Classes in CLVTools

Patrik Schilter Markus Meierer Jeffrey Näf Patrick Bachmann

September 16, 2025

Abstract

This document provides an overview of the classes used in the R package `CLVTools`. It serves as a technical guide for developers and advanced users who wish to understand the internal object-oriented architecture. The package employs a hybrid S4/S3 approach, using the formal S4 system for its core modeling structure, while leveraging canonical S3 generic methods for a familiar and user-friendly interface.

Contents

1 Overview: An S4 class system with S3 interfaces

The `CLVTools` package is built on S4 classes but offers S3 interfaces to canonical methods. This hybrid approach separates the internal complexity from the user interface.

- **S4 for the core architecture:** The formal and strict S4 system is used to define the core data structures and model behaviors. This provides robustness and a clear inheritance structure what allows us to construct rich and rather complex models.
- **S3 for the user interface:** To ensure the package is intuitive and integrates with the broader R ecosystem, the canonical S3 generics (e.g. `plot()`, `predict()`, `summary()`) are used for the main user-facing functions. For this, the package registers these S3 methods for its S4 classes.

In the following, we discus both in detail.

2 S3 methods: User interface

While the core of the package is the S4 class system, the user interacts with the package primarily through S3 generic functions what allows to write idiomatic R code. The key S3 generics in `CLVTools` include:

- `print()` and `show()`: For concise object display.
- `summary()`: For statistical summaries of fitted models.
- `plot()`: For diagnostic plots of fitted models and of the transaction data itself.
- `predict()`: For generating forecasts and estimating Customer Lifetime Value.
- `coef()`: For extracting estimated model coefficients.
- `vcov()`: For extracting the variance-covariance matrix.
- `logLik()`: For extracting the value of the log-likelihood function at final parameters.
- `nobs()`: For extracting the number of observations (customers).

In addition, there are further methods for which S3 as well as S4 generic methods have been defined and exported:

- `lrtest()`: To conduct a likelihood ratio test of nested models.
- `hessian()`: To calculate the Hessian matrix at final parameters.

3 S4 classes: Internal architecture

The foundation of `CLVTools` is built on a set of well-defined S4 classes that manage data, time, model logic, and results. These can be grouped into four primary hierarchies:

- `clv.data`: Pre-processing and storing transaction and covariate data.
- `clv.time`: Handling all time-related logic.
- `clv.fitted`: Storing all estimation results. Returned to the user for all down-stream interactions.
- `clv.model`: Handling all model-specific logic.

3.1 The `clv.data` hierarchy: Managing transaction and covariate data

The `clv.data` hierarchy is visualized in Figure 1. These classes are responsible for pre-processing, storing, and managing user-provided transaction and covariate data.

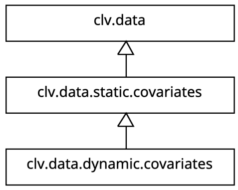

Figure 1: The `clv.data` class hierarchy

Below, we provide a brief description of each class and list – if applicable – their key slots.

`clv.data`
The base class that holds only processed transaction data.

Key Slots

- `data.transactions (data.table)`: The full transaction log.
- `data.repeat.trans (data.table)`: Transaction log of only repeat transactions.
- `clv.time (clv.time)`: For managing time-related logic at the user-defined time unit.

`clv.data.static.covariates (clv.data)`
Extends `clv.data` to include time-invariant covariate data.

Key Slots

- `data.cov.life (data.table)`: Covariate data for the lifetime process.
- `data.cov.trans (data.table)`: Covariate data for the transaction process.
- `names.cov.data.life (character)`: Names of covariates for the lifetime process.
- `names.cov.data.trans (character)`: Names of covariates for the transaction process.

`clv.data.dynamic.covariates (clv.data.static.covariates)`
Extends `clv.data.static.covariates` to handle time-varying covariates. Contains additional columns in the covariate data.

3.2 The `clv.time` hierarchy: Handling time-related logic

The `clv.time` hierarchy is visualized in Figure 2. These classes handle all time-related logic, parsing user inputs, as well as the definitions of estimation and holdout periods. Each class represents a specific time unit and implements methods to do "time-unit math" such as measuring a time span in number of time units. This encapsulation of time unit functionality allows to easily define custom types of time units such as bi-weekly or irregularly spaced time units.

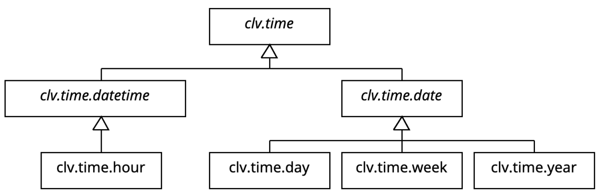

Figure 2: The `clv.time` class hierarchy.

Below, we provide a brief description of each class and list – if applicable – their key slots and key generic methods.

`clv.time (VIRTUAL)`
The base class for all time units.

Key Slots

- `timepoint.estimation.start (ANY)`: Start of the estimation period.
- `timepoint.estimation.end (ANY)`: End of the estimation period.
- `timepoint.holdout.start (ANY)`: Start of the holdout period.
- `timepoint.holdout.end (ANY)`: End of the holdout period.
- `estimation.period.in.tu (numeric)`: Length of the estimation period in time units.
- `holdout.period.in.tu (numeric)`: Length of the holdout period in time units.
- `time.format (character)`: Format used to parse dates and times given as characters.
- `name.time.unit (character)`: Display name of the time unit for output.

Key Generic Methods

- `clv.time.epsilon()`: Minimal possible time step.
- `clv.time.convert.user.input.to.timepoint()`: Convert user-given date/datetimes.
- `clv.time.interval.in.number.tu()`: Measure time between time points in time units.
- `clv.time.number.timeunits.to.timeperiod()`: Create a period of given length time units.
- `clv.time.tu.to.ly()`: Name of time unit with post-fix "ly".
- `clv.time.floor.date()`: Round a time point down to the enclosing time unit.
- `clv.time.ceiling.date()`: Round a time point up to the enclosing time unit.
- `clv.time.format.timepoint()`: Format a given time point as string.

`clv.time.date (clv.time, VIRTUAL)`
Base class for all time units that operate at whole day granularity. Ignores the time of day and stores all data using type `Date`.

`clv.time.datetime (clv.time, VIRTUAL)`
Base class for all time units that operate using `POSIXct`. Processes time and dates at the second level and stores all data using `POSIXct`.

`clv.time.hour (clv.time.datetime)`
Represents a time unit of 1 hour.

`clv.time.day (clv.time.date)`
Represents a time unit of 1 day.

`clv.time.week (clv.time.date)`
Represents a time unit of 1 week.

`clv.time.year (clv.time.date)`
Represents a time unit of 1 year.

3.3 The `clv.fitted` hierarchy: Storing model fitting results

The `clv.fitted` hierarchy is visualized in Figure 3. These classes are the primary objects returned to the user after fitting a model. They store the results of the estimation process and are used for any downstream user interactions such as plotting or predicting.

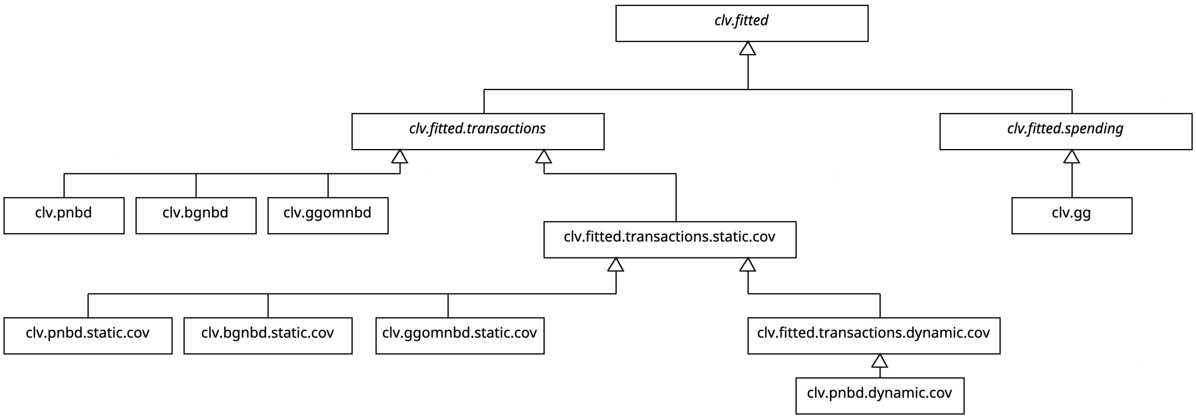

Figure 3: The `clv.fitted` class hierarchy.

Below, we provide a brief description of each class and list – if applicable – their key slots and key generic methods.

`clv.fitted (VIRTUAL)`
Base class for all fitted model results.

Key Slots

- `call (language)`: Call used to fit the model.
- `clv.model (clv.model)`: Object for model-specific behavior.
- `clv.data (clv.data)`: Transaction and covariate data.
- `cbs (data.table)`: Model input data.
- `model.specification.args (list)`: Model specification as given by user.
- `prediction.params.model (numeric)`: Model parameters for downstream usage.
- `optimx.estimation.output (optimx)`: Optimizer output from fitting LL.
- `optimx.hessian (matrix)`: Hessian matrix returned by the optimizer.

Key Generic Methods

- `clv.controlflow.estimate.check.inputs()`: Verify user inputs are valid for estimation.
- `clv.controlflow.estimate.put.inputs()`: Store user inputs for estimation.
- `clv.controlflow.estimate.generate.start.params()`: Generate start parameters if not given.
- `clv.controlflow.estimate.prepare.optimx.args()`: Build call args for LL optimization.
- `clv.controlflow.estimate.process.post.estimation()`: Steps after LL optimization finished.
- `clv.controlflow.check.prediction.params()`: Verify prediction parameters are valid.
- `clv.controlflow.predict.set.prediction.params()`: Set parameters for downstream usage.
- `clv.controlflow.plot.check.inputs()`: Verify user inputs for plotting.
- `clv.fitted.bootstrap.predictions()`: Bootstrap predictions.
- `clv.fitted.estimate.same.specification.on.new.data()`: Re-estimate model on new data.

`clv.fitted.transactions (clv.fitted, VIRTUAL)`
Base class for results of a latent attrition transaction model.

`clv.pnbd (clv.fitted.transactions)`
Results for the standard Pareto/NBD.

`clv.bgnbd (clv.fitted.transactions)`
Results for the standard BG/NBD.

`clv.ggomnbd (clv.fitted.transactions)`
Results for the standard GGom/NBD.

`clv.fitted.transactions.static.cov (clv.fitted.transactions)`
Base class for results of a latent attrition transaction models with static covariates.

Key Slots

- `estimation.used.constraints (logical)`: Whether equality constraints were used.
- `names.original.params.constr (character)`: Display names of constraint covariates.
- `names.original.params.free.life (character)`: Display names of unconstraint lifetime covariates.
- `names.original.params.free.trans (character)`: Display names of unconstraint transaction covariates.
- `names.prefixed.params.constr (character)`: Names during optimization of constraint covariates.
- `names.prefixed.params.free.life (character)`: Names during optimization of unconstraint lifetime covariates.
- `names.prefixed.params.free.trans (character)`: Names during optimization of unconstraint transaction covariates.
- `names.prefixed.params.after.constr.life (character)`: Names during optimization of constraint covariates when they are used for the lifetime process.
- `names.prefixed.params.after.constr.trans (character)`: Names during optimization of constraint covariates when they are used for the transaction process.
- `estimation.used.regularization (logical)`: Whether regularization was used.
- `reg.lambda.life (numeric)`: Regularization lambda used for the lifetime process.
- `reg.lambda.trans (numeric)`: Regularization lambda used for the transaction process.
- `prediction.params.life (numeric)`: Lifetime covariate parameters for downstream usage.
- `prediction.params.trans (numeric)`: Transaction covariate parameters for downstream usage.

`clv.pnbd.static.cov (clv.fitted.transactions.static.cov)`
Results for the Pareto/NBD with static covariates.

`clv.bgnbd.static.cov (clv.fitted.transactions.static.cov)`
Results for the BG/NBD with static covariates.

`clv.ggomnbd.static.cov (clv.fitted.transactions.static.cov)`
Results for the GGom/NBD with static covariates.

`clv.fitted.transactions.dynamic.cov (clv.fitted.transactions.static.cov)`
Base class for results of transaction models with dynamic covariates.

`clv.model.pnbd.dynamic.cov (clv.fitted.transactions.dynamic.cov)`
Results for the Pareto/NBD with dynamic covariates.

`clv.fitted.spending (clv.fitted, VIRTUAL)`
Base class for spending model results.

Key Slots

- `estimation.removed.first.transaction (logical)`: Whether first transactions are counted.

`clv.gg (clv.fitted.spending)`
Results for the standard Gamma/Gamma.

3.4 The `clv.model` hierarchy: Implementing model-specific steps

The `clv.model` hierarchy is visualized in Figure 4. The package architecture separates model-specific logic from the main result object by encapsulating it in a dedicated class. Each result object (subclass of `clv.fitted`) contains an instance of `clv.model` that is called whenever a model-related step is performed (strategy pattern).

This addresses the inheritance challenge that arises when creating result classes for covariate models. If only result classes from the `clv.fitted` hierarchy were used, a covariate model would need to inherit from two parent classes in order to obtain both model-specific functionality and covariate-specific functionality. For example, for the PNBD model with static covariates, the class would need to inherit from both `clv.pnbd` (for PNBD-specific methods) and `clv.fitted.transactions.static.cov` (for covariate handling methods). This creates a diamond inheritance problem where both parents inherit from the common ancestor `clv.fitted.transactions`, leading to method resolution ambiguity. By using composition instead of inheritance and moving all model-specific logic into the `clv.model` classes, model-specific logic can be inherited between covariate and no covariate models (`clv.model.pnbd.static` inherits from `clv.model.pnbd.no.cov`). At the same time, all covariate data logic remains in the result class (`clv.pnbd.static.cov` inherits from `clv.fitted.transactions.static.cov` and in addition, contains an instance of `clv.model.pnbd.static`).

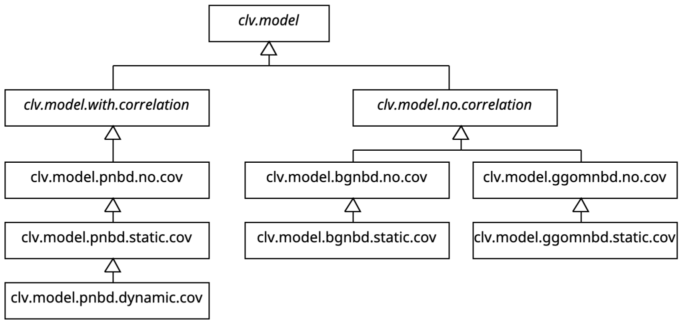

Figure 4: The `clv.model` class hierarchy.

Below, we provide a brief description of each class and list – if applicable – their key slots and key generic methods.

`clv.model(VIRTUAL)`
Base class for model-specific functionalities.

Key Slots

- `name.model (character)`: Display name of the model.
- `fn.model.generic (standardGeneric)`: Method to apply to `clv.data` object to fit the model.
- `names.original.params.model (character)`: Display names of model parameters.
- `names.prefixed.params.model (character)`: Model parameter names during LL optimization.
- `start.params.model (numeric)`: Default start parameters for the LL optimization if none provided.
- `optimx.defaults (list)`: Default arguments with which the optimizer is called.

Key Generic Methods

- `clv.model.check.input.args()`: Verify user inputs are valid for estimation.
- `clv.model.put.estimation.input()`: Store inputs used for estimation.
- `clv.model.prepare.optimx.args()`: Build call args for LL optimization.
- `clv.model.process.post.estimation()`: Steps after LL optimization finished.
- `clv.model.transform.start.params.model()`: Transform model parameters for estimation.
- `clv.model.backtransform.estimated.params.model()`: Reverse transform model parameters.
- `clv.model.transform.start.params.cov()`: Transform covariate parameters for estimation.
- `clv.model.backtransform.estimated.params.cov()`: Reverse transform covariate parameters.
- `clv.model.vcov.jacobi.diag()`: Build Jacobian matrix for variance-covariance corrections.
- `clv.model.cor.to.m()`: Transform correlation to parameter m.
- `clv.model.m.to.cor()`: Transform parameter m to correlation.
- `clv.model.expectation()`: Calculate unconditional expectation.
- `clv.model.predict()`: Calculate all prediction metrics.
- `clv.model.process.newdata()`: Steps to replace `clv.data` object in fitted model result object.
- `clv.model.probability.density()`: Calculate model pdf.

`clv.model.no.correlation (clv.model, VIRTUAL)`
Base class for models without correlation.

`clv.model.with.correlation (clv.model, VIRTUAL)`
Base class for models with correlation.

Key Slots

- `estimation.used.correlation (logical)`: Whether correlation was used.
- `name.prefixed.cor.param.m (character)`: Internal name used during estimation.
- `name.correlation.cor (character)`: Display name for correlation parameter.

`clv.model.pnbd.no.cov (clv.model.with.correlation)`
Logic for the standard Pareto/NBD.

`clv.model.pnbd.static.cov (clv.model.pnbd.no.cov)`
Logic for the Pareto/NBD with static covariates.

`clv.model.pnbd.dynamic.cov (clv.model.pnbd.static.cov)`
Logic for the Pareto/NBD with dynamic covariates.

`clv.model.bgnbd.no.cov (clv.model.no.correlation)`
Logic for the standard BG/NBD.

`clv.model.bgnbd.static.cov (clv.model.bgnbd.static.cov)`
Logic for the BG/NBD with static covariates.

`clv.model.ggomnbd.no.cov (clv.model.no.correlation)`
Logic for the standard GGom/NBD.

`clv.model.ggomnbd.static.cov (clv.model.ggomnbd.no.cov)`
Logic for the GGom/NBD with static covariates.

`clv.model.gg (clv.model.no.correlation)`
Logic for the Gamma/Gamma.

4 Adding a new model

This architecture is designed for extensibility. To add a new probabilistic model (e.g., "NewModel"), the following topics have to be covered:

- **Define the model logic:** A new S4 class for the model's logic (e.g., `clv.model.newmodel.no.cov`) has to be created. This class serves as the "brain" of the new model and should inherit from an appropriate `clv.model` subclass.
- **Define the result object:** A corresponding S4 class to store results has to be defined (e.g., `clv.newmodel`). It must inherit from `clv.fitted.transactions` or a sub-class thereof and acts as the final object returned to the user, bundling the model logic, the data, and estimation results.
- **Create the user entrypoint:** A user-facing S4 generic function, such as `newmodel()`, must be created to act as the entry point for fitting the model. This generic method should dispatch on the class of the `clv.data` object, allowing for different implementations for data with and without covariates.