Package ‘strategize’

January 19, 2025

Type Package

Title Tools for Learning Adversarial or Non-Adversarial Optimal Distributions in High-Dimensional Conjoint Experiments

Version 0.0.1

Date 2025-01-08

Description The 'strategize' package implements methods for learning an optimal or adversarial probability distribution over high-dimensional factors in conjoint (or factorial) experiments. It supports both single-agent (non-adversarial) optimization and adversarial two-player settings, such as multi-stage electoral contexts. Users can estimate the distribution of factor levels that best achieves a chosen objective, optionally under institutional constraints (e.g., two-stage primaries). The package offers a variety of estimation routines, including closed-form solutions for some linear models, gradient-based optimizers for more complex outcome models, and built-in support for inference (via the delta method). It is particularly suitable for exploring and comparing candidate strategies in forced-choice conjoint studies, but is general enough for broader use in high-dimensional policy learning.

URL `https://github.com/cjerzak/strategize-software`

BugReports `https://github.com/cjerzak/strategize-software/issues`

Depends R (>= 3.3.3)

License GPL-3

Encoding UTF-8

LazyData true

Imports stats,
graphics,
utils,
methods,
reticulate

Suggests testthat (>= 3.0.0),
Matrix,
sandwich,
compositions,
Rcpp,
devtools,
knitr,
rmarkdown

VignetteBuilder knitr

RoxygenNote 7.3.2

NeedsCompilation no

Contents

`build_backend` *A function to build the environment for strategize Builds a conda environment in which 'JAX' and 'np' are installed. Users can also create a conda environment where 'JAX' and 'np' are installed themselves.*

Description

A function to build the environment for strategize Builds a conda environment in which 'JAX' and 'np' are installed. Users can also create a conda environment where 'JAX' and 'np' are installed themselves.

Usage

```
build_backend(conda_env = "strategize", conda = "auto")
```

Arguments

conda_env	(default = `"strategize"`) Name of the conda environment in which to place the backends.
conda	(default = `auto`) The path to a conda executable. Using `"auto"` allows reticulate to attempt to automatically find an appropriate conda binary.

Value

Invisibly returns NULL; this function is used for its side effects of creating and configuring a conda environment for `strategize`. This function requires an Internet connection. You can find out a list of conda Python paths via: `Sys.which("python")`

Examples

```
## Not run:
# Create a conda environment named "strategize"
# and install the required Python packages (jax, numpy, etc.)
build_backend(conda_env = "strategize", conda = "auto")

# If you want to specify a particular conda path:
# build_backend(conda_env = "strategize", conda = "/usr/local/bin/conda")

## End(Not run)
```

`cv.OptiConjoint` *Cross-validation for Optimal Stochastic Interventions in Conjoint Analysis*

Description

Performs cross-validation to select the regularization parameter λ (and, if desired, other hyperparameters) for the `OptiConjoint` function. This function splits the data by respondent (or user-specified units), trains candidate models under a grid of λ values, and evaluates out-of-sample performance, returning the model that maximizes a chosen criterion (e.g., out-of-sample expected utility or log-likelihood).

Usage

```
cv.OptiConjoint(
  Y,
  W,
  X = NULL,
  lambda_seq = NULL,
  lambda = NULL,
  folds = 2L,
  varcov_cluster_variable = NULL,
  competing_group_variable_respondent = NULL,
  competing_group_variable_candidate = NULL,
  competing_group_competition_variable_candidate = NULL,
  pair_id = NULL,
  respondent_id = NULL,
  respondent_task_id = NULL,
  profile_order = NULL,
  p_list = NULL,
  slate_list = NULL,
  UseOptax = F,
  K = 1,
  nSGD = 100,
  diff = F,
  MaxMin = F,
  UseRegularization = F,
  OpenBrowser = F,
  ForceGaussianFamily = F,
  A_INIT_SD = 0.001,
  TypePen = "KL",
  ComputeSEs = T,
  conda_env = NULL,
  conda_env_required = F,
  confLevel = 0.9,
  nFolds_glm = 3L,
  nMonte_MaxMin = 5L,
  nMonte_Qglm = 100L,
  jax_seed = as.integer(Sys.time()),
  OptimType = "default"
)
```

Arguments

`Y`	A numeric or binary response vector. If binary (e.g., 0–1), it should correspond to forced-choice outcomes (1 if candidate `A` is chosen; 0 if candidate `B` is chosen). If numeric, please see details in `OptiConjoint` for how outcomes are handled.
`W`	A data frame or matrix representing the randomized conjoint attributes. Each column is a factor or character vector indicating attribute levels for a particular dimension. Multiple columns can be used if the conjoint has multiple attributes.
`X`	Optional covariate matrix or data frame for modeling systematic heterogeneity. If `K > 1`, this is typically required for multi-class or cluster-based models. Otherwise, set `X = NULL`.
`lambda_seq`	A numeric vector of candidate λ values for cross-validation. If `NULL` and `lambda` is also `NULL`, a sequence of values is automatically generated (e.g., via `10^seq(-4, 0, length.out = 5) * sd(Y)`).
`lambda`	A single user-specified λ value. If provided, cross-validation is effectively disabled unless `lambda_seq` is also supplied.
`folds`	An integer or user-specified partitioning indicating the number of cross-validation folds. Defaults to 2. See Details for how data splitting is done.
`varcov_cluster_variable`	An optional clustering variable for robust standard errors. For instance, if the data is from multiple respondents, specify respondent IDs here for cluster-robust inference (via sandwich estimation). If `NULL`, no cluster-based variance correction is used.
`competing_group_variable_respondent`	Optional vector for multi-round or multi-group setups, indicating which respondent belongs to which group. Used for advanced or adversarial designs (e.g., dual-party contexts). If `NULL`, standard usage is assumed.
`competing_group_variable_candidate`	Similar to `competing_group_variable_respondent`, but for candidate-level grouping. If `NULL`, standard usage is assumed.
`competing_group_competition_variable_candidate`	An optional variable for specifying which candidate is in competition with which group. Relevant if multi-step adversarial frameworks are used.
`pair_id`	An optional vector (same length as `Y`) identifying which rows (candidate pairs) belong to the same forced choice. For example, if each respondent evaluates multiple pairs, this ID ensures correct grouping. Required only in certain advanced difference-in-differences or paired analyses.
`respondent_id`	A user-specified ID to denote respondent-level grouping, typically used to cluster standard errors or to perform out-of-sample validation by respondent. If `NULL`, a simple row index is used for splitting.
`respondent_task_id`	Another optional ID for tasks (e.g., each respondent might see multiple tasks). Helps in advanced designs. If `NULL`, ignored.
`profile_order`	An optional vector capturing the ordering of candidate profiles within tasks, if multiple profiles are being shown. Used in difference or extended hierarchical modeling.
`p_list`	A list of assignment probabilities for each attribute, if known or desired as a baseline. If `NULL`, each level is assumed to have uniform probability or derived from empirical frequencies in `W`.

`slate_list`	An optional list specifying alternative or restricted sets of attribute levels. Used when a subset of attributes is feasible or when bounding certain strategies in an adversarial design.
`UseOptax`	Logical. If `TRUE`, uses the **optax** Python library (via **reticulate**) for gradient-based optimization. If `FALSE`, uses a default gradient-based approach from **jax**.
`K`	An integer specifying the number of mixture components or clusters if `X` is used (e.g., for multi-class analysis). Defaults to 1 (no mixture).
`nSGD`	An integer number of iterations for gradient-based training. Defaults to 100 but can be increased if convergence has not been reached.
`diff`	Logical indicating whether a difference-based model (e.g., for forced-choice or difference-in-outcomes) is used. Defaults to `FALSE`, but set `TRUE` in certain difference-of-utility designs.
`MaxMin`	Logical indicating whether to use a two-party or multi-agent *adversarial* approach in the optimization. If `TRUE`, a min-max (zero-sum) formulation is employed. Defaults to `FALSE` (single-agent or average-case optimization).
`UseRegularization`	Logical; if `TRUE`, penalty-based regularization is used for the outcome model. Usually set to `TRUE` for large designs. Defaults to `FALSE`.
`OpenBrowser`	Logical used in debugging (e.g., opening a browser for interactive inspection within **jax** or **reticulate**). Defaults to `FALSE`.
`ForceGaussianFamily`	Logical indicating whether a Gaussian family (`lm`-style) is forced for the outcome model, even if `Y` is binary. Defaults to `FALSE`.
`A_INIT_SD`	A numeric controlling the random initialization scale for unconstrained parameters in gradient-based optimization. Defaults to 0.001. Larger values can help avoid local minima in complex outcome landscapes.
`TypePen`	A character string specifying the type of penalty for the *optimal stochastic intervention*, e.g., `"KL"`, `"L2"`, or `"LogMaxProb"`. The default is `"KL"`.
`ComputeSEs`	Logical; if `TRUE`, attempts to compute standard errors using M-estimation or the Delta method. Defaults to `TRUE`.
`conda_env`	A character specifying the name of a Conda environment for **reticulate**. If `NULL`, the default environment is used.
`conda_env_required`	Logical. If `TRUE`, errors if the specified Conda environment `conda_env` cannot be found. Otherwise tries to fall back gracefully.
`confLevel`	The confidence level (between 0 and 1) for interval estimation, default 0.90.
`nFolds_glm`	An integer specifying the number of folds in internal regression-based cross-validation (if used) for outcome model selection. Defaults to 3.
`nMonte_MaxMin`	A positive integer specifying the number of Monte Carlo draws for the min-max (adversarial) stage, if `MaxMin = TRUE`. Defaults to 5.
`nMonte_Qglm`	An integer specifying the number of Monte Carlo draws for evaluating certain integrals in `glm`-based approximations, default 100.
`jax_seed`	An integer seed for the JAX random number generator. Defaults to `as.integer(Sys.time())`.
`OptimType`	A character describing the optimization routine. Typically `"default"` uses a standard gradient-based approach; set `"tryboth"` or `"SecondOrder"` for testing or advanced routines.

Details

`cv.OptiConjoint` implements a cross-validation routine for `OptiConjoint`. First, the data is split into `folds` parts. For each fold, we train candidate outcome models and compute out-of-sample performance. The best-performing λ is selected. Finally, a refit on the full data is done using the chosen hyperparameters, returning the results of the final `OptiConjoint` call with λ set to the best value.

The function supports a wide range of conjoints, including forced-choice (where `diff = TRUE`), multi-cluster outcome modeling (where $K > 1$), and adversarial designs (where `MaxMin = TRUE`). Regularization for the outcome model or for the candidate distribution can be enabled via `UseRegularization` and `TypePen`. Cross-validation is particularly helpful when the data is limited or highly dimensional.

Value

A named list with components:

PiStar_point The estimated optimal probability distribution(s) over candidate profiles ($\hat{\pi}^*$).

Q_point_mEst The estimated expected outcome (e.g., vote share) under the selected optimal distribution.

lambda The chosen λ value from cross-validation (and any other relevant hyperparameters).

CVInfo A data frame or matrix summarizing cross-validation results, e.g., in-sample and out-of-sample estimates for each candidate λ.

Other components Various additional objects useful for inference and debugging (e.g., final model fits, standard error estimates, weighting details).

See Also

`OptiConjoint` for direct optimization of stochastic interventions in conjoint analysis, including average and adversarial settings.

Examples

```
# A minimal example using hypothetical data
set.seed(123)
# Suppose Y is a binary forced choice outcome, W has several attributes (factors)
Y <- rbinom(200, size = 1, prob = 0.5)
W <- data.frame(
  Gender = sample(c("Male","Female"), 200, TRUE),
  Age    = sample(c("35","50","65"),  200, TRUE),
  Party  = sample(c("Dem","Rep"),     200, TRUE)
)

# Cross-validate over a range of lambda
lam_seq <- c(0, 0.001, 0.01, 0.1)
cv_fit <- cv.OptiConjoint(
  Y = Y,
  W = W,
  lambda_seq = lam_seq,
  folds = 2
)

# Extract optimal lambda and final fit
print(cv_fit$lambda)
print(cv_fit$CVInfo)
print(names(cv_fit$PiStar_point))
```

`OneStep.OptiConjoint` *Estimate an Optimal (or Adversarial) Stochastic Intervention for Conjoint Analysis Using a One-Step M-estimation Approach*

Description

`OneStep.OptiConjoint` implements a single-step ("one-step") approach to estimating a target quantity of interest in high-dimensional conjoint (or factorial) experiments, such as finding an *optimal stochastic intervention* over factor levels. This method can incorporate adversarial or non-adversarial settings, regularization, multi-stage structures (e.g., primaries followed by a general election), and a variety of user-specified outcome models. It returns estimated distributions of factor levels *(e.g., candidate attributes)* that maximize or minimize an outcome (e.g., vote share), including optional standard errors via M-estimation or the delta method.

Usage

```
OneStep.OptiConjoint(
  W,
  Y,
  X = NULL,
  K = 1,
  warmStart = FALSE,
  automatic_scaling = TRUE,
  p_list = NULL,
  hypotheticalProbList = NULL,
  pi_init_vec = NULL,
  constrain_ub = NULL,
  nLambda = 10,
  penaltyType = "LogMaxProb",
  testFraction = 0.5,
  log_PrW = NULL,
  LEARNING_RATE_BASE = 0.01,
  cycle_width  = 50,
  cycle_number = 4,
  nSGD = 500,
  nEpoch = NULL,
  X_factorized = NULL,
  momentum = 0.99,
  nFullCycles_noRestarts = 1,
  optim_method = "tf",
  sg_method = NULL,
  forceSEs = FALSE,
  clipAT_factor = 100000,
  adaptiveMomentum = FALSE,
  PenaltyType = "L2",
  knownNormalizationFactor = NULL,
  split1_indices = NULL,
  split2_indices = NULL,
  openBrowser = FALSE,
  useHajekInOptimization = TRUE,
  findMax = TRUE,
  quiet = TRUE,
  lambda_seq = NULL,
  lambda_coef = 0.0001,
  nFolds = 3,
  batch_size = 50,
  confLevel = 0.90,
  conda_env = NULL,
  conda_env_required = FALSE,
  hypotheticalNInVarObj = NULL
)
```

Arguments

`W`	A matrix or data frame of assigned factor levels in the conjoint. For forced-choice designs, each row generally represents a single profile or a single respondent-task-profile combination. Must have integer, factor, or character columns representing factor levels.
`Y`	A numeric or binary outcome vector. Typically `Y` is `1` if a profile is chosen over its competitor, and `0` otherwise. If `findMax = FALSE`, the sign is flipped, effectively minimizing `Y`.
`X`	Optional matrix or data frame of covariates (or respondent-level features). Can be used for multi-cluster modeling with `K>1`.
`K`	An integer for the number of mixture components or latent clusters if multi-cluster modeling is desired. Defaults to `1` (no clusters).
`warmStart`	Logical. If `TRUE`, attempts to re-initialize from previous solutions each time `lambda` or other hyperparameters change. Defaults to `FALSE`.
`automatic_scaling`	Logical indicating whether to center or scale `X` and `Y` automatically. Defaults to `TRUE`.
`p_list`	A list of baseline factor-level probabilities in the design or assignment mechanism (e.g., the original random assignment distribution). If `NULL`, the function may assume uniform or empirical distributions.
`hypotheticalProbList`	An optional list specifying a counterfactual distribution over factor levels. If provided, `OneStep.OptiConjoint` directly computes and returns the performance or value under that distribution instead of estimating a new optimal distribution.
`pi_init_vec`	A numeric vector for initializing the simplex-based representation of factor-level probabilities to be optimized. If `NULL`, a random initialization is used internally.
`constrain_ub`	Optional numeric or vector of upper bounds on factor probabilities. If not `NULL`, can help to enforce constraints in optimization.
`nLambda`	Integer specifying the number of penalty values considered if cross-validation is performed. Defaults to 10.
`penaltyType`	A character specifying the type of penalty for shifting probabilities (e.g., `"LogMaxProb"`, `"L2"`, or `"KL"`). This is an additional penalization on top of `PenaltyType` for the outcome model. Defaults to `"LogMaxProb"`.

`testFraction`	Fraction of samples used for holdout in cross-validation. Defaults to 0.5 for a basic split. If `NULL`, no split is performed.
`log_PrW`	Optional numeric vector of log probabilities for each row in `W`. If omitted, the function will compute `log_PrW` from `p_list` given the assumption of independent factor-level assignments.
`LEARNING_RATE_BASE`	Base learning rate for gradient-based optimizers. Defaults to 0.01.
`cycle_width`	Numeric controlling the frequency of restarts or adaptive learning-rate schedules.
`cycle_number`	Number of cycles used in the learning-rate schedule.
`nSGD`	Number of gradient-descent updates. If `nEpoch` is provided, that takes precedence.
`nEpoch`	Number of epochs, each pass including `length(availableTrainIndices) / batch_size` mini-batches. If provided, overrides `nSGD`.
`X_factorized`	An optional matrix or data frame representing factorized (dummy-coded) versions of `X` for advanced modeling. If `NULL`, the function may factorize internally.
`momentum`	Numeric specifying momentum for stochastic gradient descent. Defaults to 0.99.
`nFullCycles_noRestarts`	If `>1`, repeats training cycles without restarts for a total number of gradient steps. Useful for stability checks.
`optim_method`	A character specifying the optimization backend (e.g., `"tf"` for TensorFlow-based, or `"jax"` for JAX-based). Defaults to `"tf"` if available.
`sg_method`	A character controlling the type of gradient updates (e.g., `"adanorm"`, `"wngrad"`). If `NULL`, a default method is chosen.
`forceSEs`	Logical. If `TRUE`, attempts to compute standard errors by M-estimation or the delta method, even if no cross-validation is done.
`clipAT_factor`	A large numeric to clip gradient norms if they exceed `clipAT_factor`.
`adaptiveMomentum`	Logical. If `TRUE`, momentum is adapted automatically as the optimization proceeds. Defaults to `FALSE`.
`PenaltyType`	A character specifying the type of penalty (e.g., `"L2"`) for the outcome model. Used only if a penalized approach to outcome model fitting is internally performed.
`knownNormalizationFactor`	An optional numeric to normalize reweighting for Hajek-based estimators. If `NULL`, it is inferred from the sum of weights.
`split1_indices`, `split2_indices`	Optional vectors of indices partitioning the data for cross-validation or holdout. If `NULL`, a random partition is done internally.
`openBrowser`	Logical for debugging. If `TRUE`, may open an interactive browser for advanced inspection.
`useHajekInOptimization`	Logical. If `TRUE`, uses a Hajek-based reweighting in objective functions for computing the expected outcome under counterfactual probability shifts. Defaults to `TRUE`.
`findMax`	Logical. If `TRUE`, maximizes `Y`; if `FALSE`, treats `Y` as negative of interest (e.g., adversity minimization).

`quiet`	Logical controlling the verbosity of printed messages.
`lambda_seq`	Optional numeric vector of penalty values for cross-validation. If `NULL`, the function attempts a default sequence or single value.
`lambda_coef`	Numeric constant controlling the magnitude of the penalty for the outcome model. Defaults to 0.0001.
`nFolds`	Number of folds for cross-validation. Defaults to 3.
`batch_size`	Positive integer specifying the size of mini-batches in each gradient iteration. Defaults to 50.
`confLevel`	Numeric in \((0,1)\), specifying the confidence level for intervals around estimated probabilities or performance measures. Defaults to 0.90.
`conda_env`	Optional name of a Conda environment with **jax**, **optax**, etc. If `NULL`, attempts a default environment.
`conda_env_required`	Logical. If `TRUE`, errors if the environment `conda_env` cannot be found. Otherwise attempts to proceed gracefully.
`hypotheticalNInVarObj`	Optional numeric specifying an alternative `n` for certain variance calculations (e.g., hypothesized population size). If `NULL`, uses the observed sample size.

Details

This function implements a *one-step M-estimation* approach for directly estimating the “optimal” probability distributions over high-dimensional factors in conjoint or factorial experiments. Rather than a multi-step procedure of (1) outcome modeling followed by (2) re-optimizing factor distributions, the one-step approach can iteratively re-estimate distribution parameters while simultaneously adjusting the outcome model. This allows regularization or advanced modeling to be integrated into the *same* optimization objective, potentially improving finite-sample performance. Support for adversarial or multiple clusters is also available.

By default, `OneStep.OptiConjoint` attempts to find the distribution(s) $\boldsymbol{\pi}^*$ that maximizes the average outcome if `findMax = TRUE` (e.g., maximizing candidate choice share). In adversarial contexts, each cluster or “player” can simultaneously learn a best response. The function is flexible enough to incorporate sub-populations or multiple stages (e.g., primaries plus general elections).

If a user-supplied `hypotheticalProbList` is given, the function directly computes $Q(\boldsymbol{\pi})$ for that distribution instead of estimating. This is useful for evaluating the performance of a known or hypothesized distribution (e.g., “status quo”).

Most users do not need to call `OneStep.OptiConjoint` directly, as this is a lower-level routine. The `OptiConjoint` or `cv.OptiConjoint` functions may suffice in many typical workflows.

Value

A named `list` with components, often including:

- `PiStar_point`: The estimated optimal or learned distribution(s) over factor levels. If `K>1` or if adversarial competition is considered, may return multiple distributions (`k1`, `k2`, etc.).
- `Q_point`: The estimated performance measure under the learned distribution(s). For example, the average or adversarially optimized outcome.
- `Q_se_mEst`: If available, standard errors via M-estimation or the delta method.
- `PiStar_lb`, `PiStar_ub`: Lower and upper confidence intervals for factor-level probabilities, if standard errors are computed.

- `CVInfo`: A data frame or list summarizing cross-validation performance for each candidate `lambda`.
- `ClassProbsXobs`, `VarCov_ProbClust`, `pi_init_next`, `optim_max_hajek_list`: Additional objects storing advanced details of the optimization or M-estimation procedure.
- `Output.Description`: Additional messages describing the run.

Note

Advanced arguments like `X_factorized`, `conda_env`, `optim_method`, or specifying `adaptiveMomentum` are only needed for specialized or larger-scale (GPU-based) computations.

References

- Goplerud, M. & Titiunik, R. (2022). *Analysis of High-Dimensional Factorial Experiments: Estimation of Interactive and Non-Interactive Effects.* ArXiv preprint. - Egami, N. & Imai, K. (2019). *Causal Interaction in Factorial Experiments: Application to Conjoint Analysis.* Journal of the American Statistical Association, 114(526), 529–540. - Hainmueller, J., Hopkins, D. J., & Yamamoto, T. (2014). *Causal Inference in Conjoint Analysis: Understanding Multidimensional Choices via Stated Preference Experiments.* Political Analysis, 22(1), 1–30. - (Paper Reference) A forthcoming or accompanying manuscript describing in detail the methods for *optimal* or *adversarial* stochastic interventions in conjoint settings.

See Also

`OptiConjoint` for an approach that first fits an outcome model and then re-optimizes factor-level probabilities. \ `cv.OptiConjoint` for cross-validation across candidate values of `lambda`.

Examples

```
## Not run:
# Suppose we have a forced-choice conjoint dataset (W, Y) and baseline probabilities p_list.
# We want to estimate an optimal distribution that maximizes average Y.

set.seed(123)
# X could be respondent covariates, if any
X <- matrix(rnorm(nrow(W)*2), nrow(W), 2)

result_one_step <- OneStep.OptiConjoint(
  W = W,
  Y = Y,
  X = X,
  p_list = p_list,
  nSGD = 400,
  useHajekInOptimization = TRUE,
  penaltyType = "LogMaxProb",
  PenaltyType = "L2",
  lambda_seq = c(0.01, 0.1),
  testFraction = 0.3
)

# Inspect the estimated distribution over factor levels
str(result_one_step$PiStar_point)

# Evaluate estimated performance
print( result_one_step$Q_point )
```

```
## End(Not run)
```

`OptiConjoint` *Estimate Optimal (or Adversarial) Stochastic Interventions for Conjoint Experiments*

Description

`OptiConjoint` implements the core methods described in the accompanying paper for learning an optimal or adversarial probability distribution over conjoint factor levels. It is specifically designed for forced-choice conjoint settings (e.g., candidate-choice experiments) and can accommodate scenarios in which a single agent optimizes its strategy in isolation, or in which two (potentially adversarial) agents simultaneously optimize against each other.

This function can be used to find the *optimal stochastic intervention* for maximizing an outcome of interest (e.g., vote choice, rating, or utility), possibly subject to a penalty that keeps the learned distribution close to the original design distribution. It can also incorporate institutional rules (e.g., primaries, multiple stages of choice) by specifying additional arguments. Estimation can be done under standard generalized linear modeling assumptions or more advanced approaches. The function returns estimates of the learned distribution and the associated performance quantity ($Q(\boldsymbol{\pi}^*)$) along with optional inference based on the (asymptotic) delta method.

Usage

```
OptiConjoint(
  Y,
  W,
  X = NULL,
  lambda,
  varcov_cluster_variable = NULL,
  competing_group_variable_respondent = NULL,
  competing_group_variable_candidate = NULL,
  competing_group_competition_variable_candidate = NULL,
  pair_id = NULL,
  respondent_id = NULL,
  respondent_task_id = NULL,
  profile_order = NULL,
  p_list = NULL,
  slate_list = NULL,
  K = 1,
  nSGD = 100,
  diff = FALSE,
  MaxMin = FALSE,
  UseRegularization = FALSE,
  OpenBrowser = FALSE,
  ForceGaussianFamily = FALSE,
  A_INIT_SD = 0,
  TypePen = "KL",
  ComputeSEs = TRUE,
  conda_env = NULL,
  conda_env_required = FALSE,
  confLevel = 0.90,
  nFolds_glm = 3L,
  folds = NULL,
  nMonte_MaxMin = 5L,
  nMonte_Qglm = 100L,
  UseOptax = FALSE,
  jax_seed = as.integer(Sys.time()),
  OptimType = "tryboth"
)
```

Arguments

`Y` A numeric or binary vector of observed outcomes, typically in `{0,1}` for forced-choice conjoint tasks, indicating whether the profile was selected. For instance, `Y = 1` if candidate A was chosen over candidate B, and `Y = 0` otherwise. The length must match the number of rows in `W`.

`W` A matrix or data frame representing the assigned levels of each factor in a conjoint design (one column per factor). Each row corresponds to a single profile. For forced-choice tasks, a given respondent may have contributed multiple rows if you reshape pairwise choices into long format. If the experiment used multiple factors D, with each factor having L_d levels, `W` should capture all factor assignments accordingly.

`X` An optional matrix or data frame of additional covariates, often respondent-level features (e.g., respondent demographics). If `K > 1`, `X` may be used internally to fit multi-cluster or multi-component outcome models, or to allow cluster-specific effect estimation for more granular insights. Defaults to `NULL`.

`lambda` A numeric scalar or vector giving the regularization penalty (e.g., in Kullback-Leibler or L2 sense) used to shrink the learned probability distribution(s) of factor levels toward a baseline distribution `p_list`. Typically set via either domain knowledge or cross-validation.

`varcov_cluster_variable`
An optional vector of cluster identifiers (e.g., respondent IDs) used to form a robust variance-covariance estimate of the outcome model. If `NULL`, the usual IID assumption is made. Defaults to `NULL`.

`competing_group_variable_respondent`, `competing_group_variable_candidate`
competing_group_competition_variable_candidate Optional variables that mark competition group membership of respondents or candidate profiles. Particularly relevant in adversarial settings (`MaxMin = TRUE`) or multi-stage electoral settings, e.g., capturing the party of each respondent or candidate. Defaults to `NULL`.

`pair_id` A factor or numeric vector identifying the forced-choice pair. If each row of `W` is a single profile, `pair_id` groups the rows belonging to the same choice set. Defaults to `NULL`.

`respondent_id`, `respondent_task_id`
Another set of optional identifiers. `respondent_id` marks each respondent across tasks, while `respondent_task_id` can define unique IDs for repeated measurements from the same respondent across multiple tasks. Useful for advanced clustering or robust SEs. Defaults to `NULL`.

`profile_order` If each forced-choice is shown with different ordering (e.g., `Candidate A` vs. `Candidate B`), `profile_order` can label each row accordingly. Helpful for ensuring consistent labeling of reference vs. opposing profiles. Defaults to `NULL`.

`p_list`	An optional list describing the baseline probability distribution over factor levels in `W`. Typically derived from the initial design distribution or uniform assignment distribution. If `NULL`, the function may assume uniform or attempt to estimate the distribution from `W`.
`slate_list`	An optional list (or lists) providing custom "slates" of candidate features (and their associated probabilities). Used in more advanced or adversarial setups where certain combinations must be included or excluded. If `NULL`, no special constraints beyond the usual factor-level distributions are applied.
`K`	Integer specifying the number of latent clusters for multi-component outcome models. If `K = 1`, no latent clustering is done. Defaults to `1`.
`nSGD`	Integer specifying the number of stochastic gradient descent (or gradient-based) iterations to use when learning the optimal distributions. Defaults to `100`.
`diff`	Logical indicating whether the outcome `Y` represents a first-difference or difference-based metric. In forced-choice contexts, typically `diff = FALSE`. Defaults to `FALSE`.
`MaxMin`	Logical controlling whether to enable the max-min adversarial scenario. When `TRUE`, the function searches for a pair of distributions (one for each competing party or group) such that each party's distribution is optimal given the other party's distribution. Defaults to `FALSE`.
`UseRegularization`	Logical indicating whether to regularize the outcome model (in addition to any penalty `lambda` on the distribution shift). This can help avoid overfitting in high-dimensional designs. Defaults to `FALSE`.
`OpenBrowser`	Logical for debugging; if `TRUE`, the function may open an interactive browser window upon encountering certain conditions. Typically `FALSE`.
`ForceGaussianFamily`	Logical indicating whether to force a Gaussian-based outcome modeling approach, even if `Y` is binary or forced-choice. If `FALSE`, the function attempts to choose a more appropriate link (e.g., `"binomial"`). Defaults to `FALSE`.
`A_INIT_SD`	Numeric scalar specifying the standard deviation for random initialization of unconstrained parameters used in the gradient-based search over factor-level probabilities. Defaults to `0`.
`TypePen`	A character string specifying the type of penalty (e.g., `"KL"`, `"L2"`, or `"LogMaxProb"`) used in the objective function for shifting the factor-level probabilities away from the baseline `p_list`. Defaults to `"KL"`.
`ComputeSEs`	Logical indicating whether standard errors should be computed for the final estimates (via the delta method or related expansions). Defaults to `TRUE`.
`conda_env`	A character string naming a Python conda environment that includes **jax**, **optax**, and other dependencies. If not `NULL`, the function attempts to activate that environment. Defaults to `NULL`.
`conda_env_required`	Logical; if `TRUE`, raises an error if the environment given by `conda_env` cannot be activated. Otherwise, the function attempts to proceed with any available installation. Defaults to `FALSE`.
`confLevel`	Numeric in $(0, 1)$, specifying the confidence level for intervals or credible bounds. Defaults to `0.90`.
`nFolds_glm`	Integer specifying the number of folds (default `3L`) for internal cross-validation used in certain outcome model or regularization steps. Defaults to `3L`.

folds	An optional user-supplied partitioning or CV scheme, overriding nFolds_glm. Defaults to NULL.
nMonte_MaxMin	Integer specifying the number of Monte Carlo samples used in adversarial or max-min steps, e.g., sampling from the opposing candidate's distribution to approximate expected payoffs. Defaults to 5L.
nMonte_Qglm	Integer specifying the number of Monte Carlo samples for evaluating or approximating the quantity of interest under certain outcomes or distributions. Defaults to 100L.
UseOptax	Logical indicating whether to use the optax library for gradient-based optimization in JAX (TRUE) or a built-in method (FALSE). Defaults to FALSE.
jax_seed	Integer seed for reproducible JAX-based computations. Defaults to as.integer(Sys.time()).
OptimType	A character string for choosing which optimizer or approach is used internally (e.g., "default", "SecondOrder", or "tryboth"). Defaults to "tryboth".

Details

Modeling the outcome: Internally, OptiConjoint may fit a generalized linear model or a more flexible approach (such as multi-cluster factorization) to learn the mapping from factor-level assignments W (and optional covariates X) onto outcomes Y. Once these outcome coefficients are estimated, the function uses gradient-based or closed-form solutions to find the *optimal stochastic intervention(s)*, i.e., new factor-level probability distributions that maximize an expected outcome (or solve the max-min adversarial problem).

Adversarial or strategic design: When MaxMin = TRUE, the function attempts to solve a zero-sum game in which one agent (say, "A") chooses its distribution to maximize vote share, while the other ("B") simultaneously chooses its distribution to minimize "A"'s vote share. In many settings, competing_group_variable_respondent and related arguments help define which respondents belong to the "A" or "B" sub-electorate (e.g., a primary). The final solution is a mixed-strategy Nash equilibrium, if it exists, for the forced-choice environment. This can be used to compare or interpret real-world candidate positioning in multi-stage elections.

Regularization: The argument lambda penalizes how far the learned distribution strays from the baseline distribution p_list. This helps avoid overfitting in high-dimensional designs. Different penalty types can be selected via TypePen.

Implementation details: Under the hood, this function may rely on **jax** for automatic differentiation. By default, it uses an internal gradient-based approach. If UseOptax = TRUE, the optax library is used for optimization. The function can automatically detect or load a **conda** environment if specified, though advanced users can pass conda_env_required = TRUE to enforce that environment activation is mandatory.

Value

A named list containing:

PiStar_point An estimate of the (possibly multi-cluster or adversarial) optimal distribution(s) over the factor levels. If MaxMin = TRUE and K = 1, returns a pair of distributions. If K > 1, returns a list, with each element corresponding to a separate cluster.

```
\item{\code{PiStar_se}}{A list of standard errors for the entries of \code{PiStar_point},
computed if \code{ComputeSEs = TRUE}.}

\item{\code{Q_point_mEst}}{Point estimates of the corresponding average or adversarially
optimized outcome, e.g., estimated expected utility or vote share.}
```

```
\item{\code{Q_se_mEst}}{Standard error (if computed) for \code{Q_point_mEst}.}

\item{\code{PiStar_lb}, \code{PiStar_ub}}{Lower and upper bounds for the factor-level
probabilities in \code{PiStar_point}, if \code{ComputeSEs = TRUE} and a confidence
level is provided.}

\item{\code{CVInfo}}{A data frame or list summarizing cross-validation performance for
different values of \code{lambda}, if relevant.}

\item{\code{estimationType}}{A string describing whether the solution was found via a
single-step or multi-step approach (e.g., \code{"TwoStep"}, \code{"OneStep"}).}

\item{\code{...}}{Additional elements storing internal details (e.g., the fitted
outcome model, jacobian matrix, or optimization logs).}
```

References

- Hainmueller, J., Hopkins, D. J., & Yamamoto, T. (2014). Causal Inference in Conjoint Analysis: Understanding Multidimensional Choices via Stated Preference Experiments. *Political Analysis*, 22(1), 1–30.
- Egami, N., & Imai, K. (2019). Causal Interaction in Factorial Experiments: Application to Conjoint Analysis. *Journal of the American Statistical Association*, 114(526), 529–540.
- Goplerud, M., & Titiunik, R. (2022). Analysis of High-Dimensional Factorial Experiments: Estimation of Interactive and Non-Interactive Effects. *arXiv preprint arXiv:2207.XXXX*.
- (Paper Reference) A forthcoming or accompanying manuscript describing in detail the methods for *optimal* or *adversarial* stochastic interventions in conjoint settings.

See Also

`cv.OptiConjoint` for cross-validation across candidate values of `lambda`. See also `OneStep.OptiConjoint` for a function that implements a “one-step” approach to M-estimation of the same target quantity.

Examples

```
## Not run:
  # Suppose we have a forced-choice conjoint dataset with
  # factor matrix W, outcome Y, and baseline probabilities p_list

  # Basic usage: single agent optimizing expected outcome
  opt_result <- OptiConjoint(
      Y = Y,
      W = W,
      lambda = 0.1,
      p_list = p_list,
      MaxMin = FALSE,         # No adversarial component
      TypePen = "KL",         # Kullback-Leibler penalty
      nSGD = 200              # # of gradient descent iterations
  )

  # Inspect the learned distribution and performance
  print(opt_result$PiStar_point)
  print(opt_result$Q_point_mEst)
  print(opt_result$CVInfo)            # If cross-validation was used
  # Adversarial scenario with multi-stage structure
  # E.g., define 'competing_group_variable_respondent' for two parties' supporters
  adv_result <- OptiConjoint(
      Y = Y,
      W = W,
      lambda = 0.2,
      p_list = p_list,
      MaxMin = TRUE,         # Solve zero-sum game across two sets of respondents
      competing_group_variable_respondent = partyID,
      nSGD = 300
  )

  # 'adv_result' now contains distributions for each party's candidate
  # that approximate a mixed-strategy Nash equilibrium
  print(adv_result$PiStar_point$k1)   # Party A distribution
  print(adv_result$PiStar_point$k2)   # Party B distribution

## End(Not run)
```

`strategize.plot` *Implements...*

Description

Implements...

Usage

```
OneStep.OptiConjoint(...)
```

Arguments

`x`	Description

Details

`OneStep.OptiConjoint` Description

- Description

Value

z Description

Examples

```
# Analysis
OptiConjoint_analysis <- OneStep.OptiConjoint()

print( OptiConjoint_analysis )
```

Index