A Data Analysis

Baek kimin

2020-12-15

Contents

Prerequisites

Install R

1. Open an internet browser and go to www.r-project.org.
2. Click the "download R" link in the middle of the page under "Getting Started."
3. Select a CRAN location (a mirror site) and click the corresponding link.

4. Click on the "Download R for Windows" link at the top of the page.

5. Click on the "install R for the first time" link at the top of the page.
6. Click "Download R for Windows" and save the executable file somewhere on your computer. Run the .exe file and follow the installation instructions.

7. Now that R is installed, you need to download and install RStudio.

Install RStudio

1. Go to www.rstudio.com and click on the "Download RStudio" button.
2. Click on "Download RStudio Desktop."
3. Click on the version recommended for your system, or the latest Windows version, and save the executable file. Run the .exe file and follow the installation instructions.

Software information and conventions

The R session information when compiling this book is shown below:

```
sessionInfo()
```

```
## R version 4.0.3 (2020-10-10)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19042)
```

```
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=Korean_Korea.949  LC_CTYPE=Korean_Korea.949
## [3] LC_MONETARY=Korean_Korea.949 LC_NUMERIC=C
## [5] LC_TIME=Korean_Korea.949
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## loaded via a namespace (and not attached):
##  [1] compiler_4.0.3  magrittr_2.0.1  bookdown_0.21   tools_4.0.3
##  [5] htmltools_0.5.0 rstudioapi_0.13 yaml_2.2.1      stringi_1.5.3
##  [9] rmarkdown_2.5   knitr_1.30      stringr_1.4.0   xfun_0.19
## [13] digest_0.6.27   rlang_0.4.8     evaluate_0.14
```

We do not add prompts (> and +) to R source code in this book, and we comment out the text output with two hashes ## by default, as you can see from the R session information above. This is for your convenience when you want to copy and run the code (the text output will be ignored since it is commented out). Package names are in bold text (e.g., rmarkdown), and inline code and filenames are formatted in a typewriter font (e.g., knitr::knit('foo.Rmd')). Function names are followed by parentheses (e.g., bookdown::render_book()). The double-colon operator :: means accessing an object from a package.

Chapter 1

A Study on Determinants of Employment Status of University Graduates Youth

1.1 Introduction

Using a sample from Graduate Occupational Mobility Survey(GOMS), this study attempts to ascertain determinants of employment status of university graduates youth in labor market. In this case, employment status was measured by two aspects, whether employed or unemployed, and being full-time job at the large corporation or not.

The results reveal that variables such as location of university, major, GPA, the level of satisfaction with university providing services, whether setting a concrete career goal before graduating or not, and cost spent in language training abroad have a significant effect on labor market status (employed/unemployed, and being full-time job in large firms/part-time job) in common.

Also, the household income level in those days of entering university exert an influence on probability of being employed. Likewise, those graduates who are male and have experiences of supplementary private education to be employed are advantageous than female and those who did not.

1.2 Data Explanation

data filename : employ.csv

variable	description	value
school	type of school	College, University
location	school location	Metro, Non-Metro
gender	gender	Female, Male
gpa	grade point average	0-4.5
double_major	double major	No, Yes
language_course	language training abroad	No, Yes
english_test	take a TOEIC, TOEFL, … exam	No, Yes
work_experience	have a work experience	No, Yes
job_train	do a vocational course	No, Yes
license	have a qualification certificate	No, Yes
job_preparation	job preparation	0-7
father_edu	father's education level	High School',…
mother_edu	mother's education level	High School',…
parent_income	parent income	over 500', 'under 500'
major	major	Arts and Physical',…
employ	be employed	no, yes

We do not add prompts to R source code in this manual, and we comment out the text output by two hashes ## by default. Package names are in bold text (e.g., **rpart**), function names are formatted as code (e.g., `read.csv()`).

1.3 Install and load package

Packages are the way that third party R code is distributed. The Comprehensive R Archive Network (CRAN) provides hosting and listing of third party R packages that you can download. You can install a package hosted on CRAN by calling a function. It will then pop-up a dialog to ask you which mirror you would like to download the package from. For example, here is how you can install packages which are very useful for machine learning in R and will be used in later session:

```r
install.packages("tidyverse")
install.packages("rpart")
install.packages("rpart.plot")
install.packages("caret")
install.packages("DMwR")
install.packages("pROC")
install.packages("randomForest")
install.packages("ranger")
install.packages("ipred")
install.packages("gbm")
```

You can load a package in R by calling the `library()` function and passing the package name as the argument (it's a confusing function name, I know). For example, you can load the package as follows:

```r
library(tidyverse)
library(rpart)
library(rpart.plot)
library(caret)
library(DMwR)
library(pROC)
library(randomForest)
library(ranger)
library(ipred)
library(gbm)
```

1.4 Import Data

We will use the built in `read.csv()` function call, which reads the data in as a data frame, and assign the data frame to a variable (using `<-`) so that it is stored in R's memory. Then we will explore some of the basic arguments that can be supplied to the function. The default for `read.csv()` is to set the `header` argument to TRUE. This means that the first row of values in the .csv is set as header information (column names). If your data set does not have a header, set the `header` argument to FALSE. The `stringsAsFactors` argument is perhaps the most important argument in `read.csv()`, particularly if you are working with categorical data. This is because the default behavior of R is to convert character strings into factors, which may make it difficult to do such things as replace values.

```r
employ_all <- read.csv("employ.csv", stringsAsFactors=TRUE)
```

R is always pointed at a directory on your computer. You can find out which directory by running the `getwd()` (get working directory) function; this function has no arguments. To change your working directory, use `setwd()` and specify the path to the desired folder.

- `getwd()`
- `setwd(dir)` -dir – Specify a working directory. This may be from the root directory (starting with / on a Mac), it may include a double-dot (..) to move locally up a folder from the current directory, and it may include a path from the current directory.

```r
getwd()
```

```
## [1] "C:/Users/kimin/OneDrive/project/bookdown-demo-master"
```

```
setwd("c:/project")
```

1.5 Exploring and Preparing the Data

The first questions to ask in an investigation of a new dataset should be about how the dataset is organized. If you are fortunate, your source will provide a data dictionary, a document that describes the dataset's features. In our case, the used car data does not come with this documentation, so we'll need to create our own. The `str()` function provides a method for displaying the structure of R objects, such as data frames, vectors, or lists. It can be used to create the basic outline for our data dictionary:

```
str(employ_all)
```

```
## 'data.frame':    13331 obs. of  16 variables:
##  $ school         : Factor w/ 2 levels "College","University": 1 1 1 1 1 1 1 1 1 1
##  $ location       : Factor w/ 2 levels "Metro","Non-Metro": 1 1 1 1 2 2 2 1 1 1 ...
##  $ gender         : Factor w/ 2 levels "Female","Male": 1 1 1 2 2 1 2 2 1 1 ...
##  $ gpa            : num  3.5 2 2.8 3.8 4.2 4 4 2.9 2.9 4.4 ...
##  $ double_major   : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 1 1 1 ...
##  $ language_course: Factor w/ 2 levels "No","Yes": 1 1 1 1 2 1 1 1 1 1 ...
##  $ english_test   : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 1 2 1 2 ...
##  $ work_experience: Factor w/ 2 levels "No","Yes": 2 1 2 2 1 1 2 1 2 2 ...
##  $ job_train      : Factor w/ 2 levels "No","Yes": 2 1 1 1 1 1 1 1 1 1 ...
##  $ license        : Factor w/ 2 levels "No","Yes": 2 2 1 2 1 1 2 1 1 1 ...
##  $ job_preparation: int  3 3 2 1 0 3 1 1 3 3 ...
##  $ father_edu     : Factor w/ 2 levels "High School",..: 2 1 1 2 1 2 2 1 1 1 ...
##  $ mother_edu     : Factor w/ 2 levels "High School",..: 2 2 1 2 2 2 1 1 1 1 ...
##  $ parent_income  : Factor w/ 2 levels "over 500","under 500": 1 2 1 1 2 1 2 2 2 2
##  $ major          : Factor w/ 7 levels "Arts and Physical",..: 4 4 4 4 4 4 4 4 4 4
##  $ employ         : Factor w/ 2 levels "no","yes": 2 1 2 2 2 2 2 2 1 2 ...
```

For such a simple command, we learn a wealth of information about the dataset. The statement `13331 obs` informs us that the data includes 13,331 observations, which is just another way of saying that the dataset contains 13,331 records or examples. The number of observations is often simply abbreviated as n. The `16 variables` statement refers to the sixteen features that were recorded in the data. These features are listed by name on separate lines. Looking at the line for the feature called `location`, we note some additional details:

```
location : Factor w/ 2 levels "Metro","Non-Metro": 1 1 1 1 2 2 2
```

After the variable's name, the Factor label tells us that the feature is factor type. In this dataset, fourteen of the variables are factor while one is noted as num, int which refers to the numeric type and integer type. In R, we can get summary statistics for a dataset by using the `summary()` function. To get the summary statistics for our employ dataset, we pass the name of the dataset to the `summary()` function.

```
summary(employ_all)
```

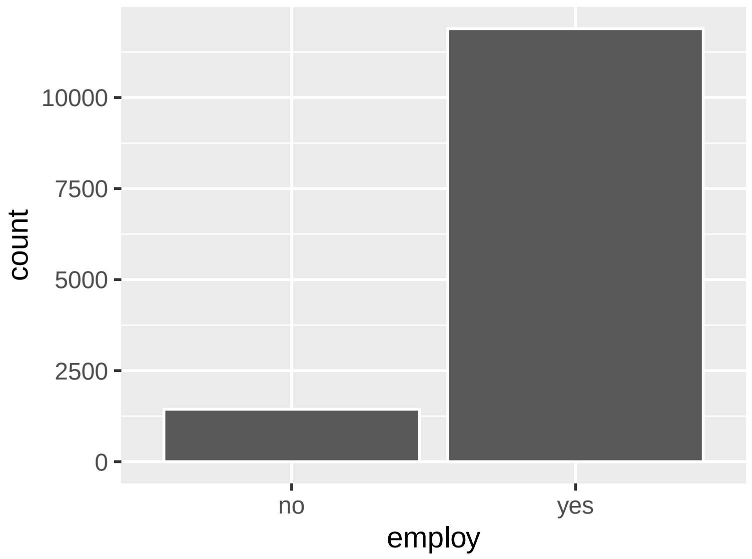
```
##         school           location       gender          gpa        double_major
##  College   : 2677   Metro    :5991   Female:6083   Min.   :1.000   No :11201
##  University:10654   Non-Metro:7340   Male  :7248   1st Qu.:3.300   Yes: 2130
##                                                    Median :3.700
##                                                    Mean   :3.602
##                                                    3rd Qu.:3.900
##                                                    Max.   :4.500
##                                                    NA's   :21
##  language_course english_test work_experience job_train  license
##  No :11825       No :8342     No :4591        No :9622   No :5650
##  Yes: 1506       Yes:4989     Yes:8740        Yes:3709   Yes:7681
##
##
##
##
##
##  job_preparation       father_edu         mother_edu     parent_income
##  Min.   :0.000   High School:7704   High School:9666   over 500 :4135
##  1st Qu.:1.000   University :5627   University :3665   under 500:9196
##  Median :2.000
##  Mean   :2.534
##  3rd Qu.:4.000
##  Max.   :8.000
##
##                major      employ
##  Arts and Physical:1427   no : 1440
##  Education        : 717   yes:11891
##  Engineering      :4038
##  Humanity         :1553
##  Medical          : 995
##  Nature           :1829
##  Social           :2772
```

Sometimes you need to explicitly convert integer to factor. To do this, you use the functions factor().

```
employ_all$job_preparation <- factor(employ_all$job_preparation)
```

The tidyverse provides us with a powerful and flexible package for visualizing data called ggplot2. The functions provided by ggplot2 follow a principle and consistent syntax known as the grammar of graphics. For readers who are interested in an in-depth explanation of ggplot2 and the grammar of graphics, we refer you to the books ggplot2 by Hadley Wickham and The Grammar of Graphics by Leland Wilkinson.

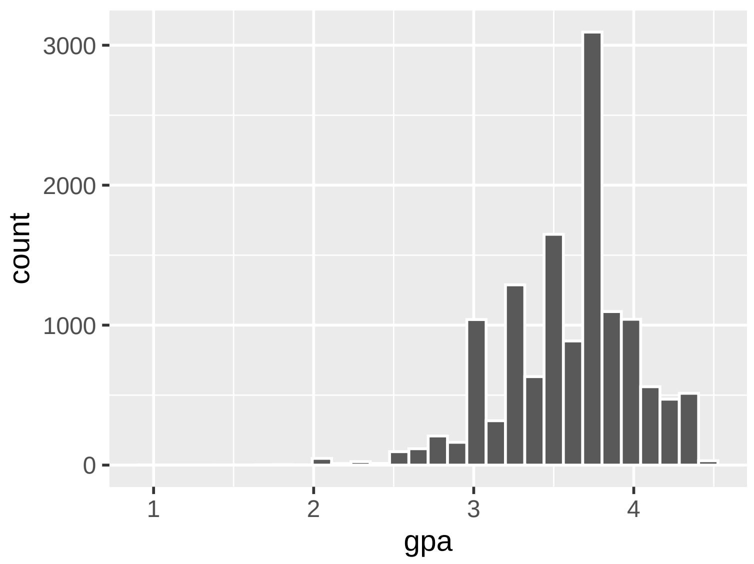
```
employ_all %>%
  ggplot(aes(employ)) +
  geom_bar(color="white")
```



```
employ_all %>%
  ggplot(aes(gpa)) +
  geom_histogram(color="white")
```

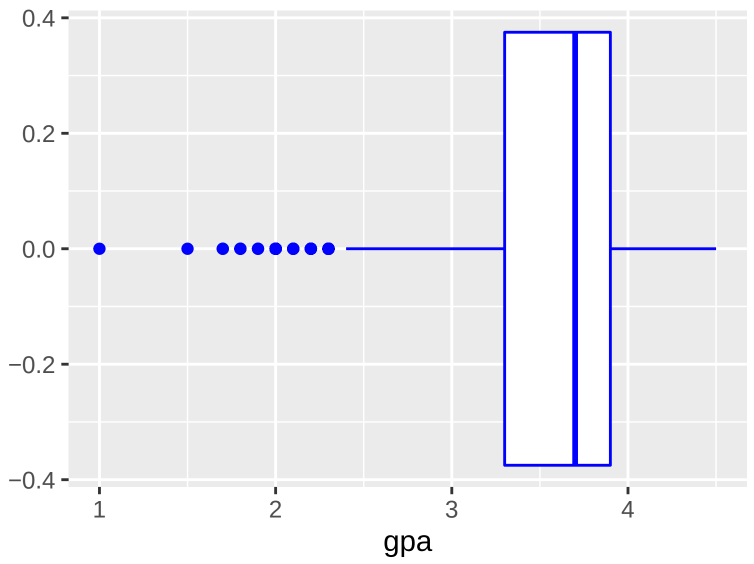
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 21 rows containing non-finite values (stat_bin).
```



```
employ_all %>%
  ggplot(aes(gpa)) +
  geom_boxplot(color="blue")
```

```
## Warning: Removed 21 rows containing non-finite values (stat_boxplot).
```



1.6 Dealing with the missing data

For continuous features, the most commonly used approach for dealing with missing values is the mean or median imputation approach. As the name suggests, the approach involves the use of the mean or median of the observed values as a substitute for the missing value.

```
employ_all$gpa <- ifelse(is.na(employ_all$gpa),
                         mean(employ_all$gpa, na.rm=TRUE),
                         employ_all$gpa)
```

1.7 Splitting the data (train set & test set)

The `createDataPartition()` function in **caret** package creates stratified random samples from the original data and takes three main arguments. The first argument (`y`) specifies the class or dependent variable, the second argument (`p`) specifies the proportion of examples that should be assigned to the training set, and the third argument (list) specifies the format of the results that are returned. This argument can be either `TRUE` or `FALSE`. If it is `TRUE`, then the results of the function are returned as a list (single row), but if it is `FALSE`, then the results are returned as a matrix (several rows). Note that we use the `set.seed()` function here again. By setting the seed value, we ensure that we get the same data partitions every time we run the code.

```
# Create a vector of indices which is an 80% random sample
set.seed(1234)
index <- createDataPartition(employ_all$employ, p = 0.8, list = FALSE)

employ_train <- employ_all[index, ]
prop.table(table(employ_train$employ))
```

```
##
##        no       yes
## 0.1080169 0.8919831
```

```
employ_test <- employ_all[-index, ]
prop.table(table(employ_test$employ))
```

```
##
##       no      yes
## 0.108027 0.891973
```

1.8 Dealing with Class Imbalance

Class imbalance is a common problem when working with real-world data. It degrades the performance of a machine learning model because it biases the model toward the majority class at the expense of the minority class. Before we build a model, we need to address this problem. There are several approaches to solving class imbalance problems; one of them is by using a synthetic minority oversampling technique (SMOTE).

This technique works by creating new synthetic samples from the minority class to resolve the imbalance. In R, the **DMwR** package provides us with a function called `SMOTE()`, which we can use to balance our training data. The `SMOTE()` function accepts arguments that describe the formula for the prediction problem, the data frame containing the original imbalanced data, a specification of how many extra cases from the minority class are to be generated (`perc.over`), and a specification for how many extra cases from the majority class should be selected for each case generated from the minority class (`perc.under`). The output of the function is a data frame of the balanced data.

```
set.seed(1234)
employ_train <- SMOTE(employ ~ ., data.frame(employ_train),
                      perc.over = 100, perc.under = 200)
```

1.9 Resampling methods to estimate model accuracy

When you are building a predictive model, you need to evaluate the capability of the model on unseen data. This is typically done by estimating accuracy using data that was not used to train the model. The caret package in R provides a number of methods to estimate the accuracy of a machine learning algorithm. In this section you will discover 5 approaches for estimating model performance on unseen data.

1.9.1 Estimating model accuracy

When working on a project you often only have a limited set of data and you need to choose carefully how you use it. Predictive models required data on which to train. You also need a dataset that the model has not seen during training on which it can make predictions. The accuracy of the model predictions on data unseen during training can be used as an estimate for the accuracy of the model on unseen data in general.

You cannot estimate the accuracy of the model on the data used to train it. An ideal model could just remember all of the training instances and make perfect predictions. You must hold data aside primarily for the purposes of model evaluation.

There are methods that you can use if you have a lot of data and do not need to be careful about how it is spent during training. More commonly your dataset has a fixed size and you need to use statistical techniques that make good use of a limited size dataset. These statistical methods are often called resampling methods as they take multiple samples or make multiple splits of your dataset into portions that you can use for model training and model testing. In the following sections you are going to discover how to use five different resampling methods that you can use to evaluate the accuracy of your data in R.

1.9.2 k-fold cross validation

Of all the approaches to cross-validation, the most commonly used is k-fold cross-validation. In this approach, after the test data has been sequestered, the remaining data is divided into k completely separate random partitions of approximately equal size. These partitions are known as folds. The folds represent the data that will be used to validate the model during each of the k iterations of the repeated holdout. Although k can be set to any value, in practice, k is often set to either 5 or 10.

```
ctrl_cv <- trainControl(method = "cv", number = 5)
set.seed(1234)
model_rpart_cv <- train(employ ~ .,
                          data = employ_train,
                          metric = "Accuracy",
                          method = "rpart",
                          trControl = ctrl_cv)
```

1.9.3 repeated k-fold cross validation

Repeated k-fold is the most preferred cross-validation technique for both classification and regression machine learning models. Shuffling and random sampling of the data set multiple times is the core procedure of repeated k-fold algorithm and it results in making a robust model as it covers the maximum training and testing operations. The working of this cross-validation technique to evaluate the accuracy of a machine learning model depends upon 2 parameters. The first parameter is k which is an integer value and it states that the given dataset will be split into K folds(or subsets). Among the k folds, the model is trained on the k-1 subsets and the remaining subset will be used to evaluate the model's performance. These steps will be repeated up to a certain number of times which will be decided by the second parameter of this algorithm and thus it got its name as Repeated k-fold i.e., the k-fold cross-validation algorithm is repeated a certain number of times.

```
ctrl_repeatedcv <- trainControl(method = "repeatedcv",
                                number = 10, repeats = 3)
set.seed(1234)
model_rpart_repeatedcv <- train(employ ~ .,
                        data = employ_train,
                        metric = "Accuracy",
                        method = "rpart",
                        trControl = ctrl_repeatedcv)
```

1.9.4 leave one out cross validation

Another common approach to cross-validation is the leave-one-out cross-validation method (LOOCV). The approach is essentially k-fold cross-validation with k set to n (the number of instances in the dataset). In the LOOCV approach, during the first iteration, the first instance is held out for validation while the rest of the data is used to train the model. Then the performance of the model is evaluated against the single instance that was held out. This process is repeated n-1 additional times until all the instances in the dataset have been used once for validation. After the last iteration, we end up with n estimates of the model's performance from each of the iterations. The average of these estimates is used as the LOOCV estimate of model performance.

```
ctrl_loocv <- trainControl(method = "LOOCV")
set.seed(1234)
model_rpart_loocv <- train(employ ~ .,
                           data = employ_train,
                           metric = "Accuracy",
                           method = "rpart",
                           trControl = ctrl_loocv)
```

1.9.5 leave group out cross validation

This approach is similar to k-fold cross-validation but with one notable difference. In this approach, instead of creating a set number of folds (validation sets) at the beginning of the process, as we do in k-fold cross-validation, the random sample that makes up the validation set is created during each iteration During the first iteration, a random sampling without replacement approach is used to create the validation set. This dataset is held out for validation, and the remainder of the data is used to train the model. In the second iteration, a new independent validation set is randomly selected. Because of the random nature of the sampling approach, it is possible and likely that some of the instances selected as part of this new validation set were also selected as part of the validation set in the previous iteration.

```
ctrl_lgocv <- trainControl(method = "LGOCV", p = .1, number = 10)
set.seed(1234)
model_rpart_lgocv <- train(employ ~ .,
                           data = employ_train,
                           metric = "Accuracy",
                           method = "rpart",
                           trControl = ctrl_lgocv)
```

1.9.6 bootstrap

The basic idea behind bootstrap sampling is to create a training dataset from the original data using a random sampling with replacement approach. A version of this technique, known as the 0.632 bootstrap, involves random sampling a dataset with n instances, n different times with replacement, to create another dataset also with n instances. This new dataset is used for training, while the instances from the original data, which were not selected as part of the training data, are used for validation.

```r
ctrl_boot <- trainControl(method = "boot632", number = 3)
set.seed(1234)
model_rpart_boot <- train(employ ~ .,
                          data = employ_train,
                          metric = "Accuracy",
                          method = "rpart",
                          trControl = ctrl_boot)
```

1.10 Spot-Check Machine Learning Algorithms

What algorithm should you use on your dataset? This is the most common question in applied machine learning. It's a question that can only be answered by trial and error, or what I call: spot-checking algorithms. In this session you will discover how to spot-check algorithms on a dataset using R.

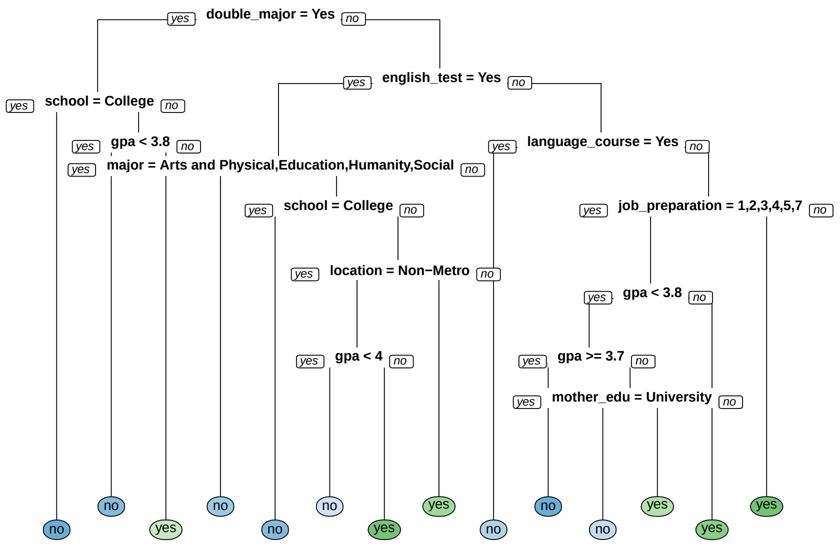
1.10.1 CART Algorithm

The CART algorithm is implemented in R as part of the rpart package. This package provides a similarly named function `rpart()`, which provides an implementation of CART (Classification And Regression Trees) for classification and regression. This function takes three primary arguments. The first is the prediction formula, which we specify as `employ ~ .` to mean that our model should use all the other variables in the dataset as predictors for the `employ` variable. The second argument is the method, which we specify as `class`. This means that we are building a classification tree. The final argument is the training dataset that will be used to build the model.

```r
# Building a CART model
model_rpart <- rpart(employ ~ .,method = "class",
                     data = employ_train,
                     control = rpart.control(maxdepth = 7,
                                             cp = 0.005))
```

Now that we've trained our decision tree model, let's visualize it. To do so, we use the `rpart.plot()` function from the similarly named rpart.plot package.

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



1.10.2 Bagging Algorithm

One of the most common ensemble learning approaches is known as bagging, which stands for bootstrap aggregating. The name comes from the fact that bagging ensembles use a bootstrap sampling approach for the allocation function, which is used to generate the data assigned to each model in the ensemble. Bagging ensembles are typically made up of homogenous learners, which are trained independently and in parallel.

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).

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

```
model_bagging <- bagging(formula = employ ~ .,
                         data = employ_train,
                         coob = TRUE)
```

1.10.3 randomforest Algorithm

One of the most popular bagging ensemble methods is the random forests or decision tree forests ensemble technique. It gets its name from the fact that the ensemble consists of a large number of decision tree learners (which are collectively called a forest) and that its allocation function combines both bootstrap sampling and random feature selection to generate the data assigned to each learner in the ensemble. By using only a random subset of the full feature set, random forests are able to handle very wide datasets (datasets with a large number of features). To illustrate the random forests ensemble technique in R, we use the `rf` method in caret, which depends on the aptly named randomForest package. Using the `modelLookup()` command for the `rf` method reveals that it has only one tunable parameter: `mtry`. This is the number of randomly selected features to consider at each split (more on this shortly).

```
modelLookup("rf")
```

```
##   model parameter                         label forReg forClass probModel
## 1    rf      mtry #Randomly Selected Predictors   TRUE     TRUE      TRUE
```

Based on the documentation provided by the randomForest package, the default value for mtry is the square root of the number of features in the dataset when working on a classification problem.

```
# Building a Random Forest model
set.seed(123)  # for reproducibility
model_rf <- randomForest(formula = employ ~ .,
                         data = employ_train)
```

1.10.4 gbm Algorithm

Boosting is a method of converting weak learners into strong learners. In boosting, each new tree is a fit on a modified version of the original data set. The gradient boosting algorithm (gbm) can be most easily explained by first introducing the AdaBoost Algorithm.The AdaBoost Algorithm begins by training a decision tree in which each observation is assigned an equal weight. After evaluating the first tree, we increase the weights of those observations that are difficult to classify and lower the weights for those that are easy to classify. The second tree is therefore grown on this weighted data. Here, the idea is to improve upon the predictions of the first tree. Our new model is therefore Tree 1 + Tree 2. We then compute the classification error from this new 2-tree ensemble model and grow a third tree to predict the revised residuals. We repeat this process for a specified number of iterations. Subsequent trees help us to classify observations that are not well classified by the previous trees. Predictions of the final ensemble model is therefore the weighted sum of the predictions made by the previous tree models. In order to train a `gbm` model in R, you will first have to install and call the gbm library. The gbm function requires you to specify certain arguments. You will begin by specifying the formula. This will include your response and predictor variables. Next, you will specify the distribution of your response variable. If nothing is specified, then gbm will try to guess. Some commonly used distributions include- “`bernoulli`” (logistic regression for 0–1 outcome), “`gaussian`” (squared errors), “`tdist`”(t-distribution loss), and “`poisson`” (count outcomes). Finally, we will specify the data and the n.trees argument (after all gbm is an ensemble of trees!) By default, the gbm model will assume 100 trees, which can provide is a good estimate of our gbm’s performance.

```
# Building a 10000-tree GBM model
# Convert "yes" to 1, "no" to 0
gbm_train <- employ_train
gbm_train$employ <- ifelse(gbm_train$employ == "no", 1, 0)

set.seed(123)
model_gbm <- gbm(formula = employ ~ .,
                 distribution = "bernoulli",
                 data = gbm_train,
                 n.trees = 10000)
```

1.10.5 logistic model

you used the `glm()` function to build a logistic regression model. As with many of R’s machine learning methods, you can apply the `predict()` function to the model object to forecast future behavior. By default, `predict()` outputs predictions in terms of log odds unless `type = "response"` is specified. This converts the log odds to probabilities.

```
model_logistic <- glm(employ ~. ,data = employ_train,
                      family="binomial")
```

1.11 Model Evaluation Metrics

What metrics can you use to evaluate your machine learning algorithms? In this lesson you will discover how you can evaluate your machine learning algorithms in R using a number of standard evaluation metrics. After completing this lesson you will know:

- How to use Accuracy and Kappa to evaluate model skill on classification problems.

- How to use Area Under ROC Curve, sensitivity and specificity to evaluate model skill on binary classification problems.

1.11.1 Accuracy and Kappa

Accuracy and Kappa are the default metrics used to evaluate algorithms on binary and multiclass classification datasets in caret. Accuracy is the percentage of correctly classified instances out of all instances. It is more useful on a binary classification than multi-class classification problem because it can be less clear exactly how the accuracy breaks down across those classes. Kappa or Cohen's Kappa is like classification accuracy, except that it is normalized at the baseline of random chance on your dataset. It is a more useful measure to use on problems that have an imbalance in the classes (e.g. a 70% to 30% split for classes 0 and 1 and you can achieve 70% accuracy by predicting all instances are for class 0).

```
# Accuracy and Kappa
ctrl_cv <- trainControl(method = "cv", number = 5)
set.seed(1234)
model_glm_cv_accuracy <- train(employ ~ .,
                              data = employ_train,
                              metric = "Accuracy",
                              method = "glm", family = "binomial",
                              trControl = ctrl_cv)
```

1.11.2 Area Under ROC Curve

ROC metrics are only suitable for binary classification problems (e.g. two classes). To calculate ROC information, you must change the summaryFunction in your trainControl to be twoClassSummary. This will calculate the Area Under ROC Curve (AUROC) also called just Area Under curve (AUC), sensitivity and specificity. ROC is actually the area under the ROC curve or AUC. The AUC represents a models ability to discriminate between positive and negative classes. An area of 1.0 represents a model that predicts perfectly. An area of 0.5 represents a model as good as random.

- Sensitivity is the true positive rate also called the recall. It is the number of instances from the positive (first) class that actually predicted correctly.
- Specificity is also called the true negative rate. Is the number of instances from the negative class (second class) that were actually predicted correctly.

ROC can be broken down into sensitivity and specificity. A binary classification problem is really a trade-off between sensitivity and specificity.

```
# Area Under ROC Curve

ctrl_cv <- trainControl(method = "cv", number = 5, classProbs = TRUE,
                        summaryFunction = twoClassSummary)
set.seed(1234)
model_glm_cv_roc <- train(employ ~ .,
                      data = employ_train,
                      metric = "ROC",
                      method = "glm", family = "binomial",
                      trControl = ctrl_cv)
```

1.12 The Performance of Machine Learning Algorithms

How do you compare the estimated accuracy of different machine learning algorithms effectively? In this lesson you will discover techniques that you can use to compare machine learning algorithms in R. it's time to compare their out-of-sample predictions and choose which one is the best model for your dataset. You can compare models in caret using the `resamples()` function, provided they have the same training data and use the same `trainControl` object with preset cross-validation folds. `resamples()` takes as input a list of models and can be used to compare dozens of models at once (though in this case you are only comparing five models).

```
# Create custom indices: myFolds
set.seed(1234)
myFolds <- createFolds(employ_train$employ, k = 5)

# Create reusable trainControl object: myControl
myControl <- trainControl(
  summaryFunction = twoClassSummary,
  classProbs = TRUE, # IMPORTANT!
  verboseIter = FALSE,
  savePredictions = TRUE,
  index = myFolds
)

model_caret_rpart <- train(
  employ ~ .,
  data = employ_train,
  metric = "ROC",
  method = "rpart",
  trControl = myControl)

model_caret_bagging <- train(
  employ ~ .,
  data = employ_train,
  metric = "ROC",
  method = "treebag",
  trControl = myControl)

model_caret_ranger <- train(
  employ ~ .,
  data = employ_train,
  metric = "ROC",
  method = "ranger",
  trControl = myControl)

model_caret_logistic <- train(
  employ ~ .,
  data = employ_train,
  metric = "ROC",
  method = "glm", family = "binomial",
  trControl = myControl)

model_caret_gbm <- train(
  employ ~ .,
  data = employ_train,
  metric = "ROC",
  method = "gbm",
  trControl = myControl)
```

```
# Create model_list
model_list <- list(rpart = model_caret_rpart,
                   rf = model_caret_ranger,
                   logistic = model_caret_logistic,
                   bagging = model_caret_bagging,
                   gbm = model_caret_gbm)
```

caret provides a variety of methods to use for comparing models. All of these methods are based on the `resamples()` function. The box-and-whisker plot allows you to compare the distribution of predictive accuracy (in this case AUC) for the two or more models.

In general, you want the model with the higher median AUC, as well as a smaller range between min and max AUC.

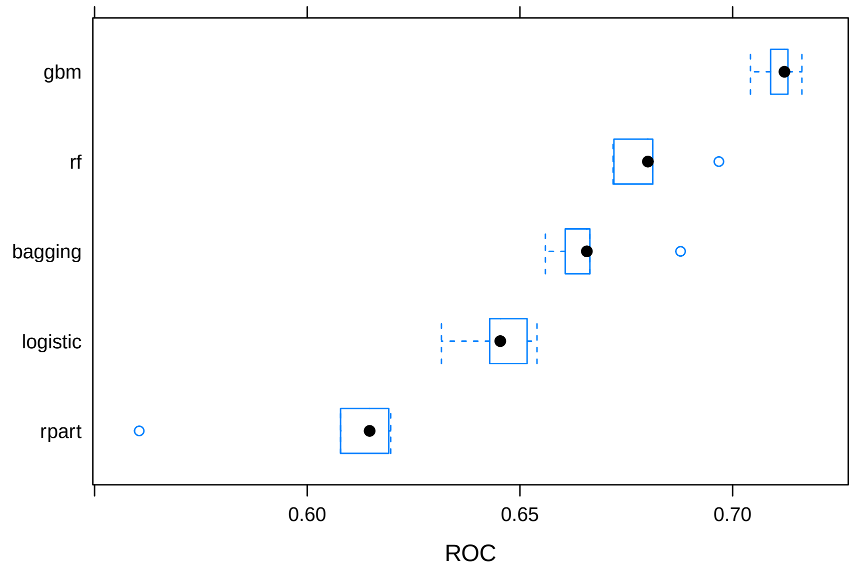
You can make this plot using the `bwplot()` function, which makes a box and whisker plot of the model's out of sample scores. Box and whisker plots show the median of each distribution as a line and the interquartile range of each distribution as a box around the median line. You can pass the metric = "ROC" argument to the `bwplot()` function to show a plot of the model's out-of-sample ROC scores and choose the model with the highest median ROC.

```r
# Pass model_list to resamples(): resamples
resamps <- resamples(model_list)

# Summarize the results
summary(resamps)
```

```
##
## Call:
## summary.resamples(object = resamps)
##
## Models: rpart, rf, logistic, bagging, gbm
## Number of resamples: 5
##
## ROC
##               Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## rpart    0.5604992 0.6078309 0.6146635 0.6043555 0.6191619 0.6196219    0
## rf       0.6719003 0.6720966 0.6800829 0.6804170 0.6812313 0.6967741    0
## logistic 0.6315494 0.6428938 0.6453851 0.6451043 0.6516840 0.6540091    0
## bagging  0.6559887 0.6606517 0.6657322 0.6673172 0.6664473 0.6877659    0
## gbm      0.7041982 0.7089350 0.7121882 0.7109240 0.7130095 0.7162891    0
##
## Sens
##               Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## rpart    0.4074878 0.4831887 0.4948454 0.5499204 0.6630494 0.7010309    0
## rf       0.5838307 0.6073753 0.6180141 0.6179482 0.6283234 0.6521975    0
## logistic 0.5355399 0.5849159 0.6001085 0.5865863 0.6055345 0.6068330    0
## bagging  0.5556158 0.5789474 0.6117137 0.6029721 0.6245252 0.6440586    0
## gbm      0.5952252 0.6245252 0.6250678 0.6273867 0.6317787 0.6603364    0
##
## Spec
##               Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## rpart    0.4199674 0.5287419 0.6988606 0.6283745 0.7075421 0.7867607    0
## rf       0.6160521 0.6250678 0.6342919 0.6394340 0.6500271 0.6717309    0
## logistic 0.6011937 0.6122560 0.6185567 0.6216373 0.6310364 0.6451438    0
## bagging  0.6209328 0.6256104 0.6266956 0.6363949 0.6299512 0.6787846    0
## gbm      0.6538253 0.6722735 0.6820402 0.6859803 0.6919740 0.7297884    0
```

```r
# Create bwplot
bwplot(resamps, metric = "ROC")
```



1.13 Improving Model Performance

It is difficult to find a good machine learning algorithm for your problem. But once you do, how do you get the best performance out of it. In this session you will discover three ways that you can tune the parameters of a machine learning algorithm in R. Most machine learning techniques have one or more parameters that need to be set before the learning process begins. These parameters are commonly known as hyperparameters.

1.13.1 Random Search

One search strategy that we can use is to try random values within a range. This can be good if we are unsure of what the value might be and we want to overcome any biases we may have for setting the parameter. To use random search, another option is available in `trainControl` called `search`. Possible values of this argument are `"grid"` and `"random"`. The built-in models contained in caret contain code to generate random tuning parameter combinations. The total number of unique combinations is specified by the tuneLength option to train.

```
# random search

ctrl_random <- trainControl(method = "repeatedcv" , number=5,
                             repeats = 3,
                            search = "random")
```

```r
set.seed(1234)
gbm_random <- train(employ ~ ., data=employ_train, method="gbm",
                  metric = "Accuracy",
                  tuneLength = 5, verbose = FALSE,
                  trControl = ctrl_random)
```

1.13.2 Grid Search

Another search you can use is to define a grid of algorithm parameters to try. Each axis of the grid is an algorithm parameter, and points in the grid are specific combinations of parameters. Because we are only tuning one parameter, the grid search is a linear search through a vector of candidate values.

```r
# grid search
ctrl_grid <- trainControl(method = "repeatedcv" , number = 5,
                          repeats = 3,
                          search = "grid")
set.seed(1234)
tune_grid <- expand.grid(n.trees = c(100, 200, 250),
                        interaction.depth = c(1, 4, 6),
                        shrinkage = 0.1,
                        n.minobsinnode = 10)
gbm_grid <- train(employ ~ ., data = employ_train, method = "gbm",
                  metric = "Accuracy", tuneGrid = tune_grid,
                  verbose = FALSE,
                  trControl = ctrl_grid)
```

1.13.3 Adaptive Resampling

Models can benefit significantly from tuning but the optimal values are rarely known beforehand. train can be used to define a grid of possible points and resampling can be used to generate good estimates of performance for each tuning parameter combination. However, in the nominal resampling process, all the tuning parameter combinations are computed for all the resamples before a choice is made about which parameters are good and which are poor. caret contains the ability to adaptively resample the tuning parameter grid in a way that concentrates on values that are the in the neighborhood of the optimal settings. To use the adaptive procedure, the trainControl option needs some additional arguments: - min is the minimum number of resamples that will be used for each tuning parameter. The default value is 5 and increasing it will decrease the speed-up generated by adaptive resampling but should also increase the likelihood of finding a good model. - alpha is a confidence level that is used to remove parameter settings. To date, this value has not shown much of an effect. - method is either "`gls`" for a linear model or "`BT`" for a Bradley-Terry model. The latter may be more useful when you expect the model to do very well (e.g. an area under the ROC curve near 1) or when there are a large number of tuning parameter settings. - complete is a logical value that specifies whether train should generate the full resampling set if it finds an optimal solution before the end of resampling. If you want to know the optimal parameter settings and don't care much for the estimated performance value, a value of FALSE would be appropriate here

```
# adaptive resampling
fitControl <- trainControl(method = "adaptive_cv",
                           number = 3, repeats = 3,
                           adaptive = list(min = 2,
                                           alpha = 0.05,
                                           method = "gls",
                                           complete = TRUE),
                           search = "random")

set.seed(1234)
gbm_adaptive <- train(employ ~ ., data = employ_train,
                      method = "gbm",
                  trControl = fitControl,
                  verbose = FALSE,
                  tuneLength = 7)
```

```
## ------------------------------------------------------------------------------

## You have loaded plyr after dplyr - this is likely to cause problems.
## If you need functions from both plyr and dplyr, please load plyr first, then dplyr:
## library(plyr); library(dplyr)

## ------------------------------------------------------------------------------

##
## Attaching package: 'plyr'

## The following object is masked from 'package:DMwR':
##
##     join

## The following objects are masked from 'package:dplyr':
##
##     arrange, count, desc, failwith, id, mutate, rename, summarise,
##     summarize

## The following object is masked from 'package:purrr':
##
##     compact
```

```
gbm_adaptive
```

```
## Stochastic Gradient Boosting
```

```
##
## 4608 samples
##   15 predictor
##    2 classes: 'no', 'yes'
##
## No pre-processing
## Resampling: Adaptively Cross-Validated (3 fold, repeated 3 times)
## Summary of sample sizes: 3072, 3072, 3072, 3072, 3072, 3072, ...
## Resampling results across tuning parameters:
##
##   shrinkage   interaction.depth  n.minobsinnode  n.trees  Accuracy   Kappa
##   0.02495755   6                  8              3330     0.7039931  0.4079861
##   0.09626856  10                  7              1163     0.6972656  0.3945312
##   0.11284695   4                  8              3117     0.6976997  0.3953993
##   0.14010332   2                 25              4304     0.6982422  0.3964844
##   0.16082565   6                  8              3046     0.6855469  0.3710938
##   0.18231333   6                 24                48     0.6842448  0.3684896
##   0.19065086   7                 12              3201     0.6861979  0.3723958
##   Resamples
##   9
##   3
##   3
##   6
##   2
##   2
##   2
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were n.trees = 3330, interaction.depth =
##  6, shrinkage = 0.02495755 and n.minobsinnode = 8.
```

1.14 Finalize Your Machine Learning Model

Once you have an accurate model on your test harness you are nearly done. But not yet. There are still a number of tasks to do to finalize your model. The whole idea of creating an accurate model for your dataset was to make predictions on unseen data. There are three tasks you may be concerned with:

- Making new predictions on unseen data.
- Creating a standalone model using all training data.
- Saving your model to file for later loading and making predictions on new data.

```
# Finalize Your Machine Learning Model

finalModel <- gbm(employ ~ .,
```

```r
  distribution = "bernoulli",
  data = gbm_train,
  n.trees = 3330,
  interaction.depth = 6,
  n.minobsinnode = 8,
  shrinkage = 0.02495755,
  verbose = FALSE,
  )
final_predictions <- predict(finalModel, employ_test,
                             type = "response")
```

```
## Using 3330 trees...
```

```r
final_class <- factor(ifelse(final_predictions >= 0.5,
                             "no","yes"),
                       levels=levels(employ_test[["employ"]]))
confusionMatrix(final_class, employ_test$employ, positive= "no")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   no  yes
##        no   107  609
##        yes  181 1769
##
##                Accuracy : 0.7037
##                  95% CI : (0.6859, 0.721)
##     No Information Rate : 0.892
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : 0.0698
##
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.37153
##             Specificity : 0.74390
##          Pos Pred Value : 0.14944
##          Neg Pred Value : 0.90718
##              Prevalence : 0.10803
##          Detection Rate : 0.04014
##    Detection Prevalence : 0.26857
##       Balanced Accuracy : 0.55772
##
##        'Positive' Class : no
##
```

Chapter 2

A Study on Job Satisfaction and Turnover Behavior

2.1 Introduction

Job satisfaction impacts on the turnover intention of employee, which affects the turnover behavior. This analysis concerns with the impact of job satisfaction on the turnover behavior. Since turnover intention is highly correlated with job satisfaction, salary, employment status and etc, we should pay careful attention for modeling of those variables as independent variables and turnover behavior as a dependent variable in the empirical study for the impact of factors on turnover behavior.

2.2 Data Explanation

data filename : satisfy.csv

variable	description
turnover	Whether you are preparing for job change
work_status	Employment status
work_hour_type	Working Hour Form
overtime	Overtime working per week
satisfaction_1	Wages or income
satisfaction_2	Employment stability
satisfaction_3	Contents of work (job content)
satisfaction_4	Working environment (facilities, safety, hygiene, etc.)
satisfaction_5	Working hours (working hours, number of working days)
satisfaction_6	Personal development potential
satisfaction_7	Relationships
satisfaction_8	Welfare benefits system (social insurance and additional benefits)
satisfaction_9	Personnel system (promotion system)
satisfaction_10	Social reputation for what you are doing
satisfaction_11	Autonomy and authority in what you are doing
satisfaction_12	Social reputation for work
satisfaction_13	The match between what you do and your aptitude and interest
satisfaction_14	Job-related education or training
satisfaction_job	Overall satisfaction with the current job (job)
satisfaction_task	Overall satisfaction with the main job (task)
match_1	Education level-level of match at work level
match_2	Work skill (skill) level-the degree to which one's skill (skill) level matches
match_3	Contents of work-Degree of coincidence with your major (major) at the time of ac
major_knowledge	Degree of help in work of knowledge in college major

We do not add prompts to R source code in this manual, and we comment out the text output by two hashes ## by default. Package names are in bold text (e.g., **rpart**), function names are formatted as code (e.g., `read.csv()`).

2.3 Import Data

We will use the built in `read.csv()` function call, which reads the data in as a data frame, and assign the data frame to a variable (using `<-`) so that it is stored in R's memory. Then we will explore some of the basic arguments that can be supplied to the function. The default for `read.csv()` is to set the `header` argument to TRUE. This means that the first row of values in the .csv is set as header information (column names). If your data set does not have a header, set the `header` argument to FALSE. The `stringsAsFactors` argument is perhaps the most important argument in `read.csv()`, particularly if you are working with categorical data. This is because the default behavior of R is to convert character strings into factors, which may make it difficult to do such things as replace values.

```
satisfy_all <- read.csv(file="satisfy.csv",header=TRUE,stringsAsFactors = TRUE)
```

R is always pointed at a directory on your computer. You can find out which directory by running the `getwd()` (get working directory) function; this function has no arguments. To change your working directory, use `setwd()` and specify the path to the desired folder.

- `getwd()`
- `setwd(dir)` -dir – Specify a working directory. This may be from the root directory (starting with / on a Mac), it may include a double-dot (..) to move locally up a folder from the current directory, and it may include a path from the current directory.

```
getwd()
```

```
## [1] "C:/Users/kimin/OneDrive/project/bookdown-demo-master"
```

```
setwd("c:/project")
```

2.4 Exploring and Preparing the Data

The first questions to ask in an investigation of a new dataset should be about how the dataset is organized. If you are fortunate, your source will provide a data dictionary, a document that describes the dataset's features. In our case, the used car data does not come with this documentation, so we'll need to create our own. The `str()` function provides a method for displaying the structure of R objects, such as data frames, vectors, or lists. It can be used to create the basic outline for our data dictionary:

```
str(satisfy_all)
```

```
## 'data.frame':    10839 obs. of  24 variables:
##  $ turnover         : Factor w/ 2 levels "No","Yes": 1 1 1 1 1 1 2 2 2 2 ...
##  $ work_status      : Factor w/ 3 levels "Daily","Regular",..: 3 2 2 2 2 2 1 2 2 3 ...
##  $ work_hour_type   : Factor w/ 2 levels "Full Time","Part Time": 1 1 1 1 1 1 2 1 1 1 ...
##  $ overtime         : int  0 5 5 2 0 0 4 10 0 2 ...
##  $ satisfaction_1   : int  1 2 3 4 3 3 4 3 3 2 ...
##  $ satisfaction_2   : int  4 4 4 4 3 4 3 3 5 4 ...
##  $ satisfaction_3   : int  5 5 4 4 3 4 3 4 3 4 ...
##  $ satisfaction_4   : int  5 4 3 2 4 4 3 3 4 5 ...
##  $ satisfaction_5   : int  3 4 3 4 3 5 3 3 5 4 ...
##  $ satisfaction_6   : int  5 3 3 4 4 4 2 4 3 4 ...
##  $ satisfaction_7   : int  2 4 4 4 3 4 2 3 5 4 ...
##  $ satisfaction_8   : int  1 4 3 3 3 4 3 3 4 4 ...
##  $ satisfaction_9   : int  1 3 4 3 3 3 2 3 4 4 ...
##  $ satisfaction_10  : int  3 4 3 4 3 4 2 4 4 4 ...
##  $ satisfaction_11  : int  5 4 3 4 3 4 2 3 5 4 ...
##  $ satisfaction_12  : int  3 3 3 4 3 3 3 3 4 4 ...
##  $ satisfaction_13  : int  4 5 4 4 3 4 3 4 2 4 ...
##  $ satisfaction_14  : int  2 3 3 4 3 3 3 3 4 4 ...
##  $ satisfaction_job : int  3 4 3 4 3 4 3 3 3 5 ...
##  $ satisfaction_task: int  4 5 4 4 3 4 3 3 3 4 ...
##  $ match_1          : int  2 2 3 2 3 2 3 3 2 4 ...
##  $ match_2          : int  1 4 3 2 3 2 3 4 2 4 ...
##  $ match_3          : int  1 3 2 1 3 5 1 4 1 5 ...
##  $ major_knowledge  : int  1 4 2 2 3 4 2 4 1 5 ...
```

For such a simple command, we learn a wealth of information about the dataset. The statement `10839 obs` informs us that the data includes 10,839 observations, which is just another way of saying that the dataset contains 10,839 records or examples. The number of observations is often simply abbreviated as n. The `24 variables` statement refers to the sixteen features that were recorded in the data. These features are listed by name on separate lines. Looking at the line for the feature called `work_status`, we note some additional details:

```
work_status      : Factor w/ 3 levels "Daily","Regular",..: 3 2 2
2 2
```

After the variable's name, the Factor label tells us that the feature is factor type. In this dataset, 3 of the variables are factor while others are noted as int which refers to the integer type. In R, we can get summary statistics for a dataset by using the `summary()` function. To get the summary statistics for our satisfy dataset, we pass the name of the dataset to the `summary()` function.

```
summary(satisfy_all)
```

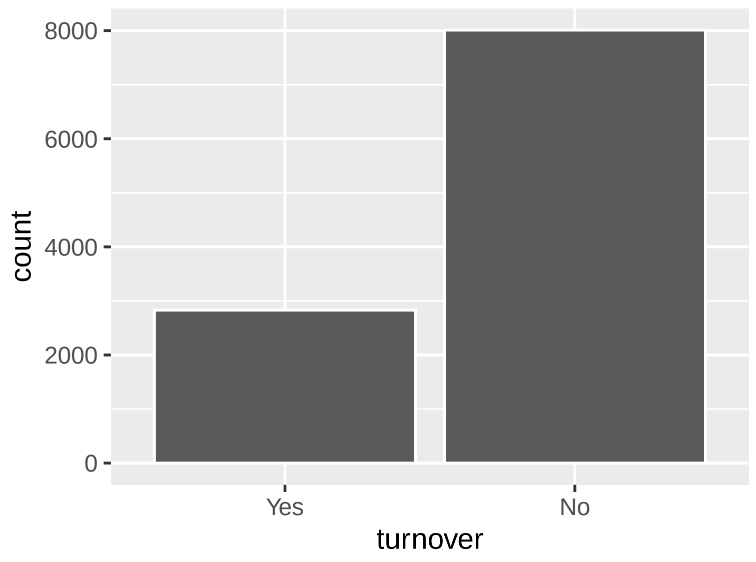
```
##  turnover      work_status     work_hour_type    overtime      satisfaction_1
##  No :8010   Daily    : 166   Full Time:9803   Min.   : 0.000   Min.   :1.000
##  Yes:2829   Regular  :8789   Part Time:1036   1st Qu.: 0.000   1st Qu.:3.000
##             Temporary:1884                    Median : 0.000   Median :3.000
##                                               Mean   : 3.739   Mean   :3.231
##                                               3rd Qu.: 5.000   3rd Qu.:4.000
##                                               Max.   :60.000   Max.   :5.000
##  satisfaction_2  satisfaction_3 satisfaction_4 satisfaction_5  satisfaction_6
##  Min.   :1.000   Min.   :1.00   Min.   :1.00   Min.   :1.000   Min.   :1.000
##  1st Qu.:3.000   1st Qu.:3.00   1st Qu.:3.00   1st Qu.:3.000   1st Qu.:3.000
##  Median :4.000   Median :4.00   Median :4.00   Median :4.000   Median :4.000
##  Mean   :3.739   Mean   :3.72   Mean   :3.82   Mean   :3.653   Mean   :3.503
##  3rd Qu.:4.000   3rd Qu.:4.00   3rd Qu.:4.00   3rd Qu.:4.000   3rd Qu.:4.000
##  Max.   :5.000   Max.   :5.00   Max.   :5.00   Max.   :5.000   Max.   :5.000
##  satisfaction_7  satisfaction_8  satisfaction_9  satisfaction_10
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.00
##  1st Qu.:3.000   1st Qu.:3.000   1st Qu.:3.000   1st Qu.:3.00
##  Median :4.000   Median :4.000   Median :3.000   Median :4.00
##  Mean   :3.831   Mean   :3.465   Mean   :3.207   Mean   :3.59
##  3rd Qu.:4.000   3rd Qu.:4.000   3rd Qu.:4.000   3rd Qu.:4.00
##  Max.   :5.000   Max.   :5.000   Max.   :5.000   Max.   :5.00
##  satisfaction_11 satisfaction_12 satisfaction_13 satisfaction_14
##  Min.   :1.000   Min.   :1.000   Min.   :1.000   Min.   :1.00
##  1st Qu.:3.000   1st Qu.:3.000   1st Qu.:3.000   1st Qu.:3.00
##  Median :4.000   Median :4.000   Median :4.000   Median :4.00
##  Mean   :3.658   Mean   :3.604   Mean   :3.672   Mean   :3.48
##  3rd Qu.:4.000   3rd Qu.:4.000   3rd Qu.:4.000   3rd Qu.:4.00
##  Max.   :5.000   Max.   :5.000   Max.   :5.000   Max.   :5.00
##  satisfaction_job satisfaction_task    match_1         match_2
##  Min.   :1.000    Min.   :1.000     Min.   :1.000   Min.   :1.000
##  1st Qu.:3.000    1st Qu.:3.000     1st Qu.:3.000   1st Qu.:3.000
##  Median :4.000    Median :4.000     Median :3.000   Median :3.000
##  Mean   :3.541    Mean   :3.619     Mean   :2.942   Mean   :2.959
##  3rd Qu.:4.000    3rd Qu.:4.000     3rd Qu.:3.000   3rd Qu.:3.000
##  Max.   :5.000    Max.   :5.000     Max.   :5.000   Max.   :5.000
##     match_3      major_knowledge
##  Min.   :1.000   Min.   :1.000
##  1st Qu.:2.000   1st Qu.:2.000
##  Median :3.000   Median :3.000
##  Mean   :3.175   Mean   :3.208
##  3rd Qu.:4.000   3rd Qu.:4.000
##  Max.   :5.000   Max.   :5.000
```

Changing the order of factor levels, You can use `fct_relevel()`. It takes a factor, f, and then any number of levels that you want to move to the front of the line.

```
satisfy_all$turnover <- fct_relevel(satisfy_all$turnover,"Yes","No")
satisfy_all$work_status <- fct_relevel(satisfy_all$work_status,"Regular","Temporary","Daily")
```

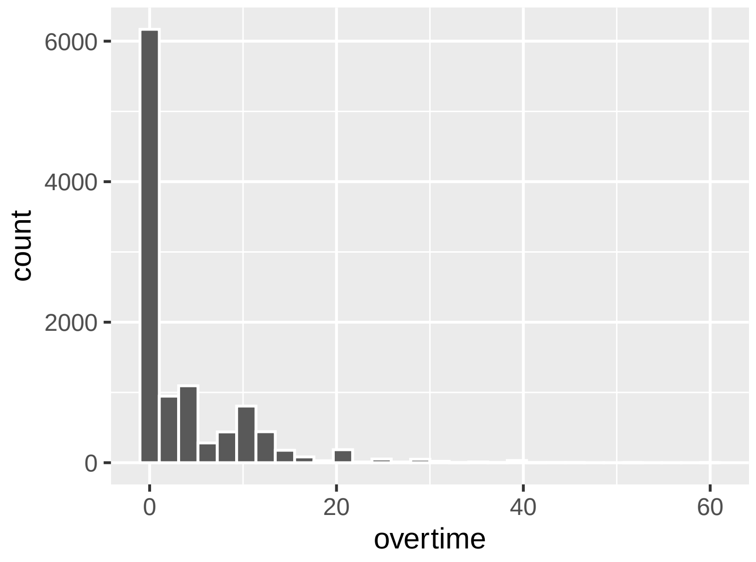
The tidyverse provides us with a powerful and flexible package for visualizing data called ggplot2. The functions provided by ggplot2 follow a principle and consistent syntax known as the grammar of graphics. For readers who are interested in an in-depth explanation of ggplot2 and the grammar of graphics, we refer you to the books ggplot2 by Hadley Wickham and The Grammar of Graphics by Leland Wilkinson.

```
satisfy_all %>%
  ggplot(aes(turnover)) +
  geom_bar(color="white")
```

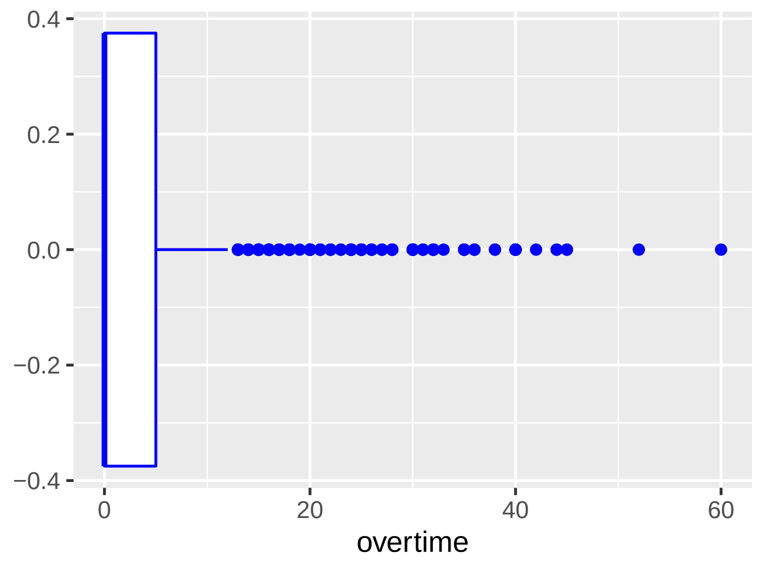


```
satisfy_all %>%
  ggplot(aes(overtime)) +
  geom_histogram(color="white")
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



```
satisfy_all %>%
  ggplot(aes(overtime)) +
  geom_boxplot(color="blue")
```



2.5 Splitting the data (train set & test set)

The `createDataPartition()` function in **caret** package creates stratified random samples from the original data and takes three main arguments. The first argument (`y`) specifies the class or dependent variable, the second argument (`p`) specifies the proportion of examples that should be assigned to the training set, and the third argument (list) specifies the format of the results that are returned. This argument can be either `TRUE` or `FALSE`. If it is `TRUE`, then the results of the function are returned as a list (single row), but if it is `FALSE`, then the results are returned as a matrix (several rows). Note that we use the `set.seed()` function here again. By setting the seed value, we ensure that we get the same data partitions every time we run the code.

```
# Create a vector of indices which is an 80% random sample
set.seed(1234)
index <- createDataPartition(satisfy_all$turnover, p = 0.8, list = FALSE)

satisfy_train <- satisfy_all[index, ]
prop.table(table(satisfy_train$turnover))
```

```
##
##       Yes        No
## 0.2610701 0.7389299
```

```
satisfy_test <- satisfy_all[-index, ]
prop.table(table(satisfy_test$turnover))
```

```
##
##       Yes        No
## 0.2607291 0.7392709
```

2.6 Dealing with Class Imbalance

Class imbalance is a common problem when working with real-world data. It degrades the performance of a machine learning model because it biases the model toward the majority class at the expense of the minority class. Before we build a model, we need to address this problem. There are several approaches to solving class imbalance problems; one of them is by using a synthetic minority oversampling technique (SMOTE).

This technique works by creating new synthetic samples from the minority class to resolve the imbalance. In R, the **DMwR** package provides us with a function called `SMOTE()`, which we can use to balance our training data. The `SMOTE()` function accepts arguments that describe the formula for the prediction problem, the data frame containing the original imbalanced data, a specification of how many extra cases from the minority class are to be generated (`perc.over`), and a specification for how many extra cases from the majority class should be selected for each case generated from the minority class (`perc.under`). The output of the function is a data frame of the balanced data.

```
set.seed(1234)
satisfy_train <- SMOTE(turnover ~ ., data.frame(satisfy_train),
                       perc.over = 100, perc.under = 200)
```

2.7 Resampling methods to estimate model accuracy

When you are building a predictive model, you need to evaluate the capability of the model on unseen data. This is typically done by estimating accuracy using data that was not used to train the model. The caret package in R provides a number of methods to estimate the accuracy of a machine learning algorithm. In this section you will discover 5 approaches for estimating model performance on unseen data.

2.7.1 Estimating model accuracy

When working on a project you often only have a limited set of data and you need to choose carefully how you use it. Predictive models required data on which to train. You also need a dataset that the model has not seen during training on which it can make predictions. The accuracy of the model predictions on data unseen during training can be used as an estimate for the accuracy of the model on unseen data in general.

You cannot estimate the accuracy of the model on the data used to train it. An ideal model could just remember all of the training instances and make perfect predictions. You must hold data aside primarily for the purposes of model evaluation.

There are methods that you can use if you have a lot of data and do not need to be careful about how it is spent during training. More commonly your dataset has a fixed size and you need to use statistical techniques that make good use of a limited size dataset. These statistical methods are often called resampling methods as they take multiple samples or make multiple splits of your dataset into portions that you can use for model training and model testing. In the following sections you are going to discover how to use five different resampling methods that you can use to evaluate the accuracy of your data in R.

2.7.2 k-fold cross validation

Of all the approaches to cross-validation, the most commonly used is k-fold cross-validation. In this approach, after the test data has been sequestered, the remaining data is divided into k completely separate random partitions of approximately equal size. These partitions are known as folds. The folds represent the data that will be used to validate the model during each of the k iterations of the repeated holdout. Although k can be set to any value, in practice, k is often set to either 5 or 10.

```
ctrl_cv <- trainControl(method = "cv", number = 5)
set.seed(1234)
model_rpart_cv <- train(turnover ~ .,
                        data = satisfy_train,
                        metric = "Accuracy",
                        method = "rpart",
                        trControl = ctrl_cv)
```

2.7.3 repeated k-fold cross validation

Repeated k-fold is the most preferred cross-validation technique for both classification and regression machine learning models. Shuffling and random sampling of the data set multiple times is the core procedure of repeated k-fold algorithm and it results in making a robust model as it covers the maximum training and testing operations. The working of this cross-validation technique to evaluate the accuracy of a machine learning model depends upon 2 parameters. The first parameter is k which is an integer value and it states that the given dataset will be split into K folds(or subsets). Among the k folds, the model is trained on the k-1 subsets and the remaining subset will be used to evaluate the model's performance. These steps will be repeated up to a certain number of times which will be decided by the second parameter of this algorithm and thus it got its name as Repeated k-fold i.e., the k-fold cross-validation algorithm is repeated a certain number of times.

```r
ctrl_repeatedcv <- trainControl(method = "repeatedcv",
                                number = 10, repeats = 3)
set.seed(1234)
model_rpart_repeatedcv <- train(turnover ~ .,
                        data = satisfy_train,
                        metric = "Accuracy",
                        method = "rpart",
                        trControl = ctrl_repeatedcv)
```

2.7.4 leave one out cross validation

Another common approach to cross-validation is the leave-one-out cross-validation method (LOOCV). The approach is essentially k-fold cross-validation with k set to n (the number of instances in the dataset). In the LOOCV approach, during the first iteration, the first instance is held out for validation while the rest of the data is used to train the model. Then the performance of the model is evaluated against the single instance that was held out. This process is repeated n-1 additional times until all the instances in the dataset have been used once for validation. After the last iteration, we end up with n estimates of the model's performance from each of the iterations. The average of these estimates is used as the LOOCV estimate of model performance.

```r
ctrl_loocv <- trainControl(method = "LOOCV")
set.seed(1234)
model_rpart_loocv <- train(turnover ~ .,
                           data = satisfy_train,
                           metric = "Accuracy",
                           method = "rpart",
                           trControl = ctrl_loocv)
```

2.7.5 leave group out cross validation

This approach is similar to k-fold cross-validation but with one notable difference. In this approach, instead of creating a set number of folds (validation sets) at the beginning of the process, as we do in k-fold cross-validation, the random sample that makes up the validation set is created during each iteration During the first iteration, a random sampling without replacement approach is used to create the validation set. This dataset is held out for validation, and the remainder of the data is used to train the model. In the second iteration, a new independent validation set is randomly selected. Because of the random nature of the sampling approach, it is possible and likely that some of the instances selected as part of this new validation set were also selected as part of the validation set in the previous iteration.

```
ctrl_lgocv <- trainControl(method = "LGOCV", p = .1, number = 10)
set.seed(1234)
model_rpart_lgocv <- train(turnover ~ .,
                           data = satisfy_train,
                           metric = "Accuracy",
                           method = "rpart",
                           trControl = ctrl_lgocv)
```

2.7.6 bootstrap

The basic idea behind bootstrap sampling is to create a training dataset from the original data using a random sampling with replacement approach. A version of this technique, known as the 0.632 bootstrap, involves random sampling a dataset with n instances, n different times with replacement, to create another dataset also with n instances. This new dataset is used for training, while the instances from the original data, which were not selected as part of the training data, are used for validation.

```
ctrl_boot <- trainControl(method = "boot632", number = 3)
set.seed(1234)
model_rpart_boot <- train(turnover ~ .,
                          data = satisfy_train,
                          metric = "Accuracy",
                          method = "rpart",
                          trControl = ctrl_boot)
```

2.8 Spot-Check Machine Learning Algorithms

What algorithm should you use on your dataset? This is the most common question in applied machine learning. It's a question that can only be answered by trial and error, or what I call: spot-checking algorithms. In this session you will discover how to spot-check algorithms on a dataset using R.

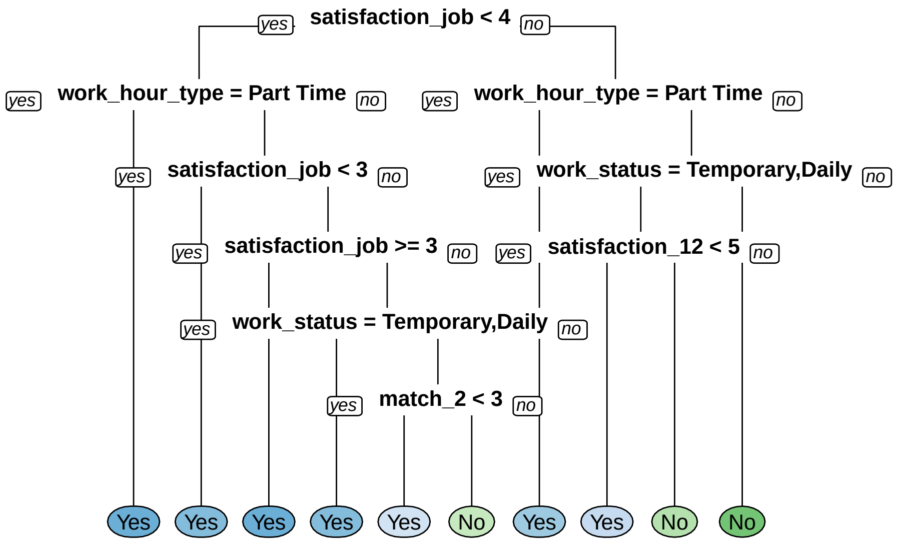
2.8.1 CART Algorithm

The CART algorithm is implemented in R as part of the rpart package. This package provides a similarly named function `rpart()`, which provides an implementation of CART (Classification And Regression Trees) for classification and regression. This function takes three primary arguments. The first is the prediction formula, which we specify as `employ ~ .` to mean that our model should use all the other variables in the dataset as predictors for the `employ` variable. The second argument is the method, which we specify as `class`. This means that we are building a classification tree. The final argument is the training dataset that will be used to build the model.

```r
# Building a CART model
model_rpart <- rpart(turnover ~ .,method = "class",
                     data = satisfy_train,
                     control = rpart.control(maxdepth = 7,
                                             cp = 0.005))
```

Now that we've trained our decision tree model, let's visualize it. To do so, we use the `rpart.plot()` function from the similarly named rpart.plot package.

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



2.8.2 Bagging Algorithm

One of the most common ensemble learning approaches is known as bagging, which stands for bootstrap aggregating. The name comes from the fact that bagging ensembles use a bootstrap sampling approach for the allocation function, which is used to generate the data assigned to each model in the ensemble. Bagging ensembles are typically made up of homogenous learners, which are trained independently and in parallel.

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).

```
# Bagging is a randomized model, so let's set a seed (123) for reproducibility
set.seed(123)
# Building a bagged model
model_bagging <- bagging(formula = turnover ~ .,
                         data = satisfy_train,
                         coob = TRUE)
```

2.8.3 randomforest Algorithm

One of the most popular bagging ensemble methods is the random forests or decision tree forests ensemble technique. It gets its name from the fact that the ensemble consists of a large number of decision tree learners (which are collectively called a forest) and that its allocation function combines both bootstrap sampling and random feature selection to generate the data assigned to each learner in the ensemble. By using only a random subset of the full feature set, random forests are able to handle very wide datasets (datasets with a large number of features). To illustrate the random forests ensemble technique in R, we use the `rf` method in caret, which depends on the aptly named randomForest package. Using the `modelLookup()` command for the `rf` method reveals that it has only one tunable parameter: `mtry`. This is the number of randomly selected features to consider at each split (more on this shortly).

```
modelLookup("rf")
```

```
##   model parameter                         label forReg forClass probModel
## 1    rf      mtry #Randomly Selected Predictors   TRUE     TRUE      TRUE
```

Based on the documentation provided by the randomForest package, the default value for mtry is the square root of the number of features in the dataset when working on a classification problem.

```
# Building a Random Forest model
set.seed(123)  # for reproducibility
model_rf <- randomForest(formula = turnover ~ .,
                         data = satisfy_train)
```

2.8.4 gbm Algorithm

Boosting is a method of converting weak learners into strong learners. In boosting, each new tree is a fit on a modified version of the original data set. The gradient boosting algorithm (gbm) can be most easily explained by first introducing the AdaBoost Algorithm.The AdaBoost Algorithm begins by training a decision tree in which each observation is assigned an equal weight. After evaluating the first tree, we increase the weights of those observations that are difficult to classify and lower the weights for those that are easy to classify. The second tree is therefore grown on this weighted data. Here, the idea is to improve upon the predictions of the first tree. Our new model is therefore Tree 1 + Tree 2. We then compute the classification error from this new 2-tree ensemble model and grow a third tree to predict the revised residuals. We repeat this process for a specified number of iterations. Subsequent trees help us to classify observations that are not well classified by the previous trees. Predictions of the final ensemble model is therefore the weighted sum of the predictions made by the previous tree models. In order to train a `gbm` model in R, you will first have to install and call the gbm library. The gbm function requires you to specify certain arguments. You will begin by specifying the formula. This will include your response and predictor variables. Next, you will specify the distribution of your response variable. If nothing is specified, then gbm will try to guess. Some commonly used distributions include- "`bernoulli`" (logistic regression for 0–1 outcome), "`gaussian`" (squared errors), "`tdist`"(t-distribution loss), and "`poisson`" (count outcomes). Finally, we will specify the data and the n.trees argument (after all gbm is an ensemble of trees!) By default, the gbm model will assume 100 trees, which can provide is a good estimate of our gbm's performance.

```r
# Building a 10000-tree GBM model
# Convert "yes" to 1, "no" to 0
gbm_train <- satisfy_train
gbm_train$turnover <- ifelse(gbm_train$turnover == "Yes", 1, 0)

set.seed(123)
model_gbm <- gbm(formula = turnover ~ .,
                 distribution = "bernoulli",
                 data = gbm_train,
                 n.trees = 10000)
```

2.8.5 logistic model

you used the `glm()` function to build a logistic regression model. As with many of R's machine learning methods, you can apply the `predict()` function to the model object to forecast future behavior. By default, `predict()` outputs predictions in terms of log odds unless `type = "response"` is specified. This converts the log odds to probabilities.

```r
model_logistic <- glm(turnover ~. ,data = satisfy_train,
                      family="binomial")
```

2.9 Model Evaluation Metrics

What metrics can you use to evaluate your machine learning algorithms? In this lesson you will discover how you can evaluate your machine learning algorithms in R using a number of standard evaluation metrics. After completing this lesson you will know:

- How to use Accuracy and Kappa to evaluate model skill on classification problems.
- How to use Area Under ROC Curve, sensitivity and specificity to evaluate model skill on binary classification problems.

2.9.1 Accuracy and Kappa

Accuracy and Kappa are the default metrics used to evaluate algorithms on binary and multiclass classification datasets in caret. Accuracy is the percentage of correctly classified instances out of all instances. It is more useful on a binary classification than multi-class classification problem because it can be less clear exactly how the accuracy breaks down across those classes. Kappa or Cohen's Kappa is like classification accuracy, except that it is normalized at the baseline of random chance on your dataset. It is a more useful measure to use on problems that have an imbalance in the classes (e.g. a 70% to 30% split for classes 0 and 1 and you can achieve 70% accuracy by predicting all instances are for class 0).

```
# Accuracy and Kappa
ctrl_cv <- trainControl(method = "cv", number = 5)
set.seed(1234)
model_glm_cv_accuracy <- train(turnover ~ .,
                        data = satisfy_train,
                        metric = "Accuracy",
                        method = "glm", family = "binomial",
                        trControl = ctrl_cv)
```

2.9.2 Area Under ROC Curve

ROC metrics are only suitable for binary classification problems (e.g. two classes). To calculate ROC information, you must change the summaryFunction in your trainControl to be twoClassSummary. This will calculate the Area Under ROC Curve (AUROC) also called just Area Under curve (AUC), sensitivity and specificity. ROC is actually the area under the ROC curve or AUC. The AUC represents a models ability to discriminate between positive and negative classes. An area of 1.0 represents a model that predicts perfectly. An area of 0.5 represents a model as good as random.

- Sensitivity is the true positive rate also called the recall. It is the number of instances from the positive (first) class that actually predicted correctly.

- Specificity is also called the true negative rate. Is the number of instances from the negative class (second class) that were actually predicted correctly.

ROC can be broken down into sensitivity and specificity. A binary classification problem is really a trade-off between sensitivity and specificity.

```
# Area Under ROC Curve

ctrl_cv <- trainControl(method = "cv", number = 5, classProbs = TRUE,
                        summaryFunction = twoClassSummary)
set.seed(1234)
model_glm_cv_roc <- train(turnover ~ .,
                      data = satisfy_train,
                      metric = "ROC",
                      method = "glm", family = "binomial",
                      trControl = ctrl_cv)
```

2.10 The Performance of Machine Learning Algorithms

How do you compare the estimated accuracy of different machine learning algorithms effectively? In this lesson you will discover techniques that you can use to compare machine learning algorithms in R. it's time to compare their out-of-sample predictions and choose which one is the best model for your dataset. You can compare models in caret using the `resamples()` function, provided they have the same training data and use the same `trainControl` object with preset cross-validation folds. `resamples()` takes as input a list of models and can be used to compare dozens of models at once (though in this case you are only comparing five models).

```
# Create custom indices: myFolds
set.seed(1234)
myFolds <- createFolds(satisfy_train$turnover, k = 5)

# Create reusable trainControl object: myControl
myControl <- trainControl(
  summaryFunction = twoClassSummary,
  classProbs = TRUE, # IMPORTANT!
  verboseIter = FALSE,
  savePredictions = TRUE,
  index = myFolds
)

model_caret_rpart <- train(
  turnover ~ .,
```

```r
  data = satisfy_train,
  metric = "ROC",
  method = "rpart",
  trControl = myControl)

model_caret_bagging <- train(
  turnover ~ .,
  data = satisfy_train,
  metric = "ROC",
  method = "treebag",
  trControl = myControl)

model_caret_ranger <- train(
  turnover ~ .,
  data = satisfy_train,
  metric = "ROC",
  method = "ranger",
  trControl = myControl)

model_caret_logistic <- train(
  turnover ~ .,
  data = satisfy_train,
  metric = "ROC",
  method = "glm", family = "binomial",
  trControl = myControl)

model_caret_gbm <- train(
  turnover ~ .,
  data = satisfy_train,
  metric = "ROC",
  method = "gbm",
  trControl = myControl)
```

```r
# Create model_list
model_list <- list(rpart = model_caret_rpart,
                   rf = model_caret_ranger,
                   logistic = model_caret_logistic,
                   bagging = model_caret_bagging,
                   gbm = model_caret_gbm)
```

caret provides a variety of methods to use for comparing models. All of these methods are based on the `resamples()` function. The box-and-whisker plot allows you to compare the distribution of predictive accuracy (in this case AUC) for the two or more models.

In general, you want the model with the higher median AUC, as well as a smaller range between min and max AUC.

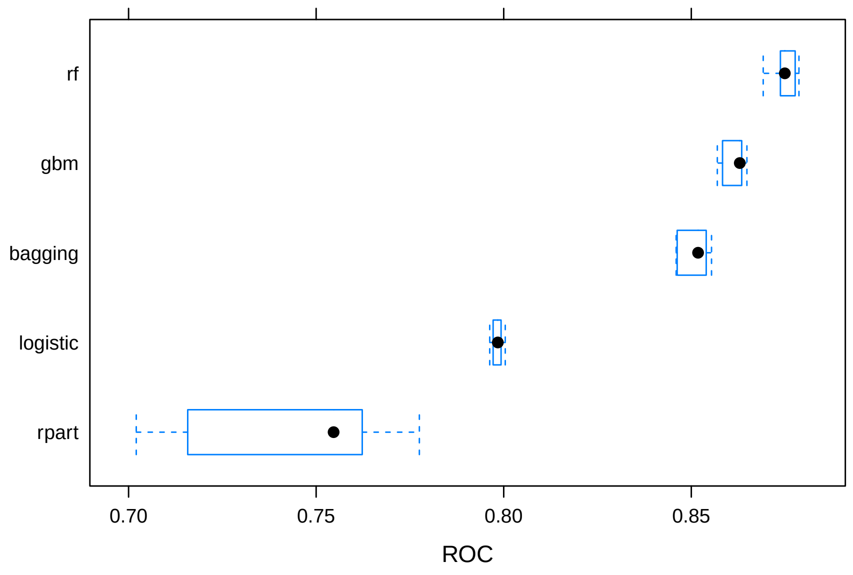
You can make this plot using the `bwplot()` function, which makes a box and whisker plot of the model's out of sample scores. Box and whisker plots show the median of each distribution as a line and the interquartile range of each distribution as a box around the median line. You can pass the metric = "ROC" argument to the `bwplot()` function to show a plot of the model's out-of-sample ROC scores and choose the model with the highest median ROC.

```r
# Pass model_list to resamples(): resamples
resamps <- resamples(model_list)

# Summarize the results
summary(resamps)
```

```
##
## Call:
## summary.resamples(object = resamps)
##
## Models: rpart, rf, logistic, bagging, gbm
## Number of resamples: 5
##
## ROC
##               Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## rpart    0.7020505 0.7157649 0.7546550 0.7424550 0.7622874 0.7775173    0
## rf       0.8691685 0.8737564 0.8749277 0.8748478 0.8776903 0.8786960    0
## logistic 0.7962687 0.7971766 0.7984307 0.7983164 0.7992928 0.8004131    0
## bagging  0.8459675 0.8462392 0.8518077 0.8506775 0.8539794 0.8553938    0
## gbm      0.8569389 0.8583258 0.8629288 0.8612928 0.8634412 0.8648294    0
##
## Sens
##               Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## rpart    0.6645500 0.6861717 0.7300580 0.7518268 0.8108780 0.8674765    0
## rf       0.7291552 0.7297819 0.7421314 0.7435405 0.7524151 0.7642187    0
## logistic 0.6736610 0.6952802 0.6979569 0.6952289 0.6998896 0.7093569    0
## bagging  0.7269464 0.7452388 0.7506902 0.7461349 0.7510351 0.7567642    0
## gbm      0.7310878 0.7424786 0.7512424 0.7486751 0.7513111 0.7672557    0
##
## Spec
##               Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
## rpart    0.5480398 0.5911099 0.7030086 0.6840147 0.7868581 0.7910571    0
## rf       0.7962452 0.8211427 0.8235781 0.8248667 0.8401436 0.8432238    0
## logistic 0.7614578 0.7615236 0.7653878 0.7650178 0.7669796 0.7697405    0
## bagging  0.7614578 0.7775324 0.7836047 0.7863854 0.7934843 0.8158476    0
## gbm      0.7733297 0.8012697 0.8175041 0.8094624 0.8210933 0.8341154    0
```

```r
# Create bwplot
bwplot(resamps, metric = "ROC")
```



2.11 Improving Model Performance

It is difficult to find a good machine learning algorithm for your problem. But once you do, how do you get the best performance out of it. In this session you will discover three ways that you can tune the parameters of a machine learning algorithm in R. Most machine learning techniques have one or more parameters that need to be set before the learning process begins. These parameters are commonly known as hyperparameters.

2.11.1 Random Search

One search strategy that we can use is to try random values within a range. This can be good if we are unsure of what the value might be and we want to overcome any biases we may have for setting the parameter. To use random search, another option is available in `trainControl` called `search`. Possible values of this argument are `"grid"` and `"random"`. The built-in models contained in caret contain code to generate random tuning parameter combinations. The total number of unique combinations is specified by the tuneLength option to train.

```
# random search

ctrl_random <- trainControl(method = "repeatedcv" , number=5,
                             repeats = 3,
                            search = "random")
set.seed(1234)
ranger_random <- train(turnover ~ ., data=satisfy_train, method="ranger",
                  metric = "Accuracy",
                  tuneLength = 5, verbose = FALSE,
                  trControl = ctrl_random)
ranger_random
```

```
## Random Forest
##
## 9056 samples
##   23 predictor
##    2 classes: 'Yes', 'No'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 3 times)
## Summary of sample sizes: 7244, 7246, 7245, 7244, 7245, 7246, ...
## Resampling results across tuning parameters:
##
##   min.node.size  mtry  splitrule   Accuracy   Kappa
##    5              6    extratrees  0.8296153  0.6592304
##    5              7    extratrees  0.8294681  0.6589360
##    9              2    gini        0.8224011  0.6448018
##   12             16    extratrees  0.8208174  0.6416346
##   15              4    extratrees  0.8064625  0.6129253
##
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 6, splitrule = extratrees
##  and min.node.size = 5.
```

2.11.2 Grid Search

Another search you can use is to define a grid of algorithm parameters to try. Each axis of the grid is an algorithm parameter, and points in the grid are specific combinations of parameters. Because we are only tuning one parameter, the grid search is a linear search through a vector of candidate values.

```
# grid search
ctrl_grid <- trainControl(method = "repeatedcv" , number = 5,
                          repeats = 3,
                          search = "grid")
set.seed(1234)
```

```r
tune_grid <- expand.grid( min.node.size= seq(5,10,2),
                         mtry= seq(2,7),
                         splitrule= c("extratrees","gini"))
ranger_grid <- train(turnover ~ ., data = satisfy_train, method = "ranger",
                  metric = "Accuracy", tuneGrid = tune_grid,
                  verbose = FALSE,
                  trControl = ctrl_grid)
```

2.11.3 Adaptive Resampling

Models can benefit significantly from tuning but the optimal values are rarely known beforehand. train can be used to define a grid of possible points and resampling can be used to generate good estimates of performance for each tuning parameter combination. However, in the nominal resampling process, all the tuning parameter combinations are computed for all the resamples before a choice is made about which parameters are good and which are poor. caret contains the ability to adaptively resample the tuning parameter grid in a way that concentrates on values that are the in the neighborhood of the optimal settings. To use the adaptive procedure, the trainControl option needs some additional arguments: - min is the minimum number of resamples that will be used for each tuning parameter. The default value is 5 and increasing it will decrease the speed-up generated by adaptive resampling but should also increase the likelihood of finding a good model. - alpha is a confidence level that is used to remove parameter settings. To date, this value has not shown much of an effect. - method is either "`gls`" for a linear model or "`BT`" for a Bradley-Terry model. The latter may be more useful when you expect the model to do very well (e.g. an area under the ROC curve near 1) or when there are a large number of tuning parameter settings. - complete is a logical value that specifies whether train should generate the full resampling set if it finds an optimal solution before the end of resampling. If you want to know the optimal parameter settings and don't care much for the estimated performance value, a value of FALSE would be appropriate here

```r
# adaptive resampling
fitControl <- trainControl(method = "adaptive_cv",
                           number = 3, repeats = 3,
                           adaptive = list(min = 2,
                                           alpha = 0.05,
                                           method = "gls",
                                           complete = TRUE),
                           search = "random")

set.seed(1234)
ranger_adaptive <- train(turnover ~ ., data = satisfy_train,
                         method = "ranger",
                         trControl = fitControl,
                         verbose = FALSE,
                         tuneLength = 7)
ranger_adaptive
```

```
## Random Forest
##
## 9056 samples
##   23 predictor
##    2 classes: 'Yes', 'No'
##
## No pre-processing
## Resampling: Adaptively Cross-Validated (3 fold, repeated 3 times)
## Summary of sample sizes: 6038, 6038, 6036, 6037, 6038, 6037, ...
## Resampling results across tuning parameters:
##
##   min.node.size  mtry  Accuracy   Kappa      Resamples
##    5              4    0.8178010  0.6356021  9
##    5              6    0.8184270  0.6368542  9
##    6             15    0.8145767  0.6291542  4
##    9             22    0.8125066  0.6250142  4
##   12              2    0.7955600  0.5911199  2
##   15              7    0.8127899  0.6255799  2
##   16             14    0.8122929  0.6245858  2
##
## Tuning parameter 'splitrule' was held constant at a value of extratrees
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were mtry = 6, splitrule = extratrees
##  and min.node.size = 5.
```

2.12 Finalize Your Machine Learning Model

Once you have an accurate model on your test harness you are nearly done. But not yet. There are still a number of tasks to do to finalize your model. The whole idea of creating an accurate model for your dataset was to make predictions on unseen data. There are three tasks you may be concerned with:

- Making new predictions on unseen data.
- Creating a standalone model using all training data.
- Saving your model to file for later loading and making predictions on new data.

```
# Finalize Your Machine Learning Model

finalModel <- randomForest(turnover ~ .,
  data = satisfy_train,
```

```r
  min.node.size = 5,
  mtry = 6,
  splitrule = "extratrees",
  verbose = FALSE,
  )
final_predictions <- predict(finalModel, satisfy_test,type="class")
confusionMatrix(final_predictions, satisfy_test$turnover, positive= "Yes")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  Yes   No
##        Yes  354  395
##        No   211 1207
##
##                Accuracy : 0.7204
##                  95% CI : (0.7009, 0.7392)
##     No Information Rate : 0.7393
##     P-Value [Acc > NIR] : 0.9783
##
##                   Kappa : 0.3437
##
##  Mcnemar's Test P-Value : 1.055e-13
##
##             Sensitivity : 0.6265
##             Specificity : 0.7534
##          Pos Pred Value : 0.4726
##          Neg Pred Value : 0.8512
##              Prevalence : 0.2607
##          Detection Rate : 0.1634
##    Detection Prevalence : 0.3456
##       Balanced Accuracy : 0.6900
##
##        'Positive' Class : Yes
##
```