Supervised Learning - Trees, Bagging, Boosting

Jan-Philipp Kolb

17 Mai, 2019

s ## What is supervised learning?

Supervised learning includes tasks for “labeled” data (i.e. you have a target variable).

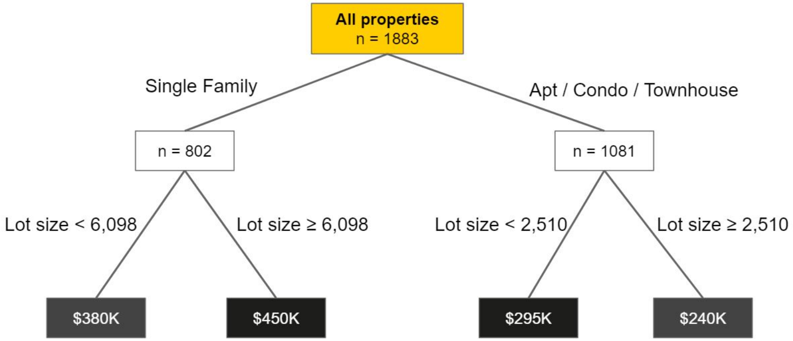
- In practice, it’s often used as an advanced form of predictive modeling.
- Each observation must be labeled with a “correct answer.”
- Only then can you build a predictive model because you must tell the algorithm what’s “correct” while training it (hence, “supervising” it).
- Regression is the task for modeling continuous target variables.
- Classification is the task for modeling categorical (a.k.a. “class”) target variables.

Tree-Based Models

- Trees are good for interpretation because they are simple
- Tree based methods involve stratifying or segmenting the predictor space into a number of simple regions. (Hastie and Tibshirani)
- These methods do not deliver the best results concerning prediction accuracy.

Explanation: **decision tree**

Decision trees model data as a “tree” of hierarchical branches. They make branches until they reach “leaves” that represent predictions.



Summary decission trees

Due to their branching structure, decision trees can easily model nonlinear relationships.

- For example, let's say for Single Family homes, larger lots command higher prices.
- However, let's say for Apartments, smaller lots command higher prices (i.e. it's a proxy for urban / rural).
- This reversal of correlation is difficult for linear models to capture unless you explicitly add an interaction term (i.e. you can anticipate it ahead of time).
- On the other hand, decision trees can capture this relationship naturally.

```r
library(rpart)
```

Regression Trees - preparation

- The following slides are based on

```
library(rsample)     # data splitting
library(dplyr)       # data wrangling
library(rpart)       # performing regression trees
library(rpart.plot)  # plotting regression trees
library(ipred)       # bagging
library(caret)       # bagging
```

The Ames Housing data

```r
set.seed(123)
ames_split <- initial_split(AmesHousing::make_ames(), prop = .7)
ames_train <- training(ames_split)
ames_test  <- testing(ames_split)
```

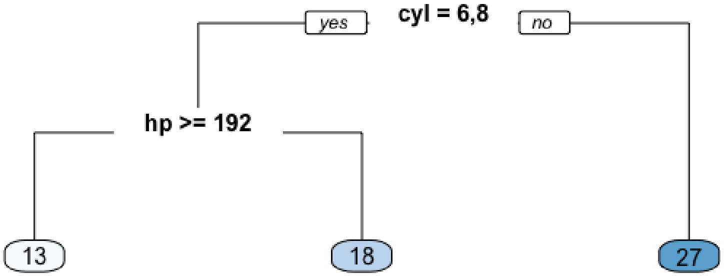
Basic regression trees partition a data set into smaller groups and then fit a simple model (constant) for each subgroup. Unfortunately, a single tree model tends to be highly unstable and a poor predictor. However, by bootstrap aggregating (bagging) regression trees, this technique can become quite powerful and effective. Moreover, this provides the fundamental basis of more complex tree-based models such as random forests and gradient boosting machines. This tutorial will get you started with regression trees and bagging.

The Idea

There are many methodologies for constructing regression trees but one of the oldest is known as the classification and regression tree (CART) approach developed by Breiman et al. (1984). This tutorial focuses on the regression part of CART. Basic regression trees partition a data set into smaller subgroups and then fit a simple constant for each observation in the subgroup. The partitioning is achieved by successive binary partitions (aka recursive partitioning) based on the different predictors. The constant to predict is based on the average response values for all observations that fall in that subgroup.

For example, consider we want to predict the miles per gallon a car will average based on cylinders (cyl) and horsepower (hp). All observations go through this tree, are assessed at a particular node, and proceed to the left if the answer is "yes" or proceed to the right if the answer is "no". So, first, all observations that have 6 or 8 cylinders go to the left branch, all other observations proceed to the right branch. Next, the left branch is further partitioned by horsepower. Those 6 or 8 cylinder observations with horsepower equal to or greater than 192 proceed to the left branch; those with less than 192 hp proceed to the right. These branches lead to terminal nodes or leafs which contain our predicted response value. Basically, all observations (cars in this example) that do not have 6 or 8 cylinders (far right branch) average 27 mpg. All observations that have 6 or 8 cylinders and have more than 192 hp (far left branch) average 13 mpg.

Predicting mpg based on cyl and hp.



This simple example can be generalized to state we have a continuous response variable Y and two inputs X_1 and X_2. The recursive partitioning results in three regions (R_1,R_2,R_3) where the model predicts Y with a constant c_m for region R_m:

$$\hat{f}(X) = \sum_{m=1}^{3} c_m I(X_1, X_2) \in R_m$$

However, an important question remains of how to grow a regression tree.

Deciding on splits

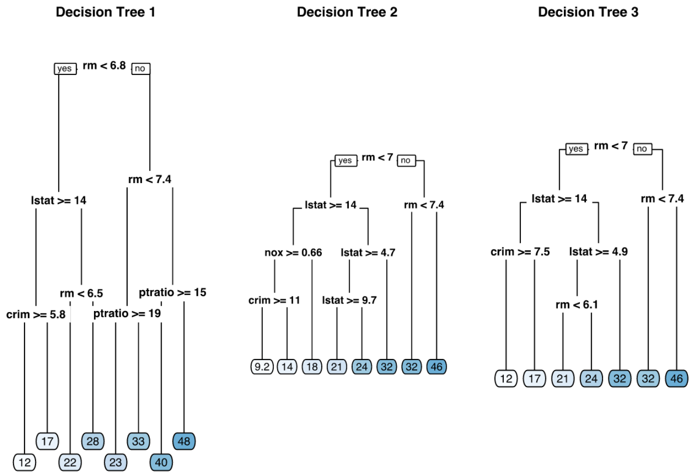
First, its important to realize the partitioning of variables are done in a top-down, greedy fashion. This just means that a partition performed earlier in the tree will not change based on later partitions. But how are these partions made? The model begins with the entire data set, S, and searches every distinct value of every input variable to find the predictor and split value that partitions the data into two regions (R_1 and R_2) such that the overall sums of squares error are minimized:

$$\text{minimize}\{SSE = \sum_{i \in R_1} (y_i - c_1)^2 + \sum_{i \in R_2} (y_i - c_2)^2\}$$

Having found the best split, we partition the data into the two resulting regions and repeat the splitting process on each of the two regions. This process is continued until some stopping criterion is reached. What results is, typically, a very deep, complex tree that may produce good predictions on the training set, but is likely to overfit the data, leading to poor performance on unseen data.

For example, using the well-known **Boston housing data set**, I create three decision trees based on three different samples of the data. You can see that the first few partitions are fairly similar at the top of each tree; however, they tend to differ substantially closer to the terminal nodes. These deeper nodes tend to overfit to specific attributes of the sample data; consequently, slightly different samples will result in highly variable estimate/predicted values in the terminal nodes. By pruning these lower level decision nodes, we can introduce a little bit of bias in our model that help to stabilize predictions and will tend to generalize better to new, unseen data.

Three decision trees based on slightly different samples.



Cost complexity criterion

There is often a balance to be achieved in the depth and complexity of the tree to optimize predictive performance on some unseen data. To find this balance, we typically grow a very large tree as defined in the previous section and then prune it back to find an optimal subtree. We find the optimal subtree by using a cost complexity parameter (α) that penalizes our objective function in Eq. 2 for the number of terminal nodes of the tree (T) as in Eq. 3.

$$\text{minimize}\{SSE + \alpha|T|\}$$

##

For a given value of α, we find the smallest pruned tree that has the lowest penalized error. If you are familiar with regularized regression, you will realize the close association to the lasso L_1 norm penalty. As with these regularization methods, smaller penalties tend to produce more complex models, which result in larger trees. Whereas larger penalties result in much smaller trees. Consequently, as a tree grows larger, the reduction in the SSE must be greater than the cost complexity penalty. Typically, we

Strengths

There are several advantages to regression trees:

- They are very interpretable.
- Making predictions is fast (no complicated calculations, just looking up constants in the tree).
- It's easy to understand what variables are important in making the prediction. The internal nodes (splits) are those variables that most largely reduced the SSE.
- If some data is missing, we might not be able to go all the way down the tree to a leaf, but we can still make a prediction by averaging all the leaves in the sub-tree we do reach.
- The model provides a non-linear “jagged” response, so it can work when the true regression surface is not smooth. If it is smooth, though, the piecewise-constant surface can approximate it arbitrarily closely (with enough leaves).
- There are fast, reliable algorithms to learn these trees.

WEAKNESSES

But there are also some significant weaknesses:

- Single regression trees have high variance, resulting in unstable predictions (an alternative subsample of training data can significantly change the terminal nodes).
- Due to the high variance single regression trees have poor predictive accuracy.

Basic Implementation

We can fit a regression tree using rpart and then visualize it using rpart.plot. The fitting process and the visual output of regression trees and classification trees are very similar. Both use the formula method for expressing the model (similar to lm). However, when fitting a regression tree, we need to set method = "anova". By default, rpart will make an intelligent guess as to what the method value should be based on the data type of your response column, but it's recommened that you explictly set the method for reproducibility reasons (since the auto-guesser may change in the future).

```
m1 <- rpart(
  formula = Sale_Price ~ .,
  data    = ames_train,
  method  = "anova"
  )
```

Once we've fit our model we can take a peak at the m1 output. This just explains steps of the splits. For example, we start with 2051 observations at the root node (very beginning) and the first variable we split on (the first variable that optimizes a reduction in SSE) is Overall_Qual. We see that at the first node all observations with

```
Overall_Qual=Very_Poor,Poor,Fair,Below_Average,Average,Above_Ave
```

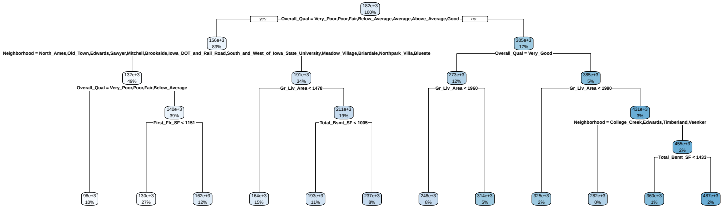
go to the 2nd (2)) branch. The total number of observations that follow this branch (1699), their average sales price (156147.10) and SSE (4.001092e+12) are listed. If you look for the 3rd branch (3)) you will see that 352 observations with `Overall_Qual=Very_Good,Excellent,Very_Excellent` follow this branch and their average sales prices is 304571.10 and the SEE in this region is 2.874510e+12. Basically, this is telling us the most important variable that has the largest reduction in SEE initially is Overall_Qual with those homes on the upper end of the quality spectrum having almost double the average sales price.

```
m1

## n= 2051
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 2051 1.329920e+13 181620.20
##    2) Overall_Qual=Very_Poor,Poor,Fair,Below_Average,Average,
##      4) Neighborhood=North_Ames,Old_Town,Edwards,Sawyer,Mitch
##        8) Overall_Qual=Very_Poor,Poor,Fair,Below_Average 195
##        9) Overall_Qual=Average,Above_Average,Good 805 8.52605
##         18) First_Flr_SF< 1150.5 553 3.023384e+11 129936.80 *
##         19) First_Flr_SF>=1150.5 252 3.743907e+11 161810.90 *
##      5) Neighborhood=College_Creek,Somerset,Northridge_Height
##       10) Gr_Liv_Area< 1477.5 300 2.472611e+11 164045.20 *
##       11) Gr_Liv_Area>=1477.5 399 6.311990e+11 211259.60
##         22) Total_Bsmt_SF< 1004.5 232 1.640427e+11 192946.30
##         23) Total_Bsmt_SF>=1004.5 167 2.812570e+11 236700.80
##    3) Overall_Qual=Very_Good,Excellent,Very_Excellent 352 2.8
```

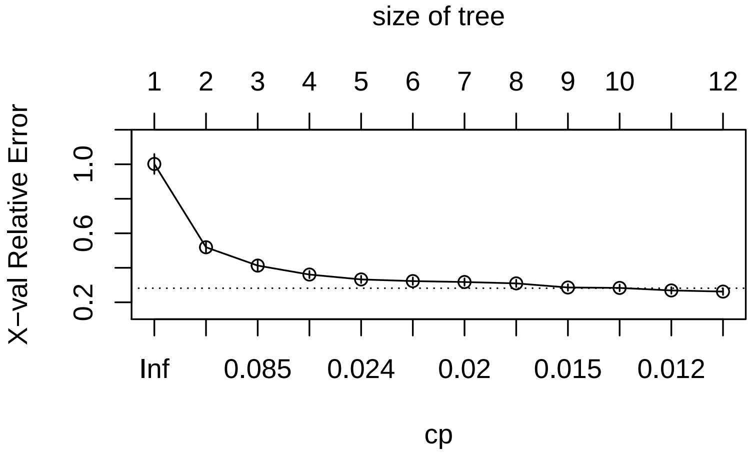
We can visualize our model with rpart.plot. rpart.plot has many plotting options, which we'll leave to the reader to explore. However, in the default print it will show the percentage of data that fall to that node and the average sales price for that branch. One thing you may notice is that this tree contains 11 internal nodes resulting in 12 terminal nodes. Basically, this tree is partitioning on 11 variables to produce its model. - There are 80 variables in ames_train. So what happened?

```
rpart.plot(m1)
```


182e+3
100%
yes
Overall_Qual = Very_Poor,Poor,Fair,Below_Average,Average,Above_Average,Good
no
156e+3
83%
Neighborhood = North_Ames,Old_Town,Edwards,Sawyer,Mitchell,Brookside,Iowa_DOT_and_Rail_Road,South_and_West_of_Iowa_State_University,Meadow_Village,Briardale,Northpark_Villa,Blueste
305e+3
17%
Overall_Qual = Very_Good
132e+3
49%
Overall_Qual = Very_Poor,Poor,Fair,Below_Average
191e+3
34%
Gr_Liv_Area < 1478
273e+3
12%
Gr_Liv_Area < 1960
385e+3
5%
Gr_Liv_Area < 1990
140e+3
39%
First_Flr_SF < 1151
211e+3
19%
Total_Bsmt_SF < 1005
431e+3
3%
Neighborhood = College_Creek,Edwards,Timberland,Veenker
455e+3
2%
Total_Bsmt_SF < 1433
98e+3
10%
130e+3
27%
162e+3
12%
164e+3
15%
193e+3
11%
237e+3
8%
248e+3
8%
314e+3
5%
325e+3
2%
282e+3
0%
360e+3
1%
487e+3
2%

Behind the scenes rpart is automatically applying a range of cost complexity (α values to prune the tree. To compare the error for each α value, rpart performs a 10-fold cross validation so that the error associated with a given α value is computed on the hold-out validation data. In this example we find diminishing returns after 12 terminal nodes as illustrated below (y-axis is cross validation error, lower x-axis is cost complexity (α) value, upper x-axis is the number of terminal nodes (tree size $= |T|$). You may also notice the dashed line which goes through the point $|T| = 9$. Breiman et al. (1984) suggested that in actual practice, its common to instead use the smallest tree within 1 standard deviation of the minimum cross validation error (aka the 1-SE rule). Thus, we could use a tree with 9 terminal nodes and reasonably expect to experience similar results within a small margin of error.

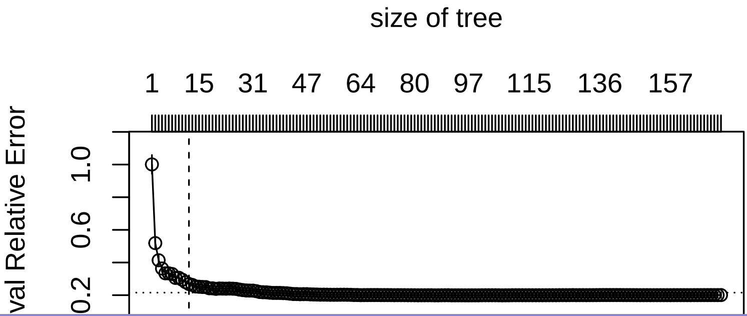
```r
plotcp(m1)
```


size of tree
1
2
3
4
5
6
7
8
9
10
12
X−val Relative Error
0.2
0.6
1.0
Inf
0.085
0.024
0.02
0.015
0.012
cp

To illustrate the point of selecting a tree with 12 terminal nodes (or 9 if you go by the 1-SE rule), we can force rpart to generate a full tree by using cp = 0 (no penalty results in a fully grown tree). We can see that after 12 terminal nodes, we see diminishing returns in error reduction as the tree grows deeper. Thus, we can signifcantly prune our tree and still achieve minimal expected error.

```
m2 <- rpart(
    formula = Sale_Price ~ .,
    data    = ames_train,
    method  = "anova",
    control = list(cp = 0, xval = 10)
)

plotcp(m2)
abline(v = 12, lty = "dashed")
```



So, by default, rpart is performing some automated tuning, with an optimal subtree of 11 splits, 12 terminal nodes, and a cross-validated error of 0.272 (note that this error is equivalent to the PRESS statistic but not the MSE). However, we can perform additional tuning to try improve model performance.

```
m1$cptable
```

```
##            CP nsplit rel error    xerror       xstd
## 1  0.48300624      0 1.0000000 1.0017486 0.05769371
## 2  0.10844747      1 0.5169938 0.5189120 0.02898242
## 3  0.06678458      2 0.4085463 0.4126655 0.02832854
## 4  0.02870391      3 0.3417617 0.3608270 0.02123062
## 5  0.02050153      4 0.3130578 0.3325157 0.02091087
## 6  0.01995037      5 0.2925563 0.3228913 0.02127370
## 7  0.01976132      6 0.2726059 0.3175645 0.02115401
## 8  0.01550003      7 0.2528446 0.3096765 0.02117779
## 9  0.01397824      8 0.2373446 0.2857729 0.01902451
## 10 0.01322455      9 0.2233663 0.2833382 0.01936841
## 11 0.01089820     10 0.2101418 0.2687777 0.01917474
## 12 0.01000000     11 0.1992436 0.2621273 0.01957837
```

Tuning

In addition to the cost complexity (α) parameter, it is also common to tune:

- `minsplit`: the minimum number of data points required to attempt a split before it is forced to create a terminal node. The default is 20. Making this smaller allows for terminal nodes that may contain only a handful of observations to create the predicted value.
- `maxdepth`: the maximum number of internal nodes between the root node and the terminal nodes. The default is 30, which is quite liberal and allows for fairly large trees to be built.

`rpart` uses a special control argument where we provide a list of hyperparameter values. For example, if we wanted to assess a model with minsplit $=$ 10 and maxdepth $=$ 12, we could execute the following:

```
m3 <- rpart(
    formula = Sale_Price ~ .,
    data    = ames_train,
    method  = "anova",
    control = list(minsplit = 10, maxdepth = 12, xval = 10)
)
```

```
m3$cptable
```

```
##            CP nsplit rel error    xerror       xstd
## 1  0.48300624      0 1.0000000 1.0007911 0.05768347
## 2  0.10844747      1 0.5169938 0.5192042 0.02900726
## 3  0.06678458      2 0.4085463 0.4140423 0.02835387
## 4  0.02870391      3 0.3417617 0.3556013 0.02106960
## 5  0.02050153      4 0.3130578 0.3251197 0.02071312
## 6  0.01995037      5 0.2925563 0.3151983 0.02095032
## 7  0.01976132      6 0.2726059 0.3106164 0.02101621
## 8  0.01550003      7 0.2528446 0.2913458 0.01983930
## 9  0.01397824      8 0.2373446 0.2750055 0.01725564
## 10 0.01322455      9 0.2233663 0.2677136 0.01714828
## 11 0.01089820     10 0.2101418 0.2506827 0.01561141
## 12 0.01000000     11 0.1992436 0.2480154 0.01583340
```

Although useful, this approach requires you to manually assess multiple models. Rather, we can perform a grid search to automatically search across a range of differently tuned models to identify the optimal hyerparameter setting.

To perform a grid search we first create our hyperparameter grid. In this example, I search a range of minsplit from 5-20 and vary maxdepth from 8-15 (since our original model found an optimal depth of 12). What results is 128 different combinations, which requires 128 different models.

```r
hyper_grid <- expand.grid(
  minsplit = seq(5, 20, 1),
  maxdepth = seq(8, 15, 1)
)

head(hyper_grid)
```

```
##   minsplit maxdepth
## 1        5        8
## 2        6        8
## 3        7        8
## 4        8        8
## 5        9        8
## 6       10        8
```

```r
nrow(hyper_grid)
```

```
## [1] 128
```

To automate the modeling we simply set up a for loop and iterate through each minsplit and maxdepth combination. We save each model into its own list item.

```r
models <- list()

for (i in 1:nrow(hyper_grid)) {

  # get minsplit, maxdepth values at row i
  minsplit <- hyper_grid$minsplit[i]
  maxdepth <- hyper_grid$maxdepth[i]

  # train a model and store in the list
  models[[i]] <- rpart(
    formula = Sale_Price ~ .,
    data    = ames_train,
    method  = "anova",
    control = list(minsplit = minsplit, maxdepth = maxdepth)
    )
}
```

We can now create a function to extract the minimum error associated with the optimal cost complexity α value for each model. After a little data wrangling to extract the optimal α value and its respective error, adding it back to our grid, and filter for the top 5 minimal error values we see that the optimal model makes a slight improvement over our earlier model (xerror of 0.242 versus 0.272).

```
# function to get optimal cp
get_cp <- function(x) {
  min    <- which.min(x$cptable[, "xerror"])
  cp <- x$cptable[min, "CP"]
}

# function to get minimum error
get_min_error <- function(x) {
  min    <- which.min(x$cptable[, "xerror"])
  xerror <- x$cptable[min, "xerror"]
}

hyper_grid %>%
```

If we were satisfied with these results we could apply this final optimal model and predict on our test set. The final RMSE is 39145.39 which suggests that, on average, our predicted sales prices are about 39,145 Dollar off from the actual sales price.

```
optimal_tree <- rpart(
    formula = Sale_Price ~ .,
    data    = ames_train,
    method  = "anova",
    control = list(minsplit = 11, maxdepth = 8, cp = 0.01)
    )

pred <- predict(optimal_tree, newdata = ames_test)
RMSE(pred = pred, obs = ames_test$Sale_Price)
```

```
## [1] 39145.39
```

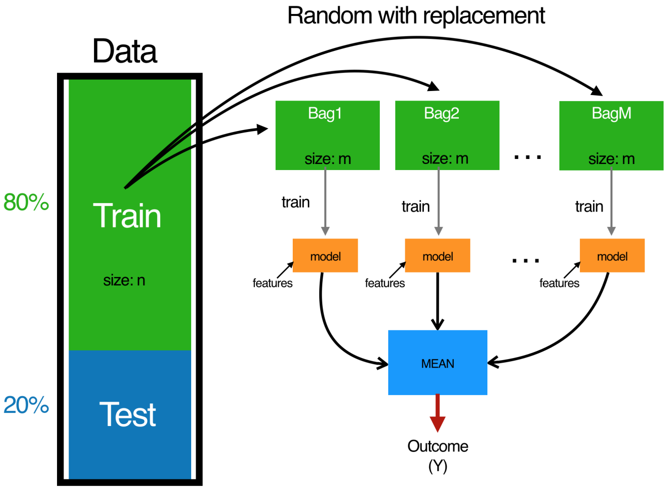
Bagging - The idea

As previously mentioned, single tree models suffer from high variance. Although pruning the tree helps reduce this variance, there are alternative methods that actually exploite the variability of single trees in a way that can significantly improve performance over and above that of single trees. Bootstrap aggregating (bagging) is one such approach (originally proposed by Breiman, 1996).

Bagging combines and averages multiple models. Averaging across multiple trees reduces the variability of any one tree and reduces overfitting, which improves predictive performance. Bagging follows three simple steps:

- Create m bootstrap samples from the training data. Bootstrapped samples allow us to create many slightly different data sets but with the same distribution as the overall training set.
- For each bootstrap sample train a single, unpruned regression tree.
- Average individual predictions from each tree to create an overall average predicted value.

The bagging process.



This process can actually be applied to any regression or classification model; however, it provides the greatest improvement for models that have high variance. For example, more stable parametric models such as linear regression and multi-adaptive regression splines tend to experience less improvement in predictive performance.

One benefit of bagging is that, on average, a bootstrap sample will contain 63 per cent (23) of the training data. This leaves about 33 per cent ($\frac{1}{3}$) of the data out of the bootstrapped sample. We call this the out-of-bag (OOB) sample. We can use the OOB observations to estimate the model's accuracy, creating a natural cross-validation process.

Bagging with ipred

Fitting a bagged tree model is quite simple. Instead of using rpart we use ipred::bagging. We use coob = TRUE to use the OOB sample to estimate the test error. We see that our initial estimate error is close to $3K less than the test error we achieved with our single optimal tree (36543 vs. 39145)

```r
# make bootstrapping reproducible
set.seed(123)

# train bagged model
bagged_m1 <- bagging(
  formula = Sale_Price ~ .,
  data    = ames_train,
  coob    = TRUE
)

bagged_m1
```

One thing to note is that typically, the more trees the better. As we add more trees we are averaging over more high variance single trees. What results is that early on, we see a dramatic reduction in variance (and hence our error) and eventually the reduction in error will flatline signaling an appropriate number of trees to create a stable model. Rarely will you need more than 50 trees to stabilize the error.

By default bagging performs 25 bootstrap samples and trees but we may require more. We can assess the error versus number of trees as below. We see that the error is stabilizing at about 25 trees so we will likely not gain much improvement by simply bagging more trees.

```
# assess 10-50 bagged trees
ntree <- 10:50

# create empty vector to store OOB RMSE values
rmse <- vector(mode = "numeric", length = length(ntree))

for (i in seq_along(ntree)) {
  # reproducibility
  set.seed(123)

  # perform bagged model
  model <- bagging(
  formula = Sale_Price ~ .,
  data    = ames_train,
  coob    = TRUE,
```

Classification Tree example

The purpose of this dataset is to predict which people are more likely to survive after the collision with the iceberg. The dataset contains 13 variables and 1309 observations. The dataset is ordered by the variable X.

```
path <- 'https://raw.githubusercontent.com/thomaspernet/data_csv
titanic <-read.csv(path)
shuffle_index <- sample(1:nrow(titanic))
kable(head(titanic))
```

X	pclass	survived	name	sex
1	1	1	Allen, Miss. Elisabeth Walton	fema
2	1	1	Allison, Master. Hudson Trevor	male
3	1	0	Allison, Miss. Helen Loraine	fema
4	1	0	Allison, Mr. Hudson Joshua Creighton	male
5	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	fema
6	1	1	Anderson, Mr. Harry	male

Unfortunately, decision trees suffer from a major flaw as well. If you allow them to grow limitlessly, they can completely "memorize" the training data, just from creating more and more and more branches.

As a result, individual unconstrained decision trees are very prone to being overfit.

Conditional inference tree

```
library(party)
```

```
?ctree
```

- performs recursively univariate split recursively
- **Vignette** package `party`

CTREE EXAMPLE

```r
install.packages("party")
```

The data behind

```
airq <- subset(airquality, !is.na(Ozone))
summary(airq$Temp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   57.00   71.00   79.00   77.87   85.00   97.00
```

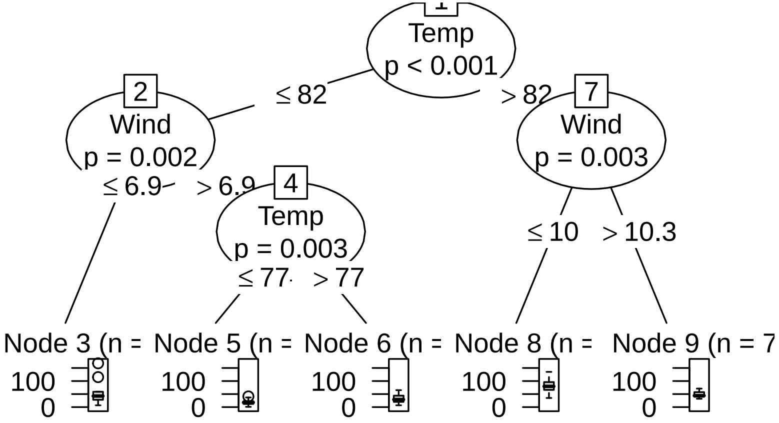
A first model

```r
library(party)

air.ct <- ctree(Ozone ~ ., data = airq, controls = ctree_control
```

The plot for ctree

```
plot(air.ct)
```



Recursive partitioning algorithms are special cases of a

simple two-stage algorithm

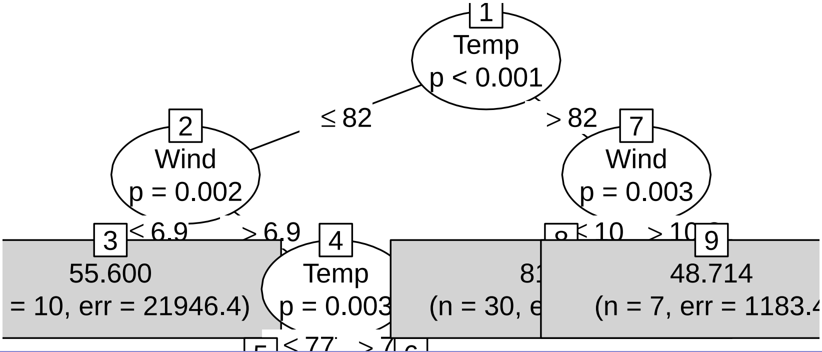
- First partition the observations by univariate splits in a recursive way and
- second fit a constant model in each cell of the resulting partition.

ctree - Regression

```r
library(partykit)
```

```r
?ctree
```

```r
airq <- subset(airquality, !is.na(Ozone))
airct <- ctree(Ozone ~ ., data = airq)
plot(airct, type = "simple")
```



Decision Trees

Regression tree vs. classification tree

```
library(rpart)
```

Grow a tree

```
fit <- rpart(Kyphosis ~ Age + Number + Start,
   method="class", data=kyphosis)

printcp(fit) # display the results
```

```
##
## Classification tree:
## rpart(formula = Kyphosis ~ Age + Number + Start, data = kypho
##     method = "class")
##
## Variables actually used in tree construction:
## [1] Age   Start
##
```

Ensembling

Ensembles are machine learning methods for combining predictions from multiple separate models.

Bagging

attempts to reduce the chance overfitting complex models.

- It trains a large number of "strong" learners in parallel.
- A strong learner is a model that's relatively unconstrained.
- Bagging then combines all the strong learners together in order to "smooth out" their predictions.

Boosting

attempts to improve the predictive flexibility of simple models.

- It trains a large number of "weak" learners in sequence.
- A weak learner is a constrained model (i.e. you could limit the max depth of each decision tree).
- Each one in the sequence focuses on learning from the mistakes of the one before it.
- Boosting then combines all the weak learners into a single strong

Random Forest

Random forest aims to reduce the previously mentioned correlation issue by choosing only a subsample of the feature space at each split. Essentially, it aims to make the trees de-correlated and prune the trees by setting a stopping criteria for node splits, which I will cover in more detail later.

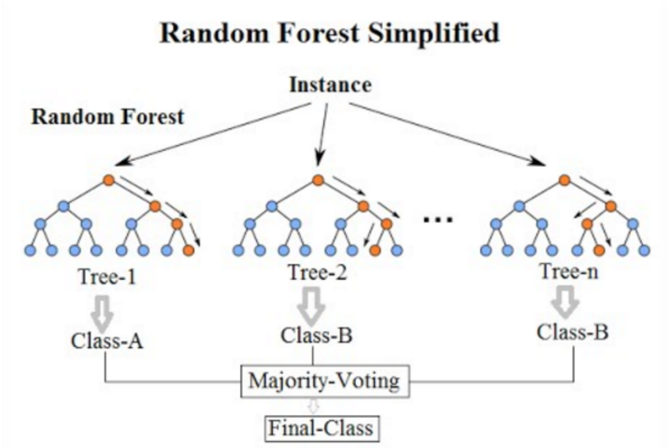
What are the advantages and disadvantages of decision trees?

Advantages: Decision trees are easy to interpret, nonparametric (which means they are robust to outliers), and there are relatively few parameters to tune.

Disadvantages: Decision trees are prone to be overfit. However, this can be addressed by ensemble methods like random forests or boosted trees.

Random forest

- Ensemble learning method - multitude of decision trees
- Random forests correct for decision trees' habit of overfitting to their training set.



Bagging

- Bagging is also known as bootstrap aggregation.
- Bagging is a method for combining predictions from different regression or classification models and was developed by Leo Breiman.
- The results of the models are then averaged in the simplest case.
- The result of each model prediction is included in the prediction with the same weight.
- The weights could depend on the quality of the model prediction, i.e. "good" models are more important than "bad" models.
- Bagging leads to significantly improved predictions in the case of unstable models.

Gradient boosting

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

The idea of gradient boosting originated in the observation by Leo Breiman that boosting can be interpreted as an optimization algorithm on a suitable cost function.

Breiman, L. (1997). "Arcing The Edge". Technical Report 486. Statistics Department, University of California, Berkeley.

Explicit algorithms

Explicit regression gradient boosting algorithms were subsequently developed by Jerome H. Friedman,[2][3] simultaneously with the more general functional gradient boosting perspective of Llew Mason, Jonathan Baxter, Peter Bartlett and Marcus Frean.[4][5]

The latter two papers introduced the view of boosting algorithms as iterative functional gradient descent algorithms. That is, algorithms that optimize a cost function over function space by iteratively choosing a function (weak hypothesis) that points in the negative gradient direction. This functional gradient view of boosting has led to the development of boosting algorithms in many areas of machine learning and statistics beyond regression and classification.

Advantages of gradient boosting

- Often provides predictive accuracy that cannot be beat.
- Lots of flexibility - can optimize on different loss functions and provides several hyperparameter tuning options that make the function fit very flexible.
- No data pre-processing required - often works great with categorical and numerical values as is.
- Handles missing data - imputation not required.

Disadvantages of gradient boosting

- GBMs will continue improving to minimize all errors. This can overemphasize outliers and cause overfitting. Must use cross-validation to neutralize.
- Computationally expensive - GBMs often require many trees ($>$1000) which can be time and memory exhaustive.
- The high flexibility results in many parameters that interact and influence heavily the behavior of the approach (number of iterations, tree depth, regularization parameters, etc.). This requires a large grid search during tuning.
- Less interpretable although this is easily addressed with various tools (variable importance, partial dependence plots, LIME, etc.).

Two types of errors for tree methods

Bias related errors

- Adaptive boosting
- Gradient boosting

Variance related errors

- Bagging
- Random forest

Gradient Boosting for Linear Regression - why does it not work?

While learning about Gradient Boosting, I haven't heard about any constraints regarding the properties of a "weak classifier" that the method uses to build and ensemble model. However, I could not imagine an application of a GB that uses linear regression, and in fact when I've performed some tests - it doesn't work. I was testing the most standard approach with a gradient of sum of squared residuals and adding the subsequent models together.

The obvious problem is that the residuals from the first model are populated in such manner that there is really no regression line to fit anymore. My another observation is that a sum of subsequent linear regression models can be represented as a single regression model as well (adding all intercepts and corresponding coefficients) so I cannot imagine how that could ever improve the model. The last observation is that a linear regression (the most typical approach) is using sum of squared residuals as a loss function - the same one that GB is using.

Links

- **Vignette** for package `partykit`
- Conditional Inference Trees
- Conditional inference trees vs traditional decision trees
- Video on tree based methods

Links - Boosting

- **Gradient Boosting Machines**
- How to Visualize Gradient Boosting Decision Trees With XGBoost in Python