Gradient Boosting Trees

Applied Machine Learning for Educational Data Science

true

11/25/2021

Contents

[Updated: Sun, Nov 28, 2021 - 16:03:22]

In Bagged Trees or Random Forests models, the trees are developed independently by taking a random sample of rows and columns from the dataset. The main difference between Gradient Boosting Trees and Bagged Trees or Random Forests is that the trees are developed sequentially and each tree model is built upon the errors of the previous tree models. The sequential process of model building and predictions in gradient boosted trees can be conceptually demonstrated as below.

Sequential Development of Trees from Residuals

Let's try to implement this idea in a toy dataset we use to predict a readability score from the number of sentences.

```
readability_sub <- read.csv('https://raw.githubusercontent.com/uo-datasci-specialization/c4-ml-fall-202

readability_sub[,c('sents','target')]
```

```
   sents   target
1      7 -2.58591
2     23  0.45993
3     17 -1.07471
4      7 -1.81700
5      6 -1.81492
```

```
6     18 -0.94968
7     10 -0.12103
8      4 -2.82201
9      9 -0.74845
10    28  0.73949
11    15 -0.96219
12    10 -2.21515
13    10 -1.21845
14     8 -1.89544
15    19 -0.04101
16    15 -1.83717
17     6 -0.18819
18     6 -0.81739
19     7 -1.86308
20    19 -0.41630
```

Iteration 0

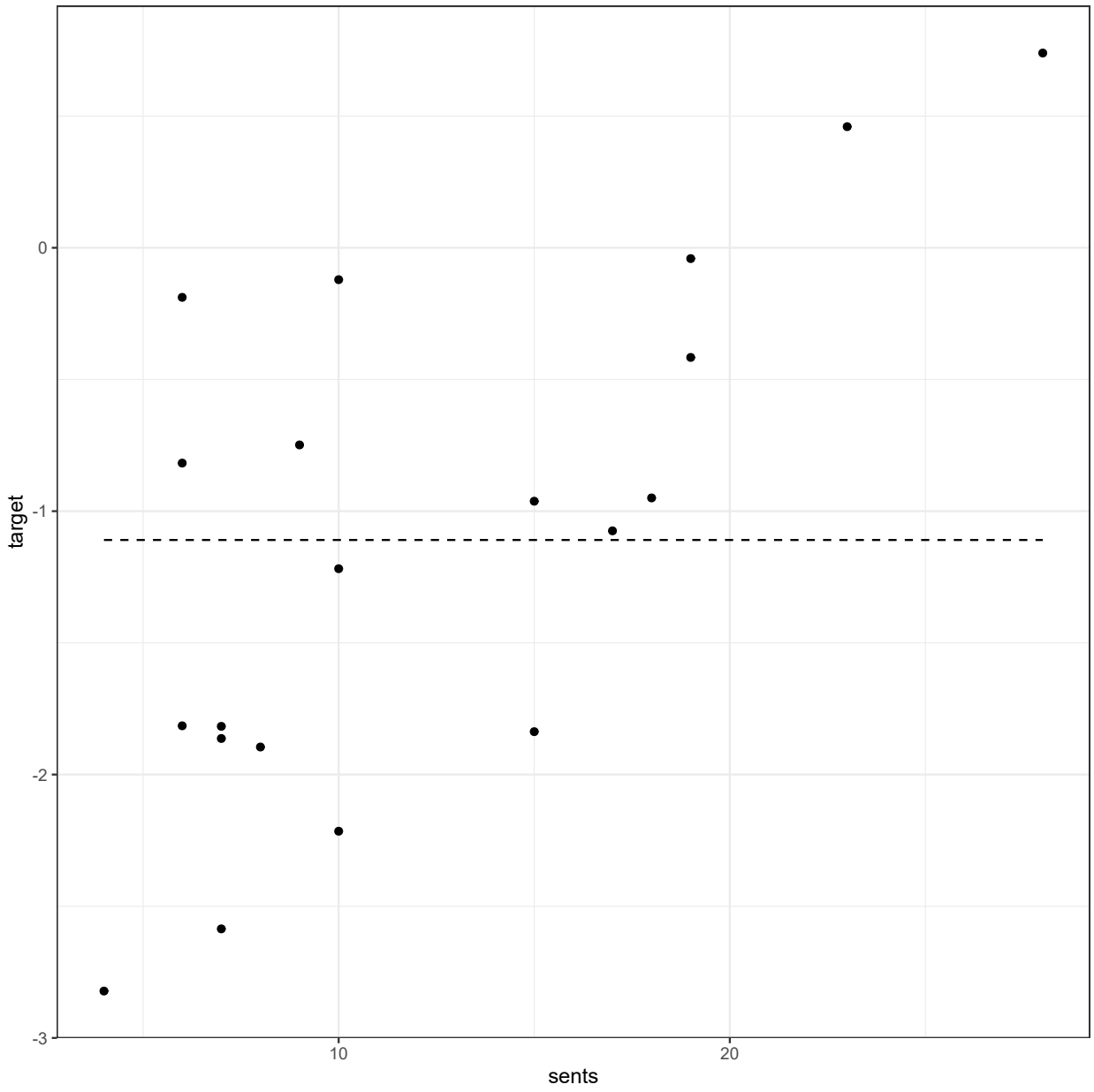
We first start with a simple model that uses the average target outcome to predict the readability for all observations in this toy dataset. We calculate the predictions and residuals from our first model, and fit a tree model to the residuals using the number of sentences as the predictor.

```
readability_sub$pred1 <- mean(readability_sub$target)
readability_sub$res1  <- readability_sub$target - readability_sub$pred


readability_sub[,c('sents','target','pred1','res1')]
```

```
   sents   target  pred1     res1
1      7 -2.58591 -1.109 -1.47648
2     23  0.45993 -1.109  1.56936
3     17 -1.07471 -1.109  0.03473
4      7 -1.81700 -1.109 -0.70757
5      6 -1.81492 -1.109 -0.70548
6     18 -0.94968 -1.109  0.15975
7     10 -0.12103 -1.109  0.98840
8      4 -2.82201 -1.109 -1.71257
9      9 -0.74845 -1.109  0.36098
10    28  0.73949 -1.109  1.84892
11    15 -0.96219 -1.109  0.14724
12    10 -2.21515 -1.109 -1.10572
13    10 -1.21845 -1.109 -0.10902
14     8 -1.89544 -1.109 -0.78601
15    19 -0.04101 -1.109  1.06842
16    15 -1.83717 -1.109 -0.72773
17     6 -0.18819 -1.109  0.92125
18     6 -0.81739 -1.109  0.29204
19     7 -1.86308 -1.109 -0.75364
20    19 -0.41630 -1.109  0.69313
```

```
# SSE at the end of Iteration 0

sum(readability_sub$res1^2)
```

```
[1] 18.65
```

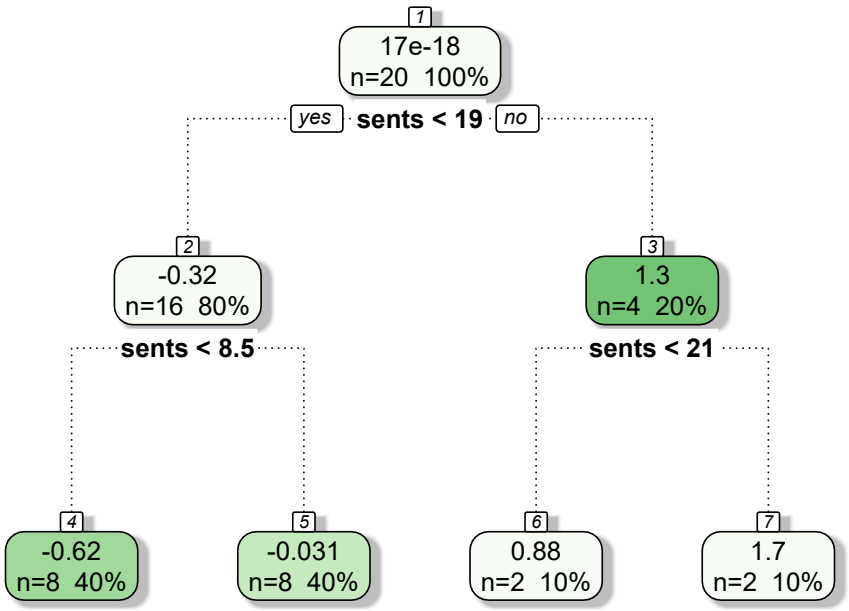


Iteration 1

Now, we fit a tree model to predict the residuals from the number of sentences. Notice that I fix the value of certain parameters while fitting the tree model (e.g., complexity parameter, minsplit, maxdepth).

```
require(rpart)
require(rattle)

model1 <- rpart(formula = res1 ~ sents,
                data    = readability_sub,
                method  = "anova",
```

```
                control = list(minsplit=2,
                               cp=0,
                               minbucket = 2,
                               maxdepth = 2)
                )


fancyRpartPlot(model1,type=2,sub='')
```



Let's see the predictions of residuals from Model 1.

```
predict(model1, readability_sub)
```

```
      1       2       3       4       5       6       7       8
-0.61606  1.70914 -0.03142 -0.61606 -0.61606 -0.03142 -0.03142 -0.61606
      9      10      11      12      13      14      15      16
-0.03142  1.70914 -0.03142 -0.03142 -0.03142 -0.61606  0.88078 -0.03142
     17      18      19      20
-0.61606 -0.61606 -0.61606  0.88078
```

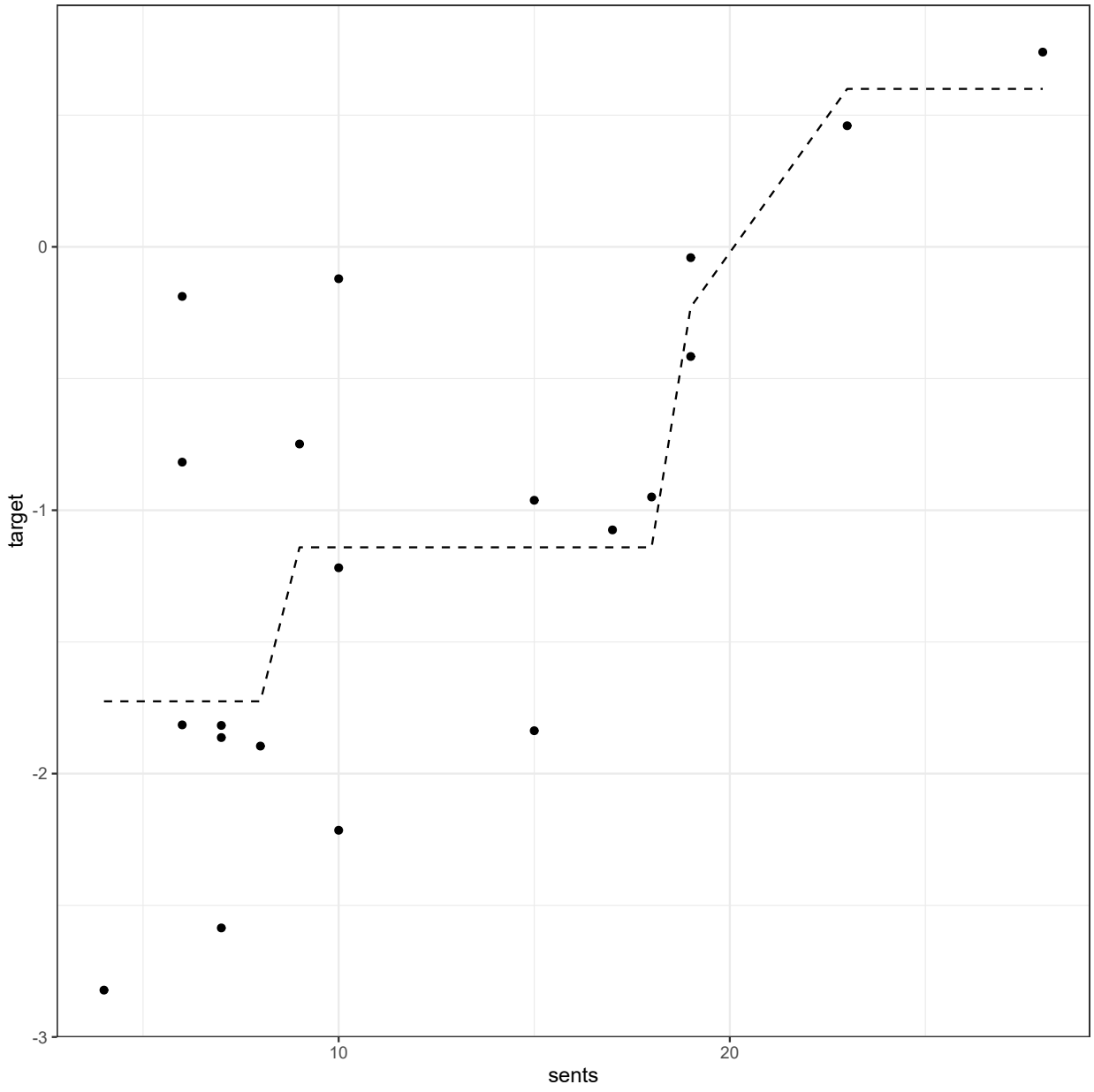
Now, add the predicted residuals from Model 1 to the predictions from Iteration 0 to obtain the new predictions at the end of Iteration 1.

```
readability_sub$pred2  <- readability_sub$pred1 + predict(model1, readability_sub)
readability_sub$res2   <- readability_sub$target - readability_sub$pred2


readability_sub[,c('sents','target','pred1','res1','pred2','res2')]
```

```
   sents   target  pred1     res1   pred2     res2
1      7 -2.58591 -1.109 -1.47648 -1.7255 -0.86042
2     23  0.45993 -1.109  1.56936  0.5997 -0.13978
3     17 -1.07471 -1.109  0.03473 -1.1409  0.06615
4      7 -1.81700 -1.109 -0.70757 -1.7255 -0.09151
5      6 -1.81492 -1.109 -0.70548 -1.7255 -0.08943
6     18 -0.94968 -1.109  0.15975 -1.1409  0.19117
7     10 -0.12103 -1.109  0.98840 -1.1409  1.01982
8      4 -2.82201 -1.109 -1.71257 -1.7255 -1.09651
9      9 -0.74845 -1.109  0.36098 -1.1409  0.39240
10    28  0.73949 -1.109  1.84892  0.5997  0.13978
11    15 -0.96219 -1.109  0.14724 -1.1409  0.17866
12    10 -2.21515 -1.109 -1.10572 -1.1409 -1.07430
13    10 -1.21845 -1.109 -0.10902 -1.1409 -0.07760
14     8 -1.89544 -1.109 -0.78601 -1.7255 -0.16995
15    19 -0.04101 -1.109  1.06842 -0.2287  0.18765
16    15 -1.83717 -1.109 -0.72773 -1.1409 -0.69631
17     6 -0.18819 -1.109  0.92125 -1.7255  1.53731
18     6 -0.81739 -1.109  0.29204 -1.7255  0.90810
19     7 -1.86308 -1.109 -0.75364 -1.7255 -0.13758
20    19 -0.41630 -1.109  0.69313 -0.2287 -0.18765
```

```
# SSE at the end of Iteration 1

sum(readability_sub$res2^2)
```

```
[1] 8.216
```

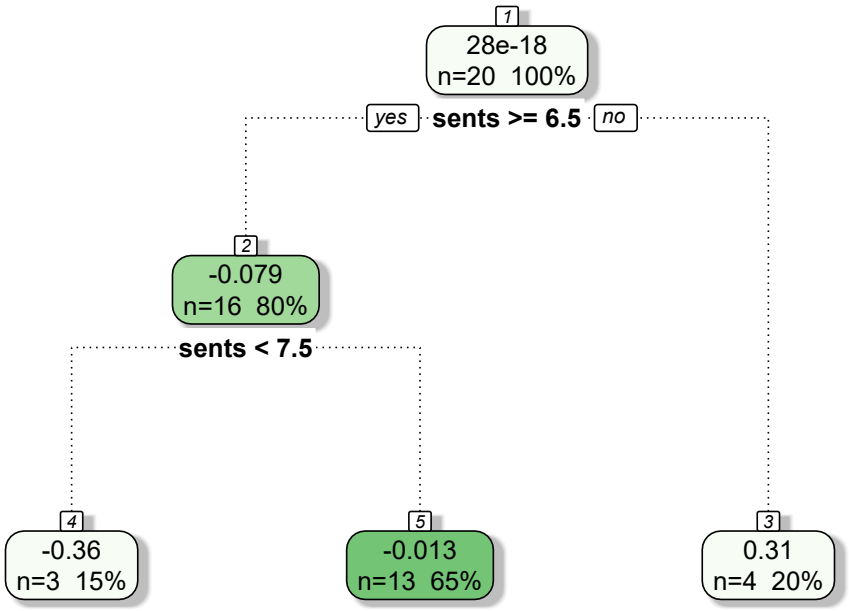


Iteration 2

We repeat Iteration 1, but the only difference is that we now fit a tree model again to predict the residuals obtained after the updated predictions at the end of Iteration 1.

```
model2 <- rpart(formula = res2 ~ sents,
                data    = readability_sub,
                method  = "anova",
                control = list(minsplit=2,
                               cp=0,
                               minbucket = 2,
                               maxdepth = 2)
                )
```

```
fancyRpartPlot(model2,type=2,sub='')
```



Let's see the predictions of residuals from Model 2.

```
predict(model2, readability_sub)
```

```
       1        2        3        4        5        6        7        8
-0.36317 -0.01307 -0.01307 -0.36317  0.31487 -0.01307 -0.01307  0.31487
       9       10       11       12       13       14       15       16
-0.01307 -0.01307 -0.01307 -0.01307 -0.01307 -0.01307 -0.01307 -0.01307
      17       18       19       20
 0.31487  0.31487 -0.36317 -0.01307
```

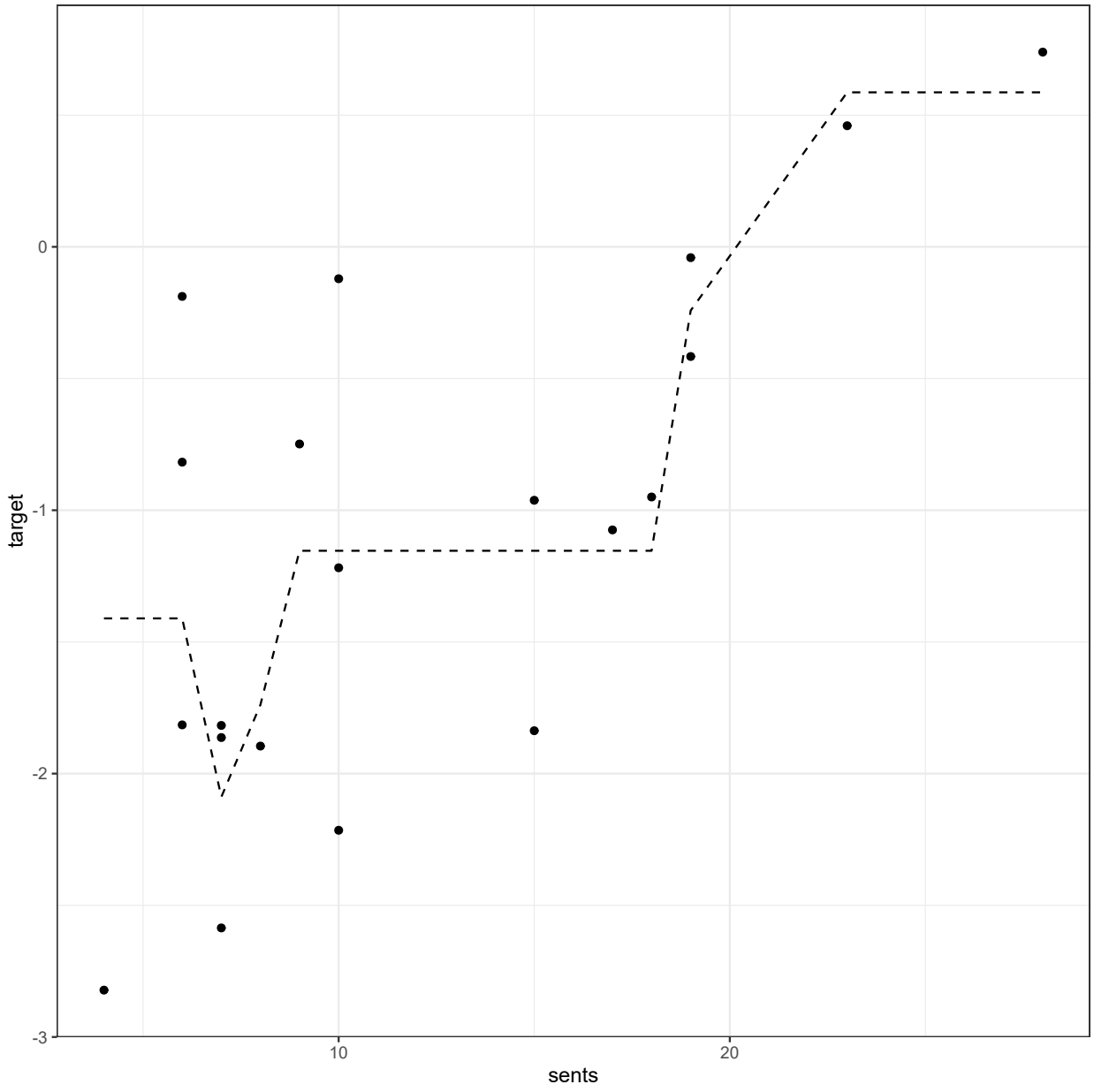
Now, add the predicted residuals from Model 2 to the predictions from Iteration 1 to obtain the new predictions at the end of Iteration 2.

```
readability_sub$pred3  <- readability_sub$pred2 + predict(model2, readability_sub)
readability_sub$res3   <- readability_sub$target - readability_sub$pred3


readability_sub[,c('sents','target','pred1','res1','pred2','res2','pred3','res3')]
```

```
   sents   target  pred1     res1   pred2     res2   pred3     res3
1      7 -2.58591 -1.109 -1.47648 -1.7255 -0.86042 -2.0887 -0.49725
2     23  0.45993 -1.109  1.56936  0.5997 -0.13978  0.5866 -0.12670
3     17 -1.07471 -1.109  0.03473 -1.1409  0.06615 -1.1539  0.07922
4      7 -1.81700 -1.109 -0.70757 -1.7255 -0.09151 -2.0887  0.27166
5      6 -1.81492 -1.109 -0.70548 -1.7255 -0.08943 -1.4106 -0.40429
6     18 -0.94968 -1.109  0.15975 -1.1409  0.19117 -1.1539  0.20424
7     10 -0.12103 -1.109  0.98840 -1.1409  1.01982 -1.1539  1.03290
8      4 -2.82201 -1.109 -1.71257 -1.7255 -1.09651 -1.4106 -1.41138
9      9 -0.74845 -1.109  0.36098 -1.1409  0.39240 -1.1539  0.40547
10    28  0.73949 -1.109  1.84892  0.5997  0.13978  0.5866  0.15285
11    15 -0.96219 -1.109  0.14724 -1.1409  0.17866 -1.1539  0.19174
12    10 -2.21515 -1.109 -1.10572 -1.1409 -1.07430 -1.1539 -1.06122
13    10 -1.21845 -1.109 -0.10902 -1.1409 -0.07760 -1.1539 -0.06452
14     8 -1.89544 -1.109 -0.78601 -1.7255 -0.16995 -1.7386 -0.15688
15    19 -0.04101 -1.109  1.06842 -0.2287  0.18765 -0.2417  0.20072
16    15 -1.83717 -1.109 -0.72773 -1.1409 -0.69631 -1.1539 -0.68324
17     6 -0.18819 -1.109  0.92125 -1.7255  1.53731 -1.4106  1.22244
18     6 -0.81739 -1.109  0.29204 -1.7255  0.90810 -1.4106  0.59323
19     7 -1.86308 -1.109 -0.75364 -1.7255 -0.13758 -2.0887  0.22559
20    19 -0.41630 -1.109  0.69313 -0.2287 -0.18765 -0.2417 -0.17457
```

```
# SSE at the end of Iteration 2

sum(readability_sub$res3^2)
```

```
[1] 7.422
```



We can keep iterating and add tree models as long as we find a tree model that improves our predictions (minimizing SSE).

A more formal introduction of Gradient Boosting Trees

Let $\mathbf{x}_i = (x_{i1}, x_{i2}, x_{i3}, ..., x_{ij})$ represent a vector of observed values for the i^{th} observation on j predictor variables, and y_i is the value of the target outcome for the i^{th} observation. A gradient boosted tree model is an ensemble of T different tree models sequentially developed and the final prediction of the outcome is obtained by using an additive function as

$$\hat{y}_i = \sum_{t=1}^{T} f_t(\mathbf{x}_i),$$

where f_t is a tree model obtained at Iteration t from the residuals at Iteration $t-1$.

The algorithm optimizes an objective function $\mathfrak{L}(\mathbf{y}, \hat{\mathbf{y}})$ in an additive manner. This objective loss function can be defined as sum of squared errors when the outcome is continuous or logistic loss when the outcome is categorical.

The algorithm starts with a constant prediction. For instance, in the above example, we start with the average outcome. Then, a new tree model that minimizes the objective loss function is searched and added at each iteration.

$$\hat{y}_i^{(0)} = \bar{y}$$
$$\hat{y}_i^{(1)} = \hat{y}_i^{(0)} + \alpha f_1(\mathbf{x}_i)$$

$$\hat{y}_i^{(2)} = \hat{y}_i^{(1)} + \alpha f_2(\mathbf{x}_i)$$

$$\vdots$$

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \alpha f_t(\mathbf{x}_i)$$

Notice that I added a multiplier, α, while adding our predictions at each iteration. In the above example, we simply fixed this multiplier to 1, $\alpha = 1$, as we added the whole of new prediction to the previous prediction. This is not necessary. We can also choose to add only a fraction of new predictions (e.g., $\alpha = 0.1, 0.05, 0.01, 0.001$) at each iteration. This multiplier in machine learning literature is called **learning rate**. The smaller the learning rate, the more iterations (more tree models) we will need to achieve the same level of performance. So, the number of iterations (number of tree models, T) and the learning rate (α) play in tandem. Both of these parameters are known as the **boosting hyperparameters** and need to be tuned.

Think about choosing a **learning rate** as choosing your speed on a highway and **number of trees** as the time it takes to arrive at your destination. Suppose you are traveling from Eugene to Portland on I-5. If you drive 40 miles/hour, you are less likely to involve in an accident because you are more aware of your surroundings, but it will take 3-4 hours to arrive at your destination. If you are 200 miles/hour, it will only take you an hour to arrive at your destination, assuming you will not have an accident on the way (which is very likely). So, you try to find a speed level that is fast enough to arrive at your destination and also safe enough not to have an accident on your way.

TECHNICAL NOTE

Why do people call it **Gradient** Boosting? It turns out that the updates at each iteration based on the residuals from a previous model is related to the concept of negative gradient (first derivative) of the objective loss function with respect to the predicted values from the previous step.

$$-g_i^t = -\frac{\partial \mathfrak{L}(y_i, \hat{y}_i^{t-1})}{\partial \hat{y}_i^{t-1}} = \hat{y}_i^{(t)} - \hat{y}_i^{(t-1)}$$

The general logic of gradient boosting works as

- take a differentiable loss function, $\mathfrak{L}(\mathbf{y}, \hat{\mathbf{y}})$, that summarizes the distance between observed and predicted values,
- start with an initial model to obtain initial predictions, $f_0(\mathbf{x}_i)$,

- iterate until termination:
  - calculate the negative gradients of the loss funciton with respect to predictions from previous step
  - fit a tree model to the negative gradients
  - update the predictions (with a multiplier, a.k.a learning rate).

Most software use mathematical approximations and computational hocus pocus to do these compuations for faster implementation.

Fitting Gradient Boosting Trees using the `gbm` package

The gradient boosting trees can be fitted using the `gbm` function from the 'gbm package. The code below tries to replicate our example above using the toy dataset.

```
require(gbm)

gbm.model <-   gbm(formula           = target ~ sents,
                   data              = readability_sub,
                   distribution       = 'gaussian',
                   n.trees           = 2,
                   shrinkage         = 1,
                   interaction.depth = 2,
                   n.minobsinnode    = 2,
                   bag.fraction      = 1,
                   cv.folds          = 0,
                   n.cores           = 1)
```

Model and Data:

- `formula`, a description of the outcome and predictive variables in the model using column names
- `data`, name of the data object to look for the variables in the formula statement
- `distribution`, a character to specify the type of objective loss function to optimize. 'gaussian' is typically used for continuous outcomes(minimize the squared error) and 'bernoulli' is typically used for the binary outcomes (minimizes the logistic loss)

Hyperparameters:

- `n.trees`, number of trees to fit (the number of iterations)
- `shrinkage`, learning rate.
- `interaction.depth`, the maximum depth of each tree developed at each iteration
- `n.minobsinnode`, the minimum number of observations in each terminal note of tree models at each iteration

Stochastic Gradient Boosting:

- `bag.fraction`, the proportion of observations to be randomly selected for developing a new tree at each iteration.

In Gradient Boosting Trees, when we develop a new tree model at each iteration, we use all observations (100% of rows). So, we can set `bag.fraction=1` and `gbm` fits a gradient boosting tree model. On the other hand, adding a random component may sometimes help and yield better performance. You can think about this as an integration of Bagging and Boosting. So, we may want to take a random sample of observations to develop a tree model at each iteration. For instance, if you set `bag.fraction=.9`, the algorithm will first randomly sample 90% of the observations at each iteration before fitting the new tree model to residuals from the previous step. When `bag.fraction` is anything lower than 1, this is called Stochastic Gradient Boosting Trees. `bag.fraction` can also be considered as a hyperparameter to tune by trying different values to find an optimal value, or it can be fixed to a certain number.

Cros validation:

- `cv.folds`, number of cross-validation folds to perform.

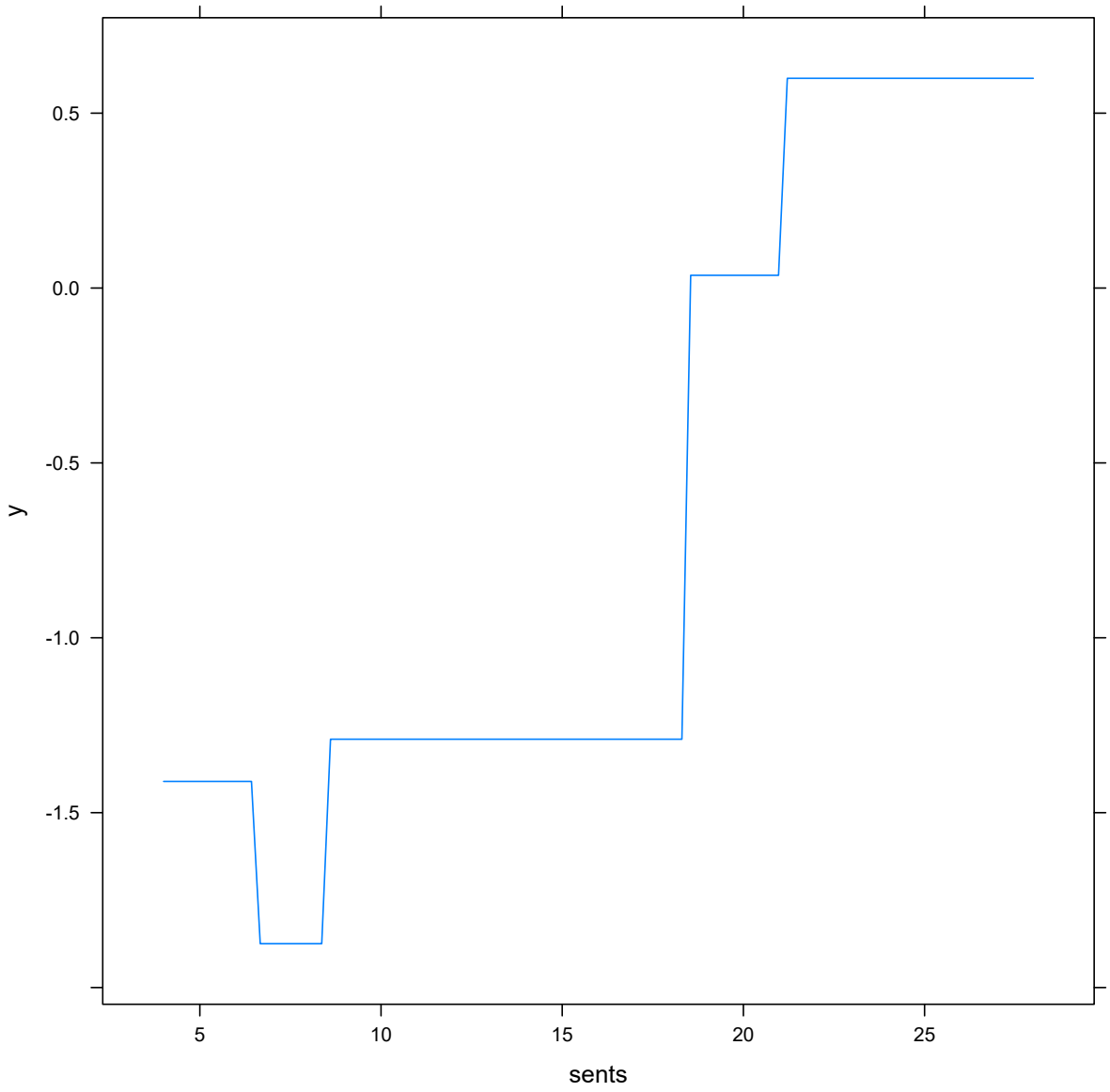
Parallel Processing:

- `n.cores`, the number of CPU cores to use.

```
# Obtain predictions from the model

predict(gbm.model)
```

```
 [1] -1.8746  0.5997 -1.2900 -1.8746 -1.4106 -1.2900 -1.2900 -1.4106 -1.2900
[10]  0.5997 -1.2900 -1.2900 -1.2900 -1.8746  0.0364 -1.2900 -1.4106 -1.4106
[19] -1.8746  0.0364
```

```
# Plot the final model

plot(gbm.model)
```



Fitting Gradient Boosting Trees using the `caret` package and Hyperparameter Tuning

The `gbm` algorithm is available in the `caret` package. Let's check the hyperparameters available to tune.

```
require(caret)

getModelInfo()$gbm$parameters
```

```
          parameter   class                     label
1           n.trees numeric   # Boosting Iterations
2 interaction.depth numeric          Max Tree Depth
```

```
3         shrinkage numeric               Shrinkage
4    n.minobsinnode numeric Min. Terminal Node Size
```

The most important four parameters are all available to tune. Note that it is very challenging to find the best combination of values for all these four hyperparameters unless you implement a full grid search which may take a very long time. You may appply a general sub-optimal strategy to tune the hyperparameters step by step either in pairs or one by one. Below is one way to implement such strategy:

1. Fix the `interaction.depth` and `n.minobsinnode` to a certain value (e.g., interaction.depth = 5, n.minobsinnode = 10),
2. Pick a small value of learning rate (`shrinkage`) such as 0.05 or 0.1,
3. Do a grid search and find the optimal number of trees (`n.trees`) using the fixed values at #1 and #2,
4. Fix the `n.trees` at its optimal value from #3, keep `shrinkage` same as in #2, and do a two-dimensional grid search for `interaction.depth` and `n.minobsinnode` and find the optimal number of depth and minimum observation in a terminal node,
5. Fix the `interaction.depth` and `n.minobsinnode`at their optimal values from #4, lower the learning rate and increase the number of trees to see if the model performance can be further improved.
6. Fix `interaction.depth`,`n.minobsinnode`, `shrinkage`, and `n.trees` at their optimal values from previus steps, and do a grid search for `bag.fraction`.

At the link below you will find an interactive app you can play to understand the dynamics among these hyperparameters and optimize them in toy examples.

http://arogozhnikov.github.io/2016/07/05/gradient_boosting_playground.html

Predicting Readability Scores Using Gradient Boosting Trees

First, we import and prepare data for modeling. Then, we split the data into training and test pieces.

```
# Import the dataset

readability <- read.csv('https://raw.githubusercontent.com/uo-datasci-specialization/c4-ml-fall-2021/ma

# Remove the variables with more than 80% missingness

require(finalfit)

missing_    <- ff_glimpse(readability)$Continuous
flag_na     <- which(as.numeric(missing_$missing_percent) > 80)
readability <- readability[,-flag_na]

# Write the recipe

require(recipes)

blueprint_readability <- recipe(x     = readability,
                                vars  = colnames(readability),
                                roles = c(rep('predictor',990),'outcome')) %>%
  step_zv(all_numeric()) %>%
  step_nzv(all_numeric()) %>%
  step_impute_mean(all_numeric()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_corr(all_numeric(),threshold=0.9)
```

```
# Train/Test Split

set.seed(10152021)  # for reproducibility

  loc      <- sample(1:nrow(readability), round(nrow(readability) * 0.9))
  read_tr  <- readability[loc, ]
  read_te  <- readability[-loc, ]
```

Prepare the data partitions for 10-fold cross validation.

```
# Cross validation settings

read_tr = read_tr[sample(nrow(read_tr)),]

# Create 10 folds with equal size

folds = cut(seq(1,nrow(read_tr)),breaks=10,labels=FALSE)

# Create the list containing the row indices for each fold

my.indices <- vector('list',10)
for(i in 1:10){
  my.indices[[i]] <- which(folds!=i)
}

cv <- trainControl(method = "cv",
                   index  = my.indices)
```

Set the multiple cores for parallel processing.

```
require(doParallel)

ncores <- 10

cl <- makePSOCKcluster(ncores)

registerDoParallel(cl)
```

Step 1: Tune the number of trees

Now, we will fix the learning rate at 0.1 (`shrinkage=0.1`), interaction depth at 5 (`interaction.depth=5`), and minimum number of observations at 10 (`n.minobsinnode = 10`). We will do a grid search for number of trees from 1 to 1000 (`n.trees = 1:1000`). Note that I fix the bag fraction at 1 and pass it as an argument in the `caret::train` function because it is not allowed in the hyperparameter grid.

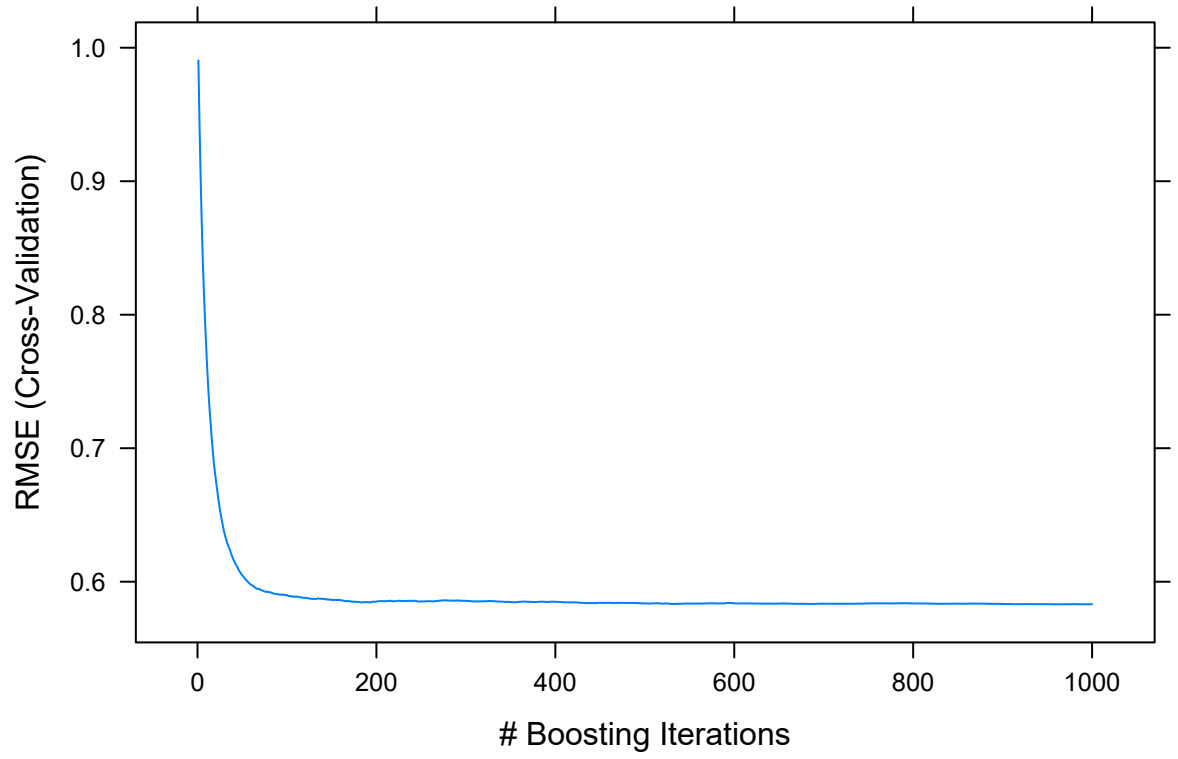
```
# Grid Settings

grid <- expand.grid(shrinkage         = 0.1,
                    n.trees           = 1:1000,
                    interaction.depth = 5,
                    n.minobsinnode    = 10)
```

```
gbm1 <- caret::train(blueprint_readability,
                     data         = read_tr,
                     method       = 'gbm',
                     trControl    = cv,
                     tuneGrid     = grid,
                     bag.fraction = 1,
                     verbose      = FALSE)

gbm1$times
```

```
$everything
   user  system elapsed
 176.45    1.31  354.86

$final
   user  system elapsed
  156.3     0.0   156.3

$prediction
[1] NA NA NA
```

This took about 6 minutes to run. We now look at the plot, and how the cross-validated RMSE changes as a function of number of trees.

```
plot(gbm1,type='l')
```



This indicates that there is not much improvement after 200 trees with these settings (this is just eyeballing, nothing specific about how to come up with this number). So, I will fix the number of trees to 200 for the next step.

Step 2: Tune the interaction depth and minimum number of observations

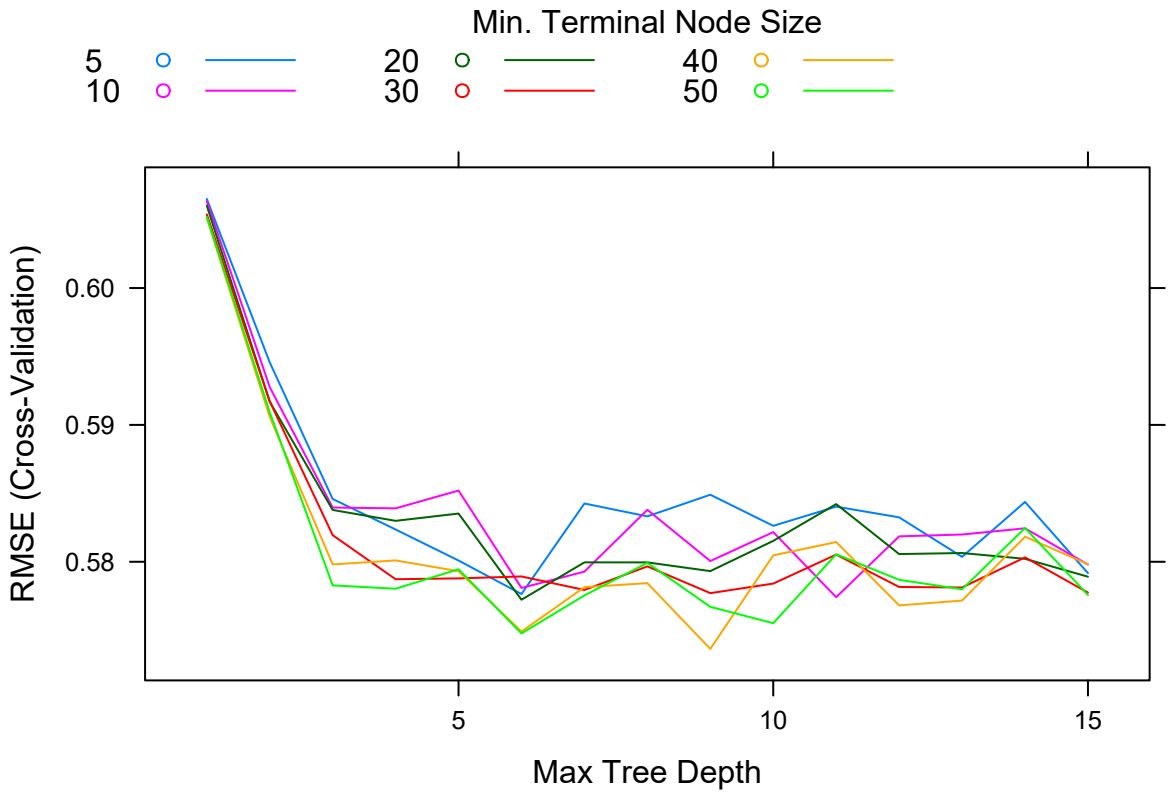
Now, we will fix the number of trees at 200 (`n.trees = 200`) and learning rate at 0.1 (`shrinkage=0.1`). Then, we will do a grid search by assigning values for the interaction depth from 1 to 15 and values for the minimum number of observations at 5, 10, 20, 30, 40, and 50. We still keep bag fraction as 1.

```
grid <- expand.grid(shrinkage         = 0.1,
                    n.trees           = 200,
                    interaction.depth = 1:15,
                    n.minobsinnode    = c(5,10,20,30,40,50))


gbm2 <- caret::train(blueprint_readability,
                     data      = read_tr,
                     method    = 'gbm',
                     trControl = cv,
                     tuneGrid  = grid,
                     bag.fraction = 1,
                     verbose = FALSE)


gbm2$times
```

```
$everything
   user  system elapsed
  92.92    4.94 6202.95

$final
   user  system elapsed
  69.83    0.14   69.84

$prediction
[1] NA NA NA
```

This search took about 1 hour and 43 minutes. If we look at the cross-validates RMSE for all these 90 possible conditions, we see that the best result comes out when interaction depth is equal to 9 and minimum number of observations is equal to 40.

```
plot(gbm2,type='l')
```




```
gbm2$bestTune
```

```
   n.trees interaction.depth shrinkage n.minobsinnode
53     200                 9       0.1             40
```

```
gbm3$results[which.min(gbm3$results$RMSE),]
```

```
     shrinkage interaction.depth n.minobsinnode n.trees   RMSE Rsquared    MAE
3377      0.01                 9             40    3377 0.5678    0.701 0.4506
      RMSESD RsquaredSD   MAESD
3377 0.02724    0.02495 0.02143
```

Step 3: Lower the learning rate and increase the number of trees

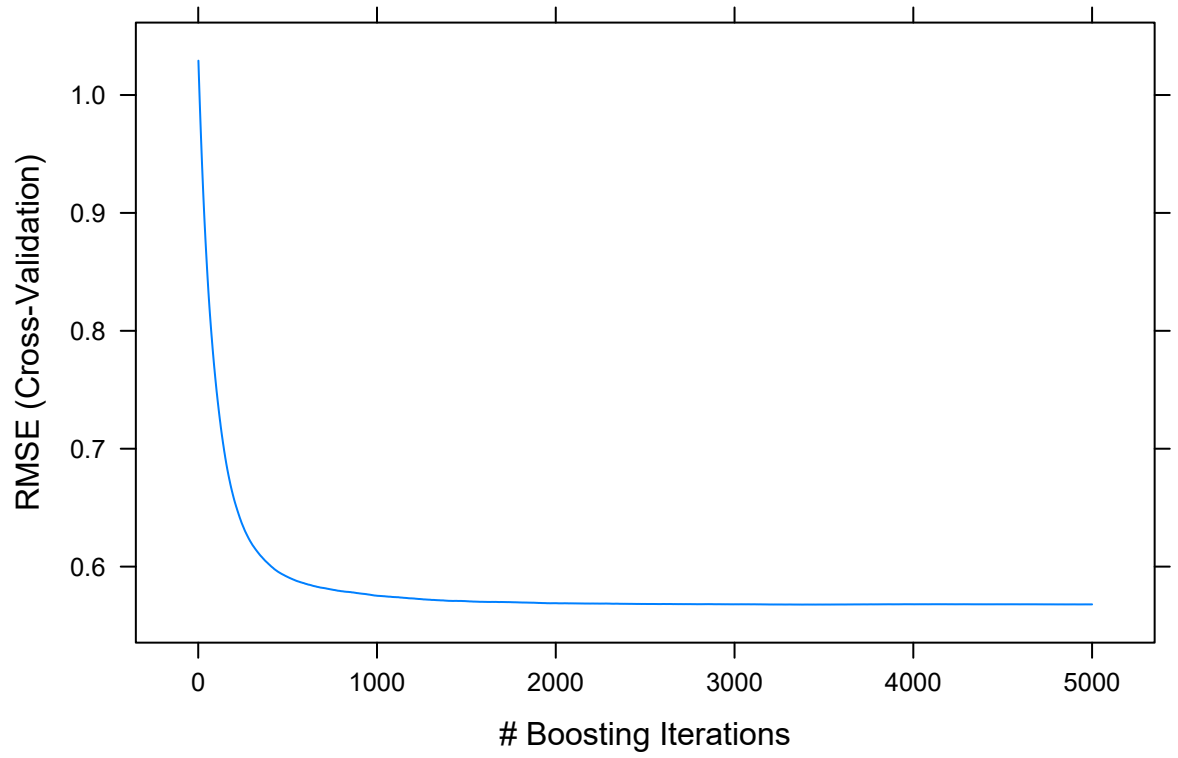
Now, we will fix the interaction depth at 9 (`interaction.depth = 9`) and minimum number of observations at 40 (`n.minobsinnode = 40`). We will lower the learning rate to 0.01 (`shrinkage=0.01`) and increase the number of trees to 5000 (`n.trees = 1:5000`) to explore if lower learning rate improves the performance.

```
grid <- expand.grid(shrinkage         = 0.01,
                    n.trees           = 1:5000,
                    interaction.depth = 9,
                    n.minobsinnode    = 40)
```

```
gbm3 <- caret::train(blueprint_readability,
                     data       = read_tr,
                     method     = 'gbm',
                     trControl = cv,
                     tuneGrid   = grid,
                     bag.fraction = 1,
                     verbose= FALSE)

gbm3$times
```

```
$everything
   user  system elapsed
 839.83    6.74 2224.38

$final
   user  system elapsed
 818.95    0.85  819.30

$prediction
[1] NA NA NA
```

```
plot(gbm3,type='l')
```



```
gbm3$bestTune
```

```
     n.trees interaction.depth shrinkage n.minobsinnode
3377    3377                 9      0.01             40
```

```
gbm3$results[which.min(gbm3$results$RMSE),]
```

```
     shrinkage interaction.depth n.minobsinnode n.trees   RMSE Rsquared    MAE
3377      0.01                 9             40    3377 0.5678    0.701 0.4506
      RMSESD RsquaredSD   MAESD
3377 0.02724    0.02495 0.02143
```

This run took about another 37 minutes. The best performance was obtained with a model 3377 trees, and yielded an RMSE valueof 0.5678. We can stop here, and decide that this is our final model. Or, we can play with `bag.fraction` and see we can squeeze a little bit more.

Step 4: Tune Bag Fraction

In order to play with the `bag.fraction`, we should write our own syntax as `caret::train` doesn allow it to be manipulated as a hyperparameter.

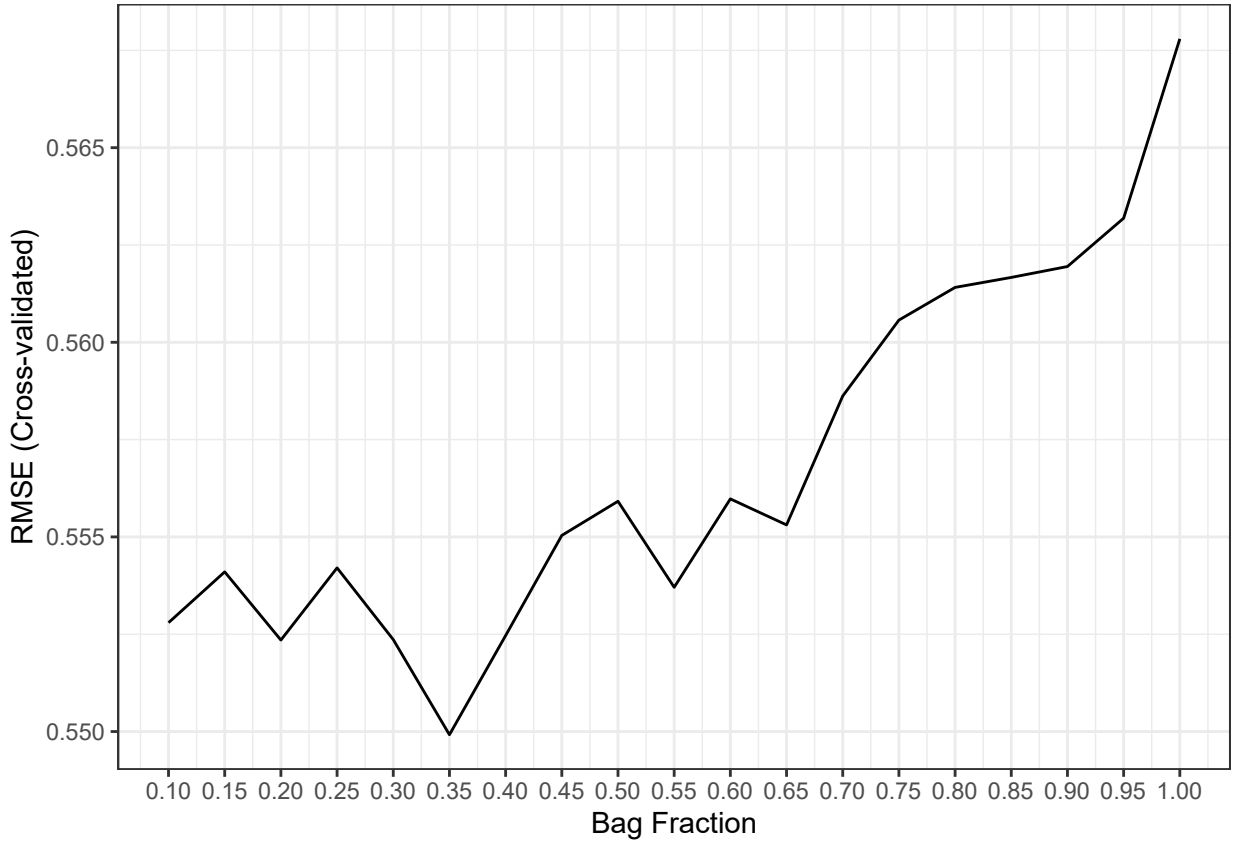
Notice that I fixed the values of `shrinkage`, `n.trees`,`interaction.depth`,`n.minobsinnode` at their optimal values. Then, I write a `for` loop to iterate over different values of `bag.fraction` from 0.1 to 1 with increments of 0.05. I save the model object from each iteration in a list object.

```
grid <- expand.grid(shrinkage         = 0.01,
                    n.trees           = 3377,
                    interaction.depth = 9,
                    n.minobsinnode    = 40)

bag.fr <- seq(0.1,1,.05)

my.models <- vector('list',length(bag.fr))

for(i in 1:length(bag.fr)){

  my.models[[i]] <- caret::train(blueprint_readability,
                                 data      = read_tr,
                                 method    = 'gbm',
                                 trControl = cv,
                                 tuneGrid  = grid,
                                 bag.fraction = bag.fr[i],
                                 verbose= FALSE)
}
```

This took about 8 hours to complete with 10 cores. Let's check if it improved the performance.

```
cv.rmse <- c()

for(i in 1:length(bag.fr)){
  cv.rmse[i] <- my.models[[i]]$results$RMSE
}
```

```
ggplot()+
  geom_line(aes(x=bag.fr,y=cv.rmse))+
  theme_bw()+
  xlab('Bag Fraction')+
  ylab('RMSE (Cross-validated)')+
  scale_x_continuous(breaks = bag.fr)
```



The best performance was obtained when `bag.fr` is equal to 0.35, so 35% of the observations randomly sampled at each iteration.

Finally, we can check the performance of the final model with these settings on the test dataset and compare it to other methods.

```
final.gbm <- my.models[[6]]

# Predictions from a Bagged tree model with 158 trees

predicted_te <- predict(final.gbm,read_te)

# MAE

mean(abs(read_te$target - predicted_te))
```

```
[1] 0.4332
```

```
# RMSE

sqrt(mean((read_te$target - predicted_te)^2))
```

```
[1] 0.538
```

```
# R-square

cor(read_te$target,predicted_te)^2
```

```
[1] 0.7244
```

	R-square	MAE	RMSE
Ridge Regression	0.727	0.435	0.536
Lasso Regression	0.725	0.434	0.538
Gradient Boosting	0.724	0.433	0.538
Random Forests	0.671	0.471	0.596
Bagged Trees	0.656	0.481	0.604
Linear Regression	0.644	0.522	0.644
KNN	0.623	0.500	0.629
Decision Tree	0.497	0.577	0.729

Predicting Recidivism Using Gradient Boosting Trees

The code below implements a similar strategy and demonstrates step by step how to fit a Gradient Boosting Tree model for the Recidivism dataset to predict recidivism in Year 2.

** Import the dataset, and and initial data preparation

```
# Import data

  recidivism <- read.csv('https://raw.githubusercontent.com/uo-datasci-specialization/c4-ml-fall-2021/ma

# List of variable types in the dataset

  outcome <- c('Recidivism_Arrest_Year2')

  categorical <- c('Residence_PUMA',
                   'Prison_Offense',
                   'Age_at_Release',
                   'Supervision_Level_First',
                   'Education_Level',
                   'Prison_Years',
                   'Gender',
                   'Race',
                   'Gang_Affiliated',
                   'Prior_Arrest_Episodes_DVCharges',
                   'Prior_Arrest_Episodes_GunCharges',
                   'Prior_Conviction_Episodes_Viol',
                   'Prior_Conviction_Episodes_PPViolationCharges',
```

```
                  'Prior_Conviction_Episodes_DomesticViolenceCharges',
                  'Prior_Conviction_Episodes_GunCharges',
                  'Prior_Revocations_Parole',
                  'Prior_Revocations_Probation',
                  'Condition_MH_SA',
                  'Condition_Cog_Ed',
                  'Condition_Other',
                  'Violations_ElectronicMonitoring',
                  'Violations_Instruction',
                  'Violations_FailToReport',
                  'Violations_MoveWithoutPermission',
                  'Employment_Exempt')

  numeric   <- c('Supervision_Risk_Score_First',
                 'Dependents',
                 'Prior_Arrest_Episodes_Felony',
                 'Prior_Arrest_Episodes_Misd',
                 'Prior_Arrest_Episodes_Violent',
                 'Prior_Arrest_Episodes_Property',
                 'Prior_Arrest_Episodes_Drug',
                 'Prior_Arrest_Episodes_PPViolationCharges',
                 'Prior_Conviction_Episodes_Felony',
                 'Prior_Conviction_Episodes_Misd',
                 'Prior_Conviction_Episodes_Prop',
                 'Prior_Conviction_Episodes_Drug',
                 'Delinquency_Reports',
                 'Program_Attendances',
                 'Program_UnexcusedAbsences',
                 'Residence_Changes',
                 'Avg_Days_per_DrugTest',
                 'Jobs_Per_Year')

  props      <- c('DrugTests_THC_Positive',
                  'DrugTests_Cocaine_Positive',
                  'DrugTests_Meth_Positive',
                  'DrugTests_Other_Positive',
                  'Percent_Days_Employed')

# Convert all nominal, ordinal, and binary variables to factors
# Leave the rest as is

  for(i in categorical){

    recidivism[,i] <- as.factor(recidivism[,i])

  }

# For variables that represent proportions, add/substract a small number
# to 0s/1s for logit transformation

  for(i in props){
    recidivism[,i] <- ifelse(recidivism[,i]==0,.0001,recidivism[,i])
    recidivism[,i] <- ifelse(recidivism[,i]==1,.9999,recidivism[,i])
```

```
  }

# Blueprint for processing variables

  require(recipes)

  blueprint_recidivism <- recipe(x  = recidivism,
                                 vars  = c(categorical,numeric,props,outcome),
                                 roles = c(rep('predictor',48),'outcome')) %>%
    step_indicate_na(all_of(categorical),all_of(numeric),all_of(props)) %>%
    step_zv(all_numeric()) %>%
    step_impute_mean(all_of(numeric),all_of(props)) %>%
    step_impute_mode(all_of(categorical)) %>%
    step_logit(all_of(props)) %>%
    step_ns(all_of(numeric),all_of(props),deg_free=3) %>%
    step_normalize(paste0(numeric,'_ns_1'),
                   paste0(numeric,'_ns_2'),
                   paste0(numeric,'_ns_3'),
                   paste0(props,'_ns_1'),
                   paste0(props,'_ns_2'),
                   paste0(props,'_ns_3')) %>%
    step_dummy(all_of(categorical),one_hot=TRUE)  %>%
    step_num2factor(Recidivism_Arrest_Year2,
                    transform = function(x) x + 1,
                    levels=c('No','Yes'))
```

Train/Test Split, and Cross-validation Settings

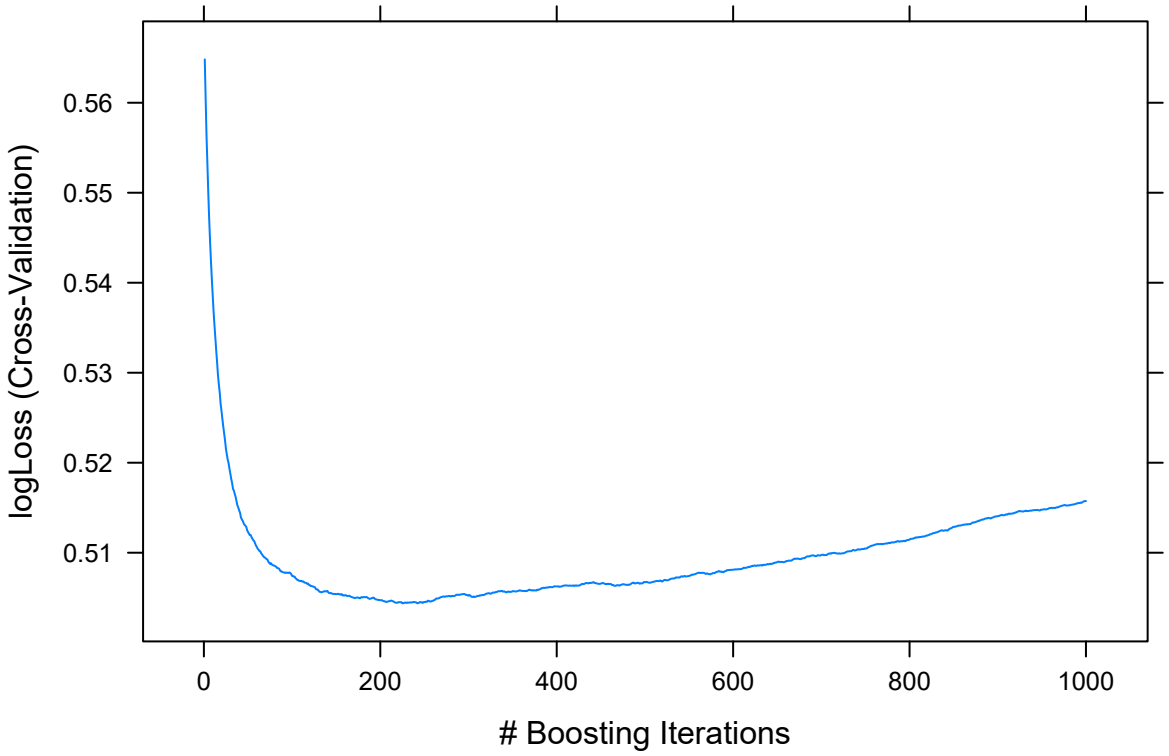
```
# Train/Test Split

  loc <- which(recidivism$Training_Sample==1)

  recidivism_tr  <- recidivism[loc, ]
  recidivism_te  <- recidivism[-loc, ]


# Cross validation settings

  set.seed(10302021) # for reproducibility

  recidivism_tr = recidivism_tr[sample(nrow(recidivism_tr)),]

  # Create row indices for 10 folds with equal size

  folds = cut(seq(1,nrow(recidivism_tr)),breaks=10,labels=FALSE)

  # Create the list object with each element representing
  # the row indices for each fold

  my.indices <- vector('list',10)
  for(i in 1:10){
    my.indices[[i]] <- which(folds!=i)
  }
```

```
  cv <- trainControl(method = "cv",
                     index  = my.indices,
                     classProbs = TRUE,
                     summaryFunction = mnLogLoss)

      # Because the outcome is binary, I specify classProbs=TRUE and
      # summaryFunction = mnLogloss, so we will minimize the negative LogLoss
      # based on predicted class probabilities
```

Initial model fit to tune the number of trees

We fix the learning rate at 0.1 (`shrinkage=0.1`), interaction depth at 5 (interaction.depth=5), and minimum number of observations at 10 (`n.minobsinnode = 10`). We do a grid search for optimal number of trees from 1 to 1000 (`n.trees = 1:1000`).

```
grid <- expand.grid(shrinkage         = 0.1,
                    n.trees           = 1:1000,
                    interaction.depth = 5,
                    n.minobsinnode    = 10)


gbm1 <- caret::train(blueprint_recidivism,
                     data         = recidivism_tr,
                     method       = 'gbm',
                     trControl    = cv,
                     tuneGrid     = grid,
                     bag.fraction = 1,
                     metric       = 'logLoss')

plot(gbm1,type='l')

gbm1$bestTune
```




```
    n.trees interaction.depth shrinkage n.minobsinnode
242     242                 5       0.1             10
```

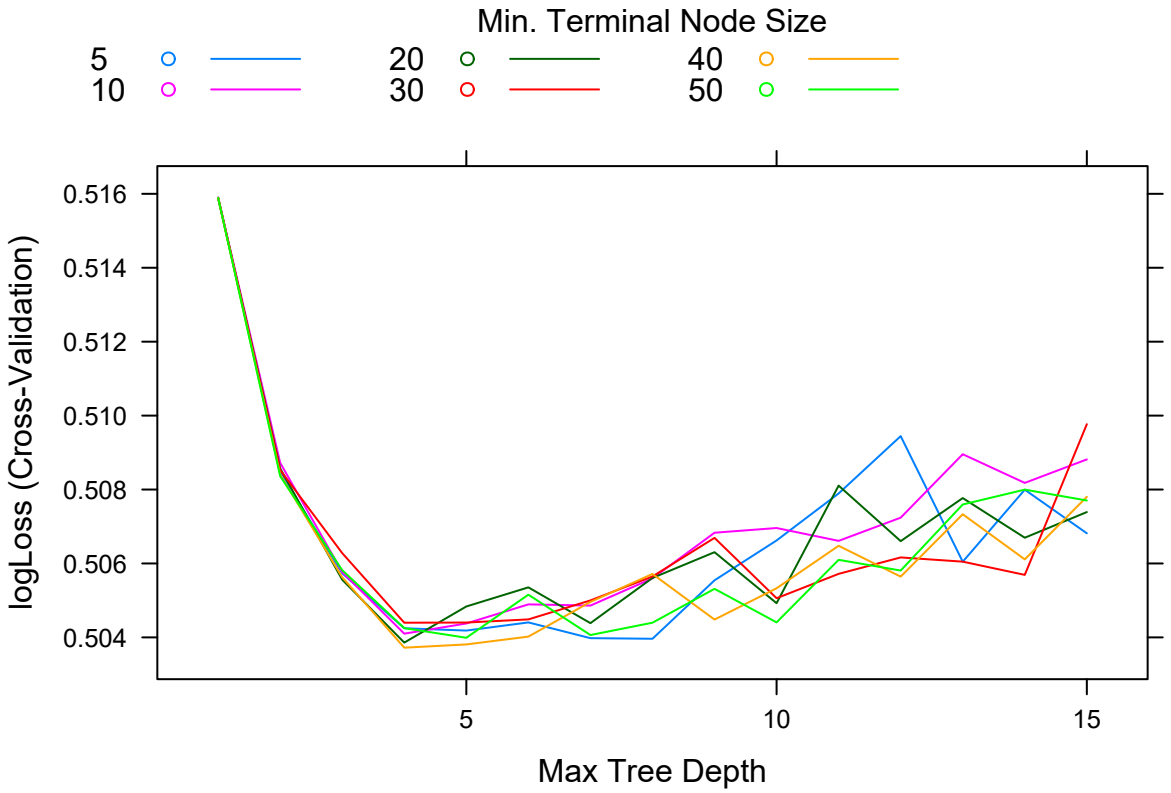
This indicates that 242 trees is the optimal at the initial search.

Tune the interaction depth and minimum number of observations

We fix the number of trees at 242 (`n.trees = 242`) and learning rate at 0.1 (`shrinkage=0.1`). Then, we do a grid search by assigning values for the interaction depth from 1 to 15 and values for the minimum number of observations at 5, 10, 20, 30, 40, and 50.

```
grid <- expand.grid(shrinkage         = 0.1,
                    n.trees           = 242,
                    interaction.depth = 1:15,
                    n.minobsinnode    = c(5,10,20,30,40,50))


gbm2 <- caret::train(blueprint_recidivism,
                     data         = recidivism_tr,
                     method       = 'gbm',
                     trControl    = cv,
                     tuneGrid     = grid,
                     bag.fraction = 1,
                     metric       = 'logLoss')
```

```
plot(gbm2)

gbm2$bestTune
```




```
   n.trees interaction.depth shrinkage n.minobsinnode
23     242                 4       0.1             40
```

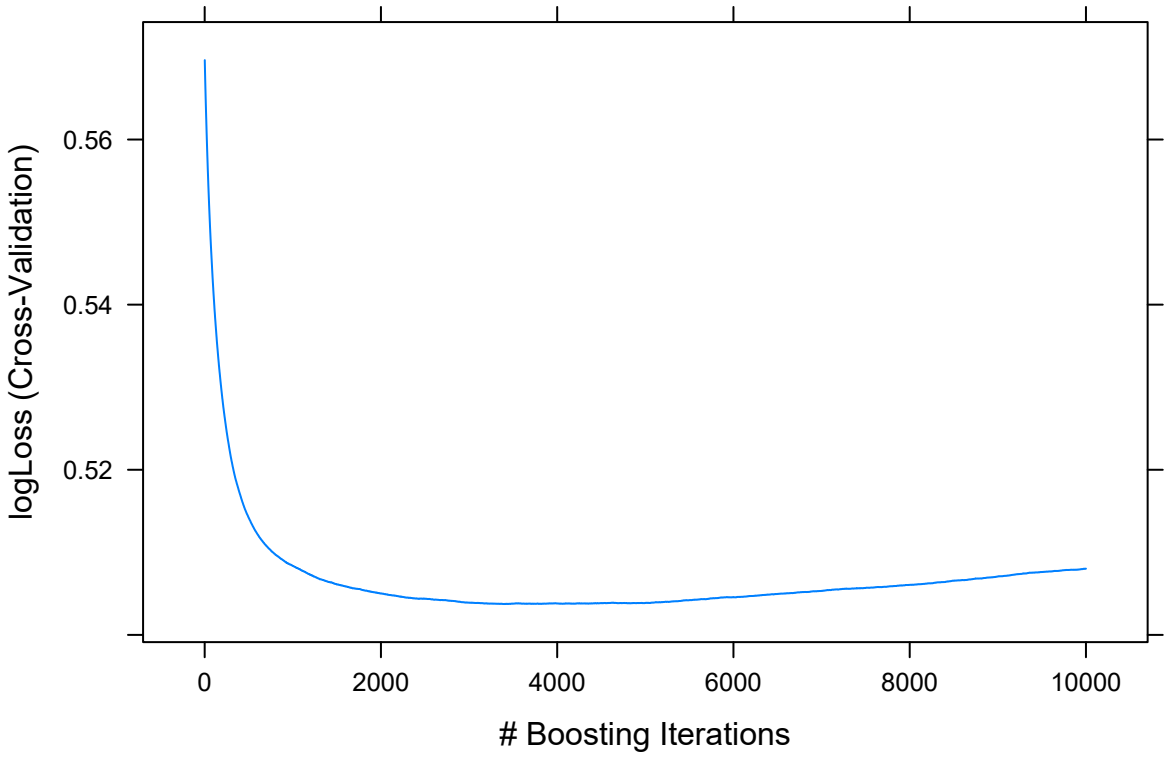
The search indicates that the best performance is obtained when the interaction depth is equal to 4 and minimum number of observations is equal to 40.

Lower the learning rate and increase the number of trees

We fix the interaction depth at 4 (`interaction.depth = 4`) and minimum number of observations at 40 (`n.minobsinnode = 40`). We will lower the learning rate to 0.01 (shrinkage=0.01) and increase the number of trees to 10000 (`n.trees = 1:10000`) to explore if lower learning rate improves the performance.

```
grid <- expand.grid(shrinkage         = 0.01,
                    n.trees           = 1:10000,
                    interaction.depth = 4,
                    n.minobsinnode    = 40)


gbm3 <- caret::train(blueprint_recidivism,
                     data         = recidivism_tr,
                     method       = 'gbm',
```

```
                    trControl    = cv,
                    tuneGrid     = grid,
                    bag.fraction = 1,
                    metric       = 'logLoss')

plot(gbm3,type='l')

gbm3$bestTune
```



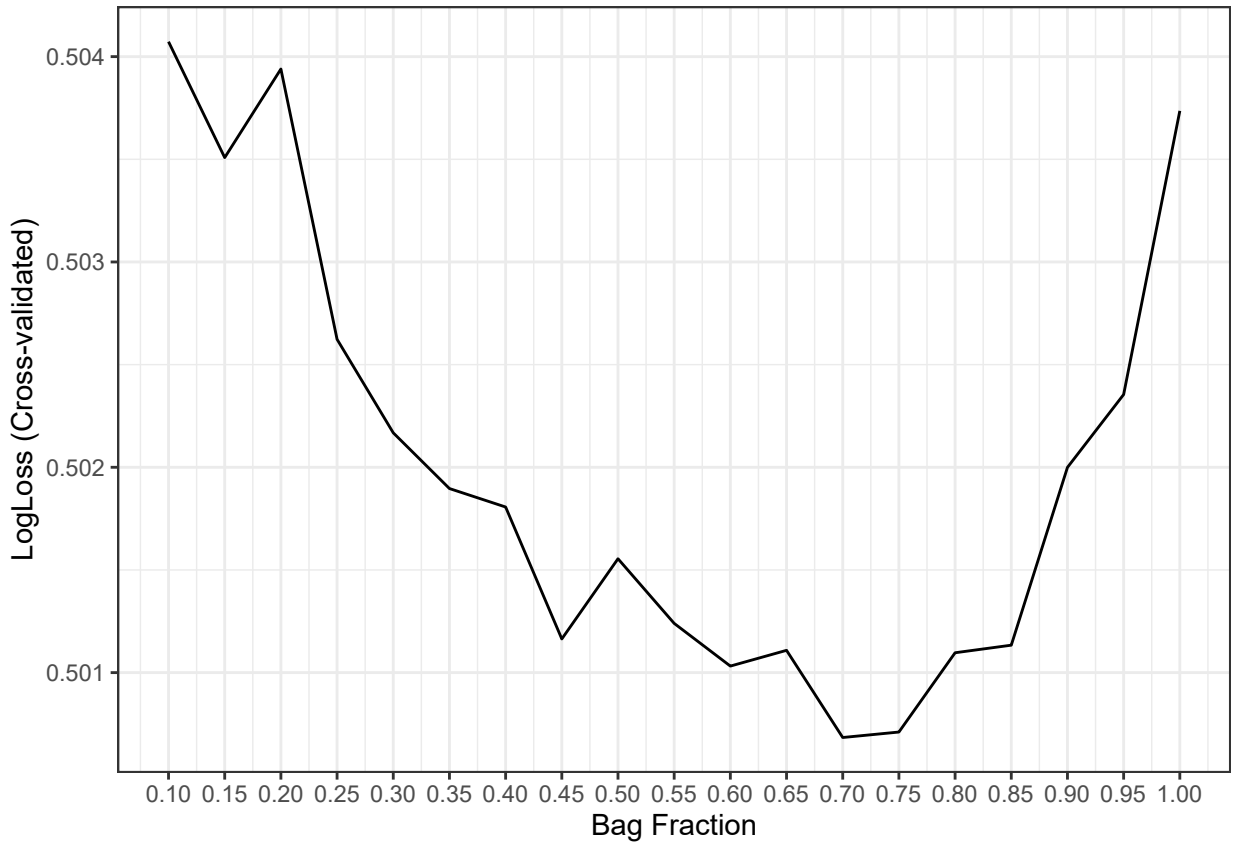

```
     n.trees interaction.depth shrinkage n.minobsinnode
3397    3397                 4      0.01             40
```

Tune the bag fraction

```
grid <- expand.grid(shrinkage         = 0.01,
                    n.trees           = 3397,
                    interaction.depth = 4,
                    n.minobsinnode    = 40)

bag.fr <- seq(0.1,1,.05)

my.models <- vector('list',length(bag.fr))

for(i in 1:length(bag.fr)){
```

```
  my.models[[i]] <- caret::train(blueprint_recidivism,
                                 data       = recidivism_tr,
                                 method     = 'gbm',
                                 trControl  = cv,
                                 tuneGrid   = grid,
                                 bag.fraction = bag.fr[i])
}
```

```
cv.LogL <- c()

for(i in 1:length(bag.fr)){
  cv.LogL[i] <- my.models[[i]]$results$logLoss
}

ggplot()+
  geom_line(aes(x=bag.fr,y=cv.LogL))+
  theme_bw()+
  xlab('Bag Fraction')+
  ylab('LogLoss (Cross-validated)')+
  scale_x_continuous(breaks = bag.fr)
```



The best result is obtained when the bag fraction is 0.7. So, we will proceed with that as our final model.

Final Predictions

```r
final.gbm <- my.models[[13]]

# Predict the probabilities for the observations in the test dataset

predicted_te <- predict(final.gbm, recidivism_te, type='prob')

head(predicted_te)
```

```
      No     Yes
1 0.9321 0.06793
2 0.5283 0.47173
3 0.7162 0.28380
4 0.5000 0.50001
5 0.7985 0.20145
6 0.8021 0.19785
```

```r
# Compute the AUC

require(cutpointr)

cut.obj <- cutpointr(x     = predicted_te$Yes,
                     class = recidivism_te$Recidivism_Arrest_Year2)

auc(cut.obj)
```

```
[1] 0.7364
```

```r
# Confusion matrix assuming the threshold is 0.5

pred_class <- ifelse(predicted_te$Yes>.5,1,0)

confusion <- table(recidivism_te$Recidivism_Arrest_Year2,pred_class)

confusion
```

```
   pred_class
       0    1
  0 3946  200
  1 1098  216
```

```r
# True Negative Rate

confusion[1,1]/(confusion[1,1]+confusion[1,2])
```

```
[1] 0.9518
```

```r
# False Positive Rate

confusion[1,2]/(confusion[1,1]+confusion[1,2])
```

```
[1] 0.04824
```

```
# True Positive Rate

confusion[2,2]/(confusion[2,1]+confusion[2,2])
```

```
[1] 0.1644
```

```
# Precision

confusion[2,2]/(confusion[1,2]+confusion[2,2])
```

```
[1] 0.5192
```

Comparison of Results

	-LL	AUC	ACC	TPR	TNR	FPR	PRE
Gradient Boosted Trees	0.5007	0.7364	0.762	0.164	0.952	0.048	0.519
Random Forests	0.5097	0.7242	0.762	0.126	0.963	0.037	0.519
Bagged Trees	0.5088	0.7225	0.758	0.130	0.957	0.043	0.489
Logistic Regression with Lasso Penalty	0.5090	0.7200	0.754	0.127	0.952	0.048	0.458
Logistic Regression with Elastic Net	0.5091	0.7200	0.753	0.127	0.952	0.048	0.456
Logistic Regression	0.5096	0.7192	0.755	0.142	0.949	0.051	0.471
Logistic Regression with Ridge Penalty	0.5111	0.7181	0.754	0.123	0.954	0.046	0.461
Decision Tree	0.5522	0.6521	0.752	0.089	0.962	0.038	0.427