ChoiceModels Documentation

Release 0.2.2dev0

Urban Data Science Toolkit

Apr 23, 2019

Contents

ChoiceModels is a Python library for discrete choice modeling, with utilities for sampling, simulation, and other ancillary tasks. It's part of the Urban Data Science Toolkit (UDST).

v0.2.2dev0, released April 23, 2019

CHAPTER 1

Contents

1.1 Getting started

1.1.1 Intro

ChoiceModels is a Python library for discrete choice modeling, with utilities for sampling, simulation, and other ancillary tasks. It's part of the Urban Data Science Toolkit (UDST).

The library currently focuses on tools to help integrate discrete choice models into larger workflows, drawing on other packages such as the excellent PyLogit for most estimation of models. ChoiceModels can automate the creation of choice tables for estimation or simulation, using uniform or weighted random sampling of alternatives, as well as interaction terms or cartesian merges. It also provides general-purpose tools for Monte Carlo simulation of choices given probability distributions from fitted models, with fast algorithms for independent or capacity-constrained choices. ChoiceModels includes a custom engine for Multinomial Logit estimation that's optimized for fast performance with large numbers of alternatives.

ChoiceModels is hosted on Github with a BSD 3-Clause open source license. The code repository includes some material not found in this documentation: a change log, a contributor's guide, and instructions for running the tests, updating the documentation, and creating a new release. Another useful resource is the issues and pull requests on Github, which include detailed feature proposals and other discussions.

ChoiceModels was created in 2016, with contributions from Sam Maurer (maurer@urbansim.com), Timothy Brathwaite, Geoff Boeing, Paul Waddell, Max Gardner, Eddie Janowicz, Arezoo Besharati Zadeh, Jacob Finkelman, Catalina Vanoli, and others. It includes earlier code written by Matt Davis, Fletcher Foti, and Paul Sohn.

1.1.2 Installation

ChoiceModels is tested with Python 2.7, 3.5, 3.6, and 3.7. It should run on any platform.

Production releases

ChoiceModels can be installed using the Pip or Conda package managers. We recommend Conda because it resolves dependency conflicts better.

```
pip install choicemodels
```

```
conda install choicemodels --channel conda-forge
```

When new production releases of ChoiceModels come out, you can upgrade like this:

```
pip install choicemodels --upgrade
```

```
conda update choicemodels --channel conda-forge
```

Developer pre-releases

Developer pre-releases of ChoiceModels can be installed using the Github URL. Additional information about the developer releases can be found in Github pull requests.

```
pip install git+git://github.com/udst/choicemodels.git
```

You can use the same command to upgrade.

Cloning the repository

You can also install ChoiceModels by cloning the Github repository, which is the best way to do it if you'll be modifying the code. The main branch contains the latest developer release.

```
git clone https://github.com/udst/choicemodels.git
cd choicemodels
python setup.py develop
```

Update it with `git pull`.

1.1.3 Basic usage

You can use components of ChoiceModels individually, or combine them together to streamline model estimation and simulation workflows. Other UDST libraries like UrbanSim Templates use ChoiceModels objects as inputs and outputs.

If you have choosers and alternatives as Pandas DataFrames, you can prepare them for model estimation like this:

```
mct = choicemodels.tools.MergedChoiceTable(obs, alts, chosen_alternatives='chosen',
                                           sample_size=10, ..)
```

Then, you can estimate a Multinomial Logit model like this:

```
results = choicemodels.MultinomialLogit(mct, model_expression='x1 + x2 + x3')
```

This provides a `choicemodels.MultinomialLogitResults` object, from which you can obtain probability distributions for out-of-sample choice scenarios in order to generate simulated choices.

```
mct2 = choicemodels.tools.MergedChoiceTable(obs2, alts, sample_size=10, ..)
probs = results.probabilities(mct2)
choices = choicemodels.tools.monte_carlo_choices(probs)
```

1.2 Choice table utilities API

Working with discrete choice models can require a lot of data preparation. Each chooser has to be matched with hypothetical alternatives, either to simulate choice probabilities or to compare them with the chosen alternative for model estimation.

ChoiceModels includes a class called `MergedChoiceTable` that automates this. To build a merged table, create an instance of the class and pass it one `pd.DataFrame` of choosers and another of alternatives, with whatever other arguments are needed (see below for full API).

The merged data table can be output to a DataFrame, or passed directly to other ChoiceModels tools as a `MergedChoiceTable` object. (This retains metadata about indexes and other special columns.)

```
mct = choicemodels.tools.MergedChoiceTable(obs, alts, ..)
df = mct.to_frame()
```

This tool is designed especially for models that need to sample from large numbers of alternatives. It supports:

- uniform random sampling of alternatives, with or without replacement
- weighted random sampling based either on characteristics of the alternatives or on combinations of chooser and alternative
- interaction terms to be merged onto the final data table
- cartesian merging of all the choosers with all the alternatives, without sampling

All of the sampling procedures work for both estimation (where the chosen alternative is known) and simulation (where it is not).

1.2.1 MergedChoiceTable

class choicemodels.tools.**MergedChoiceTable**(*observations*, *alternatives*, *chosen_alternatives=None*, *sample_size=None*, *replace=True*, *weights=None*, *availability=None*, *interaction_terms=None*, *random_state=None*)

Generates a merged long-format table of observations (choosers) and alternatives, for discrete choice model estimation or simulation.

Supports random sampling of alternatives (uniform or weighted). Supports sampling with or without replacement. Supports merging observations and alternatives without sampling them. Supports alternative-specific weights, as well as interaction weights that depend on both the observation and alternative. Supports automatic merging of interaction terms onto the final data table.

Support is PLANNED for specifying availability of alternatives, specifying random state, and passing interaction-type parameters as callable generator functions.

Does NOT support cases where the number of alternatives in the final table varies across observations.

Reserved column names: 'chosen'.

Parameters

- **observations** (*pandas.DataFrame*) – Table with one row for each chooser or choice scenario, with unique ID's in the index field. Additional columns can contain fixed attributes of the choosers. Index name is set to 'obs_id' if none provided. All observation/alternative column names must be unique except for the join key.
- **alternatives** (*pandas.DataFrame*) – Table with one row for each alternative, with unique ID's in the index field. Additional columns can contain fixed attributes of the alternatives. Index name is set to 'alt_id' if none provided. All observation/alternative column names must be unique except for the join key.
- **chosen_alternatives** (*str or pandas.Series, optional*) – List of the alternative ID selected in each choice scenario. (This is required for preparing estimation data, but not for simulation data.) If str, interpreted as a column name from the observations table. If Series, it will be joined onto the obserations table before processing. The column will be dropped from the merged table and replaced with a binary column named 'chosen'.
- **sample_size** (*int, optional*) – Number of alternatives to sample for each choice scenario. If 'None', all of the alternatives will be available for each chooser in the merged table. The sample size includes the chosen alternative, if applicable. If replace=False, the sample size must be less than or equal to the total number of alternatives.
- **replace** (*boolean, optional*) – Whether to sample alternatives with or without replacement, at the level of a single chooser or choice scenario. If replace=True (default), alternatives may appear multiple times in a single choice set. If replace=False, an alternative will appear at most once per choice set. Sampling with replacement is much more efficient, so setting replace=False may have performance implications if there are very large numbers of observations or alternatives.
- **weights** (*str, pandas.Series, optional*) – Numerical weights to apply when sampling alternatives. If str, interpreted as a column from the alternatives table. If Series, it can contain either (a) one weight for each alternative or (b) one weight for each combination of observation and alternative. The former should include a single index with ID's from the alternatives table. The latter should include a MultiIndex with the first level corresponding to the observations table and the second level corresponding to the alternatives table. If callable, it should accept two arguments (obs_id, alt_id) and return the corresponding weight.

  TO DO - accept weights specified with respect to derivative characteristics, like how the interaction terms work (for example weights could be based on home census tract rather than observation id if there are multiple observations per tract)

  TO DO - implement support for a callable
- **availability** (*pandas.Series or callable, optional (NOT YET IMPLEMENTED)*) – Binary representation of the availability of alternatives. Specified and applied similarly to the weights.
- **interaction_terms** (*pandas.Series, pandas.DataFrame, or list of either, optional*) – Additional column(s) of interaction terms whose values depend on the combination of observation and alternative, to be merged onto the final data table. If passed as a Series or DataFrame, it should include a two-level MultiIndex. One level's name and values should match an index or column from the observations table, and the other should match an index or column from the alternatives table.

  TO DO - implement support for a callable
- **random_state** (*NOT YET IMPLEMENTED*) – Representation of random state, for replicability of the sampling.

alternative_id_col
Name of column in the merged table containing the alternative id. Name and values will match the index of the alternatives table.

Returns

Return type str

choice_col
Name of the generated column containing a binary representation of whether each alternative was chosen in the given choice scenario, if applicable.

Returns

Return type str or None

observation_id_col
Name of column in the merged table containing the observation id. Name and values will match the index of the observations table.

Returns

Return type str

to_frame()
Long-format DataFrame of the merged table. The rows representing alternatives for a particular chooser are contiguous, with the chosen alternative listed first if applicable. (Unless no sampling is performed, in which case the alternatives are listed in order.) The DataFrame includes a two-level MultiIndex. The first level corresponds to the index of the observations table and the second to the index of the alternatives table.

Returns

Return type pandas.DataFrame

1.3 Multinomial Logit API

ChoiceModels has built-in functionality for Multinomial Logit estimation and simulation. This can use either the PyLogit MNL estimation engine or a custom engine optimized for fast performance with large numbers of alternatives. The custom engine is originally from `urbansim.urbanchoice`.

Fitting a model yields a results object that can generate choice probabilities for out-of-sample scenarios.

1.3.1 MultinomialLogit

class choicemodels.**MultinomialLogit**(*data*, *model_expression*, *observation_id_col=None*, *choice_col=None*, *model_labels=None*, *alternative_id_col=None*, *initial_coefs=None*, *weights=None*)
A class with methods for estimating multinomial logit discrete choice models. Each observation is a choice scenario in which a chooser selects one alternative from a choice set of two or more. The fitted parameters represent a joint optimization of utility expressions that explains observed choices based on attributes of the alternatives and of the choosers.

The input data needs to be in "long" format, with one row for each combination of chooser and alternative. Columns contain relevant attributes and identifiers. (If the choice sets are large, sampling of alternatives should be carried out before data is passed to this class.)

The class constructor supports two use cases:

1. The first use case is simpler and requires fewer inputs. Each choice scenario must have the same number of alternatives, and each alternative must have the same model expression (utility equation). This is typical when the alternatives are relatively numerous and homogenous, for example with travel destination choice or household location choice.

   The following parameters are required: 'data', 'observation_id_col', 'choice_col', 'model_expression' in Patsy format. If data is provided as a MergedChoiceTable, the observation id and choice column names can be read directly from its metadata.

   To fit this type of model, ChoiceModels will use its own estimation engine adapted from the UrbanSim MNL codebase.

   Migration from 'urbansim.urbanchoice': Note that these requirements differ from the old UrbanSim codebase in a couple of ways. (1) The chosen alternatives need to be indicated in a column of the estimation data table instead of in a separate matrix, and (2) in lieu of indicating the number of alternatives in each choice set, the estimation data table should include an observation id column. These changes make the API more consistent with other use cases. See the MergedChoiceTable() class for tools and code examples to help with migration.

2. The second use case is more flexible. Choice scenarios can have varying numbers of alternatives, and the model expression (utility equation) can be different for distinct alternatives. This is typical when there is a small number of alternatives whose salient characteristics vary, for example with travel mode choice.

   The following parameters are required: 'data', 'observation_id_col', 'alternative_id_col', 'choice_col', 'model_expression' in PyLogit format, 'model_labels' in PyLogit format (optional).

   To fit this type of model, ChoiceModels will use the PyLogit estimation engine.

With either use case, the model expression can include attributes of both the choosers and the alternatives. Attributes of a particular alternative may vary for different choosers (distance, for example), but this must be set up manually in the input data.

Note that prediction methods are in a separate class: see MultinomialLogitResults().

Parameters

- **data** (*pd.DataFrame or choicemodels.tools.MergedChoiceTable*) – A table of estimation data in "long" format, with one row for each combination of chooser and alternative. Column labeling must be consistent with the 'model_expression'. May include extra columns.
- **model_expression** (*Patsy 'formula-like' or PyLogit 'specification'*) – For the simpler use case where each choice scenario has the same number of alternatives and each alternative has the same model expression, this should be a Patsy formula representing the right-hand side of the single model expression. This can be a string or a number of other data types. See here: https://patsy.readthedocs.io/en/v0.1.0/API-reference.html#patsy.dmatrix

  For the more flexible use case where choice scenarios have varying numbers of alternatives or the model expessions vary, this should be a PyLogit OrderedDict model specification. See here: https://github.com/timothyb0912/pylogit/blob/master/pylogit/pylogit.py#L116-L130
- **observation_id_col** (*str, optional*) – Name of column or index containing the observation id. This should uniquely identify each distinct choice scenario. Not required if data is passed as a MergedChoiceTable.
- **choice_col** (*str, optional*) – Name of column containing an indication of which alternative has been chosen in each scenario. Values should evaluate as binary: 1/0, True/False, etc. Not required if data is passed as a MergedChoiceTable.

- **model_labels** (*PyLogit 'names', optional*) – If the model expression is a PyLogit OrderedDict, you can provide a corresponding OrderedDict of labels. See here: https://github.com/timothyb0912/pylogit/blob/master/pylogit/pylogit.py#L151-L165
- **alternative_id_col** (*str, optional*) – Name of column or index containing the alternative id. This is only required if the model expression varies for different alternatives. Not required if data is passed as a MergedChoiceTable.
- **initial_coefs** (*numeric or list-like of numerics, optional*) – Initial coefficients (beta values) to begin the optimization process with. Provide a single value for all coefficients, or an array containing a value for each one being estimated. If None, initial coefficients will be 0.
- **weights** (*1D array, optional*) – NOT YET IMPLEMENTED - Estimation weights.

estimation_engine
'ChoiceModels' or 'PyLogit'.

fit()
Fit the model using maximum likelihood estimation. Uses either the ChoiceModels or PyLogit estimation engine as appropriate.

Returns

Return type *MultinomialLogitResults*() object.

1.3.2 MultinomialLogitResults

class choicemodels.**MultinomialLogitResults**(*model_expression*, *results=None*, *fitted_parameters=None*, *estimation_engine='ChoiceModels'*)
The results class represents a fitted model. It can report the model fit, generate choice probabilties, etc.

A full-featured results object is returned by MultinomialLogit.fit(). A results object with more limited functionality can also be built directly from fitted parameters and a model expression.

Parameters

- **model_expression** (*str or OrderedDict*) – Patsy 'formula-like' (str) or PyLogit 'specification' (OrderedDict).
- **results** (*dict or object, optional*) – Raw results as currently provided by the estimation engine. This should be replaced with a more consistent and comprehensive set of inputs.
- **fitted_parameters** (*list of floats, optional*) – If not provided, these will be extracted from the raw results.
- **estimation_engine** (*str, optional*) – 'ChoiceModels' (default) or 'PyLogit'.

get_raw_results()
Return the raw results as provided by the estimation engine. Dict or object.

probabilities(*data*)
Generate predicted probabilities for a table of choice scenarios, using the fitted parameters stored in the results object.

Parameters

- **data** (choicemodels.tools.MergedChoiceTable) – Long-format table of choice scenarios. TO DO - accept other data formats.

- **class parameters** (*Expected*) –
- **------------------------** –
- **self.model_expression** (*patsy string*) –
- **self.fitted_parameters** (*list of floats*) –

Returns

Return type pandas.Series with indexes matching the input

report_fit()
Print a report of the model estimation results.

1.4 Simulation utilities API

ChoiceModels provides general-purpose tools for Monte Carlo simulation of choices among alternatives, given probability distributions generated from fitted models.

`monte_carlo_choices()` is equivalent to applying `np.random.choice()` in parallel for many independent choice scenarios, but it's implemented as a single-pass matrix calculation that is much faster.

`iterative_lottery_choices()` is for cases where the alternatives have limited capacitiesxs, requiring multiple passes to match choosers and alternatives. Effectively, choices are simulated sequentially, each time removing the chosen alternative or reducing its available capacity. (It's actually done in batches for better performance.)

`parallel_lottery_choices()` works functionally the same as the above but the batches run in parallel rather than sequentially.

1.4.1 Independent choices

choicemodels.tools.**monte_carlo_choices**(*probabilities*)
Monte Carlo simulation of choices for a set of K scenarios, each having different probability distributions (and potentially different alternatives).

Choices are independent and unconstrained, meaning that the same alternative can be chosen in multiple scenarios.

This function is equivalent to applying np.random.choice() to each of the K scenarios, but it's implemented as a single-pass matrix calculation. When the number of scenarios is large, this is about 50x faster than using df.apply() or a loop.

If all the choice scenarios have the same probability distribution among alternatives, you don't need this function. You can use np.random.choice() with size=K, which will be more efficient. (For example, that would work for a choice model whose expression includes only attributes of the alternatives.)

NOTE ABOUT THE INPUT FORMATS: It's important for the probabilities to be structured correctly. This is computationally expensive to verify, so you will not get a warning if it's wrong! (TO DO: we should provide an option to perform these checks, though)

1. Probabilities (pd.Series) must include a two-level MultiIndex, the first level representing the scenario (observation) id and the second the alternative id.
2. Probabilities must be sorted so that each scenario's alternatives are consecutive.
3. Each scenario must have the same number of alternatives. You can pad a scenario with zero-probability alternatives if needed.
4. Each scenario's alternative probabilities must sum to 1.

Parameters **probabilities** (*pd.Series*) – List of probabilities for each observation (choice scenario) and alternative. Please verify that the formatting matches the four requirements described above.

Returns List of chosen alternative id's, indexed with the observation id.

Return type pd.Series

1.4.2 Capacity-constrained choices

choicemodels.tools.**iterative_lottery_choices**(*choosers*, *alternatives*, *mct_callable*, *probs_callable*, *alt_capacity=None*, *chooser_size=None*, *max_iter=None*, *chooser_batch_size=None*)

Monte Carlo simulation of choices for a set of choice scenarios where (a) the alternatives have limited capacity and (b) the choosers have varying probability distributions over the alternatives.

Effectively, we simulate the choices sequentially, each time removing the chosen alternative or reducing its available capacity. (It's actually done in batches for better performance, but the outcome is equivalent.) This requires sampling alternatives and calculating choice probabilities multiple times, which is why callables for those actions are required inputs.

Chooser priority is randomized. Capacities can be specified as counts (number of choosers that can be accommodated) or as amounts (e.g. square footage) with corresponding chooser sizes. If total capacity is insufficient to accommodate all the choosers, as many choices will be simulated as possible.

Note that if all the choosers are the same size and have the same probability distribution over alternatives, you don't need this function. You can use np.random.choice() with size=K to draw chosen alternatives, which will be more efficient. (This function also works, though.)

Parameters

- **choosers** (*pd.DataFrame*) – Table with one row for each chooser or choice scenario, with unique ID's in the index field. Additional columns can contain fixed attributes of the choosers. (Reserved column names: '_size'.)
- **alternatives** (*pd.DataFrame*) – Table with one row for each alternative, with unique ID's in the index field. Additional columns can contain fixed attributes of the alternatives. (Reserved column names: '_capacity'.)
- **mct_callable** (*callable*) – Callable that samples alternatives to generate a table of choice scenarios. It should accept subsets of the choosers and alternatives tables and return a choicemodels.tools.MergedChoiceTable.
- **probs_callable** (*callable*) – Callable that generates predicted probabilities for a table of choice scenarios. It should accept a choicemodels.tools.MergedChoiceTable and return a pd.Series with indexes matching the input.
- **alt_capacity** (*str, optional*) – Name of a column in the alternatives table that expresses the capacity of alternatives. If not provided, each alternative is interpreted as accommodating a single chooser.
- **chooser_size** (*str, optional*) – Name of a column in the choosers table that expresses the size of choosers. Choosers might have varying sizes if the alternative capacities are amounts rather than counts – e.g. square footage or employment capacity. Chooser sizes must be in the same units as alternative capacities. If not provided, each chooser has a size of 1.
- **max_iter** (*int or None, optional*) – Maximum number of iterations. If None (default), the algorithm will iterate until all choosers are matched or no alternatives remain.

- **chooser_batch_size** (*int or None, optional*) – Size of the batches for processing smaller groups of choosers one at a time. Useful when the anticipated size of the merged choice tables (choosers X alternatives X covariates) will be too large for python/pandas to handle.

Returns List of chosen alternative id's, indexed with the chooser (observation) id.

Return type pd.Series

1.4.3 Parallelized capacity-constrained choices

choicemodels.tools.**parallel_lottery_choices**(*choosers*, *alternatives*, *mct_callable*, *probs_callable*, *alt_capacity=None*, *chooser_size=None*, *chooser_batch_size=None*)

A parallelized version of the iterative_lottery_choices method. Chooser batches are processed in parallel rather than sequentially.

NOTE: In it's current form, this method is only supported for simulating choices where every alternative has a capacity of 1.

Parameters

- **choosers** (*pd.DataFrame*) – Table with one row for each chooser or choice scenario, with unique ID's in the index field. Additional columns can contain fixed attributes of the choosers. (Reserved column names: '_size'.)
- **alternatives** (*pd.DataFrame*) – Table with one row for each alternative, with unique ID's in the index field. Additional columns can contain fixed attributes of the alternatives. (Reserved column names: '_capacity'.)
- **mct_callable** (*callable*) – Callable that samples alternatives to generate a table of choice scenarios. It should accept subsets of the choosers and alternatives tables and return a choicemodels.tools.MergedChoiceTable.
- **probs_callable** (*callable*) – Callable that generates predicted probabilities for a table of choice scenarios. It should accept a choicemodels.tools.MergedChoiceTable and return a pd.Series with indexes matching the input.
- **alt_capacity** (*str, optional*) – Name of a column in the alternatives table that expresses the capacity of alternatives. If not provided, each alternative is interpreted as accommodating a single chooser.
- **chooser_size** (*str, optional*) – Name of a column in the choosers table that expresses the size of choosers. Choosers might have varying sizes if the alternative capacities are amounts rather than counts – e.g. square footage or employment capacity. Chooser sizes must be in the same units as alternative capacities. If not provided, each chooser has a size of 1.
- **max_iter** (*int or None, optional*) – Maximum number of iterations. If None (default), the algorithm will iterate until all choosers are matched or no alternatives remain.
- **chooser_batch_size** (*int or None, optional*) – Size of the batches for processing smaller groups of choosers one at a time. Useful when the anticipated size of the merged choice tables (choosers X alternatives X covariates) will be too large for python/pandas to handle.

Returns List of chosen alternative id's, indexed with the chooser (observation) id.

Return type pd.Series

1.5 Distance utilities API

ChoiceModels also includes tools for constructing pairwise distance matrices and calculating which geographies are within various distance bands of some reference geography.

1.5.1 Distance matrices

choicemodels.tools.**great_circle_distance_matrix**(*df*, *x*, *y*, *earth_radius=6371009*, *return_int=True*)

Calculate a pairwise great-circle distance matrix from a DataFrame of points. Distances returned are in units of earth_radius (default is meters).

Parameters

- **df** (*pandas DataFrame*) – a DataFrame of points, uniquely indexed by place identifier (e.g., tract ID or parcel ID), represented by x and y coordinate columns
- **x** (*str*) – label of the x coordinate column in the DataFrame
- **y** (*str*) – label of the y coordinate column in the DataFrame
- **earth_radius** (*numeric*) – radius of earth in units in which distance will be returned (default is meters)
- **return_int** (*bool*) – if True, convert all distances to integers

Returns Multi-indexed distance vector in units of df's values, with top-level index representing "from" and second-level index representing "to".

Return type pandas Series

choicemodels.tools.**euclidean_distance_matrix**(*df*)

Calculate a pairwise euclidean distance matrix from a DataFrame of points. Distances returned are in units of x and y columns.

Parameters **df** (*pandas DataFrame*) – a DataFrame of points, uniquely indexed by place identifier (e.g., tract ID or parcel ID), represented by x and y coordinate columns

Returns Multi-indexed distance vector in units of df's values, with top-level index representing "from" and second-level index representing "to".

Return type pandas Series

choicemodels.tools.**distance_matrix**(*df*, *method='euclidean'*, *x='lng'*, *y='lat'*, *earth_radius=6371009*, *return_int=True*)

Calculate a pairwise distance matrix from a DataFrame of two-dimensional points.

Parameters

- **df** (*pandas DataFrame*) – a DataFrame of points, uniquely indexed by place identifier (e.g., tract ID or parcel ID), represented by x and y coordinate columns
- **method** (*str*) – {'euclidean', 'greatcircle', 'network'} which algorithm to use for calculating pairwise distances
- **x** (*str*) – if method='greatcircle' or 'network', label of the x coordinate column in the DataFrame
- **y** (*str*) – if method='greatcircle' or 'network', label of the y coordinate column in the DataFrame

- **earth_radius** (*numeric*) – if method='greatcircle', radius of earth in units in which distance will be returned (default is meters)
- **return_int** (*bool*) – if method='greatcircle', if True, convert all distances to integers

Returns Multi-indexed distance vector in units of df's values, with top-level index representing "from" and second-level index representing "to".

Return type pandas Series

1.5.2 Distance bands

choicemodels.tools.**distance_bands**(*dist_vector*, *distances*)

Identify all geographies located within each distance band of each geography.

The list of distances is treated pairwise to create distance bands, with the first element of each pair forming the band's inclusive lower limit and the second element of each pair forming the band's exclusive upper limit. For example, if distances=[0, 10, 30], band 0 will contain all geographies with a distance >= 0 and < 10 units (e.g., meters) from the reference geography, and band 1 will contain all geographies with a distance >= 10 and < 30 units from the reference geography.

To make the final distance band include all geographies beyond a certain distance, make the final value in the distances list np.inf.

Parameters

- **dist_vector** (*pandas Series*) – Multi-indexed distance vector in units of df's values, with top-level index representing "from" and second-level index representing "to".
- **distances** (*list*) – a list of distance band increments

Returns a series multi-indexed by geography ID and distance band number, with values of arrays of geography IDs with the corresponding distances from that ID

Return type pandas Series

Index