Module 9: Recommended Exercises (Solution)

TMA4268 Statistical Learning V2025

Kenneth Aase, Sara Martino, Stefanie Muff
Department of Mathematical Sciences, NTNU

March 19, 2025

We recommend you to work through the Section 8.3.4 in the course book (Lab: Boosting)

Problem 1

Both **bagging** and **boosting** are *ensemble* methods.

Solution:

- What is an ensemble method?

A statistical learning model that combines the output from many statistical learning models, to perform better than any of the models individually.

- What are the main conceptual differences between bagging and boosting?

Bagging: Bootstrap aggregation. Based on building many models *independently of each other*, and combining the results. Boosting: Based on building many models *sequentially*, where we let each model improve the fit for the parts of the problem that the model in the previous iteration struggled with.

Problem 2

Consider the Gradient Tree Boosting Algorithm (given on slide 35 in this week's lectures).

Explain the meaning behind each of the following symbols. If relevant, also explain what the impact of changing them will be, and explain how one might find or choose them.

Solution:

- $L(.)$

This is the loss function we are trying to minimize when fitting the base learners in each iteration. Our choice of loss function will of course depend on the type of data we are modelling (i.e., regression vs. classification), but the choice can also be highly impactful on the robustness of our method - for example, the quadratic loss function might place too much weight on extreme observations.

- $f_m(.)$

This is our gradient boosted tree at the m^{th} iteration of the algorithm. It is found by fitting a tree to the pseudo-residuals r_{im}, and within the terminal regions of that tree, adding a "correction" (γ_{jm} for terminal region j) to the current f_{m-1} in a way that minimizes the loss function L. For details on how to fit the individual trees, see the previous module.

- M

This is the number of trees (base learners) that we fit. This is a hyperparameter that should be tuned with e.g. cross-validation, or chosen adaptively with some stopping criterion. Too low M leads to underfitting (we don't have enough base learners to capture the structure in the data), while too high M leads to overfitting.

- J_m

This is the number of terminal regions of the tree (base learner) we fit in iteration m. J_m will be determined by the procedure we use to fit each base learner, but we commonly constrain what the maximum value of J_m can be. This constraint (the maximum value of J_m) is a hyperparameter that should be tuned with e.g. cross-validation. The lower we set the maximum allowed J_m to be, the less complex each tree will be - i.e. the simpler the base learners in the ensemble will be. And vice versa. More complicated base learners will also mean a higher computational cost per iteration.

Problem 3 - Learning rate

The algorithm mentioned in Problem 2 does not include the so-called *learning rate* parameter ν. In the context of boosting methods, what is the interpretation behind this parameter, and how would one choose it? If you were to include the parameter in the Gradient Tree Boosting Algorithm, which equation(s) in the algorithm would you change and how?

Solution:

The learning rate $0 < \nu < 1$ is a factor that we use to scale the base learner at each iteration of the boosting algorithm. Thus, the lower its value, the less impact each base learner will have on the ensemble. Empirically, this parameter is often found to be crucial in determining the accuracy of the model. The smaller the value of this parameter, the higher we need M to be, since we slow down the learning. ν can be chosen by cross-validation, or set to a fixed (small) value if M is chosen adaptively (see above).

The learning rate would be added to step 2d) on lecture slide 35 (Algorithm 10.3 in "The Elements of Statistical Learning"), so that it would read

$$\text{Update} \quad f_m(x) = f_{m-1}(x) + \nu \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm}).$$

Problem 4 - Boosting methods

In the code chunk below we simulate data meant to mimic a genomic data set for a set of markers (locations in the human genome) that underlie a complex trait (such as height, or whether you have cancer or not). You don't need to understand the details of the simulation, but the main takeaways are

- y is a continuous response, representing some biological trait (for example height) depending on both genetic (the predictors) and environmental (here included as random noise) factors.
- We have many more predictors (genetic markers) M than observations (individuals) N.
- The predictors have a complicated structure: most of them have a very small effect on y, but a few of them have a larger effect.

```
# install.packages("DirichletReg")
# install.packages("magrittr")
library(DirichletReg)
library(magrittr)

# Simulate
set.seed(1)
N <- 1000 # number of individuals
M <- 10000 # number of genotyped markers
p <- rbeta(n = M, shape1 = 2.3, shape2 = 7.4) # Allele frequencies
p %<>% ifelse(. < 0.5, ., 1 - .)
```

```r
X <- sapply(p, rbinom, n = N, size = 2) # genotype matrix

effect_weights <- rdirichlet(M, c(0.9, 0.08, 0.015, 0.005))

marker_effects <- apply(effect_weights, 1, function(w) {
  effects <- c(0, sapply(sqrt(c(1e-4, 1e-3, 1e-2)), rnorm, n = 1, mean = 0))
  sum(w * effects)
})

genetics <- as.vector(X %*% marker_effects)
environment <- rnorm(N, sd = sqrt(var(genetics) / 2))
y <- genetics + environment
dat <- cbind(y, as.data.frame(X))

# Looking at a few entries in the data:
dat[1:6,1:20]
```

```
##            y V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19
## 1 -0.6553370  0  1  0  0  0  0  1  0  0   0   0   0   1   1   1   1   1   2   2
## 2 -0.6780554  1  0  0  2  0  2  1  0  2   0   0   0   0   0   0   0   0   1   1
## 3 -0.2311866  0  1  0  0  1  0  0  0  1   1   0   0   1   0   1   0   0   1   1
## 4 -1.0386650  2  0  0  1  0  1  0  0  1   0   1   0   1   0   0   0   0   0   0
## 5 -0.4776310  0  1  1  1  0  0  0  2  2   2   2   0   2   1   1   0   1   1   0
## 6 -0.4600658  0  0  1  1  1  0  0  1  1   0   0   0   0   0   0   0   0   1   1
```

We will now try to predict y using various tree boosting methods, adapted from code provided in https://bradleyboehmke.github.io/HOML/gbm.html.

a) A basic GBM

We first implement a standard gradient boosting tree using the package `gbm`. To make sure everything works, we run a very small ($M = 3$) model below.

```r
# install.packages("dplyr")
library(dplyr)    # for general data wrangling needs

# Modeling packages
# install.packages("gbm")
# install.packages("h2o")
# install.packages("xgboost")
library(gbm)      # for original implementation of regular and stochastic GBMs
library(h2o)      # for a java-based implementation of GBM variants
library(xgboost)  # for fitting extreme gradient boosting

gbm1 <- gbm(
  formula = y ~ .,
  data = dat,
  distribution = "gaussian",  # SSE loss function
  n.trees = 3,
  shrinkage = 0.1,
  interaction.depth = 7,
  n.minobsinnode = 10,
  cv.folds = 10
)
```

We note that running the above code block is very slow - even for a *very* low number of trees $M = 3$ (M is typically in the thousands). How could we modify the algorithm to be less computationally intensive?

Solution:

We could for example:

- We could reduce the allowed complexity of each base learner (reduce the interaction depth).
- We could subsample observations (like in bagging) and/or predictors (like in random forest). In other words, go for a stochastic gradient boosting approach.
- Change our validation approach. The above code does a 10-fold cross validation for each M up to 3. To save on computations, we could go to 5-fold CV, or even a training/test set approach.

b) Stochastic GBMs

We now move on to stochastic gradient boosting tree models for the same data set, as implemented in the `h2o` package. Explain what is done in the below code (`?h2o.grid`, `h2o.gbm` and https://bradleyboehmke.github.io/HOML/gbm.html might be helpful).

```
# Set maximum memory usage
h2o.init(max_mem_size = "14g")

# refined hyperparameter grid
hyper_grid <- list(
  sample_rate = c(0.2, 0.5),              # row subsampling
  col_sample_rate = 0.1,          # col subsampling for each split
  col_sample_rate_per_tree = 0.1,  # col subsampling for each tree
  min_rows = 10,
  learn_rate = 0.05,
  max_depth = c(3, 5, 7),
  ntrees = 10000
)

# random grid search strategy
search_criteria <- list(
  strategy = "RandomDiscrete",
  stopping_metric = "mse",
  stopping_tolerance = 0.001,
  stopping_rounds = 10,
  max_runtime_secs = 20*60
)

# perform grid search
grid <- h2o.grid(
  algorithm = "gbm",
  y = "y",
  training_frame = as.h2o(dat),
  hyper_params = hyper_grid,
  nfolds = 5,
  search_criteria = search_criteria,
  seed = 123,
  parallelism = 0
)

# collect the results and sort by our model performance metric of choice
grid_perf <- h2o.getGrid(
```

```
  grid_id = grid@grid_id,
  sort_by = "mse",
  decreasing = FALSE
)

grid_perf
```

Solution:

We run a Stochastic gradient boosting tree with subsampling of both rows (observations) and columns (predictors). In each tree we subsample 10% of the predictors, and 10% of these are subsampled when creating each split within a tree. So, $10000 \cdot 0.1 \cdot 0.1 = 100$ predictors are considered at each split in the trees. The number of observations to sample for each tree (20% or 50%) is chosen by 5-fold cross validation, as is the allowed complexity of each tree (`max_depth`).

We fix the learning rate to a low value of $\nu = 0.05$, and choose M adaptively: we either stop the model if the (cross-validated) MSE has not improved by 0.001 over the last 10 iterations, if we reach 10000 iterations or if the runtime exceeds twenty minutes.

c) XGboost

Below we also provide code for applying cross-validated XGBoost to the same data set. Expand this code to perform a search of the hyperparameter space, similar to b).

```
set.seed(123)
xgb <- xgb.cv(
  data = X,
  label = y,
  nrounds = 6000,
  early_stopping_rounds = 50,
  nfold = 5,
  params = list(
    eta = 0.05,
    max_depth = 3,
    min_child_weight = 3,
    subsample = 0.2,
    colsample_bytree = 0.1),
  verbose = TRUE
)

# minimum test CV RMSE
min(xgb$evaluation_log$test_rmse_mean)
```

Solution:

Below we do a hyperparameter tuning, again adapted from https://bradleyboehmke.github.io/HOML/gbm.html. Note that XGBoost has *many* parameters that can be tuned on a much finer grid than what is done here.

```
# hyperparameter grid
hyper_grid <- expand.grid(
  eta = 0.05,
  max_depth = c(3, 5, 7),
  min_child_weight = 3,
  subsample = c(0.2, 0.5),
  colsample_bytree = 0.1,
  rmse = 0,          # a place to dump RMSE results
  trees = 0          # a place to dump required number of trees
```

```
)

# grid search
for(i in seq_len(nrow(hyper_grid))) {
  set.seed(123)
  m <- xgb.cv(
    data = X,
    label = y,
    nrounds = 6000,
    early_stopping_rounds = 50,
    nfold = 5,
    verbose = TRUE,
    params = list(
      eta = hyper_grid$eta[i],
      max_depth = hyper_grid$max_depth[i],
      min_child_weight = hyper_grid$min_child_weight[i],
      subsample = hyper_grid$subsample[i],
      colsample_bytree = hyper_grid$colsample_bytree[i]
    )
  )
  hyper_grid$rmse[i] <- min(m$evaluation_log$test_rmse_mean)
  hyper_grid$trees[i] <- m$best_iteration
}

hyper_grid %>%
  filter(rmse > 0) %>%
  arrange(rmse) %>%
  glimpse()

# minimum test CV RMSE
min(xgb$evaluation_log$test_rmse_mean)
```

d) Model evaluation

Finally, experimenting with the code from a), b) and c), what is the best model you are able to find? Fit that model using all available training data (no cross-validation).

Solution:

The answer will depend on what models you decided to investigate, and how you decided to tune the hyperparameters. The final fit should involve a call to `gbm()`, `h2o.gbm()` or `xgb.train()`.

Notice that here we have skipped a model assessment of the final model, as we used all the available data to train the final model. If we were in a data-rich situation we might have held out a independent test set until now to check the accuracy of the final model. Or, if we had a lot of computing power, we might have done a nested cross-validation to perform the model assessment.