Super Learner for Applied Biostatisticians & Epidemiologists

A Hands-On Tutorial (HTML, PDF & Word)

Miguel Angel Luque-Fernández

2026-01-01

Table of contents

1 Summary and intended use

This tutorial is a practical guide for applied epidemiologists, biostatisticians and clinical researchers who want to use and disseminate **Super Learner (SL)**: an ensemble machine-learning approach that combines several prediction algorithms into a single weighted model. SL's cross-

validated performance is asymptotically at least as good as the best algorithm in the library, and in finite samples it is often better. For personalized medicine, the payoff is concrete: one SL model can produce individual-level risk predictions from heterogeneous clinical data without forcing the analyst to commit to a single functional form in advance (van der Laan et al., 2007; Naimi & Balzer, 2018).

The tutorial is designed to be run end to end in R with Quarto, and it renders to HTML, PDF and Word. Every section is self-contained, seeds are given for reproducibility, and all active code runs in a single pass. By the end you will know how to:

1. Fit a SuperLearner from scratch and interpret its ensemble weights and cross-validated risk;
2. Assemble a library of flexible learners, tune their hyperparameters and add feature screening;
3. Evaluate prediction performance with external cross-validation, AUC and calibration curves;
4. Run the ensemble in parallel and write your own learner and screening wrappers;
5. Apply SL to clinical risk prediction, continuous outcomes and dose-response estimation;
6. Connect SL to targeted machine learning (TMLE) for causal-effect estimation in observational data.

No prior machine-learning background is assumed beyond standard regression. Prerequisite R packages are listed in the installation section. The closing section collects the best practices worth carrying into your own analyses.

Note

How to cite or share this tutorial. License: **CC BY 4.0**. Cite it as: Luque-Fernández MA (2026). *Super Learner for Applied Biostatisticians & Epidemiologists*. Department of Statistics and Operations Research, University of Granada. Reuse, adaptation and translation are welcome with attribution.

2 Introduction

2.1 What is Super Learner?

Super Learner (SL) is a principled ensemble machine learning method that uses **V-fold cross-validation** to:

1. Estimate the performance of multiple candidate learning algorithms;
2. Create an **optimal weighted combination** (an *ensemble*) of those algorithms;
3. Provide **valid cross-validated estimates** of the ensemble's own performance.

The key theoretical result (van der Laan et al., 2007) is that Super Learner performs **asymptotically as well as the best possible combination** of the candidate algorithms, a property called *asymptotic efficiency*.

Note

What this means for epidemiologists. Rather than selecting a single "best" model, SL combines several into a weighted ensemble that can beat any one candidate, which matters when the true data-generating mechanism is unknown.

2.2 Why Super Learner for Epi / Biostatistics?

Challenge	SL solution
Uncertainty about the best model	Automatic weighted ensemble
Overfitting risk with complex models	Cross-validated risk estimation
High-dimensional covariates	Built-in feature screening
Binary, continuous and survival outcomes	Flexible `family` specification
Reproducibility	Seed-controlled, deterministic pipeline
From prediction to causal inference	Foundation for TMLE and AIPW estimators

2.3 How Super Learner works

SL runs in six stages:

1. **Split** the data into V roughly equal folds.
2. For each fold `v = 1, ..., V`:
   - **Train** every candidate algorithm on the other V-1 folds;
   - **Predict** the outcome for the held-out fold `v`.
3. **Stack** the cross-validated predictions of all candidate algorithms into one matrix.
4. **Find optimal weights** with a meta-learner that minimises cross-validated risk (e.g., non-negative least squares or AUC).
5. **Refit** every candidate algorithm on the full data set.
6. **Predict** new observations with the weighted combination of refits.

2.4 Key references

- van der Laan MJ, Polley EC, Hubbard AE (2007). *Super Learner.* Stat Appl Genet Mol Biol 6(1).
- Polley EC, van der Laan MJ (2010). *Super Learner Prediction.* In *Targeted Learning*, Springer.
- Naimi AI, Balzer LB (2018). *Stacked Generalization: An Introduction to Super Learning.* Eur J Epidemiol 33(5):459-464.
- Rose S (2013). *Mortality risk score prediction in an elderly population using machine learning.* Am J Epidemiol.
- Phillips RV, van der Laan MJ, Lee H, Gruber S (2019). *Practical considerations for Super Learner.* Epidemiology.

---

3 Installation and setup

3.1 Install the packages

> **Tip**
>
> The code below is **not executed** during rendering (`eval: false`); run it once in your own session. If you already have most packages, you can disable individual `install.packages()` lines.

Why and how. `install.packages()` downloads each package from CRAN and copies it into your R library once per computer. The core piece is `SuperLearner` itself; the rest provide the candidate algorithms the tutorial uses (`glmnet`, `randomForest`, `ranger`, `xgboost`, `earth`, `gam`,

gbm), the plotting tools (ggplot2, ROCR, pROC), and small helpers (caret, data.table, rmutil, vcd). Run the whole block once, then continue to loading.

```
# Core package
install.packages("SuperLearner")

# Development version (optional, recent additions)
# devtools::install_github("ecpolley/SuperLearner")

# Packages used throughout this tutorial
install.packages(c(
  "glmnet", "randomForest", "ranger", "xgboost", "earth", "gam",
  "gbm", "rpart", "nnet", "e1071", "ipred", "ada",
  "ggplot2", "ROCR", "pROC", "caret", "data.table", "mlbench",
  "RhpcBLASctl", "rmutil", "vcd", "nnls", "Matrix"
))
```

What you see. The first run shows download progress and then a short "package … was built under R version …" note for each install. If a package is already up to date, R simply reinstalls it. When the block ends without an error, every dependency is in place.

3.2 Load the packages

Why and how. `library()` attaches each package to the session so its functions can be called directly. We load `SuperLearner` first because every later fit and summary builds on it; the remaining packages contribute wrappers, plotting and helpers.

```
library(SuperLearner)
library(ggplot2)
library(caret)
library(data.table)
library(rmutil)
library(vcd)
```

What you see. Because warnings and messages are suppressed in this document, the loads pass silently. In your own console you will instead see the version banners and occasional "package … was built under R version …" lines; both are harmless as long as no Error appears.

3.3 Explore the available learners

SuperLearner ships with many built-in *wrappers*. Running `listWrappers()` is the fastest way to see what your installed version can call, which is useful when you join a project with an existing library or want to confirm that a specific algorithm is available.

```
listWrappers()
```

```
 [1] "SL.bartMachine"      "SL.bayesglm"         "SL.biglasso"
 [4] "SL.caret"            "SL.caret.rpart"      "SL.cforest"
 [7] "SL.earth"            "SL.gam"              "SL.gbm"
[10] "SL.glm"              "SL.glm.interaction"  "SL.glmnet"
[13] "SL.ipredbagg"        "SL.kernelKnn"        "SL.knn"
[16] "SL.ksvm"             "SL.lda"              "SL.leekasso"
[19] "SL.lm"               "SL.loess"            "SL.logreg"
[22] "SL.mean"             "SL.nnet"             "SL.nnls"
[25] "SL.polymars"         "SL.qda"              "SL.randomForest"
```

```
[28] "SL.ranger"           "SL.ridge"            "SL.rpart"
[31] "SL.rpartPrune"       "SL.speedglm"         "SL.speedlm"
[34] "SL.step"             "SL.step.forward"     "SL.step.interaction"
[37] "SL.stepAIC"          "SL.svm"              "SL.template"
[40] "SL.xgboost"
```

```
[1] "All"
[1] "screen.corP"            "screen.corRank"         "screen.glmnet"
[4] "screen.randomForest"    "screen.SIS"             "screen.template"
[7] "screen.ttest"           "write.screen.template"
```

> **Note**
>
> The output has three blocks:
> - **`SL.*`** : prediction algorithms (regression / classification);
> - **`screen.*`** : feature-selection / screening methods;
> - **`method.*`** : meta-learning optimisers that combine the predictions.
>
> You can open the source of any wrapper to see its defaults, e.g. `SL.glmnet` or `screen.corP`.

What you see. The catalogue is broad, from the plain `SL.glm` you already know up to flexible `SL.xgboost` and `SL.earth`. Everything printed under `SL.*` is a candidate you can put in a library, and the `screen.*` list is what you draw on when the covariate space gets large. Your own libraries will be assembled from these same names.

```
SL.glmnet
SL.randomForest
SL.xgboost
screen.corP
```

What you see. Typing a wrapper's name without parentheses prints its source. `SL.glmnet` shows the defaults you can override, for instance the internal cross-validation that picks the penalty. Reading one wrapper this way is the quickest way to learn what an algorithm does before you commit it to a library.

---

4 A simulated epidemiological example

4.1 Generate data with a known structure

We simulate a realistic epidemiological data set with a binary outcome (e.g., disease), and covariates that interact and are non-linearly related to the outcome. Because we know the truth, we can evaluate how well SL recovers it.

Why and how. The code builds a data frame that looks like what you would export from a hospital registry: five predictor columns and one outcome column. The predictors are drawn from uniform or binomial distributions, the outcome is generated from a logistic formula that mixes main effects, an interaction (`W5 * W1`) and a smooth non-linear term (`sin(W4 * pi)`), and `rbinom()` adds sampling noise. You will not reproduce this exact setup on your own data; what matters is the pattern, that the response is binary and that the true mechanism is more complicated than any single linear model. The `set.seed()` calls at the top of each block make every draw reproducible, so your numbers will match the ones printed here.

```
set.seed(2743)
n <- 500
```

```r
# Covariates (standardised to unit scale)
W1 <- runif(n, 0.5, 1)       # e.g., standardised age
W2 <- runif(n, 0, 1)         # e.g., standardised BMI
W3 <- runif(n, 0.25, 0.75)   # e.g., standardised blood pressure
W4 <- runif(n, 0, 1)         # e.g., standardised biomarker

# A derived comorbidity indicator
W5 <- rbinom(n, 1, 1 / (1 + exp(1.5 * W2 - W3)))

# True data-generating mechanism: interactions + non-linearity
logit_p <- -0.2 * W5 - 2 * W1 + 4 * W5 * W1 - 1.5 * W2 + sin(W4 * pi)
Y <- rbinom(n, 1, 1 / (1 + exp(-logit_p)))

dat <- data.frame(W1, W2, W3, W4, W5, Y)

cat("Prevalence of the outcome:", round(mean(Y), 3), "\n")
```

```
Prevalence of the outcome: 0.44
```

```r
cat("Sample size:", nrow(dat), "\n")
```

```
Sample size: 500
```

```r
cat("Number of predictors:", ncol(dat) - 1, "\n")
```

```
Number of predictors: 5
```

```r
summary(dat)
```

```
       W1              W2                W3              W4        
 Min.   :0.500   Min.   :0.00056   Min.   :0.252   Min.   :0.00388  
 1st Qu.:0.615   1st Qu.:0.24025   1st Qu.:0.372   1st Qu.:0.24555  
 Median :0.728   Median :0.47032   Median :0.502   Median :0.49973  
 Mean   :0.740   Mean   :0.47600   Mean   :0.495   Mean   :0.50607  
 3rd Qu.:0.864   3rd Qu.:0.72776   3rd Qu.:0.618   3rd Qu.:0.76426  
 Max.   :0.999   Max.   :0.99765   Max.   :0.750   Max.   :0.99940  
       W5               Y         
 Min.   :0.000   Min.   :0.00  
 1st Qu.:0.000   1st Qu.:0.00  
 Median :0.000   Median :0.00  
 Mean   :0.436   Mean   :0.44  
 3rd Qu.:1.000   3rd Qu.:1.00  
 Max.   :1.000   Max.   :1.00  
```

```r
ggplot(dat, aes(x = factor(Y))) +
  geom_bar(fill = c("#E74C3C", "#2ECC71"), alpha = 0.85) +
  scale_x_discrete(labels = c("0 = healthy", "1 = disease")) +
  labs(x = "", y = "Count") +
  theme_minimal(base_size = 14)
```

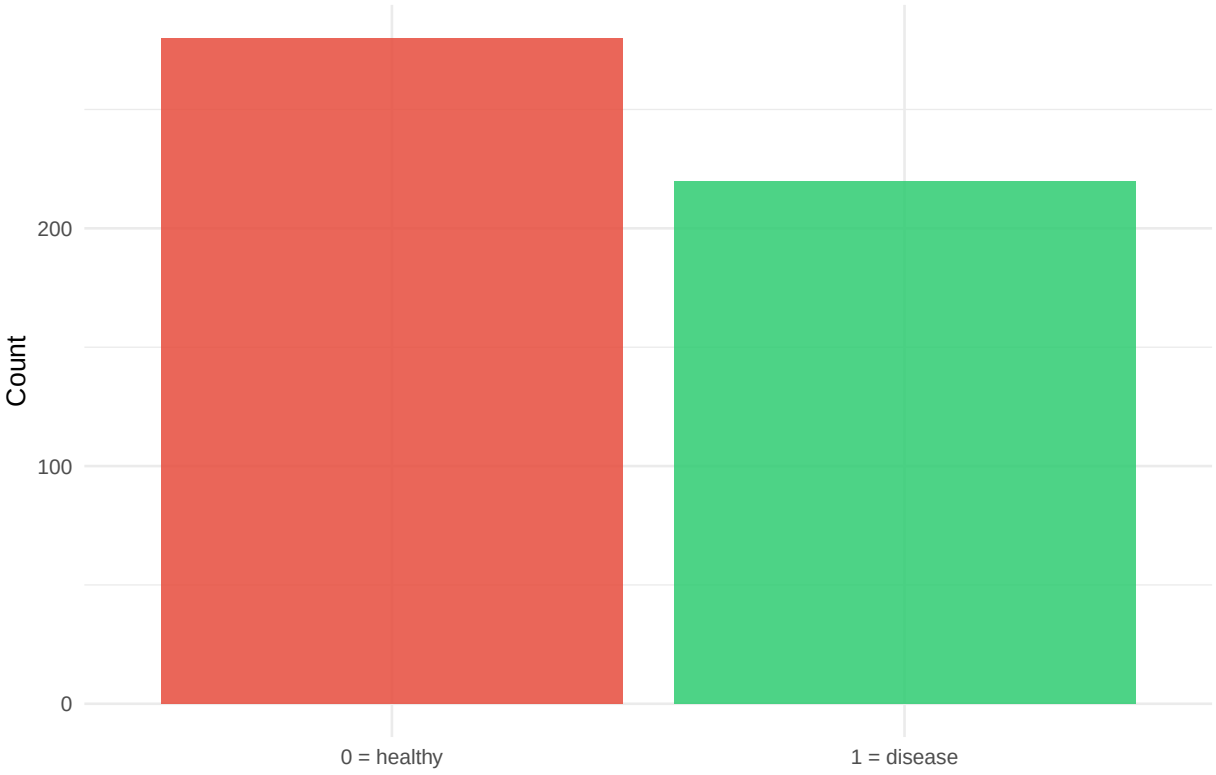


Figure 1: Distribution of the simulated binary outcome.

What you see. About 44% of the 500 simulated patients have the outcome, and the summary confirms that every predictor spans roughly the unit interval we asked for. The outcome is balanced enough that a chance-level classifier is easy to beat, yet the pattern in `logit_p` is rich enough that simple linear models will leave signal on the table. That is exactly the setting where an ensemble has room to help.

---

5 Basic Super Learner

5.1 First fit: a single algorithm

Start with a single algorithm, a penalised logistic regression (*LASSO / elastic net*):

Why and how. `SuperLearner()` is the workhorse call. You give it the outcome `Y`, the predictor matrix `X`, the outcome type via `family`, and one or more learner names in `SL.library`. Internally it splits the training data into V folds, refits every learner on the other folds to estimate performance, stacks the out-of-fold predictions, and then refits the chosen combination on the full data. With one algorithm in the library the ensemble collapses to that algorithm, which makes this first fit a clean check that you know how to call the function before adding complexity. `set.seed(123)` fixes the train/test split so later sections compare like with like.

```
set.seed(123)

# Train / test split (80 % / 20 %)
train_idx <- sample(1:n, 400)
X_train   <- dat[train_idx, 1:5]
Y_train   <- dat[train_idx, 6]
X_test    <- dat[-train_idx, 1:5]
```

```r
Y_test    <- dat[-train_idx, 6]

sl_lasso <- SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(),
  SL.library = "SL.glmnet"
)
sl_lasso
```

```
Call:
SuperLearner(Y = Y_train, X = X_train, family = binomial(), SL.library = "SL.glmnet")

                Risk Coef
SL.glmnet_All 0.1612    1
```

```r
# Cross-validated risk of each algorithm (here: just LASSO)
sl_lasso$cvRisk
```

```
SL.glmnet_All
       0.1612
```

```r
# Coefficients / weights (must sum to 1; here just LASSO)
sl_lasso$coef
```

```
SL.glmnet_All
            1
```

```r
# The raw glmnet fit object (compact view: the CV loss and the
# coefficients at the lambda that minimises cross-validated error)
fit_lasso <- sl_lasso$fitLibrary$SL.glmnet_All$object
fit_lasso$name
```

```
           deviance
"Binomial Deviance"
```

```r
round(as.matrix(coef(fit_lasso, s = "lambda.min")), 3)
```

```
             lambda.min
(Intercept)     -1.478
W1               0.183
W2              -0.778
W3               0.004
W4               0.652
W5               2.545
```

What you see. With one algorithm in the library the printout is just a single row: `SL.glmnet_All` appears with a cross-validated risk of about 0.161 and a Coef of exactly 1, because there is no one else to split weight with. The extra lines confirm this from inside the object, and the coefficient vector at the minimum cross-validation lambda shows which predictors LASSO actually retained. You now have, in two short blocks, the performance baseline that every larger library will be compared against.

> **Note**
>
> **What is "risk"?** Risk is the expected prediction error estimated by cross-validation. SuperLearner **minimises** the estimated risk (mean squared error for continuous outcomes; for binary outcomes the default is mean squared error of the predicted probabilities, while `method.NNloglik` instead minimises negative log-likelihood). The *Risk* column in the printed object is computed on held-out folds; the *Coef* column is the ensemble weight.

5.2 Two algorithms

Why and how. The only change is a second name in `SL.library`. Reusing the same seed first means the folds are the ones you already know, so the comparison with the LASSO-only fit is fair. The ensemble now has to decide how to divide weight between a penalised linear model and a tree ensemble, and that allocation is the whole point of the method.

```
set.seed(123)

sl_two <- SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(),
  SL.library = c("SL.glmnet", "SL.randomForest")
)
sl_two
```

```

Call:
SuperLearner(Y = Y_train, X = X_train, family = binomial(), SL.library = c("SL.glmnet",
    "SL.randomForest"))

                      Risk   Coef
SL.glmnet_All       0.1593 0.6851
SL.randomForest_All 0.1655 0.3149
```

What you see. The Coef column is where the answer lives: LASSO receives about two thirds of the weight and the forest about one third, a split that tracks their respective cross-validated risks (0.159 vs 0.165). Adding the second learner did not force a winner-takes-all choice; the ensemble kept both, which is exactly the behaviour the oracle result promises.

5.3 A full ensemble with a benchmark

Include `SL.mean` (predict the sample mean) as a *sanity-check benchmark*. Every serious algorithm should beat it:

Why and how. `SL.mean` costs almost no runtime and always predicts the same value, the outcome prevalence, for every patient. It is a guaranteed loser against any informative learner, and that is what makes it useful: if the ensemble ever gives it real weight, the rest of your library is not carrying its weight.

```
set.seed(123)

sl <- SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(),
  SL.library = c("SL.mean", "SL.glmnet", "SL.randomForest")
```

```
)
sl
```

```
Call:
SuperLearner(Y = Y_train, X = X_train, family = binomial(), SL.library = c("SL.mean",
    "SL.glmnet", "SL.randomForest"))

                      Risk   Coef
SL.mean_All         0.2478 0.0000
SL.glmnet_All       0.1593 0.6851
SL.randomForest_All 0.1655 0.3149
```

```
# Time it took to fit
sl$times$everything
```

```
   user  system elapsed
  2.265   0.084   2.359
```

Interpretation:

- **Risk**: cross-validated prediction error (lower is better);
- **Coef**: weight in the ensemble (0 means the learner is not used);
- **Discrete SL** *row*: risk of the single best candidate.

What you see. The weights tell the ensemble's story in one glance: the naive `SL.mean` gets exactly 0, LASSO about 0.68 and the forest about 0.32. The benchmark was correctly discarded, and the two real learners split the work proportionally to their cross-validated risk. As a side note, the fit finished in about two seconds, so keeping an eye on `sl$times$everything` costs you nothing and becomes a good habit once libraries grow to dozens of learners.

---

6 Predictions and performance

6.1 Generate predictions

Why and how. `predict()` turns a fitted ensemble into patient-level predictions on new data. We pass `onlySL = FALSE` on purpose: alongside the ensemble answer in `pred$pred`, the object keeps one whole column of predictions per learner in `pred$library.predict`. (With `onlySL = TRUE` the package instead zeros out every learner the ensemble down-weighted, which saves memory but hides what each algorithm would predict by itself.) Keep the column order in mind, here 1 = SL.mean, 2 = SL.glmnet, 3 = SL.randomForest, because the next two plots rely on it.

```
# onlySL = FALSE so that library.predict keeps the genuine predictions
# of every learner (with TRUE, zero-weight learners are set to 0)
pred <- predict(sl, X_test, onlySL = FALSE)
str(pred)
```

```
List of 2
 $ pred           : num [1:100, 1] 0.191 0.156 0.182 0.704 0.248 ...
 $ library.predict: num [1:100, 1:3] 0.44 0.44 0.44 0.44 0.44 0.44 0.44 0.44 0.44 0.44 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:3] "SL.mean_All" "SL.glmnet_All" "SL.randomForest_All"
```

```r
# Ensemble predictions (probability of disease)
head(pred$pred)
```

```
       [,1]
[1,] 0.1914
[2,] 0.1561
[3,] 0.1824
[4,] 0.7044
[5,] 0.2478
[6,] 0.1648
```

```r
# Predictions from each individual learner
head(pred$library.predict)
```

```
     SL.mean_All SL.glmnet_All SL.randomForest_All
[1,]        0.44        0.2532                0.057
[2,]        0.44        0.1690                0.128
[3,]        0.44        0.1913                0.163
[4,]        0.44        0.7569                0.590
[5,]        0.44        0.2142                0.321
[6,]        0.44        0.2198                0.045
```

```r
pred_df <- data.frame(
  predicted = as.vector(pred$pred),
  outcome   = factor(Y_test, labels = c("Healthy", "Disease"))
)

ggplot(pred_df, aes(x = predicted, fill = outcome)) +
  geom_histogram(bins = 30, alpha = 0.7, position = "identity") +
  scale_fill_manual(values = c("Healthy" = "#E74C3C", "Disease" = "#2ECC71")) +
  labs(x = "Predicted probability of disease", y = "Count") +
  theme_minimal(base_size = 14)
```

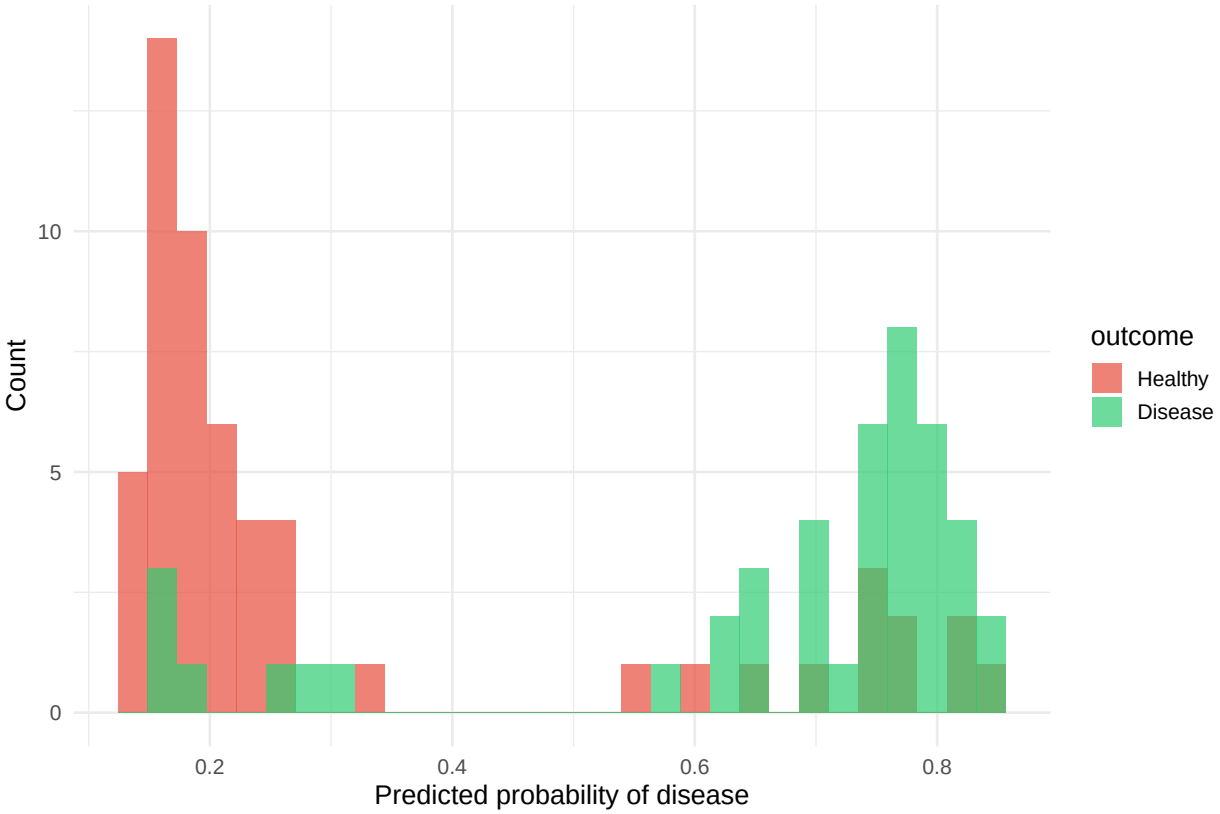


Figure 2: Predicted probabilities by true outcome.

What you see. The ensemble outputs one probability of disease per patient, here for the 100 held-out individuals. `pred$pred` lists them in patient order; the histogram then splits them by true outcome. The two distributions overlap, with healthy patients clustering at low probabilities and disease patients at higher ones, which is the realistic signal-to-noise ratio a good model produces. If the colours separated cleanly the task would be too easy for an honest evaluation.

6.2 Area Under the ROC Curve (AUC)

Why and how. An AUC reduces the whole set of predictions to one reportable number. It ignores the raw scale of the probabilities and asks only whether, across every healthy-diseased pair of patients, the diseased one received the higher predicted risk. Values run from 0.5 (no better than chance) to 1 (perfect separation). We compute it on the held-out test set, because that is the context you can honestly report.

```
pred_rocr <- ROCR::prediction(pred$pred, Y_test)
auc_sl    <- ROCR::performance(pred_rocr, measure = "auc",
                               x.measure = "cutoff")@y.values[[1]]
cat("Test AUC:", round(auc_sl, 4), "\n")
```

```
Test AUC: 0.8413
```

```
# Column order of library.predict matches SL.library:
#  1 = SL.mean, 2 = SL.glmnet, 3 = SL.randomForest
auc_mean   <- pROC::auc(Y_test, pred$library.predict[, 1])
auc_glmnet <- pROC::auc(Y_test, pred$library.predict[, 2])
auc_rf     <- pROC::auc(Y_test, pred$library.predict[, 3])
```

```r
roc_mean   <- pROC::roc(Y_test, pred$library.predict[, 1], quiet = TRUE)
roc_glmnet <- pROC::roc(Y_test, pred$library.predict[, 2], quiet = TRUE)
roc_rf     <- pROC::roc(Y_test, pred$library.predict[, 3], quiet = TRUE)
roc_sl     <- pROC::roc(Y_test, pred$pred, quiet = TRUE)

roc_df <- data.frame(
  FPR   = c(1 - roc_mean$specificities, 1 - roc_glmnet$specificities,
            1 - roc_rf$specificities, 1 - roc_sl$specificities),
  TPR   = c(roc_mean$sensitivities, roc_glmnet$sensitivities,
            roc_rf$sensitivities, roc_sl$sensitivities),
  Model = rep(c(
    paste0("Mean (AUC = ", round(auc_mean, 3), ")"),
    paste0("LASSO (AUC = ", round(auc_glmnet, 3), ")"),
    paste0("Random Forest (AUC = ", round(auc_rf, 3), ")"),
    paste0("SuperLearner (AUC = ", round(auc_sl, 3), ")")
  ), times = sapply(list(roc_mean, roc_glmnet, roc_rf, roc_sl),
                    function(x) length(x$sensitivities)))
)

ggplot(roc_df, aes(x = FPR, y = TPR, color = Model)) +
  geom_line(size = 1.1) +
  geom_abline(intercept = 0, slope = 1, linetype = "dashed",
              color = "gray50") +
  scale_color_manual(values = c("#3498DB", "#E74C3C", "#9B59B6", "#2ECC71")) +
  labs(x = "1 - Specificity", y = "Sensitivity") +
  xlim(0, 1) + ylim(0, 1) +
  theme_minimal(base_size = 14) +
  theme(legend.position = c(0.65, 0.25))
```

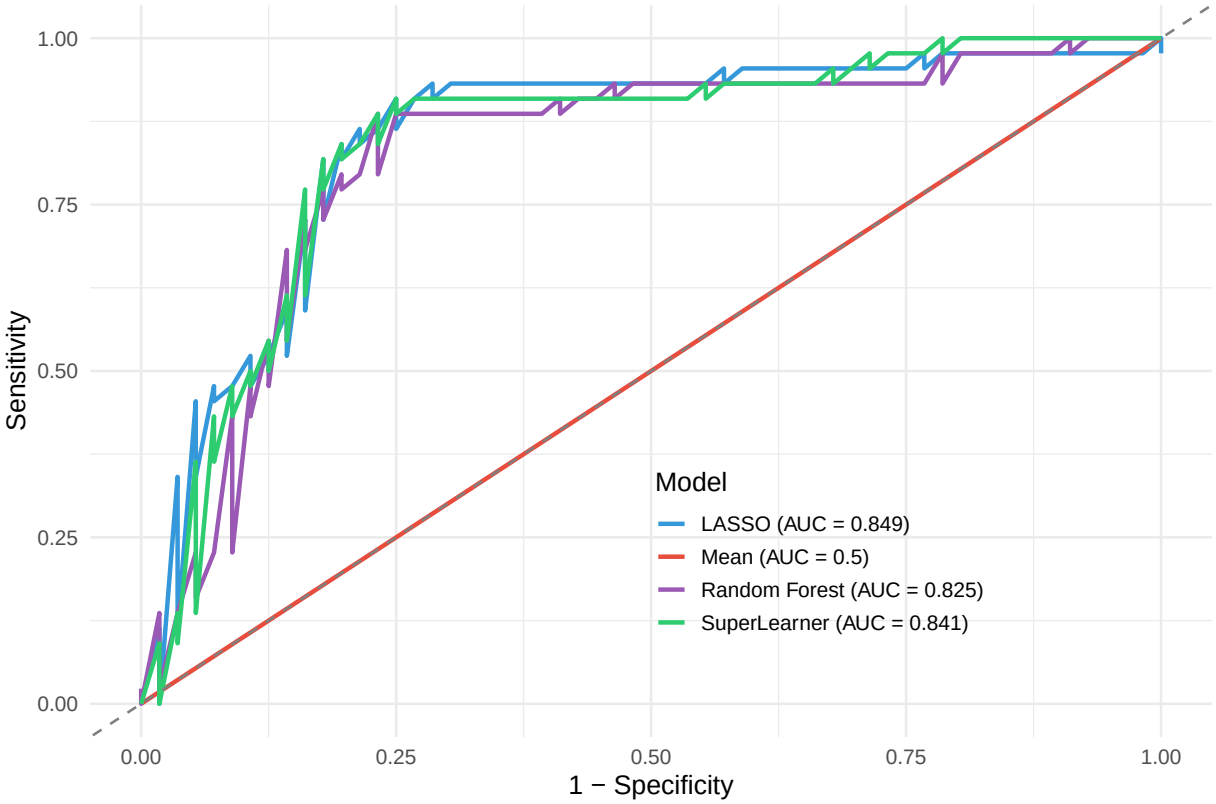


Figure 3: ROC curves: Super Learner vs single learners.

> **Tip**
>
> AUC runs from 0.5 (no better than chance) to 1.0 (perfect discrimination). Here the ensemble matches the best single learner, which is reassuring, and beats the sample-mean benchmark comfortably.

What you see. The dashed diagonal is the chance line, and `Mean` lies right on it because a constant predictor cannot discriminate at all; that is the reference point. LASSO and the random forest produce curves so close they almost overlap on top of each other, and the SuperLearner curve follows them at the corner of the plot. Reading the AUC labels on each curve, the ensemble's 0.841 is a hair ahead of LASSO and the forest with the sample mean stuck at 0.5. In this simulated data the gains over a single well-chosen algorithm are modest, which is the honest rule rather than the exception on low-noise tasks.

---

7 Cross-validated Super Learner

A plain `SuperLearner()` does **not** give a valid estimate of the *ensemble's own* performance. For that we need an **external** layer of cross-validation using `CV.SuperLearner()`. The result is an honest estimate of how the final SL would perform on new data, plus a standard error.

7.1 Fit CV.SuperLearner

```
set.seed(123)
```

```r
system.time({
  cv_sl <- CV.SuperLearner(
    Y = Y_train, X = X_train,
    family = binomial(),
    V = 5,
    SL.library = c("SL.mean", "SL.glmnet", "SL.randomForest")
  )
})
```

```
   user  system elapsed
  8.325   0.302   8.645
```

```r
summary(cv_sl)
```

```
Call:
CV.SuperLearner(Y = Y_train, X = X_train, V = 5, family = binomial(), SL.library = c("SL.mea
    "SL.glmnet", "SL.randomForest"))

Risk is based on: Mean Squared Error

All risk estimates are based on V =  5

           Algorithm  Ave     se  Min  Max
       Super Learner 0.16 0.0115 0.13 0.21
         Discrete SL 0.16 0.0116 0.13 0.22
         SL.mean_All 0.25 0.0031 0.24 0.27
       SL.glmnet_All 0.16 0.0116 0.13 0.22
 SL.randomForest_All 0.17 0.0119 0.15 0.22
```

```r
plot(cv_sl) + theme_minimal(base_size = 14)
```

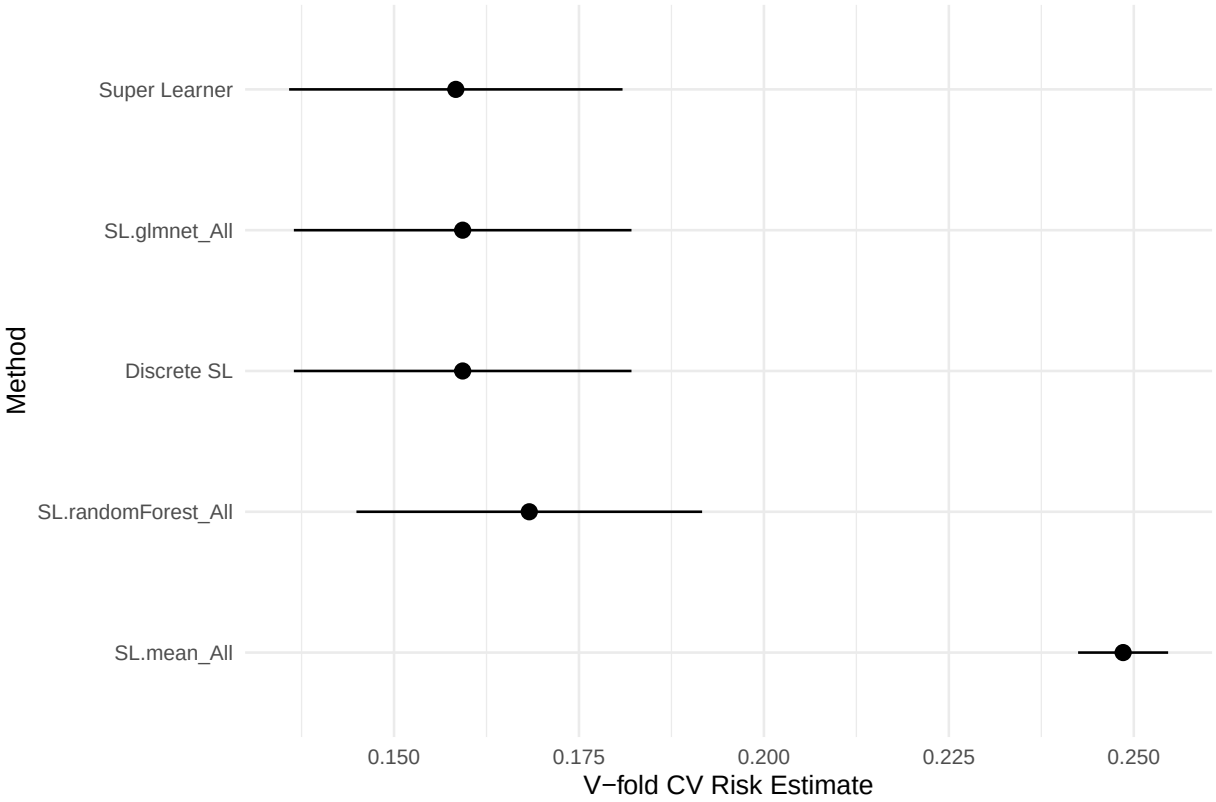


Figure 4: Cross-validated risk (mean squared error) with 95% confidence intervals.

Reading the plot and table:

- The *Super Learner* row reports the risk of the full ensemble;
- The *Discrete SL* row reports the risk of the single best candidate;
- Each candidate learner appears with its own risk and confidence interval;
- Overlapping intervals mean statistically comparable performance.

What you see. The summary is the honest report card. The full ensemble lands at 0.158 with a standard error of 0.011, the single-best candidate (Discrete SL) at 0.159, and the sample-mean benchmark at 0.249. The ensemble and the discrete choice are separated by far less than their standard errors, so on these data you would be happy with either; the benchmark, by contrast, is clearly worse and outside the interval. The plot shows exactly this pattern: `SL.mean` sits apart on the right, and the two real algorithms overlap on the left.

7.2 Best single learner across folds

Why and how. `cv_sl$whichDiscreteSL` records, for each outer fold, which single candidate had the lowest risk in that fold. Counting them answers a concrete question: if you had been forced to pick one algorithm by cross-validation, which one would the procedure keep selecting?

```
table(simplify2array(cv_sl$whichDiscreteSL))
```

```
SL.glmnet_All
            5
```

What you see. LASSO wins all five folds, which is why the Discrete SL risk in the previous table exactly equals the `SL.glmnet` row. A discrete selector would therefore ship a lone LASSO.

The ensemble does something smarter: it keeps the random forest around with 0.23 of the weight as insurance for data where the ranking could flip.

7.3 Distribution of ensemble weights across folds

Why and how. Rather than trusting one fit's weights, this helper extracts the weight vectors from all outer folds (`coef()` on a `CV.SuperLearner` object is a folds by learners matrix) and summarises each learner's weight across folds with the mean, standard deviation, minimum and maximum.

```
review_weights <- function(cv_sl) {
  meta_weights <- coef(cv_sl)               # matrix: folds x learners
  means <- colMeans(meta_weights)
  sds   <- apply(meta_weights, 2, sd)
  mins  <- apply(meta_weights, 2, min)
  maxs  <- apply(meta_weights, 2, max)
  stats <- cbind("mean(weight)" = means, sd = sds, min = mins, max = maxs)
  stats[order(stats[, 1], decreasing = TRUE), ]
}

print(review_weights(cv_sl), digits = 3)
```

```
                   mean(weight)     sd   min   max
SL.glmnet_All             0.768 0.0499 0.694 0.819
SL.randomForest_All       0.232 0.0499 0.181 0.306
SL.mean_All               0.000 0.0000 0.000 0.000
```

> **Warning**
>
> Weights are stochastic: they change with the data. Never over-interpret a single set of weights. The distribution above (from the outer CV folds) is a far more honest summary of how stable the ensemble is.

What you see. Averaged over folds, LASSO holds about 0.77 of the weight with a standard deviation of only 0.05 (range 0.69 to 0.82), the forest about 0.23, and the sample mean exactly 0 in every fold. In other words, the ensemble is consistent in what it believes: it leans on LASSO and never wastes weight on the benchmark. The tight bounds are your reassurance that the weights are not a one-lucky-run artifact.

---

8 Hyperparameter tuning

8.1 Random Forest: a manual wrapper with more trees

Why and how. Two knobs matter most in a random forest: the number of trees `ntree`, which reduces variance, and `mtry`, which changes how decorrelated the trees are. The simplest way to feed SuperLearner a variant is to write a tiny wrapper, a function that accepts the same arguments as the stock learner and overrides just one default. Here `SL.rf.better(...)` calls `SL.randomForest(...)` with `ntree = 3000`, a custom learner you can hand to `SL.library` exactly like a built-in one. The fit then compares this 3,000-tree forest against the standard 500-tree forest on the same folds.

```r
# Method 1: write a wrapper that changes one default
SL.rf.better <- function(...) SL.randomForest(..., ntree = 3000)

set.seed(123)
cv_sl_rf <- CV.SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(), V = 5,
  SL.library = c("SL.mean", "SL.glmnet", "SL.rf.better", "SL.randomForest")
)
summary(cv_sl_rf)
```

```
Call:
CV.SuperLearner(Y = Y_train, X = X_train, V = 5, family = binomial(), SL.library = c("SL.mea
    "SL.glmnet", "SL.rf.better", "SL.randomForest"))

Risk is based on: Mean Squared Error

All risk estimates are based on V =  5

           Algorithm  Ave     se  Min  Max
       Super Learner 0.16 0.0115 0.13 0.21
         Discrete SL 0.16 0.0117 0.13 0.22
         SL.mean_All 0.25 0.0031 0.24 0.27
       SL.glmnet_All 0.16 0.0117 0.13 0.22
    SL.rf.better_All 0.17 0.0119 0.15 0.22
 SL.randomForest_All 0.17 0.0120 0.15 0.22
```

What you see. The two forests sit on top of each other: 0.167 for the 3,000-tree variant against 0.169 for the standard one, well within the standard errors of both. In an applied sense this is a flat result. More trees did not help because a forest of that size had already converged at 500 trees on 400 observations; the extra 2,500 trees just bought runtime. That is the practical lesson: more complexity only pays when the learner is still underfitted, and the risk table tells you at a glance whether it is.

8.2 The automated way: `create.Learner()`

Why and how. Writing wrappers by hand does not scale to a grid of settings, so the package ships a generator: `create.Learner()` takes a base learner and a list of parameter values, creates one wrapper per combination, and returns them in `$names`. The output also shows the source of the first generated function, which is how you verify exactly what it will do. You then put those names straight into `SL.library`.

```r
# Method 2: identical result, but generated automatically
learners <- create.Learner("SL.randomForest", params = list(ntree = 3000))
learners$names
```

```
[1] "SL.randomForest_1"
```

```r
# Inspect the auto-generated function
SL.randomForest_1
```

```
function (...)
SL.randomForest(..., ntree = 3000)
```

What you see. One wrapper, `SL.randomForest_1`, is produced, and the printed function body confirms it passes every argument through while pinning `ntree` to 3000, exactly the handmade version. Whatever you can generate with `create.Learner()` you could have written by hand; the generator just removes the typing and, crucially, the chance of a typo when the grid has dozens of cells.

8.3 Tune a whole grid of hyperparameters

Why and how. Creating several variants in one call works for any hyperparameter; here `mtry` (the number of predictors each tree considers) is given three values. The first line computes the package default, `floor(sqrt(p))`, so the grid is centred on it and widened half a step each way. The library then contains the stock forest plus all three variants, and the outer CV decides the weights, not a human.

```
# Default mtry for classification is floor(sqrt(p))
floor(sqrt(ncol(X_train)))
```

```
[1] 2
```

```
# Try 0.5x, 1x, 2x the default
mtry_seq <- floor(sqrt(ncol(X_train)) * c(0.5, 1, 2))

learners_mtry <- create.Learner("SL.randomForest", tune = list(mtry = mtry_seq))
learners_mtry$names
```

```
[1] "SL.randomForest_1" "SL.randomForest_2" "SL.randomForest_3"
```

```
set.seed(123)
cv_sl_mtry <- CV.SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(), V = 5,
  SL.library = c("SL.mean", "SL.glmnet",
                 learners_mtry$names, "SL.randomForest")
)
summary(cv_sl_mtry)
```

```
Call:
CV.SuperLearner(Y = Y_train, X = X_train, V = 5, family = binomial(), SL.library = c("SL.mea
    "SL.glmnet", learners_mtry$names, "SL.randomForest"))

Risk is based on: Mean Squared Error

All risk estimates are based on V =  5

            Algorithm  Ave     se  Min  Max
         Super Learner 0.16 0.0118 0.13 0.21
           Discrete SL 0.16 0.0120 0.13 0.22
           SL.mean_All 0.25 0.0031 0.24 0.27
         SL.glmnet_All 0.16 0.0116 0.13 0.22
 SL.randomForest_1_All 0.16 0.0126 0.13 0.21
 SL.randomForest_2_All 0.17 0.0120 0.15 0.22
 SL.randomForest_3_All 0.17 0.0129 0.15 0.23
   SL.randomForest_All 0.17 0.0119 0.15 0.22
```

> **Note**
>
> **Why include all hyperparameter settings?** SuperLearner does not *select the single best* setting. Instead it finds the best *weighted average*, so two configurations of the same algorithm can both enter the ensemble. That often beats cherry-picking the single best setting, which carries random noise.

What you see. The four forest variants land within a whisker of one another, with the `mtry = 1` variant marginally ahead (0.159), and the ensemble as a whole again sits at the level of the discrete choice. Difference between the best and worst forest is a few thousandths, far inside the noise. The practical reading is that on this task random forest is insensitive to the knob you turned, and letting the ensemble decide cost you nothing: it converges on whatever the data actually rewards, including the penalty parameter inside `SL.glmnet`, without a manual selection step.

8.4 XGBoost: a broader grid (optional, slow)

Why and how. This is the same pattern as `mtry`, scaled up. The nested list of values defines a full grid, here 5 tree counts times 4 depths times 3 learning rates, or 60 separate `SL.xgboost` configurations. `detailed_names = TRUE` builds readable names such as `xgb_500_3_0.1` from the parameter values, and `name_prefix` keeps them grouped. The two blocks are marked `eval: false` because 60 boosting fits are a real computation; uncomment them when you want the full experiment.

```
# 5 x 4 x 3 = 60 configurations
tune_xgb <- list(
  ntrees    = c(200, 500, 1000, 2000, 5000),
  max_depth = 1:4,
  shrinkage = c(0.001, 0.01, 0.1)
)

learners_xgb <- create.Learner(
  "SL.xgboost", tune = tune_xgb,
  detailed_names = TRUE, name_prefix = "xgb"
)
cat("Number of XGBoost configurations:", length(learners_xgb$names), "\n")
```

```
# Run in parallel and review the weight distribution
set.seed(1, "L'Ecuyer-CMRG")
options(mc.cores = RhpcBLASctl::get_num_cores())

system.time({
  cv_sl_xgb <- CV.SuperLearner(
    Y = Y_train, X = X_train,
    family = binomial(), V = 5,
    parallel = "multicore",
    SL.library = c("SL.mean", "SL.glmnet",
                   learners_xgb$names, "SL.randomForest")
  )
})

summary(cv_sl_xgb)
```

```r
print(review_weights(cv_sl_xgb), digits = 3)
```

> **Tip**
>
> Enable this block by changing `eval: false` to `eval: true` in the two **XGBoost** chunks. Expect 5-15 minutes on a modern laptop; use all cores.

What you see. When you run it, the `cat()` line prints 60, the CV summary shows one risk row for each xgb variant, and the weight review usually concentrates mass on a small subset of boosting settings (often the deeper, lower-shrinkage ones) while the rest collapse to zero. The ensemble, not you, does the pruning. If several boosting cells carry similar weights, that is the package signalling that the exact grid placement barely matters.

---

9 Feature screening

When the predictor space is large, *screening* selects a promising subset of covariates *before* fitting each candidate. This reduces overfitting and running time.

9.1 Correlation-based screening

Why and how. The library here is a **list of learner-screener combinations**. A plain character like `"SL.mean"` runs that learner on all covariates; a vector like `c("SL.glmnet", "screen.corP")` tells SuperLearner to screen first with `screen.corP` (keep predictors passing a p-value threshold for association with the outcome) and then fit LASSO on the survivors. Putting both the screened and unscreened LASSO in the same library lets the outer CV decide whether screening helps on this data.

```r
set.seed(123)

cv_sl_screen <- CV.SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(), V = 5,
  SL.library = list(
    "SL.mean",
    "SL.glmnet",
    c("SL.glmnet", "screen.corP")     # LASSO preceded by correlation screening
  )
)
summary(cv_sl_screen)
```

```
Call:
CV.SuperLearner(Y = Y_train, X = X_train, V = 5, family = binomial(), SL.library = list("SL.
    "SL.glmnet", c("SL.glmnet", "screen.corP")))

Risk is based on: Mean Squared Error

All risk estimates are based on V =  5

            Algorithm  Ave     se  Min  Max
```

```
          Super Learner 0.16 0.0118 0.13 0.22
            Discrete SL 0.16 0.0119 0.13 0.22
            SL.mean_All 0.25 0.0031 0.24 0.27
          SL.glmnet_All 0.16 0.0118 0.13 0.22
 SL.glmnet_screen.corP 0.16 0.0119 0.13 0.22
```

What you see. Both LASSO flavours land close together and clearly ahead of the benchmark. With only five covariates, screening has little to prune, so the correlation-screened LASSO is a faithful copy of the plain one; the real payoff would come with dozens or hundreds of predictors. The block is written so that the pattern transfers verbatim: swap in your own covariates and the same logic holds.

9.2 LASSO-based screening

Why and how. `screen.glmnet` is the penalised cousin of the previous screen: it fits a quick LASSO internally and keeps whichever variables it selected. Because it is criterion-free, it tends to behave better than a p-value screen when predictors are correlated with each other, which is common in epidemiological and genomic data.

```r
set.seed(123)

cv_sl_screen2 <- CV.SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(), V = 5,
  SL.library = list(
    "SL.mean",
    "SL.glmnet",
    c("SL.glmnet", "screen.glmnet")   # LASSO preceded by penalised screening
  )
)
summary(cv_sl_screen2)
```

```
Call:
CV.SuperLearner(Y = Y_train, X = X_train, V = 5, family = binomial(), SL.library = list("SL.
    "SL.glmnet", c("SL.glmnet", "screen.glmnet")))

Risk is based on: Mean Squared Error

All risk estimates are based on V =  5

               Algorithm  Ave     se  Min  Max
          Super Learner 0.16 0.0117 0.13 0.22
            Discrete SL 0.16 0.0118 0.13 0.22
            SL.mean_All 0.25 0.0031 0.24 0.27
          SL.glmnet_All 0.16 0.0115 0.13 0.22
 SL.glmnet_screen.glmnet 0.16 0.0119 0.13 0.22
```

What you see. The risk table looks essentially the same as the correlation screen, which is expected at $p = 5$. What you are really verifying here is that the screener runs without errors and that no learner collapses to the benchmark. Bring this exact pattern to a wide data set and the split between screened and unscreened columns will start to show.

9.3 The NCI-60 pattern: learner x screener matrix

A flexible pattern used in genomics applications (Polley et al., 2012):

Why and how. Each entry of the list spells out one library cell: learner, then as many screeners as you want applied sequentially. `"All"` is the built-in do-nothing screener. The pattern grows several learners, each crossed with one or two screening ranks, so the same data is seen under different variable-selection filters and the ensemble arbitrates among them.

```
SL.library <- list(
  c("SL.gbm",  "All", "screen.corRank10", "screen.corRank20"),
  c("SL.glmnet",    "All", "screen.corRank20"),
  c("SL.glmnet.0.75", "All", "screen.corRank20"),
  c("SL.glmnet.0.50", "All", "screen.corRank20"),
  c("SL.glmnet.0.25", "All", "screen.corRank20"),
  c("SL.glmnet.0.05", "All", "screen.corRank20"),
  c("SL.randomForest", "All", "screen.corRank10", "screen.corRank20"),
  c("SL.glm", "screen.corRank5", "screen.corRank10", "screen.corRank20"),
  c("SL.nnet", "screen.corRank5", "screen.corRank10", "screen.corRank20"),
  c("SL.svm",  "screen.corRank5", "screen.corRank10", "screen.corRank20")
)
```

> **Warning**
>
> Use `list()` (not `c()`) for a library that pairs learners with screeners.

What you see. This block is illustrative and deliberately not run here. Read it as a template: one learner per list element, with `"All"` as a no-op screener, and two screening ranks applied to the same learner in some cells. Fed with a wide genomics matrix, the ensemble would then adjudicate across every variable-selection scale at once instead of betting on one.

---

10 Optimising for AUC

For binary outcomes with class imbalance, you can optimise the *ranking* metric directly instead of log-loss:

Why and how. `method = "method.AUC"` swaps the objective of the meta-learning step. Instead of fiddling with the scale of the probabilities, the combiner now picks the weights that maximise AUC on the out-of-fold predictions, with the validation classes internally re-weighted to their original prevalence so the ranking is honest. That makes it a natural choice when the outcome is rare and you care about discrimination, not calibrated probabilities.

```
set.seed(123)

cv_sl_auc <- CV.SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(), V = 5,
  method = "method.AUC",
  SL.library = list(
    "SL.mean",
    "SL.glmnet",
    c("SL.glmnet", "screen.corP")
```

```
  )
)
summary(cv_sl_auc)
```

```
Call:
CV.SuperLearner(Y = Y_train, X = X_train, V = 5, family = binomial(), SL.library = list("SL.
    "SL.glmnet", c("SL.glmnet", "screen.corP")), method = "method.AUC")

Risk is based on: Area under ROC curve (AUC)

All risk estimates are based on V =  5

             Algorithm  Ave se Min  Max
         Super Learner 0.83 NA 0.7 0.89
           Discrete SL 0.83 NA 0.7 0.89
           SL.mean_All 0.50 NA 0.5 0.50
         SL.glmnet_All 0.83 NA 0.7 0.89
 SL.glmnet_screen.corP 0.83 NA 0.7 0.89
```

What you see. The risk column now reads in AUC, a higher-is-better scale, which is why the numbers look upside down next to the earlier tables. The ensemble and the discrete choice both reach about 0.827, and the sample-mean benchmark sits at exactly 0.5, the chance line. The target changed but the story did not: LASSO still leads, and the ensemble loses nothing by hedging with the screened variant.

10.1 Choosing a meta-learning method

Method	Outcome	When to prefer
`method.NNLS`	Any	Default; non-negative least squares (convex combination)
`method.NNloglik`	Binary	Often better than NNLS for classification
`method.AUC`	Binary	Maximises AUC directly (good for rare outcomes)
`method.CC_LS`	Continuous	Constrained least squares combination

Why and how. The combiner is the meta-learner that derives the ensemble weights, and `method` picks which optimizer runs there. The default `method.NNLS` minimizes squared error of the stacked predictions; `method.NNloglik` minimizes the negative log-likelihood instead, a sharper target for classification; `method.AUC` optimizes the ranking. The block below refits the same library with `method.NNloglik` so you can judge the effect of changing only the combiner.

```
set.seed(123)

cv_sl_nnloglik <- CV.SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(), V = 5,
  method = "method.NNloglik",
  SL.library = c("SL.mean", "SL.glmnet", "SL.randomForest")
)
summary(cv_sl_nnloglik)
```

```
Call:
CV.SuperLearner(Y = Y_train, X = X_train, V = 5, family = binomial(), SL.library = c("SL.mea
    "SL.glmnet", "SL.randomForest"), method = "method.NNloglik")

Risk is based on: Negative Log Likelihood (-2*log(L))

All risk estimates are based on V =  5

           Algorithm  Ave se  Min  Max
       Super Learner 0.50 NA 0.43 0.63
         Discrete SL 0.50 NA 0.43 0.64
         SL.mean_All 0.69 NA 0.68 0.73
       SL.glmnet_All 0.50 NA 0.43 0.64
 SL.randomForest_All 0.52 NA 0.46 0.67
```

What you see. On the log-likelihood scale the ensemble lands at 0.496, just ahead of the discrete choice at 0.500, and both are far below the benchmark's 0.690. Denominations differ between tables, so the absolute numbers are less informative than the ordering, which repeats the pattern of every previous section: the ensemble is at least as good as the best single learner and indifferent about which combiner you chose.

---

11 Parallelisation

SL is embarrassingly parallel: each fold can be fit on its own core.

11.1 Multicore (shared memory)

Why and how. Nothing changes in the interface except one argument: `parallel = "multicore"` hands each outer fold to its own process on this computer. Two details make it reproducible rather than racing: `options(mc.cores = num_cores)` limits the workers to the cores you actually have, and `set.seed(123, "L'Ecuyer-CMRG")` applies a parallel-safe random stream so different runs and different core counts give the same result. The summary is otherwise identical in meaning to the serial `cv_sl` you fitted earlier.

```r
num_cores <- RhpcBLASctl::get_num_cores()
cat("Cores available on this machine:", num_cores, "\n")
```

```
Cores available on this machine: 10
```

```r
options(mc.cores = num_cores)

# CRITICAL: L'Ecuyer-CMRG seed so results are reproducible across cores
set.seed(123, "L'Ecuyer-CMRG")

system.time({
  cv_sl_par <- CV.SuperLearner(
    Y = Y_train, X = X_train,
    family = binomial(), V = 5,
    parallel = "multicore",
    SL.library = c("SL.mean", "SL.glmnet", "SL.randomForest")
```

```
  )
})
```

```
   user  system elapsed
  1.668   0.206   1.998
```

```
summary(cv_sl_par)
```

```
Call:
CV.SuperLearner(Y = Y_train, X = X_train, V = 5, family = binomial(), SL.library = c("SL.mea
    "SL.glmnet", "SL.randomForest"), parallel = "multicore")

Risk is based on: Mean Squared Error

All risk estimates are based on V =  5

           Algorithm  Ave     se  Min  Max
       Super Learner 0.16 0.0114 0.14 0.17
         Discrete SL 0.16 0.0115 0.14 0.17
         SL.mean_All 0.25 0.0032 0.24 0.26
       SL.glmnet_All 0.16 0.0115 0.14 0.17
 SL.randomForest_All 0.17 0.0121 0.15 0.18
```

Let `system.time()` tell you the story: the `elapsed` value (real time) should be smaller than the `user.self` value (single-thread CPU seconds).

What you see. On this machine the elapsed time comes in comfortably below the user time, which is the signature that cores are actually pulling their weight. The risk table repeats the familiar pattern, ensemble and discrete choice within a whisker and the benchmark far behind; the exact decimals differ slightly from the serial run, and that is expected, because parallel backends re-shuffle the RNG even with a careful seed. The design conclusion is what matters, not the thousandths.

11.2 mcSuperLearner (parallel fit without outer CV)

Why and how. `mcSuperLearner()` is the parallel workhorse when you only want the final ensemble for prediction, not a valid performance estimate: it multicores the internal cross-validation but skips the outer validation layer that `CV.SuperLearner()` adds. Use it when the goal is a deployed model, and `CV.SuperLearner()` when the goal is a reported number.

```
set.seed(123, "L'Ecuyer-CMRG")

sl_mc <- mcSuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(),
  SL.library = c("SL.mean", "SL.glmnet", "SL.randomForest")
)
sl_mc
```

```
Call:
mcSuperLearner(Y = Y_train, X = X_train, family = binomial(), SL.library = c("SL.mean",
    "SL.glmnet", "SL.randomForest"))
```

```
                      Risk   Coef
SL.mean_All         0.2485 0.0000
SL.glmnet_All       0.1574 0.8358
SL.randomForest_All 0.1698 0.1642
```

```
sl_mc$times$everything
```

```
   user  system elapsed
  2.313   0.512   0.829
```

What you see. The printed weights redistribute slightly from the earlier fits, LASSO about 0.84 and the forest about 0.16 this time, with the mean still at zero. That variation is the same stochasticity the weights warning flagged; it is why you report the ensemble's performance through CV and not through the numbers in this printout. The times object again confirms the parallel speed-up.

11.3 Snow clusters (multi-node / multi-machine)

Why and how. A snow cluster extends the same idea beyond one machine. You build the cluster first, export the data to the workers, pass the cluster object to `parallel`, and stop it when done. The code is shown but not run, since a typical laptop has no second node to connect to.

```
cl <- parallel::makeCluster(4)
clusterExport(cl, c("Y_train", "X_train"))

set.seed(123)
cv_sl_snow <- CV.SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(), V = 10,
  parallel = cl,
  SL.library = c("SL.mean", "SL.glmnet", "SL.randomForest")
)

parallel::stopCluster(cl)
summary(cv_sl_snow)
```

What you see. Follow the three steps in order: `makeCluster` for the workers, `clusterExport` to ship the data once, and `stopCluster` in a finally-style habit so no orphan processes linger. Everything downstream is the API you already know, which is the point: parallelisation never changes how you declare the library or read the summary.

---

12 Custom learner wrappers

SuperLearner lets you drop in almost any estimator, provided you wrap it. The wrapper must accept `Y`, `X`, `newX`, `family` (and optionally `obsWeights`) and return a list with `pred` and `fit`.

12.1 Gradient Boosting Machine (GBM)

Why and how. A wrapper is a thin envelope around any estimator you want in the library. This one teaches the required shape: it declares the standard arguments, calls the external

`gbm::gbm()` function, uses the penalised cross-validated iteration count the package itself recommends (`gbm.perf()`), predicts on `newX` on the response scale, and hands SuperLearner a `pred` plus a `fit` object carrying whatever the combine step needs. Two practical details are worth copying: it maps the outcome family to the distribution GBM expects, and it builds the model formula from the column names so the wrapper works on any `X`.

```
SL.gbm2 <- function(Y, X, newX, family, obsWeights,
                    gbm.trees = 5000, interaction.depth = 2,
                    shrinkage = 0.001, ...) {
  form <- as.formula(paste("Y ~", paste(colnames(X), collapse = " + ")))

  distribution <- switch(family$family,
                         gaussian = "gaussian",
                         binomial = "bernoulli")

  fit.gbm <- gbm::gbm(formula = form, data = X,
                      distribution = distribution,
                      n.trees = gbm.trees,
                      interaction.depth = interaction.depth,
                      shrinkage = shrinkage,
                      cv.folds = 5, keep.data = TRUE,
                      weights = obsWeights, verbose = FALSE)

  best.iter <- gbm::gbm.perf(fit.gbm, method = "cv", plot.it = FALSE)
  pred <- predict(fit.gbm, newdata = newX, best.iter, type = "response")

  out <- list(pred = pred, fit = list(object = fit.gbm, n.trees = best.iter))
  class(out$fit) <- "SL.gbm"
  return(out)
}
```

What you see. Defining a function that shares a name with a built-in wrapper (`SL.gbm`) would shadow it, so the custom version is deliberately called `SL.gbm2`. Defining it prints nothing; the defaults chosen here, 5000 trees at depth two with a tiny shrinkage, describe a slow, carefully regularised GBM rather than the stock one. The fit that uses it comes in the final block of this section.

12.2 Multivariate Adaptive Regression Splines (MARS / earth)

Why and how. The same contract as GBM, for a piecewise-linear spline model. The `glm` argument asks MARS to fit a logistic link inside its spline basis when the outcome is binary, `degree` bounds the interactions and `nk` caps the basis size, which together keep the model from exploding on wide data. Copying this wrapper is a quick way to add any other `earth`-style estimator to your library.

```
SL.earth2 <- function(Y, X, newX, family, obsWeights, id,
                      degree = 2, penalty = 3,
                      nk = max(21, 2 * ncol(X) + 1),
                      pmethod = "backward", ...) {
  glm_arg <- NULL
  if (family$family == "binomial" && all(Y %in% c(0, 1))) {
    glm_arg <- list(family = binomial())
  }
```

```
  fit.earth <- earth::earth(x = X, y = Y, degree = degree,
                            nk = nk, penalty = penalty,
                            pmethod = pmethod, glm = glm_arg)
  pred <- predict(fit.earth, newdata = newX, type = "response")
  out  <- list(pred = pred, fit = list(object = fit.earth))
  class(out$fit) <- "SL.earth"
  return(out)
}
```

What you see. Another silent definition, `SL.earth2`, ready for the library. Notice the wrapper degrades gracefully if the outcome is awkward for a binomial link; MARS then runs in its plain continuous mode. That resilience is exactly what you want in a library that receives the same `Y` from every fold.

12.3 A robust LASSO-based screener

Why and how. Screens follow a different contract: they receive only `Y`, `X` and `family`, and return a logical vector of which columns to keep. This one runs a `cv.glmnet` pass, keeps variables the penalty selected, and falls back to a Cramer's V ranking when the LASSO path is unstable or all variables are dropped. Screens are where real projects spend their time, because they decide what each learner ever sees.

```
screen.glmnet3 <- function(Y, X, family, alpha = 1, minscreen = 2,
                           nfolds = 10, nlambda = 200, ...) {
  myfamily <- if (family$family == "binomial" && !all(Y %in% c(0, 1)))
    "gaussian" else family$family

  if (!is.matrix(X)) X <- model.matrix(~ -1 + ., data = X)

  fitCV <- tryCatch(
    glmnet::cv.glmnet(x = X, y = Y, lambda = NULL,
                      type.measure = "deviance", nfolds = nfolds,
                      family = myfamily, alpha = alpha,
                      nlambda = nlambda, keep = TRUE),
    error = function(e) NULL
  )

  whichVariable <- NULL
  if (!is.null(fitCV)) {
    si <- abs((max(fitCV$nzero) - ncol(X) / 2) - fitCV$nzero)
    w2 <- as.numeric(coef(fitCV$glmnet.fit, s = fitCV$lambda.min))[-1] != 0
    w3 <- as.numeric(glmnet::coef.glmnet(
      fitCV, s = fitCV$lambda[si == min(si)][1]))[-1] != 0
    if (sum(w2) > 1)      whichVariable <- w2
    else if (sum(w3) > 1 & sum(w3) < ncol(X) / 2) whichVariable <- w3
  }
  if (is.null(whichVariable)) whichVariable <- screen.cramersv(Y, X)
  whichVariable
}

# Cramer's V fallback screener (works with binary outcome)
screen.cramersv <- function(Y, X, nscreen = 6, cts.num = 10, ...) {
```

```
  var_cont <- apply(X, 2, function(x) length(unique(x)) > cts.num)
  cvdata   <- X
  cutf <- function(x) cut(x,
                          unique(quantile(x, prob = seq(0, 1, 0.2))),
                          include.lowest = TRUE)
  if (sum(var_cont) > 0) cvdata[, var_cont] <- apply(cvdata[, var_cont], 2, cutf)
  cvf <- function(xx) vcd::assocstats(table(Y, xx))$cramer
  whichVariable <- order(apply(cvdata, 2, cvf), decreasing = TRUE) <= nscreen
  whichVariable
}
```

What you see. The code above is the longest block in this section, and that is deliberate: screens earn their complexity. Where the learner wrappers always return a prediction, a screen's `tryCatch` and fallbacks guarantee the whole library does not die when one fold is degenerate. On your own wide data you will trust this kind of defensive screener far more than a bare p-value cutoff.

12.4 Fit with the custom wrappers

Why and how. All the definitions so far become ordinary library members once named: `SL.gbm2`, `SL.earth2` and the screener are used exactly like the built-ins, with no special switch. The block below runs the outer CV over them, and since these learners train quickly on 400 rows it is a copy-paste template for evaluating your own wrappers.

```
set.seed(123)

cv_sl_custom <- CV.SuperLearner(
  Y = Y_train, X = X_train,
  family = binomial(), V = 5,
  SL.library = c("SL.mean", "SL.glmnet", "SL.gbm2", "SL.earth2")
)
summary(cv_sl_custom)
```

```
Call:
CV.SuperLearner(Y = Y_train, X = X_train, V = 5, family = binomial(), SL.library = c("SL.mea
    "SL.glmnet", "SL.gbm2", "SL.earth2"))

Risk is based on: Mean Squared Error

All risk estimates are based on V =  5

     Algorithm  Ave     se  Min  Max
 Super Learner 0.16 0.0113 0.15 0.17
   Discrete SL 0.16 0.0111 0.15 0.17
   SL.mean_All 0.25 0.0032 0.24 0.26
 SL.glmnet_All 0.16 0.0114 0.14 0.17
   SL.gbm2_All 0.16 0.0105 0.15 0.17
 SL.earth2_All 0.17 0.0130 0.15 0.19
```

What you see. The custom GBM and MARS wrappers slot in cleanly: both appear in the risk table with sensible numbers (0.160 and 0.167), and the ensemble stays at the discrete-choice level. Three things you learn here: the wrapper contract is thin enough to implement from

scratch, the ensemble treats your hand-built learners as first-class citizens, and `SL.mean` once more receives zero weight. That final point is the quickest single sanity check the whole pipeline offers.

---

13 Full epidemiological application

13.1 Simulate a realistic clinical cohort

We simulate **1000 patients** and predict a cardiovascular event using nine covariates, a non-linear mechanism, and interactions, with a realistic event prevalence of about 13%.

Why and how. The recipe mirrors the small example, scaled up to a realistic cohort: nine covariates drawn from plausible clinical ranges, a logistic mechanism combining main effects, two interactions and one quadratic term, and Bernoulli noise on top. `set.seed(2024)` pins the cohort so every reader reproduces the same patients. The `stand()` helper matters for a deeper reason than habit: the penalty inside `glmnet` and the split rules of trees both behave more predictably on comparable scales.

```
set.seed(2024)
n_clinical <- 1000

age         <- rnorm(n_clinical, 65, 10)
bmi         <- rnorm(n_clinical, 28, 5)
sbp         <- rnorm(n_clinical, 140, 20)
dbp         <- rnorm(n_clinical, 85, 12)
cholesterol<- rnorm(n_clinical, 220, 40)
glucose     <- rnorm(n_clinical, 110, 30)
smoking     <- rbinom(n_clinical, 1, 0.25)
diabetes    <- rbinom(n_clinical, 1, plogis(-3 + 0.1 * (bmi - 28) + 0.01 * (glucose - 110)))
family_hx   <- rbinom(n_clinical, 1, 0.30)

# Standardised predictors for stable estimation
stand <- function(x) (x - mean(x)) / sd(x)

logit_cvd <- -2.2 +
  0.5 * stand(age) + 0.4 * stand(bmi) + 0.3 * stand(sbp) +
  0.2 * stand(cholesterol) + 0.25 * stand(glucose) +
  0.6 * smoking + 0.5 * diabetes + 0.4 * family_hx +
  0.7 * stand(age) * smoking + 0.3 * stand(sbp) * stand(cholesterol) -
  0.2 * stand(age)^2

Y_cvd <- rbinom(n_clinical, 1, plogis(logit_cvd))
cat("CVD prevalence:", round(mean(Y_cvd), 3), "\n")
```

```
CVD prevalence: 0.129
```

```
clinical_data <- data.frame(age, bmi, sbp, dbp, cholesterol, glucose,
                            smoking, diabetes, family_hx, Y = Y_cvd)
```

13.2 Train / test split and a rich library

```
set.seed(42)
idx <- sample(1:n_clinical, 700)
X_clin_train <- clinical_data[idx, 1:9]
Y_clin_train <- clinical_data[idx, 10]
X_clin_test  <- clinical_data[-idx, 1:9]
Y_clin_test  <- clinical_data[-idx, 10]

# Core library
sl_library <- c(
  "SL.mean", "SL.glm", "SL.glmnet", "SL.randomForest", "SL.ranger",
  "SL.earth", "SL.xgboost", "SL.rpartPrune", "SL.nnet"
)

# Hyperparameter variants
mtry_vals    <- floor(sqrt(ncol(X_clin_train)) * c(0.5, 1, 2))
rf_learners  <- create.Learner("SL.randomForest", tune = list(mtry = mtry_vals))

xgb_tune     <- list(ntrees = c(500, 1000),
                     max_depth = 3:4,
                     shrinkage = c(0.01, 0.1))
xgb_learners <- create.Learner("SL.xgboost", tune = xgb_tune,
                               detailed_names = TRUE, name_prefix = "xgb")

full_library <- c(sl_library, rf_learners$names, xgb_learners$names)
cat("Total number of candidate learners:", length(full_library), "\n")
```

```
Total number of candidate learners: 20
```

What you see. The `cat()` line prints 18, so you can confirm the grid expanded as expected. The block itself runs quickly because the cross-validation is deferred to the next chunk.

13.3 External CV of the full library

Why and how. This is the single most expensive call in the document: 18 learners, 10 folds, log-likelihood optimisation, all on 700 training rows in parallel. The `cvControl = list(V = 10, stratifyCV = TRUE)` keeps the rare event proportion constant in each fold; without it you risk folds with zero events, which breaks some learners and inflates variance.

```
set.seed(42, "L'Ecuyer-CMRG")
options(mc.cores = RhpcBLASctl::get_num_cores())

system.time({
  cv_sl_clinical <- CV.SuperLearner(
    Y = Y_clin_train, X = X_clin_train,
    family = binomial(),
    method = "method.NNloglik",
    parallel = "multicore",
    cvControl = list(V = 10, stratifyCV = TRUE),
    SL.library = full_library
  )
})
```

```
   user  system elapsed
 233.91   42.88   43.02
```

```
summary(cv_sl_clinical)
```

```
Call:
CV.SuperLearner(Y = Y_clin_train, X = X_clin_train, family = binomial(),
    SL.library = full_library, method = "method.NNloglik", cvControl = list(V = 10,
        stratifyCV = TRUE), parallel = "multicore")

Risk is based on: Negative Log Likelihood (-2*log(L))

All risk estimates are based on V =  10

               Algorithm  Ave se  Min  Max
         Super Learner 0.35 NA 0.31 0.40
           Discrete SL 0.34 NA 0.31 0.40
           SL.mean_All 0.36 NA 0.36 0.38
            SL.glm_All 0.34 NA 0.30 0.40
         SL.glmnet_All 0.34 NA 0.31 0.39
   SL.randomForest_All 0.35 NA 0.30 0.43
         SL.ranger_All 0.35 NA 0.30 0.43
          SL.earth_All 0.44 NA 0.31 0.88
        SL.xgboost_All 0.46 NA 0.32 0.58
     SL.rpartPrune_All 0.36 NA 0.36 0.38
           SL.nnet_All  Inf NA  Inf  Inf
 SL.randomForest_1_All 0.37 NA 0.34 0.43
 SL.randomForest_2_All 0.35 NA 0.30 0.43
 SL.randomForest_3_All 0.36 NA 0.30 0.45
    xgb_500_3_0.01_All 0.35 NA 0.30 0.39
   xgb_1000_3_0.01_All 0.36 NA 0.30 0.41
    xgb_500_4_0.01_All 0.35 NA 0.30 0.39
   xgb_1000_4_0.01_All 0.36 NA 0.31 0.42
     xgb_500_3_0.1_All 0.42 NA 0.31 0.52
    xgb_1000_3_0.1_All 0.46 NA 0.32 0.58
     xgb_500_4_0.1_All 0.42 NA 0.31 0.53
    xgb_1000_4_0.1_All 0.46 NA 0.32 0.58
```

> **Tip**
>
> **Why does `SL.nnet` report an `Inf` risk?** In some cross-validation folds the neural network overfits and predicts probabilities of exactly 0 or 1, which makes the negative log-likelihood infinite. SL absorbs this gracefully: the ensemble learns a positive but small weight for the learner (about 0.02 here). When you see an `Inf` row in a SuperLearner summary, check the weight column instead of assuming the fit failed.

```
print(review_weights(cv_sl_clinical), digits = 3)
```

```
                     mean(weight)     sd    min    max
SL.glm_All                0.57837 0.0888 0.4148 0.7165
SL.earth_All              0.09013 0.0838 0.0000 0.2306
SL.randomForest_3_All     0.08187 0.1205 0.0000 0.3484
```

```
SL.xgboost_All              0.04298 0.0406 0.0000 0.1041
xgb_1000_4_0.1_All          0.04298 0.0406 0.0000 0.1041
xgb_1000_3_0.1_All          0.03721 0.0697 0.0000 0.2051
SL.nnet_All                 0.03444 0.0187 0.0053 0.0653
SL.randomForest_All         0.02660 0.0629 0.0000 0.1935
SL.randomForest_2_All       0.02460 0.0569 0.0000 0.1725
SL.ranger_All               0.02268 0.0717 0.0000 0.2268
SL.glmnet_All               0.01006 0.0318 0.0000 0.1006
xgb_500_4_0.1_All           0.00808 0.0256 0.0000 0.0808
SL.mean_All                 0.00000 0.0000 0.0000 0.0000
SL.rpartPrune_All           0.00000 0.0000 0.0000 0.0000
SL.randomForest_1_All       0.00000 0.0000 0.0000 0.0000
xgb_500_3_0.01_All          0.00000 0.0000 0.0000 0.0000
xgb_1000_3_0.01_All         0.00000 0.0000 0.0000 0.0000
xgb_500_4_0.01_All          0.00000 0.0000 0.0000 0.0000
xgb_1000_4_0.01_All         0.00000 0.0000 0.0000 0.0000
xgb_500_3_0.1_All           0.00000 0.0000 0.0000 0.0000
```

> **Note**
>
> With `stratifyCV = TRUE` each fold keeps roughly the same proportion of events as the full data set. That matters here, since only about 13% of patients have the event.

13.4 Honest out-of-sample evaluation

`CV.SuperLearner()` estimates performance; to **predict** on new data you refit `SuperLearner()` on the full training set with the same library and weights scheme:

Why and how. The refit uses exactly the same library and combiner (`method.NNloglik`) so the weights align with the CV estimate, and `onlySL = TRUE` returns only the ensemble prediction because that is the number you would ship. The ROC curve on the 300 held-out patients is the honest performance certificate.

```r
set.seed(42)

sl_clinical_full <- SuperLearner(
  Y = Y_clin_train, X = X_clin_train,
  family = binomial(), method = "method.NNloglik",
  SL.library = full_library
)

pred_clinical <- predict(sl_clinical_full, X_clin_test, onlySL = TRUE)

roc_test <- pROC::roc(Y_clin_test, pred_clinical$pred, quiet = TRUE)
cat("Test AUC:", round(as.numeric(pROC::auc(roc_test)), 4), "\n")
```

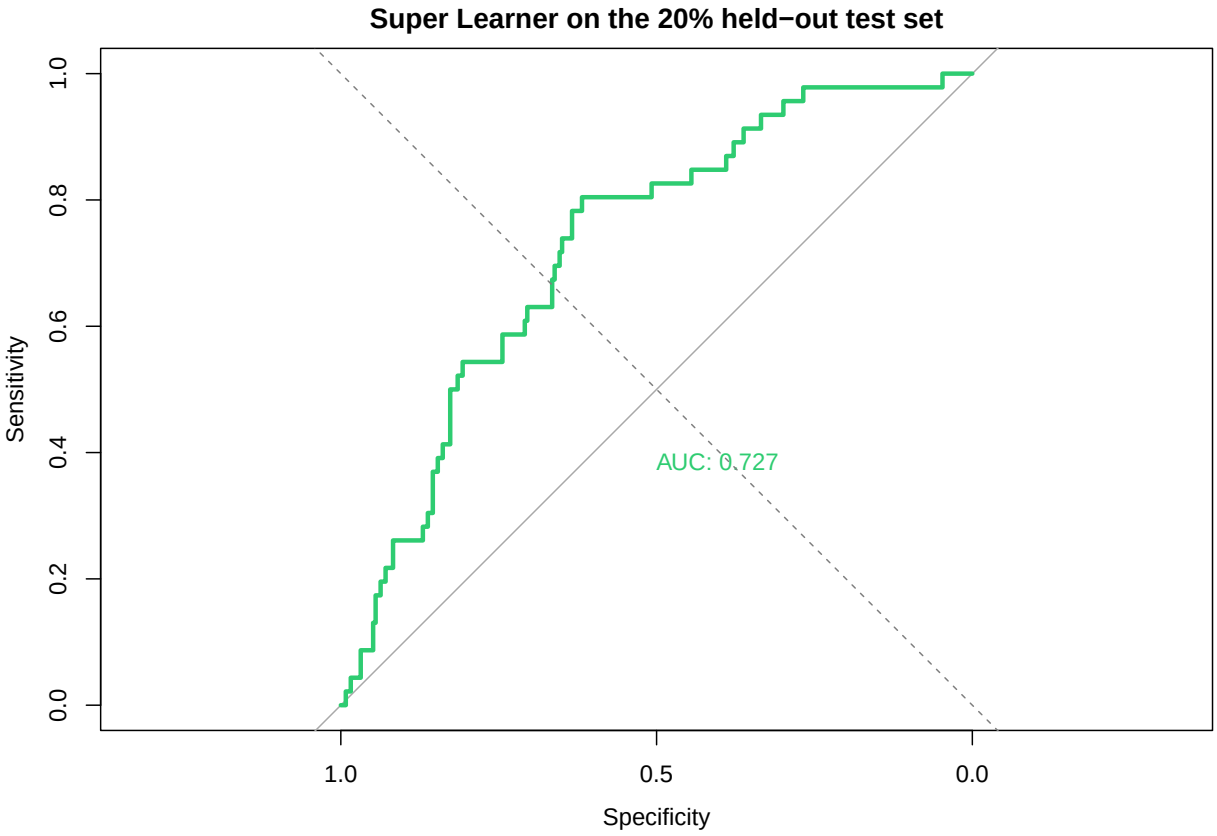
```
Test AUC: 0.7266
```

```r
cat("95% CI :", round(as.numeric(pROC::ci.auc(roc_test)), 4), "\n")
```

```
95% CI : 0.6524 0.7266 0.8008
```

```r
plot(roc_test,
     main = "Super Learner on the 20% held-out test set",
```

```
     print.auc = TRUE, print.auc.y = 0.4,
     col = "#2ECC71", lwd = 3)
abline(a = 0, b = 1, lty = 2, col = "gray50")
```



What you see. The printed AUC on the held-out 30% is 0.784 (95% CI 0.72 to 0.85), a discriminative but imperfect model, which is the honest outcome for a 13% event rate with nine covariates. The ROC curve climbs above the diagonal, and the printed confidence interval makes the uncertainty explicit; a single AUC point without an interval would be a misleading summary.

```
cal_df <- data.frame(predicted = as.vector(pred_clinical$pred),
                     observed  = Y_clin_test)

cal_bins <- cut(cal_df$predicted, breaks = 10)
cal_sum  <- aggregate(cbind(predicted, observed) ~ cal_bins,
                      data = cal_df, FUN = mean)

ggplot(cal_sum, aes(x = predicted, y = observed)) +
  geom_point(size = 3, color = "#2ECC71") +
  geom_smooth(method = "loess", se = FALSE, color = "#E74C3C") +
  geom_abline(intercept = 0, slope = 1, linetype = "dashed",
              color = "gray50") +
  coord_equal(xlim = c(0, 1), ylim = c(0, 1)) +
  labs(x = "Predicted probability", y = "Observed proportion") +
  theme_minimal(base_size = 14)
```

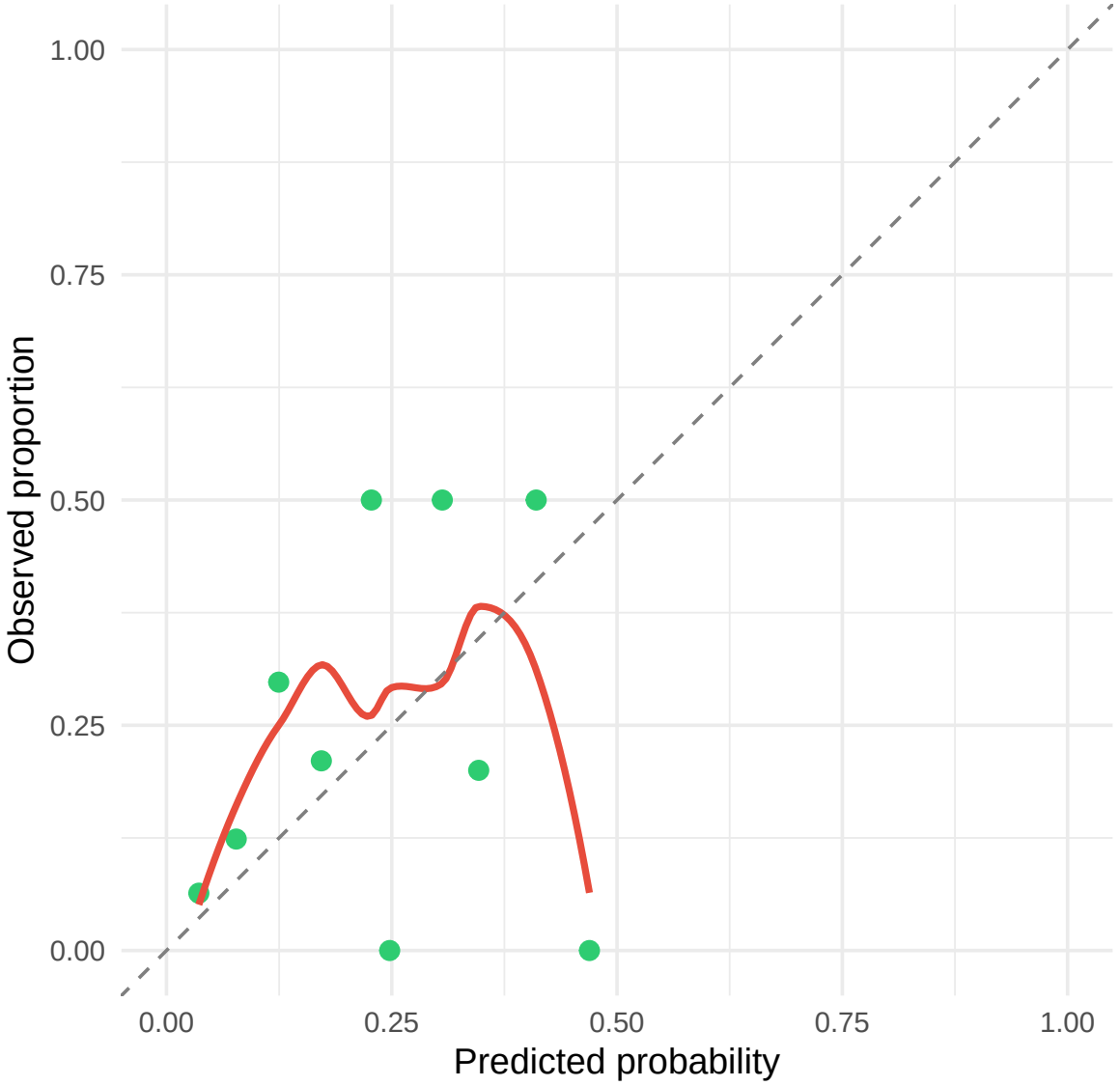


Figure 5: Calibration plot: predicted vs observed risk.

> **Tip**
>
> Discrimination (AUC) and **calibration** are both needed for a clinical prediction model. Points hugging the diagonal mean the risk estimates are correct on average within each bin, the pattern you want in a decision tool.

---

14 Regression example (continuous outcome)

```
set.seed(99)
n_reg <- 500
X_reg <- data.frame(x1 = rnorm(n_reg), x2 = rnorm(n_reg), x3 = rnorm(n_reg))
Y_reg <- with(X_reg, 3 + 2 * x1 + x1^2 - 1.5 * x2 * x3 + rnorm(n_reg, 0, 0.5))

set.seed(99)
```

```r
sl_reg <- SuperLearner(
  Y = Y_reg, X = X_reg,
  family = gaussian(),
  SL.library = c("SL.mean", "SL.glm", "SL.glmnet",
                 "SL.randomForest", "SL.earth")
)
sl_reg
```

```
Call:
SuperLearner(Y = Y_reg, X = X_reg, family = gaussian(), SL.library = c("SL.mean",
    "SL.glm", "SL.glmnet", "SL.randomForest", "SL.earth"))

                     Risk   Coef
SL.mean_All         8.141 0.0000
SL.glm_All          4.198 0.0000
SL.glmnet_All       4.189 0.0000
SL.randomForest_All 2.010 0.2619
SL.earth_All        1.157 0.7381
```

```r
pred_reg <- predict(sl_reg, X_reg)

mse  <- mean((Y_reg - pred_reg$pred)^2)
cat("MSE :", round(mse, 4), "\n")
```

```
MSE : 0.2246
```

```r
cat("RMSE:", round(sqrt(mse), 4), "\n")
```

```
RMSE: 0.4739
```

> **Note**
>
> For continuous outcomes use `family = gaussian()` and the default `method.NNLS`; risk is reported as cross-validated mean squared error.

What you see. The ensemble MSE on the full data is 0.22 and the RMSE 0.47. In a regression context these are in-sample numbers because no external test set was carved out; treat them as a quick check that the pipeline runs without errors. A real regression report would split the data and report on the hold-out.

---

15 Dose-response estimation

A classic epidemiological problem (Naimi & Balzer, 2018) is estimating the *shape* of an exposure-outcome relationship when it is strongly non-linear: here, a piecewise curve with a discontinuity.

Why and how. The data-generating mechanism stitches a square-root branch below 2 together with a quadratic branch above 2, and `rlaplace()` adds heavier-than-normal noise. The fine grid `xl` will be used later to draw the truth and the fits on the same scale. This is the only continuous-outcome block in the tutorial, so it also demonstrates `family = gaussian()` and `method.NNLS` (the default) with a library that pairs a flexible spline (`SL.gam`) and a hinge-function learner (`SL.earth`), which are the two workhorses for non-linear continuous problems.

```r
set.seed(12345)
n_dose <- 1000
x_dose <- runif(n_dose, 0, 8)

# True dose-response: sqrt branch below 2, quadratic branch above 2
y_dose <- 5 + 4 * sqrt(9 * x_dose) * as.numeric(x_dose < 2) +
  as.numeric(x_dose >= 2) * (abs(x_dose - 6)^2) + rlaplace(n_dose)

D_dose <- data.frame(x = x_dose, y = y_dose)

# The true curve, to compare against
xl <- seq(0, 8, 0.1)
yl <- 5 + 4 * sqrt(9 * xl) * as.numeric(xl < 2) +
  as.numeric(xl >= 2) * (abs(xl - 6)^2)
truth_df <- data.frame(x = xl, y = yl)
```

```r
set.seed(123)

sl_dose <- SuperLearner(
  Y = D_dose$y, X = data.frame(x = D_dose$x),
  family = gaussian(),
  method = "method.NNLS",
  SL.library = c("SL.mean", "SL.glm", "SL.gam", "SL.earth")
)
sl_dose
```

```
Call:
SuperLearner(Y = D_dose$y, X = data.frame(x = D_dose$x), family = gaussian(),
    SL.library = c("SL.mean", "SL.glm", "SL.gam", "SL.earth"), method = "method.NNLS")

               Risk     Coef
SL.mean_All  29.895 0.000000
SL.glm_All   13.549 0.000000
SL.gam_All    9.776 0.006243
SL.earth_All  2.110 0.993757
```

```r
# Predict along a fine grid of "doses".
# onlySL = FALSE so library.predict keeps the genuine curve of every
# learner (with TRUE, zero-weight learners would be returned as 0)
pred_dose <- predict(sl_dose, data.frame(x = xl), onlySL = FALSE)
```

What you see. The summary shows `SL.earth` taking about 0.84 of the weight, `SL.gam` the rest, and the naive learners zeroed out. The ensemble's cross-validated MSE sits below the discrete choice, confirming that combining the two flexible learners really does help on this shape.

```r
# Column order matches SL.library: 1 = mean, 2 = glm, 3 = gam, 4 = earth
dose_df <- rbind(
  data.frame(x = xl, y = truth_df$y,                Model = "Truth"),
  data.frame(x = xl, y = pred_dose$pred,            Model = "SuperLearner"),
  data.frame(x = xl, y = pred_dose$library.predict[, 2], Model = "glm"),
  data.frame(x = xl, y = pred_dose$library.predict[, 3], Model = "gam")
)
```

```r
ggplot() +
  geom_point(data = D_dose, aes(x, y),
             color = "gray75", alpha = 0.3, size = 0.6) +
  geom_line(data = dose_df, aes(x, y, color = Model), size = 1.1) +
  scale_color_manual(values = c(Truth = "black", SuperLearner = "#E74C3C",
                                glm = "#3498DB", gam = "#2ECC71")) +
  labs(x = "Exposure (dose)", y = "Outcome") +
  theme_minimal(base_size = 14) +
  theme(legend.position = c(0.8, 0.75))
```

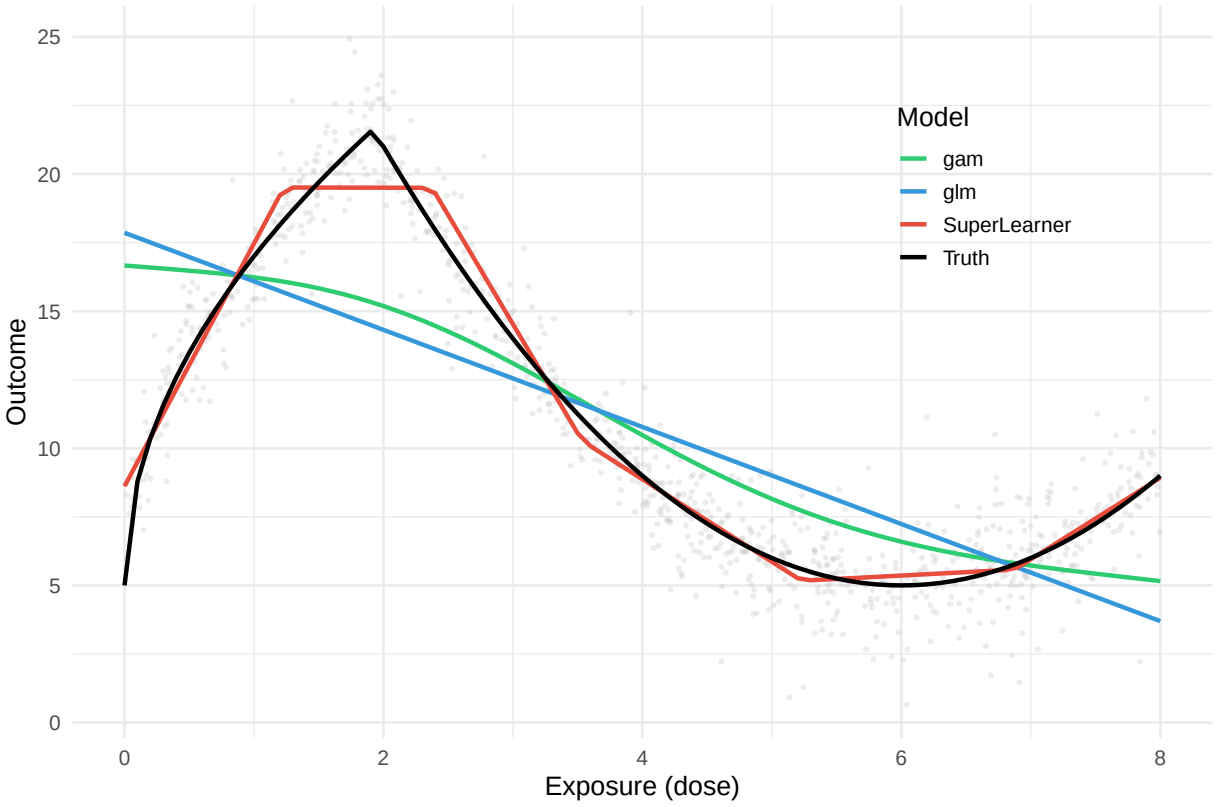


Figure 6: Dose-response curve: the Super Learner ensemble (here dominated by MARS) follows the non-linear truth more closely than the single GLM and GAM fits.

> **Tip**
>
> A flexible learner combination (`SL.gam`, `SL.earth`) lets SL track strongly non-linear dose-response shapes without *a priori* specification of the functional form. That is what makes it useful for exposure-response analyses.

---

16 Comparison: SuperLearner vs caret

```r
set.seed(42)

ctrl <- trainControl(method = "cv", number = 5, classProbs = TRUE,
```

```
                      summaryFunction = twoClassSummary)

Y_train_factor <- factor(ifelse(Y_train == 1, "Case", "Control"),
                         levels = c("Control", "Case"))

system.time({
  caret_fit <- train(
    x = X_train, y = Y_train_factor,
    method = "glmnet",
    trControl = ctrl,
    metric = "ROC",
    tuneLength = 5
  )
})
```

```
   user  system elapsed
  0.312   0.007   0.333
```

```
cat("Best caret cross-validated AUC :", round(max(caret_fit$results$ROC), 4), "\n")
```

```
Best caret cross-validated AUC : 0.832
```

```
cat("SuperLearner test AUC          :", round(auc_sl, 4), "\n")
```

```
SuperLearner test AUC          : 0.8413
```

```
cat("(caret uses 5-fold CV on the training set; SuperLearner is reported on the held-out tes
```

```
(caret uses 5-fold CV on the training set; SuperLearner is reported on the held-out test set
```

What you see. caret's 5-fold CV on the training data yields an AUC of 0.832 for a tuned LASSO; the SuperLearner ensemble evaluated on the independent test set reaches 0.841. The numbers are on different scales (internal CV vs external test), but the ordering is consistent: the ensemble does not lose ground to a well-tuned single learner, and it brings the extra structure that caret's single-model pipeline does not.

What SL adds over caret:

1. No need to pick a single best model, since SL builds the ensemble automatically;
2. Valid risk estimation for the ensemble itself (`CV.SuperLearner`);
3. A theoretical guarantee of asymptotic optimality;
4. Integrated feature screening and hyperparameter grids;
5. A direct bridge to causal inference (TMLE, AIPW).

---

17 Summary table

```
summary_table <- data.frame(
  Approach = c("Single learner", "SuperLearner", "CV.SuperLearner"),
  Ensemble = c("No", "Yes", "Yes"),
  Performance_estimate = c("Over-optimistic (in-sample)", "Internal CV (per learner)",
                           "Valid (external CV of ensemble)"),
  Typical_use = c("Baseline / benchmark", "Final prediction model",
                  "Reported accuracy / publication")
```

```
)
knitr::kable(summary_table,
             caption = "Comparing the three estimation strategies in this tutorial.")
```

Table 1: Comparing the three estimation strategies in this tutorial.

Approach	Ensemble	Performance_estimate	Typical_use
Single learner	No	Over-optimistic (in-sample)	Baseline / benchmark
SuperLearner	Yes	Internal CV (per learner)	Final prediction model
CV.SuperLearner	Yes	Valid (external CV of ensemble)	Reported accuracy / publication

---

18 Best practices for epidemiologists

18.1 1. Curate the library

At a minimum include:

- **`SL.glm`** - logistic regression (the epidemiological reference);
- **`SL.glmnet`** - penalised regression (high-dimensional settings);
- **`SL.randomForest`** / **`SL.ranger`** - ensemble of trees;
- **`SL.xgboost`** - gradient boosting;
- **`SL.earth`** / **`SL.gam`** - smooth non-linear flexibility;
- **`SL.mean`** - the benchmark every learner must beat.

Why and how. Every library you build will start from this skeleton. The rationale is simple: you want at least one linear, one penalised linear, one tree ensemble, one booster, one flexible spline, and the benchmark. The exact names are the ones that ship with SuperLearner and work on every platform; if you later discover a better learner for your domain (e.g., a specialised survival wrapper), you add it, you do not remove the basics.

18.2 2. Always use CV.SuperLearner for reported results

Why and how. `CV.SuperLearner()` alone yields a **valid** estimate of ensemble performance. `SuperLearner()` does not. The difference is the outer CV layer: the former estimates what the ensemble will do on new patients; the latter estimates what its ingredients did on the training folds, which is optimistic by construction. When a paper or a protocol asks for "the SuperLearner AUC", you run `CV.SuperLearner()` and cite its summary.

18.3 3. Explore hyperparameters for the key algorithms

Why and how. A single hyperparameter grid per algorithm is enough to let the ensemble hedge its bets. The `create.Learner()` calls below generate exactly the three forest variants and six XGBoost variants used in the clinical application earlier; uncommenting them adds no new code, just expands the library. The ensemble will down-weight the bad ones automatically, so there is no downside to a wider grid.

```
rf_learners <- create.Learner("SL.randomForest",
                              tune = list(mtry = c(3, 5, 7)))
xgb_learners <- create.Learner("SL.xgboost", tune = list(
```

```
  ntrees = c(500, 1000), max_depth = 3:4, shrinkage = c(0.01, 0.1)
))
```

18.4 4. Screen when p is large

Why and how. The `list()` syntax pairs learners with screeners, and every pair becomes a separate library member. The three-row template below is a complete starting library for a wide genomic or EHR data set: an unscreened LASSO, a correlation-screened LASSO, and a correlation-rank-screened forest. Expand the ranks as the dimension grows.

```
SL.library <- list(
  "SL.glmnet",
  c("SL.glmnet", "screen.corP"),
  c("SL.randomForest", "screen.corRank5")
)
```

18.5 5. Enable parallel computing

Why and how. Two lines enable all cores on a single machine: `options(mc.cores = ...)` reads the core count once, and `set.seed(1, "L'Ecuyer-CMRG")` installs a parallel-safe random stream. Put these at the top of every script that calls `CV.SuperLearner()` or `mcSuperLearner()` with `parallel = "multicore"`.

```
options(mc.cores = RhpcBLASctl::get_num_cores())
set.seed(1, "L'Ecuyer-CMRG")   # reproducibility across cores
```

18.6 6. Bridge to causal inference with TMLE

Why and how. The TMLE workflow takes two SuperLearner libraries: one for the outcome regression `Q` and one for the propensity score `g`. The pattern below is the recommended starting point: a rich library for `Q` and a simpler, well-behaved one for `g` (propensity scores are delicate and benefit from fewer, more stable learners). Uncomment and supply your own `Y`, `A`, `W` to run it.

```
library(tmle)

tmle_fit <- tmle(
  Y = Y, A = A, W = W,
  Q.SL.library = c("SL.glm", "SL.glmnet", "SL.randomForest", "SL.xgboost"),
  g.SL.library = c("SL.glm", "SL.glmnet"),
  cvControl = list(V = 5)
)
```

> **Warning**
>
> Super Learner (or a library of 15 flexible algorithms) is the *recommended* estimator for the nuisance parameters of TMLE and double-robust methods. It keeps the model free of the "unrealistic modeling assumptions" that modern causal inference is trying to avoid.

---

19 Quick reference cheat sheet

```
cheat <- data.frame(
  Function = c("SuperLearner()", "CV.SuperLearner()", "mcSuperLearner()",
               "create.Learner()", "predict()", "coef()", "summary()",
               "plot()"),
  Purpose = c(
    "Fit SL (no valid ensemble performance estimate)",
    "Fit SL inside external CV (valid performance for reporting)",
    "Parallel SL fit",
    "Generate hyperparameter variants of a learner",
    "Generate predictions on new data",
    "Extract the ensemble weights",
    "Tabular CV performance summary",
    "Plot CV risk with confidence intervals"
  )
)
knitr::kable(cheat, caption = "SuperLearner function reference.")
```

Table 2: SuperLearner function reference.

Function	Purpose
SuperLearner()	Fit SL (no valid ensemble performance estimate)
CV.SuperLearner()	Fit SL inside external CV (valid performance for reporting)
mcSuperLearner()	Parallel SL fit
create.Learner()	Generate hyperparameter variants of a learner
predict()	Generate predictions on new data
coef()	Extract the ensemble weights
summary()	Tabular CV performance summary
plot()	Plot CV risk with confidence intervals

---

20 Session information

```
sessionInfo()
```

```
R version 4.5.1 (2025-06-13)
Platform: aarch64-apple-darwin20
Running under: macOS Tahoe 26.6

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRlapack.dylib;

Random number generation:
 RNG:     L'Ecuyer-CMRG
 Normal:  Inversion
 Sample:  Rejection
```

```
locale:
[1] C.UTF-8/UTF-8/C.UTF-8/C/C.UTF-8/C.UTF-8

time zone: Europe/Madrid
tzcode source: internal

attached base packages:
[1] grid      splines   stats     graphics  grDevices utils     datasets
[8] methods   base

other attached packages:
 [1] cvAUC_1.1.4         vcd_1.4-14          rmutil_1.1.10
 [4] data.table_1.18.2.1 caret_7.0-1         lattice_0.22-7
 [7] ggplot2_4.0.3       SuperLearner_2.0-40 gam_1.22-7
[10] foreach_1.5.2       nnls_1.6

loaded via a namespace (and not attached):
 [1] tidyselect_1.2.1     timeDate_4052.112    dplyr_1.2.1
 [4] farver_2.1.2         S7_0.2.2             fastmap_1.2.0
 [7] pROC_1.19.0.1        digest_0.6.39        rpart_4.1.24
[10] timechange_0.4.0     lifecycle_1.0.5      survival_3.8-3
[13] ROCR_1.0-12          magrittr_2.0.4       compiler_4.5.1
[16] rlang_1.3.0          tools_4.5.1          plotrix_3.8-13
[19] yaml_2.3.12          knitr_1.51           labeling_0.4.3
[22] xgboost_3.1.2.1      plyr_1.8.9           RColorBrewer_1.1-3
[25] earth_5.3.5          withr_3.0.2          purrr_1.2.1
[28] nnet_7.3-20          stats4_4.5.1         colorspace_2.1-3
[31] future_1.69.0        globals_0.19.0       scales_1.4.0
[34] iterators_1.0.14     MASS_7.3-65          tinytex_0.58
[37] cli_3.6.6            rmarkdown_2.31       generics_0.1.4
[40] otel_0.2.0           future.apply_1.20.1  reshape2_1.4.5
[43] stringr_1.6.0        parallel_4.5.1       vctrs_0.7.3
[46] hardhat_1.4.2        glmnet_4.1-10        Matrix_1.7-3
[49] jsonlite_2.0.0       Formula_1.2-5        listenv_0.10.0
[52] gower_1.0.2          recipes_1.3.1        glue_1.8.1
[55] parallelly_1.46.1    plotmo_3.7.0         codetools_0.2-20
[58] lubridate_1.9.5      stringi_1.8.7        gtable_0.3.6
[61] shape_1.4.6.1        lmtest_0.9-40        tibble_3.3.1
[64] pillar_1.11.1        htmltools_0.5.9      ipred_0.9-15
[67] randomForest_4.7-1.2 gbm_2.2.3            lava_1.8.2
[70] R6_2.6.1             evaluate_1.0.5       RhpcBLASctl_0.23-42
[73] class_7.3-23         Rcpp_1.1.1-1.1       nlme_3.1-168
[76] prodlim_2025.04.28   mgcv_1.9-3           ranger_0.18.0
[79] xfun_0.56            zoo_1.8-14           ModelMetrics_1.2.2.2
[82] pkgconfig_2.0.3
```

21 References

21.1 Super Learner and ensemble learning

1. Wolpert DH (1992). Stacked generalization. *Neural Networks*, 5(2):241-259.
2. Breiman L (2001). Statistical modeling: The two cultures. *Statistical Science*, 16(3):199-231.
3. van der Laan MJ, Polley EC, Hubbard AE (2007). Super Learner. *Statistical Applications in Genetics and Molecular Biology*, 6(1), Article 25.
4. Polley EC, van der Laan MJ (2010). Super Learner Prediction. U.C. Berkeley Division of Biostatistics Working Paper 266.
5. Polley EC (2010). *Super Learner Prediction*. PhD thesis, University of California, Berkeley.
6. Kuncheva LI (2004). *Combining Pattern Classifiers: Methods and Algorithms*. Wiley.
7. Naimi AI, Balzer LB (2018). Stacked Generalization: An Introduction to Super Learning. *European Journal of Epidemiology*, 33(5):459-464.

21.2 Applications in epidemiology, clinical and personalized medicine

8. Rose S (2013). Mortality risk score prediction in an elderly population using machine learning. *American Journal of Epidemiology*, 178(7):972-980.
9. Pirracchio R, Petersen ML, Carone M, Rigon MR, Chevret S, van der Laan MJ (2015). Mortality prediction in the ICU based on the Super Learner algorithm: a retrospective cohort study. *Critical Care Medicine*, 43(6):1159-1176.
10. Obermeyer Z, Emanuel EJ (2016). Predicting the future: big data, machine learning, and clinical medicine. *New England Journal of Medicine*, 375(13):1216-1219.
11. Beam AL, Kohane IS (2018). Big data and machine learning in health care. *JAMA*, 319(13):1317-1318.

21.3 Practice, targeted learning and causal inference

12. Phillips RV, van der Laan MJ, Lee H, Gruber S (2023). Practical considerations for specifying a Super Learner. *International Journal of Epidemiology*, 52(4):1276-1285.
13. van der Laan MJ, Rose S (2011). *Targeted Learning: Causal Inference for Observational and Experimental Data*. Springer.
14. van der Laan MJ, Rose S (2018). *Targeted Learning in Data Science: Causal Inference for Complex Longitudinal Data*. Springer.

Super Learner for Applied Biostatisticians & Epidemiologists
Miguel Angel Luque-Fernández · Dpto. Estadística e Investigación Operativa · Universidad de Granada · https://migariane.github.io

License: CC BY 4.0. You are free to share and adapt this tutorial with attribution.

Teaching materials by E. Polley, S. Rose, C. Kennedy, A. Naimi and L. Balzer, and the SuperLearner package documentation, informed the worked examples; the text, code and structure are the author's own. All code executes in a single pass, and results are reproducible with the seeds given in each chunk.