Boosting Tree and Package

Gireg Willame

Detralytics

2023-03-15

1. Practical info and litterature review
2. Boosting Tree Method
3. `BT` package and parameters
4. Case-study

Section 1

Practical info and litterature review

Link to package and books/articles

- The Boosting Tree package can be downloaded via `CRAN` or via github https://github.com/GiregWillame/BT. The latter contains the on-going version with latest changes.
- M. Denuit, D. Hainaut and J. Trufin (2019). **Effective Statistical Learning Methods for Actuaries |: GLMs and Extensions**, *Springer Actuarial*.
- M. Denuit, D. Hainaut and J. Trufin (2019). **Effective Statistical Learning Methods for Actuaries ||: Tree-Based Methods and Extensions**, *Springer Actuarial*.
- M. Denuit, D. Hainaut and J. Trufin (2019). **Effective Statistical Learning Methods for Actuaries |||: Neural Networks and Extensions**, *Springer Actuarial*.

Link to package and books/articles (cont'd)

- M. Denuit, D. Hainaut and J. Trufin (2022). **Response versus gradient boosting trees, GLMs and neural networks under Tweedie loss and log-link**. Accepted for publication in *Scandinavian Actuarial Journal*.
- M. Denuit, J. Huyghe and J. Trufin (2022). **Boosting cost-complexity pruned trees on Tweedie responses: The ABT machine for insurance ratemaking**. Paper submitted for publication.
- M. Denuit, J. Trufin and T. Verdebout (2022). **Boosting on the responses with Tweedie loss functions**. Paper submitted for publication.
- L. Breiman, J. H. Friedman, R.A. Olshen, C.J. Stone (1984). **Classification and Regression Trees**. *Wadsworth Statistics/Probability Series*.
- Interesting code resource: Florian Pechon's repo.

Section 2

Boosting Tree Method

Tweedie family

A random variable Y belongs to the *Tweedie* family if it's a member of the *exponential dispersion family* and satisfies the following criteria:

$$\mathbb{V}\text{ar}(\mathbb{E}(Y)) = \mathbb{E}(Y)^{\xi}, \text{ for some } \xi \in \mathbb{R} \backslash]0, 1[.$$

In particular, the following well-known distributions are part of this family:

- Gaussian distribution, with $\xi = 0$.
- Poisson distribution, with $\xi = 1$.
- Poisson-Gamma compound distribution, with $\xi \in]1, 2[$.
- Gamma distribution, with $\xi = 2$.
- Inverse Gaussian distribution, with $\xi = 3$.

Deviance

We remind that the deviance loss function L corresponds to 2 times the log-likelihood ratio of the saturated model compared to the reduced (fitted) one, that is

$$L(y, \mathbb{E}(Y|\widehat{\theta}_f)) = 2\Big(\log \mathcal{L}(\widehat{\theta}_s|y) - \log \mathcal{L}(\widehat{\theta}_f|y)\Big),$$

where

- $\widehat{\theta}_s$ denotes the estimated parameters for the saturated model
- $\widehat{\theta}_f$ denotes the estimated parameters for the fitted model f.

BT Algorithm

The idea is to combine multiple weak learners (trees in our case) to approximate the response variable.

Algorithm 1: Boosting Tree Algorithm

```
output: $\widehat{\text{score}}(\mathbf{x}) = \widehat{\text{score}}_M(\mathbf{x})$
1 init: First score to be a constant, e.g.
2     $$\widehat{\text{score}}_0(x) = \operatorname*{argmin}_{\beta} \sum_{i \in \mathcal{I}} L\Big(y_i, g^{-1}(\beta)\Big).$$
3 for m=1 to M do
4     Fit a regression tree $T(\mathbf{x}; \widehat{\mathbf{a}_m})$ with
          $$\widehat{\mathbf{a}_m} = \operatorname*{argmin}_{\mathbf{a}_m} \sum_{i \in \mathcal{I}} L\Bigg(y_i, g^{-1}\Big(\widehat{\text{score}}_{m-1}(\mathbf{x}_i) + T(\mathbf{x}_i; \mathbf{a}_m)\Big)\Bigg).$$
5     Update $\widehat{\text{score}}_m(\mathbf{x}) = \widehat{\text{score}}_{m-1}(\mathbf{x}) + T(\mathbf{x}; \widehat{\mathbf{a}_m})$.
6 end
```

Boosting - Remarks on subsampling

- While fitting the tree for a given iteration m, one can introduce some randomness.
- This approach aims to fit the current weak learner on a random sample of the training database, without replacement.
- It has basically two benefits:
  - The variance of each weak learner estimates at each iteration increases. However, there will be less correlation between these estimates at different iterations. It finally leads to a **variance reduction** of the combined model.
  - Reduction in the computational time.

Boosting - Remarks on overfitting

- In opposite to ensemble methods, it's important to define the optimal number of trees to avoid overfitting. For a given set of parameters, one would therefore need to find the best iteration. This is often achieved thanks to validation set, cross-validation or even out-of-bag improvements.
- One can also introduce a **shrinkage** parameter τ. This parameter will scale the contribution of each tree in the expansion by modifying the update step as

$$\widehat{\text{score}}_m(\mathbf{x}) = \widehat{\text{score}}_{m-1}(\mathbf{x}) + \tau T(\mathbf{x}; \widehat{\mathbf{a}_m})$$

  - Having a small shrinkage factor will require more iterations to be fitted and so higher computational time.
  - It however tends to improve the performances.

Example in Poisson distribution case

Given an observation i of the training set, let us denote by y_i the claims count, $\mathbf{x}_i$ the features vector and d_i the exposure-to-risk. One can then rewrite the algorithm optimization step m as

$$\begin{aligned}
\widehat{\mathbf{a}_m} &= \operatorname*{argmin}_{\mathbf{a}_m} \sum_{i \in \mathcal{I}} L\Big(y_i, d_i \exp\Big(\widehat{\text{score}}_{m-1}(\mathbf{x}_i) + T(\mathbf{x}_i; \mathbf{a}_m)\Big)\Big) \\
&= \operatorname*{argmin}_{\mathbf{a}_m} \sum_{i \in \mathcal{I}} L\Big(y_i, d_i \exp\Big(\widehat{\text{score}}_{m-1}(\mathbf{x}_i)\Big) \exp\big(T(\mathbf{x}_i; \mathbf{a}_m)\big)\Big) \\
&= \operatorname*{argmin}_{\mathbf{a}_m} \sum_{i \in \mathcal{I}} L\Big(y_i, d_{mi} \exp\big(T(\mathbf{x}_i; \mathbf{a}_m)\big)\Big)
\end{aligned}$$

The value d_{mi} is a constant and can be considered as a working exposure-to-risk applied to observation i.

$\rightarrow$ **In a Poisson log-link context, a boosting model as previously described is no more than fitting trees with updated/working exposure-to-risk.**

Tweedie deviance-based boosting under log-link

- Let us denote ν_i the records' weight and ξ the Tweedie power.
- For iteration m, let us define the working weights

$$\nu_{mi} = \nu_i \exp\left(\widehat{\text{score}}_{m-1}(\mathbf{x}_i)\right)^{2-\xi}$$

and the working response

$$r_{mi} = \frac{y_i}{\exp\left(\widehat{\text{score}}_{m-1}(\mathbf{x}_i)\right)}.$$

- Denuit et al. established that

$$\widehat{\mathbf{a}_m} = \underset{\mathbf{a}_m}{\operatorname{argmin}} \sum_{i \in \mathcal{I}} \nu_i L\left(y_i, \exp\left(\widehat{\text{score}}_{m-1}(\mathbf{x}_i) + T(\mathbf{x}_i; \mathbf{a}_m)\right)\right)$$

can be rewritten as

$$\widehat{\mathbf{a}_m} = \underset{\mathbf{a}_m}{\operatorname{argmin}} \sum_{i \in \mathcal{I}} \nu_{mi} L\left(r_{mi}, \exp\left(T(\mathbf{x}_i; \mathbf{a}_m)\right)\right).$$

ABT Algorithm - Cost-complexity measure

- Idea: Fit cost-complexity pruned trees in a forward stage-wise additive and adaptive way.
- In this case, the interaction depth (=Number terminal nodes - 1) is free to vary amongst the trees entering the score.
- Let us define the cost-complexity measure of tree $T(\mathbf{x}; \mathbf{a}_m)$ as

$$R_\alpha(T(\mathbf{x}; \mathbf{a}_m)) = \sum_{i \in \mathcal{I}} \nu_{mi} L\Big(r_{mi}, \exp\Big(T(\mathbf{x}_i; \mathbf{a}_m)\Big)\Big) + \alpha |T(\mathbf{x}; \mathbf{a}_m)|$$

- Having deeper tree will always favor in-sample deviance. It will however be balanced by the complexity penalization.
- Given $T(\mathbf{x}; \mathbf{a}'_m)$ a 1-leaf more complex tree than $T(\mathbf{x}; \mathbf{a}_m)$, we've $R_\alpha(T(\mathbf{x}, \mathbf{a}'_m)) \geq R_\alpha(T(\mathbf{x}, \mathbf{a}_m))$ iif

$$\alpha \geq \sum_{i \in \mathcal{I}} \nu_{mi} L\Big(r_{mi}, \exp\Big(T(\mathbf{x}_i; \mathbf{a}_m)\Big)\Big) - \sum_{i \in \mathcal{I}} \nu_{mi} L\Big(r_{mi}, \exp\Big(T(\mathbf{x}_i; \mathbf{a}'_m)\Big)\Big).$$

ABT Algorithm - Minimal cost-complexity pruning

- A tree $T(\mathbf{x};\mathbf{a}_m)$ that is obtained from $T(\mathbf{x};\mathbf{a}'_m)$ by successively pruning branches is called a subtree and is denoted as $T(\mathbf{x};\mathbf{a}_m) \preceq T(\mathbf{x};\mathbf{a}'_m)$.
- Consider tree $T(\mathbf{x};\mathbf{a}_m)$ and a fixed value of α. We define $T(\mathbf{x};\mathbf{a}_m(\alpha))$ as the subtree of $T(\mathbf{x};\mathbf{a}_m)$ such that

$$R_\alpha(T(\mathbf{x};\mathbf{a}_m(\alpha))) = \min_{T(\mathbf{x};\mathbf{a}) \preceq T(\mathbf{x};\mathbf{a}_m)} R_\alpha(T(\mathbf{x};\mathbf{a})).$$

  Obviously, for every value of α there exists at least one subtree of $T(\mathbf{x};\mathbf{a}_m)$ minimizing R_α.
- Because there can be multiple subtrees satisfying the above condition, we supplement it by (Proven by Breiman et al. for all α)

$$R_\alpha(T(\mathbf{x};\mathbf{a})) = R_\alpha(T(\mathbf{x};\mathbf{a}_m(\alpha))) \Rightarrow T(\mathbf{x};\mathbf{a}_m(\alpha)) \preceq T(\mathbf{x};\mathbf{a}).$$

ABT Algorithm - Minimal cost-complexity pruning(Cont'd)

- When $\alpha = 0$, there's no penalization and the biggest tree i.e. $T(\mathbf{x};\mathbf{a}_m)$ will be retained. As α increases, the retained subtrees $T(\mathbf{x};\mathbf{a}_m(\alpha))$ will have fewer terminal nodes, up to the point where $T(\mathbf{x};\mathbf{a}_m(\alpha))$ will consist in root node.
- Tree $T(\mathbf{x};\mathbf{a}_m)$ has a finite number of subtrees. Hence, even if α vary in continuous way on $\mathbb{R}^+$, $\{T(\mathbf{x};\mathbf{a}_m(\alpha))\}_{\alpha>0}$ is finite.
- Precisely, Breiman et al. established that there exists an increasing sequence $0 = \alpha_0 < \alpha_1 < \cdots < \alpha_{\kappa_m+1} = \infty$ such that

$$T(\mathbf{x};\mathbf{a}_m(\alpha)) = T(\mathbf{x};\mathbf{a}_m(\alpha_k)), \forall k = 0, 1, \ldots, \kappa_m, \forall \alpha \in [\alpha_k, \alpha_{k+1}[$$

$$T(\mathbf{x};\mathbf{a}_m(\alpha_{\kappa_m})) \preceq T(\mathbf{x};\mathbf{a}_m(\alpha_{\kappa_m-1})) \preceq \cdots \preceq T(\mathbf{x};\mathbf{a}_m(\alpha_0)).$$

$\rightarrow$ **The optimal subtrees appear to be nested: Any subtree $T(\mathbf{x};\mathbf{a}_m(\alpha_k))$ in $\{T(\mathbf{x};\mathbf{a}_m(\alpha_k))\}_{k=0,\ldots,\kappa_m}$ can be obtained by pruning the previous subtree $T(\mathbf{x};\mathbf{a}_m(\alpha_{k-1}))$.**

ABT Algorithm

- At iteration m, we first fit a tree $T(\mathbf{x}; \widehat{\mathbf{a}_m}(\alpha_0))$ with depth D, on the working training set $\mathcal{D}^{(m)} = \{(\nu_{mi}, r_{mi}, \mathbf{x}_i), i \in \mathcal{I}\}$.
- This tree is built such that any tree with D internal nodes is strictly included in $T(\mathbf{x}; \widehat{\mathbf{a}_m}(\alpha_0))$.
- We then prune $T(\mathbf{x}; \widehat{\mathbf{a}_m}(\alpha_0))$ up to get tree $T(\mathbf{x}; \widehat{\mathbf{a}_m}(\alpha_{\kappa_m^*}))$ with at most D internal nodes, that is

$$|T(\mathbf{x}; \widehat{\mathbf{a}_m}(\alpha_{\kappa_m^*-1}))| > D + 1 \text{ and } |T(\mathbf{x}; \widehat{\mathbf{a}_m}(\alpha_{\kappa_m^*}))| \leq D + 1$$

- The m^{th} tree $T(\mathbf{x}; \widehat{\mathbf{a}_m}(\alpha_{\kappa_m^*}))$ entering the score has therefore D internal nodes unless there's no tree in the sequence $\{T(\mathbf{x}; \widehat{\mathbf{a}_m}(\alpha_k))\}_{k=0,\ldots,\kappa_m}$ of that size, in which case the chosen tree has a smaller size.

Section 3

BT package and parameters

Main function

The package has been built in a Tweedie framework, using log-link function. Let's import it and have a look at the main function parameters.

```
##
## 1  function (formula = formula(data), data = list(), tweedie.power = 1,
## 2      ABT = TRUE, n.iter = 100, train.fraction = 1, interaction.depth = 4,
## 3      shrinkage = 1, bag.fraction = 1, colsample.bytree = NULL,
## 4      keep.data = TRUE, is.verbose = FALSE, cv.folds = 1, folds.id = NULL,
## 5      n.cores = 1, tree.control = rpart.control(xval = 0, maxdepth = (if (!is.null(interaction.depth)) {
## 6          interaction.depth
## 7      } else {
## 8          10
## 9      }), cp = -Inf, minsplit = 2), weights = NULL, seed = NULL,
## 10     ...)
```

One can basically split this set into four subsets:

- Global configuration
- Algorithm configuration
- Tree specific parameters
- Learning task parameters: Only implemented for Poisson distribution.

Global configuration

- `formula` a symbolic description of the model to be fit. Note that the offset isn't supported in this algorithm.
- `data` a data frame containing the variables in the model.
- `keep.data` a boolean variable indicating whether to keep the data frames or not. This is particularly useful if one wants to keep track of the initial data frames and is further used for predicting in case any data frame is specified.
- `is.verbose` if set to `TRUE`, the function will print out the algorithm progress.
- `weights` optional vector of weights used in the fitting process. These weights must be positive but do not need to be normalized. By default, an uniform weight of 1 is given for each observation.

Global configuration (Cont'd)

- `seed` optional number used as seed. Please note that if `cv.folds`, `parLapply` function is called. Therefore, the seed (if defined) used inside each fold will be a multiple of the `seed` parameter.
- `n.cores` the number of cores to use for parallelization. This parameter is used during the cross-validation.

Please note that the NA values are currently not handled. In fact, all rows having at least one NA are dropped before calling the algorithm.

Algorithm configuration

- `ABT` a boolean parameter. If `TRUE` an adaptive boosting tree algorithm is built whereas if `FALSE` a boosting tree algorithm is run.
- `n.iter` the total number of iterations to fit. This is equivalent to the number of trees and the number of basis function in the additive expansion.
- `shrinkage` a shrinkage parameter (in the interval (0,1]) applied to each tree in the expansion.
- `colsample.bytree` each tree will be trained on a random subset of `colsample.bytree` number of features, adding variability to the algorithm and reducing computation time.

Algorithm configuration (Cont'd)

- `train.fraction` the first `train.fraction` % observations are used to fit the boosting tree and the remainder are used for computing out-of-sample estimates (also known as validation error) of the loss function.
- `bag.fraction` the fraction of independent training observations randomly selected to propose the next tree in the expansion. This introduces randomness into the model fit.
- `cv.folds` a positive integer representing the number of cross-validation folds to perform. If higher than 1 then `BT`, in addition to the usual fit, will perform a cross-validation and calculate an estimate of generalization error.
- `folds.id` an optional vector of values identifying what fold each observation is in. If supplied, this parameter prevails over `cv.folds`.

Algorithm configuration - Remarks

- Generally, the database is split between **train** and **test** sets before starting. The latter will be used to assess final generalization performance only.
- There are basically two ways to assess model performances:
  - Using a **validation set**. The model is obviously built on the train set and its performance will then be computed on the validation set.
  - Using **cross-validation**. In the case of 3-cv, the train set will be split into 3 parts. Each part will successively play the role of validation set while the two others will represent the on-going train set on which a model will be fitted. In the end, the cross-validation error is obtained by averaging the obtained cv-errors.

Algorithm configuration - Remarks (Cont'd)

- In case of bagging, the optimal number of iterations can also be found via the so-called **out-of-bag improvements**, which is the difference between the following errors:
  - At the beginning of iteration *m*, some observations are left out-of-sample while building the model. Current model performances are computed on this sample.
  - At the end of iteration *m*, a new tree is added to the additive model. Once again, performances are evaluated on the out-of-sample.

Tree specific parameters

- `interaction.depth` the maximum depth of variable interactions: 1 builds an additive model, 2 builds a model with up to two-way interactions, etc. This parameter can also be interpreted as the maximum number of non-terminal nodes.

Each tree in the expansion are built thanks to `rpart` package. While it could totally be managed within the code, we decided to leave the tree specific parameters as input.

- `tree.control` allows to define additional `rpart` tree parameters that will be used at each iteration. It can also be useful if one wants to e.g. manage the minimal number of records within leaves.

Section 4

Case-study