Assisted specification with Biogeme

Michel Bierlaire Nicola Ortelli

July 7, 2022

Report TRANSP-OR 220707
Transport and Mobility Laboratory
School of Architecture, Civil and Environmental Engineering
Ecole Polytechnique Fédérale de Lausanne
transp-or.epfl.ch

Series on Biogeme

The package Biogeme (`biogeme.epfl.ch`) is designed to estimate the parameters of various models using maximum likelihood estimation. It is particularly designed for discrete choice models.

This document describes how to use Biogeme to apply the methodology for assisted specification described by Ortelli et al. (2021). In a nutshell, an optimization algorithm is used to generate models based on a minimal number of inputs provided by the analyst. These inputs are used to build a space of possible specifications that may contain any form of variable interaction, nonlinear transformation, segmentation of the population in the dataset and potential choice models; the space is then explored by an algorithm that sequentially introduces small modifications to an initial set of promising specifications.

We assume that the reader is already familiar with discrete choice models, and has successfully installed Biogeme. Biogeme is a Python package written in Python and C++, that relies on the Pandas library for the management of the data. This document has been written using Biogeme 3.2.9.

1 Modeling elements

We first define the modeling elements that are combined for the specification of a discrete choice model.

Alternatives An alternative is an element of the choice set $\mathcal{C}$.

Variable A variable is an explanatory, or independent, variable that is present in the data set.

Characteristic A characteristic is a variable that does not vary across alternatives. Typical characteristics are the socio-economic characteristics of the decision-maker (age, income, gender, etc.), and the variables describing the choice context (day of the week, weather conditions, etc.) In our context, they are used to segment the population into mutually exclusive segments.

Attribute An attribute is a variable that does vary across alternatives.

Group of attributes A group captures the same attribute in different utility functions. All attributes in the same group play similar roles in the specification of the model. A group of attributes is said to be *active* if it is involved in the specification of the choice model.

Coefficient When the utility function is a linear combination of attributes, the coefficients of this combination are unknown parameters to be estimated.

Segmentation The population can be segmented using one or several characteristics. In that case, a different coefficient is associated with each segment of the population.

Generic group of attributes A group of attributes is said to be generic if the same coefficient is used for each attribute in the group.

Alternative specific group of attributes A group of attributes that is not generic is said to be "alternative specific".

Transformation A transformation is a function that transforms the value of the attributes in a group. It may involve unknown parameters and nonlinear specifications.

2 A step by step specification

We describe now how to prepare a Python script for the assisted specification of a choice model. We use the Swissmetro example to illustrate each step.

1. Data preparation: this step is exactly the same as for any Biogeme model. An object of type biogeme.database has to be defined. Each variable is associated with an object of type biogeme.expressions.Variable, which can be a column of the original dataset, or a new variable defined by the user. We refer the reader to the documentation of Biogeme and to the online examples for more details.

2. Dictionary of attributes: the attributes that may be involved in the model specification are identified and named, in the form of a dictionary. The keys of the dictionary are the names, and the values are the corresponding biogeme.expressions.Variable. For example,

```
attributes = {
  'Train travel time': TRAIN_TT_SCALED,
  'Swissmetro travel time': SM_TT_SCALED,
  'Car travel time': CAR_TT_SCALED,
  'Train travel cost': TRAIN_COST,
  'Swissmetro travel cost': SM_COST,
  'Car travel cost': CAR_COST,
  'Train headway': TRAIN_HE,
  'Swissmetro headway': SM_HE,
}
```

3. Dictionary of groups of attributes: the keys of the dictionary are the names of the group. The values are a list of names of attributes. Note that no attribute can appear in more than one group.

```
groupsOfAttributes = {
  'Travel time': [
    'Train travel time',
    'Swissmetro travel time',
    'Car travel time',
  ],
  'Travel cost': [
    'Train travel cost',
    'Swissmetro travel cost',
    'Car travel cost',
  ],
  'Headway': ['Train headway', 'Swissmetro headway'],
}
```

4. The algorithm tries to change the status of groups of attributes from generic to alternative specific. It is possible to force some groups to always be alternative specific by defining the list of their names. If there is no such restriction, set the list to None.

```
genericForbiden = None
```

5. The algorithm tries to include or not each group of attributes in the model. It is possible to force some groups to always be active, that is, to be in the model, by defining the list of their names. If there is no such restriction, set the list to None.

```
forceActive = ['Travel time', 'Travel cost']
```

6. Define possible transformations of the attributes. Each of these transformations is characterized by a Python function that takes a real number as input, and returns a tuple with two elements:

   (a) the name of the transformation,

   (b) a Biogeme expression (that is, an object of type biogeme.expressions) to calculate the transformation.

```
def mylog(x):
"""Log of the attribute, or 0 if it is zero"""
return 'log', Elem({0: log(x), 1: Numeric(0)}, x == 0)
```

   More examples are discussed in Section 5. Then, associate each group of attributes with possible transformations. Note that the option not to transform the attribute is automatically considered. Define a dictionary where the keys are the names of the groups of attributes, and the values are lists of functions defined in the previous step.

```
transformations = {
  'Travel time': [
    mylog,
    sqrt,
    square,
    piecewise_time_1,
    piecewise_time_2,
    boxcox_time,
  ],
  'Travel cost': [
    mylog,
    sqrt,
    square,
    piecewise_cost_1,
```

```
        piecewise_cost_2,
        boxcox_cost,
      ],
      'Headway': [mylog, sqrt, square, boxcox_headway],
    }
```

7. Define the potential segmentations. A segmentation is based on a discrete characteristic z that can take several values: $z_1, \ldots, z_\ell$. It is defined by a tuple with two elements:

   (a) an object of type biogeme.expressions.Variable that captures the characteristic,

   (b) a dictionary associating each possible value with a name describing it.

   As a group of attributes can potentially be associated with several segmentations, we define a dictionary where the keys are the names of the segmentations, and the values are the tuples described above.

```
segmentations_cost = {
    'GA': DiscreteSegmentationTuple(
        variable=GA,
        mapping={1: 'GA', 0: 'noGA'}
    ),
    'gender': DiscreteSegmentationTuple(
        variable=MALE,
        mapping={0: 'female', 1: 'male'}
    ),
    'income': DiscreteSegmentationTuple(
        variable=INCOME,
        mapping={
            1: 'inc-under50',
            2: 'inc-50-100',
            3: 'inc-100+',
            4: 'inc-unknown',
        },
    ),
    'class': DiscreteSegmentationTuple(
        variable=FIRST,
        mapping={0: 'secondClass', 1: 'firstClass'}
    ),
    'who': DiscreteSegmentationTuple(
        variable=WHO,
        mapping={1: 'egoPays', 2: 'employerPays', 3: 'fiftyFifty'},
    ),
}
```

Each of these potential segmentations is then associated with a name, using a dictionary. It also clarifies if the segmentation should be done for each characteristic separately, or if all combinations of all possible values of the characteristics should be used for the segmentation. In the latter case, combinatorial must be set to True.

```
segmentations = {
    'Seg. cte': SegmentedParameterTuple(
        dict=segmentations_cte, combinatorial=False
    ),
    'Seg. cost': SegmentedParameterTuple(
        dict=segmentations_cost, combinatorial=False
    ),
    'Seg. time': SegmentedParameterTuple(
        dict=segmentations_time, combinatorial=False
    ),
    'Seg. headway': SegmentedParameterTuple(
        dict=segmentations_headway, combinatorial=False
    ),
}
```

Note that this feature allows the algorithm to investigate interactions between attributes and discrete characteristics. In order to model interactions between attributes and continuous characteristics, see the example provided in Section 5.

8. Specification of the utility functions as a list of terms. Each term is a tuple with the following elements:

   (a) the name of an attribute, or None for the alternative specific constant,

   (b) the name of a list of segmentations,

   (c) the bounds on the associated coefficients, in the form of a tuple (lower_bound, upper_bound), where each of these entries can be set to None is no bound is needed,

   (d) a function that verifies the validity of the estimated coefficient. Each model where the estimated value of the parameter is invalid is rejected. The function takes a value as input and returns a boolean. For instance, if we expect a coefficient to be negative, we can define the following function:

   ```
   def negativeParameter(val):
       return val < 0
   ```

Note that the bounds apply to the coefficients of each segment of the population. Here is an example of the definition of the utility function:

```
utility_train = [
    TermTuple(
        attribute=None,
        segmentation='Seg. cte',
        bounds=(None, None),
        validity=None,
    ),
    TermTuple(
        attribute='Train travel time',
        segmentation='Seg. time',
        bounds=(None, 0),
        validity=None,
    ),
    TermTuple(
        attribute='Train travel cost',
        segmentation='Seg. cost',
        bounds=(None, 0),
        validity=None,
    ),
    TermTuple(
        attribute='Train headway',
        segmentation='Seg. headway',
        bounds=(None, 0),
        validity=None,
    ),
]
```

Then, associate each utility function with the ID of the alternative, and with a name. Define a dictionary such that the keys are the ID of the alternatives, and the values are a tuple with the following elements:

(a) the name of the alternative,

(b) a list describing the specification of the utility function, as described in the previous step.

For example,

```
    utilities = {
      1: ('train', utility_train),
      2: ('Swissmetro', utility_sm),
      3: ('car', utility_car),
    }
```

9. Define the availability condition, in the exact same way as for any Biogeme specification, that is a dictionary where the keys are the ID of the

alternatives, and the values are Biogeme expressions (biogeme.expressions.Expression). For example,

```
availabilities = {
  1: TRAIN_AV_SP,
  2: SM_AV,
  3: CAR_AV_SP
}
```

10. We define potential candidates for the choice model. Each candidate is a function that takes as input three arguments:

    (a) the dictionary of utility functions,

    (b) the dictionary of availability conditions, and

    (c) the expression to calculate the chosen alternative.

    It returns, as output, an expression representing the contribution of each observation to the log likelihood function. For example,

```
def nested1(V, av, choice):
    existing = Beta('mu_existing', 1, 1, None, 0), [1, 3]
    future = 1.0, [2]
    nests = existing, future
    return models.lognested(V, av, nests, choice)
```

    Each of these functions is associated with a name in a dictionary:

```
myModels = {
  'Logit': logit,
  'Nested one stop': nested1,
  'Nested same': nested2,
  'CNL alpha fixed': cnl1,
  'CNL alpha est.': cnl2,
}
```

11. The last step consists in defining the optimization problem, gathering the ingredients defined above. An object of type biogeme.assisted.specificationProblem is created, with the following arguments for the constructor:

    (a) the name of the problem,

    (b) the biogeme.database object containing the data (from step 1),

    (c) the dictionary of all attributes, defined at step 2,

    (d) the dictionary of all groups of attributes, defined at step 3,

    (e) the list of groups that must be alternative specific, defined at step 4,

(f) the list of groups that must be in the model, defined at step 5,

(g) the dictionary of transformations of attributes, defined at step 6,

(h) the dictionary of segmentations, defined at step 7,

(i) the dictionary of utility functions, defined at step 8,

(j) the dictionary of availability conditions, defined at step 9,

(k) the Biogeme expression calculating the chosen alternative,

(l) the dictionary of possible choice models, defined at step 10.

For example,

```
theProblem = assisted.specificationProblem(
    'Swissmetro',
    database,
    attributes,
    groupsOfAttributes,
    genericForbiden,
    forceActive,
    transformations,
    segmentations,
    utilities,
    availabilities,
    CHOICE,
    myModels,
)
```

3 Running the algorithm

In order to run the algorithm, we need to provide initial specifications that the algorithm tries to improve. A specification is characterized by two dictionaries. The first one associates each group of attributes with the index of a transformation (or None) and a boolean that is True if the coefficient is generic, and False if it is alternative specific. For instance,

```
attr = {
    'Travel time': (1, False),
    'Travel cost': (0, True),
    'Headway': (0, False),
}
```

If n transformations have been associated with a group of attributes, the index must be between 0 and $n-1$. The second dictionary activates specific dimensions of the segmentations. For instance, the following dictionary activates one dimension for the constant and the headway coefficients, and two

dimensions for the cost coefficient. The time coefficient is not segmented, and is the same for all observations.

```
sg = {
    'Seg. cte': ['GA'],
    'Seg. cost': ['class', 'who'],
    'Seg. time': [],
    'Seg. headway': ['class'],
}
```

The actual model is generated by the function generateSolution that takes three arguments:

1. the dictionary describing the transformation, as introduced above,
2. the dictionary describing the segmentations, as introduced above,
3. the name of the choice model.

For instance, we can create five models, with the same specification for the utility functions, but different assumptions for the error term as follows:

```
initSolutions = [
    theProblem.generateSolution(attr, sg, 'Logit'),
    theProblem.generateSolution(attr, sg, 'Nested one stop'),
    theProblem.generateSolution(attr, sg, 'Nested same'),
    theProblem.generateSolution(attr, sg, 'CNL alpha fixed'),
    theProblem.generateSolution(attr, sg, 'CNL alpha est.'),
]
```

The algorithm investigates many specifications, and estimates the corresponding set of parameters. All these specifications are stored in a pickle file (a binary representation of Python objects). Therefore, the algorithm can be interrupted at any time, and restarted at the point it was interrupted. Moreover, the pickle file contains the Pareto optimal models, as discussed later.

The following instruction launches the algorithm:

```
vns.vns(
    theProblem,
    initSolutions,
    archiveInputFile='swissmetroPareto.pickle',
    pickleOutputFile='swissmetroPareto.pickle',
)
```

Note that it is simpler to use the same pickle file for input and output, to avoid any specific manipulation between two runs. For the first run, when no pickle file is available yet, simply define archiveInputFile=None.

4 Exploring the Pareto set

The algorithm generates a pickle file where the Pareto optimal solutions are stored. These solutions can be explored using the following statements:

```
from biogeme import vns
pickleFile = 'swissmetroPareto.pickle'
pareto = vns.paretoClass(
    0,
    archiveInputFile=pickleFile
)
for p in pareto.pareto:
    print('----------------------------')
    print(p)
```

For each model, it prints the name of the model, the description of the specification of the utility function, the value of the final log likelihood function and the number of parameters, as in the following examples:

```
Nested one stop-----------------------
Alternative train [1]
--------------------

Cte. <GA, gender>
Train travel time_log [alt. spec.] <GA>
Train travel cost_log [alt. spec.] <class, who>
Train headway_log [generic] <class>
--------------------------

Alternative Swissmetro [2]
--------------------------

Cte. <GA, gender>
Swissmetro travel time_log [alt. spec.] <GA>
Swissmetro travel cost_log [alt. spec.] <class, who>
Swissmetro headway_log [generic] <class>
----------------------

Alternative car [3]
-------------------

Car travel time_log [alt. spec.] <GA>
Car travel cost_log [alt. spec.] <class, who>

Neg. log likelihood: 7786.078102793665
#parameters: 32
```

Actually, the pickle file contains three sets of models:

- the set of Pareto optimal solutions that are not dominated by any other model (illustrated above),
- the set of considered solutions that contains all models that have been investigated by the algorithm,

- the set of removed solutions that contains the models that have been at some point in the Pareto set, but have been removed when a better model has been found.

It is interesting to visualize these three sets using the following code:

```
import matplotlib.pyplot as plt
from biogeme import vns

pickleFile = 'swissmetroPareto.pickle'
pareto = vns.paretoClass(0, archiveInputFile=pickleFile)

objectives = list(pareto.pareto)[0].objectivesNames
par_obj = [p.objectives for p in pareto.pareto]
par_x, par_y = zip(*par_obj)
con_obj = [p.objectives for p in pareto.considered]
con_x, con_y = zip(*con_obj)
rem_obj = [p.objectives for p in pareto.removed]
rem_x, rem_y = zip(*rem_obj)

x_buffer = 10
y_buffer = 0.1

plt.axis(
    [
        min(par_x) - x_buffer,
        max(par_x) + x_buffer,
        min(par_y) - y_buffer,
        max(par_y) + y_buffer,
    ]
)
plt.plot(par_x, par_y, 'o', label='Pareto')
plt.plot(rem_x, rem_y, 'x', label='Removed')
plt.plot(con_x, con_y, ',', label='Considered')
plt.xlabel(objectives[0])
plt.ylabel(objectives[1])
plt.legend()
plt.show()
```

The resulting plot is represented in Figure 1, where the x-axis represents the value of the negative log likelihood, and the y-axis represents the number of parameters. The idea is that each of these objectives should be as small as possible. The trade-off between these two objectives is well illustrated by this figure.

The results of the algorithm allow to better appreciate the trade-off between the goodness of fit and the parsimony. But the final decision about which model in the Pareto set must be preferred lies with the modeler.

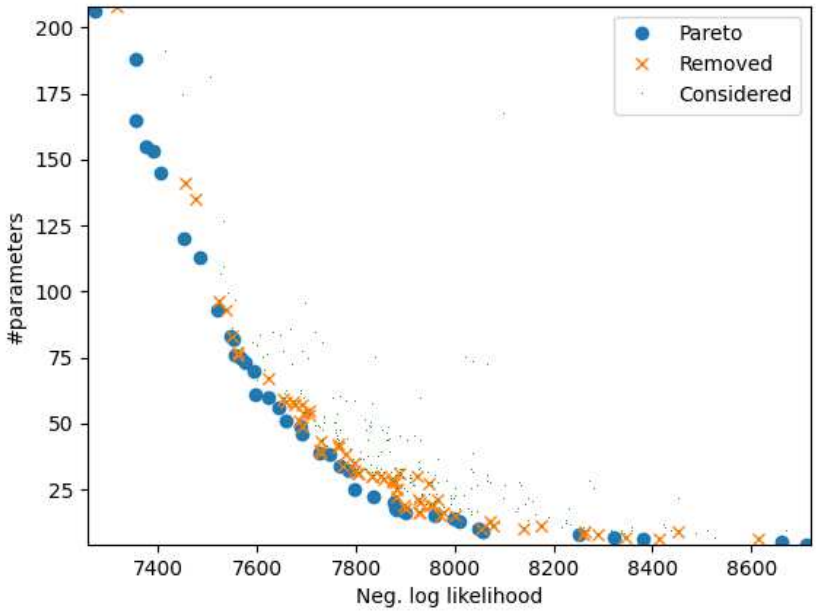

Figure 1: Illustration of the models investigated by the algorithm

5 Transformation of variables

The transformation of variables provides a great flexibility to investigate various specifications of a choice model. In general, it is used to investigate nonlinear specifications, but it does not have to.

We provide here some examples of transformations.

First, any simple transformation can be investigated. For instance, the square of the variable:

```
def square(x):
    return 'square', x ** 2
```

The Box-Cox transformation can also be investigated. In that case, it involves an additional parameter to be estimated. We first define a generic Box-Cox function, as follows.

```
def boxcox(x, name):
    ell = Beta(f'lambda_{name}', 1, None, None, 0)
    return f'Box-Cox_{name}', models.boxcox(x, ell)
```

It defines first a parameter, which is initialized at the value 1. Then, it combines the name of the transformation, with the Box-Cox model implemented in the Biogeme module models.

This generic function can now be used to implement several specific transformations. For instance, the Box-Cox transformation of the travel time variable can be coded as follows:

```
def boxcox_time(x):
    return boxcox(x, 'time')
```

Similarly, the Box-Cox transformation of the cost variable can be coded as follows:

```
def boxcox_cost(x):
    return boxcox(x, 'cost')
```

The reason why we need two different functions is to associate a different parameter with each transformation.

A similar implementation can be used to investigate piecewise linear specifications. Indeed, different variables are associated with different thresholds for such a specification. We first implement a generic function, that also relies on the Biogeme module models:

```
def piecewise(x, thresholds, name):
    piecewiseVariables = models.piecewiseVariables(x, thresholds)
    formula = piecewiseVariables[0]
    for k in range(1, len(thresholds) - 1):
        formula += (
            Beta(
```

```
                f'pw_{name}_{thresholds[k-1]}_{thresholds[k]}',
                0,
                None,
                None,
                0,
            )
            * piecewiseVariables[k]
        )
    return (f'piecewise_{name}_{thresholds}', formula)
```

Now, we can define specific transformations of the same variable. For example, we transform the time variable based on the threshold $[0, 0.1, \infty[$ as follows:

```
def piecewise_time_1(x):
    return piecewise(x, [0, 0.1, None], 'time')
```

The transformation of the same variable with a different list of thresholds is obtained as follows:

```
def piecewise_time_2(x):
    return piecewise(x, [0, 0.25, None], 'time')
```

Transformation of variables can also be used to investigate interactions with continuous characteristics. In the following example, the variable distance_km is a continuous characteristic. It provides the distance between an origin and a destination in a mode choice context. In order to investigate its interaction with an attribute, the following transformation can be implemented:

```
from biogeme.expressions import Variable
def distanceInteraction(x):
    return (
        'dist. interaction',
        x * log(1 + Variable('distance_km') / 1000)
    )
```

If income is coded as a continuous variable, its interaction with an attribute can be coded as follows:

```
from biogeme.expressions import Variable
def incomeInteraction(x):
    return (
        'income interaction',
        x / Variable('income)
    )
```

References

Ortelli, N., Hillel, T., Pereira, F., de Lapparent, M. and Bierlaire, M. (2021). Assisted specification of discrete choice models, *Journal of Choice Modelling* **39**(100285).