier()
abstract property, must be overriden. Should be str, representing name of VCheck tier

title()
abstract property, must be overriden. Should be str, representing title of VCheck type

class validation.**Passing**(*message*)
Bases: *validation.VCheck*

VCheck subclass representing a passed check in data validation, where there is no problem.

Examples

```
>>> Passing("This is a passing data validation check").expand()
Tier                                          Passing
Bullet                                            [X]
Level                                               4
Title                                   CHECKS PASSED
Message    This is a passing data validation check
dtype: object
```

bullet()
abstract property, must be overriden. Should be a str, representing a bullet point

level()
abstract property, must be overriden. Should be int ,representing VCheck tier

tier()
abstract property, must be overriden. Should be str, representing name of VCheck tier

title()
abstract property, must be overriden. Should be str, representing title of VCheck type

class validation.**Suggest**(*message*)
Bases: *validation.VCheck*

VCheck subclass representing a minor problem with data that does not prevent data validation.

Examples

```
>>> Suggest("This is a data validation suggestion").expand()
Tier                                     Suggestion
Bullet                                          [i]
Level                                             3
Title                                   SUGGESTIONS
Message    This is a data validation suggestion
dtype: object
```

bullet()
abstract property, must be overriden. Should be a str, representing a bullet point

level()
abstract property, must be overriden. Should be int ,representing VCheck tier

tier()
abstract property, must be overriden. Should be str, representing name of VCheck tier

title()
abstract property, must be overriden. Should be str, representing title of VCheck type

class validation.**VCheck**(*message*)
Bases: object

Abstract class fior a single validation check

bullet
abstract property, must be overriden. Should be a str, representing a bullet point

expand()
Expands VCheck information as a Pandas Series

Parameters **None** –

Returns representing VCheck attributes as a Pandas Series

Return type Pandas Series

Examples

```
>>> Err("Error Message").expand()
Tier               Error
Bullet               [!]
Level                  1
Title             ERRORS
Message    Error Message
dtype: object
```

level
abstract property, must be overriden. Should be int ,representing VCheck tier

tier
abstract property, must be overriden. Should be str, representing name of VCheck tier

title
abstract property, must be overriden. Should be str, representing title of VCheck type

class validation.**Validation**(*name=''*)
Bases: object

Validation object represents an organized dataframe of validation checks

Variables **vchecks** (*Pandas DataFrame*) – a dataframe containing the expanded form of the VCheck instances that have been added.

affected_by_absence(*missing_grped*)
adds a validation check as *Warn* describing the items in missing_grped, which detail the impact that missing columns have on newly created mappings.

missing_grped (Pandas Series): series where the index is the name of the missing source column, and the values are a list of affected values.

Returns: None

all_valid(*given*, *valid*, *definition*)
adds a validation check where all values in *given* must be in *valid* to pass. *fail_check* is *Err* (fails validation).

Parameters

- **given** (*Pandas Series*) – the items representing input given
- **valid** (*Pandas Series*) – list of all possible valid items accepted in *given*
- **definition** (*str*) – string describing what makes an item in *given* be in *valid*

Returns None

Examples

```
>>> v = Validation()
>>> v.all_valid(pd.Series(["a","b"], name="example input"),  pd.Series(["a","b
↪","c"],  name="valid value(s)"), "pre-defined")
>>> v.all_valid(pd.Series(["a","b","c"], name="example input"),  pd.Series(["a
↪","d"],  name="valid value(s)"), "'a' or 'd'")
>>> v.report(verbose=4)
Validating  . . .
<BLANKLINE>
 CHECKS PASSED
[X]          All values in example input are valid.
<BLANKLINE>
 ERRORS
[!]          2 values in example input were invalid ('b', and 'c').
These must be 'a' or 'd' to be valid.
```

check_na(*df*)

Adds a validation check flagging the rows in every column of *df* that are *None*

Parameters **df** (*Pandas DataFrame*) – a Pandas DataFrame with columns that should have no NA values

Returns None

Examples

```
>>> v = Validation()
>>> test_df = pd.DataFrame({"A":["a","B","c"], "B":["D","e",None]})
>>> v.check_na(test_df)
>>> v.report(verbose=4)
Validating  . . .
<BLANKLINE>
 CHECKS PASSED
[X]          No NA's in column A detected.
<BLANKLINE>
 WARNINGS
[?]          1 NA's in column B detected in row(s) #2.
```

fix_alnum(*df*)

Adds a validation check flagging the rows in every column of *df* that contain non-alphanumeric characters. Regex removes all characters that are not alpha-numeric, but leaves periods that are part of a number.

Parameters **df** (*Pandas DataFrame*) – a Pandas DataFrame with columns that should have only alphanumeric characters

Returns *df* where alphanumeric characters are removed

Return type Pandas DataFrame

Examples

```
>>> v = Validation()
>>> test_df = pd.DataFrame({"A":["a","3.0","c"],  "B":["??.test","test<>!",";
↪test_data"]})
>>> v.fix_alnum(test_df)
   A          B
0  a       test
```

(continues on next page)

(continued from previous page)

```
1  3.0     test
2  c  test_data
>>> v.report(verbose=4)
Validating  . . .
<BLANKLINE>
 CHECKS PASSED
[X]      No non-alphanumeric value(s) in column A detected.
<BLANKLINE>
 WARNINGS
[?]      3 non-alphanumeric value(s) in column B detected in row(s)
#0, #1, and #2. This text should be alphanumeric. Non-alphanumeric
characters will be removed.
```

fix_lowcase(*df*)

Adds a validation check flagging the rows in every column of *df* that contain lowercase characters.

Parameters **df** (*Pandas DataFrame*) – a Pandas DataFrame with columns that should have only uppercase characters

Returns *df* where all characters are uppercase

Return type Pandas DataFrame

Examples

```
>>> v = Validation()
>>> test_df = pd.DataFrame({"A":["a","B","c"], "B":["D","e","F"]})
>>> v.fix_lowcase(test_df)
   A  B
0  A  D
1  B  E
2  C  F
>>> v.report(verbose=4)
Validating  . . .
<BLANKLINE>
 WARNINGS
[?]      2 lower case value(s) in  column A detected in row(s) #0,
and #2. Convention to have this text be uppercase. Lower case text
will be made uppercase.
[?]      1 lower case value(s) in  column B detected in row(s) #1.
Convention to have this text be uppercase. Lower case text will be
made uppercase.
```

fix_upcase(*df*)

Adds a validation check flagging the rows in every column of *df* that contain uppercase characters

Parameters **df** (*Pandas DataFrame*) – a Pandas DataFrame with columns that should have only lowercase characters

Returns *df* where all characters are lowercase

Return type Pandas DataFrame

Examples

```
>>> v = Validation()
>>> test_df = pd.DataFrame({"A":["a","B","c"], "B":["D","e","F"]})
>>> v.fix_upcase(test_df)
   A  B
0  a  d
1  b  e
2  c  f
>>> v.report(verbose=4)
Validating  . . .
<BLANKLINE>
 WARNINGS
[?]      1 upper case value(s) in column A detected in row(s) #1.
Convention is to have this text be lowercase. Upper case text will
be made lowercase.
[?]      2 upper case value(s) in column B detected in row(s) #0,
and #2. Convention is to have this text be lowercase. Upper case
text will be made lowercase.
```

fix_whitespace(*df*)

Adds a validation check flagging the rows in every column of *df* that contain whitespace

Parameters **df** (*Pandas DataFrame*) – a Pandas DataFrame with columns that should have no whitespace

Returns

***df* where whitespace is replaced with an** underscore

Return type Pandas DataFrame

Examples

```
>>> v = Validation()
>>> test_df = pd.DataFrame({"A":["a"," B ","Test Data"],  "B":["D"," e","F "]}
↪)
>>> v.fix_whitespace(test_df)
           A  B
0          a  D
1          B  e
2  Test_Data  F
>>> v.report(verbose=4)
Validating  . . .
<BLANKLINE>
 CHECKS PASSED
[X]          No whitespace in column B detected.
<BLANKLINE>
 WARNINGS
[?]          1 leading/trailing spaces column A detected in row(s) #1. ␣
↪Leading/trailing spaces will be removed.
[?]          2 leading/trailing spaces column B detected in row(s)  #1, and
↪#2. Leading/trailing spaces will be removed.
[?]          1 whitespace in column A detected in row(s) #2.  Whitespace will␣
↪be converted to '_'
```

flag_elements(*flag_where*, *flag_elements*, *criteria*)

Adds a validation check seeing if any values in flag_where are true, and then reports on the corresponding

items in flag_elements.

Parameters

- **flag_where** (*Pandas Series*) – a boolean Pandas Series where True represents a failed check
- **flag_elements** (*Pandas Series*) – a boolean Pandas Series listing elements that are affected by True values in *flag_where*
- **criteria** (*String*) – a brief description of what elements are being flagged and reported on

Returns None

Examples

```
>>> v = Validation("element test")
>>> v.flag_elements(pd.Series([False, False]),  pd.Series(["A", "B"]), "red␣
↪flag(s)")
>>> v.flag_elements(pd.Series([False, True]),  pd.Series(["A", "B"]), "blue␣
↪flag(s)")
>>> v.report(verbose=4)
Validating element test . . .
<BLANKLINE>
 CHECKS PASSED
[X]          No red flag(s) in element test detected.
<BLANKLINE>
 WARNINGS
[?]          1 blue flag(s) in element test detected. These ('B') will be␣
↪treated as NA.
```

flag_rows (*flag_where*, *flag_criteria*, *flag_action=''*, *flag_tier=<class 'validation.Warn'>*)

Adds a validation check seeing if any values in flag_where are true, where fail_check is of type flag_tier. Note that rows are reported counting from 0.

Parameters

- **flag_where** (*Pandas Series*) – a boolean Pandas Series where True represents a failed check.
- **flag_criteria** (*str*) – a noun clause describing the criteria for an item to be flagged in *flag_where*
- **flag_action** (*str*) – string describing the action to be taken if an item is flagged. Defaults to "".
- **flag_tier** (VCheck) – should be either Suggest, Warn, or Err, is the seriousness of the failed check.

Returns None

Examples

```
>>> v = Validation()
>>> v.flag_rows(pd.Series([False, False]),  flag_criteria="true values")
>>> v.flag_rows(pd.Series([False, True]),  flag_criteria="true values")
>>> v.report(verbose=4)
```

(continues on next page)

(continued from previous page)

```
Validating  . . .
<BLANKLINE>
CHECKS PASSED
[X]      No true values detected.
<BLANKLINE>
WARNINGS
[?]      1 true values detected in row(s) #1.
```

is_valid()

Checks to see if instance is valid.

Parameters **None** –

Returns

True if is valid (has no errors in vchecks) and False if instance has errors or where vchecks is empty.

Return type bool

Examples

```
>>> Validation().is_valid()
False
>>> v = Validation()
>>> v.must_contain(pd.Series(["A", "B"]), pd.Series(["B"]))
>>> v.is_valid()
True
>>> v.must_contain(pd.Series(["A", "B"]), pd.Series(["C"]))
>>> v.is_valid()
False
```

must_contain(*given*, *required*, *passing_msg=''*, *fail=<class 'validation.Err'>*)

adds a validation check where *given* must contain every item in *required* at least once to pass, and *fail_check* is *fail*, (fails validation).

Parameters

- **given** (*Pandas Series*) – the items representing input given
- **required** (*Pandas Series*) – the items required to be in *given*
- **passing_msg** (*str*) – Message to return if all items in *expected* are listed in *given*. Defaults to "".
- **fail** (VCheck) – the outcome if the check fails. Default is Err.
- **impact** (*Pandas Series*) – a corresponding series to *required* that represents the affected information when

Returns None

Examples

```
>>> v = Validation()
>>> v.must_contain(pd.Series(["a","b","c"], name="example input"),  pd.
↪Series(["a","b"],  name="example requirement(s)"),  "all included")
```

(continues on next page)

(continued from previous page)

```
>>> v.must_contain(pd.Series(["a","b","c"], name="example input"),  pd.
↪Series(["a","b","d"],  name="example requirement(s)"))
>>> v.report(verbose=4)
Validating  . . .
<BLANKLINE>
 CHECKS PASSED
[X]          all included
<BLANKLINE>
 ERRORS
 [!]          1 (33.3%) example requirement(s) ('d') were not found in␣
↪example input. Their values will be NA.
```

no_duplicates (*my_series*)

adds a validation check as *Err* if any items in my_series are duplicates. Intended to alert users of issues where there are duplicate columns before an exception is raised.

my_series (Pandas Series): series where there should not be dupes

Returns: None

no_extraneous (*given*, *relevant*, *value_type*)

adds a validation check where all values in *given* should also be in *relevant* to pass. *fail_check* is *Warn*

Parameters

- **given** (*Pandas Series*) – the items representing input given
- **relevant** (*Pandas Series*) – all items in *given* that will be used
- **value_type** (*str*) – string describing the kind of noun that is listed in *given*

Returns None

Examples

```
>>> v = Validation()
>>> v.no_extraneous(pd.Series(["a","b"], name="example input"),  pd.Series(["a
↪","b","c"],  name="relevant value(s)"), "example")
>>> v.no_extraneous(pd.Series(["a","b","c"], name="example input"),  pd.
↪Series(["a","d"],  name="relevant value(s)"), "example")
>>> v.report(verbose=4)
Validating  . . .
<BLANKLINE>
CHECKS PASSED
[X]      No extraneous example found in example input.
<BLANKLINE>
ERRORS
[!]      2 extraneous example(s) found in example input
('b', and 'c') Extraneous example(s) will be ommitted.
```

report (*verbose=2*)

Prints the checks in the vchecks attribute

Parameters **verbose** (*int*) – Parameter controlling how much to print by filtering for the level in each vchecks row to be less than or equal to verbose. Defaults to 2 (print only converted *Warn* and *Err* checks)

Returns None

Examples

```
>>> v = Validation("Testing Tests")
>>> v._add_condition(pd.Series([False, False, False]),  Passing("Passed test
↪"), Err("Failed test"))
>>> v._add_condition(pd.Series([False, False, False]),  Passing("Passed test 2
↪"), Err("Failed test"))
>>> v._add_condition(pd.Series([False, False, True]),  Passing("Passed test"),
↪ Err("Error test"))
>>> v._add_condition(pd.Series([False, False, True]),  Passing("Passed test"),
↪ Warn("Warn test"))
>>> v._add_condition(pd.Series([False, False, True]),  Passing(""), Suggest(
↪"Suggest test"))
>>> v.report(verbose=1)
Validating Testing Tests . . .
<BLANKLINE>
 ERRORS
[!]      Error test
>>> v.report(verbose=4)
Validating Testing Tests . . .
<BLANKLINE>
 CHECKS PASSED
[X]      Passed test
[X]      Passed test 2
<BLANKLINE>
 ERRORS
[!]      Error test
<BLANKLINE>
 SUGGESTIONS
[i]      Suggest test
<BLANKLINE>
 WARNINGS
[?]      Warn test
```

class validation.**Warn**(*message*)

Bases: *validation.VCheck*

VCheck subclass representing a problem in data validation that can be fixed in place, but would otherwise prevent validation.

Examples

```
>>> Warn("This is a data validation warning").expand()
Tier                                  Warning
Bullet                                    [?]
Level                                       2
Title                                WARNINGS
Message    This is a data validation warning
dtype: object
```

bullet()

abstract property, must be overriden. Should be a str, representing a bullet point

level()

abstract property, must be overriden. Should be int ,representing VCheck tier

tier()
abstract property, must be overriden. Should be str, representing name of VCheck tier

title()
abstract property, must be overriden. Should be str, representing title of VCheck type

validation.**report_row**(*flag_where*)
A helper method to return an english explanation of what rows have been flagged with a failed validation check.

Parameters **flag_where** (*Pandas Series*) – boolean Pandas Series representing failed validation checks.

Returns a string reporting the index of the flagged rows

Return type str

Examples

```
>>> report_row(pd.Series([True, True, False, True, False]))
'#0, #1, and #3'
```

CHAPTER 5

mappings module

Defines MapCondition class and its subclasses, each represent a single condition that uses a relationship to transform raw data into a boolean column while preserving the NA values.

class mappings.**BetweenCondition**(*condition_row*)

Bases: *mappings.NumMapCondition*

Subclass of NumMapCondition that overrides __init__ and .check() methods for the *between* relationship

Variables

- **low** (*float*) – a float representing the lowest acceptable value (incl)
- **high** (*float*) – a float representing the highest acceptable value (incl)

possible_values()

generate a non-exhaustive list of possible values implied by the condition

Args: None

Returns a list of integers between self.low - 1 and self.high + 2

Return type list

Examples

```
>>> BetweenCondition({"Condition" : "3 to 5",  "New Column Name" : "test new␣
↪column name",  "Relationship" : "between",  "Prerequisite" : None,  "Source␣
↪Column ID" : "source_test_2"}  ).possible_values()
[2.0, 3.0, 4.0, 5.0, 6.0]
```

class mappings.**ContainsCondition**(*condition_row*)

Bases: *mappings.StrMapCondition*

Subclass of StrMapCondition that overrides ._run_check() method for the *contains* relationship

class mappings.**MapCondition**(*condition_row*)

Bases: abc.ABC

Abstract class representing a single mapped condition in the mapping data, which gives instructions to transform the raw input data into the form needed for a VA instrument. The main configuration class is composed of these.

Variables

- **name** (*str*) – the name of the new column to be created
- **relationship** (*str*) – the relationship of the input data to the condition Should be one of "ge" (greater than or equal to), "gt" (greater than), "le" (less than or equal to), "lt" (less than), "eq" (equal to), "ne" (not equal to), "contains" (if string contains) or "between" (between the two numbers, inclusive).
- **preq_column** (*str or None*) – name of the pre-requisite column if it exists, or *None* if no pre-requisite
- **source** (*str*) – the name of the column to be checked

check (*prepared_data*)

Checks the condition against dataframe. Do not check NAs, just add them back afterward.

Parameters **prepared_data** (*Pandas DataFrame*) – a dataframe containing a created column with the name specified in self.source_dtype

Returns returns a boolean array where the condition is met (as float)

Return type Array

Examples

```
>>> test_df = pd.DataFrame({"source_test_str": ["test condition", "test␣
↪condition 2", np.nan], "source_test_num": [4, 5, np.nan]})
>>> StrMapCondition({"Condition" : "test condition", "New Column Name" :
↪"test new column name", "Relationship" : "eq", "Prerequisite" : None,
↪"Source Column ID" : "source_test"}).check(test_df)
array([ 1., 0., nan])
```

```
>>> NumMapCondition({"Condition" : 4.5, "New Column Name" : "test new column␣
↪name", "Relationship" : "ge", "Prerequisite" : None, "Source Column ID" :
↪"source_test"}).check(test_df)
array([ 0., 1., nan])
```

check_prereq (*transformed_data*)

checks for pre-req column status; if there is no pre-req, returns true, else looks up values of pre-req column from transformed_data

Parameters **transformed_data** (*Pandas DataFrame*) – the new dataframe being created, which contains any pre-req columns

Returns

representing whether pre-req is satisfied

Return type boolean or boolean pd.series

Examples

```
>>> test_df = pd.DataFrame({"preq_one": np.repeat(True,5),  "preq_two": np.
↪repeat(False, 5)})
```

If there is no pre-req, simply returns True (1) Pandas can interpret this in boolean indexing.

```
>>> NumMapCondition({"Condition" : 4.5,  "New Column Name" : "test new column␣
↪name",  "Relationship" : "ge",  "Prerequisite" : None,  "Source Column ID"␣
↪: "source_test"}  ).check_prereq(test_df)
1
```

If there is a pre-req, then returns the value of transformed_data with that column.

```
>>> NumMapCondition({"Condition" : 4.5,  "New Column Name" : "test new column␣
↪name",  "Relationship" : "ge",  "Prerequisite" : "preq_one",  "Source␣
↪Column ID" : "source_test"}  ).check_prereq(test_df)
0    True
1    True
2    True
3    True
4    True
Name: preq_one, dtype: bool
```

```
>>> NumMapCondition({"Condition" : 4.5,  "New Column Name" : "test new column␣
↪name",  "Relationship" : "ge",  "Prerequisite" : "preq_two",  "Source␣
↪Column ID" : "source_test"}  ).check_prereq(test_df)
0    False
1    False
2    False
3    False
4    False
Name: preq_two, dtype: bool
```

describe()
just a wrapper for the __str__ function

factory(*condition=''*)
static class factory method, which determines which subclass to return

Parameters

- **relationship** (*str*) – a relationship in (gt, ge, lt, le, ne, eq, contains, between) that represents a comparison to be made to the raw data
- **condition** (*str or int*) – the condition being matched. if relationship is ambiguous, then this determins if condition is numerical or string. Defaults to empty string.

Returns returns specific subclass that corresponds to the correct relationship

Return type *MapCondition*

Examples

```
>>> MapCondition.factory("ge") #doctest: +ELLIPSIS
<class '...NumMapCondition'>
```

```
>>> MapCondition.factory("eq", 0) #doctest: +ELLIPSIS
<class '...NumMapCondition'>
```

```
>>> MapCondition.factory("eq") #doctest: +ELLIPSIS
<class '...StrMapCondition'>
```

```
>>> MapCondition.factory("contains") #doctest: +ELLIPSIS
<class '...ContainsCondition'>
```

```
>>> MapCondition.factory("between") #doctest: +ELLIPSIS
<class '...BetweenCondition'>
```

```
>>> MapCondition.factory("eqq") #doctest: +ELLIPSIS
Traceback (most recent call last):
AssertionError: No defined Condition class for eqq type
```

possible_values
abstract method stub generate a non-exhaustive list possible values implied by condition

prepare_data(*raw_data*)
prepares raw_data by ensuring dtypes are correct for each comparison

Parameters **raw_data** (*dataframe*) – a data frame containing raw data, including the column given in self.source_name.

Returns the column in *raw_data* named in self.source_name, with the attribute self.prep_func applied to it.

Return type Pandas Series

class mappings.**NumMapCondition**(*condition_row*, *cast_cond=True*)
Bases: *mappings.MapCondition*

class representing a numerical condition, inherits from MapCondition

Variables

- **source_dtype** (*str*) – a copy of the instance attribute self.source_name with "_num" appended, to represent the expected dtype
- **prep_func** (*function*) – class attr, a function to apply before making a numerical-based comparison. pd.to_numeric() coerces non-number data to NaN.

possible_values()
generate a non-exhaustive list of possible values implied by condition

Args: None

Returns

list containing range of possible values. If a greater than relationship, the list will include ints from self.condition + 1 to self.condition*2. If a less than relationship, it will include values from 0 to self.condition. If the condition includes "equal to", then self.condition will also be included.

Return type list

Examples

```
>>> NumMapCondition({"Condition" : 3,  "New Column Name" : "test new name",
↪"Relationship" : "ge",  "Prerequisite" : None,  "Source Column ID" :
↪"source_test"}).possible_values()
[4.0, 5.0, 3.0]
```

(continues on next page)

(continued from previous page)

```
>>> NumMapCondition({"Condition" : 3,  "New Column Name" : "test new name",
↪"Relationship" : "lt",  "Prerequisite" : None,  "Source Column ID" :
↪"source_test"}).possible_values()
[0.0, 1.0, 2.0]
```

class mappings.**StrMapCondition**(*condition_row*)

Bases: *mappings.MapCondition*

class representing a str condition, inherits from MapCondition

Variables

- **source_dtype** (*str*) – instance attribute, a copy of the instance attribute self.source_name with "_str" appended, to represent the expected dtype
- **prep_func** (*function*) – class attribute, a function to apply before making a string-based comparison. It preserves null values but changes all else to str.

possible_values()

generate a non-exhaustive list possible values implied by condition

Args: None

Returns

list containing 4 possible values (empty string, NA, None, and the self.condition attribute) that might be expected by this condition

Return type list

Examples

```
>>> StrMapCondition({"Condition" : "test condition",  "New Column Name" :
↪"test new column name",  "Relationship" : "eq",  "Prerequisite" : None,
↪"Source Column ID" : "source_test"}  ).possible_values()
['', nan, None, 'test condition', 'yes', 'no', 'dk', 'ref']
```

CHAPTER 6

pyCrossVA

6.1 Simple Usage - Python

The simplest way to get started with CrossVA is to invoke the `transform` function with a default mapping, and the path to a csv containing your raw verbal autopsy data.

```
from pycrossva.transform import transform

transform(("2016WHOv151", "InterVA4"), "path/to/data.csv")
```

You can also call the transform function on a Pandas DataFrame, if you wanted to read in and process the data before calling the function.

```
from pycrossva.transform import transform

input_data = pd.read_csv("path/to/data.csv")
input_data = some_special_function(input_data)
final_data = transform(("2016WHOv151", "InterVA4"), input_data)
```

The transform function returns a Pandas DataFrame object. To write the Pandas DataFrame to a csv, you can do:

```
final_data.to_csv("filename.csv")
```

pyCrossVA is a python package for transforming verbal autopsy data collected using the 2016 WHO VA instrument (v1.5.1, or v1.4.1), 2012 WHO VA instrument, and the PHRMC short questionnaire into a format suitable for openVA.

The flagship function of this package is the transform() function, which prepares raw data for use in a verbal autopsy algorithm. The user can either choose to use a default mapping, or create a custom one of their own design. The

default mappings are listed in *Currently Supported* and can be invoked by passing in a tuple as the mapping argument in `("input", "output")` format.

6.2 Command Line

pycrossva also contains a command line tool, *pycrossva-transform* that acts as a wrapper for the *transform* python function in the pycrossva package. Once you have installed pycrossva, you can run this from the command line in order to process verbal autopsy data without having to touch python code. If you have multiple input files to process from the same input type (or source format) to the same output type (or algorithm), you can run them all in a single command.

If no destination (–dst) is specified, the default behavior will be to write the resulting data to a csv in the current working directory with a name in the pattern of "output_type_from_src_mmddyy", where mmddyy is the current date. If *dst* is a directory, then the result file will still have the default name. If *dst* ends in '.csv' but multiple input files are given, then the output files will be written to dst_1.csv, dst_2.csv, etc.

***pycrossva-transform* takes 3 positional arguments:**

- *input_type*: source type of the input data (the special input type of 'AUTODETECT' specifies that the type should be detected automatically if possible)
- *output_type*: format of output data (which algorithm the data should be prepared for)
- *src*: filepath to the input data - can take multiple arguments, separated by a space

Examples:

```
$ pycrossva-transform 2012WHO InterVA4 path/to/mydata.csv
2012WHO 'path/to/my/data.csv' data prepared for InterVA4 and written to csv at 'my/
↪current/directory/InterVA4_from_mydata_042319.csv'

$ pycrossva-transform 2012WHO InterVA4 path/to/mydata1.csv path/to/another/data2.csv -
↪-dst outputfolder
2012WHO 'path/to/mydata1.csv' data prepared for InterVA4 and written to csv at
↪'outputfolder/InterVA4_from_mydata1_042319.csv'
2012WHO 'path/to/another/data2.csv' data prepared for InterVA4 and written to csv at
↪'outputfolder/InterVA4_from_data2_042319.csv'

$ pycrossva-transform 2012WHO InterVA4 path/to/mydata1.csv path/to/another/data2.csv -
↪-dst outputfolder/results.csv
2012WHO 'path/to/mydata1.csv' data prepared for InterVA4 and written to csv at
↪'outputfolder/results_1.csv'
2012WHO 'path/to/another/data2.csv' data prepared for InterVA4 and written to csv at
↪'outputfolder/results_2.csv'

$ pycrossva-transform AUTODETECT InterVA4 path/to/mydata.csv
Detected input type: 2012WHO
2012WHO 'path/to/my/data.csv' data prepared for InterVA4 and written to csv at 'my/
↪current/directory/InterVA4_from_mydata_042319.csv'
```

6.3 Running Tests

To run unit tests, first make sure all requirements are installed

```
pip install -r requirements.txt
```

Also make sure that pytest is installed

    pip install pytest

Finally, run the tests

    python setup.py install && cd pycrossva && python -m pytest –doctest-modules

6.4 Currently Supported

6.4.1 Inputs

- 2021 WHO Questionnaire from ODK export
- 2016 WHO Questionnaire from ODK export, v1.5.1
- 2016 WHO Questionnaire from ODK export, v1.4.1
- 2012 WHO Questionnaire from ODK export
- PHRMC Shortened Questionnaire

6.4.2 Outputs

- InSilicoVA
- InterVA4
- InterVA5

6.5 Roadmap

This is an alpha version of package functionality, with only limited support.

6.5.1 Expanding outputs

One component of moving to a production version will be to offer additional mapping files to support more output formats. The package currently supports mapping to the InterVA4 and InSilicoVA format.

The following is a list of additional outputs for other algorithms to be supported in future versions:

- Tariff
- Tariff 2.0

6.6 Style

This package was written using google style guide for Python and PEP8 standards. Tests have been written using doctest.

6.7 Background

6.7.1 About Verbal Autopsy

From Wikipedia:

> A verbal autopsy (VA) is a method of gathering health information about a deceased individual to determine his or her cause of death. Health information and a description of events prior to death are acquired from conversations or interviews with a person or persons familiar with the deceased and analyzed by health professional or computer algorithms to assign a probable cause of death.
>
> Verbal autopsy is used in settings where most deaths are undocumented. Estimates suggest a majority of the 60 million annual global deaths occur without medical attention or official medical certification of the cause of death. The VA method attempts to establish causes of death for previously undocumented subjects, allowing scientists to analyze disease patterns and direct public health policy decisions.
>
> Noteworthy uses of the verbal autopsy method include the Million Death Study in India, China's national program to document causes of death in rural areas, and the Global Burden of Disease Study 2010.

6.8 License

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

R

S

T

V

W