Journal of Statistical Software

July 2018, Volume 85, Issue 13. doi: 10.18637/jss.v085.i13

Developer-Friendly and Computationally Efficient Predictive Modeling without Information Leakage: The emil Package for R

Christofer L. Bäcklin
Uppsala University

Mats G. Gustafsson
Uppsala University

Abstract

Data driven machine learning for predictive modeling problems (classification, regression, or survival analysis) typically involves a number of steps beginning with data preprocessing and ending with performance evaluation. A large number of packages providing tools for the individual steps are available for R, but there is a lack of tools for facilitating rigorous performance evaluation of the complete procedures assembled from them by means of cross-validation, bootstrap, or similar methods. Such a tool should strictly prevent test set observations from influencing model training and meta-parameter tuning, so-called information leakage, in order to not produce overly optimistic performance estimates.

Here we present a new package for R denoted **emil** (evaluation of modeling without information leakage) that offers this form of performance evaluation. It provides a transparent and highly customizable framework for facilitating the assembly, execution, performance evaluation, and interpretation of complete procedures for classification, regression, and survival analysis. The components of package **emil** have been designed to be as modular and general as possible to allow users to combine, replace, and extend them if needed. Package **emil** was also developed with scalability in mind and has a small computational overhead, which is a key requirement for analyzing the very big data sets now available in fields like medicine, physics, and finance.

First package **emil**'s functionality and usage is explained. Then three specific application examples are presented to show its potential in terms of parallelization, customization for survival analysis, and development of ensemble models. Finally a brief comparison to similar software is provided.

Keywords: predictive modeling, machine learning, performance evaluation, resampling, high performance computing.

1. Introduction

Data driven machine learning for predictive modeling problems (classification, regression, and survival analysis) is employed in virtually all scientific domains and rapidly grows in importance in the context of what is called "big data" (Snijders, Matzat, and Reips 2012; Wu, Zhu, Wu, and Ding 2014). In this context a prediction model is a computable function f that transforms an input feature vector x into a prediction y determined as $y = f(x)$ where y may have one or several dimensions. Usually the main purpose of a predictive modeling exercise is to provide decision support in the form of actual predictions for new inputs of interest as well as in the form of assigning importance to the features in the input vector x. Predictive modeling typically involves the following three steps: (1) data pre-processing including imputation of missing values, raw data transformations and/or feature selection; (2) parameter fitting and model family selection including tuning of meta-parameters or model selection by an information theory based criterion, like AIC (Akaike 1973) or BIC (Schwarz 1978); and (3) prediction using the selected final prediction model.

When performing predictive modeling using a particular computational procedure consisting of a pre-processing part (step 1 above) and a modeling procedure part (step 2 above), a problem of fundamental interest is to characterize and quantify the overall prediction performance of the models produced by the procedure. This is because the computational procedure used might be too liberal or too constrained for the underlying prediction problem of interest. If too liberal the final prediction models produced will have been fitted too well to the design/training examples used (overfitting) and will therefore perform poorly on new external test examples. If the computational procedure would be too constrained the models would not be flexible enough to enable sufficiently close fitting to the design examples. Typically the overall prediction performance, especially the expected average performance of the prediction models produced, is estimated using testing on external examples that have not at all been part of the pre-processing or the modeling procedure. In order to obtain a reliable characterization that is not dependent on one particular split of the available data into one set of examples for training and one set of examples for external performance estimation, usually cross-validation (CV), or some other resampling method, is employed to build multiple prediction models, each tested on independent external test examples (Hastie, Tibshirani, and Friedman 2001; Varma and Simon 2006; Lawless and Yuan 2010). If test examples are allowed to influence the modeling because they are not excluded from pre-processing and the modeling procedure there is a risk of positively biasing the resulting performance estimate. We will refer to such influences as *information leaks*. Examples of such leaks include imputation or feature selection methods applied to an entire data set prior to resampling based performance evaluation.

When performing the kind of performance evaluation of a computational procedure used for predictive modeling described above, it is desirable to use an existing computational framework that takes care of the tedious but straight-forward book-keeping aspects, yet is flexible enough to allow for easy customization and comparison to alternative solutions, while not adding an unnecessarily large computational or memory burden. The package **emil** (Bäcklin and Gustafsson 2018) for R (R Core Team 2018) is designed to achieve precisely this. It provides a simple, transparent, light-weight, and well documented framework encompassing all of the aspects inherent to data driven machine learning based predictive modeling. Rather than providing a high level analysis platform rich in pre-defined features package **emil** strives towards being an application programming interface (API) with highly reusable components

and consistent syntax and data structures. This results in analysis scripts that are easy to read, low in maintenance, and easy to debug. Package **emil** does however include several methods for each analysis step that can be used to do analyses out-of-the-box and serve as examples of how to implement customized methods. The usability of package **emil** has recently been appreciated in a master of science engineering program course focused on multivariate data analysis as well as in a few master thesis projects.

First we present an overview of the functionality of the **emil** package, followed by three application examples, and lastly we provide a comparison to the **caret** package (Kuhn 2008; Kuhn *et al.* 2018) which provides partially overlapping functionality. Functions used in the examples from packages other than **emil** and **caret** are summarized in Table 1. For a comprehensive guide to all aspects of the **emil** package's usage, please refer to its included documentation (accessible from within R by entering `?emil`). Package **emil** is publicly available for download at the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=emil`, and all code and data required to run the examples and benchmarking is publicly available at `http://github.com/Molmed/Backlin-2017`.

1.1. Notation and terminology

In this work the following notation and terminology is used (as partly outlined by Hastie *et al.* 2001). The structure and adjustable parameters of a prediction model f yielding predictions $y = f(x)$ is said to be "fitted" to a dataset. A dataset consists of a matrix or data frame x of size $n \times p$, where n denotes individual "observations" and p denotes "features", and a vector of "response values" y of length n. The words "train" and "design" commonly used elsewhere have the same meaning as "fit" and can be used interchangeably. Such a relationship is also the case for the groups of words "observations", "objects", and "examples"; "variables", "attributes", "independent variables", and "predictor variables"; and "response", "response variable" and "dependent variable". Also note that x and y may have a different structure elsewhere. For example, x may be a collection of documents stored as text and y may be a matrix with multiple response values for each observation.

Often the fitting procedure used has one or several meta-parameters that also need to be selected based on the data available. One classical example is ridge regression (Hastie *et al.* 2001) where the resulting prediction model can be written as the linear model $y = w_\theta^\top x$ where the coefficient vector w_θ is determined using a fitting procedure that involves a penalty (regularization) term having an influence determined by a meta-parameter θ. Another classical example is the number of latent variables when performing linear regression using the partial least squares algorithm (PLS, Hastie *et al.* 2001). Also in this case a linear prediction model $y = w_k^\top x$ is built where the coefficient vector w_k is determined using ordinary least squares fitting in a space constrained by k latent variables where k has to be tuned/selected based on the set of training examples available.

In addition to the kind of meta-parameters discussed above, in many cases one also has to choose between different model classes. For example, in regression modeling it is common to choose between prediction models that are linear (i.e., have the form $y = w^\top x$), prediction models based on standard binary regression trees (RTs), prediction models based on k-nearest-neighbor regression (kNNR), and models based on multilayer perceptron artificial neural networks (MLPANNs) (Hastie *et al.* 2001). For each of these classes there are meta-parameters to tune (for linear models see above, for RTs the depth of the tree, for kNNR one has to choose

Function	Package	Description
`%>%`	**magrittr**	Forward-pipe operator. Like the Unix pipe operator `|` it passes all left hand side contents as an argument to the right hand side function. For example `x %>% f` is evaluated as `f(x)` and `x %>% f %>% g(arg = "value")` is evaluated as `g(f(x), arg = "value")`.
`aes`	**ggplot2**	Generates aesthetic mappings that control how the contents of data are presented in a '`ggplot`' object.
`c`	**base**	Concatenate multiple values or vectors into a single vector.
`do.call`	**base**	Use the elements of a list as arguments to a function call, e.g., `do.call(f, x)` where `x = list(x1, x2, x3)` executes `f(x1, x2, x3)`. This is useful if the structure of `x` may vary.
`exprs`	**Biobase**	Extracts gene expression data from a **Bioconductor** dataset.
`findInterval`	**base**	Given a set of breakpoints and a set of elements, return the interval indexes in which each element occurs, e.g., `findInterval(c(4, 6, 8), 7.83)` is 2.
`geom_point`	**ggplot2**	Adds a layer of scatter plot points to a '`ggplot`' object.
`ggplot`	**ggplot2**	Creates a '`ggplot`' object onto which layers of graphics can be added e.g., using `geom_point`.
`gl`	**base**	Generate a categorical vector containing class labels.
`lapply`	**base**	Applies a function to each element of a list and returns the output as a new list.
`Map`	**base**	Similar to `lapply` but accepts multiple lists of input values to be processed together.
`mclapply`	**parallel**	Executes a function on each item of a list in parallel.
`pData`	**Biobase**	Extract phenotype data from a **Bioconductor** dataset.
`select`	**dplyr**	Selects a subset of columns in a data frame. The **emil** package implements a class specific version of `select` for its own result objects (Section 2.6).
`spread`	**tidyr**	Converts data frames from long format to wide format.
`t`	**base**	Matrix transpose operator.
`table`	**base**	Counts the number of occurrences of each value in a vector.
`tempdir`	**base**	Path to a directory where temporary results can be saved (created automatically by R).
`try`	**base**	Wrapper that catches and silences errors in a block of code instead of breaking execution.
`tryCatch`	**base**	Similar to `try` but provides more error handling capabilities.
`with`	**base**	Evaluates an expression with the variables of a data frame. This is only for improving code readability, since `with(d, x + y + z)` is sometimes preferable to `d$x + d$y + d$z`.

Table 1: Key functions used in the code examples of the paper but which are not part of the packages **emil** and **caret**. The packages **base**, **parallel**, and **stats** are included in the standard distribution of R (R Core Team 2018). The package **Biobase** (Huber *et al.* 2015) contains the core functionality of the **Bioconductor** platform (Gentleman *et al.* 2004), which is widely used to do bioinformatic analyses.

the value of k, for MLPANN one has to choose the number of hidden layers and hidden nodes) and on top of this one also has to choose the class of the final model, which is to be used for prediction of unknown examples and for interpretation of important variables etc. Therefore one should regard the variable used to encode the model class as another meta-parameter that has to be tuned based on the training examples available. Thus the "fitting" of the structure and the parameters of one prediction model to data can also be viewed and interpreted as the result of a fitting procedure that selects one model (defined by its specific structure and associated parameter values) in the space of all conceivable models spanned by all plausible structures and parameter values.

2. Functionality

The main objective of the **emil** package is to build and evaluate different computational procedures for predictive modeling using resampling based testing. It is particularly developed for providing unbiased performance estimation, an important task when the number of samples is small compared to the number of features and/or compared to the number of model parameters including alternative model structures. Under such conditions there is always a serious risk of overfitting models (structure and parameters) as well as making overly optimistic performance estimates. As already mentioned, most fitting algorithms contain meta-parameters for controlling overall model complexity, e.g., the amount of penalty (regularization) in ridge regression or the depth of a regression tree, apart from the ordinary parameters obtained as a consequence of the choice of meta-parameter value, i.e., the actual coefficients in the resulting linear prediction model (ridge regression) and the variables and corresponding split points in the regression tree. These meta-parameters are typically also tuned with resampling based testing and it is important to keep this exercise strictly inside the training set, requiring a two-layered resampling scheme, for combined tuning and unbiased performance estimation (Hastie *et al.* 2001; Varma and Simon 2006; Lawless and Yuan 2010).

Consider the task where a data set D is to be pre-processed according to a pre-processing algorithm A and then used by an modeling procedure $B(\theta)$, defined by a user-defined meta-parameter θ, to create a prediction model f_θ. Returning to the ridge regression example above as one illustration, θ is known as the ridge/penalty parameter and controls the amount of regularization used when determining the optimal coefficient vector w_θ. In order to obtain an unbiased estimate/characterization of the performance of such a modeling procedure one needs to perform a two-layered resampling procedure outlined in Algorithm 1. Such a procedure inevitably requires many data subsets to be generated and processed, and it is of vital importance to make sure that no test set is allowed to influence the contents of any training set, including all data pre-processing steps, in order to not bias the final performance estimate.

Although A generally is an unsupervised method that only serves to prepare the features of a data set for analysis, such as a normalization, imputation or compression technique, it is important to acknowledge the fact that it is the combination of A and $B(\theta)$ that constitutes the complete computational procedure for producing a model from a raw data set. There is no theoretical reason for separating A and $B(\theta)$ into different entities, but in practice it is convenient to do so to enable different implementations of A and $B(\theta)$ to be used and combined independently of one another. This also allows the results from A to be reused together with a different $B(\theta)$, which can save a lot of time if A is a time consuming algorithm.

Algorithm 1 Two-layered resampling.

loop for performance evaluation.
  Randomly split D into mutually exclusive subsets D_f (fitting) and D_t (testing).
  loop for meta-parameter tuning.
    Randomly split D_f into mutually exclusive subsets D_{ff} and D_{ft}.
    Define a normalization transform n_{ff} using the algorithm A applied to D_{ff} alone.
    for each possible value of the meta-parameter θ **do**
      Fit one model f_θ using $B(\theta)$ applied to the transformed data set $n_{ff}(D_{ff})$.
      Estimate the performance of f_θ using the transformed test set $n_{ff}(D_{ft})$.
    end for
    return Performance estimates for all parameter values.
  end loop
  Determine the most promising value θ^* among the tested values for θ.
  Define a normalization transform n_f using the algorithm A applied to D_f alone.
  Fit one model f_{θ^*} using $B(\theta^*)$ applied to the transformed data set $n_f(D_f)$.
  Estimate the performance of f_{θ^*} using the transformed test set $n_f(D_t)$.
end loop
return Performance estimates for each random split.

In the **emil** framework, A is referred to as *pre-processing* method and B as *modeling procedure*. The **emil** function `evaluate` can be used to perform the work of Algorithm 1, and the code below presents an example of how it can be used (referred to as *the main example* throughout Section 2). The different concepts needed to understand how it works are explained in Sections 2.1–2.4, how to work with the results produced is explained in Sections 2.5–2.7, and some notes on scalability are provided in Section 2.8.

```
R> data("prostate", package = "ElemStatLearn")
R> cv <- resample(method = "crossvalidation", y = prostate$lpsa,
+    nfold = 3, nrepeat = 2)
R> result <- evaluate(procedure = "lasso", x = prostate[1:8],
+    y = prostate$lpsa, resample = cv,
+    pre_process = function(x, y, fold) {
+      data <- pre_split(x, y, fold)
+      data <- pre_scale(data, center = TRUE)
+      data <- pre_convert(data, x_fun = as.matrix)
+      return(data)
+    })
R> get_performance(result)

       fold error
1 rep1fold1 0.735
2 rep1fold2 0.767
3 rep1fold3 0.742
4 rep2fold1 0.807
5 rep2fold2 0.758
6 rep2fold3 0.669
```

Briefly, the code of the example sets up a LASSO regression procedure (Tibshirani 1996) and evaluates its performance using 3-fold CV repeated two times. The result obtained with the `get_performance` function is the estimated root mean square error obtained in each fold.

2.1. Data

By default a data set is represented in package **emil** as a matrix or data frame `x` with observations as rows and features as columns, accompanied by a response vector `y`. There are no restrictions on the types of data `x` and `y` may contain as long as appropriate modeling functions are used to handle them. The most common case is that `y` is either a numeric implying regression, a factor implying classification, or time-to-event vector of class '`Surv`' implying survival analysis, but any type of response could, in principle, be modeled. To improve compatibility with other packages, data can alternatively be supplied as `x` holding the entire data set and `y` holding a formula or character scalar marking which feature to be used as response.

The data set of the main example, `prostate`, was first presented by Stamey *et al.* (1989) and is available in the R package **ElemStatLearn** (Halvorsen 2015). It contains the level of prostate specific antigen (PSA) in 97 men prior to prostate cancer treatment by radical prostatectomy, together with 8 clinical features that the model building is to be based upon.

2.2. Resampling

Resampling schemes are used to define how the data set is to be split and used for model fitting and testing. Such schemes are created with the function `resample` that takes a method and the response vector `y`. This vector `y` is needed to allow the resampling method to generate folds that are balanced with respect to the different classes (i.e., preserving their relative frequencies).

```
R> cv <- resample(method = "crossvalidation", y = prostate$lpsa,
+    nrepeat = 2, nfold = 3)
R> head(cv)
```

```
  rep1fold1 rep1fold2 rep1fold3 rep2fold1 rep2fold2 rep2fold3
1     FALSE      TRUE      TRUE      TRUE      TRUE     FALSE
2      TRUE     FALSE      TRUE      TRUE     FALSE      TRUE
3      TRUE      TRUE     FALSE     FALSE      TRUE      TRUE
4      TRUE      TRUE     FALSE     FALSE      TRUE      TRUE
5      TRUE      TRUE     FALSE      TRUE     FALSE      TRUE
6      TRUE     FALSE      TRUE     FALSE      TRUE      TRUE
```

Each column in the output above represents one *fold*, describing which observations are to be used for training (`TRUE`) and testing (`FALSE`) for a particular split of the data set. Internally, `resample` calls the function `resample_crossvalidation` to generate the scheme and adds a couple of additional attributes to it (not visible in the code above) to allow inner resampling schemes to be generated from it (see `?subresample`). Inner resampling schemes are needed for meta-parameter tuning in each of the folds in the outer loop.

The user may implement and use a custom resampling algorithm by creating a function `resample_my_method` and then call it with `resample(method = "my_method", y)` (see `?resample` for details).

2.3. Splitting and pre-processing

Given a fold defining how the data is to be split the user can now proceed to carry out data pre-processing. This is typically done using a sequence of functions that represent the various actions and transformations the user wants to apply to the data set. The sequence is started with `pre_split` that sifts out the relevant data subsets and passes them on in a list that can be modified by the subsequent functions. The code below first splits the data, then scales all features to have mean 0 and unit standard deviation to allow for fair comparison between them, and finally converts `x` to a matrix rather than a data frame since the LASSO method used in the main example requires that form:

```
R> prostate_split <- pre_split(x = prostate[1:8], y = prostate$lpsa,
+    fold = cv[[1]])
R> prostate_split <- pre_scale(prostate_split, center = TRUE)
R> prostate_split <- pre_convert(prostate_split, x_fun = as.matrix)
R> prostate_split

Pre-processed data set `prostate[1:8]` of 8 features.
64 observations for model fitting,
33 observations for model evaluation.
```

Equivalently, the pipe notation `%>%` introduced in the **magrittr** package (Milton Bache and Wickham 2014) can be used to express the sequence of pre-processing functions as a single chained command, which is less bulky and arguably easier to read (see Table 1 for details):

```
R> prostate_split <-
+    pre_split(x = prostate[1:8], y = prostate$lpsa, fold = cv[[1]]) %>%
+    pre_scale %>% pre_convert(x_fun = as.matrix)
```

Package **emil** contains a number of pre-processing functions ready to be combined and new ones can easily be written by the user (see `?pre_process`). The `pre_pca` function performing principal component analysis (PCA, Hastie *et al.* 2001, Chapter 14.5) constitutes a concise example of how such a function should be defined:

```
R> pre_pca

function (data, ...)
{
    pca <- prcomp(data$fit$x, ...)
    data$fit$x <- pca$x
    data$test$x <- predict(pca, data$test$x)
    data
}
<environment: namespace:emil>
```

In such well structured code it is easy to spot and correct any source of information leakage between the fitting and testing data. The pre-processing routine can be passed to `evaluate` using a wrapper function, as in the main example, or as a list of functions as shown here:

```
R> result <- evaluate(procedure = "lasso", x = prostate[1:8],
+    y = prostate$lpsa, resample = cv,
+    pre_process = list(pre_split, pre_scale,
+      function(data) pre_convert(data, x_fun = as.matrix)))
```

It is worth highlighting the fact that the data is only extracted and pre-processed right before model fitting in order to avoid producing any unnecessary memory copies of the data (or subsets of the data). This means that no outer datasets (D_f and D_t in Algorithm 1) are created until the parameter tuning is completed, producing at most two copies of the data set simultaneously in memory.

2.4. Model fitting and testing

When evaluating a procedure using the `evaluate` function, models are automatically fitted and tested according to the resampling scheme (Section 2.2). This means a collection of models will be produced and tested, one per fold in the scheme. The user can however also create and test individual models in the following way, where `rmse` denotes the calculation of the root mean squared error:

```
R> model <- fit(procedure = "lasso", x = prostate_split$fit$x,
+    y = prostate_split$fit$y)
R> prediction <- predict(object = model, x = prostate_split$test$x)
R> rmse(prostate_split$test$y, prediction)

[1] 0.737
```

The prediction object is a list containing various quantities the user might be interested in depending on the modeling method of choice, such as predicted class labels, estimated class probabilities, or prediction intervals. Functions such as `error_rate` or `roc_curve` can then be used to summarize the performance (see `?error_fun`). Predictions from `evaluate` can be tabulated with the function `get_prediction`.

```
R> head(get_prediction(result, resample = cv, format = "wide"))

  id rep1fold1 rep1fold2 rep1fold3 rep2fold1 rep2fold2 rep2fold3
1  1     0.965        NA        NA        NA        NA     0.885
2  2        NA     1.073        NA        NA      1.07        NA
3  3        NA        NA     1.048     1.004        NA        NA
4  4        NA        NA     0.895     0.849        NA        NA
5  5        NA        NA     1.894        NA      1.92        NA
6  6        NA     0.991        NA     0.887        NA        NA
```

The argument `procedure` to the function `fit` is the key to choosing or customizing modeling algorithms. It takes an object of class '`modeling_procedure`' created with the function

`modeling_procedure` or alternatively any object that can be coerced to that class, such as the character `"lasso"` used in the main example. Information on which methods are available in package **emil** and other installed compatible packages can be found in the manual page `?list_method`.

```
R> rf <- modeling_procedure(method = "randomForest",
+    parameter = list(mtry = c(1, 3, 6)))
R> rf

`randomForest` modeling procedure.

   model fitting function:       yes
   prediction function:          yes
   feature importance function:  yes
   individual error function:    no

   number of parameter sets to tune over: 3
   tuned: no
```

The result is a list of wrapper functions that define how to fit and evaluate models of a particular method and what parameter values to use. To tune a parameter simply supply all values to consider in a list or vector (see `?modeling_procedure` for different options). If multiple parameters are to be tuned, all combinations are automatically tested unless otherwise specified. Parameter tuning statistics can then be extracted with the function `get_tuning`.

```
R> tuned_rf <- tune(rf, x = prostate[1:8], y = prostate$lpsa, resample = cv)
R> get_tuning(tuned_rf)

  parameter_set mtry rep1fold1 rep1fold2 rep1fold3 rep2fold1 rep2fold2
1             1    1     0.694     0.878     0.877     0.967     0.773
2             2    3     0.633     0.845     0.814     0.913     0.723
3             3    6     0.649     0.811     0.773     0.908     0.727
  rep2fold3
1     0.697
2     0.673
3     0.670
```

Isolating all wrapper functions in a single object removes any ambiguity of what function is actually called when performing the work, as the user is free to inspect them, e.g., with the functions `print` or `page`.

```
R> rf$fit_fun

function (x, y, ...)
{
    nice_require("randomForest")
    tryCatch(randomForest::randomForest(x, y, ...),
```

```
        error = function(e) {
            if (any(is.na(x))) {
                stop("Random forest does not accept any missing values.")
            }
            else {
                stop(e)
            }
        })
}
<environment: namespace:emil>
```

It also enables the user to easily change plug-in functions (see `?extension`) and facilitates debugging with the standard facilities of R. The example below invokes the debugger when the function used for fitting is called for the first time. From the progress log, printed by setting `.verbose = TRUE` (see Section 2.8 for more details), the user can tell that it happened in the inner loop of the two-layered resampling procedure (Algorithm 1), the parameter values and data to be used to fit this particular model can be inspected, and the code can be stepped through line by line while looking for problems.

```
R> debugonce(rf$fit_fun)
R> rf_result <- evaluate(procedure = rf, x = prostate[1:8],
+    y = prostate$lpsa, resample = cv,
+    pre_process = list(pre_split, pre_scale), .verbose = TRUE)


08 jul 09:51  Evaluating modeling performance...
08 jul 09:51    Repeat 1, fold 1
08 jul 09:51      Tuning parameters...
08 jul 09:51        Evaluating modeling performance...
08 jul 09:51          Repeat 1, fold 1
08 jul 09:51            Extracting fitting and testing datasets.
08 jul 09:51            Fitting randomForest (1)
debugging in: p$fit_fun(x, y, ...)
debug: {
    nice_require("randomForest")
    tryCatch(randomForest::randomForest(x, y, ...),
        error = function(e) {
            if (any(is.na(x))) {
                stop("Random forest does not accept any missing values.")
            }
            else {
                stop(e)
            }
        })
}
```

2.5. Model interpretation

Package **emil** contains a number of functions for extracting and summarizing results produced during the modeling process. Apart from the functions `get_performance`, `get_prediction`, and `get_tuning` introduced earlier there is also a function `get_importance` for extracting feature importance estimates.

```
R> lasso <- modeling_procedure("lasso")
R> model <- fit(procedure = lasso, x = prostate_split$fit$x,
+    y = prostate_split$fit$y)
R> get_importance(model)

  feature coefficient
1  lcavol       0.715
2 lweight       0.100
3     age      -0.135
4    lbph       0.246
5     svi       0.315
6     lcp       0.000
7 gleason       0.000
8   pgg45       0.000
```

How such estimates are calculated depends on the modeling procedure used. In the case of LASSO above the importance estimates are simply the fitted model coefficients w_θ, mentioned in Section 1.1. To see exactly how they were obtained the user can view the code of the procedure specific importance function by entering the following command (output not shown due to its length).

```
R> lasso$importance_fun
```

The reason `get_importance` should be called on a fitted model rather that the procedure itself is that the final model parameters are typically needed to estimate the features' importance. The model also contains a copy of the procedure used to create it, allowing it to find the correct extraction function.

2.6. Downstream analysis

The functions presented in Sections 2.2–2.5 constitute the base of the **emil** framework. Figure 1 provides a schematic overview of them (blue and green) together with the types of input and output they require and produce (yellow). For most applications the objects can be thought of as belonging to one of three groups: those related to the data, those related to the modeling, and those that are end results. Data and result objects are generally represented as data frames that can directly be passed to high level data manipulation and plotting tools, such as **dplyr** (Wickham and François 2017), **tidyr** (Wickham 2014), **ggplot2** (Wickham 2009), or **lattice** (Sarkar 2008). The modeling objects are all lists to allow for any type of contents that the user might want to create during the modeling, including elements that have not been foreseen by the authors of package **emil**. Internally, the modeling functions reuse the components of Figure 1 (see Figure 2) giving all modeling objects a consistent structure.

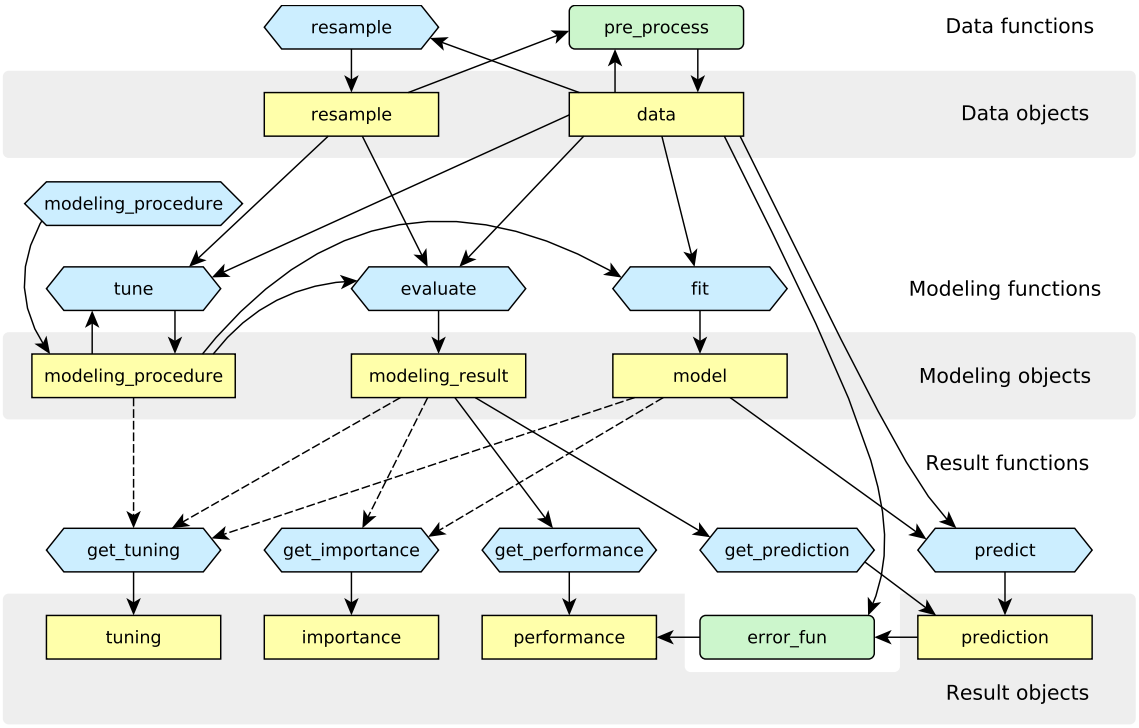

Figure 1: A schematic overview of the most common functions (blue) and object types (yellow) that the user works with in the **emil** package. The green rounded squares mark functions for which there are several alternatives from which the user must choose. Solid incoming arrows to a node mark required data objects for producing the output marked by the single outgoing arrow and dashed incoming arrows mark optional inputs, e.g., `fit` requires data and a modeling procedure to create a model. As another example, performance estimates of a procedure can be obtained by calling the function `get_performance` on an object of class `modeling_result` created by `evaluate`. `evaluate` in turn requires data, a resampling scheme, and a modeling procedure.

This allows partial results to be easily retrieved and condensed, typically using the functions `subtree` or `select`, which is an **emil** specific extension to the **dplyr** function `select`.

For example, objects of class '`modeling_result`' are nested lists where each top-level-element corresponds to a fold in the resampling scheme. All top-level-elements contain the same types of information (a model, predictions, a performance estimate, etc.) and these are of the same form as returned by `fit` or `predict` etc. These are referred to as second-level-elements. The following commands can be used to extract only the performance estimates.

```
R> subtree(x = result, TRUE, "error")

rep1fold1 rep1fold2 rep1fold3 rep2fold1 rep2fold2 rep2fold3
    0.737     0.768     0.752     0.817     0.759     0.671
```

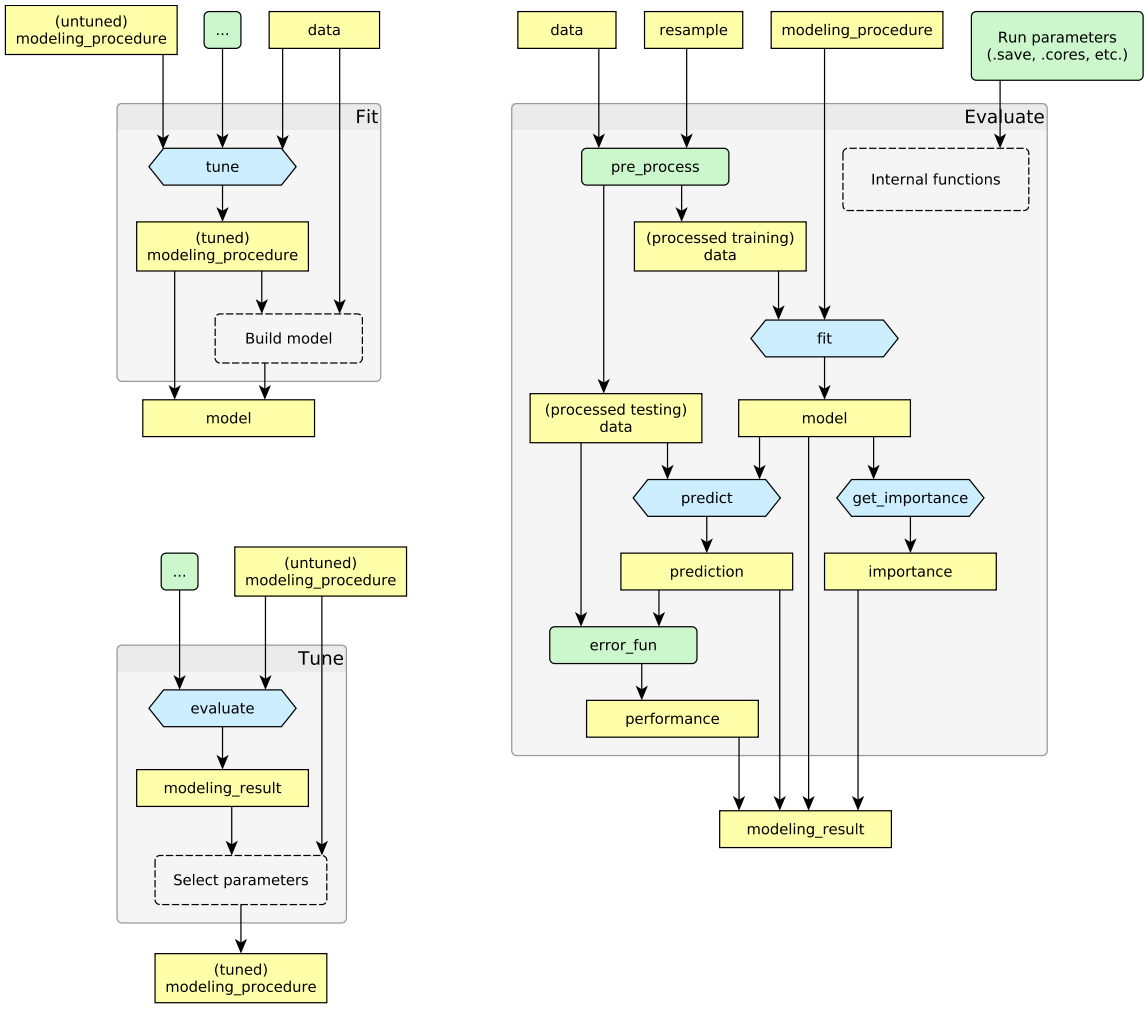

Figure 2: The inner structure of the functions `fit`, `tune`, and `evaluate`. The same components that the user has access to are reused internally, making the framework produce results and logs that are consistent in structure and easy to process and follow. The `fit` function creates predictive models by first tuning any untuned meta-parameters contained in the modeling procedure and then calling its fitting function. The `tune` function selects meta-parameter values by splitting a modeling procedure into separate procedures with fixed meta-parameter values, evaluating all of them and selecting the one that performs the best. The `evaluate` function asserts how well the complete computational procedure including pre-processing and modeling does. `evaluate` performs the work outlined in its box once per fold of the resampling scheme.

The first unnamed argument[1] to `subtree` above specifies which top-level-elements of `result` that should be selected (`TRUE` meaning all folds), and the second unnamed argument specifies

[1]The second and third arguments are unnamed since they lack an assignment operator (`=`) and a left-hand-side name. Unnamed arguments are assigned to the appropriate variable inside the function based on their position in the call. Due to the recursive structure of `subtree` and use of the dots argument "`...`" the indexing arguments to `subtree` are typically not named.

that what second-level-elements are to be selected (only elements named `"error"`).

The `select` function works in a similar way but always produces a data frame[2], which can be directly parsed by the high level data manipulation tools mentioned above.

```
R> select(result, Fold = TRUE, RMSE = "error")

       Fold  RMSE
1 rep1fold1 0.737
2 rep1fold2 0.768
3 rep1fold3 0.752
4 rep2fold1 0.817
5 rep2fold2 0.759
6 rep2fold3 0.671
```

In cases where the conversion from list element to data frame is not obvious a function returning a data frame can be used instead of an indexing vector or value. Let us return to the main example for a demonstration of this functionality. The modeling procedure internally used the **glmnet** package (Friedman, Hastie, and Tibshirani 2010) for fitting LASSO models, but rather than using the meta-parameter tuning framework of package **emil** to tune its complexity parameter λ, the fitting function calls a native tuning function provided in the **glmnet** package (for improved computational efficiency). Because of this, the function `get_tuning` will not be able to extract the tuning statistics of λ, but this can easily be done using `select`.

```
R> internal_tuning <- select(result, Fold = TRUE, "model", "model",
+    function(m) data.frame(Lambda = m$lambda, TuningRMSE = m$cvm))
R> head(internal_tuning)

       Fold Lambda TuningRMSE
1 rep1fold1  0.950       1.55
2 rep1fold1  0.865       1.44
3 rep1fold1  0.789       1.30
4 rep1fold1  0.719       1.19
5 rep1fold1  0.655       1.09
6 rep1fold1  0.597       1.02

R> library("ggplot2")
R> ggplot(internal_tuning, aes(x = Lambda, y = TuningRMSE, group = Fold)) +
+    geom_line()
```

The plot of the above code is presented in Figure 3.

The result functions introduced earlier, such as `get_performance`, can also be used with `select` (see the example in the end of Section 2.7 for a demonstration).

[2]Named arguments to `select` will produce columns in the resulting data frame and non-named arguments will continue the traversing without producing any columns.

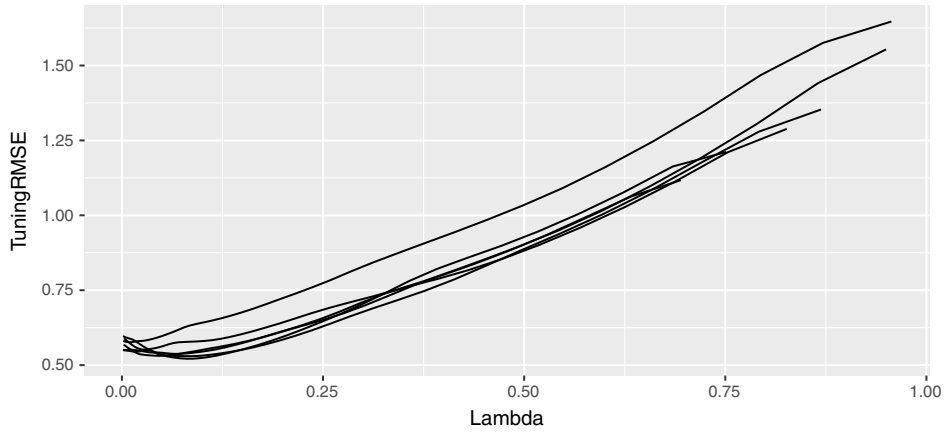

Figure 3: The figure created by the `select` and **ggplot2** commands of Section 2.6.

2.7. Procedure comparison

Comparing different modeling procedures on the same problem is a common task in predictive modeling. Since the resampling scheme and pre-processing steps must be the same for all modeling procedures to allow for a fair comparison a lot of time may be saved by not repeating it. All modeling functions therefore allow the user to supply multiple modeling procedures, in which case the same splitting and pre-processing chain is used for all.

If the user wishes to compare LASSO regression to ridge regression on the task set up in the main example, the only change needed in the code is to supply multiple procedures in a vector or list.

```
R> comparison <- evaluate(
+    procedure = c(LASSO = "lasso", RR = "ridge_regression"),
+    x = prostate[1:8], y = prostate$lpsa, resample = cv,
+    pre_process = function(x, y, fold) {
+      pre_split(x, y, fold) %>% pre_scale %>%
+        pre_convert(x_fun = as.matrix)
+    })
R> get_performance(comparison, format = "wide")

       fold LASSO    RR
1 rep1fold1 0.737 0.718
2 rep1fold2 0.769 0.762
3 rep1fold3 0.752 0.748
4 rep2fold1 0.807 0.833
5 rep2fold2 0.761 0.743
6 rep2fold3 0.663 0.652
```

The example above also illustrates how procedures can be named, by setting `c(LASSO = "lasso", RR = "ridge_regression")`. The names (i.e., `"LASSO"` and `"RR"`) are preserved throughout the logs and results, and are automatically guessed from the procedures themselves if not given by the user.

2.8. Scalability

The **emil** package offers several tools that can help the user perform time-consuming and resource intensive computations: parallelization, checkpointing and an advanced progress logging system.

Multicore parallelization is controlled using the argument `.cores` to `evaluate`, specifying how many cores to use. In the current version of package **emil**, cluster parallelization must be set up manually as demonstrated below. Although the setup is quite simple, we thought it best to leave it to the user since cluster interfaces may differ between platforms, but more assistance to the user may be added in the future. It is also worth mentioning that cluster and multicore parallelization may be performed together, which is useful when analyzing large datasets with multiple computers (the code below uses 4 computers with 8 cores each).

```
R> library("parallel")
R> cluster <- makeCluster(spec = 4)
R> clusterEvalQ(cluster, library("emil"))
R> clusterExport(cluster, "prostate")
R> my_evaluate <- function(fold) {
+    my_pre_process <- function(x, y, fold) {
+      pre_split(x, y, fold) %>% pre_scale %>%
+        pre_convert(x_fun = as.matrix)
+    }
+   evaluate(procedure = "lasso", x = prostate[1:8], y = prostate$lpsa,
+     resample = fold, pre_process = my_pre_process, .cores = 8,
+     .checkpoint_dir = tempdir())
+  }
R> result <- parLapply(cluster, cv, my_evaluate)
```

Checkpointing (see argument on the second to last row above) is a technique for saving the results of each fold to disk as soon as it is completed, which reduces the need to redo work in case the execution is interrupted. When working on large computer clusters the user is typically required to request resources for a limited time ahead of execution, and since it is usually difficult to accurately estimate the running time beforehand this feature can rescue much results in case the running time was underestimated.

Finally, logging can be enabled by setting the argument `.verbose = TRUE` to `fit`, `tune`, or `evaluate`. The logging system uses the **emil** functions `log_message` and `indent` which can also be utilized by the user to create nicely formatted progress reports in time consuming custom written fitting functions.

2.9. Distribution of user created methods

Custom methods for generating resampling schemes, pre-processing, fitting models, estimating performance etc. can be written and distributed in separate packages or plain R script files. The internal documentation page `?extension` provides details on how the functions should be written to be fully compatible with the framework. Once a package extending package **emil** is installed and loaded its contained methods will automatically be detected by package **emil**.

3. Application examples

In this section three applications are presented, using the **emil** package with custom preprocessing and fitting functions to show its flexibility. Section 3.1 shows an implementation of random forest with parallelization at a different level than what package **emil** provides by default. Section 3.2 demonstrates a method to combine PCA and proportional hazards regression (referred to as *Cox regression* below, Cox 1972) to solve a high dimensional survival analysis task. Section 3.3 demonstrates a method to create an ensemble classifier on the fly by combining already implemented classification methods.

3.1. An alternative parallelization routine

How to perform a modeling task in the most resource efficient way is not the same across all modeling algorithms and applications. Ensemble methods such as random forest provide a nice illustration of this dilemma through the fact that parallelization may be performed on different levels, which leads to different memory requirements and computation times.

Consider the following two ways to parallelize the **emil** framework on a high performance machine with 16 CPU cores. The aim is to evaluate a procedure in which random forest classifiers containing 8000 decision trees are trained and used for classification by 8-fold cross-validation repeated 4 times (32 folds in total). The default implementation executes the 32 folds in parallel, distributed on the 16 cores, where each core fits all 8000 trees of 2 forests (Figure 4 top). The alternative implementation executes the 32 folds sequentially, but distributes the trees of each fold so that each core fits 500 trees of 32 forests (Figure 4 bottom).

First the standard sequential solution (i.e., no parallelization) is set up and tested as a reference:

```
R> x <- matrix(rnorm(100 * 10000), 100, 10000)
R> y <- gl(2, 50)
R> cv <- resample("crossvalidation", y, nrepeat = 4, nfold = 8)
R> proc <- modeling_procedure("randomForest", param = list(ntree = 8000))
R> system.time(result_seq <- evaluate(procedure = proc, x = x, y = y,
+    resample = cv))
```

This yielded the following results where the `user` element denotes the CPU time in seconds used by the R session, `system` denotes the CPU time used by the operating system on behalf of the R session (e.g., writing files, forking processes, looking at the system clock etc.), and `elapsed` denotes the real world time used to complete the work.

```
    user   system  elapsed
5878.389    0.170 5874.807
```

Next, the standard parallel implementation is set up and tested:

```
R> system.time(result_par1 <- evaluate(procedure = proc, x = x, y = y,
+    resample = cv, .cores = 16))
```

By spawning multiple R processes elapsed time is greatly reduced at the expense of a slight increase in user and system times.

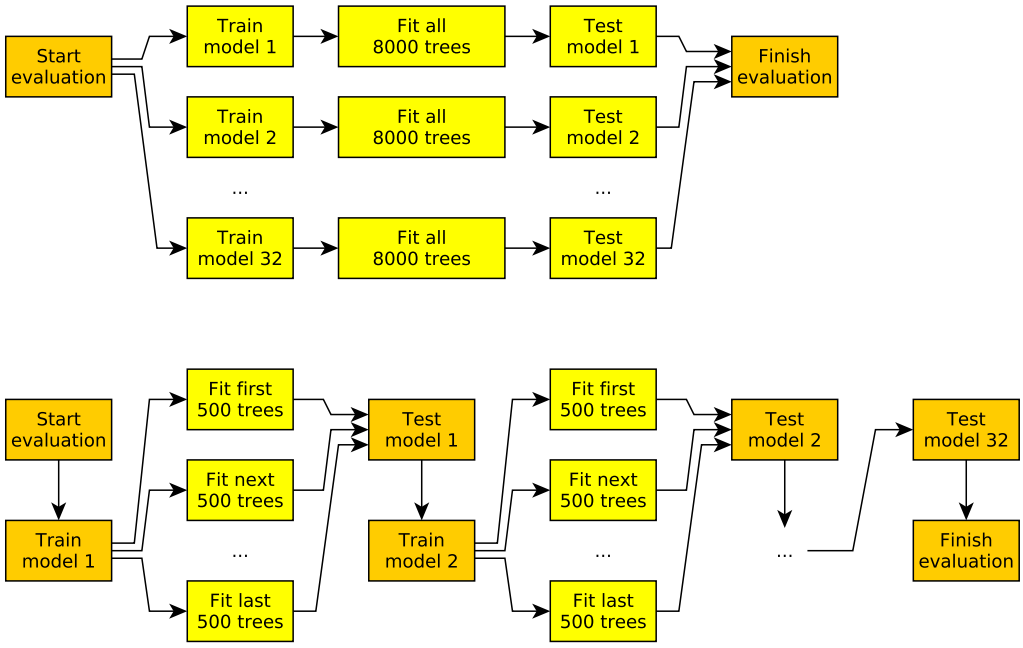

Figure 4: Two alternatives for parallelizing the same analysis presented in Section 3.1. Serial steps are orange and parallel steps are yellow. The task involves fitting and testing 32 random forest models containing 8000 decision trees each (one forest per fold of a 4×8 CV scheme). Top: The first implementation fits all 32 models in parallel, and fits the 8000 trees of each sequentially. Bottom: The second implementation trains the models sequentially, but splits up the 8000 trees on the CPU cores, producing more but smaller subtasks.

```
    user   system  elapsed
6216.822    1.624  524.425
```

Finally, the alternative parallel implementation is set up and tested by replacing the fitting function of the sequential process. The variable `ntree` below is a vector holding the number of trees to be fitted by each core:

```
R> library("parallel")
R> options(mc.cores = 16)
R> par_proc <- proc
R> par_proc$fit_fun <- function(..., ntree) {
+    nc <- getOption("mc.cores")
+    ntree <- table(findInterval(1:ntree - 1, ntree / nc * 1:nc))
+    forests <- mclapply(ntree, function(nt) randomForest(..., ntree = nt))
+    do.call(combine, forests)
+  }
R> system.time(result_par2 <- evaluate(procedure = par_proc, x = x, y = y,
+    resample = cv))
```

This time the elapsed time is somewhat increased and the user and system times are increased

further, since the alternative implementation produces many more parallelized subtasks with more overhead than the standard parallel implementation.

```
    user   system  elapsed
6618.851   33.675  617.991
```

On the other hand, the alternative implementation required much less memory since all sub-processes can work on the same pair of training and test sets, whereas the standard parallel implementation needs to keep 16 unique pairs in memory simultaneously. When working with large data sets this can be a trade-off well worth making.

3.2. High dimensional survival modeling

The next example is based on a publicly available gene expression data set with breast cancer tumor samples (Miller *et al.* 2005). The following analysis was not used in the original paper, but was invented *ad hoc* to demonstrate the **emil** framework. The data set contained expression levels of 18822 genes measured with 44928 probes across 251 samples. The response modeled was the time to relapse. Cox regression is commonly used for survival analysis problems, but since this data set has far more features than observations, and Cox regression lacks feature selection or regularization, the Cox regression algorithm will not be able to find a unique solution. To remedy this, PCA was used to pre-process the data set to compress it to a manageable number of features. However, since not all patients received the same treatment an additional treatment feature must also be included for stratification, but should of course not be part of the PCA.

First the data set is loaded and prepared for the analysis. The package **breastCancerUPP** (Schroeder, Haibe-Kains, Culhane, Sotiriou, Bontempi, and Quackenbush 2011) is used to load the data set `upp`, which is attached to the workspace by calling `data("upp", package = "breastCancerUPP")`. The same object contains both experimental and clinical/phenotypic data, which are accessed using the **Biobase** functions `exprs` and `pData`.

```
R> library("Biobase")
R> data("upp", package = "breastCancerUPP")
R> x <- data.frame(treatment = pData(upp)$treatment, t(exprs(upp)))
R> y <- with(pData(upp), Surv(t.rfs, e.rfs))
```

Next, the custom functions for pre-processing and modeling are defined:

```
R> pre_cox_pca <- function(data) {
+    pca <- prcomp(data$fit$x[-1])
+    data$fit$x <- data.frame(treatment = data$fit$x$treatment, pca$x)
+    data$test$x <- data.frame(treatment = data$test$x$treatment,
+      predict(pca, data$test$x[-1]))
+    data
+  }
R> pca_cox <- modeling_procedure(
+    method = "pca-cox",
+    fit_fun = function(x, y, nfeat) {
+      terms <- c("treatment", sprintf("PC%i", seq_len(nfeat)))
```

```
+      formula <- as.formula(sprintf("y ~ %s",
+        paste(terms, collapse = " + ")))
+      coxph(formula, x)
+    }, predict_fun = predict_coxph,
+    param = list(nfeat = c(0, 1, 2, 3, 5, 9, 15)))
```

Finally, the procedure is carried out and evaluated:

```
R> ho <- resample("holdout", y, nfold = 10, fraction = 1/4,
+    subset = complete.cases(x))
R> result <- evaluate(procedure = pca_cox, x = x, y = y, resample = ho,
+    pre_process = list(pre_split, pre_cox_pca))
```

3.3. Implementation of a custom ensemble classifier

This application example serves to illustrate the reusability of **emil**'s components by implementing a custom ensemble classification method. The fitting function is defined taking both a data set and a list of modeling procedures to fit models from. Using bootstrapping (Efron 1979), different random subsets of the training data are defined for each procedure, and each pair of data subset (`fold`) and procedure is analyzed together using the `Map` function. The `try` statement below is needed to safe-guard against failing model training, which might occasionally occur depending on what procedures are given to the ensemble.

```
R> fit_ensemble <- function(x, y, procedure_list) {
+    samples <- resample("bootstrap", y, nfold = length(procedure_list))
+    Map(function(procedure, fold) {
+      try(fit(procedure, x[index_fit(fold), ], y[index_fit(fold)]),
+        silent = TRUE)
+    }, procedure_list, samples)
+  }
```

The function for using a fitted ensemble to make predictions consists of several steps, explained in detail below.

```
R> library("tidyr")
R> library("dplyr")
R> predict_ensemble <- function(object, x) {
+    prediction <- lapply(object, function(model) {
+      if (inherits(model, "model")) {
+        data.frame(id = seq_len(nrow(x)),
+          prediction = predict(model, x)$prediction)
+      } else {
+        NULL
+      }
+    })
+    vote <- do.call(rbind, prediction)
+    vote <- count(vote, id, prediction)
+    vote <- spread(vote, prediction, n, fill = 0)
```

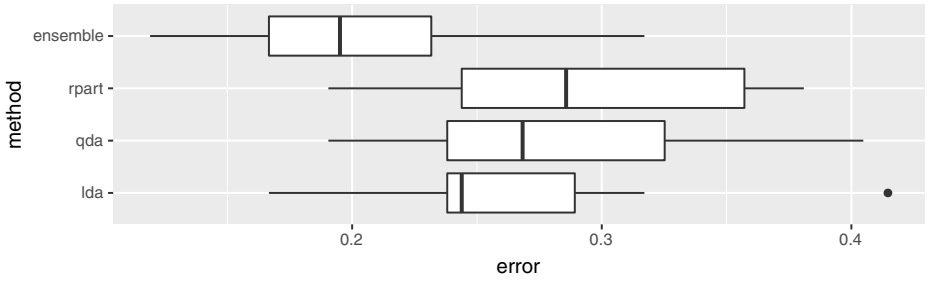

Figure 5: Estimated performance of the ensemble method presented in Section 3.3 and the three types of methods incorporated into it. The x-axis shows estimated error rate.

```
+    n_model <- sum(sapply(object, inherits, "model"))
+    return(list(
+      prediction = factor(apply(vote[-2], 1, which.max),
+        levels = 2:ncol(vote) - 1, labels = colnames(vote)[-1]),
+      vote = as.data.frame(vote[-1] / n_model)
+    ))
+  }
```

The first step is to let all the classifiers of the ensemble (`object`) make individual predictions for all observations in the test set (`x`). For each classifier the `if-inherits` statement checks whether it was trained successfully or caught by the `try` function. Class predictions are returned if it was successfully trained and `NULL` is returned if not.

The second step is to aggregate all the classifiers' predictions into votes for the different classes. This is done by putting all predictions into a single large data frame, using `do.call("rbind", prediction)`, and then pass it on to `count` for summarizing, and `spread` for reshaping.

The final step is to return the class with the most votes for each observation (the element `prediction` in the returned list). The voting statistics are also returned in case the user would be interested.

Using the above functions an ensemble classifier is defined containing 100 linear discriminants (`lda`), 100 quadratic discriminants (`qda`, Venables and Ripley 2002), and 100 decision trees (`rpart`, Therneau, Atkinson, and Ripley 2015).

```
R> ensemble <- modeling_procedure(method = "ensemble",
+    parameter = list(procedure = list(rep(c("lda", "qda", "rpart"),
+      each = 100))))
```

The performances of the ensemble classifiers built are then estimated using cross-validation on a data set containing sonar frequency profiles (60 features) of 208 objects that were either rocks or metal cylinders, first published by Gorman and Sejnowski (1988). The data set was later published in the UCI Machine Learning Repository (Newman, Hettich, Blake, and Merz 1998) and on CRAN in the package **mlbench** (Leisch and Dimitriadou 2010). The performance of the methods incorporated into the ensemble was finally estimated and compared to that of the ensemble (Figure 5).

```
R> data("Sonar", package = "mlbench")
```

```
R> cv <- resample("crossvalidation", Sonar$Class, nrepeat = 3, nfold = 5)
R> comparison <- evaluate(procedure = list("lda", "qda", "rpart", ensemble),
+    x = Sonar, y = "Class", resample = cv)
R> perf <- get_performance(comparison, format = "long")
R> ggplot(perf, aes(x = method, y = error)) + geom_boxplot() + coord_flip()
```

4. Comparison to similar software

There are a number of tools available for automating predictive modeling, from graphical workbenches such as SAS **Enterprise Miner** (SAS Institute Inc. 2013), the Konstanz Information Miner (**KNIME**, Berthold *et al.* 2008) and **RapidMiner** (**RapidMiner** Inc. 2013) to packages for different programming languages such as **caret** (Kuhn 2008; Kuhn *et al.* 2018) for R, **scikit-learn** (Pedregosa *et al.* 2011) for Python (Van Rossum *et al.* 2011), and **PRTools** (Duin *et al.* 2007) for MATLAB (The MathWorks Inc. 2017). Packages **emil** and **caret** partially overlap in functionality, and have actually been developed independently in parallel for a few years, but both also have some unique features. This section serves to highlight the differences between them.

4.1. Performance evaluation of complete procedures

The most notable difference between packages **emil** and **caret** is that package **emil** focuses on evaluating whole computational procedures whereas package **caret** is only focused on training and tuning them. During training package **caret** does use resampling based performance evaluation to tune meta-parameters (the inner loop of Algorithm 1), but the package does not contain any function for unbiased performance estimation of the entire procedure. It might be tempting to use the performance estimates obtained during tuning as a substitute, but since the tuning procedure always selects the best performing parameter values there is a risk that the corresponding performance estimate is positively biased (Varma and Simon 2006; Lawless and Yuan 2010). Users of the **caret** package may of course implement the outer loop of Algorithm 1 themselves to obtain more reliable performance estimates, but this will lead to suboptimal memory usage given the way package **caret** is implemented.

The memory efficiency and computation time of packages **emil** (version 2.2.8) and **caret** (version 6.0-79) was studied on five tasks involving commonly used predictive modeling techniques. Each task was performed in five replicates and monitored using the **Syrupy** tool (Sukumaran 2015) that repeatedly polled the Unix function `ps` once per second for the given R processes. Figure 6 shows the computation time and average memory usage over all the replicates. The "ALL" tasks used the data set presented in Section 3.2 with the goal of predicting the samples' estrogen-receptor status from their gene expression profiles, while the "Sonar" task used the data presented in Section 3.3. Task specific details were as follows:

- The "ALL-kNN-forest" task consisted of performing kNN-imputation in combination with random forest classification (Breiman 2001) with a pre-selected meta-parameter value, including evaluation by 5-fold cross-validation repeated 3 times.
- The "ALL-PAM" task consisted of performing nearest shrunken centroids classification (Tibshirani, Hastie, Narasimhan, and Chu 2002, NSC) using the **pamr** package (Hastie,

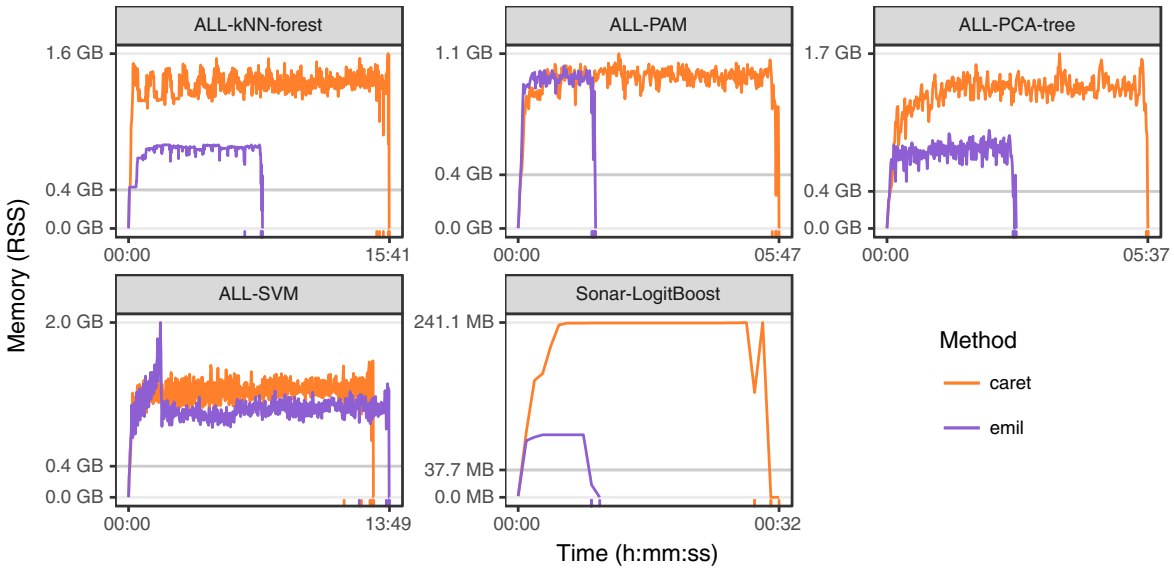

Figure 6: Comparison of memory usage and computation time between packages **emil** and **caret** on the 5 benchmarking tasks described in Section 4.1. All curves are averages over five replicates performed sequentially on a high performance machine. Each replicate was run in a separate R process. Garbage collection was performed prior to the fitting of each model and a 3 second waiting period was added to the end of each job to ensure the memory profiler had time to get a final reading. The gray horizontal lines mark the memory requirement of only loading the data set. The small vertical tick marks on the x-axis mark the completion time of individual replicates. The y-axis shows the amount of non-swapped RAM used by the R processes, known as resident set size (RSS).

Tibshirani, Narasimhan, and Chu 2014), including tuning the meta-parameter `threshold` and evaluation by 5-fold cross-validation repeated 3 times.

- The "ALL-PCA-tree" task consisted of performing PCA pre-processing, selecting the first 20 components, and training decision trees (Breiman, Friedman, Olshen, and Stone 1984) using the **rpart** package (Therneau *et al.* 2015), including tuning the meta-parameter `maxdepth` and evaluation by 3-fold cross-validation repeated twice.

- The "ALL-SVM" task consisted of performing support vector machine (SVM) classification with polynomial kernels using the **kernlab** package (Karatzoglou, Smola, Hornik, and Zeileis 2004), including tuning the meta-parameter `degree` and evaluation by 3-fold cross-validation repeated twice.

- The "Sonar-LogitBoost" task consisted of training a single logit boost model (Dettling and Bühlmann 2003) using the **caTools** package (Tuszynski 2014), including tuning of its meta-parameter `nIter` and evaluation by external testing on a single test set. This task was taken from package **caret**'s online manual (Kuhn 2015).

Package **emil** required less memory than package **caret** on all tasks except "ALL-PAM", mainly because package **emil** avoids creating some unnecessary in-memory copies of the data set.

When analyzing large data sets (on the order of several gigabytes) it is crucially important to avoid unnecessary copies of the data set, as it may considerably slow down the analysis or sometimes inhibit it entirely. The issue is less serious on the relatively small data sets analyzed in the benchmarking study but the existence of the effect is still apparent, especially when using pre-processing steps such as kNN imputation or PCA. Package **emil**'s core functions `fit`, `tune`, and `evaluate` are designed to never modify the data set in any way, but to leave that to the user-defined pre-processing functions. Since the pre-processing functions operate in a sequential manner on the training and test sets exclusively package **emil** always leaves the original copy of the data set untouched. This holds true even if a method incorporated into the **emil** framework requires data in a non-standard form, such as the **pamr** method for NSC classification. Data should be supplied to the `pamr.train` function as a list of two elements `x` and `y` where `x` is a matrix with observations as rows and `y` is a matching response vector. This can easily be accomplished by using a custom pre-processing function[3].

```
R> pre_pamr <- function(data) {
+    data$fit$x <- list(x = t(data$fit$x), y = data$fit$y)
+    data$fit$y <- NULL
+    data$test$x <- t(data$test$x)
+    data
+  }
R> y <- factor(findInterval(prostate$lpsa, quantile(prostate$lpsa, 1:2/3)),
+    labels = c("low", "intermediate", "high"))
R> cv <- resample("crossvalidation", y, nfold = 5, nrep = 1)
R> result <- evaluate(procedure = "pamr", x = prostate[1:8], y = y,
+    resample = cv, pre_process = list(pre_split, pre_pamr))
```

If the data set instead had been reshaped within the training function, an additional in-memory copy would have been created.

Some of the advantages of both packages can be combined through calling the **emil** function `evaluate` with a modeling procedure with `method = "caret"`. The fitting function will then pass on the data and parameters to the **caret** function `train` for tuning and fitting. This may add some additional memory overhead compared to running package **emil** on its own, but provides a quick way of adding the outer loop of Algorithm 1 to any existing code written for package **caret**. The example below shows how to define a modeling procedure using package **caret** to train random forest models[4].

```
R> modeling_procedure(method = "caret", parameter = list(
+    method = "rf", ntree = 10000,
+    trControl = list(trainControl(method = "repeatedcv", number = 5,
+      repeats = 5, returnData = FALSE, allowParallel = TRUE,
+      verboseIter = TRUE)),
+    tuneGrid = list(data.frame(mtry = c(10, 50, 200)))))
```

[3]The `pre_pamr` function actually used in the package also contains additional steps but these have been omitted for clarity.

[4]The object returned from `trainControl` is a list in itself requiring it to be wrapped in an outer list to be interpreted as a single entity. If not, all the elements of the `trainControl` object would be interpreted as separate values to try for the `trControl` parameter, which does not work. The same principle also applies to `tuneGrid`.

4.2. Other functionality

In addition to the differences highlighted in the previous section, the **emil** and **caret** packages both provide some functionality with superior implementation or not present in the other package. The advantages of package **emil** include a more flexible data pre-processing system (Section 2.3), check-pointing and a more developed progress logging system (Section 2.8), and learning curve analyzes (see `?learning_curve`, Duda, Hart, and Stork 2000). The advantages of package **caret** include a larger number of methods ready to be used by the user, developed parallelization capabilities, and adaptive resampling, which is a technique for reducing the meta-parameter search space as the resampling based testing progresses (Maron and Moore 1997; Shen, Welch, and Hughes-Oliver 2011).

5. Concluding remarks

When using machine learning-based tools for prediction modeling problems, rigorous performance evaluation is critical in order to obtained unbiased information about the true performance of the prediction models built (Varma and Simon 2006). Resampling based methods such as cross-validation and repeated holdout are useful tools for this but can be misleading unless the test sets by these procedures are excluded from the entire modeling procedure and only used for the final performance evaluation step. The **emil** package strictly adheres to this principle and provides a versatile toolbox for implementing and evaluating customized modeling procedures for predictive modeling problems in a statistically rigorous way.

Since the applications of predictive modeling are incredibly diverse, the **emil** framework is designed to be as general and flexible as possible while at the same time staying transparent and light weight. In this paper and the accompanying benchmarking study we have shown several examples of how the components can be combined and reused in different settings to implement alternative routines for pre-processing, parallelization, and parameter tuning, as well as creating custom ensemble classifiers while at the same time staying computationally efficient. This is possible thanks to the API like design of the framework.

While great care has been taken to make the **emil** package as resource efficient as possible, it should be noted that the memory requirement and computation time of a modeling procedure is heavily dependent on which methods are incorporated into it. The gains by using the **emil** package will differ from case to case, but a substantial reduction in development time can nevertheless be achieved by not having to re-implement the general analysis framework.

Potential drawbacks of the **emil** framework is that since the user is empowered to re-implement many features it might require a higher level of programming proficiency than similar packages, for example when setting up cluster parallelization. There are also points where memory efficiency could be improved further, such as delaying the pre-processing of the test sets until the training is completed, but since the sum of the training and test sets never exceeds the size of the entire data set (unless oversampling is used) this would only provide a minor improvement.

Acknowledgments

This work was supported by grants from the Swedish Foundation for Strategic Research (RBc08-008) as well as the Swedish Research Council (621-2008-5854). The computations were performed on resources provided by SNIC through the Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). We thank Elias Castegren at the Department of Information Technology, Uppsala University for providing feedback on implementation design and terminology.

CB and MG conceived of the approach and discussed the computational results as they appeared. CB implemented the package and drafted the manuscript. MG supervised the project and wrote the final manuscript together with CB.

References

Akaike H (1973). "Information Theory and an Extension of the Maximum Likelihood Principle." In BN Petrov, F Csaki (eds.), *2nd International Symposium on Information Theory*, pp. 267–281.

Bäcklin CL, Gustafsson MG (2018). ***emil**: Evaluation of Modeling without Information Leakage*. R package version 2.2.10, URL https://CRAN.R-project.org/package=emil.

Berthold MR, Cebron N, Dill F, Gabriel TR, Kötter T, Meinl T, Ohl P, Sieb C, Thiel K, Wiswedel B (2008). "**KNIME**: The Konstanz Information Miner." In C Preisach, H Burkhardt, L Schmidt-Thieme, R Decker (eds.), *Data Analysis, Machine Learning and Applications*, Studies in Classification, Data Analysis, and Knowledge Organization. Springer-Verlag.

Breiman L (2001). "Random Forests." *Machine Learning*, **45**(1), 5–32. doi:10.1023/a:1010933404324.

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984). *Classification and Regression Trees*. 1st edition. Wadsworth.

Cox DR (1972). "Regression Models and Life-Tables." *Journal of the Royal Statistical Society B*, **34**(2), 187–220.

Dettling M, Bühlmann P (2003). "Boosting for Tumor Classification with Gene Expression Data." *Bioinformatics*, **19**(9), 1061–1069. doi:10.1093/bioinformatics/btf867.

Duda RO, Hart PE, Stork DG (2000). *Pattern Classification*. 2nd edition. John Wiley & Sons.

Duin RPW, Juszczak P, Paclík P, Pękalska E, De Ridder D, Tax DMJ, Verzakov S (2007). "**PRTools4.1**, A MATLAB Toolbox for Pattern Recognition." URL http://prtools.org.

Efron B (1979). "Bootstrap Methods: Another Look at the Jackknife." *The Annals of Statistics*, **7**(1), 1–26. doi:10.1214/aos/1176344552.

Friedman J, Hastie T, Tibshirani R (2010). "Regularization Paths for Generalized Linear Models via Coordinate Descent." *Journal of Statistical Software*, **33**(1), 1–22. doi:10.18637/jss.v033.i01.

Gentleman RC, Carey VJ, Bates DM, Bolstad B, Dettling M, Dudoit S, Ellis B, Gautier L, Ge Y, Gentry J, Hornik K, Hothorn T, Huber W, Iacus S, Irizarry R, Leisch F, Li C, Maechler M, Rossini AJ, Sawitzki G, Smith C, Smyth G, Tierney L, Yang JYH, Zhang J (2004). "**Bioconductor**: Open Software Development for Computational Biology and Bioinformatics." *Genome Biology*, **5**, R80. doi:10.1186/gb-2004-5-10-r80.

Gorman RP, Sejnowski TJ (1988). "Analysis of Hidden Units in a Layered Network Trained to Classify Sonar Targets." *Neural Networks*, **1**(1), 75–89. doi:10.1016/0893-6080(88)90023-8.

Halvorsen K (2015). ***ElemStatLearn**: Data Sets, Functions and Examples from the Book: "The Elements of Statistical Learning, Data Mining, Inference, and Prediction" by Trevor Hastie, Robert Tibshirani and Jerome Friedman*. R package version 2015.6.26, URL https://CRAN.R-project.org/package=ElemStatLearn.

Hastie T, Tibshirani R, Friedman J (2001). *The Elements of Statistical Learning*. 1st edition. Springer-Verlag. doi:10.1007/978-0-387-21606-5.

Hastie T, Tibshirani R, Narasimhan B, Chu G (2014). ***pamr**: PAM: Prediction Analysis for Microarrays*. R package version 1.55, URL https://CRAN.R-project.org/package=pamr.

Huber W, Carey VJ, Gentleman R, Anders S, Carlson M, Carvalho BS, Bravo HC, Davis S, Gatto L, Girke T, Gottardo R, Hahne F, Hansen KD, Irizarry RA, Lawrence M, Love MI, MacDonald J, Obenchain V, Oleś AK, Pagès H, Reyes A, Shannon P, Smyth GK, Tenenbaum D, Waldron L, Morgan M (2015). "Orchestrating High-Throughput Genomic Analysis with **Bioconductor**." *Nature Methods*, **12**(2), 115–121. doi:10.1038/nmeth.3252.

Karatzoglou A, Smola A, Hornik K, Zeileis A (2004). "**kernlab** – An S4 Package for Kernel Methods in R." *Journal of Statistical Software*, **11**(9), 1–20. doi:10.18637/jss.v011.i09.

Kuhn M (2008). "Building Predictive Models in R Using the **caret** Package." *Journal of Statistical Software*, **28**(5), 1–26. doi:10.18637/jss.v028.i05.

Kuhn M (2015). "The **caret** Package: Using Your Own Model in `train`." http://topepo.github.io/caret/custom_models.html. Accessed 2015-06-18.

Kuhn M, Wing J, Weston S, Williams A, Keefer C, Engelhardt A, Cooper T (2018). ***caret**: Classification and Regression Training*. R package version 6.0-79, URL https://CRAN.R-project.org/package=caret.

Lawless JF, Yuan Y (2010). "Estimation of Prediction Error for Survival Models." *Statistics in Medicine*, **29**(2), 262–272. doi:10.1002/sim.3758.

Leisch F, Dimitriadou E (2010). ***mlbench**: Machine Learning Benchmark Problems*. R package version 2.1-1, URL https://CRAN.R-project.org/package=mlbench.

Maron O, Moore AW (1997). "The Racing Algorithm: Model Selection for Lazy Learners." In DW Aha (ed.), *Lazy Learning*, pp. 193–225. Springer-Verlag. doi:10.1007/978-94-017-2053-3_8.

Miller LD, Smeds J, George J, Vega VB, Vergara L, Ploner A, Pawitan Y, Hall P, Klaar S, Liu ET, Bergh J (2005). "An Expression Signature for P53 Status in Human Breast Cancer Predicts Mutation Status, Transcriptional Effects, and Patient Survival." *Proceedings of the National Academy of Sciences of the United States of America*, **102**(38), 13550–13555. doi:10.1073/pnas.0506230102.

Milton Bache S, Wickham H (2014). ***magrittr**: A Forward-Pipe Operator for R*. R package version 1.5, URL https://CRAN.R-project.org/package=magrittr.

Newman DJ, Hettich S, Blake CL, Merz CJ (1998). "UCI Repository of Machine Learning Databases." URL https://www.ics.uci.edu/~mlearn/MLRepository.html.

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011). "**scikit-Learn**: Machine Learning in Python." *Journal of Machine Learning Research*, **12**, 2825–2830.

RapidMiner Inc (2013). ***RapidMiner** Studio, Version 6*. Cambridge. URL http://www.rapidminer.com/.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Sarkar D (2008). ***lattice**: Multivariate Data Visualization with R*. Springer-Verlag, New York. URL http://lmdvr.R-Forge.R-project.org/.

SAS Institute Inc (2013). *SAS **Enterprise Miner**, Version 13.1*. URL http://www.sas.com/.

Schroeder M, Haibe-Kains B, Culhane A, Sotiriou C, Bontempi G, Quackenbush J (2011). ***breastCancerUPP**: Gene Expression Dataset Published by Miller et al. [2005] (UPP)*. doi:10.18129/B9.bioc.breastCancerUPP. R package version 1.0.5.

Schwarz G (1978). "Estimating the Dimension of a Model." *The Annals of Statistics*, **6**(2), 461–464. doi:10.1214/aos/1176344136.

Shen H, Welch WJ, Hughes-Oliver JM (2011). "Efficient, Adaptive Cross-Validation for Tuning and Comparing Models, with Application to Drug Discovery." *The Annals of Applied Statistics*, **5**(4), 2668–2687. doi:10.1214/11-aoas491.

Snijders C, Matzat U, Reips UD (2012). ""Big Data": Big Gaps of Knowledge in the Field of Internet." *International Journal of Internet Science*, **1**(7), 1–5.

Stamey TA, Kabalin JN, McNeal JE, Johnstone IM, Freiha F, Redwine EA, Yang N (1989). "Prostate Specific Antigen in the Diagnosis and Treatment of Adenocarcinoma of the Prostate. II. Radical Prostatectomy Treated Patients." *The Journal of Urology*, **141**(5), 1076–1083. doi:10.1016/s0022-5347(17)41175-x.

Sukumaran J (2015). "**Syrupy**: System Resource Usage Profiler." https://github.com/jeetsukumaran/Syrupy. Commit: 2d6e963d03233388afb9cafd5360435e66557dd9.

The MathWorks Inc (2017). *MATLAB – The Language of Technical Computing, Version R2017b*. Natick. URL http://www.mathworks.com/products/matlab/.

Therneau T, Atkinson B, Ripley B (2015). ***rpart**: Recursive Partitioning and Regression Trees*. R package version 4.1-9, URL https://CRAN.R-project.org/package=rpart.

Tibshirani R (1996). "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society B*, **58**(1), 267–288.

Tibshirani R, Hastie T, Narasimhan B, Chu G (2002). "Diagnosis of Multiple Cancer Types by Shrunken Centroids of Gene Expression." *Proceedings of the National Academy of Sciences of the United States of America*, **99**(10), 6567–6572. doi:10.1073/pnas.082099299.

Tuszynski J (2014). ***caTools**: Tools – Moving Window Statistics, GIF, Base64, ROC, AUC, etc.* R package version 1.17.1, URL https://CRAN.R-project.org/package=caTools.

Van Rossum G, *et al.* (2011). *Python Programming Language*. URL http://www.python.org/.

Varma S, Simon R (2006). "Bias in Error Estimation When Using Cross-Validation for Model Selection." *BMC Bioinformatics*, **7**(91). doi:10.1186/1471-2105-7-91.

Venables WN, Ripley BD (2002). *Modern Applied Statistics with S*. 4th edition. Springer-Verlag, New York. URL http://www.stats.ox.ac.uk/pub/MASS4.

Wickham H (2009). ***ggplot2**: Elegant Graphics for Data Analysis*. Springer-Verlag, New York. URL http://ggplot2.org/.

Wickham H (2014). "Tidy Data." *Journal of Statistical Software*, **59**(10), 1–23. doi:10.18637/jss.v059.i10.

Wickham H, François R (2017). ***dplyr**: A Grammar of Data Manipulation*. R package version 0.7.4, URL https://CRAN.R-project.org/package=dplyr.

Wu X, Zhu X, Wu GQ, Ding W (2014). "Data Mining with Big Data." *IEEE Transactions on Knowledge and Data Engineering*, **26**(1), 97–107. doi:10.1109/tkde.2013.109.

Affiliation:

Christofer L. Bäcklin
Uppsala University, Department of Medical Sciences
Cancer Pharmacology and Computational Medicine
Uppsala Academic Hospital
Entrance 61, floor 3
751 85 Uppsala, Sweden
E-mail: emil@christofer.backlin.se

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
July 2018, Volume 85, Issue 13 *Submitted:* 2014-05-11
doi:10.18637/jss.v085.i13 *Accepted:* 2017-01-10