Classification and Regression Trees
Part 2: Evaluating the Performance of a Classification Tree and Avoiding Overfitting

Aram Balagyozyan

Department of Operations and Information Management
Kania School of Management
The
University of Scranton

February 20, 2020

1. Evaluating the Performance of a Classification Tree
2. Avoiding Overfitting

Evaluating the Performance of a Classification Tree

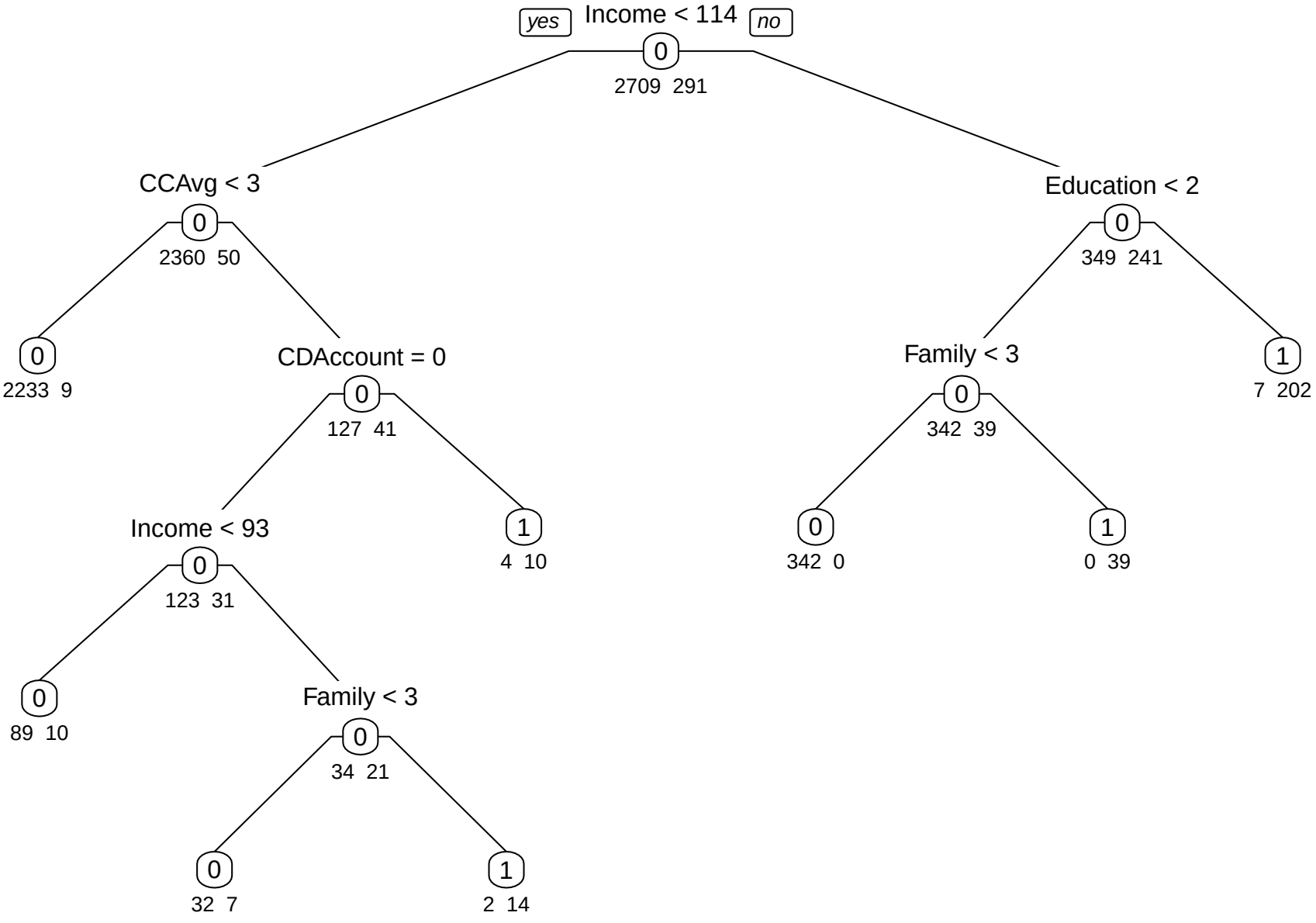
- As we have seen in previous modules, the modeling job is not completed by fitting a model to training data; we need out-of-sample data to assess and tune the model. This is particularly true with classification and regression trees, for two reasons:
  1. Tree structure can be quite unstable, shifting substantially depending on the sample chosen.
  2. Continuous recursive partitioning will eventually lead to a perfectly fit model. However, such model will certainty be overfit.
- A prudent starting approach would be to split the data into a training and validation sets and see how increasingly more partitioned trees would perform on the validation set.
- Do demonstrate this idea, let's partition our starter case into a training set (60% or 3,000 records) and validation set (40% or 2,000 records).

Evaluating the Performance of a Classification Tree

```
# partition
set.seed(1) # for reproducibility
train.index <- sample(c(1:dim(bank.df)[1]),
                      dim(bank.df)[1]*0.6)
train.df <- bank.df[train.index, ]
valid.df <- bank.df[-train.index, ]

# classification tree
default.ct <- rpart(PersonalLoan ~ .,
                    data = train.df,
                    method = "class")
# plot tree
prp(default.ct, type = 1, extra = 1,
       under = TRUE, split.font = 1,
                     varlen = -10)
```

Evaluating the Performance of a Classification Tree



Evaluating the Performance of a Classification Tree

- After randomly partitioning the data into training (3000 records) and validation (2000 records), we use the training data to construct a tree.
- A tree with 7 splits is shown on the previous slide (this is the default tree produced by rpart() for this data).
- Of the eight terminal nodes, five lead to classification of "did not accept" and three lead to classification of "accept."
- Note that when compared to the corresponding plot in the textbook, ours is a bit different. The difference seems to be engendered by different versions of **rpart** used.

Confusion Matrix and Accuracy of the Default Tree: Training Set

- We can easily produce the confusion matrix and accuracy of the "default" **rpart** tree produced above using the training set:

```
library(forecast) #for predict
library (caret) # for confusionMatrix
# classify records in the training data.
# set argument type = "class" in predict() to
# generate predicted class membership rather
# than propensities.
  default.ct.pred.train <- predict(
  default.ct,train.df , type = "class")
# generate confusion matrix for training data
confMatDefTrain <- confusionMatrix(
              default.ct.pred.train,
              factor(train.df$PersonalLoan))
```

Confusion Matrix and Accuracy of the Default Tree: Training Set

```
#display confusion matrix: default tree, train set
confMatDefTrain$table
```

```
##            Reference
## Prediction    0    1
##          0 2696   26
##          1   13  265
```

```
#display accuracy: default tree, training set
confMatDefTrain$overall[1]
```

```
## Accuracy
##    0.987
```

Confusion Matrix and Accuracy of the Default Tree: Validation Set

- We can also produce the confusion matrix and accuracy of the default **rpart** tree using the validation set.

```
# classify records in the validation data.
  default.ct.pred.valid <- predict(
  default.ct, valid.df , type = "class")
# generate confusion matrix for validation data
confMatDefValid <- confusionMatrix(
              default.ct.pred.valid,
              factor(valid.df$PersonalLoan))
```

Confusion Matrix and Accuracy of the Default Tree: Validation Set

```
#display confusion matrix: default tree, valid set
confMatDefValid$table
```

```
##            Reference
## Prediction    0    1
##          0 1792   18
##          1   19  171
```

```
#display accuracy:default tree, valid set
confMatDefValid$overall[1]
```

```
## Accuracy
##   0.9815
```

Confusion Matrix and Accuracy of the Default Tree: Validation Set

- As you can see, the accuracy of the validation set is only slightly lower than on the accuracy of the training set. Thus there is little evidence of an overfit tree.
- We next produce a tree that is deeper that the default tree. As you will see, in terms of accuracy it will perform better on the training set and worse on the validation set, a clear sign of overfitting.

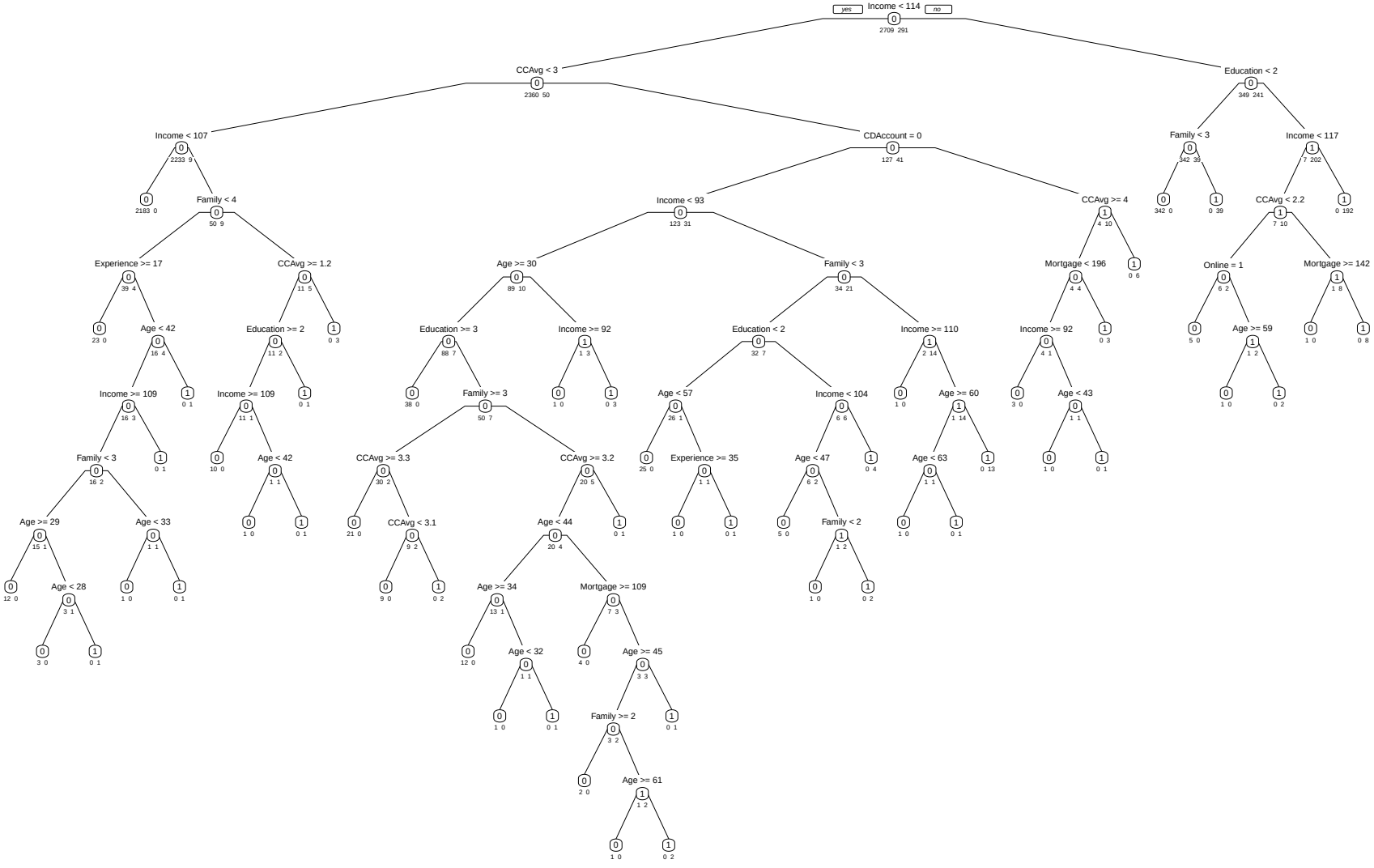
Confusion Matrix and Accuracy of a "Deep" Tree: Training Set

- We use the `minsplit` option of the `rpart()` function to produce a tree that is deeper than the default tree above.

```
deeper.ct <- rpart(PersonalLoan ~ .,
                   data = train.df,
                   method = "class",
                   cp = 0,
                   minsplit = 1)
# Plot the deeper tree
prp(deeper.ct, type = 1, extra = 1, under = TRUE,
split.font = 1, varlen = -10)
```

- Also, cp is the complexity parameter. For the `rpart()` routine, any split that does not decrease the overall lack of fit by a factor of cp is not attempted. Thus, the smaller this parameter the deeper is the tree.

Confusion Matrix and Accuracy of a "Deep" Tree: Training Set



- The resulting deep tree has 53 terminal nodes.

Confusion Matrix and Accuracy of a "Deep" Tree: Training Set

- The following code generate predictions of the training set using the deep tree:

```
  deeper.ct.pred.train <- predict(
  deeper.ct,train.df , type = "class")
# generate confusion matrix for training data
confMatDeepTrain <- confusionMatrix(
              deeper.ct.pred.train,
              factor(train.df$PersonalLoan))
```

Confusion Matrix and Accuracy of a "Deep" Tree: Training Set

```
#display confusion matrix: default tree, valid set
confMatDeepTrain$table
```

```
##           Reference
## Prediction    0    1
##          0 2709    0
##          1    0  291
```

```
#display accuracy:default tree, valid set
confMatDeepTrain$overall[1]
```

```
## Accuracy
##        1
```

- The accuracy of the deep-tree model on the training set is 1 (the deep-tree model correctly predicts the loan-offer response of every single customer in the training set!)
- But how accurately can the deep-tree model predict records in the validation set?

Confusion Matrix and Accuracy of a "Deep" Tree: Validation Set

- The code below produces the predictions of the deep-tree model on the validation set. The code also predicts the confusion matrix and accuracy.

```
# classify records in the validation data.
  deeper.ct.pred.valid <- predict(
  deeper.ct, valid.df , type = "class")
# generate confusion matrix for validation data
confMatDeepValid <- confusionMatrix(
              deeper.ct.pred.valid,
              factor(valid.df$PersonalLoan))
```

Confusion Matrix and Accuracy of a "Deep" Tree: Validation Set

```
#display confusion matrix: default tree, valid set
confMatDeepValid$table
```

```
##            Reference
## Prediction    0    1
##          0 1788   19
##          1   23  170
```

```
#display accuracy:default tree, valid set
confMatDeepValid$overall[1]
```

```
## Accuracy
##    0.979
```

Accuracy of the Default and Deep Trees: Summary

- Just to summarize, the accuracies of the two models are presented in the table below

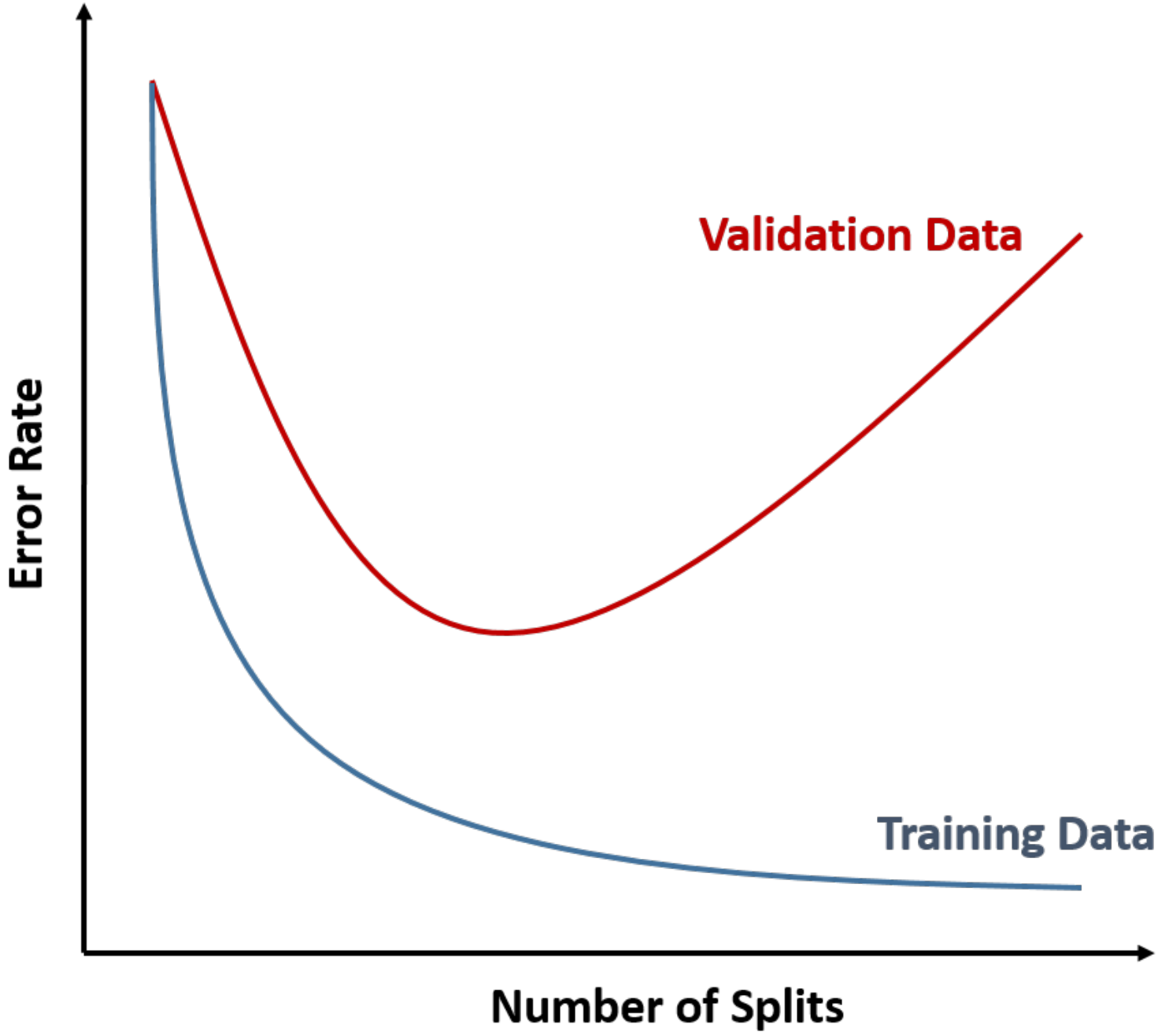
Model	Training Set	Validation
Default Tree	0.987	0.98
Deeper Tree	1	0.979

- For the default **rpart** tree, the accuracy measure is almost the same across the two data sets.
- The deeper tree is clearly overfit as it is very accurate on the training set but not nearly as fit on the validation set.

Avoiding overfitting

- It must be obvious that by continually increasing the number of splits, we can reduce the overall error rate of the training data down to zero.
- However, for the validations set, the overall error is expected to decrease until the point where the tree fully models the relationship between class and the predictors. After that, the tree starts to model the noise in the training set, and we expect the overall error for the validation set to start increasing.
- The figure on the following slide demonstrates the idea

Avoiding overfitting



Stopping Tree Growth: Conditional Inference Trees

- One approach on determining the optimal tree size is to start with a small tree and grow it to the optimal size before it starts overfitting the data.
- The problem is that in practice, it is not simple to determine what is a good stopping point using such rules.

Pruning the Tree

- An alternative popular solution that has proven to be more successful than stopping tree growth is pruning the full-grown tree.
- The idea behind pruning is to recognize that a very large tree is likely to overfit the training data, and that the weakest branches, which hardly reduce the error rate, should be removed.
- In our `deeper.ct` tree, the last leaves resulted in rectangles with as few as 2 records. Intuitively, these last splits are likely just capturing noise in the training set rather than reflecting patterns that would occur in future data, such as the validation data.

Pruning the Tree

- Pruning consists of successively selecting a decision node and re-designating it as a terminal node [lopping off the branches extending beyond that decision node (its sub-tree) and thereby reducing the size of the tree].
- The pruning process trades off misclassification error in the validation dataset against the number of decision nodes in the pruned tree to arrive at a tree that captures the patterns—but not the noise—in the training data.

Pruning the Tree

- In R's `rpart()`, we can control the depth of the tree with the *complexity parameter* (cp) which imposes a penalty to the tree for having two many splits. The default value is 0.01. The higher the cp, the smaller the tree.
- A too small value of cp leads to overfitting and a too large cp value will result in a too small of a tree. Both cases decrease the predictive performance of the model.

Pruning the Tree

- There are different approaches to pruning fully grown trees. C4.5 and CART algorithms are two of the most popular ones. The **rpart** library uses the CART approach.
- The C4.5 approach uses the *training data* for both, growing and pruning the tree.
- The CART approach is more robust as it uses the *validation data* to prune back the tree that has deliberately overgrown using the the training data.

Cross-Validation

- In principle, we could follow the CART procedure (grow a tree using the training set and prune it using the validation set) and be done with it. However, this procedure solves the overfitting problem but it does not solve the problem with instability.
- The CART algorithm may be unstable in choosing one or another variable for the top level splits, and this effect then cascades down and produces highly variable rule set.
- The solution is to avoid using just one partition of the data into training and validation. Rather we run multiple cycles, each consisting of (1) partitioning of the data into training and validation, (2) growing a deep tree using the training set, (3) pruning the tree using the validation set so that the error rate is minimized and (4) recording the cp value of the final tree.
- After we repeat the above process several times, we average the cp values obtained on all cycles.
- Finally, we go to the original data and grow a tree, stopping at optimum CP value.

Cross-Validation: Summary

- Below is the summary of the algorithm.
  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. Re-partition 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.
- The number of times we re-partition the data is also referred to as "folds."
- Typically, in the cross-validation process, folds are non-overlapping.

Cross-Validation in Practice

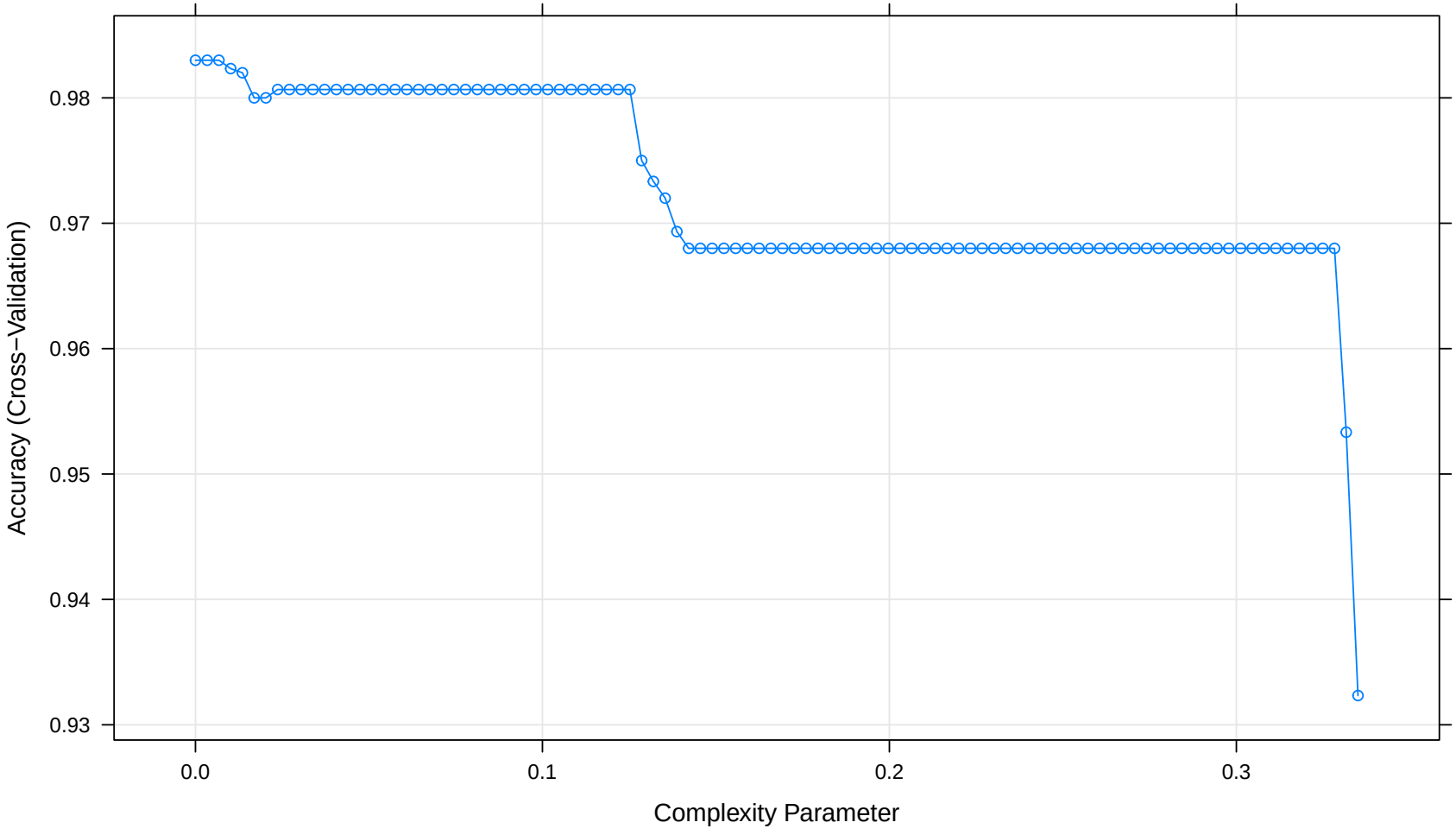
- Pruning can be easily performed in the **caret** package workflow, which invokes the rpart method for automatically testing different possible values of cp, then choose the optimal cp that maximize the cross-validation ("cv") accuracy, and fit the final best CART model that explains the best our data.
- You can use the following arguments in the function `train()` [from **caret** package]:
  1. `trControl`, to set up 10-fold cross validation
  2. `tuneLength`, to specify the number of possible cp values to evaluate. Default value is 3, here we'll use 100.

Cross-Validation in Practice

- Below is the code:

```
library(caret)
# convert PersonalLoan to a factor (req. by caret)
train.df$PersonalLoan<-factor(train.df$PersonalLoan)
# Fit the model on the training set
set.seed(123)
model1 <- train(
  PersonalLoan ~., data = train.df,
  method = "rpart",
  trControl = trainControl("cv", number = 10),
  tuneLength = 100
  )
# Plot model accuracy vs different values of
# cp (complexity parameter)
plot(model1)
```

Cross-Validation in Practice




```
model1$bestTune
```

```
##           cp
## 3 0.006768718
```

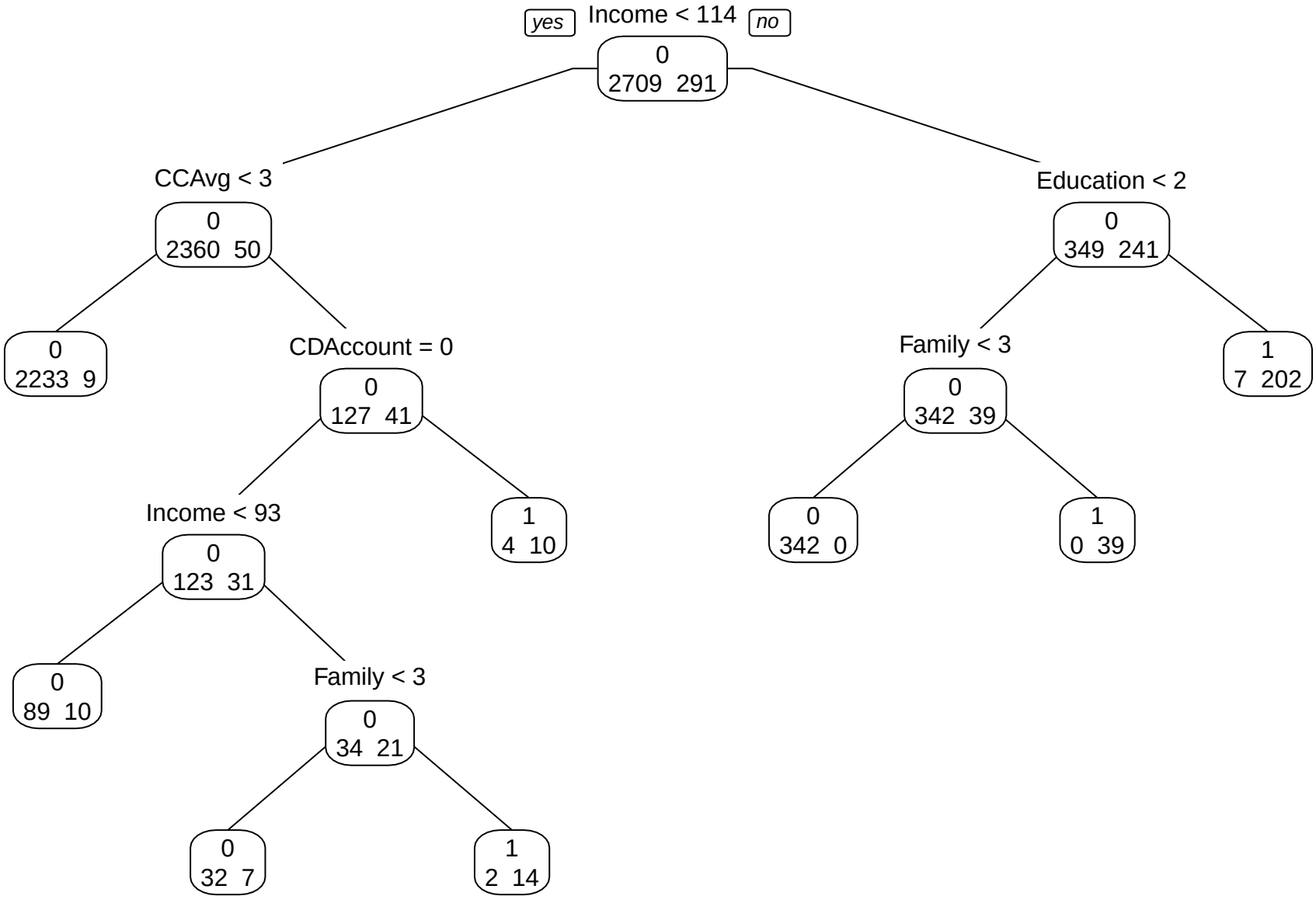
Cross-Validation in Practice

- Now, knowing that the value of cp = 0.006768718 provides the highest accuracy on the training set, we can proceed and prune and plot the best tree.

```
# prune the tree using prune.rpart()
pruned.ct<-prune.rpart(default.ct,
                cp=model1$bestTune[1,1])
prp(pruned.ct,
type=1,
extra=1,
split.font = 1,
varlen = 10)
```

- For the `prune.rpart()` routine, the `cp` parameter controls to what complexity the tree should be cut.
- Note also that the optimal tree `pruned.ct` could have been produced by executing command `pruned.ct<-model1$finalModel`

Cross-Validation in Practice



Cross-Validation in Practice

- Note that the best tree above (chosen by the manual cross-validation procedure) happened to be the same "default" tree chosen by the `rpart()` function (`default.ct`).
- `rpart()` has an internal cross-validation procedure that happened to the the same result as what we did manually above. The `xval` argument of the `rpart()` function ($= 10$ by default) controls the number of cross-validation folds.
- However, I found that more often than not, the `train()` routine of the **caret** package is more robust than the default **rpart** pruning engine.

Classification Rules from Trees

- As described earlier, classification trees provide easily understandable classification rules.
- There is a nice `rpart.rules()` method that allows you to print the rules of any **rpart** tree.

```
rpart.rules(pruned.ct)
```

PersonalLoan																				
0.00	when	Income	>=		114								&	Family	<	3	&	Education	<	2
0.00	when	Income	<	114		&	CCAvg	<	3											
0.10	when	Income	<	93		&	CCAvg	>=	3	&	CDAccount	is	0							
0.18	when	Income	is	93	to	114	&	CCAvg	>=	3	&	CDAccount	is	0	&	Family	<	3		
0.71	when	Income	<	114		&	CCAvg	>=	3	&	CDAccount	is	1							
0.88	when	Income	is	93	to	114	&	CCAvg	>=	3	&	CDAccount	is	0	&	Family	>=	3		
0.97	when	Income	>=		114												&	Education	>=	2
1.00	when	Income	>=		114								&	Family	>=	3	&	Education	<	2

- The first column in the above table represents the fraction of customers in the terminal node of the `pruned.ct` tree (described by the rule in the corresponding row) that accepted the loan offer.
- For instance, the proportion of customers with Income $>= 114$, Family $>= 3$, & Education < 2 (last row) who accepted the loan offer is 1 (100%). We see from the tree itself that there were 39 customers in this particular rectangle.

Classification Trees for More Than Two Classes

- You shouldn't be surprise that classification trees can be used with an outcome variable that has more than two classes.
- In terms of measuring impurity, the two measures presented earlier (the Gini impurity index and the entropy measure) were defined for m classes and hence can be used for any number of classes.
- The tree itself would have the same structure, except that its terminal nodes would take one of the m–class labels.