CMSC/LING/STAT 208: Machine Learning

Abhishek Chakraborty [Much of the content in these slides have been adapted from *ISLR2* by James et al. and *HOMLR* by Boehmke & Greenwell]

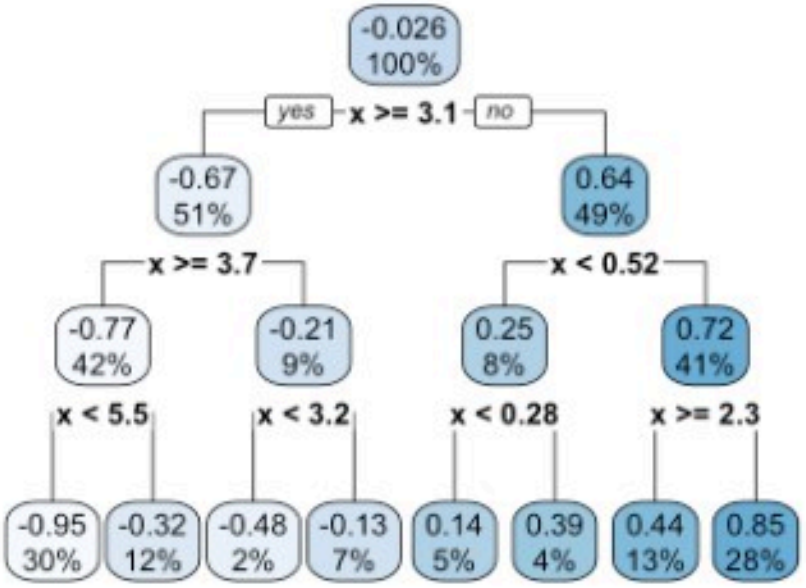
Tree-Based Methods

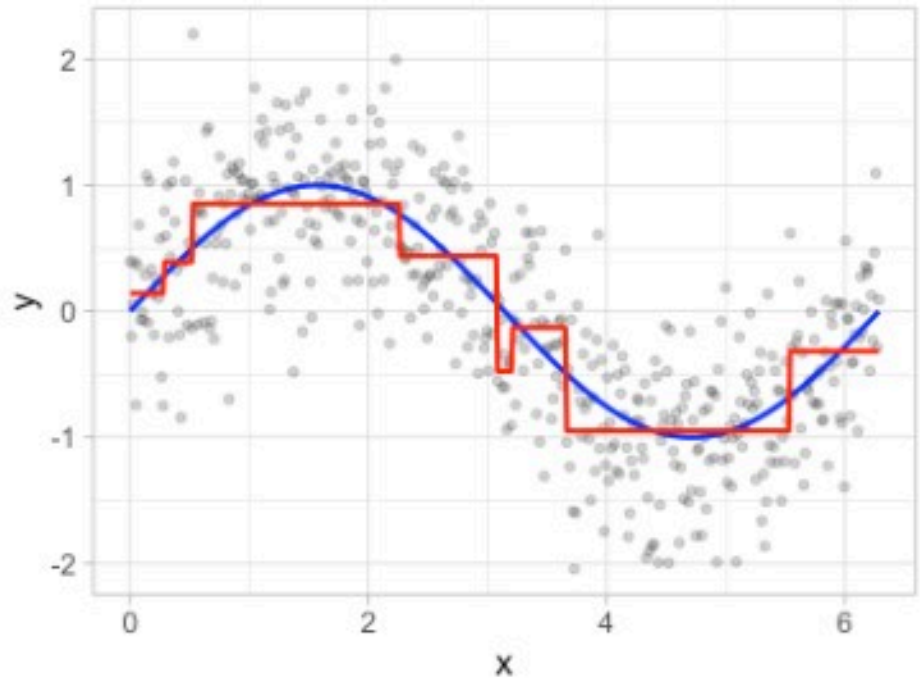
- Involves **stratifying** or **segmenting** the predictor space into a number of simple regions.
- The set of splitting rules used to segment the predictor space can be summarized in a tree, thus, the name **decision tree** methods.
- Can be used for both classification and regression.
- Tree-based methods are simple and useful for interpretation, however, not the best in terms of prediction accuracy.
- Methods such as **bagging**, **random forests**, and **boosting** grow multiple trees and then combine their results.

Terminology for Trees

- Every split is considered to be a **node**.
- We refer to the first node at the top of the tree as the **root node** (this node contains all of the training data).
- The final nodes at the bottom of the tree are called the **terminal nodes** or **leaves**.
- Decision trees are typically drawn **upside down**, in the sense that the leaves are at the bottom of the tree.
- The points along the tree where the predictor space is split are referred to as **internal nodes**, that is, every node in between the **root node** and **terminal nodes** is referred to as an **internal node**.
- The segments of the trees that connect the nodes are known as **branches**.

Terminology for Trees





Adapted from HMLR, Boehmke & Greenwell

Building a Tree

- First select the predictor X_j and the cutpoint s such that splitting the predictor space into the regions $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ leads to the greatest possible reduction in SSE.

For any j and s, define

$$R_1 = \{X|X_j < s\} \text{ and } R_2 = \{X|X_j > s\}$$

Find j and s that minimize

$$SSE = \sum_{i \in R_1} \left(y_i - \hat{y}_{R_1}\right)^2 + \sum_{i \in R_2} \left(y_i - \hat{y}_{R_2}\right)^2$$

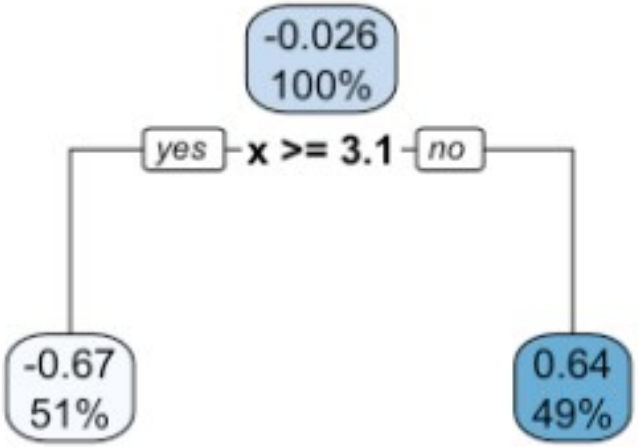
- Next, repeat the process, look for the best predictor and best cutpoint in order to split the data further. However, this time, instead of splitting the entire predictor space, split one of the two previously identified regions.
- The process continues until a stopping criterion is reached; say, we may continue until no region contains more than five observations.

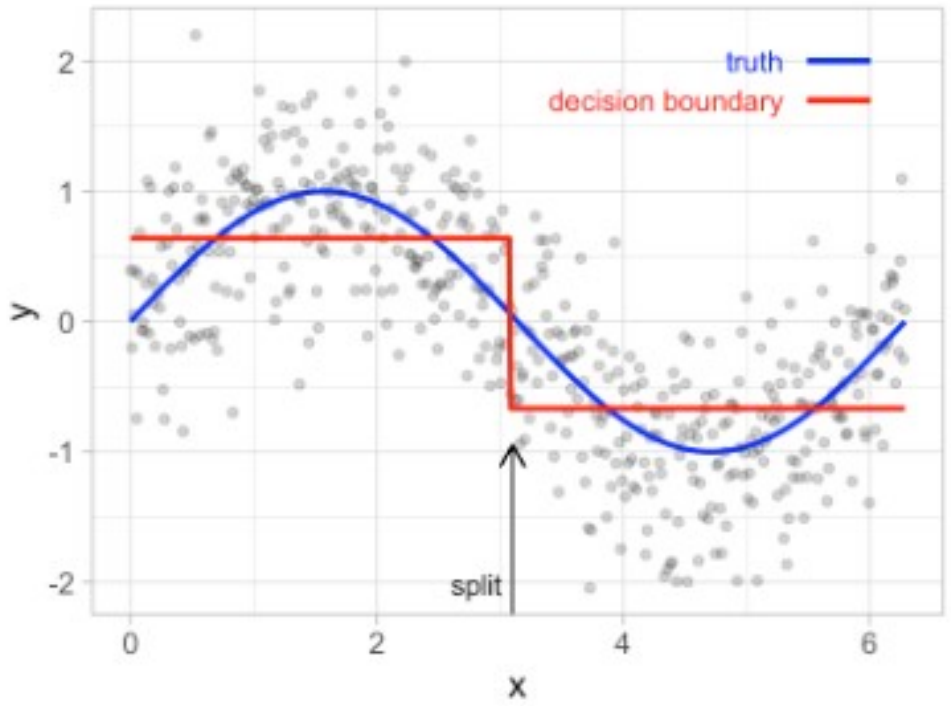
Prediction

For every observation that falls into the region R_j, make the same prediction, which is

- the mean response of the training set observations in R_j (for regression problems),
- majority vote response of the training set observations in R_j (for classification problems).

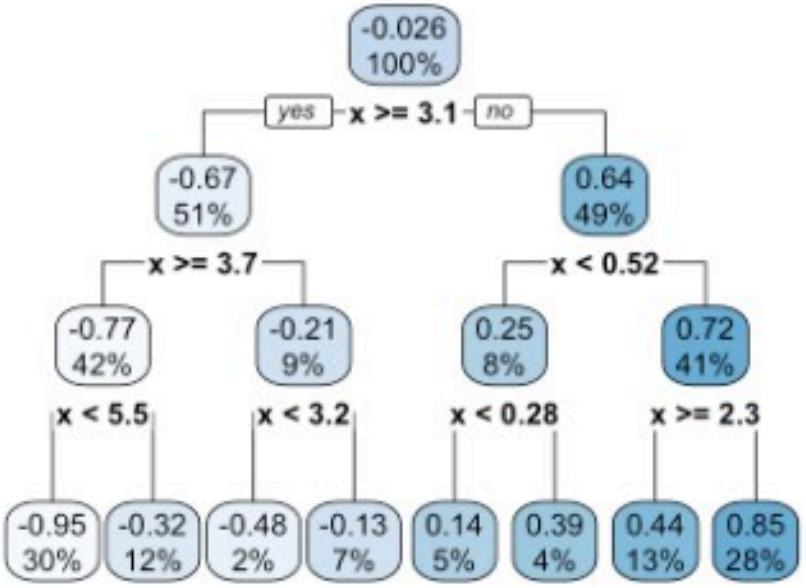
Building a Tree and Prediction

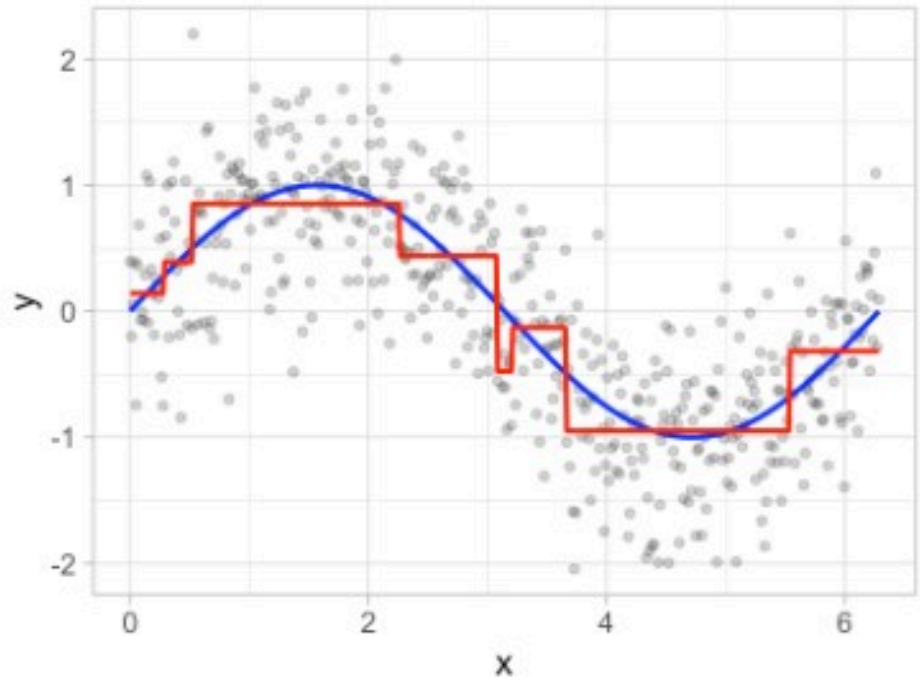




Adapted from HMLR, Boehmke & Greenwell

Building a Tree and Prediction

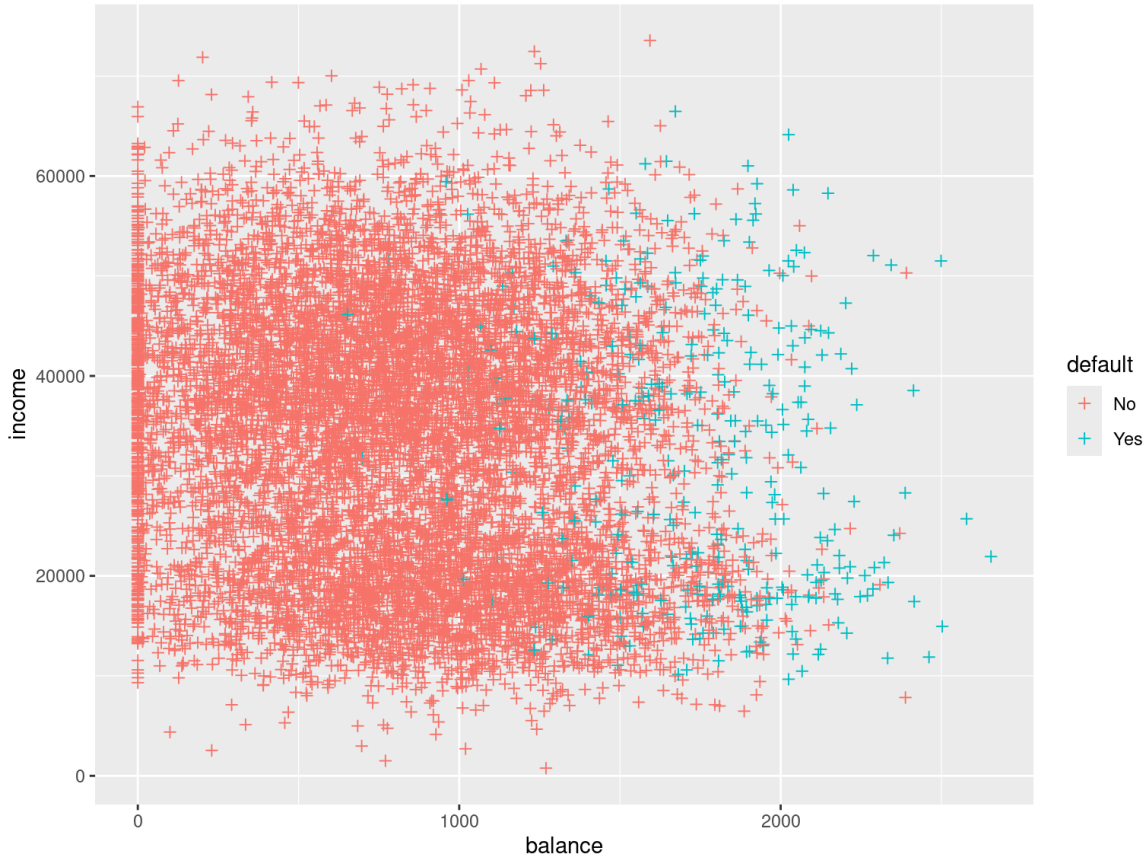




Adapted from HMLR, Boehmke & Greenwell

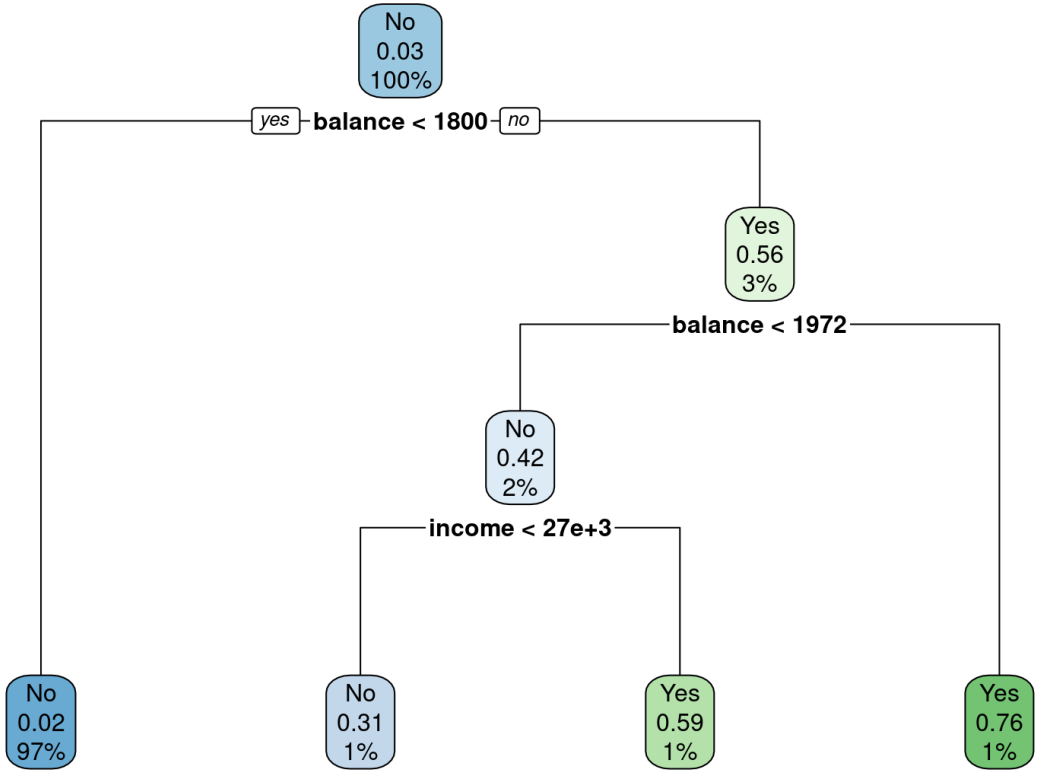
Building a Tree and Prediction

Default Dataset

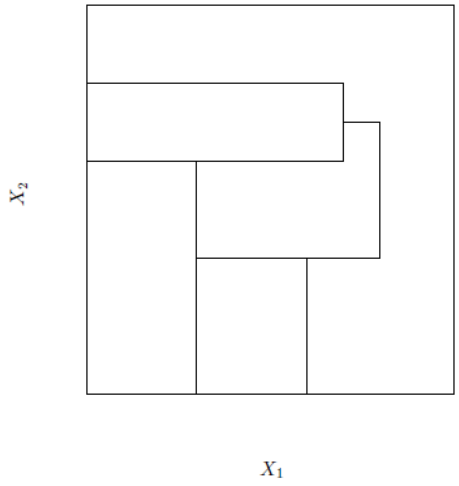


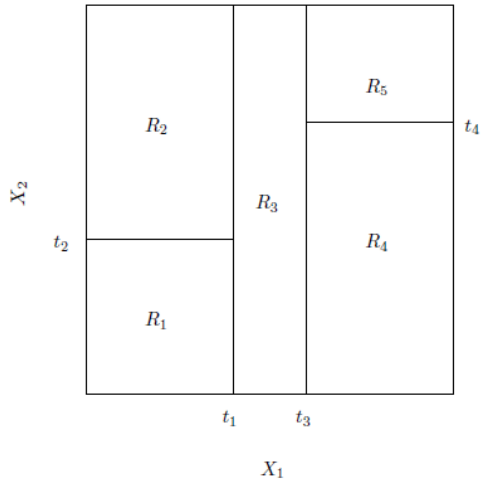
Building a Tree and Prediction

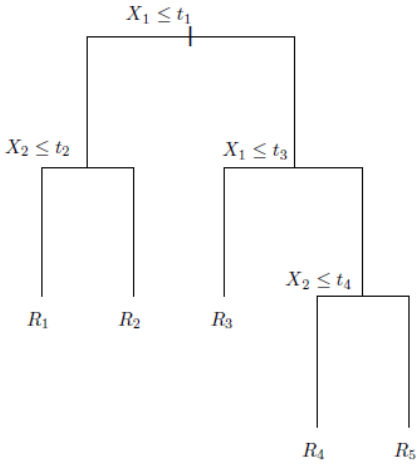
Default Dataset

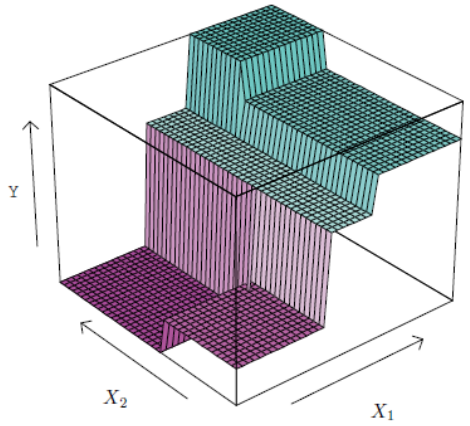


Building a Tree and Prediction









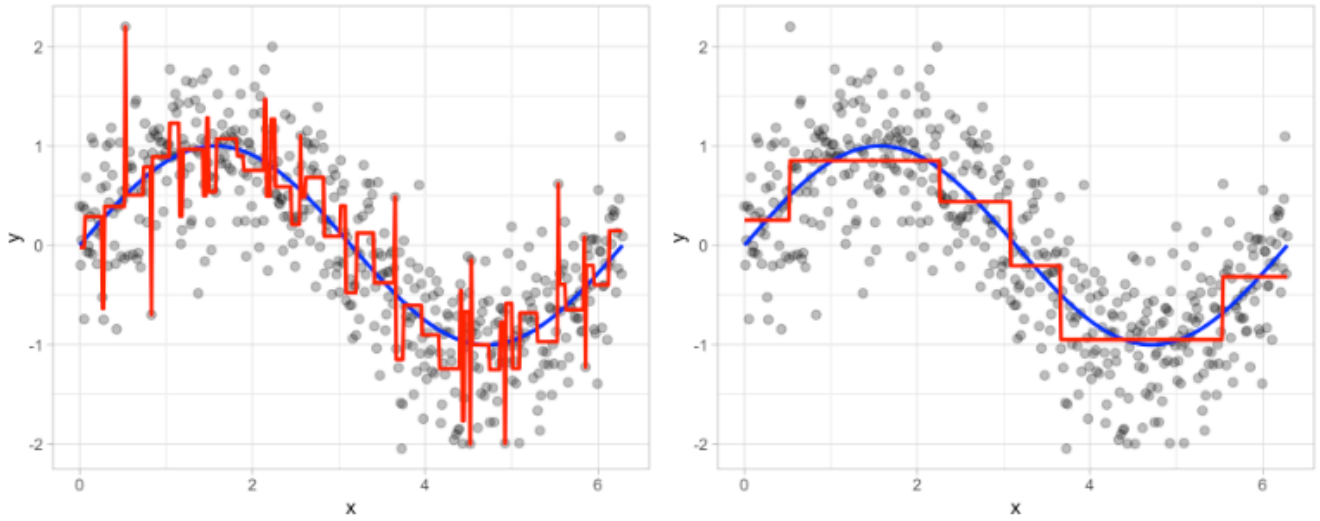
Adapted from ISLR, James et al.

Building a Tree

- It is computationally infeasible to consider every possible partition of the feature space into J boxes.
- For this reason, we take a **top-down, greedy** approach known as **recursive binary splitting**.
  - **top-down** because it begins at the top of the tree and then successively splits the predictor space.
  - **greedy** because at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

Tree Pruning

The process described above may **overfit** the data.



Tree Pruning

- Grow a very large tree, and then **prune** it back to obtain a **subtree**.
- The technique uses is known as **cost complexity pruning** (also known as **weakest link pruning**).
- Consider a sequence of trees indexed by α. For each α, consider the tree that minimizes

$$SSE + \alpha \left|T\right|$$

- Choose optimal α by CV.

Regression Tree: Implementation

Ames Housing Dataset

```
ames <- readRDS("AmesHousing.rds")   # load dataset
```

```
# reorder levels of 'Overall_Qual'
ames$Overall_Qual <- factor(ames$Overall_Qual, levels = c("Very_Poor", "Poor", "Fair", "Below_Average",
                                                          "Average", "Above_Average", "Good", "Very_Good",
                                                          "Excellent", "Very_Excellent"))
```

```
# split data

set.seed(050924)   # set seed

train_index <- createDataPartition(y = ames$Sale_Price, p = 0.7, list = FALSE)   # consider 70-30 split

ames_train <- ames[train_index,]   # training data

ames_test <- ames[-train_index,]   # test data
```

Regression Tree: Implementation

Ames Housing Dataset

```
# create recipe and blueprint, prepare and apply blueprint

set.seed(050924)   # set seed

ames_recipe <- recipe(Sale_Price ~ ., data = ames_train)   # set up recipe

blueprint <- ames_recipe %>%
  step_nzv(Street, Utilities, Pool_Area, Screen_Porch, Misc_Val) %>%   # filter out zv/nzv predictors
  step_impute_mean(Gr_Liv_Area) %>%                                    # impute missing entries
  step_integer(Overall_Qual) %>%                                       # numeric conversion of levels of the predictors
  step_center(all_numeric(), -all_outcomes()) %>%                      # center (subtract mean) all numeric predictors
  step_scale(all_numeric(), -all_outcomes()) %>%                       # scale (divide by standard deviation) all numeric predictors
  step_other(Neighborhood, threshold = 0.01, other = "other") %>%      # lumping required predictors
  step_dummy(all_nominal(), one_hot = FALSE)                           # one-hot/dummy encode nominal categorical predictors


prepare <- prep(blueprint, data = ames_train)    # estimate feature engineering parameters based on training data


baked_train <- bake(prepare, new_data = ames_train)   # apply the blueprint to training data

baked_test <- bake(prepare, new_data = ames_test)    # apply the blueprint to test data
```

Regression Tree: Implementation

Ames Housing Dataset

Implement CV to tune the hyperparameter.

```
set.seed(050924)   # set seed

cv_specs <- trainControl(method = "repeatedcv", number = 5, repeats = 5)   # CV specifications


library(rpart)   # for trees

tree_cv <- train(blueprint,
                 data = ames_train,
                 method = "rpart",
                 trControl = cv_specs,
                 tuneLength = 20,                  # considers a grid of 20 possible tuning parameter values
                 metric = "RMSE")
```

```
# results from the CV procedure

tree_cv$bestTune    # optimal hyperparameter
```

```
##           cp
## 1 0.002631828
```

```
min(tree_cv$results$RMSE)   # optimal CV RMSE
```

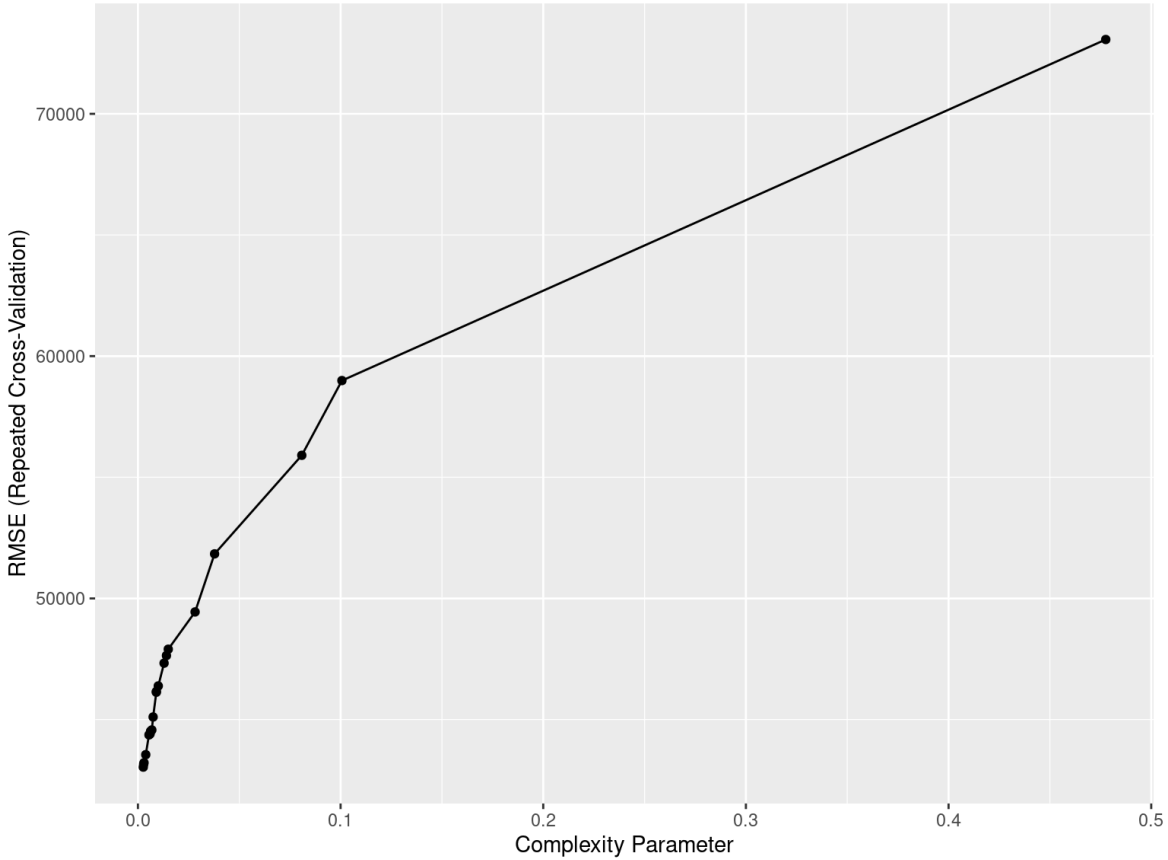
```
## [1] 43039.21
```

Regression Tree: Implementation

Ames Housing Dataset

Results from the CV procedure.

```
ggplot(tree_cv)
```



Regression Tree: Implementation

Ames Housing Dataset

```
# build final model

final_model <- rpart(formula = Sale_Price ~ .,
                     data = baked_train,
                     cp = tree_cv$bestTune$cp,
                     xval = 0,                # no further CV
                     method = "anova")        # for regression
```

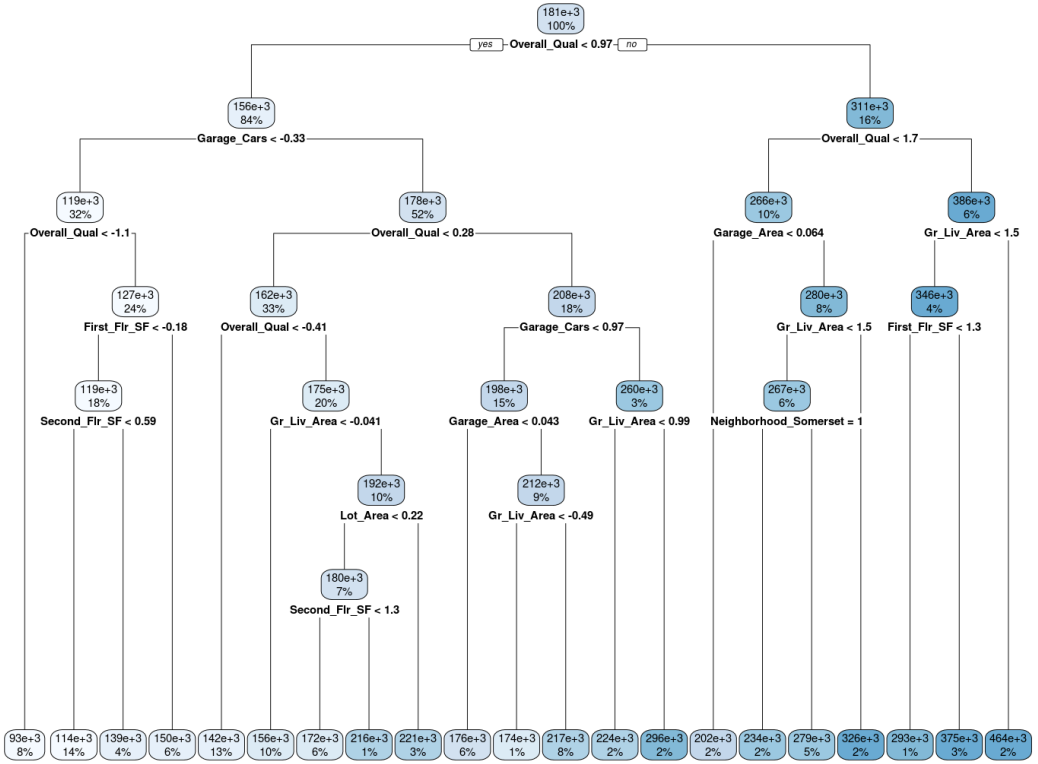
```
# obtain predictions and test set RMSE

final_model_preds <- predict(object = final_model, newdata = baked_test, type = "vector")    # obtain test set predictions

sqrt(mean((final_model_preds - baked_test$Sale_Price)^2))   # calculate test set RMSE
```

```
## [1] 39241.28
```

Regression Tree: Implementation

Ames Housing Dataset

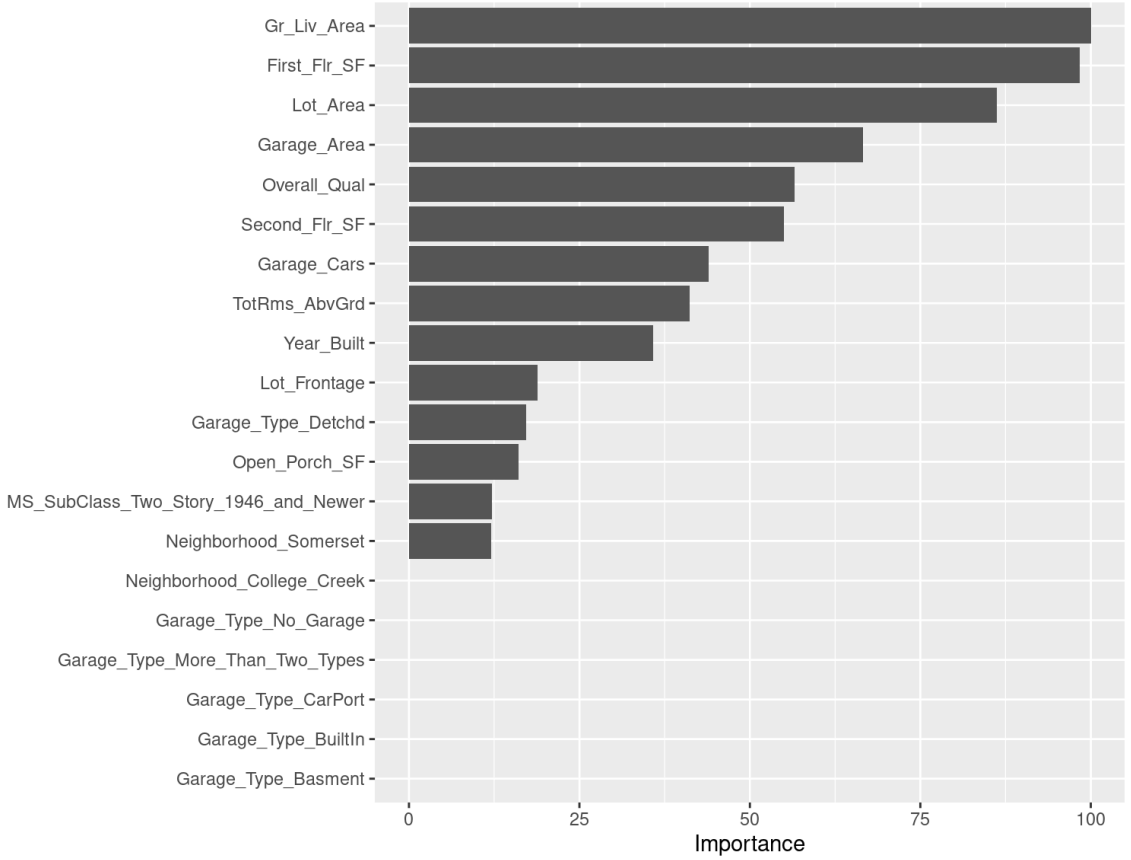
```
library(rpart.plot)
rpart.plot(final_model)
```



Regression Tree: Implementation

Ames Housing Dataset

```
# variable importance

vip(object = tree_cv, num_features = 20, method = "model")
```



Regression Tree: Implementation

Ames Housing Dataset

```
# build full grown tree (no pruning)

final_model_no_prune <- rpart(formula = Sale_Price ~ .,
                              data = baked_train,
                              cp = 0,                  # no pruning
                              xval = 0,                # no CV
                              method = "anova")        # for regression
```

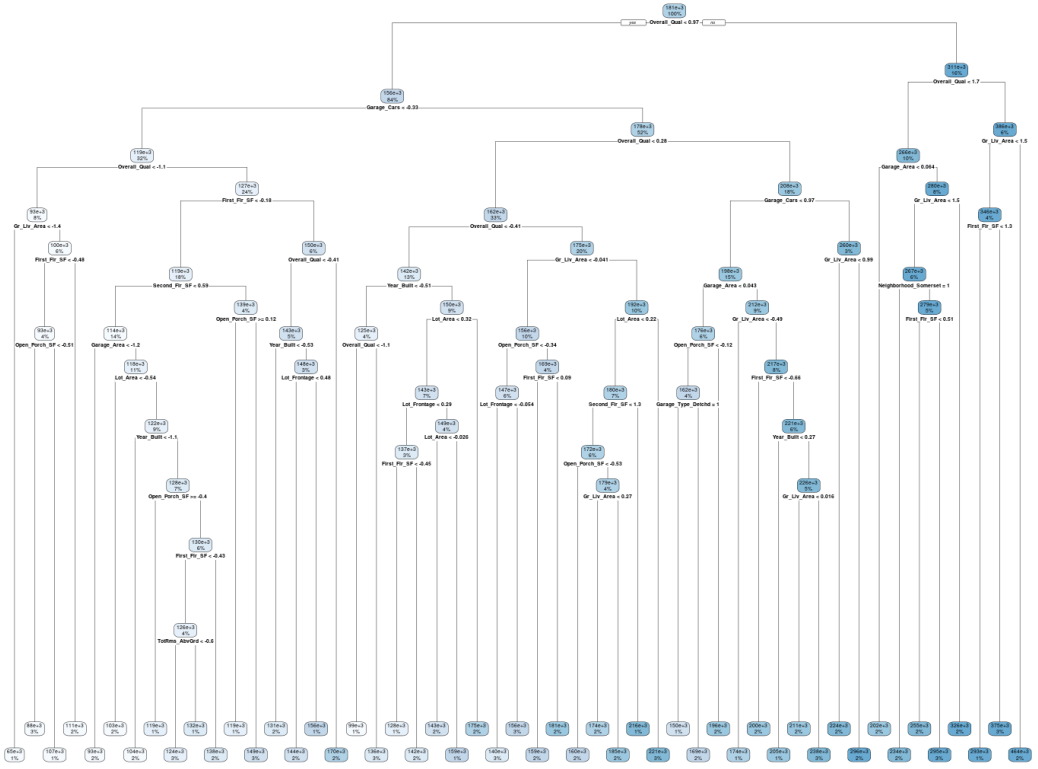
```
# obtain predictions and test set RMSE

final_model_no_prune_preds <- predict(object = final_model_no_prune, newdata = baked_test, type = "vector")    # test set predictions

sqrt(mean((final_model_no_prune_preds - baked_test$Sale_Price)^2))   # calculate test set RMSE
```

```
## [1] 37327.49
```

Regression Tree: Implementation

Ames Housing Dataset

```
rpart.plot(final_model_no_prune)
```



Trees

Advantages

- Easy to explain.
- Closely mirror human decision-making.
- Can be displayed graphically, and are easily interpreted by non-experts.
- Handle qualitative predictors without creating dummy variables. Does not require standardization of predictors.

Disadvantages

- Do not have same level of prediction accuracy.
- Can be very non-robust.

Regression Tree: Implementation

Ames Housing Dataset (minimal feature engineering)

```
# create new blueprint (minimal feature engineering), prepare and apply blueprint

set.seed(050924)   # set seed

ames_recipe <- recipe(Sale_Price ~ ., data = ames_train)   # set up recipe

blueprint_minimal <- ames_recipe %>%
  step_impute_mean(Gr_Liv_Area)                                    # impute missing entries


prepare_minimal <- prep(blueprint_minimal, data = ames_train)    # estimate feature engineering parameters based on training data


baked_train_minimal <- bake(prepare_minimal, new_data = ames_train)   # apply the blueprint to training data

baked_test_minimal <- bake(prepare_minimal, new_data = ames_test)    # apply the blueprint to test data
```

Regression Tree: Implementation

Ames Housing Dataset

Implement CV to tune the hyperparameter.

```
set.seed(050924)   # set seed

cv_specs <- trainControl(method = "repeatedcv", number = 5, repeats = 5)   # CV specifications


tree_cv_min_fe <- train(blueprint_minimal,
                        data = ames_train,
                        method = "rpart",
                        trControl = cv_specs,
                        tuneLength = 20,                  # considers a grid of 20 possible tuning parameter values
                        metric = "RMSE")
```

```
# results from the CV procedure

tree_cv_min_fe$bestTune    # optimal hyperparameter
```

```
##           cp
## 1 0.003232143
```

```
min(tree_cv_min_fe$results$RMSE)   # optimal CV RMSE
```

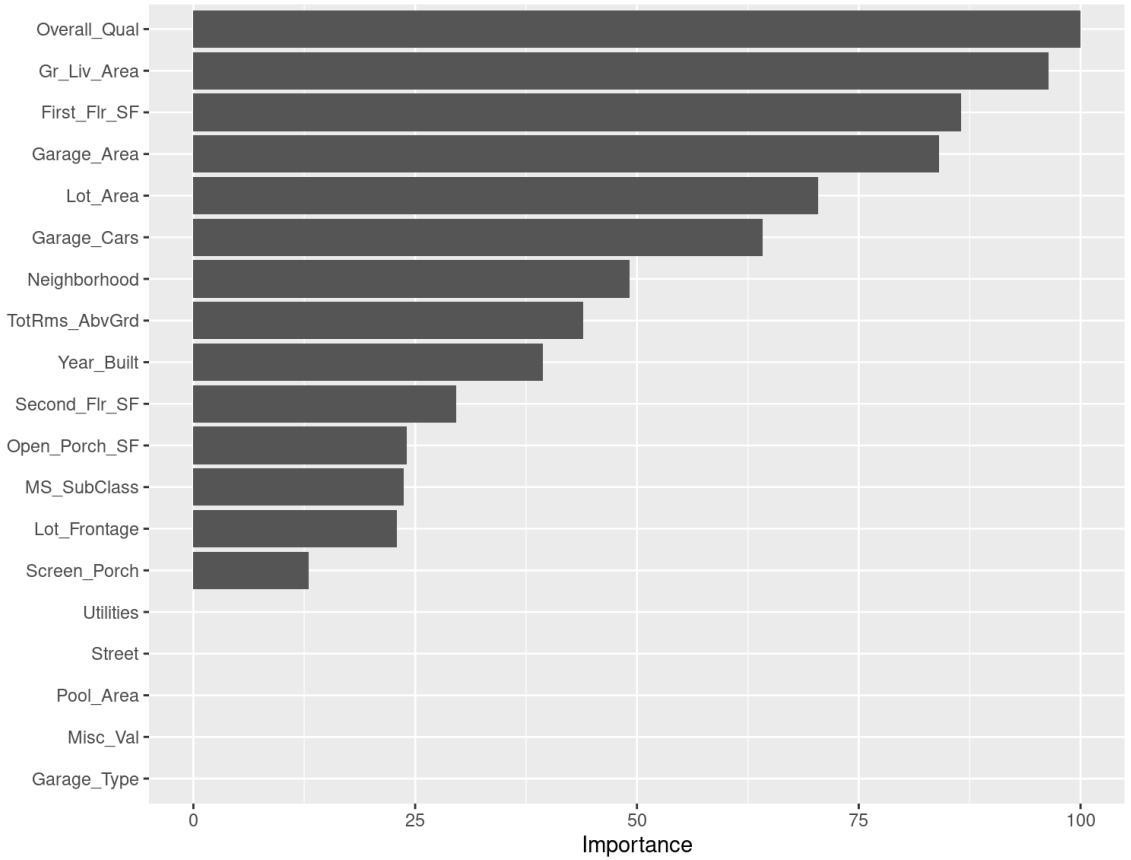
```
## [1] 43521.51
```

```
# doing feature engineering seems to result in a slightly better performance
```

Regression Tree: Implementation

Ames Housing Dataset

```
# variable importance

vip(object = tree_cv_min_fe, num_features = 20, method = "model")
```



Classification Trees

- Still use **recursive binary splitting** to grow a classification tree.
- SSE can be replaced by
  - **classification error rate**, the fraction of the training observations in that region that do not belong to the most common class.

$$E = 1 - \max_k (\hat{p}_{mk})$$

  - **Gini index**, a measure of node purity—a small value indicates that a node contains predominantly observations from a single class.

$$G = \sum_{k=1}^{K} \hat{p}_{mk} (1 - \hat{p}_{mk})$$

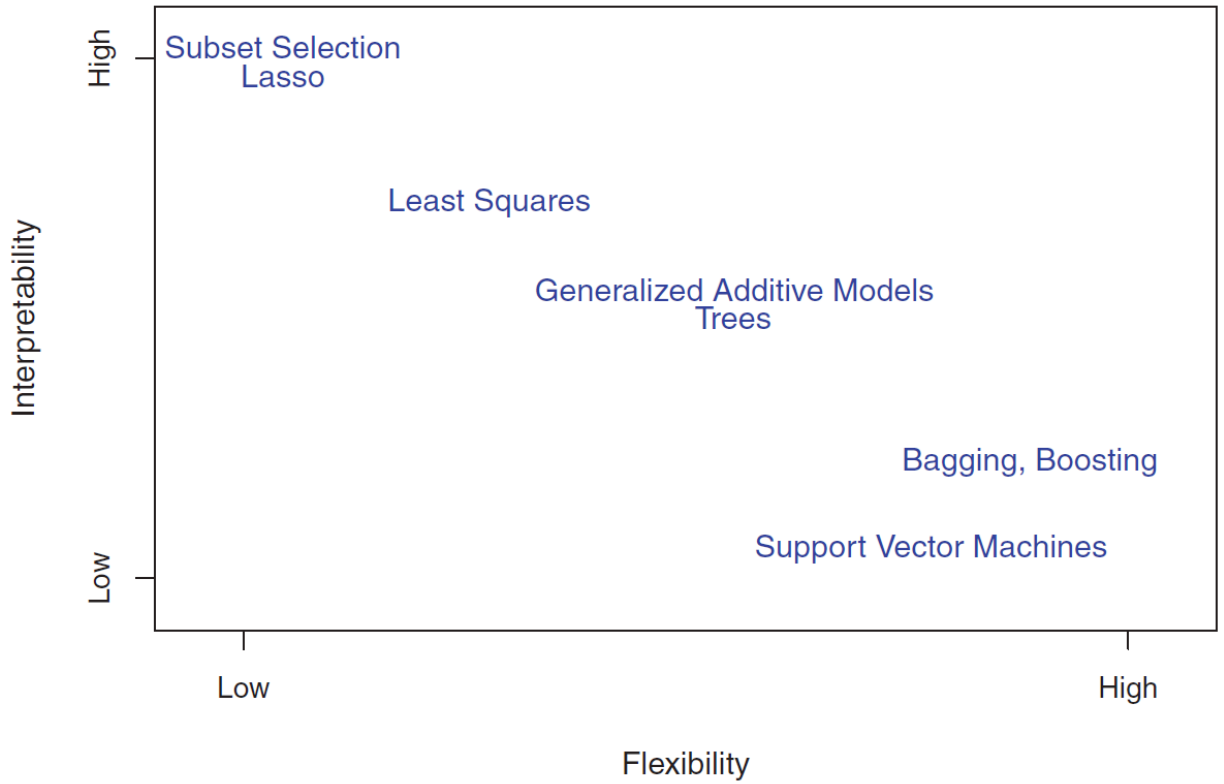
Here $\hat{p}_{mk}$ represents the proportion of training observations in the m^{th} region that are from the k^{th} class.

Ensemble Methods

Single regression or classification trees usually have poor predictive performance.

Ensemble Methods use a collection of multiple trees to improve the predictive performance at the cost of interpretability.

- Bagging
- Random Forests
- Boosting



Adapted from ISLR, James et al.

Bagging

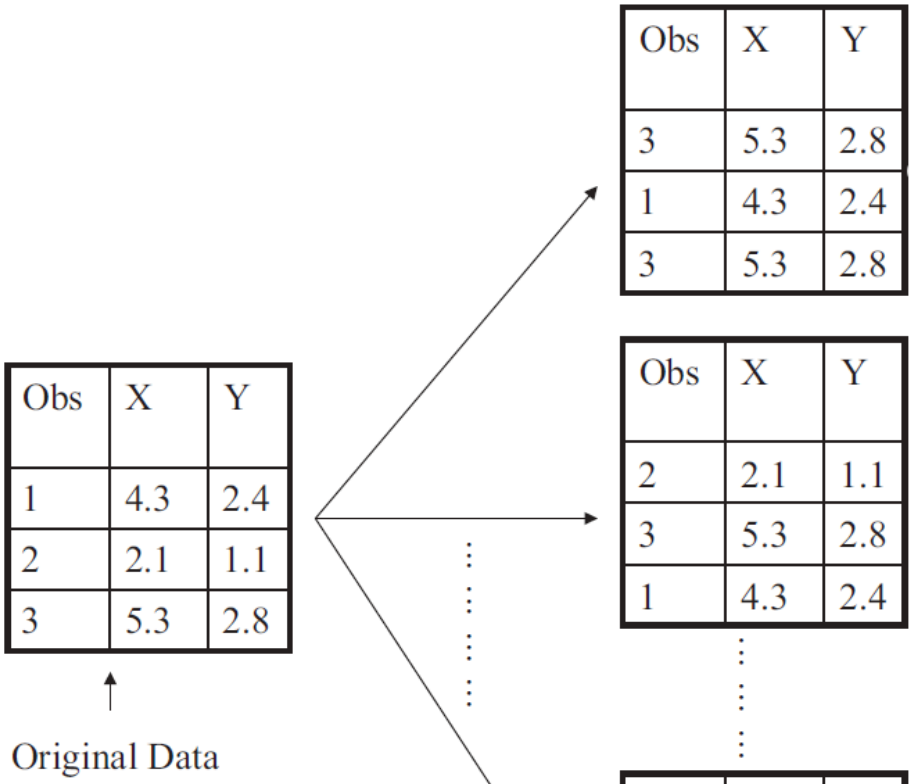
- **Bootstrap aggregation** or **bagging** is a general-purpose procedure for reducing the variance of a statistical learning method.
- **Idea**: Build multiple trees and average their results.
- **Result**: Given a set of n independent observations (random variables) $Z_1, \ldots, Z_n$, each with variance σ^2, the variance of the mean/average $\bar{Z} = \dfrac{Z_1 + Z_2 + \cdots + Z_n}{n}$ of the observations is σ^2/n.

In other words, **averaging a set of observations reduces variance**.

- In reality, we do not have multiple training datasets.

Bagging

Bootstrapping



Adapted from ISLR, James et al.

Bagging

- Take repeated bootstrap samples (say B) from the original (single) available dataset.
- Build a tree on each bootstrap sample and obtain predictions $\hat{f}^{*b}(x),\ b = 1, 2, \ldots, B$.
- Average all the predictions.

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x)$$

- Individual trees are grown deep and are not pruned. They have high variance, but low bias.
- For classification trees, take **majority vote**: the overall prediction is the most commonly occurring class among the B predictions.

Out-of-Bag Error Estimation

- A straightforward way to estimate the test error of a bagged model, without performing CV.
- It can be shown that on average, each bagged tree (constructed on each bootstrap sample) makes use of around two-thirds of the observations.
- Remaining one-third observations not used train a bagged tree are referred to as **out-of-bag (OOB)** observations.
- For i^{th} observation, use the trees in which that observation was OOB. This will yield around $B/3$ predictions for the i^{th} observation. Take their average to obtain a single prediction.
- Equivalent to LOOCV if B is large.

Variable Importance Measures

- Bagging improves prediction accuracy at the expense of interpretability.
- However, one can still obtain an overall summary of the importance of each predictor.
- To measure feature importance, the reduction in the loss function (e.g., RSS) attributed to each variable at each split is tabulated. In some instances, a single variable could be used multiple times in a tree; consequently, the total reduction in the loss function across all splits by a variable are summed up and used as the total feature importance.
- A large value indicates an important predictor.

Bagging: Implementation

Ames Housing Dataset

Data splitting and feature engineering has been done in the previous slides.

```
set.seed(051424)   # set seed

library(ipred)   # for bagging

bag_fit <- bagging(formula = Sale_Price ~ .,
                   data = baked_train,
                   nbagg = 500,                              # number of trees to grow (bootstrap samples) usually 500
                   coob = TRUE,                              # yes to computing OOB error estimate
                   control = rpart.control(minsplit = 2,     # split a node if at least 2 observations present
                                           cp = 0,           # no pruning (let the trees grow tall)
                                           xval = 0))        # no CV
```

```
bag_fit   # results of bagging
```

```
##
## Bagging regression trees with 500 bootstrap replications
##
## Call: bagging.data.frame(formula = Sale_Price ~ ., data = baked_train,
##     nbagg = 500, coob = TRUE, control = rpart.control(minsplit = 2,
##         cp = 0, xval = 0))
##
## Out-of-bag estimate of root mean squared error:  36038.11
```

```
bag_fit$err   # OOB RMSE estimate
```

```
## [1] 36038.11
```

Bagging: Implementation

Ames Housing Dataset

CV with bagging (NOT recommended since computationally expensive)

```
set.seed(051424)   # set seed

cv_specs <- trainControl(method = "repeatedcv", number = 5, repeats = 1)   # CV specifications

library(ipred)
library(e1071)

bagging_cv <- train(blueprint,
                    data = ames_train,
                    method = "treebag",
                    trControl = cv_specs,
                    nbagg = 500,
                    control = rpart.control(minsplit = 2, cp = 0),
                    metric = "RMSE")
```

Bagging: Implementation

```
# obtain predictions on the test set

final_model_preds <- predict(object = bag_fit, newdata = baked_test)     # use 'type = "class"' for classification trees

sqrt(mean((final_model_preds - baked_test$Sale_Price)^2))   # test set RMSE
```

```
## [1] 28590.5
```

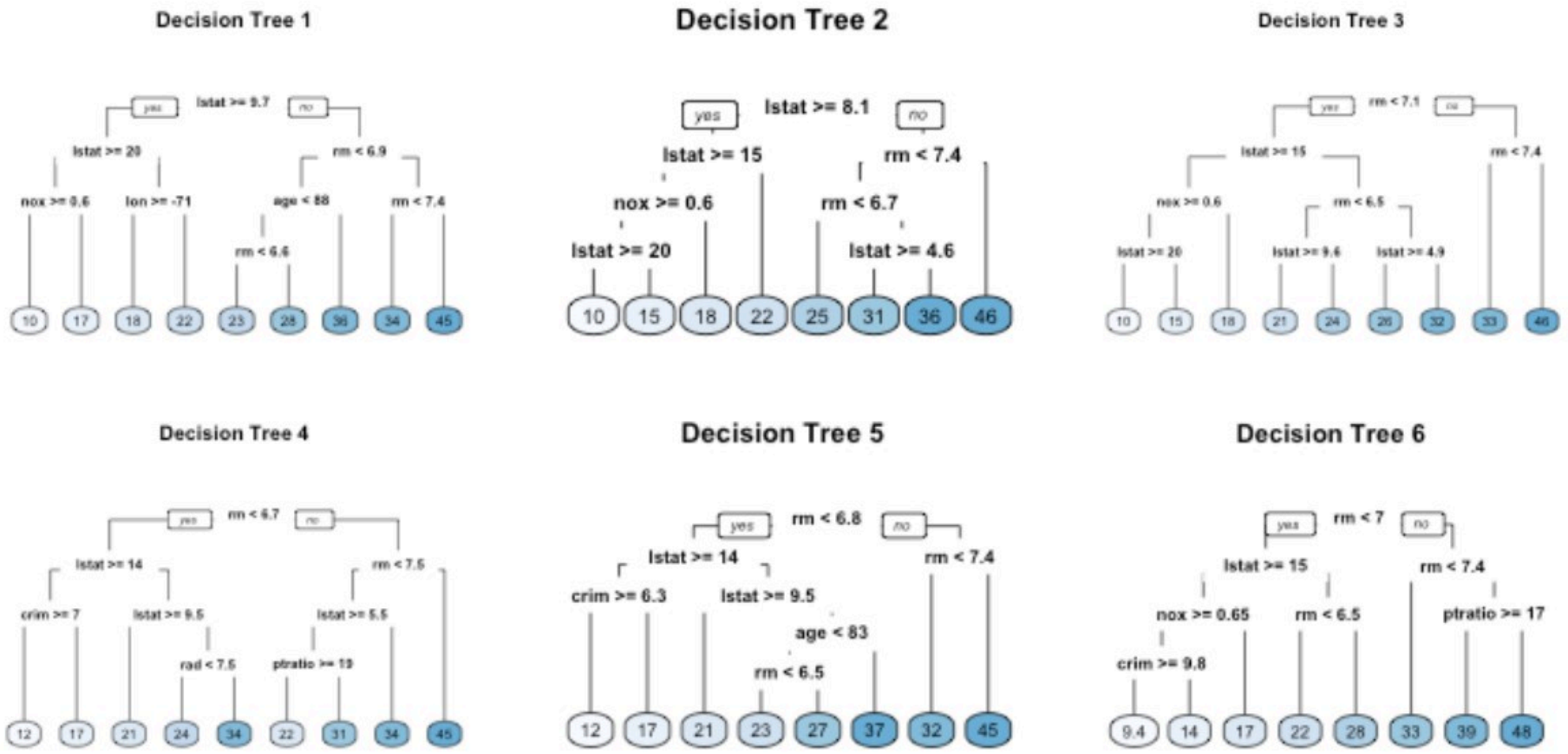
```
# variable importance

imp <- varImp(bag_fit)      # look at the object created
```

Bagging: Disadvantages

- Bagging improves the prediction accuracy for high variance (and low bias) models at the expense of interpretability and computational speed.
- However, although the model building steps are independent, the trees in bagging are not completely independent of each other since all the original features are considered at every split of every tree. Rather, trees from different bootstrap samples typically have similar structure to each other (especially at the top of the tree) due to any underlying strong relationships.
- This characteristic is known as **tree correlation** and prevents bagging from further reducing the variance of the individual models. **Random forests** extend and improve upon bagged decision trees by reducing this correlation and thereby improving the accuracy of the overall ensemble.

Bagging: Disadvantages

Tree Correlation in Bagging



Adapted from HMLR, Boehmke & Greenwell

Random Forests

- Provide an improvement over bagged trees by reducing the variance further (by **decorrelating**) when we average the trees.
- As in bagging, we build a number of decision trees on bootstrapped training samples.
- For each tree, each time a split is considered, a **random selection of m predictors** is chosen (split candidates) from the full set of p predictors. The split is allowed to use only one of those m predictors. Note that in bagging, each split for each tree considers all p predictors as split candidates.
- A fresh sample of m predictors is taken at each split. Typical default values are $m = p/3$ (regression) and $m = \sqrt{p}$ (classification) but this should be considered a tuning parameter, to be chosen by CV.

Random Forests: Implementation

Ames Housing Dataset

Data splitting and feature engineering has been done in the previous slides.

```
set.seed(051424)   # set seed

cv_specs <- trainControl(method = "cv", number = 5)   # CV specifications


library(ranger)
library(e1071)


param_grid <- expand.grid(mtry = seq(1, 30, 1),     # sequence of 1 to at least half the number of predictors
                          splitrule = "variance",   # use "gini" for classification
                          min.node.size = 2)        # for each tree

rf_cv <- train(blueprint,
               data = ames_train,
               method = "ranger",
               trControl = cv_specs,
               tuneGrid = param_grid,
               metric = "RMSE")

rf_cv$bestTune$mtry   # optimal tuning parameter
```

```
## [1] 15
```

```
min(rf_cv$results$RMSE)   # optimal CV RMSE
```

```
## [1] 31758.42
```

Random Forests: Implementation

Ames Housing Dataset

```
# fit final model

final_model <- ranger(formula = Sale_Price ~ .,
                      data = baked_train,
                      num.trees = 500,
                      mtry = rf_cv$bestTune$mtry,
                      splitrule = "variance",
                      min.node.size = 2,
                      importance = "impurity")
```

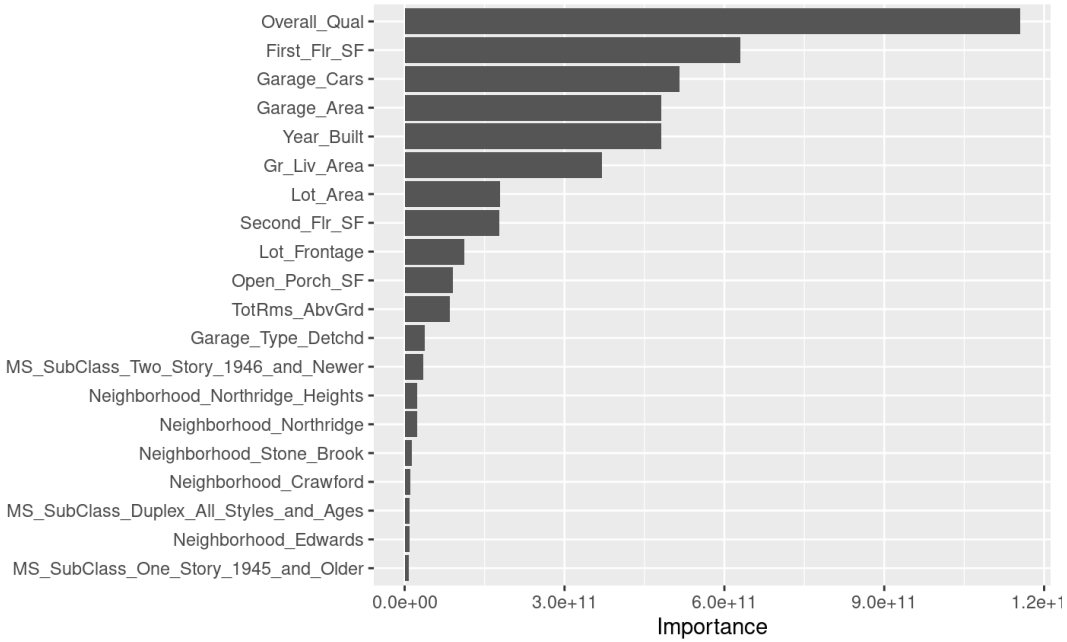
```
# obtain predictions on the test set

final_model_preds <- predict(object = final_model, data = baked_test, type = "response")  # predictions on test set

sqrt(mean((final_model_preds$predictions - baked_test$Sale_Price)^2))  # test set RMSE
```

```
## [1] 26371.64
```

Random Forests: Implementation

Ames Housing Dataset

```
# variable importance

vip(final_model, num_features = 20)      # top 20 most important features
```



Gradient Boosting

Like bagging and random forests, boosting involves combining a large number of decision trees. However, the main idea of boosting is to add new models to the ensemble **sequentially**.

Each model in the process is a weak model, referred to as the **base learner**. The idea behind boosting is that each model in the sequence slightly improves upon the performance of the previous one, thereby **learning slowly**.

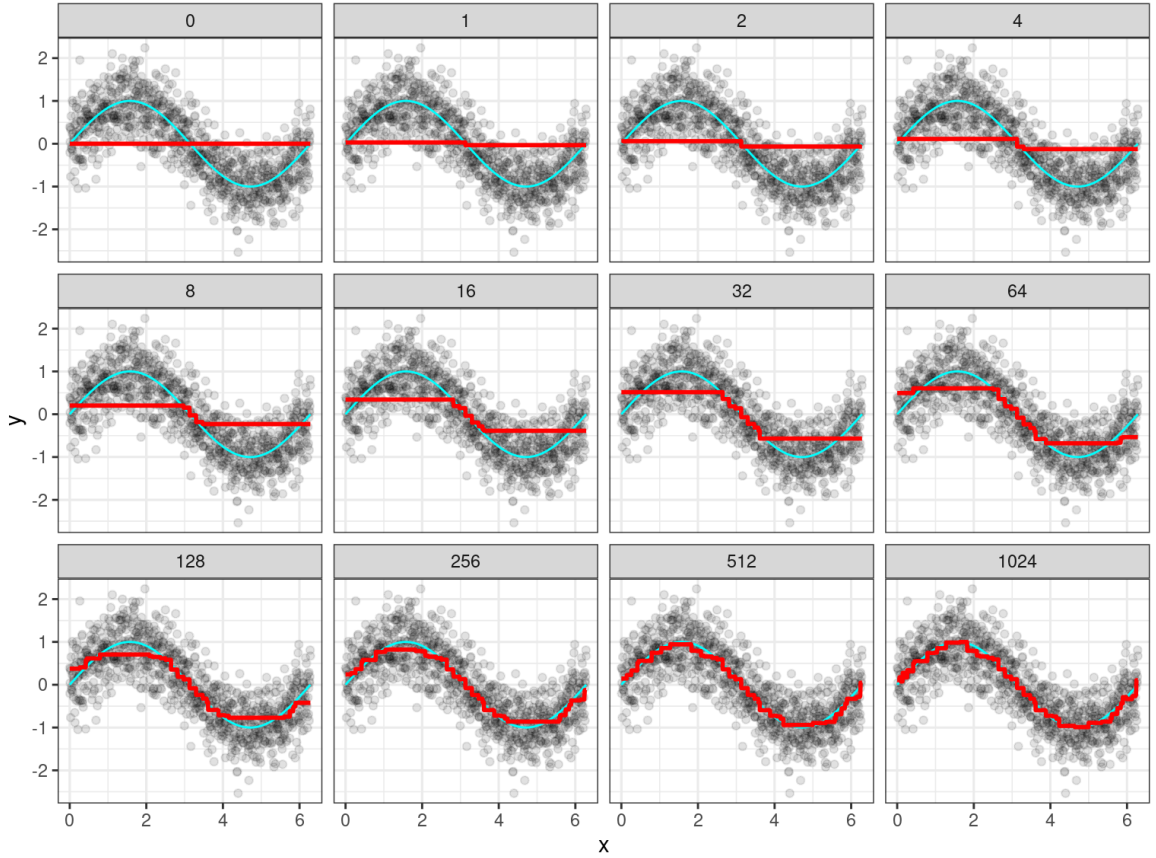
At each step of the process, we fit a tree to the residuals from the previous model, rather than the outcome Y, as the response. The final model is a stagewise additive model of B individual trees.

$$\hat{f}(x) = \sum_{b=1}^{B} \lambda \hat{f}^{b}(x)$$

The process is called **gradient boosting** since it uses the **gradient descent** algorithm to minimize the loss function.

Gradient Boosting

Gradient Boosting



Gradient Boosting

The hyperparameters (to be tuned by CV) include:

- **Number of trees**: Having too many trees can overfit the training data.

- **Shrinkage** (λ): Determines the contribution of each tree to the final model and controls how quickly the algorithm learns.

- **Tree Depth**: Controls the depth of the individual trees.

- **Minimum number of observations in terminal nodes**: Also, controls the complexity of each tree.

Gradient Boosting: Implementation

Ames Housing Dataset

EDA and data splitting has been done in the previous slides.

```
# create recipe and blueprint, prepare and apply blueprint

set.seed(051624)   # set seed

ames_recipe <- recipe(Sale_Price ~ ., data = ames_train)   # set up recipe

blueprint <- ames_recipe %>%
  step_impute_mean(Gr_Liv_Area)                                    # impute missing entries


prepare <- prep(blueprint, data = ames_train)    # estimate feature engineering parameters based on training data


baked_train <- bake(prepare, new_data = ames_train)   # apply the blueprint to training data

baked_test <- bake(prepare, new_data = ames_test)    # apply the blueprint to test data
```

Gradient Boosting: Implementation

Ames Housing Dataset

```
set.seed(051624)   # set seed

cv_specs <- trainControl(method = "repeatedcv", number = 5, repeats = 1)   # CV specifications

library(gbm)

out <- capture.output(
  gbm_cv <- train(blueprint,
                  data = ames_train,
                  method = "gbm",
                  trControl = cv_specs,
                  tuneLength = 10,
                  metric = "RMSE")
  )
```

```
gbm_cv$bestTune   # optimal tuning parameters
```

```
##    n.trees interaction.depth shrinkage n.minobsinnode
## 15     250                 2       0.1             10
```

```
min(gbm_cv$results$RMSE)   # optimal CV RMSE
```

```
## [1] 43471.13
```

Gradient Boosting: Implementation

Ames Housing Dataset

```
# fit final model

final_model <- gbm(formula = Sale_Price ~ .,
                   data = baked_train,
                   n.trees = gbm_cv$bestTune$n.trees,
                   interaction.depth = gbm_cv$bestTune$interaction.depth,
                   n.minobsinnode = gbm_cv$bestTune$n.minobsinnode,
                   shrinkage = gbm_cv$bestTune$shrinkage)
```

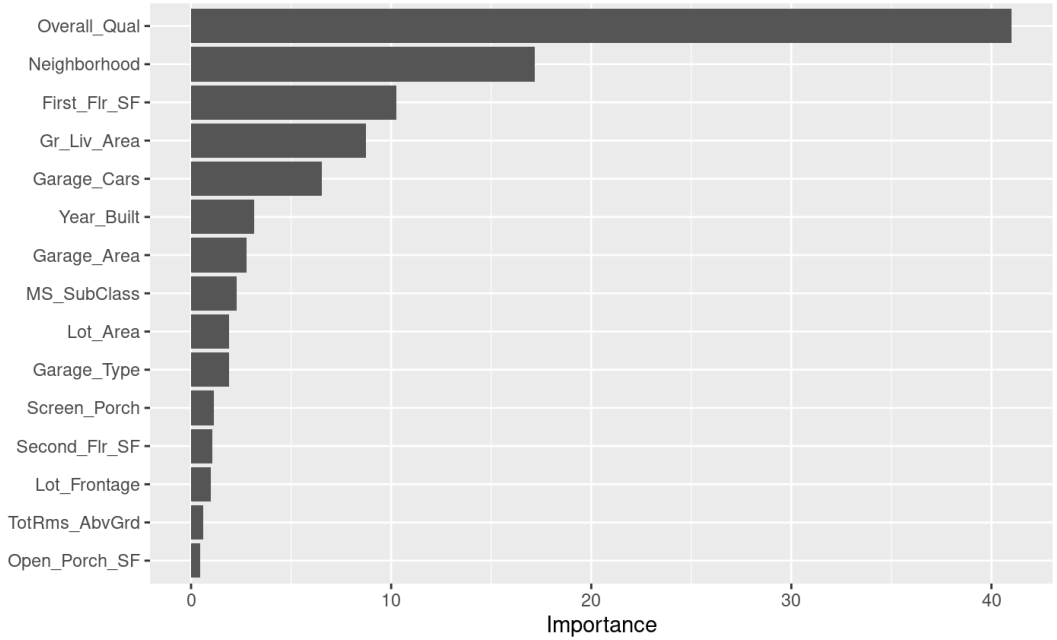
```
## Distribution not specified, assuming gaussian ...
```

```
# obtain predictions on the test set

final_model_preds <- predict(object = final_model, newdata = baked_test, type = "response")  # test set predictions

sqrt(mean((final_model_preds - baked_test$Sale_Price)^2))   # test set RMSE
```

```
## [1] 37795.54
```

Gradient Boosting: Implementation

```
vip(final_model, num_features = 15)
```



Multi-Class Classification: Iris Dataset

We will work with the `iris` dataset which contains measurements in centimeters of four variables for 50 flowers from each of 3 species of iris: setosa, versicolor, and virginica. Please load the dataset using the following code.

```
data(iris)        # load dataset
```

We are interested in predicting `Species` using the rest of the variables in the dataset. Note that this is a **multi-class classification** problem where our response has 3 classes/categories.

Multi-Class Classification: Iris Dataset

Let's investigate the dataset.

```
glimpse(iris)   # all features are numerical
```

```
## Rows: 150
## Columns: 5
## $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4.…
## $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.…
## $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.…
## $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.…
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa, s…
```

```
summary(iris)   # summary of variables
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

Multi-Class Classification: Iris Dataset

Let's investigate the dataset.

```
sum(is.na(iris))  # no missing entries
```

```
## [1] 0
```

```
nearZeroVar(iris, saveMetrics = TRUE)  # no zv/nzv features
```

```
##              freqRatio percentUnique zeroVar   nzv
## Sepal.Length  1.111111      23.33333   FALSE FALSE
## Sepal.Width   1.857143      15.33333   FALSE FALSE
## Petal.Length  1.000000      28.66667   FALSE FALSE
## Petal.Width   2.230769      14.66667   FALSE FALSE
## Species       1.000000       2.00000   FALSE FALSE
```

We do not need to write a blueprint for implementing tree-based methods since trees don't require much feature engineering and there are no missing entries in the dataset.

Note, if you are implementing KNN you would want to scale the features (using `step_center` and `step_scale`)

Multi-Class Classification: Iris Dataset

Data splitting

```
# split the dataset

set.seed(051624)   # set seed

# split the data into training and test sets

index <- createDataPartition(iris$Species, p = 0.8, list = FALSE)

iris_train <- iris[index, ]

iris_test <- iris[-index, ]
```

Multi-Class Classification: Iris Dataset

```
cv_specs <- trainControl(method = "repeatedcv", number = 10, repeats = 5)   # CV specifications
```

```
set.seed(051624)   # set seed

# CV with logistic regression

logistic_cv <- train(Species ~ .,
                     data = iris_train,
                     method = "glm",
                     family = "binomial",
                     trControl = cv_specs,
                     metric = "Accuracy")
```

The code above will throw an error since this is a 3-class classification problem and logistic regression (with family = binomial) works for a binary (2-class) problem.

Multi-Class Classification: Iris Dataset

```
set.seed(051624)   # set seed

# CV with gradient boosting

library(gbm)

out <- capture.output(
  iris_gbm_cv <- train(Species ~ .,
                       data = iris_train,
                       method = "gbm",
                       trControl = cv_specs,
                       tuneLength = 5,
                       metric = "Accuracy")
  )
```

```
iris_gbm_cv$bestTune   # optimal tuning parameters
```

```
##   n.trees interaction.depth shrinkage n.minobsinnode
## 5     250                 1       0.1             10
```

```
max(iris_gbm_cv$results$Accuracy)   # optimal CV Accuracy
```

```
## [1] 0.95
```

Multi-Class Classification: Iris Dataset

```
# final model

gbm_fit <- gbm(formula = Species ~ .,
               data = iris_train,
               n.trees = iris_gbm_cv$bestTune$n.trees,
               interaction.depth = iris_gbm_cv$bestTune$interaction.depth,
               n.minobsinnode = iris_gbm_cv$bestTune$n.minobsinnode,
               shrinkage = iris_gbm_cv$bestTune$shrinkage)
```

```
## Distribution not specified, assuming multinomial ...
```

We will need to write a bit of code to convert the probability predictions into class label predictions. The predicted class is the label with the highest (maximum) probability out of the 3 classes.

```
# probability predictions for each class

final_model_prob_preds <- matrix(predict(object = gbm_fit, newdata = iris_test, type = "response"),
                                 nrow = length(iris_test$Species),
                                 ncol = n_distinct(iris_test$Species))



# class label predictions for each class

final_model_class_preds <- factor(levels(iris_test$Species)[apply(final_model_prob_preds, 1, which.max)])
```

Multi-Class Classification: Iris Dataset

```
# confusion matrix

confusionMatrix(data = final_model_class_preds, reference = iris_test$Species)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa         10          0         0
##   versicolor      0         10         3
##   virginica       0          0         7
##
## Overall Statistics
##
##                Accuracy : 0.9
##                  95% CI : (0.7347, 0.9789)
##     No Information Rate : 0.3333
##     P-Value [Acc > NIR] : 1.665e-10
##
##                   Kappa : 0.85
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: setosa Class: versicolor Class: virginica
## Sensitivity                 1.0000            1.0000           0.7000
## Specificity                 1.0000            0.8500           1.0000
## Pos Pred Value              1.0000            0.7692           1.0000
## Neg Pred Value              1.0000            1.0000           0.8696
## Prevalence                  0.3333            0.3333           0.3333
## Detection Rate              0.3333            0.3333           0.2333
## Detection Prevalence        0.3333            0.4333           0.2333
## Balanced Accuracy           1.0000            0.9250           0.8500
```