Introduction to Machine Learning

Working Group "Computational Statistics" – Bernd Bischl et al.

Code demo for Nested Resampling

In this code demo we

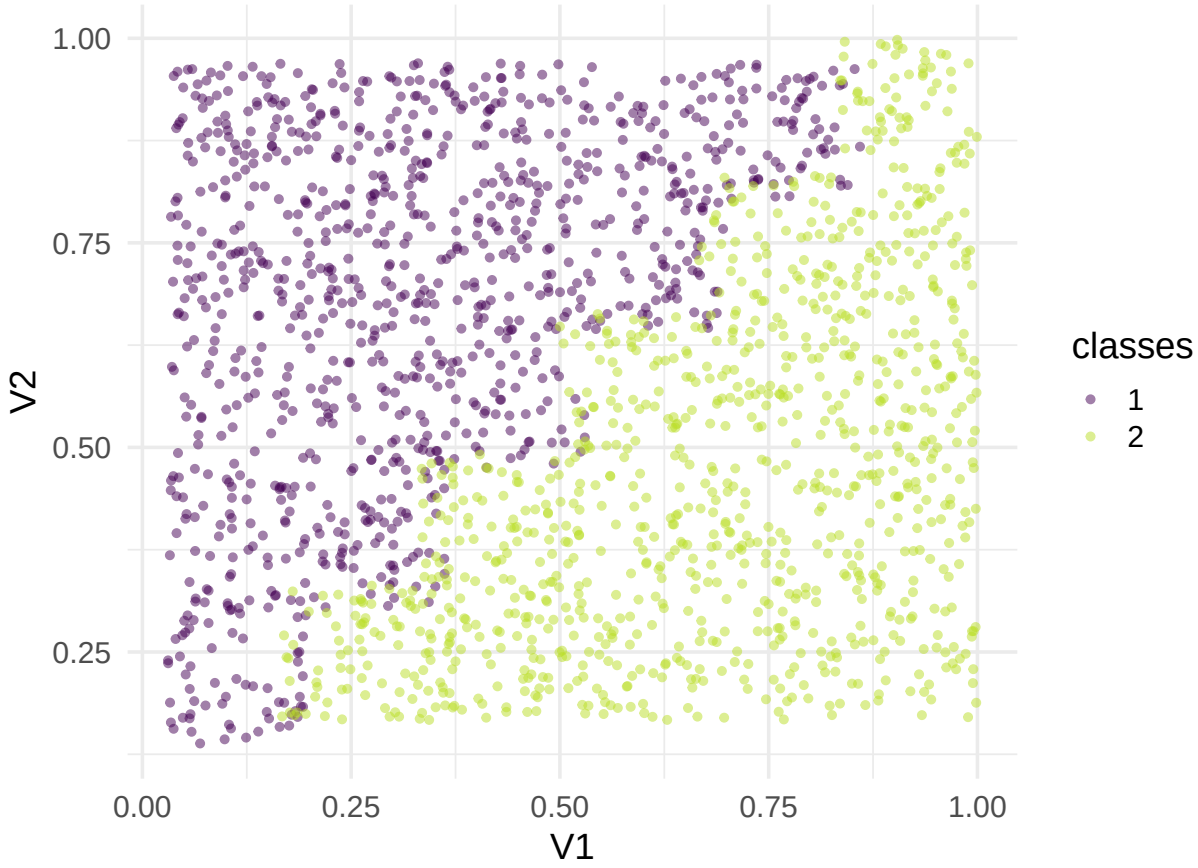
- compare resampling and nested resampling,
- see how we can tune hyperparemeters,
- learn how to evaluate the generalization error in this scheme.

Setup

Firstly we generate stairs-like data:

```
library(mlr3)
library(mlr3learners)
library(mlr3tuning)
library(mlbench)
library(ggplot2)

# function to simulate stairs data
stairs <- function(n, steps) {
  xs <- as.data.frame(matrix(runif(2 * n, max = 1 / steps, min = 0), ncol = 2))
  num_blocks <- steps * (steps - 1) / 2
  xs$block <- sample(1:num_blocks, n, replace = T)

  block <- 1
  for (i in 1:(steps - 1)) {
    for (j in 1:i) {
      xs[xs$block == block, 1:2] <- t(t(xs[xs$block == block, 1:2]) + c(
        i / steps,
        j / steps
      ))

      block <- block + 1
    }
  }
  return(xs[, 1:2])
}

# function to generate two stairs which share an overlap
two_stairs <- function(n, steps, overlap) {
  v1 <- stairs(n / 2, steps)
  v1$V2 <- v1$V2 - 1 / steps + overlap
  v1$V1 <- v1$V1 - overlap
  v1[, c(1, 2)] <- v1[, c(2, 1)] # "mirror" the stairs
  v1$classes <- factor("1")
  v2 <- stairs(n / 2, steps)
  v2$classes <- factor("2")

  return(rbind(v1, v2))
}

set.seed(123)
```

```
sim_stairs <- two_stairs(2000, 6, 0.03)
ggplot(sim_stairs, aes(x = V1, y = V2)) + geom_point(aes(color = classes),
                                                     alpha = 0.5)
```



Now we want to find the optimal hyperparameter k from a set of candidates for a knn model. As we have already learned we can use CV for this task (We use the `knn_cv()` function from the resampling code demo):

```
k_candidates <- seq(1, 30, 3)
mmces <- NULL
for (k in k_candidates) {
  set.seed(10)

  if (is.null(mmces)) {
    mmces <- knn_cv(sim_stairs, "classes", 10, k)
  } else {
    mmces <- cbind(mmces, knn_cv(sim_stairs, "classes", 10, k))
  }
}

results <-
  as.data.frame(t(apply(mmces, 2, mean)))
colnames(results) <- as.character(k_candidates)
results
```

```
##         1      4      7     10     13     16     19     22      25     28
```

```
## 1 0.0575 0.0505 0.0515 0.053 0.051 0.05 0.053 0.051 0.0505 0.049
```

So from this we would choose $k = 28$, since it has the smallest MSE[1]. But how can we estimate the generalization error? A naive approach would be to average the minimal MSE of all folds:

```
apply(mmces, 1, min)
```

```
##  [1] 0.055 0.030 0.060 0.045 0.040 0.030 0.040 0.030 0.040 0.070
```

```
mean(apply(mmces, 1, min))
```

```
## [1] 0.044
```

From the lecture we know that this gives us a biased estimate. Therefore we use nested resampling with the following options:

- 10 outer CV loops
- 10 inner CV loops
- 10 Candidates

Basically, we want to do two things: find the optimal hyperparameter and estimate the generalization error of models tuned in this way. We can not do two operations in one CV and therefore use two nested CVs, thus we apply **nested resampling**.

```
kknn_nr <- function(data, target, outer_folds = 3, inner_folds = 4, k_candidates,
                    inform = FALSE) {
  # data: whole data set including the target variable
  # target: string indicating the target variable of the task
  # outer_folds: the amount of folds in the outer loop of the nested resampling
  # inner_folds: the amount of folds in the inner loop of the nested resampling
  # max_depths: the potential values for k from which we want to select
  #              the optimal
  # inform: boolean controling if the user is getting informed about the
  #         progress of the procedure or not


  # counter for inform functionality
  counter <- 0

  # indices for the outer loops
  outer_indices <- sample(
    x = seq(1, nrow(data), by = 1),
    size = nrow(data), replace = FALSE
  )
  index_outer_mat <- matrix(
    data = outer_indices, byrow = TRUE,
    nrow = outer_folds
  )
```

[1] NB: If your tuning procedure yields an optimum at the end of the range of the parameter you are tuning, you should extend the range...

```r
# frame to store the winner and its test-gen_error from all outer folds
winner_cv <- as.data.frame(matrix(0, nrow = outer_folds, ncol = 2))
colnames(winner_cv) <- c("k", "gen_error")

for (i in 1:outer_folds) {

  # split in validation and data for the inner loop
  test_data <- data[index_outer_mat[i, ], ]
  inner_data <- data[ -index_outer_mat[i, ], ]

  # frame to store the CV-gen_errors for each candidate
  candidate_gen_error <-
    as.data.frame(matrix(0, nrow = length(k_candidates), ncol = 2))
  colnames(candidate_gen_error) <- c("k", "gen_error")

  # calculate gen_error for each of the candidates via CV
  for (l in 1:length(k_candidates)) {
    inner_indices <-
      sample(
        x = seq(1, nrow(inner_data), by = 1),
        size = nrow(inner_data), replace = FALSE
      )

    # index matrix for inner folds
    index_inner_mat <- matrix(
      data = inner_indices,
      byrow = TRUE, nrow = inner_folds
    )

    # storage for CV errors for one candidate
    storage_inner_cv <- numeric(inner_folds)

    for (j in 1:inner_folds) {

      # data split in validation and train data
      eval_data <- inner_data[index_inner_mat[j, ], ]
      train_data <- inner_data[ -index_inner_mat[j, ], ]

      # model
      task <- TaskClassif$new(
        id = "train_data",
        backend = train_data,
        target = target
      )
      learner <- lrn("classif.kknn", k = k_candidates[l])
      learner$train(task)
      # inform user about progress
      counter <- counter + 1
      if (inform) {
        print(paste0(
          "model: ", counter,
          " inner fold: ", j, " outer fold: ", i
        ))
```

```
        }
        # store gen_error estimates from test on validation data
        storage_inner_cv[j] <- learner$predict_newdata(
          newdata = eval_data
        )$score()[[1]]
      }

      # CV gen_error for candidate l
      candidate_gen_error[l, "gen_error"] <- mean(storage_inner_cv)
      candidate_gen_error[l, "k"] <- k_candidates[l]
    }

    # get gen_error for best candidate in this outer_fold
    # in case of equally good Candidates, choose the first one
    best_candidate <- candidate_gen_error[
      which(candidate_gen_error$gen_error ==
        min(candidate_gen_error$gen_error)), "k"
    ][1]
    winner_cv[i, "k"] <- best_candidate

    # model
    task <- TaskClassif$new(
      id = "inner_data",
      backend = inner_data,
      target = target
    )
    learner <- lrn("classif.kknn", k = best_candidate)
    learner$train(task)
    # store gen_error estimates from test on test data
    winner_cv[i, "gen_error"] <- learner$predict_newdata(
      newdata = test_data
    )$score()[[1]]
  }
  return(winner_cv[order(winner_cv$gen_error), ])
}
```

Run it and check the results of the outer CV:

```
# let's run it
set.seed(1337)
result_nr <- kknn_nr(
  data = sim_stairs, target = "classes", outer_folds = 10, inner_folds = 10,
  k_candidates = k_candidates, inform = FALSE
)
# order results by gen_error
result_nr
```

```
##     k gen_error
## 5  28     0.035
## 7  28     0.040
## 9   7     0.040
## 10 22     0.045
## 8  16     0.050
```

```
## 3  13     0.055
## 4  19     0.060
## 1  19     0.065
## 6   4     0.070
## 2  22     0.080
```

```
mean(result_nr$gen_error)
```

```
## [1] 0.054
```

So we get a bigger generalization error than the one we got for the simple CV. So how can we check what the real generalization error for our best candidate k is? Neat thing about the stairs data: we can simulate arbitrarily many new unseen data points, s.t. we can assess the real generalization error.

```
set.seed(1337)
unseen_data <- two_stairs(100000, 6, 0.03)

task <- TaskClassif$new(id = "stairs", backend = sim_stairs, target = "classes")
learner <- lrn("classif.kknn", k = 28)
learner$train(task = task)
learner$predict_newdata(newdata = unseen_data)$score()
```

```
## classif.ce
##     0.0533
```

As we see with the simple CV we arrived to an overly optimistic estimate (overtuning effect), while a more realistic estimate could be achieved with the nested resampling algorithm.

mlr3 implementation

```
# show only warning messages (the number of info messages is quite large
# for our nested resampling design)
lgr::get_logger("mlr3")$set_threshold("warn")

task <- TaskClassif$new(
  id = "stairs",
  backend = sim_stairs,
  target = "classes"
)

learner <- lrn("classif.kknn")
# define the the inner cv
resampling <- rsmp("cv", folds = 10)
measures <- msr("classif.ce")

# replicate the k_candidates test design
param_set <- paradox::ParamSet$new(
  params = list(paradox::ParamInt$new("k",
    lower = min(k_candidates),
    upper = max(k_candidates)
```

```
  ))
)
# we can use a grid design since the candidates are uniformly spaced
design <- paradox::generate_design_grid(param_set,
  resolution = length(k_candidates)
)
dt <- tnr("design_points", design = design$data)
# we want to test every candidate
terminator <- term("evals", n_evals = length(k_candidates))

at <- AutoTuner$new(
  learner = learner,
  resampling = resampling,
  measures = measures,
  tune_ps = param_set,
  terminator = terminator,
  tuner = dt
)

# define the outer cv
resampling_outer <- rsmp("cv", folds = 10)
# do nested resampling
rr <- resample(task = task, learner = at, resampling = resampling_outer)
```

Get the tuning results for the outer loops

```
# tune results
rr$score()[, c("iteration", "classif.ce")]
```

```
##     iteration classif.ce
##  1:         1      0.045
##  2:         2      0.060
##  3:         3      0.050
##  4:         4      0.070
##  5:         5      0.085
##  6:         6      0.045
##  7:         7      0.045
##  8:         8      0.045
##  9:         9      0.045
## 10:        10      0.045
```

```
rr$aggregate()
```

```
## classif.ce
##     0.0535
```