The cbcTools Package: Tools for Designing and Testing Choice-Based Conjoint Surveys in R

John Paul Helveston, Ph.D.*

Abstract

Traditional tools for designing choice-based conjoint survey experiments focus on optimizing the design of experiment for statistical power under ideal conditions. But these tools rarely provide guidance on important design decisions for less ideal conditions, such as when preference heterogeneity may be expected in respondent choices or when strong interactions may be expected between certain attributes. The `cbcTools` R package was developed to provide researchers tools for creating and assessing experiment designs and sample size requirements under a variety of different conditions prior to fielding an experiment. The package contains functions for generating experiment designs and surveys as well as functions for simulating choice data and conducting power analyses. Since the package data format matches that of designs exported from Sawtooth Software, it should integrate into the Sawtooth workflow. Detailed package documentation can be found at https://jhelvy.github.io/cbcTools/.

Introduction

Designing a choice-based conjoint survey is almost never a simple, straightforward process. Designers must consider multiple trade offs between design parameters (e.g., which attributes and levels to include, how many choice questions to ask each respondent, and how many alternatives per choice question) and the design outcomes in terms of the user experience and the statistical power available for identifying effects. The process is typically highly iterative.

As a quick example, consider a simple conjoint experiment about cars with just two attributes with three levels each:

- **Price**: $20,000, 40,000, & 100,000
- **Brand**: GM, BMW, & Ferrari

A simple starting point is to generate a design by randomly choosing combinations of brands and prices from the full set of all possible profiles. Once created, one of the first things designers examine is the count of how often each level of each attribute is shown. The

*Engineering Management and Systems Engineering, George Washington University, Washington, D.C. USA table below shows an example of the counts from a random design with 9 choice sets of 3 alternatives per question:

Table 1: Individual and pairwise counts of attributes.

Price:		\$20,000	\$40,000	\$100,000
Brand		9	9	9
GM	10	3	0	7
BMW	11	4	5	2
Ferrari	6	2	4	0

Based on the counts alone, it is clear that this design has several problems. First, while the price levels are perfectly balanced (each level is shown 9 times), the brand levels are not – GW and BMW are shown 10 and 11 times, respectively, whereas Ferrari is only shown 6 times. And the pairwise counts are particularly troubling. The Ferrari brand is only shown with a price of \$20,000 and \$40,000 and never at the \$100,000 level (the most logical level for a Ferrari!). Likewise, GM brand is shown with a price of \$100,000 in 7 out of 10 times and never with a price of \$40,000.

This rather poor design is a common outcome when using a randomized design with a small number of choice sets. Simply increasing the number of choice sets often results in a much better balance. For example, if we increase the number from 9 to 90, we obtain the following counts:

Table 2: Individual and pairwise counts of attributes.

Price:		\$20,000	\$40,000	\$100,000
Brand		91	84	95
GM	92	31	31	30
BMW	80	25	25	30
Ferrari	98	35	30	35

This design has a much better balance across the attribute levels than the one created from just 9 choice sets. However, it too has issues. Consider, for example, that about 1/3 of the time the Ferrari brand is shown it has a price of just \$20,000. This is obviously an unrealistic profile, and if a user saw such a profile multiple times they may not take the choice exercise seriously.

One approach to try to correct this outcome is to use a Bayesian D-efficient design. This approach allows the designer to list a set of priors on the expected coefficients for each attribute level and then use an algorithm to select combinations of profiles that maximize the information available to identify main effects.

Using a Bayesian D-efficient design with 90 choice sets and the prior utilities in Table 3, the counts become much more reasonable (see Table 4). Now the individual attribute counts are

Table 3: Example conjoint attributes and levels.

Attribute	Level	Prior
Price	$20,000	0
	$40,000	-1
	$100,000	-4
Brand	GM	0
	BMW	1
	Ferrari	2

relatively balanced, and the pairwise combinations of attributes are much more plausible. For example, the GM brand is only shown at price levels of $20,000 and $40,000, and about half of the time Ferrari is shown at the $100,000 price level.

Table 4: Individual and pairwise counts of attributes.

Price:		$20,000	$40,000	$100,000
Brand		97	93	78
GM	93	52	41	0
BMW	90	30	30	30
Ferrari	86	15	22	49

Of course, even Bayesian D-efficient designs have their downsides. In particular, they can become problematic if there are significant interaction effects between any of the attributes. Figure 1 highlights this trade off. The plots show the standard errors of each coefficients from the same model estimated using increasing subsets of two simulated data sets: a randomized design and a Bayesian D-efficient design. In the first row, it is clear that the D-efficient design produces lower standard errors for the same sample size compared to the randomized design. However, the second row shows the standard errors on interaction effects, which are not identifiable using the D-efficient design. This is because interaction effects can become confounded with main effects in D-efficient designs. Thus, if interaction effects are expected, a randomized design may be a more appropriate choice.

The `cbcTools` Package

As the previous example illustrates, designing a choice-based conjoint experiment is an iterative process, and designers must consider a wide range of factors. The `cbcTools` package was designed as a tool for helping designers navigate this process and to understand the impacts different design decisions could have on the statistical power of the experiment prior to fielding the survey.

The package provides a set of functions (each starting with `cbc_`) for designing surveys and

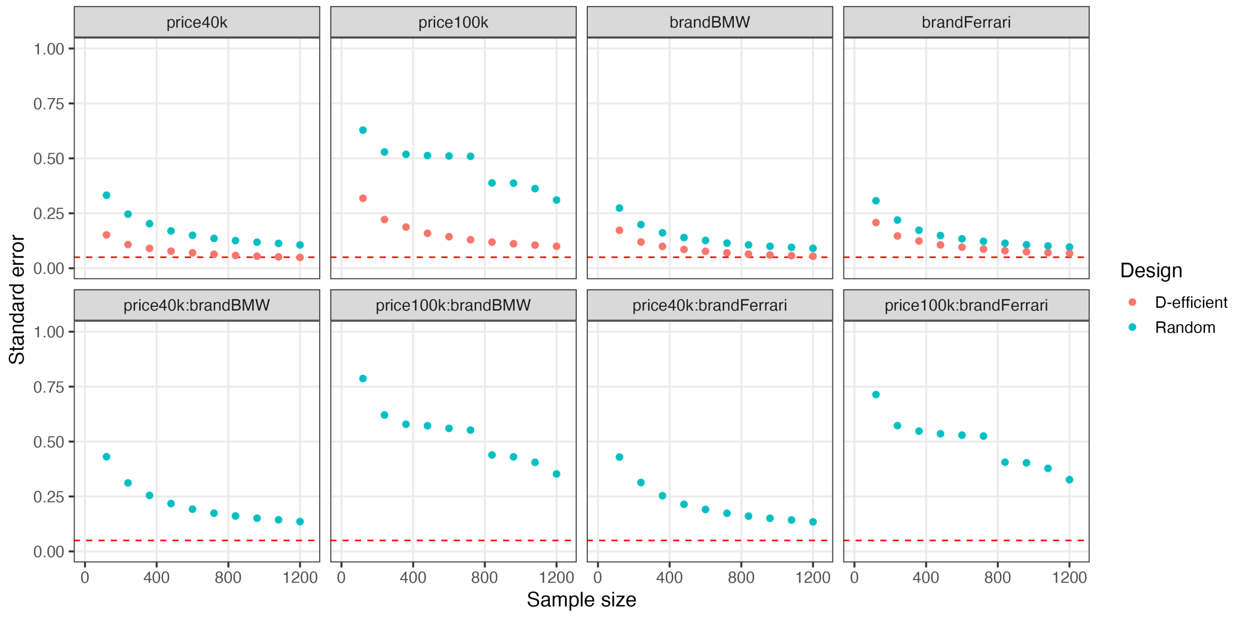

Figure 1: Standard errors with increasing sample size for a multinomial logit model with interaction effects estimated using a randomized design versus a Bayesian D-efficient design

conducting power analyses for choice-based conjoint survey experiments in R. The typical workflow involves a series of steps, and each step has a function associated with it (see Figure 2).

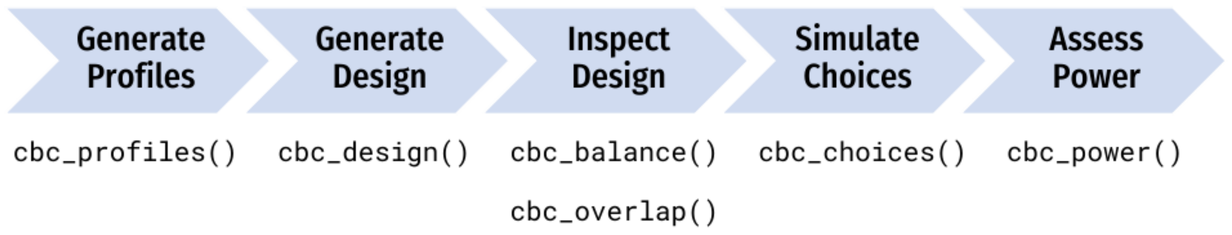

Figure 2: Diagram of the choice-based conjoint survey design process.

The rest of this paper explains each of these steps using a detailed example of a simple conjoint experiment about apples.

Generating profiles

The first step in designing an experiment is to define the attributes and levels for the experiment and then generate all of the `profiles` of each possible combination of those attributes and levels. For example, consider designing a conjoint experiment about apples with the following three attributes: `price`, `type`, and `freshness`. All of the possible profiles for these attributes can be obtained using the `cbc_profiles()` function:

```r
profiles <- cbc_profiles(
  price     = seq(1, 4, 0.5), # $ per pound
  type      = c('Fuji', 'Gala', 'Honeycrisp'),
  freshness = c('Poor', 'Average', 'Excellent')
)

nrow(profiles)
```

```
#> [1] 63
```

```r
head(profiles)
```

```
#>   profileID price type freshness
#> 1         1   1.0 Fuji      Poor
#> 2         2   1.5 Fuji      Poor
#> 3         3   2.0 Fuji      Poor
#> 4         4   2.5 Fuji      Poor
#> 5         5   3.0 Fuji      Poor
#> 6         6   3.5 Fuji      Poor
```

```r
tail(profiles)
```

```
#>    profileID price       type freshness
#> 58        58   1.5 Honeycrisp Excellent
#> 59        59   2.0 Honeycrisp Excellent
#> 60        60   2.5 Honeycrisp Excellent
#> 61        61   3.0 Honeycrisp Excellent
#> 62        62   3.5 Honeycrisp Excellent
#> 63        63   4.0 Honeycrisp Excellent
```

Depending on the context of the survey, some profiles may need to be eliminated (e.g., some profile combinations may be illogical or unrealistic).[1] To do so, each level of an attribute can be defined as a list defining those contraints. In the example below, the `type` attribute has constraints such that only certain price levels will be shown for each level. In addition, for the `"Honeycrisp"` level of the `type` attribute, only two of the three `freshness` levels are included: `"Excellent"` and `"Average"`. Note that both the other attributes (`price` and `freshness`) should contain all of the possible levels. With these constraints, only 30 profiles are available compared to 63 without constraints.

```r
profiles <- cbc_profiles(
  price = c(1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5),
  freshness = c('Poor', 'Average', 'Excellent'),
  type = list(
    "Fuji" = list(
```

[1]Note that including hard constraints in your designs can substantially reduce the statistical power of your design, so use them cautiously and avoid them if possible.

```
        price = c(2, 2.5, 3)
    ),
    "Gala" = list(
        price = c(1, 1.5, 2)
    ),
    "Honeycrisp" = list(
        price = c(2.5, 3, 3.5, 4, 4.5, 5),
        freshness = c("Average", "Excellent")
    )
  )
)

nrow(profiles)
```

```
#> [1] 30
```

```
head(profiles)
```

```
#>   profileID price freshness type
#> 1         1   2.0      Poor Fuji
#> 2         2   2.5      Poor Fuji
#> 3         3   3.0      Poor Fuji
#> 4         4   2.0   Average Fuji
#> 5         5   2.5   Average Fuji
#> 6         6   3.0   Average Fuji
```

```
tail(profiles)
```

```
#>    profileID price freshness       type
#> 25        25   2.5 Excellent Honeycrisp
#> 26        26   3.0 Excellent Honeycrisp
#> 27        27   3.5 Excellent Honeycrisp
#> 28        28   4.0 Excellent Honeycrisp
#> 29        29   4.5 Excellent Honeycrisp
#> 30        30   5.0 Excellent Honeycrisp
```

Generating designs

Randomized designs

Once a set of profiles is obtained, a randomized conjoint survey can then be generated using the `cbc_design()` function:

```
design <- cbc_design(
  profiles = profiles,
  n_resp   = 900, # Number of respondents
  n_alts   = 3,   # Number of alternatives per question
```

```r
  n_q      = 6    # Number of questions per respondent
)

dim(design)  # View dimensions
```

```
#> [1] 16200     8
```

```r
head(design) # Preview first 6 rows
```

```
#>   respID qID altID obsID profileID price       type freshness
#> 1      1   1     1     1        25   2.5       Fuji   Average
#> 2      1   1     2     1        56   4.0       Gala Excellent
#> 3      1   1     3     1        59   2.0 Honeycrisp Excellent
#> 4      1   2     1     2         4   2.5       Fuji      Poor
#> 5      1   2     2     2        52   2.0       Gala Excellent
#> 6      1   2     3     2        39   2.5 Honeycrisp   Average
```

For now, the `cbc_design()` function only generates a randomized design. Other packages, such as the `idefix` package, are able to generate other types of designs, including Bayesian D-efficient designs, and future versions of the `cbcTools` package will add a wrapper around the `idefix` package to integrate some of these features. The randomized design simply samples from the set of `profiles` while ensuring that no two profiles are the same in any choice question.

The structure of the resulting `design` data frame is such that each row specifies the levels shown for one alternative in a choice question. In addition to the attribute levels, the `design` data frame also includes the following variables for identifying different aspects of the survey:

- `respID`: Identifies each survey respondent.
- `qID`: Identifies the choice question answered by the respondent.
- `altID`:Identifies the alternative in any one choice observation.
- `obsID`: Identifies each unique choice observation across all respondents.
- `profileID`: Identifies the profile in `profiles`.

Labeled designs (a.k.a. "alternative-specific" designs)

A "labeled" design (also known as an "alternative-specific" design) can also be generated. In labeled designs, the levels of one attribute are used as the labels for each alternative. To do so, the `label` argument can be set to one of the attributes. This by definition also sets the number of alternatives in each question to the number of levels in the chosen attribute, so the `n_alts` argument is overridden. In the example below, the `type` attribute is used as the label:

```r
design_labeled <- cbc_design(
  profiles  = profiles,
  n_resp    = 900, # Number of respondents
  n_alts    = 3,   # Number of alternatives per question
  n_q       = 6,   # Number of questions per respondent
  label     = "type" # Set the "type" attribute as the label
)

dim(design_labeled)
```

```
#> [1] 16200     8
```

```
head(design_labeled)
```

```
#>   respID qID altID obsID profileID price       type freshness
#> 1      1   1     1     1        49   4.0       Fuji Excellent
#> 2      1   1     2     1        51   1.5       Gala Excellent
#> 3      1   1     3     1        59   2.0 Honeycrisp Excellent
#> 4      1   2     1     2         7   4.0       Fuji      Poor
#> 5      1   2     2     2        12   3.0       Gala      Poor
#> 6      1   2     3     2        36   1.0 Honeycrisp   Average
```

In the above example, the `type` attribute is now fixed to be the same order for every choice question, ensuring that each level in the `type` attribute will always be shown in each choice question.

Adding a "no choice" option (a.k.a. "outside good")

Often times designers may wish to allow respondents to opt out from choosing any of the alternatives shown in any one choice question. Such a "no choice" (or "outside good") option can be included by setting `no_choice = TRUE`. If included, all categorical attributes will be dummy-coded to appropriately dummy-code the "no choice" alternative.

```
design_nochoice <- cbc_design(
  profiles  = profiles,
  n_resp    = 900, # Number of respondents
  n_alts    = 3, # Number of alternatives per question
  n_q       = 6, # Number of questions per respondent
  no_choice = TRUE
)

dim(design_nochoice)
```

```
#> [1] 21600    13
```

```
head(design_nochoice)
```

```
#>   respID qID altID obsID profileID price type_Fuji type_Gala type_Honeycrisp
#> 1      1   1     1     1        43   1.0         1         0               0
#> 2      1   1     2     1        30   1.5         0         1               0
#> 3      1   1     3     1        58   1.5         0         0               1
#> 4      1  1    4    1         0  0.0         0         0              0
#> 5      1  2    1    2        13  3.5         0         1              0
#> 6      1  2    2    2        61  3.0         0         0              1
#>   freshness_Poor freshness_Average freshness_Excellent no_choice
#> 1              0                 0                   1         0
#> 2              0                 1                   0         0
#> 3              0                 0                   1         0
#> 4              0                 0                   0         1
#> 5              1                 0                   0         0
#> 6              0                 0                   1         0
```

Inspecting designs

The package includes some functions to quickly inspect some basic metrics of a design. The `cbc_balance()` function prints out a summary of the counts of each level for each attribute across all choice questions as well as the two-way counts across all pairs of attributes for a given design:

```
cbc_balance(design)
```

```
#> ==============================
#> price x type
#> 
#>          Fuji Gala Honeycrisp
#>       NA 5396 5429       5375
#> 1   2290  751  773        766
#> 1.5 2250  764  735        751
#> 2   2307  792  778        737
#> 2.5 2354  789  817        748
#> 3   2404  764  821        819
#> 3.5 2287  788  724        775
#> 4   2308  748  781        779
#> 
#> price x freshness
#> 
#>          Poor Average Excellent
#>       NA 5368    5465      5367
#> 1   2290  769     779       742
#> 1.5 2250  754     749       747
#> 2   2307  780     748       779
#> 2.5 2354  791     784       779
#> 3   2404  763     836       805
#> 3.5 2287  734     806       747
#> 4   2308  777     763       768
#> 
```

```
#> type x freshness
#>
#>                 Poor Average Excellent
#>              NA 5368    5465      5367
#> Fuji       5396 1808    1782      1806
#> Gala       5429 1762    1857      1810
#> Honeycrisp 5375 1798    1826      1751
```

Similarly, the `cbc_overlap()` function prints out a summary of the amount of "overlap" across attributes within the choice questions. For example, for each attribute, the count under `"1"` is the number of choice questions in which the same level was shown across all alternatives for that attribute (because there was only one level shown). Likewise, the count under `"2"` is the number of choice questions in which only two unique levels of that attribute were shown, and so on:

```
cbc_overlap(design)
```

```
#> ==============================
#> Counts of attribute overlap:
#> (# of questions with N unique levels)
#>
#> price:
#>
#>    1    2    3
#>   62 1864 3474
#>
#> type:
#>
#>    1    2    3
#>  560 3583 1257
#>
#> freshness:
#>
#>    1    2    3
#>  546 3650 1204
```

Simulating choices

Choices for a given `design` can be simulated using the `cbc_choices()` function. By default, random choices are simulated:

```
data <- cbc_choices(
  design = design,
  obsID  = "obsID"
)
```

```
head(data)
```

```
#>   respID qID altID obsID profileID price       type freshness choice
#> 1      1   1     1     1        25   2.5       Fuji   Average      0
#> 2      1   1     2     1        56   4.0       Gala Excellent      1
#> 3      1   1     3     1        59   2.0 Honeycrisp Excellent      0
#> 4      1   2     1     2         4   2.5       Fuji      Poor      0
#> 5      1   2     2     2        52   2.0       Gala Excellent      0
#> 6      1   2     3     2        39   2.5 Honeycrisp   Average      1
```

Choices can also be simulated according to an assumed prior model. The default model used is a multinomial logit model with fixed parameters. In the example below, the choices are simulated using a utility model with the following parameters:

- 1 continuous parameter for `price`
- 2 categorical parameters for `type` (`'Gala'` and `'Honeycrisp'`)
- 2 categorical parameters for `freshness` (`"Average"` and `"Excellent"`)

Note that for categorical variables (`type` and `freshness` in this example), the first level defined when using `cbc_profiles()` is set as the reference level. The example below defines the following utility model for simulating choices for each alternative j:

$$u_j = 0.1price_j + 0.1type_j^{\text{Gala}} + 0.2type_j^{\text{Honeycrisp}} + 0.1freshness_j^{\text{Average}} + 0.2freshness_j^{\text{Excellent}} + \varepsilon_j$$

```
data <- cbc_choices(
  design = design,
  obsID = "obsID",
  priors = list(
    price     = 0.1,
    type      = c(0.1, 0.2),
    freshness = c(0.1, 0.2)
  )
)
```

The prior model used for simulating choices can also include other, more complex models, such as models that include interaction terms or mixed logit models. For example, the example below is the same as the previous example but with an added interaction between `price` and `type`:

```
data <- cbc_choices(
  design = design,
  obsID = "obsID",
  priors = list(
    price = 0.1,
    type = c(0.1, 0.2),
```

```r
    freshness = c(0.1, 0.2),
    `price*type` = c(0.1, 0.5)
  )
)
```

To simulate choices according to a mixed logit model where parameters follow a normal or log-normal distribution across the population, the `randN()` and `randLN()` functions can be used inside the `priors` list. The example below models the `type` attribute with two random normal parameters using a vector of means (`mean`) and standard deviations (`sd`) for each level of `type`:

```r
data <- cbc_choices(
  design = design,
  obsID = "obsID",
  priors = list(
    price = 0.1,
    type = randN(mean = c(0.1, 0.2), sd = c(1, 2)),
    freshness = c(0.1, 0.2)
  )
)
```

Analyzing power

The simulated choice data can be used to conduct a power analysis by estimating the same model multiple times with incrementally increasing sample sizes. As the sample size increases, the estimated coefficient standard errors will decrease (i.e. coefficient estimates become more precise), allowing the designer to identify the sample size required to achieve a desired level of precision.

The `cbc_power()` function achieves this by partitioning the choice data into multiple sizes (defined by the `nbreaks` argument) and then estimating a user-defined choice model on each data subset. In the example below, 10 different sample sizes are used. All models are estimated using the `logitr` package, and any arguments for estimating models with the `logitr` package can be passed through the `cbc_power()` function.

```r
power <- cbc_power(
  data    = data,
  pars    = c("price", "type", "freshness"),
  outcome = "choice",
  obsID   = "obsID",
  nbreaks = 10,
  n_q     = 6
)

head(power)
```

```
#>   sampleSize                coef         est         se
#> 1         90               price -0.04285506 0.05172937
#> 2         90            typeGala  0.02308611 0.13081166
#> 3         90      typeHoneycrisp  0.09320432 0.12773123
#> 4         90    freshnessAverage  0.02004300 0.13193111
#> 5         90 freshnessExcellent  0.14198531 0.13105651
#> 6        180               price -0.04361896 0.03689761
```

```
tail(power)
```

```
#>    sampleSize                coef          est         se
#> 45        810 freshnessExcellent -0.076357960 0.04303338
#> 46        900              price -0.012316440 0.01654186
#> 47        900           typeGala -0.009044929 0.04071941
#> 48        900     typeHoneycrisp  0.061467662 0.04041809
#> 49        900   freshnessAverage -0.080280718 0.04075264
#> 50        900 freshnessExcellent -0.060804549 0.04063969
```

The `power` data frame contains the coefficient estimates and standard errors for each sample size. The standard errors can be quickly visualized to identify a required sample size for a desired level of parameter precision by using the `plot()` method:

```
plot(power)
```

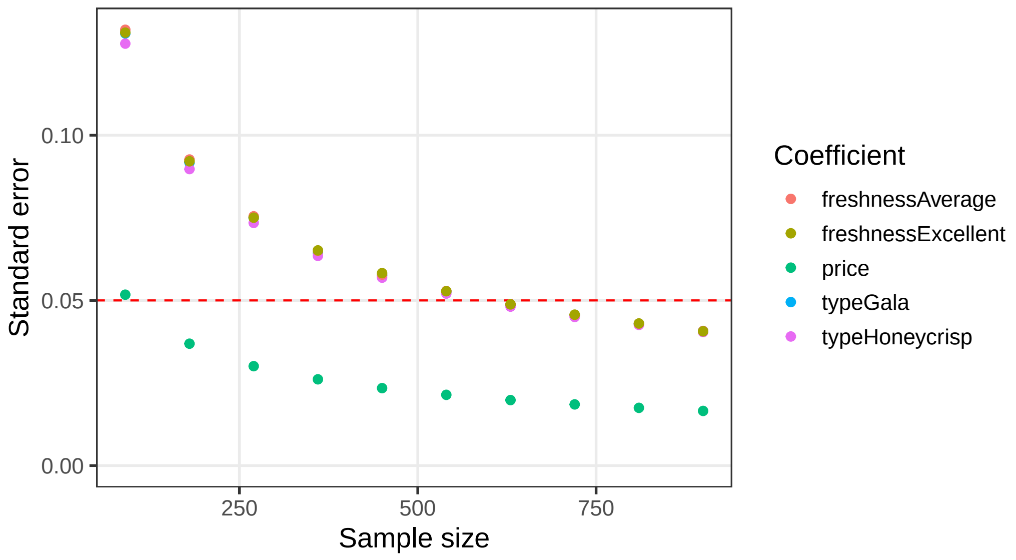

Figure 3: Standard errors of each model coefficient with increasing sample size.

Of course, designers may be interested in aspects other than standard errors. By setting `return_models = TRUE`, the `cbc_power()` function will return a list of estimated models (one for each sample size increment), which can then be used to examine other model objects. The example below prints a summary of the last model in the list of returned models.

```r
library(logitr)

models <- cbc_power(
  data    = data,
  pars    = c("price", "type", "freshness"),
  outcome = "choice",
  obsID   = "obsID",
  nbreaks = 10,
  n_q     = 6,
  return_models = TRUE
)

summary(models[[10]])
```

```
#> =================================================
#> Call:
#> FUN(data = X[[i]], outcome = ..1, obsID = ..2, pars = ..3, randPars = ..4,
#>     panelID = ..5, clusterID = ..6, robust = ..7, predict = ..8)
#>
#> Frequencies of alternatives:
#>       1       2       3
#> 0.35315 0.31537 0.33148
#>
#> Exit Status: 3, Optimization stopped because ftol_rel or ftol_abs was reached.
#>
#> Model Type:    Multinomial Logit
#> Model Space:          Preference
#> Model Run:                1 of 1
#> Iterations:                    9
#> Elapsed Time:        0h:0m:0.04s
#> Algorithm:        NLOPT_LD_LBFGS
#> Weights Used?:             FALSE
#> Robust?                    FALSE
#>
#> Model Coefficients:
#>                       Estimate Std. Error z-value Pr(>|z|)
#> price               -0.0123164  0.0165419 -0.7446  0.45654
#> typeGala            -0.0090449  0.0407194 -0.2221  0.82421
#> typeHoneycrisp       0.0614677  0.0404181  1.5208  0.12831
#> freshnessAverage    -0.0802807  0.0407526 -1.9700  0.04884 *
#> freshnessExcellent -0.0608045  0.0406397 -1.4962  0.13461
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
```

```
#> Log-Likelihood:          -5.928261e+03
#> Null Log-Likelihood:     -5.932506e+03
#> AIC:                      1.186652e+04
#> BIC:                      1.189949e+04
#> McFadden R2:              7.155816e-04
#> Adj McFadden R2:         -1.272325e-04
#> Number of Observations:   5.400000e+03
```

Piping it all together!

One of the convenient features of how the package is written is that the object generated in each step is used as the first argument to the function for the next step. Thus, just like in the overall program diagram (see Figure 2), the functions can be piped together. The example below will generate the same power analysis plot as in Figure 3 by sequentially evaluating each of the main design steps.

```
cbc_profiles(
  price     = seq(1, 4, 0.5), # $ per pound
  type      = c('Fuji', 'Gala', 'Honeycrisp'),
  freshness = c('Poor', 'Average', 'Excellent')
) |>
cbc_design(
  n_resp   = 900, # Number of respondents
  n_alts   = 3,   # Number of alternatives per question
  n_q      = 6    # Number of questions per respondent
) |>
cbc_choices(
  obsID = "obsID",
  priors = list(
    price     = 0.1,
    type      = c(0.1, 0.2),
    freshness = c(0.1, 0.2)
  )
) |>
cbc_power(
    pars    = c("price", "type", "freshness"),
    outcome = "choice",
    obsID   = "obsID",
    nbreaks = 10,
    n_q     = 6
) |>
plot()
```

One benefit of this piped structure is that it enables the designer to quickly observe the implications of different design choices on statistical power. For example, consider the following “what if” design questions:

- What if one more choice question was added to each respondent?
- What if the number of alternatives per choice question was decreased to two?
- What if a labeled design were used for the `type` attribute?
- What if there was an interaction effect between `price` and `type`?

Obtaining an answer to each of these questions requires only small modifications to the arguments in one or more functions inside the analysis “pipeline”. Once the change is made, the entire analysis can be quickly re-executed and the results compared to those of an alternative design. Because `cbcTools` leverages the `logitr` package (which is highly optimized for speed) for simulating choices and estimating models, re-executing each analysis should be relatively fast, taking only seconds to compute the entire process on most modern computers and laptops.

Conclusions

References