Supervised Learning - Regression Trees and Bagging

Jan-Philipp Kolb

22 Mai, 2019

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**)

But:

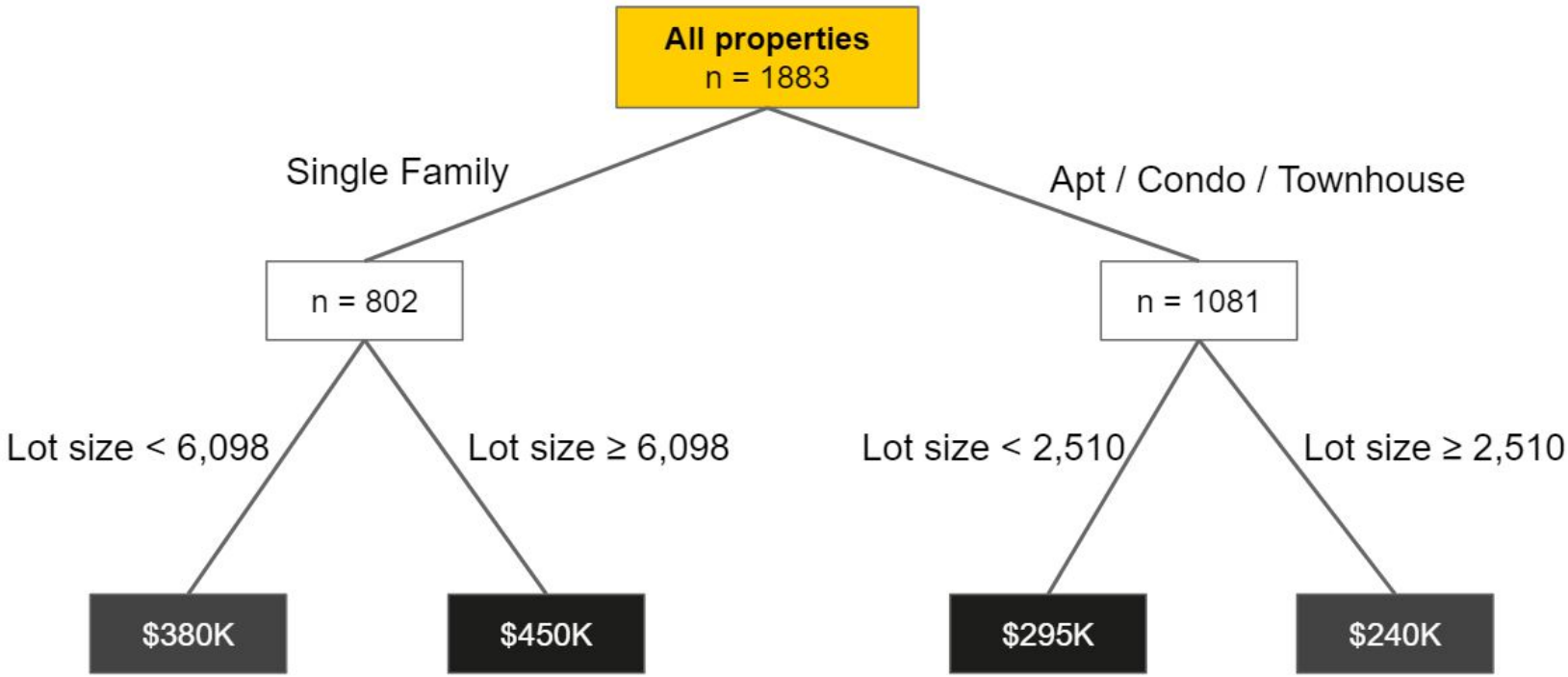
- These methods do not deliver the best results concerning prediction accuracy.

The Idea

- There are many methodologies for constructing regression trees but one of the oldest is the classification and regression tree (CART) approach by Breiman et al. (1984).
- 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.

Explanation: decision tree

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



Example decission trees

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

- Let's say for Single Family homes, larger lots command higher prices,
- and 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

- Decision trees can capture this relationship naturally.

Regression Trees - preparation

- The following slides are based on the UC Business Analytics R Programming Guide on **regression trees**

```
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
```

Model foundations

- This simple example can be generalized
- 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:

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

How to grow a regression tree - deciding on splits

- It is important to realize the partitioning of variables are done in a top-down, greedy fashion.
- A partition performed earlier in the tree will not change based on later partitions.

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:

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

The best split

- 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.
- We typically get 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.

The Ames Housing data

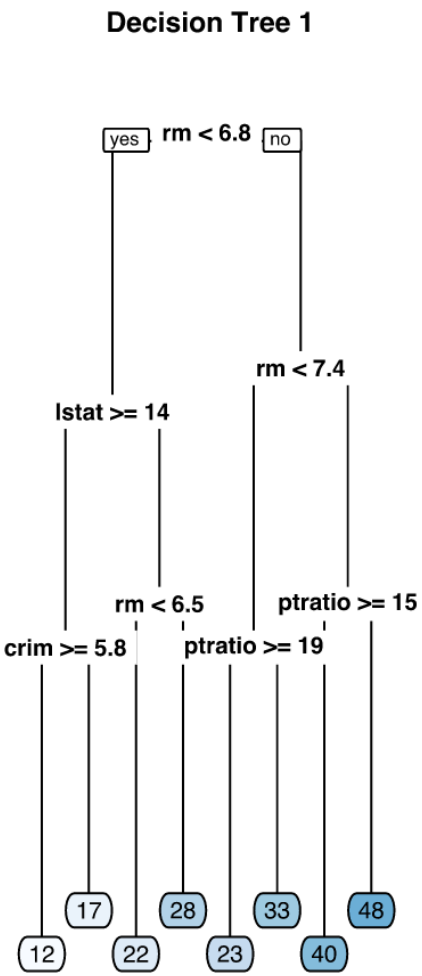
- Again we use the Ames dataset and split it in a test and training dataset

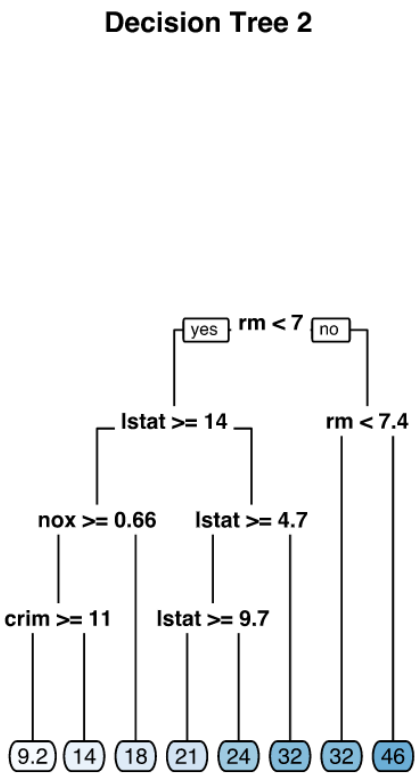
```
set.seed(123)
ames_data <- AmesHousing::make_ames()
ames_split <- initial_split(ames_data,prop = .7)
ames_train <- training(ames_split)
ames_test  <- testing(ames_split)
```

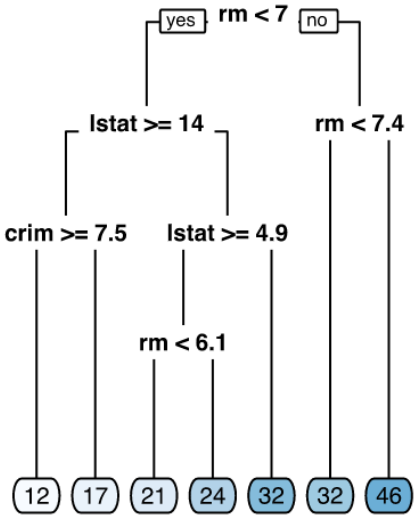
Example Boston housing data set

- We 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; - they tend to differ substantially closer to the terminal nodes.
- These deeper nodes tend to overfit to specific attributes of the sample data;
- 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 three 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 grow a very large tree as showed and then prune it back to find an optimal subtree.
- We find this subtree by using a cost complexity parameter (α) that penalizes our objective function for the number of terminal nodes of the tree (T).

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

The complexity value α

- For a given value of α, we find the smallest pruned tree that has the lowest penalized error.
- $\rightarrow$ A close association to the lasso L_1 norm penalty.
- Smaller penalties tend to produce more complex models, which result in larger trees.
- Larger penalties result in much smaller trees.
- As a tree grows larger, the reduction in the SSE must be greater than the cost complexity penalty.
- Typically, we evaluate multiple models across a spectrum of α and use cross-validation to identify the optimal α and the optimal subtree.

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.
- The model provides a non-linear response, so it can work when the true regression surface is not smooth.
- If it is smooth, the piecewise-constant surface can approximate it arbitrarily closely (with enough leaves).
- There are fast, reliable algorithms to learn these trees.

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.

Fit a regression tree using `rpart`

- 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`).
- When fitting a regression tree, we need to set `method = "anova"`.
- By default, `rpart` will make an intelligent guess based on the data type of the response column
- But it's recommened to explictly set the method for reproducibility reasons (auto-guesser may change in future).

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

THE m1 OUTPUT.

```
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,Above_Average,Good 1699 4.001092e+12 156147.10

     4) 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 1000 1.298629e+12 131787.90
       8) Overall_Qual=Very_Poor,Poor,Fair,Below_Average 195 1.733699e+11  98238.33 *
       9) Overall_Qual=Average,Above_Average,Good 805 8.526051e+11 139914.80
        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_Heights,Gilbert,Northwest_Ames,Sawyer_West,Crawford,T
imberland,Northridge,Stone_Brook,Clear_Creek,Bloomington_Heights,Veenker,Green_Hills 699 1.260199e+12 190995
.90
      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 2.874510e+12 304571.10
     6) Overall_Qual=Very_Good 254 8.855113e+11 273369.50
      12) Gr_Liv_Area< 1959.5 155 3.256677e+11 247662.30 *
      13) Gr_Liv_Area>=1959.5 99 2.970338e+11 313618.30 *
     7) Overall_Qual=Excellent,Very_Excellent 98 1.100817e+12 385440.30
      14) Gr_Liv_Area< 1990 42 7.880164e+10 325358.30 *
      15) Gr_Liv_Area>=1990 56 7.566917e+11 430501.80
        30) Neighborhood=College_Creek,Edwards,Timberland,Veenker 8 1.153051e+11 281887.50 *
        31) Neighborhood=Old_Town,Somerset,Northridge_Heights,Northridge,Stone_Brook 48 4.352486e+11 455270.
80
          62) Total_Bsmt_SF< 1433 12 3.143066e+10 360094.20 *
          63) Total_Bsmt_SF>=1433 36 2.588806e+11 486996.40 *
```

m1

Steps of the splits explained

- E.g., we start with 2051 observations at the root node (very beginning) and the first variable we split on (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_Average,Good`

go to the 2nd branch.

The 3rd branch

- The number of observations in this branch (1699), their average sales price (156147.10) and SSE (4.001092e+12) are listed.
- In the 3rd branch we have 352 observations with

```
Overall_Qual=Very_Good,Excellent,Very_Excellent
```

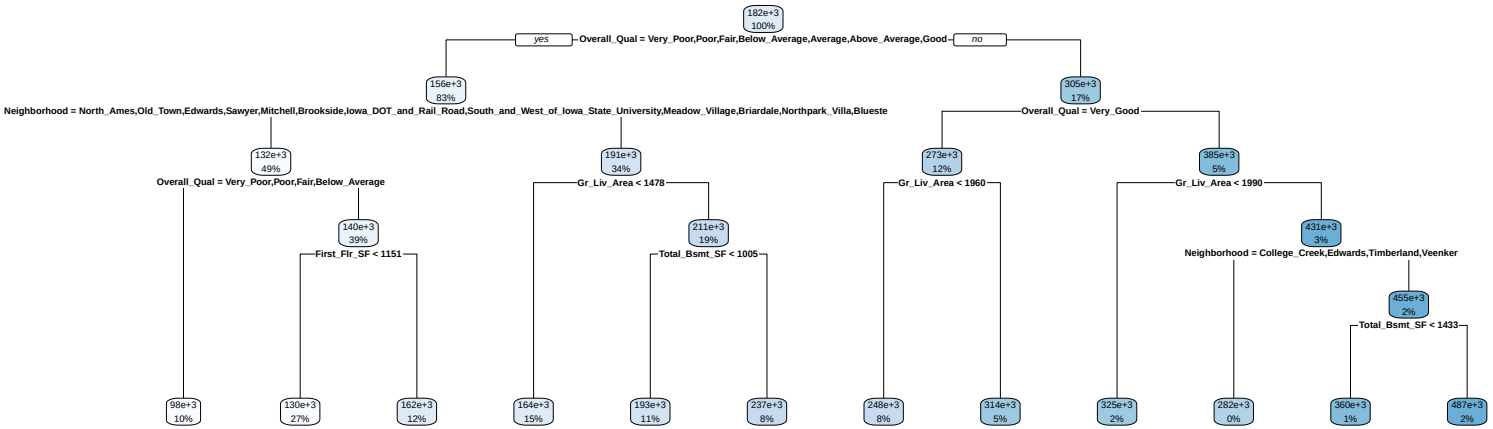
- their average sales prices is 304571.10 and the SEE in this region is 2.874510e+12.

Visualization with `rpart.plot`

- `rpart.plot` has many plotting options
- In the default print it will show the percentage of data that fall to that node and the average sales price for that branch.
- This tree contains 11 internal nodes resulting in 12 terminal nodes.
- This tree is partitioning on 11 variables to produce its model.

The rpart.plot

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



Behind the scenes

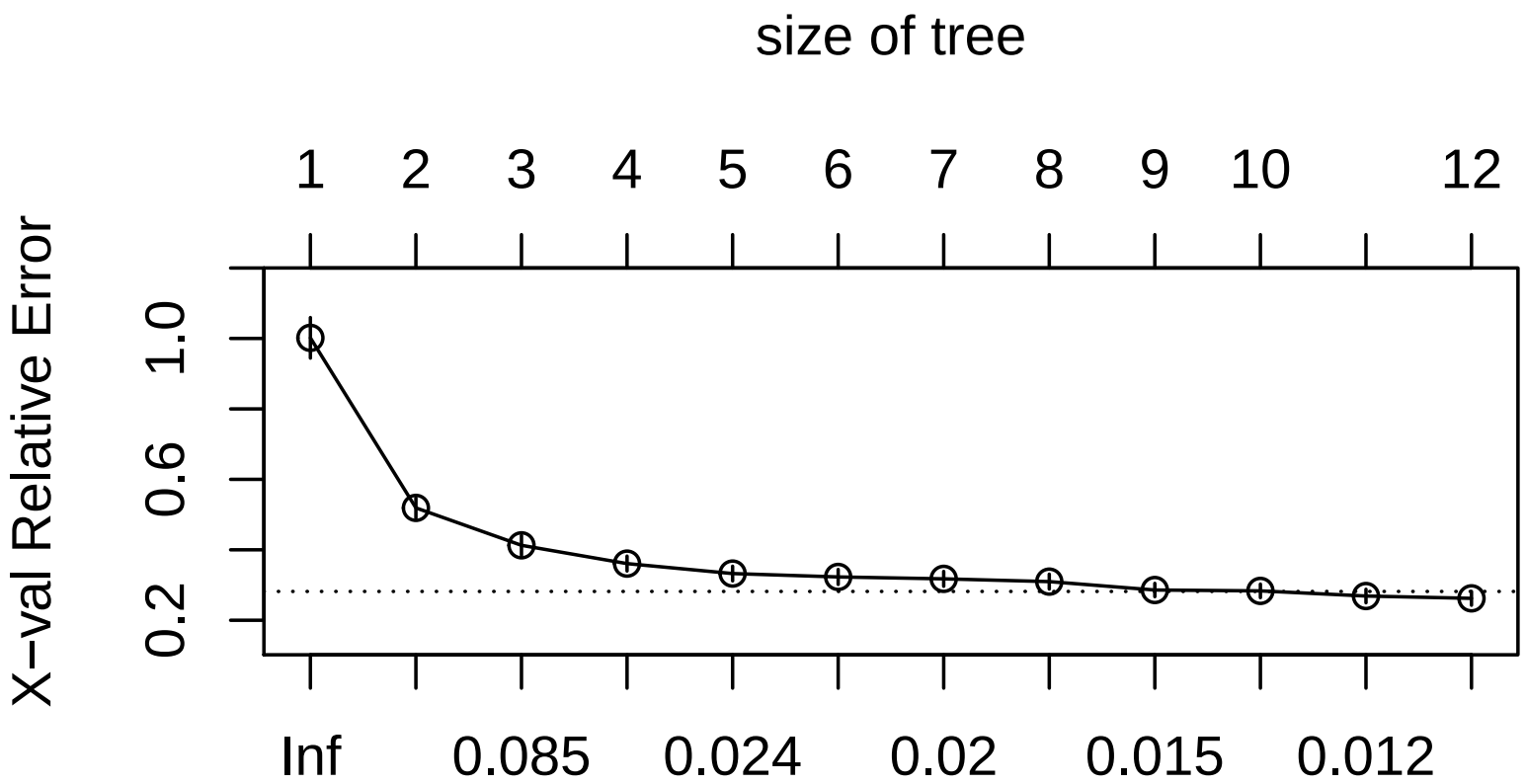
There are 80 variables in `ames_train`. So what happened?

- `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 α value is computed on the hold-out validation data.

The plotcp

- In this example we find diminishing returns after 12 terminal nodes (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 $= |T|$).

```
plotcp(m1)
```



The 1-SE rule - how many terminal nodes

- Breiman et al. (1984) suggested to 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 expect to get similar results within a small margin of error.
- 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).

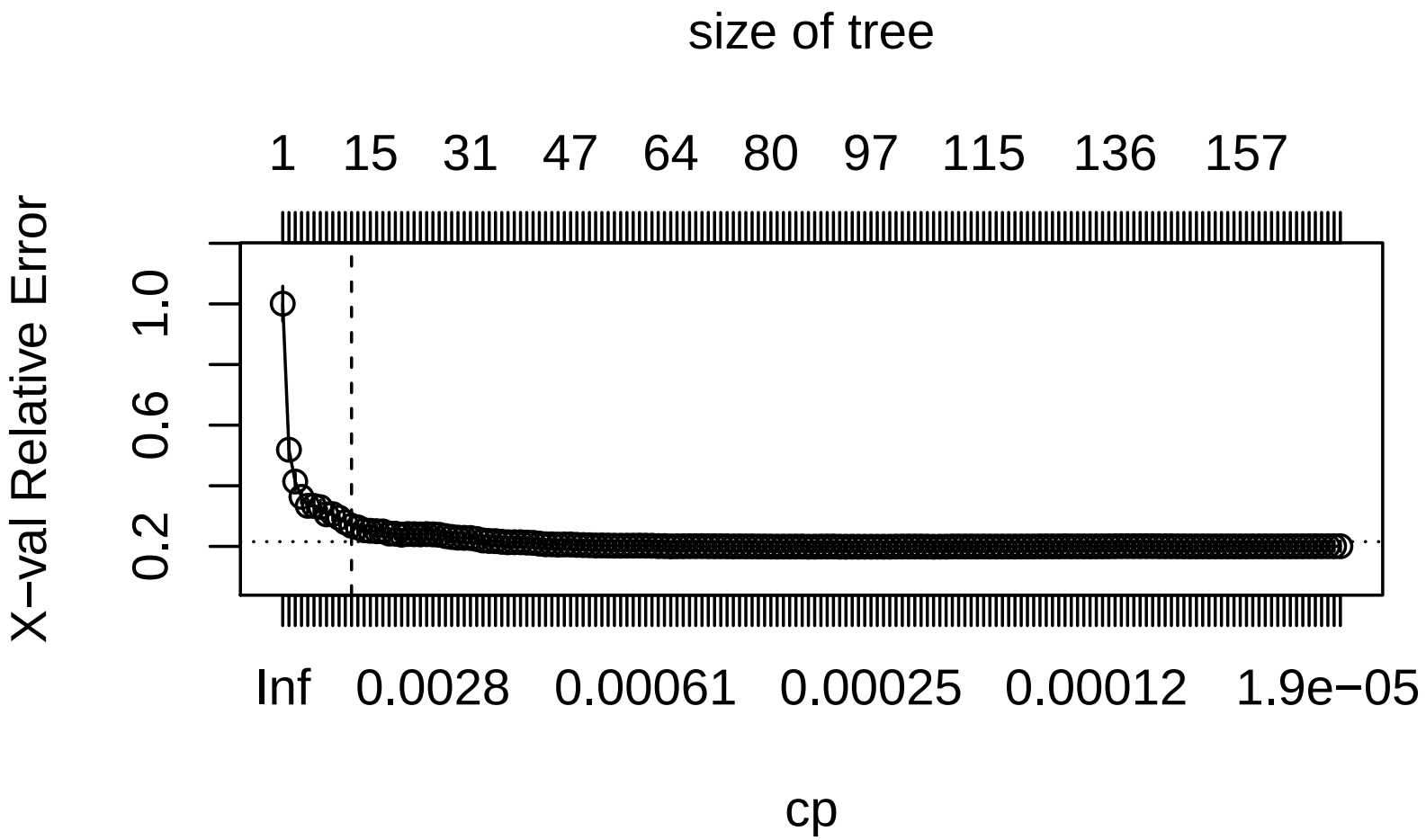
Generate a full tree

- 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))
```

Plot the result

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



Automated tuning by default

- `rpart` is performing some automated tuning by default, 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).
- We can perform additional tuning to try improve model performance.

The output cptable

```
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.

Special control argument

- `rpart` uses a special control argument where we provide a list of hyperparameter values.
- E.g., if we want 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)
)
```

The output cptable of model 3

```
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
```

Grid search

- We can avoid it to manually assess multiple models, by performing 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.

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

- The result are 128 combinations - 128 different models.

```
head(hyper_grid)
```

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

A loop to autimate modeling

- 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.

```
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)
    )
}
```

A function to extract the minimum error

- 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"]
}
```

Apply the functions

```
hyper_grid %>%
  mutate(
    cp    = purrr::map_dbl(models, get_cp),
    error = purrr::map_dbl(models, get_min_error)
    ) %>%
  arrange(error) %>%
  top_n(-5, wt = error)
```

```
##   minsplit maxdepth        cp     error
## 1        5       13 0.0108982 0.2421256
## 2        6        8 0.0100000 0.2453631
## 3       12       10 0.0100000 0.2454067
## 4        8       13 0.0100000 0.2459588
## 5       19        9 0.0100000 0.2460173
```

The final optimal model

- 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
```

What is bagging?

- Basic regression trees divide a data set into smaller groups and then fit a simple model (constant) for each subgroup.
- But a single tree model tends to be highly unstable and a poor predictor.
- Bootstrap aggregating (bagging) regression trees is quite powerful and effective.
- This provides the fundamental basis of more complex tree-based models such as random forests and gradient boosting machines.

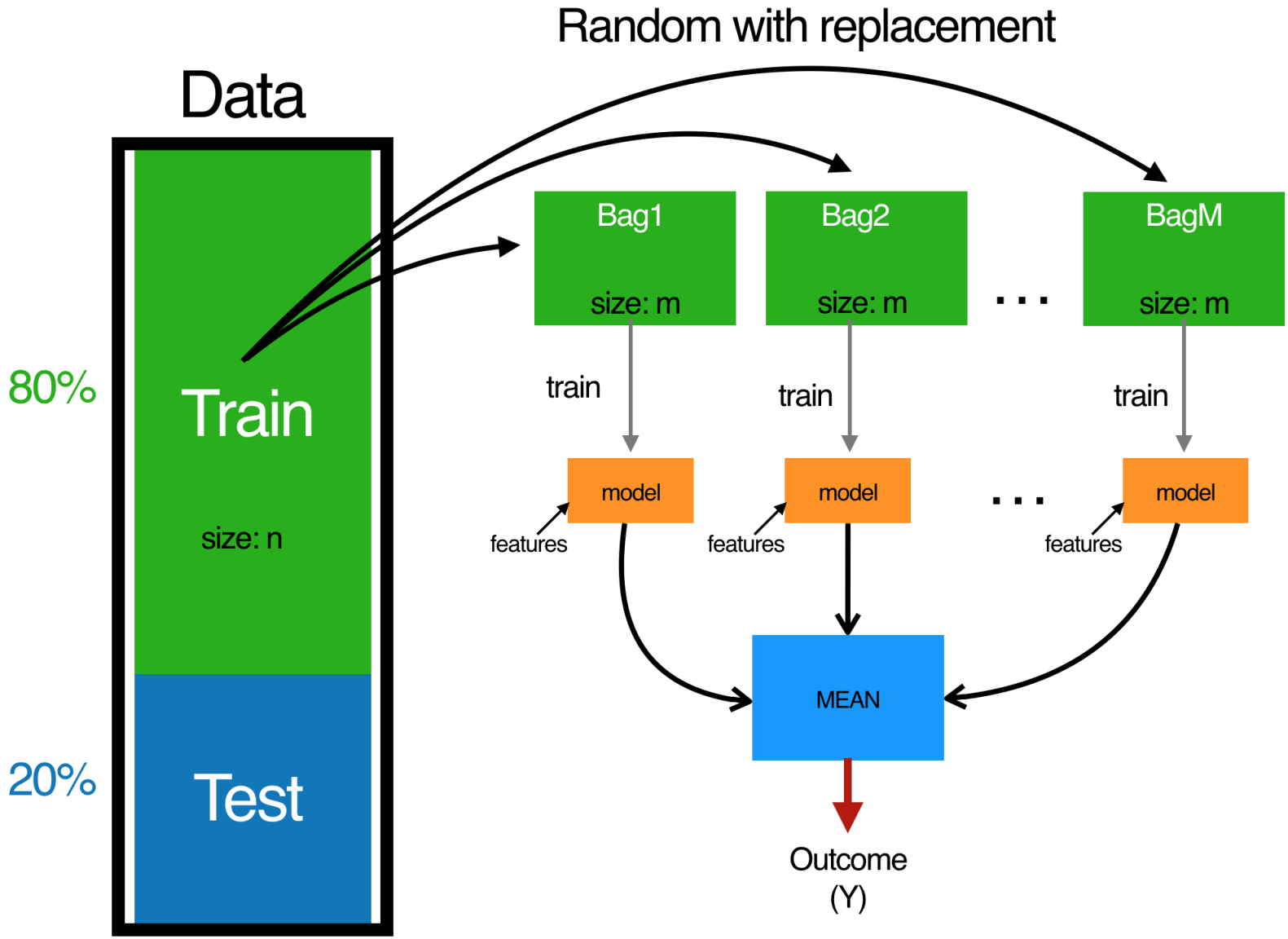
Bagging

- Single tree models suffer from high variance.
- Pruning helps, but there are alternative methods that exploite the variability of single trees in a way that can significantly improve performance.
- Bootstrap aggregating (bagging) is one such approach (originally proposed by Breiman, 1996).
- 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
- Model predictions are included in the prediction with the same weight (simplest case).
- 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.

Bagging follows three simple steps:

- 1.) 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.
- 2.) For each bootstrap sample train a single, unpruned regression tree.
- 3.) Average individual predictions from each tree to create an overall average predicted value.

The bagging process.



About bagging

- This process can be applied to any regression or classification model;
- It provides the greatest improvement for models that have high variance.
- More stable parametric models such as linear regression and multi-adaptive regression splines tend to experience less improvement in predictive performance.
- 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)

Bagging with ipred (II)

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

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


##
## Bagging regression trees with 25 bootstrap replications
##
## Call: bagging.data.frame(formula = Sale_Price ~ ., data = ame
##     coob = TRUE)
##
## Out-of-bag estimate of root mean squared error:  36543.37
```

One thing to note typically

- The more trees the better - we are averaging over more high variance single trees.
- We see a dramatic reduction in variance (and hence our error) and eventually the reduction in error will flatline
- You need less than 50 trees to stabilize the error.

Number of bootstrap samples

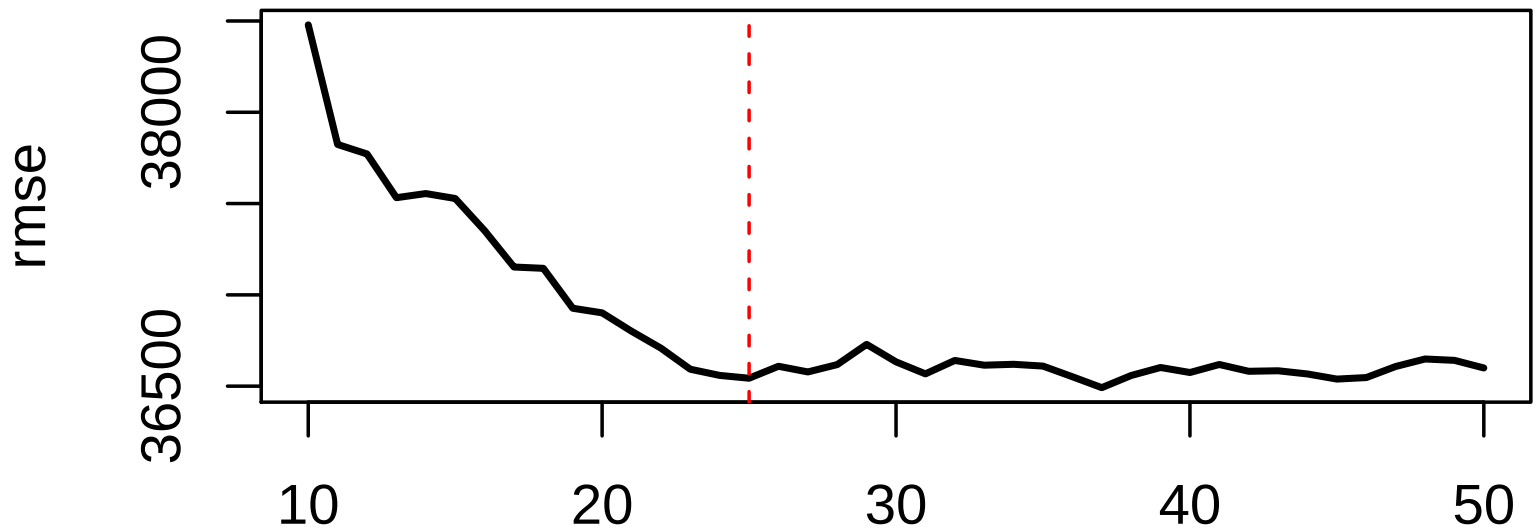
- By default bagging performs 25 bootstrap samples and trees but we may require more.

```
ntree <- 10:50 # assess 10-50 bagged trees
# create empty vector to store OOB RMSE values
rmse <- vector(mode = "numeric", length = length(ntree))
for (i in seq_along(ntree)) {
  set.seed(123)   # reproducibility
  # perform bagged model
  model <- bagging(
  formula = Sale_Price ~ .,
  data    = ames_train,
  coob    = TRUE,
  nbagg   = ntree[i]
)
  rmse[i] <- model$err   # get OOB error
}
```

Plot the result

- We see that the error is stabilizing at about 25 trees so we will likely not gain much improvement by simply bagging more trees.

```
plot(ntree, rmse, type = 'l', lwd = 2)
abline(v = 25, col = "red", lty = "dashed")
```



Bagging with caret

- Bagging with `ipred` is simple but there are some additional benefits of bagging with `caret`.

1.) Its easier to perform cross-validation. Although we can use the OOB error, performing cross validation will also provide a more robust understanding of the true expected test error.

2.) We can assess variable importance across the bagged trees.

A 10-fold cross-validated model.

- We see that the cross-validated RMSE is 36,477 dollar.
- We also assess the top 20 variables from our model.
- Variable importance for regression trees is measured by assessing the total amount SSE is decreased by splits over a given predictor, averaged over all m trees.
- The predictors with the largest average impact to SSE are considered most important. - The importance value is simply the relative mean decrease in SSE compared to the most important variable (provides a 0-100 scale).

CV BAGGED MODEL

```
# Specify 10-fold cross validation
ctrl <- trainControl(method = "cv",  number = 10)

bagged_cv <- train(Sale_Price ~ .,data = ames_train,
  method = "treebag",trControl = ctrl,importance = TRUE
  )
```

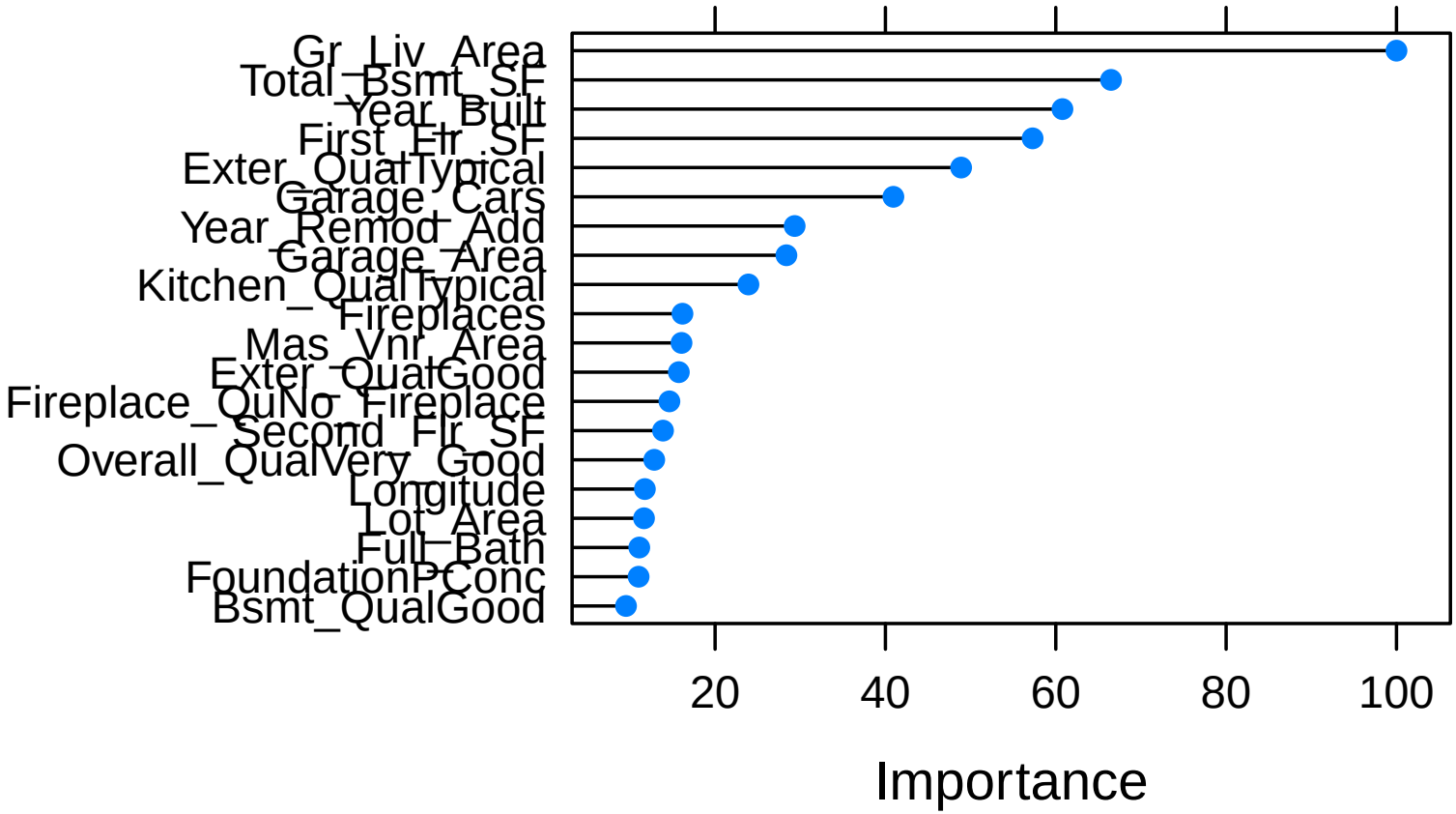
Assess results

```
bagged_cv
```

```
## Bagged CART
##
## 2051 samples
##   80 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 1846, 1845, 1847, 1845, 1846, 1847,
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   36477.25  0.8001783  24059.85
```

Assess results with a plot

```
plot(varImp(bagged_cv), 20)
```



Extensions

- If we compare this to the test set out of sample we see that our cross-validated error estimate was very close.
- We have successfully reduced our error to about $35,000;
- Extensions of this bagging concept (random forests and GBMs) can significantly reduce this further.

```
pred <- predict(bagged_cv, ames_test)
RMSE(pred, ames_test$Sale_Price)
```

```
## [1] 35262.59
```

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**