Apollo: a flexible, powerful and customisable freeware package for choice model estimation and application

version 0.0.8

User manual

www.ApolloChoiceModelling.com

Stephane Hess & David Palma
Choice Modelling Centre
University of Leeds

9 September 2019

While the Apollo package is the result of many years of development, the core of this work was carried out under the umbrella of the European Research Council (ERC) funded consolidator grant 615596-DECISIONS. We are grateful to the many colleagues who provided suggestions and/or tested the code extensively, including Chiara Calastri, Romain Crasted dit Sourd, Andrew Daly, Jeff Dumont, Joe Molloy and Basil Schmid. We would like to especially thank Thijs Dekker for his contributions to precursors of Apollo and his guidance on the EM algorithm, Thomas Hancock for his implementation of Decision Field Theory and Annesha Enam for her contributions on MDCEV without an outside good. We are also grateful to Kay Axhausen for making the Swiss public transport route choice dataset available.

Contents

List of Figures

List of Tables

1 Introduction

Choice modelling techniques have been used across different disciplines for over four decades (see McFadden 2000 for a retrospective and Hess and Daly 2014 for recent contributions and applications across fields). For the majority of that time, the number of users of especially the most advanced models was rather small, and similarly, a small number of software packages was used by this community. In the last two decades, the pool of users of choice models has expanded dramatically, in terms of their number as well as the breadth of disciplines covered. At the same time, we have seen the development of new modelling approaches, and gains in computer performance as well as software availability have given an ever broader group of users access to ever more advanced models.

These developments have also seen a certain fragmentation of the community in terms of software, which in part runs along discipline lines[1]. Notwithstanding the most advanced users who develop their own code for often their own models, there is first a split between the users of commercial software and those using freeware tools. The former are generally computationally more powerful but may have more limitations in terms of available model structures or the possibility for customisation. On the other hand, the latter may have limitations in terms of performance and user friendliness but may benefit from more regular developments to accommodate new model structures.

A further key differentiation between packages is the link between user inputs and interface and the actual underlying methodology. Many existing packages, both freeware and commercial, are black box tools where the user has little or no knowledge of what goes on "under the hood". While this has made advanced models accessible to a broader group of users, a disconnect between theory and software not only increases the risk of misinterpretations and misspecifications, but can also hide relevant nuances of the modelling process and mistakenly give the impression that choice models are "easy tools" to use. On the other hand, software that relies on users to code all components from scratch arguably imposes too high a bar in terms of access.

Software also almost exclusively allows the use of only either classical estimation techniques or Bayesian techniques. This fragmentation again runs largely in parallel with discipline boundaries and has only served to further contribute to the lack of interaction/dialogue between the classical and Bayesian communities.

A final difference arises in terms of software environment. While commercial software usually provides a custom user interface, freeware options in general (though not exclusively) rely on existing statistical or econometric software and are made available as packages within these. The latter at times means that *freeware* packages are not really free to use (if the host software is not), while there are also cases of software being accessible only in either Windows or Linux, not both.

The above points served in large part as the motivation for the development of *Apollo*. Our aims were:

Free access: *Apollo* is a completely free package which does not rely on commercial statistical software as a host environment.

[1] We intentionally do not refer to specific packages, so as not to risk any misrepresentations but also given the growing number of freeware tools, some of which we might not be aware of.

Big community: *Apollo* relies on `R` (R Core Team, 2017), which is very widely used across disciplines and works well across different operating systems.

Transparent, yet accessible: *Apollo* is neither a blackbox nor does it require expert econometric skills. The user can see as much or as little detail of the underlying methodology as desired, but the link between inputs and outputs remains.

Ease of use: *Apollo* combines easy to use `R` functions with new intuitive functions without unnecessary jargon or complexity.

Modular nature: *Apollo* uses the same code structure independently of whether the simplest multinomial logit model is to be estimated, or a complex structure using random coefficients and combining multiple model components.

Fully customisable: *Apollo* provides functions for many well known models but the user is able to add new structures and still make use of the overall code framework. This for example extends to coding expectation-maximisation routines.

Discrete and continuous: *Apollo* incorporates functions not just for commonly used discrete choice models but also for a family of models that looks jointly at discrete and continuous choices.

Novel structures: *Apollo* goes beyond standard choice models by incorporating the ability to estimate Decision Field Theory (DFT) models, a popular accumulator model from mathematical psychology.

Classical and Bayesian: *Apollo* does not restrict the user to either classical or Bayesian estimation but easily allows changing from one to the other.

Easy multi-threading: *Apollo* allows users to split the computational work across multiple processors without making changes to the model code.

Not limited to estimation: *Apollo* provides a number of pre and post-estimation tools, including diagnostics as well as prediction/forecasting capabilities and posterior analysis of model estimates.

While *Apollo* is easy to use, we also remain of the opinion that users of choice modelling software should understand the actual process that happens during estimation. For this reason, the user needs to explicitly include or exclude calls to specific functions that are model and dataset specific. For example, in the case of repeated choice data, the user needs to include a call to a function that takes the product across choices for the same person (`apollo_panelProd`). Or in the case of a mixed logit model, the user needs to include a call to a function that averages across draws (`apollo_avgInterDraws` and/or `apollo_avgIntraDraws`). If calls to these functions are missing when needed, or if a user makes a call to a function that should not be used in the specific model, the code will fail, and provide the user with feedback about why this happened. This is in our view much better than the software permitting users to make mistakes and fixing them behind the scenes.

Users of *Apollo* are asked to acknowledge the use of the software by citing the academic paper (Hess, S. & Palma, D. (2019), *Apollo*: a flexible, powerful and customisable freeware package for choice model estimation and application, Journal of Choice Modelling) and the manual for the version used in their work (e.g. Hess, S. & Palma, D. (2019), *Apollo* version 0.0.8, user manual, www.ApolloChoiceModelling.com).

Apollo is the culmination of many years of development of individual choice modelling routines, starting with code developed by Hess while at Imperial College (cf. Hess, 2005) using Ox (Doornik, 2001). This code was gradually transitioned to `R` at the University of Leeds, with substantial further developments once Palma joined the team in Leeds, bringing with him ideas developed at Pontificia Universidad Católica de Chile (cf. Palma, 2016). No code is an island, and we have been inspired especially by ALogit (ALogit, 2016) and Biogeme (Bierlaire, 2003), and *Apollo* mirrors at least some of their features.

This manual presents an overview of the capabilities of the *Apollo* package and serves as a user manual. It is accompanied online[2] by numerous example files (some of which are used in this manual) and a number of free to use datasets. In adddition to the detailed information in this manual, users can also obtain help on specific functions directly in R, using e.g. `?apollo_mnl` for help on the `apollo_mnl` function.

In line with our earlier point about other software, this manual does not include any comparisons with other packages, in terms of capabilities or speed. The code has been widely tested to ensure accuracy. In our view, any speed comparison offers little practical benefit. For simple models, there is a clear advantage for highly specialised code, while, for complex models, any benchmarking is impacted substantially by the specific implementation and degree of optimisation used.

In the remainder of this manual, we do not provide details on common `R` functions and syntax used in the code, or how to run `R` code, and the reader is instead referred to R Core Team (2017). For the syntax shown in this manual, it is just worth noting that in `R`, a line starting with one or more `#` characters is a comment. We tend to use a single `#` for optional lines that a user can comment in and out, and `###` for actual comments. In addition, two other points are worth raising. In complex models, the `R` syntax file for *Apollo* can become quite large, and a user may wish to split this into separate files, e.g. one for loading and processing the data, one for the actual model definition, etc, and then have a master file which calls the individual files (using `source`). Secondly, for the predefined functions, the order of arguments passed to the function should be kept in the order specified in this manual[3].

Another point to raise concerns the specific naming conventions we have adopted for functions and inputs to functions. All functions within the code start with the prefix apollo_. This is then followed by the "name" of the actual function in a single word, where any new part of the name starts with a capital letter, for example `apollo_modelOutput`. The prefix `apollo_` is also used for four key non-function objects in the code, namely the user defined settings `apollo_control`, `apollo_HB` and `apollo_draws`, the list of parameters `apollo_beta` and fixed parameters `apollo_fixed`, and the code generated combined inputs variable `apollo_inputs`. The functions in *Apollo* take numerous inputs and for ease of programming, these are often combined

[2] www.ApolloChoiceModelling.com

[3] Unless a user explicitly prefaces each argument with the name used in the function. For example, if a function is defined to take two inputs, namely `dependent` and `explanatory`, e.g. `model_prob(dependent,explanatory)`, and the user wants to use `choice` and `utility` as the inputs, then the function can be called as `model_prob(choice,utility)` but not as `model_prob(utility,choice)`. The latter change in order is only possible if the function is called explicitly as `model_prob(explanatory=utility,dependent=choice)`, which is the same as `model_prob(dependent=choice,explanatory=utility)`.

into a *list* object. The naming convention used for these is to have the name of the function (without the `apollo_` prefix) followed by `_settings`, for example in `modelOutput_settings`. Finally, individual variables/settings do not have a prefix and again use the convention of capitalising the first letter of any new word except for the start, for example in `printDiagnostics`.

Before we proceed, a brief explanation is needed as to our choice of the name *Apollo*. Several existing packages refer to specific models in their name (e.g. ALogit, NLogit) which is not applicable in our case given the wider set of models we cover. We failed miserably in our efforts to come up with an imaginative acronym like Biogeme and so went back to Greek mythology. The obvious choice would have been Cassandra, with her gift of prophecy and the curse that nobody listened to her (a bit like choice modellers trying to sell their ideas to policy makers). Alas, the name has already been used for a large database package, so we resorted to Apollo, the Greek god of prophecy who gave this gift to Cassandra in the first place.

The remainder of this manual is organised as follows. The following section talks about installation before Section 3 introduce a number of datasets used throughout the manual. Section 4 provides an in-depth introduction to the code structure, using the example of a simple Multinomial Logit model. This is followed in Section 5 by an overview of other available model components, and a description of how the user can add his/her own models. Section 6 covers random heterogeneity, both discrete and continuous while Section 7 discusses joint estimation of multiple model components, with a focus on hybrid choice models. Bayesian estimation is covered in Section 8 with (mainly) post-estimation capabilities discussed in Section 9. Finally, a few extensions are discussed in Section 10. A number of appendices are also included. Appendix A summarises changes across different versions of *Apollo*. Appendix B contains data dictionaries, Appendix C a list of the example files and Appendix D an index of functions and variables in *Apollo*.

2 Installing *Apollo*, loading the libraries and running the code

Apollo runs in R, with a minimum `R` version of 3.1.0. *Apollo* can be installed in two ways. If an internet connection is available, the easiest way to install it is to type the following command into the `R` console. This will also install all dependencies, i.e. other routines used by the *Apollo* package[4].

```
install.packages("apollo")
```

The second way is to install it from a file. A file containing the source code can be obtained at www.ApolloChoiceModelling.com. Then, the following command must be typed into the `R` console.

```
install.packages("C:\...\apollo_v0.0.8.tar.gz", repos = NULL, type = "source")
```

where `C:\...\apollo_v0.0.8.tar.gz` must be replaced by the correct path to the file in the user's computer, using the version that was downloaded. This will not automatically install dependencies.

The installation of the package does not need to be repeated every time `R` is started nor every time a model is to be estimated. Instead, it only need to be done once (unless R itself is updated, then the installation must be repeated).

Every time users want to estimate a model, they should load *Apollo* into memory. This can be achieved by simply running the following line of code in R, or by including it in the source file of each model, prior to running any *Apollo* functions.

```
library(apollo)
```

Users are encouraged to check for updated versions of the package every few months. Updates, when available, can be acquired by simply re-installing the package. Installation from CRAN will install the latest release. Previous releases will be available from the software website, where users also have access to versions with new features that are under development prior to a full release. These versions need to be compiled locally, and users require Rtools for this purpose.

Most users will run `R` from a shell such as `RStudio` (RStudio Team, 2015). A full *Apollo* model file, or any other `R` script, can also be run from the command line, without accessing `R` directly. This can be useful when running many scripts unattended, or when submitting jobs to a computer cluster. The command to do this changes depending on the operation system and the local directory structure. In Linux, the command is as follows: `R CMD BATCH model.R`. In Windows, the command is for example as follows: `"C:\Program Files\R\R-3.5.1\bin\R.exe" CMD BATCH model.R`. Note that in both cases the working directory should be set within the model file using the `setwd` function. The output that would normally be printed to the `R` Terminal will instead be written in a file called `model.Rout`, which can be opened with any plain text editor.

[4]For installation on macOS, users should install from binaries, rather than source.

3 Data format and datasets used for examples

Apollo makes use of a format where all relevant information for a given observation is stored in the same row. Using a simple discrete choice context, this would imply that the data for all alternatives is included in the same row, rather than one row per alternative. Some choice modellers refer to this as the *wide* format, as opposed to the *long* format, which would have one row per alternative. This terminology is however not very helpful as, in the context of repeated measurements data, the term *wide* refers to a format where all measurements for the same person are included in one line. In the one row per observation format in a choice modelling context, there will still be multiple rows for different choices for the same person.

This section presents a number of datasets used throughout the manual and in the online examples. All datesets are saved as comma separated (*csv*) files, with details on variable names are provided in Appendix B.

3.1 RP-SP mode choice dataset: `apollo_modeChoiceData.csv`

Our first dataset is a synthetic dataset looking at mode choice for 500 travellers. For each individual, the data contains two revealed preference (RP) inter-city trips, where the possible modes were car, bus, air and rail, and where each individual had at least two of these four modes available to them. The journey options were described on the basis of access time (except for car), travel time and cost, with times in minutes, and costs in £. The data then also contains 14 stated preference (SP) tasks per person, using the same alternatives as those available on the RP journey for that person, but with an additional categorical quality of service attribute added in for air and rail, taking three levels, namely *no frills*, *wifi available*, or *food available*. For each individual, the dataset additionally contains information on gender, whether the journey was a business trip or not, and the individual's income.

3.2 SP route choice dataset: `apollo_swissRouteChoiceData.csv`

Our second dataset comes from an actual SP survey of public transport route choice conducted in Switzerland (Axhausen et al., 2008). A set of 388 people were faced with 9 choices each between two public transport routes, both using train (leading to 3,492 observations in the data). The two alternatives were described on the basis of travel time, travel cost, headway (time between subsequent trains/busses) and the number of interchanges. For each individual, the dataset additionally contains information on income, car availability in the household, and whether the journey was made for commuting, shopping, business or leisure.

3.3 Health attitudes SP: `apollo_drugChoiceData.csv`

Our third dataset is a synthetic dataset looking at drug choices for the treatment of headaches for 1,000 individuals. For each person, the data contains 10 SP tasks, each giving a choice between four alternatives, the first two being products by recognised drug companies while the final two are generic products. In each choice task, a full ranking of the four alternatives is given. The drugs are described in terms of brand (two recognised brands and three generic brands), country

of origin (six countries), drug features (three types of features), risk of side effects and price. The possible levels for the attributes differ between the first two (branded) and last two (generic) alternatives. For each individual, the dataset additionally contains answers to four attitudinal questions as well as information on whether an individual is a regular user, their education and their age.

3.4 Time use data: `apollo_timeUseData.csv`

Our fourth dataset comes from a revealed preferences survey on time use conducted in the UK (Calastri et al., 2019). A set of 447 individuals completed a digital activity log for up to 14 days, providing 2,826 days of data (first day discarded for each person). For each day, the amount of time spent in each of twelve activities is recorded, as well as some of the individual's characteristics, the weather during the day, and information about land use at the individual's home location. The activities considered were dropping-off or picking-up, working, going to school, shopping, private business, getting petrol, social or leisure activities, vacation, doing exercise, being at home, travelling, and a last activity grouping the time allocated to other activities by the individual.

4 General code structure and components: illustration for MNL

The structure of an *Apollo* model file varies across specifications, but a general overview is shown in Figure 1.

In this section, we provide an introduction to the general capabilities of the *Apollo* package by using the example of a Multinomial Logit (MNL) model (McFadden, 1974) on the simple mode choice stated preference survey introduced in Section 3.1, where we use the SP part of this data, i.e. 14 choices each for 500 individuals. This example is available in the file `Apollo_example_3.r` and uses a very detailed specification of the utility function. More barebones examples are also available in `Apollo_example_1.r` and `Apollo_example_2.r`, which are models without any socio-demographics, estimated on the `RP` and `SP` data, respectively.

4.1 Initialising the code

The first step in every use of *Apollo* is to initialise the code. These steps are illustrated in Figure 2. In an optional step, we clear the memory/workspace by using `rm(list = ls())`, before loading the *Apollo* library. This is followed by calling the `apollo_initialise` function, which 'detaches' variables[5] and makes sure that output is directed to the console rather than a file. This function is called without any arguments and does not return any output variables, i.e.:

```
apollo_initialise()
```

The user next sets a number of core controls, where in our case, we only give the name of the model (where any output files will use this name too), provide a brief description of the model (for use in the output) and indicate the name (in quotes) of the column in the data which contains the identifier variable for individual decision makers. Each time, the entry on the left is an *Apollo*-defined variable whose name is not to be changed, and the user provides the value on the right, followed by a comma, except for the last element.

Only this final setting in Figure 2, i.e. setting the individual ID, is a requirement without which the code will not run. For any other settings, the code will use default values when not provided by the user, as illustrated in Figure 6. These other settings include:

`mixing`: A boolean variable which needs to be set to TRUE when the model uses continuous random coefficients, as discussed in Section 6.1 (default is set to FALSE).

`nCores`: An integer setting the number of cores used during estimation discussed, as discussed in Section 6.4 (default is set to 1).

`workInLogs`: A boolean variable, which, when set to TRUE, means that the logs of probabilities are used when processing probabilities inside `apollo_probabilities`. This can avoid numerical issues with complex models and datasets where there are large numbers of observations per individual. This is only really useful with repeated choice, and slows down estimation (default is set to FALSE).

`seed`: An integer setting the seed used for any random number generation (default is 13).

[5] In `R`, a user can 'attach' an object, which means that individual components in it can be called by name.

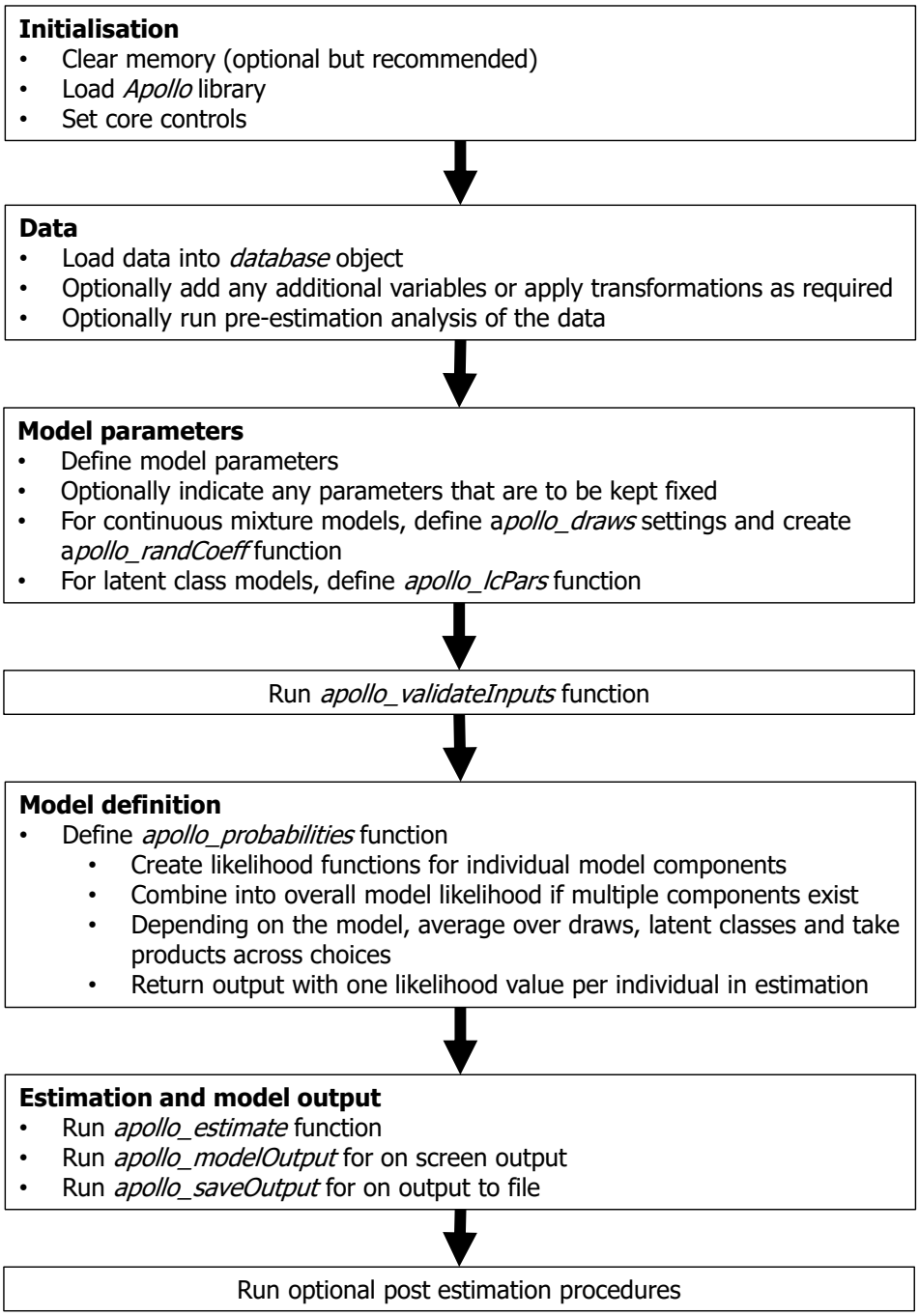

Figure 1: General structure of an *Apollo* model file

```
rm(list = ls())

library(apollo)

apollo_initialise()

apollo_control = list(
  modelName  ="Apollo_example_3",
  modelDescr ="MNL model with socio-demographics on mode choice SP data",
  indivID    ="ID"
)
```

Figure 2: Code initialisation

`HB`: A boolean variable which needs to be set to TRUE for using Bayesian estimation, as discussed in Section 8 (default is FALSE).

`noValidation` A boolean variable, which, when set to TRUE, means that no validation checks are performed (default is FALSE).

`noDiagnostics` A boolean variable, which, when set to TRUE, means that no model diagnostics are reported. This setting is provided primarily to avoid excessively verbose output with complex models using many components (cf. Section 7) but will be set to FALSE (default) for most models by most users.

`weights`: The name of a variable in the database containing weights for each observation, which can then be used in estimation if also using the function `apollo_weighting` (default is for weights to be missing).

4.2 Reading and processing the data

Figure 3 illustrates the process of loading the data, in this case from a *csv* file, working with only a subset of the data (in this case removing the RP observations) and creating additional variables in the data (in this case a variable with the mean income in the data). In our example, we read the data file from the working directory, which we had set to be the same as the directory with the model file. Users may need to add the path of the file depending on their local setup and file structure.

Three additional points need to be mentioned here. Firstly, the code is not limited to using *csv* files, and `R` allows the user to read in tab separated files too, for example[6]. Secondly, some applications may combine data from multiple files. The user can either combine the data outside of `R` or do so inside `R` using appropriate merging functions, but at the point of validating the user inputs (Section 4.4), all data needs to be combined in a single `R` data.frame called `database`. Thirdly, any new variables created by the user, such as mean income in our case, need to be created in the database object rather than the global environment, and this needs to happen prior to validating the user inputs.

[6]The reader is referred to R Core Team (2017).

```
database = read.csv("apollo_modeChoiceData.csv",header=TRUE)
database = subset(database,database$SP==1)
database$mean_income = mean(database$income)
```

Figure 3: Loading data, selecting a subset and creating an additional variable

4.3 Model parameters

In this simple model, we estimate alternative specific constants (ASCs), mode specific travel time coefficients, a cost and access time coefficient and dummy coded coefficients for the service quality attribute. In addition, we interact the constants with gender, allow for differences in the time and cost sensitivities for business travellers (generic across modes), and incorporate an income elasticity on the cost sensitivity.

With the above, the utilities for the four modes in choice situation t for individual n are given by:

$$
\begin{aligned}
U_{car,n,t} &= \delta_{car} \\
&+ (\beta_{tt,car} + \beta_{tt,business-shift} \cdot z_{business,n}) \cdot x_{tt,car,n,t} \\
&+ (\beta_{tc} + \beta_{tc,business-shift} \cdot z_{business,n}) \cdot \left(\frac{z_{income,n}}{\overline{z_{income}}} \right)^{\lambda_{income}} \cdot x_{tc,car,n,t} \\
&+ \varepsilon_{car,n,t} \\
U_{bus,n,t} &= \delta_{bus} + \delta_{bus,female-shift} \cdot z_{female,n} \\
&+ (\beta_{tt,bus} + \beta_{tt,business-shift} \cdot z_{business,n}) \cdot x_{tt,bus,n,t} \\
&+ (\beta_{tc} + \beta_{tc,business-shift} \cdot z_{business,n}) \cdot \left(\frac{z_{income,n}}{\overline{z_{income}}} \right)^{\lambda_{income}} \cdot x_{tc,bus,n,t} \\
&+ \varepsilon_{bus,n,t} \\
U_{air,n,t} &= \delta_{air} + \delta_{air,female-shift} \cdot z_{female,n} \\
&+ (\beta_{tt,air} + \beta_{tt,business-shift} \cdot z_{business,n}) \cdot x_{tt,air,n,t} \\
&+ (\beta_{tc} + \beta_{tc,business-shift} \cdot z_{business,n}) \cdot \left(\frac{z_{income,n}}{\overline{z_{income}}} \right)^{\lambda_{income}} \cdot x_{tc,air,n,t} \\
&+ \beta_{no\,frills} \cdot (x_{service,air,n,t} == 1) + \beta_{wifi} \cdot (x_{service,air,n,t} == 2) + \beta_{food} \cdot (x_{service,air,n,t} == 3) \\
&+ \varepsilon_{air,n,t} \\
U_{rail,n,t} &= \delta_{rail} + \delta_{rail,female-shift} \cdot z_{female,n} \\
&+ (\beta_{tt,rail} + \beta_{tt,business-shift} \cdot z_{business,n}) \cdot x_{tt,rail,n,t} \\
&+ (\beta_{tc} + \beta_{tc,business-shift} \cdot z_{business,n}) \cdot \left(\frac{z_{income,n}}{\overline{z_{income}}} \right)^{\lambda_{income}} \cdot x_{tc,rail,n,t} \\
&+ \beta_{no\,frills} \cdot (x_{service,rail,n,t} == 1) + \beta_{wifi} \cdot (x_{service,rail,n,t} == 2) + \beta_{food} \cdot (x_{service,rail,n,t} == 3) \\
&+ \varepsilon_{rail,n,t},
\end{aligned}
\tag{1}
$$

where all parameters are estimated except for δ_{car} and $\beta_{no\,frills}$, which are both fixed to a value of zero.

In the code, the user needs to define the parameters and their starting values, and also indicate whether any of the parameters are to be kept at their starting values. This process is illustrated in Figure 4. We first create an `R` object of the *named vector* type, called `apollo_beta`, with the name and starting value for each parameter, including any that are later on fixed to their starting values. In our case, we then keep two of these parameters, namely `asc_car` and `b_no_frills`, fixed to their starting values by including their names in the character vector `apollo_fixed`, where this vector is kept empty (`apollo_fixed = c()`) if all parameters are to be estimated.

```
apollo_beta=c(asc_car                 = 0,
              asc_bus                 = 0,
              asc_air                 = 0,
              asc_rail                = 0,
              asc_bus_shift_female    = 0,
              asc_air_shift_female    = 0,
              asc_rail_shift_female   = 0,
              b_tt_car                = 0,
              b_tt_bus                = 0,
              b_tt_air                = 0,
              b_tt_rail               = 0,
              b_tt_shift_business     = 0,
              b_acc                   = 0,
              b_cost                  = 0,
              b_cost_shift_business   = 0,
              cost_income_elast       = 0,
              b_no_frills             = 0,
              b_wifi                  = 0,
              b_food                  = 0
             )

apollo_fixed = c("asc_car","b_no_frills")
```

Figure 4: Setting names and starting values for model parameters, and fixing some parameters to their starting values

For complex models especially, it can sometimes be beneficial to read in starting values from an earlier model, albeit that users should be mindful that this can lead to problems with convergence to the values of the old model. This process is made possible by the function `apollo_readBeta`, which is called as:

```
apollo_beta = apollo_readBeta(apollo_beta,
                              apollo_fixed,
                              inputModelName,
                              overwriteFixed)
```

The function returns an updated version of `apollo_beta`. The first two arguments passed to the function are already known to the reader, the remaining two are:

`inputModelName`: The name of a previously estimated model, given as a string.

`overwriteFixed`: A boolean variable indicating whether parameters that are not to be estimated should have their starting values overwritten by the input file (set to FALSE by default).

To use `apollo_readBeta`, the outputs from the input model need to have been saved in the same directory as the current model file. We illustrate the use of this function in Figure 5, where we read in parameters from the earlier RP model (`Apollo_example_1.r`) which did not include the socio-demographic effects or the quality of service attribute, thus meaning that only values for the 9 estimated parameters were read in, with the fixed parameter `asc_car` kept to the value from `apollo_beta` given the use of `overwriteFixed=FALSE`, where, with `overwriteFixed=TRUE`, the value from the input file would also be used for fixed parameters.

```
> apollo_beta=apollo_readBeta(apollo_beta,apollo_fixed,"Apollo_example_1",overwriteFixed=FALSE)

Out of the 19 parameters in apollo_beta, 9 were updated with values from the input file.
1 parameter in apollo_beta was kept fixed at its starting value rather than being updated from the input
    ↪ file.
```

Figure 5: Using `apollo_readBeta` to load results from an earlier model as starting values

4.4 Validation and preparing user inputs

The final step in preparing the code and data for model estimation or application is to make a call to `apollo_validateInputs`. The function runs a number of checks and produces a consolidated list of model inputs. It is called as:

```
apollo_inputs=apollo_validateInputs()
```

This function takes no arguments but looks in the global environment for the various inputs required for a model. This always includes the control settings `apollo_control`, the model parameters `apollo_beta`, the vector with names of fixed parameters `apollo_fixed` and finally the data object `database`. If any of these objects are missing from the global environment, the execution of `apollo_validateInputs` fails. The function also looks for a number of optional objects, namely `apollo_HB`, which is used for Bayesian estimation (cf. Section 8), `apollo_draws` and `apollo_randCoeff`, which are used for continuous random coefficients (cf. Section 6.1), and `apollo_lcPars`, which is used for latent class (cf. Section 6.2).

Before returning the list of model inputs, `apollo_validateInputs` runs a number of validation tests on the `apollo_control` settings and the `database`. It then uses default values for any missing settings, sorts the data by ID and adds an extra column called `apollo_sequence` which is a running index of observations for each individual in the data. Finally, the code also checks for the presence of multiple rows per individual in the data and accordingly sets `apollo_control$panelData` to TRUE or FALSE[7]. The running of `apollo_validateInputs` is illustrated in Figure 6. The list that is returned, `apollo_inputs`, contains the validated versions of the various objects mentioned above, e.g. `database`.

4.5 Likelihood component: the *apollo_probabilities* function

The core part of the code is contained in the `apollo_probabilities` function, where we show this function for our simple MNL model in Figure 7. An important distinction arises between

[7] In R, elements of a list such as `apollo_control` can be referred to via `apollo_control$panelData`.

```
> apollo_inputs = apollo_validateInputs()
Missing setting for mixing, set to default of FALSE
Missing setting for nCores, set to default of 1
Missing setting for workInLogs, set to default of FALSE
Missing setting for seed, set to default of 13
Missing setting for HB, set to default of FALSE
Several observations per individual detected based on the value of ID.
  Setting panelData set to TRUE.
All checks on apollo_control completed.
All checks on data completed.
```

Figure 6: Running `apollo_validateInputs`

`apollo_probabilities` and other functions in *Apollo*. While the other functions we have encountered are part of the package, `apollo_probabilities` needs to be defined by the user as it is specific to the model to be estimated. The function itself is never called by the user, but is used for example by the function for model estimation `apollo_estimate` discussed below. The function returns probabilities, where the specific format depends on `functionality`, which takes a default value for model estimation, but other values apply for example in prediction. The value used depends on which function makes the call to `apollo_probabilities` and is controlled internally.

This function takes three inputs, namely the vector of parameters `apollo_beta`, the list of combined model inputs `apollo_inputs`, and the argument `functionality`, which takes a default value for model estimation, but other values apply for example in prediction, as discussed in Section 9.5. The value used depends on which function makes the call to `apollo_probabilities`.

In the following three subsections, we look at the individual components of the code shown in Figure 7.

4.5.1 Initialisation

Any use of the `apollo_probabilities` function begins with a call to `apollo_attach` which enables the user to then call individual elements within for example the database by name, e.g. using `female` instead of `database$female`. This function is called as:

```
apollo_attach(apollo_beta,
              apollo_inputs)
```

The function does not return an object as output and the user does not need to change the arguments for this function. The call to this function is immediately followed by a command instructing `R` to run the function `apollo_detach` once the code exits `apollo_probabilities`. This ensures that this call is made even if there is an error that leads to a failure (and hence hard exit) from `apollo_probabilities`. This call is made as:

```
on.exit(apollo_detach(apollo_beta,
                      apollo_inputs))
```

We next initialise a list (a flexible `R` object) called `P` which will contain the probabilities for the model, where this is a requirement for any type of model used with the code.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Create alternative specific constants and coefficients using interactions with socio-demographics
  asc_bus_value    = asc_bus  + asc_bus_shift_female * female
  asc_air_value    = asc_air  + asc_air_shift_female * female
  asc_rail_value   = asc_rail + asc_rail_shift_female * female
  b_tt_car_value   = b_tt_car + b_tt_shift_business * business
  b_tt_bus_value   = b_tt_bus + b_tt_shift_business * business
  b_tt_air_value   = b_tt_air + b_tt_shift_business * business
  b_tt_rail_value  = b_tt_rail + b_tt_shift_business * business
  b_cost_value     = ( b_cost +  b_cost_shift_business * business ) * ( income / mean_income ) ^
      ↪ cost_income_elast

  ### List of utilities: these must use the same names as in mnl_settings, order is irrelevant
  V = list()
  V[['car']]  = asc_car + b_tt_car_value * time_car + b_cost_value * cost_car
  V[['bus']]  = asc_bus_value + b_tt_bus_value * time_bus + b_acc * access_bus + b_cost_value * cost_bus
  V[['air']]  = asc_air_value + b_tt_air_value * time_air + b_acc * access_air + b_cost_value * cost_air
      ↪  + b_no_frills * ( service_air == 1 ) + b_wifi * ( service_air == 2 ) + b_food * ( service_air
      ↪ == 3 )
  V[['rail']] = asc_rail_value + b_tt_rail_value * time_rail + b_acc * access_rail + b_cost_value *
      ↪ cost_rail  + b_no_frills * ( service_rail == 1 ) + b_wifi * ( service_rail == 2 ) + b_food * (
      ↪ service_rail == 3 )

  ### Define settings for MNL model component
  mnl_settings = list(
    alternatives = c(car=1, bus=2, air=3, rail=4),
    avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
    choiceVar    = choice,
    V            = V
  )

  ### Compute probabilities using MNL model
  P[['model']] = apollo_mnl(mnl_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 7: The `apollo_probabilities` function: example for MNL model

4.5.2 Model definition

With $\varepsilon_{car,n,t}$, $\varepsilon_{bus,n,t}$, $\varepsilon_{air,n,t}$ and $\varepsilon_{rail,n,t}$ in Equation 1 being distributed identically and independently (*iid*) across individuals and choice scenarios following a type I extreme value distribution, we obtain an MNL model, with the probability for alternative i in choice task t for person n given by:

$$P_{i,n,t}(\boldsymbol{\beta}) = \frac{z_{avail,i,n,t} \cdot e^{V_{i,n,t}}}{\sum_{j=1}^{J} z_{avail,j,n,t} \cdot e^{V_{j,n,t}}}, \tag{2}$$

where $\boldsymbol{\beta}$ is a vector combining all model parameters, $V_{j,n,t}$ refers to the part of the utility functions in Equation 1 that excludes the error term $\varepsilon_{j,n,t}$, and where $z_{avail,j,n,t}$ takes a value of 1 if alternative j is available in choice set t for person n, and 0 otherwise.

In the central part of the `apollo_probabilities` function, the user defines the actual model, where in our example, this is a simple MNL model. No limits on flexibility are imposed on the user with the *Apollo* package. A number of prewritten functions for common models are made available in the package, going beyond MNL, as discussed in Section 5. Additionally, the user can define his/her own models, as discussed in Section 5.5. Finally, this part of the code can contain either a single model, as shown here, or multiple individual model components, as discussed in Section 7.

The `apollo_mnl` function is called via:

```
P[["model"]] = apollo_mnl(mnl_settings,
                          functionality)
```

The function returns probabilities for the model, where depending on `functionality`, this is for the chosen alternative only or for all alternatives. The output of the function is saved in a component of the list `P` where for single component models such as here, this element is called `P[['model']]`. The function takes as its core input a list called `mnl_settings` which has four compulsory inputs and one optional input. We will now look at these in turn.

`alternatives`: A named vector containing the names of the alternatives as defined by the user, and for each alternative, giving the value used in the dependent variable in the data. In our case, these simply go from 1 to 4.

`avail`: A list containing one element per alternative, using the same names as in `alternatives`. For each alternative, we define the availability either through a vector of values of the same length as the number of observations (i.e. a column from the data) or by a scalar of 1 if an alternative is always available. A user can also set `avail=1` which implies that all of the alternatives are available for every choice observation in the data.

`choiceVar`: A vector of length equal to the number of observations, containing the chosen alternative for each observation. In our example, this column is simply called `choice`.

`V`: A list object containing one utility for each alternative, using the same names as in `alternatives`, where any linear or non-linear specification is possible. The contents of `V` are complicated and are thus generally defined prior to calling the function, as in Figure 7. In our case, we pre-compute the interactions with socio-demographic variables in the lines preceding the definition of the actual utilities, creating for example the new parameter `b_tt_car_value`. This helps keep the code organised, makes it easier to add additional interactions and also avoids unnecessary calculations. The latter point can be understood by noting that in our example, the impact of income and purpose on the cost coefficient is calculated just once and then used in each of the four utilities, rather than being calculated four times.

`rows`: This is an optional argument which is missing by default. It allows the user to specify a vector called `rows` of the same length as the number of rows in the data. This vector needs to use logical statements to identify which rows in the data are to be used for this model. For any observations in the data where the entry in rows is set to FALSE, the probability for the model will be set to 1. This means that this observation does not contribute to the calculation of the likelihood and hence estimation of the model parameters. It is useful for example in

the case of hybrid choice models, a point we return to in Section 7. When omitted from the call to `apollo_mnl`, all rows are used, as in our example in Figure 7.

In the code example, we actually create the utilities `V` outside `mnl_settings` first just for ease of coding, but they can similarly be created directly inside the list. What matters is that they are then copied into a component called `V` inside `mnl_settings`.

4.5.3 Function output

The final component of the `apollo_probabilities` function prepares the output of the function. This performs further processing of the `P` list, which needs to include an element called `model`, where, in our example, this is the only element in `P`. The specific functions to be called in this part of the code depend on the data and model, where once again, the actual inputs to these functions are not to be changed by the user.

In our specific example, the only additional manipulation of the raw probabilities produced by `apollo_mnl` is a call to `apollo_panelProd` which multiplies the probabilities across individual choice observations for the same individual, thus recognising the repeated choice nature of our data. This function is only to be used in the presence of multiple observations per individual. When estimating a model, the code computes the probability for the chosen alternative, say $j^*_{n,t}$ in choice task t for person n, i.e. $P_{j^*_{n,t}}$, using Equation 2. The contribution by person n to the likelihood function for person n, with a given value for the vector of model parameters $\boldsymbol{\beta}$, is then given by:

$$L_n(\boldsymbol{\beta}) = \prod_{t=1}^{T_n} P_{j^*_{n,t}}, \tag{3}$$

where T_n is the number of separate choice situations for person n. This function is called as:

```
P = apollo_panelProd(P,
                     apollo_inputs,
                     functionality)
```

All arguments of this function have been described already. When called in model prediction (cf. Section 9.5), the multiplication across choices is omitted, i.e. the function returns an unmodified version of `P`, with one row per observation.

Independent of the model specification, the function `apollo_probabilities` always ends with the same two commands. First is `apollo_prepareProb` which prepares the output of the function depending on `functionality`, e.g. with different output for estimation and prediction. This is called as:

```
P = apollo_prepareProb(P,
                       apollo_inputs,
                       functionality)
```

This is followed by

```
return(P)
```

which ensures that `P` is returned as the output of `apollo_probabilities`.

We earlier mentioned the possible use of weights by including the setting `weights` in `apollo_control`. Weights are only used in estimation, and if the user wants to use weights, then in addition to including the setting in `apollo_control`, the function `apollo_weighting` needs to be called prior to `apollo_prepareProb`. This is called as:

```
P = apollo_weighting(P,
                     apollo_inputs,
                     functionality)
```

We illustrate the use of this function in the EM algorithm discussions in Section 10.5.

4.6 Estimation

Now that we have defined our model, we can perform model estimation by calling the function `apollo_estimate` and saving the output from it in an object called `model`. This function uses the `maxLik` package (Henningsen and Toomet, 2011) for classical estimation, where Bayesian estimation is discussed in Section 8. *Apollo* relies on numerical gradients and Hessians only for classical estimation. In its simplest form, this function is called via:

```
model = apollo_estimate(apollo_beta,
                        apollo_fixed,
                        apollo_probabilities,
                        apollo_inputs)
```

where we have already covered all four arguments. The function may also be called with an additional argument, namely `estimate_settings`, i.e.:

```
model = apollo_estimate(apollo_beta,
                        apollo_fixed,
                        apollo_probabilities,
                        apollo_inputs,
                        estimate_settings)
```

The additional input, i.e. `estimate_settings`, is a list which contains a number of settings for estimation. None of these settings is compulsory and default settings will be used for any omitted settings, or indeed all settings when calling `apollo_estimate` without the `estimate_settings` argument. The possible settings to include in this list are:

`estimationRoutine`: A character object which can take the values BFGS (for the Broyden-Fletcher-Goldfarb-Shanno algorithm), BHHH (for the Berndt-Hall-Hall-Hausman algorithm) or NR (for the Newton-Raphson algorithm), where, the specific syntax is for example `estimationRoutine="BFGS"` (default is set to BFGS).

`maxIterations`: An integer setting a maximum on the number of iterations (default is set to 200).

`writeIter`: A boolean variable, which, when set to TRUE, means that the values of parameters at each iteration are saved into a file in the working directory, saved as `modelName_iterations.csv` where `modelName` is as defined in `apollo_control`. This allows the user to monitor progress during estimation, which is useful especially for complex models. It is only possible when using BFGS (default is set to TRUE).

`hessianRoutine`: A character variable indicating what routine to use for calculating the final Hessian. Possible values are `numDeriv` for the numDeriv package (Gilbert and Varadhan, 2016), `maxLik` for the same package as used in estimation, and `none` for no covariance matrix calculation. We have generally found that `numDeriv` is more reliable than `maxLik` for computing the covariance matrix, but its use may lead to issues with complex mixture models. If, when using numDeriv, this fails, the code reverts to using maxLik (default is set to `numDeriv`).

`printLevel`: A numeric variable which can take levels from 0 to 3 and controls the level of detail printed out during estimation, with higher levels meaning more detail (default is set to 3).

`silent`: A boolean variable, which, when set to TRUE, means that no information is printed to the screen during estimation (default is set to FALSE).

`constraints`: A list of constraints to be applied in estimation. This is only possible with `BFGS` and uses the coding approach in `maxLik` (Henningsen and Toomet, 2011).

`scaling`: A named vector of scalings to be applied to individual parameters during estimation. This can help estimation if the scale of individual parameters at convergence is very different. In classical estimation, the user can specify scales for individual model parameters. For example, if the unscaled specification involves a component $\beta_k x_k$ in the utility function, and if the user wishes to apply a scale of s_{β_k}, the starting value will be automatically adjusted to $\beta_k^* = \frac{1}{s_{\beta_k}} x_k$, the utility component will be adjusted to $s_{\beta_k}\beta_k^* x_k$ and the maximum likelihood estimation will optimise the value of β_k^*. The final model estimates will be translated to the original scale, i.e. returning estimates for β_k[8]. The aim of this process is to have the parameters that are actually used in model estimation, i.e. β_k^* to be of a similar scale. An example of this is given for the MDCNEV model in Section 5.4.2. For Bayesian estimation, the scales are applied to the posterior parameter chains.

`bootstrapSE`: A numeric variable indicating the number of bootstrap samples to calculate standard errors. The default is 0, meaning no bootstrap standard errors will be calculated.

[8]When using scaling in Bayesian estimation in *Apollo*, not all estimates are returned to their original scale after estimation. Indeed, the scaling is applied to the parameter chains directly, and as producing scaled values for the underlying Normals is not convenient. We thus report the scaled outputs only for the fixed parameters, the random parameters after transformation to the actual distributions used, and the posterior means.

The number must zero or a positive integer.

`bootstrapSeed:` A numeric variable indicating the seed for the bootstrap sampling. The default is 24. If changed, it must be a positive integer. This value is only used if `bootstrapSE` > 0. In general, there is no need to change this value. However, if the user wants to add new repetitions to a completed or interrupted estimation with bootstrap standard errors, then this value should be changed to ensure samples are not repeated.

Figure 8 illustrates what happens when running `apollo_estimate` on our simple MNL model. The model first checks the model specification used inside `apollo_probabilities` and reports some basic diagnostics. These validation and diagnostic steps are skipped if the user has set a value of TRUE for `noValidation` or `noDiagnostics`, respectively, in `apollo_control`. For MNL, the checks include for example ensuring that no unavailable alternatives are chosen. This is followed by the main estimation process and finally the calculation of the Hessian. Prior to that step, which can take a long time in complex models (and may fail), the code also prints out the final estimates.

We can see from Figure 8 that the estimation uses minimisation of the negative of the log-likelihood, hence the positive values, which is of course equivalent to maximisation of the log-likelihood itself.

Model estimation is the most likely step during which failures are encountered when working with the *Apollo* package. These could be either caused by errors in using the `R` syntax, resulting in generic `R` error messages, or errors made in the use of the various *Apollo* functions, leading to more specific error or warning messages. Not all warnings will be terminal and the code will continue to run and report warnings after completion. It is in this case entirely possible that the estimation has reached an acceptable solution with the warning messages for example being a result of the estimation process trying parameter values that lead to numeric issues in some iterations.

If a user runs the entire script contained in a model file including any post-estimation processes in one go, then errors during estimation will cause further problems in the steps that follow, but the reporting of those problems will likely become less intuitive further down the line. The user should in that case return to the first error message obtained and identify the cause of this and remedy it in the code. In general, running the code section by section is advisable to avoid this issue as far as possible.

The outcomes of model estimation are saved in a list called `model`, which contains amongst other things the estimates (`model$estimates`) and the classical and robust covariance matrices (`model$varcov` and `model$robvarcov`).

4.7 Reporting and saving results

Now that we have completed model estimation, the user can output the results to the console (screen) and/or a set of different output files. Two separate functions are used for this, namely `apollo_modelOutput` for output to the screen, and `apollo_saveOutput` for output to files. These two commands do not return an object as output, i.e. are called without an object to assign the output to. In their default versions, these functions are called with only the `model` object as

```
model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)

Testing probability function (apollo_probabilities)

Overview of choices for MNL model component:
                                    car     bus     air    rail
Times available                 5446.00 6314.00 5264.00 6118.00
Times chosen                    1946.00  358.00 1522.00 3174.00
Percentage chosen overall         27.80    5.11   21.74   45.34
Percentage chosen when available  35.73    5.67   28.91   51.88

Starting main estimation
Initial function value: -6413.433
Initial gradient value:
             asc_bus               asc_air              asc_rail  asc_bus_shift_female
            -471.8336            -405.7411             906.9119             -217.4302
 asc_air_shift_female asc_rail_shift_female              b_tt_car              b_tt_bus
            -170.4289             454.6112           -19956.9923           -174060.3443
           b_tt_air              b_tt_rail  b_tt_shift_business              b_access
          -25629.1953          134032.5302         -130444.4796           -14953.5089
              b_cost b_cost_shift_business     cost_income_elast                 b_wifi
           -4769.1384           17472.3701           -360.0784             507.5689
              b_food
            120.8984
initial  value 6413.432774
iter   2 value 5891.018325
iter   3 value 5752.822505
...
iter  26 value 4830.944739
final  value 4830.944739
converged

Estimated values:
                          [,1]
asc_car                 0.0000
asc_bus                 0.2864
asc_air                -0.9034
asc_rail               -2.0926
asc_bus_shift_female    0.3402
asc_air_shift_female    0.2682
asc_rail_shift_female   0.1896
b_tt_car               -0.0131
b_tt_bus               -0.0213
b_tt_air               -0.0166
b_tt_rail              -0.0071
b_tt_shift_business    -0.0062
b_access               -0.0212
b_cost                 -0.0762
b_cost_shift_business   0.0334
cost_income_elast      -0.6138
b_no_frills             0.0000
b_wifi                  1.0267
b_food                  0.4221

Computing covariance matrix using numDeriv package.
 (this may take a while)
0%....25%....50%....75%....100%
Hessian estimated with numDeriv will be used.
Calculating LL(0)... -8196.02
Updating inputs... Done.
Calculating LL of each model component...Done.
```

Figure 8: Running `apollo_estimate` on MNL model

input, i.e.:

```
apollo_modelOutput(model)
```

and

```
apollo_saveOutput(model)
```

In addition, it is possible to call both functions with an additional argument that is a list of settings, i.e.

```
apollo_modelOutput(model,
                   modelOutput_settings)
```

and

```
apollo_saveOutput(model,
                  saveOutput_settings)
```

The two lists `modelOutput_settings` and `saveOutput_settings` have a number of arguments that are all optional, namely:

`printClassical`: If set to TRUE, the code will output classical standard errors as well as robust standard errors, computed using the sandwich estimator (cf. Huber, 1967). This setting then also affects the reporting of t-ratios, p-values and covariance/correlation matrices. If the computation of classical standard errors fails for some parameters, the user is alerted to this even if classical standard errors are not reported (default is TRUE for `apollo_modelOutput` and `apollo_saveOutput`).

`printPVal`: If set to TRUE, p-values are reported (default is FALSE for `apollo_modelOutput` and `apollo_saveOutput`).

`printT1`: if set to TRUE, t-ratios against 1 are reported in addition to t-ratios against 0, where this is useful for nested logit models and for multipliers (default is FALSE for `apollo_modelOutput` and `apollo_saveOutput`).

`printDiagnostics`: If set to TRUE, model diagnostics such as choice shares are reported (default is TRUE for `apollo_modelOutput` and `apollo_saveOutput`).

`printCovar`: If set to TRUE, the covariance matrix of parameters is reported (default is FALSE for `apollo_modelOutput` and TRUE for `apollo_saveOutput`).

`printCorr`: if set to TRUE, the correlation matrix of parameters is reported (default is FALSE for `apollo_modelOutput` and TRUE for `apollo_saveOutput`).

`printOutliers`: If set to TRUE, the 20 worst outliers in terms of lowest average probabilities for the chosen alternative are reported (default is FALSE for `apollo_modelOutput` and TRUE for `apollo_saveOutput`). Alternatively, a scalar can be provide to use instead of 20.

`printChange`: If set to TRUE, the changes from the starting values are reported for the estimated parameters (default is FALSE for `apollo_modelOutput` and TRUE for `apollo_saveOutput`).

The main outputs controlled by the above settings will determine what `apollo_saveOutput` writes into the main output file, which will be called `modelName_output.txt` where `modelName` is as defined in `apollo_control`.

`saveOutput_settings` list has five additional possible settings, namely:

`saveEst`: If set to TRUE, the code will save a *csv* file with the parameter estimates, standard errors and t-ratios, saved as `modelName_estimates.csv` (default is TRUE).

saveCov: If set to TRUE, a *csv* file will be produced with the covariance matrix, where, if `printClassical==TRUE`, a separate file will be produced with the classical covariance matrix, saved as `modelName_covar.csv` (default is TRUE).

saveCorr: If set to TRUE, a *csv* file will be produced with the correlation matrix, where, if `printClassical==TRUE`, a separate file will be produced with the classical correlation matrix, saved as `modelName_corr.csv` (default is TRUE).

saveModelObject: If set to TRUE, an output file of the *rds* (an `R` format) will be produced containing the `model` object, saved as `modelName_model.rds` (default is TRUE).

writeF12: If set to TRUE, the code will produce an `F12` file, which is an output format used by the ALogit software (ALogit, 2016)[9], saved as `modelName.f12` (default is FALSE).

An example of the on screen output is shown in Figure 9. For `apollo_saveOutput`, a text file containing output using the above settings will be produced, using a filename corresponding to `apollo_control$modelName`. The default settings imply a more verbose output for the log file as opposed to the on screen output, in addition to files with estimates, covariance matrices etc, unless instructed not to.

[9] This is file containing all key model outputs. It is also produced by Biogeme and ALogit provides a shell to compare the results across models using these files, which can come from different estimation packages. See www.alogit.com

```
apollo_modelOutput(model)

Model run using Apollo for R, version 0.0.8
www.cmc.leeds.ac.uk

Model name                              : Apollo_example_3
Model description                       : MNL model with socio-demographics on mode choice SP data
Model run at                            : 2019-02-28 16:08:14
Estimation method                       : bfgs
Model diagnosis                         : successful convergence
Number of individuals                   : 500
Number of observations                  : 7000

Number of cores used                    :  1
Model without mixing

LL(start)                               : -6413.433
LL(0)                                   : -8196.021
LL(final)                               : -4830.945
Rho-square (0)                          :  0.4106
Adj.Rho-square (0)                      :  0.4085
AIC                                     :  9695.89
BIC                                     :  9812.4
Estimated parameters                    :  17
Time taken (hh:mm:ss)                   :  00:00:19.47
Iterations                              :  28

Estimates:
                       Estimate  Std.err.  t.ratio(0)  Rob.std.err.  Rob.t.ratio(0)
asc_car                  0.0000        NA          NA            NA              NA
asc_bus                  0.2864    0.5830        0.49        0.5490            0.52
asc_air                 -0.9034    0.3735       -2.42        0.3612           -2.50
asc_rail                -2.0926    0.3534       -5.92        0.3507           -5.97
asc_bus_shift_female     0.3402    0.1328        2.56        0.1451            2.34
asc_air_shift_female     0.2682    0.0915        2.93        0.0952            2.82
asc_rail_shift_female    0.1896    0.0738        2.57        0.0781            2.43
b_tt_car                -0.0131    0.0007      -17.85        0.0008          -17.09
b_tt_bus                -0.0213    0.0016      -13.31        0.0015          -14.02
b_tt_air                -0.0166    0.0028       -5.98        0.0027           -6.21
b_tt_rail               -0.0071    0.0018       -3.89        0.0018           -4.00
b_tt_shift_business     -0.0062    0.0006      -10.38        0.0006          -10.56
b_access                -0.0212    0.0029       -7.38        0.0027           -7.82
b_cost                  -0.0762    0.0021      -36.33        0.0021          -36.40
b_cost_shift_business    0.0334    0.0027       12.18        0.0026           12.99
cost_income_elast       -0.6138    0.0301      -20.40        0.0306          -20.08
b_no_frills              0.0000        NA          NA            NA              NA
b_wifi                   1.0267    0.0561       18.29        0.0578           17.78
b_food                   0.4221    0.0550        7.67        0.0564            7.48


Overview of choices for MNL model component:
                                    car      bus      air     rail
Times available                 5446.00  6314.00  5264.00  6118.00
Times chosen                    1946.00   358.00  1522.00  3174.00
Percentage chosen overall         27.80     5.11    21.74    45.34
Percentage chosen when available  35.73     5.67    28.91    51.88
```

Figure 9: On screen output obtained using `apollo_modelOutput` for MNL model

5 Other model components

In Section 4, we gave a detailed overview of the approach to specifying and estimating models in the *Apollo* package. In this section, we look at the use of other model components, thus replacing the part of the code discussed in Section 4.5.2. We discuss ready-to-use functions for a number of commonly used models before explaining how a user can add his/her own model functions. We group these model structures

5.1 Other RUM-consistent discrete choice models

5.1.1 Nested logit

For the nested logit (NL) model (Daly and Zachary, 1978; McFadden, 1978; Williams, 1977), we adopt the efficient implementation of Daly (1987) but adapt it to the more commonly used version which divides the utilities by the nesting parameter in the within nest probabilities (see the discussions in Train 2009, chapter 4, and Koppelman and Wen 1998). Let us assume we have a nesting structure with three levels and that alternative i falls into nest o_m on the lowest level of nesting, which itself is a member of nest m on upper level of nesting, with m being in the root nest. We would then have that $0 < \lambda_{o_m} \leq \lambda_m \leq \lambda_r \leq 1$. The probability[10] of person n choosing alternative i in choice situation t is then given by:

$$P_{i,n,t} = P_{m,n,t}\, P_{(o_m|m),n,t}\, P_{(i|o_m),n,t} \tag{4}$$

where

$$P_{(i|o_m),n,t} = \frac{e^{(\frac{V_{i,n,t}}{\lambda_{o_m}})}}{\sum_{j\in o_m} e^{(\frac{V_{j,n,t}}{\lambda_{o_m}})}} \tag{5}$$

$$P_{(o_m|m),n,t} = \frac{e^{(\frac{\lambda_{o_m}}{\lambda_m} I_{o_m,n,t})}}{\sum_{l_m=1}^{M_m} e^{(\frac{\lambda_{l_m}}{\lambda_m} I_{l_m,n,t})}} \tag{6}$$

$$P_{m,n,t} = \frac{e^{(\frac{\lambda_m}{\lambda_r} I_{m,n,t})}}{\sum_{m=1}^{M} e^{(\frac{\lambda_l}{\lambda_r} I_{l,n,t})}} \tag{7}$$

with

$$I_{m,n,t} = \log \sum_{l_m=1}^{M_m} e^{(\frac{\lambda_{l_m}}{\lambda_m} I_{l_m,n,t})} \tag{8}$$

$$I_{o_m,n,t} = \log \sum_{j\in o_m} e^{(\frac{V_{j,n,t}}{\lambda_{o_m}})}, \tag{9}$$

[10] For the sake of simplicity of notation, we assume here that all alternatives are available in every choice situation for every person. In the code, we of course allow for departures from this assumption.

where $0 < \lambda_{o_m} \leq \lambda_m \leq \lambda_r \leq 1$. We also specify the model so that the nest parameter of the root is normalised to 1.

In the efficient implementation of Daly (1987), we then work in logs, where we define a set of elementary alternatives $\boldsymbol{E}$ and a tree function $\boldsymbol{t}$. The tree function gives us a set of composite nodes $\boldsymbol{C} = (\boldsymbol{t}(j), \boldsymbol{t}(\boldsymbol{t}(j)), \ldots \forall j \in \boldsymbol{E})$. For each elementary alternative, there is a single path up to the root r, where, for alternative i, this is given by: $\boldsymbol{A}(i, r, \boldsymbol{t}) = (i, \boldsymbol{t}(i), \boldsymbol{t}(\boldsymbol{t}(i)) \ldots r)$. We then have that:

$$log(P_{i,n,t}) = \sum_{a \in \boldsymbol{A}(j,r,\boldsymbol{t})} \frac{1}{\lambda_{\boldsymbol{t}(a)}} \left(V_{a,n,t} - \widetilde{V_{\boldsymbol{t}(a),n,t}} \right), \tag{10}$$

where $\lambda_{\boldsymbol{t}(a)}$ is the nesting parameter for the nest that contains a, and for any non-elementary elements a, we have:

$$\widetilde{V_{a,n,t}} = \lambda_a \, log \sum_{l \in a} exp(\frac{V_{l,n,t}}{\lambda_a}) \tag{11}$$

where $l \in a$ gives all the elements contained in a, which can be a mixture of nests and elementary alternatives. For normalisation, we set $\lambda_r = 1$, and for consistency with utility maximisation, we then have that $0 < \lambda_a \leq 1$, $\forall a$ and $\lambda_a \leq \lambda_{\boldsymbol{t}(a)}$, i.e. the λ terms in a given chain $\boldsymbol{A}(j, r, \boldsymbol{t})$ decrease as we go from the root down the tree.

In the actual user syntax, we adopt an approach inspired by ALogit[11] (ALogit, 2016) where a user needs to specify a chain going from the root to each of the elementary alternatives. To illustrate this, we look at an example on the data described in Section 3.1, where we implement a three-level NL (`Apollo_example_5.r`). A simpler two-level model is available in `Apollo_example_4.r`.

In the first level of the tree, alternatives are divided into public transport (PT) alternatives and car, while the PT alternatives are then further split into a nest containing rail and air (fastPT), where bus is then on its own. To estimate this model, we specify two additional parameters compared to the MNL model in Section 4.5.2 in the `apollo_beta` vector, say `lambda_PT` and `lambda_fastPT`.

Just like `apollo_mnl`, the `apollo_nl` function is called as follows:

```
P[['model']] = apollo_nl(nl_settings,
                         functionality)
```

The list `nl_settings` contains all the same elements as for MNL, i.e. the compulsory inputs `alternatives`, `avail`, `choiceVar`, `V` and the optional input `rows`. For further details on these, the reader is referred back to Section 4.5.2. For Nested Logit, the list `nl_settings` needs to contain two additional arguments, namely:

`nlNests`: A named vector containing the names of the nests and the associated structural parameters λ. For each λ, we give the name of the associated parameter. This list needs to include the `root`, which is the only nest for which the choice of name is not free for the user to determine.

[11] www.alogit.com

```
### Specify nests for NL model
nlNests      = list(root=1, PT=lambda_PT, fastPT=lambda_fastPT)

### Specify tree structure for NL model
nlStructure= list()
nlStructure[["root"]]   = c("car","PT")
nlStructure[["PT"]]     = c("bus","fastPT")
nlStructure[["fastPT"]] = c("air","rail")

### Define settings for NL model
nl_settings <- list(
  alternatives = c(car=1, bus=2, air=3, rail=4),
  avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
  choiceVar    = choice,
  V            = V,
  nlNests      = nlNests,
  nlStructure  = nlStructure
)

### Compute probabilities using NL model
P[["model"]] = apollo_nl(nl_settings, functionality)
```

Figure 10: Nested logit implementation (extract)

`nlStructure`: A list containing one element per nest, where each element is a vector with the names of the contents of that nest, which can itself be a mix of nests and elementary alternatives.

In our example, as illustrated in Figure 10 (where we do not show the definition of alternatives, availabilities, choices and utilities, as these remain the same as in the MNL model in Section 4.5.2), we have three nests, where this includes the root. The order of elements is of no importance as they are identified by the nest names, yet for consistency, using the same order as in the model structure which follows is advisable. For each nest, we give the nesting parameter, using the parameter names previously defined in `apollo_beta`. The final step in the definition of the NL model is a call to `apollo_nl` with the appropriate inputs. Extensive checks are performed by this function, notably ensuring that for each alternative, there is exactly one chain from the root to the bottom of the tree.

After estimation, the model reports estimates for all parameters, as for any model, but in addition prints out (except if `apollo_control$noDiagnostics==FALSE`) the resulting tree structure with the estimated nesting parameters in brackets (and this is repeated with `apollo_modelOutput` and `apollo_saveOutput` if `printDiagnostics` is set to TRUE in their respective settings). In the case of our example, shown in Figure 11, we see that, as intended, there is a direct link from the root to the car alternative, while all other alternatives are nested in a public transport nest, with a further layer of nesting for rail and air within that nest. The nesting parameters also follow the required decreasing trend when going from the root down the tree.

5.1.2 Cross-nested logit

For our implementation of the cross-nested logit (CNL) model (Vovsha, 1997), we follow the "Generalised Nested Logit" (GNL) model of Wen and Koppelman (2001), with all nesting parameters freely estimated, and the constraint on the allocation parameters (showing the

```
Nest: root (1)
|---------Alternative: car
'-Nest: PT (0.6953)
   |-----Alternative: bus
   '-Nest: fastPT (0.5862)
      |-Alternative: air
      '-Alternative: rail
```

Figure 11: Nested logit tree structure after estimation

membership of alternative j in nest m) that $0 \leq \alpha_{j,m} \leq 1, \forall j, m$ and $\sum_j \alpha_{j,m} = 1, \forall m$. Only two-level versions of CNL are available through the `apollo_cnl` function, i.e. one layer of nests below the root, with the membership of a non-root nest being made up entirely of elementary choice alternatives.

In our implementation, each alternative needs to fall into at least one nest on the second level of the tree, where this can be a single alternative nest. We then have M nests, S_1 to S_M, where $\alpha_{j,m}$ represents allocation of alternative j to nest S_m. We have that $0 \leq \alpha_{j,m} \leq 1 \ \forall \, j, m$ and $\sum_{m=1}^{M} \alpha_{j,m} = 1 \ \forall \, j$. The probabilities are then given by a sum over nests:

$$P_{i,n,t} = \sum_{m=1}^{M} P_{S_m,n,t} \, P_{(i|S_m),n,t} \tag{12}$$

where

$$P_{S_m,n,t} = \frac{\left(\sum_{j \in S_m} \left(\alpha_{j,m} e^{V_{j,n,t}} \right)^{\frac{1}{\lambda_m}} \right)^{\lambda_m}}{\sum_{l=1}^{M} \left(\sum_{j \in S_l} \left(\alpha_{j,l} e^{V_{j,n,t}} \right)^{\frac{1}{\lambda_l}} \right)^{\lambda_l}} \tag{13}$$

$$P_{(i|S_m),n,t} = \frac{\left(\alpha_{i,m} e^{V_{i,n,t}} \right)^{\frac{1}{\lambda_m}}}{\sum_{j \in S_m} \left(\alpha_{j,m} e^{V_{j,n,t}} \right)^{\frac{1}{\lambda_m}}} \tag{14}$$

Just like `apollo_mnl` and `apollo_nl`, the `apollo_nl` function is called as follows:

```
P[['model']] = apollo_cnl(cnl_settings,
                          functionality)
```

The list `cnl_settings` contains all the same elements as for MNL, i.e. the compulsory inputs `alternatives`, `avail`, `choiceVar`, `V` and the optional input `rows`. For further details on these, the reader is referred back to Section 4.5.2. For Cross-Nested Logit, the list `cnl_settings` needs to contain two additional arguments, namely:

`cnlNests`: A named vector containing the names of the nests and the associated structural parameters λ. For each λ, we give the name of the associated parameter. Unlike in `apollo_nl`, the `root` is not included for `apollo_cnl` as only two-level structures are used.

`cnlStructure`: A matrix showing the allocation of alternatives to nests, with one row per nest and one column per alternative, using the same ordering as in `alternatives` and `cnlNests`.

For our implementation example on the simple mode choice data, we define a structure where air is nested together with rail (fast PT), and bus is nested together with rail (ground-based PT), but there is no joint nest membership for bus and air. Finally, car is nested on its own. This means that the only alternative for which allocation parameters need to be estimated is the rail alternative, where we have that $\alpha_{rail,fastPT} + \alpha_{rail,groundPT} = 1$, with both $\alpha_{rail,fastPT}$ and $\alpha_{rail,groundPT}$ being constrained to be between 0 and 1. Imposing constraints directly on the estimation routine is inefficient and can affect the standard error calculations. We instead recommend the use of a logistic transform, where, with alternative j having an estimated allocation parameter for M different nests, we have that, for nest m:

$$\alpha_{j,m} = \frac{e^{(\alpha_{0,j,m})}}{\sum_{l=1}^{M} e^{(\alpha_{0,j,l})}}, \tag{15}$$

where a normalisation is required, for example fixing $\alpha_{0,j,1} = 0$.

The definitions of the alternatives `alternatives`, availabilities `avail`, the choice variable `choiceVar` and the utilities `V` remains the same as in the MNL and NL codes. Like in the NL model, we define a vector of names for the nests, `cnlNests`, which defines the names of the nests and the associated structural parameters λ, using the parameter names previously defined in `apollo_beta`.

In our example (`Apollo_example_6.r`), as illustrated in Figure 12 (where we do not show the definition of alternatives, availabilities, choices and utilities, as these remain the same as in the MNL and NL models), we have three nests, one for air and rail (fastPT), one for bus and rail (groundPT), and one for car, which is nested on its own. The nesting parameter for the car nest is set to 1 given this is a single alternative nest. Our CNL implementation is limited to a two-level structure, and all elementary alternatives need to belong to at least one nest below the root, even if these are single alternative nests. This means that all nests defined by the user are automatically positioned below the root and the root is thus not included in the definition of the nest or tree structure given by the user.

For the allocation or nest membership parameters, car and bus both fall into one nest exactly, so they have $\alpha_{j,m} = 1$ for the specific nest m they fall into and the remaining ones are set to zero. For rail, we define $\alpha_{rail,fastPT} = \frac{e^{(\alpha_{0,rail,fastPT})}}{e^{(\alpha_{0,rail,fastPT})} + e^{(\alpha_{0,rail,groundPT})}}$, and obviously $\alpha_{rail,groundPT} = 1 - \alpha_{rail,fastPT}$, while we use the normalisation that $\alpha_{0,rail,groundPT} = 0$, by including the parameters `alpha0_rail_fastP` in `apollo_fixed`. The crucial part of the definition of a CNL model is again the actual model structure, which in our code is again called `cnlStructure`, where this is now made up of a matrix with one row per nest, and one alternative per column, where the entry in a given cell corresponds to the appropriate allocation parameter. The order of rows and columns needs to be consisted with the order in `cnlNests` and `alternatives`, respectively.

In the model output, the code reports the resulting tree structure with the estimated allocation and nesting parameters. In the case of our example, shown in Figure 13, we see that, as intended, car, bus and air all belong to one nest only, while the estimation has shown that the split for rail is almost 50-50, with the λ parameter being smaller in the fastPT nest. In reporting the

```
### Specify nests for CNL model
cnlNests = list(fastPT=lambda_fastPT, groundPT=lambda_groundPT, car=1)

### Specify nest allocation parameters for alternatives included in multiple nests
alpha_rail_fastPT    = exp(alpha0_rail_fastPT)/(exp(alpha0_rail_fastPT) + exp(alpha0_rail_groundPT))
alpha_rail_groundPT = 1 - alpha_rail_fastPT

### Specify tree structure, showing membership in nests (one row per nest, one column per alternative)
cnlStructure       = matrix(0, nrow=length(cnlNests), ncol=length(V))
cnlStructure[1,] = c( 0,   0, 1, alpha_rail_fastPT   ) # fastPT
cnlStructure[2,] = c( 0,   1, 0, alpha_rail_groundPT) # groundPT
cnlStructure[3,] = c( 1,   0, 0, 0                   ) # car

### Define settings for CNL model
cnl_settings <- list(
  alternatives = c(car=1, bus=2, air=3, rail=4),
  avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
  choiceVar    = choice,
  V            = V,
  cnlNests     = cnlNests,
  cnlStructure = cnlStructure
)

### Compute probabilities using CNL model
P[["model"]] = apollo_cnl(cnl_settings, functionality)
```

Figure 12: Cross-nested logit implementation (extract)

allocation parameters, the code uses the final values used inside `cnlStructure`, i.e. after the logistic transform in our example.

```
Final structure for CNL model component
          car bus air rail       lambda
fastPT    0   0   1   0.4928     0.4012
groundPT  0   1   0   0.5072     0.5284
car       1   0   0   0          1
```

Figure 13: Cross-nested logit structure after estimation

5.2 Non-RUM decision rules for discrete choice

In this section, we present the use of two alternatives to RUM in *Apollo*, namely random regret minimisation (RRM) and Decision Field Theory (DFT).

5.2.1 Random regret minimisation (RRM)

The fundamental assumption in regret theory is that what matters is not only the realised outcome but also on what could have been obtained by selecting a different course of action. This means that the model incorporates anticipated feelings of regret that would be experienced once ex-post decision outcomes are revealed to be "unfavourable". The value of an alternative can thus only be assigned following a cross-wise evaluation of alternatives, and this is the cause for substantial increases in computational complexity with large choice sets.

Following (Chorus, 2010), the deterministic regret for alternative i ($i = 1, \ldots, I$) for respondent n in choice task t is given:

$$R_{i,n,t} = \sum_{k=1}^{K} \sum_{j \neq i} \ln\left(1 + e^{\beta_k (x_{j,n,t,k} - x_{i,n,t,k})}\right) \tag{16}$$

where β_k is the coefficient associated with attribute x_k, with $k = 1, \ldots, K$. The regret is informed by all the pairwise comparisons, where regret for alternative i increases whenever an alternative $j \neq i$ performs better than i on a given attribute. When using extreme value error terms, RRM models are in fact Logit models, albeit not RUM-consistent models. With the assumption of type I extreme value errors, the probability of respondent n choosing alternative i in choice task t, is now simply given by a MNL formula as:

$$P_{i,n,t} = \frac{e^{-R_{i,n,t}}}{\sum_{j=1}^{J} e^{-R_{j,n,t}}}. \tag{17}$$

In RRM, we minimise the regret rather than maximising the utility, and this is achieved by maximising the negative regret in Equation 17.

Given the above point, any of the logit family models in *Apollo* can be used also for regret minimisation, by simply replacing the utilities (i.e. `V`) by the negative of regret. The labour intensive part comes in specifying the regret functions for the alternatives, i.e. implementing Equation 16.

An example of an RRM implementation is given in Figure 14, where we apply a MNL (i.e. non-nested) version of RRM to the mode choice data from Section 3.1. We use a simpler implementation than in Section 4.5.2, with no socio-demographics. This example is available in `Apollo_example_7.r`.

In a RUM model, only the attributes applying to a given alternative are used in the utility for that alternative, and the absence of access time for car or service quality for car and bus is of no importance. In a RRM model, we need to create these attributes given that the differences across the alternatives are used for all attributes. We next define the regret function for each alternative, where we here only show the regret for the car alternative, with a corresponding formulation applying for other modes, each time using Equation 16. For the alternative specific constants (ASCs), we adopt the convention of entering them directly into the regret function rather than using Equation 16. In the `mnl_settings` list, we now define `V` to be the negative of `R` by multiplying each element in `R` by `-1`. We finally make the call to `apollo_mnl`.

5.2.2 Decision field theory (DFT)

Decision field theory (DFT) originates in mathematical psychology (Busemeyer and Townsend, 1992, 1993) and is very different to both RUM and RRM. The key assumption under a DFT model is that the preferences for alternatives update over time. The decision-maker considers the alternatives until they reach an internal threshold (similar to the concept of satisficing, where one of the options is deemed 'good enough') or some external threshold (i.e. some time constraint, where a decision-maker stops deliberating on the alternatives as a result of running out of time to make the decision).

An example of a decision process under DFT is given in Figure 15. In this particular example, the decision-maker chooses different alternatives if they make their choice after reaching an internal threshold (which is represented by the horizontal line) on the 4^{th} preference updating timestep or if they conclude after 10 steps upon reaching a time threshold. Mathematically, DFT has been

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Prepare regret components for categorical variables
  access_car  = 0
  RFrills_car = b_no_frills
  RFrills_bus = b_no_frills
  RFrills_air  = b_no_frills * ( service_air  == 1 ) + b_wifi * ( service_air  == 2 ) + b_food * (
      ↪ service_air  == 3 )
  RFrills_rail = b_no_frills * ( service_rail == 1 ) + b_wifi * ( service_rail == 2 ) + b_food * (
      ↪ service_rail == 3 )

  ### List of regret functions: these must use the same names as in mnl_settings, order is irrelevant
  R = list()
  R[['car']]  = asc_car  +
    log(1+exp(b_tt_bus*time_bus   - b_tt_car*time_car)) +
    log(1+exp(b_tt_air*time_air   - b_tt_car*time_car)) +
    log(1+exp(b_tt_rail*time_rail - b_tt_car*time_car)) +
    log(1+exp(b_cost*(cost_bus  - cost_car))) +
    log(1+exp(b_cost*(cost_air  - cost_car))) +
    log(1+exp(b_cost*(cost_rail - cost_car))) +
    log(1+exp(b_access*(access_bus - access_car))) +
    log(1+exp(b_access*(access_air - access_car))) +
    log(1+exp(b_access*(access_rail- access_car))) +
    log(1+exp(RFrills_bus  - RFrills_car)) +
    log(1+exp(RFrills_air  - RFrills_car))  +
    log(1+exp(RFrills_rail - RFrills_car))
...

  ### Define settings for RRM model, which is MNL with negative regret as utility
  mnl_settings <- list(
    alternatives = c(car=1, bus=2, air=3, rail=4),
    avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
    choiceVar    = choice,
    V            = lapply(R, "*", -1)
  )

  ### Compute probabilities using MNL model
  P[['model']] = apollo_mnl(mnl_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 14: Implementation of random regret MNL model

operationalised differently depending on whether internal or external thresholds are used. A full specification of DFT with internal thresholds is given by Busemeyer and Townsend (1993), while we focus here on DFT with external thresholds (c.f. Roe et al. (2001) for the first adaptation of DFT with external thresholds for multiple alternatives). For DFT with an external threshold, the preference values update stochastically as a result of the assumption that a decision-maker considers just one attribute of an alternative at each timestep. Consequently, the preference values for each alternative update iteratively:

$$P_t = S \cdot P_{t-1} + V_t, \tag{18}$$

where P_t is a column vector containing the preference values of each alternative i at time t. S is a feedback matrix with memory and sensitivity parameters (detailed in Equation 19) and V_t is a

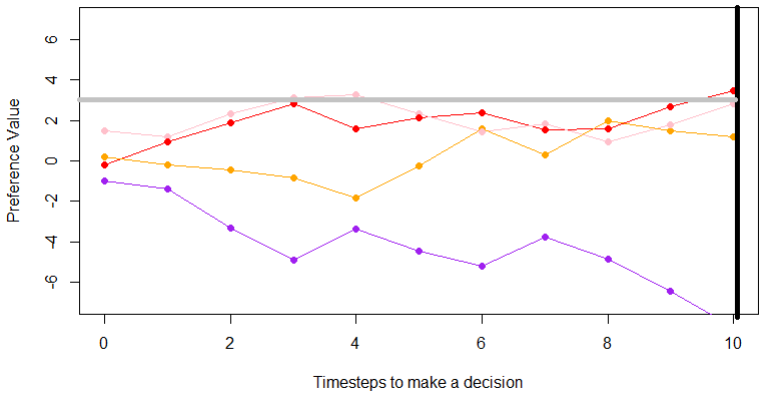

Figure 15: An example of a decision-maker stopping upon reaching either an internal or external threshold

valence vector (Equation 20), which varies depending on which attribute is attended to at time t. The feedback matrix used in *Apollo* is based on the definition by Hotaling et al. (2010):

$$S = I - \phi_2 \times \exp(-\phi_1 \times D^2), \tag{19}$$

where I is the identity matrix of size n and n is the number of alternatives. The feedback parameter has two free parameters. The first, ϕ_1, is a sensitivity parameter, which allows for competition between alternatives that are more similar (in terms of attribute values). The second, ϕ_2, is a memory parameter, which captures whether attributes considered at the start of the deliberation process or attributes considered at the end are more important. Finally, D is some measure of distance between the alternatives. In our code, we use the Euclidean distance for simplicity. Next, the valence vector can be described as:

$$V_t = C \cdot M \cdot W_t + \varepsilon_t, \tag{20}$$

where C is a contrast matrix used to rescale the attribute values such that they sum to zero, M is a matrix containing the attribute values for all of the alternatives, $W_t = [0..1..0]'$ with entry $k = 1$ if and only if attribute k is the attribute being attended to by the decision-maker at timestep t, and ε_t is an error term.

The implementation of DFT in *Apollo* allows for two different ways of accounting for the relative importance of attributes. A user may define attribute importance weights w_k, for each attribute, that are to be estimated and which correspond to the likelihood of a decision-maker attending to that attribute k. These however have the limitation that they must sum to one, which consequently requires the user to have a priori knowledge on the directionality of attributes (Hancock et al., 2018). Alternatively, the analyst may use 'attribute scaling coefficients'. These have many benefits (see Hancock et al. 2019 for a detailed explanation of these), including, most importantly, avoiding the limitation of having to sum to one. By instead assuming that each attribute is attended to with the same likelihood (all weights, $w_k = 1/n$), the relative importance

can instead enter as a set of scaling coefficients, s_k, which are applied to the attributes before they are entered (through M in Equation 20) into the calculation of the valence vector at each timestep.

The error term ε is drawn from independent and identically distributed normal draws with mean 0 and a standard deviation which is an estimated parameter. Consequently, the preference values P_t converge to a multivariate normal distribution (Roe et al., 2001). To calculate the probabilities of alternatives under DFT we thus simply require the expectation and covariance of P_t (ξ_t and Ω_t, respectively). Hence, the probability of choosing alternative j from a set of J alternatives at time t is:

$$P_{DFT}\left[\max_{i\in J} P_t[i] = P_t[j]\right] = \int_{X>0} exp\left[-(X-\Gamma)'\Lambda^{-1}(X-\Gamma)/2\right]/(2\pi|\Lambda|^{0.5})dX, \tag{21}$$

with X the set of differences between the preference value for the chosen alternatives and each other alternative, $X = [P_t[j] - P_t[1], ..., P_t[j] - P_t[J]]'$. Additionally, we require transformations of the expectation and covariance, $\Gamma = L\xi_t$, $\Lambda = L\Omega_t L'$, with L a matrix comprised of a column vector of 1s and a negative identity matrix of size $J-1$ where J is the number of alternatives. The column vector of 1s is placed in the i^{th} column where i is the chosen alternative.

An implementation of DFT is given in `apollo_dft`, which is called as:

```
P[['model']] = apollo_dft(dft_settings,
                          functionality)
```

where `dft_settings` contains the following elements:

- `alternatives`: A named vector containing the names of the alternatives, as for other discrete choice models.
- `avail`: A list containing availabilities, as for other discrete choice models.
- `choiceVars`: A variable indicting the column in the database which identifies the alternative chosen in a given choice situation, as for other discrete choice models.
- `attrValues`: A list with attribute values for alternatives, where this list contains one list per alternative, using the names from `alternatives`. Each alternative-specific list then contains the attribute values for that alternative, with one entry per attribute, where these are all column vectors with one entry per observation. DFT requires all alternatives to have each of the specified attributes, so by default will set attribute values of zero for any attributes not provided for a given alternative. Note that attributes specified here that are not included in either `attrWeights` or `attrScalings` will be ignored.
- `altStart`: A list containing a starting preference value for each alternative, using the same names as `alternatives`. As with other models, these are generally defined in `apollo_beta`, but could involve interactions with socio-demographics or be randomly distributed across individuals and/or observations.
- `attrWeights`: A list of weights, with one for each attribute. These should sum to one and will be adjusted accordingly if they do not. As mentioned above, any attributes included in `attrValues` but missing from `attrWeights` will be ignored. Conversely, any attribute

missing in `attrValues` but included in `attrWeights` will be created in `attrValues` but set to zero. Note that `attrWeights` should be set to 1 if `attrScalings` is provided.

- `attrScalings`: A list of scaling parameters that are applied to attribute values before they are passed into M in Equation 20. These do not need to sum to 1 across the set of attributes. As mentioned above, any attributes included in `attrValues` but missing from `attrScalings` will be ignored. Conversely, any attribute missing in `attrValues` but included in `attrScalings` will be created in `attrValues` but set to zero. Note that `attrScalings` should be set to 1 if `attrWeights` is provided.
- `procPars`: A list containing the four DFT 'process parameters'. The first of these is `error_sd`, which corresponds to the standard deviation of the error term in Equation 20. The second, `timesteps`, is the number of preference updating timesteps (t in Equations 18 and 20). `apollo_dft` will automatically adjust the number of timesteps such that there is at least one timestep. The final process parameters are the sensitivity and process parameters, `phi1` and `phi2`, from Equation 19. All of these parameters can be entered as single values to be used across the dataset, or can take choice-set dependent values.
- `rows`: The optional rows argument already described for the earlier models.

An example of a DFT implementation is given in Figure 16, where we apply a DFT model with scale parameters to the mode choice data from Section 3.1. We use an identical implementation to that of the MNL model in Section 4.5.2, with the same socio-demographics parameters. This is a key advantage of using scaling parameters (with the weights instead being fixed) in a DFT model, as it allows us to make equivalent adjustments to the parameters. This example is available in `Apollo_example_9.r`, where a simpler DFT model without covariates applied to the Swiss route choice data is available in `Apollo_example_8.r`.

Values for alternatives without a given attribute (wifi, food and access time for car, for example) are set to zero (and would be automatically set to zero if not initially provided). Additionally, DFT weights are automatically rescaled to sum to one, therefore attribute specific scalings (such as the one for the travel time coefficient in this example) are more efficiently employed through the use of attribute scaling parameters. Consequently, `attrWeights` is set to 1 in `dft_settings`.

Note that DFT process parameters can often cause identification or estimation issues (c.f. (Hancock et al., 2019)). Consequently, care is required, particularly when estimating DFT models on datasets where the process parameters are unlikely to have an impact, as poor initial starting values for the parameters can result in convergence to poor local optima. Here, we adjust the process parameters to aid estimation. We use exponentials to restrict the number of deliberation timesteps to be greater than 1 and the sensitivity parameter to be positive, and a logistic transform to ensure the memory parameter falls between 0 and 1. Additionally, with this data, we fix `error_sd` by including it in `apollo_fixed`. Finally, it is preferable to use non-zero starting values for all parameters.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Create alternative specific constants and coefficients using interactions with socio-demographics
  asc_bus_value   = asc_bus  + asc_bus_shift_female * female
  asc_air_value   = asc_air  + asc_air_shift_female * female
  asc_rail_value  = asc_rail + asc_rail_shift_female * female
  b_tt_car_value  = b_tt_car + b_tt_shift_business * business
  b_tt_bus_value  = b_tt_bus + b_tt_shift_business * business
  b_tt_air_value  = b_tt_air + b_tt_shift_business * business
  b_tt_rail_value = b_tt_rail + b_tt_shift_business * business
  b_cost_value    = ( b_cost +  b_cost_shift_business * business ) * ( income / mean_income ) ^
      ↪ cost_income_elast

  ### List of attribute values
  attrValues = list()
  attrValues[['car']]  = list(time=time_car,  access=0           ,cost=cost_car ,wifi=0
      ↪                      ,food=0                    )
  attrValues[['bus']]  = list(time=time_bus,  access=access_bus ,cost=cost_bus ,wifi=0
      ↪                      ,food=0                    )
  attrValues[['air']]  = list(time=time_air,  access=access_air ,cost=cost_air ,wifi=1*(service_air  ==
      ↪ 2),food=1*(service_air ==  3))
  attrValues[['rail']] = list(time=time_rail, access=access_rail,cost=cost_rail,wifi=1*(service_rail ==
      ↪ 2),food=1*(service_rail == 3))

  ### List of initial preference values
  altStart = list()
  altStart[['car']]  = asc_car
  altStart[['bus']]  = asc_bus_value
  altStart[['air']]  = asc_air_value
  altStart[['rail']] = asc_rail_value

  ### List of attribute scaling factors
  attrScalings = list(time      = list(car = b_tt_car_value, bus = b_tt_bus_value, air = b_tt_air_value,
      ↪  rail = b_tt_rail_value),
                      access    = b_acc,
                      cost      = b_cost_value,
                      wifi      = b_wifi,
                      food      = b_food)

  ### List of process parameters
  procPars  = list(
    error_sd=p_error_sd,
    timesteps=1+exp(p_timesteps),
    phi1=exp(p_phi1),
    phi2=exp(p_phi2)/(1+exp(p_phi2))
  )

  ### Define settings for DFT model component
  dft_settings <- list(
    alternatives = c(car=1, bus=2, air=3, rail=4),
    avail        = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
    choiceVar    = choice,
    attrValues   = attrValues,
    altStart     = altStart,
    attrWeights  = 1,            ### Using scaling factors, so attrWeights must be set to 1.
    attrScalings = attrScalings,
    procPars     = procPars
  )

  ### Compute choice probabilities using DFT model
  P[['model']] = apollo_dft(dft_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 16: DFT implementation for the SP dataset

5.3 Models for ranking, rating and continuous dependent variables

Especially when developing hybrid choice models (cf. Section 7.3, some of the dependent variables in the model will not be of the discrete choice type. We now look at how to model such dependent variables in *Apollo*.

5.3.1 Exploded logit

Datasets may include the full ranking for alternatives, in which case an exploded logit model can be used. In particular, with J different alternatives for individual n in choice situation t, we may observe the ranking $R_{nt} = \langle R_{nt,1}, \ldots, R_{nt,J} \rangle$, where $R_{nt,1}$ is the index for the alternative which is ranked the highest, i.e. the choice in a simple discrete choice setting. Note that this is different from the convention where $R_{nt,j}$ is the rank of alternative j; here, $R_{nt,j}$ refers to the specific alternative ranked in j^{th} place.

We then have that the probability of the observed ranking is given by:

$$P_{nt} = \prod_{i=1}^{J-1} \frac{e^{\mu_i V_{R_{nt,i}}}}{\sum_{j=i}^{J} e^{\mu_i V_{R_{nt,j}}}}, \tag{22}$$

where this is given by a product of logit probabilities for all but the last ranking (which is just a single alternative), where the denominator gradually omits alternatives, and where we allow for differences in scale across the stages, with an appropriate normalisation, e.g. $\mu_1 = 1$.

In *Apollo*, the exploded logit model is implemented in the function `apollo_el`, which is called called as follows:

```
P[['model']] = apollo_el(el_settings,
                         functionality)
```

where the contents of `el_settings` are a little different from the earlier MNL, NL and CNL models. In particular, we have:

`alternatives`: A named vector containing the names of the alternatives, as for MNL, NL and CNL.

`avail`: A list containing availabilities, as for MNL, NL and CNL.

`choiceVars`: A list containing the names of the variables indicating the column in the database which identify the choices at each stage in the ranking, except for the final (worst) alternative. If not all alternatives are available for all individuals, then some of the later rankings will not apply for these individuals, and the user should put a value of NA in the data for those entries. For example, if a given person only has two out of the four alternatives available, then the third and fourth ranking should be given as `-1` in the data for that individual.

`V`: A list of utilities, as for MNL, NL and CNL.

`scales`: An optional argument given by a list, with one entry per stage in the ranking, giving the scale parameter to be used in that stage.

`rows`: The optional `rows` argument already described for the earlier models.

An example using the exploded logit model is given in `Apollo_example_10.r`, using the drug choice data from Section 3.3. We use a dummy coded specification for the three categorical variables, along with a continuous specification for risk and cost.

The utility for alternative j in choice situation t for individual n is given by:

$$\begin{aligned} V_{j,n,t} = & \sum_{s=1}^{5} \beta_{brand_s} \cdot \left(x_{brand_{j,n,t}} == s\right) \\ & + \sum_{s=1}^{6} \beta_{country_s} \cdot \left(x_{country_{j,n,t}} == s\right) \\ & + \sum_{s=1}^{3} \beta_{characteristic_s} \cdot \left(x_{characteristic_{j,n,t}} == s\right) \\ & + \beta_{side\_effects} \cdot x_{side\_effects_{j,n,t}} \\ & + \beta_{price} \cdot x_{price_{j,n,t}} \end{aligned} \tag{23}$$

For the first three rows in Equation 23, one of the β parameters in each row is constrained to zero (dummy coded), and not all levels apply for each alternative, as described in Appendix B.

The implementation of the model is shown in Figure 17. Special care is required for the qualitative attributes. These are coded as text in the data, and one parameter needs to be associated with each level, where we impose an appropriate normalisation in `apollo_fixed` to set the parameter for one level to zero for each attribute. The levels that are included in the utility functions differ across alternatives, as reflected in the design of the survey (cf. Table A3). The key different from an MNL model arises in the inclusion of `choiceVars` instead of `choiceVar` in `el_settings` where this differs from giving a single preferred alternative for each observation and instead giving one column for each stage in the ranking except for the final stage. We also provide scale parameters for these three stages in `el_settings$scales`, where the scale for the first stage is normalised to 1.

5.3.2 Ordered logit

For ordinal dependent variables, the function `apollo_ol` provides an implementation of the ordered logit model. Specifically, let $Y_{n,t}$ be the observed value for the dependent variable for the t^{th} observation for individual n, where $Y_{n,t}$ can take S different possible values, going from $s = 1, \ldots, S$. The probability of observing value s is then given by:

$$P_{Y_{n,t}=s} = \frac{e^{\tau_s - V_{n,t}}}{1 + e^{\tau_s - V_{n,t}}} - \frac{e^{\tau_{s-1} - V_{n,t}}}{1 + e^{\tau_{s-1} - V_{n,t}}} \tag{24}$$

The likelihood of the observed value $Y_{n,t}$ is then given by:

$$L_{Y_{n,t}} = \sum_{s=1}^{S} \delta_{(Y_{n,t}=s)} \left[\frac{e^{\tau_s - V_{n,t}}}{1 + e^{\tau_s - V_{n,t}}} - \frac{e^{\tau_{s-1} - V_{n,t}}}{1 + e^{\tau_{s-1} - V_{n,t}}} \right], \tag{25}$$

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### List of utilities: these must use the same names as in el_settings, order is irrelevant
  V = list()
  V[['alt1']] = ( b_brand_Artemis*(brand_1=="Artemis") + b_brand_Novum*(brand_1=="Novum")
                  + b_country_CH*(country_1=="Switzerland") + b_country_DK*(country_1=="Denmark") +
                    ↪ b_country_USA*(country_1=="USA")
                  + b_char_standard*(char_1=="standard") + b_char_fast*(char_1=="fast acting") +
                    ↪ b_char_double*(char_1=="double strength")
                  + b_risk*side_effects_1
                  + b_price*price_1)
  V[['alt2']] = ( b_brand_Artemis*(brand_2=="Artemis") + b_brand_Novum*(brand_2=="Novum")
                  + b_country_CH*(country_2=="Switzerland") + b_country_DK*(country_2=="Denmark") +
                    ↪ b_country_USA*(country_2=="USA")
                  + b_char_standard*(char_2=="standard") + b_char_fast*(char_2=="fast acting") +
                    ↪ b_char_double*(char_2=="double strength")
                  + b_risk*side_effects_2
                  + b_price*price_2)
  V[['alt3']] = ( b_brand_BestValue*(brand_3=="BestValue") + b_brand_Supermarket*(brand_3=="Supermarket
      ↪ ") + b_brand_PainAway*(brand_3=="PainAway")
                  + b_country_USA*(country_3=="USA") + b_country_IND*(country_3=="India") + b_country_RUS
                    ↪ *(country_3=="Russia") + b_country_BRA*(country_3=="Brazil")
                  + b_char_standard*(char_3=="standard") + b_char_fast*(char_3=="fast acting")
                  + b_risk*side_effects_3
                  + b_price*price_3 )
  V[['alt4']] = ( b_brand_BestValue*(brand_4=="BestValue") + b_brand_Supermarket*(brand_4=="Supermarket
      ↪ ") + b_brand_PainAway*(brand_4=="PainAway")
                  + b_country_USA*(country_4=="USA") + b_country_IND*(country_4=="India") + b_country_RUS
                    ↪ *(country_4=="Russia") + b_country_BRA*(country_4=="Brazil")
                  + b_char_standard*(char_4=="standard") + b_char_fast*(char_4=="fast acting")
                  + b_risk*side_effects_4
                  + b_price*price_4 )

  ### Define settings for exploded logit
  el_settings = list(
    alternatives = c(alt1=1, alt2=2, alt3=3, alt4=4),
    avail        = list(alt1=1, alt2=1, alt3=1, alt4=1),
    choiceVars   = list(best, second_pref, third_pref),
    V            = V,
    scales       = list(1,scale_2,scale_3)
  )

  ### Compute exploded logit probabilities
  P[["model"]]=apollo_el(el_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 17: Exploded logit implementation

where, for normalisation, we set $\tau_S = +\infty$ and $\tau_0 = -\infty$, such that the probability of $Y_{n,t} = 1$ is given by $\frac{e^{\tau_1 - V_{n,t}}}{1+e^{\tau_1 - V_{n,t}}}$ while the probability of $Y_{n,t} = S$ is given by $1 - \frac{e^{\tau_S - V_{n,t}}}{1+e^{\tau_{S-1} - V_{n,t}}}$. In our notation, $V_{n,t}$ is the utility used inside the ordered logit model, which will be a function of characteristics of the decision maker and the scenario that the dependent variable relates to.

For an example using `apollo_ol`, see the section on hybrid choice models (Section 7.3). In

Apollo, the `apollo_ol` function is called as follows:

```
P[['model']] = apollo_ol(ol_settings,
                         functionality)
```

where the contents of `ol_settings` are a little different from the earlier MNL, NL and CNL models. In particular, we have:

`outcomeOrdered`: A variable indicating the column in the database which identifies the level selected for the ordinal variable in each observation.
`V`: A numeric vector containing the explanatory variable used in the ordered logit model, i.e. the utility in Equation 24.
`tau`: A vector containing the names of the threshold parameters that are used in the model. These need to be defined in `apollo_start`, and should have one fewer element than the number of possible values for the dependent variable `Y`. Extreme thresholds at $-\inf$ and $+\inf$ are added automatically by the code.
`coding`: An optional argument of numeric or character vector type which is only required as an input if the dependent variable does not use an incremental coding from 1 to a value equal to the number of possible values for the dependent variable `Y`. This can be used both if the dependent variable is numeric in the data but not monotonic or with unequal increment, or if the dependent variable is given in string format.
`rows`: The optional `rows` argument already described for the earlier models.

5.3.3 Normally distributed continuous variables

For continuous dependent variables (or ordinal dependent variables that are treated as continuous) the function `apollo_normalDensity` is available, which is an implementation of the Normal probability density function. This implies that the probability of observing the specific value for the dependent variable Y in situation t for person n is given by:

$$P\left(Y_{n,t}\right) = \frac{\phi\left(\frac{Y_{n,t}-X_{n,t}-\mu}{\sigma}\right)}{\sigma}, \tag{26}$$

where $X_{n,t}$ is the explanatory variable used, μ and σ are the estimated means and standard deviations, and ϕ is the standard Normal density function.

For an example using `apollo_normalDensity`, see the section on hybrid choice models (Section 7.3). The `apollo_normalDensity` is called as follows:

```
P[['model']] = apollo_normalDensity(normalDensity_settings,
                                    functionality)
```

where the contents of `normalDensity_settings` now include:

`outcomeNormal`: A variable indicating the column in the database which contains the value for the dependent variable in each observation.
`xNormal`: A numeric vector containing the explanatory variable used in Equation 26.
`mu`: The parameter used as the mean for the Normal density.
`sigma`: The parameter used as the standard deviation for the Normal density.
`rows`: The optional `rows` argument already described for the earlier models.

5.4 Discrete-continuous models

While choice modelling is generally best known for the study of the choice between mutually exclusive alternatives, a large body of research has also looked at the joint choice of multiple alternatives and the *consumption* of different quantities of each of these. Especially the family of Multiple Discrete Continuous Extreme Value models has received extensive interest in recent years, and two of these models are implemented in *Apollo*.

5.4.1 Multiple Discrete Continuous Extreme Value (MDCEV) model

The MDCEV model (Bhat, 2008) is a representation of a multiple discrete-continuous decisions process. Such a process consist of choosing one or more elements from a set of alternatives, and then choosing a non-negative amount of each of the chosen elements. Examples of such a process are consumption (what products or services to buy and how much of each), and time use (what activities to engage with and for how long). More formally, the MDCEV model is a stochastic implementation of the classical consumer maximization processes, where consumers allocate resources (e.g. their income) in a way that maximizes their utility. This problem can be formulated as follows:

$$\begin{aligned} \underset{x_k \forall k}{\text{Max}} \quad & \sum_{k=1}^{K} \frac{\gamma_k}{\alpha_k} \psi_k \left(\left(\frac{x_k}{\gamma_k} + 1 \right)^{\alpha_k} - 1 \right) \\ \text{subject to} \quad & \sum_{k=1}^{K} x_k p_k = B \end{aligned} \tag{27}$$

$$\psi_k = \exp(V_k + \varepsilon_k), \tag{28}$$

where K is the number of alternatives, x_k is the amount consumed of product k, and p_k is the unit price or cost of alternative k, and B is the budget available to the individual for consumption. The term ε_k is an independent and identically distributed random disturbance following a *Gumbel(0, σ)* distribution. Finally, α_k and γ_k are parameters determining satiation, while V_k determines each alternative's base utility (i.e. its marginal utility at zero consumption).

The probability of an observed vector of consumptions is then given by:

$$\begin{aligned} & P({x_1}^*, {x_2}^*, \cdots, {x_M}^*, 0, \cdots, 0) \\ =& \frac{1}{p_1} \frac{1}{\sigma^{M-1}} \left(\prod_{m=1}^{M} f_m \right) \left(\sum_{m=1}^{M} \frac{p_m}{f_m} \right) \left(\frac{\prod_{m=1}^{M} e^{V_i/\sigma}}{\left(\sum_{k=1}^{K} e^{W_k/\sigma} \right)^M} \right) (M-1)!, \end{aligned} \tag{29}$$

where $f_i = \frac{1-\alpha_i}{x_i^* + \gamma_i}$ and $W_k = V_k + (\alpha_k - 1)\log(\frac{x_k^*}{\gamma_k} + 1) - \log(p_k)$, and x_k^* is the observed (optimum) consumption of product k.

A revised formulation as shown in Equation 30 is obtained when an outside good is included among the alternatives. An outside good is a product that is consumed by all individuals in the sample. The outside good usually represents an aggregate measure of the consumption of all products that are not of interest for the study. For example, if a study focuses on use of leisure time, the outside good might be all activities that are not leisure (such as sleeping, work, travelling, etc.), while the inside goods (i.e. all alternatives that are not the outside good) could deal with leisure in a more detailed way (e.g. going to the park, hiking, going to the cinema, meeting friends, etc.).

$$\begin{aligned} & P({x_1}^*, {x_2}^*, \cdots, {x_M}^*, 0, \cdots, 0) \\ =& \frac{1}{\sigma^{M-1}} \left(\prod_{m=1}^{M} f_m \right) \left(\sum_{m=1}^{M} \frac{p_m}{f_m} \right) \left(\frac{\prod_{m=1}^{M} e^{V_i/\sigma}}{\left(\sum_{k=1}^{K} e^{W_k/\sigma} \right)^M} \right) (M-1)!, \end{aligned} \tag{30}$$

The function `apollo_mdcev` calculates the loglikelihood of an MDCEV model, using equation 29 if no outside good is provided and uses equation 30 if an outside good is provided. An example of a function call, as well as a definition of its arguments follow. The function is called as:

```
P[['model']] = apollo_mdcev(mdcev_settings,
                            functionality)
```

The list `mdcev_settings` contains the following objects:

`alternatives`: Character vector containing the name of all alternatives. If one of these alternatives is called `outside`, it will automatically be used as the outside good.

`avail`: List of availabilities, using the names from `alternatives`. Each element can be scalar (0 or 1) or a vector detailing availability for each observation.

`continuousChoice`: List of continuous consumption, using the names from `alternatives`. Each element must be a vector of length N (number of observations) indicating the amount consumed.

`V`: A list of length K (i.e. number of alternatives), containing the deterministic part of the base utility of each alternative. The outside good should have `V=0` if included.

`alpha`: List containing the α parameter for each alternative, using the names from `alternatives`.

gamma: List containing the γ parameter for each alternative, using the names from `alternatives`, excluding any outside good.
sigma: Scalar representing the scale parameter of the error term. If there is no price variation across products, this should be fixed to 1.
cost: List containing the cost or price of each alternative, using the names from `alternatives`. Each element can be a scalar if the price does not change across observations, or a vector of length equal to the number of observations in the data, detailing the price for each observation.
budget: Vector with the amount of the resource (e.g. money or time) available for each observation. It must be equal to the total consumption of that observation.
minConsumption: Optional argument, which, if provided, should be a list with as many elements as alternatives. Each element can be either a scalar or a vector defining the minimum consumption of an alternative if it is consumed.
outside: Optional argument with the name of an outside good. This is not needed if one of the alternatives is already called `outside`.
rows: The optional `rows` argument already described for the earlier models.

As discussed at length by Bhat (2008), different profiles exist for normalisation of a MDCEV model, either using a generic α and alternative-specific γ parameters, an α parameter only for the outside good (and set to zero for others) along with alternative-specific γ parameters, or alternative specific α terms with $\gamma = 1$ for all goods. In our examples below, we use a generic α and alternative-specific γ parameters. Other profiles can be implemented by simply changing which parameters are generic and which are alternative specific, and making some α terms equal to zero, as appropriate.

We include two examples of the MDCEV model on the time use data described in Section 3.4. The first example, `Apollo_example_11.r` does not include an outside good. We illustrate this in Figure 18.

We begin by defining the names of the alternatives, availabilities and continuous consumptions, where we turn minutes into hours. We then creating the list of utilities, where, in our example, these include alternative specific constants only, where we fix δ_{home} to zero for identification. This is followed by the definition of a generic α parameter, which is constrained to be below 1 by using a logistic transform, with $\alpha = \frac{1}{1+e^{-\alpha_{base}}}$, and the set of γ parameters, where these are alternative-specific in our case. We finally define the costs, turn the budget into hours, and make the call to `apollo_mdcev`. In this example, we also fix `sigma` to its starting value of 1 in `apollo_fixed`.

The second example, `Apollo_example_12.r`, groups together some alternatives to create an outside good. It also incorporates socio-demographics in the utility function, though not in the α and γ terms, which is however also possible in *Apollo*. We illustrate this example in Figure 19.

To create the new activities, we sum some of the activities up after reading in the data, where we create an outside good by combining time spent travelling with time spent at home. We also create a generic leisure activity. The remainder of the specification is no different in principle from that in Figure 19 with the exception of there being an alternative called `outside`, and with using more detailed utility functions.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Define individual alternatives
  alternatives = c("dropOff",
...
                   "other")

  ### Define availabilities
  avail = list(dropOff  = 1,
...
               other = 1)

  ### Define continuous consumption for individual alternatives
  continuousChoice = list(dropOff  = t_a01/60,
...
                          other = t_a12/60)

  ### Define utilities for individual alternatives
  V = list()
  V[["dropOff" ]] = delta_dropOff
...
  V[["other"]] = delta_other

  ### Define alpha parameters
  alpha = list(dropOff  = 1 /(1 + exp(-alpha_base)),
...
               other = 1 /(1 + exp(-alpha_base)))

  ### Define gamma parameters
  gamma = list(dropOff  = gamma_dropOff,
...
               other = gamma_other)

  ### Define costs for individual alternatives
  cost = list(dropOff  = 1,
...
              other = 1)

  ### Define budget
  budget = budget/60

  ### Define settings for MDCEV model
  mdcev_settings <- list(alternatives       = alternatives,
                         avail              = avail,
                         continuousChoice   = continuousChoice,
                         V                  = V,
                         alpha              = alpha,
                         gamma              = gamma,
                         sigma              = sigma,
                         cost               = cost,
                         budget             = budget)

  ### Compute probabilities using MDCEV model
  P[["model"]] = apollo_mdcev(mdcev_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 18: MDCEV implementation without outside good

```
### Load data
database = read.csv("apollo_time_use_data.csv",header=TRUE)
database$t_outside = rowSums(database[,c("t_a01", "t_a06", "t_a10", "t_a11", "t_a12")])
database$t_leisure = rowSums(database[,c("t_a07", "t_a08", "t_a09")])

...

apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Define individual alternatives
  alternatives  = c("outside",
...
                    "leisure")

  ### Define availabilities
  avail = list(outside  = 1,
...
               leisure  = 1)

  ### Define continuous consumption for individual alternatives
  continuousChoice = list(outside  = t_outside/60,
...
                          leisure  = t_leisure/60)

  ### Define utilities for individual alternatives
  V = list()
  V[["outside"]]  = 0
  V[["work"]]     = delta_work     + delta_work_FT * occ_full_time + delta_work_wknd * weekend
  V[["school"]]   = delta_school   + delta_school_young * (age<=30)
  V[["shopping"]] = delta_shopping
  V[["private"]]  = delta_private
  V[["leisure"]]  = delta_leisure  + delta_leisure_wknd*weekend

  ### Define alpha parameters
  alpha = list(outside  = 1 /(1 + exp(-alpha_base)),
...
               leisure  = 1 /(1 + exp(-alpha_base)))

  ### Define gamma parameters
  gamma = list(work     = gamma_work,
...
               leisure  = gamma_leisure)

  ### Define costs for individual alternatives
  cost = list(outside  = 1,
...
              leisure  = 1)

  ### Define budget
  budget = budget/60

  ### Define settings for MDCEV model
  mdcev_settings <- list(alternatives      = alternatives,
                         avail             = avail,
                         continuousChoice = continuousChoice,
                         V                 = V,
                         alpha             = alpha,
                         gamma             = gamma,
                         sigma             = sigma,
                         cost              = cost,
                         budget            = budget)

  ### Compute probabilities using MDCEV model
  P[["model"]] = apollo_mdcev(mdcev_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 19: MDCEV implementation with an outside good

5.4.2 Multiple Discrete Continuous Nested Extreme Value (MDCNEV) model

The MDCNEV is an extension to the MDCEV model, proposed by Pinjari and Bhat (2010a). It incorporates correlation between alternatives, in a similar way to the nested logit (NL), where correlation can be introduced by nesting, i.e. grouping alternatives that are correlated among them. The implementation of MDCNEV in *Apollo* allows for only a single level of nesting and is also only valid for models with an outside good, i.e. a product that is consumed in every observation. The likelihood function of the model is as follows.

$$
\begin{aligned}
&P({x_1}^*, {x_2}^*, \ldots, {x_M}^*, 0, \ldots, 0) \\
&= |J| \frac{\prod_{i \in \text{chosen alts}} e^{\frac{V_i}{\theta_i}}}{\prod_{s=1}^{S_M} \left(\sum_{i \in s^{th}_{nest}} e^{\frac{V_i}{\theta_s}} \right)^{q_s}} \\
&\cdot \sum_{r_1=1}^{q_1} \cdots \sum_{r_s=1}^{q_s} \cdots \sum_{r_{S_M}=1}^{q_{S_M}} \left\{ \prod_{s=1}^{S_M} \left[\frac{\left(\sum_{i \in s^{th}_{nest}} e^{\frac{V_i}{\theta_s}} \right)^{\theta_s}}{\sum_{s=1}^{S_k} \left\{ \left(\sum_{i \in s^{th}_{nest}} e^{\frac{V_i}{\theta_s}} \right)^{\theta_s} \right\}} \right]^{q_s - r_s + 1} \left(\prod_{s=1}^{S_M} sum(X_r s) \right) \left(\sum_{s=1}^{S_M} (q_s - r_s + 1) - 1 \right)! \right\},
\end{aligned}
\tag{31}
$$

For a detailed explanation of the values in the equation, see Pinjari and Bhat (2010a).

The `apollo_mdcnev` function is called as:

```
P[['model']] = apollo_mdcev(mdcnev_settings,
                            functionality)
```

Aside from the previously defined contents in `mdcnev_settings`, we now have two additional inputs, namely::

mdcnevNests: A named vector containing the names of the nests and the associated structural parameters *theta*. For each *theta*, we give the name of the associated parameter. Unlike in `apollo_nl`, the `root` is not included for `apollo_cnl` as only two-level structures are used.

mdcnevStructure: A matrix showing the allocation of alternatives to nests, with one row per nest and one column per alternative, using the same ordering as in `alternatives` and `mdcnevStructure`.

The example `Apollo_example_13.r` is a nested version of the model with an outside good used in Figure 19, i.e. `Apollo_example_12.r`. The model uses two nests, one for *mandatory* activities (work, school, private) and one for *optional* activities (all others, including the outside good). In Figure 20, we only show the part of the code that differs from the standard MDCEV model. We define two nests, and assign the appropriate θ parameter to each, where in our example, `theta_optional` is further fixed to 1 via `apollo_fixed` as its estimate was not significantly different from 1. We then describe the allocation of alternatives to nests using a matrix of ones and zeros, with one row per nest and one column per alternative, where each alternative falls into exactly one nest. Finally, we make the call to `apollo_mdcnev`. For this model, we use scaling of some of the parameters in model estimation given the earlier findings of very diverse scales for the individual parameters in the corresponding simple MDCEV model, i.e. `Apollo_example_12.r`.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

...

  ### Define nesting structure
  mdcnevNests = list(mandatory = theta_mandatory,
                     optional  = theta_optional)

  mdcnevStructure  = matrix(0, nrow=length(mdcnevNests), ncol=length(V))
  ###                        outside work school shopping private leisure
  mdcnevStructure[1,] = c(      0,    1,      1,        0,       1,       0) # mandatory
  mdcnevStructure[2,] = c(      1,    0,      0,        1,       0,       1) # optional

...

  mdcnev\_settings <- list(...
                           mdcnevNests        = mdcnevNests,
                           mdcnevStructure    = mdcnevStructure)

  P[["model"]] = apollo_mdcnev(mdcnev\_settings, functionality)
...
}

model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs,
                        estimate_settings=list(scaling=c(alpha_base=10,
                                                         gamma_work=5,
                                                         gamma_school=3,
                                                         gamma_leisure=2,
                                                         delta_work=4,
                                                         delta_school=7,
                                                         delta_shopping=4,
                                                         delta_private=4,
                                                         delta_leisure=4,
                                                         delta_work_wknd=3,
                                                         delta_school_young=3,
                                                         delta_leisure_wknd=0.3)))
```

Figure 20: MDCNEV implementation and call to apollo_estimate using scaling

5.5 Adding new model types

As already mentioned, users of *Apollo* are not restricted to those models for which functions are available in the code. Any model that yields a probability for an outcome can be used in the code and parameters for the model can be estimated using either classical or Bayesian estimation. The advantage of the predefined functions is of course that they run a large number of checks to avoid issues with mis-specification and produce output for different user needs. The level of these checks and output flexibility that a user implements for new models will vary as a function of the user's needs.

The user has the option of either creating new functions in `R` that are defined outside `apollo_probabilities` much in the same way as for example `apollo_mnl` or to simply code the probabilities for a model inside `apollo_probabilities`. An example of the latter approach is shown in Section 10.5.

Clearly, coding models as separate functions is preferable in terms of reusability as well as code organisation. Users who are interested in coding their own functions should inspect the code for some of the implemented functions for guidance, for example using `apollo_ol` as a simple start. For user defined models to be compatible with *Apollo*, a number of simple basic requirements need to be fulfilled. In particular, the function needs to take `functionality` as an argument to be able to produce different output depending on the value passed to it for `functionality`. Not all possible values discussed for `functionality` in this manual need to be implemented for new

models, but essential capabilities include the ability to deal with the following four settings for `functionality`:

estimate: Return the probabilities for each row in the data, using a vector, matrix or cube (array) depending on the presence of random coefficients (cf. Section 6.1).
validate: Return TRUE if all tests are passed, or TRUE if no tests are implemented.
zero_LL: Return the log-likelihood of the model component with all parameters at zero, set to NA if not applicable for given model.
output: Same as `functionality="estimate"`.

If the models are to be compatible with random coefficients, they furthermore need to be able to produce probabilities as three-dimensional arrays, as discussed in Section 6.1. Additionally, if the models are components of an overall model from which predictions are to be made (cf. Section 9.5, then output from the function is also needed with `functionality==prediction`, even if returning `NA` for that model component.

6 Incorporating random heterogeneity

In this section, we describe how to use the *Apollo* package to incorporate random coefficients. We look first at continuous random heterogeneity before looking at discrete mixtures (DM) and latent class (LC) models, and also a combination of the two. Finally, we discuss multi-core estimation, which is beneficial for models with random heterogeneity. In this section, we use the simple binary public transport route choice SP data described in Section 3.2.

6.1 Continuous random coefficients

6.1.1 Introduction

The *Apollo* package allows for a very general use of continuous random coefficients. The code works for models allowing for intra-individual mixing (i.e. heterogeneity at the level of individual choices), inter-individual mixing (i.e. heterogeneity at the level of individual people), as well as a mixture of the two. For background, we provide a brief recap of the discussions in Hess and Train (2011) on this topic.

In cross-sectional data, we would have a sample of N individuals, indexed as $n = 1, \ldots, N$, where each individual is observed to face only one choice situation. Let β_n be a vector of the true, but unobserved taste coefficients for consumer n. We assume that $\beta_n \, \forall n$ is *iid* over consumers with density $g(\beta \mid \Omega)$, where Ω is a vector of parameters of this distribution, such as the mean and variance. Let j_{n^*} be the alternative chosen by consumer n, such that $P_n(j_{n^*} \mid \beta)$ gives the probability of the observed choice for consumer n, conditional on β. The mixed logit probability of consumer n's chosen alternative is

$$P_n(j_{n^*} \mid \Omega) = \int_\beta P_n(j_n^* \mid \beta) \, g(\beta \mid \Omega) \, d\beta. \tag{32}$$

The log-likelihood function is then given by:

$$\mathrm{LL}(\Omega) = \sum_{n=1}^{N} \ln \left(\int_\beta P_n(j_n^* \mid \beta) \, g(\beta \mid \Omega) \, \mathrm{d}\beta \right), \tag{33}$$

Since the integrals do not take a closed form, they are approximated by simulation. The simulated log-likelihood is:

$$\mathrm{SLL}(\Omega) = \sum_{n=1}^{N} \ln \left(\frac{1}{R} \sum_{r=1}^{R} P_n(j_n^* \mid \beta_{r,n}) \right). \tag{34}$$

where $\beta_{r,n}$ gives the r^{th} draw (out of R) from $g(\beta \mid \Omega)$ for individual n. Different draws are used for the N consumers, for a total of NR draws.

When we have multiple observations per individual, we typically make the assumption that sensitivities vary across people, but stay constant across individuals. We would then have that the likelihood of the sequence of choices for person n is given by:

$$P_n(\Omega) = \int_\beta \prod_{t=1}^{T_n} P_{n,t}(j_{n,t}^* \mid \beta) \, g(\beta \mid \Omega) \, d\beta, \tag{35}$$

where $j^*_{n,t}$ be the alternative chosen by individual n in choice situation t Note that, since the same sensitivities apply to all choices by a given consumer, the integration over the density of β applies to all the consumer's choices combined, rather than each one separately.

The log-likelihood function for the observed choices is then:

$$\text{LL}(\Omega) = \sum_{n=1}^{N} \ln\left(\int_{\beta}\left[\prod_{t=1}^{T_n}\left(P_{n,t}\left(j^*_{n,t} \mid \beta\right)\right)\right] g(\beta \mid \Omega)\,\mathrm{d}\beta\right). \tag{36}$$

The simulated LL (SLL) is:

$$\text{SLL}(\Omega) = \sum_{n=1}^{N} \ln\left(\frac{1}{R}\sum_{r=1}^{R}\left[\prod_{t=1}^{T_n}\left(P_{n,t}\left(j^*_{n,t} \mid \beta_{r,n}\right)\right)\right]\right). \tag{37}$$

Note that in this formulation, the product over choice situations is calculated for each draw; the product is averaged over draws; and *then* the log of the average is taken. The SLL is the sum over consumers of the log of the average (across draws) of products. The calculation of the contribution to the SLL function for consumer n involves the computation of RT_n logit probabilities.

Instead of utilising the panel nature of the data, the model could be estimated *as if* each choice were from a different consumer. That is, the panel data could be *treated as if* they were cross-sectional. The objective function is similar to Equation 33 except that the multiple choice situations by each consumer are represented as being for different individuals:

$$\text{LL}(\Omega) = \sum_{n=1}^{N}\sum_{t=1}^{T_n} \ln\left(\int_{\beta} P_{n,t}\left(j^*_{n,t} \mid \beta\right) g(\beta \mid \Omega)\,\mathrm{d}\beta\right), \tag{38}$$

where the integration across the distribution of taste coefficients is applied to each choice, rather than to each consumer's sequence of choices. This function is simulated as:

$$\text{SLL}(\Omega) = \sum_{n=1}^{N}\sum_{t=1}^{T_n} \ln\left(\frac{1}{R}\sum_{r=1}^{R} P_{n,t}\left(j^*_{n,t} \mid \beta_{r,t,n}\right)\right). \tag{39}$$

where $\beta_{r,t,n}$ is the r^{th} draw from $g(\beta \mid \Omega)$ for choice situation t for individual n. Different draws are used for the T_n choice situations for consumer n, as well as for the N consumers. Consumer n's contribution to the SLL function utilises RT_n draws of β rather than R draws as in Equation 37, but involves the computation of the same number of logit probabilities as before, namely, RT_n. The difference is that the averaging across draws is performed before taking the product across choice situations.

We now generalise the specification on panel data to include intra-personal taste heterogeneity in addition to inter-personal heterogeneity. Let $\beta_{n,t} = \alpha_n + \gamma_{n,t}$ where α_n is distributed across consumers but not over choice situations for a given consumer, and $\gamma_{n,t}$ is distributed over choice situations as well as consumers. That is, α_n captures inter-personal variation in tastes while $\gamma_{n,t}$ captures intra-personal variation. Their densities are denoted as $f(\alpha)$ and $h(\gamma)$, respectively,[12]

[12]The mean of β_n is captured in α_n such that the mean of $\gamma_{n,t}$ is zero.

where their dependence on underlying parameters, contained collectively in Ω, is suppressed for convenience.

The LL function is given by:

$$LL\left(\Omega\right) = \sum_{n=1}^{N} \ln\left[\int_{\alpha}\left(\prod_{t=1}^{T_n}\left(\int_{\gamma} P_{n,t}\left(j_{n,t}^{*} \mid \alpha,\gamma\right) h\left(\gamma\right) \mathrm{d}\gamma\right)\right) f\left(\alpha\right) \mathrm{d}\alpha\right]. \tag{40}$$

The two levels of integration create two levels of simulation, which can be specified as:

$$SLL = \sum_{n=1}^{N} \ln\left[\frac{1}{R}\sum_{r=1}^{R}\left(\prod_{t=1}^{T_n}\frac{1}{K}\sum_{k=1}^{K}\left(P_{n,t}\left(j_{n,t}^{*} \mid \alpha_{r,n},\gamma_{k,t,n}\right)\right)\right)\right]. \tag{41}$$

This simulation uses R draws of α for consumer n, along with $K\,T_n$ draws of γ. Note that, in this specification, the same draws of γ are used for all draws of α. That is, $\gamma_{k,t,n}$ does not have an additional subscript for r. The total number of evaluations of a logit probability for consumer n is equal to $R\,K\,T_n$, compared to $R\,T_n$ when there is only inter-personal variation.

The *Apollo* package allows the user to incorporate continuous random heterogeneity for all types of models. In a model using `apollo_mnl` inside `apollo_probabilities`, we would thus obtain a mixed multinomial logit (MMNL) model, while, with a CNL core, i.e. `apollo_cnl`, we would have a mixed CNL model. Users can similarly specify and estimate mixed MDCEV models (an example is included in `Apollo_example_17.r`, and clearly also hybrid choice structures, as described in Section 7. It is straightforward to combine continuous random heterogeneity with deterministic heterogeneity for individual parameters, as shown in our example. There are very few limits imposed on what parameters can incorporate continuous random heterogeneity, opening up the use of error components for correlation across alternatives and heteroskedasticity. The parameters (in the models made available with *Apollo* for which random heterogeneity is not allowed are:

- the allocation parameters α in a CNL model
- the σ parameter in a MDCEV model
- the θ parameters in a MDCNEV model

A very flexible implementation is used that minimises the changes in the code that are required to introduce random coefficients or to change between the different layers of integration. In particular, the package works with arrays in three dimensions. For a model without continuous random coefficients, the likelihood for a model (prior to multiplying across observations for the same individual) is contained in a column vector of length O, where O is the number of observations in the data. If we introduce continuous random heterogeneity at the level of individual people, with multiple choices per person, the likelihood is given by a OxR_1 matrix, with one row per observation, and one column per draw from the random coefficients, where we use R_1 draws per random coefficient and per individual. Here, the same draws would be reused across the T_n rows for a given individual n, meaning that we would have N sets of draws, where N is the number of individuals. In the presence of additional heterogeneity at the level of individual observations, the likelihood becomes a cube of dimensions OxR_1xR_2, where in this third

dimension, different draws are used across different choices for the same individual. As described by Hess and Train (2011), a given inter-individual draw is then associated with multiple intra-individual draws. If only intra-individual heterogeneity is used, the cube collapses to an array of dimensions Ox1xR_2, i.e. a matrix but with columns going into the third dimension rather than second dimension. Depending on the type of heterogeneity (inter and inter) present in the model, different operations are required in terms of averaging across draws and multiplying across choices, and we discuss these in detail in our example below.

6.1.2 Example model specification

In what follows, we show the specification of a MMNL model with various levels of heterogeneity on the route choice data described in Section 3.2. We specify the utility of alternative j for individual n in choice situation t as:

$$V_{n,j,t} = \delta_j + \beta_{TC,n}\left(\beta_{VTT,n,t}TT_{n,j,t} + TC_{n,j,t} + \beta_{VHW,n}HW_{n,j,t} + \beta_{VCH}CH_{n,j,t}\right), \tag{42}$$

where $TT_{n,j,t}$, $TC_{n,j,t}$, $HW_{n,j,t}$ and $CH_{n,j,t}$ refer to the travel time, travel cost, headway and interchanges attributes, respectively, for alternative j in choice situation t for individual n. The treatment in terms of deterministic and random heterogeneity differs across the various parameters, as we will now explain in turn:

- Alternative specific constants (ASC) are included to capture any left-right bias in the survey, where we set $\delta_2 = 0$ for normalisation. No random or deterministic heterogeneity is incorporated for δ_1.
- The travel time coefficient $\beta_{TC,n}$ multiplies the entire remainder of the utility function, meaning that our model produces direct estimates of willingness-to-pay (WTP) measures through working in WTP space (Train and Weeks, 2005). We use a negative log-uniform distribution (cf. Hess et al., 2017) for this coefficient, capturing inter-individual heterogeneity only, with

  $$\beta_{TC,n} = -exp\left(a_{\log(\beta_{TC})} + b_{\log(\beta_{TC})} \cdot \xi_{tc,n}\right), \tag{43}$$

  where $\xi_{tc,n}$ follows a uniform distribution across individuals (but is constant across choices for the same individual), and $a_{\log(\beta_{TC})}$ and $b_{\log(\beta_{TC})}$ are the offset and range, respectively, for the Uniform distribution used for the log of β_{TC}.
- The value of travel time parameter $\beta_{VTT,nt}$ gives a direct estimate of the monetary valuation of travel time (VTT). We use a very flexible distribution for this coefficient. We being with a lognormal distribution at the inter-individual level, but add additional heterogeneity across choices for the same individual, a semi-non-parametric term to allow for deviation from the lognormal distribution at the individual level (Fosgerau and Mabit, 2013), and a deterministic multiplier to allow for differences between business and non-business travellers.

We have:

$$\begin{aligned}\beta_{VTT,n,t} = exp[&\mu_{\log(\beta_{VTT})} \\ &+ \sigma_{\log(\beta_{VTT}),inter} \cdot \xi_{tt,n} \\ &+ \sigma_{\log(\beta_{VTT}),inter,2} \cdot \xi_{tt,n}^2 \\ &+ \sigma_{\log(\beta_{VTT}),intra} \cdot \xi_{tt,nt}] \\ &\cdot (\gamma_{VTT,business} \cdot x_{business,n} + (1 - x_{business,n}))\,, \end{aligned} \tag{44}$$

where $\mu_{\log(\beta_{VTT})}$ is the estimated mean for the log of β_{VTT}, $\sigma_{\log(\beta_{VTT}),inter}$ and $\sigma_{\log(\beta_{VTT}),inter,2}$ are the standard deviation and first additional Fosgerau and Mabit (2013) polynomial term at the inter-individual level, multiplying the inter-individual level standard normally distributed $\xi_{tt,n}$ error term and its square, respectively, and $\sigma_{\log(\beta_{VTT}),intra}$ captures additional intra-individual heterogeneity by multiplying a standard normally distributed error term which also varies across individual choices, $\xi_{tt,nt}$. Finally $\gamma_{VTT,business}$ is a multiplier for business travellers, for whom $x_{business,n} = 1$. The subscript t on $\beta_{VTT,n,t}$ reflects the fact that β_{VTT} is distributed across individuals and across choices.

- The value of headway parameter $\beta_{VHW,n}$ again follows a lognormal distribution only at the inter-individual level, but with correlation with the inter-individual heterogeneity in the value of travel time parameter $\beta_{VTT,nt}$, such that:

$$\beta_{VHW,n} = exp\left(\mu_{\log(\beta_{VHW})} + \sigma_{\log(\beta_{VHW})} \cdot \xi_{hw,n} + \sigma_{\log(\beta_{VHW},\beta_{VTT})} \cdot \xi_{tt,n}\right), \tag{45}$$

  where $\xi_{hw,n}$ again follows a standard Normal distribution across individuals, and where the reuse of $\xi_{tt,n}$ from the $\beta_{VTT,nt}$ definition allows us to capture correlation betwen $\beta_{VHW,n}$ and $\beta_{VTT,nt}$ through the estimate $\sigma_{\log(\beta_{VHW},\beta_{VTT})}$.
- The value of interchanges parameter β_{VCH} is estimated without any heterogeneity, hence the lack of subscript.

We use this highly complex specification with a view to illustrating both the flexibility of the code and the ease of implementation of complex models.

6.1.3 Implementation

We explain the implementation of the model from Section 6.1.2 in four simple steps. We do not revisit obvious steps such as the definition of parameters to estimate. The model is implemented in `Apollo_example_16.r`, with simpler Mixed Logit models also available in `Apollo_example_14.r` and `Apollo_example_15.r`.

Settings

The first step is to set `mixing=TRUE` in `apollo_control`. This setting is a requirement for using continuous random heterogeneity. A user may additionally set `nCores` to a value larger than 1 in

`apollo_control`, a point we return to in Section 6.4. We would thus for example have:

```
apollo_control = list(modelName = "Apollo_example_16",
                      modelDescr = "Mixed logit model on Swiss route choice data",
                      indivID = "ID",
                      mixing = TRUE,
                      nCores = 3)
```

Draws

The second step concerns the generation of draws for random distributions. In our case, we need to produce uniformly distributed inter-individual draws for $\xi_{tc,n}$, normally distributed inter-individual draws for $\xi_{tt,n}$ and $\xi_{hw,n}$, and normally distributed intra-individual draws for $\xi_{tt,n,t}$. Draws are generated by *Apollo* whenever `mixing==TRUE` in `apollo_control`, using the settings defined in a list called `apollo_draws`. This process happens during `apollo_validateInputs`. The process used for this is illustrated in Figure 21.

```
apollo_draws = list(
  interDrawsType = "halton",
  interNDraws    = 100,
  interUnifDraws = c("draws_tc_inter"),
  interNormDraws = c("draws_hw_inter","draws_tt_inter"),
  intraDrawsType = "mlhs",
  intraNDraws    = 100,
  intraUnifDraws = c(),
  intraNormDraws = c("draws_tt_intra")
)
```

Figure 21: Defining settings for generation of draws

In `apollo_draws`, the user needs to create settings for the type of draws, both for inter (`interDrawsType`) and intra-individual (`intraDrawsType`) draws. Seven pre-defined types of draws are available in *Apollo*, namely:

pmc for pseudo-Monte Carlo draws;
halton for Halton draws Halton (1960);
mlhs for MLHS draws (Hess et al., 2006);
sobol for Sobol draws (Sobol', 1967);
sobolOwen for Sobol draws with Owen scrambling (Owen, 1995);
sobolFaureTezuka for Sobol draws with Faure-Tezuka scrambling (Faure and Tezuka, 2000); and
sobolOwenFaureTezuka for Sobol draws with both Owen and Faure-Tezuka scrambling

While the type of draws used can differ between the inter and intra-individual sets of draws, multiple sets of draws within either category will come from the same type. In our case, we use Halton draws for the inter-individual draws and MLHS draws for the intra-individual draws. When using the same type of draws for both inter and intra-individual draws, different parts of the sequence (e.g. primes for Halton) are used for the two types.

The user needs to next specify how many draws are to be used per individual for inter-individual draws, and per observation for intra-individual draws. This is set via `interNDraws` and `intraNDraws`. The number can differ between these two dimensions of integration. We use 100 inter-individual draws per parameter and per individual, and 100 intra-individual draws per parameter and per choice situation. If only inter-individual draws are to be used, then a setting of `intraNDraws = 0` is used, with a corresponding approach for intra-individual draws only. Alternatively, these settings can be omitted by the user.

Finally, the user needs to define the actual random disturbances or sets of draws, by giving each set of draws a name which can be used later in the model specification, and by determining whether the draws are Normally or Uniformly distributed, by including their names in `interNormDraws` and `interUnifDraws`, respectively, in the case of inter-individual draws, and `intraNormDraws` and `intraUnifDraws`, respectively, in the case of inter-individual draws. These two distributions (standard Normal and Uniform between 0 and 1) are used as the base for any other distributions later in the code. All the draws in our example follow standard Normal distributions, except $\xi_{tc,n}$, which comes from a Uniform distribution between 0 and 1. A user can either specify empty vectors for any settings that are not in use, such as `intraUnifDraws = c()` in our case, or omit these settings entirely.

Some users may want additional flexibility to combine different types of draws or to generate their own draws. This is possible in *Apollo* by giving the name of a user generated object in `apollo_draws$interDrawsType` and/or `apollo_draws$intraDrawsType` instead of providing one of the seven specific types of draws listed above. Using the example from Figure 21, let us assume the user wants to provide his/her own draws for inter-individual mixing, but continue to use the *Apollo* generated MLHS draws for intra-individual mixing. In that case, the user needs to replace `halton` by for example `ownInterDraws`, where this is a list, with one element per random set of draws. Each entry in the list needs to have a name, where this same set of names is then used across `interUnifDraws` and `interNormDraws` to instruct the code to either leave the draws untransformed or apply an inverse Normal CDF. The draws provided in the list `ownInterDraws` should thus be uniformly distributed. The user also still needs to specify `interNDraws` and `intraNDraws`. Each element of the list of draws provided by the user (`ownInterDraws` in our example) should be a matrix containing the user-generated draws. In the case of inter-individual draws, each matrix must have one row per individual in the database and `interNDraws` columns, while, for intra-individual draws, the matrix must have one row per observation in the database, and `intraNDraws` columns.

Random coefficients

The third step concerns the actual definition of those coefficients in the model that follow a random distribution. For this, the code includes an additional function defined outside the `apollo_probabilities` function, namely `apollo_randCoeff`. Just as with `apollo_probabilities`, this is a function that the user does not call but which the user defines.

This function takes `apollo_beta` and `apollo_inputs` as inputs and generates a new list which contains the random coefficients, incorporating any deterministic effects too. This step is shown in Figure 22, where the correspondence with Equations 43 to 45 should be clear. The contents of

```
apollo_randCoeff = function(apollo_beta, apollo_inputs){
  randcoeff = list()

  randcoeff[["b_tc"]] = -exp( mu_log_b_tc
                            + sigma_log_b_tc_inter       * draws_tc_inter )

  randcoeff[["v_tt"]] =  ( exp( mu_log_v_tt
                            + sigma_log_v_tt_inter       * draws_tt_inter
                            + sigma_log_v_tt_inter_2     * draws_tt_inter ^ 2
                            + sigma_log_v_tt_intra       * draws_tt_intra   )
                          * ( gamma_vtt_business    * business + ( 1 - business ) ) )

  randcoeff[["v_hw"]] =  exp( mu_log_v_hw
                            + sigma_log_v_hw_inter       * draws_hw_inter
                            + sigma_log_v_hw_v_tt_inter * draws_tt_inter )

  return(randcoeff)
}
```

Figure 22: The `apollo_randCoeff` function

`apollo_randCoeff` will vary across model specifications, only the first line (`randCoeff = list()`) and final line (`return(randCoeff)`) are to remain as in the example.

Model definition

The final step consists of adapting the `apollo_probabilities` function to work with random coefficients. This step is in essence the easiest as the writing of the utility functions and probabilities is equivalent to the approach used in the models without random heterogeneity. This is thanks to having defined the actual random coefficients in the `apollo_randCoeff` function which means that the user can now simply use the elements contained in the `randCoeff` list.

We illustrate this in Figure 23. As we can see, we still define the model as MNL, as this is the model structure conditional on the random coefficients. The only distinction with the earlier MNL example is that we make calls to two additional functions towards the end of `apollo_probabilities`. In the MNL example in Figure 7, we made a call to `apollo_panelProd`, which takes the product across choices for the same individual, before preparing the probabilities for output using `apollo_prepareProb`. In our MMNL model, the probabilities are however not now given by a vector with one value per choice task, but a cube with one colum per inter-individual draw in the second dimension, and one colum per intra-individual draw in the third dimension. The actual log-likelihood function for our model is thus given by:

$$L(\mathbf{\Omega}) = \prod_{n=1}^{N} \int_{\xi_{tc,n}} \int_{\xi_{tt,n}} \int_{\xi_{hw,n}} \prod_{t=1}^{T_n} \int_{\xi_{tt,n,t}} P_{j^*_{n,t}} \mathrm{d}\xi_{tt,n,t} \mathrm{d}\xi_{hw,n} \mathrm{d}, \xi_{tt,n} \mathrm{d}\xi_{tc,n}, \tag{46}$$

The two layers of integration need to be approximated using numerical simulation, where different functions are used for simulation at the inter-individual and intra-individual level. These two functions, `apollo_avgIntraDraws` and `apollo_avgInterDraws` are called as:

```
P = apollo_avgIntraDraws(P,
                         apollo_inputs,
                         functionality)
```

and

```
P = apollo_avgInterDraws(P,
                         apollo_inputs,
                         functionality)
```

In our example, we first average across intra-individual draws, using `apollo_avgIntraDraws`. We then take the product over choices, using `apollo_panelProd` before averaging across the inter-individual draws using `apollo_avgInterDraws` to obtain a column vector once again, with one row per individual. We finally call `apollo_prepareProb`.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Function initialisation: do not change the following three commands
  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### List of utilities: these must use the same names as in mnl_settings, order is irrelevant
  V = list()
  V[['alt1']] = asc_1 + b_tc*(v_tt*tt1 + tc1 + v_hw*hw1 + v_ch*ch1)
  V[['alt2']] = asc_2 + b_tc*(v_tt*tt2 + tc2 + v_hw*hw2 + v_ch*ch2)

  ### Define settings for MNL model component
  mnl_settings = list(
    alternatives  = c(alt1=1, alt2=2),
    avail         = list(alt1=1, alt2=1),
    choiceVar     = choice,
    V             = V
  )

  ### Compute probabilities using MNL model
  P[['model']] = apollo_mnl(mnl_settings, functionality)

  ### Average across intra-individual draws
  P = apollo_avgIntraDraws(P, apollo_inputs, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Average across inter-individual draws
  P = apollo_avgInterDraws(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 23: The `apollo_probabilities` function for a MMNL model

While the example here is for a MMNL model, i.e. a mixture of a MNL kernel, it is similarly possible to use for example a Mixed Nested Logit model, and in *Apollo*, this is straightforward by replacing `apollo_mnl` with `apollo_nl` and defining appropriate additional arguments.

6.1.4 Estimation

The estimation of a continuous mixed logit model uses the same routine `apollo_estimate` as our other models, and the code automatically finds the draws and random coefficients in `apollo_inputs`. This is illustrated in Figure 24, where we use 3 cores in estimation, and where the use of a MNL kernel inside the MNL model is made clear by the output.

```
> model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)

Testing probability function (apollo_probabilities)

Overview of choices for MNL model component:
                                    alt1     alt2
Times available                  3492.00  3492.00
Times chosen                     1734.00  1758.00
Percentage chosen overall          49.66    50.34
Percentage chosen when available   49.66    50.34

Attempting to split data into 3 pieces.
Number of observations per worker (thread):
worker_1 worker_2 worker_3
    1170     1170     1152
 1133.3Mb of RAM in use before splitting.
Splitting draws... Done. 2199.1Mb of RAM in use.
Splitting database... Done. 2199.4Mb of RAM in use.
Creating workers and loading libraries... Done. 2286.4Mb of RAM in use.
Copying data to workers... Done. 2286.6Mb of RAM in use (max was 3704.2Mb)


Starting main estimation
Initial function value: -2406.92
Initial gradient value:
                     asc_1                  mu_log_b_tc       sigma_log_b_tc_inter
             -12.818236428             -15.520025045              -7.743425613
                mu_log_v_tt       sigma_log_v_tt_inter     sigma_log_v_tt_inter_2
              11.501068457              -0.015208570              11.487185475
      sigma_log_v_tt_intra                 mu_log_v_hw       sigma_log_v_hw_inter
               0.004143203              33.799592529              -0.013273166
sigma_log_v_hw_v_tt_inter                      v_ch         gamma_vtt_business
               0.004037247              44.469883505               5.229098406
initial  value 2406.919551
iter   2 value 2383.804709
...
```

Figure 24: Running `apollo_estimate` for MMNL using 3 cores

6.2 Discrete mixtures and latent class

Apollo offers the same degree of flexibility with latent class and discrete mixture models as with continuous mixture models, with only the α parameters in CNL, the σ parameters in MDCEV and the θ parameters in MDCNEV need to be kept non-random.

In a latent class model, heterogeneity is accommodated by making use of separate classes with different values for the vector β in each class. With S classes, we have S instances of β, say β_1 to β_S, with the possibility of some elements staying fixed across some classes. Individual n belongs to class s with probability $\pi_{n,s}$, where $0 \leq \pi_{n,s} \leq 1 \; \forall s$ and $\sum_{s=1}^{S} \pi_{n,s} = 1$.

Let $P_{i,n,t}(\beta_s)$ give the probability of respondent n choosing alternative i in choice situation t, conditional on n falling into class s, where $P_{i,n,t}$ is typically specified as a MNL model, but this is not a requirement in theory or in *Apollo*. The unconditional (on s) choice probability is then given by:

$$P_{i,n,t}(\beta_1, \ldots, \beta_S) = \sum_{s=1}^{S} \pi_{n,s} \, P_{i,n,t}(\beta_s) \tag{47}$$

In the presence of repeated choice data, it is natural to perform the mixing at the level of individual

people, and we then have that the probability of the sequence of choices for person n is given by:

$$L_n(\beta) = \sum_{s=1}^{S} \pi_{n,s} \prod_{t=1}^{T_n} P_{j^*_{n,t}}(\beta_s) \tag{48}$$

where $\beta = \langle \beta_1, \ldots, \beta_S \rangle$, and where $j^*_{n,t}$ gives the alternative chosen by person n in choice situation t.

In the most basic version, the class allocation probabilities $\pi_{n,s}$ are constant across respondents, i.e. $\pi_{n,s} = \pi_s \, \forall n$. The real flexibility of the model arises when linking class allocation to socio-demographics, where we use a class allocation model, with typically an underlying logit structure, such that:

$$\pi_{n,s} = \frac{e^{\delta_s + g(\gamma_s, z_n)}}{\sum_{l=1}^{S}, e^{\delta_l + g(\gamma_l, z_n)}}, \tag{49}$$

where δ_s is an offset, and γ_s is a vector of parameters capturing the influence of the vector of individual characteristics z_n on the class allocation probabilities. For normalisation, δ_s is fixed to 0 for one of the S classes, as is γ_s. In a model with constant class allocation probabilities across individuals, we would only estimate the vector of constants δ.

The focus in our discussion is on latent class models rather than discrete mixtures (cf. Hess et al., 2007). In a discrete mixture model, we have S_k values for parameter β_k, where the number of possible values S_k can vary across parameters (and can be 1 for some). A weight $\pi_{n,k,s}$ is assigned to the s^{th} value for β_k for person n, with $\sum_{s=1}^{S_k} \pi_{n,k,s} = 1, \forall n, k$. We thus have $\sum_{k=1}^{K} S_k$ possible values across the K different β parameters, and each combination is possible in the model. This effectively means that the discrete mixture model can be written as a latent class model with $\prod_{k=1}^{K} S_k$ classes, where, for example, the first class might use the first value for each of the coefficients, and thus have a class allocation probability $\pi^*_{n,s} = \prod_{k=1}^{K} \pi_{n,k,1}$. Discrete mixtures can thus be estimated using software for latent class, including in *Apollo*, and an example is given in `Apollo_example_19.r`. The use of discrete mixture models leads to a larger number of parameters, as we now have separate $\pi_{n,k,s}$ for different β parameters, as well as generally a larger number of overall classes (and hence a more complex likelihood function) given that $S^* = \prod_{k=1}^{K} S_k$. Latent class models also provide a more natural way of capturing correlation in the heterogeneity across different coefficients.

To illustrate the implementation of latent class models in *Apollo*, we provide an example on the Swiss route choice data also used for the Mixed Logit model in Section 6.1.2. We develop a model with two classes, where all four marginal utility parameters (time, cost, headway and interchanges) vary across the classes, but where the ASCs are kept fixed across classes. For the class allocation model, we use two socio-demographic characteristics, namely whether an individual was on a commute journey or not, and whether they had a car available to them. This example is implemented in `Apollo_example_20.r`, where a simpler latent class model without covariates is available in `Apollo_example_18.r`.

The development of a latent class model in *Apollo* consists of two steps, which we now look at it turn.

Defining latent class parameters

We first implement a function called `apollo_lcPars`, which performs a role analagous to `apollo_randCoeff` for continuous mixtures. This is thus another function that is not called by the user but which is developed by the user for the specific model that is to be used. Like `apollo_randCoeff`, this function takes `apollo_beta` and `apollo_inputs` as inputs and generates a new list which contains the parameters that vary across classes as well as the class allocation probabilities. The contents of `apollo_lcPars` will vary across model specifications, only the first line (`lcPars = list()` and final line (`return(lpars)`) are to remain as in the example.

The implementation for our example is shown in Figure 25. We create a list called `lcPars` which contains the values for the different parameters across classes, as well as the class allocation probabilities. As can be seen from Figure 25, we first produce one element in the list for each of the four marginal utility coefficients. Each one of these elements is a list in itself, and contains the values for the coefficients across the two classes. If more classes are to be used, more entries are added into each one of these lists, where the possibility exists of keeping the values constants for some parameters across some or all of the classes (in which case the number of values still needs to be the same as the number of classes). Note that for parameters that are kept constant across all (i.e. not just some) of the classes, such as the ASCs in our example, there is no need (though also no harm) to include them in `lcPars`.

```
apollo_lcPars=function(apollo_beta, apollo_inputs){
  lcpars = list()
  lcpars[["beta_tt"]] = list(beta_tt_a, beta_tt_b)
  lcpars[["beta_tc"]] = list(beta_tc_a, beta_tc_b)
  lcpars[["beta_hw"]] = list(beta_hw_a, beta_hw_b)
  lcpars[["beta_ch"]] = list(beta_ch_a, beta_ch_b)

  V=list()
  V[["class_a"]] = delta_a + gamma_commute_a*commute + gamma_car_av_a*car_availability
  V[["class_b"]] = delta_b + gamma_commute_b*commute + gamma_car_av_b*car_availability

  mnl_settings = list(
    alternatives = c(class_a=1, class_b=2),
    avail        = 1,
    choiceVar    = NA,
    V            = V
  )
  lcpars[["pi_values"]] = apollo_mnl(mnl_settings, functionality="raw")

  lcpars[["pi_values"]] = apollo_firstRow(lcpars[["pi_values"]], apollo_inputs)

  return(lcpars)
}
```

Figure 25: The `apollo_lcPars` function

We next calculate the class allocation probabilities, i.e. $\pi_{n,s}, \forall n, s$. We use a MNL model for the class allocation probabilities, and thus produce utility functions for the two classes. The utility for the second class could in our case simply be set to 0 as the parameters are all normalised to 0 in `apollo_fixed`. We then use the `apollo_mnl` function to calculate the probabilities, with two alternatives which are always both available (hence `avail=1`). Two points need noting here. First, unlike in other applications of the in-built functions, we now explicitly use the functionality

`raw` when calling `apollo_mnl` - this ensures that the probabilities are returned for all alternatives, or in this case all classes. When using `raw`, we also do not need to define the chosen alternative, and thus set `choiceVar=NA`.

At this point, we obtain a value for the probability for the two classes for each row in the data. However, for the calculation in Equation 48, we require the class allocation probabilities at the individual rather than observation level, i.e. $\pi_{n,s}$. This is achieved by running the additional function `apollo_firstrow` on the part of `lcPars` containing the class allocation probabilities. This is a general function that can be applied to probabilities, data elements, etc. In particular, by calling:

```
x = apollo_firstrow(x,
                    apollo_inputs)
```

we replace `x` by a version where only the first entry for each individual is retained. The object `x` can be a vector, matrix or cube. In our example, calling `apollo_firstrow(lcPars[["pi_values"]],apollo_inputs)` retains the first row in each element of `lcPars[["pi_values"]]` for each individual.

Model definition

We next turn to the calculation of the actual latent class choice probabilities, a process that is illustrated in Figure 26. As can be seen, we first create a generic version of `mnl_settings` that contains those settings which will be constant across classes, namely the alternatives, availabilities and choice variable.

We then incorporate a loop over classes, where we calculate the utilities for the two alternatives in each class, using the appropriate values for those parameters that vary across classes. In each class, we then update `mnl_settings` to use the utilities in that specific class, before creating one element in the `P` list for each class. The reader will note that in class s, we are using the coefficients for that class (e.g. `beta_tc[[s]]` uses the s^{th} element in `beta_tc` created in `apollo_lcPars`), and the call to `apollo_mnl` in each class uses the appropriate utilities for that class as these are updated inside the overall `mnl_settings` using `mnl_settings$V = V` in each step of the loop. In our example, we calculate the within class probabilities using a MNL model, where it would again also be possible to use different models inside the latent class structure, e.g. Nested Logit. In preparation for the averaging across classes, we take the product across choices for the same individual in each class, using `apollo_panelProd`, in line with Equation 48.

We now have the likelihoods in each class, i.e. for class s, we have $L_{n,s} = \prod_{t=1}^{T_n} P_{j^*_{n,t}}(\beta_s)$. The remaining step is to take the weighted average across classes, i.e. $\sum_{s=1}^{S} \pi_{n,s} L_{n,s}$. This is achieved by the `apollo_lc` function, which uses the within class probabilities contained in the S existing elements of `P`, multiplies each one by the appropriate class allocation probability in `pi_values`, and then sums across classes. This function is called as:

```
P[["model"]] = apollo_lc(lc_settings,
                         apollo_inputs,
                         functionality)
```

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Define settings for MNL model component that are generic across classes
  mnl_settings = list(
    alternatives = c(alt1=1, alt2=2),
    avail        = list(alt1=1, alt2=1),
    choiceVar    = choice
  )

  ### Loop over classes
  s=1
  while(s<=2){

    ### Compute class-specific utilities
    V=list()
    V[['alt1']]  = asc_1 + beta_tc[[s]]*tc1 + beta_tt[[s]]*tt1 + beta_hw[[s]]*hw1 + beta_ch[[s]]*ch1
    V[['alt2']]  = asc_2 + beta_tc[[s]]*tc2 + beta_tt[[s]]*tt2 + beta_hw[[s]]*hw2 + beta_ch[[s]]*ch2

    mnl_settings$V = V

    ### Compute within-class choice probabilities using MNL model
    P[[s]] = apollo_mnl(mnl_settings, functionality)

    ### Take product across observation for same individual
    P[[s]] = apollo_panelProd(P[[s]], apollo_inputs ,functionality)

    s=s+1}

  ### Compute latent class model probabilities
  lc_settings   = list(inClassProb = P, classProb=pi_values)
  P[["model"]] = apollo_lc(lc_settings, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 26: Implementing choice probabilities for latent class

The list `lc_settings` contains two elements, namely:

- `inClassProb`: A list of in class probabilities, i.e. the $L_{n,s}$ for different classes. These need to already have all continuous random heterogeneity averaged out and contain one entry per individual, i.e. having been multiplied across observations for the same individual.
- `classProb`: A list of class allocation probabilities, which can be either scalars (if constant across people), vectors (if using only deterministic heterogeneity) or matrices or cubes (if including continuous random heterogeneity in the class allocation probabilities).

The output from this function is the actual latent class model probability in Equation 48 and is stored in the `model` component of the list `P`.

6.3 Combining latent class with continuous random heterogeneity

Apollo also allows users to combine continuous random heterogeneity with latent classes (cf. Greene and Hensher, 2013). Continuous heterogeneity can be allowed for both in the within-class probabilities and in the class membership probabilities. Specifically, let us assume that the vector

π is distributed according to $f(\pi \mid \Omega_\pi)$ where Ω_π is a vector of parameters, while the vector β_s, which contains the parameters for the within-class model in class s is distributed according to $g_s(\beta_s \mid \Omega_{\beta_s})$, where Ω_{β_s} is a vector of parameters, and where $\Omega_\beta = \langle \Omega_{\beta_1}, ..., \Omega_{\beta_S} \rangle$. We then have:

$$L_n(\Omega_\pi, \Omega_\beta) = \int_\pi \sum_{s=1}^{S} \pi_{n,s} \left(\int_{\beta_s} \prod_{t=1}^{T_n} P_{j^*_{n,t}}(\beta_s)\, g_s(\beta_s \mid \Omega_{\beta_s})\, \mathrm{d}\beta_s \right) f(\pi \mid \Omega_\pi)\, \mathrm{d}\pi. \tag{50}$$

The integration across the distribution for heterogeneity in the within-class model is thus carried out prior to averaging across classes, while the integration across the distribution for heterogeneity in the class-allocation model is carried out outside the averaging across classes. For estimation, this implies averaging across draws in two distinct places, as we will now illustrate.

We extend the model from Section 6.2 by allowing the travel time coefficient to follow a negative lognormal distribution, with separate parameters in the two classes. In addition, we allow the constant in the class allocation model for the first class, i.e. δ_a, to follow a Normal distribution. The specification of the random parameters is illustrated in Figure 27, and is available in `Apollo_example_21.r`.

We have that `draws_tt` and `draws_pi` are standard Normal draws defined in `apollo_draws` (not shown here). We then use a negative Lognormal distribution for `tt_a` and `tt_b`, and a Normal distribution for `delta_a`. These random time coefficients are then also used inside `apollo_lcPars` when defining `lcPars[["tt"]]`, while the randomly distributed `delta_a` is used when defining `V[["class_a"]]`.

The definition of the model probabilities differs from that of the simple latent class model in Figure 26 in only two ways. In particular, as seen in Figure 28, in line with Equation 50, we now average across the random draws in the within class likelihoods via `P[[s]] = apollo_avgInterDraws(P[[s]],apollo_inputs,functionality)`, after taking the product across observations for the same individual using `apollo_panelProd`. This gives one likelihood for the observed choices for each person within each class. We then perform the weighted summation across classes using `P[["model"]] = apollo_lc(lc_settings, apollo_inputs, functionality)`. As `pi_values` again incorporates random terms, we now have a version of the combined model likelihood for each draw for each individual. As a final step we then average across the continuous heterogeneity in the class allocation probabilities, using `P[["model"]] = apollo_avgInterDraws(P[["model"]], apollo_inputs, functionality)` to again give one value per person.

6.4 Multi-threading capabilities

Apollo allows for multi-threaded estimation for classical estimation[13], leading to significant estimation speed improvements for some models. This can easily be activated by specifying the number of threads to use in `apollo_control$nCores`. The recommended number of threads is equal to the number of available processor cores in the machine minus one, which can be

[13] When using Bayesian estimation, the reliance on `RSGHB` means only single core processing is possible.

```
apollo_randCoeff = function(apollo_beta, apollo_inputs){
  randcoeff = list()

  randcoeff[["tt_a"]] = -exp(log_tt_a_mu + log_tt_a_sig*draws_tt)
  randcoeff[["tt_b"]] = -exp(log_tt_b_mu + log_tt_b_sig*draws_tt)
  randcoeff[["delta_a"]] = delta_a_mu  + delta_a_sig*draws_pi

  return(randcoeff)
}

apollo_lcPars = function(apollo_beta, apollo_inputs){
  lcpars = list()
  lcpars[["tt"]] = list(tt_a, tt_b)
  lcpars[["tc"]] = list(tc_a, tc_b)
  lcpars[["hw"]] = list(hw_a, hw_b)
  lcpars[["ch"]] = list(ch_a, ch_b)

  V=list()
  V[["class_a"]] = delta_a      + gamma_commute_a*commute + gamma_car_av_a*car_availability
  V[["class_b"]] = delta_b      + gamma_commute_b*commute + gamma_car_av_b*car_availability

  mnl_settings = list(
    alternatives = c(class_a=1, class_b=2),
    avail        = 1,
    choiceVar    = NA,
    V            = V
  )
  lcpars[["pi_values"]] = apollo_mnl(mnl_settings, functionality="raw")

  lcpars[["pi_values"]] = apollo_firstRow(lcpars[["pi_values"]], apollo_inputs)

  return(lcpars)
}
```

Figure 27: The `apollo_randCoeff` and `apollo_lcPars` functions for a latent class model with continuous random heterogeneity

determined by typing `parallel::detectCores()` in the R console. The use of multi-threaded estimation comes with some restrictions.

apollo_probabilities can only access its arguments: In other words, the likelihood can only use data stored inside `apollo_beta` and `apollo_inputs`, where the latter combines `database`, `apollo_control`, `draws`, `apollo_randCoeff` and `apollo_lcPars`. All other variables created by the user in the global environment before estimation cannot be accessed. This issue is easily avoided by creating any new variables inside the `database` object prior to calling `apollo_validateInputs`, which is good practice anyway.

Data splitting: The dataset is split among several threads, so statistics such as the mean, maximum and minimum of variables, among others, will not be reliably calculated during estimation when using multi-threading. To avoid this issue, any such statistic (for example the mean income in our MNL example in Section 4.2) need to be calculated before estimation and saved as a new variable inside `database`[14].

Increased memory consumption: Memory consumption is approximately doubled when using multi-threading. This is because the dataset and draws (usually the biggest objects

[14]To illustrate this issue, in our earlier example in Section 4.5.2, we created a variable called `mean_income` inside the database, prior to calling `apollo_validateInputs`, by calling `database$mean_income = mean(database$income)`. This ensures that with multi-threading, the same mean income would be used in each core, while, if the variable had been created inside `apollo_probabilities`, a different mean income would have been used across cores.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Define settings for MNL model component that are generic across classes
  mnl_settings = list(
    alternatives = c(alt1=1, alt2=2),
    avail        = list(alt1=1, alt2=1),
    choiceVar    = choice
  )

  ### Loop over classes
  s=1
  while(s<=2){

    ### Compute class-specific utilities
    V=list()
    V[['alt1']]  = asc1 + tc[[s]]*tc1 + tt[[s]]*tt1 + hw[[s]]*hw1 + ch[[s]]*ch1
    V[['alt2']]  = asc2 + tc[[s]]*tc2 + tt[[s]]*tt2 + hw[[s]]*hw2 + ch[[s]]*ch2

    mnl_settings$V = V

    ### Compute within-class choice probabilities using MNL model
    P[[s]] = apollo_mnl(mnl_settings, functionality)

    ### Take product across observation for same individual
    P[[s]] = apollo_panelProd(P[[s]], apollo_inputs ,functionality)

    ### Average across inter-individual draws within classes
    P[[s]] = apollo_avgInterDraws(P[[s]], apollo_inputs, functionality)

    s=s+1
  }

  ### Compute latent class model probabilities
  lc_settings   = list(inClassProb = P, classProb=pi_values)
  P[["model"]] = apollo_lc(lc_settings, apollo_inputs, functionality)

  ### Average across inter-individual draws in class allocation probabilities
  P[["model"]] = apollo_avgInterDraws(P[["model"]], apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 28: Implementing choice probabilities for latent class with continuous random heterogeneity

in memory) need to be duplicated, split, and copied into several threads. Any threads after the second thread will also increase memory requirements, but to a much lesser extent.

Speed gains are dependent on the model: In general, models using few iterations that each take a long time will benefit the most. This applies to models using big datasets or a large number of draws in the case of mixture models. Marginal speed gains also decrease with the number of threads used. For small models, speed gains due to multi-threading might be negligible, or even negative due to overhead.

To help decide how many cores to use, we provide the function `apollo_speedTest`, which calculates the loglikelihood function several times using different number of threads and draws,

and reports both the calculation time and the memory usage. This function is called as:

```
apollo_speedTest(apollo_beta,
                 apollo_fixed,
                 apollo_probabilities,
                 apollo_inputs,
                 speedTest_settings)
```

The final argument, `speedTest_settings`, is optional and allows the user to change the following settings:

- `nDrawsTry`: A vector with the number of draws to try (default is `c(100, 200, 500)`). Note that this may need to be reduced for very complex models if memory issues arise.
- `nCoresTry`: A vector with the number of threads to try (default is to try all cores present in the machine).
- `nRep`: An integer setting the number of times `apollo_probabilities` is calculated for each possible pair of elements from `nDrawsTry` and `nCoresTry` (default is `10`), ensuring stable results for the calculation of runtimes.

We illustrate the use of this function in Figure 29, which shows a big benefit especially by using a second core. When running `apollo_speedTest`, progress and results of the test are printed to the console. Each row displays the set-up, progress, and results of a given configuration. The first column (*nCores*) indicates the number of computational threads in use, i.e. how many processor cores are being used simultaneously by `R`. The second (*inter*) and third (*intra*) columns indicate the number of inter-individual and intra-individual draws used. The third column (*progress*) indicates the progress of the test for each set-up, each dot representing 10% of the repetitions requested. The fifth column (*sec/LLCal*) indicates the average time in seconds required to complete one evaluation of the `apollo_probabilities` function. The sixth and last column (*RAM(MB)*) presents a lower bound of the memory required to evaluate the `apollo_probabilities` function. After completing the test, results are summarised in a table indicating the time required to evaluate `apollo_probabilities` under each configuration, as well as in a plot.

```
> speedTest_settings=list(nDrawsTry = c(50, 75, 100),nCoresTry = 1:3,nRep = 10)

> apollo_speedTest(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, speedTest_settings)

         ___Draws___                    sec/
nCores  inter  intra   progress    LLCall  RAM(MB)
      1     50     50  ..........    2.54   1440.2
      2     50     50  ..........    1.65   1764.8
      3     50     50  ..........    1.61   1793.7
      1     75     75  ..........    5.59   1773.3
      2     75     75  ..........    4.06   2430.9
      3     75     75  ..........    3.57   2459.9
      1    100    100  ..........   10.90   2239.5
      2    100    100  ..........    7.35   3363.3
      3    100    100  ..........    6.89   3392.3

Summary of results (sec. per call to LL function)
        draws50 draws75 draws100
cores1   2.5383  5.5874   10.9029
cores2   1.6472  4.0645    7.3518
cores3   1.6055  3.5699    6.8887
```

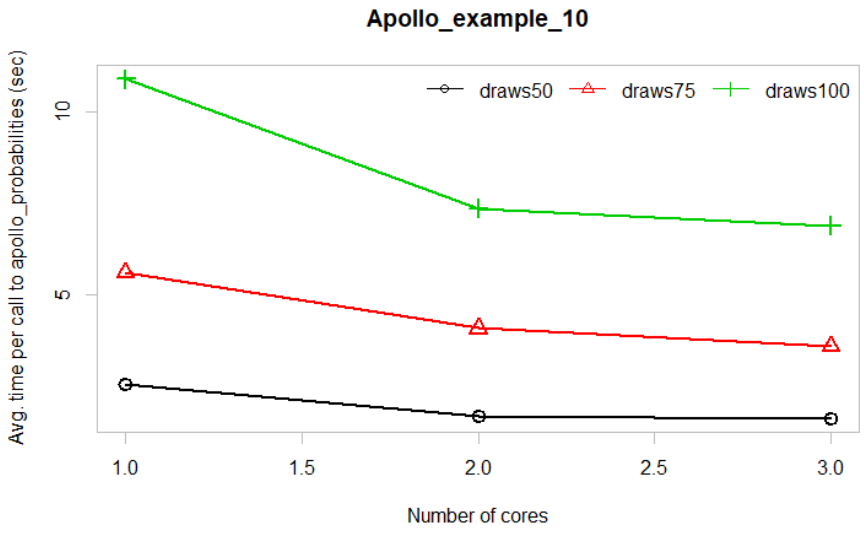

Figure 29: Running `apollo_speedTest`

7 Joint estimation of multiple model components

The use of models made up of several separate components is made possible by the function `apollo_combineModels`, which is called as follows:

```
P = apollo_combineModels(P,
                         apollo_inputs,
                         functionality)
```

This function takes the list `P` which contains several individual model components and produces a combined model. There is no limit on the number of subcomponents. The obvious case is estimation, where, with $L_{n,m}$ giving the likelihood of model component m for person n, the overall likelihood for person n is given by $L_n = \prod_{m=1}^{M} L_{n,m}$ (not showing here the presence of any integration over random terms, which would be carried out outside the product). The function `apollo_combineModels` creates the `model` object inside `P` as the product across individual components - when working with multiple model components, the individual components should thus not be called `model` themselves.

The most widely used case in recent years of models with multiple components is that of hybrid choice models. Before we turn to that example, we illustrate the joint estimation capabilities of *Apollo* by looking at two simpler cases of combining two models, namely the case of joint estimation on RP and SP data, and the estimation of best-worst data.

7.1 Joint estimation on RP and SP data

The example `Apollo_example_22.r` uses the mode choice data used also for the earlier MNL model (cf. Section 4.5.2) but combines the RP and SP data. To allow for scale differences between the two data sources (Bradley and Daly, 1996; Hensher et al., 1998), we incorporate separate scale parameters μ_{RP} and μ_{SP} where the former is kept fixed to 1 for normalisation.

The basic setup is the same as in Section 4.5.2 with the exception that we omit `database = subset(database,database$SP==1)` used earlier in Figure 3 as we now utilise the entire sample. The earlier part of the code remains the same as in Figure 7 and is largely omitted here for conciseness of presentation - this includes the definition of `mu_RP` and `mu_SP` in `apollo_beta`, and the inclusion of `mu_RP` in `apollo_fixed`.

The key differences arise in the `apollo_probabilities` function, where we illustrate this in Figure 30. The definition of the utilities remains the same, with the difference that they are now calculated for all rows in the data, i.e. for RP rows as well as SP rows. In the example used here, the service quality attributes are coded as zero for the RP data and thus do not enter into the utility calculation for these rows. Special care is required in datasets where attributes for part of the data are coded for example as -999. The user can then either calculate the utilities separately for RP and SP or add additional conditional statements.

We first calculate the probabilities for RP choices. The definition of `mnl_settings` differs from that in the SP model in Figure 7 in that we multiply the utilities by the RP scale parameter, μ_{RP}, e.g. $V_{i,n,t,RP} = \mu_{RP} V_{i,n,t}$, where we use `lapply` to cycle over the list of utilities. RP probabilities

should only be calculated for RP rows in the data, and we thus include `rows=(RP==1)`, meaning that for SP rows in the data, the probability of RP choices is simply fixed to 1 so as not to contribute to model estimation. We then make the call to `apollo_mnl`, saving the output not in `P[["model"]]` which is reserved for the overall model, but in a subcomponent called `P[["RP"]]`. For the SP part of the data, we only need to change two components[15] in `mnl_settings`, namely using `lapply(V, "*", mu_SP)` instead of `lapply(V, "*", mu_RP)`, and `rows = (SP==1)` instead of `rows = (RP==1)`. We then calculate the probabilities for the SP rows in the data.

The probability for the combined model is obtained by multiplying the RP and SP components together in `P[["model"]]`, which is the component used for estimation. Rather than doing this manually, we use `P = apollo_combineModels(P,apollo_inputs,functionality)`, as this function also prepares different output depending on the setting of `functionality`, allowing the use of joint models also in prediction, for example. The multiplication of probabilities across all observations for the same respondent happens after combining the two model components, using `apollo_panelProd`.

A subset of the model output is shown in Figure 31. We see that the model reports the joint log-likelihood as well as the subcomponents for the two separate model components, and we obtain a scale parameter for the SP data (μ_{SP}) which is significantly larger than 1. It should be noted that in models with multiple components, the output in terms of diagnostics (cf. Figure 8) can become quite verbose as this information is reported for each model component, and a user may thus set `noDiagnostics` to FALSE in `apollo_control`. The diagnostics are reported in the order that the model components appear in the overall structure, in this case RP before SP, with both being MNL components.

7.2 Joint best-worst model

Many stated choice surveys ask respondents for the most and least preferred alternatives, otherwise known as best-worst or BW (Lancsar et al., 2013). Although there is evidence that the behaviour in these two stages is not necessarily symmetrical (Giergiczny et al., 2017), such data is commonly analysed jointly, using in the simplest form a model where the probability for a given person n in choice task t is given by:

$$P_{n,t} = \frac{e^{V_{b_{n,t}}}}{\sum_{j=1} e^{V_{j,n,t}}} \cdot \frac{e^{-\mu_w V_{w_{n,t}}}}{\sum_{j \neq b_{n,t}} e^{-\mu_w V_{j,n,t}}}, \tag{51}$$

where $b_{n,t}$ is the most preferred alternative for respondent n in choice situation t while $w_{n,t}$ is the least preferred option. The above specification assumes that the best option is chosen first, and the worst is then chosen from the remaining set of alternatives, where the alternative with the lowest utility has the highest probability of being chosen (given the multiplication of the utilities by -1). A scale difference between the two stages is allowed for with the estimation of μ_w.

We illustrate the estimation of best-worst choice models on the drug choice data, using the same utility specification as in 5.3.1, but looking at the best and worst choice stages only,

[15] As in previous examples, we do not rewrite the entire `mnl_settings` but just update individual components inside it.

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Create alternative specific constants and coefficients using interactions with socio-demographics
  asc_bus_value   = asc_bus  + asc_bus_shift_female * female
  asc_air_value   = asc_air  + asc_air_shift_female * female
  asc_rail_value  = asc_rail + asc_rail_shift_female * female
  b_tt_car_value  = b_tt_car + b_tt_shift_business * business
  b_tt_bus_value  = b_tt_bus + b_tt_shift_business * business
  b_tt_air_value  = b_tt_air + b_tt_shift_business * business
  b_tt_rail_value = b_tt_rail + b_tt_shift_business * business
  b_cost_value    = ( b_cost +  b_cost_shift_business * business ) * ( income / mean_income ) ^
      ↪ cost_income_elast

  ### List of utilities (before applying scales): these must use the same names as in mnl_settings,
      ↪ order is irrelevant
  V = list()
  V[['car']]  = asc_car        + b_tt_car_value  * time_car                          + b_cost_value *
      ↪ cost_car
  V[['bus']]  = asc_bus_value  + b_tt_bus_value  * time_bus  + b_access * access_bus  + b_cost_value *
      ↪ cost_bus
  V[['air']]  = asc_air_value  + b_tt_air_value  * time_air  + b_access * access_air  + b_cost_value *
      ↪ cost_air   + b_no_frills * ( service_air == 1 )  + b_wifi * ( service_air == 2 )  + b_food * (
      ↪ service_air == 3 )
  V[['rail']] = asc_rail_value + b_tt_rail_value * time_rail + b_access * access_rail + b_cost_value *
      ↪ cost_rail  + b_no_frills * ( service_rail == 1 ) + b_wifi * ( service_rail == 2 ) + b_food * (
      ↪ service_rail == 3 )

  ### Compute probabilities for the RP part of the data using MNL model
  mnl_settings = list(
    alternatives  = c(car=1, bus=2, air=3, rail=4),
    avail         = list(car=av_car, bus=av_bus, air=av_air, rail=av_rail),
    choiceVar     = choice,
    V             = lapply(V, "*", mu_RP),
    rows          = (RP==1)
  )

  P[['RP']] = apollo_mnl(mnl_settings, functionality)

  ### Compute probabilities for the SP part of the data using MNL model
  mnl_settings$V = lapply(V, "*", mu_SP)
  mnl_settings$rows = (SP==1)

  P[['SP']] = apollo_mnl(mnl_settings, functionality)

  ### Combined model
  P = apollo_combineModels(P, apollo_inputs, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 30: Joint RP-SP model on mode choice data

where we allow for a difference in scale between the two stages. This example is available in `Apollo_example_23.r`, where we do not repeat the code showing the specification of the utilities, which is the same as in Figure 17. The model component for the first preference (best) is as in the exploded logit model. For the worst stage, we make three changes. We first adapt the availabilities by making the alternative chosen as the best alternative in the first stage unavailable in the second stage (assuming sequential choices). We next change the dependent variable to be `worst` rather than `best`, before multiplying the utilities by the negative of μ_{RP}, as in Equation 51. Of course,

```
> apollo_modelOutput(model)
Model name                              : Apollo_example_22
Model description                       : RP-SP model on mode choice data

LL(final, whole model)                  : -5802.644
  LL(RP)                                : -971.2433
  LL(SP)                                : -4831.4

Estimates:
                         Estimate Std.err. t.ratio(0) Rob.std.err. Rob.t.ratio(0)
asc_car                    0.0000       NA         NA          NA             NA
asc_bus                    0.1262   0.2811       0.45      0.2618           0.48
asc_air                   -0.3960   0.1837      -2.16      0.1776          -2.23
asc_rail                  -0.9787   0.1806      -5.42      0.1774          -5.52
asc_bus_shift_female       0.1818   0.0647       2.81      0.0714           2.55
asc_air_shift_female       0.1344   0.0455       2.95      0.0473           2.84
asc_rail_shift_female      0.0980   0.0366       2.67      0.0384           2.55
b_tt_car                  -0.0064   0.0005     -12.62      0.0005         -13.02
b_tt_bus                  -0.0105   0.0010     -10.94      0.0009         -12.05
b_tt_air                  -0.0087   0.0015      -5.91      0.0014          -6.06
b_tt_rail                 -0.0038   0.0009      -4.17      0.0009          -4.29
b_tt_shift_business       -0.0032   0.0003      -9.25      0.0003          -9.19
b_acc                     -0.0106   0.0015      -6.89      0.0015          -7.23
b_cost                    -0.0382   0.0025     -15.55      0.0024         -15.78
b_cost_shift_business      0.0167   0.0016      10.19      0.0015          10.76
cost_income_elast         -0.6131   0.0292     -20.99      0.0297         -20.62
b_no_frills                0.0000       NA         NA          NA             NA
b_wifi                     0.5233   0.0430      12.16      0.0435          12.02
b_food                     0.2201   0.0308       7.14      0.0315           6.99
mu_RP                      1.0000       NA         NA          NA             NA
mu_SP                      1.9941   0.1263      15.79      0.1225          16.28

The following parameters were fixed (they have no std.err.):
asc_car, b_no_frills, mu_RP

Overview of choices for MNL model component:
                                  car    bus    air    rail
Times available                778.00 902.00 752.00 874.00
Times chosen                   332.00 126.00 215.00 327.00
Percentage chosen overall       33.20  12.60  21.50  32.70
Percentage chosen when available 42.67  13.97  28.59  37.41

Overview of choices for MNL model component:
                                  car     bus     air     rail
Times available               5446.00 6314.00 5264.00 6118.00
Times chosen                  1946.00  358.00 1522.00 3174.00
Percentage chosen overall       27.80    5.11   21.74   45.34
Percentage chosen when available 35.73   5.67   28.91   51.88
```

Figure 31: On screen output for RP-SP model

the same result could be achieved by making use of the `apollo_el` function with two stages, using the best and worst outcomes and a negative scale multiplier for the second stage. We show the use of two separate components here as this would also allow a user to change the actual utility function between the best and worst stage by for example allowing for differences in individual β terms going beyond just a generic scale difference (by redefining the utilities for the worst stage). This is not possible with `apollo_el` which allows for scale differences only.

7.3 Hybrid choice model

We next turn to the use of *Apollo* for hybrid choice models (see Abou-Zeid and Ben-Akiva, 2014, for a recent overview), where we look at a simple implementation of a model with a single latent variable on the drug choice data described in Section 3.3. We use a dummy coded specification for the three categorical variables, along with a continuous specification for risk and cost. We specify a structural model for the latent variable that uses the three socio-demographic characteristics

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

...

  ### Compute probabilities for 'best' choice using MNL model
  mnl_settings = list(
    alternatives = c(alt1=1, alt2=2, alt3=3, alt4=4),
    avail        = list(alt1=1, alt2=1, alt3=1, alt4=1),
    choiceVar    = best,
    V            = V
  )
  P[['choice_best']] = apollo_mnl(mnl_settings, functionality)

  ### Compute probabilities for 'worst' choice using MNL model
  mnl_settings$avail        = list(alt1=(best!=1), alt2=(best!=2), alt3=(best!=3), alt4=(best!=4))
  mnl_settings$choiceVar    = worst
  mnl_settings$V            = lapply(V,"*",-mu_worst)

  P[['choice_worst']] = apollo_mnl(mnl_settings, functionality)

  ### Combined model
  P = apollo_combineModels(P, apollo_inputs, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 32: Best-Worst model on drug choice data

included in the data, and then use this latent variable in the utilities for the two branded alternatives as well as in the measurement models for the four attitudinal indicators. In our example, we do not incorporate additional random heterogeneity not linked to the latent variable, but this is entirely straightforward to do by including additional terms in `apollo_randCoeff`. Similarly, it is possible to combine latent variables with latent class structures. Indeed, latent variables are simply additional random components in a model.

Two different versions of the model are provided. The first of these, `Apollo_example_24.r` uses an ordered logit model for the indicators, as discussed by Daly et al. (2012b). The second example, `Apollo_example_25.r` uses the common simplification of treating the indicators as being normally distributed. We will now look at these two models in turn.

Specifically, we have that the latent variable for individual n is given by:

$$\alpha_n = \gamma' z_n + \eta_n, \tag{52}$$

where z_n is a vector combining the three socio-demographic variables for individual n, γ is a vector of estimated parameters capturing the impact of these variables on α_n and η_n is a random disturbance which follows a standard Normal distribution across individuals, i.e. $\eta_n \sim N(0,1)$.

The utility for alternative j in choice situation t for individual n is given by:

$$\begin{aligned} V_{j,n,t} = & \sum_{s=1}^{5} \beta_{brands_s} \cdot \left(x_{brand_{j,n,t}} == s\right) \\ & + \sum_{s=1}^{6} \beta_{country_s} \cdot \left(x_{country_{j,n,t}} == s\right) \\ & + \sum_{s=1}^{3} \beta_{characteristic_s} \cdot \left(x_{characteristic_{j,n,t}} == s\right) \\ & + \beta_{side\_effects} \cdot x_{side\_effects_{j,n,t}} \\ & + \beta_{price} \cdot x_{price_{j,n,t}} \\ & + \lambda \cdot \alpha_n \cdot (j \leq 2)\,. \end{aligned} \tag{53}$$

For the first five rows, the same specification as in Section 5.3.1 and Section 7.2 is used, with dummy coding for the categorical variables and a continuous treatment of risk and price. Finally, the inclusion of the latent variable, i.e. $\lambda \cdot \alpha_n$ only applies to the first two alternatives, i.e. the branded products. We thus get that the likelihood of the observed sequence of T_n choices for person n, conditional on β and α_n, is given by:

$$L_{C_n}\left(\beta, \alpha_n\right) = \prod_{t=1}^{T_n} \frac{e^{V_{j^*_{n,t}}}}{\sum_{j=1}^{4} e^{V_{j,n,t}}}, \tag{54}$$

where $j^*_{n,t}$ is the alternative chosen by respondent n in task t.

The latent variable is also used to explain the value of the four attitudinal questions, where two different specifications are used in our example.

With the ordered logit model, we have that:

$$L_{I_n,ordered}\left(\tau, \zeta, \alpha_n\right) = \prod_{i=1}^{4} \left(\sum_{s=1}^{S} \delta_{(I_{n,i}=s)} \left[\frac{e^{\tau_{i,s} - \zeta_i \alpha_n}}{1 + e^{\tau_{i,s} - \zeta_i \alpha_n}} - \frac{e^{\tau_{i,s-1} - \zeta_i \alpha_n}}{1 + e^{\tau_{i,s-1} - \zeta_i \alpha_n}} \right] \right), \tag{55}$$

where ζ_i is an estimated parameter that measures the impact of α_n on the attitudinal indicator I_i, and $\tau_{i,.}$ is a vector of threshold parameters for this indicator.

With the continuous measurement model, we instead have that:

$$L_{I_n,normal}\left(\sigma, \zeta, \alpha_n\right) = \prod_{i=1}^{4} \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{\left(I_{n,i} - \bar{I}_i - \zeta_i \alpha_n\right)^2}{2\sigma_i^2}} \tag{56}$$

where ζ_i is an estimated parameter that measures the impact of α_n on the attitudinal indicator I_i, and σ_i is an estimated standard deviation. By subtracting the mean of the indicator across the sample, i.e. using $I_{n,i} - \bar{I}_i$, we avoid the need to estimate the mean of the normal density.

The combined log-likelihood for the model is then given by:

$$LL\left(\gamma,\zeta,\tau,\beta\right)=\sum_{n=1}^{N}log\int_{\eta_n}L_{C_n}\left(\beta,\alpha_n\right)L_{I_n,ordered}\left(\tau,\zeta,\alpha\right)\phi\left(\eta_n\right)\mathrm{d}\eta_n, \tag{57}$$

with the ordered model, where we would replace $L_{I_n,ordered}\left(\tau,\zeta,\alpha\right)$ by $L_{I_n,normal}\left(\sigma,\zeta,\alpha\right)$ for the continuous measurement model (Equation 56 instead of Equation 55). This log-likelihood function requires integration over the random component in the latent variable, where this integral is then approximated using numerical simulation.

For conciseness, we do not here reproduce the obvious parts of the code relating to the definition of parameters or basic settings. In Figure 33, we start by creating 100 inter-individual standard Normal draws for η based on Halton draws. We then define a single random component inside `apollo_randCoeff`, where this is for the latent attitude α_n, in line with Equation 52, which includes deterministic heterogeneity through the inclusion of socio-demographic effects.

```
### Set parameters for generating draws
apollo_draws = list(
  interDrawsType="halton",
  interNDraws=100,
  interUnifDraws=c(),
  interNormDraws=c("eta"),

  intraDrawsType='',
  intraNDraws=0,
  intraUnifDraws=c(),
  intraNormDraws=c()
)

### Create random parameters
apollo_randCoeff=function(apollo_beta, apollo_inputs){
  randcoeff = list()

  randcoeff[["LV"]] = gamma_reg_user*regular_user + gamma_university*university_educated + gamma_age_50*
      ↪ over_50 + eta

  return(randcoeff)
}
```

Figure 33: Hybrid choice model: draws and latent variable

Figure 34 shows the implementation of the hybrid model in the `apollo_probabilities` function for the example with an ordered measurement model, i.e. `Apollo_example_24.r`. We create a list `P` which will in the end have five components, namely the probabilities of the four measurement models and the probabilities from the choice model. We first compute the probabilities for the four ordered logit measurement models, one for each attitudinal statement, where these explain the values for the attitudinal indicators as a function of the latent variable, as detailed in Equation 55. For details on the syntax of `apollo_ol`, refer to 5.3.2. One point to note here is the inclusion of `rows=(task==1)` which ensures that the measurement model is only used once for each attitudinal statement and for each individual, rather than contributing to the overall model likelihood in each row for that person. This is in line with the rows in the data being for choice tasks, and the answers to attitudinal questions being repeated in the data in each row.

We next turn to the calculation of the probabilities for the choice model component of the hybrid model. The definition of `alternatives`, `avail` and `choiceVar` is as before. The core

```
apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Likelihood of indicators
  ol_settings1 = list(outcomeOrdered=attitude_quality,
                      V=zeta_quality*LV,
                      tau=c(tau_quality_1, tau_quality_2, tau_quality_3, tau_quality_4),
                      rows=(task==1))
...
  ol_settings4 = list(outcomeOrdered=attitude_dominance,
                      V=zeta_dominance*LV,
                      tau=c(tau_dominance_1, tau_dominance_2, tau_dominance_3, tau_dominance_4),
                      rows=(task==1))
  P[["indic_quality"]]      = apollo_ol(ol_settings1, functionality)
...
  P[["indic_dominance"]]    = apollo_ol(ol_settings4, functionality)

  ### Likelihood of choices
  ### List of utilities: these must use the same names as in mnl_settings, order is irrelevant
  V = list()
  V[['alt1']] = ( b_brand_Artemis*(brand_1=="Artemis") + b_brand_Novum*(brand_1=="Novum")
                + b_country_CH*(country_1=="Switzerland") + b_country_DK*(country_1=="Denmark") +
                    ↪ b_country_USA*(country_1=="USA")
                + b_char_standard*(char_1=="standard") + b_char_fast*(char_1=="fast acting") +
                    ↪ b_char_double*(char_1=="double strength")
                + b_risk*side_effects_1
                + b_price*price_1
                + lambda*LV )
  V[['alt2']] = ( b_brand_Artemis*(brand_2=="Artemis") + b_brand_Novum*(brand_2=="Novum")
                + b_country_CH*(country_2=="Switzerland") + b_country_DK*(country_2=="Denmark") +
                    ↪ b_country_USA*(country_2=="USA")
                + b_char_standard*(char_2=="standard") + b_char_fast*(char_2=="fast acting") +
                    ↪ b_char_double*(char_2=="double strength")
                + b_risk*side_effects_2
                + b_price*price_2
                + lambda*LV )
  V[['alt3']] = ( b_brand_BestValue*(brand_3=="BestValue") + b_brand_Supermarket*(brand_3=="Supermarket
      ↪ ") + b_brand_PainAway*(brand_3=="PainAway")
                + b_country_USA*(country_3=="USA") + b_country_IND*(country_3=="India") + b_country_RUS
                    ↪ *(country_3=="Russia") + b_country_BRA*(country_3=="Brazil")
                + b_char_standard*(char_3=="standard") + b_char_fast*(char_3=="fast acting")
                + b_risk*side_effects_3
                + b_price*price_3 )
...

  ### Define settings for MNL model component
  mnl_settings = list(
    alternatives = c(alt1=1, alt2=2, alt3=3, alt4=4),
    avail        = list(alt1=1, alt2=1, alt3=1, alt4=1),
    choiceVar    = best,
    V            = V
  )

  ### Compute probabilities for MNL model component
  P[["choice"]] = apollo_mnl(mnl_settings, functionality)

  ### Likelihood of the whole model
  P = apollo_combineModels(P, apollo_inputs, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Average across inter-individual draws
  P = apollo_avgInterDraws(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 34: Hybrid choice model with ordered measurement model: defining probabilities

```
### Load data
database <- read.csv("apollo_drugChoiceData.csv")

database$attitude_quality=database$attitude_quality-mean(database$attitude_quality)
database$attitude_ingredients=database$attitude_ingredients-mean(database$attitude_ingredients)
database$attitude_patent=database$attitude_patent-mean(database$attitude_patent)
database$attitude_dominance=database$attitude_dominance-mean(database$attitude_dominance)

...

apollo_probabilities=function(apollo_beta, apollo_inputs, functionality="estimate"){

...

  ### Likelihood of indicators
  normalDensity_settings1 = list(outcomeNormal=attitude_quality,
                                 xNormal=zeta_quality*LV,
                                 mu=0,
                                 sigma=sigma_qual,
                                 rows=(task==1))
  normalDensity_settings2 = list(outcomeNormal=attitude_ingredients,
                                 xNormal=zeta_ingredient*LV,
                                 mu=0,
                                 sigma=sigma_ingr,
                                 rows=(task==1))
  normalDensity_settings3 = list(outcomeNormal=attitude_patent,
                                 xNormal=zeta_patent*LV,
                                 mu=0,
                                 sigma=sigma_pate,
                                 rows=(task==1))
  normalDensity_settings4 = list(outcomeNormal=attitude_dominance,
                                 xNormal=zeta_dominance*LV,
                                 mu=0,
                                 sigma=sigma_domi,
                                 rows=(task==1))
  P[["indic_quality"]]     = apollo_normalDensity(normalDensity_settings1, functionality)
  P[["indic_ingredients"]] = apollo_normalDensity(normalDensity_settings2, functionality)
  P[["indic_patent"]]      = apollo_normalDensity(normalDensity_settings3, functionality)
  P[["indic_dominance"]]   = apollo_normalDensity(normalDensity_settings4, functionality)

...

}
```

Figure 35: Hybrid choice model with continuous measurement model: zero-centering indicators and defining probabilities

part of the utility functions is as in Figure 17, with the addition that the latent variable α_n is introduced into the utilities for the first two alternatives only, multiplied by a common parameter (λ), as shown in Equation 53.

The list `P` now contains five individual components, and the call to `apollo_combineModels` combines these into a joint model, prior to multiplying across choices for the same individual and averaging across draws, using the by now well known functions.

In `Apollo_example_25.r`, we use a continuous measurement model for the indicators. In line with Equation 56, we wish to avoid the estimation of the means for the latent variable. This is achieved by zero-centering the indicators, a process that needs to take place at the database level, prior to the call to `apollo_validateInputs` to ensure that these new variables are identical across cores in a multi-core setting. We show this part of the code in Figure 35, along with the part of `apollo_probabilities` which changes, which is only the treatment of the indicators, where we now use the function `apollo_normalDensity`, with details available in Section 5.3.3.

8 Bayesian estimation

Apollo allows the user to replace classical estimation by Bayesian estimation, for all models. We do not provide details here on Bayesian theory but instead refer the reader to Lenk (2014) and the references therein. Bayesian estimation in *Apollo* makes use of the `RSGHB` package, and the user is referred to the documentation in Dumont and Keller (2019) for `RSGHB`-specific settings.

The key advantage for the user is that *Apollo* provides a wrapper around `RSGHB` so that the syntax in `apollo_probabilities` does not change when a user moves from classical to Bayesian estimation. To explain the process, we now look at the estimation of a mixed logit version of the model from Section 4.5.2, which is included in `Apollo_example_26.r`. We use Normal distributions for the three ASCs, negative Lognormal distributions for the time and cost coefficients, censored Normal distributions (with negative values fixed to zero) for the wifi and food parameters, and fixed parameters for all other terms.

```
apollo_beta=c(asc_car                = 0,
              asc_bus                = 0,
              asc_air                = 0,
              asc_rail               = 0,
              asc_bus_shift_female   = 0,
              asc_air_shift_female   = 0,
              asc_rail_shift_female  = 0,
              b_tt_car               = -3,
              b_tt_bus               = -3,
              b_tt_air               = -3,
              b_tt_rail              = -3,
              b_tt_shift_business    = -3,
              b_acc                  = -3,
              b_cost                 = -3,
              b_cost_shift_business  = 0,
              cost_income_elast      = 0,
              b_no_frills            = 0,
              b_wifi                 = 0,
              b_food                 = 0)

apollo_fixed = c("asc_car","b_no_frills")

apollo_HB = list(
  hbDist          = c(asc_car                = "F",
                      asc_bus                = "N",
                      asc_air                = "N",
                      asc_rail               = "N",
                      asc_bus_shift_female   = "F",
                      asc_air_shift_female   = "F",
                      asc_rail_shift_female  = "F",
                      b_tt_car               = "LN-",
                      b_tt_bus               = "LN-",
                      b_tt_air               = "LN-",
                      b_tt_rail              = "LN-",
                      b_tt_shift_business    = "F",
                      b_acc                  = "LN-",
                      b_cost                 = "LN-",
                      b_cost_shift_business  = "F",
                      cost_income_elast      = "F",
                      b_no_frills            = "F",
                      b_wifi                 = "CN+",
                      b_food                 = "CN+"),
  gNCREP          = 100000,
  gNEREP          = 50000,
  gINFOSKIP       = 500)
```

Figure 36: Bayesian estimation in *Apollo*: model settings

The first steps in the model definition are shown in Figure 36, where we define the individual coefficients and their starting values in `apollo_beta` as before, where we use starting values of -3 for the underlying mean of the Lognormally distributed coefficients, i.e. the mean of the logarithm

of the coefficients. The definition of `apollo_fixed` is also as in the MNL model.

We next create a list called `apollo_HB` which can contain any of the settings used in `RSGHB` (Dumont and Keller, 2019), where we only use a small subset. One further difference arises. `RSGHB` requires the user to create an element called `gdist` with a numeric coding for distributions. *Apollo* instead requires the user to create a named character vector inside `apollo_HB` that is called `hbDist` and which contains one entry for each of the parameters in `apollo_beta`, setting the distribution to use, with the following definitions:

`"F"`: non-random (fixed) parameters[16];
`"N"`: normally distributed random parameters;
`"LN-"`: negative lognormally distributed random parameters;
`"LN+"`: positive lognormally distributed random parameters;
`"CN-"`: normally distributed random parameters, bounded above at 0;
`"CN+"`: normally distributed random parameters, bounded below at 0; and
`"JSB"`: Johnson SB distributed random parameters.

The entry `hbDist` is the only compulsory setting when using Bayesian estimation in *Apollo*. In our example, we also define three additional settings, namely:

`gNCREP`: number of burn-in iterations to use prior to convergence (default=100000);
`gNEREP`: number of iterations to keep for averaging after convergence has been reached (default=100000); and
`gINFOSKIP`: number of iterations between printing/plotting information about the iteration process (default=250)

The `apollo_probabilities` function is exactly the same as for the MNL model shown in Figure 7 and is thus not reproduced here. When using Bayesian estimation, the use of `apollo_avgInterDraws` and `apollo_avgIntraDraws` does not apply even in the presence of random coefficients. In addition, the call to `apollo_panelProd` is ignored as `RSGHB` automatically groups together observations for the same individual. The inclusion of any of these three commands however does no harm.

The call to `apollo_estimate` is made in exactly the same way as with classical estimation. The estimation process is illustrated in Figure 37 for the text output and Figure 38 for a graphical output of the chains. In the text output, we show the first and final iteration, where this also highlights the way in which `RSGHB` confirms the distributions used at the outset.

The post-estimation output from a model using Bayesian estimation is substantially different from that with classical estimation, and is summarised in Figure 39. The early information on model name etc is the same as with classical estimation. This is followed by average model fit statistics across the post burn-in iterations. Next, we have convergence reports for the parameter chains, where these use the Geweke test (Geweke, 1992). The next four parts of the output look at summaries of the parameter chains, each time giving the mean and standard deviation across the

[16] This is also the obvious choice for parameters that are to be kept fixed at their starting values. RSGHB also allows users to have random parameters where the mean and standard are fixed, please see Dumont and Keller (2019).

```
> model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)

Diagnostic checks passed. Please review before proceeding

...
                        Initial Log-Likelihood: -300144.175

      Fixed Parameters Start
  asc_bus_shift_female   0.0
  asc_air_shift_female   0.0
asc_rail_shift_female    0.0
   b_tt_shift_business  -0.1
b_cost_shift_business    0.0
    cost_income_elast    0.0

 Random Parameters Start Dist.
          asc_bus   0.0      N
          asc_air   0.0      N
         asc_rail   0.0      N
         b_tt_car  -3.0    LN-
         b_tt_bus  -3.0    LN-
         b_tt_air  -3.0    LN-
        b_tt_rail  -3.0    LN-
            b_acc  -3.0    LN-
           b_cost  -3.0    LN-
           b_wifi   0.0    CN+
           b_food   0.0    CN+

...

Iteration:      150000

...
           Log-Likelihood:      -4662.79949
                     RLH:      0.5336594011

      Fixed Parameters      Estimate
  asc_bus_shift_female:   0.350513792
  asc_air_shift_female:   0.179325718
asc_rail_shift_female:    0.186593885
   b_tt_shift_business:  -0.007608993
b_cost_shift_business:    0.029969039
    cost_income_elast:   -0.722956104

 Random Parameters    Estimate
          asc_bus:  -1.9398931
          asc_air:  -1.1445669
         asc_rail:  -2.2408746
         b_tt_car:  -4.5583159
         b_tt_bus:  -4.2736961
         b_tt_air:  -5.0724452
        b_tt_rail:  -6.8548800
            b_acc:  -4.2750151
           b_cost:  -2.6319158
           b_wifi:   0.8191816
           b_food:   0.1901212

Time per iteration: 0.0169 secs
Time to completion: 0 mins

Estimation complete.
```

Figure 37: Bayesian estimation in *Apollo*: estimation process

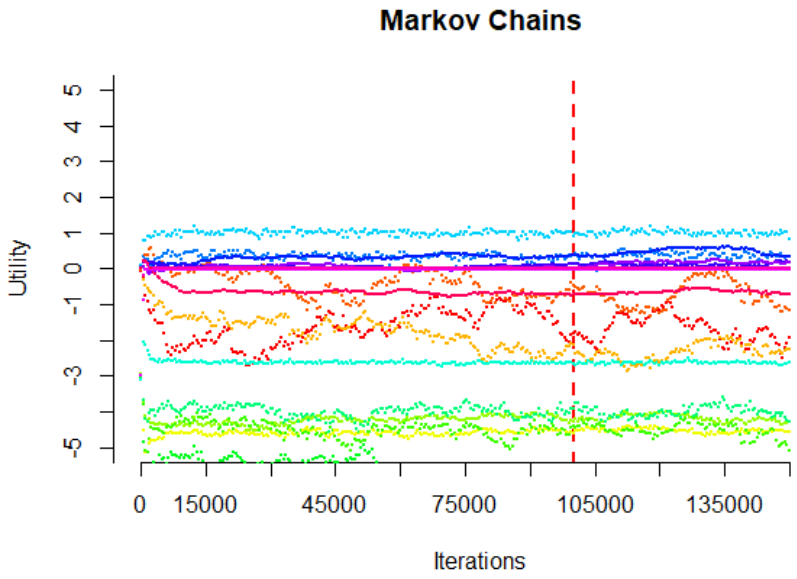

Figure 38: Bayesian estimation in *Apollo*: estimation process (parameter chains)

post burn-in iterations for each parameter, where these results are divided into the non-random coefficients, the means for the underlying Normals, and the covariance matrix (split across two tables, with the mean and standard deviations of each entry in the covariance matrix). Finally, the output reports the means and standard deviations for the posteriors, where these are for the actual coefficients, i.e. taking into account the distributions used, rather than looking at the underlying Normals. All the values used for these components are also available in the `model` object after estimation and can be used for plotting. The use of `apollo_saveOutput` operates as before, but if `saveEst==TRUE`, the code additionally saves the output files produced by `RSGHB`, which can be very large in size (cf. Dumont and Keller, 2019).

In classical estimation, *Apollo* creates an object `estimates` in the `model` list created after estimation, containing the final parameter values. When using Bayesian estimation, `model$estimates` is also produced, combining non-random parameters with individual specific posteriors for random parameters. This allows the user to use `apollo_prediction` and `apollo_llCalc` on such outputs, where care is of course required in interpretation of outputs based on posterior means.

```
> apollo_modelOutput(model)

Model run using Apollo for R, version 0.0.8
www.cmc.leeds.ac.uk

Model name                              : Apollo_example_26
Model description                       : HB model on mode choice SP data
Estimation method                       : Hierarchical Bayes

Average posterior log-likelihood post burn-in -4627.541
Average posterior RLH post burn-in 0.5363667

Chain convergence report:
Fixed (non random) parameters:
 asc_bus_shift_female  asc_air_shift_female asc_rail_shift_female
              -1.8659                 0.4337                -6.3352
...

Summary of parameters chains:
Non-random coefficients
                        Mean     SD
asc_car                0.0000     NA
asc_bus_shift_female   0.4762 0.0953
...

Upper level model results for mean parameters for underlying Normals
             Mean     SD
asc_bus   -1.6887 0.3612
asc_air   -0.7266 0.3707
...

Upper level model results for covariance matrix for underlying Normals (means across iterations)
          asc_bus asc_air asc_rail b_tt_car b_tt_bus b_tt_air b_tt_rail
asc_bus    0.1951 -0.0014   0.0130  -0.0058   0.0252  -0.0088  -0.0035
asc_air   -0.0014  0.1760   0.0580  -0.0144  -0.0064   0.0379  -0.0133
...

Upper level model results for covariance matrix for underlying Normals (SD across iterations)
          asc_bus asc_air asc_rail b_tt_car b_tt_bus b_tt_air b_tt_rail
asc_bus    0.0834  0.0537   0.0537   0.0205   0.0235   0.0475   0.0572
asc_air    0.0537  0.0958   0.0567   0.0206   0.0220   0.0540   0.0567
...

Results for posterior means for random coefficients
             Mean     SD
asc_bus   -1.6887 0.0273
asc_air   -0.7267 0.0459
...
```

Figure 39: Bayesian estimation in *Apollo*: output (extracts)

9 Pre and post-estimation capabilities

A large number of additional functions are provided in *Apollo* to allow the user to analyse the results after estimation. The outputs from these functions are not saved in the model output files, and it is then helpful for a user to dump the additional output to a text file, for example using `sink(paste(model$apollo_control$modelName,"_additional_output.txt",sep=""), split=TRUE)` which produces a new text file using the name of the current model. Outputs in the console are then also written into this text file, and writing to file can be stopped via `if(sink.number()>0) sink()`. We will now look at these various functions in turn.

9.1 Pre-estimation analysis of choices

With labelled choice data (or even unlabelled data where there may be strong left-right effects), it can be useful to analyse the choices before model estimation to determine whether the characteristics of individuals choosing specific alternatives differ across alternatives. This is made possible by the function called `apollo_choiceAnalysis`, which is called as follows:

```
apollo_choiceAnalysis(choiceAnalysis_settings,
                      apollo_control,
                      database)
```

where `choiceAnalysis_settings` has the following contents:

- `alternatives`: A named vector containing the names of the alternatives, as in e.g. an MNL model.
- `avail`: A list containing one element with availabilities per alternative, as in e.g. an MNL model, but where reference to database needs to be made given that we are operating outside `apollo_probabilities` (cf. Figure 40).
- `choiceVar`: A vector of length equal to the number of observations, containing the chosen alternative for each observation.
- `explanators`: A dataframe containing a set of variables, one per column and one entry per choice observation, that are to be used to analyse the choices. This could include explantory variables describing the alternatives but is most useful for characteristics of the decision makers. In order to be able to define this object outside `apollo_probabilities`, reference to the database again needs to be made(cf. Figure 40).
- `rows`: This is an optional argument which is missing by default. It allows the user to specify a vector called `rows` of the same length as the number of rows in the data. This vector needs to use logical statements to identify which rows in the data are to be used for the analysis of choices.

The function produces a *csv* file with one row per alternative, and three columns per variable included in `explanators`. In a given row, i.e. for a given alternative, these three columns contain the mean value for the given explanatory variable for those choices where the alternative is chosen,

```
choiceAnalysis_settings <- list(
  alternatives = c(car=1, bus=2, air=3, rail=4),
  avail        = list(car=database$av_car, bus=database$av_bus, air=database$av_air, rail=
      ↪ database$av_rail),
  choiceVar    = database$choice,
  explanators  = database[,c("female","business","income")]
)

apollo_choiceAnalysis(choiceAnalysis_settings, apollo_control, database)
```

	Mean for female if chosen	Mean for female if not chosen	t-test for difference
car	0.4699	0.5112	1.14
bus	0.4841	0.4704	-0.29
air	0.4744	0.473	-0.04
rail	0.4801	0.4808	0.02

Figure 40: Running `apollo_choiceAnalysis` (syntax and excerpt of output)

the mean value where it is not chosen (but available), and the test statistic for the two-sample t-test comparing the means in these two groups (where the null hypothesis states that the difference between the means is equal to 0, and the alternative hypothesis says that it is different from zero.). These value are also returned silently by the function, so they can be stored in a variable, by using e.g. `output=apollo_choiceAnalysis(choiceAnalysis_settings,apollo_control,database)`. An example application of this function is included in `Apollo_example_1.r`, where we show the results for the first explanator only.

9.2 Reading in a previously saved `model` object

As mentioned in Section 4.7, the call to `apollo_saveOutput` (with default settings) saves the `model` object in a *.rds* file. It is then possible to read this in as a new model object using the function `apollo_loadModel` which is called as:

`oldModel` = `apollo_loadModel`(`modelName`)

where `modelName` needs to be replaced by the name of previously run model (for which the output was saved in the current directory), where this name needs to be given as a string, i.e. with quotation marks. The output from this function is then a model object, which in this case is saved into `oldModel`. The benefit of this function is that it is then easy for a user to return to a previously estimated model and compute additional output with the estimates from that model and having access to the full covariance matrix without needing to reestimate the model. An example is included in `apollo_example_16.r`.

9.3 Calculating model fit for given parameter values

Especially with complex models, it can be useful for testing purposes to calculate the log-likelihood of the model (and subcomponents) for given parameter values, before or after estimation. This is

made possible by the function `apollo_llCalc`, which is called as follows:

```
apollo_llCalc(apollo_beta,
              apollo_probabilities,
              apollo_inputs)
```

where we illustrate this in Figure 41 for the case of the hybrid choice model (`Apollo_example_24.r`) from Section 7.3. It should be noted that when calling this function, the format of `apollo_beta` needs to be compatible with what is used inside `apollo_probabilities`. All parameters used inside the model need to be included, with either one value per parameter (classical estimation) or one value per parameter and per observation (Bayesian estimation).

```
> apollo_llCalc(apollo_beta, apollo_probabilities, apollo_inputs)
Updating inputs... Done.
Calculating LL of each model component...Done.
$model
[1] -19734.47

$indic_quality
[1] -1505.83

$indic_ingredients
[1] -1531.451

$indic_patent
[1] -1543.356

$indic_dominance
[1] -1465.061

$choice
[1] -13314.75
```

Figure 41: Running `apollo_llCalc`

9.4 Likelihood ratio tests against other models

A core step in many model fitting exercises is the comparison of models of different levels of complexity. When comparing two models where one model is a more general version of a base model, i.e. the *base* model is nested within the *general* model, a likelihood-ratio test can be used to compare the two models (cf. Train, 2009, Section 3.8.2.). This test is implemented in the function `apollo_lrTest`.

The function `apollo_lrTest` can either be called to compare two models for which the output has been saved in earlier runs of *Apollo* or to compare a model run in the current instance of `R` with a model for which the output has been saved. In Figure 42, we illustrate the function by comparing the MNL model from Section 4.5.2, i.e. `Apollo_example_3`, and the NL model from Section 5.1.1, i.e. `Apollo_example_5`. We show both the version where the outputs for both models have been saved earlier[17] as well as the version where the more general model has just been run and still exists in the `R` instance. This function is not suitable when for example comparing a joint model with two separate models (e.g. RP-SP vs separate RP and SP models) and the user in that case needs to calculate the LR test manually, which is of course trivial.

[17] Note that the output needs to have been saved in the same directory.

```
> apollo_lrTest("Apollo_example_3", "Apollo_example_5")

Likelihood ratio test-value:    120.94
Degrees of freedom:             2
Likelihood ratio test p-value: 0

> apollo_lrTest("Apollo_example_3", model)

Likelihood ratio test-value:    120.94
Degrees of freedom:             2
Likelihood ratio test p-value: 0
```

Figure 42: Running `apollo_lrTest`

9.5 Model predictions

A core capability of *Apollo* is that it covers model application (i.e. prediction) in addition to estimation. This is implemented in the function `apollo_prediction`. The function is called as follows:

```
forecast = apollo_prediction(model,
                             apollo_probabilities,
                             apollo_inputs,
                             modelComponent)
```

The majority of these arguments have been discussed already. The only additional new argument is `modelComponent`, where this is the name of the model component for which predictions are requested. This argument is required for models with multiple components, and needs to be the name (string) of one of the elements in the list `P` used inside `apollo_probabilities`.

The application of this function to the `Apollo_example_3.r` model is illustrated in Figure 43, which also shows how to look at changes in choices following a change in an explanatory variable, as well as how elasticities can be calculated. Model predictions in *Apollo* always use `database` as an input, whether applying the model to the base data or a forecast scenario. This means that for looking at the impact of changes to explanatory variables, these changes need to be made in `database`, and can then of course be reversed after applying `apollo_prediction`.

The output of `apollo_prediction` will depend on the underlying model component, but will always include the ID and choice situation index for each row. In particular:

- For **MNL**, **NL**, **CNL** and **OL** models, `apollo_prediction` will return the probability of the chosen alternative, as well as the probability of each alternative at the observation level (rather than person level). In particular, these models return a list containing one vector per alternative plus an additional vector for the chosen alternative, where each vector is as long as the number of observations in the `database`, contain the probability of that alternative. In the presence of continuous random heterogeneity, the draws are averaged out before presenting the results.
- The discrete continuous models `MDCEV` and `MDCNEV` do not return probabilities, but instead expected values of consumption for each alternative at the observation level. In particular,

```
> predictions_base = apollo_prediction(model, apollo_probabilities, apollo_inputs)
Running predictions from model
> summary(predictions_base[,3:ncol(predictions_base)])
      car                bus                air                rail             chosen
 Min.   :0.00000   Min.   :0.000000   Min.   :0.0000000   Min.   :0.0000   Min.   :0.005085
 1st Qu.:0.02813   1st Qu.:0.003839   1st Qu.:0.0001073   1st Qu.:0.1785   1st Qu.:0.382050
 Median :0.19006   Median :0.015744   Median :0.0803034   Median :0.4434   Median :0.628007
 Mean   :0.27800   Mean   :0.051144   Mean   :0.2174293   Mean   :0.4534   Mean   :0.592764
 3rd Qu.:0.46307   3rd Qu.:0.044569   3rd Qu.:0.3685864   3rd Qu.:0.7268   3rd Qu.:0.815799
 Max.   :0.99658   Max.   :0.999608   Max.   :0.9992502   Max.   :0.9994   Max.   :0.999608

### Now imagine the cost for rail increases by 10%
> database$cost_rail=1.1*database$cost_rail
> predictions_new = model, apollo_probabilities, apollo_inputs)

### Return to original data
> database$cost_rail=1/1.1*database$cost_rail

### Compute change in probabilities
> change=(predictions_new-predictions_base)/predictions_base
### Not interested in chosen alternative now, so drop last column
> change=change[,-ncol(change)]
### First two columns (change in ID and task) also meaningless
> change=change[,-c(1,2)]
>### Look at person 9, who has all 4 modes available
> change[database$ID==9,]
              car         bus         air         rail
 [1,] 0.09920605 0.09920605 0.09920605 -0.17890617
 [2,] 0.12886030 0.12886030 0.12886030 -0.05239268
...
[14,] 0.16101011 0.16101011 0.16101011 -0.06258327
>### Look at mean changes for subsets of the data, ignoring NAs
> colMeans(change,na.rm=TRUE)
       car        bus        air        rail
 0.1391721  0.1568854  0.1469683 -0.1894928
> colMeans(subset(change,database$business==1),na.rm=TRUE)
       car        bus        air        rail
 0.1286203  0.1332535  0.1084991 -0.1065921
> colMeans(subset(change,database$business==0),na.rm=TRUE)
       car        bus        air        rail
 0.1444075  0.1688990  0.1641315 -0.2302359
> colMeans(subset(change,(database$income<quantile(database$income,0.25))),na.rm=TRUE)
       car        bus        air        rail
 0.1775089  0.1978315  0.1936918 -0.2787185
> colMeans(subset(change,(database$income>=quantile(database$income,0.25))|(database$income<=quantile(database$income,0.75))), na.rm=TRUE)
       car        bus        air        rail
 0.1391721  0.1568854  0.1469683 -0.1894928
> colMeans(subset(change,(database$income>quantile(database$income,0.75))),na.rm=TRUE)
       car        bus        air        rail
 0.1079734  0.1193161  0.1074423 -0.1364650

>### Own elasticity for rail:
> log(sum(predictions_new[,4])/sum(predictions_base[,4]))/log(1.1)
[1] -1.340772

>### Cross-elasticities for other modes
> log(sum(predictions_new[,1])/sum(predictions_base[,1]))/log(1.1)
[1] 1.049468
> log(sum(predictions_new[,2])/sum(predictions_base[,2]))/log(1.1)
[1] 1.343074
> log(sum(predictions_new[,3])/sum(predictions_base[,3]))/log(1.1)
[1] 0.8418118
```

Figure 43: Running `apollo_prediction`

they return a matrix detailing the expected (continuous) consumption for each alternative, and a proxy for the probability of consuming each alternative (discrete choice), as well as the standard deviations for both of these measurements. These outputs are calculated using the efficient forecasting method proposed by Pinjari and Bhat 2010b, and its modification for the **MDCNEV** model by Calastri et al. 2017. These methods are based on simulation (200 repetitions are used), and can therefore be computationally demanding. The probability of

consuming each alternative is calculated as the percentage of simulation repetitions in which the alternative is consumed, and is not calculated using an analytical formula. Again, in the presence of continuous random coefficients, the results are averaged across draws.

- The **EL** (exploded logit) and **Normal Density** models do not return any prediction, as it is not evident what precise outcome would be the most useful for the biggest share of users.

9.6 Market share recovery for subgroups of data

With labelled choice data (or even unlabelled data where there may be strong left-right effects), it can be useful to test after model estimation how well the choice shares in the data are recovered by the model. With a full set of ASCs, a linear in attributes MNL model will perfectly recover market shares at the sample level (see e.g. Train, 2009, Section 2.6.1.). This is however likely not the case in subsets of the data, and this test can thus be a useful input for model refinements. The function `apollo_sharesTest` is based on the "apply" tables approach in ALogit (ALogit, 2016), and is called as follows:

```
apollo_sharesTest(model,
                  apollo_probabilities,
                  apollo_inputs,
                  sharesTest_settings)
```

The list `sharesTest_settings` has the following components:

`alternatives`: A named vector containing the names of the alternatives as defined by the user, and for each alternative, giving the value used in the dependent variable in the data. In our case, these simply go from 1 to 4.

`choiceVar`: A variable indicating the column in the database which identifies the alternative chosen in a given choice situation. In our example, this column is simply called `choice`. This is not a character variable (i.e. text) but the name use to identify a column in the database. As we are now operating outside `apollo_probabilities`, we need to use `database$choice` for example.

`subsamples`: The list `subsamples` is an optional input which contains one column for each subset of the data to be used in the test, where it is possible for a given row to be included in multiple subsets, i.e. the values across column vectors in `subsamples` may exceed 1.

`modelComponent`: The name of the model component for which predictions are requested. This argument is required for models with multiple components, and needs to be the name (string) of one of the elements in the list `P` used inside `apollo_probabilities`.

The function produces one table per column in `subsamples`, along with an overall table for the entire sample. In each table, the code reports the number of times an alternative is chosen in the data, the number of times the model predicts it to be chosen, the difference between prediction and data, and a t-ratio and p-value for this difference. An example application of this function is included in `Apollo_example_3.r`. As we can see from Figure 44, in our example, the model

```
> sharesTest_settings = list()
> sharesTest_settings=list()
> sharesTest_settings[["alternatives"]] = c(car=1, bus=2, air=3, rail=4)
> sharesTest_settings[["choiceVar"]] = database$choice
> sharesTest_settings[["subsamples"]] = list(business=(database$business==1),+ leisure=(database$business==0))

> apollo_sharesTest(model,apollo_probabilities,apollo_inputs,sharesTest_settings)
Updating inputs...Done.
Running predictions from model... Done.

Running share prediction tests


Prediction tests for group: business (2310 observations)

                              car     bus      air      rail   All
Times chosen (data)       366.000   8.000 771.000 1165.000 2310
Times chosen (prediction) 350.443  24.348 739.725 1195.484 2310
Diff (prediction-data)    -15.557  16.348 -31.275   30.484    0
t-ratio                    -1.093   3.463  -1.846    1.612   NA
p-value                     0.275   0.001   0.065    0.107   NA

Prediction tests for group: leisure (4690 observations)

                              car      bus      air      rail   All
Times chosen (data)       1580.000 350.000 751.000 2009.000 4690
Times chosen (prediction) 1595.563 333.656 782.280 1978.501 4690
Diff (prediction-data)      15.563 -16.344  31.280  -30.499    0
t-ratio                      0.606  -1.075   1.601   -1.160   NA
p-value                      0.544   0.282   0.109    0.246   NA

Prediction tests for group: All data (7000 observations)

                              car      bus       air      rail   All
Times chosen (data)       1946.000 358.000 1522.000 3174.000 7000
Times chosen (prediction) 1946.006 358.004 1522.005 3173.984 7000
Diff (prediction-data)       0.006   0.004    0.005   -0.016    0
t-ratio                      0.000   0.000    0.000    0.000   NA
p-value                      1.000   1.000    1.000    1.000   NA
```

Figure 44: Running `apollo_sharesTest`

significantly overpredicts the rate at which business travellers choose bus and underpredicts the rate at which they choose air. A revised model specification may thus incorporate shifts in these ASCs for business travellers.

9.7 Comparison of model fit across subgroups of data

An additional function is implemented to compare the performance of the estimated model to predict the chosen alternative for different subsets of the data. The function `apollo_fitsTest` is called as follows:

```
apollo_fitsTest(model,
                apollo_probabilities,
                apollo_inputs,
                fitsTest_settings)
```

The list `fitsTest_settings` has the following components:

`subsamples`: The list `subsamples` is an optional input which contains one column for each subset of the data to be used in the test, where it is possible for a given row to be included in

multiple subsets, i.e. the values across column vectors in `subsamples` may exceed 1.

`modelComponent:` The name of the model component for which predictions are requested. This argument is required for models with multiple components, and needs to be the name (string) of one of the elements in the list `P` used inside `apollo_probabilities`.

The function calculates various statistics for the probability for the chosen alternative, as illustrate in Figure 45 for `Apollo_example_3.r`, where the last row in the output compares the mean predicted probability for the chosen alternative in the specific subsample compared to the mean in all other subsamples.

```
> fitsTest_settings = list()

> fitsTest_settings[["subsamples"]] = list()
> fitsTest_settings$subsamples[["business"]] = database$business==1
> fitsTest_settings$subsamples[["leisure"]] = database$business==0
> apollo_fitsTest(model,apollo_probabilities,apollo_inputs,fitsTest_settings)
Updating inputs...Done.
Running predictions from model... Done.
                          All data business leisure
Min P(chosen)                 0.01     0.01    0.01
Mean P(chosen)                0.59     0.64    0.57
Median P(chosen)              0.63     0.67    0.61
Max P(chosen)                 1.00     1.00    1.00
SD P(chosen)                  0.27     0.26    0.27
mean vs mean of all other       NA     0.06   -0.06
```

Figure 45: Running `apollo_fitsTest`

Users need to exercise caution when using this function in the case where the choice set size varies across individuals in a manner that is correlated with the subgroups as the prediction performance for individuals with smaller choice sets will be likely to be larger, all else being equal.

9.8 Functions of model parameters and associated standard errors

A key use of estimates from choice models is the calculation of functions of these estimates, for example in the form of ratios of coefficients, leading to marginal rates of substitution, and in the case of a cost coefficient being used as the denominator, willingness-to-pay (WTP) measures. It is then important to be able to calculate standard errors for these derived measures, where this can be done straightforwardly and accurately with the Delta method, as discussed by Daly et al. (2012a). The function `apollo_deltaMethod` is implemented for this purpose for a limited number of operations, and is called as follows:

```
apollo_deltaMethod(model,
                   deltaMethod_settings)
```

The list `deltaMethod_settings` has the following components:

`operation:` A character object `operation`, which determines which function is to be applied to the parameters. Possible values are:

`sum:` two-parameter function, with $f(\beta_1, \beta_2) = \beta_1 + \beta_2$
`diff:` two-parameter function, with $f(\beta_1, \beta_2) = \beta_1 - \beta_2$
`ratio:` two-parameter function, with $f(\beta_1, \beta_2) = \frac{\beta_1}{\beta_2}$

`exp`: one-parameter function, with $f(\beta_1) = e^{\beta_1}$

`logistic`: either one-parameter function, with $f_1(\beta_1) = \frac{e^{\beta_1}}{e^{\beta_1}+1}$ and $f_2(\beta_1) = \frac{1}{e^{\beta_1}+1}$, or two-parameter function, with $f_1(\beta_1, \beta_2) = \frac{e^{\beta_1}}{e^{\beta_1}+e^{\beta_2}+1}$, $f_2(\beta_1, \beta_2) = \frac{e^{\beta_2}}{e^{\beta_1}+e^{\beta_2}+1}$, and $f_3(\beta_1, \beta_2) = \frac{1}{e^{\beta_1}+e^{\beta_2}+1}$.

`lognormal`: two-parameter function giving the mean and standard deviation for a Lognormal distribution on the basis of the mean and standard deviation for the logarithm of the coefficient, i.e. with $\beta = e^{N(\beta_1,\beta_2)}$, we have $f_1(\beta_1, \beta_2) = \mu_\beta = e^{\beta_1 + \frac{\beta_2^2}{2}}$ and $f_2(\beta_1, \beta_2) = \sigma_\beta = \mu_\beta\sqrt{e^{\beta_2^2} - 1}$

`parName1`: A character object giving the name of the first parameter.

`parName2`: A character object giving the name of the second parameter, optional if `operation=logistic`.

`multPar1`: An optional numerical value used to multiply the first parameter, set to 1 if omitted.

`multPar2`: An optional numerical value used to multiply the second parameter, set to 1 if omitted.

An example application of this function is included in `Apollo_example_3.r`, and is illustrate in Figure 46 for the car value of travel time, i.e. the ratio between the car travel time and cost coefficients (in both minutes and hours) as well as for the difference between the car and rail travel time coefficients. The values here are all calculated for an individual in the base socio-demographic group.

```
> deltaMethod_settings=list(operation="ratio",parName1="b_tt_car",parName2="b_cost")
> apollo_deltaMethod(model, deltaMethod_settings)

Running Delta method computations
                              Value Robust s.e. Rob t-ratio (0)
Ratio of b_tt_car and b_cost:  0.172      0.0097           17.72

> deltaMethod_settings=list(operation="ratio",parName1="b_tt_car",parName2="b_cost",multPar1 = 60)
> apollo_deltaMethod(model, deltaMethod_settings)

Running Delta method computations
                                                    Value Robust s.e. Rob t-ratio (0)
Ratio of b_tt_car (multiplied by 60) and b_cost:  10.3222      0.5826           17.72


> deltaMethod_settings=list(operation="diff",parName1="b_tt_car",parName2="b_tt_rail")
> apollo_deltaMethod(model, deltaMethod_settings)

Running Delta method computations
                                          Value Robust s.e. Rob t-ratio (0)
Difference between b_tt_car and b_tt_rail: -0.0061      0.0019          -3.18
```

Figure 46: Running `apollo_deltaMethod`

Only a limited number of functions of parameters are covered by `apollo_deltaMethod`. Rather than relying on sampling based approaches such as the Krinsky & Robb method (Krinsky and Robb, 1986) for calculating the standard errors for more complex functions, users who wish to compute standard errors of other functions can for example use the `R` function `deltamethod` from the `alr3` package (Weisberg, 2005). This uses symbolic differentiation of the user provided function.

9.9 Unconditionals for random parameters

After model estimation, it may be useful to an analyst to have at their disposal the actual values used for random coefficients, especially if these included interactions with socio-demographics or (non-linear) transforms that may lead to a requirement for simulation to calculate moments (as in the semi-non-parametric approach of Fosgerau and Mabit 2013 used in Section 6.1.2). We look separately at continuous random parameters and latent class.

9.9.1 Continuous random heterogeneity

For continuous random coefficients, the function `apollo_unconditionals` is called as follows:

```
unconditionals = apollo_unconditionals(model,
                                       apollo_probabilities,
                                       apollo_inputs)
```

The function produces a list as output, with one element per random coefficient, where this is a matrix for inter-individual draws, and a cube with inter and intra-individual draws. The outputs from this function can then readily be used for summary statistics or to produce plots. An example of this is included in `apollo_example_16.r`, and also illustrated in the discussion of conditionals in Section 9.10.1.

9.9.2 Latent class

For latent class models, the function `apollo_lcUnconditionals` is called as follows:

```
unconditionals = apollo_lcUnconditionals(model,
                                         apollo_probabilities,
                                         apollo_inputs)
```

The `apollo_lcUnconditionals` produces a list which has one element for each model parameter that varies across classes, where these are given by lists, with one element per class. The entry for each class could be either a scalar (if fixed coefficients are used inside the classes), a vector (if interactions with socio-demographics are used), or a matrix or cube if continuous heterogeneity is also incorporated. The final component in the list produced by `apollo_lcUnconditionals` is a list containing the class allocation probabilities, with one element per class, where these could again be scalars, vectors, matrices or cubes, depending on the extent of heterogeneity allowed for by the user. An example of this is included in `apollo_example_20.r`, and also illustrated in the discussion of conditionals in Section 9.10.2.

9.10 Conditionals for random coefficients

There is extensive interest by choice modellers in posterior model parameter distributions, as discussed in Train (2009, chapter 11) for continuous mixture models and Hess (2014) for latent class. We implement functions for this for both continuous mixed logit and latent class models.

9.10.1 Continuous random coefficients

Let β give a vector of taste coefficients that are jointly distributed according to $f(\beta \mid \Omega)$, where Ω is a vector of distributional parameters that is to be estimated from the data. Let Y_n give the sequence of observed choices for respondent n (which could be a single choice), and let $L(Y_n \mid \beta)$ give the probability of observing this sequence of choices with a specific value for the vector β. Then it can be seen that the probability of observing the specific value of β given the choices of respondent n is equal to:

$$L(\beta \mid Y_n) = \frac{L(Y_n \mid \beta) f(\beta \mid \Omega)}{\int_\beta L(Y_n \mid \beta) f(\beta \mid \Omega) \, \mathrm{d}\beta} \tag{58}$$

The integral in the denominator of Equation 58 does not have a closed form solution, such that its value needs to be approximated by simulation. This is a simple (albeit numerically expensive) process, with as an example the mean for the conditional distribution for respondent n being given by:

$$\widehat{\beta_n} = \frac{\sum_{r=1}^{R} [L(Y_n \mid \beta_r) \beta_r]}{\sum_{r=1}^{R} L(Y_n \mid \beta_r)}, \tag{59}$$

where β_r with $r = 1, \ldots, R$ are independent multi-dimensional draws[18] with equal weight from $f(\beta \mid \Omega)$ at the estimated values for Ω. Here, $\widehat{\beta_n}$ gives the most likely value for the various marginal utility coefficients, conditional on the choices observed for respondent n.

It is important to stress that the conditional estimates for each respondent follow themselves a random distribution, and that the output from Equation 59 simply gives the expected value of this distribution. As such, a distribution of the output from Equation 59 across respondents should not be seen as a conditional distribution of a taste coefficient across respondents, but rather a distribution of the means of the conditional distributions (or conditional means) across respondents. Here, it is similarly possible to produce a measure of the conditional standard deviation, given by:

$$\widetilde{\beta_n} = \sqrt{\frac{\sum_{r=1}^{R} \left[L(Y_n \mid \beta_r) \left(\beta_r - \widehat{\beta_n} \right)^2 \right]}{\sum_{r=1}^{R} L(Y_n \mid \beta_r)}}, \tag{60}$$

with $\widehat{\beta_n}$ taken from Equation 59.

The calculation of posteriors for models with continuous random heterogeneity is implemented in the function `apollo_conditionals`, which is called as follows:

```
conditionals = apollo_conditionals(model,
                                   apollo_probabilities,
                                   apollo_inputs)
```

[18] The term *independent* relates to independence across different multivariate draws, where the individual multivariate draws allow for correlation between univariate draws.

The function produces a list object with one component per continuous random coefficient (element defined in `apollo_randCoeff`). Each of these components is a matrix with one row per individual, containing the ID for that individual, the mean of the posterior distribution for that individual for the coefficient in question, and the standard deviation. As `apollo_conditionals` uses the contents of `apollo_randCoeff`, any socio-demographic interactions included in `apollo_randCoeff` will also be included in the calculation for the conditionals, where, if these vary across observations for the same individual, they will be averaged across observations. Similarly, any intra-individual random heterogeneity will also be averaged out.

```
> unconditionals = apollo_unconditionals(model, apollo_probabilities, apollo_inputs)

> conditionals <- apollo_conditionals(model, apollo_probabilities, apollo_inputs)
Your model contains intra-individual draws which will be averaged over for conditionalsFor information: this
     function is meant for use only with continuous mixture models, i.e. no latent class.

> mean(unconditionals[["v_tt"]])
[1] 0.4190822
> sd(unconditionals[["v_tt"]])
[1] 0.4811322

> summary(conditionals[["v_tt"]])
       ID           post. mean        post. sd
 Min.   : 2439   Min.   :0.1189   Min.   :0.0450
 1st Qu.:15308   1st Qu.:0.2674   1st Qu.:0.1295
 Median :18533   Median :0.3347   Median :0.1612
 Mean   :22181   Mean   :0.4166   Mean   :0.2003
 3rd Qu.:21948   3rd Qu.:0.4647   3rd Qu.:0.2238
 Max.   :84525   Max.   :2.2283   Max.   :1.4226

> income_n=apollo_firstRow(database$hh_inc_abs, apollo_inputs)

Call:
lm(formula = conditionals[["v_tt"]][, 2] ~ income_n)

Residuals:
     Min       1Q   Median       3Q      Max
-0.31612 -0.14816 -0.07318  0.05123  1.78066

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 3.506e-01  2.696e-02   13.01  < 2e-16 ***
income_n    8.628e-07  3.048e-07    2.83  0.00489 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2664 on 386 degrees of freedom
Multiple R-squared:  0.02033,    Adjusted R-squared:  0.01779
F-statistic: 8.011 on 1 and 386 DF,  p-value: 0.004892
```

Figure 47: Running `apollo_unconditionals` and `apollo_conditionals`

Figure 47 illustrates the use of this function for the value of travel time coefficient in the `Apollo_example_16.r` example, where we show how the conditional means can then for example also be used in regression analysis against characteristics of the individual, as discussed by Train (2009, chapter 11), in our case showing a significant impact of income on the conditionals for the VTT[19]. We also include a comparison with the unconditionals.

9.10.2 Latent class

It is similarly possible to calculate a number of posterior measures from latent class models. A key example comes in the form of posterior class allocation probabilities, where the posterior

[19] Note that as `apollo_conditionals` produces one value per individual, we also need to reduce the dimensionality of the income variable to one per individual, using `apollo_firstRow`.

probability of individual n for class s is given by:

$$\widehat{\pi_{ns}} = \frac{\pi_{ns} L_n(\beta_s)}{L_n(\beta, \pi_n)}, \tag{61}$$

where $L_n(\beta_s)$ gives the likelihood of the observed choices for individual n, conditional on class s.

To explain the benefit of these posterior class allocation probabilities, let us assume that we have calculated for each class in the model a given measure $w_s = \frac{\beta_{s1}}{\beta_{s2}}$, i.e. the ratio between the first two coefficients. Using $\overline{w_n} = \sum_{s=1}^{S} \pi_{ns} w_s$ simply gives us a sample level mean for the measure w for an individual with the specific observed characteristics of person n. These characteristics (in terms of socio-demographics used in the class allocation probabilities) will however be common to a number of individuals who still make different choices, and the most likely value for w for individual n, conditional on his/her observed choices, can now be calculated as $\widehat{w_n} = \sum_{s=1}^{S} \widehat{\pi_{ns}} w_s$.

Finally, it might also be useful to produce a profile of the membership in each class. From the parameters in the class allocation probabilities, we know which class is more or less likely to capture individuals who posses a specific characteristic, but this is not taking into account the multivariate nature of these characteristics. Let us for example assume that a given socio-demographic characteristic z_c is used in the class allocation probabilities, with associated parameter γ_c, and using a linear parameterisation in Equation 49. We can then calculate the likely value for z_c for an individual in class s as:

$$\widehat{z_{cs}} = \frac{\sum_{n=1}^{N} \widehat{\pi_{ns}} z_{cn}}{\sum_{n=1}^{N} \widehat{\pi_{ns}}}, \tag{62}$$

where we again use the posterior probabilities to take into account the observed choices. Alternatively, we can also calculate the probability of an individual in class s having a given value κ for z_c by using:

$$P(\widehat{z_{cs}} = \kappa) = \frac{\sum_{n=1}^{N} \widehat{\pi_{ns}} (z_{cn} = \kappa)}{\sum_{n=1}^{N} \widehat{\pi_{ns}}}. \tag{63}$$

The calculation of posteriors for latent class models is implemented in the function `apollo_lcConditionals`, which is called as follows:

```
conditionals = apollo_lcCconditionals(model,
                                      apollo_probabilities,
                                      apollo_inputs)
```

This function is only applicable for latent class models that do not incorporate additional continuous random heterogeneity.

The function produces a list object with one component per continuous random coefficient (element defined in `apollo_randCoeff`). Each of these components is a matrix with one row per individual, containing the ID for that individual, the mean of the posterior distribution for that individual for the coefficient in question, and the standard deviation.

Figure 48 illustrates the use of this for the `Apollo_example_20.r` example. We first produce the output from the `apollo_unconditionals_lc` function to compare to the conditionals later on, and also calculate the value of travel time (VTT) in each class, e.g. $VTT_a = \frac{\beta_{t,a}}{\beta_{c,a}}$, where $\beta_{t,a}$ and $\beta_{c,a}$ are the time and cost coefficients, respectively, in class a. We then calculate the unconditional VTT obtained by taking the weighted average across classes, where this varies across individuals as the class allocation probabilities do, i.e. $VTT_n = \pi_{n,a}VTT_a + \pi_{n,b}VTT_b$. We next calculate the conditional class allocation probabilities using `apollo_lcConditionals`. As can be seen from the output, the means of the conditionals is identical to the mean of the unconditionals, but the range is much wider. Similarly, when we calculate the conditional VTT, we see a wider range for that too.

We finally use the conditional class allocation probabilities to calculate some posterior statistics for class membership. To do this, we first retain only one value for the two socio-demographic variables `commute` and `car_availability` for each individual (by using `apollo_firstRow` before using the formula in Eq. 62 to calculate the most likely value for these two variables for individuals in the two classes, given the posterior class allocation probabilities. These posteriors class allocation probabilities can of course then also be used in regression.

```
> unconditionals=apollo_lcUnconditionals(model, apollo_probabilities, apollo_inputs)

> vtt_class_a=unconditionals[["beta_tt"]][[1]]/unconditionals[["beta_tc"]][[1]]
> vtt_class_b=unconditionals[["beta_tt"]][[2]]/unconditionals[["beta_tc"]][[2]]
> vtt_unconditional=unconditionals[["pi_values"]][[1]]*vtt_class_a+unconditionals[["pi_values"]][[2]]*vtt_class_b

> conditionals=apollo_lcConditionals(model, apollo_probabilities, apollo_inputs)
> summary(conditionals)
     Class_1           Class_2
 Min.   :0.000003   Min.   :0.0000
 1st Qu.:0.151559   1st Qu.:0.1209
 Median :0.381015   Median :0.6190
 Mean   :0.483881   Mean   :0.5161
 3rd Qu.:0.879099   3rd Qu.:0.8484
 Max.   :1.000000   Max.   :1.0000
> summary(as.data.frame(unconditionals[["pi_values"]]))
     class_a           class_b
 Min.   :0.3924   Min.   :0.4140
 1st Qu.:0.4467   1st Qu.:0.4140
 Median :0.4467   Median :0.5533
 Mean   :0.4839   Mean   :0.5161
 3rd Qu.:0.5860   3rd Qu.:0.5533
 Max.   :0.5860   Max.   :0.6076

> vtt_conditional=conditionals[,1]*vtt_class_a+conditionals[,2]*vtt_class_b

> summary(vtt_unconditional)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.4221  0.4544  0.4544  0.4766  0.5374  0.5374
> summary(vtt_conditional)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.1885  0.2787  0.4153  0.4766  0.7118  0.7838

> commute_n = apollo_firstRow(database$commute,apollo_inputs)
> car_availability_n = apollo_firstRow(database$car_availability,apollo_inputs)

> post_commute=colSums(commute_n*conditionals)/colSums(conditionals)
> post_car_availability=colSums(car_availability_n*conditionals)/colSums(conditionals)

> post_commute
  Class_1   Class_2
0.2629875 0.3077349
> post_car_availability
  Class_1   Class_2
0.4465377 0.3154211
```

Figure 48: Running `apollo_lcUnconditionals` and `apollo_lcConditionals`

9.11 Summary of results for multiple models

It is often useful to produce an output file combining the estimates from multiple models run on the same data. This is facilitated by the function `apollo_combineResults`. This function allows the user to combine the results from a number of models (which can be larger than 2) for which the outputs have all been saved in the same directory. The function is called as follows:

```
apollo_combineResults(combineResults_settings)
```

where the list `combineResults_settings` has the following contents:

`modelNames`: a vector of model names, e.g. `c("Apollo_example_1", "Apollo_example_2", "Apollo_example_3")`. If this argument is not given, all models within the directory are used.
`printClassical`: if set to TRUE, the code will save classical standard errors as well as robust standard errors, computed using the sandwich estimator (cf. Huber, 1967). This setting then also affects the reporting of t-ratios and p-values (default is FALSE).
`printPVal`: if set to TRUE, p-values are saved (default is FALSE).
`printT1`: if set to TRUE, t-ratios against 1 are saved in addition to t-ratios against 0 (default is FALSE).
`estimateDigits`: number of digits used for estimates (default set to 4).
`tDigits`: number of digits used for t-ratios (default set to 2).
`pDigits`: number of digits used for p-values (default set to 2).

The function produces a *csv* file with the name `model_comparison_time` where `time` is a numerical value defined by the current date and time. The file contains for each model the name, the number of individuals and observations, the number of estimated parameters, as well as four model fit statistics, namely the final log-likelihood, the adjusted ρ^2 measure, the AIC and the BIC. Note that not all these measures will be reported for all models, e.g. ρ^2 is not calculated for models with continuous components. The actual model outputs are then included in a number of columns where this depends on the level of detail requested by the user as described above (e.g. including classical t-ratios).

The function can be called as `apollo_combineResults()`, i.e. without any arguments. In that case, the default settings are used for all arguments, and all model files within the directory are combined into the output. An example of how to call this function is included in `apollo_example_6.r`, combining the MNL, NL and CNL results.

10 Extensions

10.1 Iterative coding of utilities for large choice sets

In the examples shown in this manual, the user codes the utilities of all alternatives one by one. With very large choice sets, this may not be practical, and a user may create the utilities recursively, for example. We illustrate this in Figure 49 for a simple example, where we have 100 alternatives, and where the utility includes two attributes (`x1` and `x2`). Values for these exist in the data for each alternative, with for example `x1_1` being the value for the first attribute for the first alternative, and there is also a vector of availabilities for each alternative, e.g. `av1` for the first alternative.

```
J = 100
V = list()
for(j in 1:J) V[[paste0("alt",j)]] = b1*database[,paste0("x1_",j)] + b2*database[,paste0("x2_",j)]

mnl_settings = list(
  alternatives  = setNames(1:J, names(V)),
  avail         = setNames(database[,paste0("av",1:J)], names(V)),
  choiceVar     = choice,
  V             = V
)
```

Figure 49: Defining utilities for large choice sets

10.2 Starting value search

In classical estimation, convergence to the global maximum of the likelihood function is not guaranteed by any optimization algorithm. While this is not a problem for simple linear in attributes MNL models due to their concave likelihood function, it might be for other more complex models, such as mixed logit or latent class models. A popular approach to reduce the probability of reaching a poor local maximum is starting the optimization process from several different candidate points (i.e. sets of parameters), and keep the solution with the highest likelihood. However, this approach is very computationally intensive. To reduce its cost, algorithms have been proposed to dynamically eliminate unpromising candidates.

The function `apollo_searchStart` implements a simplified version of the algorithm proposed by Bierlaire et al. (2010), which is called as follows:

```
apollo_beta = apollo_searchStart(apollo_beta,
                                 apollo_fixed,
                                 apollo_probabilities,
                                 apollo_inputs,
                                 searchStart_settings)
```

The function returns an updated list of starting values. The list `searchStart_settings` has the following contents:

`apolloBetaMin`: a vector of the minimum possible value for each parameter (default is `apollo_beta-0.5`).

`apolloBetaMax`: a vector of the maximum possible value for each parameter (default is `apollo_beta+0.5`).
`nCandidates`: the number of initial candidates (default is 100).
`maxStages`: the maximum number of iterations of the algorithm, i.e. maximum number of times the algorithm jumps from step 6 to 3 described below (default is 10).
`bfgsIter`: the maximum number of BFGS iterations to apply to each candidate in each iteration of the main algorithm (default is 10).
`smartStart`: if `TRUE`, the Hessian of `apollo_probabilities` is calculated at `apollo_beta`, and the initial candidates are drawn with a higher probability from the area where the Hessian indicates an improvement in the likelihood. This adds a significant amount of time to the initialization of the algorithm (default is `FALSE`).
`dTest`: the tolerance of test 4.2 described below (default is 1).
`gTest`: the tolerance for the gradient in test 4.3 described below (default is 10^{-3}).
`llTest`: the tolerance for the LL in test 4.3 described below (default is 3).

The main difference in our implementation lies in the fact that `apollo_searchStart` uses only two out of three tests on the candidates described by Bierlaire et al. (2010). The implemented algorithm has the following steps.

1. Randomly draw `nCandidates` candidates from an interval given by the user.
2. Label all candidates with a valid log-likelihood (LL) as *active*.
3. Apply `bfgsIter` iterations of the BFGS algorithm to each active candidate.
4. Apply the following tests to each candidate:
   (a) Has the BGFS search converged?
   (b) Are the candidate parameters after BFGS closer than `dTest` from any other candidate with higher LL?
   (c) Is the LL of the candidate after BFGS further than `distLL` from a candidate with better LL, and is its gradient smaller than `gTest`?
5. Mark any candidates for which at least one of these tests is passed as *inactive*.
6. Go back to step 3 for he remaining candidates.

The performance of the function varies across models and datasets and depends on the settings used. In particular, we advise to adjust the `bfgsIter`, `dTest`, `distLL` and `gTest` parameters to suit each user's particular model characteristics, as their default values might not be suitable for some model specifications. The running of the function is illustrated in Figure 50 for example `apollo_example_20.r`, where only a small part of the output is shown.

10.3 Out of sample fit

A common way to test for overfitting of a model is to measure its fit on a sample not used during estimation, i.e. measuring out-of-sample fit. A simple way to do this is splitting the available dataset into two parts: an estimation sample, and a validation sample. The model of interest is estimated using only the estimation sample, and then those estimated parameters are used

```
> apollo_beta=apollo_searchStart(apollo_beta, apollo_fixed,apollo_probabilities, apollo_inputs)

...

Initializing cluster.
Creating initial set of 100 candidate values.
Calculating LL of candidates 0%....50%.....100%

Stage 1, 100 active candidates.
 Estimating 20 BFGS iteration(s) for each active candidate.
 Candidate ..... LLstart ..... LLfinish ..... GradientNorm ... Converged
         1          -1756          -1562            78.273             0
         2          -1804          -1551           350.796             0
...
       100          -1867          -1562             8.991             0
 Candidate 1 dropped.
  Failed test 1 against 22 26 29 40 44 46 58 72 75 78 91 95 97 98 100
...
 Candidate 100 dropped.
  Failed test 1 against 29 40 44 72 91 95 98

 Best candidate so far (LL=-1551.2)

...

Stage 2, 20 active candidates.
 Estimating 20 BFGS iteration(s) for each active candidate.
 Candidate ..... LLstart ..... LLfinish ..... GradientNorm ... Converged
         2          -1551          -1549             0.012             1
...
        92          -1565          -1552           191.749             0

...

Stage 4, 5 active candidates.
 Estimating 20 BFGS iteration(s) for each active candidate.
 Candidate ..... LLstart ..... LLfinish ..... GradientNorm ... Converged
        15          -1579          -1559            23.317             0
        42          -1549          -1549             1.595             1
        61          -1562          -1562             0.243             1
        76          -1562          -1562                 0             1
        77          -1599          -1578           920.526             0
 Candidate 77 dropped.
  Failed test 1 against 76
  Failed test 2 against 76
 Best candidate so far (LL=-1549.3)
                    [,1]
asc_1             0.0724
asc_2             0.0000
beta_tt_a        -0.0701
beta_tt_b        -0.0709
beta_tc_a        -0.1101
beta_tc_b        -0.3758
beta_hw_a        -0.0437
beta_hw_b        -0.0383
beta_ch_a        -0.7840
beta_ch_b        -1.8961
delta_a          -0.1457
gamma_commute_a   0.2682
gamma_car_av_a   -0.0247
delta_b           0.0000
gamma_commute_b   0.0000
gamma_car_av_b    0.0000
```

Figure 50: Running `apollo_searchStart`

to measure the fit of the model (e.g. the log-likelihood of the model) on the validation sample. Doing this with only one validation sample may however lead to biased results, as a particular validation sample need not be representative of the population. One way to minimise this issue is to randomly draw several pairs of estimation and validation samples from the complete dataset, and apply the procedure to each pair. This also allows the calculation of a confidence interval for the out-of-sample measure of fit.

The function `apollo_outOfSample` implements the process described above. It is called as follows:

```
apollo_outOfSample(apollo_beta,
                   apollo_fixed,
                   apollo_probabilities,
                   apollo_inputs,
                   estimate_settings,)
                   outOfSample_settings)
```

The only new input here is `outOfSample_settings`, which has the following contents:

`nRep`: Number of times a different pair of estimation and validation sets are to be extracted from the full database (default is 10).

`validationSize`: Size of the validation sample. It can be provided as a fraction of the whole database (number between 0 and 1), or a number of individuals (number bigger than 1). The splitting of the database is done at the individual level, not at the observation level (default is 0.1).

`samples`: An optional numeric matrix or data.frame with as many rows as observations in the `database`, and as many columns as number of repetitions wanted. Each column represents a re-sample, and each element must be a 0 if the observation should be assigned to the estimation sample, or 1 if the observation should be assigned to the prediction sample. If this argument is provided, then `nRep` and `validationSize` are ignored. Note that this allows sampling at the observation rather than the individual level.

`apollo_outOfSample` saves to disk a file called `name_outOfSample_params.csv`, where `name` is the name of the model as defined in `apollo_control`. This file contains the estimates from each of the `nRep` estimation runs, as well as the estimation and out of sample log-likelihoods. It also saves a file `name_outOfSample_samples.csv` which contains information on the samples. In addition, the function prints to screen the per observation log-likelihood for each subsample, both for the estimation sample and the holdout sample. The running of the function is illustrated in Figure 51 for example `apollo_example_18.r`

10.4 Bootstrap estimation

Apollo also allows the user to use bootstrap estimation. Given a number of repetitions, this function generates as many new samples as requested, by sampling individuals (i.e. blocks of observations) *with replacement* from the original dataset. Then parameters are estimated for each of these new samples. Finally, the covariance matrix of the sequence of estimated parameters is calculated. This matrix is in itself an estimator of the covariance matrix of the parameter estimates.

The function `apollo_bootstrap` implements the process described above. It is called as

```
> apollo_outOfSample(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)

 Number of individuals for estimation : 349
 Number of individuals for forecasting: 39
 Total number of individuals in sample: 388

 Preparing loop.

 Estimated parameters and loglikelihoods of each sample will be
   written to: Apollo_example_18_outOfSampe_params.csv
 The matrix defining the observations used in each repetition will
   be written to: Apollo_example_18_outOfSampe_samples.csv

Estimation cycle 1
Using 3141 observations
Initial function value: -1557.318
Initial gradient value:
       asc_1     beta_tt_a     beta_tt_b     beta_tc_a     beta_tc_b     beta_hw_a     beta_hw_b     beta_ch_a
   -1.757518 -3544.366104   -797.898719    275.206767    -55.030148    851.256897    208.102798     66.955306
   beta_ch_b       delta_a
   -5.855591     10.292963
initial  value 1557.317915
iter   2 value 1487.178858
iter   3 value 1477.380459
iter   4 value 1467.882161
...

Estimation cycle 10
...
iter  45 value 1375.158283
iter  46 value 1375.158178
iter  47 value 1375.158142
iter  47 value 1375.158138
iter  47 value 1375.158123
final  value 1375.158123
converged
Estimation results written to file.

Processing time: 4.225905 mins
      LL per obs in estimation sample LL per obs in validation sample
 [1,]                      -0.4406197                       -0.5191336
 [2,]                      -0.4491659                       -0.4072436
 [3,]                      -0.4437753                       -0.4891126
 [4,]                      -0.4506910                       -0.4243149
 [5,]                      -0.4422518                       -0.5056741
 [6,]                      -0.4505546                       -0.4262517
 [7,]                      -0.4425831                       -0.4629453
 [8,]                      -0.4478721                       -0.4518991
 [9,]                      -0.4490924                       -0.4442491
[10,]                      -0.4378090                       -0.5066031
```

Figure 51: Running `apollo_outOfSample`

follows:

```
apollo_bootstrap(apollo_beta,
                 apollo_fixed,
                 apollo_probabilities,
                 apollo_inputs,
                 estimate_settings,)
                 bootstrap_settings)
```

The only new input here is `bootstrap_settings`, which has the following contents:

`nRep`: Number of bootstrap samples to use (default is 30).

samples: An optional numeric matrix or data.frame with as many rows as observations in the `database`, and as many columns as number of repetitions wanted. Each column represents a re-sample, and each element must be a 0 or a positive integer representing the number of times that row is used in that given sample. If this argument is provided, then `nRep` is ignored. Note that this allows sampling at the observation rather than the individual level.

seed: An optional positive integer used as seed for the bootstrap sampling generation process. Default is 24. It is only used if `samples` is `NA`. Changing the seed allows drawing new samples when re-starting a bootstrap process. This is useful when a bootstrap process has been interrupted, or when additional repetitions are needed.

`apollo_bootstrap` saves to disk a file called `name_bootstrap_params.csv`, where `name` is the name of the model as defined in `apollo_control`. This file contains the estimates from each of the `nRep` estimation runs, as well as the log-likelihoods. It also saves a file `name_bootstrap_samples.csv` which contains information on the samples and a file called `name_bootstrap_vcov.csv` containing the covariance matrix. In addition, the function prints to screen the covariance matrix. The running of the function is illustrated in Figure 52 for example `apollo_example_18.r`

```
> apollo_bootstrap(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs)

> apollo_bootstrap(apollo_beta, apollo_fixed,apollo_probabilities, apollo_inputs)

 30 new dataset will be constructed by randomly sampling
 388 individuals with replacement from the original dataset.

 Preparing bootstrap.

 Estimated parameters and loglikelihoods of each sample will be
   written to: Apollo_example_18_bootstrap_params.csv
 The matrix defining the observations used in each re-sampling will
   be written to: Apollo_example_18_bootstrap_samples.csv

Estimation cycle 1 (3492 obs)
initial   value 1640.462629
iter    2 value 1569.398593
iter    3 value 1550.356703
iter    4 value 1538.379789
...

Estimation cycle 30 (3492 obs)
...
iter  21 value 1401.723717
final   value 1401.723717
converged
Estimation results written to file.

Estimated parameters and loglikelihoods of each sample written to
  Apollo_example_18_bootstrap_params.csv

Matrix defining the observations used in each re-sampling written to
  Apollo_example_18_bootstrap_samples.csv

Covariance matrix of parameters written to
  Apollo_example_18_bootstrap_vcov.csv

Processing time: 14.95309 mins

                    asc_1 asc_2      beta_tt_a      beta_tt_b      beta_tc_a      beta_tc_b      beta_hw_a
asc_1       0.0026254245      0 -4.919614e-04  0.0024256232 -5.725252e-04  0.0143542141  2.200878e-04
asc_2       0.0000000000      0  0.000000e+00  0.0000000000  0.000000e+00  0.0000000000  0.000000e+00
...
```

Figure 52: Running `apollo_bootstrap`

The function can also be called directly during estimation, as described in Section .

10.5 Expectation-maximisation (EM) algorithm

Apollo allows the user to estimate models using Expectation - Maximisation (EM) algorithms. These are iterative algorithms were the updating of the parameters is usually achieved through the maximization of a simplified version of the model likelihood function. EM algorithms do not provide standard error estimates for the parameters. To obtain them, a Maximum Likelihood estimation with the EM estimated parameters as the starting values is typically run afterwards. This guarantees quick convergence and standard errors for all parameters. For a detailed discussion of EM algorithms, see Train (2009, ch. 14).

The precise steps of these algorithms change depending on the kind of model, making them hard to generalize and implement in a flexible way. It is thus up to the user to write the specific EM algorithm required by the desired model. Nevertheless, we provide three examples covering the most common choice models estimated using EM, so that a user can easily modify them to their own needs.

10.6 LC model without covariates in the allocation function

In this subsection, we describe the EM estimation of a choice model with S different classes. The conditional choice probability of class s (i.e. the in-class probability) is determined by a MNL model. All preference parameters β are allowed to vary across classes - this is a requirement when using EM, although it is possible to fix some of the parameters to zero in some of the classes. The class allocation probability for class s is determined by a single parameter π_s, and is the same for all individuals in the sample, i.e. there is no class allocation model using covariates. Estimation is achieved through an iterative five step process detailed below (cf. Train, 2009, ch. 14).

1. Definition of starting allocation probabilities π_s^0, and initial values for the preference parameters β_s^0 in each class. Preferences parameters should be different across classes.
2. Calculate class allocation probabilities for each individual conditional on observed choices for each class $s \in \{1, ..., S\}$, using the following expression.

$$h_{n,s}^0 = \frac{\pi_s^0 L_{n,s}(\beta_s^0)}{\sum_{s=1}^{S} \pi_s^0 L_{n,s}(\beta_s^0)} \tag{64}$$

   Where $L_{n,s}(\beta_s^0)$ is the likelihood of the observed choice for individual n assuming class s.
3. Update the allocation probabilities as follows.

$$\pi_s^1 = \frac{\sum_n h_{n,s}^0}{\sum_n \sum_{s'} h_{n,s'}^0} \tag{65}$$

4. Update the preference parameters for each class by estimating separate weighted MNL models. Estimation of each MNL model can be done using Maximum Likelihood, using $h_{n,s}^0$ as weights.

$$\beta_s^1 = argmax_{\beta_s^0}(\sum_{n=1}^{N} h_{n,s}^0 log(L_{n,s})) \; \forall s \tag{66}$$

5. Calculate the likelihood of the whole model and check for convergence. Convergence is achieved when the change on this likelihood is smaller than an pre-defined value. If convergence is not achieved, return to step 2.

We illustrate this example using the EM analogue of `Apollo_example_18.r`, i.e. the simple two-class LC model on the Swiss route choice data. This example is available in `Apollo_example_27.r`.

In terms of implementation, we have to write two different likelihood (probability) functions: (i) a class-specific conditional likelihood $L_{n,s}$, and (ii) the whole model likelihood $\sum_{s=1}^{S} \pi_s L_{n,s}$, where n enumerates individuals. The latter is given as before in `apollo_probabilities`. The conditional likelihoods are the only additional functions that must be written, compared to Maximum Likelihood estimation.

Figure 53 to 55 present code implementing this algorithm. In Figure 53, we define the name of the attribute containing the weights in `apollo_control`, where this is then used later in the code. In this implementation, we directly estimate the class allocation probabilities rather than using a logistic transform as in Section 6.2, and we thus simply define a parameter `pi_a` in `apollo_beta`, which is then also used in setting the list entry `lcpars[["pi_values"]]` in `apollo_lcPars`.

```
### Set core controls
apollo_control = list(
  modelName  ="Apollo_example_27",
...
  weights    = "weights",
...
)

...

### Vector of parameters, including any that are kept fixed in estimation
apollo_beta = c(asc_1            = 0,
...
                beta_ch_b        =-2.1725,
                pi_a             = 0.5)

...

# ################################################################# #
#### DEFINE LATENT CLASS COMPONENTS                                 ####
# ################################################################# #

apollo_lcPars=function(apollo_beta, apollo_inputs){
  lcpars = list()
...

  lcpars[["pi_values"]] = list(pi_a,1-pi_a)

  return(lcpars)
}
```

Figure 53: EM algorithm for simple latent class: initial steps

Figure 54 shows the definition of an additional probabilities function, `apollo_probabilities_within_class`, using a format consistent with the by now regular `apollo_probabilities`, but looking at the within class models only. As can be seen, this uses the scalar `apollo_inputs$s`, which is set inside the EM algorithm as we will see below, where this is then used to determine which set of parameters to use, i.e. which class we are working in. Additionally, as we are now using weighted estimation of the within class models, we make a call to `apollo_weighting` after taking the product across choices for the same individual. The remainder of this code is standard.

```
# ################################################################# #
#### DEFINE MODEL AND LIKELIHOOD FUNCTION FOR WITHIN CLASS       ####
# ################################################################# #

apollo_probabilities_within_class=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Determine which class we're working in
  s=apollo_inputs$s

  ### List of utilities: these must use the same names as in mnl_settings, order is irrelevant
  V=list()
  V[['alt1']]  = asc_1 + beta_tc[[s]]*tc1 + beta_tt[[s]]*tt1 + beta_hw[[s]]*hw1 + beta_ch[[s]]*ch1
  V[['alt2']]  = asc_2 + beta_tc[[s]]*tc2 + beta_tt[[s]]*tt2 + beta_hw[[s]]*hw2 + beta_ch[[s]]*ch2

  ### Define settings for MNL model component
  mnl_settings = list(
    alternatives = c(alt1=1, alt2=2),
    avail        = list(alt1=1, alt2=1),
    choiceVar    = choice,
    V            = V)

  ### Compute probabilities using MNL model
  P[["model"]] = apollo_mnl(mnl_settings, functionality)

  ### Take product across observation for same individual
  P = apollo_panelProd(P, apollo_inputs, functionality)

  ### Apply weights
  P = apollo_weighting(P, apollo_inputs, functionality)

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 54: EM algorithm for simple latent class: separate probabilities function for within class model

The key steps in the EM algorithm are implemented in Figure 55, following the five steps outlined above. We first create a backup of `apollo_fixed` as the fixed parameters will change throughout the algorithm. We then in step 1 initialise the class allocation probabilities and set a stopping criterion before beginning the loop over iterations. Steps 2 and 3 are direct implementations of the formulae shown above. Step 4 is split into two parts, one per class. We first create the vector of weights to be used in estimation, where these are the conditional class allocation probabilities for that class, where we replicate each of these once per observation per individual to achieve the same length as the number of rows in the data. We then place this inside `apollo_inputs$database` to be accessible inside `apollo_probabilities_within_class`. We then include all parameters that do not relate to the within class model for class 1 in `apollo_fixed` and set `apollo_inputs$s` to 1 for use inside `apollo_probabilities_within_class`, before estimating the within class parameters and updating `apollo_beta`. The process is repeated for class 2. Finally, step 5 checks for convergence by calculating the overall likelihood and comparing it to the previous iteration. Once convergence has been reached, we make a call to classical estimation with 0 iterations, i.e. only estimating the covariance matrix.

```
### Keep backup of vector of fixed parameters as this changes throughout
apollo_fixed_base = apollo_fixed

### Step 1 ###

### Initialise class allocation probabilities
apollo_beta["pi_a"]=0.5

### Loop over repeated EM iterations until convergence has been reached
stopping_criterion=10^-5
iteration=1
stop=0
while(stop==0){
  cat("Starting iteration: ",iteration,"\n",sep="")

  ### Step 2 ###

  ### Calculate model likelihood and class specific likelihoods
  L=apollo_probabilities(apollo_beta, apollo_inputs, functionality="output")

  ### Calculate class specific conditional likelihoods
  h1=as.vector(apollo_beta["pi_a"]*L[[1]]/L[[3]])
  h2=as.vector((1-apollo_beta["pi_a"])*L[[2]]/L[[3]])

  ### Calculate current log-likelihood for LC model
  Lcurrent=sum(log(L[[3]]))
  cat("Current LL: ",Lcurrent,"\n",sep="")

  ### Step 3 ###

  ### Update shares in classes
  apollo_beta["pi_a"]=sum(h1)/(sum(h1)+sum(h2))

  ### Step 4 ###

  ### Update coefficients in class specific models by estimating class specific models
  ### using posterior class allocation probabilities as weights

  ### Class 1

  ### Replicate individual-specific weights for each observation
  nObsPerIndiv <- as.vector(table(database[,apollo_control$indivID]))
  apollo_inputs$database$weights=rep(h1,times=nObsPerIndiv)

  ### Set fixed parameters (only estimating parameters for class 1)
  apollo_fixed=c("asc_2","beta_tt_b","beta_tc_b","beta_hw_b","beta_ch_b","pi_a")

  ### Set class index to use inside apollo_probabilities_within_class
  apollo_inputs$s=1

  ### Estimate class-specific weighted MNL model
  model = apollo_estimate(apollo_beta, apollo_fixed,
                          apollo_probabilities_within_class, apollo_inputs,
                          estimate_settings=list(writeIter=FALSE,silent=TRUE,hessianRoutine="none"))

  ### Update overall parameters
  apollo_beta=model$estimate

  ### Class 2

...

  ### Step 5 ###

  ### Calculate new log-likelihood and compute improvement
  Lnew=sum(log(apollo_probabilities(apollo_beta, apollo_inputs, functionality="output")[[3]]))
  change=Lnew-Lcurrent
  cat("New LL: ",Lnew,"\n",sep="")
  cat("Improvement: ",change,"\n\n",sep="")

  ### Determine whether convergence has been reached
  if(change<stopping_criterion) stop=1
  iteration=iteration+1
}

#### CLASSICAL ESTIMATION FOR COVARIANCE MATRIX                    ####
### Reinstate original vector of fixed parameters
apollo_fixed=apollo_fixed_base

model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs,
    ↪ estimate_settings=list(maxIterations=0))
```

Figure 55: EM algorithm for simple latent class: EM process

10.7 LC model with covariates in the allocation function

We next turn to a latent class model with C different classes where the class allocation is determined by a MNL model using covariates Z_n such as an individual's income, causing the allocation probabilities $\pi_{n,s}$ to change from one individual to the next.

Estimation is achieved through an iterative process detailed below, where this incorporates some departures from the approach in Section 10.6, drawing on Bhat (1997).

1. Definition of starting values for γ and β_s parameters, where β_s should be different across classes.
2. Calculate class allocation probabilities conditional on observed choices for each class $s \in \{1, ..., S\}$, using the following expression.

$$h^0_{n,s} = \frac{\pi^0_{n,s}(\gamma^0) L_{n,s}(\beta^0_s)}{\sum_{s=1}^{S} \pi^0_{n,s}(\gamma^0) L_{n,s}(\beta^0_s)} \tag{67}$$

   where $L_{n,s}(\beta^0_s)$ is the likelihood of the observed choice for individual n assuming class s, and where $\pi^0_{n,s}(\gamma^0)$ is the class allocation probability for individual n for class s, using γ^0 as parameters.
3. Update the parameters γ used in the class allocation model by maximising the allocation probabilities weighted by $h^0_{n,s}$.

$$\gamma^1 = argmax_{\gamma^0}(\sum_{n=1}^{N}\sum_{s=1}^{S} h^0_{n,s} log(\pi_{n,s})) \tag{68}$$

4. Update the parameters β_s for the within class model for each class by estimating separate weighted MNL models, just as in the procedure in Section 10.6. Estimation of each MNL model can be done using Maximum Likelihood, using $h^0_{n,s}$ as weights.

$$\beta^1_s = argmax_{\beta^0_s}(\sum_{n=1}^{N} h^0_{n,s} log(L_{n,s}))\ \forall s \tag{69}$$

5. Calculate the likelihood of the whole model and check for convergence. Convergence is achieved when the change on this likelihood is smaller than an pre-defined value. If convergence is not achieved, return to step 2.

We illustrate this example using the EM analogue of `Apollo_example_20.r`, i.e. the two-class LC model on the Swiss route choice data, with covariates in the class allocation model. This example is available in `Apollo_example_28.r`. In our discussions here, we focus on those parts that differ from the example in the previous section. The model parameters are the same as in `Apollo_example_20.r`, as shown in Section 6.2, as is the definition of `apollo_lcPars` and `apollo_probabilities`. Furthermore, the definition of the within class probabilities function, i.e. `apollo_probabilities_within_class`, is the same as in `Apollo_example_27.r`, cf. Figure 54.

The first difference arises in the need to create an additional probabilities function for the class allocation model, defined as `apollo_probabilities_class`, and shown in Figure 56. This loads the posterior class allocation probabilities from inside `apollo_inputs` where they are set inside the main algorithm described later, and then manually implements the model from Equation 68.

We next turn to the actual EM algorithm, shown in Figure 57. The first difference arises in step 2 in the code. In the example without covariates, the unconditional class allocation probabilities were simply given by a parameter in `apollo_beta`. In this model, we create a temporary model object which contains `apollo_beta` as estimates and then make a call to `apollo_lcUnconditionals` to compute the class allocation probabilities. The remainder of this step is the same as in Figure 55. Step 3 is entirely different in that it estimates the parameters for the class allocation model through maximisation of `apollo_probabilities_class`, where the earlier conditional class allocation probabilities are placed into `apollo_inputs` to be accessible inside the model. Steps 4 and 5 remain the same as before.

```
apollo_probabilities_class=function(apollo_beta, apollo_inputs, functionality="estimate"){

  ### Attach inputs and detach after function exit
  apollo_attach(apollo_beta, apollo_inputs)
  on.exit(apollo_detach(apollo_beta, apollo_inputs))

  ### Create list of probabilities P
  P = list()

  ### Load posterior class allocation probabilities from inputs
  h_1=apollo_inputs$h1
  h_2=apollo_inputs$h2
  h_grouped=list(h_1,h_2)

  ### Take logs of class allocation probabilities
  log_pi_values=lapply(pi_values,log)

  ### Define model that aims to minimise difference between posterior and unconditional class allocation
      ↪ probabilities
  P[["model"]]=exp(Reduce('+', mapply('*',h_grouped,log_pi_values,SIMPLIFY = FALSE)))

  ### Prepare and return outputs of function
  P = apollo_prepareProb(P, apollo_inputs, functionality)
  return(P)
}
```

Figure 56: EM algorithm for latent class with covariates: separate probabilities function for class allocation model

10.8 MMNL model with full covariance matrix for random coefficients

In this subsection, we describe the EM estimation of a MMNL model in which all parameters are random and where we estimate a full covariance matrix between them. More formally, we assume the preference parameters to follow a joint random normal distribution $\beta \sim N(\mu, \Sigma)$, where transformations to other distributions are straightforward, as explained in our example below.

The iterative process is described below (cf. Train, 2009, chapter 11).

1. Define starting values for μ^0 and Σ^0.
2. Generate R multivariate draws for each individual n, say $\beta^0_{n,r}$ for draw r, where $\beta^0_{n,r}$ contains one value for each random parameter.

```
### Keep backup of vector of fixed parameters as this changes throughout
apollo_fixed_base = apollo_fixed

### Create temporary model object
model=list()

### Loop over repeated EM iterations until convergence has been reached
stopping_criterion=10^-5
iteration=1
stop=0

while(stop==0){
  cat("Starting iteration: ",iteration,"\n",sep="")

  ### Step 2 ###

  ### calculate model likelihood and class specific likelihoods
  L=apollo_probabilities(apollo_beta, apollo_inputs, functionality="output")

  ### Calculate class specific conditional likelihoods
  model$estimate=apollo_beta
  pi=apollo_lcUnconditionals(model, apollo_inputs)[["pi_values"]]
  h1=as.vector(pi[[1]]*L[[1]]/L[[3]])
  h2=as.vector(pi[[2]]*L[[2]]/L[[3]])

  ### Calculate current log-likelihood for LC model
  Lcurrent=sum(log(L[[3]]))
  cat("Current LL: ",Lcurrent,"\n",sep="")

  ### Step 3 ###

  ### Update shares in classes by optimising class allocation model only

  ### Fix all parameters for within class models
  apollo_fixed=c("asc_1",
                 "asc_2",
                 "beta_tt_a",
                 "beta_tt_b",
                 "beta_tc_a",
                 "beta_tc_b",
                 "beta_hw_a",
                 "beta_hw_b",
                 "beta_ch_a",
                 "beta_ch_b",
                 "delta_b",
                 "gamma_commute_b",
                 "gamma_car_av_b" )

  ### Put posterior class allocation probabilities into input object for use inside
      ↪ apollo_probabilities_class
  apollo_inputs$h1=h1
  apollo_inputs$h2=h2

  ### Estimate class allocation model
  model = apollo_estimate(apollo_beta, apollo_fixed,
                          apollo_probabilities_class, apollo_inputs,
                          estimate_settings=list(writeIter=FALSE,silent=TRUE,hessianRoutine="none"))

  ### Update overall parameters
  apollo_beta=model$estimate

  ### Step 4 ###
...
  iteration=iteration+1
}
```

Figure 57: EM algorithm for latent class with covariates: EM process

3. Calculate the model likelihood at the individual level for each draw, i.e. $L_{n,r}(\beta^0_{n,r})$.
4. Calculate weights for each draw and for each individual using the following expression.

$$w^0_{n,r} = \frac{L_{n,r}}{\frac{\sum_r L_{n,r}}{R}} \; \forall r, n \tag{70}$$

5. Update the means of the random parameters.

$$\mu^1 = \frac{w^0_{n,r}\beta^0_{n,r}}{RN} \tag{71}$$

   Where N is the number of individuals in the sample.
6. Update the covariance matrix of the random parameters, given by

$$\Sigma^0 = \frac{w^0_{n,r}\Sigma^0_{n,r}}{RN}, \tag{72}$$

   where this requires calculating the covariance matrix $\Sigma^0_{n,r}$ at the individual draw level, given by:

$$\Sigma^0_{n,r} = \left(\beta^0_{n,r} - \mu^1\right) \cdot \left(\beta^0_{n,r} - \mu^1\right)'. \tag{73}$$

7. Calculate the likelihood of the whole model and check for convergence. Convergence is achieved when the change on this likelihood is smaller than an pre-defined value. If convergence is not achieved, return to step 2.

Unlike previous examples in this section, the EM estimation of MMNL models does not require writing additional likelihood functions, as compared to the Maximum Likelihood implementation. Functions `apollo_randCoeff` and `apollo_probabilities` are sufficient to do EM estimation. Furthermore, this algorithm has the benefit of not needing any maximisation at all. Figure 58 presents code implementing this algorithm for the EM analogue of `Apollo_example_15.r`, i.e. using a MMNL model with correlated negative Lognormals on the Swiss route choice data. This example is implemented in `Apollo_example_29.r`. We focus solely on the EM algorithm steps as the definition of `apollo_randCoeff` and `apollo_probabilities` remains the same as in `Apollo_example_15.r`.

The actual implementation is straightforward and only a few points need discussing. Firstly, we again make use of a temporary model object in which we place the current estimates, allowing us to then make the call to `apollo_unconditionals` to obtain the actual coefficient values. These are then negative lognormals, and we first translate them back to Normals, using the logarithm of the negative draws. The implementation of Equation 72 is followed by the calculation of the Cholesky matrix for that covariance of the draws, as this corresponds to the parameters used in `apollo_beta` and then `apollo_randCoeff`.

```
### Create temporary model object
model=list()

### Calculate initial likelihood at the level of each draw
Ln=apollo_probabilities(apollo_beta, apollo_inputs, functionality="conditionals")
cat("Initial LL:",sum(log(rowMeans(Ln))),"\n")

### Loop over repeated EM iterations until convergence has been reached
stopping_criterion=10^-5
iteration=1
stop=0

while(stop==0){
  cat("Starting iteration: ",iteration,"\n",sep="")
  cat("Current LL: ",sum(log(rowMeans(Ln))),"\n",sep="")

  ### Calculate weight for each individual and for each draw
  wn=Ln/(rowMeans(Ln))

  ### Copy current parameter values into temporary model object
  model$estimate=apollo_beta

  ### Produce draws for random coefficients with current vector of parameters
  d=apollo_unconditionals(model,apollo_probabilities,apollo_inputs)

  ### Translate draws back to Normal from negative Lognormal
  d=lapply(d,function(x) log(-x))

  ### Apply weights to individual draws and turn into a matrix
  dwn=lapply(d,"*",wn)
  dwn=lapply(dwn,as.vector)
  dwn=do.call(cbind,dwn)

  ### Calculate means for weighted draws
  mu=colMeans(dwn)

  ### Calculate weighted covariance matrix
  K = length(mu)    # n of coefficients
  R = ncol(d[[1]]) # n of draws
  N = nrow(d[[1]]) # n of individuals
  tmp=matrix(0, nrow=N, ncol=K)
  Omega=matrix(0, nrow=K, ncol=K)
  for(r in 1:R){
    for(k in 1:K) tmp[,k] = d[[k]][,r] - mu[k]
    for(n in 1:N) Omega = Omega + wn[n,r]*(tmp[n,] %*% t(tmp[n,]))
  }

  ### Compute Cholesky of average weighted covariance matrix
  cholesky = chol(Omega/(N*R))
  cholesky = cholesky[upper.tri((cholesky),diag=TRUE)]

  ### Update vector of model parameters on the basis of calculated mu and Omega
  apollo_beta[1:4]=mu
  apollo_beta[5:14]=cholesky

  ### Calculate likelihood with new parameters
  Lnew=((apollo_probabilities(apollo_beta, apollo_inputs, functionality="conditionals")))

  ### Compute improvement
  change=sum(log(rowMeans(Lnew)))-sum(log(rowMeans(Ln)))
  cat("New LL: ",sum(log(rowMeans(Lnew))),"\n",sep="")
  cat("Improvement: ",change,"\n\n",sep="")
  Ln=Lnew

  ### Determine whether convergence has been reached
  if(change<stopping_criterion) stop=1
  iteration=iteration+1
}

#### CLASSICAL ESTIMATION FOR COVARIANCE MATRIX                    ####

### Reinstate original vector of fixed parameters
apollo_fixed=apollo_fixed_base

model = apollo_estimate(apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs,
                        estimate_settings=list(hessianRoutine="maxLik",maxIterations=0))
```

Figure 58: EM estimation of Mixed Logit with correlated negative Lognormals

A *Apollo* versions: timeline, changes and backwards compatibility

Version 0.0.6 (13 March 2019)

This is the first fully functioning release of *Apollo*.

Version 0.0.7 (8 May 2019)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

Inputs changed for `apollo_choiceAnalysis`
Functions affected: `apollo_choiceAnalysis`
Detailed description: inputs changed so function can be called prior to `apollo_validateInputs`
Backwards compatibility of code: function call changed from version 0.0.7 onwards

Constraints for classical estimation
Functions affected: `apollo_estimate`
Detailed description: *Apollo* now allows the user to include a list called `constraints` in `estimate_settings` for use with BFGS for classical model estimation.
Backwards compatibility of code: no backwards compatibility issues for existing functions

Scaling of parameters during model estimation
Functions affected: `apollo_estimate`
Detailed description: scaling of model parameters can be used during estimation
Backwards compatibility of code: no backwards compatibility issues for existing functions

Validation output
Functions affected: `apollo_estimate`
Detailed description: *Apollo* no longer reports that all pre-estimation checks were passed for a model component and instead only reports if there are an issues.
Backwards compatibility of code: no backwards compatibility issues for existing functions

Bayesian estimation produces `model$estimate`
Functions affected: `apollo_estimate`, `apollo_prediction`, `apollo_llCalc`
Detailed description: until version 0.0.6, Bayesian estimation in *Apollo* did not produce a `model$estimate` output. We have retained the various existing outputs, but in addition, `model$estimate` is now produced, combining non-random parameters with individual specific

posteriors for random parameters. This now allows the user to use `apollo_prediction` and `apollo_llCalc` on such outputs, where care is of course required in interpretation of outputs based on posterior means.
Backwards compatibility of code: no backwards compatibility issues for existing functions

Changes to *Apollo* examples:

Examples affected: `apollo_example_1.r` and `apollo_example_2.r`
Detailed description: Use of `apollo_choiceAnalysis` added
Backwards compatibility of examples: affected part only works from version 0.0.7 onwards

Examples affected: `apollo_example_12.r`
Detailed description: Scaling in estimation implemented
Backwards compatibility of examples: only works from version 0.0.7 onwards

Examples affected: `apollo_example_26.r`
Detailed description: HB prediction component added
Backwards compatibility of examples: affected part only works from version 0.0.7 onwards

Bug fixes:

apollo_speedTest
This function was unintentionally hidden from users in previous versions

Version 0.0.8 (9 September 2019)

Changes to *Apollo* code:

General
Minor improvements to efficiency, stability and reporting of user errors.

Bootstrap estimation added
Functions affected: `apollo_bootstrap`, `apollo_estimate`
Detailed description: the user can now perform bootstrap estimation. This can also be called directly with `apollo_estimate` during estimation
Backwards compatibility of code: new function from version 0.0.8 onwards, new optional arguments for `apollo_estimate`, but function called in the same way

Allow user to use subset of rows for analysis of choices
Functions affected: `apollo_choiceAnalysis`
Detailed description: An additional `rows` argument can be entered into `choiceAnalysis_settings`.
Backwards compatibility of code: optional argument added from version 0.0.8 onwards, but function called in the same way

Outputs changed for `apollo_choiceAnalysis`
Functions affected: `apollo_choiceAnalysis`
Detailed description: outputs changed so that t-test value is reported instead of p-value, and order of outputs is changed
Backwards compatibility of code: outputs changed from version 0.0.8 onwards, but function called in the same way

Allow user to change name and location of outside good in MDCEV and MDCNEV
Functions affected: `apollo_mdcev` and `apollo_mdcnev`
Detailed description: An additional `outside` argument can be entered into `mdcev_settings` and `mdcnev_settings` with the name of the outside good which can now differ from `outside`. It also no longer needs to be in first position in the list of alternatives.
Backwards compatibility of code: optional argument added from version 0.0.8 onwards, but function called in the same way

No need to define superfluous γ for outside good in MDCEV and MDCNEV
Functions affected: `apollo_mdcev` and `apollo_mdcnev`
Detailed description: The user no longer needs to create a `gamma` term for the outside good.
Backwards compatibility of code: function called in the same way

Chosen unavailable alternatives have a likelihood of zero
Functions affected: `apollo_mnl, apollo_mdcev, apollo_mdcnev`
Detailed description: MNL, MDCEV and MDCNEV now return a likelihood equal to zero for chosen alternatives that are not available. This change is only relevant if `apollo_control$noValidation` is `TRUE`.
Backwards compatibility of code: new likelihood values for unavailable chosen alternatives on MNL, MDCEV and MDCNEV models from version 0.0.8 onwards

Allow user to specify number of outliers to report
Functions affected: `apollo_modelOutput, apollo_saveOutput`
Detailed description: In addition to specifying TRUE/FALSE for printOutliers, the user can provide the number of outliers to report (instead of the default of 20).
Backwards compatibility of code: optional argument added from version 0.0.8 onwards, but function called in the same way

Ability to define estimation/validation subsets for `apollo_outOfSample`
Functions affected: `apollo_outOfSample`
Detailed description: the user can now provide a matrix or data.frame describing which observations are to be used in the estimation and validation subsets
Backwards compatibility of code: optional argument added from version 0.0.8 onwards, but function called in the same way

Individual IDs and choice scenario numbers added in predictions
Functions affected: `apollo_prediction`
Detailed description: The output from `apollo_prediction` now includes the IDs and choice observation numbers as the first two columns.
Backwards compatibility of code: function called in the same way

Changes to *Apollo* examples:

Examples affected: `apollo_example_3.r`, `apollo_example_11.r`, `apollo_example_13.r`, `apollo_example_22.r`, `apollo_example_24.r`, `apollo_example_25.r`, `apollo_example_26.r`
Detailed description: `apollo_prediction` now includes the IDs and observation numbers as the first two columns, meaning some output is shifted.
Backwards compatibility of examples: use of outputs needs adjusting to reflect change in columns

Bug fixes:

apollo_combineResults
This function failed in earlier versions when using only a single model

apollo_firstRow
This function mistakenly replicated the first row for each person T_n times

apollo_estimate with scaling and HB
HB estimation failed in earlier versions for models without any random parameters

B Data dictionaries

Tables A1 to A4 present data dictionaries for the four datasets made available with *Apollo*.

C Index of example files

Table A5 presents an overview of the example files made available with *Apollo*, while Table A6 shows which function is used with what example.

D Overview of functions and elements

Table A7 presents an overview of all *Apollo* functions, together with their inputs and outputs. Table A8 and Table A9 give an overview of all the lists and elements used by *Apollo*.

Table A1: Data dictionary for `apollo_modeChoiceData.csv`

Individuals	500
Observations	8,000

Variable	Description	Values
ID	Unique individual ID	1 to 500
RP	RP data identifier	1 for RP, 0 for SP
SP	SP data identifier	1 for SP, 0 for RP
RP_journey	Index for RP observations	1 to 2, NA for SP
SP_task	Index for SP observations	1 to 14, NA for RP
av_car	availability for alternative 1 (car)	1 for available, 0 for unavailable
av_bus	availability for alternative 2 (bus)	1 for available, 0 for unavailable
av_air	availability for alternative 3 (air)	1 for available, 0 for unavailable
av_rail	availability for alternative 4 (rail)	1 for available, 0 for unavailable
time_car	travel time (mins) for alternative 1 (car)	Min: 250, mean: 311.79, max: 390 (0 if not available)
cost_car	travel cost (£) for alternative 1 (car)	Min: 30, mean: 39.99, max: 50 (0 if not available)
time_bus	travel time (mins) for alternative 2 (bus)	Min: 300, mean: 370.29, max: 420 (0 if not available)
cost_bus	travel cost (£) for alternative 2 (bus)	Min: 15, mean: 25.02, max: 35 (0 if not available)
access_bus	access time (mins) for alternative 2 (bus)	Min: 5, mean: 15.02, max: 25 (0 if not available)
time_air	travel time (mins) for alternative 3 (air)	Min: 50, mean: 70.07, max: 90 (0 if not available)
cost_air	travel cost (£) for alternative 3 (air)	Min: 50, mean: 79.94, max: 110 (0 if not available)
access_air	access time (mins) for alternative 3 (air)	Min: 35, mean: 45.02, max: 55 (0 if not available)
service_air	service quality for alternative 3 (air)	1 to 3 (0 if not available)
time_rail	travel time (mins) for alternative 4 (rail)	Min: 120, mean: 142.93, max: 170 (0 if not available)
cost_rail	travel cost (£) for alternative 4 (rail)	Min: 35, mean: 55.03, max: 75 (0 if not available)
access_rail	access time (mins) for alternative 4 (rail)	Min: 5, mean: 14.96, max: 25 (0 if not available)
service_rail	service quality for alternative 4 (rail)	1 to 3 (0 if not available)
female	dummy variable for female individuals	1 for female, 0 otherwise
business	dummy variable for business trips	1 for business trips, 0 otherwise
income	income variable (£ per annum)	Min: 15,490, mean: 44,748.27, max: 74,891
choice	choice variable	1 for car, 2 for bus, 3 for air, 4 for rail

Table A2: Data dictionary for `apollo_swissRouteChoiceData.csv`

Individuals	388
Observations	3,492

Variable	Description	Values
ID	Unique individual ID	2,439 to 84,525
choice	choice variable	1 for alternative 1, 2 for alternative 2
tt1	travel time (mins) for alternative 1	Min: 2, mean: 52.59, max: 389
tc1	travel cost (CHF) for alternative 1	Min: 1, mean: 19.67, max: 206
hw1	headway (mins) for alternative 1	Min: 15, mean: 32.48, max: 60
ch1	interchanges for alternative 1	Min: 0, mean: 0.94, max: 2
tt2	travel time (mins) for alternative 2	Min: 2, mean: 52.47, max: 385
tc2	travel cost (CHF) for alternative 2	Min: 1, mean: 19.69, max: 268
hw2	headway (mins) for alternative 2	Min: 15, mean: 32.38, max: 60
ch2	interchanges for alternative 2	Min: 0, mean: 0.95, max: 2
hh_inc_abs	household income (CHF per annum)	Min: 10,000, mean: 76,507.73, max: 167,500
car_availability	car availability	1 for yes, 0 otherwise
commute	dummy variable for commute trips	1 for commute trips, 0 otherwise
shopping	dummy variable for shopping trips	1 for shopping trips, 0 otherwise
business	dummy variable for business trips	1 for business trips, 0 otherwise
leisure	dummy variable for leisure trips	1 for leisure trips, 0 otherwise

Table A3: Data dictionary for `apollo_drugChoiceData.csv`

Individuals	1,000
Observations	10,000

Variable	Description	Values
ID	Unique respondent ID	1 to 1,000
task	Index for SP choice tasks	1 to 10
best	first ranked alternative	1 to 4
second_pref	second ranked alternative	1 to 4
third_pref	third ranked alternative	1 to 4
worst	worst ranked alternative	1 to 4
brand_1	brand for first alternative	Artemis; Novum
country_1	country for first alternative	Switzerland; Denmark; USA
char_1	characteristics for first alternative	standard; fast acting; double strength
side_effects_1	rate of side effects for first alternative (out of 100,000)	Min: 1, mean: 37, max: 100
price_1	price (£) for first alternative	Min: 2.25, mean: 3.15, max: 4.5
brand_2	brand for second alternative	Artemis; Novum
country_2	country for second alternative	Switzerland; Denmark; USA
char_2	characteristics for second alternative	standard; fast acting; double strength
side_effects_2	rate of side effects for second alternative (out of 100,000)	Min: 1, mean: 37, max: 100
price_2	price (£) for second alternative	Min: 2.25, mean: 3.15, max: 4.5
brand_3	brand for third alternative	BestValue; Supermarket; PainAway
country_3	country for third alternative	USA; India; Russia; Brazil
char_3	characteristics for third alternative	standard; fast acting
side_effects_3	rate of side effects for third alternative (out of 100,000)	Min: 10, mean: 370, max: 1,000
price_3	price (£) for third alternative	Min: 0.75, mean: 1.75, max: 2.5
brand_4	brand for fourth alternative	BestValue; Supermarket; PainAway
country_4	country for fourth alternative	USA; India; Russia; Brazil
char_4	characteristics for fourth alternative	standard; fast acting
side_effects_4	rate of side effects for fourth alternative (out of 100,000)	Min: 10, mean: 370, max: 1,000
price_4	price (£) for fourth alternative	Min: 0.75, mean: 1.75, max: 2.5
regular_user	dummy variable for regular users	1 for regular users, 0 otherwise
university_educated	dummy variable for university educated	1 for university educated, 0 otherwise
over_50	dummy variable for age over 50 years	1 for age over 50 years, 0 otherwise
attitude_quality	Answer to *"I am concerned about the quality of drugs developed by unknown companies"*	Likert scale from 1 (strongly disagree) to 5 (strongly agree)
attitude_ingredients	Answer to *"I believe that ingredients are the same no matter what the brand"*	Likert scale from 1 (strongly disagree) to 5 (strongly agree)
attitude_patent	Answer to *"The original patent holders have valuable experience with their medecines"*	Likert scale from 1 (strongly disagree) to 5 (strongly agree)
attitude_dominance	Answer to *"I believe the dominance of big pharmaceutical companies is unhelpful"*	Likert scale from 1 (strongly disagree) to 5 (strongly agree)

Table A4: Data dictionary for `apollo_timeUseData.csv`

Individuals	447
Observations	2,826

Variable	Description	Values
indivID	Unique respondent ID	19209 to 9959342
day	Index of the day for the individual (day 1 excluded from data)	2 to 14
date	Date in format yyyymmdd	20161014 to 20170308
budget	Total amount of time registered during the day (in minutes)	1440 to 1440
t_a01	time spent dropping-off or picking up other people (in minutes)	0 to 1153
t_a02	time spent working (in minutes)	0 to 1425
t_a03	time spent on educational activities (in minutes)	0 to 1050
t_a04	time spent shopping (in minutes)	0 to 1434
t_a05	time spent on private business (in minutes)	0 to 1077
t_a06	time spent getting petrol (in minutes)	0 to 896
t_a07	time spent on social or leasure activities (in minutes)	0 to 1425
t_a08	time spent on vacation or on long (intercity) travel (in minutes)	0 to 828
t_a09	time spent doing exercise (in minutes)	0 to 1416
t_a10	time spent at home (in minutes)	0 to 1440
t_a11	time spent travelling (everyday travelling) (in minutes)	0 to 1182
t_a12	Non-allocated time (in minutes)	0 to 1160
female	dummy variable for female individuals	1 for female, 0 otherwise
age	age of the respondent (in years, approximate)	21 to 80
occ_full_time	dummy for respondents working full time	1 for respondents working full time, 0 otherwise
weekend	dummy for weekend days	1 for weekend, 0 otherwise

Table A5: Index of example files

File	Description	Data file
Apollo_example_1.r	Simple MNL model on mode choice RP data	apollo_modeChoiceData.csv
Apollo_example_2.r	Simple MNL model on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_3.r	MNL model with socio-demographics on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_4.r	Two-level NL model with socio-demographics on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_5.r	Three-level NL model with socio-demographics on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_6.r	CNL model with socio-demographics on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_7.r	Simple RRM model on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_8.r	DFT model on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_9.r	DFT model on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_10.r	Exploded logit model on drug choice data	apollo_drugChoiceData.csv
Apollo_example_11.r	MDCEV model on time use data without outside good and constants only in utilities	apollo_timeUseData.csv
Apollo_example_12.r	MDCEV model on time use data with outside good and with covariates in utilities	apollo_timeUseData.csv
Apollo_example_13.r	MDCNEV model on time use data with outside good and with covariates in utilities	apollo_timeUseData.csv
Apollo_example_14.r	Mixed logit model on Swiss route choice data, uncorrelated Lognormals in utility space	apollo_swissRouteChoiceData.csv
Apollo_example_15.r	Mixed logit model on Swiss route choice data, correlated Lognormals in utility space	apollo_swissRouteChoiceData.csv
Apollo_example_16.r	Mixed logit model on Swiss route choice data, WTP space with correlated and flexible distributions, inter and intra-individual heterogeneity	apollo_swissRouteChoiceData.csv
Apollo_example_17.r	Mixed MDCEV model on time use data, alpha-gamma profile, no outside good and random constants only in utilities	apollo_timeUseData.csv
Apollo_example_18.r	Simple LC model on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_19.r	Simple DM model on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_20.r	LC model with class allocation model on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_21.r	Latent class with continuous random parameters on Swiss route choice data	apollo_swissRouteChoiceData.csv
Apollo_example_22.r	RP-SP model on mode choice data	apollo_modeChoiceData.csv
Apollo_example_23.r	Best-worst model on drug choice data	apollo_drugChoiceData.csv
Apollo_example_24.r	ICLV model on drug choice data, using ordered measurement model for indicators	apollo_drugChoiceData.csv
Apollo_example_25.r	ICLV model on drug choice data, using continuous measurement model for indicators	apollo_drugChoiceData.csv
Apollo_example_26.r	HB model on mode choice SP data	apollo_modeChoiceData.csv
Apollo_example_27.r	Simple LC model on Swiss route choice data, EM algorithm	apollo_swissRouteChoiceData.csv
Apollo_example_28.r	LC model with class allocation model on Swiss route choice data, EM algorithm	apollo_swissRouteChoiceData.csv
Apollo_example_29.r	Mixed logit model on Swiss route choice data, correlated Lognormals in utility space, EM algorithm	apollo_swissRouteChoiceData.csv

Table A6: Functions used in example files

	Apollo_example_1.r	Apollo_example_2.r	Apollo_example_3.r	Apollo_example_4.r	Apollo_example_5.r	Apollo_example_6.r	Apollo_example_7.r	Apollo_example_8.r	Apollo_example_9.r	Apollo_example_10.r	Apollo_example_11.r	Apollo_example_12.r	Apollo_example_13.r	Apollo_example_14.r	Apollo_example_15.r	Apollo_example_16.r	Apollo_example_17.r	Apollo_example_18.r	Apollo_example_19.r	Apollo_example_20.r	Apollo_example_21.r	Apollo_example_22.r	Apollo_example_23.r	Apollo_example_24.r	Apollo_example_25.r	Apollo_example_26.r	Apollo_example_27.r	Apollo_example_28.r	Apollo_example_29.r
apollo_attach	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_avgInterDraws														x	x	x	x				x			x	x				x
apollo_avgIntraDraws																x													
apollo_bootstrap																		x		x									
apollo_choiceAnalysis	x	x																											
apollo_cnl						x																							
apollo_combineModels																						x	x	x	x				
apollo_combineResults						x																							
apollo_conditionals																x								x	x				
apollo_deltaMethod			x																										
apollo_detach	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_dft								x	x																				
apollo_el										x																			
apollo_estimate	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_firstRow																x		x	x	x	x							x	
apollo_fitsTest			x																										
apollo_initialise	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_lc																		x	x	x	x						x	x	
apollo_lcConditionals																				x									
apollo_lcUnconditionals																				x								x	
apollo_lcPars																		x	x	x	x						x	x	
apollo_llCalc																								x					
apollo_loadModel																x													
apollo_lrTest			x	x	x																								
apollo_mdcev											x	x					x												
apollo_mdcnev													x																
apollo_mnl	x	x	x				x							x	x	x		x	x	x	x	x	x	x	x	x	x	x	x
apollo_modelOutput	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_nl				x	x																								
apollo_normalDensity																									x				
apollo_ol																								x					
apollo_outOfSample																		x		x									
apollo_panelProd	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_prediction			x								x		x									x		x	x				
apollo_prepareprob	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_probabilities	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_randCoeff														x	x	x	x				x			x	x				x
apollo_readBeta			x	x	x	x							x		x														
apollo_saveOutput	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_searchStart																		x	x	x									
apollo_sharesTest			x																										
apollo_speedTest																x													x
apollo_unconditionals																x								x	x				x
apollo_validateInputs	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
apollo_weighting																											x	x	

Table A7: Functions used by *Apollo*, with inputs and outputs

Function	Arguments	Description	Output
apollo_attach	apollo_beta, apollo_inputs	Attaches parameters and data to allow users to refer to individual variables by name without reference to the object they are contained in.	None
apollo_avgInterDraws	P, apollo_inputs, functionality	Averages individual-specific likelihood across inter-individual draws.	Likelihood values (vector, one element per individual)
apollo_avgIntraDraws	P, apollo_inputs, functionality	Averages observation-specific likelihood across intra-individual draws.	Likelihood values (vector or matrix, one row per observation)
apollo_bootstrap	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, estimate_settings, bootstrap_settings	Randomly samples individuals with replacement from the database, and estimates the model in each sample	Files with parameters, samples matrix and covariance matrix
apollo_choiceAnalysis	choiceAnalysis_settings, apollo_control, database	Compares market shares across subsamples in dataset, and writes results to a file.	None (only written to file)
apollo_cnl	cnl_settings, functionality	Calculates probabilities of a cross nested logit model.	Likelihood values (vector, matrix or 3-dim array)
apollo_combineModels	P, apollo_inputs, functionality	Calculates the combined likelihood from several model components.	Likelihood values (vector, matrix or 3-dim array)
apollo_combineResults	combineResults_settings	Writes results from a series of models into a single CSV file.	None (only written to file)
apollo_conditionals	model, apollo_probabilities, apollo_inputs	Computes posterior distributions from continuous mixture models, and reports conditional means and standard deviations at the person level for each random coefficient.	A list of matrices, one per random coefficient.
apollo_deltaMethod	model, deltaMethod_settings	Calculates the standard errors of transformations of parameters.	None (only printed to screen)
apollo_detach	apollo_beta, apollo_inputs	Detaches variables attached by apollo_attach	None
apollo_dft	dft_settings, functionality	Calculate probabilities of a Decision Field Theory (DFT) with external thresholds.	Likelihood values (vector, matrix or 3-dim array)
apollo_el	el_settings, functionality	Calculate probabilities of an exploded logit model.	Likelihood values (vector, matrix or 3-dim array)

apollo_estimate	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, estimate_settings	Estimates parameters for a model by maximising the log-likelihood.	Model object
apollo_firstRow	P, apollo_inputs	Extracts the values from the input object (typically probabilities) for the first observation for each individual.	Same format as input, but with reduced rows.
apollo_fitsTest	model, apollo_probabilities, apollo_inputs, fitsTest_settings	Compares the fit across subsamples in the data for a given model.	None (only printed to screen)
apollo_initialise	none	Prepares global (i.e. user's) environment for a new model estimation.	None
apollo_lc	lc_settings, apollo_inputs, functionality	Calculates likelihood of latent class model as the sum across class of within clas probabilities and class allocation probabilities.	Likelihood values (vector, matrix or 3-dim array)
apollo_lcConditionals	model, apollo_probabilities, apollo_inputs	Calculates the posterior expected values (conditionals) of the class allocation probability for each individual.	A matrix with the posterior class allocation probabilites for each individual.
apollo_lcPars	apollo_beta, apollo_inputs	User-defined function used for latent class models, which receives two arguments: apollo_beta and apollo_inputs.	List of: allocation probabilities (named "pi_values"), and one list per parameter (each with as many elements as classes).
apollo_lcUnconditionals	model, apollo_probabilities, apollo_inputs	Returns unconditionals for latent class parameters in model, including interactions with deterministic covariates	List of vectors, with the values of random coefficients for each draw.
apollo_llCalc	apollo_beta, apollo_probabilities, apollo_inputs	Calculates the likelihood of each model component as well as for the whole model.	A list of likelihood values (vector)
apollo_loadModel	modelName	Loads to memory an estimated model object from a file in the current working directory.	Model object
apollo_lrTest	baseModel, generalModel	Calculates the likelihood ratio test and prints result.	None (only printed to screen)
apollo_mdcev	mdcev_settings, functionality	Calculates the likelihood of a Multiple Discrete Continuous Extreme Value (MDCEV)	Likelihood values (vector, matrix or 3-dim array)
apollo_mdcnev	mdcnev_settings, functionality	Calculates the likelihood of a Multiple Discrete Continuous Nested Extreme Value (MDCEV) model with an outside good.	Likelihood values (vector, matrix or 3-dim array)

apollo_mnl	mnl_settings, functionality	Calculates probabilities of a multinomial logit model.	Likelihood values (vector, matrix or 3-dim array)
apollo_modelOutput	model, modelOutput_settings	Prints estimation results to console. Amount of information presented can be adjusted through arguments.	None (only printed to screen)
apollo_nl	nl_settings, functionality	Calculates probabilities of a nested logit model.	Likelihood values (vector, matrix or 3-dim array)
apollo_normalDensity	normalDensity_settings, functionality	Calculates the density from a Normal distribution at a specific value with a specified mean and standard deviation.	Likelihood values (vector, matrix or 3-dim array)
apollo_ol	ol_settings, functionality	Calculates probabilities of an ordered logit model.	Likelihood values (vector, matrix or 3-dim array)
apollo_outOfSample	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, estimate_settings, outOfSample_settings	Generates estimation and validation samples, estimates the model on the first and calculates the likelihood for the second, then repeats.	Files with parameters and fits and samples matrix
apollo_panelProd	P, apollo_inputs, functionality	Calculates liklelihood of sequence of observations for each individual.	Likelihood values (vector or matrix)
apollo_prediction	model, apollo_probabilities, apollo_inputs, modelComponent	Makes model predictions using the model specified by the user.	List of forecasted values, one element per alternative.
apollo_prepareProb	P, apollo_inputs, functionality	Prepares the output from the user specified probabilities function	Likelihood values (vector)
apollo_probabilities	apollo_beta, apollo_inputs, functionality	User defined function determining the model likelihood. It must receive three arguments: apollo_beta, apollo_inputs, and functionality.	Likelihood of the model (or log of it if workInLogs is TRUE) (vector)
apollo_randCoeff	apollo_beta, apollo_inputs	User defined function used for mixture models, which receives two arguments: apollo_beta and apollo_inputs.	List of random parameters.
apollo_readBeta	apollo_beta, apollo_fixed, inputModelName, overwriteFixed	Retrieves values of parameters with matching names from a previously estimated model.	Named numeric vector of parameters.
apollo_saveOutput	model, saveOutput_settings	Writes estimation results into various output files.	None (only written to file)

apollo_searchStart	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, searchStart_settings	Given a set of starting values and a range for them, searches for points with a better starting likelihood.	Named numeric vector of parameters.
apollo_sharesTest	model, apollo_probabilities, apollo_inputs, sharesTest_settings	Prints tables comparing the shares predicted by the model with the shares observed in the data, across subsamples.	None (only printed to screen)
apollo_speedTest	apollo_beta, apollo_fixed, apollo_probabilities, apollo_inputs, speedTest_settings	Gives an estimate of the computation time as a function of the number of cores and number of draws.	None (only printed to screen)
apollo_unconditionals	model, apollo_probabilities, apollo_inputs	Returns draws (unconditionals) for random parameters in model, including interactions with deterministic covariates	List of vectors, with the values of random coefficients for each draw.
apollo_validateInputs	none	Searches the user work space for all necessary input to run \codeapollo_estimate, and packs it in a single list.	List grouping several required input for model estimation.
apollo_weighting	P, apollo_inputs, functionality	Applies weights to probabilities	Likelihood values (vector, one element per individual)

Table A8: Lists used by *Apollo*

Name of list	Contents	Description
apollo_control	modelName, modelDescr, indivId, mixing, nCores, workInLogs, seed, HB, noValidation, noDiagnostics, weights	Settings for apollo_control
apollo_draws	inter_drawsType, inter_nDraws, inter_unifDraws, inter_normDraws, intra_drawsType, intra_nDraws, intra_unifDraws, intra_normDraws	Settings for apollo_draws
apollo_HB	hb_dist, other parameters described in the RSGHB package	Settings for apollo_HB
apollo_inputs	List	List grouping most common inputs. Created by function apollo_validateInput.
bootstrap_settings	nRep, samples	Settings for apollo_bootstrap
choiceAnalysis_settings	alternatives, avail, choiceVar, explanators	Settings for apollo_choiceAnalysis
cnl_settings	alternatives, avail, choiceVar, V, cnlNests, cnlStructure, rows	Settings for apollo_cnl
combineResults_settings	printClassical, printPVal, printT1, estimateDigits, tDigits, pDigits	Settings for apollo_combineResults
deltaMethod_settings	operation, par_1, par_2, mult_1, mult_2	Settings for apollo_deltaMethod
dft_settings	alternatives, avail, choiceVar, attrValues, altStart, attrWeights, attrScalings, procPars, rows	Settings for apollo_dft
el_settings	alternatives, avail, choiceVars, V, scales, rows	Settings for apollo_el
estimate_settings	estimationRoutine, maxIterations, writeIter, hessianProc, printLevel, numDeriv_settings, silent, constraints, scaling	Settings for apollo_estimate
fitsTest_settings	fits, categories	Settings for apollo_fitsTest
lc_settings	inClassProb, classProb	Settings for apollo_lc
mdcev_settings	V, alternatives, alpha, gamma, sigma, cost, avail, continuous_choice, budget, min_consumption, rows	Settings for apollo_mdcev
mdcnev_settings	V, alternatives, alpha, gamma, sigma, mdcnevNests, mdcnevStructure, cost, avail, continuous_choice, budget, min_consumption, rows	Settings for apollo_mdcnev
mnl_settings	alternatives, avail, choiceVar, V, rows	Settings for apollo_mnl
modelOutput_settings	printClassical, printPVal, printT1, printDiagnostics, printCovar, printCorr, printOutliers, printChange	Settings for apollo_modelOutput
nl_settings	alternatives, avail, choiceVar, V, nlNests, nlSTructure, rows	Settings for apollo_nl
normalDensity_settings	Y, X, mu, sigma	Settings for apollo_normalDensity
ol_settings	Y, V, tau, coding, rows	Settings for apollo_ol

outOfSample_settings	nRep, validationSize, samples	Settings for apollo_outOfSample
procPars	error_sd, timesteps, phi1, phi2	List containing the four DFT 'process parameters'
saveOutput_settings	printClassical, printPVal, printT1, printDiagnostics, printCovar, printCorr, printOutliers, printChange, saveEst, saveCorr, saveModelObject, writeF12	Settings for apollo_saveOutput
searchStart_settings	nCandidates, smartStart, apollo_beta_min, apollo_beta_max, maxStages, dtest, gtest, Ltest, bfgsIter	Settings for apollo_searchStart
sharesTest_settings	true_shares, predicted_shares, subsamples	Settings for apollo_sharesTest
speedTest_settings	nDrawsTry, nCoresTry, nRep	Settings for apollo_speedTest

Table A9: Elements in lists and functions used by *Apollo*

Element	Type	Description
alpha	Named list of numeric vectors	Alpha parameters of MDC(N)EV associated to each alternative, including for the outside good. As many elements as alternatives.
alternatives	[MNL, NL, CNL] Named numeric vector	Names of alternatives and their corresponding value in choiceVar. Should have as many elements as V.
alternatives	[MDCEV, MDCNEV] Character vector	Names of alternatives, elements must match the names in list 'V'. If using an outside good, it must include "outside".
altStart	Named list of numeric vectors	As many elements as alternatives. Each elment can be a scalar or vector containing the starting preference value for the alternative.
apollo_beta	Named numeric vector	Names and starting values for parameters.
apollo_fixed	Character vector	Names of parameters whose values are to remain fixed throughout estimation.
apolloBetaMax	Numeric vector	Minimum possible value of parameters when generating candidates. Ignored if smartStart is TRUE. Default is apollo_beta - 0.1.
apolloBetaMin	Numeric vector	Maximum possible value of parameters when generating candidates. Ignored if smartStart is TRUE. Default is apollo_beta + 0.1.
attrScalings	Named list of vectors, matrices or 3-dim arrays	A named list with as many elements as attributes, or fewer. Each element is a factor scaling the attribute value. attrWeights and attrScalings should not be both defined for an attribute. Default is 1 for all attributes.
attrValues	Named list of lists	As many elements as alternatives. Each sub-list contains the alternative attributes for each observation (usually a column from the database). All alternatives must have the same attributes (can be set to zero if not relevant).
attrWeights	Named list of vectors, matrices or 3-dim arrays	As many elements as attributes, or fewer. Each element is the weight of the attribute. They should add up to one for each observation and draw (if present), and will be re-scaled if they do not. attrWeights and attrScalings should not be both defined for an attribute. Default is 1 for all attributes.
avail	Named list of numeric vectors or scalars	Availabilities of alternatives, one element per alternative. Names of elements must match those in alternatives. Values can be a vector of 0 or 1.
base_model	Character	Name of a previously estimated model. This is the restricted model, i.e. the one with fewer parameters.
bfgsIter	Numeric scalar	Number od BFGS iterations to perform at each stage to each remaining candidate. Default is 1.
budget	Numeric vector	Budget for each observation.
choiceVar	Numeric vector	Contains choices for all observations. It will usually be a column from the database. Values are defined in alternatives.

choiceVars	List of numeric vectors	Contain choices for each position of the ranking. The list must be ordered with the best choice first, second best second, etc. It will usually be a list of columns from the database.
classProb	List of probabilities	Allocation probability of each class. One element per class, in the same order than inClassProb.
cnlNests	List of numeric scalars or vectors	Lambda parameters for each nest. Elements must be named with the nest name. The lambda at the root is fixed to 1, and therefore must be no be defined.
cnlStructure	Numeric matrix	One row per nest and one column per alternative. Each element of the matrix is the alpha parameter of each (nest, alternative) pair.
coding	Numeric or character vector	Optional argument. Defines the order of the levels in outcomeOrdered. The first value is associated with the lowest level of V, and the last one with the highest value. If not provided, is assumed to be 1, 2, ... (length(tau) + 1).
constraints	List	Optional argument. List of constraints for `maxLik`.
continuous_choice	Named list of numeric vectors	Amount of consumption of each alternative. One element per alternative, as long as the number of observations (or a scalar). Names must match those in alternatives.
cost	Named list of numeric vectors	Price of each alternative. One element per alternative, each one as long as the number of observations (or a scalar). Names must match those in alternatives.
database	data.frame	Data used by model.
dTest	Numeric scalar	Tolerance for test 1. A candidate is discarded if its distance in parameter space to a better one is smaller than \codedTest. Default is 1.
error_sd	Numeric scalar or vector	The standard deviation of the the error term in each timestep.
estimateDigits	Numeric scalar	Number of decimal places to print on estimate values. Default is four.
estimationRoutine	Character	Estimation algorithm. Can take values "bfgs" (recommended), "bhhh", or "nr". Used only if apollo_control$HB is FALSE. Default is "bfgs".
explanators	data.frame	Variables determining subsamples of the database. Values in each column must describe a group or groups of individuals (e.g. socio-demographics). Most usually a subset of columns from database.
fits	Numeric vector	Predicted probability for chosen alternative for each observation.
functionality	Character	Description of the desired output from apollo_probabilities. Can take the values: "estimate", "prediction", "validate", "zero_LL", "conditionals", "output", "raw".
gamma	Named list of numeric vectors	Gamma parameters associated to each alternative. As many elements as alternatives.
general_model	Character or model object	Name or model object of a previously estimated model. This model should nests base_model, meaning it should more parameters than it.

gTest	Numeric scalar	Tolerance for test 2. A candidate is discarded if the norm of its gradient is smaller than gTest AND its LL is further than llTest from a better candidate. Default is 10^(-3).
HB	Boolean	TRUE to estimate model using Hierarchical Bayes techniques (using the RSGHB package).
hbDist	Vector	Contains numbers indicating the distributional assumptions of each parameter (in the same order as apollo_beta).
hessianRoutine	Character	Name of routine used to calculate the Hessian of the loglikelihood function. Valid values are "numDeriv" (default) and "maxLik" to use the routines in those packages. Only used if apollo_control$HB=FALSE.
inClassProb	List of probabilities	Conditional likelihood of each class. One element per class, in the same order than classProb.
indivId	Character	Name of the variable in database containing the identification of individuals.
input_modelName	Character	Name of model whose estimated parameters are to be loaded.
interDrawsType	Character	Type of inter-individual draws ('halton','mlhs','pmc','sobol','sobolOwen','sobolFaureTezuka', 'sobolOwenFaureTezuka' or the name of an object loaded in memory).
interNDraws	Numeric	Number of inter-individual draws per individual. Should be set to 0 if not using them.
interNormDraws	Character vector	Names of normalyl distributed inter-individual draws.
interUnifDraws	Character vector	Names of uniformly distributed inter-individual draws.
intraDrawsType	Character	Type of intra-individual draws ('halton','mlhs','pmc','sobol','sobolOwen','sobolFaureTezuka', 'sobolOwenFaureTezuka' or the name of an object loaded in memory).
intraNDraws	Numeric	Number of intra-individual draws per individual. Should be set to 0 if not using them.
intraNormDraws	Character vector	Names of normalyl distributed intra-individual draws.
intraUnifDraws	Character vector	Names of uniformly distributed intra-individual draws.
llTest	Numeric scalar	Tolerance for test 2. A candidate is discarded if the norm of its gradient is smaller than gTest AND its LL is further than llTest from a better candidate. Default is 3.
maxIterations	Numeric scalar	Maximum number of iterations of the estimation routine before stopping. Used only if apollo_control$HB is FALSE. Default is 200.
maxStages	Numeric scalar	Maximum number of search stages. The algorithm will stop when there is only one candidate left, or if it reaches this number of stages. Default is 5.
mdcnevNests	List of numeric scalars or vectors	Lambda parameters for each nest. Elements must be named with the nest name. The lambda at the root is fixed to 1, and therefore must be no be defined.

mdcnevStructure	Numeric matrix	One row per nest and one column per alternative. Each element of the matrix is 1 if an alternative belongs to the corresponding nest.
minConsumption	Named list of scalars or numeric vectors	Minimum consumption of the alternatives, if consumed. As many elements as alternatives. Names must match those in alternatives.
mixing	Boolean	Should be TRUE for models with mixing, and FALSE otherwise.
model	Model object	Estimated model object as returned by function apollo_estimate.
model_component	Character	Name of component of apollo_probabilities output to calculate predictions for. Default is "model", i.e. the whole model.
modelDescr	Character	Description of the model.
modelName	Character	Name of the model.
mu	Numeric scalar	Intercept of the linear model.
multPar1	Numeric scalar	A value to scale parName1
multPar2	Numeric scalar	A value to scale parName2
nCandidates	Numeric scalar	Number of candidate sets of parameters to be used at the start. Default is 100.
nCores	Numeric	Number of threads (processors) to use in estimation of the model.
nCoresTry	Numeric vector	Number of threads to try. Default is from 1 to the detected number of cores.
nDrawsTry	Numeric vector	Number of inter and intra-person draws to try. Default value is c(50, 100, 200)
nlNests	List of numeric scalars or vectors	Lambda parameters for each nest. Elements must be named with the nest name. The lambda at the root is fixed to 1 if excluded (recommended).
nlStructure	Named list of character vectors	As many elements as nests, it must include the "root". Each element contains the names of the nests or alternatives that belong to that nest. Element names must match those in \codenlNests.
noDiagnostics	Boolean	TRUE to avoid printing of diagnostics data during estimation. Default is FALSE.
noValidation	Boolean	TRUE to avoid input validation by model functions (apollo_mnl, apollo_mdcev, etc.) during estimation. Default is FALSE.
nRep	Numeric scalar	Number of times the procedure is repeated.
numDeriv_settings	List	Additional arguments to the Richardson method used by numDeriv to calculate the Hessian. See argument method.args in numDeriv::grad for more details.
operation	Character	Function to calculate the delta method for. Valid values are "sum", "diff", "ratio", "exp", "logistic", "lognormal".
outcomeNormal	Numeric vector	Dependant variable of linear model.
outcomeOrdered	Numeric or Character vector	Dependant variable of ordered logit.
overwrite_fixed	Boolean	TRUE if values for fixed parameters should also be read from input file. Default is FALSE.

P	List of vectors, matrices or 3-dim arrays	Likelihood of the model components.
parName1	Character	Name of the first parameter.
parName2	Character	Name of the second parameter. Optional depending on operation.
pDigits	Numeric scalar	Number of decimal places to print on p-values. Default is two.
phi1	Numeric scalar or vector	Sensitivity
phi2	Numeric scalar or vector	Process parameter
predicted_shares	Named list of numeric vectors	Predicted probability of each alternative. Can be calculated using apollo_prediction. Names must match alternatives.
printChange	Boolean	TRUE (default) for printing difference between starting values and estimates.
printClassical	Boolean	TRUE (default) for printing classical standard errors.
printCorr	Boolean	TRUE (default) for printing parameters correlation matrix. If printClassical=TRUE, both classical and robust matrices are printed.
printCovar	Boolean	TRUE (default) for printing parameters covariance matrix. If printClassical=TRUE, both classical and robust matrices are printed.
printDiagnostics	Boolean	TRUE (default) for printing summary of choices in database and other diagnostics.
printLevel	Numeric scalar	Higher values render more verbous outputs. Can take values 0, 1, 2 or 3. Ignored if apollo_control$HB is TRUE. Default is 3.
printOutliers	Boolean	TRUE (default) for printing 20 individuals with worst average fit across observations.
printPVal	Boolean	TRUE for printing p-values. FALSE by default.
printT1	Boolean	If TRUE, t-test for H0: apollo_beta=1 are printed. FALSE by default.
rows	Boolean vector	Consideration of rows in the likelihood calculation, FALSE to exclude. Length equal to the number of observations (nObs). Default is \code"all", equivalent for rep(TRUE, nObs)
samples	Matrix or data.frame	Indicates which observation is included in the estimation/validation sample in each run, or in the case of bootstrap, the number of times it is used
saveCorr	Boolean	TRUE (default) for saving estimated correlation matrix to a CSV file.
saveEst	Boolean	TRUE (default) for saving estimated parameters and standard errors to a CSV file.
saveModelObject	Boolean	TRUE (default) to save the R model object to a file (use apollo_loadModel to load it to memory)
scales	List of numeric vectors	Scale factors of each logit model. Should have one element less than choiceVars. At least one element should be normalized to 1. If omitted, scale=1 for all positions is assumed.
scalings	Named numeric vector	Scalings to be applied to individual model parameters or posterior chains during estimation.

seed	Numeric	Seed for random number generation.
sigma	Numeric scalar	Scale parameter of the model extreme value type I error.
silent	Boolean	If TRUE, no information is printed to the console by the function. Default is FALSE.
smartStart	Boolean	If TRUE, candidates are randomly generated with more chances in the directions the Hessian indicates improvement of the LL function. Default is FALSE.
subsamples	Named list of boolean vectors	Each element of the list defines a subsample (e.g. sociodemographics) for each observation.
tau	Numeric vector	Thresholds. As many as number of different levels in the dependent variable - 1. Extreme thresholds are fixed at -inf and +inf. No mixing allowed in thresholds.
tDigits	Numeric scalar	Number of decimal places to print on t-ratios values. Default is two.
timesteps	Numeric scalar or vector	Number of timesteps to consider. Should be an integer bigger than 0.
V	[MNL, NL, CNL, MDCECV, MDCNEV] List of deterministic utilities	Utilities of the alternatives. Names of elements must match those in 'alternatives'.
V	[OL] Deterministic utility	Deterministic part of the utility of the ordered logit.
validationSize	Numeric scalar	Size of the validation sample. Can be a percentage of the sample (0-1) or the number of individuals in the validation sample (>1). Default is 0.1.
weights	Numeric vector	Vector of weights, of length equal to number of rows in the data.
workInLogs	Boolean	TRUE for higher numeric stability during estimation, at the expense of computational time. Mostly useful for panel models.
writeF12	Boolean	TRUE for writing results into an F12 file (ALOGIT format). Default is FALSE.
writeIter	Boolean	Writes value of the parameter on each iteration on a CSV file. Works only if estimation_routine="bfgs". Default is TRUE.
xNormal	Numeric vector	Deterministic part of the linear model.

References

Abou-Zeid, M., Ben-Akiva, M., 2014. Hybrid choice models. In: Hess, S., Daly, A. (Eds.), Handbook of Choice Modelling. Edward Elgar.

ALogit, 2016. ALOGIT 4.3. ALOGIT Software & Analysis Ltd.
URL www.alogit.com

Axhausen, K. W., Hess, S., König, A., Abay, G., Bates, J. J., Bierlaire, M., 2008. State of the art estimates of the swiss value of travel time savings. Transport Policy 15 (3), 173–185.

Bhat, C., 1997. An endogenous segmentation mode choice model with an application to intercity travel. Transportation Science 31 (1), 34–48.

Bhat, C. R., 2008. The multiple discrete-continuous extreme value (mdcev) model: role of utility function parameters, identification considerations, and model extensions. Transportation Research Part B: Methodological 42 (3), 274–303.

Bierlaire, M., 2003. BIOGEME: a free package for the estimation of discrete choice models. Proceedings of the 3^{rd} Swiss Transport Research Conference, Monte Verità, Ascona.

Bierlaire, M., Thémans, M., Zufferey, N., 2010. A heuristic for nonlinear global optimization. INFORMS Journal on Computing 22 (1), 59–70.

Bradley, M. A., Daly, A., 1996. Estimation of logit choice models using mixed stated-preference and revealed-preference information. In: Stopher, P. R., Lee-Gosselin, M. (Eds.), Understanding Travel Behaviour in an Era of Change. Elsevier, Oxford, Ch. 9, pp. 209–231.

Busemeyer, J. R., Townsend, J. T., 1992. Fundamental derivations from decision field theory. Mathematical Social Sciences 23 (3), 255–282.

Busemeyer, J. R., Townsend, J. T., 1993. Decision field theory: a dynamic-cognitive approach to decision making in an uncertain environment. Psychological Review 100 (3), 432.

Calastri, C., Crastes dit Sourd, R., Hess, S., 2019. We want it all: experiences from a survey seeking to capture social network structures, lifetime events and short-term travel activity planning. Transportation forthcoming.

Calastri, C., Hess, S., Daly, A., Carrasco, J. A., 2017. Does the social context help with understanding and predicting the choice of activity type and duration? an application of the multiple discrete-continuous nested extreme value model to activity diary data. Transportation Research Part A: Policy and Practice 104, 1–20.

Chorus, C., 2010. A new model of random regret minimization. European Journal of Transport and Infrastructure Research 10 (2), 181–196.

Daly, A., 1987. Estimating Tree Logit models. Transportation Research Part B 21 (4), 251–267.

Daly, A., Hess, S., de Jong, G., 2012a. Calculating errors for measures derived from choice modelling estimates. Transportation Research Part B 46 (2), 333–341.

Daly, A., Zachary, S., 1978. Improved multiple choice models. In: Hensher, D. A., Dalvi, Q. (Eds.), Identifying and Measuring the Determinants of Mode Choice. Teakfields, London.

Daly, A. J., Hess, S., Patruni, B., Potoglou, D., Rohr, C., 2012b. Using ordered attitudinal indicators in a latent variable choice model: A study of the impact of security on rail travel behaviour. Transportation 39 (2), 267–297.

Doornik, J. A., 2001. Ox: An Object-Oriented Matrix Language. Timberlake Consultants Press, London.

Dumont, J., Keller, J., 2019. RSGHB: Functions for Hierarchical Bayesian Estimation: A Flexible Approach. R package version 1.2.1.
URL https://CRAN.R-project.org/package=RSGHB

Faure, H., Tezuka, S., 2000. Another random scrambling of digital (t,s)-sequences. In: Fang, K.-T., Hickernell, F. J., Niederreiter, H. (Eds.), Monte Carlo and Quasi-Monte Carlo Methods. Springer.

Fosgerau, M., Mabit, S. L., 2013. Easy and flexible mixture distributions. Economics Letters 120 (2), 206 – 210.

Geweke, J., 1992. Evaluating the accuracy of sampling-based approaches to the calculations of posterior moments. Bayesian statistics 4, 641–649.

Giergiczny, M., Dekker, T., Hess, S., Chintakayala, P., September 2017. Testing the stability of utility parameters in repeated best, repeated best-worst and one-off best-worst studies. European Journal of Transport and Infrastructure Research 17 (4), 457–476, © 2017, Author(s). Reproduced in accordance with the publisher's self-archiving policy.
URL http://eprints.whiterose.ac.uk/118496/

Gilbert, P., Varadhan, R., 2016. numDeriv: Accurate Numerical Derivatives. R package version 2016.8-1.
URL https://CRAN.R-project.org/package=numDeriv

Greene, W. H., Hensher, D. A., 2013. Revealing additional dimensions of preference heterogeneity in a latent class mixed multinomial logit model. Applied Economics 45 (14), 1897–1902.

Halton, J., 1960. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. Numerische Mathematik 2 (1), 84–90.

Hancock, T. O., Hess, S., Choudhury, C. F., 2018. Decision field theory: Improvements to current methodology and comparisons with standard choice modelling techniques. Transportation Research Part B: Methodological 107, 18–40.

Hancock, T. O., Hess, S., Choudhury, C. F., 2019. An accumulation of preference: two alternative dynamic models for understanding transport choices. Submitted.

Henningsen, A., Toomet, O., 2011. maxlik: A package for maximum likelihood estimation in R. Computational Statistics 26 (3), 443–458.
URL http://dx.doi.org/10.1007/s00180-010-0217-1

Hensher, D. A., Louviere, J. J., Swait, J., 1998. Combining sources of preference data. Journal of Econometrics 89 (1-2), 197–221.

Hess, S., 2005. Advanced discrete choice models with applications to transport demand. Ph.D. thesis, Centre for Transport Studies, Imperial College London.

Hess, S., 2014. 14 latent class structures: taste heterogeneity and beyond. In: Handbook of choice modelling. Edward Elgar Publishing Cheltenham, pp. 311–329.

Hess, S., Bierlaire, M., Polak, J. W., 2007. A systematic comparison of continuous and discrete mixture models. European Transport 36, 35–61.

Hess, S., Daly, A., 2014. Handbook of Choice Modelling. Edward Elgar publishers, Cheltenham.

Hess, S., Daly, A., Dekker, T., Cabral, M. O., Batley, R., 2017. A framework for capturing heterogeneity, heteroskedasticity, non-linearity, reference dependence and design artefacts in value of time research. Transportation Research Part B: Methodological 96, 126 – 149.

Hess, S., Train, K., 2011. Recovery of inter- and intra-personal heterogeneity using mixed logit models. Transportation Research Part B 45 (7), 973–990.

Hess, S., Train, K., Polak, J. W., 2006. On the use of a Modified Latin Hypercube Sampling (MLHS) method in the estimation of a Mixed Logit model for vehicle choice. Transportation Research Part B 40 (2), 147–163.

Hotaling, J. M., Busemeyer, J. R., Li, J., 2010. Theoretical developments in decision field theory: comment on Tsetsos, Usher, and Chater (2010). Psychological Review 117 (4), 1294–1298.

Huber, P., 1967. The behavior of maximum likelihood estimation under nonstandard conditions. In: LeCam, L., Neyman, J. (Eds.), Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability. University of California Press, pp. 221–233.

Koppelman, F. S., Wen, C.-H., 1998. Alternative Nested Logit Models: structure, properties and estimation. Transportation Research Part B 32 (5), 289–298.

Krinsky, I., Robb, A., 1986. On approximating the statistical properties of elasticities. Review of Economics and Statistics 68, 715–719.

Lancsar, E., Louviere, J., Donaldson, C., Currie, G., Burgess, L., 2013. Best worst discrete choice experiments in health: Methods and an application. Social Science & Medicine 76, 74 – 82.
URL http://www.sciencedirect.com/science/article/pii/S0277953612007290

Lenk, P., February 2014. Bayesian estimation of random utility models. In: Handbook of Choice Modelling. Chapters. Edward Elgar Publishing, Ch. 20, pp. 457–497.
URL https://ideas.repec.org/h/elg/eechap/14820_20.html

McFadden, D., 1974. Conditional logit analysis of qualitative choice behaviour. In: Zarembka, P. (Ed.), Frontiers in Econometrics. Academic Press, New York, pp. 105–142.

McFadden, D., 1978. Modelling the choice of residential location. In: Karlqvist, A., Lundqvist, L., Snickars, F., Weibull, J. W. (Eds.), Spatial Interaction Theory and Planning Models. North Holland, Amsterdam, Ch. 25, pp. 75–96.

McFadden, D., 2000. Economic Choices. Nobel Prize Lecture.
URL https://www.nobelprize.org/uploads/2018/06/mcfadden-lecture.pdf

Owen, A. B., 1995. Randomly permuted (t,m,s)-nets and (t,s)-sequences. In: Niederreiter, H., Shiue, J.-S. (Eds.), Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing. Springer, New York, pp. 351–368.

Palma, D., 2016. Modelling wine consumer preferences using hybrid choice models: inclusion of intrinsic and extrinsic attributes. Ph.D. thesis, School of Engineering, Pontificia Universidad Católica de Chile.

Pinjari, A. R., Bhat, C., 2010a. A multiple discrete–continuous nested extreme value (mdcnev) model: formulation and application to non-worker activity time-use and timing behavior on weekdays. Transportation Research Part B: Methodological 44 (4), 562–583.

Pinjari, A. R., Bhat, C. R., 2010b. An efficient forecasting procedure for kuhn-tucker consumer demand model systems: application to residential energy consumption analysis. Technical paper, Department of Civil and Environmental Engineering, University of South Florida, 263–285.

R Core Team, 2017. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria.
URL https://www.R-project.org/

Roe, R. M., Busemeyer, J. R., Townsend, J. T., 2001. Multialternative decision field theory: A dynamic connectionist model of decision making. Psychological Review 108 (2), 370.

RStudio Team, 2015. Rstudio: Integrated development for r. RStudio, Inc., Boston, MA URL http://www.rstudio.com/.

Sobol', I. M., 1967. On the distribution of points in a cube and the approximate evaluation of integrals. Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki 7 (4), 784–802.

Train, K., 2009. Discrete Choice Methods with Simulation, second edition Edition. Cambridge University Press, Cambridge, MA.

Train, K., Weeks, M., 2005. Discrete choice models in preference space and willingness-to-pay space. In: Scarpa, R., Alberini, A. (Eds.), Application of simulation methods in environmental and resource economics. Springer, Dordrecht, Ch. 1, pp. 1–16.

Vovsha, P., 1997. Application of a Cross-Nested Logit model to mode choice in Tel Aviv, Israel, Metropolitan Area. Transportation Research Record 1607, 6–15.

Weisberg, S., 2005. Applied Linear Regression, 3rd Edition. Wiley, Hoboken NJ.
URL http://www.stat.umn.edu/alr

Wen, C.-H., Koppelman, F. S., 2001. The Generalized Nested Logit Model. Transportation Research Part B 35 (7), 627–641.

Williams, H. C. W. L., 1977. On the Formulation of Travel Demand Models and Economic Evaluation Measures of User Benefit. Environment & Planning A 9 (3), 285–344.