MA8701 Advanced methods in statistical inference and learning

L7: Random forest, Super Learner, Hyperparameter tuning

Mette Langaas IMF/NTNU

25 February, 2021

Ensembles - third act

So far

L5:

- Trees
- Many trees with bootstrap aggregation
- Many trees into a random forest (not finished)

L6: with Berent Å. S. Lunde (UiB)

- Boosting
- Xgboost
- Avoiding hyperparameter tuning in xgboost

Outline

- Ensembles
- Finishing the random forest from L5
- Super Learner
- Hyperparameter tuning

Left for L8: How to use statistical inference to give CI and compare predictions from different methods.

Ensembles - overview

(ELS Ch 16.1)

With ensembles we want to build *one prediction model* which combines the strength of *a collection of models*.

These models may be simple base models - or more elaborate models.

We have studied bagging - where we take a simple average of the prediction from many models (or majority vote), and the base models can be trees - or other type of models.

Random forest is a version of bagging, with trees made to be different (decorrelated).

We have studied boosting, where the models are trained on sequentially different data - from residuals or gradients of loss functions - and the ensemble members cast weighted votes. We have in particular looked into the xgboost variant of boosting, and also evaluated possible parameters to tune to optimize performance.

In L7 we also look into an ensemble built of elaborate models with the Super Learner and study some possible methods for tuning hyperparameters.

Random forest

(ELS Ch 15) see learning material in L5.

Super Learner

(References: Le Dell (2015) Section 2.2 and van der Laan et al (2016), and Polley et al (2011))

Why do we want to study the Super Learner?

The Super Learner or *generalized stacking* is an algorithm that combines

- multiple, (typically) diverse prediction methods (learning algorithms) called *base learners* into a
- a *metalearner* - which can be seen as a *single* method.

Development:

- 1992: stacking introduce for neural nets by Wolpert
- 1996: adapted to regression problems by Breiman - but only for one type of methods at once (cart with different number of terminal nodes, glms with subset selection, ridge regression with different ridge penalty parameters)
- 2006: proven to have asymptotic theoretical oracle property by van der Laan, Polley and Hubbard.

Ingredients:

- *Training data* (level-zero data) $O_i = (X_i, Y_i)$ of N i.i.d observations.
- A total of L *base learning algorithms* Ψ^l for $l = 1, \ldots, L$, each from some algorithmic class and each with a specific set of model parameters.
- A *metalearner* is used to find an *optimal combination* of the L base learners.

Algorithm

Step 1: Produce level-one data $\mathbf{Z}$

a) Divide the training data $\mathbf{X}$ randomly into V roughly-equally sized validation folds $\mathbf{X}_{(1)}, \ldots, \mathbf{X}_{(V)}$. V is often 5 or 10. (The responses $\mathbf{Y}$ are also needed.)

b) For each base learner Ψ^l perform V-fold cross-validation to produce prediction.

This gives the level-one data set $\mathbf{Z}$ consisting prediction of all the level-zero data - that is a matrix with N rows and L columns.

What could the base learners be?

“Any” method that produces a prediction - “all” types of problems.

- linear regression
- lasso
- cart
- random forest with mtry=value 1
- random forest with mtry=value 2
- xgboost with hyperparameter set 1
- xgboost with hyperparameter set 2
- neural net with hyperparameter set 1

Step 2: Fit the metalearner

a) The starting point is the level-one prediction data $\mathbf{Z}$ together with the responses $(Y_1, \dots, Y_N)$.

b) The metalearner is used to estimate the weights given to each base learner: $\hat{\eta}_i = \alpha_1 z_{1i} + \cdots + \alpha_L z_{Li}$.

What could the metalearner be?

- the mean (bagging)
- constructed by minimizing the
  - squared loss (ordinary least squares) or
  - non-negative least squares
- ridge or lasso regression
- constructed by minimizing 1-ROC-AUC

(Class notes: Study Figure 3.2 from Polley et al)

Step 3: Re-estimate base learners and combine into superlearner on full training data

a) Fit each of the L base learners to the full training set.
b) The *ensemble fit* consists the L base learner fits together with the metalearner fit.

Step 4: Using the ensemble for prediction

For a new observaton $\mathbf{x}^*$

a) Use each of the L base learners to produce a prediction $\mathbf{z}^*$, and
b) feed this to the metalearner-fit to produce the final prediction y^*.

The metalearning

Some observations

- The term *discrete super learner* is used if the base learner with the lowest risk (i.e. CV-error) is selected.
- Since the predictions from multiple base learners may be highly correlated - the chosen method should perform well in that case (i.e. ridge and lasso).
- When minimizing the squared loss it has been found that adding a non-negativity constraint $\alpha_l \leq 0$ works well,
- and also the additivity constraint $\sum_{l=1}^{L} \alpha_l = 1$ - the ensemble is a *convex combination* of the base learners.
- Non-linear optimization methods may be employed for the metalearner if no existing algorithm is available
- Historically a regularized linear model has “mostly” been used
- For classification the logistic response function can be used on the linear combination of base learners (Figure 3.2 Polley).

Examples

Simulations examples

(Class notes: Study Figure 3.3 and Table 3.2 from Polley et al)

Real data

(Class notes: Study Figure 3.4 and Table 3.3 from Polley et al. RE=MSE relative to the linear model OLS.)

Theoretical result

(Le Dell 2015, page 6)

- Oracle selector: the estimator among all possible weighted combinations of the base prediction function that minimizes the risk under the *true data generating distribution*.
- The *oracle result* was established for the Super Learner by van der Laan et al (2006).
- If the *true prediction function* cannot be represented by a combination of the base learners (available), then "optimal" will be the closest linear combination that would be optimal if the true data-generating function was known.
- The oracle result require an *uniformly bounded loss function*. Using the convex restriction (sum alphas $=1$) implies that if each based learner is bounded so is the convex combination. In practice: truncation of the predicted values to the range of the outcome in the training set is sufficient to allow for unbounded loss functions

Uncertainty in the ensemble

(Class notes: Study “Road map” 2 from Polley et al)

- Add an outer (external) cross validation loop (where the super learner loop is inside). Suggestion: use 20-fold, especially when small sample size.
- Overfitting? Check if the super learner does as well or better than any of the base learners in the ensemble.
- Results using *influence functions* for estimation of the variance for the Super Learner are based on asymptotic variances in the use of V-fold cross-validation (see Ch 5.3 of Le Dell, 2015)

Other issues

- Many different implementations available, and much work on parallell processing and speed and memory efficient execution.
- Super Learner implicitly can handle hyperparameter tuning by including the same base learner with different model parameter sets in the ensemble.
- Speed and memory improvements for large data sets involves subsampling, and the R `subsemble` package is one solution, the H2O Ensemble project another.

R example

Code is copied from Guide to SuperLearner and the presentation follows this guide. The data used is the Boston housing dataset from MASS, but with the median value of a house dichotomized into a classification problem.

Observe that only 150 of the 560 observations is used (to speed up things, but of cause that gives less accurate results).

```
data(Boston, package = "MASS")
#colSums(is.na(Boston)) # no missing values
outcome = Boston$medv
# Create a dataframe to contain our explanatory variables.
data = subset(Boston, select = -medv)
#Set a seed for reproducibility in this random sampling.
set.seed(1)
# Reduce to a dataset of 150 observations to speed up mode
train_obs = sample(nrow(data), 150)
# X is our training sample.
x_train = data[train_obs, ]
# Create a holdout set for evaluating model performance.
# Note: cross-validation is even better than a single hold
```

Choosing hyperparameters

- What are *hyperparameters*?
- Which hyperparameters have we encountered in the course so far?

Hyperparameters are parameters than cannot be directly estimated from data. This may be model parameters (the λ in lasso) or parameters that influence the fitting of the model (e.g. related to some optimization algorithm).

We have already studied hyperparameter tuning for the lasso, ridge, elastic net, random forest, and boosting - with the use of cross-validation of some loss function for a predefined set of hyperparameter values.

The use of ensembles like the Super Learner may be seen as an alternative to hyperparameter tuning.

Overview of advices from Berent Å.S. Lunde on tuning parameters in xbgoost (written out after the lecture and approved by Berent):

Ways to speed-up computation:

- Higher learning-rate, then tune the number of boosting iterations.
- When tuning, do not use too high k in k-fold CV (for both speed and also to avoid high variance), or drop CV and use a validation set.
- Speedups with histogram algorithms of order n (avoid exact enumeration of all splits, nlogn).
- Use sparsity when possible!

Comments on hyperparameters:

- Learning rate (eta in xgb): Set as low as computation times allow. Typically in between 0.01 and 0.1 is sufficient.
- Number of trees (boosting iterations): Very much affected by the learning rate. Most important parameter to tune, given a learning rate. Maximum depth of trees: start low, then increase. High values takes significantly longer to train. Think about the problem, and the number of possible interaction effects. If max-depth = J, then interactions among J-1 features is possible. How much is needed?
- Gamma: Very traditional tree-hyperparameter to tune. Tune.
- Colsampling for tree is usually sufficient.
- subsample: in 0.5-0.9
- Min child weight: Something I usually do not think about. Default=1 (low) works. For me

Every problem is different, and there will exist exceptions to the above guidelines. *I look forward to a world without hyperparameters.* (Berent Å.S. Lunde)

The hyperparameters may be continuous (penalty parameter), discrete (number of layers in a neural network, applying early stopping or not) or categorical (choose between different optimizers).

The choice of hyperparameters is important, and will often directly affect the model complexity, and unwise choices may lead to overfitting.

Hyperparametertuning is performed using a separate validation set or by cross-validation. Different loss functions or selection criteria may be used (MSE, AUC, misclassification rate, ...).

The hyperparameter tuning is often referred to as a black-box optimization because we (usually) only calculate loss function values (with CV) and do not get to compute gradients.

What may be challenges with hyperparameter optimization?

Some challenges with hyperparameter optimization (Feurer and Hutter, Ch1):

- expensive evaluation of the model under study (large networks, large data sets)
- unclear which of possibly many hyperparameters that need to be selected carefully (refer to the discussion for xgboost)
- gradient of selection criterion with respect to the hyperparameters not (generally) available, and criterion not convex or smooth in the hyperparameters
- and the need for external validation or CV

There exist many ways to *group* methods for hyperparameter tuning. One way to look at this is (Kuhn and Silge, 2021, Ch 12)

- grid search: specify a set of possible values a priori and investigate only these values, choose the value where the chosen selection criterion is optimal. This is also called "model free methods".
- iterative search: start with a set of values, fit/evaluate some (surrogate) model (might also be the loss function), and based on this choose new values to evaluate next.

For grid search also methods for *speeding up calculations* exists - for example by stopping evaluation at a grid point where the loss is seen to be high after some CV-folds, for example the method of *racing* described by Kuhn and Silge, Ch 13.4.

(Class notes: see example from Kuhn and Silge, 2021, Spacefilling grid vs global search)

Surrogate methods

We will look at two types of surrogate models: Bayesian regression with Gaussian processes (in Bayesian optimization) and regression-type models in response surface methods (presented by group 2).

Design of experiments and response surface methodology

Article presentation by group 2.
G. A. Lujan-Moreno, P. R. Howard, O. G. Rojas and D. C. Montgomery (2018): Design of experiments and response surface methodology to tune machine learning hyperparameters, with a random forest case- study. Expert Systems with Applications. 109, 195-205.

Bayesian optimization

Bayesian optimization is an iterative method - where we start with evaluating some loss function at some predefined set of points in the hyperparameter space. New position in the hyperparameter space are chosen iteratively.
Two key ingredients:

- a surrogate model (we will only look at Bayesian regression with Gaussian processes) to fit to the observed values of the loss function in the hyperparameter space
- an *acquisition* function to decide a new point in the hyperparameter space to evaluate next

Underlying idea: given some “observations” in the hyperparameter space, the task is to decide where to place a new point. We should try a point where:

- we expect a good value and/or
- we have little information so far

To do that we need information on both expected value *and* variance - or preferably the distribution of the loss function for our problem.

We now look at the multivariate Gaussian distribution and conditional distribution, a Gaussian process

Gaussian processes

(Eidsvik 2017, page 6-7, note in TMA4265)
A Gaussian process is defined for

- times or locations x_i, $i = 1, \ldots, n$ in $\Re^d$, where
- $Y_i = Y(x_i)$ is a random variable at x_i
- such that $\mathbf{Y} = (Y_1, \ldots, Y_n)$ is multivariate Gaussian.

The process is *first order (mean) stationary* if $\text{E}(Y(x)) = \mu$ for all x, and this can be extended to depend on covariates.
The process is *second order stationary* if $\text{Var}(Y(x)) = \sigma^2$ for all x and the correlation $\text{Corr}(Y(x), Y(x'))$ only depends on differences between x and x'.
The multivariate Gaussian distribution is defined by the mean and covariance alone.

Correlation functions

(Eidsvik 2017, page 7, Frazier 2018, Ch 3.1)
Correlation functions are also referred to as *kernels*.
We assume that points at positions close to each other have a stronger correlation than point far apart.

Power exponential or Gaussian kernel

$$\mathrm{Corr}(Y(x), Y(x')) = \exp(-\phi_G \|x - x'\|^2)$$

where the L2 distance is used and ϕ_G is a parameter that determine the decay in the correlations.

Matern-type kernel

$$\mathsf{Corr}(Y(x), Y(x\,\acute{})) = (1 + \phi_M \|x - x'\|) \exp(-\phi_M \|x - x'\|)$$

now with decay-describing parameter ϕ_M.

The parameters of the kernels need to be estimated, see Ch 3.2 of Frazier 2018 (who use a slightly different parameterization). We will just assume that these parameters are known.

(Class notes: study Figure 4 of Eidsvik, 2018.)

From correlations into covariance matrix

For simplicty assume that $d = 1$. The number of positions to consider is n.

To get from correlation function to a $n \times n$ covariance matrix first construct a $n \times n$ matrix of distances for each pair of positions, denote this $\mathbf{H}$.

For the Matern-type correlation function the covariance matrix can then be written

$$\Sigma = \sigma^2 (1 + \phi_M \mathbf{H}) \otimes \exp(-\phi_M \mathbf{H}))$$

where $\otimes$ is elementwise multiplication.

See Eidsvik (2018, Ch 3.2 and 3.3) for how to build covariance matrices in an efficient way.

Multivariate normal distribution

aka multivariate Gaussian distribution. Known from TMA4265 and TMA4267.

The random vector $\mathbf{Y}_{p\times 1}$ is multivariate normal N_p with mean and (positive definite) covariate matrix Σ. The pdf is:

$$f(\mathbf{Y}) = \frac{1}{(2\pi)^{\frac{p}{2}}|\Sigma|^{\frac{1}{2}}}\exp\{-\frac{1}{2}(\mathbf{Y}-\ )^T\Sigma^{-1}(\mathbf{Y}-\ )\}$$

Six useful properties of the mvN - we need number 6.

Let $\mathbf{Y}_{(p\times 1)}$ be a random vector from $N_p(\ ,\Sigma)$.

1. The grapical contours of the mvN are ellipsoids (can be shown using spectral decomposition).
2. Linear combinations of components of $\mathbf{Y}$ are (multivariate) normal (can be easily proven using moment generating functions MGF).
3. All subsets of the components of $\mathbf{Y}$ are (multivariate) normal (special case of the above).

4. Zero covariance implies that the corresponding components are independently distributed (can be proven using MGF).
5. $\mathbf{A}\Sigma\mathbf{B}^T = \mathbf{0} \Leftrightarrow \mathbf{AY}$ and $\mathbf{BY}$ are independent.
6. The conditional distributions of the components are (multivariate) normal.

$$\mathbf{Y}_2 \mid (\mathbf{Y}_1 = \mathbf{Y}_1) \sim N_{p2}(\;_2 + \Sigma_{21}\Sigma_{11}^{-1}(\mathbf{Y}_1 - \;_1), \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}).$$

Acquisition function: Expected improvement

(Frazier 2018 page 7)

Thought experiment:

1) we have evaluated our function at all possible points x, and must return a solution based on what we already have evaluated. If the evaluation is noise-less we need to return the point with the largest observed value f.
2) Correction: We may perform one more evaluation. If we choose x we observe $f(x)$, and the best point before that was f_n^*. The improvement at the new observation is then

$$\max(f(x) - f_n^*, 0)$$

3) We define the *expected improvement* as

$$\mathsf{EI}_n(x) = \mathsf{E}_n[\max(f(x) - f_n^*, 0)]$$

where the expectation is taken at the posterior distribution given that we have evaluated f at n observations $x_1, \ldots, x_n$, and the posterior distribution is that f conditional on $x_1, \ldots, x_n, y_1, \ldots, y_n$ is normal with mean $\mu_n(x)$ and variance $\sigma_n^2(x)$.

4) How to evaluate the expected improvement? Integration by parts gives

$$\mathsf{EI}_n(x) = \max(\mu_n(x) - f_n^*, 0)]) + \sigma_n(x)\phi(\frac{\mu_n(x) - f_n^*}{\sigma_n(x)})$$

$$-\mathsf{abs}(\mu_n(x) - f_n^*)\Phi(\frac{\mu_n(x) - f_n^*}{\sigma_n(x)})$$

$\mu_n(x) - f_n^*$ is expected proposed vs previously best

5) We choose to evaluate the point with the largest expected improvement

$$x_{n+1} = \mathsf{argmax}\mathsf{EI}_n(x)$$

Is often found using quasi-Newton optimization.

Algorithm for Bayesian optimization of a function f

(Frazier 2018, page 3, noise-free evaluation)

```
Place a Gaussian process prior on $f$.
Observe $f$ at $n_0$ points from some experimental design. Set $n = n_0$.
while $n \leq N$ do
    Update the posterior on f with all available data
    Let $x_n$ be a maximizer of the acquisition function over $x$, computed using the current posterior
    Observe $y_n = f(x_n)$
    Increment $n$
end while
Return a solution: a point with largest $f(x)$ or the point with the largest posterior mean
```

What does the steps mean?

- Gaussian prior: choose (estimate?) mean and correlation function for the problem.
- Observe n_0 points: calculate the loss function at each of the points (remark: we have noise)
- Update the posterior: calculate the conditional distribution for f for a new point given the observed loss at all previously observed points
- Acquisition function: find argmax$\mathsf{EI}_n(x)$.

(Class notes: Figure 1 of Frazier 2018.)

- For a point x we model the distribution of $f(x)$,
- which is normally distributed with mean $\mu_n(x)$ and variance $\sigma_n^2(x)$. The mean and variance is found from the conditional distribution.
- With 95% credibility interval $\mu_n(x) \pm 1.95\sigma_n(x)$.
- The width of the credibility interval at observations is 0.

Extension

What is the objection function is not observed noise-less? Independent normal error term ε can be added to the previously defined $Y = f(x)$ to make a new $Y = f(x) + \varepsilon$. This (only) adds a diagonal term to the covariance matrix, and it is common to assume that the variance is the same for all x and treat the variance as a hyperparameter.

Example

(Kuhn and Silge, Ch 14, the example is for SVM)

First just grid search to test what is best value for mtry

```
data(Boston, package = "MASS")
# first using a grid
tune_grid <- expand.grid(
  mtry = (1:13))
#  ntree=seq(100,500,length=10)) # how to also include ntr
tune_control <- caret::trainControl(
  method = "oob", # cross-validation #eller cv
  #number = 3, # with n folds
  verboseIter = FALSE, # no training log
  allowParallel = FALSE # FALSE for reproducible results
)
rf_tune <- caret::train(
  medv~crim+zn+indus+chas+nox+rm+age+dis+rad+tax+ptratio+bl
  data=Boston,
  na.action=na.roughfix,
  trControl = tune_control,
  tuneGrid = tune_grid
```

The R the function `tune_bayes` is available in the package `tune`, and requires that the analyses is done with a workflow. Default in the GP is exponential correlation function, but first we try the Matern.

```
tree_rec <- recipe(medv~crim+zn+indus+chas+nox+rm+age+dis+r

tune_spec <- rand_forest( # parsnip interface to random fo
  mode="regression",
  mtry = tune(),
  trees = tune(),
#  min_n = tune()
) %>%
#  set_mode("regression") %>%
#  set_engine("ranger",objective="reg:rmse") # errors with
  set_engine("randomForest") # randomforest ok

tune_wf <- workflow() %>%
  add_recipe(tree_rec) %>%
  add_model(tune_spec)
```

Here we try the default exponential correlation.

```
bayesres2<- tune_bayes(tune_wf,
    resamples = vfold,
    #metrics = rmse,
    #corr=list(type="matern",nu=5/2),
    #default in corr_mat(GPfit) is "exponential" power 1.9
    initial = 10,
    param_info = tune_param,
    iter = 10,
    objective=exp_improve(),
    control = ctrl
  )
dput(bayesres2,"bayesres2.dd")
```

```
bayesres2=dget("bayesres2.dd")
show_best(bayesres2,n=10)
```

```
## # A tibble: 10 x 9
##      mtry trees .metric .estimator  mean     n std_err .c
##     <int> <int> <chr>   <chr>      <dbl> <int>   <dbl> <c
```

Suggested software

(Frazier 2018, Ch 6)

- R: DiceOptim (on CRAN)
- R: tune_bayes in `tune` (also CRAN)
- Python: Spearmint https://github.com/HIPS/Spearmint
- Python: GPyOpt https://github.com/SheffieldML/GPyOpt
- Python: GPFlow (Tensorflow) https://github.com/GPflow/GPflow and GPYTorch (PyTorch) https://github.com/cornellius-gp/gpytorch

References

Super Learner

- Erin Le Dell (2015): Scalable Ensemble Learning and Computationally Efficient Variance Estimation. PhD Thesis, University of California, Berkeley.
- Mark J. van der Laan, Eric C. Polley, Alan E. Hubbard (2007): Super Learner, Statistical Applications in Genetics and Molecular Biology, 6, 1, 25
- Eric C. Polley, Sherri Rose, Mark J. van der Laan (2011): Super Learning. Chapter 3 of M. J. van der Laan and S. Rose. Targeted Learning, Springer.

Ensembles

- [ELS] The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition (Springer Series in Statistics, 2009) by Trevor Hastie, Robert Tibshirani, and Jerome Friedman. Ebook.

Hyperparameter tuning

- M. Feurer and F. Hutter (2019). In F. Hutter et al (eds.)