

Chapter 9 – Classification and Regression Trees

**Machine Learning for Business
Analytics in R (2nd ed)**

Shmueli, Bruce, Gedeck, Yahav & Patel

Trees and Rules

Goal: Classify or predict an outcome based on a set of predictors

The output is a set of **rules**

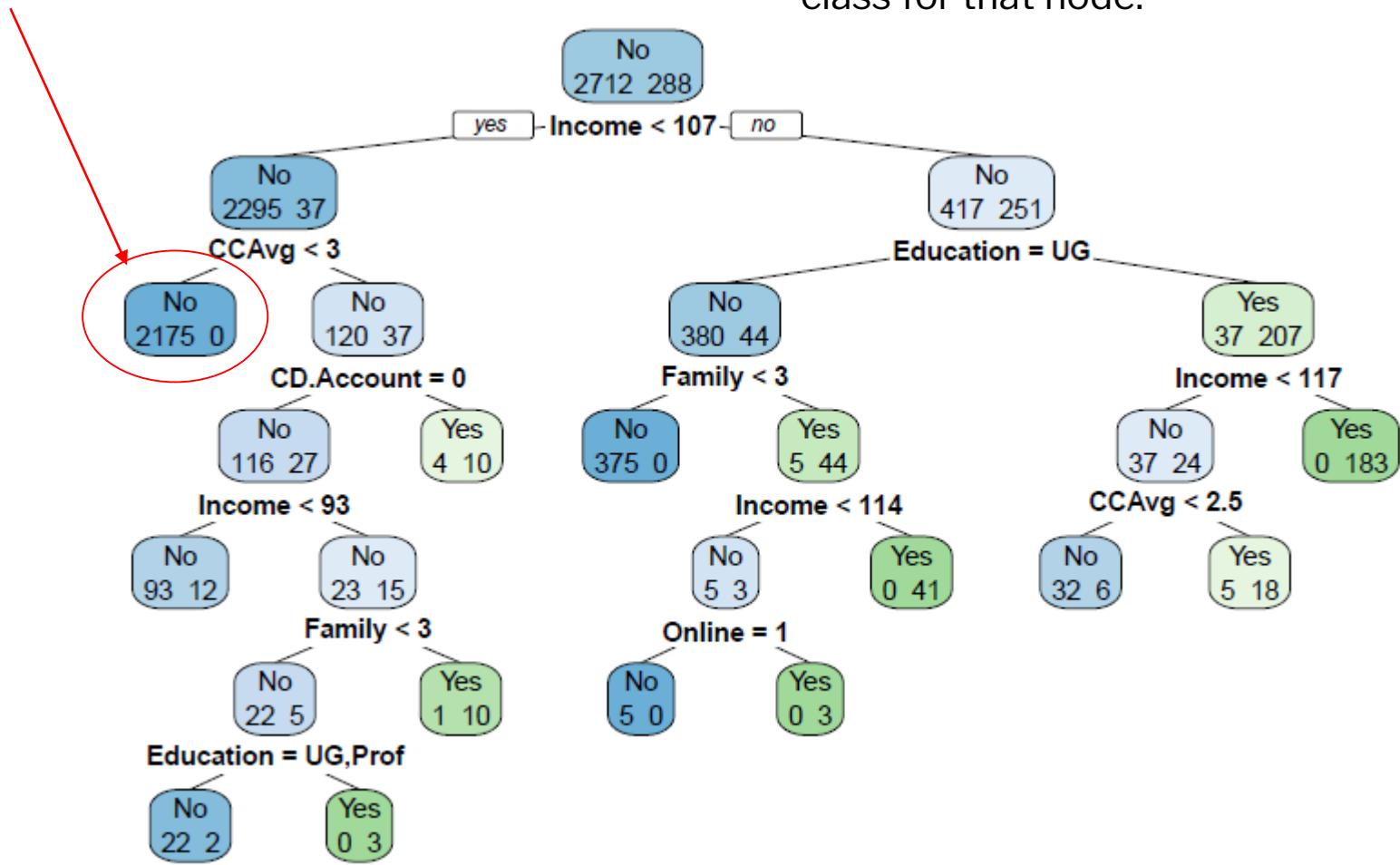
Example:

- Goal: classify a record as “will accept credit card offer” or “will not accept”
- Rule might be “IF (Income ≥ 106) AND (Education < 1.5) AND (Family ≤ 2.5) THEN Class = 0 (nonacceptor)
- Also called CART, Decision Trees, or just Trees
- Rules are represented by tree diagrams

Will bank customers accept offer to take a loan?

Read top-down to get rules,
e.g. for this node: “IF (Income < 107) AND (CCAvg < 3) THEN Class = No (nonacceptor)

Cases are “dropped” down the tree until they reach a decision (terminal) node, where they are assigned the class for that node.



How Is the Tree Produced?

Recursive partitioning: Repeatedly split the records into two parts so as to achieve maximum homogeneity of outcome within each new part

Stopping Tree Growth: A fully grown tree is too complex and will overfit

Recursive Partitioning

Recursive Partitioning Steps

- Pick one of the predictor variables, x_i
- Pick a value of x_i , say s_i , that divides the training data into two (not necessarily equal) portions
- Measure how “pure” or homogeneous each of the resulting portions is
“Pure” = containing records of mostly one class (or, for prediction, records with similar outcome values)
- Algorithm tries different values of x_i and s_i to maximize purity in initial split
- After you get a “maximum purity” split, repeat the process for a second split (on any variable), and so on

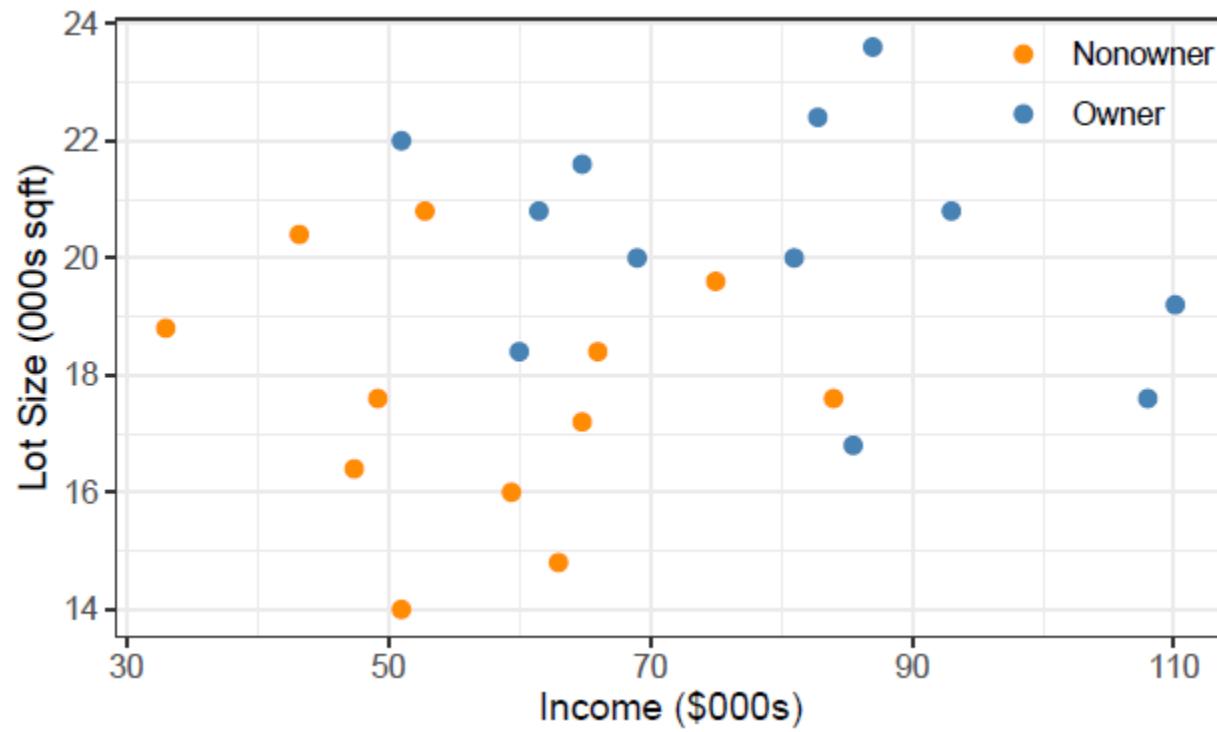
Example: Riding Mowers

- Goal: Classify 24 households as owning or not owning riding mowers
- Predictors = Income, Lot Size

`library rpart` for running trees, function `prp` in
`library rpart.plot` to plot them

Income	Lot_Size	Ownership
60.0	18.4	owner
85.5	16.8	owner
64.8	21.6	owner
61.5	20.8	owner
87.0	23.6	owner
110.1	19.2	owner
108.0	17.6	owner
82.8	22.4	owner
69.0	20.0	owner
93.0	20.8	owner
51.0	22.0	owner
81.0	20.0	owner
75.0	19.6	non-owner
52.8	20.8	non-owner
64.8	17.2	non-owner
43.2	20.4	non-owner
84.0	17.6	non-owner
49.2	17.6	non-owner
59.4	16.0	non-owner
66.0	18.4	non-owner
47.4	16.4	non-owner
33.0	18.8	non-owner
51.0	14.0	non-owner
63.0	14.8	non-owner

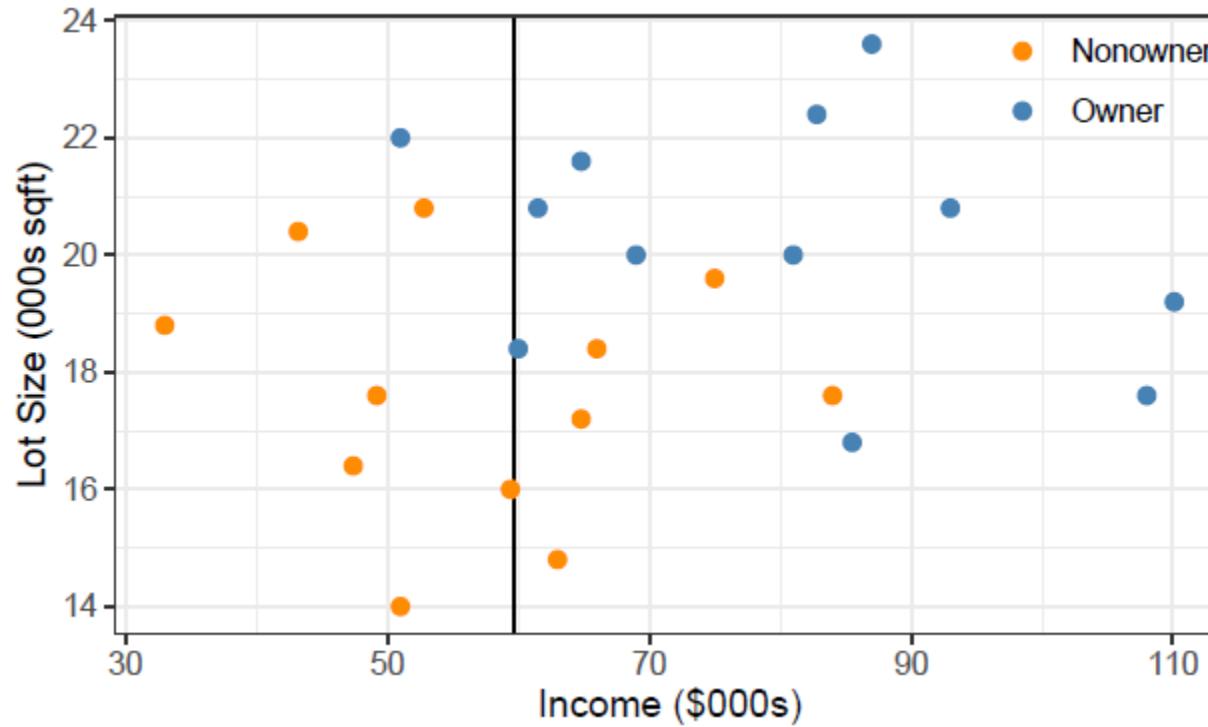
Riding Mowers - Scatter Plot



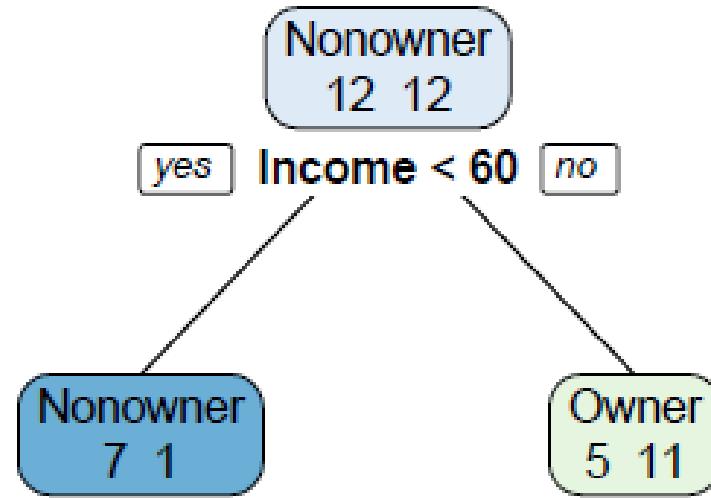
How to split

- Order records according to one variable, say income
- Take a predictor value, say 60 (the first record) and divide records into those with $\text{income} \geq 60$ and those < 60
- Measure resulting purity (homogeneity) of class in each resulting portion
- Try all other split values
- Repeat for other variable(s)
- Select the one variable & split that yields the most purity

The first split: Income = 60

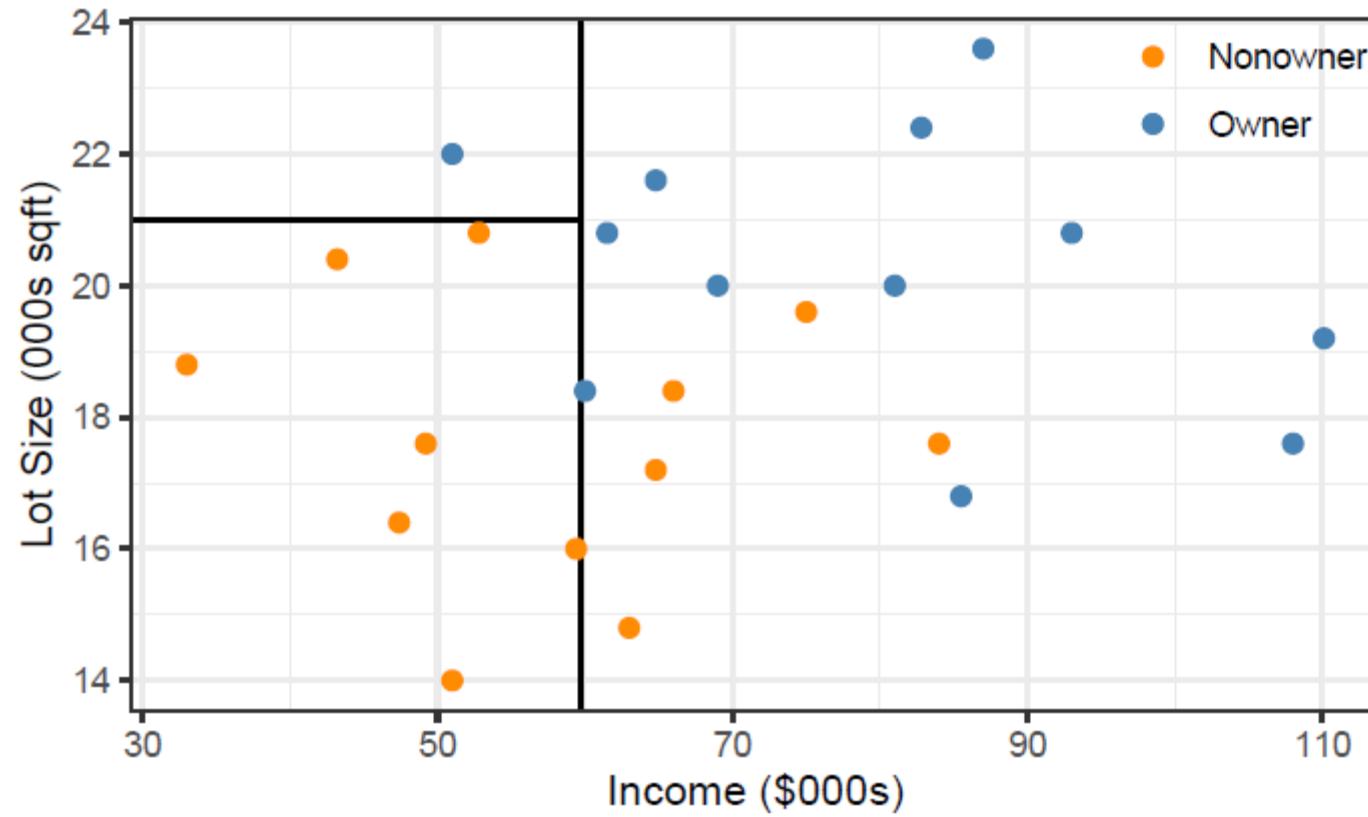


First Split – The Tree

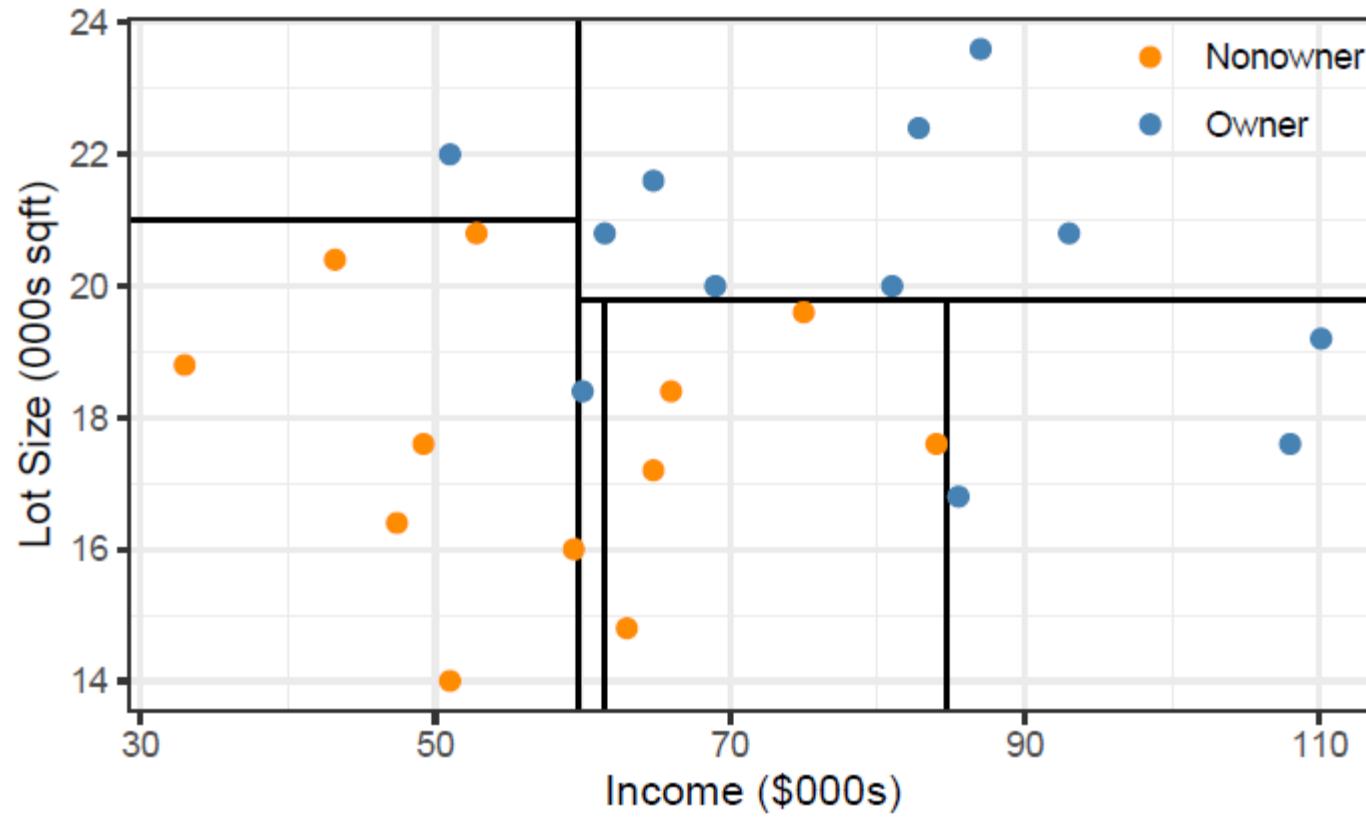


In this node there are 11 owners and 5 non-owners, the dominant class is owner.

Second Split: Lot size = 21

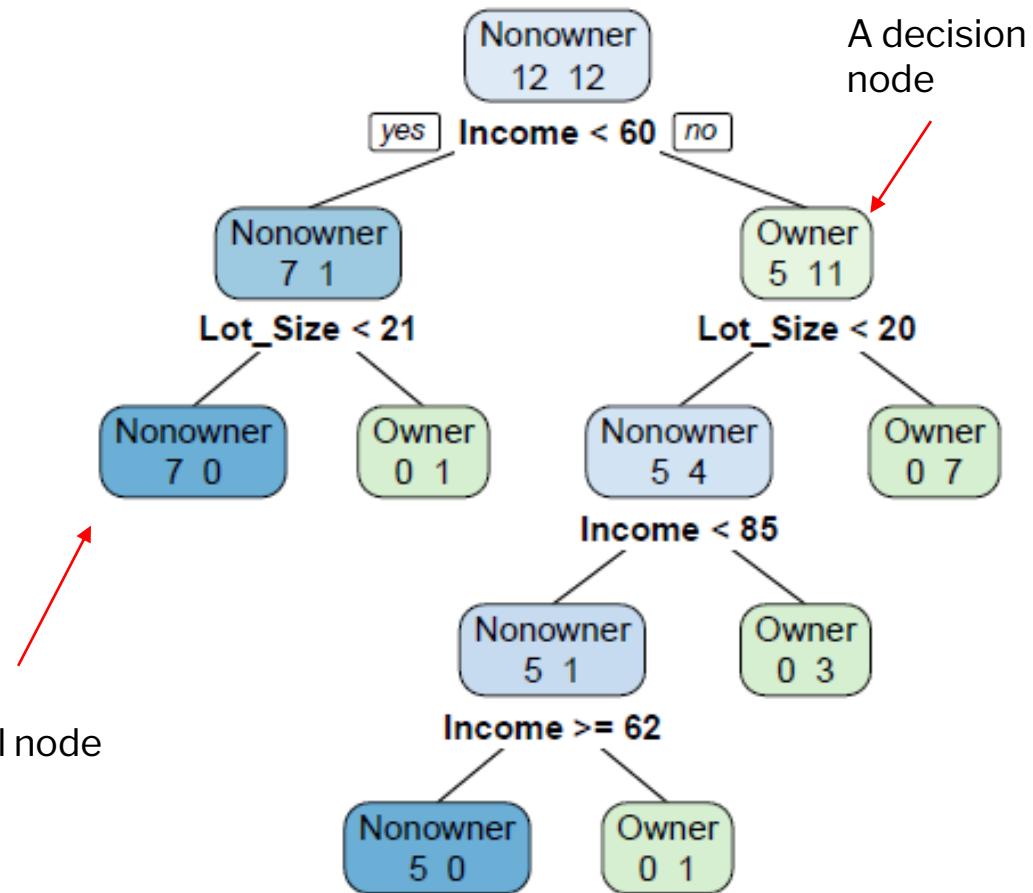


After All Splits



Tree after all splits

The first split is on Income at 60, then the next split is on Lot Size for both the low income group (at lot size 21) and the high income split (at lot size 20), then there are further splits.



Handling Categorical Variables

- Examine all possible ways in which the categories can be split.
- E.g., categories A, B, C can be split 3 ways
 - {A} and {B, C}
 - {B} and {A, C}
 - {C} and {A, B}
- With many categories, # of splits becomes huge

Code for the loan acceptance example (tree shown at the beginning)

Use tidyverse and caret libraries

```
# Load and preprocess data
bank.df <- mlba::UniversalBank %>%
  # Drop ID and zip code columns.
  select(-c(ID, ZIP.Code)) %>%
  # convert Personal.Loan to a factor with labels Yes and No
  mutate(Personal.Loan = factor(Personal.Loan, levels=c(0, 1), labels=c("No",
    "Yes")), Education = factor(Education, levels=c(1, 2, 3), labels=c("UG",
    "Grad", "Prof")))

# partition
set.seed(1)
idx <- createDataPartition(bank.df$Personal.Loan, p=0.6, list=FALSE)
train.df <- bank.df[idx, ]
holdout.df <- bank.df[-idx, ]

# classification tree
default.ct <- rpart(Personal.Loan ~ ., data=train.df, method="class")
# plot tree
rpart.plot(default.ct, extra=1, fallen.leaves=FALSE)
```

Measuring Impurity

Gini Index

Gini Index for rectangle A

$$I(A) = 1 - \sum_{k=1}^m p_k^2$$

p = proportion of cases in rectangle A that belong to class k (out of m classes)

- $I(A) = 0$ when all cases belong to same class
- Max value when all classes are equally represented (= 0.50 in binary case)

Entropy

$$\text{entropy}(A) = - \sum_{k=1}^m p_k \log_2(p_k)$$

p = proportion of cases in rectangle A that belong to class k (out of m classes)

- Entropy ranges between 0 (most pure) and $\log_2(m)$ (equal representation of classes)

Impurity and Recursive Partitioning

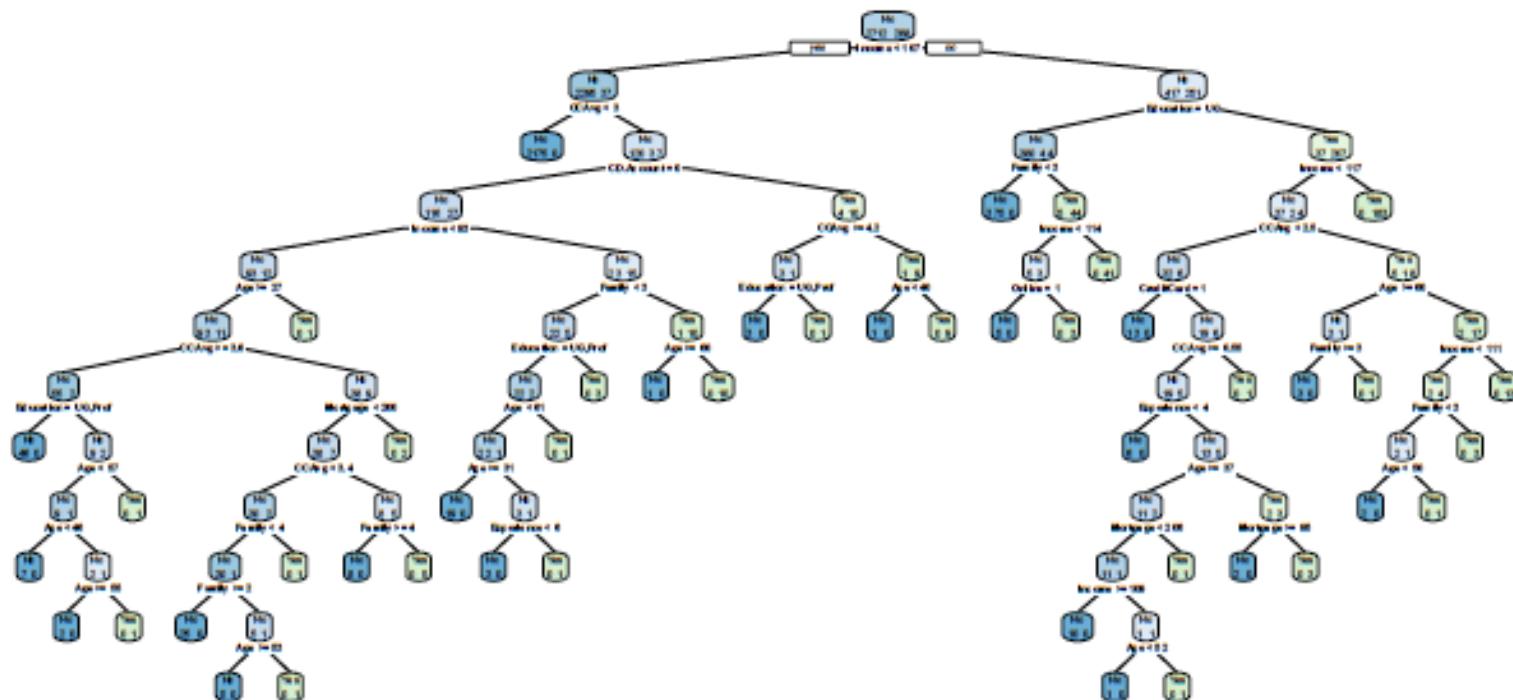
- Obtain overall impurity measure (weighted avg. of individual rectangles)
- At each successive stage, compare this measure across all possible splits in all variables
- Choose the split that reduces impurity the most
- Chosen split points become nodes on the tree

The Overfitting Problem

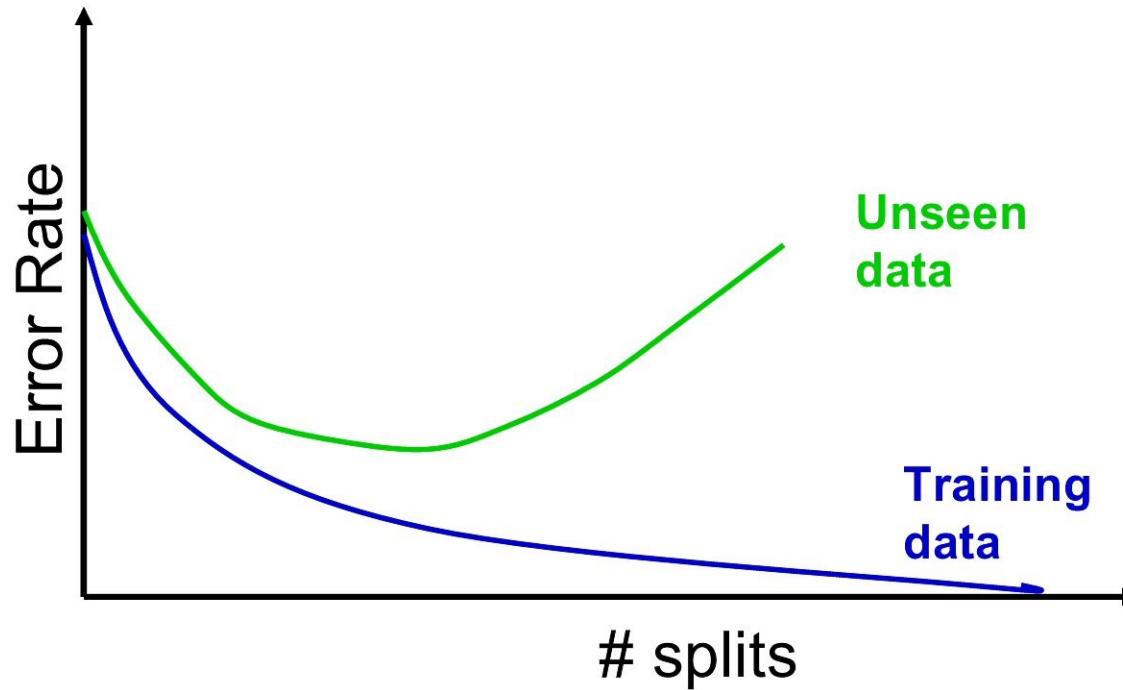
Full trees are complex and overfit the data

- Natural end of process is 100% purity in each leaf
- This **overfits** the data, which end up fitting noise in the data
- Consider the Loan Acceptance example, with more records and more variables than the Riding Mower data – the full tree is very complex

Full trees are too complex – they end up fitting noise, overfitting the data (Loan Acceptance shown)



Overfitting produces poor predictive performance – past a certain point in tree complexity, the error rate on new data starts to increase

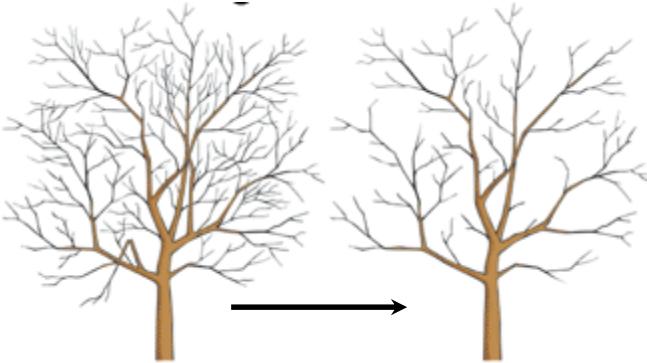


Stopping tree growth

CHAID, older than CART, uses chi-square statistical test to limit tree growth (splitting stops when purity improvement is not statistically significant)

Grid Search: Try different model parameters in search of minimum error tree (can get quite time-consuming)

Pruning



- CART lets tree grow to full extent, then prunes it back
- Idea is to find that point at which the validation error is at a minimum
- Generate successively smaller trees by pruning leaves
- At each pruning stage, multiple trees are possible
- Use *cost complexity* to choose the best tree at that stage (**cp** = complexity parameter). R code:

```
# prune by lower cp
pruned.ct <- prune(cv.ct,
  cp=cv.ct$cptable[which.min(cv.ct$cptable[, "xerror"]),"CP"])
sum(pruned.ct$frame$var == "<leaf>")
rpart.plot(pruned.ct, extra=1, fallen.leaves=FALSE)
```

Which branch to cut at each stage of pruning?

$$CC(T) = Err(T) + \alpha L(T)$$

$CC(T)$ = cost complexity of a tree

$Err(T)$ = proportion of misclassified records

α = penalty factor, also called *complexity factor*,
or **cp**) attached to tree size (set by user)

- Among trees of given size, choose the one with lowest CC
- Do this for each size of tree (stage of pruning)

Set optimal cp by cross-validation

1. Partition the data into training and validation sets.
2. Grow the tree with the training data.
3. Prune it successively, step by step, recording ***cp*** (using the training data) at each step.
4. Note the *cp* that corresponds to the minimum error on the validation data.
5. Repartition the data into training and validation, and repeat the growing, pruning and *cp* recording process.
6. Do this again and again, and average the *cp*'s that reflect minimum error for each tree.
7. Go back to the original data, or future data, and grow a tree, stopping at this optimum *cp* value.

Tabulating tree error as a function of **cp**

```
# argument xval refers to the number of folds to use in rpart's built-in  
# cross-validation procedure  
# argument cp sets the smallest value for the complexity parameter.  
  
cv.ct <- rpart(Personal.Loan ~ ., data=train.df, method="class",  
    cp=0.00001, minsplit=5, xval=5)  
# use printcp() to print the table.  
printcp(cv.ct)
```

Output

	CP	nsplit	rel error	xerror	xstd
1	0.2951389	0	1.000000	1.00000	0.056026
2	0.1354167	2	0.409722	0.43056	0.037858
3	0.0451389	3	0.274306	0.29514	0.031556
4	0.0104167	5	0.184028	0.19097	0.025514
5	0.0086806	10	0.121528	0.17361	0.024347
6	0.0069444	12	0.104167	0.15972	0.023369
7	0.0046296	14	0.090278	0.15625	0.023117
8	0.0034722	17	0.076389	0.17014	0.024106
9	0.0023148	22	0.059028	0.17014	0.024106
10	0.0017361	28	0.045139	0.19097	0.025514
11	0.0000100	30	0.041667	0.19444	0.025740

This tree has lowest cv error (xerror), so use its cp (0.0046296) in deploying model to new data





Tree instability



- If 2 or more variables are of roughly equal importance, which one CART chooses for the first split can depend on the initial partition into training and validation
- A different partition into training/validation could lead to a different initial split
- This can cascade down and produce a very different tree from the first training/validation partition
- Solution is to try many different training/validation splits – “cross validation”

Cross validation

- Do many different partitions (“folds*”) into training and validation, grow & pruned tree for each
- Problem: We end up with lots of different pruned trees. Which one to choose?
- Solution: Don’t choose a tree, choose a tree size:
 - For each iteration, record the cp that corresponds to the minimum validation error
 - Average these cp ’s
 - With future data, grow tree to that optimum cp value

*typically folds are non-overlapping, i.e. data used in one validation fold will not be used in others

Cross validation, “best pruned”

- In the above procedure, we select cp for minimum error tree
- But... simpler is better: slightly smaller tree might do just as well
- Solution: add a cushion to minimum error
 - Calculate standard error of cv estimate – this gives a rough range for chance variation
 - Add standard error to the actual error to allow for chance variation
 - Choose smallest tree within one std. error of minimum error
 - You can then use the corresponding cp to set cp for future data

Regression Trees

Regression Trees for Prediction

- Used with continuous outcome variable
- Procedure similar to classification tree
- Many splits attempted, choose the one that minimizes impurity

Differences from CT

- Prediction is computed as the **average** of numerical target variable in the rectangle (in CT it is majority vote)
- Impurity measured by **sum of squared deviations** from leaf mean
- Performance measured by RMSE (root mean squared error)

Advantages of trees

- Easy to use, understand
- Produce rules that are easy to interpret & implement
- Variable selection & reduction is automatic
- Do not require the assumptions of statistical models
- Can work without extensive handling of missing data

Disadvantage of single trees: instability and poor predictive performance

Random Forests and Boosted Trees

- Examples of “ensemble” methods, “Wisdom of the Crowd” (Chap 13)
- Predictions from many trees are combined
- Very good predictive performance, better than single trees (often the top choice for predictive modeling)
- Cost: loss of rules you can explain implement (since you are dealing with many trees, not a single tree)
 - However, RF does produce “variable importance scores,” (using information about how predictors reduce Gini scores over all the trees in the forest)

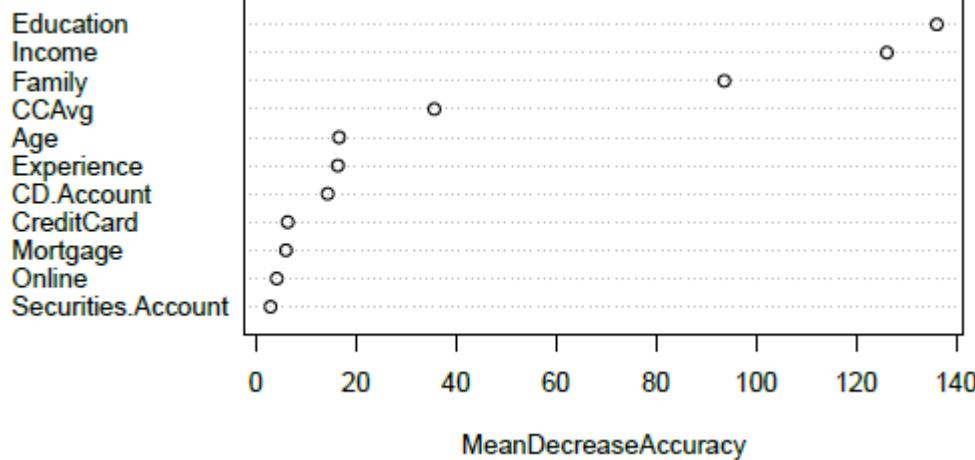
Random Forests (library randomForest)

1. Draw multiple bootstrap resamples of cases from the data
2. For each resample, use a random subset of predictors and produce a tree
3. Combine the predictions/classifications from all the trees (the “forest”)
 - Voting for classification
 - Averaging for prediction

Random Forest: Code for Loan Acceptance Example

```
library(randomForest)
## random forest
rf <- randomForest(Personal.Loan ~ ., data=train.df, ntree=500,
mtry=4, nodesize=5, importance=TRUE)

## variable importance plot
varImpPlot(rf, type=1)
```



Variable Importance Plot
shows how much removing a predictor reduces accuracy, for each predictor (summing Gini decrease across all trees in forest)

Boosted Trees (libraries xgboost and caret)

Boosting, like RF, is an ensemble method – but uses an iterative approach in which each successive tree focuses its attention on the misclassified trees from the prior tree.

1. Fit a single tree
2. Draw a bootstrap sample of records with higher selection probability for misclassified records
3. Fit a new tree to the bootstrap sample
4. Repeat steps 2 & 3 multiple times
5. Use weighted voting (classification) or averaging (prediction) with heavier weights for later trees

Boosted Trees: Code for Loan Acceptance Example (and comparison to single tree and RF)

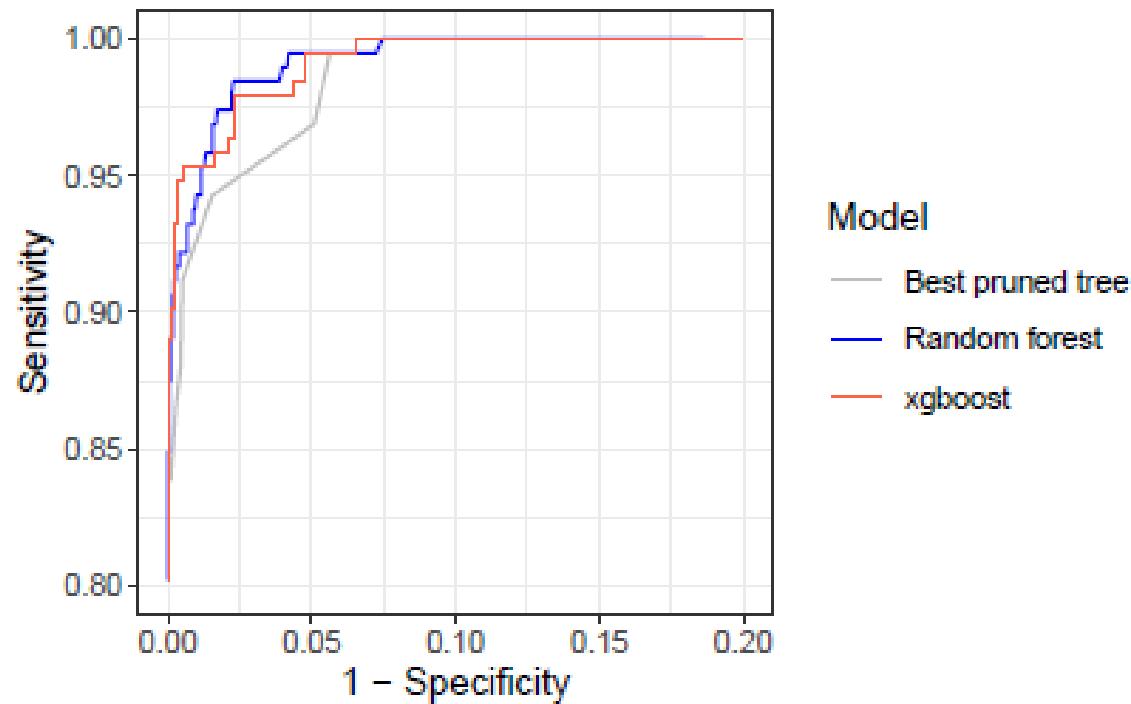
libraries `caret` and `xgboost`

```
xgb <- train(Personal.Loan ~ ., data=train.df, method="xgbTree", verbosity=0)

# compare ROC curves for single tree, random forest, and boosted tree models
library(ROCR)
rocCurveData <- function(model, data) {
  prob <- predict(model, data, type="prob")[, "Yes"]
  predob <- prediction(prob, data$Personal.Loan)
  perf <- performance(predob, "tpr", "fpr")
  return (data.frame(tpr=perf@x.values[[1]], fpr=perf@y.values[[1]]))
}

performance.df <- rbind(
  cbind(rocCurveData(best.ct, holdout.df), model="Best pruned tree"),
  cbind(rocCurveData(rf, holdout.df), model="Random forest"),
  cbind(rocCurveData(xgb, holdout.df), model="xgboost"))
colors <- c("Best pruned tree"="grey", "Random forest"="blue",
"xgboost"="tomato")
ggplot(performance.df, aes(x=tpr, y=fpr, color=model)) +
  geom_line() +
  scale_color_manual(values=colors) +
  geom_segment(aes(x=0, y=0, xend=1, yend=1), color="grey", linetype="dashed") +
  labs(x="1 - Specificity", y="Sensitivity", color="Model")
```

Personal Loan Example: Comparing Single Tree, Random Forest, and Boosted Trees (zooming in to high specificity values)



Summary

- Classification and Regression Trees are an easily understandable and transparent method for predicting or classifying new records
- A single tree is a graphical representation of a set of rules
- Tree growth must be stopped to avoid overfitting of the training data – cross-validation helps you pick the right cp level to stop tree growth
- Ensembles (random forests, boosting) improve predictive performance, but you lose interpretability and the rules embodied in a single tree