Code Development in Model Infrastructure

Leigh Alexander and Shannon T. Holloway

2025-04-18

Contents

Introduction

This document is meant to provide a general discussion of the current coding practices followed by the Model Infrastructure Team. It is meant only to serve as a tool to highlight the aspects of code development that we believe are requisite for creating robust and readable code. We welcome your feedback and thoughts.

In Chapter 1, we discuss the key elements of functions using a simple illustrative example. Though the elements are presented in a linear fashion, code development is rarely a linear process! The goal is present each element, discuss its role in the development process, and highlight some considerations that should be included in any development task.

In Chapter 2, we extend the discussion of functions by building a complete tool intended for use across a broad range of object types. We focus on overall design and highlight the pros and cons of common design choices.

In Chapter 3, we transition to one of the most important phases of development – unit testing! Here, we discuss ways to identify what elements of a new tool **need** to be tested, we discuss some techniques we have adopted for streamlining tests, and provide a framework for creating tests within the designs discussed in Chapter 2.

Chapter 4 provides general guidelines for the PR peer review process. We highlight ways to make this process a collaborative learning experience for both the reviewer and developer that yields valid, robust, and efficient tools.

Chapter 1

Elements of a Function

A formulaic approach to code development provides a strong foundation for developing readable, robust, and well-tested code. In this chapter, we will describe a basic pattern that can be followed when implementing new tools for or extending the `somaverse` codebase. There is not "one way" to develop code. The format presented here is not expected to be a one-size-fits-all approach. It is only our intention to present a general framework and highlight the components that, however implemented, are viewed by the authors as critical to any development effort.

1.1 Function Structure

There are 5 key components of a function:

- Usage definition
- Input testing
- Input processing
- Task implementation
- Result construction and testing

To illustrate each of these sections, we will step through the development of a simple function. The following are the general expectations provided, for example, through a Jira task:

Create a tool that fits a model using a random sampling of a provided dataset. The function

- accepts as input a formula, a dataset, and a random seed;
- obtains a random sample (size 100 with replacement) of the provided data;

- uses regression method `lm()` as implemented by the `stats` package to obtain parameter estimates based on the sampled dataset; and
- returns the `lm` value object augmented by attributes specifying the kind of random number generator and the seed.

1.1.1 Usage definition

As we will discuss in Chapter 2, there are multiple design options for implementing a new functionality. For the purposes of this section, we will use a single function and save the discussion of extendable tools for Chapter 2.

Currently, the `somaverse` convention for exported functions is to use camel case for function names, e.g., `myNewFunction()`. For internal functions, i.e., those that are not made available to the user or reside within a function, use camel case preceded by a dot, e.g., `.myInternalFunction()`. In contrast, argument names use a dot to separate words rather than camel case, e.g., `model.formula` or `r.seed`.

Both function and argument names should clearly indicate their purpose but not be verbose.

Naming functions and arguments can be a surprisingly difficult task; it can be a balancing act between clarity, length, and convention. As an example of a confusing argument name choice, consider the `survival::survfit()` function. This function is an S3 method for creating survival curves. The S3 generic is defined to dispatch on argument `formula` – let's assume that the original implementation was a simple function developed for formula objects that was later transitioned to an S3 method to allow for other types of objects. If that is the case, the argument name `formula` for the original function was a clear and concise choice. However, at some point, `survfit()` was extended to be an S3 method within the `survival` package to also allow for other object types, i.e., `Surv`, `coxph`, `matrix`, etc. The decision was made to keep the original formal arguments! Thus, `formula` can now be a formula **OR** a previously fit model, **OR** a matrix, etc.! Further, third-party developers have added methods, e.g., objects of class `coxnet` defined in package `glmnet`. In fact, the `somaverse` has also extended this function to objects of class `coxnet2`. So the original choice of argument name `formula` does not well describe all of the types of objects that can be provided by the user. In this case `object` would have likely been a better choice.

LPT: Don't dwell on the function and input argument names in the beginning. Use names that mean something to YOU. Save the naming debate for the final step. At that point you'll have a clear understanding of exactly what the input arguments mean within your code, how your code may or may not be extendable, etc. Then, discuss those choices with your team and explore the `somaverse` to identify if there are conventions that should be followed or if there are possible future extensions that might influence the choice.

It is also helpful to avoid using common or vague variable names within your code to minimize the chance that a preferred input argument name coincides with a variable name. For example, say a developer opts to use `adat` as an input argument and uses `data` as a variable name within their code. After discussion with the team, they realize that it is more common across the `somaverse` to use `data` as the input argument name. Unfortunately, a simple find/replace is not an option as `data` was used internally and blindly changing names may lead to unexpected consequences. Instead, use more descriptive internal variable names. If you convert the provided data object to a model frame, use for example `model_frame`.

For our example, we opt to use `sampleRegression()` as the function name and `formula`, `data`, and `r.seed` as the required input arguments. If, for example, it is anticipated that this function might be extended to allow a covariate matrix and response vector, this choice of input names would not be the best choice.

```
sampleRegression(formula, data, r.seed = 1234L) {
  ### Input Testing
  ### Input Processing
  ### Task Implementation
  ### Result Construction
}
```

Notice that we elected to include a default value for input `r.seed`. Default values can also be surprisingly tricky to set. For example, the best tolerance bound for converge of an iterative algorithm can vary widely. Default values are often kinder to our users, but care must be taken in selecting the default values and thus the default behaviors.

1.1.2 Input Testing

Ensuring that the inputs provided by the user are exactly what you expect them to be is the first step toward developing robust code. Despite our best efforts to provide clear and thorough documentation, we may not always succeed in clarity, users may not read the entire help page, and providing exhaustive examples is often not feasible w.r.t. build times. **The first defense against misuse of code is to ensure that the information provided by the user is what we expect it to be.**

Immediately following the usage specification, **ALL** of the inputs should be tested for basic characteristics. Are they the correct types? Do they have the correct dimensions? Are the values within range? This is often most easily accomplished using `stopifnot()` with clear descriptions of what the correct structure should be. For example,

```
sampleRegression(formula, data, r.seed = 1234L) {

  ### Input Testing

  stopifnot(
    "`formula` should be of the form LHS ~ RHS." =
      !missing(formula) && inherits(formula, "formula") && length(formula) == 3L,
    "`data` should be a data.frame." =
      !missing(data) && is.data.frame(data),
    "`r.seed` should be a positive integer." =
      is.numeric(r.seed) && is.vector(r.seed) && length(r.seed) == 1L &&
      r.seed > 0 && isTRUE(all.equal(r.seed, as.integer(r.seed)))
  )

  ### Input Processing
  ### Task Implementation
  ### Result Construction
}
```

Here, we ensure that the formula is provided and contains both a response variable and model covariates – information that is necessary for any regression tool. We ensure that the data is provided and is a data.frame object. And finally, that the seed for the random number generator is a positive integer. Notice that we do not check that `r.seed` is provided as we have set a default value.

1.1.3 Input Processing

Once it has been confirmed that the basic structure and contents of the inputs are what we expect in simple terms, we need to perform any non-trivial testing and processing of those inputs before using them to perform the task. Such tests should typically be implemented as early in the function as possible.

The choice of testing as much as possible before performing any calculations is often kinder to the user – imagine if the first step of a task uses a subset of the inputs to complete a 20 minute calculation and only after that calculation is complete do you realize that the inputs required for the next step are invalid!

Be kind to your users. Identify any issues arising from invalid or unexpected inputs as early as you can, especially if a task is time-consuming.

In our example, such an input processing step might be to convert the provided data.frame into a model frame. This processing step ensures that the data contains all of the required model terms as well as the validity of the formula. We can also include an informative message telling the user if cases with missing

data are removed and warn them if the provided data is smaller than the number of samples that we will be taking.

```
sampleRegression(formula, data, r.seed = 1234L) {

  ### Input Testing

  stopifnot(
    "`formula` should be of the form LHS ~ RHS." =
      !missing(formula) && inherits(formula, "formula") && length(formula) == 3L,
    "`data` should be a data.frame." =
      !missing(data) && is.data.frame(data),
    "`r.seed` should be a positive integer." =
      is.numeric(r.seed) && is.vector(r.seed) && length(r.seed) == 1L &&
      r.seed > 0 && isTRUE(all.equal(r.seed, as.integer(r.seed)))
  )

  ### Input Processing

  # check to ensure that all required `formula` terms are provided in `data`
  model_frame <- tryCatch(stats::model.frame(formula, data, na.action = na.omit),
                          error = function(e) {
                            stop("Unable to extract model.frame.\n\t",
                                 e$message, call. = FALSE)
                          })

  # if NAs are present in data, inform user
  if ( nrow(data) != nrow(model_frame) ) {
    message(nrow(data) - nrow(model_frame), " cases removed due to NA values.")
  }

  # warn user if data is particularly small
  # using model_frame to avoid counting NA cases in data
  if ( nrow(model_frame) < 100L ) {
    warning("Provided `data` may be too small for sampling.", call. = FALSE)
  }

  ### Task Implementation
  ### Result Construction
}
```

Notice in the above code that we have opted to save the model frame as a new variable, `model_frame`. However, if `data` is large, this is not a good design choice – there will now be two large objects in the environment. Because our focus is only on the structure of functions, we have opted for clarity of variable names over conservation of memory.

Note that each step is heavily documented. Why is the subsequent block of code included? Are there any assumptions we are making? What specifically are we checking? Though some of these documentation lines may seem excessive and may not remain in the final code, they help during development, especially during the development of unit tests.

It may not always be necessary to process or further test inputs before executing the task.

1.1.4 Task Implementation

At this stage, you should be as confident as possible that the provided inputs are valid and in an expected form. The details of this section of a function depend on the task being implemented, which we will not attempt to cover. Some things to consider as you develop this piece are:

- **Test to ensure that every result is valid before using or returning.** Do values fall within the expected range? Are there NA, NaN, or Inf values? Is the result always going to be the expected type (matrix, vector, list, etc.)?

- **Wrap calls to other functions in some type of `tryCatch()`, `be_quiet()`, etc.** This allows for more informative messaging to the user (and to developers if debugging is required).
- **Document!** This cannot be stressed enough! Indicate what you expect the result to be. Describe any non-standard choices you make. The goal of documentation is to provide the *next* developer with all the information they need to understand exactly what and why.

```r
sampleRegression(formula, data, r.seed = 1234L) {

  ### Input Testing

  stopifnot(
    "`formula` should be of the form LHS ~ RHS." =
      !missing(formula) && inherits(formula, "formula") && length(formula) == 3L,
    "`data` should be a data.frame." =
      !missing(data) && is.data.frame(data),
    "`r.seed` should be a positive integer." =
      is.numeric(r.seed) && is.vector(r.seed) && length(r.seed) == 1L &&
      r.seed > 0 && isTRUE(all.equal(r.seed, as.integer(r.seed)))
  )

  ### Input Processing
```

```
  # check to ensure that all required `formula` terms are provided in `data`
  model_frame <- tryCatch(stats::model.frame(formula, data),
                          error = function(e) {
                            stop("Unable to extract model.frame.\n\t",
                                 e$message, call. = FALSE)
                          })

  # if NAs are present in data, inform user
  if ( nrow(data) != nrow(model_frame) ) {
    message(value(nrow(data) - nrow(model_frame)),
            " cases removed due to NA values.")
  }

  # warn user if data is particularly small
  # using model_frame to avoid possible NA cases in data
  if ( nrow(model_frame) < 100L ) {
    warning("Provided `data` may be too small for sampling.", call. = FALSE)
  }

  ### Task Implementation

  # sample (with replacement) the complete cases data
  data_sample <- withr::with_seed(r.seed,
                                  sample(nrow(model_frame), 100L, replace = TRUE))

  # obtain parameter estimates
  # pass-on warnings such as NA coefficients, but stop on error
  fit <- tryCatch(stats::lm(formula, model_frame[data_sample, ]),
                  error = function(e) {
                    stop("Unable to obtain parameter estimates.\n\t",
                         e$message, call. = FALSE)
                  })

  ### Result Construction
}
```

1.1.5 Result Construction and Testing

Once a task is complete, you will often need to organize the results and ensure that the values being returned are both valid and match the documentation. Again, the details are task dependent, and we do not attempt to cover those bases here.

For our example, we simply need to add the requested attributes to the value object returned by `stats::lm()`, i.e., the seed used for the sampling and the

random number generator.

```
sampleRegression(formula, data, r.seed = 1234L) {

  ### Input Testing

  stopifnot(
    "`formula` should be of the form LHS ~ RHS." =
      !missing(formula) && inherits(formula, "formula") && length(formula) == 3L,
    "`data` should be a data.frame or a soma_adat." =
      !missing(data) && (is.data.frame(data) || is.soma_adat(data)),
    "`r.seed` should be a positive integer." =
      is.numeric(r.seed) && is.vector(r.seed) && length(r.seed) == 1L &&
      r.seed > 0 && isTRUE(all.equal(r.seed, as.integer(r.seed)))
  )

  ### Input Processing

  # check to ensure that all required `formula` terms are provided in `data`
  model_frame <- tryCatch(stats::model.frame(formula, data),
                          error = function(e) {
                            stop("Unable to extract model.frame.\n\t",
                                 e$message, call. = FALSE)
                          })

  # if NAs are present in data, inform user
  if ( nrow(data) != nrow(model_frame) ) {
    message(value(nrow(data) - nrow(model_frame)),
            " cases removed due to NA values.")
  }

  # warn user if data is particularly small
  # using model_frame to avoid possible NA cases in data
  if ( nrow(model_frame) < 100L ) {
    warning("Provided `data` may be too small for sampling.", call. = FALSE)
  }

  ### Task Implementation

  # sample (with replacement) the complete cases data
  data_sample <- withr::with_seed(r.seed,
                                  sample(nrow(model_frame), 100L, replace = TRUE))

  # obtain parameter estimates
  # pass-on warnings such as NA coefficients, but stop on error
  fit <- tryCatch(stats::lm(formula, model_frame[data_sample, ]),
```

```
                    error = function(e) {
                      stop("Unable to obtain parameter estimates.\n\t",
                           e$message, call. = FALSE)
                    })

  ### Result Construction and Testing

  # add attributes to fit indicating random generator and seed
  attr(fit, "RNGkind") <- RNGkind()
  attr(fit, "seed")    <- r.seed

  fit
}
```

1.1.6 Concluding Remarks

Despite our presentation here, tool development is not a linear process. The procedures for implementing new tools will vary as widely as the types of tools we implement! There are times when it may be easiest to first work out the details of the task itself and then work your way through the remaining components of the function. Or maybe the tool evolves as you develop the details, requiring changes to the original specification as well as the various components. Our object here is not to provide a formula for *how* to develop code, but to give an ingredients list for what a robust and readable implementation should contain.

Chapter 2

Extended Function Example

Let's assume that the following task is to be implemented:

- Provided
  - a fitted model (`object`),
  - data (`newdata`),
  - evaluation time (`eval.time`), and
  - a vector of bin boundaries (`bin.boundaries`)
- Obtain estimated survival probabilities for `newdata`
- Return a list containing
  - the number of cases with predictions that fall within each bin
  - a vector indicating the bin in which each case in `newdata` falls
  - the user-specified bin structure

Ideally, our new tool would support all currently used survival models in the `somaverse`. However, for the sake of brevity, we will limit the supported models to only:

- AFT as implemented by `survival::survreg()` (`survreg` objects)
- Elastic Net AFT as implemented by `SomaSurvival::fitSurvregnet()` (`survregnet` objects)
- Cox proportional hazards as implemented by `survival::coxph()` (`coxph` objects)

2.1 Procedure Specification

Before writing any code, we should have a clear picture of how the task will be accomplished. Think of this step as writing the headers for an outline. Such headers should be clear and concise – describe the step, but avoid including details of *how* the step will be accomplished.

An example of such a procedure outline for this tool could be:

- Test validity of inputs
- Predict survival probabilities
- Identify bin membership based on the predicted survival probabilities
- Tally the number of cases in each probability bin
- Create a list containing the bin boundaries, bin membership, and totals
- Verify and return list

2.2 Select a Design

There are two primary design structures for extendable tools currently used in the `somaverse`:

- An S3 based structure, where a generic specifies the availability of a tool, and individual S3 methods are implemented for each supported object type. Often the individual S3 methods are responsible for pre-processing of inputs, and each method calls a common "core" function that implements the primary functionality; and
- A framework based solution that uses a single function to specify the general framework and uses supporting utility functions (possibly S3) to extend the framework to each object type supported.

There are pros and cons to each of these design choices. We will discuss these as we work through illustrative examples of both designs.

2.3 S3 Based Design Structure

2.3.1 Usage Specification

The core elements of this design structure are the generic, default, and extension methods. For our example, there will be at least five functions with the following procedures:

- A generic method

  - `survProbBins(object, ...)`.

- A default method
  - `survProbBins.default(object, ...)`.
    * Generate error indicating that `object` is not supported
- An extension method for `coxph` objects
  - `survProbBins.coxph(object, newdata, eval.time, bin.boundaries, ...)`.
    * Test validity of inputs
    * Predict survival probabilities
    * Identify bin membership based on the predicted survival probabilities
    * Tally the number of cases in each probability bin
    * Create a list containing the bin boundaries, bin membership, and totals
    * Return list
- An extension method for `survreg` objects
  - `survProbBins.survreg(object, newdata, eval.time, bin.boundaries, ...)`.
    * Test validity of inputs
    * Predict survival probabilities
    * Identify bin membership based on the predicted survival probabilities
    * Tally the number of cases in each probability bin
    * Create a list containing the bin boundaries, bin membership, and totals
    * Return list
- An extension method for `survregnet` objects
  - `survProbBins.survregnet(object, newdata, eval.time, bin.boundaries, lambda, ...)`.
    * Test validity of inputs
    * Call `convert2Survreg()` to convert `object` to a `survreg` object
    * Call `survProbBins.survreg()` using converted object.
    * Return `.survProbBins.survreg()` value object

Notice that the method for objects of class `survregnet` is a bit different from the other extension methods. Specifically, we will convert the object to class `survreg` and call the `survreg` method. The S3 based design lends itself to situations where simple conversions can be exploited to minimize code duplication – for example, numeric vector to matrix.

Now that we have a general outline of the minimum components for this design, take a moment to identify any overlaps between methods. Are there common

steps that may benefit from being implemented as common internal functions? Above, we see that two of the extension methods have exactly the same tasks. Can these be implemented as common internal functions?

- Test validity of inputs
  - This step might be different for each model type. For example, Elastic Net models will require different inputs that non-Elastic Net models. In this design, it is typically best to keep input testing local to each object type.
- Predict survival probabilities
  - This step will be different for each model type. The survival probability for the Cox model is not directly obtained from the `survival::predict()` function, as is the case for the AFT model. This step cannot be a common internal function.
- Identify bin membership based on the predicted survival probabilities Tally the number of cases in each probability bin Create a list containing the bin boundaries, bin membership, and totals
  - Binning and tallying a vector of predicted probabilities is not dependent on *how* the vector of predictions was obtained. These steps can be implemented separately.

Let's define common function `.binData()` to implement the three steps related to binning and tallying the predicted probabilities. Our design outline is then

- A generic method
  - `survProbBins(object, ...)`.
- A default method
  - `survProbBins.default(object, ...)`.
    - Generate error indicating that `object` is not supported
- An extension method for `coxph` objects
  - `survProbBins.coxph(object, newdata, eval.time, bin.boundaries, ...)`.
    - Test validity of inputs
    - Predict survival probabilities
    - Call .binData()
    - Return .binData() value object
- An extension method for `survreg` objects
  - `survProbBins.survreg(object, newdata, eval.time, bin.boundaries, ...)`.

    * Test validity of inputs
    * Predict survival probabilities
    * Call .binData()
    * Return .binData() value object

- An extension method for `survregnet` objects
  - `survProbBins.survregnet(object, newdata, eval.time, bin.boundaries, lambda, ...)`.
    * Test validity of inputs
    * Call `convert2Survreg()` to convert `object` to a `survreg` object
    * Call `survProbBins.survreg()` using converted object.
    * Return `.survProbBins.survreg()` value object
- An internal function for binning and tallying a vector of values
  - .binData(values, bin.boundaries).
    * Test validity of inputs
    * Identify bin membership based on the provided probabilities
    * Tally the number of cases in each probability bin
    * Create a list containing the bin boundaries, bin membership, and totals
    * Return list

As a next step, try to identify points at which intermediate results should be tested or verified prior to their use. For our example, it would be a good idea to ensure that the estimated survival probabilities obey general characteristics, such as all in range [0, 1], and to explicitly address cases for which this expectation is not met. There are a few options for this:

1. Incorporate tests into `.binData()` to ensure that the provided values have the expected characteristics of a probability or
2. Add a testing step between "Predict survival probabilities" and "Call `.binData()`"

Selecting option 1 limits `.binData()` to be applicable only to vectors containing values in [0, 1]. Option 2 introduces an additional step and thus function into our design that must be fully unit tested. Because there is no explicit reason why `.binData()` should be limited to probabilities (perhaps such a binning functionality will be useful elsewhere in the future!), option 2 is a more general solution. Specifically, include an internal function that ensures the validity of the predicted survival probabilities and call it before passing the predicted probabilities to `.binData()`.

- A generic method
  - `survProbBins(object, ...)`.

- A default method
  - `survProbBins.default(object, ...)`.
    - Generate error indicating that `object` is not supported
- An extension method for `coxph` objects
  - `survProbBins.coxph(object, newdata, eval.time, bin.boundaries, ...)`.
    - Test validity of inputs
    - Predict survival probabilities
    - Call .testProbabilities()
    - Call `.binData()`
    - Return `.binData()` value object
- An extension method for `survreg` objects
  - `survProbBins.survreg(object, newdata, eval.time, bin.boundaries, ...)`.
    - Test validity of inputs
    - Predict survival probabilities
    - Call .testProbabilities()
    - Call `.binData()`
    - Return `.binData()` value object
- An extension method for `survregnet` objects
  - `survProbBins.survregnet(object, newdata, eval.time, bin.boundaries, lambda, ...)`.
    - Test validity of inputs
    - Call `convert2Survreg()` to convert `object` to a `survreg` object
    - Call `survProbBins.survreg()` using converted object.
    - Return `.survProbBins.survreg()` value object
- An internal function for binning and tallying a vector of values
  - `.binData(values, bin.boundaries)`.
    - Test validity of inputs
    - Identify bin membership based on the provided probabilities
    - Tally the number of cases in each probability bin
    - Create a list containing the bin boundaries, bin membership, and totals
    - Return list
- An internal function for testing validity of survival probabilities
  - .testProbabilities(predictions)
    - Ensure predicted probabilities are all finite and in [0,1]

We now have an outline of our design and can begin filling in the details. At this stage, we will convert the design outline to skeleton code and outline the key steps of each function using only documentation.

2.3.1.0.1 S3 Generic

The first function we must define is the "generic"

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, ...) { UseMethod("survProbBins") }
```

This function establishes `survProbBins()` as an S3 method that dispatches on the class structure of input `object`. Note that it is not necessary (nor is it advised) to specify any additional input arguments when declaring an S3 generic.

We often use this function as the primary documentation point for the method. Thus, the function title, description, dispatch argument, and expected returned value have been documented here.

The dispatch argument, `object`, could have been set as `fitted.model` based solely on the problem specification. But, we can imagine a circumstance where the method might be extended to allow a user to provide the vector of predictions rather than requiring a fitted model; in that usage, `fitted.model` would be a confusing input name, so we opt to use a more general name.

2.3.1.1 S3 Default

Next, we will define the "default" method. This method is the "last resort" method, meaning it will be triggered if `object` does not inherit from any of the types for which we have extended the generic method. Suppose that the class structure of `object` is `c("foo", "bar")`. When `survProbBins(object)` is evaluated, `R` will follow the following path to identify the correct S3 method:

- `survProbBins.foo()`
- `survProbBins.bar()`
- `survProbBins.default()`

If `survProbBins.foo()` is not defined, `R` looks for `survProbBins.bar()`. If `survProbBins.bar()` does not exist, `R` will dispatch the default method. Often the default method is used to generate an informative error message indicating that the function does not support the object provided by the user, as we have elected to implement here. However, this behavior is not a requirement; the default method can perform calculations and return a result.

```
#' S3 default method -- error only
#' @rdname survProbBins
#' @export
survProbBins.default <- function(object, ...) {
  stop("`survProbBins()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}
```

2.3.1.2 S3 Extensions

With the generic and default methods in place, we can now specify extensions of the method for each supported class of `object`, i.e., `coxph`, `survreg`, and `survregnet`. We elect to have these extensions described in the documentation of the generic (`@rdname survProbBins`) and incorporate new parameter documentation when new input parameters are introduced by a specific method, e.g., `lambda` of `survProbBins.survregnet()`. It is also possible to define all parameters in the generic documentation.

```
#' S3 method for objects returned by `survival::coxph()`
#' @rdname survProbBins
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @export
survProbBins.coxph <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # Verify that `newdata`, `eval.time` and `bin.boundaries` are provided
  # Verify that `newdata` is a data.frame or soma_adat
  # Verify that `newdata` contains the appropriate data
  # Verify that `eval.time` is a positive scalar
```

```
  # Verify that `bin.boundaries` is a numeric vector in [0, 1]

  ### Input Processing
  # Predict survival probabilities
  # Call .testProbabilities() to ensure valid predictions

  ### Task Implementation
  # Call `.binData()` to bin and tally predicted probabilities

  ### Result Construction and Testing
  # Return the `.binData()` value object
}

#' S3 method for objects returned by `survival::survreg`
#' @rdname survProbBins
#' @export
survProbBins.survreg <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # Verify that `newdata`, `eval.time`, and `bin.boundaries` are provided
  # Verify that `newdata` is a data.frame or soma_adat
  # Verify that `newdata` contains the appropriate data
  # Verify that `eval.time` is a positive scalar
  # Verify that `bin.boundaries` is a numeric vector in [0, 1]

  ### Input Processing
  # Predict survival probabilities
  # Call .testProbabilities() to ensure valid predictions

  ### Task Implementation
  # Call `.binData()` to bin and tally predicted probabilities

  ### Result Construction and Testing
  # Return the `.binData()` value object
}

#' S3 method for objects returned by `SomaSurvival::fitSurvregnet`
#' @rdname survProbBins
#' @param lambda Numeric. The Elastic Net parameter, lambda, of the model.
#' @export
survProbBins.survregnet <- function(object, newdata, eval.time, bin.boundaries,
                                    lambda, ...) {
  ### Input Testing
  # Verify that `newdata`, `eval.time`, `bin.boundaries`, and `lambda` are
  #   provided
  # Verify that `newdata` is a data.frame or soma_adat
```

```
  # Verify that `newdata` contains the appropriate data
  # Verify that `eval.time` is a positive scalar
  # Verify that `bin.boundaries` is a numeric vector in [0, 1]
  # Verify that `lambda` is non-negative scalar

  ### Input Processing
  # Call convert2survreg() to convert to a `survreg` object

  ### Task Implementation
  # Call survProbBins.survreg() passing converted object

  ### Result Construction and Testing
  # Return the `.survProbBins.survreg()` value object
}
```

An alternative and more concise usage specification for `survProbBins.survregnet()` is

```
#' S3 method for objects returned by `SomaSurvival::fitSurvregnet`
#' @rdname survProbBins
#' @param lambda Numeric. The Elastic Net parameter of the model.
#' @export
survProbBins.survregnet <- function(object, lambda, ...) {
  ### Input Testing
  # Verify that `lambda` is provided and is a non-negative scalar

  ### Input Processing
  # Call convert2survreg() to convert to a `survreg` object

  ### Task Implementation
  # Call survProbBins.survreg() passing converted object and ...

  ### Result Construction and Testing
  # Return `survProbBins.survreg()` value object
}
```

where the ellipsis (`...`) is assumed to contain the additional arguments required and used by `survProbBins.survreg()`, i.e., `newdata`, `eval.time`, and `bin.boundaries`. Though this is "tidier" in the sense that we only worry about the inputs that `survProbBins.survregnet()` actually *uses*, this is a forward-facing function, and requiring users to figure out what needs to be passed through the ellipsis can be an unkind burden.

2.3.1.3 Common Functions

The required task, bin the probabilities, and the common task of verifying that the survival probabilities are valid will be accomplished through common utility functions.

```
#' Internal function to test that probabilities are valid
#'
#' @noRd
#' @param predictions Numeric. The probabilities to test.
#' @return Numeric. The probabilities with invalid values reset to NA_real_
#' @keywords internal
.testProbabilities <- function(predictions) {
  ### Input Testing
  # Verify that `predictions` is provided and is a non-empty numeric vector

  ### Input Processing

  ### Task Implementation
  # Check for non-finite (NA, NaN, Inf) values
  # Reset non-finite values to NA_real_ and tell user
  #
  # Check for values outside of [0, 1]
  # Reset out of range values to NA_real_ and tell user

  ### Result Construction and Testing
  # Return modified probabilities as an unnamed numeric vector
}

#' Internal function to perform binning
#'
#' Provided a vector of values and a vector of bin boundaries, identify the
#'   bin in which each value falls and the total number of values in each bin.
#'
#' @noRd
#' @param values A numeric or numeric vector. The values to bin and tally.
#' @param bin.boundaries Numeric vector. The K+1 boundaries of K bins.
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @keywords internal
.binData <- function(values, bin.boundaries) {
```

```
  ### Input Testing
  # Verify that `values` is provided
  # Verify that `values` is a numeric vector
  # Verify that `values` does not contain NA/NaN/Inf
  # Verify that `bin.boundaries` is provided
  # Verify that `bin.boundaries` is a numeric vector of length > 1
  # Verify that `bin.boundaries` are all unique values
  # Verify that `bin.boundaries` does not contain NA or NaN values
  # Verify that `bin.boundaries` are in increasing order
  # Verify that `bin.boundaries` cover range of `values`

  ### Input Processing

  ### Task Implementation
  # Identify bin membership for each value
  # Tally the number of cases in each bin

  ### Result Construction and Testing
  # Return a list with elements `bin.boundaries`, `bin.id`, and `totals`
}
```

2.3.1.4 General Comments

Notice that each function adheres to the function structure described in Chapter 1 – specifically, usage definition is detailed by our design outline, and each function includes input testing, input processing, task implementation, and result construction. Each component is heavily documented. The usage definition is detailed through the `roxygen` documentation and the internal structure of each function is detailed through comments. Because we cannot always anticipate every line of code, we include all of the components regardless of whether or not we anticipate their necessity at this stage. For example, in `.binData()`, there are currently no details for component "Input Processing." Keeping the header here can serve as a reminder to ensure that we have not overlooked something as we continue to fill in the details.

For purely internal functions, such as `.binData()`, the input testing element may seem superfluous – "Only *I* am calling this function"; "I *know* what is being passed is valid"; "I've already checked most of this"; etc. However, mistakes happen, things get overlooked, maybe another developer decides this function will be helpful for another tool. Unless a test is time- or memory-intensive, it doesn't hurt to verify that you are getting what you need!

Further, the specific characteristics of the inputs that are tested in each function are not necessarily the same. For example, in `survProbBins.*()` we do not *use* `bin.boundaries` and thus only test that it is provided, that it is of

the type expected by `.binData()`, and this is contains values as required for survival probabilities (in range [0,1]). We reserve much of the testing, such as uniqueness, increasing, etc., for the function `.binData()`, which **uses** that input. This is not a hard and fast rule; remember, we also want to stop a function as soon as we know that there is a problem. There will certainly be circumstances under which it is kinder to the user to more extensively test inputs well before they will be used. However, keep in mind that incorporating tests for values outside of the function in which they are used could lead to issues if the function is later extended. Say for example, we add tests for `NA` values into the `survProbBins.*()` functions. If at a later time it is decided that a better implementation is to simply report the `NA` values rather than stop with an error, we would need to change not only the `.binData()` function to accommodate this change but also the `survProbBins.*()` functions to remove the tests for the `NA` condition.

2.3.2 Input Testing

For the S3 based design, the testing of the input `object` happens automatically through the specification of the individual methods – if a method is not implemented for objects of class `xyz`, the default method `survProbBins.default()` will trigger indicating that the object is not supported.

```
#' Bin Predicted Survival Probabilities
#'
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, ...) { UseMethod("survProbBins") }
```

```
#' S3 default method -- error only
#' @noRd
#' @export
survProbBins.default <- function(object, ...) {
  stop("`survProbBins()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' S3 method for objects returned by `survival::coxph()`
#' @rdname survProbBins
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @export
survProbBins.coxph <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  # Predict survival probabilities
  # Call .testProbabilities() to ensure valid predictions

  ### Task Implementation
  # Call `.binData()` to bin and tally predicted probabilities

  ### Result Construction and Testing
  # Return `.binData()` value object
}
```

```r
#' S3 method for objects returned by `survival::survreg`
#' @noRd
#' @export
survProbBins.survreg <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
  )

  ### Input processing
  # Predict survival probabilities
  # Call .testProbabilities() to ensure valid predictions

  ### Task Implementation
  # Call `.binData()` to bin and tally predicted probabilities

  ### Result Construction and Testing
  # Return `.binData()` value object
}

#' S3 method for objects returned by `SomaSurvival::fitSurvregnet`
#' @noRd
#' @param lambda Numeric. The Elastic Net parameter, lambda, of the model.
#' @export
survProbBins.survregnet <- function(object, newdata, eval.time, bin.boundaries,
                                    lambda, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
```

```
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
      all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
    "`lambda` must be a non-negative scalar." =
      !missing(lambda) && is.numeric(lambda) &&
      is.vector(lambda) && length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Call convert2survreg() to convert to `survreg` object

  ### Task Implementation
  # Call survProbBins.survreg() passing converted object

  ### Result Construction and Testing
  # Return `survProbBins.survreg()` value object
}

### Common Utility Functions

#' Internal function to test that probabilities are valid
#'
#' @noRd
#' @param predictions Numeric. The probabilities to test.
#' @return Numeric. The probabilities with invalid values reset to NA_real_
#' @keywords internal
.testProbabilities <- function(predictions) {
  ### Input Testing
  stopifnot(
    "`predictions` must be numeric." =
      !missing(predictions) && is.numeric(predictions) &&
      is.vector(predictions) && length(predictions) != 0L
  )

  ### Input Processing

  ### Task Implementation
  # Check for non-finite (NA, NaN, Inf) values
  # Reset non-finite values to NA_real_ and tell user
  #
  # Check for values outside of [0, 1]
  # Reset out of range values to NA_real_ and tell user

  ### Result Construction and Testing
```

```r
  # Return modified probabilities as an unnamed numeric vector
}

#' Internal function to perform binning
#'
#' Provided a vector of values and a vector of bin boundaries, identify the
#'   bin in which each value falls and the total number of values in each bin.
#'
#' @noRd
#' @param values A numeric or numeric vector. The values to bin and tally.
#' @param bin.boundaries Numeric vector. The K+1 boundaries of K bins.
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @keywords internal
.binData <- function(values, bin.boundaries) {
  ### Input Testing
  stopifnot(
    "`values` must be a non-empty, numeric vector." =
      !missing(values) && is.numeric(values) && is.vector(values) &&
        length(values) != 0L,
    "`bin.boundaries` must be a numeric vector of length > 1." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && is.vector(bin.boundaries),
    "All values in `bin.boundaries` must be unique." =
      length(bin.boundaries) == length(unique(bin.boundaries)),
    "`bin.boundaries` cannot contain NA or NaN values." =
      all(!is.na(bin.boundaries) && !is.nan(bin.boundaries)),
    "`bin.boundaries` must be in increasing order." = !is.unsorted(bin.boundaries),
    "`values` must be finite and in range of `bin.boundaries`." =
      all(is.finite(values)) && all(values >= min(bin.boundaries)) &&
      all(values <= max(bin.boundaries))
  )

  ### Input Processing

  ### Task Implementation
  # Identify bin membership based on the provided probabilities
  # Tally the number of cases in each probability bin

  ### Result Construction and Testing
```

```
  # Return a list with elements `bin.boundaries`, `bin.id`, and `totals`
}
```

2.3.3 Input Processing

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, ...) { UseMethod("survProbBins") }

#' S3 default method -- error only
#' @noRd
#' @export
survProbBins.default <- function(object, ...) {
  stop("`survProbBins()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' S3 method for objects returned by `survival::coxph()`
#' @noRd
#' @export
survProbBins.coxph <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
```

```r
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  ## Predict survival probabilities

  # retrieve baseline hazard
  base_hazard <- survival::basehaz(object)
  # identify time-point in baseline hazard nearest the specified evaluation time
  t_idx       <- which.min(abs(eval.time - base_hazard[, 2L]))

  base_hazard_at_t <- base_hazard[t_idx, 1L]

  # ensure that baseline hazard is a valid value (not NA, NaN, Inf)
  if ( !is.finite(base_hazard_at_t) ) {
    stop("Invalid baseline hazard ", value(base_hazard_at_t), call. = FALSE)
  }
  message("Survival probability evaluated at t = ", value(base_hazard[t_idx, 2L]))

  # get linear predictors for newdata
  lin_predictors <- tryCatch(predict(object, newdata, type = "lp"),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                               })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function
  surv_probabilities <- exp(-base_hazard_at_t * exp(lin_predictors))

  # Ensure valid predictions
  surv_probabilities <- .testProbabilities(surv_probabilities)

  ### Task Implementation
  # Call `.binData()` to bin and tally predicted probabilities

  ### Result Construction and Testing
```

```
  # Return `.binData()` value object
}

#' S3 method for objects returned by `survival::survreg`
#' @noRd
#' @export
survProbBins.survreg <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  ## Predict survival probabilities

  # retrieve linear predictors
  lin_predictors <- tryCatch(predict(object, newdata, type = "lp"),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                             })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function

  surv_probabilities <- 1.0 - stats::pweibull(eval.time,
                                              shape = 1.0 / exp(object$scale),
                                              scale = exp(lin_predictors))
  # Ensure valid predictions
  surv_probabilities <- .testProbabilities(surv_probabilities)

  ### Task Implementation
  # Call `.binData()` to bin and tally predicted probabilities

  ### Result Construction and Testing
  # Return `.binData()` value object
```

```
}

#' S3 method for objects returned by `SomaSurvival::fitSurvregnet`
#' @noRd
#' @export
survProbBins.survregnet <- function(object, newdata, eval.time, bin.boundaries,
                                    lambda, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
      all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
    "`lambda` must be a non-negative scalar." =
      !missing(lambda) && is.numeric(lambda) &&
      is.vector(lambda) && length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Convert to survreg object
  survreg_object <- tryCatch(SomaSurvival::convert2Survreg(object, lambda),
                             error = function(e) {
                               stop("Unable to convert `object` to `survreg`.\n\t",
                                    e$message, call. = FALSE)
                             })

  ### Task Implementation
  # Call survProbBins.survreg() passing converted object

  ### Result Construction and Testing
  # Return `survProbBins.survreg()` value object
}

#' Internal function to test that probabilities are valid
#'
#' @noRd
#' @param predictions Numeric. The probabilities to test.
#' @return Numeric. The probabilities with invalid values reset to NA_real_
```

```
#' @keywords internal
.testProbabilities <- function(predictions) {
  ### Input Testing
  stopifnot(
    "`predictions` must be numeric." =
      !missing(predictions) && is.numeric(predictions) &&
      is.vector(predictions) && length(predictions) != 0L
  )

  ### Input Processing

  ### Task Implementation
  # Check for non-finite (NA, NaN, Inf) values
  # Reset non-finite values to NA_real_ and tell user
  #
  # Check for values outside of [0, 1]
  # Reset out of range values to NA_real_ and tell user

  ### Result Construction and Testing
  # Return modified probabilities as an unnamed numeric vector
}

#' Internal function to perform binning
#'
#' Provided a vector of values and a vector of bin boundaries, identify the
#'   bin in which each value falls and the total number of values in each bin.
#'
#' @noRd
#' @param values A numeric or numeric vector. The values to bin and tally.
#' @param bin.boundaries Numeric vector. The K+1 boundaries of K bins.
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @keywords internal
.binData <- function(values, bin.boundaries) {
  ### Input Testing
  stopifnot(
    "`values` must be a non-empty, numeric vector." =
      !missing(values) && is.numeric(values) && is.vector(values) &&
        length(values) != 0L,
    "`bin.boundaries` must be a numeric vector of length > 1." =
```

```
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && is.vector(bin.boundaries),
    "All values in `bin.boundaries` must be unique." =
      length(bin.boundaries) == length(unique(bin.boundaries)),
    "`bin.boundaries` cannot contain NA or NaN values." =
      all(!is.na(bin.boundaries) && !is.nan(bin.boundaries)),
    "`bin.boundaries` must be in increasing order." = !is.unsorted(bin.boundaries),
    "`values` must be finite and in range of `bin.boundaries`." =
      all(is.finite(values)) && all(values >= min(bin.boundaries)) &&
      all(values <= max(bin.boundaries))
  )

  ### Input Processing

  ### Task Implementation
  # Identify bin membership based on the provided probabilities
  # Tally the number of cases in each probability bin

  ### Result Construction and Testing
  # Return a list with elements `bin.boundaries`, `bin.id`, and `totals`
}
```

2.3.4 Task Implementation

Next, we implement the primary task of each function.

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
```

```r
#' @export
survProbBins <- function(object, ...) { UseMethod("survProbBins") }

#' S3 default method -- error only
#' @noRd
#' @export
survProbBins.default <- function(object, ...) {
  stop("`survProbBins()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' S3 method for objects returned by `survival::coxph()`
#' @noRd
#' @export
survProbBins.coxph <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && is.data.frame(newdata) || is.soma_adat(newdata),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        length(bin.boundaries) > 1L && all(bin.boundaries >= 0.0) &&
        all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  ## Predict survival probabilities

  # retrieve baseline hazard
  base_hazard <- survival::basehaz(object)
  # identify time-point in baseline hazard nearest the specified evaluation time
  t_idx       <- which.min(abs(eval.time - base_hazard[, 2L]))

  base_hazard_at_t <- base_hazard[t_idx, 1L]

  # ensure that baseline hazard is a valid value (not NA, NaN, Inf)
  if ( !is.finite(base_hazard_at_t) ) {
    stop("Invalid baseline hazard ", value(base_hazard_at_t), call. = FALSE)
```

```
  }
  message("Survival probability evaluated at t = ", value(base_hazard[t_idx, 2L]))

  # get linear predictors for newdata
  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                             })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function
  surv_probabilities <- exp(-base_hazard_at_t * exp(lin_predictors))

  # Ensure valid predictions
  surv_probabilities <- .testProbabilities(surv_probabilities)

  ### Task Implementation

  # Bin and tally predicted probabilities
  result <- .binData(surv_probabilities, bin.boundaries)

  ### Result Construction and Testing
  # Return `.binData()` value object
}

#' S3 method for objects returned by `survival::survreg`
#' @noRd
#' @export
survProbBins.survreg <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && is.data.frame(newdata) || is.soma_adat(newdata),
    "`eval.time` must be numeric." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        length(bin.boundaries) > 1L && all(bin.boundaries >= 0.0) &&
        all(bin.boundaries <= 1.0)
  )

  ### Input Processing
```

```
  ## Predict survival probabilities

  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                               })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function

  surv_probabilities <- 1.0 - stats::pweibull(eval.time,
                                              shape = 1.0 / exp(object$scale),
                                              scale = exp(lin_predictors))
  # Ensure valid predictions
  surv_probabilities <- .testProbabilities(surv_probabilities)

  ### Task Implementation
  # Bin and tally predicted probabilities
  result <- .binData(surv_probabilities, bin.boundaries)

  ### Result Construction and Testing
  # Return `.binData()` value object
}

#' S3 method for objects returned by `SomaSurvival::fitSurvregnet`
#' @noRd
#' @export
survProbBins.survregnet <- function(object, newdata, eval.time, bin.boundaries,
                                    lambda, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && is.data.frame(newdata) || is.soma_adat(newdata),
    "`eval.time` must be numeric." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && all(bin.boundaries >= 0.0) &&
      all(bin.boundaries <= 1.0),
    "`lambda` must be a non-negative scalar." =
      !missing(lambda) && is.numeric(lambda) &&
      is.vector(lambda) && length(lambda) == 1L && lambda >= 0.0
```

```
  )

  ### Input Processing
  # Convert to survreg object and use survreg method
  survreg_object <- tryCatch(SomaSurvival::convert2Survreg(object, lambda),
                             error = function(e) {
                               stop("Unable to convert `object` to `survreg`.\n\t",
                                    e$message, call. = FALSE)
                             })

  ### Task Implementation
  result <- survProbBins(survreg_object, newdata, eval.time, bin.boundaries, ...)

  ### Result Construction and Testing
  # Return `survProbBins.survreg()` value object
}

#' Internal function to test that probabilities are valid
#'
#' @noRd
#' @param predictions Numeric. The probabilities to test.
#' @return Numeric. The probabilities with invalid values reset to NA_real_
#' @keywords internal
.testProbabilities <- function(predictions) {
  ### Input Testing
  stopifnot(
    "`predictions` must be numeric." =
      !missing(predictions) && is.numeric(predictions) &&
      is.vector(predictions) && length(predictions) != 0L
  )

  ### Input Processing

  ### Task Implementation
  # convert non-finite (NA, NaN, Inf) values to NA_real_
  # convert values outside of [0, 1] to NA_real_
  bad_values <- !is.finite(predictions) | predictions < 0.0 | predictions > 1.0
  if ( any(bad_values) ) {
    predictions[bad_values] <- NA_real_
    message(value(sum(bad_value)), " invalid probabilities have been set to NA.")
  }

  ### Result Construction and Testing
  # Return modified probabilities as an unnamed numeric vector
}
```

```r
#' Internal function to perform binning
#'
#' Provided a vector of values and a vector of bin boundaries, identify the
#'   bin in which each value falls and the total number of values in each bin.
#'
#' @noRd
#' @param values A numeric or numeric vector. The values to bin and tally.
#' @param bin.boundaries Numeric vector. The K+1 boundaries of K bins.
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @keywords internal
.binData <- function(values, bin.boundaries) {
  ### Input Testing
  stopifnot(
    "`values` must be a non-empty, numeric vector." =
      !missing(values) && is.numeric(values) && is.vector(values) &&
        length(values) != 0L,
    "`bin.boundaries` must be a numeric vector of length > 1." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && is.vector(bin.boundaries),
    "All values in `bin.boundaries` must be unique." =
      length(bin.boundaries) == length(unique(bin.boundaries)),
    "`bin.boundaries` cannot contain NA or NaN values." =
      all(!is.na(bin.boundaries) && !is.nan(bin.boundaries)),
    "`bin.boundaries` must be in increasing order." = !is.unsorted(bin.boundaries),
    "`values` must be finite and in range of `bin.boundaries`." =
      all(is.finite(values)) && all(values >= min(bin.boundaries)) &&
      all(values <= max(bin.boundaries))
  )

  ### Input Processing

  ### Task Implementation
  # Identify bin membership based on the provided probabilities
  # returns k satisfying bin_k <= x < bin_{k+1}
  # all.inside = TRUE sets anything that does not fall with the
  # range of bin.boundaries to the nearest bin
  # (i.e., 1 if x < bin_1 or K if x >= bin_K)
  bins <- findInterval(values, bin.boundaries, all.inside = TRUE)
```

```
  # Tally the number of cases in each probability bin
  totals        <- tabulate(bins)
  names(totals) <- 1L:(length(bins)-1L)

  ### Result Construction and Testing
  # Return a list with elements `bin.boundaries`, `bin.id`, and `totals`
}
```

2.3.5 Results Construction and Testing

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, ...) { UseMethod("survProbBins") }

#' S3 default method -- error only
#' @noRd
#' @export
survProbBins.default <- function(object, ...) {
  stop("`survProbBins()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' S3 method for objects returned by `survival::coxph()`
#' @noRd
```

```r
#' @export
survProbBins.coxph <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && is.data.frame(newdata) || is.soma_adat(newdata),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        length(bin.boundaries) > 1L && all(bin.boundaries >= 0.0) &&
        all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  ## Predict survival probabilities

  # retrieve baseline hazard
  base_hazard <- survival::basehaz(object)
  # identify time-point in baseline hazard nearest the specified evaluation time
  t_idx       <- which.min(abs(eval.time - base_hazard[, 2L]))

  base_hazard_at_t <- base_hazard[t_idx, 1L]

  # ensure that baseline hazard is a valid value (not NA, NaN, Inf)
  if ( !is.finite(base_hazard_at_t) ) {
    stop("Invalid baseline hazard ", value(base_hazard_at_t), call. = FALSE)
  }
  message("Survival probability evaluated at t = ", value(base_hazard[t_idx, 2L]))

  # get linear predictors for newdata
  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                             })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function
  surv_probabilities <- exp(-base_hazard_at_t * exp(lin_predictors))

  # Ensure valid predictions
  surv_probabilities <- .testProbabilities(surv_probabilities)
```

```
  ### Task Implementation

  # Bin and tally predicted probabilities
  result <- .binData(surv_probabilities, bin.boundaries)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
  result
}

#' S3 method for objects returned by `survival::survreg`
#' @noRd
#' @export
survProbBins.survreg <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && is.data.frame(newdata) || is.soma_adat(newdata),
    "`eval.time` must be numeric." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        length(bin.boundaries) > 1L && all(bin.boundaries >= 0.0) &&
        all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  ## Predict survival probabilities

  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                               })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function

  surv_probabilities <- 1.0 - stats::pweibull(eval.time,
                                              shape = 1.0 / exp(object$scale),
                                              scale = exp(lin_predictors))
  # Ensure valid predictions
  surv_probabilities <- .testProbabilities(surv_probabilities)
```

```
  ### Task Implementation
  # Bin and tally predicted probabilities
  result <- .binData(surv_probabilities, bin.boundaries)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
  result
}

#' S3 method for objects returned by `SomaSurvival::fitSurvregnet`
#' @noRd
#' @export
survProbBins.survregnet <- function(object, newdata, eval.time, bin.boundaries,
                                    lambda, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && is.data.frame(newdata) || is.soma_adat(newdata),
    "`eval.time` must be numeric." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && all(bin.boundaries >= 0.0) &&
      all(bin.boundaries <= 1.0),
    "`lambda` must be a non-negative scalar." =
      !missing(lambda) && is.numeric(lambda) &&
      is.vector(lambda) && length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Convert to survreg object and use survreg method
  survreg_object <- tryCatch(SomaSurvival::convert2Survreg(object, lambda),
                             error = function(e) {
                               stop("Unable to convert `object` to `survreg`.\n\t",
                                    e$message, call. = FALSE)
                             })

  ### Task Implementation
  result <- survProbBins(survreg_object, newdata, eval.time, bin.boundaries, ...)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
```

```
  result
}

#' Internal function to test that probabilities are valid
#'
#' @noRd
#' @param predictions Numeric. The probabilities to test.
#' @return Numeric. The probabilities with invalid values reset to NA_real_
#' @keywords internal
.testProbabilities <- function(predictions) {
  ### Input Testing
  stopifnot(
    "`predictions` must be numeric." =
      !missing(predictions) && is.numeric(predictions) &&
      is.vector(predictions) && length(predictions) != 0L
  )

  ### Input Processing

  ### Task Implementation
  # convert non-finite (NA, NaN, Inf) values to NA_real_
  # convert values outside of [0, 1] to NA_real_
  bad_values <- !is.finite(predictions) | predictions < 0.0 | predictions > 1.0
  if ( any(bad_values) ) {
    predictions[bad_values] <- NA_real_
    message(value(sum(bad_value)), " invalid probabilities have been set to NA.")
  }

  ### Result Construction and Testing
  predictions
}

#' Internal function to perform binning
#'
#' Provided a vector of values and a vector of bin boundaries, identify the
#'   bin in which each value falls and the total number of values in each bin.
#'
#' @noRd
#' @param values A numeric or numeric vector. The values to bin and tally.
#' @param bin.boundaries Numeric vector. The K+1 boundaries of K bins.
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
```

```r
#'   probability bin.}
#'
#' @keywords internal
.binData <- function(values, bin.boundaries) {
  ### Input Testing
  stopifnot(
    "`values` must be a non-empty, numeric vector." =
      !missing(values) && is.numeric(values) && is.vector(values) &&
        length(values) != 0L,
    "`bin.boundaries` must be a numeric vector of length > 1." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && is.vector(bin.boundaries),
    "All values in `bin.boundaries` must be unique." =
      length(bin.boundaries) == length(unique(bin.boundaries)),
    "`bin.boundaries` cannot contain NA or NaN values." =
      all(!is.na(bin.boundaries) && !is.nan(bin.boundaries)),
    "`bin.boundaries` must be in increasing order." = !is.unsorted(bin.boundaries),
    "`values` must be finite and in range of `bin.boundaries`." =
      all(is.finite(values)) && all(values >= min(bin.boundaries)) &&
      all(values <= max(bin.boundaries))
  )

  ### Input Processing

  ### Task Implementation
  # Identify bin membership based on the provided probabilities
  # returns k satisfying bin_k <= x < bin_{k+1}
  # all.inside = TRUE sets anything that does not fall with the
  # range of bin.boundaries to the nearest bin
  # (i.e., 1 if x < bin_1 or K if x > bin_K)
  bins <- findInterval(values, bin.boundaries, all.inside = TRUE)

  # Tally the number of cases in each probability bin
  totals        <- tabulate(bins)
  names(totals) <- 1L:(length(bins)-1L)

  ### Result Construction and Testing
  list("bin.boundaries" = bin.boundaries,
       "bin.id"         = bins,
       "totals"         = totals)
}
```

That's it! We have a fully implemented extendable tool for the requested task!

2.3.6 Extensions

If it is decided that this tool would be handy if a user could instead provide a numeric vector of predicted survival probabilities rather than a fitted model with new data, such an extension can be made by adding the following:

```
#' Bin Predicted Survival Probabilities
#'
#' Function takes a user provided vector of survival probabilities or obtains
#'   predicted survival probabilities from a provided fitted model and new data
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model or numeric vector. Fitted survival models
#'   are currently limited to classes `coxph`, `survreg`, and `survregnet`.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored. If `object` is numeric, this input is ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, ...) { UseMethod("survProbBins") }

#' S3 method for numeric vector objects
#' @noRd
#' @export
survProbBins.numeric <- function(object, bin.boundaries, ...) {
  ### Input Testing
  # We reserve testing of `object` for the testing utility function
  stopifnot(
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  # Ensure valid predictions
  surv_probabilities <- .testProbabilities(object)

  ### Task Implementation
```

```
  # Bin and tally predicted probabilities
  result <- .binData(surv_probabilities, bin.boundaries)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
  result
}
```

Easy Peasy!!

2.4 Framework and Utility Function Design Structure

2.4.1 Usage Specification

The key components of this design are a main function detailing the procedure in terms of steps and utility functions to implement each step. An example outline of this design might be:

- A main function
  - `survProbBins(object, newdata, eval.time, bin.boundaries, lambda, ...)`
    - Test validity of inputs
    - Predict survival probabilities
    - Call `.binData()` to bin and tally predicted survival probabilities
    - Return `.binData()` value object
- A utility function to predict survival probabilities
  - `.predictSurvivalProb(object, newdata, eval.time, ...)`
    - Test validity of inputs
    - Predict survival probabilities
    - Call `.testProbabilities()`
    - Return `.testProbabilities()` value object
- A utility function to test validity of survival probabilities
  - `.testProbabilities(prediction)`
    - Ensure predicted probabilities are all finite and in [0,1]
- A utility function to bin and tally predicted survival probabilities
  - `.binData(values, bin.boundaries)`
    - Test validity of inputs

        * Identify bin membership based on the provided probabilities
        * Tally the number of cases in each probability bin
        * Create a list containing the bin boundaries, bin membership, and totals
        * Return list

Notice that two of our utility functions are identical to the internal common functions of the S3 based design. For brevity, we will not repeat the development of `.testProbabilities()` and `.binData()` in this section.

Next, we need to identify the steps in the procedure that are model type dependent. In this example, the step to predict survival probabilities is the only step that may vary across model types. Such dependence lends itself to an S3 structure. For example:

- A main function
  - `survProbBins(object, newdata, eval.time, bin.boundaries, lambda, ...)`
    * Test validity of inputs
    * Call `.predictSurvivalProb()` to estimate survival probabilities
    * Call `.binData()` to bin and tally predicted survival probabilities
    * Return `.binData()` value object
- A utility method to predict survival probabilities
  - A generic method
    * .predictSurvivalProb(object, …).
  - A default method
    * .predictSurvivalProb(object, …).
      · Generate error indicating that object is not supported
  - An extension method for coxph objects
    * .predictSurvivalProb(object, newdata, eval.time, …).
      · Test validity of inputs
      · Predict survival probabilities
      · Call .testProbabilities()
  - An extension method for survreg objects
    * .predictSurvivalProb(object, newdata, eval.time, …).
      · Test validity of inputs
      · Predict survival probabilities
      · Call .testProbabilities()
  - An extension method for survregnet objects
    * .predictSurvivalProb(object, newdata, eval.time, lambda, …).
      · Test validity of inputs

        - Call convert2Survreg() to convert object to a survreg object
        - Call .predictSurvivalProb.survreg() using converted object.

- A utility function to test validity of survival probabilities
  - `.testProbabilities(prediction)`
    - Ensure predicted probabilities are all finite and in [0,1]
- A utility function to bin and tally predicted survival probabilities
  - `.binData(values, bin.boundaries)`
    - Test validity of inputs
    - Identify bin membership based on the provided probabilities
    - Tally the number of cases in each probability bin
    - Create a list containing the bin boundaries, bin membership, and totals
    - Return list

This design is very similar to the S3 based design. The key difference lies in where we place the model type dependency. For the S3 based design, that dependency rests with the task itself. The framework design shifts that dependency to the utility functions; specifically, the utility function that predicts the survival probability. An advantage of the framework design is that this functionality – predict survival probabilities – is now available to any other function in the package. In the S3 based design, this functionality is not accessible.

Because the functionality of predicting the survival probability is available throughout the package, it is most robust if it is developed to ensure that the returned predictions obey the expected behaviors of a survival probability. Specifically, survival probabilities must be in [0,1]; if they are not, an expected "invalid" value is returned (e.g., `NA_real`). To make this utility function more robust should it be used elsewhere, it is a better design choice to do the testing within the prediction method. Thus, we chose to organize our design outline such that the "test survival probabilities" is performed in the prediction utility function itself rather than as a step in the main procedure.

2.4.1.1 Main Function

In the framework design, the main function, `survProbBins()`, sets up the general procedure for completing the task. That is, the steps required to perform the task are identified, but the details of each step are deferred to utility functions.

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
```

```
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @param lambda Numeric. The Elastic Net parameter of the model.
#'   Required only if `object` is of class `survregnet`; ignored for all others.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, newdata, eval.time, bin.boundaries,
                         lambda = NULL, ...) {
  ### Input Testing
  # Verify that `object`, `newdata`, `eval.time`, and `bin.boundaries` are provided
  # Verify that object is a supported model type
  # Verify that `newdata` is a data.frame or soma_adat
  # Verify that `newdata` contains the appropriate data
  # Verify that `eval.time` is a positive scalar
  # Verify that `bin.boundaries` is a numeric vector in [0, 1]
  # Verify that `lambda` is NULL or a non-negative scalar

  ### Input Processing
  # Call `.predictSurvivalProb()` to estimate survival probabilities

  ### Task Implementation
  # Call `.binData()` to bin and tally predicted probabilities

  ### Result Construction and Testing
  # Return `.binData()` value object
}
```

Notice that in contrast to the S3 based framework, **ALL** of the inputs required

by each model type must be provided as input. Here, `lambda` is an input regardless of the model type (though `NULL` is a perfectly acceptable value if `object` is not an Elastic Net model). We can circumvent this by dictating in the documentation that "if `object` is Elastic Net, `...` must include the Elastic Net parameter, `lambda`." However, heavy reliance on the ellipsis for such one-off parameters could become confusing for the user.

2.4.1.2 Utility Functions

The implementation of the "bin and tally the predicted probabilities" and "test the predicted survival probabilities" will be identical to those developed under the S3 based design.

The details of the prediction step will depend on the model type, and thus it is a natural S3 method. Should more model types become supported, ideally we will only have to extend this method to include the new model type and make minor modifications to documentation and input testing in the main function.

```
#' Internal S3 Method for Estimating the Survival Probabilities
#'
#' @noRd
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#' @return Numeric. The estimated survival probabilities with invalid or out of
#'   range values reset to NA_real_.
#' @keywords internal
.predictSurvivalProb <- function(object, ...) {
  UseMethod(".predictSurvivalProb")
}

#' @noRd
#' @keywords internal
.predictSurvivalProb.default <- function(object, ...) {
  stop("`.predictSurvivalProb()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' @noRd
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
```

```
#'   estimated.
#' @keywords internal
#' @note inclusion of lambda as an input parameter in non-Elastic Net methods
#'   ensures that it is not passed to the prediction method through the ellipsis
.predictSurvivalProb.coxph <- function(object, newdata, eval.time, lambda, ...) {
  ### Input Testing
  # Verify that `newdata` is a data.frame or soma_adat
  # Verify that `newdata` contains the appropriate data
  # Verify that `eval.time` is a positive scalar

  ### Input Processing

  ### Task Implementation
  # Predict survival probability
  #
  # Call .testProbabilities() to ensure valid predictions

  ### Result construction and testing
  # Return predicted survival probability as an unnamed numeric vector
}

#' @noRd
#' @keywords internal
#' @note inclusion of lambda as an input parameter in non-Elastic Net methods
#'   ensures that it is not passed to the prediction method through the ellipsis
.predictSurvivalProb.survreg <- function(object, newdata, eval.time, lambda, ...) {
  ### Input Testing
  # Verify that `newdata` is a data.frame or soma_adat
  # Verify that `newdata` contains the appropriate data
  # Verify that `eval.time` is a positive scalar

  ### Input Processing

  ### Task Implementation
  # Predict survival probability
  #
  # Call .testProbabilities() to ensure valid predictions

  ### Result construction and testing
  # Return predicted survival probability as an unnamed numeric vector
}

#' @noRd
#' @param lambda Numeric. The Elastic Net parameter, lambda, of the model.
#' @keyword internal
```

```
.predictSurvivalProb.survregnet <- function(object, newdata,
                                            eval.time, lambda, ...) {
  ### Input Testing
  # Verify that `newdata` is a data.frame or soma_adat
  # Verify that `newdata` contains the appropriate data
  # Verify that `eval.time` is a positive scalar
  # Verify that `lambda` is a non-negative scalar

  ### Input Processing
  # Convert to a survreg object

  ### Task Implementation
  # call .predictSurvivalProb.survreg() using the converted object

  ### Result construction and testing
  # Return `.predictSurvivalProb()` value object
}
```

2.4.2 Input Testing

Unlike the S3 based design, input `object` must be explicitly tested in the framework and utility function design to ensure that it supported.

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @param lambda Numeric or NULL. The Elastic Net parameter, lambda, of the model.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
```

```
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, newdata, eval.time, bin.boundaries,
                         lambda = NULL, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`object` must be inherit from `coxph`, `survreg`, or `survregnet`." =
      !missing(object) && inherits(object, c("coxph", "survreg", "survregnet")),
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0),
    "`lambda` must be a non-negative scalar." =
      is.null(lambda) || (is.numeric(lambda) && is.vector(lambda) &&
                            length(lambda) == 1L && lambda >= 0.0)
  )

  ### Input Processing
  # Call `.predictSurvivalProb()` to estimate survival probabilities

  ### Task Implementation
  # Call `.binData()` to bin and tally predicted probabilities

  ### Result Construction and Testing
  # Return `.binData()` value object
}

#' Internal S3 Method for Estimating the Survival Probabilities
#'
#' @noRd
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet` and their `stripped_*` counterparts.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#' @returns Numeric. The estimated survival probabilities with invalid or out of
```

```
#'   range values reset to NA_real_.
#' @keywords internal
.predictSurvivalProb <- function(object, ...) {
  UseMethod(".predictSurvivalProb")
}

#' @noRd
#' @keywords internal
.predictSurvivalProb.default <- function(object, ...) {
  stop("`.predictSurvivalProb()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' @noRd
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @keywords internal
.predictSurvivalProb.coxph <- function(object, newdata, eval.time, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing

  ### Task Implementataion
  # Predict survival probability
  #
  # Call .testProbabilities() to ensure valid predictions

  ### Result construction and testing
  # Return predicted survival probability as an unnamed numeric vector
}

#' @noRd
#' @export
.predictSurvivalProb.survreg <- function(object, newdata, eval.time, ...) {
```

```
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing

  ### Task Implementation
  # Predict survival probability
  #
  # Call .testProbabilities() to ensure valid predictions

  ### Result construction and testing
  # Return predicted survival probability as an unnamed numeric vector
}

#' @noRd
#' @param lambda Numeric or NULL. The Elastic Net parameter, lambda, of the model.
#' @export
.predictSurvivalProb.survregnet <- function(object, newdata,
                                            eval.time, lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`lambda` must be a non-negative scalar." =
      !missing(lambda) && is.numeric(lambda) && is.vector(lambda) &&
      length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Call convert2survreg() to convert to a `survreg` object

  ### Task Implementation
  # Call .predictSurvivalProb.survreg() passing converted object

  ### Result construction and testing
  # Return `.predictSurvivalProb.survreg()` value object
```

```
}
```

2.4.3 Input Processing

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, newdata, eval.time, bin.boundaries,
                         lambda = NULL, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`object` must be inherit from `coxph`, `survreg`, or `survregnet`." =
      !missing(object) && inherits(object, c("coxph", "survreg", "survregnet")),
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
```

```
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
      all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0),
    "`lambda` must be a non-negative scalar." =
      is.null(lambda) || (is.numeric(lambda) && is.vector(lambda) &&
                          length(lambda) == 1L && lambda >= 0.0)
  )

  ### Input Processing
  # Estimate survival probabilities
  predictions <- .predictSurvivalProb(object, newdata, lambda = lambda, ...)

  ### Task Implementation
  # Call `.binData()` to bin and tally predicted probabilities

  ### Result Construction and Testing
  # Return `.binData()` value object
}

#' Internal S3 Method for Estimating the Survival Probabilities
#'
#' @noRd
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet` and their `stripped_*` counterparts.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#' @returns Numeric. The estimated survival probabilities with invalid or out of
#'   range values reset to NA_real_.
#' @keywords internal
.predictSurvivalProb <- function(object, ...) {
  UseMethod(".predictSurvivalProb")
}

.predictSurvivalProb.default <- function(object, ...) {
  stop("`.predictSurvivalProb()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' @noRd
#' @export
#' Note including lambda as an input here eliminates it from being passed to
```

```
#'   the prediction method through the ellipsis
.predictSurvivalProb.coxph <- function(object, newdata, eval.time, lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing

  ### Task Implementation
  # Predict survival probability
  #
  # Call .testProbabilities() to ensure valid predictions

  ### Result construction and testing
  # Return predicted survival probability as an unnamed numeric vector
}

#' @noRd
#' @export
#' Note including lambda as an input here eliminates it from being passed to
#'   the prediction method through the ellipsis
.predictSurvivalProb.survreg <- function(object, newdata, eval.time,
                                         lambda, ...) {
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing

  ### Task Implementation
  # Predict survival probability
  #
  # Call .testProbabilities() to ensure valid predictions

  ### Result construction and testing
  # Return predicted survival probability as an unnamed numeric vector
```

```
}

#' @noRd
#' @export
.predictSurvivalProb.survregnet <- function(object, newdata,
                                            eval.time, lambda, ...) {
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`lambda` must be a positive scalar." =
      !missing(lambda) && is.numeric(lambda) && is.vector(lambda) &&
      length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Convert to survreg object and use survreg method
  survreg_object <- tryCatch(SomaSurvival::convert2Survreg(object, lambda),
                             error = function(e) {
                               stop("Unable to convert `object` to `survreg`.\n\t",
                                    e$message, call. = FALSE)
                             })

  ### Task Implementation
  # Call `.predictSurvivalProb.survreg()` passing converted object

  ### Result construction and testing
  # Return `.predictSurvivalProb.survreg()` value object
}
```

2.4.4 Task Implementation

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
```

```
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, newdata, eval.time, bin.boundaries,
                         lambda = NULL, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`object` must be inherit from `coxph`, `survreg`, or `survregnet`." =
      !missing(object) && inherits(object, c("coxph", "survreg", "survregnet")),
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0),
    "`lambda` must be a non-negative scalar." =
      is.null(lambda) || (is.numeric(lambda) && is.vector(lambda) &&
                            length(lambda) == 1L && lambda >= 0.0)
  )

  ### Input Process

  ### Task Implementation
  # Estimate survival probabilities
  predictions <- .predictSurvivalProb(object, newdata, lambda = lambda, ...)

  # Bin and tally predicted probabilities
```

```
  result <- .binData(predictions, bin.boundaries)

  ### Result Construction and Testing
  # Return `.binData()` value object
}

#' Internal S3 Method for Estimating the Survival Probabilities
#'
#' @noRd
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet` and their `stripped_*` counterparts.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#' @returns Numeric. The estimated survival probabilities with invalid or out of
#'   range values reset to NA_real_.
#' @keywords internal
.predictSurvivalProb <- function(object, ...) {
  UseMethod(".predictSurvivalProb")
}

.predictSurvivalProb.default <- function(object, ...) {
  stop("`.predictSurvivalProb()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' @noRd
#' @export
#' Note including lambda as an input here eliminates it from being passed to
#'   the prediction method through the ellipsis
.predictSurvivalProb.coxph <- function(object, newdata, eval.time, lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing
```

```
  ### Task Implementation
  ## Predict survival probabilities

  # retrieve baseline hazard
  base_hazard <- survival::basehaz(object)
  # identify time-point in baseline hazard nearest the specified evaluation time
  t_idx       <- which.min(abs(eval.time - base_hazard[, 2L]))

  base_hazard_at_t <- base_hazard[t_idx, 1L]

  # ensure that baseline hazard is a valid value (not NA, NaN, Inf)
  if ( !is.finite(base_hazard_at_t) ) {
    stop("Invalid baseline hazard ", value(base_hazard_at_t), call. = FALSE)
  }
  message("Survival probability evaluated at t = ", value(base_hazard[t_idx, 2L]))

  # get linear predictors for newdata
  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                             })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function
  surv_probabilities <- exp(-base_hazard_at_t * exp(lin_predictors))

  # Ensure valid predictions
  result <- .testProbabilities(surv_probabilities)

  ### Result construction and testing
  # Return predicted survival probability as an unnamed numeric vector
}

#' @noRd
#' @export
#' Note including lambda as an input here eliminates it from being passed to
#'   the prediction method through the ellipsis
.predictSurvivalProb.survreg <- function(object, newdata, eval.time,
                                         lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
```

```
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing

  ### Task Implementation
  ## Predict survival probabilities

  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                               })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function

  surv_probabilities <- 1.0 - stats::pweibull(eval.time,
                                              shape = 1.0 / exp(object$scale),
                                              scale = exp(lin_predictors))
  # Ensure valid predictions
  result <- .testProbabilities(surv_probabilities)

  ### Result construction and testing
  # Return predicted survival probability as an unnamed numeric vector
}

#' @noRd
#' @export
.predictSurvivalProb.survregnet <- function(object, newdata,
                                            eval.time, lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`lambda` must be a positive scalar." =
      !missing(lambda) && is.numeric(lambda) && is.vector(lambda) &&
      length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Convert to survreg object and use survreg method
  survreg_object <- tryCatch(SomaSurvival::convert2Survreg(object, lambda),
```

```
                                error = function(e) {
                                  stop("Unable to convert `object` to `survreg`.\n\t",
                                       e$message, call. = FALSE)
                                })

  result <- .predictSurvivalProb(survreg_object, newdata, eval.time, ...)

  ### Result construction and testing
  # Return `.predictSurvivalProb.survreg()` value object
}
```

2.4.5 Results Construction and Testing

For our example, there are no additional processing steps required for the result; `.binData()` created the expected list.

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, newdata, eval.time, bin.boundaries,
                         lambda = NULL, ...) {
  ### Input Testing
```

```
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`object` must be inherit from `coxph`, `survreg`, or `survregnet`." =
      !missing(object) && inherits(object, c("coxph", "survreg", "survregnet")),
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
      all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0),
    "`lambda` must be a non-negative scalar." =
      is.null(lambda) || (is.numeric(lambda) && is.vector(lambda) &&
                          length(lambda) == 1L && lambda >= 0.0)
  )

  ### Input Process

  ### Task Implementation
  # Estimate survival probabilities
  predictions <- .predictSurvivalProb(object, newdata, lambda = lambda, ...)

  # Bin and tally predicted probabilities
  result <- .binData(predictions, bin.boundaries)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
  result
}

#' Internal S3 Method for Estimating the Survival Probabilities
#'
#' @noRd
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet` and their `stripped_*` counterparts.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#' @returns Numeric. The estimated survival probabilities with invalid or out of
#'   range values reset to NA_real_.
#' @keywords internal
```

```
.predictSurvivalProb <- function(object, ...) {
  UseMethod(".predictSurvivalProb")
}

.predictSurvivalProb.default <- function(object, ...) {
  stop("`.predictSurvivalProb()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' @noRd
#' @export
#' Note including lambda as an input here eliminates it from being passed to
#'   the prediction method through the ellipsis
.predictSurvivalProb.coxph <- function(object, newdata, eval.time, lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
       is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing

  ### Task Implementation
  ## Predict survival probabilities

  # retrieve baseline hazard
  base_hazard <- survival::basehaz(object)
  # identify time-point in baseline hazard nearest the specified evaluation time
  t_idx       <- which.min(abs(eval.time - base_hazard[, 2L]))

  base_hazard_at_t <- base_hazard[t_idx, 1L]

  # ensure that baseline hazard is a valid value (not NA, NaN, Inf)
  if ( !is.finite(base_hazard_at_t) ) {
    stop("Invalid baseline hazard ", value(base_hazard_at_t), call. = FALSE)
  }
  message("Survival probability evaluated at t = ", value(base_hazard[t_idx, 2L]))

  # get linear predictors for newdata
  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
```

```
                              error = function(e) {
                                stop("Unable to obtain linear predictors.",
                                     e$message, call. = FALSE)
                                })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function
  surv_probabilities <- exp(-base_hazard_at_t * exp(lin_predictors))

  # Ensure valid predictions
  result <- .testProbabilities(surv_probabilities)

  ### Result construction and testing
  unname(result)
}

#' @noRd
#' @export
#' Note including lambda as an input here eliminates it from being passed to
#'   the prediction method through the ellipsis
.predictSurvivalProb.survreg <- function(object, newdata, eval.time,
                                         lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing

  ### Task Implementation
  ## Predict survival probabilities

  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                              error = function(e) {
                                stop("Unable to obtain linear predictors.",
                                     e$message, call. = FALSE)
                                })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function

  surv_probabilities <- 1.0 - stats::pweibull(eval.time,
                                              shape = 1.0 / exp(object$scale),
```

```
                                                scale = exp(lin_predictors))
  # Ensure valid predictions
  result <- .testProbabilities(surv_probabilities)

  ### Result construction and testing
  unname(result)
}

#' @noRd
#' @export
.predictSurvivalProb.survregnet <- function(object, newdata,
                                            eval.time, lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`lambda` must be a positive scalar." =
      !missing(lambda) && is.numeric(lambda) && is.vector(lambda) &&
      length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Convert to survreg object and use survreg method
  survreg_object <- tryCatch(SomaSurvival::convert2Survreg(object, lambda),
                             error = function(e) {
                               stop("Unable to convert `object` to `survreg`.\n\t",
                                    e$message, call. = FALSE)
                             })

  result <- .predictSurvivalProb(survreg_object, newdata, eval.time, ...)

  ### Result construction and testing
  result
}
```

2.4.6 Extensions

If it is decided that this tool would be handy if a user could instead provide a numeric vector of predicted survival probabilities rather than a fitted model, there are a few options available for this extension. First, the step of the main function responsible for predicting the survival probability could be wrapped in

an `if/else` statement.

```
#' Bin Predicted Survival Probabilities
#'
#' Function takes a user provided vector of survival probabilities or obtains
#'   predicted survival probabilities from a provided fitted model and new data
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model or numeric vector. Survival models are
#'   currently limited to classes `coxph`, `survreg`, and `survregnet`.
#' @param newdata soma_adat, data.frame, or NULL. The model covariates, times, and
#'   status data. If NULL, `object` must be a numeric vector.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored. If `object` is numeric, input is ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, newdata = NULL, eval.time, bin.boundaries,
                         lambda = NULL, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`object` must be inherit from `coxph`, `numeric`, `survreg`, or `survregnet`." =
      !missing(object) && inherits(object, c("coxph", "numeric", "survreg", "survregnet")),
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0),
    "`lambda` must be a non-negative scalar." =
```

```
      is.null(lambda) || (is.numeric(lambda) && is.vector(lambda) &&
                            length(lambda) == 1L && lambda >= 0.0)
  )

  ### Input Process

  ### Task Implementation
  # Estimate survival probabilities
  if ( is.numeric(object) ) {
    if ( is.vector(object) && length(object) != 0L ) {
      predictions <- .testProbabilities(object)
    } else {
      stop("`object` must be a non-empty numeric vector if predictions are provided",
           call. = FALSE)
    }
  } else {
    predictions <- .predictSurvivalProb(object, newdata, lambda = lambda, ...)
  }

  # Bin and tally predicted probabilities
  result <- .binData(predictions, bin.boundaries)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
  result
}
```

Notice that this extension required some modification to our usage statement (`newdata` can now be `NULL`) as well as additional options for the input testing of `object` and `newdata`. This choice may not be ideal as should the supported types for `object` be extended to other non-model type objects, additional layering of the `if/else` would be necessary, which can become cumbersome!

Another solution is to extend the prediction method. We still need to make some adjustments to the usage of the main function.

```
#' Bin Predicted Survival Probabilities
#'
#' Function takes a user provided vector of survival probabilities or obtains
#'   predicted survival probabilities from a provided fitted model and new data
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model or numeric vector. Survival models are
#'   currently limited to classes `coxph`, `survreg`, and `survregnet`.
#' @param newdata soma_adat, data.frame, or NULL. The model covariates, times, and
```

```r
#'   status data. If NULL, `object` must be a numeric vector.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored. If `object` is numeric, input is ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, newdata = NULL, eval.time, bin.boundaries,
                         lambda = NULL, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`object` must be inherit from `coxph`, `numeric`, `survreg`, or `survregnet`." =
      !missing(object) && inherits(object, c("coxph", "numeric", "survreg", "survregnet")),
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0),
    "`lambda` must be a non-negative scalar." =
      is.null(lambda) || (is.numeric(lambda) && is.vector(lambda) &&
                            length(lambda) == 1L && lambda >= 0.0)
  )

  ### Input Process

  ### Task Implementation
  # Estimate survival probabilities
  predictions <- .predictSurvivalProb(object, newdata, lambda = lambda, ...)

  # Bin and tally predicted probabilities
```

```
  result <- .binData(predictions, bin.boundaries)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
  result
}

#' @noRd
#' @export
.predictSurvivalProb.numeric <- function(object, ...) {

  stopifnot(
    "`object` must be a non-empty numeric vector if predictions are provided" =
      is.vector(object) && length(object) != 0L
  )

  # Ensure valid predictions
  result <- .testProbabilities(object)

  ### Result construction and testing
  result
}
```

This choice may seem odd given that one is not actually "predicting a survival function" but rather processing a provided prediction. Further, one must be careful that this extension of the prediction function does not have unintended consequences for other functions that might call `.predictSurvivalProb()`. However, if proper input testing is implemented in the downstream function, this is not a significant concern.

2.5 Key Design Differences

- S3 Based Design
  - Pros
    - **Extension to new model types does not risk changes to existing functionality.** To extend the tool to new model types, a developer need only create a new function `survProbBins.newType()`, which would need to execute the verification of input, obtain the predicted survival probabilities, and call the bin/tally method. There is little possibility that the added function will impact the execution of or results from the existing methods.

    - **Freedom in type-specific input arguments.** Because each method *must* contain only the dispatching argument (in our example, `object`), each method can declare its own required inputs. In our example, the Elastic Net parameter `lambda`, is an input only for the Elastic Net model type.
    - **More familiar to our users.** This is the primary design choice of the `somaverse` – our users are accustomed to its structure, which could make debugging for them easier.
  - Cons
    - **Steps implemented within the individuals methods are not available to other functions.** In our example, the prediction step is hidden within the `survProbBins.*()` functions, i.e., it is not returned as output so this predicted value cannot be accessed by a another function in the package. It is not unreasonable to imagine that a method to predict survival probabilities for a supported model would be useful elsewhere, but under the current design, it would need to be re-implemented. (More on this later.)
    - **Full coverage of unit tests can be difficult.** If multiple steps of the procedure are implemented in each individual method, it can be challenging to get strong coverage in unit tests.
    - **Procedure evolution.** In the absence of a single function specifying the steps required to complete the task, it is possible for subtle differences to arise in the procedure across model types.

- Framework and Utility Functions Design
  - Pros
    - **Main function highlights the *structure* of the solution rather than the details required for each model type.** This structure provides "sign posts" that can make developing unit tests easier and provides future developers the explicit steps expected to accomplish the task.
    - **Each step of the procedure can be individually unit tested.** Each function can be robustly unit tested to ensure that it matches the developers expectations. A key aspect of this design is that utility functions should be developed such that very little is "hidden" from unit tests.
    - **All utility functions are available to all other functions in the package.** In our example, `.predictProbSurv()` can be used elsewhere if needed without further development or testing.
    - **Extensions to utility functions become immediately available to other functions.** This can expedite extending multiple methods to new model types.
  - Cons

    - **Extension to new model types may require changes to existing functions.** For utility steps that are implemented as functions, developers must ensure that new functionality does not impact previously written code.
    - **Extension to new model types may not be intuitive or might require cumbersome if/else structures.** In our example we illustrated how to extend to a numeric vector of predictions. The utility function `.predictSurvalProb()` does not necessarily "make sense" when provided a vector of predictions.
    - **Bloated input structure.** Because inputs are shared across all model types, we lose the freedom to add or remove inputs for each model type. For example, because we support Elastic Net models, `lambda` must be provided as an input no matter the model type (though `NULL` is a perfectly acceptable value for non-Elastic Net models). This restriction can introduce confusion for users that must be clearly addressed in the documentation. Further, adding extensions might requiring adding additional input arguments or modifying default values - all of which can have unexpected consequences downstream if not done carefully.

2.6 Blended Design

The implementations that we chose above were meant to highlight the key differences between these two general design choices. We can address some of the disadvantages of the each method by blending the two designs. Specifically, we can move the survival probability prediction and testing steps of the S3 based design to its own individual S3 utility method, as was implemented in the framework design.

- A generic method
    - `survProbBins(object, ...)`.
- A default method
    - `survProbBins.default(object, ...)`.
        - Generate error indicating that `object` is not supported
- An extension method for `coxph` objects
    - `survProbBins.coxph(object, newdata, eval.time, bin.boundaries, ...)`.
        - Test validity of inputs
        - Call `.predictSurvivalProb()`
        - Call `.binData()`
        - Return `.binData()` value object

- An extension method for `survreg` objects
  - `survProbBins.survreg(object, newdata, eval.time, bin.boundaries, ...)`.
    - Test validity of inputs
    - Call `.predictSurvivalProb()`
    - Call `.binData()`
    - Return `.binData()` value object
- An extension method for `survregnet` objects
  - `survProbBins.survregnet(object, newdata, eval.time, bin.boundaries, lambda, ...)`.
    - Test validity of inputs
    - Call `.predictSurvivalProb()`
    - Call `.binData()`
    - Return `.binData()` value object
- A utility method to predict survival probabilities
  - A generic method
    - `.predictSurvivalProb(object, ...)`.
  - A default method
    - `.predictSurvivalProb(object, ...)`.
      - Generate error indicating that `object` is not supported
  - An extension method for `coxph` objects
    - `.predictSurvivalProb(object, newdata, eval.time, ...)`.
      - Test validity of inputs
      - Predict survival probabilities
      - Call `.testProbabilities()`
      - Return `.testProbabilities()` value object
  - An extension method for `survreg` objects
    - `.predictSurvivalProb(object, newdata, eval.time, ...)`.
      - Test validity of inputs
      - Predict survival probabilities
      - Call `.testProbabilities()`
      - Return `.testProbabilities()` value object
  - An extension method for `survregnet` objects
    - `.predictSurvivalProb(object, newdata, eval.time, lambda, ...)`.
      - Test validity of inputs
      - Call `convert2Survreg()` to convert `object` to a `survreg` object

        - Call `.predictSurvivalProb.survreg()` using converted object.
        - Return `.predictSurvivalProb.survreg()` value object

- An internal function for binning and tallying a vector of values
    - `.binData(values, bin.boundaries)`.
        - Test validity of inputs
        - Identify bin membership based on the provided probabilities
        - Tally the number of cases in each probability bin
        - Create a list containing the bin boundaries, bin membership, and totals
        - Return list

- An internal function for testing validity of survival probabilities
    - `.testProbabilities(predictions)`.
        - Ensure predicted probabilities are all finite and in [0,1]

```
#' Bin Predicted Survival Probabilities
#'
#' Function uses a fitted model and new data to estimate survival probabilities
#'   and returns a vector of bin membership and the total number of cases in
#'   each user-specified probability bin.
#'
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet`.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param eval.time Numeric. The time at which survival probabilities are to be
#'   estimated.
#' @param bin.boundaries Numeric vector. The k+1 boundaries of the k
#'   survival probability bins.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @export
survProbBins <- function(object, ...) { UseMethod("survProbBins") }

#' S3 default method -- error only
```

```r
#' @noRd
#' @export
survProbBins.default <- function(object, ...) {
  stop("`survProbBins()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' S3 method for objects returned by `survival::coxph()`
#' @noRd
#' @export
survProbBins.coxph <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  # Predict survival probabilities
  surv_probabilities <- .predictSurvivalProb(object, newdata, ...)

  ### Task Implementation
  # Bin and tally predicted probabilities
  .binData(surv_probabilities, bin.boundaries)
}

#' S3 method for objects returned by `survival::survreg`
#' @noRd
#' @export
survProbBins.survreg <- survProbBins.coxph

#' S3 method for objects returned by `SomaSurvival::fitSurvregnet`
#' @noRd
#' @export
```

```
survProbBins.survregnet <- function(object, newdata, eval.time, bin.boundaries,
                                    lambda, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
    "`lambda` must be a non-negative scalar." =
      !missing(lambda) && is.numeric(lambda) && is.vector(lambda) &&
      length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Predict survival probabilities
  surv_probabilities <- .predictSurvivalProb(object, newdata, lambda = lambda, ...)

  ### Task Implementation
  # Bin and tally predicted probabilities
  .binData(surv_probabilities, bin.boundaries)
}

#' Internal S3 Method for Estimating the Survival Probabilities
#'
#' @noRd
#' @param object Fitted survival model. Currently limited to classes
#'   `coxph`, `survreg`, and `survregnet` and their `stripped_*` counterparts.
#' @param newdata soma_adat or data.frame. The model covariates, times, and
#'   status data.
#' @param ... Optional arguments passed to the `predict.*` method. Note that
#'   input `type` will be ignored.
#'
#' @returns Numeric. The estimated survival probabilities with invalid or out of
#'   range values reset to NA_real_.
#' @keywords internal
.predictSurvivalProb <- function(object, ...) {
  UseMethod(".predictSurvivalProb")
}
```

```
.predictSurvivalProb.default <- function(object, ...) {
  stop("`.predictSurvivalProb()` does not yet support objects of class ",
       value(class(object))[1L], ".", call. = FALSE)
}

### Individual methods for each of the classes we must support

#' @noRd
#' @export
#' Note including lambda as an input here eliminates it from being passed to
#'   the prediction method through the ellipsis
.predictSurvivalProb.coxph <- function(object, newdata, eval.time, lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
       is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing

  ### Task Implementation
  ## Predict survival probabilities

  # retrieve baseline hazard
  base_hazard <- survival::basehaz(object)
  # identify time-point in baseline hazard nearest the specified evaluation time
  t_idx       <- which.min(abs(eval.time - base_hazard[, 2L]))

  # ensure that baseline hazard is a valid value (not NA, NaN, Inf)
  if ( !is.finite(base_hazard[t_idx, 1L]) ) {
    stop("Invalid baseline hazard ", value(base_hazard[t_idx, 1L]), call. = FALSE)
  }
  message("Survival probability evaluated at t = ", value(base_hazard[t_idx, 2L]))

  # get linear predictors for newdata
  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                             })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function
```

```
  surv_probabilities <- exp(-base_hazard_at_t * exp(lin_predictors))

  # Ensure valid predictions
  .testProbabilities(surv_probabilities)
}

#' @noRd
#' @export
#' Note including lambda as an input here eliminates it from being passed to
#'   the prediction method through the ellipsis
.predictSurvivalProb.survreg <- function(object, newdata, eval.time,
                                         lambda, ...) {
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0
  )

  ### Input Processing

  ### Task Implementation
  ## Predict survival probabilities

  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                             })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function

  surv_probabilities <- 1.0 - stats::pweibull(eval.time,
                                              shape = 1.0 / exp(object$scale),
                                              scale = exp(lin_predictors))
  # Ensure valid predictions
  .testProbabilities(surv_probabilities)
}

#' @noRd
#' @export
.predictSurvivalProb.survregnet <- function(object, newdata,
                                            eval.time, lambda, ...) {
```

```
  ### Input Testing
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && (is.data.frame(newdata) || is.soma_adat(newdata)),
    "`eval.time` must be a positive scalar." =
      !missing(eval.time) && is.numeric(eval.time) &&
        is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`lambda` must be a positive scalar." =
      !missing(lambda) && is.numeric(lambda) && is.vector(lambda) &&
      length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Convert to survreg object and use survreg method
  survreg_object <- tryCatch(SomaSurvival::convert2Survreg(object, lambda),
                             error = function(e) {
                               stop("Unable to convert `object` to `survreg`.\n\t",
                                    e$message, call. = FALSE)
                             })

  ### Task Implementation
  .predictSurvivalProb(survreg_object, newdata, eval.time, ...)
}

#' Internal function to test that probabilities are valid
#'
#' @noRd
#' @param predictions Numeric. The probabilities to test.
#' @return Numeric. The probabilities with invalid values reset to NA_real_
#' @keywords internal
.testProbabilities <- function(predictions) {
  ### Input Testing
  stopifnot(
    "`predictions` must be numeric." =
      !missing(predictions) && is.numeric(predictions) &&
      is.vector(predictions) && length(predictions) != 0L
  )

  ### Input Processing

  ### Task Implementation
  # convert non-finite (NA, NaN, Inf) values to NA_real_
  # convert values outside of [0, 1] to NA_real_
  bad_values <- !is.finite(predictions) | predictions < 0.0 | predictions > 1.0
  if ( any(bad_values) ) {
```

```
    predictions[bad_values] <- NA_real_
    message(value(sum(bad_value)), " invalid probabilities have been set as NA.")
  }

  predictions
}

#' Internal function to perform binning
#'
#' Provided a vector of values and a vector of bin boundaries, identify the
#'   bin in which each value falls and the total number of values in each bin.
#'
#' @noRd
#' @param values A numeric or numeric vector. The values to bin and tally.
#' @param bin.boundaries Numeric vector. The K+1 boundaries of K bins.
#' @return A list containing
#' \item{bin.boundaries}{A numeric vector. The provided probability bins.}
#' \item{bin.id}{An integer vector of the probability bin to which each case in
#'   `newdata` is assigned.}
#' \item{totals}{A numeric vector of the total number of cases in each
#'   probability bin.}
#'
#' @keywords internal
.binData <- function(values, bin.boundaries) {
  ### Input Testing
  stopifnot(
    "`values` must be a non-empty, numeric vector." =
      !missing(values) && is.numeric(values) && is.vector(values) &&
        length(values) != 0L,
    "`bin.boundaries` must be a numeric vector of length > 1." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && is.vector(bin.boundaries),
    "All values in `bin.boundaries` must be unique." =
      length(bin.boundaries) == length(unique(bin.boundaries)),
    "`bin.boundaries` cannot contain NA or NaN values." =
      all(!is.na(bin.boundaries) && !is.nan(bin.boundaries)),
    "`bin.boundaries` must be in increasing order." = !is.unsorted(bin.boundaries),
    "`values` must be finite and in range of `bin.boundaries`." =
      all(is.finite(values)) && all(values >= min(bin.boundaries)) &&
      all(values <= max(bin.boundaries))
  )

  ### Input Processing

  ### Task Implementation
```

```
  # Identify bin membership based on the provided probabilities
  bins <- findInterval(values, bin.boundaries, all.inside = TRUE)

  # Tally the number of cases in each probability bin
  totals <- tabulate(bins)
  names(totals) <- 1L:(length(bins)-1L)

  # Return a list of the results
  list("bin.boundaries" = bin.boundaries,
       "bin.id"         = bins,
       "totals"         = totals)
}
```

In this blended design, the estimation and testing of the survival probabilities is now available to the broader package; the input structure is not bloated for non-Elastic Net methods; and we can avoid duplication of code when the procedure for models are identical (as for `coxph` and `survreg`). Further, the tools can be extended to other non-model types, such as a numeric vector of predictions without modification of the tool's usage statement.

```
#' S3 method for numeric vector objects
#' @noRd
#' @export
survProbBins.numeric <- function(object, bin.boundaries, ...) {
  ### Input Testing
  stopifnot(
    "`object` must be a non-empty numeric vector if predictions are provided." =
      is.vector(object) && length(object) != 0L,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
        is.vector(bin.boundaries) && length(bin.boundaries) > 1L &&
        all(bin.boundaries >= 0.0) && all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  surv_probabilities <- .testProbabilities(object)

  ### Task Implementation
  # Bin and tally predicted probabilities
  .binData(surv_probabilities, bin.boundaries)
}
```

Some disadvantages of this design are that both the tool and the survival probability functions must be extended when new model types become available and the "procedure" can be obfuscated as it is implemented within each individual S3

extension of the tool. As future developers extend our tool, the procedure may subtly evolve away from our original intention. However, overall this blended approach presents the best of both worlds with fewer drawbacks.

Chapter 3

Unit Testing

3.1 General Comments

Test what your function *does* not what it *calls*.

What we mean by this is, the goal of unit testing is to test the code that **you** wrote, not to test the code that other developers have written. We start from the premise that any third-party function that we make use of has already been rigorously tested. Take for example the `predict.survregnet()` method from `SomaSurvival` (modified slightly to facilitate this discussion).

```
predict.survregnet <- function(object, newdata, lambda,
                               type = c("response", "lp", "linear", "quantile",
                                        "uquantile", "coefficients"),
                               p = 0.5, na.action = stats::na.pass, ...) {

  ### Input Testing
  type <- match.arg(type)
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      missing(newdata) || is.data.frame(newdata) || is.soma_adat(newdata),
    "`lambda` must be a scalar numeric." =
      !missing(lambda) && is.numeric(lambda) && is.vector(lambda) &&
      length(lambda) == 1L && lambda >= 0.0,
    "`p` must be a non-negative scalar." =
      is.numeric(p) && is.vector(p) && length(p) == 1L && p >= 0.0,
    "`na.action` must be a function." = is.function(na.action)
  )

  ### Input Processing
```

```
  survreg_obj <- convert2Survreg(object, lambda)

  ### Task Implementation
  if ( type == "coefficients" ) {
    return(data.frame(pred = survreg_obj$coefficients))
  }

  if ( missing(newdata) ) {
    pred <- predict(survreg_obj, type = type, p = p, na.action = na.action, ...)
  } else {
    pred <- predict(survreg_obj, newdata = newdata, type = type, p = p,
                    na.action = na.action, ...)
  }

  ### Result construction and testing
  data.frame(pred = pred)
}
```

After some testing and manipulation of the inputs provided by the user, this function simply calls the `predict.survreg()` function of the `survival` package. It is not our goal to test if the predictions returned by `predict.survreg()` are correct; as it is an external function, we assume that it is properly tested by the developers. Rather, we want to make sure that our testing and manipulation of the inputs are doing what we expect as well as verify that the structure of the *call to* `predict.survreg()` is being properly formulated.

In the code below we have added comments specifying the tests that could be implemented for the unit test suite of this function.

```
predict.survregnet <- function(object, newdata, lambda,
                               type = c("response", "lp", "linear", "quantile",
                                        "uquantile", "coefficients"),
                               p = 0.5, na.action = stats::na.pass, ...) {

  ### Input Testing
  #### - Ensure that an error is generated when `type` is an invalid value
  type <- match.arg(type)

  #### - Ensure that an error is generated when `newdata` is provided but is not a
  ####   data.frame or soma_adat
  #### - Ensure that an error is generated when `lambda` is not provided
  #### - Ensure that an error is generated when `lambda` is not numeric
  #### - Ensure that an error is generated when `lambda` is not a vector
  #### - Ensure that an error is generated when `lambda` is provided as a vector
  ####   of length > 1
```

```
#### - Ensure that an error is generated when `lambda` is negative
#### - Ensure that an error is generated when `p` is not numeric
#### - Ensure that an error is generated when `p` is not a vector
#### - Ensure that an error is generated when `p` is provided as a vector
####   of length > 1
#### - Ensure that an error is generated when `p` is negative
#### - Ensure that an error is generated when `na.action` is not a function
stopifnot(
  "`newdata` must be a data.frame or soma_adat." =
    missing(newdata) || is.data.frame(newdata) || is.soma_adat(newdata),
  "`lambda` must be a scalar numeric." =
    !missing(lambda) && is.numeric(lambda) && is.vector(lambda) &&
    length(lambda) == 1L && lambda >= 0.0,
  "`p` must be a non-negative scalar." =
    is.numeric(p) && is.vector(p) && length(p) == 1L && p >= 0.0,
  "`na.action` must be a function." = is.function(na.action)
)

### Input Processing
#### Assume that convert2Survreg does what it is supposed to do.
#### This line of code does need to be unit tested
survreg_obj <- convert2Survreg(object, lambda)

### Task Implementation
#### - Ensure that a data.frame of the coefficients for provided lambda value is
####   returned when `type = "coefficients"`
if ( type == "coefficients" ) {
  return(data.frame(pred = survreg_obj$coefficients))
}

if ( missing(newdata) ) {
  #### - Create a survreg object from the survregnet object using
  ####   convert2survreg(survregnet_object, lambda = lambda);
  ####   assume a single lambda value
  ####   Test equality of all combinations (keeping in mind that this function
  ####    returns a data.frame and predict.survreg will return a vector)
  ####   - default inputs should be equal to
  ####     predict.survreg(survreg_object, type = "response")
  ####   - type = "lp" should be equal to
  ####     predict.survreg(survreg_object, type = "lp")
  ####   - type = "linear" should be equal to
  ####     predict.survreg(survreg_object, type = "linear")
  ####   - type = "quantile" (p taking default value) should be equal to
  ####     predict.survreg(survreg_object, type = "quantile", p = 0.5)
  ####   - type = "quantile"; p = 0.25 should be equal to
```

```
    ####     predict.survreg(survreg_object, type = "quantile", p = 0.25)
    ####   - type = "uquantile" (p taking default value) should be equal to
    ####     predict.survreg(survreg_object, type = "uquantile", p = 0.5)
    ####   - type = "uquantile"; p = 0.25 should be equal to
    ####     predict.survreg(survreg_object, type = "uquantile", p = 0.25)
    ####
    #### - pass an additional argument through the ellipsis
    ####   - default values with addition of se.fit = TRUE should be equal to
    ####   predict.survreg(survreg_object, type = "response", se.fit = TRUE)
    pred <- predict(survreg_obj, type = type, p = p, na.action = na.action, ...)
  } else {
    #### - Create a survreg object from the survregnet object using
    ####   convert2survreg(survregnet_object, lambda = lambda);
    ####   assume a single lambda value
    ####   Test equality of all combinations (keeping in mind that this function
    ####    returns a data.frame and predict.survreg will return a vector)
    ####   - default inputs should be equal to
    ####     predict.survreg(survreg_object, newdata, type = "response")
    ####   - type = "lp" should be equal to
    ####     predict.survreg(survreg_object, newdata, type = "lp")
    ####   - type = "linear" should be equal to
    ####     predict.survreg(survreg_object, newdata, type = "linear")
    ####   - type = "quantile" (p taking default value) should be equal to
    ####     predict.survreg(survreg_object, newdata, type = "quantile", p = 0.5)
    ####   - type = "quantile"; p = 0.25 should be equal to
    ####     predict.survreg(survreg_object, newdata, type = "quantile", p = 0.25)
    ####   - type = "uquantile" (p taking default value) should be equal to
    ####     predict.survreg(survreg_object, newdata, type = "uquantile", p = 0.5)
    ####   - type = "uquantile"; p = 0.25 should be equal to
    ####     predict.survreg(survreg_object, newdata, type = "uquantile", p = 0.25)
    ####
    ####   - set at least 1 case of `newdata` to NA
    ####     na.action = stats::na.omit should be equal to
    ####     predict.survreg(survreg_object, newdata_na, type = "response",
    ####                     na.action = stats::na.omit)
    ####
    #### - pass an additional argument through the ellipsis
    ####   - default values with addition of se.fit = TRUE should be equal to
    ####   predict.survreg(survreg_object, type = "response", se.fit = TRUE)
    pred <- predict(survreg_obj, newdata = newdata, type = type, p = p,
                    na.action = na.action, ...)
  }

  data.frame(pred = pred)
}
```

There are some additional tests that could be included. For example, ensuring that `na.action` is one of `na.omit`, `na.fail`, `na.omit`, or `na.fail` as well as ensuring that any inputs provided through the ellipsis are indeed formals of `predict.survreg()`. However, as these inputs are only *passed* to `predict.survreg()` (we make no internal usage of these input) we can assume that `predict.survreg()` handles inappropriate values. Technically, we could omit any testing of `na.action` because we do not use it. However, because we explicitly include it as input and pass it to the `predict()` method, it is not a bad idea to at least ensure that it is the appropriate type of object, i.e., a function.

3.2 Unit Tests for Chapter 2 Example

To highlight some general recommendations for developing unit tests, we will make use of the implementations developed in Chapter 2. We will not cover every function.

3.2.1 Common Functions

Begin the test suite by developing tests for the common internal functions. This allows us to safely make use of these resources when testing functions that call the common functions, i.e., once `.bindata()` is fully tested, we can call `.bindata()` in any later test suite without needing to verify its returned value.

Because all of the implementations make use of the same utility functions `.binData()` and `testProbabilities()`, we develop these tests without concern for the overall design choice.

It is recommended that unit tests are developed for each component of the function: input testing, input processing, task implementation, and returned result. Typically, the final two components (task and result) will be combined.

We will begin with the `.binData()` function

```
#' Internal function to perform binning
#'
#' Provided a vector of values and a vector of bin boundaries,
#'   identify the bin in which each value falls and the total number of values
#'   in each bin.
#'
#' @noRd
#' @param values A numeric or numeric vector
#' @param bin.boundaries Numeric vector. The k+1 boundaries of k bins.
#' @return A list containing
#' \item{bins}{A vector of the bin index to which each value is assigned.}
```

```
#' \item{totals}{A vector of the total number of cases in each bin.}
#'
#' @keywords internal
.binData <- function(values, bin.boundaries, ...) {
  ### Input Testing
  stopifnot(
    "`values` must be a non-empty, numeric vector." =
      !missing(values) && is.numeric(values) && is.vector(values) &&
      length(values) != 0L,
    "`bin.boundaries` must be a numeric vector of length > 1." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && is.vector(bin.boundaries),
    "All values in `bin.boundaries` must be unique." =
      length(bin.boundaries) == length(unique(bin.boundaries)),
    "`bin.boundaries` cannot contain NA or NaN values." =
      all(!is.na(bin.boundaries) && !is.nan(bin.boundaries)),
    "`bin.boundaries` must be in increasing order." = !is.unsorted(bin.boundaries),
    "`values` must be finite and in range of `bin.boundaries`." =
      all(is.finite(values)) && all(values >= min(bin.boundaries)) &&
      all(values <= max(bin.boundaries))
  )

  ### Input Processing

  ### Task Implementation
  # Identify bin membership based on the provided probabilities
  bins <- findInterval(values, bin.boundaries, all.inside = TRUE)

  # Tally the number of cases in each probability bin
  totals <- tabulate(bins)
  names(totals) <- 1L:(length(bins)-1L)

  ### Result Construction and Testing
  list("bin.boundaries"  = bin.boundaries,
       "bin.id"          = bins,
       "totals"          = totals)
}
```

3.2.1.1 Unit tests for Input Testing

Ensure that all stopping conditions are triggered appropriately. This suite can be long, but it is, in the writers' opinion, the most important test suite. As they say – GIGO or "garbage in, garbage out." This series of tests will typically involve only the `testthat::expect_error()` function.

```
test_that("Input Testing generates expected errors", {
  ## Use a fully named input structure to test for missing value failures.
  ##   This test ensures that if the function is called using a function
  ##   such as do.call, the elements of the list have been appropriate named.
  expect_error(.binData(bin.boundaries = c(0.0, 0.1, 0.5, 1.0)),
               "`values` must be a numeric vector.")
  expect_error(.binData(values = c(0.0, 0.1, 0.5, 1.0)),
               "`bin.boundaries` must be a numeric vector of length > 1.")

  ## Next use an unnamed structure leaving 1 required input out.
  ## Here, because both `values` and `bin.boundaries` are vectors, the
  ## `bin.boundaries` missing error is triggered.
  expect_error(.binData(c(0.0, 0.1, 0.5, 1.0)),
               "`bin.boundaries` must be a numeric vector of length > 1.")

  ## Input-by-input, test the required characteristics
  ## specify fully valid inputs for all other input arguments

  ## `values` must be a numeric vector of length >= 1
  ## there is no need to test EVERY type of non-numeric object
  expect_error(.binData("a", c(0.0, 0.1, 0.5, 1.0)),
               "`values` must be a numeric vector.")
  ## pass a matrix to ensure the "is.vector" fails; a numeric matrix will pass
  ## the is.numeric() test.
  expect_error(.binData(matrix(1.0, 2L, 2L), c(0.0, 0.1, 0.5, 1.0)),
               "`values` must be a numeric vector of length > 1.")
  ## pass an empty numeric vector to ensure length() != 0 test fails
  expect_error(.binData(numeric(0L), c(0.0, 0.1, 0.5, 1.0)),
               "`values` must be a numeric vector of length > 1.")


  ## `bin.boundaries` must be a numeric vector of length > 1
  expect_error(.binData(c(0.0, 0.1, 0.5, 1.0), "a"),
               "`bin.boundaries` must be a numeric vector of length > 1.")
  ## pass a matrix of length 1 to ensure that is.vector() test fails
  expect_error(.binData(c(0.0, 0.1, 0.5, 1.0), matrix(1, 1L, 1L)),
               "`bin.boundaries` must be a numeric vector of length > 1.")
  ## pass a scalar to ensure that the length() > 1 test fails
  expect_error(.binData(c(0.0, 0.1, 0.5, 1.0), 1.0),
               "`bin.boundaries` must be a numeric vector of length > 1.")

  ## `bin.boundaries` must contain unique values
  expect_error(.binData(c(0.0, 0.1, 0.5, 1.0), c(0.0, 0.1, 1.0, 1.0)),
               "All values in `bin.boundaries` must be unique.")
```

```
  ## `bin.boundaries` must be non-NA and non-NaN and unique
  expect_error(.binData(c(0.0, 0.1, 0.5, 1.0), c(0.0, 0.1, 1.0, NA)),
               "`bin.boundaries` cannot contain NA or NaN values.")
  expect_error(.binData(c(0.0, 0.1, 0.5, 1.0), c(0.0, 0.1, 1.0, NaN)),
               "`bin.boundaries` cannot contain NA or NaN values.")

  ## `bin.boundaries` must be in increasing order
  expect_error(.binData(c(0.0, 0.1, 0.5, 1.0), c(0.1, 0.4, 0.3, 1.0)),
               "`bin.boundaries` must be in increasing order.")

  ## `values` must be finite and within `bin.boundaries`
  expect_error(.binData(1:4, 2:5),
               "`values` must be finite and in range of `bin.boundaries`.")
  expect_error(.binData(c(1:3, Inf), 2:5),
               "`values` must be finite and in range of `bin.boundaries`.")
  expect_error(.binData(c(1:3, NA), 2:5),
               "`values` must be finite and in range of `bin.boundaries`.")
  expect_error(.binData(c(1:3, NaN), 2:5),
               "`values` must be finite and in range of `bin.boundaries`.")
})
```

3.2.1.2 Unit tests for Input Processing

This function does not have an input processing component.

3.2.1.3 Unit tests for Task Implementation and Returned Result

Here, it is often possible to implement the task in a different way. For example, using an explicit `for` loop rather than `apply` or using test data that can easily be manipulated or evaluated. We will show a couple of options here – exactly which tests best suit the situation depends heavily on the complexity of the task. The goal is to ensure that **ALL** new code works as expected assuming that the inputs have passed the input testing phase.

If you are not able to test every line directly, this may be an indication that an intermediate step should be broken off into its own fully testable function.

```
test_that("Task Implementation generates expected result", {
  ## First, verify that the output returned by the function has the basic
  ## characteristics that we expect: a list with 3 elements with appropriate names
  test_object <- .binData(c(1.0, 2.0, 3.0), c(1.0, 2.0, 3.0))
  expect_true(is.list(test_object))
  expect_length(test_object, 3L)
  expect_named(test_object, c("bin.boundaries", "bin.id", "totals"))
```

```
## Here, we opt to use an alternative implementation.
##   We don't need to include input tests, etc., and it doesn't need
##   to be efficient -- just COMPLETELY DIFFERENT and CORRECT.
##   It is recommended that this version of the implementation be "simple"
##   and, when possible, avoids using convenience functions
.local_task(values, bin.boundaries) {

  bins   <- integer(length(values))
  totals <- integer(length(bin.boundaries) - 1L)

  for ( i in seq_along(values) ) {
    # findInterval() returns k that satisfies bin[k] <= x < bin[k+1]
    bins[i] <- sum(values[i] >= bin.boundaries)

    # we set `all.inside = TRUE` -- need to implement this clamping
    # any value >= max(bin.boundaries) is shifted to the last bin
    # Note that we do not need to consider the condition value < min(bin.boundaries)
    # because our input testing stops if any value is outside of the range
    # of bin.boundaries
    bins[i] <- min(bins[i], length(bin.boundaries) - 1L)
    totals[bins[i]] <- totals[bins[i]] + 1L
  }
  list("bin.boundaries" = bin.boundaries,
       "bin.id"         = bins,
       "totals"         = totals)
}

### Test a variety of inputs ranges, lengths, types

## "integer-like" numeric with values at the extrema of `bin.boundaries`
expect_equal(.local_task(c(1:10, 1:10), 1:10),
             .binData(c(1:10, 1:10), 1:10))
## numeric vector
expect_equal(.local_task(withr::with_seed(42L, stats::runif(10)),
                         c(0, 0.5, 1.0)),
             .binData(withr::with_seed(42L, stats::runif(10)),
                      c(0, 0.5, 1.0)))

### Test extremes

## A single `value`
expect_equal(.local_task(10.0, c(0, 5.0, 20.0)),
             .binData(10.0, c(0, 5.0, 20.0)))

## minimum vector length for `bin.boundaries` definition
```

```
  expect_equal(.local_task(10.0, c(0, 20.0)),
               .binData(10.0, c(0, 20.0)))

  ## Infinities in boundaries are properly handled
  expect_equal(.local_task(10.0, c(-Inf, Inf)),
               .binData(10.0, c(-Inf, Inf)))

  ## Next, we assume simple inputs for which we can easily construct
  ##  the expected output, without requiring a call to an (UNTESTED)
  ##  internal function -- sometimes these types of tests are more robust.
  ##  For example, it is possible that our second implementation will
  ##  include assumptions that we didn't realize we made.
  expect_equal(.binData(c(1:3, 1:3), 1:3),
               list("bin.boundaries" = 1:3,
                    "bin.id"         = c(1L, 2L, 2L, 1L, 2L, 2L),
                    "total"          = c("1" = 2L, "2" = 4L))

  expect_equal(.binData(c(0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0),
                        c(0.0, 0.25, 0.5, 0.75, 1.0)),
               list("bin.boundaries" = c(0.0, 0.25, 0.5, 0.75, 1.0),
                    "bin.id"         = c(1L, 1L, 1L, 2L, 2L, 3L, 3L, 4L, 4L, 4L),
                    "total"          = c("1" = 3L, "2" = 2L, "3" = 2L, "4" = 3L))

  expect_equal(.binData(c(0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0),
                        c(0.0, 1.0)),
               list("bin.boundaries" = c(0.0, 0.25, 0.5, 0.75, 1.0),
                    "bin.id"         = c(1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L, 1L),
                    "total"          = c("1" = 10L))

  expect_equal(.binData(5.0, c(0.0, 10.0)),
               list("bin.boundaries" = c(0.0, 10.0),
                    "bin.id"         = 1L,
                    "total"          = c("1" = 1L))
})
```

Similar tests can be constructed for `.testProbabilities()`.

```
test_that("`.testProbabilities()` returns expected errors.", {
  ## Input Testing
  expect_error(.testProbabilities(), "`predictions` must be numeric.")
  expect_error(.testProbabilities("a"), "`predictions` must be numeric.")
  expect_error(.testProbabilities(matrix(1.0, 2L, 2L)),
               "`predictions` must be numeric.")
  expect_error(.testProbabilities(numeric(0)),
               "`predictions` must be numeric.")
```

```
})

test_that("`.testProbabilities()` returns expected messages and results.", {
  ## Task Implementation and Result
  # ensure non-finite values are reset to NA and that
  # the correct message is generated
  expect_message(out <- .testProbabilities(c(-Inf, Inf, NA, NaN)),
                 "4 invalid probabilities have been set to NA.")
  expect_equal(out, rep(NA_real_, 4L))

  # ensure that finite values out of range are reset to NA and that
  # the correct message is generated
  expect_message(out <- .testProbabilities(c(-0.1, 0.0, 0.5, 1.0, 1.1)),
                 "2 invalid probabilities have been set to NA.")
  expect_equal(out, c(NA_real_, 0.0, 0.5, 1.0, NA_real_))

  # ensure that a vector of all valid values does not generate a message
  # and returned object is the same as the input object
  expect_message(out <- .testProbabilities(c(0.1, 0.2, 0.99)), NA)
  expect_equal(out, c(0.1, 0.2, 0.99))
})
```

3.3 S3 Based Design

We shift our attention now to the inidividual S3 methods of this design, one generic, one default, and three extensions defined for object of class `coxph`, `survreg`, and `survregnet`.

There is no need to test the generic method. And, the default method is trivially tested.

```
test_that("`survProbBins.default()` is correctly triggered", {
  expect_error(survProbBins(1.0),
               "`survProbBins()` does not yet support objects of class 'numeric'.")
})
```

3.3.1 S3 Extensions

```
#' S3 method for objects returned by `survival::coxph()`
#' @noRd
#' @export
survProbBins.coxph <- function(object, newdata, eval.time, bin.boundaries, ...) {
```

```
### Input Testing
# We forego testing for appropriate covariates in `newdata`, which will
#   be inherent in the procedure used to obtain predictions
stopifnot(
  "`newdata` must be a data.frame or soma_adat." =
    !missing && is.data.frame(newdata) || is.soma_adat(newdata),
  "`eval.time` must be a positive scalar." =
    !missing(eval.time) && is.numeric(eval.time) && eval.time > 0.0,
  "`bin.boundaries` must be a numeric vector in [0, 1]." =
    !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
    length(bin.boundaries) > 1L && all(bin.boundaries >= 0.0) &&
    all(bin.boundaries <= 1.0)
)

### Input Processing
## Predict survival probabilities

# retrieve baseline hazard
base_hazard <- survival::basehaz(object)
# identify time-point in baseline hazard nearest the specified evaluation time
t_idx       <- which.min(abs(eval.time - base_hazard[, 2L]))

base_hazard_at_t <- base_hazard[t_idx, 1L]

# ensure that baseline hazard is a valid value (not NA, NaN, Inf)
if ( !is.finite(base_hazard_at_t) ) {
  stop("Invalid baseline hazard ", value(base_hazard_at_t), call. = FALSE)
}
message("Survival probability evaluated at t = ", value(base_hazard[t_idx, 2L]))

# get linear predictors for newdata
lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                           error = function(e) {
                             stop("Unable to obtain linear predictors.",
                                  e$message, call. = FALSE)
                             })
# do not need to verify linear predictors, this will be done in the
# .testProbabilities() function
surv_probabilities <- exp(-base_hazard_at_t * exp(lin_predictors))

# Ensure valid predictions
surv_probabilities <- .testProbabilities(surv_probabilities)

### Task Implementation
```

```r
  # Bin and tally predicted probabilities
  result <- .binData(surv_probabilities, bin.boundaries)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
  result
}

#' S3 method for objects returned by `survival::survreg`
#' @noRd
#' @export
survProbBins.survreg <- function(object, newdata, eval.time, bin.boundaries, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && is.data.frame(newdata) || is.soma_adat(newdata),
    "`eval.time` must be numeric." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && all(bin.boundaries >= 0.0) &&
      all(bin.boundaries <= 1.0)
  )

  ### Input Processing
  ## Predict survival probabilities

  lin_predictors <- tryCatch(predict(object, newdata, type = "lp", ...),
                             error = function(e) {
                               stop("Unable to obtain linear predictors.",
                                    e$message, call. = FALSE)
                               })
  # do not need to verify linear predictors, this will be done in the
  # .testProbabilities() function

  surv_probabilities <- 1.0 - stats::pweibull(eval.time,
                                              shape = 1.0 / exp(object$scale),
                                              scale = exp(lin_predictors))
  # Ensure valid predictions
  surv_probabilities <- .testProbabilities(surv_probabilities)

  ### Task Implementation
```

```
  # Bin and tally predicted probabilities
  result <- .binData(surv_probabilities, bin.boundaries)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
  result
}

#' S3 method for objects returned by `SomaSurvival::fitSurvregnet`
#' @noRd
#' @export
survProbBins.survregnet <- function(object, newdata, eval.time, bin.boundaries,
                                    lambda, ...) {
  ### Input Testing
  # We forego testing for appropriate covariates in `newdata`, which will
  #   be inherent in the procedure used to obtain predictions
  stopifnot(
    "`newdata` must be a data.frame or soma_adat." =
      !missing && is.data.frame(newdata) || is.soma_adat(newdata),
    "`eval.time` must be numeric." =
      !missing(eval.time) && is.numeric(eval.time) &&
      is.vector(eval.time) && length(eval.time) == 1L && eval.time > 0.0,
    "`bin.boundaries` must be a numeric vector in [0, 1]." =
      !missing(bin.boundaries) && is.numeric(bin.boundaries) &&
      length(bin.boundaries) > 1L && all(bin.boundaries >= 0.0) &&
      all(bin.boundaries <= 1.0),
    "`lambda` must be a non-negative scalar." =
      !missing(lambda) && is.numeric(lambda) &&
      is.vector(lambda) && length(lambda) == 1L && lambda >= 0.0
  )

  ### Input Processing
  # Convert to survreg object and use survreg method
  survreg_object <- tryCatch(SomaSurvival::convert2Survreg(object, lambda),
                             error = function(e) {
                               stop("Unable to convert `object` to `survreg`.\n\t",
                                    e$message, call. = FALSE)
                             })

  ### Task Implementation
  result <- survProbBins(survreg_object, newdata, eval.time, bin.boundaries, ...)

  ### Result Construction and Testing
  # No further manipulation of the result of .binData() is required
  result
```

```
}
```

3.3.1.1 Input Testing

Notice that with the exception of the `lambda` input required for `survProbBins.survregnet()`, the input testing elements are identical. One could develop the unit tests for each S3 extension separately, or streamline the procedure by defining a single internal expectation that each model type uses. Specifically, we can define the following expectation in the Setup section of the test file

```
expect_input_fails <- function(model, data) {
  expect_error(survProbBins(model),
               "`newdata` must be a data.frame or soma_adat.")
  expect_error(survProbBins(model, matrix(1, 20, 5)),
               "`newdata` must be a data.frame or soma_adat.")
  expect_error(survProbBins(model, data),
               "`eval.time` must be a positive scalar.")
  expect_error(survProbBins(model, data, matrix(1.0, 1L, 1L)),
               "`eval.time` must be a positive scalar.")
  expect_error(survProbBins(model, data, c(1.0, 1.0)),
               "`eval.time` must be a positive scalar.")
  expect_error(survProbBins(model, data, -1.5),
               "`eval.time` must be a positive scalar.")
  expect_error(survProbBins(model, data, 1.0),
               "`bin.boundaries` must be a numeric vector in [0, 1].")
  expect_error(survProbBins(model, data, 1.0, "a"),
               "`bin.boundaries` must be a numeric vector in [0, 1].")
  expect_error(survProbBins(model, data, 1.0, 1.0),
               "`bin.boundaries` must be a numeric vector in [0, 1].")
  expect_error(survProbBins(model, data, 1.0, c(-1.0, 0.0, 1.0)),
               "`bin.boundaries` must be a numeric vector in [0, 1].")
  expect_error(survProbBins(model, data, 1.0, c(0.0, 1.0, 2.0)),
               "`bin.boundaries` must be a numeric vector in [0, 1].")
}
```

And employ this function to test that the input testing is behaving as expected.

```
test_that("`survProbBins.coxph()` input tests return expected errors", {
  apts <- SomaReadr::getAnalytes(SomaSurvival::sim_surv_data)
  form <- SomaSurvival::createSurvFormula(features = apts,
                                          time     = "time_lower",
                                          status   = "status_right")
  cox_model <- survival::coxph(form, SomaSurvival::sim_surv_data)
  expect_input_fails(cox_model, SomaSurvival::sim_surv_data)
```

```
})

test_that("`survProbBins.survreg()` input tests return expected errors", {
  apts <- SomaReadr::getAnalytes(SomaSurvival::sim_surv_data)
  form <- SomaSurvival::createSurvFormula(features = apts,
                                          time = "time_lower",
                                          status = "status_right")
  survreg_model <- survival::survreg(form, SomaSurvival::sim_surv_data)
  expect_input_fails(survreg_model, SomaSurvival::sim_surv_data)
})

test_that("`survProbBins.survregnet()` input tests return expected errors", {
  apts <- SomaReadr::getAnalytes(SomaSurvival::sim_surv_data)
  form <- SomaSurvival::createSurvFormula(features = apts,
                                          time     = "time_lower",
                                          status   = "status_right")
  grid <- SomaSurvival::calcEnetGrid(form, SomaSurvival::sim_surv_data,
                                     .alpha = 0.2, n.lambda = 5)
  survregnet_model <- SomaSurvival::fitSurvregnet(form, SomaSurvival::sim_surv_data,
                                                  tune.grid = grid, alpha = 0.2)
  expect_input_fails(survregnet_model, SomaSurvival::sim_surv_data)

  expect_error(survProbBins(survregnet_model, SomaSurvival::sim_surv_data,
                            eval.time = 10.0, bin.boundaries = c(0, 0.5, 1.0)),
               "`lambda` must be a non-negative scalar.")

  expect_error(survProbBins(survregnet_model, SomaSurvival::sim_surv_data,
                            eval.time = 10.0, bin.boundaries = c(0, 0.5, 1.0),
                            lambda = "a"),
               "`lambda` must be a non-negative scalar.")

  expect_error(survProbBins(survregnet_model, SomaSurvival::sim_surv_data,
                            eval.time = 10.0, bin.boundaries = c(0, 0.5, 1.0),
                            lambda = c(1, 2)),
               "`lambda` must be a non-negative scalar.")

  expect_error(survProbBins(survregnet_model, SomaSurvival::sim_surv_data,
                            eval.time = 10.0, bin.boundaries = c(0, 0.5, 1.0),
                            lambda = -0.5),
               "`lambda` must be a non-negative scalar.")
})
```

3.3.1.2 Input Processing

This is our first indication that the S3 based design may not be the most robust choice w.r.t. unit testing. Specifically, there is no clear way to test the input processing steps. We can only implement these steps locally and validate that `survProbBins()` returns the expected results assuming that the input processing steps are accurate.

```
test_that("`.survProbBins.coxph()` returns expected results", {

})
```

STILL NEEDS TO BE WRITTEN

Mention inability to test survival vs event probabilities

Bring in framework S3 methods to show how to test this

POINTS YET TO MAKE

Avoid hard-coding values when possible– they can be much harder to maintain.

Just like function development, avoid repeated code. If the test structures are the same for several test groups, create a local function defining the test structure and pass the values to be tested (give example)

Can recreate the behavior of the testthat expect_* functions (give example)

Hoping that we will implement 2 flags. First flag is nightly checks of the full test suite – testing all internal functions, etc. Second flag is a PR check level, skips the potentially longer duration units tests of each component, but ensures that the final result is what we expect.

Chapter 4

Pull Request (PR) Peer Review

Let's face it – we are not perfect. When we are in the thick of bug hunting or developing a new tool from scratch, it is easy to get tunnel vision or code blind. The PR peer review is meant to be a collaborative process that provides an opportunity to put a fresh set of eyes on the project. All of us have very different backgrounds and areas of expertise, and we approach problems from different perspectives. We can take advantage of this diversity to make the first level review process both an opportunity to ensure our tools are general, robust, and user-friendly as well as learn from each other!

Just to get this out of the way – PR peer review is **NOT about ensuring code style conformity**. This service is a part of the process, but it is not the point. As a peer reviewer, you should keep an eye out for such inconsistencies and make note of them. But do not let this become the focus of the review. If you see one inconsistency, make a comment at the location. If you see several, just add one comment "check spacing throughout function (or file) xyz" or "variable xyz uses a dot rather than an underscore to separate words", etc. Having 95 out of 100 comments that simply point out style issues can be frustrating for both the reviewer and the developer. Trust us, we've all been in both roles of that experience – it is not fun! Remember – the goal of a peer review is to write **better** code not **prettier** code. We will not mention this again

Okay. So our goal as peer reviewers is not to be the a style police. So, what is it?

Have you ever submitted or reviewed a peer-reviewed manuscript? The premise of PR peer review is very similar.BMC describes peer review as follows:

> Peer review is the system used to assess the quality of a manuscript before it is published. Independent researchers in the relevant re-

> search area assess submitted manuscripts for originality, validity and significance to help editors determine whether a manuscript should be published in their journal.

Replace just a few words, and we have a pretty good description of the PR peer review:

> PR peer review is the system used to assess the quality of ~~a manuscript~~ code before it is published. Independent ~~researchers in the relevant research area~~ developers assess submitted ~~manuscripts~~ code for ~~originality~~ efficiency, validity, and ~~significant~~ robustness to help ~~editors~~ package maintainers determine whether ~~a manuscript~~ code should be published in thier ~~journal~~ package.

Okay, so it isn't a perfect description, but it's helpful.

To provide effective peer-review requires that the reviewer understand the field, the problem being addressed, and the proposed solution. Their responsibility is to help identify any undocumented limitations or assumptions, oversights in implementation, errors in logic, or places for improvement. It can be a demanding and time-consuming request, but it is vital to ensuring that our code base meets the needs of our users. Remember, peer reviewers are not gatekeepers, their role is to collaborate with a developer with the joint goal of ensuring that the product is as accurate, robust, and efficient as possible.

Peer review is a dialogue, not a monologue!Be respectful, responsive, and open to discussion.

4.1 Pull Request Submission

It is the developer's role to set the stage for an effective and efficient peer review. This means that code modifications must be well documented.

The developer is typically far more aware of the intricacies of an implementation than anyone else – they've been working on this task for the last x hours, days, or months. As developers, we want to be kind not only to our users but also our reviewers and future developers!

Provide detailed explanations for motivation, implementation choices, and context. Point out known limitations, assumptions, bottlenecks, etc. Document these through either the initial pull request commit message and/or in the source code. For example, the initial Pull Request comment should

- provide a brief statement regarding the reason for the proposed modifications (bug report, feature extension, etc.)

- provide a short summary of the important changes made in the code submission
- recommend the order in which functions or files should be reviewed,
- identify open questions or trouble spots, etc.

Note that this is meant to be an introduction for the peer reviewer – there is no need to provide explicit details. For example

Modifications to `exampleFunction()` to address a bug report. Method failed when provided a tibble rather than a data.frame. Modified accessing functions to allow for both data.frames and tibbles.

4.2 Peer Review Preparation

Before starting any PR peer review (without looking at the proposed changes!), the reviewer should take the time to

Understand the motivation for and need addressed by the proposed changes.

- Does it address a specific bug report? What are the conditions under which the bug was encountered?
- Is it a feature improvement? What was the original functionality?
- Is it a package extension?

If additional information is needed, try

- reviewing the associated Jira task(s);
- reviewing comments provided by the developer during the PR submission;
- asking the developer for clarification through tracked BitBucket comments. We recommend doing this in BitBucket rather than say Slack or e-mail so that later reviewers can benefit from the conversation.

If the PR addresses a bug report or a feature extension,

Understand the original function to which the proposed changes are being made.

- What is the purpose of the affected function?
- What assumptions were made in the original implementation?
- Are there any inconsistencies with current guidelines? (e.g., uses `magrittr` pipes rather than native pipes)
- Are there any design or implementation choices that could be modified to improve user interaction, efficiency, or robustness?

Make notes. Maybe create a list of things to keep an eye out for when looking at the proposed changes. The primary purpose here is to ensure that the context in which the proposed changes are being made is fully understand.

If it turns out that items identified as possible limitations or points for improvement are not addressed by the current PR, include them in final comments so the team can discuss and perhaps incorporate them into the current PR or submit them as Jira tasks for future development.

If the PR is a new feature

Understand the theory behind the new feature.

This might be a significantly more time-consuming burden on the reviewer. Exactly what this means depends on the new feature. For example,

- Is it a new performance metric or regression method? Review and understand the governing equations as provided by the submitting developer. Work through the algebra/calculus. Review the relevant sections of the supporting references.
- Is it a new plotting function for specific returned value object? Review and understand the function that creates the object to be plotted.
- Is it a new Rmd doc? Review and understand the purpose and application for which the document will be used.

The preparation step gives us the foundation needed to provide thoughtful, comprehensive, and helpful feedback to the developer.

4.3 Map Out Your Own Approach

Without going into fine detail, ask yourself how you would/could address the proposed code changes. If the PR is addressing a reported bug, how would you correct it? If it is a feature extension, how would you incorporate it into the original function? If it's a new feature, what are the key steps that define the procedure?

Having these answers in mind during the review can help to identify unintended assumptions or limitations made in either our approach or the developer's solution as well as potentially identify or learn more efficient implementation and design choices.

Again, we are not recommending that you actually write code here. Just spend some time thinking about how you would complete the task.

Developing our own solution before looking at the proposed changes helps us to avoid locking onto the implementation choices made by the developer.

4.4 Review the Proposed Changes

Now we get to the nitty gritty of PR peer review. At this stage, take your first look at the proposed changes. Initially, you should simply want to understand the proposed code and determine if it is valid and accurate. This is not the time to consider alternative solutions. Ask:

- Are the proposed changes valid code?
- Do they address the bug or incorporate the feature as specified?
- Do they respect the assumptions stated in the documentation?
- Does the new code introduce new assumptions?
- Are there any unintended or undocumented limitations?
- Are inputs tested?
- Are intermediate results tested before use?
- Is there sufficient unit testing of new code?
- Are edge cases considered?
- Are NA/NaN/Inf conditions addressed?
- Are style conventions followed?

If something about the proposed changes is unclear, take some time to pull the branch and step through the function or ask the developer (through a comment in BitBucket) to clarify. If clarification is needed, be clear and detailed in your question: what is your current understanding; what aspect or result is not clear; etc. Comments are typically read by all reviewers in a PR process, so keep future reviewers in mind when formulating your questions.

First, ensure the accuracy and validity of the proposed changes.

4.5 Discuss Alternative Solutions

At this stage, you should be confident that the implementation is meeting the requested need and that it is accurately implemented or that any deficiencies have been identified during the review. Now, consider the questions of generality, robustness, and efficiency. This is when questions about alternative implementation and design choices should be considered. The “map your own approach” step is often helpful.

- Can any step be implemented using different tools? If yes, identify the advantages and limitations of those alternative w.r.t. the proposed implementation.
- Would a different design be more maintainable or efficient?
- Should intermediate steps be pulled out to improve unit testing coverage?
- Are there aspects of one implementation that are more general, extendable, efficient, or robust than the other?

The purpose of this stage is not to push your personal coding preferences on the developer. This step is meant to facilitate a discussion between you and the developer to ensure that the tool is robust, general, and efficient. It also can help us to better understand the impact of design choices, avoid unintentional implementation assumptions, and recognize points of optimization and generalization in our own development efforts.

If it is believed that an alternative solution should be considered, add a detailed comment providing at least a skeleton of the suggested alternative. Any advantages or limitations of the alternate solution as they compare to the proposed solution should be detailed. For example, "I have found that `abc` is often slower for large datasets than `xyz`. Will that be a concern here?"

Feel free to also share equivalent solutions solely for the purposes of discussion – "This is great! I typically would have used `abc` because of `xyz`, but I see no strong reason to do to so here." Maybe the developer is not familiar with `abc`. Or, maybe the developer considered it but opted for another solution because `xyz` is not valid in the context. **This is the stage of peer review during which we learn the most from one another!**

It is through the discussion of ideas that we, as a team, learn, grow, and succeed.