Package survey\_stats

Ducumentation

# Introduction

survey\_stats is a Python package that provides a set of statistical tools for survey studies. This package includes some tools for data processing and estimation of observation weights, linear and logistic regressions, as well as tree regression and classification that are adapted for both numerical and categorical variables.

# data\_processing

## Definition

data\_process module includes classes of Data\_Types, Data, and Sample.

from survey\_stats.data\_process import Data\_Types, Data, Sample

## Data\_Types class

**Data\_Types** class is a data class that includes the types of data that will be used in statistical modeling. These types of data include cross-sectional ('cross'), time series ('time') and panel data ('panel'). Only cross-sectional data tools are currently available.

## Data

### Definition

**Data** class consists of two features:

* **type** contains values of the Data\_Types class that can be entered with just one string such as 'cross', 'time', or 'panel', or the Data\_Type class can be used for example Data\_Types.cross, Data\_Types.time, or Data\_Types.panel.
* **values** are entered as a dictionary in this format:

{'variable\_name':{'index':'value', …}, …}

For example:

values = {

    'name': {1: 'Cyrus', 2: 'Mandana', 3: 'Atossa'},

    'age': {1: 32, 2: 65, 3: 40},

    'sex': {1:'male', 2: 'female', 3: 'female'}

}

data = Data(Data\_Types.cross, values)

# or

data = Data('cross', values)

**Note:**

1- Dictionary keys are the names of variables, and must be of string type (str).

2- The values of each variable should be entered as a dictionary, the keys of this dictionary are index (can be types of both string and number) and its values are variable values for each index. For example, in cross sectional data, the index is a subject number, and in time series data, can be year, season, month, or day.

3- index should be the same for all variables. If it is not the same, you will encounter the following error:

ValueError: Error! index aren't same for all variables.

4- If a variable does not have a value for an index, must be set the value of Numpy.nan for that index.

5- This data can also be entered through the Pandas package by converting the DataFrame to a dictionary.

import pandas as pd

values = pd.read\_csv('data.csv').to\_dict()

data = Data(Data\_Types.cross, values)

### Methods

#### \_\_str\_\_()

This function provides an overview of the data, which is a table that shows middle columns and rows with ... .

**Input:**

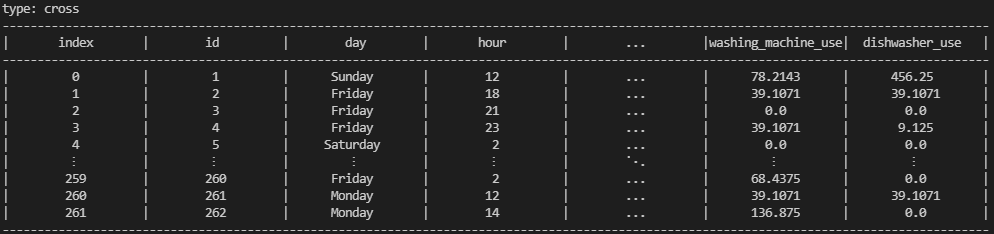
print(data)

or

data\_str = data.\_\_str\_\_()

print(data\_str)

**output:**



#### variables()

This function lists the names of variables as a list.

**Input:**

variables\_name = data.variables()

print(variables\_name)

**Output:**

['id', 'day', 'hour', 'w1', 'w2', 'sex', 'age', 'city', 'position\_in\_family', 'marital\_status', 'children', 'children\_welfare', 'religion', 'faith', 'education', 'job', 'healthy', 'income', 'pollution', 'planet\_age', 'causes\_global\_warming', 'effects\_global\_warming', 'environment\_attitudes\_validity', 'env\_pleasure', 'env\_government', 'env\_avtivist', 'env\_population', 'env\_total', 'personality\_level1', 'personality\_level2', 'personality\_level3', 'risk\_aversion', 'expected\_age', 'life\_satisfaction', 'life\_control', 'social\_capital', 'energy\_use', 'pc\_use', 'laptop\_use', 'mobile\_use', 'tv\_use', 'game\_console\_use', 'Hairdryer\_use', 'microwave\_use', 'iron\_use', 'tea\_maker\_use', 'juicer\_use', 'vacuum\_use', 'steamer\_use', 'washing\_machine\_use', 'dishwasher\_use']

Of course, it was also possible to receive it directly using the dictionary functions:

variables\_name = list(data.values.keys())

#### index(without\_checking:bool=True)

This function gives a list of indexes. If *without\_checking* is False, all variables are checked to have the same indexes, otherwise it will give an error message..

**Input:**

index = data.index()

print(index)

**Output:**

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261]

**Note:** This function checks if the indexes of different variables are not the same, returns an error with this ValueError:

ValueError: Error! index aren't same for all variables.

#### set\_index(var:str, drop\_var:bool=True)

This function sets the index values equal to the values of the 'var' variable. By default, this variable is removed from the list of variables. If you do not want to delete it, set the drop\_var equals False.

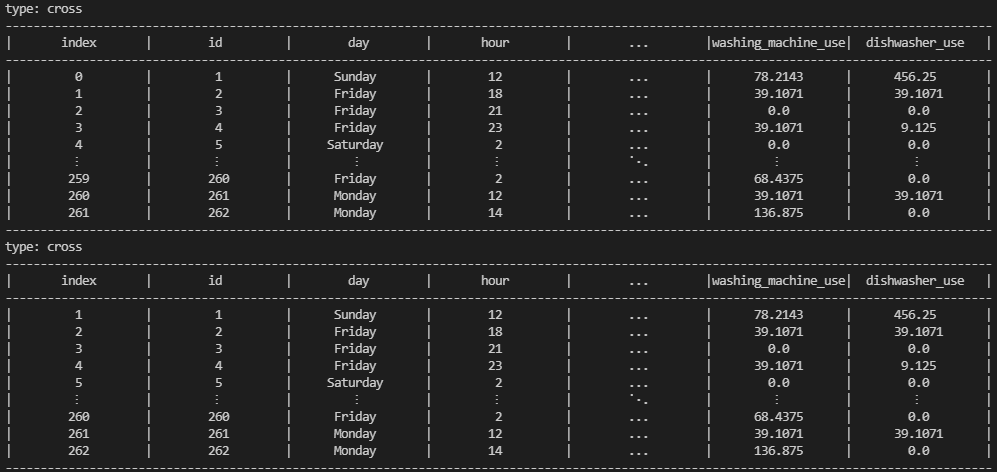
**Input:**

print(data)

data.set\_index('id', False)

print(data)

**Output:**



#### set\_names(new\_names:list[str]=[], old\_names:list[str]=[])

This function changes the name of the list of variables to new names.

**Input:**

print('Before rename:','\n',data.variables())

data.set\_names(['w1','w2'], ['weights1', 'weights2'])

print('\nAfter rename:','\n',data.variables())

**Output:**

Before rename:

['id', 'day', 'hour', 'w1', 'w2', 'sex', 'age', 'city', 'position\_in\_family', 'marital\_status', 'children', 'children\_welfare', 'religion', 'faith', 'education', 'job', 'healthy', 'income', 'pollution', 'planet\_age', 'causes\_global\_warming', 'effects\_global\_warming', 'environment\_attitudes\_validity', 'env\_pleasure', 'env\_government', 'env\_avtivist', 'env\_population', 'env\_total', 'personality\_level1', 'personality\_level2', 'personality\_level3', 'risk\_aversion', 'expected\_age', 'life\_satisfaction', 'life\_control', 'social\_capital', 'energy\_use', 'pc\_use', 'laptop\_use', 'mobile\_use', 'tv\_use', 'game\_console\_use', 'Hairdryer\_use', 'microwave\_use', 'iron\_use', 'tea\_maker\_use', 'juicer\_use', 'vacuum\_use', 'steamer\_use', 'washing\_machine\_use', 'dishwasher\_use']

After rename:

['id', 'day', 'hour', 'w1', 'w2', 'sex', 'age', 'city', 'position\_in\_family', 'marital\_status', 'children', 'children\_welfare', 'religion', 'faith', 'education', 'job', 'healthy', 'income', 'pollution', 'planet\_age', 'causes\_global\_warming', 'effects\_global\_warming', 'environment\_attitudes\_validity', 'env\_pleasure', 'env\_government', 'env\_avtivist', 'env\_population', 'env\_total', 'personality\_level1', 'personality\_level2', 'personality\_level3', 'risk\_aversion', 'expected\_age', 'life\_satisfaction', 'life\_control', 'social\_capital', 'energy\_use', 'pc\_use', 'laptop\_use', 'mobile\_use', 'tv\_use', 'game\_console\_use', 'Hairdryer\_use', 'microwave\_use', 'iron\_use', 'tea\_maker\_use', 'juicer\_use', 'vacuum\_use', 'steamer\_use', 'washing\_machine\_use', 'dishwasher\_use']

#### select\_variables(vars:list[str]=[])

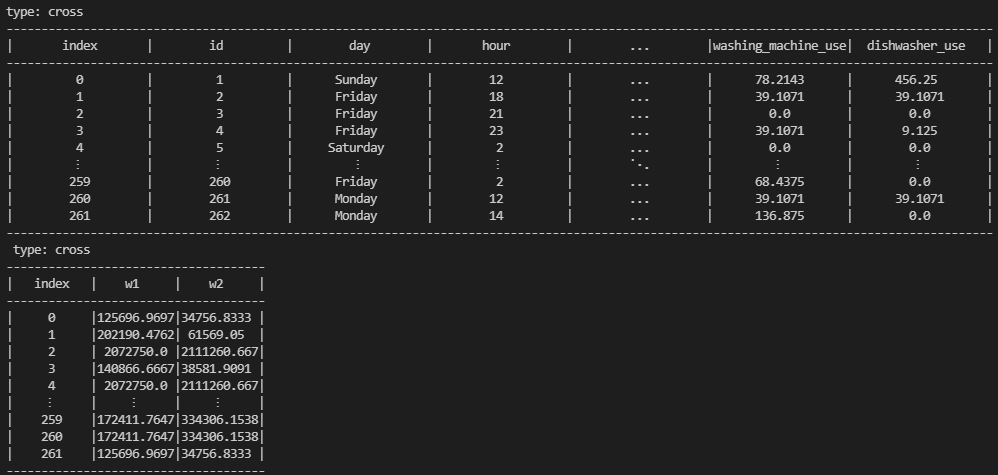
This function separates the values of a number of selected variables from the data set and returns them as a new data set.

**Input:**

new\_data = data.select\_variables(['w1','w2'])

print(data,'\n',new\_data)

**Output:**



#### select\_index(index:list)

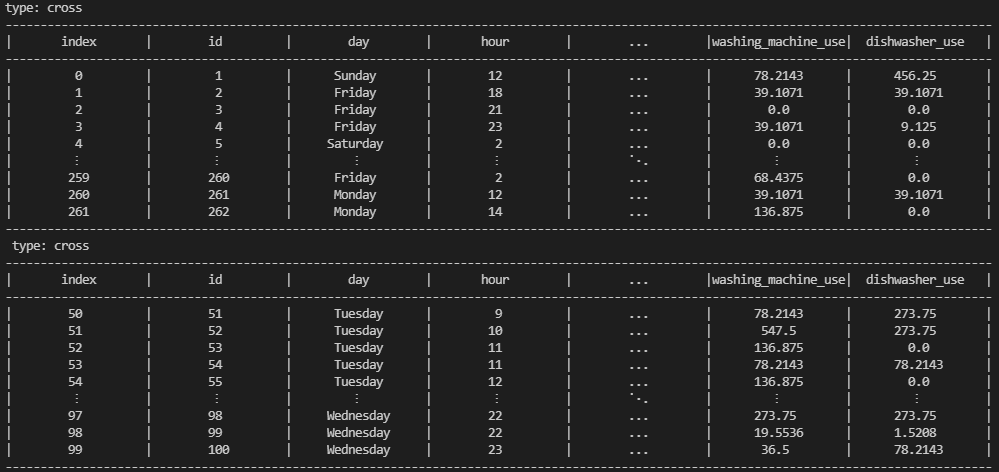
This function returns the values of all variables for the list of selected indexes.

**Input:**

new\_data = data.select\_index(range(50,100))

print(data,'\n',new\_data)

**Output:**



#### convert(new\_names:list[str]=[] , vars:list[str]=[], formula:str='', based\_value:float=0, based\_var:str='', add\_to\_data:bool=False)

Using this function, new variables can be defined based on existing variables.

new\_names are optional, if not defined, the program itself will choose a name based on the names of variables (vars), formula, based\_var and based\_value.

|  |  |  |  |
| --- | --- | --- | --- |
| Formula value | Data\_Type | Var\_Type | Math Function |
| growth | only time | numeric |  |
| lags | only time | numeric |  |
| dif | only time | numeric |  |
| Log | all types | numeric |  |
| + | all types | numeric |  |
| - | all types | numeric |  |
| .- | all types | numeric |  |
| \* | all types | numeric |  |
| / | all types | numeric |  |
| ./ | all types | numeric |  |
| ^ | all types | numeric |  |
| .^ | all types | numeric |  |
| merge\_value | all types | categorical |  |

**Example 1:** for a time data:

**Input:**

values = {

    'year': {1390: 1390, 1391: 1391, 1392: 1392, 1393: 1393, 1394: 1394, 1395: 1395, 1396: 1396, 1397: 1397, 1398: 1398, 1399: 1399, 1400: 1400},

    'income': {1390: 125, 1391: 200, 1392: 210, 1393: 300, 1394: 350, 1395: 380, 1396: 395, 1397: 410, 1398: 425, 1399: 443, 1400: 500},

    'price': {1390: 100, 1391: 170, 1392: 200, 1393: 235, 1394: 280, 1395: 320, 1396: 354, 1397: 375, 1398: 405, 1399: 480, 1400: 535}

}

data = Data(Data\_Types.time, values)

print(data)

new\_data = data.convert(formula='growth', vars=[

                        'income', 'price'], based\_value=2)

print(new\_data)

new\_data = data.convert(formula='lags', vars=[

                        'income', 'price'], based\_value=2)

print(new\_data)

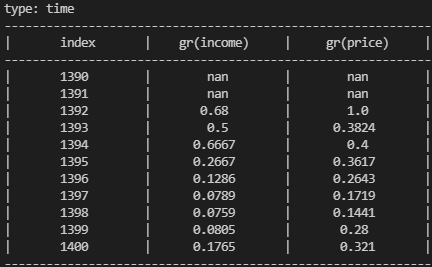
new\_data = data.convert(formula='dif', vars=[

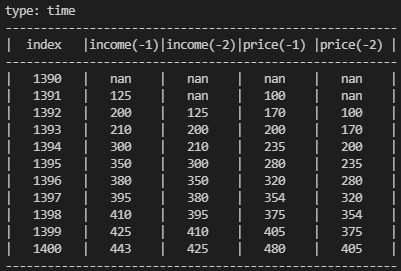
                        'income', 'price'], based\_value=2)

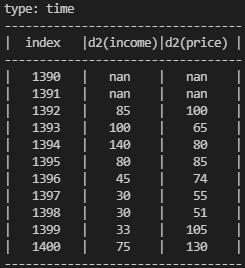
print(new\_data)

**Output:**









**Example 2:** for a cross-section data:

**Input:**

values = {

    'year': {0: 1352, 1: 1330, 2: 1359, 3: 1365, 4: 1343, 5: 1372, 6: 1364, 7: 1357, 8: 1341, 9: 1375, 10: 1367},

    'height': {0: 170, 1: 160, 2: 143, 3: 190, 4: 135, 5: 145, 6: 130, 7: 185, 8: 165, 9: 175, 10: 180},

    'weight': {0: 170, 1: 160, 2: 143, 3: 190, 4: 135, 5: 145, 6: 130, 7: 185, 8: 165, 9: 175, 10: 180}

}

data = Data(Data\_Types.cross, values)

print('data'.center(80,'-'))

print(data)

print('add age'.center(80,'-'))

data.convert(new\_names=['age'], formula='+', vars=['year'], based\_value=-1400, add\_to\_data=True)

data.convert(new\_names=['age'], vars=['age'],

             formula='\*', based\_value=-1, add\_to\_data=True)         # the new variable replaces the old variable

print(data)

print('BMI'.center(80,'-'))

data.convert(formula='^',

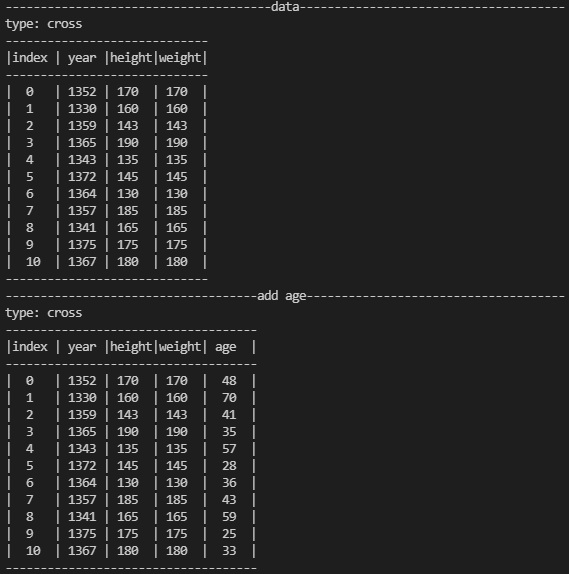
             vars=['height'], based\_value=2, add\_to\_data=True)

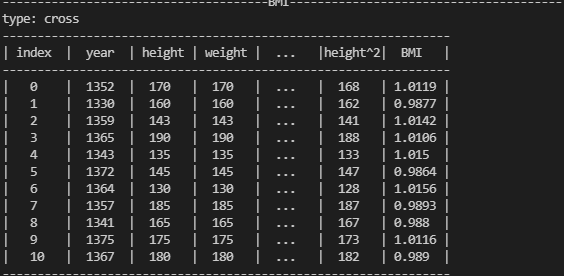
data.convert(new\_names=['BMI'], formula='/', vars=[

                        'weight'], based\_var='height^2', add\_to\_data=True)

print(data)

**Output:**





#### drop(var\_names:list[str])

Remove variables from data

**Input:**

values = {

    'year': {0: 1352, 1: 1330, 2: 1359, 3: 1365, 4: 1343, 5: 1372, 6: 1364, 7: 1357, 8: 1341, 9: 1375, 10: 1367},

    'height': {0: 170, 1: 160, 2: 143, 3: 190, 4: 135, 5: 145, 6: 130, 7: 185, 8: 165, 9: 175, 10: 180},

    'weight': {0: 170, 1: 160, 2: 143, 3: 190, 4: 135, 5: 145, 6: 130, 7: 185, 8: 165, 9: 175, 10: 180}

}

data = Data(Data\_Types.cross, values)

print('data'.center(80,'-'))

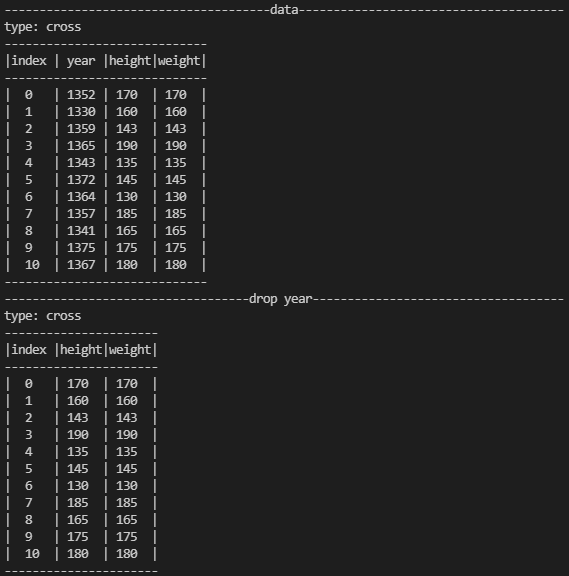
print(data)

print('drop year'.center(80,'-'))

data.drop(['year'])

print(data)

**Output:**



#### add\_a\_dummy(condition:list[tuple], add\_to\_data:bool=False)

Creates a dummy variable based on a condition.

**Condition format:**

[('var\_name\_1','operator\_1','value\_1'),( 'var\_name\_2','operator\_2','value\_2'), …]

Means:

('var\_name\_1' 'operator\_1' 'value\_1') and ('var\_name\_2' 'operator\_2' 'value\_2') and …

**Input:**

values = {

    'year': {0: 1352, 1: 1330, 2: 1359, 3: 1365, 4: 1343, 5: 1372, 6: 1364, 7: 1357, 8: 1341, 9: 1375, 10: 1367},

    'height': {0: 170, 1: 160, 2: 143, 3: 190, 4: 135, 5: 145, 6: 130, 7: 185, 8: 165, 9: 175, 10: 180},

    'weight': {0: 170, 1: 160, 2: 143, 3: 190, 4: 135, 5: 145, 6: 130, 7: 185, 8: 165, 9: 175, 10: 180}

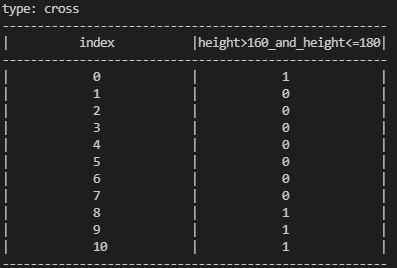
}

data = Data(Data\_Types.cross, values)

dummy = data.add\_a\_dummy([['height', '>', 160], ['height', '<=', 180]])

print(dummy)

**Output:**



#### add\_dummies(conditions:list[list[tuple]], add\_to\_data:bool=False)

Creates a number of dummy variables based on a condition.

**Condition format:**

[

[('var\_name\_1\_1','operator\_1\_1','value\_1\_1'),( 'var\_name\_2\_1','operator\_2\_1','value\_2\_1'), …],

[('var\_name\_1\_2','operator\_1\_2','value\_1\_2'),( 'var\_name\_2\_2','operator\_2\_2','value\_2\_2'), …],

…

]

**Input:**

values = {

    'year': {0: 1352, 1: 1330, 2: 1359, 3: 1365, 4: 1343, 5: 1372, 6: 1364, 7: 1357, 8: 1341, 9: 1375, 10: 1367},

    'height': {0: 170, 1: 160, 2: 143, 3: 190, 4: 135, 5: 145, 6: 130, 7: 185, 8: 165, 9: 175, 10: 180},

    'weight': {0: 70, 1: 160, 2: 43, 3: 190, 4: 35, 5: 45, 6: 30, 7: 85, 8: 65, 9: 75, 10: 80}

}

data = Data(Data\_Types.cross, values)

dummy = data.add\_dummies([

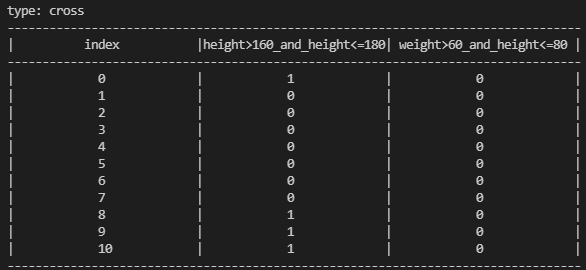
    [('height', '>', 160), ('height', '<=', 180)],

    [('weight', '>', 60), ('height', '<=', 80)]

    ])

print(dummy)

**Output:**



#### dropna(vars:list[str]=[])

Indexes in which at least one of the input variables has the value of Numpy.nan, are romved for all variables in the data.

if vars == [] then vars will be equal to all variables in data.

**Input:**

import numpy as np

values = {

    'year': {0: 1352, 1: 1330, 2: np.nan, 3: 1365, 4: 1343, 5: 1372, 6: 1364, 7: 1357, 8: 1341, 9: 1375, 10: 1367},

    'height': {0: 170, 1: 160, 2: 143, 3: 190, 4: np.nan, 5: 145, 6: 130, 7: np.nan, 8: 165, 9: 175, 10: 180},

    'weight': {0: 70, 1: 160, 2: 43, 3: 190, 4: 35, 5: 45, 6: np.nan, 7: 85, 8: 65, 9: 75, 10: 80}

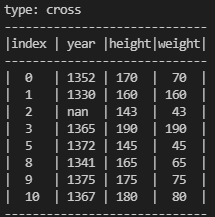
}

data = Data(Data\_Types.cross, values)

data.dropna(['height', 'weight'])

print(data)

**Output:**



#### to\_numpy(vars:list[str]=[])

It first removes all 'Numpy.nan's and then converts all numeric data to an array format in Numpy module.

**Input:**

import numpy as np

values = {

    'name': {0: 'ali', 1: 'reza', 2: np.nan, 3: 'kourosh', 4: 'tahmineh', 5: 'morteza',

             6: 'esfandiar', 7: 'mahtab', 8: 'yaas', 9: 'hadi', 10: np.nan},

    'year': {0: 1352, 1: 1330, 2: np.nan, 3: 1365, 4: 1343, 5: 1372, 6: 1364, 7: 1357, 8: 1341, 9: 1375, 10: 1367},

    'height': {0: 170, 1: 160, 2: 143, 3: 190, 4: np.nan, 5: 145, 6: 130, 7: np.nan, 8: 165, 9: 175, 10: 180},

    'weight': {0: 70, 1: 160, 2: 43, 3: 190, 4: 35, 5: 45, 6: np.nan, 7: 85, 8: 65, 9: 75, 10: 80}

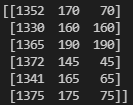
}

data = Data(Data\_Types.cross, values)

num = data.to\_numpy()

print(num)

**Output:**



#### add\_dara(new\_data:Data=None)

Merges a new data into this data.

**Input:**

import numpy as np

values = {

    'name': {0: 'ali', 1: 'reza', 2: np.nan, 3: 'kourosh', 4: 'tahmineh', 5: 'morteza'},

    'year': {0: 1352, 1: 1330, 2: np.nan, 3: 1365, 4: 1343, 5: 1372},

    'weight': {0: 70, 1: 160, 2: 43, 3: 190, 4: 35, 5: 45}

}

data = Data(Data\_Types.cross, values)

values = {

    'name': {5: 'azam', 6: 'esfandiar', 7: 'mahtab', 8: 'yaas', 9: 'hadi', 10: np.nan},

    'year': {5: 1356, 6: 1364, 7: 1357, 8: 1341, 9: 1375, 10: 1367},

    'height': {5: 145, 6: 130, 7: np.nan, 8: 165, 9: 175, 10: 180},

    'weight': {5:65, 6: 85, 7: 85, 8: 65, 9: 75, 10: 80}

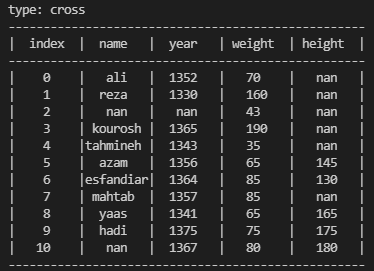
}

data\_new = Data(Data\_Types.cross, values)

data.add\_data(data\_new)

print(data)

**Output:**



#### transpose()

Reverses index (rows) and variable names (cols)

**Input:**

d = {

    'a': {'count': 25, 'weight': 0.247524752},

    'b': {'count': 5, 'weight': 0.04950495},

    'c': {'count': 26, 'weight': 0.257425743},

    'd': {'count': 12, 'weight': 0.118811881},

    'e': {'count': 33, 'weight': 0.326732673},

}

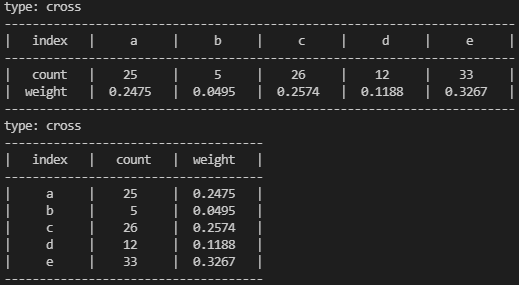
data = Data('cross', d)

print(data)

data\_t = data.transpose()

print(data\_t)

**Output:**



#### read\_csv(path\_file:str, data\_type:str='cross', na:any='', do\_faster\_instead\_of\_low\_space:bool=True)

This function reads a CSV file and converts it to a Data object.

* **na**: Specifies blank values. The default value is an empty string (“”), sometimes "NA", ".." and … are used.
* **do\_faster\_instead\_of\_low\_space:** The default value is True, meaning that a method is used to read, which is faster, but takes up more space on the RAM, if its value is False, then a method is used that uses less space on the RAM. But it will take longer.

#### to\_csv(self, path\_file:str, na:str='')

save values to a CSV file.

* **path\_file:** path and name of CSV file.
* **na**: Specifies blank values. The default value is an empty string (“”), sometimes "NA", ".." and … are used.

#### \_\_len\_\_()

Returns the number of indexes if you place a data object in the len function.

## Sample

### Definition

**Sample** class consists of 4 features , of which 2 are essential:

* **data** contains an object of Data class that contains all the population or main sample data.
* **index** is a subset of a data.index() that includes a random or non-random sample of the population or the main sample for example as a train and test samples.
* **name** is a string as a name for the sample. This name can be useful when there are several sample.
* **widghts:** The weights of the sample members are determined based on the sampling method. If the sampling is purely random, the weight of all members is equal and can be obtained by dividing the members of the population into sample members (N/n). In this case, the amount of default (=one) is sufficient for statistical inferences. But many cases, random sampling with unequal weights is used, such as cluster sampling. In this case, the weights are already known and available in the data. Therefore, in this argument, the name of the variable that contains the weights is entered.

values = {

    'name': {0: 'ali', 1: 'reza', 2: np.nan, 3: 'kourosh', 4: 'tahmineh', 5: 'morteza', 6: 'esfandiar',

                     7: 'mahtab', 8: 'yaas', 9: 'hadi', 10: np.nan},

    'year': {0: 1352, 1: 1330, 2: np.nan, 3: 1365, 4: 1343, 5: 1372, 6: 1364, 7: 1357, 8: 1341, 9: 1375, 10: 1367},

    'height': {0: 170, 1: 160, 2: 143, 3: 190, 4: np.nan, 5: 145, 6: 130, 7: np.nan, 8: 165, 9: 175, 10: 180},

    'weight': {0: 70, 1: 160, 2: 43, 3: 190, 4: 35, 5: 45, 6: 85, 7: 85, 8: 65, 9: 75, 10: 80}

}

data = Data(Data\_Types.cross, values)

s = Sample(data, [0,5,6,10])

### Methods

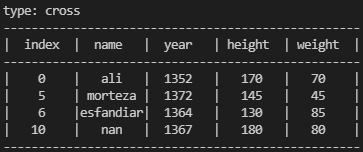
#### get\_data()

Returns data of the sample as a Data class

Input:

print(s.get\_data())

Output:



#### split\_methods

List of all methods that will be used to split the sample into two parts.

split\_methods = ['random', 'start', 'end']

**methods:**

* **random:** A specific number (fraction of the total number of members of population or main sample) is randomly selected without replacement.
* **start:** A specific number (fraction of the total number of members of population or main sample) is selected from the beginning of the data.
* **end:**A specific number (fraction of the total number of members of population or main sample) is selected from the end of the data.

**Note:** In time series data, the order of the members is important.

#### split(ratio:float, names:list, method:str='random')

In many statistical analyzes, it is necessary to split the sample into two parts: 'learn' and 'test'. The split method is the same as that described in 'split\_methods'. We need two more things to do this:

* first, what **ratio** of the sample will be allocated to the first part, the second part will include the remaining members of the sample (1-**ratio**).
* Second, what **names** are chosen for each of these two sections. **names** are entered as a list.

Input:

values = {

    'name': {0: 'ali', 1: 'reza', 2: 'mehran', 3: 'kourosh', 4: 'tahmineh', 5: 'morteza', 6: 'esfandiar',

                     7: 'mahtab', 8: 'yaas', 9: 'hadi', 10: 'razieh'},

    'year': {0: 1352, 1: 1330, 2: 1380, 3: 1365, 4: 1343, 5: 1372, 6: 1364, 7: 1357, 8: 1341, 9: 1375, 10: 1367},

    'height': {0: 170, 1: 160, 2: 143, 3: 190, 4: 110, 5: 145, 6: 130, 7: 160, 8: 165, 9: 175, 10: 180},

    'weight': {0: 70, 1: 160, 2: 43, 3: 190, 4: 35, 5: 45, 6: 85, 7: 85, 8: 65, 9: 75, 10: 80}

}

data = Data(Data\_Types.cross, values)

main\_sample = Sample(data, data.index())

train\_sample, test\_sample = main\_sample.split(0.7,['train', 'test'], 'start')

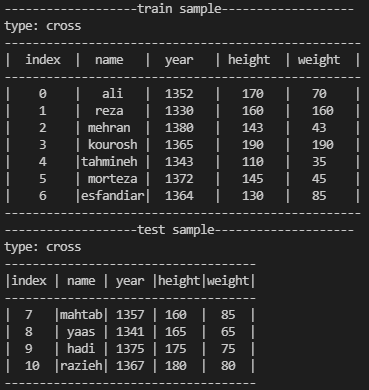
print('train sample'.center(80,'-'))

print(train\_sample.get\_data())

print('test sample'.center(80,'-'))

print(test\_sample.get\_data())

Output:



#### get\_weights(vars\_conditions:list[list], totals:list[Union[int,float]])

In many cases, probabilty sampling is very expensive. In these cases, non-probablity sampling is usually used, such as sampling via the Internet and social networks, or Mall Intercept. In this case, statistical inferences are not reliable. But research has shown that the weight of sample members based on their characteristics such as age, sex and education can increase the reliability of the results even to the extent of random sampling.

The method used here is the generalized regression estimator, GREG, based on the work of Chen, Valliant and Elliott (2001)[[1]](#footnote-1). In the general setting, there are known weights  sample, where, in the absence of true design information,  is typically set to  for all, which is equivalent to assuming a simple random-sample design. Defining the diagonal matrix of design or pseudodesign weights as, the calibrated weights  minimize an expected distance measure with respect to the design of, 



under the constraint  where  is a row vector of known population totals of from a population of size  and  is a differentiable function with respect to , strictly convex on an interval containing , and . The -distance measure  with  yields the generalized regression estimator, GREG:



The arguments of this function are:

* **vars\_conditions:** Members of population are divided into different classes according to their characteristics. Each class can be defined as a condition based on the characteristics of its members. For example, a class, include women under the age of 30 year old, could write [('sex','=', 'female'),('age','<=',30)]. Now all classes are on a list.
* totals: The number of population membrs in each class is placed into a list corresponding to the list of conditions.

**Input:**

values = {

    'name': {0: 'ali', 1: 'reza', 2: 'mehran', 3: 'kourosh', 4: 'tahmineh', 5: 'morteza', 6: 'esfandiar', 7: 'mahtab', 8: 'yaas', 9: 'hadi', 10: 'razieh'},

    'sex': {0: 'male', 1: 'male', 2: 'male', 3: 'male', 4: 'female', 5: 'male', 6: 'male', 7: 'female', 8: 'female', 9: 'male', 10: 'female'},

    'age': {0: 50, 1: 13, 2: 25, 3: 33, 4: 18, 5: 35, 6: 44, 7: 60, 8: 24, 9: 33, 10: 42},

}

data = Data(Data\_Types.cross, values)

cond = [

    [('sex','=', 'female'),('age','<=',30)],

    [('sex','=', 'female'),('age','>',30)],

    [('sex','=', 'male'),('age','<=',30)],

    [('sex','=', 'male'),('age','>',30)]

    ]

totals = [

    50,

    150,

    45,

    160

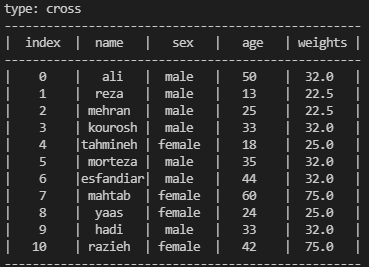
]

sample = Sample(data, data.index())

sample.get\_weights(cond, totals)

print(sample.data)

**Output:**



#### \_\_len\_\_()

Returns the number of indexes if you place a Sample object in the len function.

#### \_\_str\_\_()

Print output of get\_data() function.

# basic\_model

## Definition

basic\_model module will include classes that can be used on all models. Currently this module includes classes of Variable\_Types, Variable, and Variables. Variable\_Types Class is a data class that contains variable types. Variable class is the main class. Variables class contains functions on a list of variables. It was possible to define these functions in a Variable class as a staticmethod, but in this class it is more specialized.

## Variable\_Types

Variables are of two general types:

* **Numerical:** Variables such as age and height have numerical values. on these variables, we can be calculated statistics such as mean, median, variance, standard deviation, etc.
* **Categorical:** Variables such as gender and education are classification values. For this variable, its values and the frequency of each value in the population, which is its probability density distribution, will be important.

## Variable

### Definition

The variable consists of two properties:

* **name:** A string representing a variable name
* **type:** A string containing ‘numberic’ or ‘categorical’ that are introduced in a class of Variable\_Types.

**Note:** name is essential and default if type is ‘numeric’.

### Methods

#### \_\_str\_\_()

**Input:**

x = Variable('x', Variable\_Types.numeric)

# or

x = Variable('x', 'numeric')

print(x)

**Output:**

x:numeric

#### from\_dict(var\_dict: dict)

Users can define variables by names if they are ‘numeric’, but where the type of variable is important, the user can define the variable with a dictionary.

**Input:**

x = Variable('x')

# or

x = Variable.from\_dict({'x': Variable\_Types.numeric})

# or

x = Variable.from\_dict({'x': 'numeric'})

print(x)

**Ouput:**

x:numeric

#### from\_data(data: Data, name:str)

Sometimes you may want to rely on the values in the data. Based on this, you can define the variable by introducing the data. If the values of that variable in the data included number and non-number, the criterion is more count. If the count of numbers is more, the type of ‘numeric’ variable and otherwise ‘categorical’ is identified. If the letter is not in the list of data variables, this function returns None.

**Input:**

values = {

    'x': {0:'a', 1:'b',2:'a',3:45, 4:-5}

}

data = Data('cross', values)

x = Variable.from\_data(data, 'x')

print(x)

y = Variable.from\_data(data, 'y')

print(y)

**Output:**

x:categorical

None

#### values()

If the variable name is in the list of variables of the data, returns the list of its values without repeating, otherwise it returns None.

**Input:**

values = {

    'x': {0:'a', 1:'b',2:'a',3:'a',4:'b'}

}

data = Data('cross', values)

sample = Sample(data, data.index())

x = Variable('x', Variable\_Types.categorical)

values = x.values(sample)

print(values)

**Output:**

['a', 'b']

#### values\_set(self, sample: Sample, half\_of\_set:bool=False)

When we want to divide the values of a variable into two non-empty parts, it will be different depending on whether the variable is numeric or categorical:

* **numeric variables:** it is enough to delete the last value. In this case, these two parts are included x<=vi and x>vi.
* **categorical variables:** here we have to calculate all the subsets of the set of values, but since we want to divide the values into two parts, half of these subsets are enough. If we want to return all subsets, set the value of the half\_of\_set parameter to false.

**Input:**

values = {

    'x': {0:'a', 1:'b',2:'c',3:'a',4:'d'},

    'y': {0:12, 1:6, 2:6, 3:15, 4:3}

}

data = Data('cross', values)

sample = Sample(data, data.index())

x = Variable('x', Variable\_Types.categorical)

x\_values = x.values\_set(sample)

x\_values\_total = x.values\_set(sample, False)

print(x\_values)

print(x\_values\_total)

y = Variable('y')

y\_values = y.values\_set(sample)

print(y\_values)

**Output:**

[['a'], ['b'], ['a', 'b'], ['c'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]

[[], ['a'], ['b'], ['a', 'b'], ['c'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c'], ['d'], ['a', 'd'], ['b', 'd'], ['a', 'b', 'd'], ['c', 'd'], ['a', 'c', 'd'], ['b', 'c', 'd'], ['a', 'b', 'c', 'd']]

[3, 6, 12]

#### stats

stats is a class containing the following functions:

##### mean (sample: Sample):

Measures the mean of the variable in the sample.

##### std (sample: Sample):

Measures the standard deviation of the variable in the sample.

##### count (sample: Sample):

Measures the number of non-blank values of the variable in the sample.

##### tss (sample: Sample):

Measures the total sum of square of deviations from the mean of the variable in the sample.

##### distribution (sample: Sample):

Returns the distribution of the variable in the sample.

##### sum (sample: Sample):

Measures the sum of the values of the variables in the sample.

##### median (sample: Sample):

Measures the median of a variable in the sample.

##### mode (sample: Sample):

Measures the variable mode in the sample.

##### min (sample: Sample, k: int = 1):

Measures the minimum values of variables in the sample.

##### max (sample: Sample, k: int = 1):

Measures the maximum values of variables in the sample.

##### percentile (sample: Sample, k: float):

The k’th percentile measures the variable in the sample.

##### gini\_coef (sample: Sample):

Measures the Gini coefficient, which is an index of inequality. It is under construction.

**Input:**

values = {

    'x': {0:'a', 1:'b',2:'c',3:'a',4:'d'},

    'y': {0:12, 1:6, 2:6, 3:15, 4:3}

}

data = Data('cross', values)

sample = Sample(data)

x = Variable('x', Variable\_Types.categorical)

y = Variable('y', Variable\_Types.numeric)

print(x.stats.mode(sample))

print(y.stats.median(sample))

**Output:**

a

6

#### map(sample:Sample, old\_values:list[list|str], new\_values:list, other:str|int=np.nan, name:str='')

**map** function converts the values of a variable (old\_values) to new variable values (new\_values) based on a rule.

In general, there are two rules:

1. Return **a list of values** to **a value**. For example, the 'bachelor', 'master', and 'phd' degrees in the *'education*' variable should be converted to 'academic' values, and the 'under diploma' and 'diploma' values n the '*education'* variable should be converted to 'non-academic' values.

old\_values = [['under diploma', 'diploma'], ['bachelor', 'master', 'phd']]

new\_values = ['non-academic', 'academic']

2- For numerical variables, return a condition to a numeric value. For example, years smaller than 2012 should be returned to the amount 'before sanction', 2012 to 2015 to the amount of 'first oil sanction', 2015 to 2017 to the amount of 'JCPOA period', and the period after 2017 to the amount of 'second oil sanction'.

old\_values = ['<2012', '<=2015', '<=2017', '>2017']

new\_values = ['before sanction', 'first oil sanction',

              'JCPOA period', 'second oil sanction']

**Note 1:** Priority is given to the first condition. For example, if the list first contains '<=2017', then '<=2015', since all values '<=2015' are also '<=2017', the '<=2015' condition will never binding. But if we make the first condition '<=2015', the second condition '<=2017' will indicate values greater than 2015 and smaller than and equal to 2017 (2015<x<=2017). In the larger case (>, >=), the opposite of this case is true.

**Note 2:** 'old\_values' list includes the list of values in number 1 or the conditions in number 2. And 'new\_values' list contains values corresponding to the 'old\_values' list. So the length of the two lists must be the same.

**Input:**

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:np.nan},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:np.nan, 5:'a'},

    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

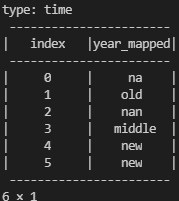
})

v1 = Variable('year', 'numeric')

v2\_data = v1.map(Sample(data), ['<1000','<1397', '<1399'],['na','old','middle'],'new')

print(v2\_data)

**Output:**



## Variables

### Definition

Receives only the list of variables.

### Methods

#### \_\_str\_\_()

**Input:**

values = {

    'x': {0:'a', 1:'b',2:'c',3:'a',4:'d'},

    'y': {0:12, 1:6, 2:6, 3:15, 4:3}

}

data = Data('cross', values)

sample = Sample(data, data.index())

x = Variable('x', Variable\_Types.categorical)

y = Variable('y')

variables = Variables([x, y])

print(variables)

**Output:**

x:categorical

y:numeric

#### from\_data(data: Data, var\_names:list = [])

Input:

values = {

    'x': {0:'a', 1:'b',2:'c',3:'a',4:'d'},

    'y': {0:12, 1:6, 2:6, 3:15, 4:3}

}

data = Data('cross', values)

variables = Variables.from\_data(data)

print(variables)

Output:

x:categorical

y:numeric

#### from\_dict(dict:dict)

**Input:**

variables = Variables.from\_dict({

    'x': Variable\_Types.categorical,

    'y': Variable\_Types.numeric

})

print(variables)

**Output:**

x:categorical

y:numeric

#### to\_variable(index:int=0)

Input:

variables = Variables.from\_dict({

    'x': Variable\_Types.categorical,

    'y': Variable\_Types.numeric

})

print(variables.to\_variable(1))

Output:

y:numeric

#### summary(sample:Sample)

Provides a summary of the statistics of the variables in the sample.

**Input:**

values = {

    'x': {0:'a', 1:'b',2:'c',3:'a',4:'d'},

    'y': {0:12, 1:6, 2:6, 3:15, 4:3},

    'z': {0:5, 1:5, 2:1, 3:np.nan, 4:13},

    'w': {0:18, 1:61, 2:65, 3:15, 4:30},

    's': {0:'ty', 1:'yu', 2:'ty', 3:'al', 4:'yu'}

}

data = Data('cross', values)

sample = Sample(data, data.index())

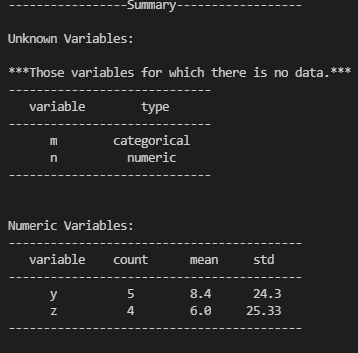
vars = Variables.from\_dict({'x':'categorical', 'y':'numeric',

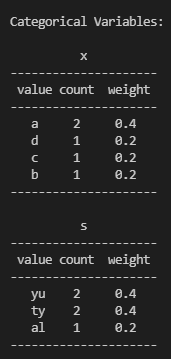
                            'z':'numeric', 's':'categorical',

                            'm':'categorical', 'n':'numeric'})

print(vars.summary(sample))

**Output:**





## Formula

### Definition

Formula is a mathematical relation. The formula class allows you to solve mathematical relationships on data, thereby either defining new variables or filtering the data set according to the values of a formula.

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:np.nan},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:np.nan, 5:'a'},

    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

})

f1 = Formula('p=a')

f2 = Formula('3\*x\*\*2+p\*x+x')

f3 = Formula('log(year)')

mathematic functions:

* time series variables:
  + lag(‘variable\_name’,’number of lags’)= var[i-lags]
  + dif(‘variable\_name’,’number of lags’)=var[i]-var[i-lags]
  + gr(‘variable\_name’,’number of lags’)=var[i]/var[i-lags]-1
* all variables:
  + log(‘variable\_name’): Napierian logarithm (loge(x))
  + exp(‘variable\_name’): Exponential function (ex)
  + statistic functions:
    - sum(‘variable\_name’): summation of variable in data.
    - count(‘variable\_name’): total non-blank values of variable in data.
    - mean(‘variable\_name’): weighted mean of variable in data.
    - std(‘variable\_name’): weighted standard deviation of variable in data.
    - min(‘variable\_name’): minimum of values of variable in data.
    - max(‘variable\_name’) maximum of values of variable in data.

### Methods

#### \_\_str\_\_()

Returns the mathematical relation of the formula.

#### calculate(data:Data, weights:str='1', skip\_collinear:bool=False)

Executes the formula on ‘*data’* and returns the output as a *Data*.

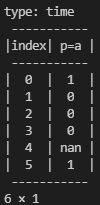
* **weights** are used for statistical functions.
* **skip\_collinear:** For categorical variables, when their value is not specified in the formula, some Dummy variable are created in the output that them values are zero and one, but if the value of skip\_collinear is set to True, one of the values removed at the output, to prevent collinearity. Because by knowing the value of other variables, the value of the final variable is determined.

**Input:**

f1 = Formula('p=a')

print(f1.calculate(data))

**Output:**

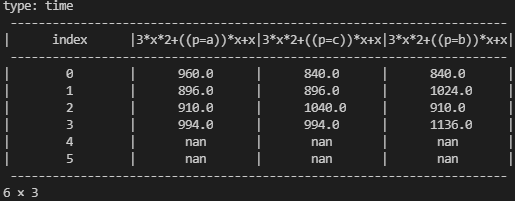


**Input:**

f2 = Formula('3\*x\*\*2+p\*x+x')

print(f2.calculate(data))

**Output:**

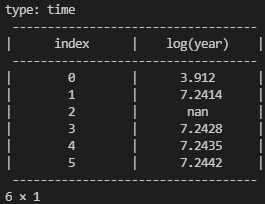


**Input:**

f3 = Formula('log(year)')

print(f3.calculate(data))

**Output:**



#### split()

split separates a formula linearly. Its output is from the Formula class.

**Input:**

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:np.nan},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:np.nan, 5:'a'},

    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

})

f = Formula('log(year)+log(x)-x\*\*2')

fs = f.split()

print(fs.formulas)

**Output:**



#### filter(value:str|int|float, data:Data)

This function filters a data set according to the calculated values of a formula.

**Input:**

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:np.nan},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:np.nan, 5:'a'},

    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

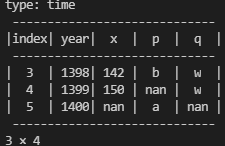
})

f = Formula('year>1397')

after1397 = f.filter(1,data)

print(after1397)

**Output:**



## Formulas

### Definition

Formulas consists of a set of formulas defined as a list of formulas.

fs = Formulas(['year>1397, p=b', '2\*x\*\*2-x'])

### Methods

#### calculate\_all(data:Data, weights:str='1', skip\_collinear:bool=False)

calculate\_all calculates all formulas and outputs an object from ‘Data’ class.

**Input:**

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:np.nan},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:np.nan, 5:'a'},

    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

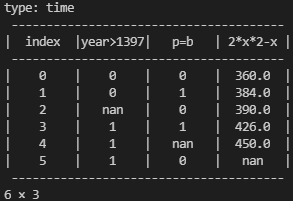
})

fs = Formulas(['year>1397’, ‘p=b', '2\*x\*\*2-x'])

calcs = fs.calculate\_all(data)

print(calcs)

**Output:**



## Table

### Definition

Table is a pivot table.

This class includes the following features:

* A **based\_variable** based on which the dataset is disaggregated.
* A list of **formulas** that are calculated for each subset of data. These formulas must start with one of the statistical functions .... If it does not include these functions, by default all of them are calculated for that formula.
* **sample**: Contains data that will be used in calculations.

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:np.nan},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:np.nan, 5:'a'},

    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

})

table = Table('p',['count(x)', 'mean(x)', 'std(x)'],Sample(data))

### Methods

#### to\_data(weights='1', skip\_collinear:bool=False)

**Input:**

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:np.nan},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:np.nan, 5:'a'},

    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

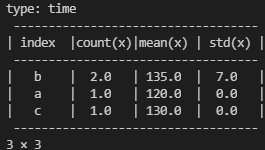
})

table = Table('p',['count(x)', 'mean(x)', 'std(x)'],Sample(data))

table\_data = table.to\_data()

print(table\_data)

**Output:**



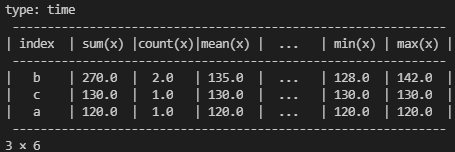
**Input:**

table = Table('p',['x'],Sample(data))

table\_data = table.to\_data()

print(table\_data)

**Output:**



#### \_\_str\_\_()

If you enter a table in the str or print function, the output of the to-data function is displayed.

#### plot(weights='1', skip\_collinear:bool=False)

This function displays the table bar chart. ‘matplotlib’ package must be installed.

**Input:**

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:np.nan},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:np.nan, 5:'a'},

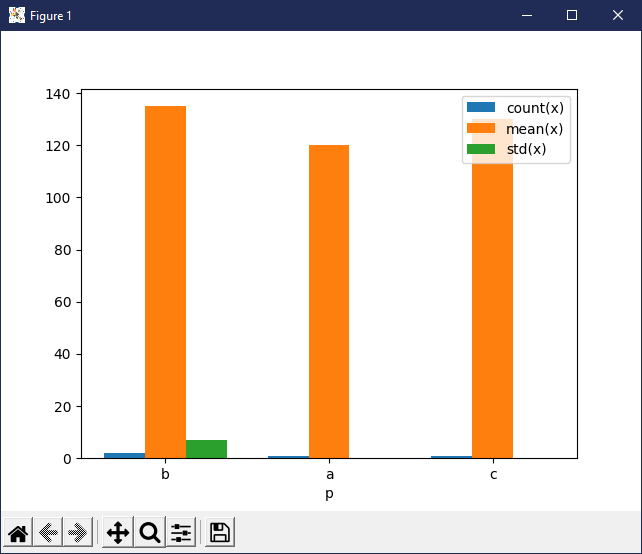
    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

})

table = Table('p',['count(x)', 'mean(x)', 'std(x)'],Sample(data))

table.plot()

**Output:**



# tree\_based\_regression

## Definition

Decision tree is one of the modelling approaches used in statistics, data mining and machine learning. Tree models where the target variable can take a discrete set of values are called **classification trees**. Decision trees where the target variable can take continuous values (typically real numbers) are called **regression trees**. Decision trees are among the most popular machine learning algorithms given their intelligibility and simplicity.

Of course, since we also have regressions with discrete variables, such as logistic regressions, so in this package we have included both regression trees and classification trees in the tree\_based\_regression module.

Currently, the sklearn package is used for the regression tree and the classification tree. But this package does not work by categorical variables, and this is very restrictive for survey researches because many of the variables in this researches are categorical. It also does not take into account the weight of the observations in its calculations. Therefore, this package has been significantly developed compared to it.

## Methods

### Definition

This class is a data class that includes the titles of regression estimation methods. There are 8 methods, 4 methods for when the dependent variable is numerical, and 4 methods for when the dependent variable is categorical.

Tree-based regression is estimated recursively, with the parent nodes being estimated first and then their children. To estimate each node, all variables and all their values are searched for the best split.

For each node, a split produces two child nodes by a variable ( is called the var\_split) and a value ( is called the value\_split), which we call left and right nodes.

* If this variable is numeric, then the **left node** contains members for which the value of this variable is less than or equal to the split value (), and the **right node** contains members for which the value of the variable is greater than the split value ().
* If this variable is categorical, then the **left node** contains members for which the value of this variable is equal to the split value (), and the **right node** contains members for which the value of the variable is not equal to the split value ().

Therefore, if the whole sample is , then the left sample and the right sample can be defined as follows:



The quality of a candidate split  is then computed using an impurity function or loss function :



is the total weight of the sample members and  and  are sum of the weight of the left and right sample. If the weights are one, the weights will be equal to the number of sample members, ,  and .

Select the parameters that minimizes the impurity



Therefore, the methods denpend on the loss function that is assumed. Traditionally, three loss functions for a categorical dependent variable and three loss functions for a numerical dependent variable are introduced. But for both of these cases, we add another function that represents the statistical significance of the mean difference between the two samples.

**Numeric criterias:**

* **Mean Squared Error (mse)** equals to OLS (ordinary Least Square) in linear regressions. This method is suitable if the dependent variable follows the normal distribution.



, 

* **Half Poisson deviance (poisson):** This method is suitable if the dependent variable follows the Poisson distribution. the Poisson distribution expresses the probability of a given number of events occurring in a fixed interval of time or space if these events occur with a known constant mean rate and independently of the time since the last event.



* **Mean Absolute Error (mae):**



 is weighted median of sample.

**Categorical criterias:** dependent variable is categorical and its values are, In this case, the probability distribution of y in the sample would be:



that 

* **gini:** 
* **entropy:** 
* **misclassification:** 

**Additional criteria:** Another method could be inspired by the notion that the S-sample is to be divided into sub-samples of S-left and S-right. If the sample members are random and independent of each other, then the statistical significance can be used as a criterion.

* **Numerical dependent variable:** If the dependent variable is numeric, the required statistic is t-statistic to compare the mean of two samples, right and left:





Since this statistic follows the T-Student distribution with the following degree of freedom



In this case, the P-Value value can be calculated.

* **Categorical dependent variable:** If the dependent variable is categorical, the required statistic is the Pearson's chi-squared statistic to compare the y-distribution in the left and right samples:



where

 = Pearson's cumulative test statistic, which asymptotically approaches a distribution.

 = the number of observations of type .

 = total number of observations

 = the expected (theoretical) count of type , asserted by the null hypothesis.

 = the number of members of the sample.

Since this statistic follows the distribution of chi-square with a degree of freedom  ( is the number of values of dependent variable), the value of P-value can be calculated.

In either case, the split that produces the most P-value will be selected.

## Model

### Definition

To estimate a tree regression, the user must first define a model. Features of a model are:

* **dep\_var:** A dependent variable that can be defined as a [Variable](#_Variable) object, or as a dictionary with 'name' key and 'type' value (-> [basic\_model](#_basic_model)->[Variable](#_Variable)->[from\_dict](#_from_dict(var_dict:_dict))), or simply as a variable name, in which case its type is considered numeric.
* **indep\_vars:** List of independent variables that can be defined list of [Variable](#_Variable)s, dictionary (-> [basic\_model](#_basic_model)->[Variable](#_Variable)->[from\_dict](#_from_dict(var_dict:_dict))), or list of variable names (in string type), in which case their types are considered numeric.
* **min\_sample:** minimum size of sample in each node. Its default value is 5.
* **min\_significant:** maximum p-value in each node. Its default value is 1. Its value is between 0 to 1.
* **method:** The model estimation method that we have described in the [Methods](#_Methods) class.

### Methods

#### \_\_str\_\_()

**Input:**

dep\_var, method = {'tv\_use':Variable\_Types.numeric}, 'p-value'

indep\_vars = {'sex':Variable\_Types.categorical, 'age':Variable\_Types.numeric,

        'city':Variable\_Types.categorical, 'position\_in\_family':Variable\_Types.categorical,

        'marital\_status':Variable\_Types.categorical,'children':Variable\_Types.numeric,

        'children\_welfare':Variable\_Types.numeric, 'religion':Variable\_Types.categorical,

        'faith':Variable\_Types.categorical, 'education':Variable\_Types.numeric,

        'job':Variable\_Types.categorical, 'healthy':Variable\_Types.numeric, 'income':Variable\_Types.numeric,

        'pollution':Variable\_Types.numeric, 'planet\_age':Variable\_Types.numeric,

        'causes\_global\_warming':Variable\_Types.categorical, 'effects\_global\_warming':Variable\_Types.numeric,

        'environment\_attitudes\_validity':Variable\_Types.numeric, 'env\_pleasure':Variable\_Types.numeric,

        'env\_government':Variable\_Types.numeric, 'env\_avtivist':Variable\_Types.numeric,

        'env\_population':Variable\_Types.numeric, 'env\_total':Variable\_Types.numeric,

        'personality\_level1':Variable\_Types.categorical,'personality\_level2':Variable\_Types.categorical,

        'personality\_level3':Variable\_Types.categorical, 'risk\_aversion':Variable\_Types.numeric,

        'expected\_age':Variable\_Types.numeric, 'life\_satisfaction':Variable\_Types.numeric,

        'life\_control':Variable\_Types.numeric, 'social\_capital':Variable\_Types.numeric}

model = tree\_based\_regression.Model(dep\_var, indep\_vars, min\_sample=25, method=method)

print(model)

**Output:**

dep\_var: 'tv\_use:numeric'

indep\_vars:

sex:categorical

age:numeric

city:categorical

position\_in\_family:categorical

marital\_status:categorical

children:numeric

children\_welfare:numeric

religion:categorical

faith:categorical

education:numeric

job:categorical

healthy:numeric

income:numeric

pollution:numeric

planet\_age:numeric

causes\_global\_warming:categorical

effects\_global\_warming:numeric

environment\_attitudes\_validity:numeric

env\_pleasure:numeric

env\_government:numeric

env\_avtivist:numeric

env\_population:numeric

env\_total:numeric

personality\_level1:categorical

personality\_level2:categorical

personality\_level3:categorical

risk\_aversion:numeric

expected\_age:numeric

life\_satisfaction:numeric

life\_control:numeric

social\_capital:numeric

min\_sample: 25

min\_significant: 1

method: p-value

#### estimate(sample:Sample, do\_print:bool=True, is\_summary:bool=True)

This function estimates the model for a given sample. By default, by performing calculations, while specifying each node, it prints them. If you want to stop the printing operation, set the value of do\_print equal to false. Also, by default, each node is printed in its summary form ([to\_summary\_str()](#_to_summary_str())), if you want it to be printed in its full form ([to\_full\_str()](#_to_full_str())), set the value of is\_summary to false.

The output of this function will be an object from '[Equation](#_Equation)' class that you can call different estimated values from this class.

**Input:**

dep\_var, method = {'tv\_use':Variable\_Types.numeric}, 'p-value'

    indep\_vars = {'sex':Variable\_Types.categorical, 'age':Variable\_Types.numeric,

            'city':Variable\_Types.categorical, 'position\_in\_family':Variable\_Types.categorical,

            'marital\_status':Variable\_Types.categorical,'children':Variable\_Types.numeric,

            'children\_welfare':Variable\_Types.numeric, 'religion':Variable\_Types.categorical,

            'faith':Variable\_Types.categorical, 'education':Variable\_Types.numeric,

            'job':Variable\_Types.categorical, 'healthy':Variable\_Types.numeric, 'income':Variable\_Types.numeric,

            'pollution':Variable\_Types.numeric, 'planet\_age':Variable\_Types.numeric,

            'causes\_global\_warming':Variable\_Types.categorical, 'effects\_global\_warming':Variable\_Types.numeric,

            'environment\_attitudes\_validity':Variable\_Types.numeric, 'env\_pleasure':Variable\_Types.numeric,

            'env\_government':Variable\_Types.numeric, 'env\_avtivist':Variable\_Types.numeric,

            'env\_population':Variable\_Types.numeric, 'env\_total':Variable\_Types.numeric,

            'personality\_level1':Variable\_Types.categorical,'personality\_level2':Variable\_Types.categorical,

            'personality\_level3':Variable\_Types.categorical, 'risk\_aversion':Variable\_Types.numeric,

            'expected\_age':Variable\_Types.numeric, 'life\_satisfaction':Variable\_Types.numeric,

            'life\_control':Variable\_Types.numeric, 'social\_capital':Variable\_Types.numeric}

    model = tree\_based\_regression.Model(dep\_var, indep\_vars, min\_sample=25, method=method)

    model.estimate(s\_total, True, False)

**Output:**

n = 217, life\_control <= 4.0000

n = 64, personality\_level3 == ['authority', 'friendly\_relations', 'reputation']

n = 33, life\_control <= 3.0000

S1: n = 33, tv\_use: mean = 503.2576, mse = 110390.0331

n = 31, life\_control > 3.0000

S2: n = 31, tv\_use: mean = 306.1290, mse = 65251.5995

n = 153, personality\_level3 == ['independence', 'fun', 'security', 'exciting', 'longevity']

n = 69, personality\_level2 == ['authority', 'exciting', 'fun', 'independence', 'longevity']

n = 30, env\_population <= 0.0000

S3: n = 30, tv\_use: mean = 358.9167, mse = 88395.5532

n = 39, env\_population > 0.0000

S4: n = 39, tv\_use: mean = 627.0513, mse = 239796.0105

n = 84, personality\_level2 == ['friendly\_relations', 'security', 'reputation']

n = 44, env\_total <= 0.3750

S5: n = 44, tv\_use: mean = 1082.5568, mse = 961075.4327

n = 40, env\_total > 0.3750

S6: n = 40, tv\_use: mean = 588.5625, mse = 304858.9383

n = 45, life\_control > 4.0000

S7: n = 45, tv\_use: mean = 401.5000, mse = 159870.0000

## Equation

### Definition

The equation is the output of estimating a [Model](#_Model). In addition to the estimated print output (summary format (is\_summary=True) and full format (is\_summary=False)), you can get all the components of the decision tree, including the list of nodes and leaves (or the list of final decision tree splits) in this class.

Additionally, you can save these results to your computer or load the previously saved results and use them to make new predictions without having to re-estimate the model.

Properties of this class include:

* dep\_var: as a [Variable](#_Variable) object.
* indep\_vars: as a list of [Variable](#_Variable) objects.
* node\_list: as a list of [Node](#_Node_1) objects.
* leafs: as a list of [Node](#_Node_1) objects.
* summary\_str: as a string.
* full\_str: as a string.

### Methods

#### \_\_str\_\_()

Returns summary format of output of estimation of a [Model](#_Node).

**Input:**

eq = model.estimate(s\_total, do\_print=False)

print('equation is:')

print(eq)

# equals to

print(eq.summary\_str)

**Output**

equation is:

n = 217, life\_control <= 4.0000

n = 64, personality\_level3 == ['authority', 'friendly\_relations', 'reputation']

n = 33, life\_control <= 3.0000

S1: n = 33, tv\_use: mean = 503.2576, mse = 110390.0331

n = 31, life\_control > 3.0000

S2: n = 31, tv\_use: mean = 306.1290, mse = 65251.5995

n = 153, personality\_level3 == ['longevity', 'independence', 'security', 'fun', 'exciting']

n = 69, personality\_level2 == ['authority', 'exciting', 'fun', 'independence', 'longevity']

n = 30, env\_population <= 0.0000

S3: n = 30, tv\_use: mean = 358.9167, mse = 88395.5532

n = 39, env\_population > 0.0000

S4: n = 39, tv\_use: mean = 627.0513, mse = 239796.0105

n = 84, personality\_level2 == ['friendly\_relations', 'reputation', 'security']

n = 44, env\_total <= 0.3750

S5: n = 44, tv\_use: mean = 1082.5568, mse = 961075.4327

n = 40, env\_total > 0.3750

S6: n = 40, tv\_use: mean = 588.5625, mse = 304858.9383

n = 45, life\_control > 4.0000

S7: n = 45, tv\_use: mean = 401.5000, mse = 159870.0000

If you want full\_str can write:

print(eq.full\_str)

#### forecast(sample: Sample, name:str='', output:str='point')

Predicts the dependent variable using the estimated model for a new sample (for example, a test sample). The output is an object of [Data](#_Data_class) class, its values of which can be in one of three forms:

* **leaf:** as a [Leaf](#_Leaf).
* **dist:** distribution of dependent variable as a dictionary that its keys are values of dependent variable and its values are the frequency of each value of dependent variable (-> [Node](#_Node_1)->[distribution](#_distribution(sorted:_bool_=))
* **point:** Central tendency of distribution of dependent variable. It is mean for numeric variables and mode for categorical variable (->[Node](#_Node_1)->[dist\_center](#_dist_center()))

**Input:**

forecast\_test\_leaf = eq.forecast(s\_test, name='sample', output ='leaf')

forecast\_test\_dist = eq.forecast(s\_test, name='sample', output ='dist')

forecast\_test\_point = eq.forecast(s\_test, name='sample', output ='point')

print('forecast leaf')

print(forecast\_test\_leaf)

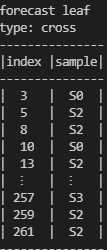
print('forecast distribution')

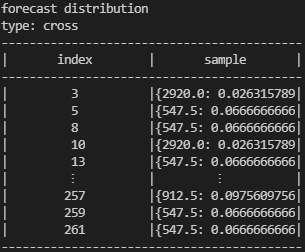
print(forecast\_test\_dist)

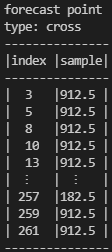
print('forecast point')

print(forecast\_test\_point)

**Output:**







#### goodness\_of\_fit(sample:Sample, decimals:int=4, do\_print:bool=True)

This function provides metrics that show the accuracy of predictions. In other words, how close the predicted values are to the actual values.

**In numerical variables**, R2 and adjusted R2 coefficients are traditionally used. The formula for calculating them is as follows:





Where:

* 
* =Number of sample members
* =Number of nodes in tree (=number of parameters in model)

But these indicators are correct in the in-sample forecast, so for out-of-sample forecasts, an pseudo R2 called McKelvey & Zavonia’s R2 is introduced, which is calculated as follows:



Where:

* 

The ANOVA table, and the F and P-Value statistics are also calculated, which is useful for fitting in-sample predictions and shows how statistically independent samples have different averages. This table is calculated as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Sum of Square** | **Degree of freedom** | **Mean sum of square** | **F** | **P-Value** |
| Between groups (Factor) |  |  |  |  |  |
| Within groups (Error) |  |  |  |  |  |
| Total |  |  |  |  |  |

* 
* 
* 
*  cumulative distribution F.

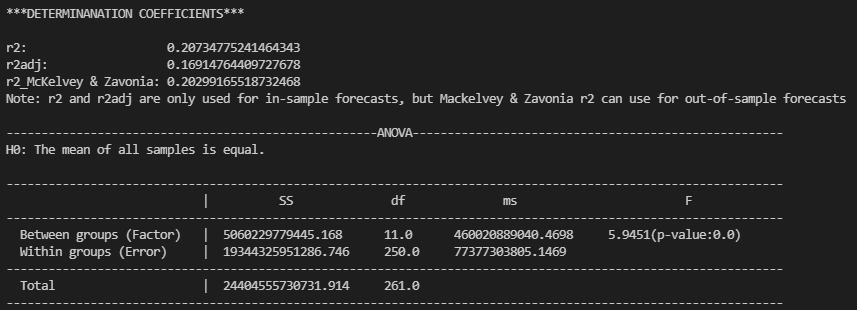
**Note:** ANOVA only uses for in-sample prediction.

**Input:**

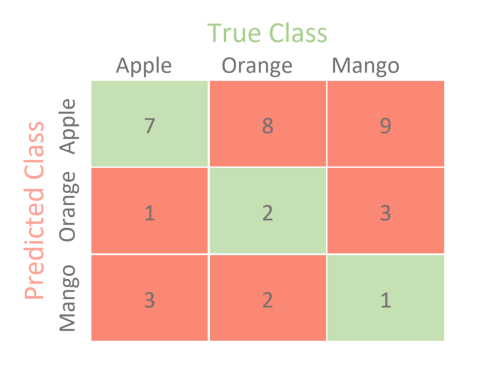
eq = model.estimate(s\_total, do\_print=False, is\_summary=True)

eq.goodness\_of\_fit(s\_total)

**Output:**



**In categorical variables**, uses Confusion Matrix for multi-class classification. First, find TP, TN, FP and FN for each individual class.



For example, if we take class Apple, then let’s see what are the values of the metrics from the confusion matrix.

* True Positive (TP) = 7
* True Negative (TN) = (2+3+2+1) = 8
* False Positive (FP) = (8+9) = 17
* False Negative (FN) = (1+3) = 4

Therefore, accuracy indicators can be calculated as follows:

* Precision = TP/(TP+FP) = 7/(7+17) = 0.29
* Recall = TP/(TP+FN) = 7/(7+4) = 0.64
* F1-score = 2\*(Precision\*Recall)/(Precision+Recall) = 0.40

Global metrics include:

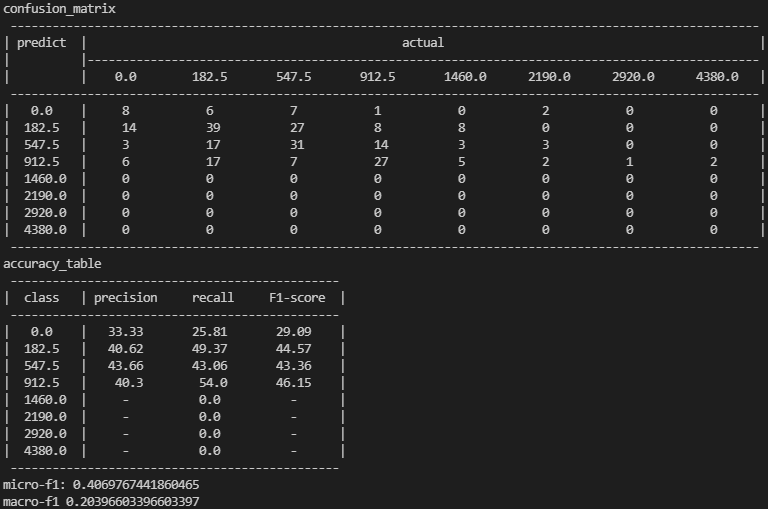
* Micro-F1 = Total TP/Total FN = (7+2+1)/(8+9+1+3+3+2)=0.28
* Macro-F1 = unweighted mean of the F1-scores = (0.40+0.22+0.11)/3 = 0.24

**Input:**

eq = model.estimate(s\_total, do\_print=False, is\_summary=True)

eq.goodness\_of\_fit(s\_total)

**Output:**



Any of the following parameters are also available in the output of this function:

For numeric variables:

* **'r2'**
* **'r2adj'**
* **'r2\_McKelvey & Zavonia'**
* **'f'**
* **'p\_value'**
* **'anova'**

**For categorical variables:**

* **'confusion\_matrix'**
* **'accuracy\_table'**
* **'precisions'**
* **'recalls'**
* **'f1\_scores'**
* **'micro-f1'**
* **'macro-f1'**

**For example:**

eq = model.estimate(s\_total, do\_print=False, is\_summary=True)

gf = eq.goodness\_of\_fit(s\_total)

ptint(gf['micro-f1'])

#### plot(size:int=10, decimals:int=2)

This function draws a tree diagram using **matplotlib** package, Therefore, you must install this package before using this function. When the tree is large, the boxes may overlap, so by reducing value of **size** from the default value by 10, you can make the chart more readable, or if the tree is too small, you can even increase value of **size**. You can also change the number of decimal places displayed in the boxes, the numbers are rounded to two decimal places by default.

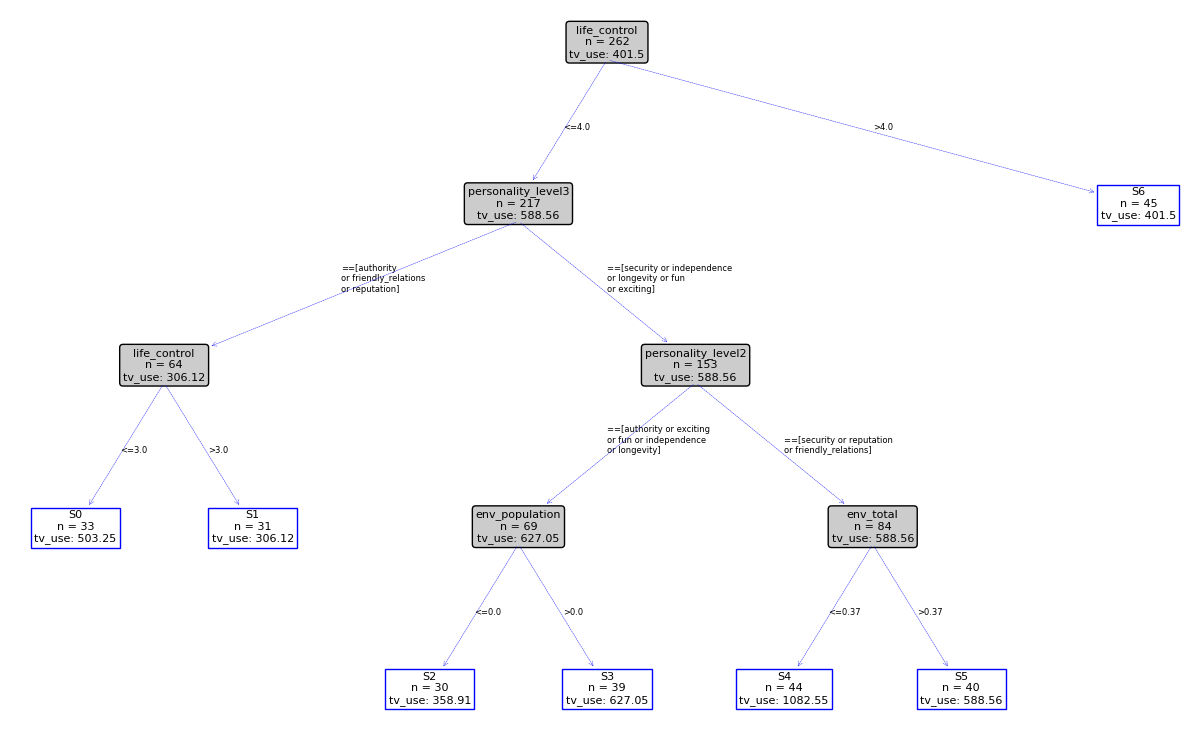
**Input:**

model = tree\_based\_regression.Model(dep\_var, indep\_vars, min\_sample=25, method=method)

eq = model.estimate(s\_total, do\_print=False, is\_summary=True)

eq.plot(8)

**Output:**



#### save(path\_name:str = 'results')

This function stores the parameters of the equation class (= model estimation results) in a file. You can specify the path and name of that file. If you just enter the name, the file will be saved in the current folder, and if no name is entered, the sample name will be used. And if the sample does not have a name, the file name will be entered as a result.

**Note:** If there is already a file with the same name, a new file will replace it.

model = tree\_based\_regression.Model(dep\_var, indep\_vars, min\_sample=25, method=method)

eq = model.estimate(s\_total, do\_print=False, is\_summary=True)

eq.save()

#### load(self, name:str = '', path:str = '')

This function reads the stored parameters, and you can re-execute the equation class's functions on them, without having to redefine and estimate the model.

eq2 = Equation.load('total')

eq2.goodness\_of\_fit(s\_total)

## Node

### Definition

A tree-based regression model involves a number of nodes. Each node has the following properties:

* **sample:** This sample is a subset of the parent sample, depending on whether this node is left or right.
* **dep\_var:** It is a dependent variable that it inherits from its parent.
* **parent:** This argument is the parent node, and it has all its properties.
* **split\_var:** This argument is the split variable that defines the node.
* **relation:** This argument is an operator that applys on the variable and the value of the split for defining the node. For example, if this node is a left node and the split variable is numeric, it is equal to <= and if the node is right, it is equal to >.
* **split\_value:** This argument is a relation that is defined on the variable and the value of the split. For example, if this node is a left node and the variable is a numerical split, it is equal to <= and if the node is right, it is equal to>
* **t:** This argument is a t-student or Chi-square statistics calculated to compare the mean or distribution of this node with its parallel node (its parent's other child).
* **p\_value:** This argument is the P-value of t argument.

Since the output of the tree-based regression estimation is a list of nodes, knowing this class and its methods can be useful.

### Methods

#### depth()

The depth of the node in the tree, which is actually equal to the number of its parents.

#### parents()

Returns a list of nodes that includes its parent and its parent's parents.

#### childs()

Returns a list of nodes that includes its children and its children's children.

#### \_\_str\_\_()

Returns to\_summary\_str()

#### to\_full\_str()

Includes:

* Number of Sample
* dependent variable name,
* dependent variable mean and mse if dependent variable is numeric else its distribution,
* split variable, relation and split value,
* t and p\_value argument for left nodes.

#### to\_summary\_str()

it only Includes: Number of Sample, and split variable, relation and split value.

#### stats\_split(var\_split:Variable, value\_split:Union[float, list], method:str = Methods.mse)

This function returns a dictionary of statistical indicators for a specific split (variable and split value) on this node.

if dependent variable is numeric, its keys are:

* **'count\_left':** the number of observation in left sample.
* **'count\_right':** the number of observation in right sample.
* **'mse':** weighted average of mean square of error of left and right sample.
* **'poisson':** weighted average of poisson of left and right sample.
* **'mae':** weighted average of mae of left and right sample.,
* **'t':** It is a t-statistic in the test comparing the means of the right and left sample.
* **'p\_value':** It is a p-value of t-statistics in the test comparing the means of the right and left sample.

if dependent variable is categorical, its keys are:

* **'count\_left':** the number of observation in left sample.
* **'count\_right':** the number of observation in right sample.
* **'gini':** weighted average of gini index of left and right sample.
* **'entropy':** weighted average of entropy index of left and right sample.
* **'misclassification':** weighted average of misclassification index of left and right sample.
* **'chi2':** It is a chi2-statistic in the test comparing the means of the right and left sample.
* **'p\_value':** It is a p-value of chi2-statistics in the test comparing the means of the right and left sample.

#### best\_value(var\_split: Variable, min\_sample:int=1, min\_significant:int=1, method=Methods.mse)

This function calculates the best split value for a specific split variable on this node. Its output is a dictionary with these keys:

* **'value':** best split value
* **'method':** the value of criteria, mse, poisson, mae, gini, entropy, misclassification, and p\_value.
* **'chi2'** or **'t':** split statistic depends on type of dependent variable.
* **'p-value':** split p-value

**Note:** When the method is p-value, the value of the method and p-value are euqal.

#### best\_split(indep\_vars:list, min\_sample:int=1, min\_significant:int=1, method=Methods.mse)

This function finds the best split on this node by receiving the list of independent variables and its output is a dictionary with two keys:

* **'left':** its value is left node.
* **'right':** its value is right node.

#### distribution(sorted: bool = True)

This function returns the distribution of the dependent variable in the node's sample. Distribution is in the form of a dictionary that its keys are values of dependent variable and its values are the frequency of each value of dependent variable.

By default, values are sorted from small to large. This increases the computation time a bit, if the order of the values is not important, you can set 'sorted' to False.

#### dist\_center()

This function returns the central measure of the dependent variable distribution in the node's sample. This measure is the arithmetic mean for numerical variables and mode for categorical variables.

## Leaf

### Definition

Leaf is a class that inherits from [Node](#_Node) and represents the last nodes in the decision tree. In fact, leaves's samples are categories of the main sample, and we can call the properties of these categories through the node and leaf object.

### Methods

#### \_\_str\_\_()

Returns the name of the final division, where 's' + is the division index, 0, ..., n.

#### definition()

Returns a string value that specifies the conditions for defining a leaf, which consists of the split variable name chain, the relation, and the split value for the node leading to the leaf.

# linear\_regressions

## Definition

Linear regression consists of the following equation, which are numerically independent variables and are combined linearly with each other.



 = number of observation

 = th independent variable

 = th independent variable of th observation

= coeficient of th independent variable

**** = intercept

**** = error of th observation

**** = the number of observations

 = the number of independent variables

 = the number of parapemeters

Categorical variables are converted to dummy variables and then used as a numerical variable in the model. We use the Formula and Formulas class to construct these variables.

Simple regression is a linear regression with a numerically dependent variable that is estimated by the least squares method.

In logistic regression, the dependent variable is a binary variable, or a numerical variable consisting of zeros and ones, or a categorical variable with only two values.

## Model

The model class should be used to introduce all linear regressions. This class has two properties, the dependent variable (**dep\_var**) and the **formula**. For the dependent variable we enter the name of the dependent variable and the formula is a linear combination of names of independent variables.

This class also has three methods:

* **\_\_str\_\_():** which works if the model object is placed in the Str or print function.
* **estimate:** that receives a sample and estimates the model on it. For all models, you can define whether or not to print the result when estimating.
* **best\_estimate:** removes meaningless independent variables from the most meaningless variables and finally returns the best estimate according to the level of meaning you choose.
* **motor:** name of a package from **statsmodels** and **sklearn** packeges, to run estimation by its. By default **motor** equals **statsmodels**.

The output of the estimation functions is an object of **Equation** class.

## Equation

All estimated parameters, including the names of the dependent and independent variables, the coefficients, the covariance matrix of the coefficients, the sample on which the estimation is performed, and the result table are stored as an object of the equation class.

This class has five functions for all linear regressions:

* **str:** Returns the printed form of the estimated equation in the str and print functions.
* **forecast:** Predicts the model on the sample you enter.
* **goodness\_of\_fit:** Includes indicators that indicate the accuracy of the model.
* **save:** saves the results in a file in the path you specify.
* **load:** Loads a saved equation and you can make new predictions based on it, without having to re-estimate the model.

**Note:** At present only simple regression is estimated independently. Other regressions are estimated using the **statsmodels** and **sklearn** packeges as you selected by **motor** argument in **estimate** function, so this library needs to be installed.

## Examples:

### simple\_regression

**Input:**

from \_models.linear\_regressions import simple\_regression as reg

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:300},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:'b', 5:'a'},

    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

})

model = reg.Model('log(x)', '1+p')

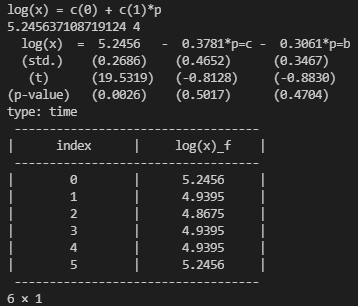
print(model)

eq= model.estimate(Sample(data), do\_print=False)

print(eq)

print(eq.forecast(Sample(data)))

**Outputs**



### logistic\_regression

**Input:**

from \_models.linear\_regressions import logistic\_regression as logit

data = Data('time', {

    'year': {0:50, 1:1396, 2:np.nan, 3:1398, 4:1399, 5:1400},

    'x': {0:120, 1:128, 2:130, 3:142, 4:150, 5:300},

    'p': {0:'a', 1:'b', 2:'c', 3:'b', 4:'b', 5:'a'},

    'q': {0:'t', 1:'t', 2:'w', 3:'w', 4:'w', 5:np.nan},

})

model = logit.Model('q', 'log(x)')

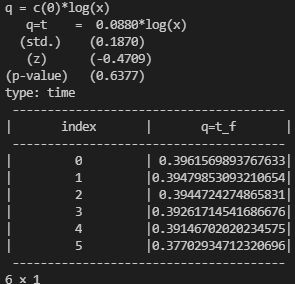
print(model)

eq= model.estimate(Sample(data), do\_print=False)

print(eq)

print(eq.forecast(Sample(data)))

**Outputs**



1. - Jack Kuang Tsung Chen, Richard L. Valliant and Michael R. Elliott, 2019, Calibrating non-probability surveys to estimated control totals using LASSO, with an application to political polling, Journal of Royal Statistical Society, Volume 68, Issue 3, Pages: 657-681 [↑](#footnote-ref-1)