Tree-Based Models

Kayla Friend

Last compiled: May 06, 2021

Contents

Chapter 1

Prerequisites

This material is from the DataCamp course Tree-Based Models by Erin L. and Gabriela de Queiroz. Before using this material, the reader should have completed and be comfortable with the material in the DataCamp module Tree-Based Models.

Chapter 2

Classification Trees

Welcome to the Course

2.1 Build a Classification Tree

A classification tree is a decision tree that performs a classification (vs regression) task.## Build a Classification Tree

Let's get started and build our first classification tree.

You will train a decision tree model to understand which loan applications are at higher risk of default using a subset of the German Credit Dataset. The response variable, `default`, indicates whether the loan went into a default or not, which means this is a binary classification problem (there are just two classes).

You will use the `rpart` package to fit the decision tree and the `rpart.plot` package to visualize the tree.

Exercise

The data frame `creditsub` is in the workspace. This data frame is a subset of the original German Credit Dataset, which we will use to train our first classification tree model.

- Take a look at the data using the `str()` function.

```
str(creditsub)
```

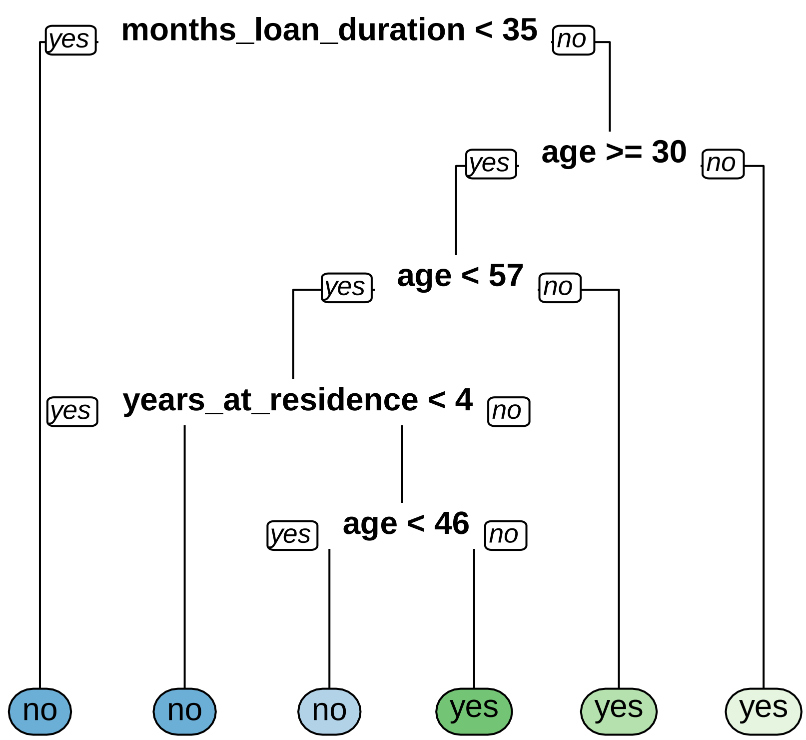
```
tibble[,5] [1,000 x 5] (S3: tbl_df/tbl/data.frame)
 $ months_loan_duration: num [1:1000] 6 48 12 42 24 36 24 36 12 30 ...
 $ percent_of_income   : num [1:1000] 4 2 2 2 3 2 3 2 2 4 ...
 $ years_at_residence  : num [1:1000] 4 2 3 4 4 4 4 2 4 2 ...
 $ age                 : num [1:1000] 67 22 49 45 53 35 53 35 61 28 ...
 $ default             : chr [1:1000] "no" "yes" "no" "no" ...
```

- In R, formulas are used to model the response as a function of some set of predictors, so the formula here is `default ~ .`, which means use all columns (except the response column) as predictors. Fit the classification decision tree using the `rpart()` function from the `rpart` package. In the `rpart()` function, note that you'll also have to provide the training data frame.

```
credit_model <- rpart(formula = default ~ .,
                      data = creditsub,
                      method = "class")
```

- Using the model object that you create, plot the decision tree model using the `rpart.plot()` function from the `rpart.plot` package.

```
rpart.plot(x = credit_model, yesno = 2, type = 0, extra = 0)
```



2.2 Introduction to Classification Trees

2.2.1 Advantages of Tree-Based Methods

What are some advantages of using tree-based methods over other supervised learning methods?

- Model interpretability (easy to understand why a prediction is made).
- Model performance (trees have superior performance compared to other machine learning algorithms).
- No pre-processing (e.g. normalization) of the data is required.
- **1 and 3 are true.**

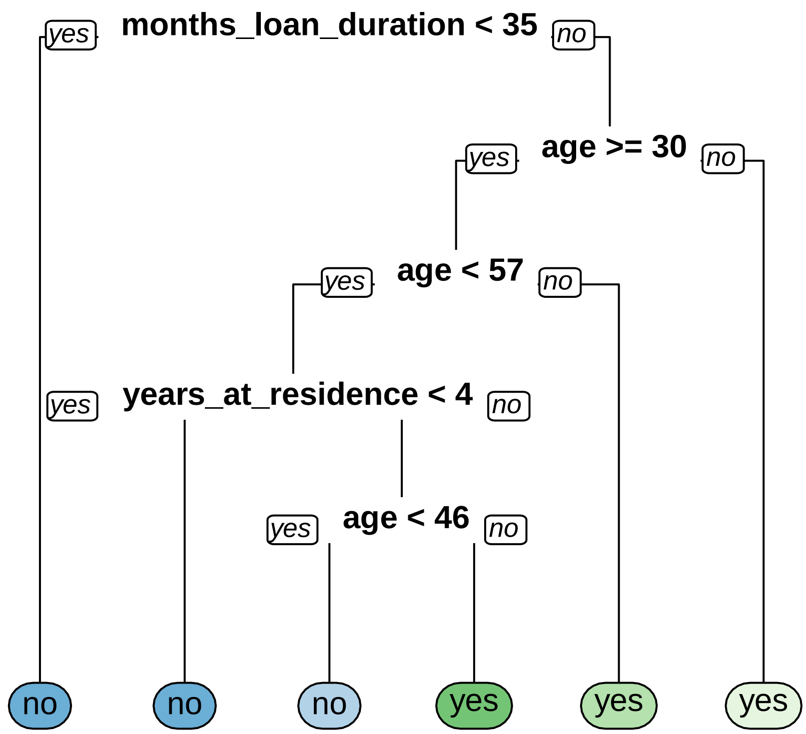
2.2.2 Prediction with a Classification Tree

Let's use the decision tree that you trained in the first exercise. The tree predicts whether a loan applicant will default on their loan (or not).

Assume we have a loan applicant who:

is applying for a 20-month loan is requesting a loan amount that is 2% of their income is 25 years old After following the correct path down the tree for this individual's set of data, you will end up in a "Yes" or "No" bucket (in tree terminology, we'd call this a "leaf") which represents the predicted class. Ending up in a "Yes" leaf means that the model predicts that this individual will default on their loan, where as a "No" prediction means that they will not default on their loan.

Starting with the top node of the tree, you must evaluate a query about a particular attribute of your data point (e.g. is `months_loan_duration < 44`?). If the answer is yes, then you go to the left at the split; if the answer is no, then you will go right. At the next node you repeat the process until you end up in a leaf node, at which point you'll have a predicted class for your data point.



According to the model this person will default on their loan.

2.3 Overview of the Modelling Process

2.3.1 Train/Test Split

For this exercise, you'll randomly split the German Credit Dataset into two pieces: a training set (80%) called `credit_train` and a test set (20%) that we will call `credit_test`. We'll use these two sets throughout the chapter. The `credit` data frame is loaded into the workspace.

Exercise

- Define `n`, the number of rows in the `credit` data frame.

```
# Total number of rows in the credit data frame
n <- nrow(credit)
```

- Define `n_train` to be ~80% of `n`.

```
# Number of rows for the training set (80% of the dataset)
n_train <- round(.8 * n)
```

- Set a seed (for reproducibility) and then sample `n_train` rows to define the set of training set indices.

```
# Create a vector of indices which is an 80% random sample
set.seed(123)
train_indices <- sample(1:n, n_train)
```

- Using row indices, subset the credit data frame to create two new datasets: `credit_train` and `credit_test`

```
# Subset the credit data frame to training indices only
credit_train <- credit[train_indices, ]

# Exclude the training indices to create the test set
credit_test <- credit[-train_indices, ]
```

2.3.2 Train a Classification Tree

In this exercise, you will train a model on the newly created training set and print the model object to get a sense of the results.

- Train a classification tree using the `credit_train` data frame.

```
# Train the model (to predict 'default')
credit_model <- rpart(formula = default ~ .,
                      data = credit_train,
                      method = "class")
```

- Look at the model output by printing the model object.

```
# Look at the model output
print(credit_model)
```

```
n= 800

node), split, n, loss, yval, (yprob)
      * denotes terminal node

  1) root 800 230 no (0.7125000 0.2875000)
```

```
   2) checking_balance=> 200 DM,unknown 365  48 no (0.8684932 0.1315068) *
   3) checking_balance=< 0 DM,1 - 200 DM 435 182 no (0.5816092 0.4183908)
     6) months_loan_duration< 22.5 259  85 no (0.6718147 0.3281853)
      12) credit_history=critical,good,poor 235  68 no (0.7106383 0.2893617)
        24) months_loan_duration< 11.5 70  11 no (0.8428571 0.1571429) *
        25) months_loan_duration>=11.5 165  57 no (0.6545455 0.3454545)
          50) amount>=1282 112  30 no (0.7321429 0.2678571) *
          51) amount< 1282 53  26 yes (0.4905660 0.5094340)
           102) purpose=business,education,furniture/appliances 34  12 no (0.6470588 0
           103) purpose=car,renovations 19   4 yes (0.2105263 0.7894737) *
      13) credit_history=perfect,very good 24   7 yes (0.2916667 0.7083333) *
     7) months_loan_duration>=22.5 176  79 yes (0.4488636 0.5511364)
      14) savings_balance=> 1000 DM,unknown 29   7 no (0.7586207 0.2413793) *
      15) savings_balance=< 100 DM,100 - 500 DM,500 - 1000 DM 147  57 yes (0.3877551 0
        30) months_loan_duration< 47.5 119  54 yes (0.4537815 0.5462185)
          60) amount>=2313.5 93  45 no (0.5161290 0.4838710)
           120) amount< 3026 19   5 no (0.7368421 0.2631579) *
           121) amount>=3026 74  34 yes (0.4594595 0.5405405)
             242) percent_of_income< 2.5 38  15 no (0.6052632 0.3947368)
               484) purpose=business,car,education 23   6 no (0.7391304 0.2608696) *
               485) purpose=car0,furniture/appliances,renovations 15   6 yes (0.40000
             243) percent_of_income>=2.5 36  11 yes (0.3055556 0.6944444) *
          61) amount< 2313.5 26   6 yes (0.2307692 0.7692308) *
        31) months_loan_duration>=47.5 28   3 yes (0.1071429 0.8928571) *
```

2.4 Evaluating Classification Model Performance

2.4.1 Compute confusion matrix

As discussed in the previous video, there are a number of different metrics by which you can measure the performance of a classification model. In this exercise, we will evaluate the performance of the model using test set classification error. A confusion matrix is a convenient way to examine the per-class error rates for all classes at once.

The `confusionMatrix()` function from the caret package prints both the confusion matrix and a number of other useful classification metrics such as "Accuracy" (fraction of correctly classified instances).

Exercise

The caret package has been loaded for you.

- Generate class predictions for the `credit_test` data frame using the `credit_model` object.

```
# Generate predicted classes using the model object
class_prediction <- predict(object = credit_model,
                            newdata = credit_test,
                            type = "class")
class_prediction
```

```
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20
 no  no  no  no yes  no  no  no  no yes  no  no  no yes  no  no  no  no  no  no
 21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40
 no  no  no  no  no  no  no  no  no yes  no  no  no  no  no  no yes yes  no yes
 41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60
 no  no  no  no  no  no  no  no  no yes  no  no  no yes yes  no yes  no yes  no
 61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
 no yes  no  no yes yes  no  no  no  no  no yes yes  no  no  no  no yes  no yes
 81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
 no  no  no  no yes  no  no yes  no  no  no  no  no yes  no  no  no  no  no  no
101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120
 no yes  no  no yes  no  no  no  no  no  no  no  no  no  no  no  no  no  no  no
121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140
yes yes  no  no  no yes  no  no  no  no  no  no  no  no yes  no yes  no  no yes
141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160
 no  no  no yes  no  no  no  no  no yes  no  no  no  no  no  no  no  no  no  no
161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180
 no  no  no  no  no  no  no  no  no  no  no  no  no  no  no  no  no  no  no  no
181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200
 no  no yes yes yes  no yes  no  no  no  no  no yes  no  no  no yes  no  no yes
Levels: no yes
```

- Using the `caret::confusionMatrix()` function, compute the confusion matrix for the test set.

```
# Calculate the confusion matrix for the test set
caret::confusionMatrix(data = class_prediction,
        reference = factor(credit_test$default))
```

```
Confusion Matrix and Statistics

          Reference
```

```
Prediction  no yes
       no  117  44
       yes  13  26

               Accuracy : 0.715
                 95% CI : (0.6471, 0.7764)
    No Information Rate : 0.65
    P-Value [Acc > NIR] : 0.03046

                  Kappa : 0.3023

 Mcnemar's Test P-Value : 7.08e-05

            Sensitivity : 0.9000
            Specificity : 0.3714
         Pos Pred Value : 0.7267
         Neg Pred Value : 0.6667
             Prevalence : 0.6500
         Detection Rate : 0.5850
   Detection Prevalence : 0.8050
      Balanced Accuracy : 0.6357

       'Positive' Class : no
```

2.5 Use of Splitting Criterion in Trees

2.5.1 Compare models with a different splitting criterion

Train two models that use a different splitting criterion and use the validation set to choose a "best" model from this group. To do this you'll use the `parms` argument of the `rpart()` function. This argument takes a named list that contains values of different parameters you can use to change how the model is trained. Set the parameter `split` to control the splitting criterion.

Exercise

The datasets `credit_test` and `credit_train` have already been loaded for you.

- Train a model, splitting the tree based on gini index.

```
# Train a gini-based model
credit_model1 <- rpart(formula = default ~ .,
                       data = credit_train,
                       method = "class",
                       parms = list(split = "gini"))
```

- Train a model, splitting the tree based on information index.

```
# Train an information-based model
credit_model2 <- rpart(formula = default ~ .,
                       data = credit_train,
                       method = "class",
                       parms = list(split = "information"))
```

- Generate predictions on the validation set using both models.

```
# Generate predictions on the validation set using the gini model
pred1 <- predict(object = credit_model1,
                 newdata = credit_test,
                 type = "class")

# Generate predictions on the validation set using the information model
pred2 <- predict(object = credit_model2,
                 newdata = credit_test,
                 type = "class")
```

- Classification error is the fraction of incorrectly classified instances. Compute and compare the test set classification error of the two models by using the `ce()` function.

```
# Compare classification error
ce(actual = credit_test$default,
     predicted = pred1)
```

```
[1] 0.285
```

```
ce(actual = credit_test$default,
     predicted = pred2)
```

```
[1] 0.285
```

Chapter 3

Regression Trees

3.1 Introduction to Regression Trees

3.1.1 Classification vs. regression

What is the difference between classification and regression?

- In classification, the response represents a category (e.g. "apples", "oranges", "bananas").
- In regression, the response represents a numeric value (e.g. price of a house).
- **All of the above.**
- None of the above.

3.2 Split the data

The goal of this exercise is to predict a student's final Mathematics grade based on the following variables: `sex`, `age`, `address`, `studytime` (weekly study time), `schoolsup` (extra educational support), `famsup` (family educational support), `paid` (extra paid classes within the course subject) and `absences`.

The response is `final_grade` (numeric: from 0 to 20, output target).

After initial exploration, split the data into training, validation, and test sets. In this chapter, we will introduce the idea of a validation set, which can be used to select a "best" model from a set of competing models.

In Chapter 1, we demonstrated a simple way to split the data into two pieces using the `sample()` function. In this exercise, we will take a slightly different approach to splitting the data that allows us to split the data into more than two parts (here, we want three: train, validation, test). We still use the `sample()` function, but instead of sampling the indices themselves, we will assign each row to either the training, validation or test sets according to a probability distribution.

Exercise

These examples will use a subset of the Student Performance Dataset from UCI ML Dataset Repository.

The dataset `grade` is already in your workspace.

- Take a look at the data using the `str()` function.

```
# Look at the data
str(grade)
```

```
spec_tbl_df[,8] [395 x 8] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ final_grade: num [1:395] 3 3 5 7.5 5 7.5 5.5 3 9.5 7.5 ...
 $ age        : num [1:395] 18 17 15 15 16 16 16 17 15 15 ...
 $ address    : chr [1:395] "U" "U" "U" "U" ...
 $ studytime  : num [1:395] 2 2 2 3 2 2 2 2 2 2 ...
 $ schoolsup  : chr [1:395] "yes" "no" "yes" "no" ...
 $ famsup     : chr [1:395] "no" "yes" "no" "yes" ...
 $ paid       : chr [1:395] "no" "no" "yes" "yes" ...
 $ absences   : num [1:395] 6 4 10 2 4 10 0 6 0 0 ...
 - attr(*, "spec")=
  .. cols(
  ..   final_grade = col_double(),
  ..   age = col_double(),
  ..   address = col_character(),
  ..   studytime = col_double(),
  ..   schoolsup = col_character(),
  ..   famsup = col_character(),
  ..   paid = col_character(),
  ..   absences = col_double()
  .. )
```

- Set a seed (for reproducibility) and then sample n_train rows to define the set of training set indices.
  - Draw a sample of size nrow(grade) from the number 1 to 3 (with replacement). You want approximately 70% of the sample to be 1 and the remaining 30% to be equally split between 2 and 3.

```
# Set seed and create assignment
set.seed(1)
assignment <- sample(1:3, size = nrow(grade), prob = c(0.7, 0.15, 0.15), replace = TRUE)
```

- Subset `grade` using the sample you just drew so that indices with the value 1 are in `grade_train`, indices with the value 2 are in `grade_valid`, and indices with 3 are in `grade_test`.

```
# Create a train, validation and tests from the original data frame
grade_train <- grade[assignment == 1, ]    # subset grade to training indices only
grade_valid <- grade[assignment == 2, ]  # subset grade to validation indices only
grade_test <- grade[assignment == 3, ]   # subset grade to test indices only
```

3.3 Train a regression tree model

In this exercise, we will use the `grade_train` dataset to fit a regression tree using `rpart()` and visualize it using `rpart.plot()`. A regression tree plot looks identical to a classification tree plot, with the exception that there will be numeric values in the leaf nodes instead of predicted classes.

This is very similar to what we did previously in Chapter 1. When fitting a classification tree, we use `method = "class"`, however, when fitting a regression tree, we need to set `method = "anova"`. By default, the `rpart()` function will make an intelligent guess as to what the method value should be based on the data type of your response column, but it's recommened that you explictly set the method for reproducibility reasons (since the auto-guesser may change in the future).

Exercise

The `grade_train` training set is loaded into the workspace.

- Using the `grade_train` dataframe and the given formula, train a regresion tree.

```r
grade_model <- rpart(formula = final_grade ~ .,
                     data = grade_train,
                     method = "anova")
```

- Look at the model output by printing the model object.

```r
# Look at the model output
print(grade_model)
```

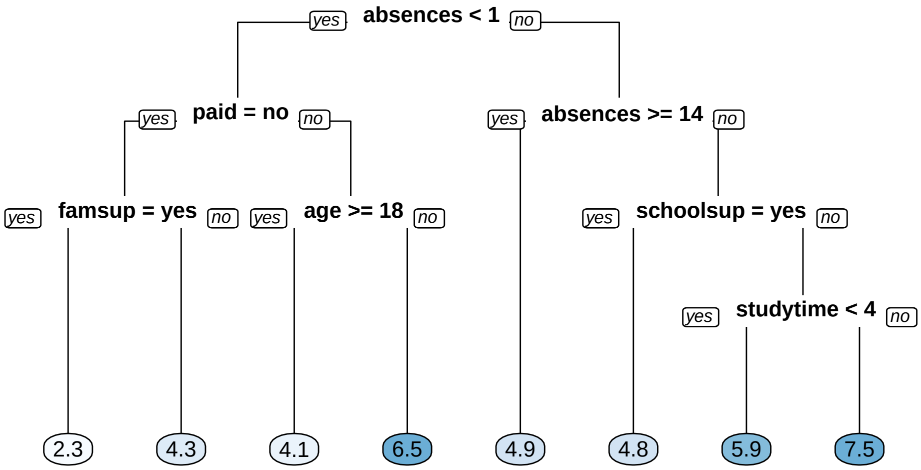
```
n= 282

node), split, n, deviance, yval
      * denotes terminal node

 1) root 282 1519.49700 5.271277
   2) absences< 0.5 82  884.18600 4.323171
     4) paid=no 50  565.50500 3.430000
       8) famsup=yes 22  226.36360 2.272727 *
       9) famsup=no 28  286.52680 4.339286 *
     5) paid=yes 32  216.46880 5.718750
      10) age>=17.5 10   82.90000 4.100000 *
      11) age< 17.5 22   95.45455 6.454545 *
   3) absences>=0.5 200  531.38000 5.660000
     6) absences>=13.5 42  111.61900 4.904762 *
     7) absences< 13.5 158  389.43670 5.860759
      14) schoolsup=yes 23   50.21739 4.847826 *
      15) schoolsup=no 135  311.60000 6.033333
        30) studytime< 3.5 127  276.30710 5.940945 *
        31) studytime>=3.5 8   17.00000 7.500000 *
```

- Plot the decision tree using `rpart.plot()`.

```r
# Plot the tree model
rpart.plot(x = grade_model, yesno = 2, type = 0, extra = 0)
```



3.4 Performance Metrics for Regression

3.4.1 Evaluate a regression tree model

Predict the final grade for all students in the test set. The grade is on a 0-20 scale. Evaluate the model based on test set RMSE (Root Mean Squared Error). RMSE tells us approximately how far away our predictions are from the true values.

Exercise

- First generate predictions on the `grade_test` data frame using the `grade_model` object.

```
# Generate predictions on a test set
pred <- predict(object = grade_model,  # model object
                newdata = grade_test)  # test dataset
```

- After generating test set predictions, use the `rmse()` function from the **Metrics** package to compute test set RMSE.

```
# Compute the RMSE
rmse(actual = grade_test$final_grade,
     predicted = pred)
```

```
[1] 2.278249
```

3.5 What are the Hyperparameters for a Decision Tree?

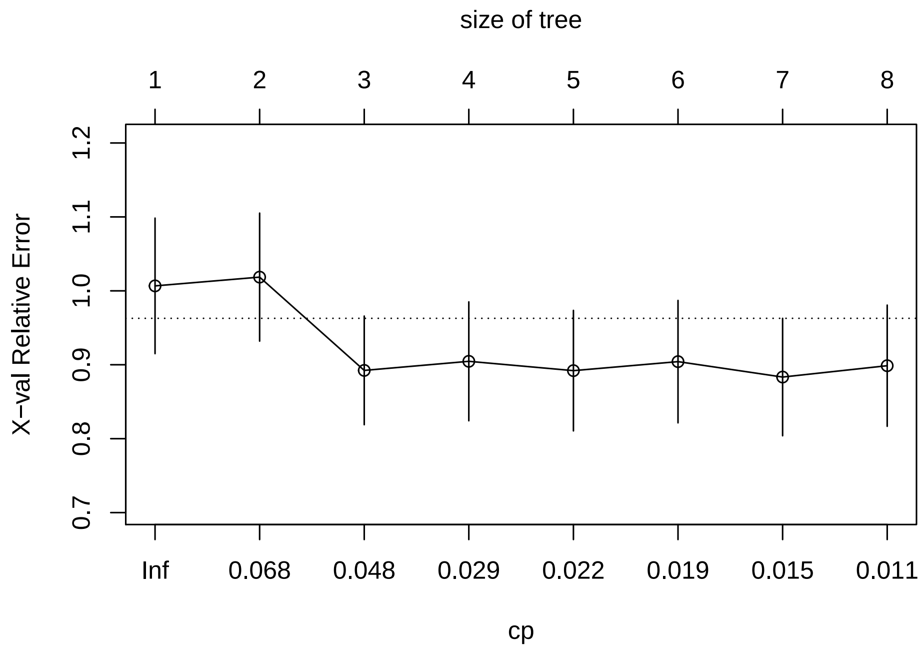
3.5.1 Tuning the Model

Tune (or "trim") the model using the `prune()` function by finding the best "CP" value (CP stands for "Complexity Parameter").

Exercise

- Print the CP Table, a matrix of information on the optimal prunings (based on CP).

```
# Plot the "CP Table"
plotcp(grade_model)
```




```
# Print the "CP Table"
print(grade_model$cptable)
```

```
          CP nsplit rel error    xerror       xstd
1 0.06839852      0 1.0000000 1.0066743 0.09169976
2 0.06726713      1 0.9316015 1.0185398 0.08663026
3 0.03462630      2 0.8643344 0.8923588 0.07351895
4 0.02508343      3 0.8297080 0.9046335 0.08045100
5 0.01995676      4 0.8046246 0.8920489 0.08153881
6 0.01817661      5 0.7846679 0.9042142 0.08283114
7 0.01203879      6 0.7664912 0.8833557 0.07945742
8 0.01000000      7 0.7544525 0.8987112 0.08200148
```

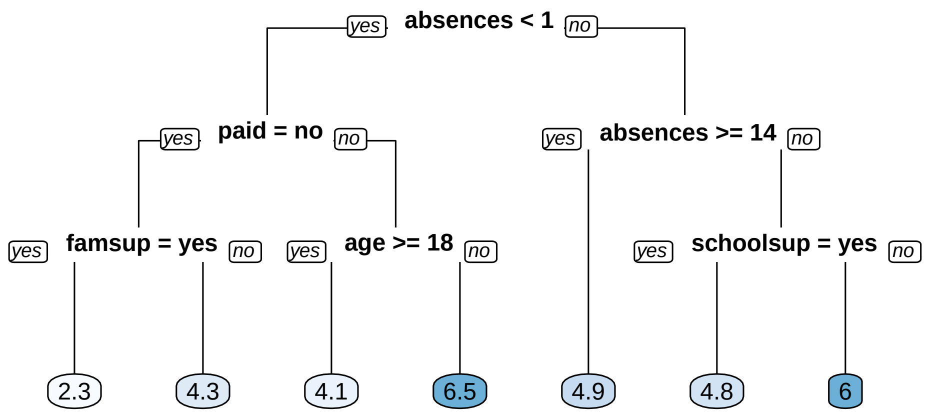
- Retrieve the optimal CP value; the value for CP which minimizes cross-validated error of the model.

```
# Retrieve optimal cp value based on cross-validated error
opt_index <- which.min(grade_model$cptable[, "xerror"])
cp_opt <- grade_model$cptable[opt_index, "CP"]
```

- Use the `prune()` function trim the tree, snipping off the least important splits, based on CP.

```
# Prune the model (to optimized cp value)
grade_model_opt <- prune(tree = grade_model,
                         cp = cp_opt)
# Plot the optimized model
```

```r
rpart.plot(x = grade_model_opt, yesno = 2, type = 0, extra = 0)
```



3.6 Grid Search for Model Selection

3.6.1 Generate a grid of hyperparameter values

Use `expand.grid()` to generate a grid of `maxdepth` and `minsplit` values.

Exercise

- Establish a list of possible values for `minsplit` and `maxdepth`.

```r
# Establish a list of possible values for minsplit and maxdepth
minsplit <- seq(1, 4, 1)
maxdepth <- seq(1, 6, 1)
```

- Use the `expand.grid()` function to generate a data frame containing all combinations

```r
# Create a data frame containing all combinations
hyper_grid <- expand.grid(minsplit = minsplit, maxdepth = maxdepth)
```

- Take a look at the resulting grid object

```r
# Check out the grid
head(hyper_grid)
```

```
  minsplit maxdepth
1        1        1
2        2        1
3        3        1
4        4        1
5        1        2
6        2        2
```

```r
# Print the number of grid combinations
nrow(hyper_grid)
```

```
[1] 24
```

3.6.2 Generate a grid of models

In this exercise, we will write a simple loop to train a "grid" of models and store the models in a list called `grade_models`. R users who are familiar with the `apply` functions in R could think about how this loop could be easily converted into a function applied to a list as an extra-credit thought experiment.

Exercise

- Create an empty list to store the models from the grid search.

```r
# Number of potential models in the grid
num_models <- nrow(hyper_grid)
# Create an empty list to store models
grade_models <- list()
```

- Write a loop that trains a model for each row in `hyper_grid` and adds it to the `grade_models` list.
  - The loop will by indexed by the rows of `hyper_grid`.
  - For each row, there is a unique combination of the `minsplit` and `maxdepth` values that will be used to train a model.

```r
# Write a loop over the rows of hyper_grid to train the grid of models
for (i in 1:num_models) {

    # 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
    grade_models[[i]] <- rpart(formula = final_grade ~ .,
                               data = grade_train,
                               method = "anova",
                               minsplit = minsplit,
                               maxdepth = maxdepth)
}
```

3.6.3 Evaluate the grid

Earlier in the chapter we split the dataset into three parts: training, validation and test.

A dataset that is not used in training is sometimes referred to as a "holdout" set. A holdout set is used to estimate model performance and although both validation and test sets are considered to be holdout data, there is a key difference:

- Just like a test set, a validation set is used to evaluate the performance of a model. The difference is that a validation set is specifically used to compare the performance of a group of models with the goal of choosing a "best model" from the group. All the models in a group are evaluated on the same validation set and the model with the best performance is considered to be the winner.
- Once you have the best model, a final estimate of performance is computed on the test set.
- A test set should only ever be used to estimate model performance and should not be used in model selection. Typically if you use a test set more than once, you are probably doing something wrong.

Exercise

- Write a loop that evaluates each model in the `grade_models` list and stores the validation RMSE in a vector called `rmse_values`.

```
# Number of potential models in the grid
num_models <- length(grade_models)

# Create an empty vector to store RMSE values
```

```r
rmse_values <- c()

# Write a loop over the models to compute validation RMSE
for (i in 1:num_models) {

    # Retrieve the i^th model from the list
    model <- grade_models[[i]]

    # Generate predictions on grade_valid
    pred <- predict(object = model,
                    newdata = grade_valid)

    # Compute validation RMSE and add to the
    rmse_values[i] <- rmse(actual = grade_valid$final_grade,
                           predicted = pred)
}
```

- The `which.min()` function can be applied to the `rmse_values` vector to identify the index containing the smallest RMSE value.
  - The model with the smallest validation set RMSE will be designated as the "best model".

```r
# Identify the model with smallest validation set RMSE
best_model <- grade_models[[which.min(rmse_values)]]
```

- Inspect the model parameters of the best model.

```r
# Print the model paramters of the best model
best_model$control
```

- Generate predictions on the test set using the best model to compute test set RMSE.

```r
# Compute test set RMSE on best_model
pred <- predict(object = best_model,
                newdata = grade_test)
rmse(actual = grade_test$final_grade,
     predicted = pred)
```

```
[1] 2.124109
```

Chapter 4

Bagged Trees

4.1 Introduction to Bagged Trees

4.1.1 Advantages of bagged trees

What are the advantages of bagged trees compared to a single tree?

- Increases the accuracy of the resulting predictions
- Easier to interpret the resulting model
- Reduces variance by averaging a set of observations
- 1 and 2 are correct
- **1 and 3 are correct**
- 2 and 3 are correct

4.2 Train a Bagged Tree Model

Let's start by training a bagged tree model. You'll be using the `bagging()` function from the `ipred` package. The number of bagged trees can be specified using the `nbagg` parameter, but here we will use the default (25).

If we want to estimate the model's accuracy using the "out-of-bag" (OOB) samples, we can set the the `coob` parameter to `TRUE`. The OOB samples are the training obsevations that were not selected into the bootstrapped sample (used in training). Since these observations were not used in training, we can use them instead to evaluate the accuracy of the model (done automatically inside the `bagging()` function).

Exercise

The `credit_train` and `credit_test` datasets from Chapter 1 are already loaded in the workspace.

- Use the `bagging()` function to train a bagged tree model.

```
# Bagging is a randomized model, so let's set a seed (123) for reproducibility
set.seed(123)

# Train a bagged model
credit_model <- bagging(formula = factor(default) ~ .,
                        data = credit_train,
                        coob = TRUE)
```

- Inspect the model by printing it.

```
# Print the model
print(credit_model)
```

```

Bagging classification trees with 25 bootstrap replications

Call: bagging.data.frame(formula = factor(default) ~ ., data = credit_train,
    coob = TRUE)

Out-of-bag estimate of misclassification error:  0.2537
```

4.3 Evaluating the Bagged Tree Performance

4.3.1 Prediction and confusion matrix

As you saw in the video, a confusion matrix is a very useful tool for examining all possible outcomes of your predictions (true positive, true negative, false positive, false negative).

In this exercise, you will predict those who will default using bagged trees. You will also create the confusion matrix using the `confusionMatrix()` function from the **caret** package.

It's always good to take a look at the output using the `print()` function.

Exercise

The fitted model object, `credit_model`, is already in your workspace.

- Use the `predict()` function with `type = "class"` to generate predicted labels on the `credit_test` dataset.

```
# Generate predicted classes using the model object
class_prediction <- predict(object = credit_model,
                            newdata = credit_test,
                            type = "class")
```

- Take a look at the prediction using the `print()` function.

```
# Print the predicted classes
print(class_prediction)
```

```
  [1] no  no  no  no  yes no  no  no  no  no  no  no  no  yes no  no  no  no
 [19] no  no  yes no  no  no  no  no  yes no  no  no  no  no  no  no  no  no
 [37] yes yes no  yes no  yes no  no  no  no  no  no  no  yes no  yes no  yes
 [55] yes no  yes no  yes no  no  yes no  no  yes yes no  yes no  no  no  yes
 [73] yes no  no  no  no  no  no  yes no  no  no  no  yes no  no  yes no  no
 [91] no  no  no  yes yes no  no  no  no  no  no  yes no  no  yes no  no  no
[109] no  no  no  no  no  no  no  no  no  no  no  no  yes no  yes no  no  yes
[127] yes no  yes no  no  no  no  no  yes no  yes yes no  no  no  no  yes no
[145] no  no  yes no  no  no  no  yes no  no  no  no  no  no  no  yes no  no
[163] yes no  yes no  no  no  no  no  no  no  no  no  no  no  no  no  no  no
[181] no  no  yes yes yes no  yes no  no  no  no  no  yes no  no  no  yes no
[199] no  yes
Levels: no yes
```

- Calculate the confusion matrix using the `confusionMatrix` function.

```
# Calculate the confusion matrix for the test set
confusionMatrix(data = class_prediction,
               reference = factor(credit_test$default))
```

```
Confusion Matrix and Statistics

          Reference
Prediction  no yes
```

```
       no  119  33
       yes  11  37

               Accuracy : 0.78
                 95% CI : (0.7161, 0.8354)
    No Information Rate : 0.65
    P-Value [Acc > NIR] : 4.557e-05

                  Kappa : 0.4787

 Mcnemar's Test P-Value : 0.001546

            Sensitivity : 0.9154
            Specificity : 0.5286
         Pos Pred Value : 0.7829
         Neg Pred Value : 0.7708
             Prevalence : 0.6500
         Detection Rate : 0.5950
   Detection Prevalence : 0.7600
      Balanced Accuracy : 0.7220

       'Positive' Class : no
```

4.3.2 Predict on a Test Set and Compute AUC

In binary classification problems, we can predict numeric values instead of class labels. In fact, class labels are created only after you use the model to predict a raw, numeric, *predicted value* for a test point.

The *predicted label* is generated by applying a threshold to the *predicted value*, such that all tests points with predicted value greater than that threshold get a predicted label of "1" and, points below that threshold get a predicted label of "0".

In this exercise, generate predicted values (rather than class labels) on the test set and evaluate performance based on AUC (Area Under the ROC Curve). The AUC is a common metric for evaluating the discriminatory ability of a binary classification model.

Exercise

- Use the `predict()` function with `type = "prob"` to generate numeric predictions on the `credit_test` dataset.

```
# Generate predictions on the test set
pred <- predict(object = credit_model,
                newdata = credit_test,
                type = "prob")

# `pred` is a matrix
class(pred)
```

```
[1] "matrix"
```

```
# Look at the pred format
head(pred)
```

```
       no  yes
[1,] 0.92 0.08
[2,] 0.92 0.08
[3,] 1.00 0.00
[4,] 1.00 0.00
[5,] 0.16 0.84
[6,] 0.84 0.16
```

- Compute the AUC using the `auc()` function from the **Metrics** package.

```
# Compute the AUC (`actual` must be a binary (or 1/0 numeric) vector)
auc(actual = ifelse(credit_test$default == "yes", 1, 0),
    predicted = pred[,"yes"])
```

```
[1] 0.8084066
```

4.4 Using `caret` for Cross-Validating Models

4.4.1 Cross-validate a bagged tree model in caret

Use `caret::train()` with the `"treebag"` method to train a model and evaluate the model using cross-validated AUC. The **caret** package allows the user to easily cross-validate any model across any relevant performance metric. In this case, we will use 5-fold cross validation and evaluate cross-validated AUC (Area Under the ROC Curve).

Exercise

The `credit_train` dataset is in your workspace. You will use this data frame as the training data.

- First specify a `ctrl` object, which is created using the `caret::trainControl()` function.

```
# Specify the training configuration
ctrl <- trainControl(method = "cv",     # Cross-validation
                     number = 5,      # 5 folds
                     classProbs = TRUE,                  # For AUC
                     summaryFunction = twoClassSummary)  # For AUC
```

- In the `trainControl()` function, you can specify many things. We will set: `method = "cv"`, `number = 5` for 5-fold cross-validation. Also, two options that are required if you want to use AUC as the metric: `classProbs = TRUE` and `summaryFunction = twoClassSummary`.

```
# Cross validate the credit model using "treebag" method;
# Track AUC (Area under the ROC curve)
set.seed(1)  # for reproducibility
credit_caret_model <- train(default ~ .,
                            data = credit_train,
                            method = "treebag",
                            metric = "ROC",
                            trControl = ctrl)

# Look at the model object
print(credit_caret_model)
```

```
Bagged CART

800 samples
 16 predictor
  2 classes: 'no', 'yes'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 640, 640, 640, 640, 640
Resampling results:
```

```
  ROC        Sens       Spec
  0.7309497  0.8789474  0.4086957
```

```
# Inspect the contents of the model list
names(credit_caret_model)
```

```
 [1] "method"       "modelInfo"    "modelType"    "results"      "pred"
 [6] "bestTune"     "call"         "dots"         "metric"       "control"
[11] "finalModel"   "preProcess"   "trainingData" "resample"     "resampledCM"
[16] "perfNames"    "maximize"     "yLimits"      "times"        "levels"
[21] "terms"        "coefnames"    "contrasts"    "xlevels"
```

```
# Print the CV AUC
credit_caret_model$results[,"ROC"]
```

```
[1] 0.7309497
```

4.4.2 Generate predictions from the caret model

Generate predictions on a test set for the caret model.

- First generate predictions on the `credit_test` data frame using the `credit_caret_model` object.

```
# Generate predictions on the test set
predc <- predict(object = credit_caret_model,
                 newdata = credit_test,
                 type = "prob")
```

- After generating test set predictions, use the `auc()` function from the `Metrics` package to compute AUC.

```
# Compute the AUC (`actual` must be a binary (or 1/0 numeric) vector)
auc(actual = ifelse(credit_test$default == "yes", 1, 0),
                    predicted = predc[,"yes"])
```

```
[1] 0.7782967
```

4.5 Compare test set performance to CV performance

In this excercise, you will print test set AUC estimates that you computed in previous exercises. These two methods use the same code underneath, so the estimates should be very similar.

- The `credit_ipred_model_test_auc` object stores the test set AUC from the model trained using the `ipred::bagging()` function.
- The `credit_caret_model_test_auc` object stores the test set AUC from the model trained using the `caret::train()` function with `method = "treebag"`.

Lastly, we will print the 5-fold cross-validated estimate of AUC that is stored within the `credit_caret_model` object. This number will be a more accurate estimate of the true model performance since we have averaged the performance over five models instead of just one.

On small datasets like this one, the difference between test set model performance estimates and cross-validated model performance estimates will tend to be more pronounced. When using small data, it's recommended to use cross-validated estimates of performance because they are more stable.

Exercise

- Print the object credit_ipred_model_test_auc.

```
# Print ipred::bagging test set AUC estimate
print(credit_ipred_model_test_auc)
```

```
[1] 0.8084066
```

- Print the object credit_caret_model_test_auc.

```
# Print caret "treebag" test set AUC estimate
print(credit_caret_model_test_auc)
```

```
[1] 0.7782967
```

- Compare these to the 5-fold cross validated AUC.

```
# Compare to caret 5-fold cross-validated AUC
credit_caret_model$results[, "ROC"]
```

```
[1] 0.7309497
```

Chapter 5

Random Forests

5.1 Introduction to Random Forests

5.1.1 Bagged trees vs. Random Forest

What is the main difference between bagged trees and the Random Forest algorithm?

- In Random Forest, the decision trees are trained on a random subset of the rows, but in bagging, they use all the rows.
- **In Random Forest, only a subset of features are selected at random at each split in a decision tree. In bagging, all features are used.**
- In Random Forest, there is randomness. In bagging, there is no randomness.

5.2 Train a Random Forest model

Here you will use the `randomForest()` function from the **randomForest** package to train a Random Forest classifier to predict loan default.

Exercise

The `credit_train` and `credit_test` datasets (from Chapter 1 & 3) are already loaded in the workspace.

- Use the `randomForest::randomForest()` function to train a Random Forest model on the `credit_train` dataset.

```
# Train a Random Forest
set.seed(1)  # for reproducibility
credit_model <- randomForest(default ~ .,
                             credit_Train)
```

- The formula used to define the model is the same as in previous chapters – we want to predict "default" as a function of all the other columns in the training set.
- Inspect the model output.

```
# Print the model output
print(credit_model)
```

```
##
## Call:
##  randomForest(formula = default ~ ., data = credit_Train)
##                Type of random forest: classification
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
##         OOB estimate of  error rate: 24.12%
## Confusion matrix:
##      no yes class.error
## no  527  43   0.0754386
## yes 150  80   0.6521739
```

5.3 Understanding Random Forest Model Output

5.3.1 Evaluate out-of-bag error

Here you will plot the OOB error as a function of the number of trees trained, and extract the final OOB error of the Random Forest model from the trained model object.

Exercise

The `credit_model` trained in the previous exercise is loaded in the workspace.

- Get the OOB error rate for the Random Forest model.

```r
# Grab OOB error matrix & take a look
err <- credit_model$err.rate
head(err)
```

```
           OOB        no       yes
[1,] 0.3170732 0.2150000 0.5517241
[2,] 0.3525641 0.2400000 0.6083916
[3,] 0.3310924 0.2091346 0.6145251
[4,] 0.3333333 0.2154812 0.6192893
[5,] 0.3264746 0.1992263 0.6367925
[6,] 0.3040000 0.1872659 0.5925926
```

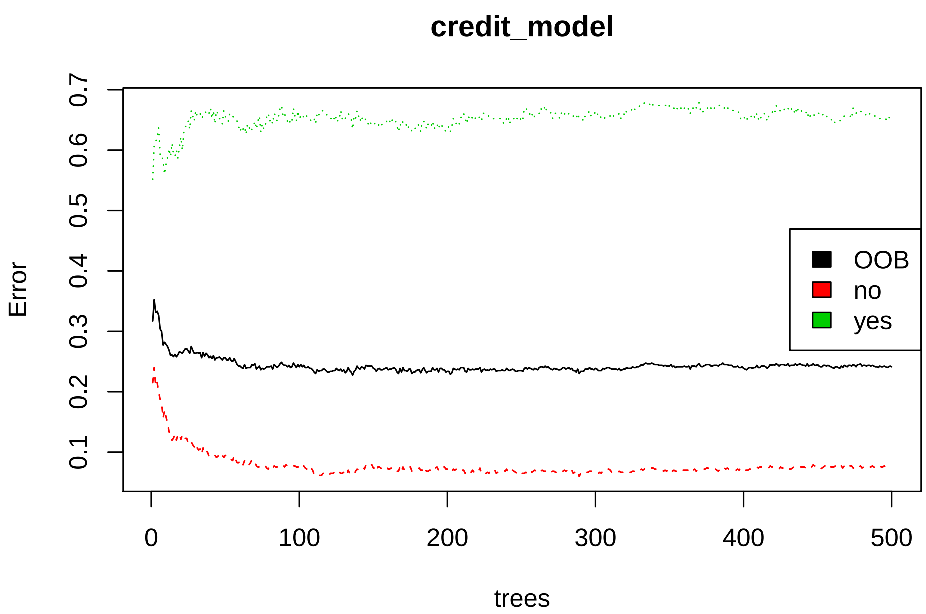
```r
# Look at final OOB error rate (last row in err matrix)
oob_err <- err[500, "OOB"]
print(oob_err)
```

```
    OOB
0.24125
```

- Plot the OOB error rate against the number of trees in the forest.

```r
# Plot the model trained in the previous exercise
plot(credit_model)

# Add a legend since it doesn't have one by default
legend(x = "right",
       legend = colnames(err),
       fill = 1:ncol(err))
```



5.3.2 Evaluate model performance on a test set

Use the `caret::confusionMatrix()` function to compute test set accuracy and generate a confusion matrix. Compare the test set accuracy to the OOB accuracy.

Exercise

- Generate class predictions for the `credit_test` data frame using the `credit_model` object.

```
# Generate predicted classes using the model object
class_prediction <- predict(object = credit_model,    # model object
                            newdata = credit_Test,   # test dataset
                            type = "class") # return classification labels
```

- Using the `caret::confusionMatrix()` function, compute the confusion matrix for the test set.

```
# Calculate the confusion matrix for the test set
cm <- confusionMatrix(data = class_prediction,       # predicted classes
```

```r
                      reference = credit_Test$default)  # actual classes
print(cm)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  no yes
       no  123  40
       yes   7  30

               Accuracy : 0.765
                 95% CI : (0.7, 0.8219)
    No Information Rate : 0.65
    P-Value [Acc > NIR] : 0.0002983

                  Kappa : 0.4205

 Mcnemar's Test P-Value : 3.046e-06

            Sensitivity : 0.9462
            Specificity : 0.4286
         Pos Pred Value : 0.7546
         Neg Pred Value : 0.8108
             Prevalence : 0.6500
         Detection Rate : 0.6150
   Detection Prevalence : 0.8150
      Balanced Accuracy : 0.6874

       'Positive' Class : no
```

- Compare the test set accuracy reported from the confusion matrix to the OOB accuracy. The OOB error is stored in `oob_err`, which is already in your workspace, and so OOB accuracy is just `1 - oob_err`.

```r
# Compare test set accuracy to OOB accuracy
paste0("Test Accuracy: ", cm$overall[1])
```

```
[1] "Test Accuracy: 0.765"
```

```r
paste0("OOB Accuracy: ", 1 - oob_err)
```

```
[1] "OOB Accuracy: 0.75875"
```

5.4 OOB Error vs. Test Set Error

5.4.1 Advantage of OOB error

What is the main advantage of using OOB error instead of validation or test error?

- Tuning the model hyperparameters using OOB error will lead to a better model.
- **If you evaluate your model using OOB error, then you don't need to create a separate test set.**
- OOB error is more accurate than test set error.

5.4.2 Evaluate Test Set AUC

In Chapter 3, we learned about the AUC metric for evaluating binary classification models. In this exercise, you will compute test set AUC for the Random Forest model.

Exercise

- Use the `predict()` function with `type = "prob"` to generate numeric predictions on the `credit_test` dataset.

```
# Generate predictions on the test set
pred <- predict(object = credit_model,
                newdata = credit_Test,
                type = "prob")

# `pred` is a matrix
class(pred)
```

```
[1] "matrix" "votes"
```

```
# Look at the pred format
head(pred)
```

```
      no   yes
1  0.904 0.096
3  0.902 0.098
7  1.000 0.000
9  0.970 0.030
12 0.216 0.784
22 0.826 0.174
```

```
credit_Model <- randomForest(default ~ .,
                             credit_Train)
rf_preds <- predict(object = credit_Model,
                newdata = credit_Test)
```

- Compute the AUC using the `auc()` function from the **Metrics** package.

```
# Compute the AUC (`actual` must be a binary 1/0 numeric vector)
auc(actual = ifelse(credit_Test$default == "yes", 1, 0),
    predicted = pred[,"yes"])
```

```
[1] 0.8187363
```

5.5 Tuning a Random Forest Model

5.5.1 Tuning a Random Forest via mtry

In this exercise, you will use the `randomForest::tuneRF()` to tune `mtry` (by training several models). This function is a specific utility to tune the `mtry` parameter based on OOB error, which is helpful when you want a quick & easy way to tune your model. A more generic way of tuning Random Forest parameters will be presented in the following exercise.

Exercise

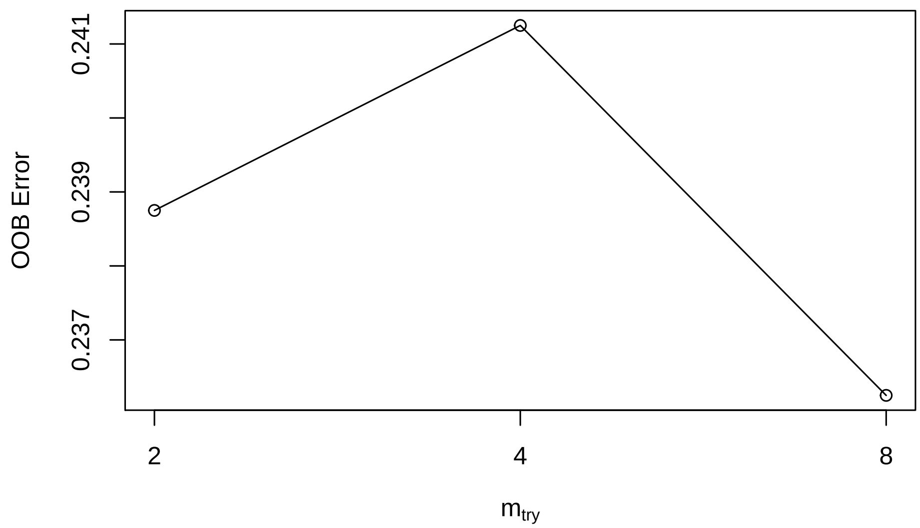
- Use the `tuneRF()` function in place of the `randomForest()` function to train a series of models with different `mtry` values and examine the the results.
  - Note that (unfortunately) the `tuneRF()` interface does not support the typical formula input that we've been using, but instead uses

two arguments, `x` (matrix or data frame of predictor variables) and `y` (response vector; must be a factor for classification).

- The `tuneRF()` function has an argument, `ntreeTry` that defaults to 50 trees. Set `nTreeTry = 500` to train a random forest model of the same size as you previously did.

```
# Execute the tuning process
set.seed(1)
res <- tuneRF(x = subset(credit_Train, select = -default),
              y = credit_Train$default,
              ntreeTry = 500)
```

```
mtry = 4  OOB error = 24.12%
Searching left ...
mtry = 2    OOB error = 23.88%
0.01036269 0.05
Searching right ...
mtry = 8    OOB error = 23.62%
0.02072539 0.05
```



- After tuning the forest, this function will also plot model performance (OOB error) as a function of the `mtry` values that were evaluated.
  - Keep in mind that if we want to evaluate the model based on AUC instead of error (accuracy), then this is not the best way to tune a model, as the selection only considers (OOB) error.

```
# Look at results
print(res)
```

```
      mtry OOBError
2.OOB    2  0.23875
```

```
4.OOB    4  0.24125
8.OOB    8  0.23625
```

```r
# Find the mtry value that minimizes OOB Error
mtry_opt <- res[,"mtry"][which.min(res[,"OOBError"])]
print(mtry_opt)
```

```
8.OOB
    8
```

5.5.2 Tuning a Random Forest via tree depth

In Chapter 2, we created a manual grid of hyperparameters using the `expand.grid()` function and wrote code that trained and evaluated the models of the grid in a loop. In this exercise, you will create a grid of `mtry`, `nodesize` and `sampsize` values. In this example, we will identify the "best model" based on OOB error. The best model is defined as the model from our grid which minimizes OOB error.

Keep in mind that there are other ways to select a best model from a grid, such as choosing the best model based on validation AUC. However, for this exercise, we will use the built-in OOB error calculations instead of using a separate validation set.

Exercise

- Create a grid of `mtry`, `nodesize` and `sampsize` values.

```r
# Establish a list of possible values for mtry, nodesize and sampsize
mtry <- seq(4, ncol(credit_Train) * 0.8, 2)
nodesize <- seq(3, 8, 2)
sampsize <- nrow(credit_Train) * c(0.7, 0.8)

# Create a data frame containing all combinations
hyper_grid <- expand.grid(mtry = mtry, nodesize = nodesize, sampsize = sampsize)

# Create an empty vector to store OOB error values
oob_err <- c()
```

- Write a simple loop to train all the models and choose the best one based on OOB error.

```r
# Write a loop over the rows of hyper_grid to train the grid of models
for (i in 1:nrow(hyper_grid)) {

    # Train a Random Forest model
    model <- randomForest(formula = default ~ .,
                          data = credit_Train,
                          mtry = hyper_grid$mtry[i],
                          nodesize = hyper_grid$nodesize[i],
                          sampsize = hyper_grid$sampsize[i])

    # Store OOB error for the model
    oob_err[i] <- model$err.rate[nrow(model$err.rate), "OOB"]
}
```

- Print the set of hyperparameters which produced the best model.

```r
# Identify optimal set of hyperparmeters based on OOB error
opt_i <- which.min(oob_err)
print(hyper_grid[opt_i,])
```

```
  mtry nodesize sampsize
2    6        3      560
```

Chapter 6

Boosted Trees

6.1 Introduction to Boosting

6.1.1 Bagged trees vs. boosted trees

What is the main difference between bagged trees and boosted trees?

- Boosted trees don't perform as well as bagged trees.
- Boosted trees have fewer hyperparameters to tune than bagged trees.
- **Boosted trees improve the model fit by considering past fits and bagged trees do not.**

6.2 Train a GBM Model

Here you will use the `gbm()` function to train a GBM classifier to predict loan default. You will train a 10,000-tree GBM on the `credit_train` dataset, which is pre-loaded into your workspace.

Using such a large number of trees (10,000) is probably not optimal for a GBM model, but we will build more trees than we need and then select the optimal number of trees based on early performance-based stopping. The best GBM model will likely contain fewer trees than we started with.

For binary classification, `gbm()` requires the response to be encoded as 0/1 (numeric), so we will have to convert from a "no/yes" factor to a 0/1 numeric response column.

Also, the the `gbm()` function requires the user to specify a `distribution` argument. For a binary classification problem, you should set `distribution = "bernoulli"`. The Bernoulli distribution models a binary response.

Exercise

- Convert from a "no/yes" factor to a 0/1 numeric response column using the `ifelse()` function.

```
# Convert "yes" to 1, "no" to 0
credit_train$default <- ifelse(credit_train$default == "yes", 1, 0)
```

- Train a 10,000-tree GBM model.

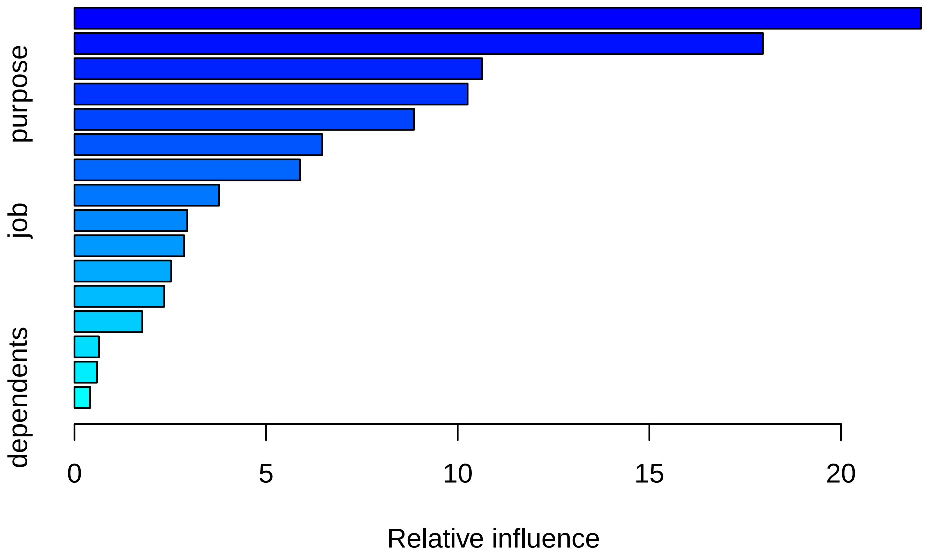
```
# Train a 10000-tree GBM model
set.seed(1)

credit_model <- gbm(formula = default ~ .,
                    distribution = "bernoulli",
                    data = credit_train,
                    n.trees = 10000)

# Print the model object
print(credit_model)
```

```
gbm(formula = default ~ ., distribution = "bernoulli", data = credit_train,
    n.trees = 10000)
A gradient boosted model with bernoulli loss function.
10000 iterations were performed.
There were 16 predictors of which 16 had non-zero influence.
```

```
# summary() prints variable importance
summary(credit_model)
```




```
                                         var    rel.inf
amount                                amount 22.0897595
age                                      age 17.9626175
credit_history                credit_history 10.6369658
purpose                              purpose 10.2584546
employment_duration      employment_duration  8.8596192
checking_balance            checking_balance  6.4650840
months_loan_duration months_loan_duration  5.8863990
savings_balance              savings_balance  3.7722735
job                                      job  2.9418015
other_credit                    other_credit  2.8613862
housing                              housing  2.5237773
years_at_residence        years_at_residence  2.3409228
percent_of_income          percent_of_income  1.7687143
phone                                  phone  0.6373101
existing_loans_count existing_loans_count  0.5870700
dependents                        dependents  0.4078447
```

6.3 Understanding GBM Model Output

6.3.1 Prediction using a GBM model

The **gbm** package uses a `predict()` function to generate predictions from a model, similar to many other machine learning packages in R. When you see a function like `predict()` that works on many different types of input (a GBM model, a RF model, a GLM model, etc), that indicates that `predict()` is an "alias" for a GBM-specific version of that function. The GBM specific version of that function is `predict.gbm()`, but for convenience sake, we can just use `predict()` (either works).

One thing that's particular to the `predict.gbm()` however, is that you need to specify the number of trees used in the prediction. There is no default, so you have to specify this manually. For now, we can use the same number of trees that we specified when training the model, which is 10,000 (though this may not be the optimal number to use).

Another argument that you can specify is `type`, which is only relevant to Bernoulli and Poisson distributed outcomes. When using Bernoulli loss, the returned value is on the log odds scale by default and for Poisson, it's on the log scale. If instead you specify `type = "response"`, then `gbm` converts the predicted values back to the same scale as the outcome. This will convert the predicted values into probabilities for Bernoulli and expected counts for Poisson.

Exercise

- Generate predictions on the test set, using 10,000 trees.

```
# Since we converted the training response col, let's also convert the test response c
credit_test$default <- ifelse(credit_test$default == "yes", 1, 0)

# Generate predictions on the test set
preds1 <- predict(object = credit_model,
                  newdata = credit_test,
                  n.trees = 10000)
```

- Generate predictions on the test set using `type = "response"` and 10,000 trees.

```
# Generate predictions on the test set (scale to response)
preds2 <- predict(object = credit_model,
                  newdata = credit_test,
                  n.trees = 10000,
                  type = "response")
```

- Compare the ranges of the two sets of predictions.

```r
# Compare the range of the two sets of predictions
range(preds1)
```

```
[1] -6.004812  4.646991
```

```r
range(preds2)
```

```
[1] 0.002460783 0.990500685
```

6.3.2 Evaluate test set AUC

Compute test set AUC of the GBM model for the two sets of predictions. We will notice that they are the same value. That's because AUC is a rank-based metric, so changing the actual values does not change the value of the AUC.

However, if we were to use a scale-aware metric like RMSE to evaluate performance, we would want to make sure we converted the predictions back to the original scale of the response.

Exercise

The `preds1` and `preds2` prediction vectors from the previous exercise are pre-loaded into the workspace.

- Compute AUC of the predictions.

```r
auc(actual = credit_test$default, predicted = preds1)
```

```
[1] 0.7142857
```

- Compute AUC of the predictions (scaled to response).

```r
auc(actual = credit_test$default, predicted = preds2)
```

```
[1] 0.7142857
```

- Notice that the AUC is the same!

6.4 GBM Hyperparameters

6.4.1 Early Stopping in GBMs

Use the `gbm.perf()` function to estimate the optimal number of boosting iterations (aka `n.trees`) for a GBM model object using both OOB and CV error. When you set out to train a large number of trees in a GBM (such as 10,000) and you use a validation method to determine an earlier (smaller) number of trees, then that's called "early stopping". The term "early stopping" is not unique to GBMs, but can describe auto-tuning the number of iterations in an iterative learning algorithm.

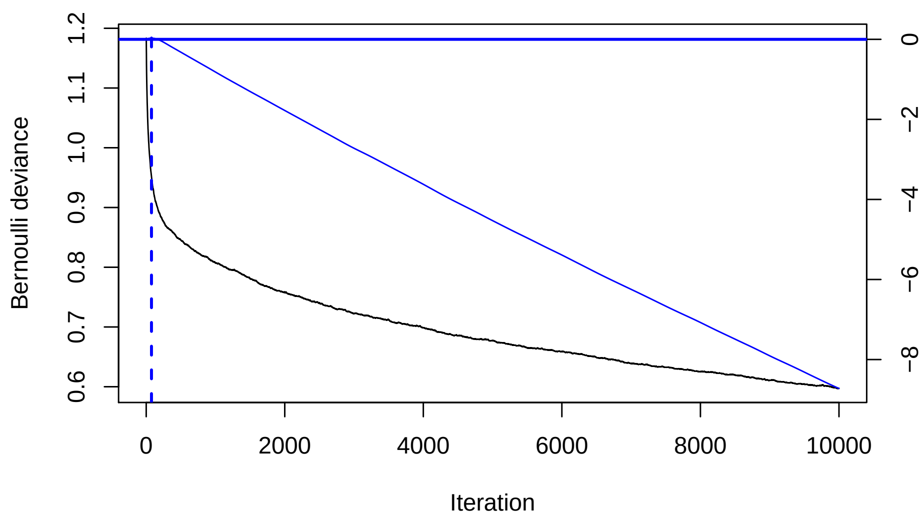
Exercise

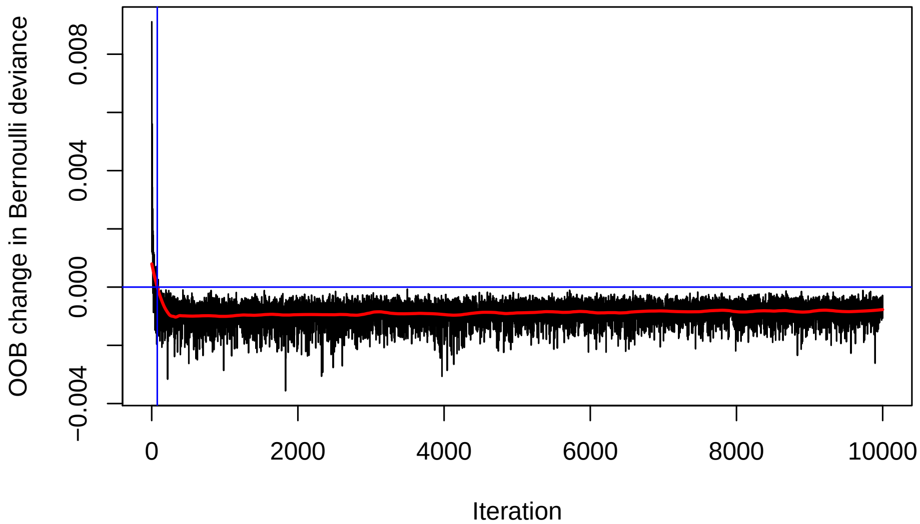
The `credit_model` object is loaded in the workspace.

- Use the `gbm.perf()` function with the "OOB" method to get the optimal number of trees based on the OOB error and store that number as `ntree_opt_oob`.

```
# Optimal ntree estimate based on OOB
ntree_opt_oob <- gbm.perf(object = credit_model,
                          method = "OOB",
                          oobag.curve = TRUE)
```

```
## OOB generally underestimates the optimal number of iterations although predictive pe
```





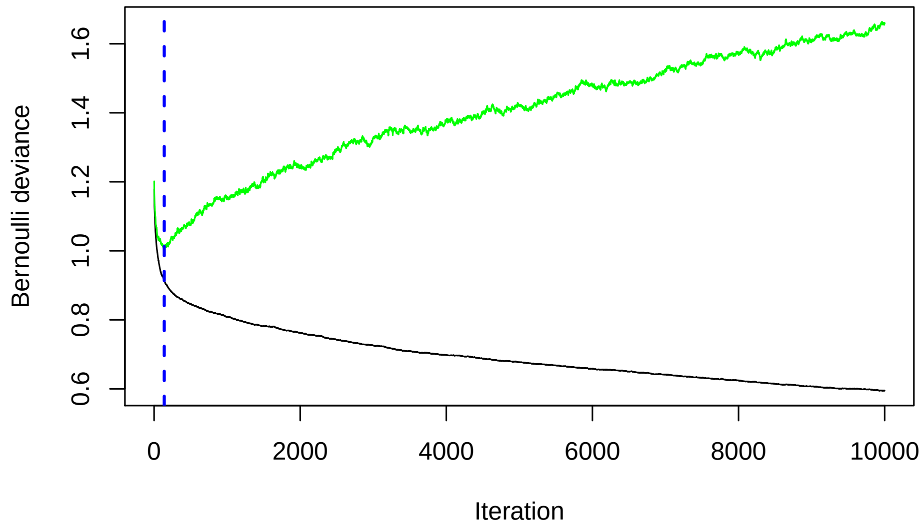
- Train a new GBM model, this time with cross-validation, so we can get a cross-validated estimate of the optimal number of trees.

```
# Train a CV GBM model
set.seed(1)
credit_model_cv <- gbm(formula = default ~ .,
                       distribution = "bernoulli",
                       data = credit_train,
                       n.trees = 10000,
                       cv.folds = 2,
                       n.cores = 1)
```

```
CV: 1
CV: 2
```

- Lastly, use the gbm.perf() function with the "cv" method to get the optimal number of trees based on the CV error and store that number as ntree_opt_cv.

```
# Optimal ntree estimate based on CV
ntree_opt_cv <- gbm.perf(object = credit_model_cv,
                         method = "cv")
```



- Compare the two numbers.

```
# Compare the estimates
print(paste0("Optimal n.trees (OOB Estimate): ", ntree_opt_oob))
```

```
[1] "Optimal n.trees (OOB Estimate): 76"
```

```
print(paste0("Optimal n.trees (CV Estimate): ", ntree_opt_cv))
```

```
[1] "Optimal n.trees (CV Estimate): 139"
```

6.4.2 OOB vs CV-Based Early Stopping

In the previous exercise, we used OOB error and cross-validated error to estimate the optimal number of trees in the GBM. These are two different ways to estimate the optimal number of trees, so in this exercise we will compare the performance of the models on a test set. We can use the same model object to make both of these estimates since the `predict.gbm()` function allows you to use any subset of the total number of trees (in our case, the total number is 10,000).

Exercise

The `ntree_opt_oob` and `ntree_opt_cv` objects from the previous exercise (each storing an “optimal” value for `n.trees`) are loaded in the workspace.

Using the `credit_model` loaded in the workspace, generate two sets of predictions:

- One using the OOB estimate of `n.trees`: 3,233 (stored in `ntree_opt_oob`)

```
# Generate predictions on the test set using ntree_opt_oob number of trees
preds1 <- predict(object = credit_model,
                  newdata = credit_test,
                  n.trees = ntree_opt_oob)
auc1 <- auc(actual = credit_test$default, predicted = preds1)
```

- And the other using the CV estimate of `n.trees`: 7,889 (stored in `ntree_opt_cv`)

```
# Generate predictions on the test set using ntree_opt_cv number of trees
preds2 <- predict(object = credit_model,
                  newdata = credit_test,
                  n.trees = ntree_opt_cv)
auc2 <- auc(actual = credit_test$default, predicted = preds2)
```

- Compare the AUCs

```
# Compare AUC
print(paste0("Test set AUC (OOB): ", auc1))
```

```
[1] "Test set AUC (OOB): 0.802527472527472"
```

```
print(paste0("Test set AUC (CV): ", auc2))
```

```
[1] "Test set AUC (CV): 0.788241758241758"
```

6.5 Model Comparison via ROC Curve & AUC

6.5.1 Compare All Models Based on AUC

In this final exercise, we will perform a model comparison across all types of models that we've learned about so far: Decision Trees, Bagged Trees, Random Forest and Gradient Boosting Machine (GBM). The models were all trained on the same training set, `credit_train`, and predictions were made for the `credit_test` dataset.

We have pre-loaded four sets of test set predictions, generated using the models we trained in previous chapters (one for each model type). The numbers stored in the prediction vectors are the raw predicted values themselves – not the predicted class labels. Using the raw predicted values, we can calculate test set AUC for each model and compare the results.

Exercise

Loaded in your workspace are four numeric vectors:

- `dt_preds`
- `bag_preds`
- `rf_preds`
- `gbm_preds`

These predictions were made on `credit_test`, which is also loaded into the workspace.

- Apply the `Metrics::auc()` function to each of these vectors to calculate test set AUC. Recall that the higher the AUC, the better the model.

```
# Generate the test set AUCs using the two sets of predictions & compare
a <- credit_Test$default
dt_auc <- auc(actual = a, predicted = dt_preds)
bag_auc <- auc(actual = a, predicted = bag_preds)
rf_auc <- auc(actual = a, predicted = rf_preds)
gbm_auc <- auc(actual = a, predicted = gbm_preds)
```

```
# Print results
sprintf("Decision Tree Test AUC: %.3f", dt_auc)
sprintf("Bagged Trees Test AUC: %.3f", bag_auc)
sprintf("Random Forest Test AUC: %.3f", rf_auc)
sprintf("GBM Test AUC: %.3f", gbm_auc)
```

6.5.2 Plot & Compare ROC Curves

We conclude this course by plotting the ROC curves for all the models (one from each chapter) on the same graph. The ROCR package provides the `prediction()` and `performance()` functions which generate the data required for plotting the ROC curve, given a set of predictions and actual (true) values.

The more "up and to the left" the ROC curve of a model is, the better the model. The AUC performance metric is literally the "Area Under the ROC Curve", so the greater the area under this curve, the higher the AUC, and the better-performing the model is.

Exercise

The **ROCR** package can plot multiple ROC curves on the same plot if you plot several sets of predictions as a list.

- The `prediction()` function takes as input a list of prediction vectors (one per model) and a corresponding list of true values (one per model, though in our case the models were all evaluated on the same test set so they all have the same set of true values). The `prediction()` function returns a "prediction" object which is then passed to the `performance()` function.

```
# List of predictions
preds_list <- list(dt_preds, bag_preds, rf_preds, gbm_preds)

# List of actual values (same for all)
m <- length(preds_list)
actuals_list <- rep(list(credit_test$default), m)

# Plot the ROC curves
pred <- prediction(preds_list, actuals_list)
```

- The `performance()` function generates the data necessary to plot the curve from the "prediction" object. For the ROC curve, you will also pass along two measures, `"tpr"` and `"fpr"`.

```
rocs <- performance(pred, "tpr", "fpr")
```

- Once you have the "performance" object, you can plot the ROC curves using the `plot()` method. We will add some color to the curves and a legend so we can tell which curves belong to which algorithm.

```
plot(rocs, col = as.list(1:m), main = "Test Set ROC Curves")
legend(x = "bottomright",
       legend = c("Decision Tree", "Bagged Trees", "Random Forest", "GBM"),
       fill = 1:m)
```