# 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/.

Designing a choice-based conjoint survey is almost never a simple, straightforward process. Designers must consider a wide variety of 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 highly iterative, yet there are few tools for quickly comparing the outcomes of different designs.

This paper introduces the `cbcTools` package, which provides a set of tools for designing surveys and conducting power analyses for choice-based conjoint survey experiments in R.

Often times the 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

*Engineering Management and Systems Engineering, George Washington University, Washington, D.C. USA

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/.

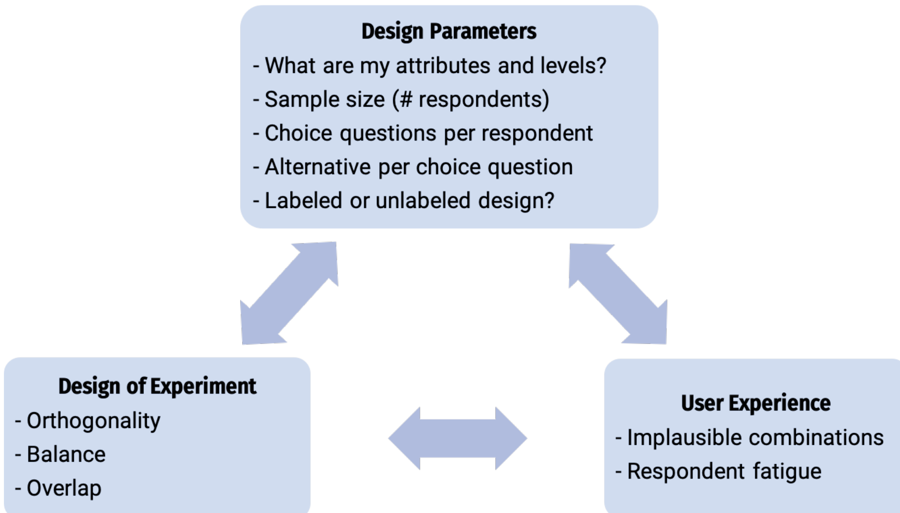



Figure 1: Caption

# .center[A simple conjoint experiment about *cars*]

| Attribute | Levels |
|---|---|
| Brand | GM, BMW, Ferrari |
| Price | $20k, $40k, $100k |

.center[**Design: .red[9] choice sets, .blue[3] alternatives each**]

```
Attribute counts:

brand:
  GM   BMW  Ferrari
  10    11     6

price:

 20k  40k 100k
  9    9    9

Pairwise attribute counts:
```

```
brand & price:

         20k 40k 100k
  GM       3   0    7
  BMW      4   5    2
  Ferrari  2   4    0
```

# .center[A simple conjoint experiment about *cars*]

| Attribute | Levels |
|---|---|
| Brand | GM, BMW, Ferrari |
| Price | \$20k, \$40k, \$100k |

.center[**Design: .red[90] choice sets, .blue[3] alternatives each**]

```
Attribute counts:

brand:
  GM    BMW   Ferrari
  92    80     98

price:

  20k  40k 100k
  91   84   95

Pairwise attribute counts:

brand & price:

         20k 40k 100k
  GM      31  31  30
  BMW     25  25  30
  Ferrari 35  28  35
```

# .center[Bayesian D-efficient designs]

**.center[Maximize information on "Main Effects" according to priors]**

| Attribute | Levels | Prior |
|---|---|---|
| Brand | GM, BMW, Ferrari | 0, 1, 2 |
| Price | \$20k, \$40k, \$100k | 0, -1, -4 |

```
Attribute counts:

brand:
  GM    BMW   Ferrari
  93    90     86

price:

  20k  40k 100k
  97   93   78

Pairwise attribute counts:

brand & price:

         20k 40k 100k
  GM      52  41  0
  BMW     30  30  30
  Ferrari 15  22  49
```

# .center[Bayesian D-efficient designs]

### .center[Attempts to maximize information on .red[Main Effects]]

"images/design_compare.png"

### .center[... but .red[interaction effects] are confounded in D-efficient designs]

"images/design_compare_int.png"

# .center[But what about other factors?]

- What if I add one more choice question to each respondent?
- What if I increase the number of alternatives per choice question?
- What if I use a labeled design (aka "alternative-specific design")?
- What if there are interaction effects?

## Make survey designs

### Generating profiles

The first step in designing an experiment is to define the attributes and levels for your experiment and then generate all of the `profiles` of each possible combination of those attributes and levels. For example, let's say you're designing a conjoint experiment about apples and you want to include `price`, `type`, and `freshness` as attributes. You can obtain all of the possible profiles for these attributes 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 your survey, you may wish to eliminate or modify some profiles before designing your conjoint survey (e.g., some profile combinations may be illogical or unrealistic). **WARNING: including hard constraints in your designs can substantially reduce the statistical power of your design, so use them cautiously and avoid them if possible**.

If you do wish to set some levels conditional on those of other attributes, you can do so by setting each level of an attribute to a list that defines these constraints. 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, 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. When these constraints you can see that there are only 30 profiles 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(
        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
```

```r
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
```

```r
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 random designs

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

```r
design <- 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
)

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        32   2.5 Gala   Average
#> 2      1   1     2     1         4   2.5 Fuji      Poor
#> 3      1   1     3     1        56   4.0 Gala Excellent
#> 4      1   2     1     2        24   2.0 Fuji   Average
#> 5      1   2     2     2         4   2.5 Fuji      Poor
#> 6      1   2     3     2         9   1.5 Gala      Poor
```

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, such as Bayesian D-efficient designs. The randomized design simply samples from the set of `profiles`. It also ensures that no two profiles are the same in any choice question.

The resulting `design` data frame includes the following columns:

- `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)

You can also make a "labeled" design (also known as "alternative-specific" design) where the levels of one attribute is used as a label by setting the `label` argument to that attribute. This by definition sets the number of alternatives in each question to the number of levels in the chosen attribute, so the `n_alts` argument is overridden. Here is an example labeled survey using the `type` attribute 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        23   1.5       Fuji   Average
#> 2      1   1     2     1        32   2.5       Gala   Average
#> 3      1   1     3     1        40   3.0 Honeycrisp   Average
#> 4      1   2     1     2         4   2.5       Fuji      Poor
#> 5      1   2     2     2         8   1.0       Gala      Poor
#> 6      1   2     3     2        39   2.5 Honeycrisp   Average
```

In the above example, you can see in the first six rows of the survey that the `type` attribute is always fixed to be the same order, 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")

You can include a "no choice" (also known as "outside good" option in your survey 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        54   3.0         0         1               0
#> 2      1   1     2     1        44   1.5         1         0               0
#> 3      1   1     3     1        43   1.0         1         0               0
#> 4      1   1     4     1         0   0.0         0         0               0
#> 5      1   2     1     2        56   4.0         0         1               0
```

```
#> 6      1   2     2     2         31   2.0          0          1                0
#>   freshness_Poor freshness_Average freshness_Excellent no_choice
#> 1              0                 0                   1         0
#> 2              0                 0                   1         0
#> 3              0                 0                   1         0
#> 4              0                 0                   0         1
#> 5              0                 0                   1         0
#> 6              0                 1                   0         0
```

## Inspecting survey 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)
```

```
#> ==============================
#> Attribute counts:
#>
#> price:
#>
#>    1  1.5    2  2.5    3  3.5    4
#> 2376 2350 2272 2332 2299 2301 2270
#>
#> type:
#>
#>       Fuji       Gala Honeycrisp
#>       5396       5438       5366
#>
#> freshness:
#>
#>      Poor   Average Excellent
#>      5398      5354      5448
#>
#> ==============================
#> Pairwise attribute counts:
#>
#> price & type:
#>
#>       Fuji Gala Honeycrisp
#>   1    796  803        777
#>   1.5  781  778        791
#>   2    737  768        767
```

```
#>   2.5  801  771        760
#>   3    758  791        750
#>   3.5  779  751        771
#>   4    744  776        750
#>
#> price & freshness:
#>
#>       Poor Average Excellent
#>   1    824     780       772
#>   1.5  790     793       767
#>   2    750     749       773
#>   2.5  734     809       789
#>   3    769     756       774
#>   3.5  816     711       774
#>   4    715     756       799
#>
#> type & freshness:
#>
#>              Poor Average Excellent
#>   Fuji       1809    1872      1715
#>   Gala       1817    1720      1901
#>   Honeycrisp 1772    1762      1832
```

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
#>    59 1856 3485
#>
#> type:
#>
#>     1    2    3
#>   565 3627 1208
#>
```

```
#> freshness:
#>
#>    1    2    3
#>  551 3590 1259
```

## Simulating choices

You can simulate choices for a given `design` 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        32   2.5 Gala   Average      0
#> 2      1   1     2     1         4   2.5 Fuji      Poor      0
#> 3      1   1     3     1        56   4.0 Gala Excellent      1
#> 4      1   2     1     2        24   2.0 Fuji   Average      1
#> 5      1   2     2     2         4   2.5 Fuji      Poor      0
#> 6      1   2     3     2         9   1.5 Gala      Poor      0
```

You can also pass a list of prior parameters to define a utility model that will be used to simulate choices. 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 $j$:

$$u_j = 0.1 price_j + 0.1 typeGala_j + 0.2 typeHoneycrisp_j + 0.1 freshnessAverage_j + 0.2 freshnessExcellent_j + \varepsilon_j$$

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

```
  )
)
```

| Attribute | Level |
|---|---|
| **Price** | Continuous |
| **Type** | Fuji |
| Gala | 0.1 |
| Honeycrisp | 0.2 |
| **Freshness** | Average |
| Excellent | 0.1 |
| Poor | -0.2 |

If you wish to include a prior model with an interaction, you can do so inside the `priors` list. For example, here is the same example as above but with an interaction between `price` and `type` added:

| Attribute | Level |
|---|---|
| **Price** | Continuous |
| **Type** | Fuji |
| Gala | 0.1 |
| Honeycrisp | 0.2 |
| **Freshness** | Average |
| Excellent | 0.1 |
| Poor | -0.2 |
| **Price x Type** | Fuji |
| Gala | 0.1 |
| Honeycrisp | 0.5 |

```
data <- cbc_choices(
  design = design,
  obsID = "obsID",
  priors = list(
    price = 0.1,
    type = c(0.1, 0.2),
    freshness = c(0.1, 0.2),
    `price*type` = c(0.1, 0.5)
  )
)
```

Finally, you can also simulate data for a mixed logit model where parameters follow a normal or log-normal distribution across the population. In the example below, the `randN()` function is used to specify the `type` attribute with 2 random normal parameters with a specified vector of means (`mean`) and standard deviations (`sd`) for each level of `type`. Log-normal parameters are specified using `randLN()`.

| Attribute | Level |
|---|---|
| **Price** | Continuous |
| **Type** | Fuji |
| Gala | N(0.1, 0.5) |
| Honeycrisp | N(0.2, 1) |
| **Freshness** | Average |
| Excellent | 0.1 |
| Poor | -0.2 |

```
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)
  )
)
```

## Conducting a power analysis

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). 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:

```
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
```

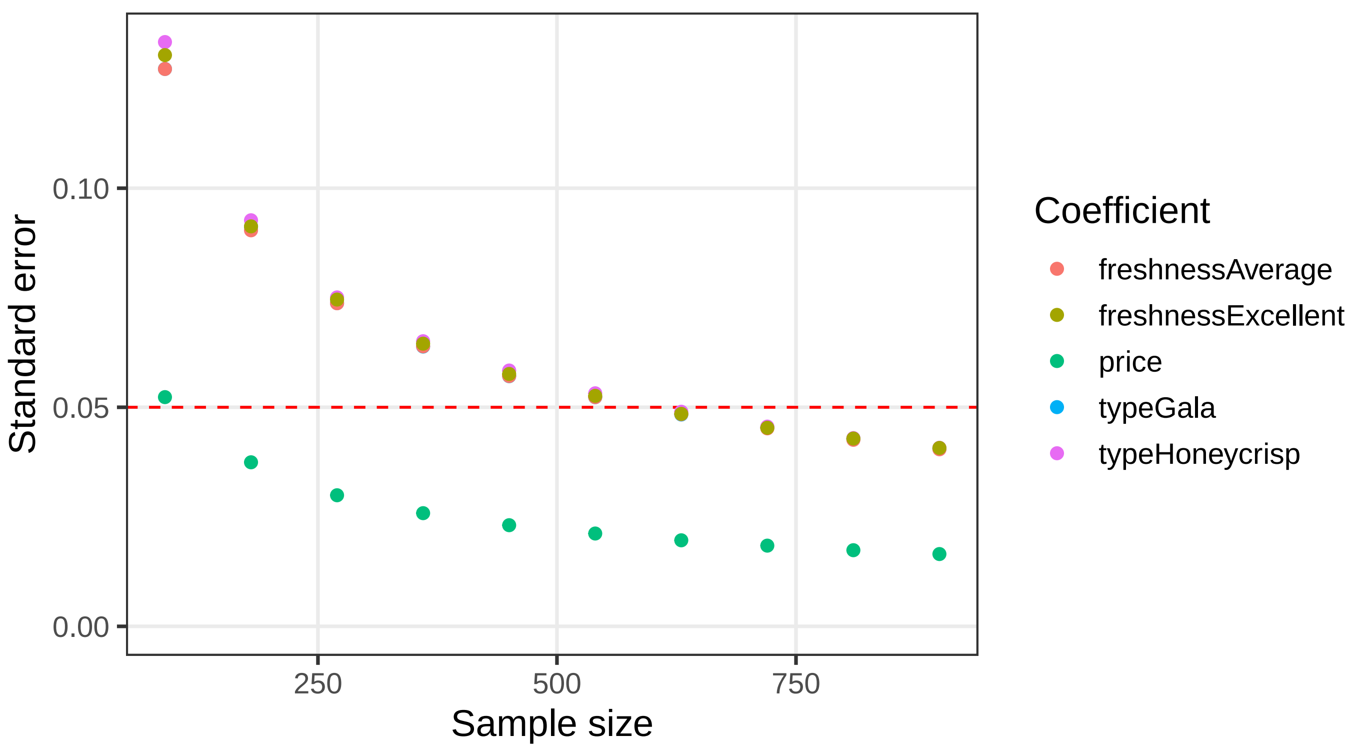
```
#> 1          90               price  0.06365535 0.05231277
#> 2          90            typeGala  0.14335624 0.12721957
#> 3          90      typeHoneycrisp  0.04351127 0.13335781
#> 4          90    freshnessAverage -0.02536818 0.12720709
#> 5          90 freshnessExcellent -0.05568708 0.13037883
#> 6         180               price  0.06250365 0.03744184
```

```
tail(power)
```

```
#>    sampleSize               coef         est         se
#> 45        810 freshnessExcellent  0.01969887 0.04286235
#> 46        900              price  0.01910029 0.01650147
#> 47        900           typeGala -0.02691359 0.04054378
#> 48        900     typeHoneycrisp -0.06048486 0.04074588
#> 49        900   freshnessAverage  0.04759586 0.04041492
#> 50        900 freshnessExcellent  0.03429806 0.04071614
```

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

```
plot(power)
```




If you want to examine any other aspects of the models other than the standard errors, you can set `return_models = TRUE` and `cbc_power()` will return a list of estimated models. The example below prints a summary of the last model in the list of 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]])
```

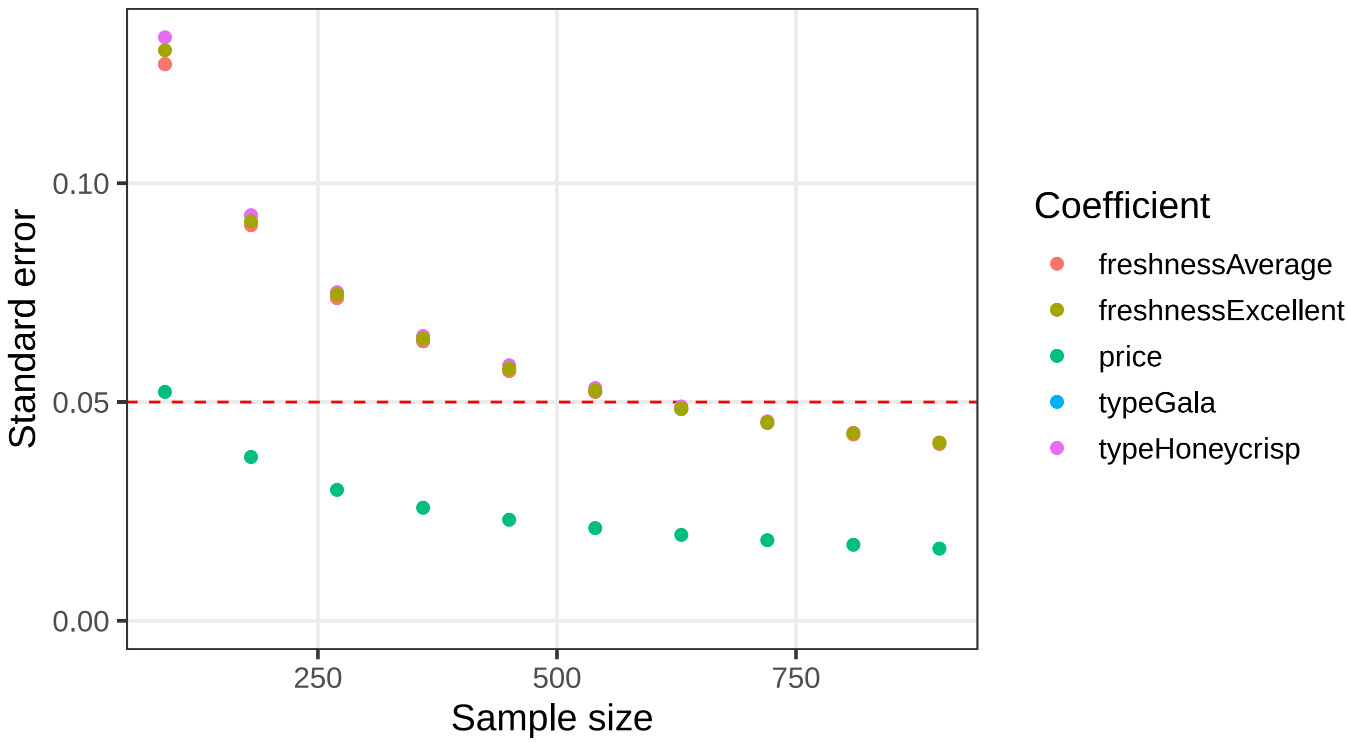
```
#> =================================================
#> Call:
#> FUN(data = X[[i]], outcome = ..1, obsID = ..2, pars = ..3, randPars = ..4,
#>     panelID = ..5, clusterID = ..6, robust = ..7, predict = ..8)
#>
#> Frequencies of alternatives:
#>       1       2       3
#> 0.33556 0.33204 0.33241
#>
#> 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:                    8
#> Elapsed Time:        0h:0m:0.02s
#> Algorithm:        NLOPT_LD_LBFGS
#> Weights Used?:             FALSE
#> Robust?                    FALSE
#>
#> Model Coefficients:
#>                     Estimate Std. Error z-value Pr(>|z|)
#> price               0.019100   0.016501  1.1575   0.2471
#> typeGala           -0.026914   0.040544 -0.6638   0.5068
#> typeHoneycrisp     -0.060485   0.040746 -1.4844   0.1377
#> freshnessAverage    0.047596   0.040415  1.1777   0.2389
#> freshnessExcellent  0.034298   0.040716  0.8424   0.3996
#>
#> Log-Likelihood:         -5.929988e+03
#> Null Log-Likelihood:    -5.932506e+03
```

```
#> AIC:                        1.186998e+04
#> BIC:                        1.190295e+04
#> McFadden R2:                4.244950e-04
#> Adj McFadden R2:           -4.183191e-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, the functions can be piped together:

```
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()
```




## Author, Version, and License Information

- Author: *John Paul Helveston* https://www.jhelvy.com/
- Date First Written: *October 23, 2020*
- License: MIT

## Citation Information

If you use this package for in a publication, I would greatly appreciate it if you cited it - you can get the citation by typing `citation("cbcTools")` into R:

```
citation("cbcTools")
```

```
#>
#> To cite cbcTools in publications use:
#>
#>   John Paul Helveston (2022). cbcTools: Tools For Designing Conjoint
#>   Survey Experiments.
#>
#> A BibTeX entry for LaTeX users is
#>
#>   @Manual{,
#>     title = {cbcTools: Tools For Designing Choice-Based Conjoint Survey Experiments},
#>     author = {John Paul Helveston},
#>     year = {2022},
```

```
#>     note = {R package version 0.0.3},
#>     url = {https://jhelvy.github.io/cbcTools/},
#>   }
```

# References