Machine Learning Mastery with R

Get Started, Build Accurate Models and Work Through Projects Step-by-Step

Jason Brownlee

Jason Brownlee

Machine Learning Mastery With R

Get Started, Build Accurate Models and Work Through Projects Step-by-Step

Machine Learning Mastery With R

Edition: v1.5

Contents

Preface iii

I Introduction 1

1 Welcome 2
1.1 Learn R The Wrong Way . . . 2
1.2 Machine Learning in R . . . 2
1.3 What This Book is Not . . . 6
1.4 Summary . . . 7

2 The R Platform 8
2.1 Why Use R . . . 8
2.2 What Is R . . . 9
2.3 Summary . . . 10

II Lessons 11

3 Installing and Starting R 12
3.1 Download and Install R . . . 12
3.2 R Interactive Environment . . . 13
3.3 R Scripts . . . 15
3.4 Summary . . . 16

4 Crash Course in R For Developers 17
4.1 R Syntax is Different, But The Same . . . 17
4.2 Assignment . . . 18
4.3 Data Structures . . . 18
4.4 Flow Control . . . 20
4.5 Functions . . . 21
4.6 Packages . . . 22
4.7 5 Things To Remember . . . 23
4.8 Summary . . . 23

5 Standard Machine Learning Datasets **25**
5.1 Practice On Small Well-Understood Datasets . . . 25
5.2 Package: datasets . . . 26
5.3 Package: mlbench . . . 27
5.4 Package: AppliedPredictiveModeling . . . 34
5.5 Summary . . . 35

6 Load Your Machine Learning Datasets **37**
6.1 Access To Your Data . . . 37
6.2 Load Data From CSV File . . . 38
6.3 Load Data From CSV URL . . . 38
6.4 Summary . . . 39

7 Understand Your Data Using Descriptive Statistics **40**
7.1 You Must Understand Your Data . . . 40
7.2 Peek At Your Data . . . 41
7.3 Dimensions of Your Data . . . 42
7.4 Data Types . . . 42
7.5 Class Distribution . . . 43
7.6 Data Summary . . . 43
7.7 Standard Deviations . . . 44
7.8 Skewness . . . 45
7.9 Correlations . . . 45
7.10 Tips To Remember . . . 46
7.11 Summary . . . 46

8 Understand Your Data Using Data Visualization **48**
8.1 Understand Your Data To Get The Best Results . . . 48
8.2 Visualization Packages . . . 49
8.3 Univariate Visualization . . . 49
8.4 Multivariate Visualization . . . 54
8.5 Tips For Data Visualization . . . 59
8.6 Summary . . . 60

9 Prepare Your Data For Machine Learning With Pre-Processing **61**
9.1 Need For Data Pre-Processing . . . 61
9.2 Data Pre-Processing in R . . . 62
9.3 Scale Data . . . 63
9.4 Center Data . . . 64
9.5 Standardize Data . . . 65
9.6 Normalize Data . . . 65
9.7 Box-Cox Transform . . . 66
9.8 Yeo-Johnson Transform . . . 67
9.9 Principal Component Analysis Transform . . . 68
9.10 Independent Component Analysis Transform . . . 69
9.11 Tips For Data Transforms . . . 71
9.12 Summary . . . 71

10 Resampling Methods To Estimate Model Accuracy **72**
10.1 Estimating Model Accuracy . . . 72
10.2 Data Split . . . 73
10.3 Bootstrap . . . 74
10.4 k-fold Cross Validation . . . 75
10.5 Repeated k-fold Cross Validation . . . 75
10.6 Leave One Out Cross Validation . . . 76
10.7 Tips For Evaluating Algorithms . . . 77
10.8 Summary . . . 77

11 Machine Learning Model Evaluation Metrics **79**
11.1 Model Evaluation Metrics in R . . . 79
11.2 Accuracy and Kappa . . . 79
11.3 RMSE and R^2 . . . 80
11.4 Area Under ROC Curve . . . 81
11.5 Logarithmic Loss . . . 82
11.6 Summary . . . 83

12 Spot-Check Machine Learning Algorithms **84**
12.1 Best Algorithm For a Problem . . . 84
12.2 Algorithms To Spot-Check in R . . . 85
12.3 Linear Algorithms . . . 86
12.4 Nonlinear Algorithms . . . 90
12.5 Other Algorithms . . . 95
12.6 Summary . . . 95

13 Compare The Performance of Machine Learning Algorithms **96**
13.1 Choose The Best Machine Learning Model . . . 96
13.2 Prepare Dataset . . . 97
13.3 Train Models . . . 97
13.4 Compare Models . . . 98
13.5 Summary . . . 105

14 Tune Machine Learning Algorithms **106**
14.1 Get Better Accuracy From Top Algorithms . . . 106
14.2 Tune Machine Learning Algorithms . . . 107
14.3 Test Setup . . . 107
14.4 Tune Using Caret . . . 108
14.5 Tune Using Algorithm Tools . . . 111
14.6 Craft Your Own Parameter Search . . . 113
14.7 Summary . . . 116

15 Combine Predictions From Multiple Machine Learning Models **117**
15.1 Increase The Accuracy Of Your Models . . . 117
15.2 Test Dataset . . . 118
15.3 Boosting Algorithms . . . 119
15.4 Bagging Algorithms . . . 120

15.5 Stacking Algorithms . . . 122
15.6 Summary . . . 125

16 Save And Finalize Your Machine Learning Model **127**
16.1 Finalize Your Machine Learning Model . . . 127
16.2 Make Predictions On New Data . . . 128
16.3 Create A Standalone Model . . . 129
16.4 Save and Load Your Model . . . 131
16.5 Summary . . . 132

III Projects **134**

17 Predictive Modeling Project Template **135**
17.1 Practice Machine Learning With Projects . . . 135
17.2 Machine Learning Project Template in R . . . 136
17.3 Machine Learning Project Template Steps . . . 137
17.4 Tips For Using The Template Well . . . 139
17.5 Summary . . . 139

18 Your First Machine Learning Project in R Step-By-Step **140**
18.1 Hello World of Machine Learning . . . 140
18.2 Load The Data . . . 141
18.3 Summarize Dataset . . . 142
18.4 Visualize Dataset . . . 145
18.5 Evaluate Some Algorithms . . . 149
18.6 Make Predictions . . . 153
18.7 Summary . . . 154

19 Regression Machine Learning Case Study Project **155**
19.1 Problem Definition . . . 155
19.2 Analyze Data . . . 156
19.3 Evaluate Algorithms: Baseline . . . 164
19.4 Evaluate Algorithms: Feature Selection . . . 167
19.5 Evaluate Algorithms: Box-Cox Transform . . . 170
19.6 Improve Results With Tuning . . . 171
19.7 Ensemble Methods . . . 174
19.8 Finalize Model . . . 178
19.9 Summary . . . 179

20 Binary Classification Machine Learning Case Study Project **181**
20.1 Problem Definition . . . 181
20.2 Analyze Data . . . 182
20.3 Evaluate Algorithms: Baseline . . . 191
20.4 Evaluate Algorithms: Transform . . . 194
20.5 Algorithm Tuning . . . 195
20.6 Ensemble Methods . . . 199

20.7 Finalize Model . . . 201
20.8 Summary . . . 203

21 More Predictive Modeling Projects **204**
21.1 Build And Maintain Recipes . . . 204
21.2 Small Projects on Small Datasets . . . 204
21.3 Competitive Machine Learning . . . 205
21.4 Summary . . . 205

IV Conclusions **206**

22 How Far You Have Come **207**

23 Getting More Help **208**
23.1 CRAN . . . 208
23.2 Q&A Websites . . . 209
23.3 Mailing Lists . . . 210
23.4 Package Websites . . . 210
23.5 Books . . . 211

Preface

I think R is an amazing platform for machine learning. There are so many algorithms and so much power sit there ready to use. I am often asked the question: *How do you use R for machine learning?* This book is my definitive answer to that question. It contains my very best knowledge and ideas on how to work through predictive modeling machine learning projects using the R platform. It is the book that I am also going to use as a refresher at the start of a new project. I'm really proud of this book and I hope that you find it a useful companion on your machine learning journey with R.

Jason Brownlee
Melbourne, Australia
2016

Part I

Introduction

Chapter 1

Welcome

Welcome to *Machine Learning Mastery With R*. This book is your guide to applied machine learning with R. You will discover the step-by-step process that you can use to get started and become good at machine learning for predictive modeling on the R platform.

1.1 Learn R The Wrong Way

Here is what you should NOT do when you start studying machine learning in R.

1. Get really good at R programming and R syntax.
2. Deeply study the underlying theory and parameters for machine learning algorithms in R.
3. Avoid or lightly touch on all of the other tasks needed to complete a real project.

I think that this approach can work for some people, but it is a really slow and a roundabout way of getting to your goal. It teaches you that you need to spend all your time learning how to use individual machine learning algorithms. It also does not teach you the process of building predictive machine learning models in R that you can actually use to make predictions. Sadly, this is the approach used to teach machine learning that I see in almost all books and online courses on the topic.

1.2 Machine Learning in R

This book focuses on a specific sub-field of machine learning called predictive modeling. This is the field of machine learning that is the most useful in industry and the type of machine learning that the R platform excels at facilitating. Unlike statistics, where models are used to *understand* data, predictive modeling is laser focused on developing models that make the *most accurate predictions* at the expense of explaining why predictions are made. Unlike the broader field of machine learning that could feasibly be used with data in any format, predictive modeling is primarily focused on tabular data (e.g. tables of numbers like a spreadsheet).

This book was written around three themes designed to get you started and using R for applied machine learning effectively and quickly. These three parts are as follows:

Lessons : Learn how the sub-tasks of a machine learning project map onto R and the best practice way of working through each task.

Projects : Tie together all of the knowledge from the lessons by working through case study predictive modeling problems.

Recipes : Apply machine learning with a catalog of standalone recipes in R that you can copy-and-paste as a starting point for new projects.

1.2.1 Lessons

You need to know how to complete the specific subtasks of a machine learning project on the R platform. Once you know how to complete a discrete task using the platform and get a result reliably, you can do it again and again on project after project. Let's start with an overview of the common tasks in a machine learning project. A predictive modeling machine learning project can be broken down into 6 top-level tasks:

1. **Define Problem**: Investigate and characterize the problem in order to better understand the goals of the project.
2. **Analyze Data**: Use descriptive statistics and visualization to better understand the data you have available.
3. **Prepare Data**: Use data transforms in order to better expose the structure of the prediction problem to modeling algorithms.
4. **Evaluate Algorithms**: Design a test harness to evaluate a number of standard algorithms on the data and select the top few to investigate further.
5. **Improve Results**: Use algorithm tuning and ensemble methods to get the most out of well-performing algorithms on your data.
6. **Present Results**: Finalize the model, make predictions and present results.

A blessing and a curse with R is that there are so many techniques and so many ways to do something with the platform. In part II of this book you will discover one easy or best practice way to complete each subtask of a general machine learning project. Below is a summary of the Lessons from Part II and the sub-tasks that you will learn about.

- Lesson 1: Install and Start R.
- Lesson 2: R Language Crash Course.
- Lesson 3: Load Standard Datasets.
- Lesson 4: Load Custom Data. (**Analyze Data**)
- Lesson 5: Understand Data With Descriptive Statistics. (**Analyze Data**)
- Lesson 6: Understand Data With Visualization. (**Analyze Data**)

- Lesson 7: Pre-Process Data. (**Prepare Data**)
- Lesson 8: Resampling Methods. (**Evaluate Algorithms**)
- Lesson 9: Algorithm Evaluation Metrics. (**Evaluate Algorithms**)
- Lesson 10: Spot-Check Algorithms. (**Evaluate Algorithms**)
- Lesson 11: Model Selection. (**Evaluate Algorithms**)
- Lesson 12: Algorithm Parameter Tuning. (**Improve Results**)
- Lesson 13: Ensemble Methods. (**Improve Results**)
- Lesson 14: Finalize Model. (**Present Results**)

These lessons are intended to be read from beginning to end in order, showing you exactly how to complete each task in a predictive modeling machine learning project. Of course, you can dip into specific lessons again later to refresh yourself. Some lessons are demonstrative, showing you how to use specific techniques for a common machine learning task (e.g. data loading and data pre-processing). Others are in a tutorial format, building throughout the lesson to culminate in a final result (e.g. algorithm tuning and ensemble methods). Each lesson was designed to be completed in under 30 minutes (depending on your level of skill and enthusiasm). It is possible to work through the entire book in one weekend. It also works if you want to dip into specific sections and use the book as a reference.

1.2.2 Projects

Recipes for common predictive modeling tasks are critically important, but they are also just the starting point. This is where most books and courses stop.

You need to piece the recipes together into end-to-end projects. This will show you how to actually deliver a model or make predictions on new data using R. This book uses small well-understood machine learning datasets from the UCI Machine learning repository[1] in both the lessons and in the example projects. These datasets are available for free as CSV downloads, and most are available directly in R by loading third party packages. These datasets are excellent for practicing applied machine learning because:

- **They are small**, meaning they fit into memory and algorithms can model them in reasonable time.
- **They are well behaved**, meaning you often don't need to do a lot of feature engineering to get a good result.
- **They are benchmarks**, meaning that many people have used them before and you can get ideas of good algorithms to try and accuracy levels you should expect.

In Part III you will work through three projects:

[1] http://archive.ics.uci.edu/ml

Hello World Project (Iris flowers dataset) : This is a quick pass through the project steps without much tuning or optimizing on a dataset that is widely used as the *hello world* of machine learning.

Regression (Boston House Price dataset) : Work through each step of the project process with a regression problem.

Binary Classification (Wisconsin Breast Cancer dataset) : Work through each step of the project process using all of the methods on a binary classification problem.

These projects unify all of the lessons from Part II. They also give you insight into the process for working through predictive modeling machine learning problems which is invaluable when you are trying to get a feeling for how to do this in practice. Also included in this section is a template for working through predictive modeling machine learning problems which you can use as a starting point for current and future projects. I find this useful myself to set the direction and setup important tasks (which are easy to forget) on new projects, and I'm sure you will too.

1.2.3 Recipes

Recipes are small code snippets in R that show you how to do one specific thing and get a result. For example, you could have a recipe that demonstrates how to use Random Forest algorithm for classification. You could have another for normalizing the attributes of a dataset.

Recipes make the difference between a beginner who is having trouble and a fast learner capable of making accurate predictions quickly on any new project. A catalog of recipes provides a repertoire of skills that you can draw from when starting a new project. More formally, recipes are defined as follows:

- Recipes are code snippets not tutorials.
- Recipes provide just enough code to work.
- Recipes are demonstrative not exhaustive.
- Recipes run as-is and produce a result.
- Recipes assume that required packages are installed.
- Recipes use built-in datasets or datasets provided in specific packages.

You are starting your journey into machine learning with R with my personal catalog of machine learning recipes provided with this book. All of the code from the lessons in Part II are available in your R recipe catalog. There are also recipes for techniques not covered in this book, including usage of a very large number of algorithms and many additional case studies. Recipes are divided into directories according to the common tasks of a machine learning project as listed above. The list below provides a summary of the recipes available.

- **Analyze Data**: Recipes to load, summarize and visualize data, including visualizations using univariate plots, multivariate plots and projection methods.

- **Prepare Data**: Recipes for data preparation including data cleaning, feature selection and data transforms.
- **Algorithms**: Recipes for using a large number of machine learning algorithms both standalone and within the popular R package `caret`, including linear, nonlinear, trees, ensembles for classification and regression.
- **Evaluate Algorithms**: Recipes for re-sampling methods, algorithm evaluation metrics and model selection.
- **Improve Results**: Recipes for algorithm tuning and ensemble methods.
- **Finalize Model**: Recipes to make final predictions, to finalize the model and save and load models to disk.
- **Other**: Recipes for managing packages and getting started with R syntax.
- **Case Studies**: Case studies for binary classification, multiclass classification and regression problems.

This is an invaluable resource that you can use to jump-start your current and future machine learning projects. You can also build upon this recipe catalog as you discover new techniques.

1.2.4 Your Outcomes From This Process

This book will lead you from being a developer who is interested in machine learning with R to a developer who has the resources and capability to work through a new dataset end-to-end using R and develop accurate predictive models. Specifically, you will know:

- How to work through a small to medium sized dataset end-to-end.
- How to deliver a model that can make accurate predictions on new unseen data.
- How to complete all subtasks of a predictive modeling problem with R.
- How to learn new and different techniques in R.
- How to get help with R.

From here you can start to dive into the specifics of the functions, techniques and algorithms used with the goal of learning how to use them better in order to deliver more accurate predictive models, more reliably in less time.

1.3 What This Book is Not

This book was written for professional developers who want to know how to build reliable and accurate machine learning models in R.

- **This is not a machine learning textbook**. We will not be getting into the basic theory of machine learning (e.g. induction, bias-variance trade-off, etc.). You are expected to have some familiarity with machine learning basics, or be able to pick them up yourself.

- **This is not an algorithm book**. We will not be working through the details of how specific machine learning algorithms work (e.g. random forest). You are expected to have some basic knowledge of machine learning algorithms or how to pick up this knowledge yourself.
- **This is not an R programming book**. We will not be spending a lot of time on R syntax and programming (e.g. basic programming tasks in R). You are expected to be a developer who can pick up a new C-like language relatively quickly.

You can still get a lot out of this book if you are weak in one or two of these areas, but you may struggle picking up the language or require some more explanation of the techniques. If this is the case, see the Resources Chapter at the end of the book and seek out a good companion reference text.

1.4 Summary

I hope you are as excited as me to get started. In this introduction chapter you learned that this book is unconventional. Unlike other books and courses that focus heavily on machine learning algorithms in R and focus on little else, this book will walk you through each step of a predictive modeling machine learning project.

- Part II of this book provides standalone lessons including a mixture of recipes and tutorials to build up your basic working skills and confidence in R.
- Part III of this book will introduce a machine learning project template that you can use as a starting point on your own projects and walks you through three end-to-end projects.
- The recipes companion to this book provides a catalog of more than 150 machine learning recipes in R. You can browse this invaluable resource, find useful recipes and copy-and-paste them into your current and future machine learning projects.
- Part IV will finish out the book. It will look back at how far you have come in developing your new found skills in applied machine learning with R. You will also discover resources that you can use to get help if and when you have any questions about R or the platform.

1.4.1 Next Step

In the next Chapter you will take a closer look at R. You will discover what R is, why it is so powerful as a platform for machine learning and the different ways you should and should not use the platform.

Chapter 2

The R Platform

R is one of the most powerful machine learning platforms and is used by the top data scientists in the world. In this Chapter you will get an introduction to R and why you should use R for machine learning.

2.1 Why Use R

There are five reasons why you should use R for your predictive modeling machine learning problems:

- **R is used by the best data scientists in the world**. In surveys on Kaggle (the competitive machine learning platform), R is by far the most used machine learning tool[1] When professional machine learning practitioners were surveyed in 2015, again the most popular machine learning tool was R[2].
- **R is powerful because of the breadth of techniques it offers in third-party packages**. Any techniques that you can think of for data analysis, visualization, data sampling, supervised learning and model evaluation are provided in R. The platform has more techniques than any that you will come across.
- **R is state-of-the-art because it is used by academics**. One of the reasons why R has so many techniques is because academics who develop new algorithms are developing them in R and releasing them as R packages. This means that you can get access to state-of-the-art algorithms in R before other platforms. It also means that you can only access some algorithms in R until someone ports them to other platforms.
- **R is free because it is open source software**. You can download it right now for free and it runs on any workstation platform you are likely to use.
- **R is a lot of fun**. I think the fun comes from the sense of exploration (you're always finding out about some new amazing technique) and because of the results you get (you can run very powerful methods on your data in a few lines of code). I use other platforms, but I always come back to R when I've got serious work to do.

Convinced?

[1] http://blog.kaggle.com/2011/11/27/kagglers-favorite-tools

[2] http://www.kdnuggets.com/2015/05/poll-r-rapidminer-python-big-data-spark.html

2.2 What Is R

R is a language, an interpreter and a platform.

R is a computer language . It can be difficult to learn but is familiar and you will figure it out quickly if you have used other scripting languages like Python, Ruby or BASH.

R is an interpreter . You can write scripts and save them as files. Like other scripting languages, you can then use the interpreter to run those scripts any time. R also provides a REPL (read-evaluate-print loop) environment where you can type in commands and see the output immediately.

R is also a platform . You can use it to create and display graphics, to save and load state and to interface with other systems. You can do all of your exploration and development in the REPL environment if you so wish.

2.2.1 Where R Came From

R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand as an implementation of the S programming language. Development started in 1993. A version was made available on FTP released under the GNU GPL in 1995. The larger core group and open source project was setup in 1997.

It started as an experiment by the authors to implement a statistical test bed in Lisp using a syntax like that provided in S. As it developed, it took on more of the syntax and features of S, eventually surpassing it in capability and scope.

2.2.2 Power In The Packages

R itself is very simple. It provides built in commands for basic statistics and data handling. The machine learning features of R that you will use come from third party packages. Packages are plug-ins to the R platform. You can search for, download and install them within the R environment.

Because packages are created by third parties, their quality can vary. It is a good idea to search for the best-of-breed packages that provide a specific technique you want to use. Packages provide documentation in the form of help for each package function and often vignettes that demonstrate how to use the package.

Before you write a line of code, always search to see if there is a package that can do what you need. You can search for packages on the Comprehensive R Archive Network or CRAN for short[3].

2.2.3 Not For Production

R is probably not the best solution for building a production model. The techniques may be state-of-the-art but they may not use the best software engineering principles, have tests or be scalable to the size of the datasets that you may need to work with.

[3] https://cran.r-project.org

That being said, R may be the best solution to discover what model to actually use in production. The landscape is changing and people are writing R scripts to run operationally and services are emerging to support larger datasets.

2.3 Summary

This chapter provided an introduction to R. You discovered:

- 5 Good reasons why you should be using R for machine learning, not least because it is used by some of the best data scientists in the world.
- That R is 3 things, a programming language, interpreter and platform.
- That the power of R comes from the free third-party packages that you can download.
- That R is excellent for R&D and one-off projects, but probably inappropriate for running production models.

2.3.1 Next Step

Next is Part II where you will begin working through the lessons, the main body of this book. The first lesson is quick and will assist you in installing R and running it for the first time.

Part II

Lessons

Chapter 3

Installing and Starting R

You may not have R installed and you may have never used the R environment before. This lesson will teach you:

1. How to download and install R.
2. How to start the R interactive environment.
3. How to run an R script.

If you have R installed, know how to start and use R, you can probably skip this lesson.

3.1 Download and Install R

You need R installed if you are going to learn how to work through predictive modeling machine learning problems using the platform. As a developer, you are already familiar with how to download and install software on your computer. So I don't need to go into too much detail. Let's download and install R:

1. Start off by visiting the R Project home page at:
   https://www.r-project.org.
   You will notice that there is no direct download link, instead you are directed to a list of international mirror websites from which you can download R:
   https://cran.r-project.org/mirrors.html.
2. Select a mirror and click the link, for example the Berkley mirror:
   http://cran.cnr.berkeley.edu/.
3. You are now presented with a list of R downloads for Linux, Mac (OS X) and Windows. Each platform has its own webpage. For Linux, you can choose the appropriate platform and you will be directed to use the package manager to install R. Mac provides binaries for modern versions of the operating system, download the package. Windows provides an executable that you can download.
4. Once you have downloaded the binary package suitable for your environment, go ahead and install it. Depending on your platform, you may or may not require administration rights.

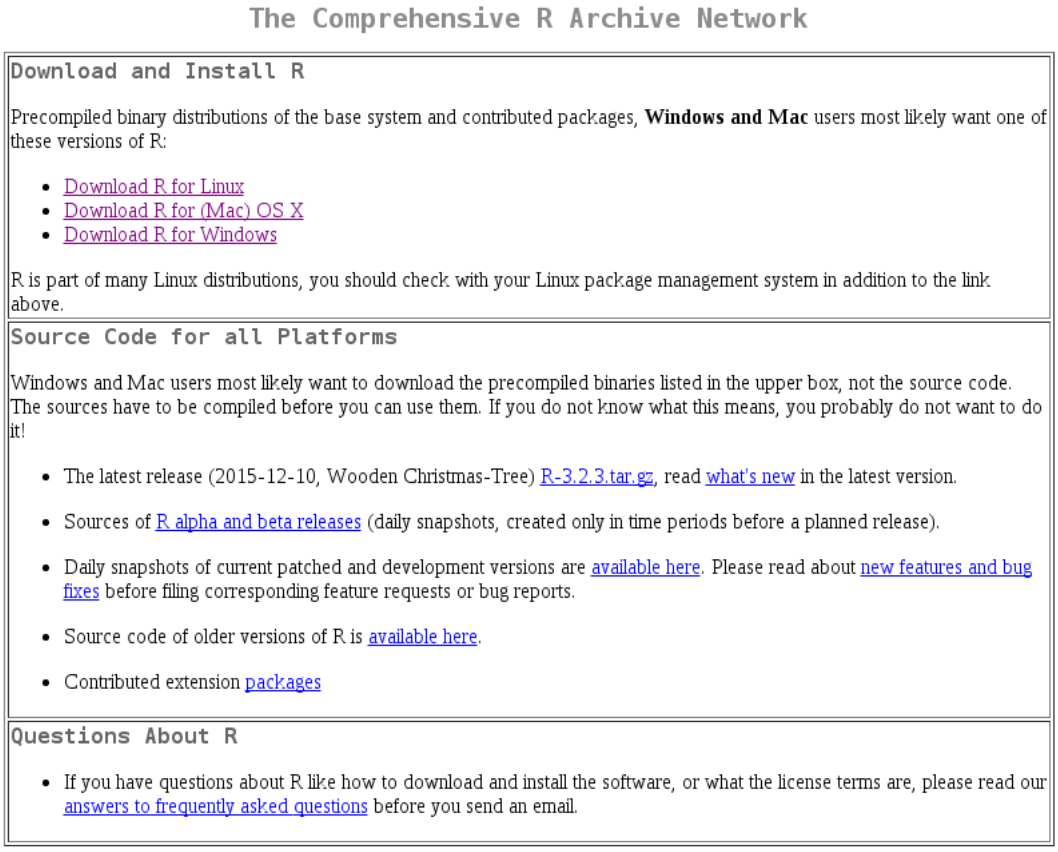

Figure 3.1: Screenshot of the R download webpage.

Which version of R should you download? Always download the latest version. The examples in this book require at least version 3.2 of R. If you are having trouble, each platform has specific installation instructions and Frequently Asked Questions (FAQs) that you may find useful. A good general place to start is the *How can R be installed?* part of the general R FAQ[1]. Another helpful guide is: R Installation and Administration[2].

3.2 R Interactive Environment

You can start R from whatever menu system you use on your operating system. This will load the R interactive environment. For example, below is a screenshot of the R interactive environment on my system.

[1] https://cran.r-project.org/doc/FAQ/R-FAQ.html
[2] https://cran.r-project.org/doc/manuals/r-release/R-admin.html

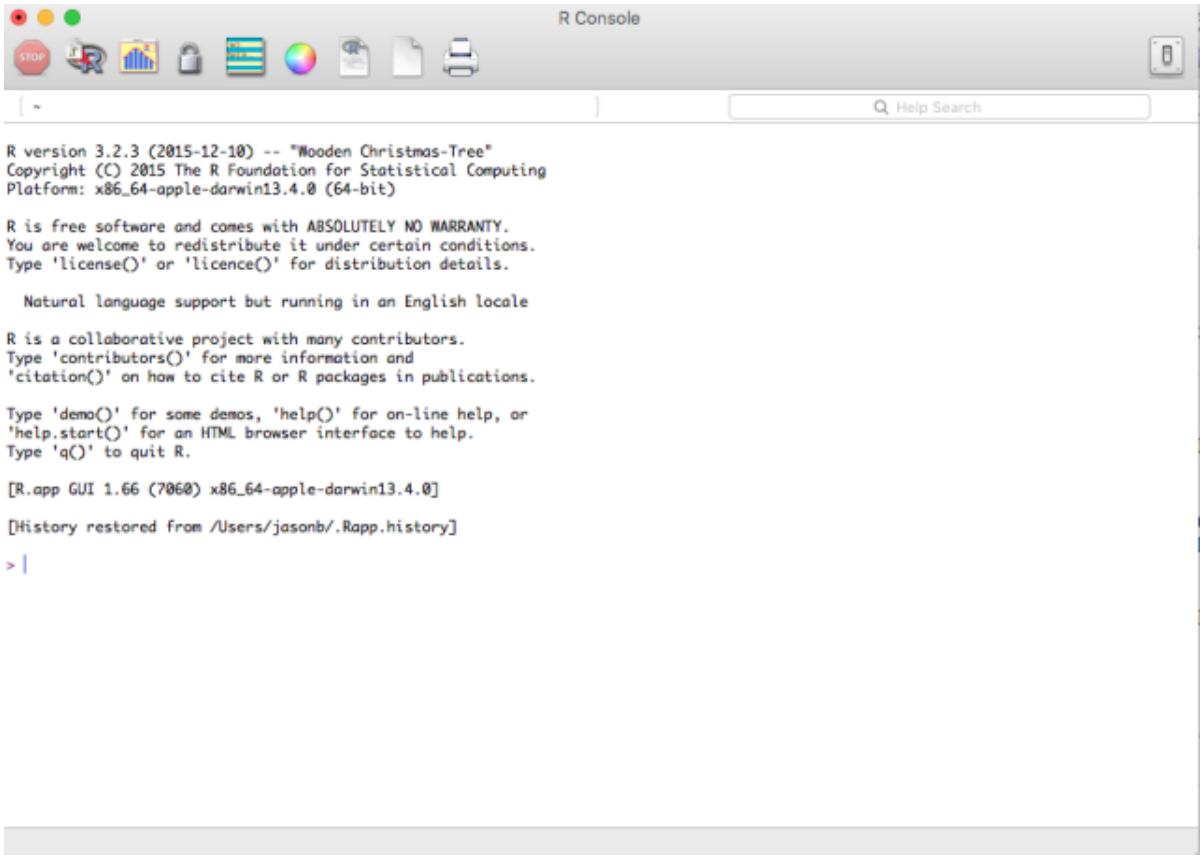

Figure 3.2: Screenshot of the R interactive environment launched from a menu.

In practice, I prefer the command line. It works everywhere and looks the same everywhere. Open your command line, change (or create) to your project directory and start R by typing:

```
R
```

Listing 3.1: The R command on the command line.

You should see something like the screenshot below in your terminal.

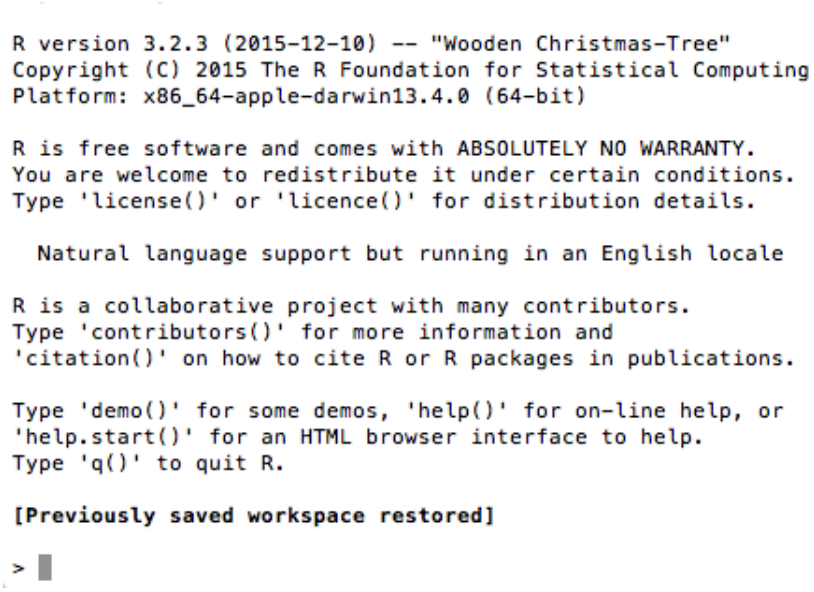

Figure 3.3: Screenshot of the R interactive environment at the command line.

You can close the interactive environment by calling the quit function `q()`. The R interactive environment is very useful for exploring and learning how to use packages and functions. You should spend a lot of time in the interactive environment when you are just starting out. The environment is also very good if you are exploring a new problem and trying what-if scenarios. It is also great if you want to use a systematic process and come up with a prototype model very quickly without the full rigmarole. I recommend that you use the R interactive environment while working through all of the lessons in this book.

3.3 R Scripts

You can save your R commands into a file with a `.R` extension. These are called R scripts. R scripts can be run from the command line, called from shell scripts and (my personal favorite) called from targets in a `Makefile`.

Below is an R script. Type this or copy-and-paste it into a new file and save it into your working directory as `your_script.R`. This simple script loads a standard machine learning dataset called `iris` (the iris flowers dataset) and summarizes all of the attributes. You will learn a lot more about these commands in later lessons.

```
data(iris)
summary(iris)
```

Listing 3.2: Sample content for your custom R script.

Open your terminal (command line) and change directory to wherever you saved the `your_script.R` file. Type the following command:

```
R CMD BATCH your_script.R your_script.log
```

Listing 3.3: Example of running your R script from the command line.

This command runs the script file `your_script.R` using R in a batch mode (non-interactively) and saves any results in the file `your_script.log`. Take a look in the log file `your_script.log`

and you will see the output of the two commands from your script, as well as some execution timing information (shown below).

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

> proc.time()
   user  system elapsed
  3.255   0.117   3.364
```

Listing 3.4: Output from running your R script.

I recommend that if you have a large machine learning project that you develop scripts. I do not recommend that you use scripts to work through the lessons in this book. Each task in your project could be described in a new script which can be documented, updated and tracked in revision control. I would also recommend using `make` with a `Makefile` (or similar build system) and create targets to call each of your scripts. This will ensure that the steps of your project are independent, repeatable, and reusable on future projects.

3.4 Summary

This lesson gave you a crash course in how to install and start R, just in case you are completely new to the platform. In this lesson you learned:

1. How to download and install R.
2. How to start the R interactive environment.
3. How to run an R script.

3.4.1 Next Step

In the next lesson you will get a crash course in the R programming language, designed specifically to get a developer like you up to speed with R very fast.

Chapter 4

Crash Course in R For Developers

As a developer you can pick up the R programming language fast. You don't need to know much about a new language to be able to read and understand code snippets and writing your own small scripts and programs. Learning R is not about learning the language, it is about learning the packages.

In this lesson you will discover the basic syntax, data structures and control structures that you need to know to start reading and writing R scripts. After completing this lesson you will know:

- How to assign variables in R.
- How to work with basic data structures in R like vectors, lists and data frames.
- How to use basic flow control structures like loops in R.
- How to work with functions and packages in R.

This lesson will refresh your R knowledge or teach you just enough to be able to understand the R code used throughout the rest of this book. Let's get started.

4.1 R Syntax is Different, But The Same

The syntax in R looks confusing, but only to begin with. It is a LISP-style language inspired by an even older language (S). The assignment syntax is probably the strangest thing you will see. Assignment uses the arrow (`<-`) rather than a single equals (`=`). Also, like many programming languages, R syntax is case sensitive.

R has all of your familiar control flow structures like if-then-else, for-loops and while loops. Comments in R use the hash (`#`) character. You can create your own functions and packages of helper functions for your scripts. If you have done any scripting before, like JavaScript, Python, Ruby, BASH or similar, then you will pick up R very quickly.

4.1.1 You Can Already Program, Just Learn the R Syntax

As a developer, you already know how to program. This means that you can take a problem and think up the type of procedure and data structures you need to implement it. The language

you are using is just a detail. You only need to map your idea of the solution onto the specifics of the language you are using. This is how you can begin using R very quickly. To get started, you need to know the absolute basics, such as:

- How do you assign data to variables?
- How do you work with different data types?
- How do you work with the data structures for handling data?
- How do you use the standard flow control structures?
- How do you work with functions and third-party packages?

This chapter is designed to answer these language jump-start questions for you.

4.2 Assignment

The assignment operator in R is the arrow operator (<-). Below are examples of assigning an integer, double, string and a boolean, and printing each out to the console in turn.

```
> # integer
> i <- 23
> i
[1] 23

> # double
> d <- 2.3
> d
[1] 2.3

> # string
> s <- 'hello world'
> s
[1] "hello world"

> # boolean
> b <- TRUE
> b
[1] TRUE
```

Listing 4.1: Examples of variable assignment in R.

Remember, do not use equals (=) for assignment. It is the biggest mistake new R programmers make.

4.3 Data Structures

Let's take a look at some of the most common and most useful data structures in R.

4.3.1 Vectors

Vectors are lists of data that can be the same or different types:

```
> # create a vector using the c() function
> v <- c(98, 99, 100)
> v
[1]  98  99 100
> v[1:2]
[1] 98 99

> # create a vector from a range of integers
> r <- (1:10)
> r
 [1]  1  2  3  4  5  6  7  8  9 10
> r[5:10]
[1]  5  6  7  8  9 10

> # add a new item to the end of a vector
> v <- c(1, 2, 3)
> v[4] <- 4
> v
[1] 1 2 3 4
```

Listing 4.2: Examples working with vectors in R.

Notice that vectors are 1-indexed (indexes start at 1 not 0). You will use the `c()` function a lot to combine variables into a vector.

4.3.2 Lists

Lists provide a group of named items, not unlike a map.

```
# create a list of named items
a <- list(aa=1, bb=2, cc=3)
a
a$aa

# add a named item to a list
a$dd=4
a
```

Listing 4.3: Examples of working with lists in R.

You can define a new list with the `list()` function. A list can be initialized with values or empty. Note that the named values in the list can be accessed using the dollar operator (`$`). Once referenced, they can be read or written. This is also how new items can be added to the list.

4.3.3 Matrices

A matrix is a table of data of the same type (e.g. numeric). It has dimensions (rows and columns) and the columns can be named.

```
# Create a 2-row, 3-column matrix with named headings
```

```
> data <- c(1, 2, 3, 4, 5, 6)
> headings <- list(NULL, c("a","b","c"))
> m <- matrix(data, nrow=2, ncol=3, byrow=TRUE, dimnames=headings)
> m
     a b c
[1,] 1 2 3
[2,] 4 5 6

> m[1,]
a b c
1 2 3

> m[,1]
[1] 1 4
```

Listing 4.4: Examples of working with a martix in R.

A lot of useful plotting and machine learning algorithms require the data to be provided as a matrix. Note the syntax to index into rows `[1,]` and columns `[,1]` of a matrix.

4.3.4 Data Frame

Data frames are useful for representing tables of your data in R.

```
# create a new data frame
years <- c(1980, 1985, 1990)
scores <- c(34, 44, 83)
df <- data.frame(years, scores)
df[,1]
df$years
```

Listing 4.5: Examples of working with a Data Frame in R.

A matrix is a much simpler structure, intended for mathematical operations. A data frame is more suited to representing a table of data and is expected by modern implementations of machine learning algorithms in R. They provide useful convenience functions and properties. Note that you can index into rows and columns of a data frame just like you can for a matrix. Also note that you can reference a column using its name just like a list (`df$years`).

4.4 Flow Control

Now, let's take a look at some standard flow control structures in R. As a developer, these are all very familiar and self explanatory.

4.4.1 If-Then-Else

```
# if then else
a <- 66
if (a > 55) {
  print("a is more than 55")
} else {
  print("A is less than or equal to 55")
```

```
}

[1] "a is more than 55"
```

Listing 4.6: Example of an If-Then-Else structure in R.

4.4.2 For Loop

```
# for loop
mylist <- c(55, 66, 77, 88, 99)
for (value in mylist) {
  print(value)
}

[1] 55
[1] 66
[1] 77
[1] 88
[1] 99
```

Listing 4.7: Example of a For-Loop in R.

4.4.3 While Loop

```
# while loop
a <- 100
while (a < 500) {
  a <- a + 100
}
a

[1] 500
```

Listing 4.8: Example of a While-Loop in R.

4.5 Functions

Functions let you group code and call it repeatedly with arguments. You have already used one function, the `c()` function for combining objects into a vector for example. R has many built in functions and additional functions can be provided by installing and loading third-party packages. Here is an example of using a statistical function to calculate the mean of a vector of numbers:

```
# call function to calculate the mean on a vector of integers
numbers <- c(1, 2, 3, 4, 5, 6)
mean(numbers)

[1] 3.5
```

Listing 4.9: Example of calling the mean() function in R.

You can get help for a function in R by using the question mark operator (`?`) followed by the function name. You can also call the `help()` function and passing the function name you need help with as the operator.

```
# help with the mean() function
?mean
help(mean)
```

Listing 4.10: Example of getting help for the mean() function in R.

Often you just want to know what arguments a function takes. To check, you can call the `args()` function and pass the function name as an argument.

```
# function arguments
args(mean)
```

Listing 4.11: Example of showing the arguments for the mean() function in R.

You can get example usage of a function by calling the `example()` function and passing the name of the function as an argument.

```
# example usage of the mean function
example(mean)
```

Listing 4.12: Example of example usage of the mean() function in R.

You can define your own functions that may or may not take arguments or return a result. Below is an example of a custom function to calculate and return the sum of three numbers:

```
# define custom function
mysum <- function(a, b, c) {
  sum <- a + b + c
  return(sum)
}
# call custom function
mysum(1,2,3)

[1] 6
```

Listing 4.13: Example of defining and calling a custom function in R.

4.6 Packages

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[1].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 the `caret` package which is very useful for machine learning in R and will be used in later lessons:

```
# install the package
install.packages("caret")
```

Listing 4.14: Example of installing the caret package in R.

[1] https://cran.r-project.org/

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 `caret` package as follows:

```
# load the package
library(caret)
```

Listing 4.15: Example of loading the caret package in R.

A package can provide a lot of new functions. You can read up on a package on its CRAN page, but you can also get help for the package within R by calling the `library()` function with the argument `help="PackageName"`. For example, you can get help for the `caret` package as follows:

```
# help for the package
library(help="caret")
```

Listing 4.16: Example of getting help with the caret package in R.

Before we move on, let's install the `caret` package and all suggested dependent packages. These packages will be needed throughout the rest of the book.

```
# Install package with dependencies
install.packages("caret", dependencies = c("Depends", "Suggests"))
```

Listing 4.17: Install the caret packages and suggested dependencies.

This may take some time to complete, but will ensure you have everything you need for subsequent lessons.

4.7 5 Things To Remember

Here are five quick tips to remember when getting started in R:

- **Assignment**: R uses the arrow operator (`<-`) for assignment, not a single equals (`=`).
- **Case Sensitive**: The R language is case sensitive, meaning that `C()` and `c()` are two different function calls.
- **Help**: You can get help on any operator or function using the `help()` function or the question mark operator `?`.
- **How To Quit**: You can exit the R interactive environment by calling the question function `q()`.
- **Documentation**: R installs with a lot of useful documentation. You can review it in the browser by typing `help.start()` in your R environment.

4.8 Summary

In this lesson you took a crash course in basic R syntax and got up to speed with the R programming language, including:

1. Variable Assignment.
2. Data Structures.
3. Flow Control.
4. Functions.
5. Packages.

As a developer, you now know enough to read other people's R scripts. You discovered that learning R is not about learning the language, but it is about learning how to use R packages. You also have the tools to start writing your own scripts in the R interactive environment.

4.8.1 Next Step

Now that you know some basic R syntax, it is time to learn how to load data. In the next lesson you will discover how you can very quickly and easily load standard machine learning datasets in R.

Chapter 5

Standard Machine Learning Datasets

It is invaluable to load standard datasets in R so that you can test, practice and experiment with machine learning techniques and improve your skill with the platform. In this lesson you will discover how you can load standard classification and regression datasets in R. After completing this lesson you will know:

1. Three R packages that you can use to load standard machine learning datasets.
2. Ten specific standard datasets that you can use to practice machine learning.

This is an important lesson because the 10 datasets that you will learn how to load will be used in all of the examples throughout the rest of the book. Let's get started.

5.1 Practice On Small Well-Understood Datasets

There are hundreds of standard test datasets that you can use to practice and get better at machine learning. Most of them are hosted for free on the UCI Machine Learning Repository[1]. These datasets are useful because they are well understood, they are well behaved and they are small. This last point is critical when practicing machine learning because:

- You can download them fast.
- You can fit them into memory easily.
- You can run algorithms on them quickly.

Please note, it is assumed that your R environment is setup with all required packages for this and subsequent lessons. If you have not installed the `caret` package and dependent packages see Section 4.6 in the previous lesson.

[1] http://archive.ics.uci.edu/ml

5.1.1 Access Standard Datasets in R

You can load the standard datasets into R as CSV files and you will discover how to do this in the next lesson. There is a more convenient approach to loading a standard dataset. They have been packaged and are available in third party R packages that you can download from the Comprehensive R Archive Network (CRAN).

Which packages should you use and what datasets are good to start with? In the following sections you will discover three packages that you can use to get access to standard machine learning datasets. You will also discover specific classification and regression problems that you can load and use to practice machine learning in R.

5.2 Package: datasets

The `datasets` package comes with base R which means you do not need to explicitly load the package. It includes a large number of datasets that you can use. You can load a dataset from this package by typing:

```
data(DataSetName)
```

Listing 5.1: Attach a dataset to your R session.

To see a list of the datasets available in this package, you can type:

```
# list all datasets in the package
library(help = "datasets")
```

Listing 5.2: Help on the datasets package.

Some datasets from this package that you could use are listed below.

5.2.1 Iris Flowers Dataset

- Description: Predict iris flower species from flower measurements.
- Type: Multiclass classification.
- Dimensions: 150 instances, 5 attributes.
- Inputs: Numeric.
- Output: Categorical, 3 class labels.
- UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Iris.
- Published accuracy results: http://www.is.umk.pl/projects/rules.html#Iris.

```
# iris flowers datasets
data(iris)
head(iris)
```

Listing 5.3: Load the Iris dataset and display some rows.

You will see:

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Listing 5.4: Peek at the iris dataset.

5.2.2 Longley's Economic Regression Data

- Description: Predict number of people employed from economic variables.
- Type: Regression.
- Dimensions: 16 instances, 7 attributes.
- Inputs: Numeric.
- Output: Numeric.

```
# Longley's Economic Regression Data
data(longley)
head(longley)
```

Listing 5.5: Load the Longley dataset and look at some rows.

You will see:

```
     GNP.deflator    GNP Unemployed Armed.Forces Population Year Employed
1947         83.0 234.289      235.6        159.0    107.608 1947   60.323
1948         88.5 259.426      232.5        145.6    108.632 1948   61.122
1949         88.2 258.054      368.2        161.6    109.773 1949   60.171
1950         89.5 284.599      335.1        165.0    110.929 1950   61.187
1951         96.2 328.975      209.9        309.9    112.075 1951   63.221
1952         98.1 346.999      193.2        359.4    113.270 1952   63.639
```

Listing 5.6: Output from the Longley dataset.

5.3 Package: mlbench

Direct from the `mlbench` package manual:

> A collection of artificial and real-world machine learning benchmark problems, including, e.g., several data sets from the UCI repository.

You can learn more about the `mlbench` package on the `mlbench` CRAN page[2]. If not installed, you can install this package as follows:

[2] https://cran.r-project.org/web/packages/mlbench/index.html

```
install.packages("mlbench")
```

Listing 5.7: Install the mlbench package.

You can load the package as follows:

```
# load the package
library(mlbench)
```

Listing 5.8: Load the mlbench package.

To see a list of the datasets available in this package, you can type:

```
# list the contents of the package
library(help = "mlbench")
```

Listing 5.9: Get help on the mlbench package.

Some highlight datasets from this package that you could use are:

5.3.1 Boston Housing Data

- Description: Predict the median house price in 1000 for suburbs in Boston.
- Type: Regression.
- Dimensions: 506 instances, 14 attributes.
- Inputs: Numeric.
- Output: Numeric.
- UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Housing.

```
# Boston Housing Data
data(BostonHousing)
head(BostonHousing)
```

Listing 5.10: Load the Boston Housing dataset.

You will see:

```
     crim zn indus chas nox    rm  age    dis rad tax ptratio    b lstat medv
1 0.00632 18 2.31   0 0.538 6.575 65.2 4.0900 1 296   15.3 396.90 4.98 24.0
2 0.02731 0  7.07   0 0.469 6.421 78.9 4.9671 2 242   17.8 396.90 9.14 21.6
3 0.02729 0  7.07   0 0.469 7.185 61.1 4.9671 2 242   17.8 392.83 4.03 34.7
4 0.03237 0  2.18   0 0.458 6.998 45.8 6.0622 3 222   18.7 394.63 2.94 33.4
5 0.06905 0  2.18   0 0.458 7.147 54.2 6.0622 3 222   18.7 396.90 5.33 36.2
6 0.02985 0  2.18   0 0.458 6.430 58.7 6.0622 3 222   18.7 394.12 5.21 28.7
```

Listing 5.11: Sample output of the Boston Housing dataset.

5.3.2 Wisconsin Breast Cancer Database

- Description: Predict whether a tissue sample is malignant or benign given properties about the tissue sample.
- Type: Binary Classification.
- Dimensions: 699 instances, 11 attributes.
- Inputs: Integer (Nominal).
- Output: Categorical, 2 class labels.
- UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Original).
- Published accuracy results: http://www.is.umk.pl/projects/datasets.html#Wisconsin.

```
# Wisconsin Breast Cancer Database
data(BreastCancer)
head(BreastCancer)
```

Listing 5.12: Load the Breast Cancer dataset.

You will see:

```
      Id Cl.thickness Cell.size Cell.shape Marg.adhesion Epith.c.size Bare.nuclei
         Bl.cromatin Normal.nucleoli Mitoses Class
1 1000025           5           1           1            1           2           1           3
                1       1    benign
2 1002945           5           4           4            5           7          10           3
                2       1    benign
3 1015425           3           1           1            1           2           2           3
                1       1    benign
4 1016277           6           8           8            1           3           4           3
                7       1    benign
5 1017023           4           1           1            3           2           1           3
                1       1    benign
6 1017122           8          10          10            8           7          10           9
                7       1 malignant
```

Listing 5.13: Sample output of the Breast Cancer dataset.

5.3.3 Glass Identification Database

- Description: Predict the glass type from chemical properties.
- Type: Regression.
- Dimensions: 214 instances, 10 attributes.
- Inputs: Numeric.
- Output: Categorical, 7 class labels.

- UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Glass+Identification.
- Published accuracy results: http://www.is.umk.pl/projects/datasets.html#Other.

```
# Glass Identification Database
data(Glass)
head(Glass)
```

Listing 5.14: Load the Glass dataset.

You will see:

```
      RI    Na   Mg   Al    Si    K   Ca Ba   Fe Type
1 1.52101 13.64 4.49 1.10 71.78 0.06 8.75 0 0.00 1
2 1.51761 13.89 3.60 1.36 72.73 0.48 7.83 0 0.00 1
3 1.51618 13.53 3.55 1.54 72.99 0.39 7.78 0 0.00 1
4 1.51766 13.21 3.69 1.29 72.61 0.57 8.22 0 0.00 1
5 1.51742 13.27 3.62 1.24 73.08 0.55 8.07 0 0.00 1
6 1.51596 12.79 3.61 1.62 72.97 0.64 8.07 0 0.26 1
```

Listing 5.15: Sample output of the Glass dataset.

5.3.4 Johns Hopkins University Ionosphere database

- Description: Predict high-energy structures in the atmosphere from antenna data.
- Type: Regression.
- Dimensions: 351 instances, 35 attributes.
- Inputs: Numeric.
- Output: Categorical, 2 class labels.
- UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Ionosphere.
- Published accuracy results: http://www.is.umk.pl/projects/datasets.html#Ionosphere.

```
# Johns Hopkins University Ionosphere database
data(Ionosphere)
head(Ionosphere)
```

Listing 5.16: Load the Ionosphere dataset.

You will see:

```
  V1 V2     V3      V4      V5      V6      V7      V8     V9     V10     V11     V12
      V13     V14     V15     V16     V17     V18     V19
1  1  0 0.99539 -0.05889 0.85243 0.02306 0.83398 -0.37708 1.00000 0.03760 0.85243 -0.17755
    0.59755 -0.44945 0.60536 -0.38223 0.84356 -0.38542 0.58212
2  1  0 1.00000 -0.18829 0.93035 -0.36156 -0.10868 -0.93597 1.00000 -0.04549 0.50874
    -0.67743 0.34432 -0.69707 -0.51685 -0.97515 0.05499 -0.62237 0.33109
3  1  0 1.00000 -0.03365 1.00000 0.00485 1.00000 -0.12062 0.88965 0.01198 0.73082 0.05346
    0.85443 0.00827 0.54591 0.00299 0.83775 -0.13644 0.75535
```

```
4  1  0 1.00000 -0.45161 1.00000 1.00000 0.71216 -1.00000 0.00000 0.00000 0.00000 0.00000
    0.00000 0.00000 -1.00000 0.14516 0.54094 -0.39330 -1.00000
5  1  0 1.00000 -0.02401 0.94140 0.06531 0.92106 -0.23255 0.77152 -0.16399 0.52798 -0.20275
    0.56409 -0.00712 0.34395 -0.27457 0.52940 -0.21780 0.45107
6  1  0 0.02337 -0.00592 -0.09924 -0.11949 -0.00763 -0.11824 0.14706 0.06637 0.03786
    -0.06302 0.00000 0.00000 -0.04572 -0.15540 -0.00343 -0.10196 -0.11575
      V20      V21      V22      V23      V24      V25      V26      V27      V28      V29      V30
               V31      V32      V33      V34 Class
1 -0.32192 0.56971 -0.29674 0.36946 -0.47357 0.56811 -0.51171 0.41078 -0.46168 0.21266
    -0.34090 0.42267 -0.54487 0.18641 -0.45300 good
2 -1.00000 -0.13151 -0.45300 -0.18056 -0.35734 -0.20332 -0.26569 -0.20468 -0.18401 -0.19040
    -0.11593 -0.16626 -0.06288 -0.13738 -0.02447 bad
3 -0.08540 0.70887 -0.27502 0.43385 -0.12062 0.57528 -0.40220 0.58984 -0.22145 0.43100
    -0.17365 0.60436 -0.24180 0.56045 -0.38238 good
4 -0.54467 -0.69975 1.00000 0.00000 0.00000 1.00000 0.90695 0.51613 1.00000 1.00000
    -0.20099 0.25682 1.00000 -0.32382 1.00000 bad
5 -0.17813 0.05982 -0.35575 0.02309 -0.52879 0.03286 -0.65158 0.13290 -0.53206 0.02431
    -0.62197 -0.05707 -0.59573 -0.04608 -0.65697 good
6 -0.05414 0.01838 0.03669 0.01519 0.00888 0.03513 -0.01535 -0.03240 0.09223 -0.07859
    0.00732 0.00000 0.00000 -0.00039 0.12011 bad
```

Listing 5.17: Sample output of the Ionosphere dataset.

5.3.5 Pima Indians Diabetes Database

- Description: Predict the onset of diabetes in female Pima Indians from medical record data.
- Type: Binary Classification.
- Dimensions: 768 instances, 9 attributes.
- Inputs: Numeric.
- Output: Categorical, 2 class labels.
- UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Pima+Indians+Diabetes.
- Published accuracy results: http://www.is.umk.pl/projects/datasets.html#Diabetes.

```
# Pima Indians Diabetes Database
data(PimaIndiansDiabetes)
head(PimaIndiansDiabetes)
```

Listing 5.18: Load the Pima Indians Diabetes dataset.

You will see:

```
  pregnant glucose pressure triceps insulin mass pedigree age diabetes
1        6     148       72      35       0 33.6    0.627  50      pos
2        1      85       66      29       0 26.6    0.351  31      neg
3        8     183       64       0       0 23.3    0.672  32      pos
4        1      89       66      23      94 28.1    0.167  21      neg
```

```
5        0     137       40       35     168 43.1   2.288 33      pos
6        5     116       74        0       0 25.6   0.201 30      neg
```

Listing 5.19: Sample output of the Pima Indians Diabetes dataset.

5.3.6 Sonar, Mines vs. Rocks Dataset

- Description: Predict metal or rock returns from sonar return data.
- Type: Binary Classification.
- Dimensions: 208 instances, 61 attributes.
- Inputs: Numeric.
- Output: Categorical, 2 class labels.
- UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks).
- Published accuracy results: http://www.is.umk.pl/projects/datasets.html#Sonar.

```
# Sonar, Mines vs. Rocks
data(Sonar)
head(Sonar)
```

Listing 5.20: Load the Sonar dataset.

You will see:

```
      V1     V2     V3     V4     V5     V6     V7     V8     V9    V10    V11    V12    V13    V14
            V15    V16    V17    V18    V19    V20    V21    V22
1 0.0200 0.0371 0.0428 0.0207 0.0954 0.0986 0.1539 0.1601 0.3109 0.2111 0.1609 0.1582
    0.2238 0.0645 0.0660 0.2273 0.3100 0.2999 0.5078 0.4797 0.5783 0.5071
2 0.0453 0.0523 0.0843 0.0689 0.1183 0.2583 0.2156 0.3481 0.3337 0.2872 0.4918 0.6552
    0.6919 0.7797 0.7464 0.9444 1.0000 0.8874 0.8024 0.7818 0.5212 0.4052
3 0.0262 0.0582 0.1099 0.1083 0.0974 0.2280 0.2431 0.3771 0.5598 0.6194 0.6333 0.7060
    0.5544 0.5320 0.6479 0.6931 0.6759 0.7551 0.8929 0.8619 0.7974 0.6737
4 0.0100 0.0171 0.0623 0.0205 0.0205 0.0368 0.1098 0.1276 0.0598 0.1264 0.0881 0.1992
    0.0184 0.2261 0.1729 0.2131 0.0693 0.2281 0.4060 0.3973 0.2741 0.3690
5 0.0762 0.0666 0.0481 0.0394 0.0590 0.0649 0.1209 0.2467 0.3564 0.4459 0.4152 0.3952
    0.4256 0.4135 0.4528 0.5326 0.7306 0.6193 0.2032 0.4636 0.4148 0.4292
6 0.0286 0.0453 0.0277 0.0174 0.0384 0.0990 0.1201 0.1833 0.2105 0.3039 0.2988 0.4250
    0.6343 0.8198 1.0000 0.9988 0.9508 0.9025 0.7234 0.5122 0.2074 0.3985
     V23    V24    V25    V26    V27    V28    V29    V30    V31    V32    V33    V34    V35    V36
           V37    V38    V39    V40    V41    V42    V43    V44
1 0.4328 0.5550 0.6711 0.6415 0.7104 0.8080 0.6791 0.3857 0.1307 0.2604 0.5121 0.7547
    0.8537 0.8507 0.6692 0.6097 0.4943 0.2744 0.0510 0.2834 0.2825 0.4256
2 0.3957 0.3914 0.3250 0.3200 0.3271 0.2767 0.4423 0.2028 0.3788 0.2947 0.1984 0.2341
    0.1306 0.4182 0.3835 0.1057 0.1840 0.1970 0.1674 0.0583 0.1401 0.1628
3 0.4293 0.3648 0.5331 0.2413 0.5070 0.8533 0.6036 0.8514 0.8512 0.5045 0.1862 0.2709
    0.4232 0.3043 0.6116 0.6756 0.5375 0.4719 0.4647 0.2587 0.2129 0.2222
4 0.5556 0.4846 0.3140 0.5334 0.5256 0.2520 0.2090 0.3559 0.6260 0.7340 0.6120 0.3497
    0.3953 0.3012 0.5408 0.8814 0.9857 0.9167 0.6121 0.5006 0.3210 0.3202
```

```
5 0.5730 0.5399 0.3161 0.2285 0.6995 1.0000 0.7262 0.4724 0.5103 0.5459 0.2881 0.0981
    0.1951 0.4181 0.4604 0.3217 0.2828 0.2430 0.1979 0.2444 0.1847 0.0841
6 0.5890 0.2872 0.2043 0.5782 0.5389 0.3750 0.3411 0.5067 0.5580 0.4778 0.3299 0.2198
    0.1407 0.2856 0.3807 0.4158 0.4054 0.3296 0.2707 0.2650 0.0723 0.1238
     V45    V46    V47    V48    V49    V50    V51    V52    V53    V54    V55    V56    V57    V58
           V59    V60 Class
1 0.2641 0.1386 0.1051 0.1343 0.0383 0.0324 0.0232 0.0027 0.0065 0.0159 0.0072 0.0167
    0.0180 0.0084 0.0090 0.0032 R
2 0.0621 0.0203 0.0530 0.0742 0.0409 0.0061 0.0125 0.0084 0.0089 0.0048 0.0094 0.0191
    0.0140 0.0049 0.0052 0.0044 R
3 0.2111 0.0176 0.1348 0.0744 0.0130 0.0106 0.0033 0.0232 0.0166 0.0095 0.0180 0.0244
    0.0316 0.0164 0.0095 0.0078 R
4 0.4295 0.3654 0.2655 0.1576 0.0681 0.0294 0.0241 0.0121 0.0036 0.0150 0.0085 0.0073
    0.0050 0.0044 0.0040 0.0117 R
5 0.0692 0.0528 0.0357 0.0085 0.0230 0.0046 0.0156 0.0031 0.0054 0.0105 0.0110 0.0015
    0.0072 0.0048 0.0107 0.0094 R
6 0.1192 0.1089 0.0623 0.0494 0.0264 0.0081 0.0104 0.0045 0.0014 0.0038 0.0013 0.0089
    0.0057 0.0027 0.0051 0.0062 R
```

Listing 5.21: Sample output of the Sonar dataset.

5.3.7 Soybean Database

- Description: Predict problems with soybean crops from crop data.
- Type: Multiclass Classification.
- Dimensions: 683 instances, 26 attributes.
- Inputs: Integer (Nominal).
- Output: Categorical, 19 class labels.
- UCI Machine Learning Repository: http://archive.ics.uci.edu/ml/datasets/Soybean+%28Large%29.

```
# Soybean Database
data(Soybean)
head(Soybean)
```

Listing 5.22: Load the Soybean dataset.

You will see:

```
Class date plant.stand precip temp hail crop.hist area.dam sever seed.tmt germ plant.growth
    leaves leaf.halo leaf.marg leaf.size leaf.shread
1 diaporthe-stem-canker 6          0      2    1    0         1        1     1        0    0
              1      1         0         2         2          0
2 diaporthe-stem-canker 4          0      2    1    0         2        0     2        1    1
              1      1         0         2         2          0
3 diaporthe-stem-canker 3          0      2    1    0         1        0     2        1    2
              1      1         0         2         2          0
4 diaporthe-stem-canker 3          0      2    1    0         1        0     2        0    1
              1      1         0         2         2          0
```

```
5 diaporthe-stem-canker 6          0     2    1    0        2       0    1       0    2
              1     1        0        2        2         0
6 diaporthe-stem-canker 5          0     2    1    0        3       0    1       0    1
              1     1        0        2        2         0
  leaf.malf leaf.mild stem lodging stem.cankers canker.lesion fruiting.bodies ext.decay
      mycelium int.discolor sclerotia fruit.pods fruit.spots seed mold.growth
1         0        0    1       1           3            1              1        1        0
              0        0         0          4    0          0
2         0        0    1       0           3            1              1        1        0
              0        0         0          4    0          0
3         0        0    1       0           3            0              1        1        0
              0        0         0          4    0          0
4         0        0    1       0           3            0              1        1        0
              0        0         0          4    0          0
5         0        0    1       0           3            1              1        1        0
              0        0         0          4    0          0
6         0        0    1       0           3            0              1        1        0
              0        0         0          4    0          0
  seed.discolor seed.size shriveling roots
1             0         0          0     0
2             0         0          0     0
3             0         0          0     0
4             0         0          0     0
5             0         0          0     0
6             0         0          0     0
```

Listing 5.23: Sample output of the Soybean dataset.

5.4 Package: AppliedPredictiveModeling

Many books that use R also include their own R package that provides all of the code and datasets used in the book.The excellent book Applied Predictive Modeling has its own package called `AppliedPredictiveModeling`. If not installed, you can install this package as follows:

```
install.packages("AppliedPredictiveModeling")
```

Listing 5.24: Install the AppliedPredictiveModeling package.

You can load the `AppliedPredictiveModeling` package as follows:

```
# load the package
library(AppliedPredictiveModeling)
```

Listing 5.25: Load the AppliedPredictiveModeling package.

Below is an example standard regression dataset from this package.

5.4.1 Abalone Data

- Description: Predict abalone age from abalone measurement data.
- Type: Regression.
- Dimensions: 4177 instances, 9 attributes.

- Inputs: Numerical and categorical.
- Output: Integer.
- UCI Machine Learning Repository: http://archive.ics.uci.edu/ml/datasets/Abalone

```
# Abalone Data
data(abalone)
head(abalone)
```

Listing 5.26: Load the Abalone dataset.

You will see:

```
  Type LongestShell Diameter Height WholeWeight ShuckedWeight VisceraWeight ShellWeight
     Rings
1    M       0.455    0.365  0.095     0.5140       0.2245       0.1010      0.150   15
2    M       0.350    0.265  0.090     0.2255       0.0995       0.0485      0.070    7
3    F       0.530    0.420  0.135     0.6770       0.2565       0.1415      0.210    9
4    M       0.440    0.365  0.125     0.5160       0.2155       0.1140      0.155   10
5    I       0.330    0.255  0.080     0.2050       0.0895       0.0395      0.055    7
6    I       0.425    0.300  0.095     0.3515       0.1410       0.0775      0.120    8
```

Listing 5.27: Sample output of the Abalone dataset.

5.5 Summary

In this lesson you discovered that you do not need to collect or load your own data in order to practice machine learning in R. You learned about 3 different packages that provide sample machine learning datasets that you can use:

- The `datasets` package.
- The `mlbench` package.
- The `AppliedPredictiveModeling` package.

You also discovered 10 specific standard machine learning datasets that you can use to practice classification and regression machine learning techniques.

- Iris flowers datasets (multiclass classification).
- Longley's Economic Regression Data (regression).
- Boston Housing Data (regression).
- Wisconsin Breast Cancer Database (binary classification).
- Glass Identification Database (multiclass classification).
- Johns Hopkins University Ionosphere database (binary classification).
- Pima Indians Diabetes Database (binary classification).

- Sonar, Mines vs. Rocks (binary classification).
- Soybean Database (multiclass classification).
- Abalone Data (regression).

5.5.1 Next Step

You have discovered that you do not need your own datasets to start practicing machine learning in R. In the next lesson you will discover how you can load your own data into R, when the time comes.

Chapter 6

Load Your Machine Learning Datasets

You need to be able to load data into R when working on a machine learning problem. In this lesson, you will discover how you can load your data files into R and start your machine learning project. After completing this lesson, you will know:

1. How to load your own data into R from local CSV files.
2. How to load your own data into R from CSV files located on the web.

Let's get started.

6.1 Access To Your Data

The most common way to work with data in machine learning is in CSV (comma separated value) files. Data may originally be stored in all manner of formats and diverse locations. For example:

- Relational database tables.
- XML files.
- JSON files.
- Fixed-width formatted file.
- Spreadsheet file (e.g. Microsoft Office).

You need to consolidate your data into a single file with rows and columns before you can work with it on a machine learning project. The standard format for representing a machine learning dataset is a CSV file. This is because machine learning algorithms, for the most part, work with data in tabular format (e.g. a matrix of input and output vectors). Datasets in R are often represented as a matrix or data frame structure. The first step of a machine learning project in R is loading your data into R as a matrix or data frame.

6.2 Load Data From CSV File

This example shows the loading of the iris dataset from a CSV file. This recipe will load a CSV file without a header (e.g. column names) located in the current directory into R as a data frame. The iris data set can be loaded from the UCI Machine Learning repository[1] and saved in your local directory as `iris.csv`.

```
# define the filename
filename <- "iris.csv"
# load the CSV file from the local directory
dataset <- read.csv(filename, header=FALSE)
# preview the first 5 rows
head(dataset)
```

Listing 6.1: Load data from a CSV file.

Running this recipe, you will see:

```
  V1  V2  V3  V4          V5
1 5.1 3.5 1.4 0.2 Iris-setosa
2 4.9 3.0 1.4 0.2 Iris-setosa
3 4.7 3.2 1.3 0.2 Iris-setosa
4 4.6 3.1 1.5 0.2 Iris-setosa
5 5.0 3.6 1.4 0.2 Iris-setosa
6 5.4 3.9 1.7 0.4 Iris-setosa
```

Listing 6.2: Output of loading data from a CSV file.

This recipe is useful if you want to store the data locally with your R scripts, such as in a project managed under revision control. If the data is not in your local directory, you can either:

1. Specify the full path to the dataset on your local file system.
2. Use the `setwd()` function to set your current working directory to where the dataset is located

6.3 Load Data From CSV URL

This example shows the loading of the iris data from a CSV file located on the UCI Machine Learning Repository website. This recipe will load a CSV file without a header from a URL into R as a data frame.

```
# load the package
library(RCurl)
# specify the URL for the Iris data CSV
urlfile <-'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
# download the file
downloaded <- getURL(urlfile, ssl.verifypeer=FALSE)
# treat the text data as a steam so we can read from it
connection <- textConnection(downloaded)
# parse the downloaded data as CSV
dataset <- read.csv(connection, header=FALSE)
# preview the first 5 rows
```

[1] https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data

```
head(dataset)
```

Listing 6.3: Load a CSV file from a URL.

Running this recipe, you will see:

```
  V1  V2  V3  V4         V5
1 5.1 3.5 1.4 0.2 Iris-setosa
2 4.9 3.0 1.4 0.2 Iris-setosa
3 4.7 3.2 1.3 0.2 Iris-setosa
4 4.6 3.1 1.5 0.2 Iris-setosa
5 5.0 3.6 1.4 0.2 Iris-setosa
6 5.4 3.9 1.7 0.4 Iris-setosa
```

Listing 6.4: Output of loading a CSV file from a URL.

This recipe is useful if your dataset is stored on a server, such as on your GitHub account. It is also useful if you want to use datasets from the UCI Machine Learning Repository but do not want to store them locally.

6.4 Summary

In this lesson, you discovered how you can load your data into R. You learned two recipes for loading your data into R:

1. How to load your data from a local CSV file.
2. How to load your data from a CSV file located on a webserver.

6.4.1 Next Step

You have now seen two ways that you can load data into R: standard datasest from R packages and your own CSV files. Now it is time to start looking at some data. In the next lesson you will discover how you can start to understand your data using descriptive statistics.

Chapter 7

Understand Your Data Using Descriptive Statistics

You must become intimate with your data. Any machine learning models that you build are only as good as the data that you provide them. The first step in understanding your data is to actually look at some raw values and calculate some basic statistics. In this lesson you will discover how you can quickly get a handle on your dataset with descriptive statistics examples and recipes in R.

After completing this lesson you will know 8 techniques that you can use in order to learn more about your data:

- How to quickly understand your data by looking at the raw values, dimensions, data types and class distributions.
- How to understand the distribution of your data with summary statistics, standard deviations and skew.
- How to understand the relationships between attributes by calculating correlations.

These recipes are perfect if you only know a little bit about statistics. Let's get started.

7.1 You Must Understand Your Data

Understanding the data that you have is critically important. You can run techniques and algorithms on your data, but it is not until you take the time to truly understand your dataset that you can fully understand the context of the results you achieve.

7.1.1 Better Understanding Equals Better Results

A deeper understanding of your data will give you better results. Taking the time to study the data you have will help you in ways that are less obvious. You build an intuition for the data and for the entities that individual records or observations represent. These can bias you towards specific techniques (for better or worse), but you can also be inspired. For example, examining your data in detail may trigger ideas for specific techniques to investigate:

Data Cleaning : You may discover missing or corrupt data and think of various data cleaning operations to perform such as marking or removing bad data and imputing missing data.

Data Transforms : You may discover that some attributes have familiar distributions such as Gaussian or exponential giving you ideas of scaling or other transforms you could apply.

Data Modeling : You may notice properties of the data such as distributions or data types that suggest the use (or to not use) specific machine learning algorithms.

7.1.2 Use Descriptive Statistics

You need to look at your data. And you need to look at your data from different perspectives. Inspecting your data will help you to build up your intuition and prompt you to start asking questions about the data that you have. Multiple perspectives on your data will challenge you to think about the data differently, helping you to ask more and better questions.

The first and best place to start is to calculate basic summary descriptive statistics on your data. You need to learn the shape, size, type and general layout of the data that you have. Let's look at some ways that you can summarize your data using R. In the next section you will discover 8 quick and simple ways to summarize your dataset.

7.2 Peek At Your Data

The very first thing to do is to just look at some raw data from your dataset. If your dataset is small you might be able to display it all on the screen. Often it is not, so you can take a small sample and review that.

```
# load the package
library(mlbench)
# load the dataset
data(PimaIndiansDiabetes)
# display first 20 rows of data
head(PimaIndiansDiabetes, n=20)
```

Listing 7.1: Display first 20 rows of data.

The `head()` function will display the first 20 rows of data for you to review and think about.

```
   pregnant glucose pressure triceps insulin mass pedigree age diabetes
1         6     148       72      35       0 33.6   0.627  50      pos
2         1      85       66      29       0 26.6   0.351  31      neg
3         8     183       64       0       0 23.3   0.672  32      pos
4         1      89       66      23      94 28.1   0.167  21      neg
5         0     137       40      35     168 43.1   2.288  33      pos
6         5     116       74       0       0 25.6   0.201  30      neg
7         3      78       50      32      88 31.0   0.248  26      pos
8        10     115        0       0       0 35.3   0.134  29      neg
9         2     197       70      45     543 30.5   0.158  53      pos
10        8     125       96       0       0  0.0   0.232  54      pos
11        4     110       92       0       0 37.6   0.191  30      neg
12       10     168       74       0       0 38.0   0.537  34      pos
13       10     139       80       0       0 27.1   1.441  57      neg
14        1     189       60      23     846 30.1   0.398  59      pos
```

```
15       5     166       72       19      175 25.8    0.587  51      pos
16       7     100        0        0        0 30.0    0.484  32      pos
17       0     118       84       47      230 45.8    0.551  31      pos
18       7     107       74        0        0 29.6    0.254  31      pos
19       1     103       30       38       83 43.3    0.183  33      neg
20       1     115       70       30       96 34.6    0.529  32      pos
```

Listing 7.2: Output of first 20 rows.

7.3 Dimensions of Your Data

How much data do you have? You may have a general idea, but it is much better to have a precise figure. If you have a lot of instances, you may need to work with a smaller sample of your data data so that modeling is computationally tractable. If you have a vast number of attributes, you may need to select those that are most relevant. If you have more attributes than instances you may need to select specific modeling techniques.

```
# load the packages
library(mlbench)
# load the dataset
data(PimaIndiansDiabetes)
# display the dimensions of the dataset
dim(PimaIndiansDiabetes)
```

Listing 7.3: Calculate dimensions.

This shows the number of rows and columns in your loaded dataset.

```
[1] 768   9
```

Listing 7.4: Output dimensions.

7.4 Data Types

You need to know the types of the attributes in your data. The types will indicate the types of further analysis, types of visualization and even the types of machine learning algorithms that you can use. Additionally, perhaps some attributes were loaded as one type (e.g. integer) and could in fact be represented as another type (e.g. a categorical factor). Inspecting the types helps expose these issues and spark ideas early.

```
# load package
library(mlbench)
# load dataset
data(BostonHousing)
# list types for each attribute
sapply(BostonHousing, class)
```

Listing 7.5: Calculate data types.

This lists the data type of each attribute in your dataset.

```
     crim       zn     indus      chas       nox       rm       age       dis       rad       tax
        ptratio        b
"numeric" "numeric" "numeric" "factor" "numeric" "numeric" "numeric" "numeric" "numeric"
    "numeric" "numeric" "numeric"
    lstat     medv
"numeric" "numeric"
```

Listing 7.6: Output data types.

7.5 Class Distribution

In a classification problem you must know the proportion of instances that belong to each class label. This is important because it may highlight an imbalance in the data, that if severe may need to be addressed with rebalancing techniques. In the case of a multiclass classification problem it may expose a class with a small number of instances that may be candidates for removing from the dataset.

```
# load the packages
library(mlbench)
# load the dataset
data(PimaIndiansDiabetes)
# distribution of class variable
y <- PimaIndiansDiabetes$diabetes
cbind(freq=table(y), percentage=prop.table(table(y))*100)
```

Listing 7.7: Calculate class breakdown.

This recipe creates a useful table showing the number of instances that belong to each class as well as the percentage that this represents from the entire dataset.

```
    freq percentage
neg  500   65.10417
pos  268   34.89583
```

Listing 7.8: Output class breakdown.

7.6 Data Summary

There is a most valuable function called `summary()` that summarizes each attribute in your dataset in turn. The function creates a table for each attribute and lists a breakdown of values. Factors are described as counts next to each class label. Numerical attributes are described using the properties:

- Min
- 25th percentile
- Median
- Mean

- 75th percentile
- Max

The breakdown also includes an indication of the number of missing values for an attribute (marked `N/A`).

```
# load the iris dataset
data(iris)
# summarize the dataset
summary(iris)
```

Listing 7.9: Calculate the summary of the attributes.

You can see that this recipe produces a lot of information for you to review. Take your time and work through each attribute in turn.

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

Listing 7.10: Output of the summary of attributes.

7.7 Standard Deviations

One thing missing from the `summary()` function above are the standard deviations. The standard deviation along with the mean are useful to know if the data has a Gaussian (or nearly Gaussian) distribution. For example, it can be useful for a quick and dirty outlier removal tool, where any values that are more than three times the standard deviation from the mean are outside of 99.7% of the data.

```
# load the packages
library(mlbench)
# load the dataset
data(PimaIndiansDiabetes)
# calculate standard deviation for all attributes
sapply(PimaIndiansDiabetes[,1:8], sd)
```

Listing 7.11: Calculate standard deviations.

This calculates the standard deviation for each numeric attribute in the dataset.

```
   pregnant    glucose   pressure    triceps    insulin       mass   pedigree        age
  3.3695781 31.9726182 19.3558072 15.9522176 115.2440024 7.8841603 0.3313286 11.7602315
```

Listing 7.12: Output of standard deviations

7.8 Skewness

If a distribution looks nearly-Gaussian but is pushed far left or right it is useful to know the skew. Getting a feeling for the skew is much easier with plots of the data, such as a histogram or density plot. It is harder to tell from looking at means, standard deviations and quartiles. Nevertheless, calculating the skew up-front gives you a reference that you can use later if you decide to correct the skew for an attribute.

```
# load packages
library(mlbench)
library(e1071)
# load the dataset
data(PimaIndiansDiabetes)
# calculate skewness for each variable
skew <- apply(PimaIndiansDiabetes[,1:8], 2, skewness)
# display skewness, larger/smaller deviations from 0 show more skew
print(skew)
```

Listing 7.13: Calculate skewness.

The further the distribution of the skew value from zero, the larger the skew to the left (negative skew value) or right (positive skew value).

```
  pregnant   glucose  pressure   triceps   insulin      mass  pedigree       age
 0.8981549 0.1730754 -1.8364126 0.1089456 2.2633826 -0.4273073 1.9124179 1.1251880
```

Listing 7.14: Output of skewness.

7.9 Correlations

It is important to observe and think about how attributes relate to each other. For numeric attributes, an excellent way to think about attribute-to-attribute interactions is to calculate correlations for each pair of attributes.

```
# load the packages
library(mlbench)
# load the dataset
data(PimaIndiansDiabetes)
# calculate a correlation matrix for numeric variables
correlations <- cor(PimaIndiansDiabetes[,1:8])
# display the correlation matrix
print(correlations)
```

Listing 7.15: Calculate correlations.

This creates a symmetrical table of all pairs of attribute correlations for numerical data. Deviations from zero show more positive or negative correlation. Values above approximately 0.75 or below -0.75 are perhaps more interesting as they show a high correlation or high negative correlation. Values of 1 and -1 show full positive or negative correlation.

```
             pregnant   glucose  pressure    triceps    insulin      mass    pedigree       age
pregnant 1.00000000 0.12945867 0.14128198 -0.08167177 -0.07353461 0.01768309 -0.03352267
    0.54434123
```

```
glucose  0.12945867 1.00000000 0.15258959 0.05732789 0.33135711 0.22107107 0.13733730
    0.26351432
pressure 0.14128198 0.15258959 1.00000000 0.20737054 0.08893338 0.28180529 0.04126495
    0.23952795
triceps -0.08167177 0.05732789 0.20737054 1.00000000 0.43678257 0.39257320 0.18392757
    -0.11397026
insulin -0.07353461 0.33135711 0.08893338 0.43678257 1.00000000 0.19785906 0.18507093
    -0.04216295
mass     0.01768309 0.22107107 0.28180529 0.39257320 0.19785906 1.00000000 0.14064695
    0.03624187
pedigree -0.03352267 0.13733730 0.04126495 0.18392757 0.18507093 0.14064695 1.00000000
    0.03356131
age      0.54434123 0.26351432 0.23952795 -0.11397026 -0.04216295 0.03624187 0.03356131
    1.00000000
```

Listing 7.16: Output of correlations.

7.10 Tips To Remember

This section gives you some tips to remember when reviewing your data using summary statistics.

Review the numbers : Generating the summary statistics is not enough. Take a moment to pause, read and really think about the numbers you are seeing.

Ask why : Review your numbers and ask a lot of questions. How and why are you seeing specific numbers. Think about how the numbers relate to the problem domain in general and specific entities that observations relate to.

Write down ideas : Write down your observations and ideas. Keep a small text file or note pad and jot down all of the ideas for how variables may relate to each other, for what numbers mean, and ideas for techniques to try later. The things you write down now while the data is fresh will be very valuable later when you are trying to think up new things to try.

7.11 Summary

In this lesson you discovered the importance of describing your dataset before you start work on your machine learning project. You discovered 8 different ways to summarize your dataset using R:

- Peek At Your Data.
- Dimensions of Your Data.
- Data Types.
- Class Distribution.
- Data Summary.

- Standard Deviations.
- Skewness.
- Correlations.

You do not need to be good at statistics. The statistics used in this lesson are very simple, but you may have forgotten some of the basics. You can quickly browse through Wikipedia for topics like Mean, Standard Deviation and Quartiles to refresh your knowledge.

7.11.1 Next Step

This lesson was all about techniques that you can use to understand your dataset analytically. Up next you will discover techniques that you can use to better understand your data quantitatively using data visualization.

Chapter 8

Understand Your Data Using Data Visualization

You must understand your data to get the best results from machine learning algorithms. Data visualization is perhaps the fastest and most useful way to summarize and learn more about your data. In this lesson you will discover exactly how you can use data visualization to better understand your data for machine learning using R. After completing this lesson, you will know:

1. How to create plots in order to understand each attribute standalone.
2. How to create plots in order to understand the relationships between attributes.

Let's get started.

8.1 Understand Your Data To Get The Best Results

A better understanding of your data will yield better results from machine learning algorithms. You will be able to clean, transform and best present the data that you have. The better that the data exposes the structure of the problem to the machine learning algorithms, the more accurate your models will be. Additionally, a deeper understanding of your data may even suggest specific machine learning algorithms to try on your data.

8.1.1 Visualize Your Data For Faster Understanding

The fastest way to improve your understanding of your dataset is to visualize it. Visualization means creating charts and plots from the raw data. Plots of the distribution or spread of attributes can help you spot outliers, strange or invalid data and give you an idea of possible data transformations you could apply.

Plots of the relationships between attributes can give you an idea of attributes that might be redundant, resampling methods that may be needed and ultimately how difficult a prediction problem might be. In the next section you will discover how you can quickly visualize your data in R. This lesson is divided into three parts:

- **Visualization Packages**: A quick note about your options when it comes to R packages for visualization.

- **Univariate Visualization**: Plots you can use to understand each attribute standalone.
- **Multivariate Visualization**: Plots that can help you to better understand the interactions between attributes.

8.2 Visualization Packages

There are many ways to visualize data in R, but a few packages have surfaced as perhaps being the most generally useful.

- **graphics package**: Excellent for fast and basic plots of data.
- **lattice package**: More pretty plots and more often useful in practice.
- **ggplot2 package**: Beautiful plots that you want to generate when you need to present results.

I recommend that you stick with simple plots from the `graphics` package for quick and dirty visualization, and use wrappers around `lattice` (via the `caret` package) for more useful multivariate plots. I think `ggplot2` plots are excellent and look lovely, but overkill for quick and dirty data visualization.

8.3 Univariate Visualization

Univariate plots are plots of individual attributes without interactions. The goal is to learn something about the distribution, central tendency and spread of each attribute.

8.3.1 Histograms

Histograms provide a bar chart of a numeric attribute split into bins with the height showing the number of instances that fall into each bin. They are useful to get an indication of the distribution of an attribute.

```
# load the data
data(iris)
# create histograms for each attribute
par(mfrow=c(1,4))
for(i in 1:4) {
  hist(iris[,i], main=names(iris)[i])
}
```

Listing 8.1: Calculate histograms.

You can see that most of the attributes show a Gaussian or multi-modal Gaussian distribution. You can see the measurements of very small flowers in the Petal width and length column.

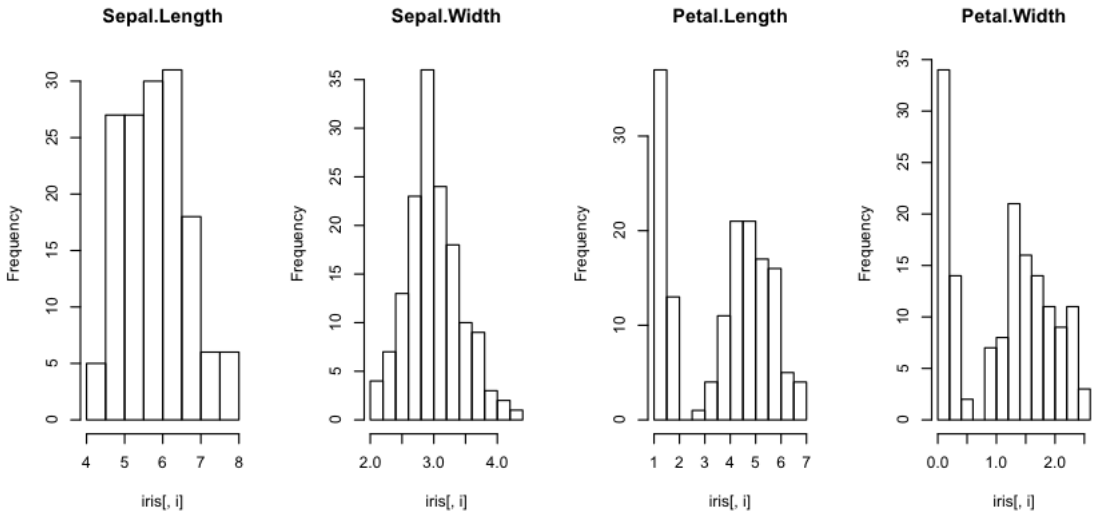

Figure 8.1: Histogram Plot in R

8.3.2 Density Plots

We can smooth out the histograms to lines using a density plot. These are useful for a more abstract depiction of the distribution of each variable.

```
# load packages
library(lattice)
# load dataset
data(iris)
# create a layout of simpler density plots by attribute
par(mfrow=c(1,4))
for(i in 1:4) {
  plot(density(iris[,i]), main=names(iris)[i])
}
```

Listing 8.2: Calculate density plots.

Using the same dataset from the previous example with histograms, we can see the double Gaussian distribution with petal measurements. We can also see a possible exponential (Lapacian-like) distribution for the Sepal width.

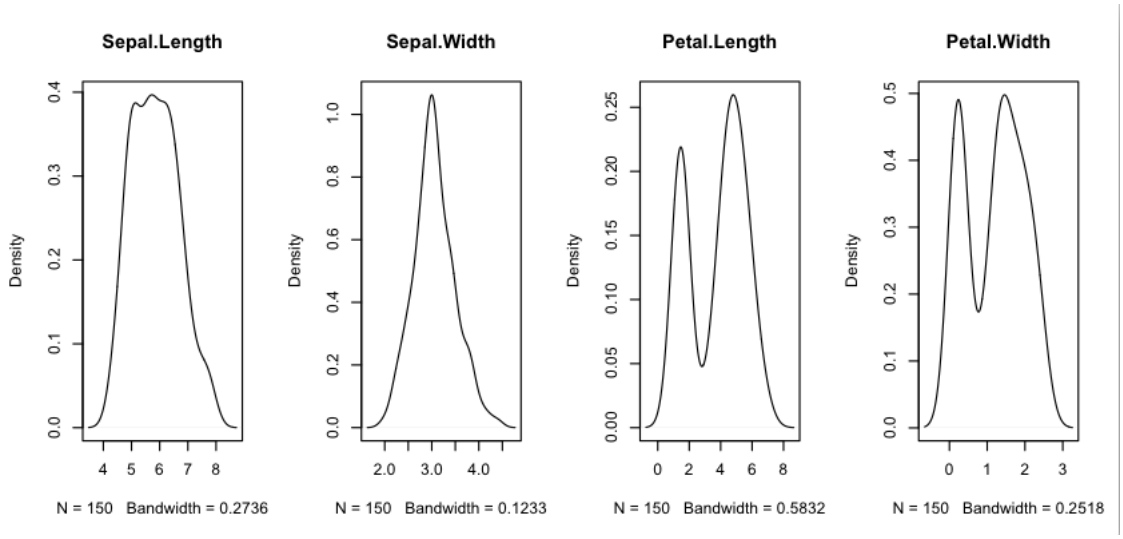

Figure 8.2: Density Plots in R

8.3.3 Box And Whisker Plots

We can look at the distribution of the data a different way using box and whisker plots. The box captures the middle 50% of the data, the line shows the median and the whiskers of the plots show the reasonable extent of data. Any dots outside the whiskers are good candidates for outliers.

```
# load dataset
data(iris)
# Create separate boxplots for each attribute
par(mfrow=c(1,4))
for(i in 1:4) {
  boxplot(iris[,i], main=names(iris)[i])
}
```

Listing 8.3: Calculate box and whisker plots.

We can see that the data all has a similar range (and the same units of centimeters). We can also see that Sepal width may have a few outlier values for this data sample.

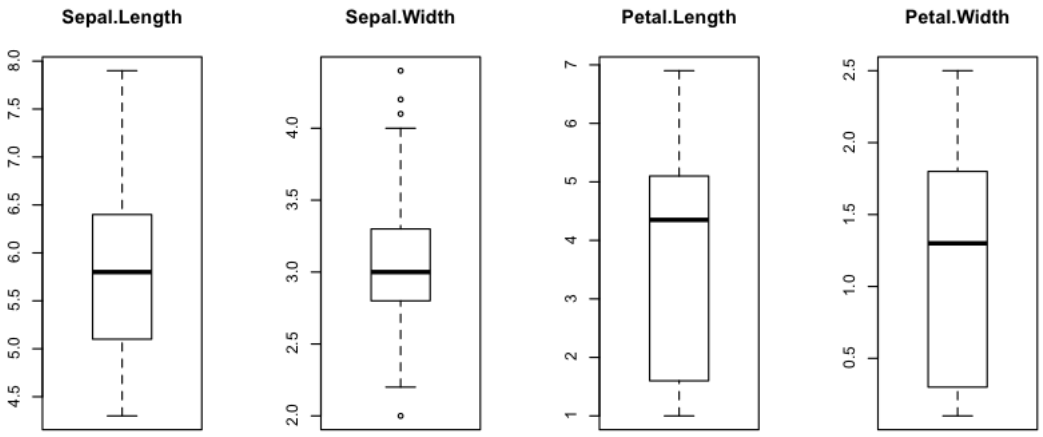

Figure 8.3: Box and Whisker Plots in R

8.3.4 Bar Plots

In datasets that have categorical rather than numeric attributes, we can create bar plots that give an idea of the proportion of instances that belong to each category.

```
# load the package
library(mlbench)
# load the dataset
data(BreastCancer)
# create a bar plot of each categorical attribute
par(mfrow=c(2,4))
for(i in 2:9) {
  counts <- table(BreastCancer[,i])
  name <- names(BreastCancer)[i]
  barplot(counts, main=name)
}
```

Listing 8.4: Calculate bar plots.

We can see that some plots have a good mixed distribution and others show a few labels with the overwhelming number of instances.

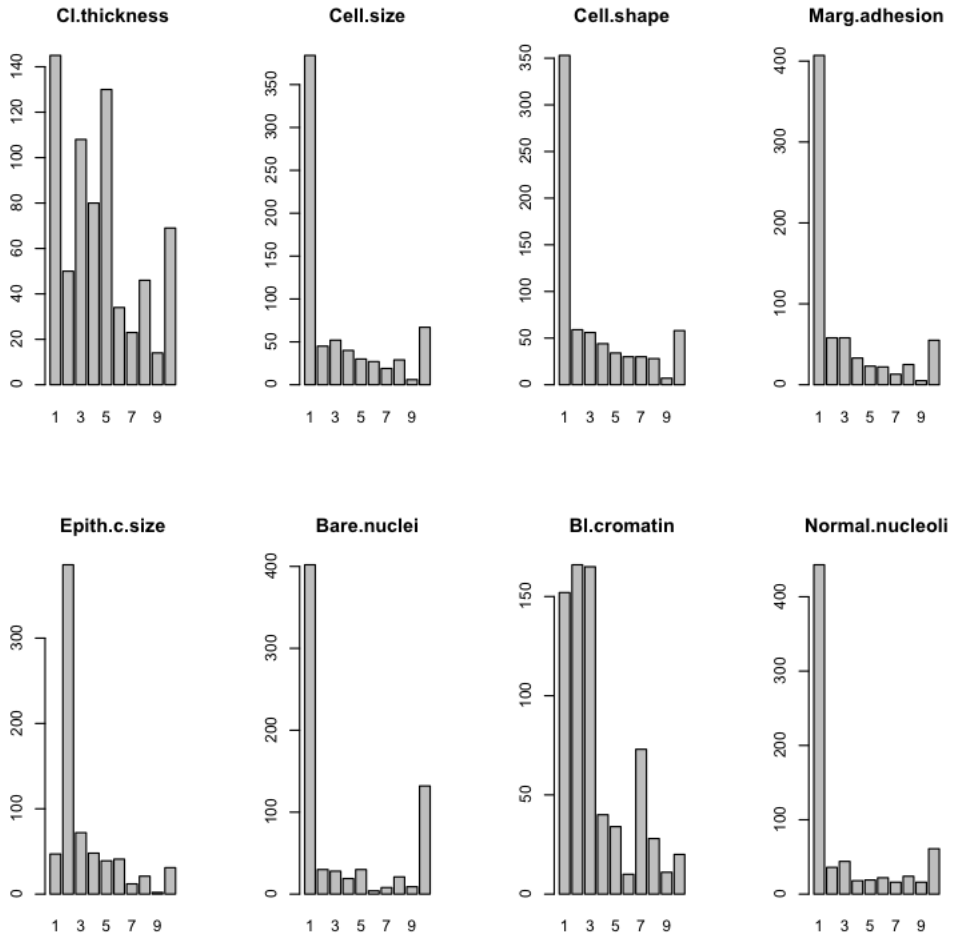

Figure 8.4: Bar Plots in R

8.3.5 Missing Plot

Missing data can have a big impact on modeling. Some techniques ignore missing data, others break. You can use a missing plot to get a quick idea of the amount of missing data in your dataset. The x-axis shows attributes and the y-axis shows instances. Horizontal lines indicate missing data for an instance, vertical blocks represent missing data for an attribute.

You may need to install the `Amelia` package. Refer to Section 4.6 for help installing packages.

```
# load packages
library(Amelia)
library(mlbench)
# load dataset
data(Soybean)
# create a missing map
missmap(Soybean, col=c("black", "grey"), legend=FALSE)
```

Listing 8.5: Calculate missing plot.

We can see that some instances have a lot of missing data across some or most of the attributes.

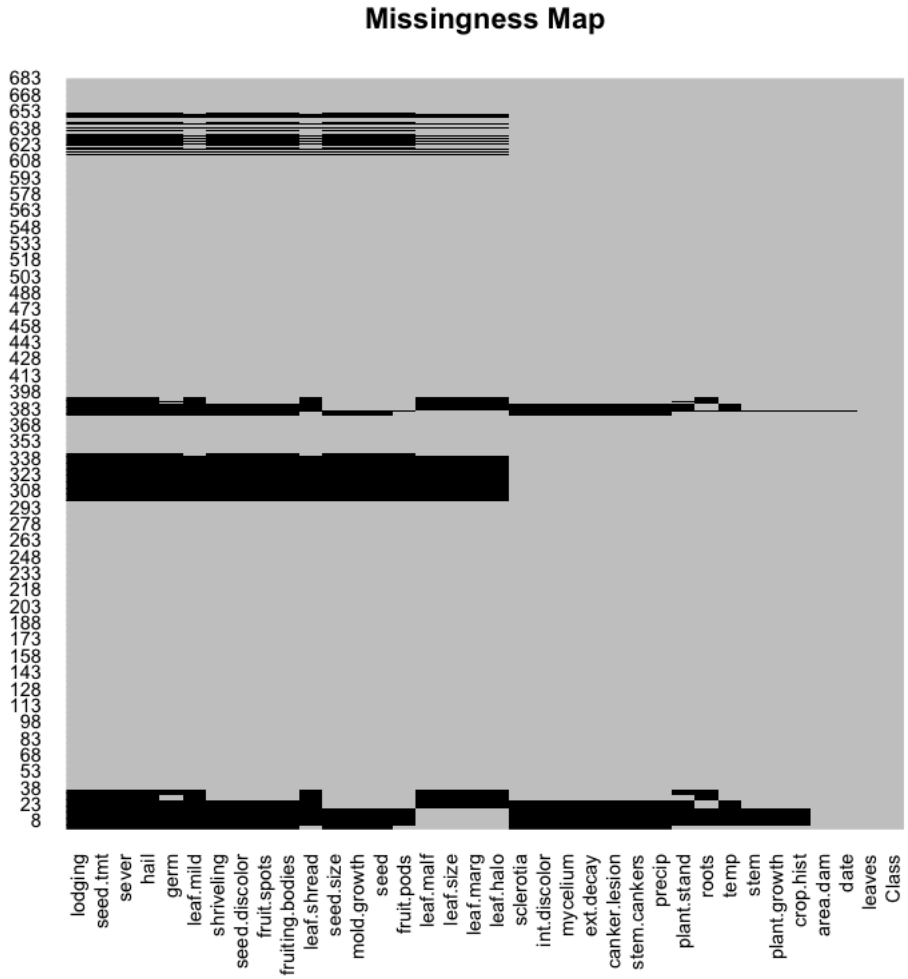

Figure 8.5: Missing Map in R

8.4 Multivariate Visualization

Multivariate plots are plots of the relationship or interactions between attributes. The goal is to learn something about the distribution, central tendency and spread over groups of data, typically pairs of attributes.

8.4.1 Correlation Plot

We can calculate the correlation between each pair of numeric attributes. These pairwise correlations can be plotted in a correlation matrix to given an idea of which attributes change together.

You may need to install the `corrplot` package. Refer to Section 4.6 for help installing packages.

```
# load package
library(corrplot)
# load the data
data(iris)
```

```
# calculate correlations
correlations <- cor(iris[,1:4])
# create correlation plot
corrplot(correlations, method="circle")
```

Listing 8.6: Calculate correlation plot.

A dot-representation was used where blue represents positive correlation and red negative. The larger the dot the larger the correlation. We can see that the matrix is symmetrical and that the diagonal attributes are perfectly positively correlated (because it shows the correlation of each attribute with itself). We can see that some of the attributes are highly correlated.

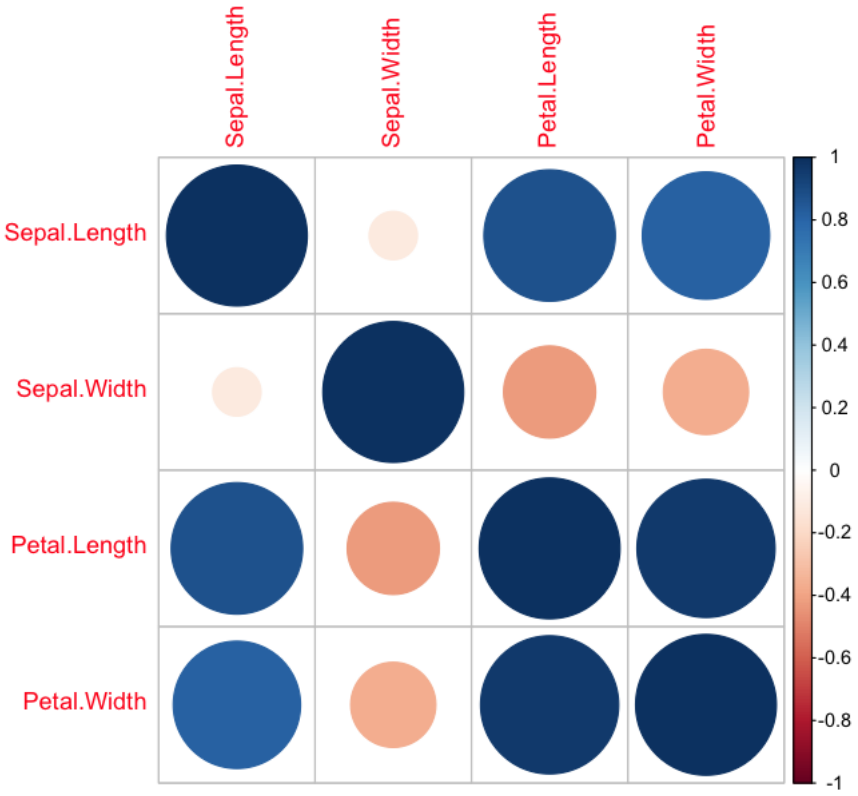

Figure 8.6: Correlation Matrix Plot in R

8.4.2 Scatter Plot Matrix

A scatter plot plots two variables together, one on each of the x- and y-axes with points showing the interaction. The spread of the points indicates the relationship between the attributes. You can create scatter plots for all pairs of attributes in your dataset, called a scatter plot matrix.

```
# load the data
data(iris)
# pairwise scatter plots of all 4 attributes
pairs(iris)
```

Listing 8.7: Calculate a scatter plot matrix.

Note that the matrix is symmetrical, showing the same plots with axes reversed. This aids in looking at your data from multiple perspectives. Note the linear (diagonal line) relationship between petal length and width.

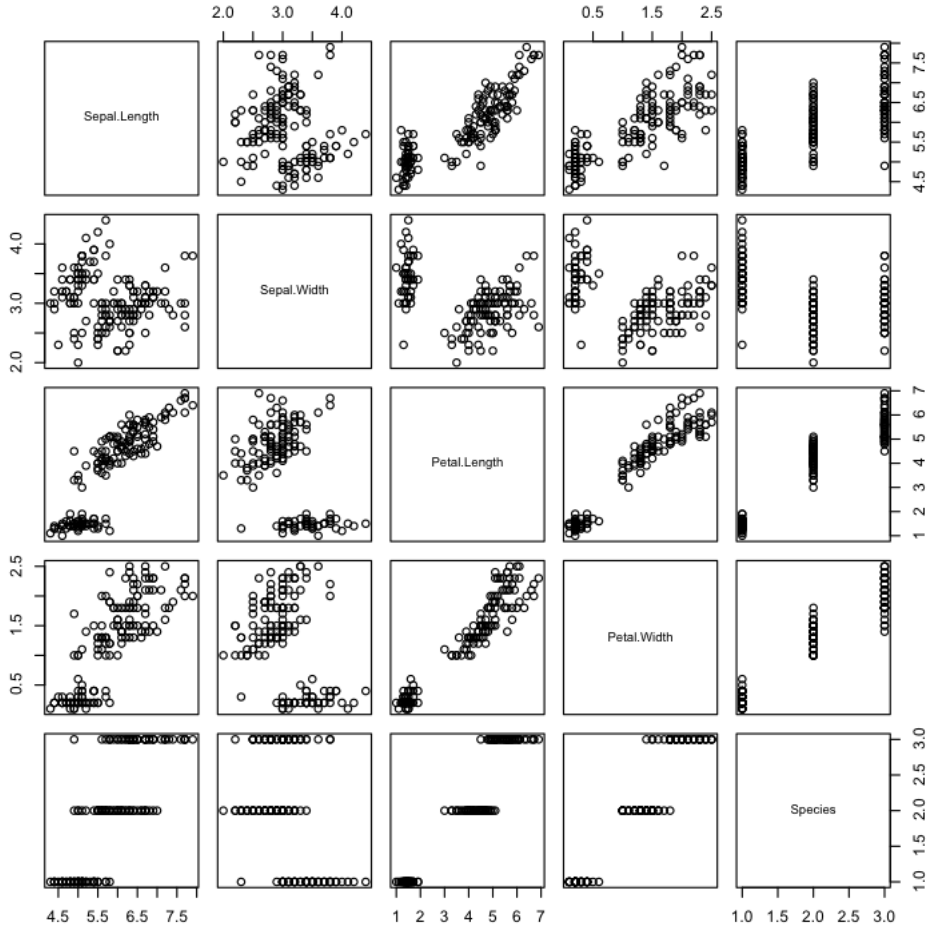

Figure 8.7: Scatter plot Matrix Plot in R

8.4.3 Scatter plot Matrix By Class

The points in a scatter plot matrix can be colored by the class label in classification problems. This can help to spot clear (or unclear) separation of classes and perhaps give an idea of how difficult the problem may be.

```
# load the data
data(iris)
# pairwise scatter plots colored by class
pairs(Species~., data=iris, col=iris$Species)
```

Listing 8.8: Calculate a scatter plot matrix by class.

Note the clear separation of the points by class label on most pairwise plots.

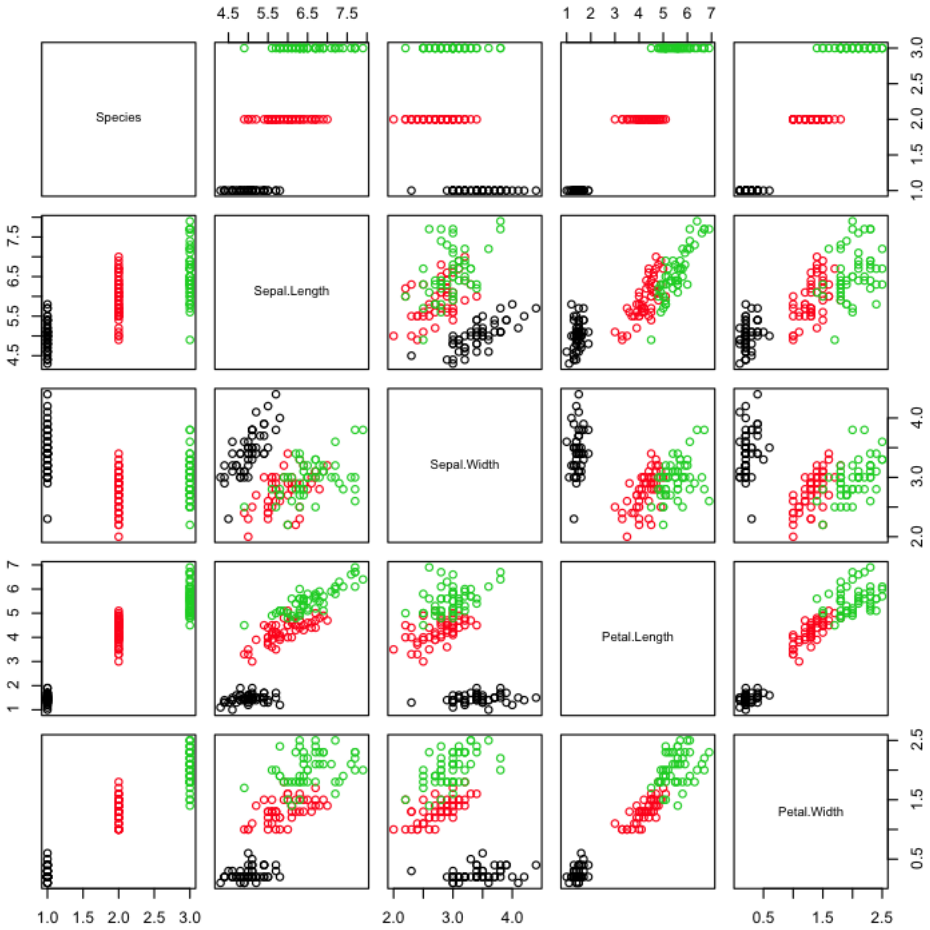

Figure 8.8: Scatter plot Matrix Plot By Class in R

8.4.4 Density Plots By Class

We can review the density distribution of each attribute broken down by class value. Like the scatter plot matrix, the density plot by class can help see the separation of classes. It can also help to understand the overlap in class values for an attribute.

```
# load the package
library(caret)
# load the data
data(iris)
# density plots for each attribute by class value
x <- iris[,1:4]
y <- iris[,5]
scales <- list(x=list(relation="free"), y=list(relation="free"))
featurePlot(x=x, y=y, plot="density", scales=scales)
```

Listing 8.9: Calculate density plots by class.

We can see that some classes do not overlap at all (e.g. Petal Length) where as with other attributes there are hard to tease apart (e.g. Sepal Width).

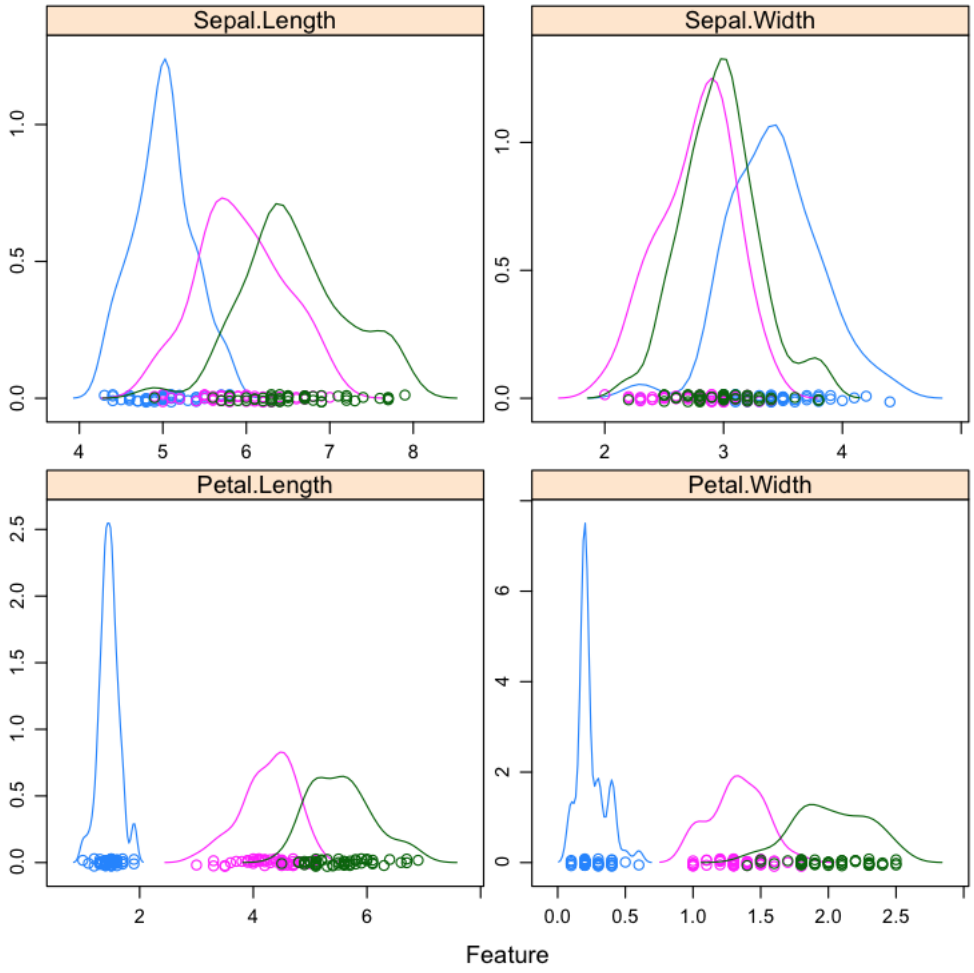

Figure 8.9: Density Plot By Class in R

8.4.5 Box And Whisker Plots By Class

We can also review the boxplot distributions of each attribute by class value. This too can help in understanding how each attribute relates to the class value, but from a different perspective to that of the density plots.

```
# load the package
library(caret)
# load the iris dataset
data(iris)
# box and whisker plots for each attribute by class value
x <- iris[,1:4]
y <- iris[,5]
featurePlot(x=x, y=y, plot="box")
```

Listing 8.10: Calculate box and whisker plots by class.

These plots help to understand the overlap and separation of the attribute-class groups. We can see some good separation of the Setosa class for the Petal Length attribute.

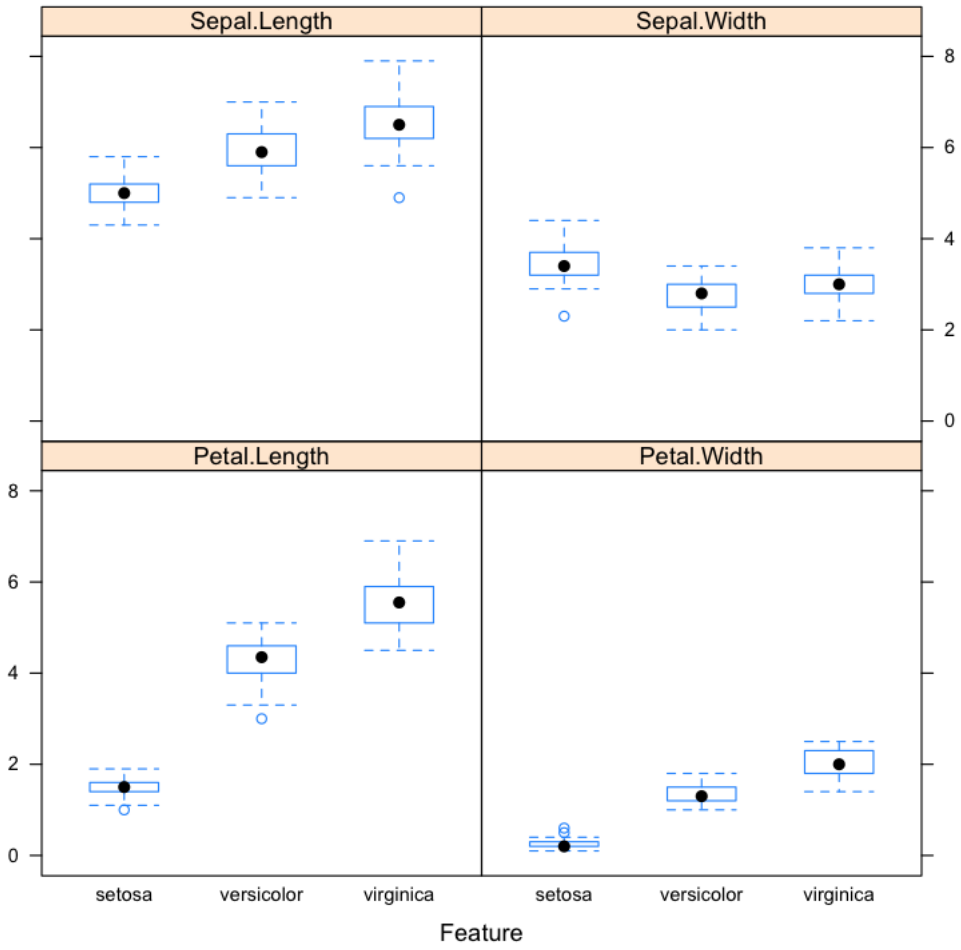

Figure 8.10: Box and Whisker Plots By Class in R

8.5 Tips For Data Visualization

- **Review Plots**. Actually take the time to look at the plots you have generated and think about them. Try to relate what you are seeing to the general problem domain as well as specific records in the data. The goal is to learn something about your data, not to generate a plot.
- **Ugly Plots, Not Pretty**. Your goal is to learn about your data not to create pretty visualizations. Do not worry if the graphs are ugly. You a not going to show them to anyone.
- **Write Down Ideas**. You will get a lot of ideas when you are looking at visualizations of your data. Ideas like data splits to look at, transformations to apply and techniques to test. Write them all down. They will be invaluable later when you are struggling to think of more things to try to get better results.

8.6 Summary

In this lesson you discovered the importance of data visualization in order to better understand your data. You discovered a number of methods that you can use to both visualize and improve your understanding of the attributes in your data using univariate plots and their interactions using multivariate plots.

- **Univariate Plots**: Histograms, Density Plots, Box And Whisker Plots, Bar Plots and Missing Plot
- **Multivariate Plots**: Correlation Plot, Scatter Plot Matrix, Scatter Plot Matrix By Class, Density By Class and Box And Whisker Plots By Class.

8.6.1 Next Step

You have now seen two ways that you can use to learn more about your data: data summarization and data visualization. In the next lesson you will start to use this understanding and pre-process your data in order to best expose the structure of the problem to the learning algorithms.

Chapter 9

Prepare Your Data For Machine Learning With Pre-Processing

Preparing data is required to get the best results from machine learning algorithms. In this lesson you will discover how to transform your data in order to best expose its structure to machine learning algorithms in R using the `caret` package. You will work through 8 popular and powerful data transforms with recipes that you can study or copy-and-paste into your current or next machine learning project. After completing this lesson you will know:

- How to use basic data transforms like scaling, centering, standardization and normalization.
- How to use power transforms like Box-Cox and Yeo-Johnson.
- How to use linear algebra transforms like PCA and ICA.

Let's get started.

9.1 Need For Data Pre-Processing

You want to get the best accuracy from machine learning algorithms on your datasets. Some machine learning algorithms require the data to be in a specific form. Whereas other algorithms can perform better if the data is prepared in a specific way, but not always. Finally, your raw data may not be in the best format to expose the underlying structure and relationships to the predicted variables.

It is important to prepare your data in such a way that it gives various different machine learning algorithms the best chance on your problem. You need to pre-process your raw data as part of your machine learning project.

9.1.1 Data Pre-Processing Methods

It is hard to know which data pre-processing methods to use. You can use rules of thumb such as:

- Instance-based methods are more effective if the input attributes have the same scale.
- Regression methods can work better if the input attributes are standardized.

These are heuristics, not hard and fast laws of machine learning, because sometimes you can get better results if you ignore them. You should try a range of data transforms with a suite of different machine learning algorithms. This will help you discover both good representations for your data and algorithms that are better at exploiting the structure that those representations expose.

It is a good idea to spot-check a number of transforms both in isolation as well as combinations of transforms. In the next section you will discover how you can apply data transforms in order to prepare your data in R using the `caret` package.

9.2 Data Pre-Processing in R

The `caret` package in R provides a number of useful data transforms. These transforms can be used in two ways:

Standalone : Transforms can be modeled from training data and applied to multiple datasets. The model of the transform is prepared using the `preProcess()` function and applied to a dataset using the `predict()` function.

Training : Transforms can be prepared and applied automatically during model evaluation. Transforms applied during training are prepared using the `preProcess()` function and passed to the `train()` function via the `preProcess` argument.

A number of data pre-processing examples are presented in this section. They are presented using the standalone method, but you can just as easily use the prepared pre-processed model during model training. All of the pre-processing examples in this section are for numerical data. Note that the `preProcess()` function will skip over non-numeric data without error.

You can learn more about the data transforms provided by the `caret` package by reading the help for the `preProcess()` function by typing `?preProcess` and by reading the Caret Pre-Processing page[1]. The data transforms presented are more likely to be useful for algorithms such as regression algorithms, instance-based methods (like KNN and LVQ), support vector machines and neural networks. They are less likely to be useful for tree and rule-based methods.

9.2.1 Summary of Transform Methods

Below is a quick summary of all of the transform methods supported in the argument to the `preProcess()` function in caret.

- `BoxCox`: apply a Box-Cox transform, values must be non-zero and positive.
- `YeoJohnson`: apply a Yeo-Johnson transform, like a BoxCox, but values can be negative.
- `expoTrans`: apply a power transform like BoxCox and YeoJohnson.
- `zv`: remove attributes with a zero variance (all the same value).
- `nzv`: remove attributes with a near zero variance (close to the same value).

[1] http://topepo.github.io/caret/preprocess.html

- `center`: subtract mean from values.
- `scale`: divide values by standard deviation.
- `range`: normalize values.
- `pca`: transform data to the principal components.
- `ica`: transform data to the independent components.
- `spatialSign`: project data onto a unit circle.

The rest of this lesson will demonstrate some of the more popular methods.

9.3 Scale Data

The `scale` transform calculates the standard deviation for an attribute and divides each value by that standard deviation. This is a useful operation for scaling data with a Gaussian distribution consistently.

```
# load packages
library(caret)
# load the dataset
data(iris)
# summarize data
summary(iris[,1:4])
# calculate the pre-process parameters from the dataset
preprocessParams <- preProcess(iris[,1:4], method=c("scale"))
# summarize transform parameters
print(preprocessParams)
# transform the dataset using the parameters
transformed <- predict(preprocessParams, iris[,1:4])
# summarize the transformed dataset
summary(transformed)
```

Listing 9.1: Calculate scale data transform.

Running the recipe, you will see:

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

Created from 150 samples and 4 variables

Pre-processing:
  - ignored (0)
  - scaled (4)

  Sepal.Length   Sepal.Width     Petal.Length    Petal.Width
 Min.   :5.193  Min.   : 4.589  Min.   :0.5665  Min.   :0.1312
```

```
 1st Qu.:6.159  1st Qu.: 6.424  1st Qu.:0.9064  1st Qu.:0.3936
 Median :7.004  Median : 6.883  Median :2.4642  Median :1.7055
 Mean   :7.057  Mean   : 7.014  Mean   :2.1288  Mean   :1.5734
 3rd Qu.:7.729  3rd Qu.: 7.571  3rd Qu.:2.8890  3rd Qu.:2.3615
 Max.   :9.540  Max.   :10.095  Max.   :3.9087  Max.   :3.2798
```

Listing 9.2: Output of scale data transform.

9.4 Center Data

The `center` transform calculates the mean for an attribute and subtracts it from each value.

```
# load packages
library(caret)
# load the dataset
data(iris)
# summarize data
summary(iris[,1:4])
# calculate the pre-process parameters from the dataset
preprocessParams <- preProcess(iris[,1:4], method=c("center"))
# summarize transform parameters
print(preprocessParams)
# transform the dataset using the parameters
transformed <- predict(preprocessParams, iris[,1:4])
# summarize the transformed dataset
summary(transformed)
```

Listing 9.3: Calculate center data transform.

Running the recipe, you will see:

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

Created from 150 samples and 4 variables

Pre-processing:
  - centered (4)
  - ignored (0)

 Sepal.Length      Sepal.Width       Petal.Length    Petal.Width
 Min.   :-1.54333  Min.   :-1.05733  Min.   :-2.758  Min.   :-1.0993
 1st Qu.:-0.74333 1st Qu.:-0.25733 1st Qu.:-2.158 1st Qu.:-0.8993
 Median :-0.04333 Median :-0.05733 Median : 0.592  Median : 0.1007
 Mean   : 0.00000  Mean   : 0.00000  Mean   : 0.000  Mean   : 0.0000
 3rd Qu.: 0.55667 3rd Qu.: 0.24267 3rd Qu.: 1.342 3rd Qu.: 0.6007
 Max.   : 2.05667  Max.   : 1.34267  Max.   : 3.142  Max.   : 1.3007
```

Listing 9.4: Output center data transform.

9.5 Standardize Data

Combining the `scale` and `center` transforms will standardize your data. Attributes will have a mean value of 0 and a standard deviation of 1.

```
# load packages
library(caret)
# load the dataset
data(iris)
# summarize data
summary(iris[,1:4])
# calculate the pre-process parameters from the dataset
preprocessParams <- preProcess(iris[,1:4], method=c("center", "scale"))
# summarize transform parameters
print(preprocessParams)
# transform the dataset using the parameters
transformed <- predict(preprocessParams, iris[,1:4])
# summarize the transformed dataset
summary(transformed)
```

Listing 9.5: Calculate standardize data transform.

Notice how we can list multiple methods in a list when specifying the `preProcess` argument to the `train()` function. Running the recipe, you will see:

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

Created from 150 samples and 4 variables

Pre-processing:
  - centered (4)
  - ignored (0)
  - scaled (4)

 Sepal.Length      Sepal.Width      Petal.Length     Petal.Width
 Min.   :-1.86378  Min.   :-2.4258  Min.   :-1.5623  Min.   :-1.4422
 1st Qu.:-0.89767 1st Qu.:-0.5904 1st Qu.:-1.2225 1st Qu.:-1.1799
 Median :-0.05233 Median :-0.1315 Median : 0.3354 Median : 0.1321
 Mean   : 0.00000  Mean   : 0.0000  Mean   : 0.0000  Mean   : 0.0000
 3rd Qu.: 0.67225 3rd Qu.: 0.5567 3rd Qu.: 0.7602 3rd Qu.: 0.7880
 Max.   : 2.48370  Max.   : 3.0805  Max.   : 1.7799  Max.   : 1.7064
```

Listing 9.6: Output standardize data transform.

9.6 Normalize Data

Data values can be scaled into the range of `[0, 1]` which is called normalization.

```
# load packages
```

```
library(caret)
# load the dataset
data(iris)
# summarize data
summary(iris[,1:4])
# calculate the pre-process parameters from the dataset
preprocessParams <- preProcess(iris[,1:4], method=c("range"))
# summarize transform parameters
print(preprocessParams)
# transform the dataset using the parameters
transformed <- predict(preprocessParams, iris[,1:4])
# summarize the transformed dataset
summary(transformed)
```

Listing 9.7: Calculate normalize data transform.

Running the recipe, you will see:

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

Created from 150 samples and 4 variables

Pre-processing:
  - ignored (0)
  - re-scaling to [0, 1] (4)


  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :0.0000  Min.   :0.0000  Min.   :0.0000  Min.   :0.00000
 1st Qu.:0.2222  1st Qu.:0.3333  1st Qu.:0.1017  1st Qu.:0.08333
 Median :0.4167  Median :0.4167  Median :0.5678  Median :0.50000
 Mean   :0.4287  Mean   :0.4406  Mean   :0.4675  Mean   :0.45806
 3rd Qu.:0.5833  3rd Qu.:0.5417  3rd Qu.:0.6949  3rd Qu.:0.70833
 Max.   :1.0000  Max.   :1.0000  Max.   :1.0000  Max.   :1.00000
```

Listing 9.8: Output normalize data transform.

9.7 Box-Cox Transform

When an attribute has a Gaussian-like distribution but is shifted, this is called a skew. The distribution of an attribute can be shifted to reduce the skew and make it more Gaussian. The `BoxCox` transform can perform this operation (assumes all values are positive).

```
# load packages
library(mlbench)
library(caret)
# load the dataset
data(PimaIndiansDiabetes)
# summarize pedigree and age
```

```
summary(PimaIndiansDiabetes[,7:8])
# calculate the pre-process parameters from the dataset
preprocessParams <- preProcess(PimaIndiansDiabetes[,7:8], method=c("BoxCox"))
# summarize transform parameters
print(preprocessParams)
# transform the dataset using the parameters
transformed <- predict(preprocessParams, PimaIndiansDiabetes[,7:8])
# summarize the transformed dataset (note pedigree and age)
summary(transformed)
```

Listing 9.9: Calculate Box-Cox data transform.

Notice, we applied the transform to only two attributes that appear to have a skew. Running the recipe, you will see:

```
   pedigree         age
 Min.   :0.0780  Min.   :21.00
 1st Qu.:0.2437  1st Qu.:24.00
 Median :0.3725  Median :29.00
 Mean   :0.4719  Mean   :33.24
 3rd Qu.:0.6262  3rd Qu.:41.00
 Max.   :2.4200  Max.   :81.00

Created from 768 samples and 2 variables

Pre-processing:
  - Box-Cox transformation (2)
  - ignored (0)

Lambda estimates for Box-Cox transformation:
-0.1, -1.1

   pedigree          age
 Min.   :-2.5510  Min.   :0.8772
 1st Qu.:-1.4116  1st Qu.:0.8815
 Median :-0.9875  Median :0.8867
 Mean   :-0.9599  Mean   :0.8874
 3rd Qu.:-0.4680  3rd Qu.:0.8938
 Max.   : 0.8838  Max.   :0.9019
```

Listing 9.10: Output Box-Cox data transform.

9.8 Yeo-Johnson Transform

The `YeoJohnson` transform another power-transform like Box-Cox, but it supports raw values that are equal to zero and negative.

```
# load packages
library(mlbench)
library(caret)
# load the dataset
data(PimaIndiansDiabetes)
# summarize pedigree and age
summary(PimaIndiansDiabetes[,7:8])
# calculate the pre-process parameters from the dataset
```

```
preprocessParams <- preProcess(PimaIndiansDiabetes[,7:8], method=c("YeoJohnson"))
# summarize transform parameters
print(preprocessParams)
# transform the dataset using the parameters
transformed <- predict(preprocessParams, PimaIndiansDiabetes[,7:8])
# summarize the transformed dataset (note pedigree and age)
summary(transformed)
```

Listing 9.11: Calculate Yeo-Johnson data transform.

Running the recipe, you will see:

```
   pedigree         age
 Min.   :0.0780  Min.   :21.00
 1st Qu.:0.2437  1st Qu.:24.00
 Median :0.3725  Median :29.00
 Mean   :0.4719  Mean   :33.24
 3rd Qu.:0.6262  3rd Qu.:41.00
 Max.   :2.4200  Max.   :81.00

Created from 768 samples and 2 variables

Pre-processing:
  - ignored (0)
  - Yeo-Johnson transformation (2)

Lambda estimates for Yeo-Johnson transformation:
-2.25, -1.15

   pedigree         age
 Min.   :0.0691  Min.   :0.8450
 1st Qu.:0.1724  1st Qu.:0.8484
 Median :0.2265  Median :0.8524
 Mean   :0.2317  Mean   :0.8530
 3rd Qu.:0.2956  3rd Qu.:0.8580
 Max.   :0.4164  Max.   :0.8644
```

Listing 9.12: Output Yeo-Johnson data transform.

9.9 Principal Component Analysis Transform

The PCA transforms the data to return only the principal components, a technique from multivariate statistics and linear algebra. The transform keeps those components above the variance threshold (default=0.95) or the number of components can be specified (`pcaComp`). The result is attributes that are uncorrelated, useful for algorithms like linear and generalized linear regression.

```
# load the packages
library(mlbench)
# load the dataset
data(iris)
# summarize dataset
summary(iris)
# calculate the pre-process parameters from the dataset
```

```
preprocessParams <- preProcess(iris, method=c("center", "scale", "pca"))
# summarize transform parameters
print(preprocessParams)
# transform the dataset using the parameters
transformed <- predict(preprocessParams, iris)
# summarize the transformed dataset
summary(transformed)
```

Listing 9.13: Calculate PCA data transform.

Notice that when we run the recipe that only two principal components are selected.

```
  Sepal.Length   Sepal.Width    Petal.Length   Petal.Width          Species
 Min.   :4.300  Min.   :2.000  Min.   :1.000  Min.   :0.100  setosa   :50
 1st Qu.:5.100  1st Qu.:2.800  1st Qu.:1.600  1st Qu.:0.300  versicolor:50
 Median :5.800  Median :3.000  Median :4.350  Median :1.300  virginica :50
 Mean   :5.843  Mean   :3.057  Mean   :3.758  Mean   :1.199
 3rd Qu.:6.400  3rd Qu.:3.300  3rd Qu.:5.100  3rd Qu.:1.800
 Max.   :7.900  Max.   :4.400  Max.   :6.900  Max.   :2.500
Created from 150 samples and 5 variables

Pre-processing:
  - centered (4)
  - ignored (1)
  - principal component signal extraction (4)
  - scaled (4)

PCA needed 2 components to capture 95 percent of the variance

      Species       PC1               PC2
 setosa   :50   Min.   :-2.7651   Min.   :-2.67732
 versicolor:50  1st Qu.:-2.0957  1st Qu.:-0.59205
 virginica :50  Median : 0.4169  Median :-0.01744
                Mean   : 0.0000  Mean   : 0.00000
                3rd Qu.: 1.3385  3rd Qu.: 0.59649
                Max.   : 3.2996  Max.   : 2.64521
```

Listing 9.14: Output PCA data transform.

9.10 Independent Component Analysis Transform

Transform the data to the independent components. Unlike PCA, ICA retains those components that are independent. You must specify the number of desired independent components with the `n.comp` argument. This transform may be useful for algorithms such as Naive Bayes.

```
# load packages
library(mlbench)
library(caret)
# load the dataset
data(PimaIndiansDiabetes)
# summarize dataset
summary(PimaIndiansDiabetes[,1:8])
# calculate the pre-process parameters from the dataset
preprocessParams <- preProcess(PimaIndiansDiabetes[,1:8], method=c("center", "scale",
    "ica"), n.comp=5)
```

```
# summarize transform parameters
print(preprocessParams)
# transform the dataset using the parameters
transformed <- predict(preprocessParams, PimaIndiansDiabetes[,1:8])
# summarize the transformed dataset
summary(transformed)
```

Listing 9.15: Calculate ICA data transform.

Running the recipe, you will see:

```
    pregnant         glucose         pressure         triceps         insulin           mass
        pedigree
 Min.   : 0.000   Min.   :  0.0   Min.   :  0.00   Min.   : 0.00   Min.   :  0.0   Min.   : 0.00
    Min.   :0.0780
 1st Qu.: 1.000   1st Qu.: 99.0   1st Qu.: 62.00   1st Qu.: 0.00   1st Qu.:  0.0   1st Qu.:27.30
    1st Qu.:0.2437
 Median : 3.000   Median :117.0   Median : 72.00   Median :23.00   Median : 30.5   Median :32.00
    Median :0.3725
 Mean   : 3.845   Mean   :120.9   Mean   : 69.11   Mean   :20.54   Mean   : 79.8   Mean   :31.99
    Mean   :0.4719
 3rd Qu.: 6.000   3rd Qu.:140.2   3rd Qu.: 80.00   3rd Qu.:32.00   3rd Qu.:127.2   3rd Qu.:36.60
    3rd Qu.:0.6262
 Max.   :17.000   Max.   :199.0   Max.   :122.00   Max.   :99.00   Max.   :846.0   Max.   :67.10
    Max.   :2.4200
     age
 Min.   :21.00
 1st Qu.:24.00
 Median :29.00
 Mean   :33.24
 3rd Qu.:41.00
 Max.   :81.00

Created from 768 samples and 8 variables

Pre-processing:
  - centered (8)
  - independent component signal extraction (8)
  - ignored (0)
  - scaled (8)

ICA used 5 components

      ICA1               ICA2               ICA3               ICA4               ICA5
 Min.   :-5.7213   Min.   :-4.89818   Min.   :-6.0289   Min.   :-2.573436   Min.   :-1.8815
 1st Qu.:-0.4873   1st Qu.:-0.48188   1st Qu.:-0.4693   1st Qu.:-0.640601   1st Qu.:-0.8279
 Median : 0.1813   Median : 0.05071   Median : 0.2987   Median : 0.007582   Median :-0.2416
 Mean   : 0.0000   Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.000000   Mean   : 0.0000
 3rd Qu.: 0.6839   3rd Qu.: 0.56462   3rd Qu.: 0.6941   3rd Qu.: 0.638238   3rd Qu.: 0.7048
 Max.   : 2.1819   Max.   : 4.25611   Max.   : 1.3726   Max.   : 3.761017   Max.   : 2.9622
```

Listing 9.16: Output ICA data transform.

9.11 Tips For Data Transforms

Below are some tips for getting the most out of data transforms.

- **Actually Use Them**. You are a step ahead if you are thinking about and using data transforms to prepare your data. It is an easy step to forget or skip over and often has a huge impact on the accuracy of your final models.
- **Use a Variety**. Try a number of different data transforms on your data with a suite of different machine learning algorithms.
- **Review a Summary**. It is a good idea to summarize your data before and after applying a transform to understand the effect it had. The `summary()` function can be very useful.
- **Visualize Data**. It is also a good idea to visualize the distribution of your data before and after to get a spatial intuition for the effect of the transform.

9.12 Summary

In this lesson you discovered 8 data pre-processing methods that you can use on your data in R via the `caret` package:

- Data scaling
- Data centering
- Data standardization
- Data normalization
- The Box-Cox Transform
- The Yeo-Johnson Transform
- PCA Transform
- ICA Transform

You can practice with the recipes presented in this section or apply them on your current or next machine learning project.

9.12.1 Next Step

You now know how to load data into R, understand it and pre-process your data ready for modeling. In the next lesson you will discover methods that you can use to estimate the accuracy of machine learning algorithms on your data.

Chapter 10

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 lesson you will discover 5 approaches for estimating model performance on unseen data. After completing this lesson, you will know:

1. How to split a dataset into train and test subsets.
2. How to evaluate model accuracy using the bootstrap method.
3. How to evaluate model accuracy using k-fold cross validation with and without repeats.
4. How to evaluate model accuracy using leave one out cross validation.

Let's get started.

10.1 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 require 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 5 different resampling methods that you can use to evaluate the accuracy of your data in R.

10.2 Data Split

Data splitting involves partitioning the data into an explicit training dataset used to prepare the model and an unseen test dataset used to evaluate the model's performance on unseen data. It is useful when you have a very large dataset so that the test dataset can provide a meaningful estimation of performance, or for when you are using slow methods and need a quick approximation of performance.

The example below splits the iris dataset so that 80% is used for training a Naive Bayes model and 20% is used to evaluate the model's performance.

```
# load the packages
library(caret)
library(klaR)
# load the iris dataset
data(iris)
# define an 80%/20% train/test split of the dataset
trainIndex <- createDataPartition(iris$Species, p=0.80, list=FALSE)
dataTrain <- iris[ trainIndex,]
dataTest <- iris[-trainIndex,]
# train a naive Bayes model
fit <- NaiveBayes(Species~., data=dataTrain)
# make predictions
predictions <- predict(fit, dataTest[,1:4])
# summarize results
confusionMatrix(predictions$class, dataTest$Species)
```

Listing 10.1: Calcualte data split.

Running this example, you will see an estimation of model accuracy on the test subset of the data:

```
Confusion Matrix and Statistics

            Reference
Prediction  setosa versicolor virginica
  setosa        10         0         0
  versicolor     0        10         2
  virginica      0         0         8

Overall Statistics

               Accuracy : 0.9333
                 95% CI : (0.7793, 0.9918)
    No Information Rate : 0.3333
    P-Value [Acc > NIR] : 8.747e-12

                  Kappa : 0.9
 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: setosa Class: versicolor Class: virginica
Sensitivity                 1.0000            1.0000           0.8000
Specificity                 1.0000            0.9000           1.0000
Pos Pred Value              1.0000            0.8333           1.0000
Neg Pred Value              1.0000            1.0000           0.9091
```

```
Prevalence             0.3333          0.3333          0.3333
Detection Rate         0.3333          0.3333          0.2667
Detection Prevalence   0.3333          0.4000          0.2667
Balanced Accuracy      1.0000          0.9500          0.9000
```

Listing 10.2: Output of data split.

10.3 Bootstrap

Bootstrap resampling involves taking random samples from the dataset (with re-selection) against which to evaluate the model. In aggregate, the results provide an indication of the variance of the model's performance. Typically, large number of resampling iterations are performed (thousands or tens of thousands). The following example uses a bootstrap with 100 resamples to estimate the accuracy of a Naive Bayes model.

```
# load the package
library(caret)
# load the iris dataset
data(iris)
# define training control
trainControl <- trainControl(method="boot", number=100)
# evalaute the model
fit <- train(Species~., data=iris, trControl=trainControl, method="nb")
# display the results
print(fit)
```

Listing 10.3: Estimate accuracy using Bootstrap.

Running this example, you will see the estimated accuracy of the Naive Bayes model with two different values for the `usekernel` model parameter.

```
Naive Bayes

150 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

No pre-processing
Resampling: Bootstrapped (100 reps)
Summary of sample sizes: 150, 150, 150, 150, 150, 150, ...
Resampling results across tuning parameters:

  usekernel Accuracy  Kappa     Accuracy SD Kappa SD
  FALSE     0.9536239 0.9298175 0.02567435  0.03882289
   TRUE     0.9572684 0.9353627 0.02484076  0.03743853

Tuning parameter 'fL' was held constant at a value of 0
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were fL = 0 and usekernel = TRUE.
```

Listing 10.4: Output of accuracy using Bootstrap.

10.4 k-fold Cross Validation

The k-fold cross validation method involves splitting the dataset into k-subsets. Each subset is held-out while the model is trained on all other subsets. This process is repeated until accuracy is determined for each instance in the dataset, and an overall accuracy estimate is provided. It is a robust method for estimating accuracy, and the size of k can tune the amount of bias in the estimate, with popular values set to 5 and 10. The following example uses 10-fold cross validation to estimate the accuracy of the Naive Bayes algorithm on the iris dataset.

```
# load the package
library(caret)
# load the iris dataset
data(iris)
# define training control
trainControl <- trainControl(method="cv", number=10)
# evaluate the model
fit <- train(Species~., data=iris, trControl=trainControl, method="nb")
# display the results
print(fit)
```

Listing 10.5: Estimate accuracy using k-fold Cross Validation.

Running this example, you will see the estimated of the accuracy of the model using 10-fold cross validation.

```
Naive Bayes

150 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 135, 135, 135, 135, 135, 135, ...
Resampling results across tuning parameters:

  usekernel Accuracy  Kappa  Accuracy SD Kappa SD
  FALSE     0.9533333 0.93   0.06324555  0.09486833
   TRUE     0.9533333 0.93   0.05488484  0.08232726

Tuning parameter 'fL' was held constant at a value of 0
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were fL = 0 and usekernel = FALSE.
```

Listing 10.6: Output of accuracy using k-fold Cross Validation.

10.5 Repeated k-fold Cross Validation

The process of splitting the data into k-folds can be repeated a number of times, this is called Repeated k-fold Cross Validation. The final model accuracy is taken as the mean from the number of repeats. The following example demonstrates 10-fold cross validation with 3 repeats to estimate the accuracy of the Naive Bayes algorithm on the iris dataset.

```
# load the package
library(caret)
# load the iris dataset
data(iris)
# define training control
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
# evaluate the model
fit <- train(Species~., data=iris, trControl=trainControl, method="nb")
# display the results
print(fit)
```

Listing 10.7: Estimate accuracy using repeated k-fold Cross Validation.

Running this example, you will see:

```
Naive Bayes

150 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 135, 135, 135, 135, 135, 135, ...
Resampling results across tuning parameters:

  usekernel Accuracy  Kappa     Accuracy SD Kappa SD
  FALSE     0.9533333 0.9300000 0.04998084  0.07497126
   TRUE     0.9577778 0.9366667 0.04789303  0.07183954

Tuning parameter 'fL' was held constant at a value of 0
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were fL = 0 and usekernel = TRUE.
```

Listing 10.8: Output of accuracy using repeated k-fold Cross Validation.

10.6 Leave One Out Cross Validation

In Leave One Out Cross Validation (LOOCV), a data instance is left out and a model constructed on all other data instances in the training set. This is repeated for all data instances. The following example demonstrates LOOCV to estimate the accuracy of the Naive Bayes algorithm on the iris dataset.

```
# load the package
library(caret)
# load the iris dataset
data(iris)
# define training control
trainControl <- trainControl(method="LOOCV")
# evaluate the model
fit <- train(Species~., data=iris, trControl=trainControl, method="nb")
# display the results
print(fit)
```

Listing 10.9: Estimate accuracy using LOOCV.

Running this example, you will see:

```
Naive Bayes

150 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

No pre-processing
Resampling: Leave-One-Out Cross-Validation
Summary of sample sizes: 149, 149, 149, 149, 149, 149, ...
Resampling results across tuning parameters:

  usekernel Accuracy  Kappa
  FALSE     0.9533333 0.93
   TRUE     0.9600000 0.94

Tuning parameter 'fL' was held constant at a value of 0
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were fL = 0 and usekernel = TRUE.
```

Listing 10.10: Output accuracy using LOOCV.

10.7 Tips For Evaluating Algorithms

Below are tips to consider when selecting test options to evaluate the accuracy of predictive modeling machine learning algorithms on unseen data.

- Using a data split into a training and test set is a good idea when you have a lot of data and you are confident that your training sample is representative of the larger dataset.
- Using a data split is very efficient and is often used to get a quick estimate of model accuracy.
- Cross validation is a gold standard for evaluating model accuracy, often with k-folds set to 5 or 10 to balance overfitting the training data with a fair accuracy estimate.
- Repeated k-fold cross validation is preferred when you can afford the computational expense and require a less biased estimate.

10.8 Summary

In this lesson you discovered 5 different methods that you can use to estimate the accuracy of your model on unseen data. Those methods were:

- Data Split.
- Bootstrap.

- k-fold Cross Validation.
- Repeated k-fold Cross Validation.
- Leave One Out Cross Validation.

10.8.1 Next Step

In this lesson you discovered how you can estimate the accuracy of models on unseen data. In the next lesson you will look at metrics that you can use to evaluate the performance of models.

Chapter 11

Machine Learning 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:

1. How to use Accuracy and Kappa to evaluate model skill on classification problems.
2. How to use RMSE and R^2 to evaluate model skill on regression problems.
3. How to use Area Under ROC Curve, sensitivity and specificity to evaluate model skill on binary classification problems.
4. How to use Logarithmic Loss to evaluate model skill on classification problems.

Let's get started.

11.1 Model Evaluation Metrics in R

There are many different metrics that you can use to evaluate your machine learning algorithms in R. When you use `caret` to evaluate your models, the default metrics used are accuracy for classification problems and RMSE for regression. But `caret` supports a range of other popular evaluation metrics.

In this lesson you will step through each of the evaluation metrics provided by `caret`. Each example provides a complete case study that you can copy-and-paste into your project and adapt to your problem. Note that this lesson does assume that you already know how to interpret these other metrics.

11.2 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 multiclass classification problem

because it can be less clear exactly how the accuracy breaks down across those classes (e.g. you need to go deeper with a confusion matrix).

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). In the example below the Pima Indians diabetes dataset is used. It has a class break down of 65% to 35% for negative and positive outcomes.

```
# load packages
library(caret)
library(mlbench)
# load the dataset
data(PimaIndiansDiabetes)
# prepare resampling method
trainControl <- trainControl(method="cv", number=5)
set.seed(7)
fit <- train(diabetes~., data=PimaIndiansDiabetes, method="glm", metric="Accuracy",
    trControl=trainControl)
# display results
print(fit)
```

Listing 11.1: Calculate Accuracy and Kappa metrics.

Running this example, we can see tables of Accuracy and Kappa for the evaluated algorithm. This includes the mean values (left) and the standard deviations (marked as SD) for each metric, taken over the population of cross validation folds and trials. You can see that the accuracy of the model is approximately 76% which is 11 percentage points above the baseline accuracy of 65% which is not really that impressive. The Kappa value on the other hand shows approximately 46% which is more interesting.

```
Generalized Linear Model

768 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 614, 614, 615, 615, 614
Resampling results

  Accuracy  Kappa     Accuracy SD Kappa SD
  0.7695442 0.4656824 0.02692468  0.0616666
```

Listing 11.2: Output of Accuracy and Kappa metrics.

11.3 RMSE and R^2

RMSE and R^2 are the default metrics used to evaluate algorithms on regression datasets in `caret`. RMSE or Root Mean Squared Error is the average deviation of the predictions from the observations. It is useful to get a gross idea of how well (or not) an algorithm is doing, in the units of the output variable.

R^2 spoken as R Squared (also called the coefficient of determination) provides a goodness-of-fit measure for the predictions to the observations. This is a value between 0 and 1 for no-fit and perfect fit respectively. In this example the longley economic dataset is used. The output variable for this dataset is a number employed people in the population. It is not clear whether this is an actual count (e.g. in millions) or a percentage.

```
# load packages
library(caret)
# load data
data(longley)
# prepare resampling method
trainControl <- trainControl(method="cv", number=5)
set.seed(7)
fit <- train(Employed~., data=longley, method="lm", metric="RMSE", trControl=trainControl)
# display results
print(fit)
```

Listing 11.3: Calculate RMSE and RSquared metrics.

Running this example, we can see tables of RMSE and R Squared for the evaluated algorithm. Again, you can see the mean and standard deviations of both metrics are provided. You can see that the RMSE was 0.38 in the units of Employed (whatever those units are). Whereas, the R Square value shows a good fit for the data with a value very close to 1 (0.988).

```
Linear Regression

16 samples
 6 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 12, 12, 14, 13, 13
Resampling results

  RMSE      Rsquared  RMSE SD   Rsquared SD
  0.3868618 0.9883114 0.1025042 0.01581824
```

Listing 11.4: Output of RMSE and RSquared metrics.

11.4 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 model's 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. It 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.

```
# load packages
library(caret)
library(mlbench)
# load the dataset
data(PimaIndiansDiabetes)
# prepare resampling method
trainControl <- trainControl(method="cv", number=5, classProbs=TRUE,
    summaryFunction=twoClassSummary)
set.seed(7)
fit <- train(diabetes~., data=PimaIndiansDiabetes, method="glm", metric="ROC",
    trControl=trainControl)
# display results
print(fit)
```

Listing 11.5: Calculate ROC metrics.

Here, you can see the good but not excellent AUC score of 0.833. The first level is taken as the positive class, in this case neg (no onset of diabetes).

```
Generalized Linear Model

768 samples
  8 predictor
  2 classes: 'neg', 'pos'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 614, 614, 615, 615, 614
Resampling results

  ROC       Sens   Spec      ROC SD     Sens SD    Spec SD
  0.8336003 0.882 0.5600978 0.02111279 0.03563706 0.0560184
```

Listing 11.6: Output ROC metrics.

11.5 Logarithmic Loss

Logarithmic Loss (or LogLoss) is used to evaluate binary classification but it is more common for multiclass classification algorithms. Specifically, it evaluates the probabilities estimated by the algorithms. In this case we see LogLoss calculated for the iris flower multiclass classification problem.

```
# load packages
library(caret)
# load the dataset
data(iris)
# prepare resampling method
trainControl <- trainControl(method="cv", number=5, classProbs=TRUE,
    summaryFunction=mnLogLoss)
```

```
set.seed(7)
fit <- train(Species~., data=iris, method="rpart", metric="logLoss", trControl=trainControl)
# display results
print(fit)
```

Listing 11.7: Calculate LogLoss metrics.

Logloss is minimized and we can see the optimal CART (rpart) model had an argument `cp` value of 0 (the first row of results).

```
CART

150 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 120, 120, 120, 120, 120
Resampling results across tuning parameters:

  cp    logLoss   logLoss SD
  0.00  0.4105613 0.6491893
  0.44  0.6840517 0.4963032
  0.50  1.0986123 0.0000000

logLoss was used to select the optimal model using the smallest value.
The final value used for the model was cp = 0.
```

Listing 11.8: Output LogLoss metrics.

11.6 Summary

In this lesson you discovered different metrics that you can use to evaluate the performance of your machine learning algorithms in R using `caret`. Specifically:

- Accuracy and Kappa.
- RMSE and R^2.
- ROC (AUC, Sensitivity and Specificity).
- LogLoss.

11.6.1 Next Step

In the previous lesson you learned how to evaluate the skill of models on unseen data and in this lesson you discovered performance metrics that you can use. In the next lesson we look at the third piece of your test harness which are the algorithms that you can use to actually model your data.

Chapter 12

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 lesson you will discover how to spot-check algorithms on a dataset using R. After completing this lesson, you will know:

1. How to model data with linear and nonlinear machine learning algorithms directly.
2. How to spot-check a suite of linear and nonlinear algorithms using caret.
3. Which linear and nonlinear algorithms to start with on a new project.

Let's get started.

12.1 Best Algorithm For a Problem

You want the most accurate model for your dataset. That is the goal of predictive modeling. No one can tell you what algorithm to use on your dataset to get the best results. If you or anyone knew what algorithm gave the best results for a specific dataset, then you probably would not need to use machine learning in the first place because of your deep knowledge of the problem.

We cannot know beforehand the best data representation or learning algorithm to use. We don't even know the best parameters to use for algorithms that we could try. We need a strategy to find the best algorithm for our dataset.

12.1.1 Use Past Experience To Choose An Algorithm

One way that you could choose an algorithm for a problem is to rely on experience. This could be your experience with working on similar problems in the past. It could also be the collective experience of the field where you refer to papers, books and other resources for similar problems to get an idea of what algorithms have worked well in the past. This is a good start, but this should not be where you stop.

12.1.2 Use Trial And Error To Choose An Algorithm

The most robust way to discover good or even best algorithms for your dataset is by trial and error. Evaluate a diverse set of algorithms on your dataset and see what works and drop what doesn't. I call this process spot-checking algorithms.

Once you have a shortlist of algorithms that you know are good at picking out the structure of your problem, you can focus your efforts on those algorithms. You can improve the results of candidate algorithms by either tuning the algorithm parameters or by combining the predictions of multiple models using ensemble methods. Next, let's take a look at how we can evaluate multiple machine algorithms on a dataset in R.

12.1.3 Which Algorithms To Spot-Check

You can guess at what algorithms might do well on your dataset, and this can be a good starting point. I recommend trying a mixture of algorithms and see what is good at picking out the structure in your data.

- Try a mixture of algorithm representations (e.g. instance-based methods and trees).
- Try a mixture of learning algorithms (e.g. different algorithms for learning the same type of representation).
- Try a mixture of modeling types (e.g. linear and nonlinear functions or parametric and nonparametric).

Let's get specific. In the next section, we will look at algorithms that you can use to spot-check on your next machine learning project in R.

12.2 Algorithms To Spot-Check in R

There are hundreds of machine learning algorithms available in R. I would recommend exploring many of them, especially, if making accurate predictions on your dataset is important and you have the time. Often you don't have the time, so you need to know the few algorithms that you absolutely must test on your problem.

In this lesson you will discover the linear and nonlinear algorithms you should spot-check on your problem in R. This excludes ensemble algorithms such as boosting and bagging, that can come later once you have a baseline. Each algorithm will be presented from two perspectives:

1. The package and function used to train and make predictions for the algorithm.
2. The wrapper in the `caret` package for the algorithm.

You need to know which package and function to use for a given algorithm. This is needed when you are researching the algorithm parameters and how to get the best performance from the algorithm. It is also needed when you have discovered the best algorithm to use and need to prepare a final model.

You need to know how to use each algorithm with the `caret` package, so that you can efficiently evaluate the accuracy of the algorithm on unseen data using the pre-processing, algorithm evaluation and tuning capabilities of `caret`. Two standard datasets first described in Chapter 5 are used to demonstrate algorithms in this lesson:

- Boston Housing dataset for regression (`BostonHousing` from the `mlbench` package).
- Pima Indians Diabetes dataset for classification (`PimaIndiansDiabetes` from the `mlbench` package).

Algorithms are presented in two groups:

- Linear Algorithms that are simpler methods that have a strong bias but are fast to train.
- Nonlinear Algorithms that are more complex methods that have a large variance but are often more accurate.

Each recipe presented in this lesson is complete and will produce a result, so that you can copy and paste it into your current or next machine learning project.

12.3 Linear Algorithms

These are methods that make large assumptions about the form of the function being modeled. As such they have a high bias but are often fast to train. The final models are also often easy (or easier) to interpret, making them desirable as final models. If the results are suitably accurate, you may not need to move onto using nonlinear methods.

12.3.1 Linear Regression

The `lm()` function is in the `stats` package and creates a linear regression model using ordinary least squares.

```
# load the package
library(mlbench)
# load data
data(BostonHousing)
# fit model
fit <- lm(medv~., BostonHousing)
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, BostonHousing)
# summarize accuracy
mse <- mean((BostonHousing$medv - predictions)^2)
print(mse)
```

Listing 12.1: Example of linear regression algorithm.

The `lm` implementation can be used in `caret` as follows:

```
# load packages
library(caret)
library(mlbench)
# load dataset
data(BostonHousing)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
```

```
fit.lm <- train(medv~., data=BostonHousing, method="lm", metric="RMSE",
    preProcess=c("center", "scale"), trControl=trainControl)
# summarize fit
print(fit.lm)
```

Listing 12.2: Example of linear regression algorithm in caret.

12.3.2 Logistic Regression

The `glm()` function is in the `stats` package and creates a generalized linear model for regression or classification. It can be configured to perform a logistic regression suitable for binary classification problems.

```
# load the package
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# fit model
fit <- glm(diabetes~., data=PimaIndiansDiabetes, family=binomial(link='logit'))
# summarize the fit
print(fit)
# make predictions
probabilities <- predict(fit, PimaIndiansDiabetes[,1:8], type='response')
predictions <- ifelse(probabilities > 0.5,'pos','neg')
# summarize accuracy
table(predictions, PimaIndiansDiabetes$diabetes)
```

Listing 12.3: Example of logistic regression algorithm.

The `glm` algorithm can be used in `caret` as follows:

```
# load packages
library(caret)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.glm <- train(diabetes~., data=PimaIndiansDiabetes, method="glm", metric="Accuracy",
    preProcess=c("center", "scale"), trControl=trainControl)
# summarize fit
print(fit.glm)
```

Listing 12.4: Example of logistic regression algorithm in caret.

12.3.3 Linear Discriminant Analysis

The `lda()` function is in the `MASS` package and creates a linear model of a classification problem.

```
# load the packages
library(MASS)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
```

```
# fit model
fit <- lda(diabetes~., data=PimaIndiansDiabetes)
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, PimaIndiansDiabetes[,1:8])$class
# summarize accuracy
table(predictions, PimaIndiansDiabetes$diabetes)
```

Listing 12.5: Example of LDA algorithm.

The `lda` algorithm can be used in `caret` as follows:

```
# load packages
library(caret)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.lda <- train(diabetes~., data=PimaIndiansDiabetes, method="lda", metric="Accuracy",
    preProcess=c("center", "scale"), trControl=trainControl)
# summarize fit
print(fit.lda)
```

Listing 12.6: Example of LDA algorithm in caret.

12.3.4 Regularized Regression

The `glmnet()` function is in the `glmnet` package and can be used for classification or regression.

Classification Example:

```
# load the package
library(glmnet)
library(mlbench)
# load data
data(PimaIndiansDiabetes)
x <- as.matrix(PimaIndiansDiabetes[,1:8])
y <- as.matrix(PimaIndiansDiabetes[,9])
# fit model
fit <- glmnet(x, y, family="binomial", alpha=0.5, lambda=0.001)
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, x, type="class")
# summarize accuracy
table(predictions, PimaIndiansDiabetes$diabetes)
```

Listing 12.7: Example of GLMNET algorithm for classification.

Regression Example:

```
# load the packages
library(glmnet)
library(mlbench)
```

```
# load data
data(BostonHousing)
BostonHousing$chas <- as.numeric(as.character(BostonHousing$chas))
x <- as.matrix(BostonHousing[,1:13])
y <- as.matrix(BostonHousing[,14])
# fit model
fit <- glmnet(x, y, family="gaussian", alpha=0.5, lambda=0.001)
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, x, type="link")
# summarize accuracy
mse <- mean((y - predictions)^2)
print(mse)
```

Listing 12.8: Example of GLMNET algorithm for regression.

It can also be configured to perform three important types of regularization: lasso, ridge and elastic net by configuring the `alpha` parameter to 1, 0 or in [0,1] respectively. The `glmnet` implementation can be used in `caret` for classification as follows:

```
# load packages
library(caret)
library(mlbench)
library(glmnet)
# Load the dataset
data(PimaIndiansDiabetes)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.glmnet <- train(diabetes~., data=PimaIndiansDiabetes, method="glmnet",
    metric="Accuracy", preProcess=c("center", "scale"), trControl=trainControl)
# summarize fit
print(fit.glmnet)
```

Listing 12.9: Example of GLMNET algorithm for classification in caret.

The `glmnet` implementation can be used in `caret` for regression as follows:

```
# load packages
library(caret)
library(mlbench)
library(glmnet)
# Load the dataset
data(BostonHousing)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.glmnet <- train(medv~., data=BostonHousing, method="glmnet", metric="RMSE",
    preProcess=c("center", "scale"), trControl=trainControl)
# summarize fit
print(fit.glmnet)
```

Listing 12.10: Example of GLMNET algorithm for regression.

12.4 Nonlinear Algorithms

These are machine learning algorithms that make fewer assumptions about the underlying function being modeled. As such, they have a higher variance but often result in higher accuracy. Their increased flexibility also can make them slower to train or increase their memory requirements.

12.4.1 k-Nearest Neighbors

The `knn3()` function is in the `caret` package and does not create a model. Instead it makes predictions from the training dataset directly. It can be used for classification or regression.

Classification Example:

```
# load the packages
library(caret)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# fit model
fit <- knn3(diabetes~., data=PimaIndiansDiabetes, k=3)
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, PimaIndiansDiabetes[,1:8], type="class")
# summarize accuracy
table(predictions, PimaIndiansDiabetes$diabetes)
```

Listing 12.11: Example of KNN algorithm for classification.

Regression Example:

```
# load the packages
library(caret)
library(mlbench)
# load data
data(BostonHousing)
BostonHousing$chas <- as.numeric(as.character(BostonHousing$chas))
x <- as.matrix(BostonHousing[,1:13])
y <- as.matrix(BostonHousing[,14])
# fit model
fit <- knnreg(x, y, k=3)
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, x)
# summarize accuracy
mse <- mean((BostonHousing$medv - predictions)^2)
print(mse)
```

Listing 12.12: Example of KNN algorithm for regression.

The `knn3` implementation can be used within the `caret train()` function for classification as follows:

```
# load packages
library(caret)
```

```
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.knn <- train(diabetes~., data=PimaIndiansDiabetes, method="knn", metric="Accuracy",
    preProcess=c("center", "scale"), trControl=trainControl)
# summarize fit
print(fit.knn)
```

Listing 12.13: Example of KNN algorithm for classification in caret.

The `knn3` implementation can be used within the `caret train()` function for regression as follows:

```
# load packages
library(caret)
data(BostonHousing)
# Load the dataset
data(BostonHousing)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.knn <- train(medv~., data=BostonHousing, method="knn", metric="RMSE",
    preProcess=c("center", "scale"), trControl=trainControl)
# summarize fit
print(fit.knn)
```

Listing 12.14: Example of KNN algorithm for regression in caret.

12.4.2 Naive Bayes

The `naiveBayes()` function is in the `e1071` package and models the probabilities of each attribute to the outcome variable independently. It can be used for classification problems.

```
# load the packages
library(e1071)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# fit model
fit <- naiveBayes(diabetes~., data=PimaIndiansDiabetes)
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, PimaIndiansDiabetes[,1:8])
# summarize accuracy
table(predictions, PimaIndiansDiabetes$diabetes)
```

Listing 12.15: Example of Naive Bayes algorithm for classification.

A very similar naive Bayes implementation (`NaiveBayes` from the `klaR` package) can be used with the `caret` package as follows:

```
# load packages
```

```
library(caret)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.nb <- train(diabetes~., data=PimaIndiansDiabetes, method="nb", metric="Accuracy",
    trControl=trainControl)
# summarize fit
print(fit.nb)
```

Listing 12.16: Example of Naive Bayes algorithm for classification in caret.

12.4.3 Support Vector Machine

The `ksvm()` function is in the `kernlab` package and can be used for classification or regression. It is a wrapper for the `LIBSVM` software package and provides a suite of kernel types and configuration options. These examples use a Radial Basis kernel.

Classification Example:

```
# load the packages
library(kernlab)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# fit model
fit <- ksvm(diabetes~., data=PimaIndiansDiabetes, kernel="rbfdot")
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, PimaIndiansDiabetes[,1:8], type="response")
# summarize accuracy
table(predictions, PimaIndiansDiabetes$diabetes)
```

Listing 12.17: Example of SVM algorithm for classification.

Regression Example:

```
# load the packages
library(kernlab)
library(mlbench)
# load data
data(BostonHousing)
# fit model
fit <- ksvm(medv~., BostonHousing, kernel="rbfdot")
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, BostonHousing)
# summarize accuracy
mse <- mean((BostonHousing$medv - predictions)^2)
print(mse)
```

Listing 12.18: Example of SVM algorithm for regression.

The SVM with Radial Basis kernel implementation can be used with `caret` for classification as follows:

```
# load packages
library(caret)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.svmRadial <- train(diabetes~., data=PimaIndiansDiabetes, method="svmRadial",
    metric="Accuracy", trControl=trainControl)
# summarize fit
print(fit.svmRadial)
```

Listing 12.19: Example of SVM algorithm for classification in caret.

The SVM with Radial Basis kernel implementation can be used with `caret` for regression as follows:

```
# load packages
library(caret)
library(mlbench)
# Load the dataset
data(BostonHousing)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.svmRadial <- train(medv~., data=BostonHousing, method="svmRadial", metric="RMSE",
    trControl=trainControl)
# summarize fit
print(fit.svmRadial)
```

Listing 12.20: Example of SVM algorithm for regression in caret.

12.4.4 Classification and Regression Trees

The `rpart()` function in the `rpart` package provides an implementation of CART (Classification And Regression Trees) for classification and regression.

Classification Example:

```
# load the packages
library(rpart)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# fit model
fit <- rpart(diabetes~., data=PimaIndiansDiabetes)
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, PimaIndiansDiabetes[,1:8], type="class")
# summarize accuracy
table(predictions, PimaIndiansDiabetes$diabetes)
```

Listing 12.21: Example of CART algorithm for classification.

Regression Example:

```
# load the packages
library(rpart)
library(mlbench)
# load data
data(BostonHousing)
# fit model
fit <- rpart(medv~., data=BostonHousing, control=rpart.control(minsplit=5))
# summarize the fit
print(fit)
# make predictions
predictions <- predict(fit, BostonHousing[,1:13])
# summarize accuracy
mse <- mean((BostonHousing$medv - predictions)^2)
print(mse)
```

Listing 12.22: Example of CART algorithm for regression.

The `rpart` implementation can be used with `caret` for classification as follows:

```
# load packages
library(caret)
library(mlbench)
# Load the dataset
data(PimaIndiansDiabetes)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=5)
fit.rpart <- train(diabetes~., data=PimaIndiansDiabetes, method="rpart", metric="Accuracy",
    trControl=trainControl)
# summarize fit
print(fit.rpart)
```

Listing 12.23: Example of CART algorithm for classification in caret.

The `rpart` implementation can be used with `caret` for regression as follows:

```
# load packages
library(caret)
library(mlbench)
# Load the dataset
data(BostonHousing)
# train
set.seed(7)
trainControl <- trainControl(method="cv", number=2)
fit.rpart <- train(medv~., data=BostonHousing, method="rpart", metric="RMSE",
    trControl=trainControl)
# summarize fit
print(fit.rpart)
```

Listing 12.24: Example of CART algorithm for regression in caret.

12.5 Other Algorithms

There are many other algorithms provided by R and available in `caret`. I would advise you to explore them and add more algorithms to your own shortlist of must-try algorithms on your next machine learning project. You can find a mapping of machine learning functions and packages to their name in the `caret` package on the Caret Model List webpage[1].

This page is useful if you are using an algorithm in `caret` and want to know which package it belongs to so that you can read up on the parameters and get more out of it. This page is also useful if you are using a machine learning algorithm directly in R and want to know how it can be used in caret.

12.6 Summary

In this lesson you discovered a diverse set of 8 algorithms that you can use to spot-check on your datasets. Specifically:

- Linear Regression.
- Logistic Regression.
- Linear Discriminant Analysis.
- Regularized Regression.
- k-Nearest Neighbors.
- Naive Bayes.
- Support Vector Machine.
- Classification and Regression Trees.

You learned which packages and functions to use for each algorithm. You also learned how you can use each algorithm with the `caret` package that provides algorithm evaluation and tuning capabilities. You can use these algorithms as a template for spot-checking on your current or next machine learning project in R.

12.6.1 Next Step

You have learned how to evaluate the skill of models, metrics and how to run the algorithms themselves. In the next lesson you will discover how you can tie all three of these elements together on a predictive modeling project and compare and choose between multiple skillful models on your dataset.

[1] https://topepo.github.io/caret/modelList.html

Chapter 13

Compare 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 8 techniques that you can use to compare machine learning algorithms in R. After completing this lesson, you will know:

1. How to compare model skill using a table summary.
2. How to review and compare model skill using a range of different plots.
3. How to compare model skill pairwise using xy-plots.
4. How to check if the difference in model skill is statistically significant.

Let's get started.

13.1 Choose The Best Machine Learning Model

How do you choose the best model for your dataset? When you work on a machine learning project, you often end up with multiple good models to choose from. Each model will have different performance characteristics. Using resampling methods like k-fold cross validation, you can get an estimate for how accurate each model may be on unseen data. You need to be able to use these estimates to choose one or two best models from the suite of models that you have created.

13.1.1 Compare Machine Learning Models Carefully

When you have a new dataset it is a good idea to visualize the data using a number of different graphing techniques in order to look at the data from different perspectives. The same idea applies to model selection. You should use a number of different ways of looking at the estimated accuracy of your machine learning algorithms in order to choose the one or two to finalize.

The way that you can do that is to use different visualization methods to show the average accuracy, variance and other properties of the distribution of estimated model accuracies. In the next section you will discover exactly how you can do that in R. This lesson is presented as a

case study. Throughout the case study you will create a number of machine learning models for the Pima Indians diabetes dataset. You will then use a suite of different visualization techniques to compare the estimated accuracy of the models. This case study is split up into three sections:

1. **Prepare Dataset**. Load the packages and dataset ready to train the models.
2. **Train Models**. Train standard machine learning models on the dataset ready for evaluation.
3. **Compare Models**. Compare the trained models using 8 different techniques.

13.2 Prepare Dataset

The dataset used in this case study is the Pima Indians diabetes dataset loaded from the `mlbench` package first described in Chapter 5. It is a binary classification problem to determine whether a patient will have an onset of diabetes within the next 5 years. The input attributes are numeric and describe medical details for female patients. Let's load the packages and dataset for this case study.

```
# load packages
library(mlbench)
library(caret)
# load the dataset
data(PimaIndiansDiabetes)
```

Listing 13.1: Load packages and dataset.

13.3 Train Models

In this section we will train the 5 machine learning models that we will compare in the next section. We will use repeated cross validation with 10 folds and 3 repeats, a common standard configuration for comparing models. The evaluation metric is accuracy and kappa because they are easy to interpret. The algorithms were chosen for their diversity of representation and learning style. They include:

- Classification and Regression Trees (CART).
- Linear Discriminant Analysis (LDA).
- Support Vector Machine with Radial Basis Function (SVM).
- k-Nearest Neighbors (KNN).
- Random Forest (RF).

After the models are trained, they are added to a list and `resamples()` is called on the list of models. This function checks that the models are comparable and that they used the same training scheme (`trainControl` configuration). This object contains the evaluation metrics for each fold and each repeat for each evaluated algorithm. The functions that we use in the next section all expect an object with this data.

```
# prepare training scheme
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
# CART
set.seed(7)
fit.cart <- train(diabetes~., data=PimaIndiansDiabetes, method="rpart",
    trControl=trainControl)
# LDA
set.seed(7)
fit.lda <- train(diabetes~., data=PimaIndiansDiabetes, method="lda", trControl=trainControl)
# SVM
set.seed(7)
fit.svm <- train(diabetes~., data=PimaIndiansDiabetes, method="svmRadial",
    trControl=trainControl)
# KNN
set.seed(7)
fit.knn <- train(diabetes~., data=PimaIndiansDiabetes, method="knn", trControl=trainControl)
# Random Forest
set.seed(7)
fit.rf <- train(diabetes~., data=PimaIndiansDiabetes, method="rf", trControl=trainControl)
# collect resamples
results <- resamples(list(CART=fit.cart, LDA=fit.lda, SVM=fit.svm, KNN=fit.knn, RF=fit.rf))
```

Listing 13.2: Train machine learning models and collect resample statistics.

13.4 Compare Models

In this section we will look at 8 different techniques for comparing the estimated accuracy of the constructed models.

13.4.1 Table Summary

The Table Summary is the easiest comparison that you can do. Simply call the `summary()` function and pass it the resamples result. It will create a table with one algorithm for each row and evaluation metrics for each column.

```
# summarize differences between models
summary(results)
```

Listing 13.3: Summarize model resample statistics.

I find it useful to look at the mean and the max columns.

```
Accuracy
      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
CART 0.6234 0.7115 0.7403 0.7382 0.7760 0.8442 0
LDA  0.6711 0.7532 0.7662 0.7759 0.8052 0.8701 0
SVM  0.6711 0.7403 0.7582 0.7651 0.7890 0.8961 0
KNN  0.6184 0.6984 0.7321 0.7299 0.7532 0.8182 0
RF   0.6711 0.7273 0.7516 0.7617 0.7890 0.8571 0

Kappa
      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
CART 0.1585 0.3296 0.3765 0.3934 0.4685 0.6393 0
```

```
LDA  0.2484 0.4196 0.4516 0.4801 0.5512 0.7048 0
SVM  0.2187 0.3889 0.4167 0.4520 0.5003 0.7638 0
KNN  0.1113 0.3228 0.3867 0.3819 0.4382 0.5867 0
RF   0.2624 0.3787 0.4516 0.4588 0.5193 0.6781 0
```

Listing 13.4: Output of model resample statistics.

13.4.2 Box and Whisker Plots

Box and Whisker Plots are a useful way to look at the spread of the estimated accuracies for different methods and how they relate.

```
# box and whisker plots to compare models
scales <- list(x=list(relation="free"), y=list(relation="free"))
bwplot(results, scales=scales)
```

Listing 13.5: Calculate box and whisker plots.

Note that the boxes are ordered from highest to lowest mean accuracy. I find it useful to look at the mean values (dots) and the overlaps of the boxes (middle 50% of results).

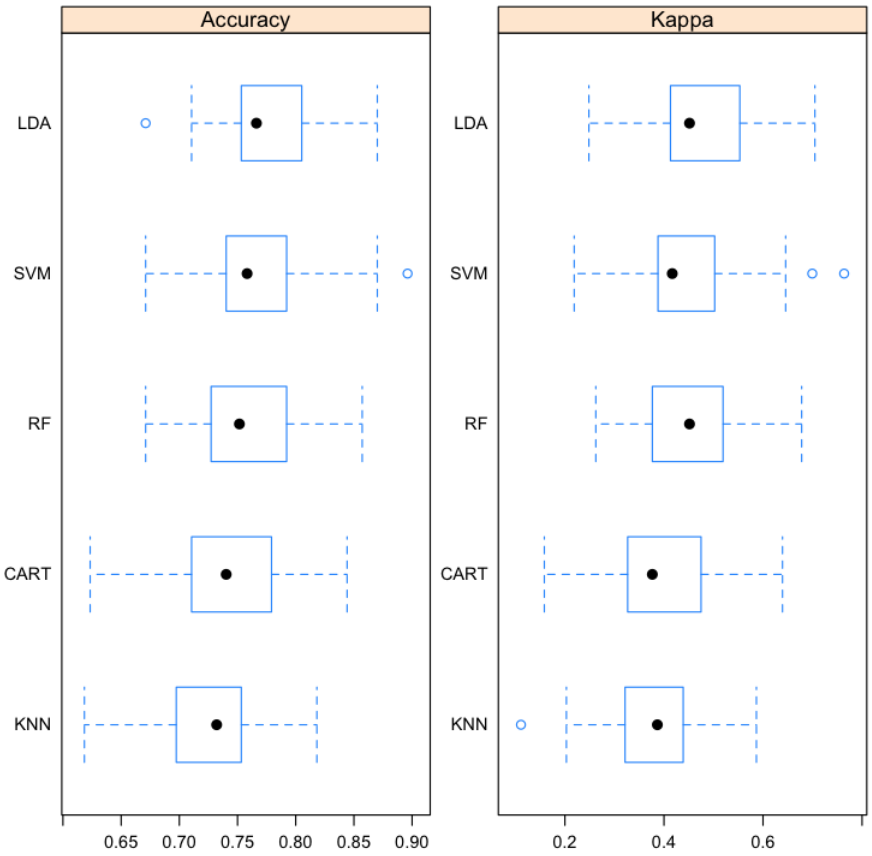

Figure 13.1: Compare Machine Learning Algorithms in R Box and Whisker Plots

13.4.3 Density Plots

You can show the distribution of model accuracy as density plots. This is a useful way to evaluate the overlap in the estimated behavior of algorithms. Note the use of | to show ticks of the data points below the distributions.

```
# density plots of accuracy
scales <- list(x=list(relation="free"), y=list(relation="free"))
densityplot(results, scales=scales, pch = "|")
```

Listing 13.6: Calculate density plots.

I like to look at the differences in the peaks as well as the spread or base of the distributions.

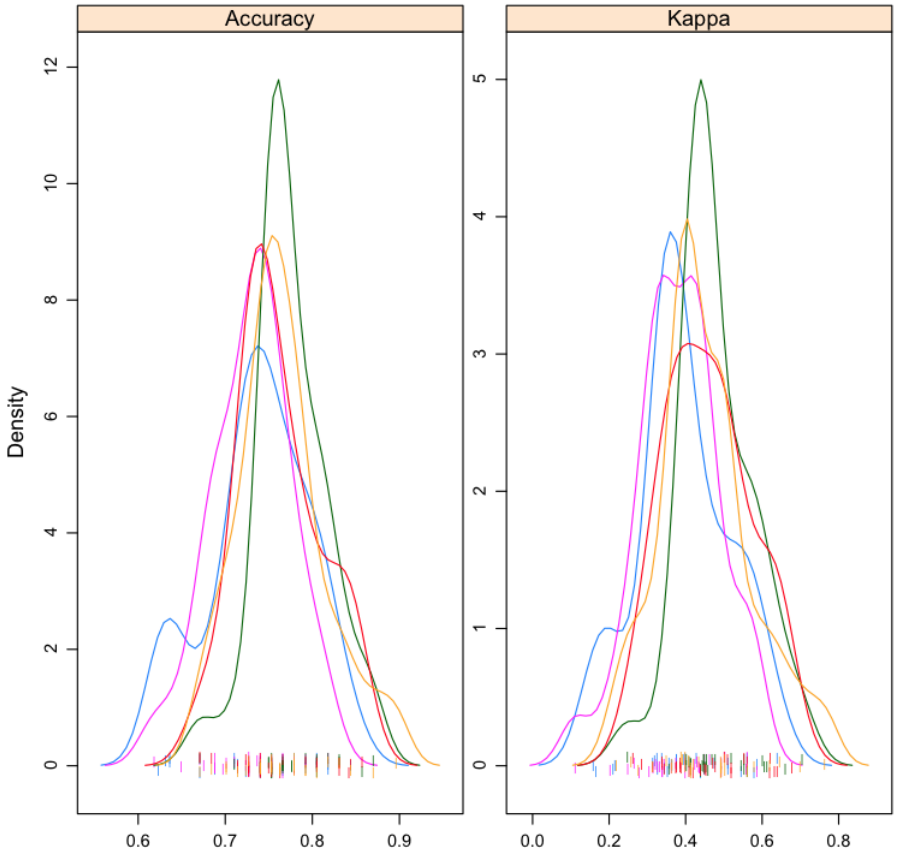

Figure 13.2: Compare Machine Learning Algorithms in R Density Plots

13.4.4 Dot Plots

These are useful plots as they show both the mean estimated accuracy as well as the 95% confidence interval (e.g. the range in which 95% of observed scores fell).

```
# dot plots of accuracy
scales <- list(x=list(relation="free"), y=list(relation="free"))
dotplot(results, scales=scales)
```

Listing 13.7: Calculate dot plots.

I find it useful to compare the means and eye-ball the overlap of the spreads between algorithms.

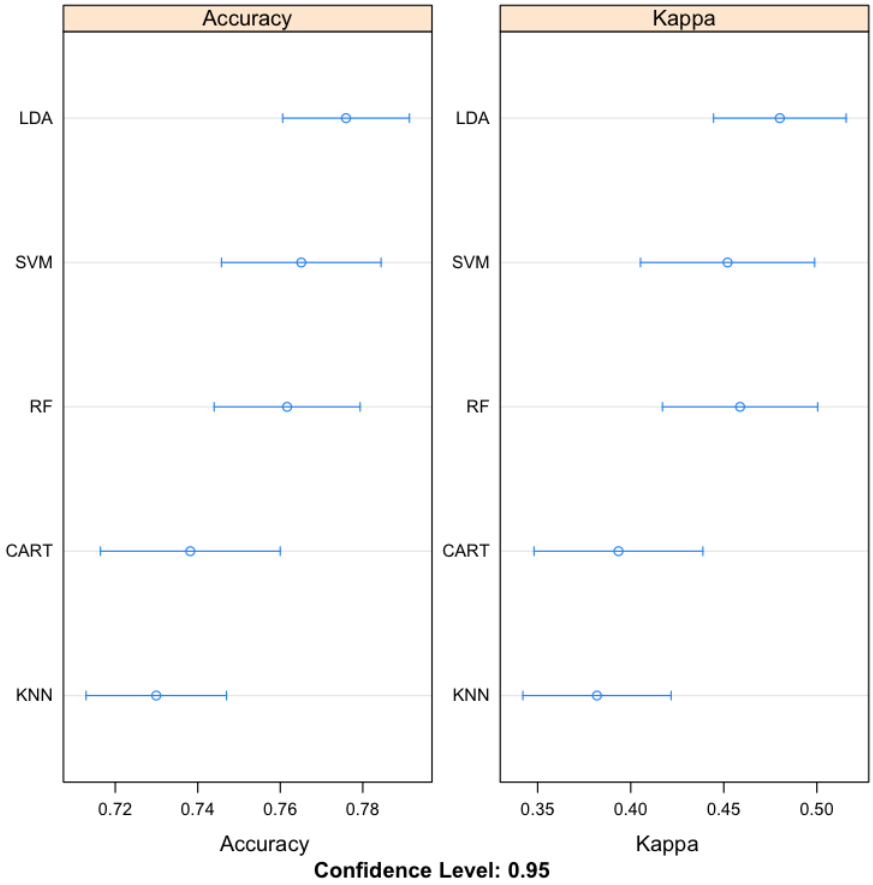

Figure 13.3: Compare Machine Learning Algorithms in R Dot Plots

13.4.5 Parallel Plots

Parallel Plots are another way to look at the data. It shows how each trial of each cross validation fold behaved for each of the algorithms tested. It can help you see how those hold-out subsets that were difficult for one algorithm affected other algorithms.

```
# parallel plots to compare models
parallelplot(results)
```

Listing 13.8: Calculate parallel plots.

This can be a tricky plot to interpret. I like to think that this can be helpful in thinking about how different methods could be combined in an ensemble prediction (e.g. stacking) at a later time, especially if you see correlated movements in opposite directions.

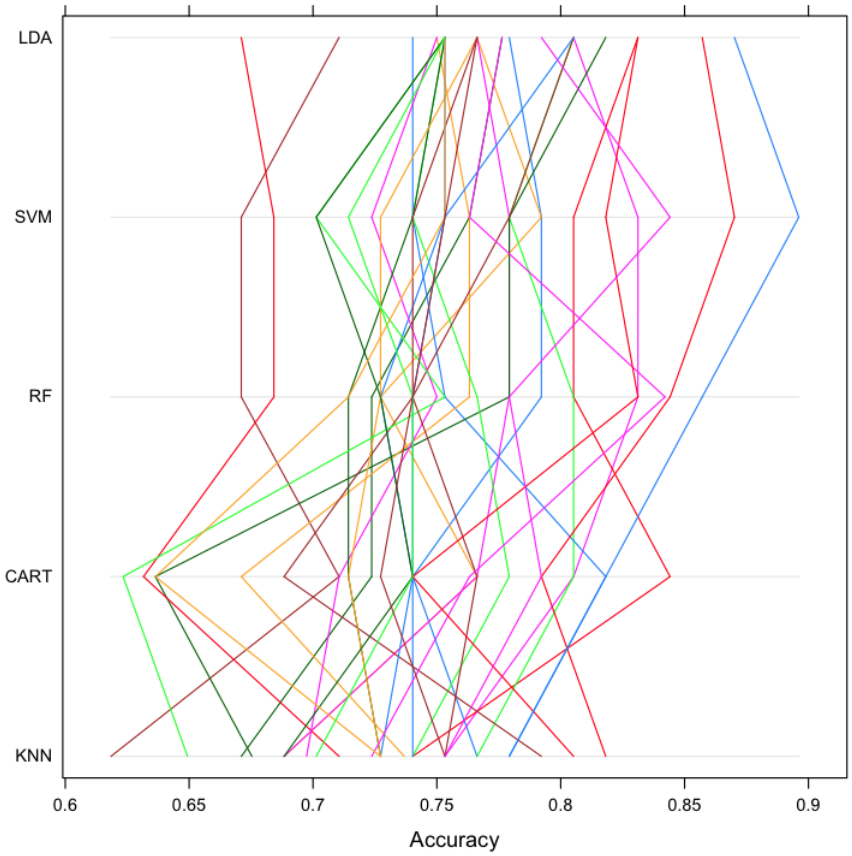

Figure 13.4: Compare Machine Learning Algorithms in R Parallel Plots

13.4.6 Scatterplot Matrix

This creates a scatter plot matrix of all fold-trial results for an algorithm compared to the same fold-trial results for all other algorithms. All pairs are plotted.

```
# pairwise scatter plots of predictions to compare models
splom(results)
```

Listing 13.9: Calculate scatter plot matrix plots.

This is invaluable when considering whether the predictions from two different algorithms are correlated. If weakly correlated, they are good candidates for being combined in an ensemble prediction. For example, eye-balling the graphs it looks like LDA and SVM look strongly correlated, as does SVM and RF. SVM and CART look weakly correlated.

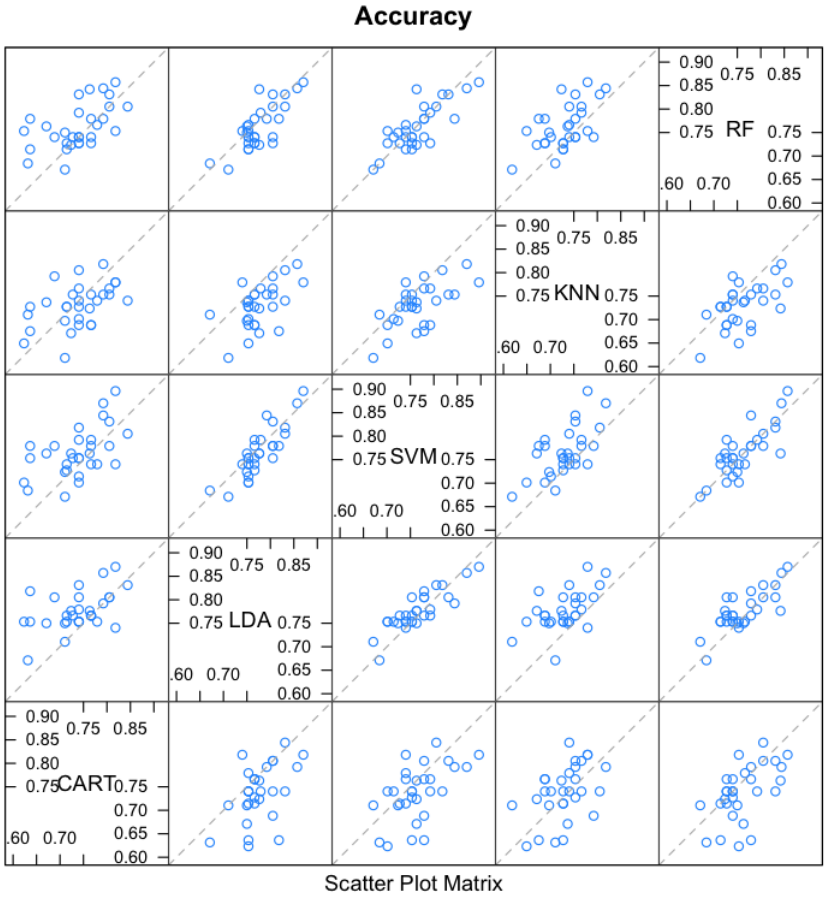

Figure 13.5: Compare Machine Learning Algorithms in R Scatter Plot Matrix

13.4.7 Pairwise xyPlots

You can zoom in on one pairwise comparison of the accuracy of trial-folds for two machine learning algorithms with an xyplot.

```
# xyplot plots to compare models
xyplot(results, models=c("LDA", "SVM"))
```

Listing 13.10: Calculate xy-plot.

In this case we can see the seemingly correlated accuracy of the LDA and SVM models.

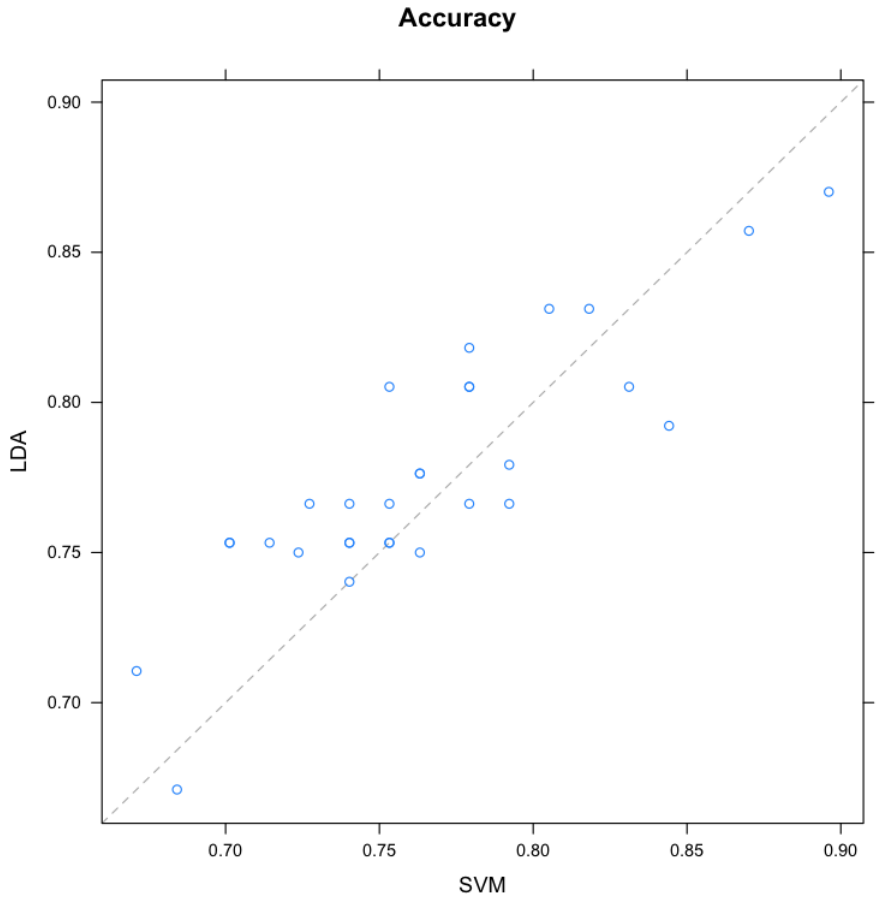

Figure 13.6: Compare Machine Learning Algorithms in R Pairwise Scatter Plot

13.4.8 Statistical Significance Tests

You can calculate the significance of the differences between the metric distributions of different machine learning algorithms. We can summarize the results directly by calling the `summary()` function.

```
# difference in model predictions
diffs <- diff(results)
# summarize p-values for pairwise comparisons
summary(diffs)
```

Listing 13.11: Calculate and summarize statistical significance.

We can see a table of pairwise statistical significance scores. The lower diagonal of the table shows p-values for the null hypothesis (distributions are the same), smaller is better. We can see no difference between CART and KNN, we can also see little difference between the distributions for LDA and SVM.

The upper diagonal of the table shows the estimated difference between the distributions. If we think that LDA is the most accurate model from looking at the previous graphs, we can get an estimate of how much better than specific other models in terms of absolute accuracy. These scores can help with any accuracy claims you might want to make between specific algorithms.

```
p-value adjustment: bonferroni
```

```
Upper diagonal: estimates of the difference
Lower diagonal: p-value for H0: difference = 0

Accuracy
     CART      LDA       SVM       KNN       RF
CART           -0.037759 -0.026908 0.008248 -0.023473
LDA  0.0050068           0.010851 0.046007 0.014286
SVM  0.0919580 0.3390336           0.035156 0.003435
KNN  1.0000000 1.218e-05 0.0007092        -0.031721
RF   0.1722106 0.1349151 1.0000000 0.0034441
```

Listing 13.12: Output of statistical significance.

A good tip is to increase the number of trials to increase the size of the populations and perhaps result in more precise p-values. You can also plot the differences, but I find the plots a lot less useful than the above summary table.

13.5 Summary

In this lesson you discovered 8 different techniques that you can use to compare the estimated accuracy of your machine learning models in R. The 8 techniques you discovered were:

- Table Summary.
- Box and Whisker Plots.
- Density Plots.
- Dot Plots.
- Parallel Plots.
- Scatter Plot Matrix.
- Pairwise xyPlots.
- Statistical Significance Tests.

13.5.1 Next Step

In the last few lessons you have learned each element involved in spot-checking algorithms and now how to compare the skill of the algorithms that you have spot-checked. In the next lesson you will look at the first of two approaches that you can use to improve the skill of your models called algorithm tuning.

Chapter 14

Tune Machine Learning Algorithms

It is difficult to find a good machine learning algorithm for your problem. But once you do find a good algorithm, how do you get the best performance out of it? In this lesson you will discover three ways that you can tune the parameters of a machine learning algorithm in R. After completing this lesson you will know:

- How to use the `caret` package to perform a grid or random search of algorithm parameters.
- How to use the tools that come with algorithms to tune parameters.
- How to extend `caret` algorithm tuning to search additional algorithm parameters.

Let's get started.

14.1 Get Better Accuracy From Top Algorithms

It is difficult to find a good or even a well performing machine learning algorithm for your dataset. Through a process of trial and error you can settle on a shortlist of algorithms that show promise, but how do you know which is the best. You could use the default parameters for each algorithm. These are the parameters set by rules of thumb or suggestions in books and research papers. But how do you know the algorithms that you are settling on are showing their best performance?

14.1.1 Use Algorithm Tuning To Search For Algorithm Parameters

The answer is to search for good or even best combinations of algorithm parameters for your problem. You need a process to tune each machine learning algorithm to know that you are getting the most out of it. Once tuned, you can make an objective comparison between the algorithms on your shortlist. Searching for algorithm parameters can be difficult. There are many options, such as:

- What parameters to tune?
- What search method to use to locate good algorithm parameters?
- What test options to use to limit overfitting the training data?

14.2 Tune Machine Learning Algorithms

You can tune your machine learning algorithm parameters in R. Generally, the approaches in this section assume that you already have a shortlist of well performing machine learning algorithms for your problem from which you are looking to get better performance. An excellent way to create your shortlist of well performing algorithms is to use the `caret` package. In this lesson we will look at three methods that you can use in R to tune algorithm parameters:

- Using the `caret` R package.
- Using tools that come with the algorithm.
- Designing your own parameter search.

Before we start tuning, let's setup our environment and test data.

14.3 Test Setup

Let's take a quick look at the data and the algorithm we will use in this lesson.

14.3.1 Test Dataset

In this case study, we will use the Sonar test problem described in Chapter 5. This is a dataset from the UCI Machine Learning Repository that describes radar returns as either bouncing off metal or rocks. It is a binary classification problem with 60 numerical input features that describe the properties of the radar return. This is not a particularly difficult dataset, but is non-trivial and interesting for this example. Let's load the required packages and load the dataset from the `mlbench` package.

```
# Load packages
library(randomForest)
library(mlbench)
library(caret)
# Load Dataset
data(Sonar)
dataset <- Sonar
x <- dataset[,1:60]
y <- dataset[,61]
```

Listing 14.1: Load packages and datasets.

14.3.2 Test Algorithm

We will use the popular Random Forest algorithm as the subject of our algorithm tuning. Random Forest is not necessarily the best algorithm for this dataset, but it is a very popular algorithm and no doubt you will find tuning it a useful exercise in you own machine learning work.

When tuning an algorithm, it is important to have a good understanding of your algorithm so that you know what effect the parameters have on the model you are creating. In this case

study, we will stick to tuning two parameters, namely the `mtry` and the `ntree` parameters that have the following effect on our Random Forest model. There are many other parameters, but these two parameters are perhaps the most likely to have the biggest effect on your final accuracy.

Direct from the help page for the `randomForest()` function in R:

- `mtry`: Number of variables randomly sampled as candidates at each split.
- `ntree`: Number of trees to grow.

Let's create a baseline for comparison by using the recommended defaults for each parameter and `mtry=floor(sqrt(ncol(x)))` or `mtry=7` and `ntree=500`.

```
# Create model with default paramters
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
seed <- 7
metric <- "Accuracy"
set.seed(seed)
mtry <- sqrt(ncol(x))
tunegrid <- expand.grid(.mtry=mtry)
rfDefault <- train(Class~., data=dataset, method="rf", metric=metric, tuneGrid=tunegrid,
    trControl=trainControl)
print(rfDefault)
```

Listing 14.2: Calculate baseline performance of algorithm.

We can see our estimated accuracy is 81.3%.

```
Resampling results

  Accuracy  Kappa     Accuracy SD Kappa SD
  0.8138384 0.6209924 0.0747572   0.1569159
```

Listing 14.3: Output of baseline performance of algorithm.

14.4 Tune Using Caret

The `caret` package provides an excellent facility to tune machine learning algorithm parameters. Not all machine learning algorithms are available in `caret` for tuning. The choice of parameters is left to the developers of the package, namely Max Kuhn. Only those algorithm parameters that have a large effect (e.g. really require tuning in Kuhn's opinion) are available for tuning in `caret`.

As such, only the `mtry` parameter is available for tuning in `caret`. The reason is its effect on the final accuracy and that it must be found empirically for a dataset. The `ntree` parameter is different in that it can be as large as you like, and continues to increase the accuracy up to some point. It is less difficult or critical to tune and could be limited by compute time available more than anything.

14.4.1 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 (like the suggested equation above). Let's try a random search for `mtry` using the `caret` package:

```
# Random Search
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3, search="random")
set.seed(seed)
mtry <- sqrt(ncol(x))
rfRandom <- train(Class~., data=dataset, method="rf", metric=metric, tuneLength=15,
    trControl=trainControl)
print(rfRandom)
plot(rfRandom)
```

Listing 14.4: Tune algorithm using random search.

Note, that we are using a test harness similar to that which we would use to spot-check algorithms. Both 10-fold cross validation and 3 repeats slows down the search process, but is intended to limit and reduce overfitting on the training dataset. It won't remove overfitting entirely. Holding back a validation set for final checking is a great idea if you can spare the data.

```
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa      Accuracy SD  Kappa SD
  11    0.8218470  0.6365181  0.09124610   0.1906693
  14    0.8140620  0.6215867  0.08475785   0.1750848
  17    0.8030231  0.5990734  0.09595988   0.1986971
  24    0.8042929  0.6002362  0.09847815   0.2053314
  30    0.7933333  0.5798250  0.09110171   0.1879681
  34    0.8015873  0.5970248  0.07931664   0.1621170
  45    0.7932612  0.5796828  0.09195386   0.1887363
  47    0.7903896  0.5738230  0.10325010   0.2123314
  49    0.7867532  0.5673879  0.09256912   0.1899197
  50    0.7775397  0.5483207  0.10118502   0.2063198
  60    0.7790476  0.5513705  0.09810647   0.2005012
```

Listing 14.5: Output of tuning algorithm using random search.

We can see that the most accurate value for `mtry` was 11 with an accuracy of 82.1%.

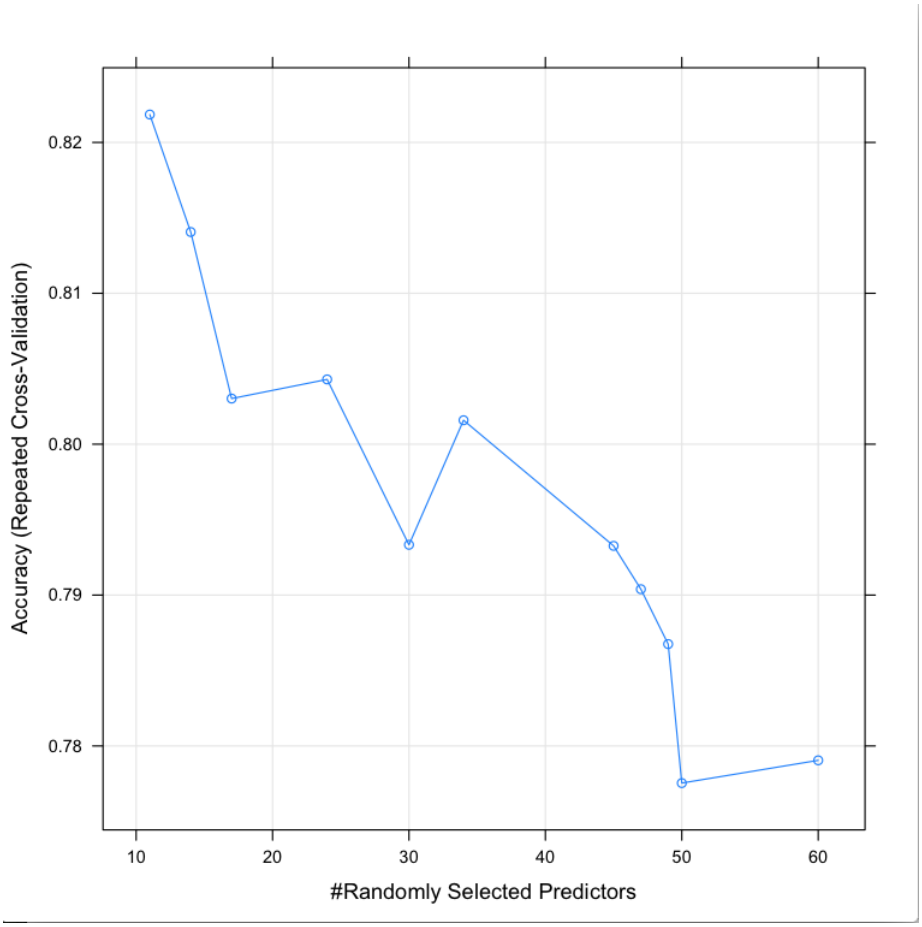

Figure 14.1: Tune Random Forest Parameters in R Using Random Search

14.4.2 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
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3, search="grid")
set.seed(seed)
tunegrid <- expand.grid(.mtry=c(1:15))
rfGrid <- train(Class~., data=dataset, method="rf", metric=metric, tuneGrid=tunegrid,
    trControl=trainControl)
print(rfGrid)
plot(rfGrid)
```

Listing 14.6: Tune algorithm using grid search.

We can see that the most accurate value for `mtry` was 2 with an accuracy of 83.78%.

```
Resampling results across tuning parameters:

  mtry  Accuracy  Kappa      Accuracy SD Kappa SD
   1    0.8377273 0.6688712 0.07154794  0.1507990
```

```
 2    0.8378932 0.6693593 0.07185686  0.1513988
 3    0.8314502 0.6564856 0.08191277  0.1700197
 4    0.8249567 0.6435956 0.07653933  0.1590840
 5    0.8268470 0.6472114 0.06787878  0.1418983
 6    0.8298701 0.6537667 0.07968069  0.1654484
 7    0.8282035 0.6493708 0.07492042  0.1584772
 8    0.8232828 0.6396484 0.07468091  0.1571185
 9    0.8268398 0.6476575 0.07355522  0.1529670
10    0.8204906 0.6346991 0.08499469  0.1756645
11    0.8073304 0.6071477 0.09882638  0.2055589
12    0.8184488 0.6299098 0.09038264  0.1884499
13    0.8093795 0.6119327 0.08788302  0.1821910
14    0.8186797 0.6304113 0.08178957  0.1715189
15    0.8168615 0.6265481 0.10074984  0.2091663
```

Listing 14.7: Output of tuning algorithm using grid search.

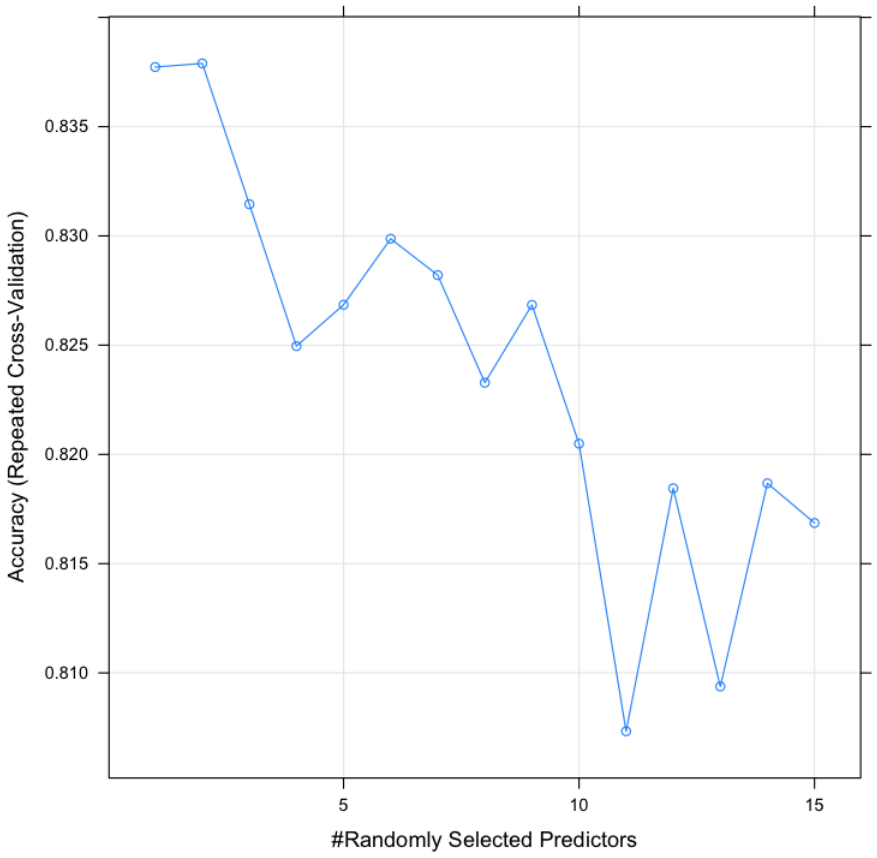

Figure 14.2: Tune Random Forest Parameters in R Using Grid Search

14.5 Tune Using Algorithm Tools

Some algorithm implementations provide tools for tuning the parameters of the algorithm. For example, the Random Forest algorithm implementation in the randomForest package provides the `tuneRF()` function that searches for optimal `mtry` values given your data.

```
# Algorithm Tune (tuneRF)
set.seed(seed)
bestmtry <- tuneRF(x, y, stepFactor=1.5, improve=1e-5, ntree=500)
print(bestmtry)
```

Listing 14.8: Tune algorithm using algorithm tools.

You can see that the most accurate value for `mtry` was 10 with an OOBError of 0.1442308. This does not really match up with what we saw in the `caret` repeated cross validation experiment above, where `mtry=10` gave an accuracy of 82.04%. Nevertheless, it is an alternate way to tune the algorithm.

```
       mtry  OOBError
5.OOB     5 0.1538462
7.OOB     7 0.1538462
10.OOB   10 0.1442308
15.OOB   15 0.1682692
```

Listing 14.9: Output of tuning algorithm using algorithm tools.

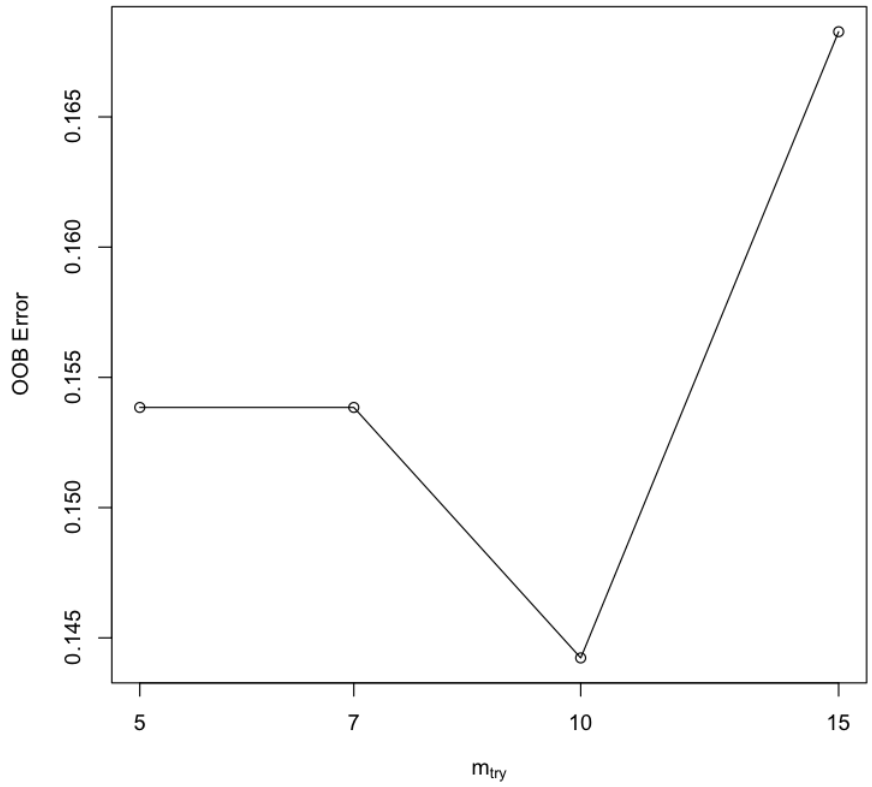

Figure 14.3: Tune Random Forest Parameters in R using tuneRF

14.6 Craft Your Own Parameter Search

Often you want to search for both the parameters that must be tuned (handled by `caret`) and the those that need to be scaled or adapted more generally for your dataset. You have to craft your own parameter search. Two popular recommendations are:

- **Tune Manually**: Write R code to create lots of models and compare their accuracy using `caret`.
- **Extend Caret**: Create an extension to `caret` that adds in additional parameters to `caret` for the algorithm you want to tune.

14.6.1 Tune Manually

We want to keep using `caret` because it provides a direct point of comparison to our previous models (apples to apples, even the same data splits) and because of the repeated cross validation test harness that we like as it reduces the severity of overfitting. One approach is to create many `caret` models for our algorithm and pass in a different set of parameters directly to the algorithm manually. Let's look at an example doing this to evaluate different values for `ntree` while holding `mtry` constant.

```
# Manual Search
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3, search="grid")
tunegrid <- expand.grid(.mtry=c(sqrt(ncol(x))))
modellist <- list()
for (ntree in c(1000, 1500, 2000, 2500)) {
  set.seed(seed)
  fit <- train(Class~., data=dataset, method="rf", metric=metric, tuneGrid=tunegrid,
      trControl=trainControl, ntree=ntree)
  key <- toString(ntree)
  modellist[[key]] <- fit
}
# compare results
results <- resamples(modellist)
summary(results)
dotplot(results)
```

Listing 14.10: Tune algorithm manually.

You can see that the most accurate value for `ntree` was perhaps 2,000 with a mean accuracy of 82.02% (a lift over our very first experiment using the default `mtry` value). The results perhaps suggest an optimal value for `ntree` between 2,000 and 2,500. Also note, we held `mtry` constant at the default value. We could repeat the experiment with a possible better `mtry=2` from the experiment above, or try combinations of `ntree` and `mtry` in case their effects on the algorithm and the result interact with each other.

```
Models: 1000, 1500, 2000, 2500
Number of resamples: 30

Accuracy
     Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
1000 0.600 0.8024 0.8500 0.8186 0.8571 0.9048 0
1500 0.600 0.8024 0.8095 0.8169 0.8571 0.9500 0
```

```
2000 0.619 0.8024 0.8095 0.8202 0.8620 0.9048 0
2500 0.619 0.8000 0.8095 0.8201 0.8893 0.9091 0
```

Listing 14.11: Output of tuning algorithm manually.

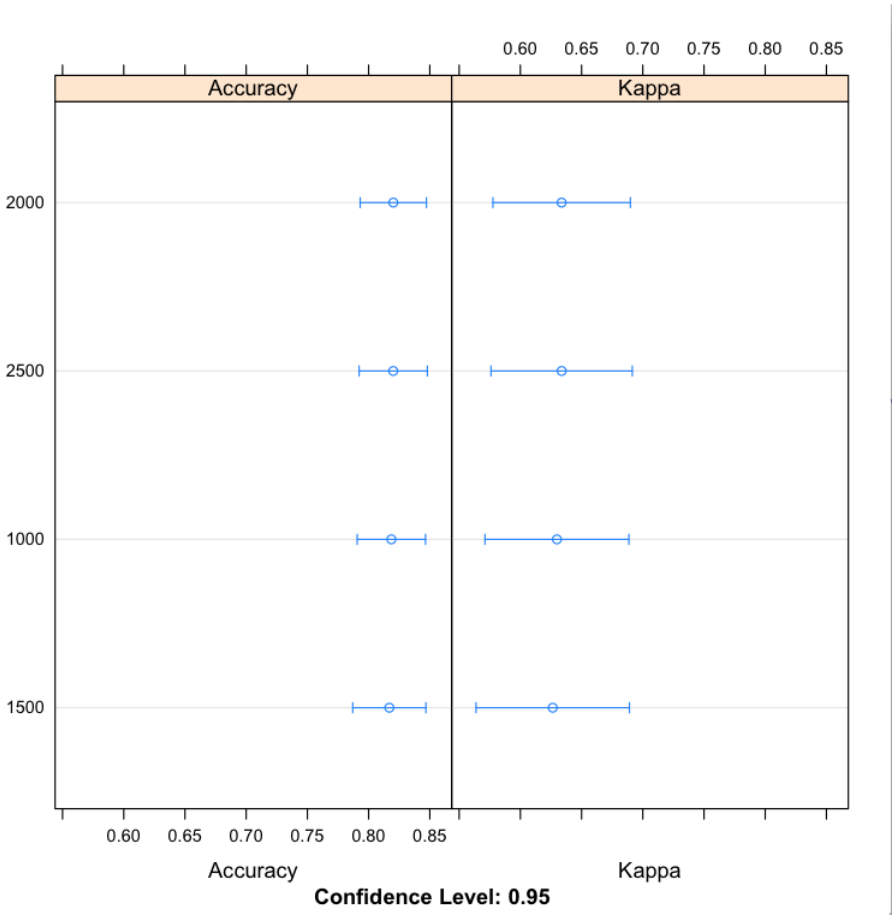

Figure 14.4: Tune Random Forest Parameters in R Manually

14.6.2 Extend Caret

Another approach is to create a new algorithm for `caret` to support. This is the same Random Forest algorithm you have been using, only modified so that it supports the tuning of multiple parameters. A risk with this approach is that the `caret` native support for the algorithm has additional or fancy code wrapping it that subtly but importantly changes its behavior. You may need to repeat prior experiments with your custom algorithm support.

We can define our own algorithm to use in `caret` by defining a list that contains a number of custom named elements that the `caret` package looks for, such as how to fit and how to predict. See below for a definition of a custom random forest algorithm for use with `caret` that takes both an `mtry` and `ntree` parameters.

```
customRF <- list(type="Classification", library="randomForest", loop=NULL)
customRF$parameters <- data.frame(parameter=c("mtry", "ntree"), class=rep("numeric", 2),
    label=c("mtry", "ntree"))
customRF$grid <- function(x, y, len=NULL, search="grid") {}
```

```
customRF$fit <- function(x, y, wts, param, lev, last, weights, classProbs, ...) {
  randomForest(x, y, mtry=param$mtry, ntree=param$ntree, ...)
}
customRF$predict <- function(modelFit, newdata, preProc=NULL, submodels=NULL)
   predict(modelFit, newdata)
customRF$prob <- function(modelFit, newdata, preProc=NULL, submodels=NULL)
   predict(modelFit, newdata, type = "prob")
customRF$sort <- function(x) x[order(x[,1]),]
customRF$levels <- function(x) x$classes
```

Listing 14.12: Define custom algorithm in caret.

Now, let's make use of this custom list in our call to the `caret` train function, and try tuning different values for `ntree` and `mtry`.

```
# train model
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
tunegrid <- expand.grid(.mtry=c(1:15), .ntree=c(1000, 1500, 2000, 2500))
set.seed(seed)
custom <- train(Class~., data=dataset, method=customRF, metric=metric, tuneGrid=tunegrid,
    trControl=trainControl)
print(custom)
plot(custom)
```

Listing 14.13: Tune algorithm custom algorithm definition in caret.

This may take a minute or two to run. You can see that the most accurate values for `ntree` and `mtry` were 2,000 and 2 with an accuracy of 84.43%. We do perhaps see some interaction effects between the number of trees and the value of `ntree`. Nevertheless, if we had chosen the best value for `mtry` found using grid search of 2 (above) and the best value of `ntree` found using grid search of 2,000 (above), in this case we would have achieved the same level of tuning found in this combined search. This is a nice confirmation.

```
  mtry  ntree  Accuracy   Kappa      Accuracy SD Kappa SD
   1    1000   0.8442424 0.6828299 0.06505226  0.1352640
   1    1500   0.8394805 0.6730868 0.05797828  0.1215990
   1    2000   0.8314646 0.6564643 0.06630279  0.1381197
   1    2500   0.8379654 0.6693773 0.06576468  0.1375408
   2    1000   0.8313781 0.6562819 0.06909608  0.1436961
   2    1500   0.8427345 0.6793793 0.07005975  0.1451269
   2    2000   0.8443218 0.6830115 0.06754346  0.1403497
   2    2500   0.8428066 0.6791639 0.06488132  0.1361329
   3    1000   0.8350216 0.6637523 0.06530816  0.1362839
   3    1500   0.8347908 0.6633405 0.06836512  0.1418106
...
  15    1000   0.8091486 0.6110154 0.08455439  0.1745129
  15    1500   0.8109668 0.6154780 0.08928549  0.1838700
  15    2000   0.8059740 0.6047791 0.08829659  0.1837809
  15    2500   0.8122511 0.6172771 0.08863418  0.1845635
```

Listing 14.14: Output of custom algorithm tuning in caret.

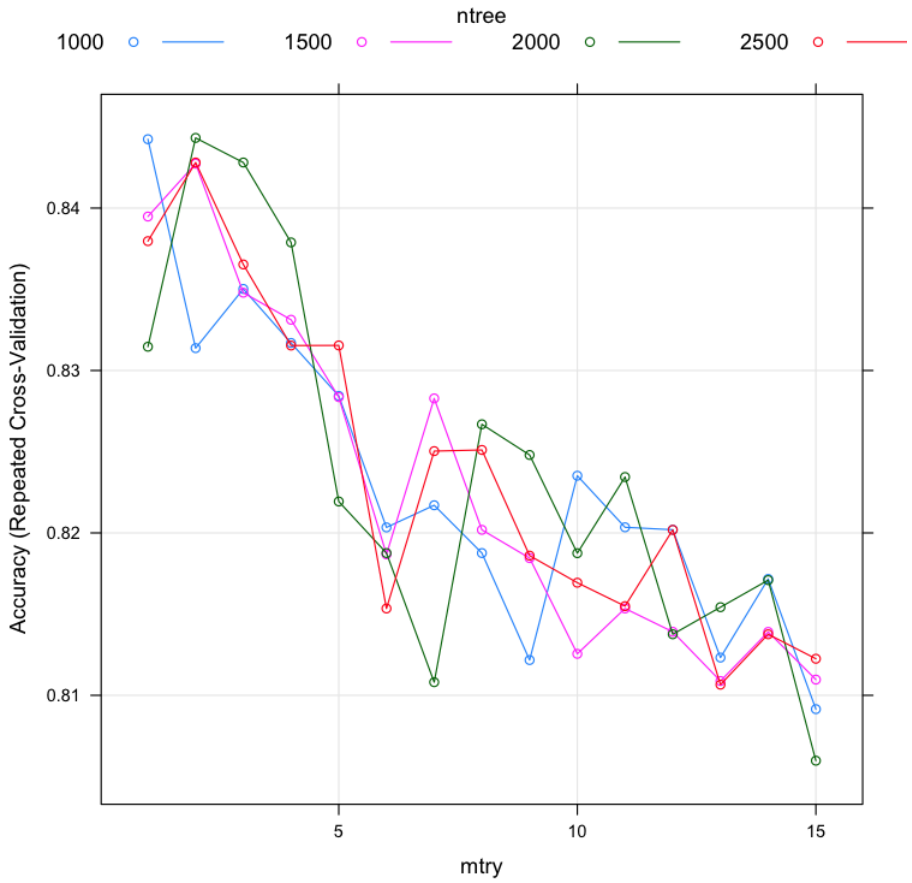

Figure 14.5: Custom Tuning of Random Forest parameters in R

14.7 Summary

In this lesson you discovered the importance of tuning well performing machine learning algorithms in order to get the best performance from them. You worked through an example of tuning the Random Forest algorithm and discovered three ways that you can tune a well performing algorithm.

1. Using the `caret` package.
2. Using tools that come with the algorithm.
3. Designing your own parameter search.

14.7.1 Next Step

In this lesson you discovered how you can search for combinations of algorithm parameters that yield the best results. In the next lesson you will discover another method that you can use to improve your results by combining predictions from multiple models called an ensemble.

Chapter 15

Combine Predictions From Multiple Machine Learning Models

Ensembles can give you a boost in accuracy on your dataset. In this lesson you will discover how you can create three of the most powerful types of ensembles in R. After completing this lesson, you will know:

1. How to use boosting and bagging algorithms on your dataset.
2. How to blend the predictions from multiple models using stacking.

Let's get started.

15.1 Increase The Accuracy Of Your Models

It can take time to find well performing machine learning algorithms for your dataset. This is because of the trial and error nature of applied machine learning. Once you have a shortlist of accurate models, you can use algorithm tuning to get the most from each algorithm. Another approach that you can use to increase accuracy on your dataset is to combine the predictions of multiple different models together. This is called an ensemble prediction.

15.1.1 Combine Model Predictions Into Ensemble Predictions

The three most popular methods for combining the predictions from different models are:

- **Bagging**. Building multiple models (typically models of the same type) from different subsamples of the training dataset.
- **Boosting**. Building multiple models (typically models of the same type) each of which learns to fix the prediction errors of a prior model in the chain.
- **Stacking**. Building multiple models (typically models of differing types) and a supervised model that learns how to best combine the predictions of the primary models.

In this lesson, we will look at each in turn. This lesson will not explain each of these ensemble methods.

15.2 Test Dataset

Before we start building ensembles, let's define our test setup. All of the examples of ensemble predictions in this case study will use the ionosphere dataset described in Chapter 5. This dataset describes high-frequency antenna returns from high energy particles in the atmosphere and whether the return shows structure or not. The problem is a binary classification that contains 351 instances and 35 numerical attributes. Let's load the packages and the dataset.

You may need to install the `caretEnsemble` package. Refer to Section 4.6 for help installing packages.

```
# Load packages
library(mlbench)
library(caret)
library(caretEnsemble)
# Load the dataset
data(Ionosphere)
dataset <- Ionosphere
dataset <- dataset[,-2]
dataset$V1 <- as.numeric(as.character(dataset$V1))
```

Listing 15.1: Load packages and dataset.

Note that the first attribute was a factor (0,1) and has been transformed to be numeric for consistency with all of the other numeric attributes. Also note that the second attribute is a constant and has been removed. Here is a sneak-peek at the first few rows of the ionosphere dataset.

```
> head(dataset)
  V1      V3       V4       V5       V6       V7       V8      V9      V10      V11      V12      V13
          V14      V15
1  1 0.99539 -0.05889 0.85243 0.02306 0.83398 -0.37708 1.00000 0.03760 0.85243 -0.17755
    0.59755 -0.44945 0.60536
2  1 1.00000 -0.18829 0.93035 -0.36156 -0.10868 -0.93597 1.00000 -0.04549 0.50874 -0.67743
    0.34432 -0.69707 -0.51685
3  1 1.00000 -0.03365 1.00000 0.00485 1.00000 -0.12062 0.88965 0.01198 0.73082 0.05346
    0.85443 0.00827 0.54591
4  1 1.00000 -0.45161 1.00000 1.00000 0.71216 -1.00000 0.00000 0.00000 0.00000 0.00000
    0.00000 0.00000 -1.00000
5  1 1.00000 -0.02401 0.94140 0.06531 0.92106 -0.23255 0.77152 -0.16399 0.52798 -0.20275
    0.56409 -0.00712 0.34395
6  1 0.02337 -0.00592 -0.09924 -0.11949 -0.00763 -0.11824 0.14706 0.06637 0.03786 -0.06302
    0.00000 0.00000 -0.04572
      V16      V17      V18      V19      V20      V21      V22      V23      V24      V25      V26
               V27      V28
1 -0.38223 0.84356 -0.38542 0.58212 -0.32192 0.56971 -0.29674 0.36946 -0.47357 0.56811
    -0.51171 0.41078 -0.46168
2 -0.97515 0.05499 -0.62237 0.33109 -1.00000 -0.13151 -0.45300 -0.18056 -0.35734 -0.20332
    -0.26569 -0.20468 -0.18401
3  0.00299 0.83775 -0.13644 0.75535 -0.08540 0.70887 -0.27502 0.43385 -0.12062 0.57528
    -0.40220 0.58984 -0.22145
4  0.14516 0.54094 -0.39330 -1.00000 -0.54467 -0.69975 1.00000 0.00000 0.00000 1.00000
    0.90695 0.51613 1.00000
5 -0.27457 0.52940 -0.21780 0.45107 -0.17813 0.05982 -0.35575 0.02309 -0.52879 0.03286
    -0.65158 0.13290 -0.53206
```

```
6 -0.15540 -0.00343 -0.10196 -0.11575 -0.05414 0.01838 0.03669 0.01519 0.00888 0.03513
    -0.01535 -0.03240 0.09223
      V29      V30      V31      V32      V33      V34 Class
1  0.21266 -0.34090 0.42267 -0.54487 0.18641 -0.45300 good
2 -0.19040 -0.11593 -0.16626 -0.06288 -0.13738 -0.02447 bad
3  0.43100 -0.17365 0.60436 -0.24180 0.56045 -0.38238 good
4  1.00000 -0.20099 0.25682 1.00000 -0.32382 1.00000 bad
5  0.02431 -0.62197 -0.05707 -0.59573 -0.04608 -0.65697 good
6 -0.07859 0.00732 0.00000 0.00000 -0.00039 0.12011 bad
```

Listing 15.2: Output of the first few rows of the dataset.

15.3 Boosting Algorithms

We can look at two of the most popular boosting machine learning algorithms:

- C5.0.
- Stochastic Gradient Boosting.

Below is an example of the C5.0 and Stochastic Gradient Boosting (using the Gradient Boosting Modeling implementation) algorithms in R. Both algorithms include parameters that could be tuned as previously discussed.

```
# Example of Boosting Algorithms
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
seed <- 7
metric <- "Accuracy"
# C5.0
set.seed(seed)
fit.c50 <- train(Class~., data=dataset, method="C5.0", metric=metric,
    trControl=trainControl)
# Stochastic Gradient Boosting
set.seed(seed)
fit.gbm <- train(Class~., data=dataset, method="gbm", metric=metric,
    trControl=trainControl, verbose=FALSE)
# summarize results
boostingResults <- resamples(list(c5.0=fit.c50, gbm=fit.gbm))
summary(boostingResults)
dotplot(boostingResults)
```

Listing 15.3: Run boosting algorithms on dataset.

We can see that the C5.0 algorithm produced a more accurate model with an accuracy of 94.58%.

```
Models: c5.0, gbm
Number of resamples: 30

Accuracy
       Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
c5.0 0.8824 0.9143 0.9437 0.9458 0.9714  1    0
gbm  0.8824 0.9143 0.9429 0.9402 0.9641  1    0
```

Listing 15.4: Output of boosting algorithms on dataset.

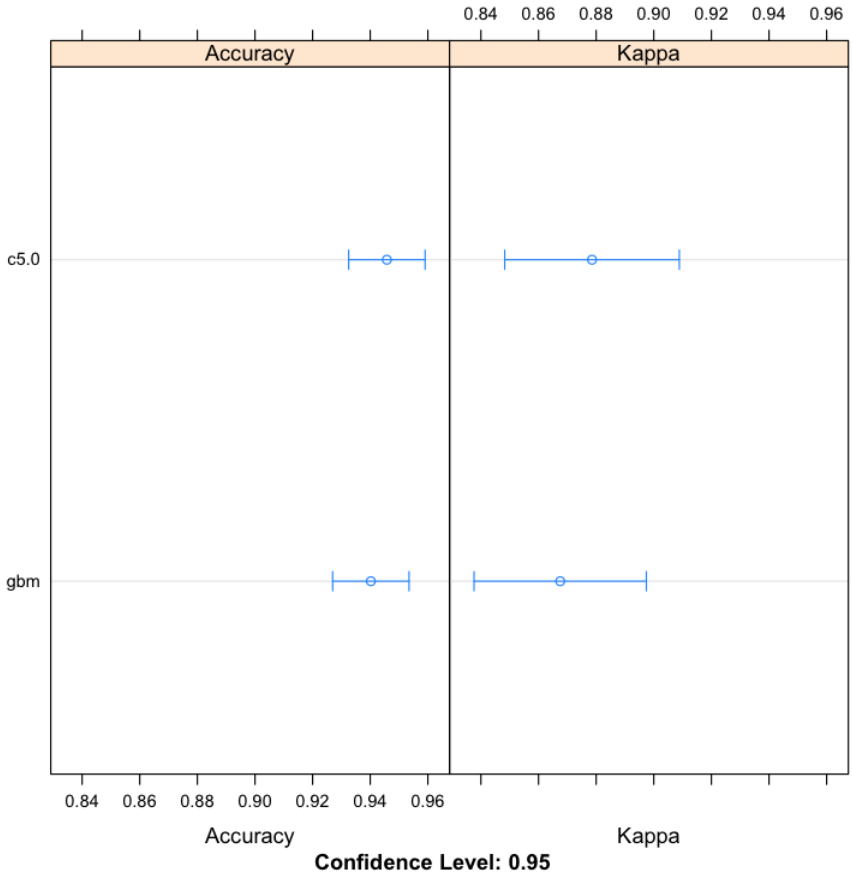

Figure 15.1: Boosting Machine Learning Algorithms in R

Learn more about `caret` package boosting models tree: Boosting Models[1].

15.4 Bagging Algorithms

Let's look at two of the most popular bagging machine learning algorithms:

- Bagged CART.
- Random Forest.

Below is an example of the Bagged CART and Random Forest algorithms in R. Both algorithms include parameters that are not tuned in this example.

```
# Example of Bagging algorithms
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
seed <- 7
metric <- "Accuracy"
# Bagged CART
set.seed(seed)
fit.treebag <- train(Class~., data=dataset, method="treebag", metric=metric,
    trControl=trainControl)
```

[1] http://topepo.github.io/caret/Boosting.html

```
# Random Forest
set.seed(seed)
fit.rf <- train(Class~., data=dataset, method="rf", metric=metric, trControl=trainControl)
# summarize results
baggingResults <- resamples(list(treebag=fit.treebag, rf=fit.rf))
summary(baggingResults)
dotplot(baggingResults)
```

Listing 15.5: Run bagging algorithms on dataset.

We can see that the Random Forest algorithm produced a more accurate model with an accuracy of 93.25%.

```
Models: treebag, rf
Number of resamples: 30

Accuracy
         Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
treebag 0.8529 0.8946 0.9143 0.9183 0.9440  1    0
rf      0.8571 0.9143 0.9420 0.9325 0.9444  1    0
```

Listing 15.6: Output of bagging algorithms on dataset.

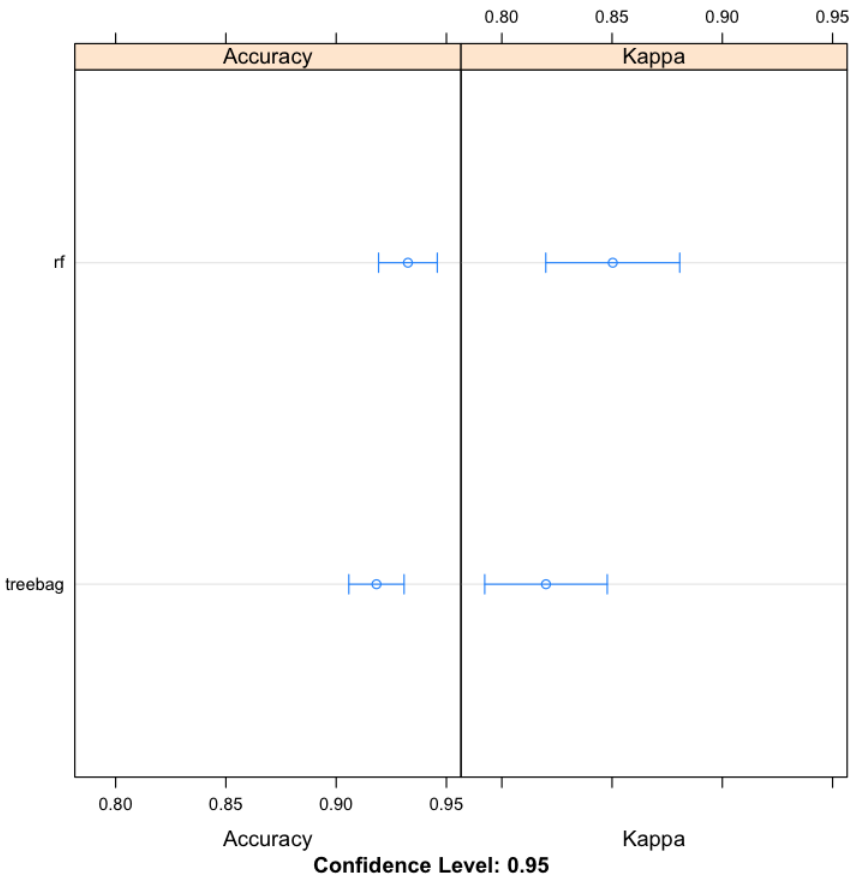

Figure 15.2: Bagging Machine Learning Algorithms in R

Learn more about `caret` package bagging model here: Bagging Models[2].

15.5 Stacking Algorithms

You can combine the predictions of multiple `caret` models using the `caretEnsemble` package. Given a list of `caret` models, the `caretStack()` function can be used to specify a higher-order model to learn how to best combine together the predictions of sub-models. Let's first look at creating five sub-models for the ionosphere dataset, specifically:

- Linear Discriminate Analysis (LDA).
- Classification and Regression Trees (CART).
- Logistic Regression (via Generalized Linear Model or GLM).
- k-Nearest Neighbors (KNN).
- Support Vector Machine with a Radial Basis Kernel Function (SVM).

Below is an example that creates these five sub-models. Note the new helpful `caretList()` function provided by the `caretEnsemble` package for creating a list of standard `caret` models.

```
# Example of Stacking algorithms
# create submodels
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3,
    savePredictions=TRUE, classProbs=TRUE)
algorithmList <- c('lda', 'rpart', 'glm', 'knn', 'svmRadial')
set.seed(seed)
models <- caretList(Class~., data=dataset, trControl=trainControl, methodList=algorithmList)
results <- resamples(models)
summary(results)
dotplot(results)
```

Listing 15.7: Run a suite of algorithms on dataset.

We can see that the SVM creates the most accurate model with an accuracy of 94.66%.

```
Models: lda, rpart, glm, knn, svmRadial
Number of resamples: 30

Accuracy
            Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
lda       0.7714  0.8286 0.8611 0.8645  0.9060 0.9429    0
rpart     0.7714  0.8540 0.8873 0.8803  0.9143 0.9714    0
glm       0.7778  0.8286 0.8873 0.8803  0.9167 0.9722    0
knn       0.7647  0.8056 0.8431 0.8451  0.8857 0.9167    0
svmRadial 0.8824  0.9143 0.9429 0.9466  0.9722 1.0000    0
```

Listing 15.8: Output of a suite of algorithms on the dataset.

[2] http://topepo.github.io/caret/Bagging.html

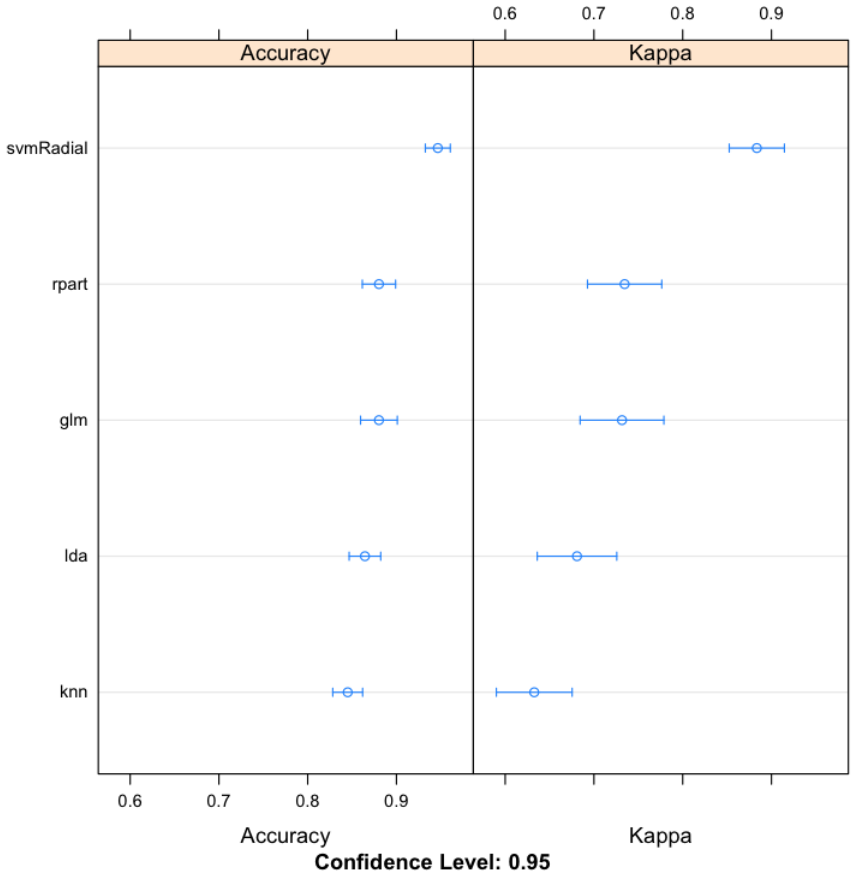

Figure 15.3: Comparison of Sub-Models for Stacking Ensemble in R

When we combine the predictions of different models using stacking, it is desirable that the predictions made by the sub-models have low correlation. This would suggest that the models are skillful but in different ways, allowing a new classifier to figure out how to get the best from each model for an improved score. If the predictions for the sub-models were highly corrected (> 0.75) then they would be making the same or very similar predictions most of the time reducing the benefit of combining the predictions.

```
# correlation between results
modelCor(results)
splom(results)
```

Listing 15.9: Calculate comparison of models on the dataset.

We can see that all pairs of predictions have generally low correlation. The two methods with the highest correlation between their predictions are Logistic Regression (GLM) and KNN at 0.517 correlation which is not considered high (> 0.75).

```
                lda     rpart       glm       knn svmRadial
lda       1.0000000 0.2515454 0.2970731 0.5013524 0.1126050
rpart     0.2515454 1.0000000 0.1749923 0.2823324 0.3465532
glm       0.2970731 0.1749923 1.0000000 0.5172239 0.3788275
knn       0.5013524 0.2823324 0.5172239 1.0000000 0.3512242
svmRadial 0.1126050 0.3465532 0.3788275 0.3512242 1.0000000
```

Listing 15.10: Output of the comparison of models on the dataset.

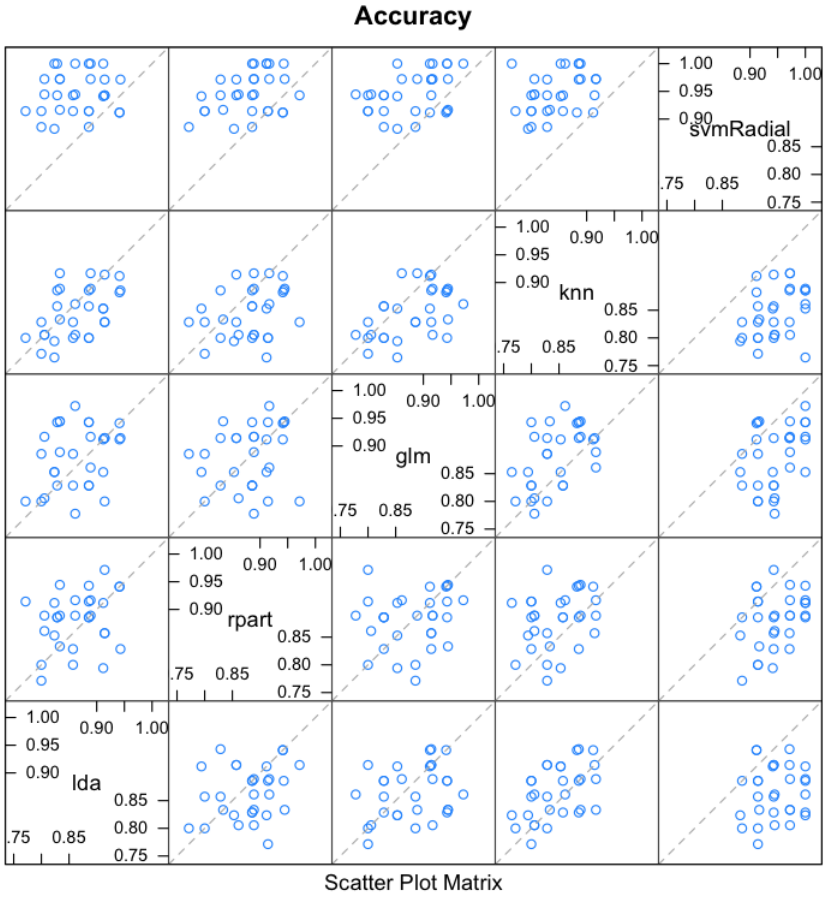

Figure 15.4: Correlations Between Predictions Made By Sub-Models in Stacking Ensemble

Let's combine the predictions of the classifiers using a simple linear model.

```
# stack using glm
stackControl <- trainControl(method="repeatedcv", number=10, repeats=3,
    savePredictions=TRUE, classProbs=TRUE)
set.seed(seed)
stack.glm <- caretStack(models, method="glm", metric="Accuracy", trControl=stackControl)
print(stack.glm)
```

Listing 15.11: Combine predictions using GLM.

We can see that we have lifted the accuracy to 94.99%, which is a small improvement over using SVM alone. This is also an improvement over using random forest alone on the dataset, as observed above.

```
A glm ensemble of 2 base models: lda, rpart, glm, knn, svmRadial

Ensemble results:
Generalized Linear Model
```

```
1053 samples
   5 predictor
   2 classes: 'bad', 'good'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 948, 947, 948, 947, 949, 948, ...
Resampling results

  Accuracy Kappa    Accuracy SD Kappa SD
  0.949996 0.891494 0.02121303  0.04600482
```

Listing 15.12: Output of combining predictions using GLM.

We can also use more sophisticated algorithms to combine predictions in an effort to tease out when best to use the different methods. In this case, we use the Random Forest algorithm to combine the predictions.

```
# stack using random forest
set.seed(seed)
stack.rf <- caretStack(models, method="rf", metric="Accuracy", trControl=stackControl)
print(stack.rf)
```

Listing 15.13: Combine predictions using Random Forest.

We can see that this has lifted the accuracy to 96.26% an impressive improvement over SVM alone.

```
A rf ensemble of 2 base models: lda, rpart, glm, knn, svmRadial

Ensemble results:
Random Forest

1053 samples
   5 predictor
   2 classes: 'bad', 'good'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 948, 947, 948, 947, 949, 948, ...
Resampling results across tuning parameters:

  mtry  Accuracy  Kappa     Accuracy SD Kappa SD
  2     0.9626439 0.9179410 0.01777927  0.03936882
  3     0.9623205 0.9172689 0.01858314  0.04115226
  5     0.9591459 0.9106736 0.01938769  0.04260672

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 2.
```

Listing 15.14: Output of combining predictions using Random Forest.

15.6 Summary

In this lesson you discovered that you can use ensembles of machine learning algorithms to improve the accuracy of your models. You discovered three types of ensembles of machine

learning algorithms that you can build in R:

- Boosting.
- Bagging.
- Stacking.

15.6.1 Next Step

In the previous lesson you discovered how you can increase the accuracy of your models using algorithm tuning, and in this lesson you discovered how you can improve the accuracy of your models by combining the predictions from multiple models. In the next lesson we wrap things up and look at how you can finalize your most accurate model.

Chapter 16

Save And Finalize Your Machine Learning Model

Finding an accurate machine learning model is not the end of the project. In this lesson you will discover how to finalize your machine learning model in R. After completing this lesson, you will know:

1. How to use your trained model to make predictions on unseen data.
2. How to re-create a well performing model from `caret` as a standalone model.
3. How to save your model to a file, load it later and make predictions on unseen data.

Let's get started.

16.1 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:

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

Once you have finalized your model you are ready to make use of it. You could use the R model directly. You could also discover the key internal representation found by the learning algorithm (like the coefficients in a linear model) and use them in a new implementation of the prediction algorithm on another platform.

The `caret` package is an excellent tool that you can use to find good or even best machine learning algorithms and parameters for machine learning algorithms. But what do you do after you have discovered a model that is accurate enough to use? Once you have found a good model in R, you have three main concerns:

- Making new predictions using your tuned `caret` model.
- Creating a standalone model using the entire training dataset.
- Saving a standalone model to file for use later.

In the following, you will look at how you can finalize your machine learning model in R.

16.2 Make Predictions On New Data

You can make new predictions using a model you have tuned using `caret` with the `predict.train()` function. In the recipe below, the dataset is split into a validation dataset and a training dataset. The validation dataset could just as easily be a new dataset stored in a separate file and loaded as a data frame. A good model of the Pima indians dataset is LDA. We can see that `caret` provides access to the best model from a training run in the `finalModel` variable.

We can use that model to make predictions by calling predict using the fit from `train()` which will automatically use the final model. We must specify the data on which to make predictions via the `newdata` argument.

```
# load packages
library(caret)
library(mlbench)
# load dataset
data(PimaIndiansDiabetes)
# create 80%/20% for training and validation datasets
set.seed(9)
validationIndex <- createDataPartition(PimaIndiansDiabetes$diabetes, p=0.80, list=FALSE)
validation <- PimaIndiansDiabetes[-validationIndex,]
training <- PimaIndiansDiabetes[validationIndex,]
# train a model and summarize model
set.seed(9)
trainControl <- trainControl(method="cv", number=10)
fit.lda <- train(diabetes~., data=training, method="lda", metric="Accuracy",
    trControl=trainControl)
print(fit.lda)
print(fit.lda$finalModel)
# estimate skill on validation dataset
set.seed(9)
predictions <- predict(fit.lda, newdata=validation)
confusionMatrix(predictions, validation$diabetes)
```

Listing 16.1: Make predictions on unseen data using caret.

Running this example, we can see that the estimated accuracy on the training dataset was 76.91%. Using the `finalModel` in the fit, we can see that the accuracy on the hold out validation dataset was 77.78%, very similar to our estimate.

```
Resampling results

  Accuracy  Kappa    Accuracy SD Kappa SD
  0.7691169 0.45993 0.06210884  0.1537133

...
```

```
Confusion Matrix and Statistics

          Reference
Prediction neg pos
       neg  85  19
       pos  15  34

               Accuracy : 0.7778
                 95% CI : (0.7036, 0.8409)
    No Information Rate : 0.6536
    P-Value [Acc > NIR] : 0.000586

                  Kappa : 0.5004
 Mcnemar's Test P-Value : 0.606905

            Sensitivity : 0.8500
            Specificity : 0.6415
         Pos Pred Value : 0.8173
         Neg Pred Value : 0.6939
             Prevalence : 0.6536
         Detection Rate : 0.5556
   Detection Prevalence : 0.6797
      Balanced Accuracy : 0.7458

       'Positive' Class : neg
```

Listing 16.2: Output of making predictions on unseen data using caret.

16.3 Create A Standalone Model

In this example, we have tuned a Random Forest with three different values for `mtry` and `ntree` set to 2,000. By printing the fit and the finalModel, we can see that the most accurate value for `mtry` was 2. Now that we know a good algorithm (Random Forest) and the good configuration (`mtry`=2, `ntree`=2000) we can create the final model directly using all of the training data.

We can lookup the `rf` Random Forest implementation used by `caret` in the Caret List of Models[1] and note that it is using the `randomForest` package and in turn the `randomForest()` function. The example creates a new model directly and uses it to make predictions on the new data, this case simulated with the verification dataset.

```
# load packages
library(caret)
library(mlbench)
library(randomForest)
# load dataset
data(Sonar)
set.seed(7)
# create 80%/20% for training and validation datasets
validationIndex <- createDataPartition(Sonar$Class, p=0.80, list=FALSE)
validation <- Sonar[-validationIndex,]
training <- Sonar[validationIndex,]
```

[1] https://topepo.github.io/caret/modelList.html

```
# train a model and summarize model
set.seed(7)
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
fit.rf <- train(Class~., data=training, method="rf", metric="Accuracy",
    trControl=trainControl, ntree=2000)
print(fit.rf)
print(fit.rf$finalModel)
# create standalone model using all training data
set.seed(7)
finalModel <- randomForest(Class~., training, mtry=2, ntree=2000)
# make a predictions on "new data" using the final model
finalPredictions <- predict(finalModel, validation[,1:60])
confusionMatrix(finalPredictions, validation$Class)
```

Listing 16.3: Create a standalone model and make predictions on unseen data.

We can see that the estimated accuracy of the optimal configuration was 85.07%. We can also see that the accuracy of the final standalone model trained on all of the training dataset and predicting for the validation dataset was 82.93%, very close to our estimate.

```
Random Forest

167 samples
 60 predictor
  2 classes: 'M', 'R'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 151, 150, 150, 150, 151, 150, ...
Resampling results across tuning parameters:

  mtry  Accuracy  Kappa     Accuracy SD Kappa SD
   2    0.8507353 0.6968343 0.07745360  0.1579125
  31    0.8064951 0.6085348 0.09373438  0.1904946
  60    0.7927696 0.5813335 0.08768147  0.1780100

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 2.

...

Call:
 randomForest(x = x, y = y, ntree = 2000, mtry = param$mtry)
               Type of random forest: classification
                     Number of trees: 2000
No. of variables tried at each split: 2

        OOB estimate of error rate: 14.37%
Confusion matrix:
   M  R class.error
M 83  6  0.06741573
R 18 60 0.23076923

...

Confusion Matrix and Statistics
```

```
          Reference
Prediction M  R
         M 20  5
         R  2 14

               Accuracy : 0.8293
                 95% CI : (0.6794, 0.9285)
    No Information Rate : 0.5366
    P-Value [Acc > NIR] : 8.511e-05

                  Kappa : 0.653
 Mcnemar's Test P-Value : 0.4497

            Sensitivity : 0.9091
            Specificity : 0.7368
         Pos Pred Value : 0.8000
         Neg Pred Value : 0.8750
             Prevalence : 0.5366
         Detection Rate : 0.4878
   Detection Prevalence : 0.6098
      Balanced Accuracy : 0.8230

       'Positive' Class : M
```

Listing 16.4: Output of predictions made by standalone model.

Some simpler models, like linear models can output their coefficients. This is useful, because from these, you can implement the simple prediction procedure in your programming language of choice and use the coefficients to get the same accuracy. This gets more difficult as the complexity of the representation used by the algorithm increases.

16.4 Save and Load Your Model

You can save your best models to a file so that you can load them up later and make predictions. In this example we split the Sonar dataset into a training dataset and a validation dataset. We take our validation dataset as new data to test our final model. We train the final model using the training dataset and our optimal parameters, then save it to a file called `finalModel.rds` in the local working directory.

The model is serialized. It can be loaded at a later time by calling the `readRDS()` function and assigning the object that is loaded (in this case a Random Forest fit) to a variable name. The loaded Random Forest is then used to make predictions on new data, in this case the validation dataset.

```
# load packages
library(caret)
library(mlbench)
library(randomForest)
# load dataset
data(Sonar)
set.seed(7)
# create 80%/20% for training and validation datasets
validationIndex <- createDataPartition(Sonar$Class, p=0.80, list=FALSE)
```

```
validation <- Sonar[-validationIndex,]
training <- Sonar[validationIndex,]
# create final standalone model using all training data
set.seed(7)
finalModel <- randomForest(Class~., training, mtry=2, ntree=2000)
# save the model to disk
saveRDS(finalModel, "./finalModel.rds")

# later...

# load the model
superModel <- readRDS("./finalModel.rds")
print(superModel)
# make a predictions on "new data" using the final model
finalPredictions <- predict(superModel, validation[,1:60])
confusionMatrix(finalPredictions, validation$Class)
```

Listing 16.5: Save and load a standalone model and make predictions on unseen data.

We can see that the accuracy on the validation dataset was 82.93%.

```
Confusion Matrix and Statistics

          Reference
Prediction M  R
         M 20  5
         R  2 14

               Accuracy : 0.8293
                 95% CI : (0.6794, 0.9285)
    No Information Rate : 0.5366
    P-Value [Acc > NIR] : 8.511e-05

                  Kappa : 0.653
 Mcnemar's Test P-Value : 0.4497

            Sensitivity : 0.9091
            Specificity : 0.7368
         Pos Pred Value : 0.8000
         Neg Pred Value : 0.8750
             Prevalence : 0.5366
         Detection Rate : 0.4878
   Detection Prevalence : 0.6098
      Balanced Accuracy : 0.8230

       'Positive' Class : M
```

Listing 16.6: Output of saving and loading a standalone model.

16.5 Summary

In this lesson you discovered three recipes for working with final predictive models:

1. How to make predictions using the best model from `caret` tuning.

2. How to create a standalone model using the parameters found during `caret` tuning.
3. How to save and later load a standalone model and use it to make predictions.

16.5.1 Next Step

This concludes your lessons on the specific predictive modeling machine learning tasks in R. You now have all of the pieces you need to be able to work through the tasks of a machine learning project. Next in Part III of this book you will discover how you can tie together all of these lessons into complete end-to-end machine learning projects.

Part III

Projects

Chapter 17

Predictive Modeling Project Template

Applied machine learning is an empirical skill. You cannot get better at it by reading books and articles. You have to practice. In this lesson you will discover the simple six-step machine learning project template that you can use to jump-start your project in R. After completing this lesson you will know:

1. How to structure an end-to-end predictive modeling project.
2. How to map the tasks you learned about in Part II onto a project.
3. How to best use the structured project template to ensure an accurate result for your dataset.

Let's get started.

17.1 Practice Machine Learning With Projects

Working through machine learning problems from end-to-end is critically important. You can read about machine learning. You can also try out small one-off recipes. But applied machine learning will not come alive for you until you work through a dataset from beginning to end.

Working through a project forces you to think about how the model will be used, to challenge your assumptions and to get good at all parts of a project, not just your favorite. The best way to practice predictive modeling machine learning projects is to use standardized datasets from the UCI Machine Learning Repository. Once you have a practice dataset and a bunch of R recipes, how do you put it all together and work through the problem end-to-end?

17.1.1 Use A Structured Step-By-Step Process

Any predictive modeling machine learning project can be broken down into about 6 common tasks:

1. Define Problem.
2. Summarize Data.
3. Prepare Data.

4. Evaluate Algorithms.
5. Improve Results.
6. Present Results.

Tasks can be combined or broken down further, but this is the general structure. To work through predictive modeling machine learning problems in R, you need to map R onto this process. The tasks may need to be adapted or renamed slightly to suit the R way of doing things (e.g. the `caret` package). The next section provides exactly this mapping and elaborates each task and the types of sub tasks and `caret` packages that you can use.

17.2 Machine Learning Project Template in R

This section presents a project template that you can use to work through machine learning problems in R end-to-end.

17.2.1 Template Summary

Below is the project template that you can use in your machine learning projects in R.

```
# R Project Template

# 1. Prepare Problem
# a) Load packages
# b) Load dataset
# c) Split-out validation dataset

# 2. Summarize Data
# a) Descriptive statistics
# b) Data visualizations

# 3. Prepare Data
# a) Data Cleaning
# b) Feature Selection
# c) Data Transforms

# 4. Evaluate Algorithms
# a) Test options and evaluation metric
# b) Spot-Check Algorithms
# c) Compare Algorithms

# 5. Improve Accuracy
# a) Algorithm Tuning
# b) Ensembles

# 6. Finalize Model
# a) Predictions on validation dataset
# b) Create standalone model on entire training dataset
# c) Save model for later use
```

Listing 17.1: Predictive modeling machine learning project template.

17.2.2 How To Use The Project Template

1. Create a new file for your project (e.g. `project_name.R`).
2. Copy the project template.
3. Paste it into your empty project file.
4. Start to fill it in, using recipes from this book and others.

17.3 Machine Learning Project Template Steps

This section gives you additional details on each of the steps of the template.

17.3.1 Prepare Problem

This step is about loading everything you need to start working on your problem. This includes:

- R packages you will use like `caret`.
- Loading your dataset from CSV.
- Using `caret` to create a separate training and validation datasets.

This is also the home of any global configuration you might need to do, like setting up any parallel packages and functions for using multiple cores. It is also the place where you might need to make a reduced sample of your dataset if it is too large to work with. Ideally, your dataset should be small enough to build a model or create a visualization within a minute, ideally 30 seconds. You can always scale up well performing models later.

17.3.2 Summarize Data

This step is about better understanding the data that you have available. This includes understanding your data using:

- Descriptive statistics such as summaries.
- Data visualizations such as plots from the `graphics` and `lattice` packages.

Take your time and use the results to prompt a lot of questions, assumptions and hypotheses that you can investigate later with specialized models.

17.3.3 Prepare Data

This step is about preparing the data in such a way that it best exposes the structure of the problem and the relationships between your input attributes with the output variable. This includes tasks such as:

- Cleaning data by removing duplicates, marking missing values and even imputing missing values.

- Feature selection where redundant features may be removed.
- Data transforms where attributes are scaled or redistributed in order to best expose the structure of the problem later to learning algorithms.

Start simple. Revisit this step often and cycle with the next step until you converge on a subset of algorithms and a presentation of the data that results in accurate or accurate-enough models to proceed.

17.3.4 Evaluate Algorithms

This step is about finding a subset of machine learning algorithms that are good at exploiting the structure of your data (e.g. have better than average skill). This involves steps such as:

- Defining test options using `caret` such as cross validation and the evaluation metric to use.
- Spot-checking a suite of linear and nonlinear machine learning algorithms.
- Comparing the estimated accuracy of algorithms.

On a given problem you will likely spend most of your time on this and the previous step until you converge on a set of 3-to-5 well performing machine learning algorithms.

17.3.5 Improve Accuracy

Once you have a shortlist of machine learning algorithms, you need to get the most out of them. There are two different ways to improve the accuracy of your models:

- Search for a combination of parameters for each algorithm using `caret` that yields the best results.
- Combine the prediction of multiple models into an ensemble prediction using standalone algorithms or the `caretEnsemble` package.

The line between this and the previous step can blur when a project becomes concrete. There may be a little algorithm tuning in the previous step. And in the case of ensembles, you may bring more than a shortlist of algorithms forward to combine their predictions.

17.3.6 Finalize Model

Once you have found a model that you believe can make accurate predictions on unseen data, you are ready to finalize it. Finalizing a model may involve sub-tasks such as:

- Using an optimal model tuned by `caret` to make predictions on unseen data.
- Creating a standalone model using the parameters tuned by `caret`.
- Saving an optimal model to file for later use.

Once you make it this far you are ready to present results to stakeholders and/or deploy your model to start making predictions on unseen data.

17.4 Tips For Using The Template Well

Below are tips that you can use to make the most of the machine learning project template in R.

- **Fast First Pass**. Make a first-pass through the project steps as fast as possible. This will give you confidence that you have all the parts that you need and a baseline from which to improve.
- **Cycles**. The process in not linear but cyclic. You will loop between steps, and probably spend most of your time in tight loops between steps 3-4 or 3-4-5 until you achieve a level of accuracy that is sufficient or you run out of time.
- **Attempt Every Step**. It is easy to skip steps, especially if you are not confident or familiar with the tasks of that step. Try and do something at each step in the process, even if it does not improve accuracy. You can always build upon it later. Don't skip steps, just reduce their contribution.
- **Ratchet Accuracy**. The goal of the project is model accuracy. Every step contributes towards this goal. Treat changes that you make as experiments that increase accuracy as the golden path in the process and reorganize other steps around them. Accuracy is a ratchet that can only move in one direction (better, not worse).
- **Adapt As Needed**. Modify the steps as you need on a project, especially as you become more experienced with the template. Blur the edges of tasks, such as 4-5 to best serve model accuracy.

17.5 Summary

In this lesson you discovered a machine learning project template in R. It laid out the steps of a predictive modeling machine learning project with the goal of maximizing model accuracy. You can copy-and-paste the template and use it to jump-start your current or next machine learning project in R.

17.5.1 Next Step

Now that you know how to structure a predictive modeling machine learning project in R, you need to put this knowledge to use. In the next lesson you will work through a simple case study problem end-to-end. This is a famous case study and the `hello world` of machine learning projects.

Chapter 18

Your First Machine Learning Project in R Step-By-Step

You need to see how all of the pieces of a predictive modeling machine learning project actually fit together. In this lesson you will complete your first machine learning project using R. In this step-by-step tutorial project you will:

1. Load a dataset and understand its structure using statistical summaries and data visualization.
2. Create five machine learning models, pick the best and build confidence that the accuracy is reliable.
3. If you are a machine learning beginner and looking to finally get started using R, this tutorial was designed for you.

Let's get started!

18.1 Hello World of Machine Learning

The best small project to start with on a new tool is the classification of iris flowers (e.g. the iris dataset described in Chapter 5). This is a good dataset for your first project because it is so well understood.

- Attributes are numeric so you have to figure out how to load and handle data.
- It is a classification problem, allowing you to practice with perhaps an easier type of supervised learning algorithm.
- It is a multiclass classification problem (multi-nominal) that may require some specialized handling.
- It only has 4 attributes and 150 rows, meaning it is small and easily fits into memory (and a screen or single sheet of paper).
- All of the numeric attributes are in the same units and the same scale not requiring any special scaling or transforms to get started.

In this tutorial we are going to work through a small machine learning project end-to-end. Here is an overview of what we are going to cover:

1. Loading the dataset.
2. Summarizing the dataset.
3. Visualizing the dataset.
4. Evaluating some algorithms.
5. Making some predictions.

Take your time and work through each step. Try to type in the commands yourself or copy-and-paste the commands to speed things up. Start your R interactive environment and let's get started with your hello world machine learning project in R.

18.2 Load The Data

Here is what we are going to do in this step:

1. Load the iris data the easy way.
2. Load the iris data from CSV (optional, for purists).
3. Separate the data into a training dataset and a validation dataset.

18.2.1 Load Data The Easy Way

You can load the iris dataset from the `stats` package as follows:

```
# load the caret package
library(caret)
# attach the iris dataset to the environment
data(iris)
# rename the dataset
dataset <- iris
```

Listing 18.1: Load data from package.

You now have the iris data loaded in R and accessible via the `dataset` variable. I like to name the loaded dataset. This is helpful if you want to copy-paste code between projects and the dataset always has the same name.

18.2.2 Load From CSV (optional alternative)

Maybe you are a purist and you want to load the data just like you would on your own machine learning project, from a CSV file.

1. Download the iris dataset from the UCI Machine Learning Repository[1].

[1] https://archive.ics.uci.edu/ml/datasets/Iris

2. Save the file as `iris.csv` your project directory.

Load the dataset from the CSV file as follows:

```
# define the filename
filename <- "iris.csv"
# load the CSV file from the local directory
dataset <- read.csv(filename, header=FALSE)
# set the column names in the dataset
colnames(dataset) <- c("Sepal.Length","Sepal.Width","Petal.Length","Petal.Width","Species")
```

Listing 18.2: Load data from CSV.

You now have the iris data loaded in R and accessible via the dataset variable.

18.2.3 Create a Validation Dataset

We need to know whether the model that we created is any good. Later, we will use statistical methods to estimate the accuracy of the models that we create on unseen data. We also want a more concrete estimate of the accuracy of the best model on unseen data by evaluating it on actual unseen data. That is, we are going to hold back some data that the algorithms will not get to see and we will use this data to get a second and independent idea of how accurate the best model might actually be. We will split the loaded dataset into two, 80% of which we will use to train our models and 20% that we will hold back as a validation dataset.

```
# create a list of 80% of the rows in the original dataset we can use for training
validationIndex <- createDataPartition(dataset$Species, p=0.80, list=FALSE)
# select 20% of the data for validation
validation <- dataset[-validationIndex,]
# use the remaining 80% of data to training and testing the models
dataset <- dataset[validationIndex,]
```

Listing 18.3: Split data into a training dataset and a validation dataset.

You now have training data in the `dataset` variable and a validation set we will use later in the `validation` variable. Note that we replaced our dataset variable with the 80% sample of the dataset. This was an attempt to keep the rest of the code simpler and readable.

18.3 Summarize Dataset

Now it is time to take a look at the data. In this step we are going to take a look at the data a few different ways:

1. Dimensions of the dataset.
2. Types of the attributes.
3. Peek at the data itself.
4. Levels of the class attribute.
5. Breakdown of the instances in each class.
6. Statistical summary of all attributes.

18.3.1 Dimensions of Dataset

We can get a quick idea of how many instances (rows) and how many attributes (columns) the data contains with the `dim` function.

```
# dimensions of dataset
dim(dataset)
```

Listing 18.4: Calculate the dimensions of the dataset.

We can see that the data has 120 instances and 5 attributes:

```
[1] 120 5
```

Listing 18.5: Output of the dimensions of the dataset.

18.3.2 Types of Attributes

It is a good idea to review the types of the attributes. They could be doubles, integers, strings, factors and other types. Knowing the types is important as it will give you an idea of how to better summarize the data you have and the types of transforms you might need to use to prepare the data before you model it.

```
# list types for each attribute
sapply(dataset, class)
```

Listing 18.6: Calculate the attribute data types of the dataset.

We can see that all of the inputs are double and that the class value is a factor:

```
Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 "numeric" "numeric" "numeric" "numeric" "factor"
```

Listing 18.7: Output of data types of the dataset.

18.3.3 Peek at the Data

It is also always a good idea to actually eyeball your data.

```
# take a peek at the first 5 rows of the data
head(dataset)
```

Listing 18.8: Display the first few rows of the dataset.

We can see the first 5 rows of the data:

```
 Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1 5.1 3.5 1.4 0.2 setosa
2 4.9 3.0 1.4 0.2 setosa
3 4.7 3.2 1.3 0.2 setosa
5 5.0 3.6 1.4 0.2 setosa
6 5.4 3.9 1.7 0.4 setosa
7 4.6 3.4 1.4 0.3 setosa
```

Listing 18.9: Output of the first few rows of the dataset.

18.3.4 Levels of the Class

The class variable is a factor. A factor has multiple class labels called levels. Let's look at the levels:

```
# list the levels for the class
levels(dataset$Species)
```

Listing 18.10: Calculate the levels of the class attribute.

Notice above how we can refer to an attribute by name as a property of the dataset. In the results we can see that the class has 3 different labels:

```
[1] "setosa" "versicolor" "virginica"
```

Listing 18.11: Output of the levels of the class attribute.

This is a multiclass or a multinomial classification problem. If there were two levels, it would be a binary classification problem.

18.3.5 Class Distribution

Let's now take a look at the number of instances (rows) that belong to each class. We can view this as an absolute count and as a percentage.

```
# summarize the class distribution
percentage <- prop.table(table(dataset$Species)) * 100
cbind(freq=table(dataset$Species), percentage=percentage)
```

Listing 18.12: Calculate the breakdown of the class attribute.

We can see that each class has the same number of instances (40 or 33% of the dataset).

```
 freq percentage
setosa     40 33.33333
versicolor 40 33.33333
virginica 40 33.33333
```

Listing 18.13: Output of the breakdown of the class attribute.

18.3.6 Statistical Summary

Now finally, we can take a look at a summary of each attribute. This includes the mean, the min and max values as well as some percentiles.

```
# summarize attribute distributions
summary(dataset)
```

Listing 18.14: Summarize all attributes in the dataset.

We can see that all of the numerical values have the same scale (centimeters) and similar ranges `[0,8]` centimeters.

```
 Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 Min.   :4.300 Min.  :2.00 Min.  :1.000 Min.  :0.100 setosa  :40
 1st Qu.:5.100 1st Qu.:2.80 1st Qu.:1.575 1st Qu.:0.300 versicolor:40
 Median :5.800 Median :3.00 Median :4.300 Median :1.350 virginica :40
```

```
Mean   :5.834 Mean   :3.07 Mean   :3.748 Mean   :1.213
3rd Qu.:6.400 3rd Qu.:3.40 3rd Qu.:5.100 3rd Qu.:1.800
Max.   :7.900 Max.   :4.40 Max.   :6.900 Max.   :2.500
```

Listing 18.15: Output of summary of all attributes in the dataset.

18.4 Visualize Dataset

We now have a basic idea about the data. We need to extend that with some visualizations. We are going to look at two types of plots:

1. Univariate plots to better understand each attribute.
2. Multivariate plots to better understand the relationships between attributes.

18.4.1 Univariate Plots

We start with some univariate plots, that is, plots of each individual variable. It is helpful with visualization to have a way to refer to just the input attributes and just the output attributes. Let's set that up and call the input attributes `x` and the output attribute (or class) `y`.

```
# split input and output
x <- dataset[,1:4]
y <- dataset[,5]
```

Listing 18.16: Split dataset into input and output attributes.

Given that the input variables are numeric, we can create box and whisker plots of each.

```
# boxplot for each attribute on one image
par(mfrow=c(1,4))
for(i in 1:4) {
  boxplot(x[,i], main=names(dataset)[i])
}
```

Listing 18.17: Calculate box and whisker plots of data.

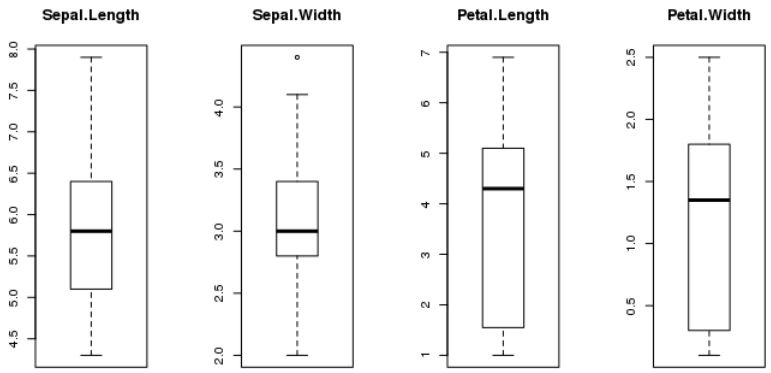

Figure 18.1: Box and Whisker Plots pf Iris flowers data.

We can also create a bar plot of the `Species` class variable to get a graphical representation of the class distribution (generally uninteresting in this case because they're even).

```
# barplot for class breakdown
plot(y)
```

Listing 18.18: Calculate bar plots of each level of the class attribute.

This confirms what we learned in the last section, that the instances are evenly distributed across the three classes:

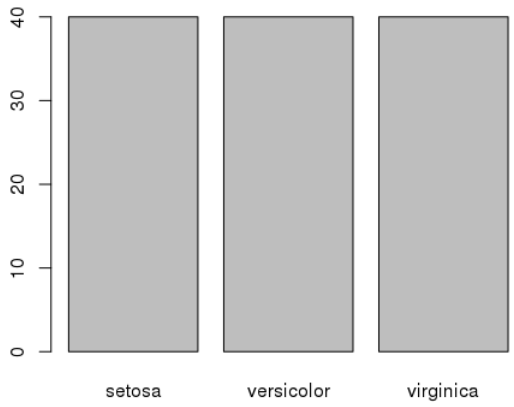

Figure 18.2: Bar Plots pf Iris flowers data.

18.4.2 Multivariate Plots

Now we can look at the interactions between the variables. First let's look at scatter plots of all pairs of attributes and color the points by class. In addition, because the scatter plots show that points for each class are generally separate, we can draw ellipses around them.

```
# scatter plot matrix
featurePlot(x=x, y=y, plot="ellipse")
```

Listing 18.19: Calculate a scatter plot matrix plot by class.

We can see some clear relationships between the input attributes (trends) and between attributes and the class values (ellipses):

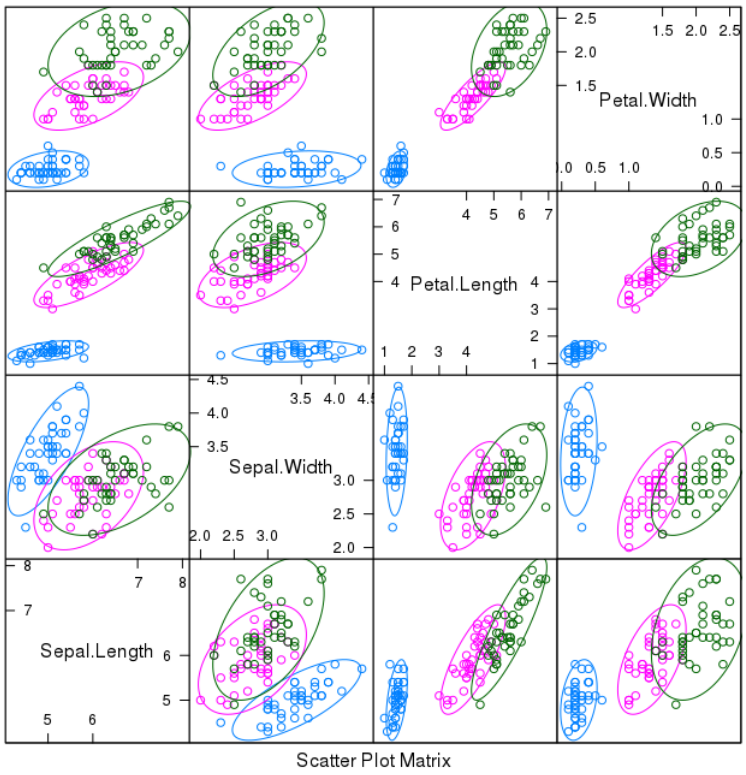

Figure 18.3: Scatterplot Matrix Iris flowers data by Class.

We can also look at box and whisker plots of each input variable again, but this time broken down into separate plots for each class. This can help to tease out obvious linear separations between the classes.

```
# box and whisker plots for each attribute
featurePlot(x=x, y=y, plot="box")
```

Listing 18.20: Calculate box and whisker plots by class.

This is useful as it shows that there are clearly different distributions of the attributes for each class value.

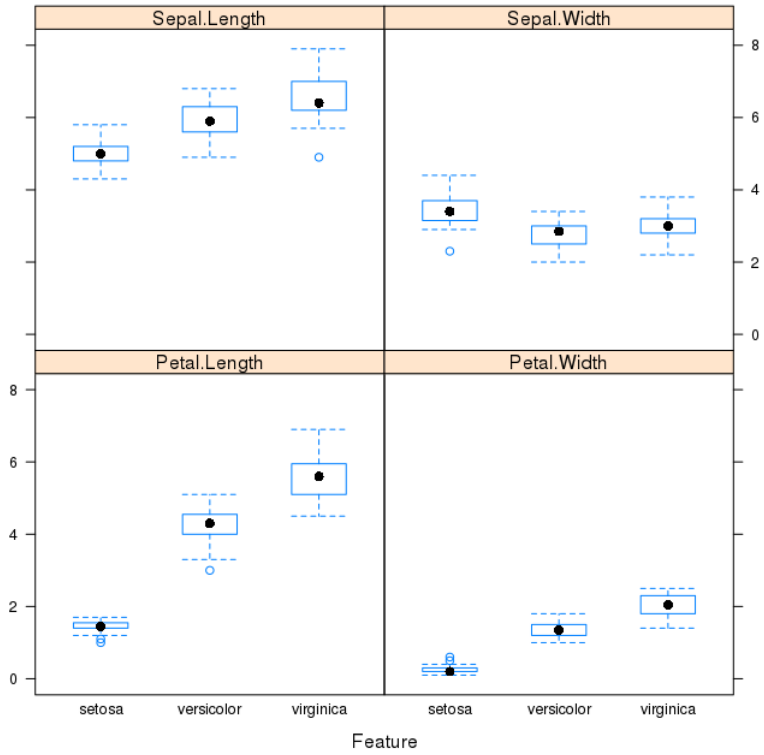

Figure 18.4: Box and Whisker Plots Iris flowers data by Class.

Next we can get an idea of the distribution of each attribute, again like the box and whisker plots, broken down by class value. Sometimes histograms are good for this, but in this case we will use some probability density plots to give nice smooth lines for each distribution.

```
# density plots for each attribute by class value
scales <- list(x=list(relation="free"), y=list(relation="free"))
featurePlot(x=x, y=y, plot="density", scales=scales)
```

Listing 18.21: Calculate density plots by class.

Like the boxplots, we can see the difference in distribution of each attribute by class value. We can also see the Gaussian-like distribution (bell curve) of each attribute.

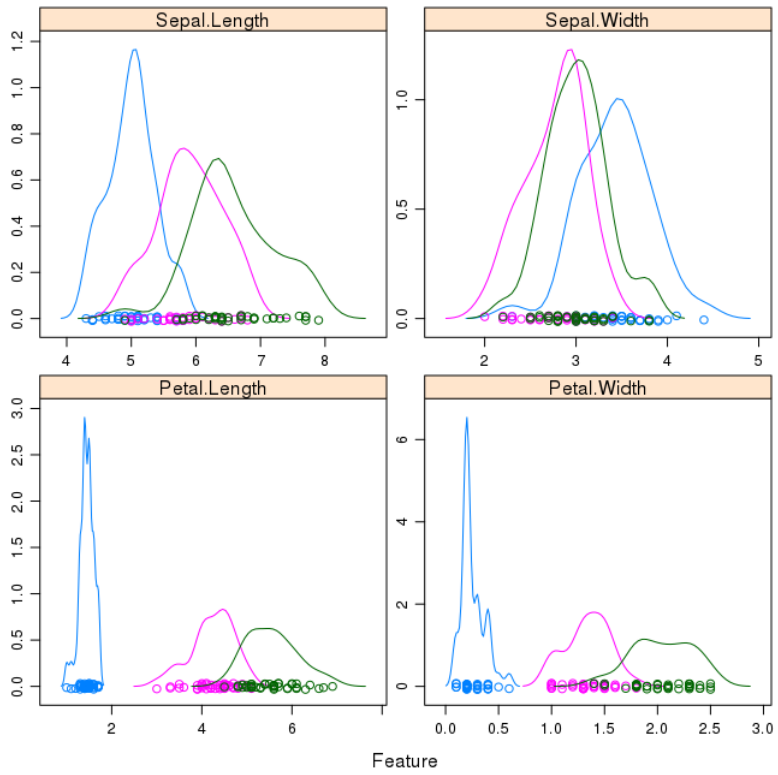

Figure 18.5: Density Plots of the Iris flowers data by Class.

18.5 Evaluate Some Algorithms

Now it is time to create some models of the data and estimate their accuracy on unseen data. Here is what we are going to cover in this step:

1. Setup the test harness to use 10-fold cross validation.
2. Build 5 different models to predict species from flower measurements
3. Select the best model.

18.5.1 Test Harness

We will use 10-fold cross validation to estimate accuracy. This will split our dataset into 10 parts, train in 9 and test on 1 and repeat for all combinations of train-test splits.

```
# Run algorithms using 10-fold cross validation
trainControl <- trainControl(method="cv", number=10)
metric <- "Accuracy"
```

Listing 18.22: Prepare the test harness for evaluating algorithms.

We are using the metric of `Accuracy` to evaluate models. This is a ratio of the number of correctly predicted instances divided by the total number of instances in the dataset multiplied by 100 to give a percentage (e.g. 95% accurate). We will be using the `metric` variable when we build and evaluate each models in the next section.

18.5.2 Build Models

We don't know which algorithms would be good on this problem or what configurations to use. We do get an idea from the plots that some of the classes are partially linearly separable in some dimensions, so we are expecting generally good results. Let's evaluate 5 different algorithms:

- Linear Discriminant Analysis (LDA).
- Classification and Regression Trees (CART).
- k-Nearest Neighbors (KNN).
- Support Vector Machines (SVM) with a radial kernel.
- Random Forest (RF).

This is a good mixture of simple linear (LDA), nonlinear (CART, KNN) and complex nonlinear methods (SVM, RF). We reset the random number seed before each run to ensure that the evaluation of each algorithm is performed using exactly the same data splits. It ensures the results are directly comparable. Let's build our five models:

```
# LDA
set.seed(7)
fit.lda <- train(Species~., data=dataset, method="lda", metric=metric,
    trControl=trainControl)
# CART
set.seed(7)
fit.cart <- train(Species~., data=dataset, method="rpart", metric=metric,
    trControl=trainControl)
# KNN
set.seed(7)
fit.knn <- train(Species~., data=dataset, method="knn", metric=metric,
    trControl=trainControl)
# SVM
set.seed(7)
fit.svm <- train(Species~., data=dataset, method="svmRadial", metric=metric,
    trControl=trainControl)
# Random Forest
set.seed(7)
fit.rf <- train(Species~., data=dataset, method="rf", metric=metric, trControl=trainControl)
```

Listing 18.23: Estimate the accuracy of a number of algorithms on the dataset.

The `caret` package does support the configuration and tuning of the configuration of each model, but we are not going to cover that in this tutorial.

18.5.3 Select Best Model

We now have 5 models and accuracy estimations for each. We need to compare the models to each other and select the most accurate. We can report on the accuracy of each model by first creating a list of the models, gathering resample statistics and using the summary function on the result.

```
# summarize accuracy of models
results <- resamples(list(lda=fit.lda, cart=fit.cart, knn=fit.knn, svm=fit.svm, rf=fit.rf))
summary(results)
```

Listing 18.24: Calculate resample statistics from the models.

In the table of results, we can see the distribution of both the Accuracy and Kappa of the models. Let's just focus on Accuracy for now.

```
Models: lda, cart, knn, svm, rf
Number of resamples: 10

Accuracy
       Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
lda  0.9167  0.9375 1.0000 0.9750     1    1    0
cart 0.8333  0.9167 0.9167 0.9417     1    1    0
knn  0.8333  0.9167 1.0000 0.9583     1    1    0
svm  0.8333  0.9167 0.9167 0.9417     1    1    0
rf   0.8333  0.9167 0.9583 0.9500     1    1    0

Kappa
      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
lda  0.875  0.9062 1.0000 0.9625     1    1    0
cart 0.750  0.8750 0.8750 0.9125     1    1    0
knn  0.750  0.8750 1.0000 0.9375     1    1    0
svm  0.750  0.8750 0.8750 0.9125     1    1    0
rf   0.750  0.8750 0.9375 0.9250     1    1    0
```

Listing 18.25: Output of resample statistics for the models.

We can also create a plot of the model evaluation results and compare the spread and the mean accuracy of each model. There is a population of accuracy measures for each algorithm because each algorithm was evaluated 10 times (10 fold cross validation).

```
# compare accuracy of models
dotplot(results)
```

Listing 18.26: Calculate dot plots of estimated model accuracy.

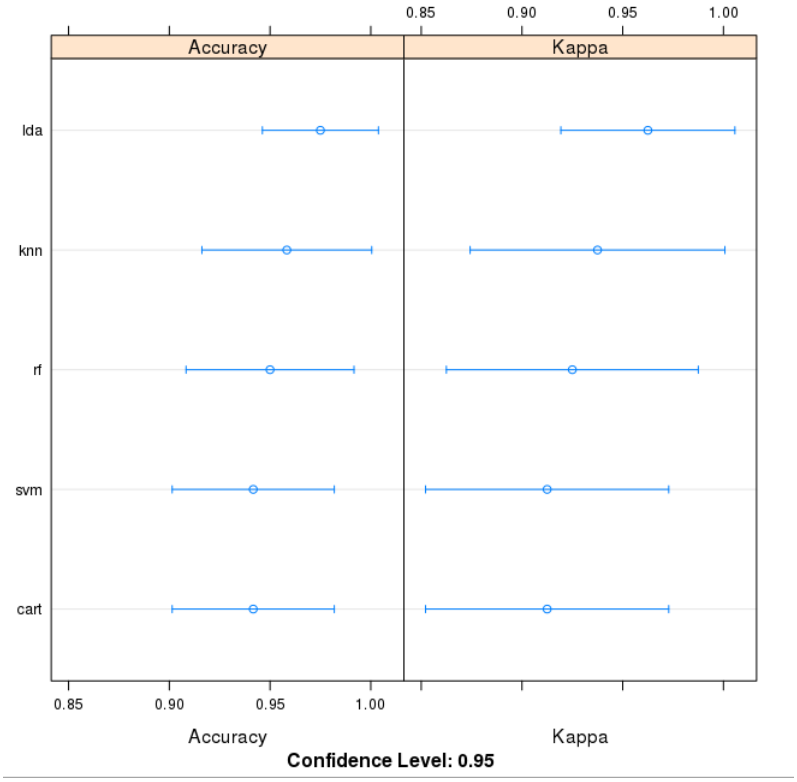

Figure 18.6: Comparison of Machine Learning Algorithms on Iris Dataset in R.

The results for just the LDA model can be summarized.

```
# summarize Best Model
print(fit.lda)
```

Listing 18.27: Display the estimated accuracy of a model.

This gives a nice summary of what was used to train the model and the mean and standard deviation (SD) accuracy achieved, specifically 97.5% accuracy +/- 4%.

```
Linear Discriminant Analysis

120 samples
  4 predictor
  3 classes: 'setosa', 'versicolor', 'virginica'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 108, 108, 108, 108, 108, 108, ...
Resampling results

  Accuracy Kappa  Accuracy SD Kappa SD
  0.975    0.9625 0.04025382  0.06038074
```

Listing 18.28: Output estimated accuracy of a model.

18.6 Make Predictions

The LDA algorithm created the most accurate model. Now we want to get an idea of the accuracy of the model on our validation set. This will give us an independent final check on the accuracy of the best model. It is valuable to keep a validation set just in case you made a slip during training, such as overfitting to the training set or a data leak. Both will result in an overly optimistic result. We can run the LDA model directly on the validation set and summarize the results in a confusion matrix.

```
# estimate skill of LDA on the validation dataset
predictions <- predict(fit.lda, validation)
confusionMatrix(predictions, validation$Species)
```

Listing 18.29: Calculate predictions on unseen data.

We can see that the accuracy is 100%. It was a small validation dataset, but this result is within our expected margin of 97% +/-4% suggesting we may have an accurate and a reliably accurate model.

```
Confusion Matrix and Statistics

           Reference
Prediction  setosa versicolor virginica
  setosa        10          0         0
  versicolor     0         10         0
  virginica      0          0        10

Overall Statistics

               Accuracy : 1
                 95% CI : (0.8843, 1)
    No Information Rate : 0.3333
    P-Value [Acc > NIR] : 4.857e-15

                  Kappa : 1
 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: setosa Class: versicolor Class: virginica
Sensitivity                 1.0000           1.0000          1.0000
Specificity                 1.0000           1.0000          1.0000
Pos Pred Value              1.0000           1.0000          1.0000
Neg Pred Value              1.0000           1.0000          1.0000
Prevalence                  0.3333           0.3333          0.3333
Detection Rate              0.3333           0.3333          0.3333
Detection Prevalence        0.3333           0.3333          0.3333
Balanced Accuracy           1.0000           1.0000          1.0000
```

Listing 18.30: Output of predictions on unseen data.

18.7 Summary

In this lesson you discovered step-by-step how to complete your first machine learning project in R. You discovered that completing a small end-to-end project from loading the data to making predictions is the best way to get familiar with a new platform.

18.7.1 Next Step

You have applied the lessons from Part II on a simple problem and completed your first machine learning project. Next you take things one step further and work through a regression predictive modeling problem. It will be a slightly more complex project and involve data transforms, algorithm tuning and use of ensemble methods to improve results.

Chapter 19

Regression Machine Learning Case Study Project

How do you work through a predictive modeling machine learning problem end-to-end? In this lesson you will work through a case study regression predictive modeling problem in R including each step of the applied machine learning process. After completing this project, you will know:

1. How to work through a regression predictive modeling problem end-to-end.
2. How to use data transforms to improve model performance.
3. How to use algorithm tuning to improve model performance.
4. How to use ensemble methods and tuning of ensemble methods to improve model performance.

Let's get started.

19.1 Problem Definition

For this project we will investigate the Boston House Price dataset described in Chapter 5. Each record in the database describes a Boston suburb or town. The data was drawn from the Boston Standard Metropolitan Statistical Area (SMSA) in 1970. The attributes are defined as follows (taken from the UCI Machine Learning Repository):

1. CRIM: per capita crime rate by town
2. ZN: proportion of residential land zoned for lots over 25,000 sq.ft.
3. INDUS: proportion of non-retail business acres per town
4. CHAS: Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
5. NOX: nitric oxides concentration (parts per 10 million)
6. RM: average number of rooms per dwelling
7. AGE: proportion of owner-occupied units built prior to 1940

8. DIS: weighted distances to five Boston employment centers
9. RAD: index of accessibility to radial highways
10. TAX: full-value property-tax rate per $10,000
11. PTRATIO: pupil-teacher ratio by town
12. B: $1000(Bk - 0.63)^2$ where Bk is the proportion of blacks by town
13. LSTAT: % lower status of the population
14. MEDV: Median value of owner-occupied homes in $1000s

We can see that the input attributes have a mixture of units.

19.1.1 Load the Dataset

The dataset is available in the `mlbench` package. Let's start off by loading the required packages and loading the dataset.

```
# load packages
library(mlbench)
library(caret)
library(corrplot)
# attach the BostonHousing dataset
data(BostonHousing)
```

Listing 19.1: Load packages and the dataset.

19.1.2 Validation Dataset

It is a good idea to use a validation hold out set. This is a sample of the data that we hold back from our analysis and modeling. We use it right at the end of our project to confirm the accuracy of our final model. It is a smoke test that we can use to see if we messed up and to give us confidence on our estimates of accuracy on unseen data.

```
# Split out validation dataset
# create a list of 80% of the rows in the original dataset we can use for training
set.seed(7)
validationIndex <- createDataPartition(BostonHousing$medv, p=0.80, list=FALSE)
# select 20% of the data for validation
validation <- BostonHousing[-validationIndex,]
# use the remaining 80% of data to training and testing the models
dataset <- BostonHousing[validationIndex,]
```

Listing 19.2: Separate the dataset into a training and validation sets.

19.2 Analyze Data

The objective of this step in the process is to better understand the problem.

19.2.1 Descriptive Statistics

Let's start off by confirming the dimensions of the dataset, e.g. the number of rows and columns.

```
# dimensions of dataset
dim(dataset)
```

Listing 19.3: Calculate the dimensions of the dataset.

We have 407 instances to work with and can confirm the data has 14 attributes including the class attribute `medv`.

```
407  14
```

Listing 19.4: Outoput of the dimensions of the dataset.

Let's also look at the data types of each attribute.

```
# list types for each attribute
sapply(dataset, class)
```

Listing 19.5: Calculate the attribute data types.

We can see that one of the attributes (`chas`) is a factor while all of the others are numeric.

```
     crim        zn     indus      chas       nox        rm       age       dis       rad       tax
        ptratio         b
"numeric" "numeric" "numeric" "factor" "numeric" "numeric" "numeric" "numeric" "numeric"
    "numeric" "numeric" "numeric"
    lstat     medv
"numeric" "numeric"
```

Listing 19.6: Output of the attribute data types.

Let's now take a peak at the first 20 rows of the data.

```
# take a peek at the first 5 rows of the data
head(dataset, n=20)
```

Listing 19.7: Peek at the first few rows of data.

We can confirm that the scales for the attributes are all over the place because of the differing units. We may benefit from some transforms later on.

```
      crim   zn indus chas  nox    rm   age    dis rad tax ptratio    b lstat medv
2  0.02731  0.0  7.07    0 0.469 6.421  78.9 4.9671   2 242   17.8 396.90  9.14 21.6
3  0.02729  0.0  7.07    0 0.469 7.185  61.1 4.9671   2 242   17.8 392.83  4.03 34.7
4  0.03237  0.0  2.18    0 0.458 6.998  45.8 6.0622   3 222   18.7 394.63  2.94 33.4
5  0.06905  0.0  2.18    0 0.458 7.147  54.2 6.0622   3 222   18.7 396.90  5.33 36.2
6  0.02985  0.0  2.18    0 0.458 6.430  58.7 6.0622   3 222   18.7 394.12  5.21 28.7
7  0.08829 12.5 7.87    0 0.524 6.012  66.6 5.5605   5 311   15.2 395.60 12.43 22.9
8  0.14455 12.5 7.87    0 0.524 6.172  96.1 5.9505   5 311   15.2 396.90 19.15 27.1
9  0.21124 12.5 7.87    0 0.524 5.631 100.0 6.0821   5 311   15.2 386.63 29.93 16.5
13 0.09378 12.5 7.87    0 0.524 5.889  39.0 5.4509   5 311   15.2 390.50 15.71 21.7
14 0.62976  0.0 8.14    0 0.538 5.949  61.8 4.7075   4 307   21.0 396.90  8.26 20.4
15 0.63796  0.0 8.14    0 0.538 6.096  84.5 4.4619   4 307   21.0 380.02 10.26 18.2
16 0.62739  0.0 8.14    0 0.538 5.834  56.5 4.4986   4 307   21.0 395.62  8.47 19.9
17 1.05393  0.0 8.14    0 0.538 5.935  29.3 4.4986   4 307   21.0 386.85  6.58 23.1
18 0.78420  0.0 8.14    0 0.538 5.990  81.7 4.2579   4 307   21.0 386.75 14.67 17.5
19 0.80271  0.0 8.14    0 0.538 5.456  36.6 3.7965   4 307   21.0 288.99 11.69 20.2
```

```
20 0.72580 0.0  8.14    0 0.538 5.727 69.5 3.7965 4 307    21.0 390.95 11.28 18.2
23 1.23247 0.0  8.14    0 0.538 6.142 91.7 3.9769 4 307    21.0 396.90 18.72 15.2
25 0.75026 0.0  8.14    0 0.538 5.924 94.1 4.3996 4 307    21.0 394.33 16.30 15.6
26 0.84054 0.0  8.14    0 0.538 5.599 85.7 4.4546 4 307    21.0 303.42 16.51 13.9
27 0.67191 0.0  8.14    0 0.538 5.813 90.3 4.6820 4 307    21.0 376.88 14.81 16.6
```

Listing 19.8: Output of the first few rows of data.

Let's summarize the distribution of each attribute.

```
# summarize attribute distributions
summary(dataset)
```

Listing 19.9: Calculate a summary of each attribute.

We can note that `chas` is a pretty unbalanced factor. We could transform this attribute to numeric to make calculating descriptive statistics and plots easier.

```
      crim                zn              indus        chas          nox
rm              age
 Min.   : 0.00906  Min.   : 0.00  Min.   : 0.46  0:376  Min.   :0.3850  Min.
:3.863  Min.   :
     2.90
 1st Qu.: 0.08556 1st Qu.: 0.00  1st Qu.: 5.19  1: 31  1st Qu.:0.4530  1st
Qu.:5.873  1st
    Qu.: 45.05
 Median : 0.28955 Median : 0.00  Median : 9.90         Median :0.5380  Median
:6.185  Median
    : 77.70
 Mean   : 3.58281  Mean   :10.57  Mean   :11.36         Mean   :0.5577  Mean
:6.279  Mean   :
    68.83
 3rd Qu.: 3.50464 3rd Qu.: 0.00  3rd Qu.:18.10         3rd Qu.:0.6310  3rd
Qu.:6.611  3rd
    Qu.: 94.55
 Max.   :88.97620  Max.   :95.00  Max.   :27.74         Max.   :0.8710  Max.
:8.780  Max.
    :100.00
      dis              rad              tax            ptratio
b               lstat
                   medv
 Min.   : 1.130  Min.   : 1.000  Min.   :188.0  Min.   :12.60  Min.   :  0.32
Min.   : 1.730
    Min.   : 5.00
 1st Qu.: 2.031  1st Qu.: 4.000  1st Qu.:279.0  1st Qu.:17.40  1st Qu.:374.50
1st Qu.: 6.895
    1st Qu.:17.05
 Median : 3.216  Median : 5.000  Median :330.0  Median :19.00  Median :391.13
Median :11.500
    Median :21.20
 Mean   : 3.731  Mean   : 9.464  Mean   :405.6  Mean   :18.49  Mean   :357.88
Mean   :12.827
    Mean   :22.61
 3rd Qu.: 5.100  3rd Qu.:24.000  3rd Qu.:666.0  3rd Qu.:20.20  3rd Qu.:396.27
3rd Qu.:17.175
    3rd Qu.:25.00
 Max.   :10.710  Max.   :24.000  Max.   :711.0  Max.   :22.00  Max.   :396.90
Max.   :37.970
    Max.   :50.00
```

Listing 19.10: Output of the summary of each attribute.

Let's go ahead and convert `chas` to a numeric attribute.

```
dataset[,4] <- as.numeric(as.character(dataset[,4]))
```

Listing 19.11: Covert an attribute to a numeric type.

Now, let's now take a look at the correlation between all of the numeric attributes.

```
cor(dataset[,1:13])
```

Listing 19.12: Calculate the correlation between attributes.

This is interesting. We can see that many of the attributes have a strong correlation (e.g. > 0.70 or < -0.70). For example:

- `nox` and `indus` with 0.77.
- `dist` and `indus` with 0.71.
- `tax` and `indus` with 0.72.
- `age` and `nox` with 0.72.
- `dist` and `nox` with 0.76.

```
             crim         zn      indus       chas        nox         rm        age        dis
                         rad        tax    ptratio          b      lstat
crim    1.00000000 -0.19790631 0.40597009 -0.05713065 0.4232413 -0.21513269 0.3543819
    -0.3905097 0.64240501 0.60622608 0.2892983 -0.3021185 0.47537617
zn      -0.19790631 1.00000000 -0.51895069 -0.04843477 -0.5058512 0.28942883 -0.5707027
    0.6561874 -0.29952976 -0.28791668 -0.3534121 0.1692749 -0.39712686
indus   0.40597009 -0.51895069 1.00000000 0.08003629 0.7665481 -0.37673408 0.6585831
    -0.7230588 0.56774365 0.68070916 0.3292061 -0.3359795 0.59212718
chas    -0.05713065 -0.04843477 0.08003629 1.00000000 0.1027366 0.08252441 0.1093812
    -0.1114242 -0.00901245 -0.02779018 -0.1355438 0.0472442 -0.04569239
nox     0.42324132 -0.50585121 0.76654811 0.10273656 1.0000000 -0.29885055 0.7238371
    -0.7708680 0.58516760 0.65217875 0.1416616 -0.3620791 0.58196447
rm      -0.21513269 0.28942883 -0.37673408 0.08252441 -0.2988506 1.00000000 -0.2325359
    0.1952159 -0.19149122 -0.26794733 -0.3200037 0.1553992 -0.62038075
age     0.35438190 -0.57070265 0.65858310 0.10938121 0.7238371 -0.23253586 1.0000000
    -0.7503321 0.45235421 0.50164657 0.2564318 -0.2512574 0.59321281
dis     -0.39050970 0.65618742 -0.72305885 -0.11142420 -0.7708680 0.19521590 -0.7503321
    1.0000000 -0.49382744 -0.52649325 -0.2021897 0.2826819 -0.49573024
rad     0.64240501 -0.29952976 0.56774365 -0.00901245 0.5851676 -0.19149122 0.4523542
    -0.4938274 1.00000000 0.92137876 0.4531232 -0.4103307 0.47306604
tax     0.60622608 -0.28791668 0.68070916 -0.02779018 0.6521787 -0.26794733 0.5016466
    -0.5264932 0.92137876 1.00000000 0.4419243 -0.4184878 0.52339243
ptratio 0.28929828 -0.35341215 0.32920610 -0.13554380 0.1416616 -0.32000372 0.2564318
    -0.2021897 0.45312318 0.44192428 1.0000000 -0.1495283 0.35375936
b       -0.30211854 0.16927489 -0.33597951 0.04724420 -0.3620791 0.15539923 -0.2512574
    0.2826819 -0.41033069 -0.41848779 -0.1495283 1.0000000 -0.37661571
lstat   0.47537617 -0.39712686 0.59212718 -0.04569239 0.5819645 -0.62038075 0.5932128
    -0.4957302 0.47306604 0.52339243 0.3537594 -0.3766157 1.00000000
```

Listing 19.13: Output of the correlation between attributes.

This is collinearity and we may see better results with regression algorithms if the correlated attributes are removed.

19.2.2 Unimodal Data Visualizations

Let's look at visualizations of individual attributes. It is often useful to look at your data using multiple different visualizations in order to spark ideas. Let's look at histograms of each attribute to get a sense of the data distributions.

```
# histograms each attribute
par(mfrow=c(2,7))
for(i in 1:13) {
  hist(dataset[,i], main=names(dataset)[i])
}
```

Listing 19.14: Calculate histograms.

We can see that some attributes may have an exponential distribution, such as `crim`, `zn`, `age` and `b`. We can see that others may have a bimodal distribution such as `rad` and `tax`.

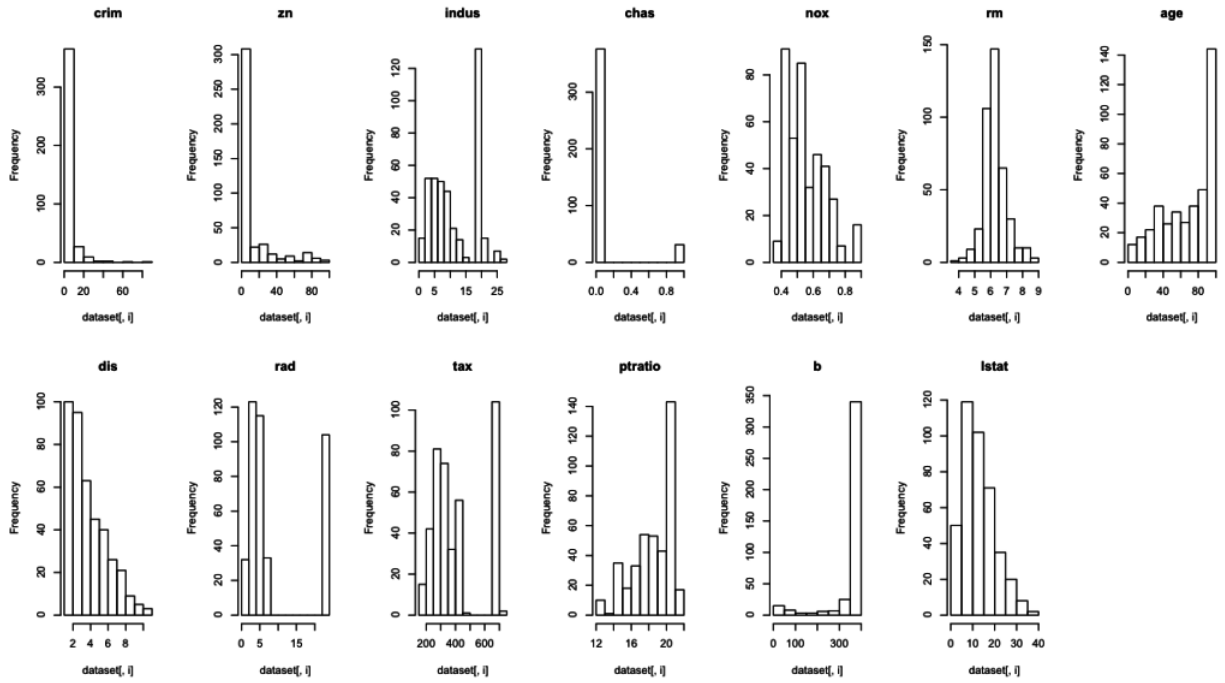

Figure 19.1: Histograms of Boston House Dataset Input Attributes

Let's look at the same distributions using density plots that smooth them out a bit.

```
# density plot for each attribute
par(mfrow=c(2,7))
for(i in 1:13) {
  plot(density(dataset[,i]), main=names(dataset)[i])
}
```

Listing 19.15: Calculate density plots.

This perhaps adds more evidence to our suspicion about possible exponential and bimodal distributions. It also looks like `nox`, `rm` and `lstat` may be skewed Gaussian distributions, which might be helpful later with transforms.

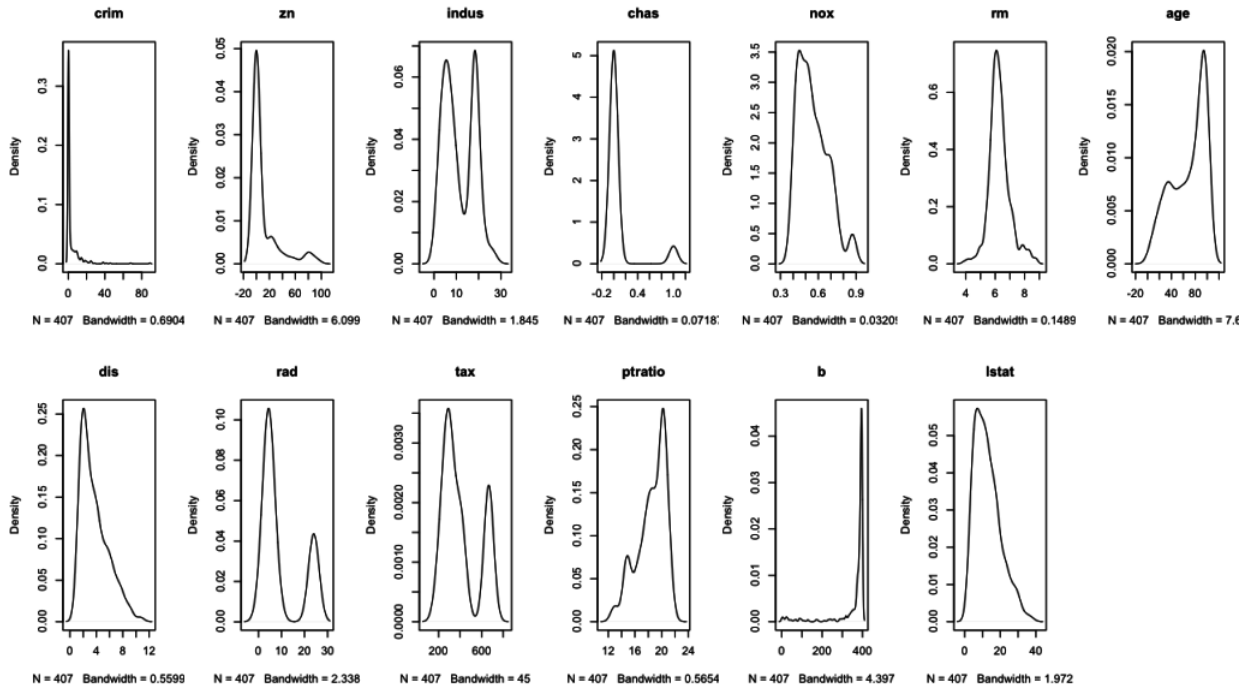

Figure 19.2: Density plots of Boston House Dataset Input Attributes

Let's look at the data with box and whisker plots of each attribute.

```
# boxplots for each attribute
par(mfrow=c(2,7))
for(i in 1:13) {
  boxplot(dataset[,i], main=names(dataset)[i])
}
```

Listing 19.16: Calculate box and whisker plots.

This helps point out the skew in many distributions so much so that data looks like outliers (e.g. beyond the whisker of the plots).

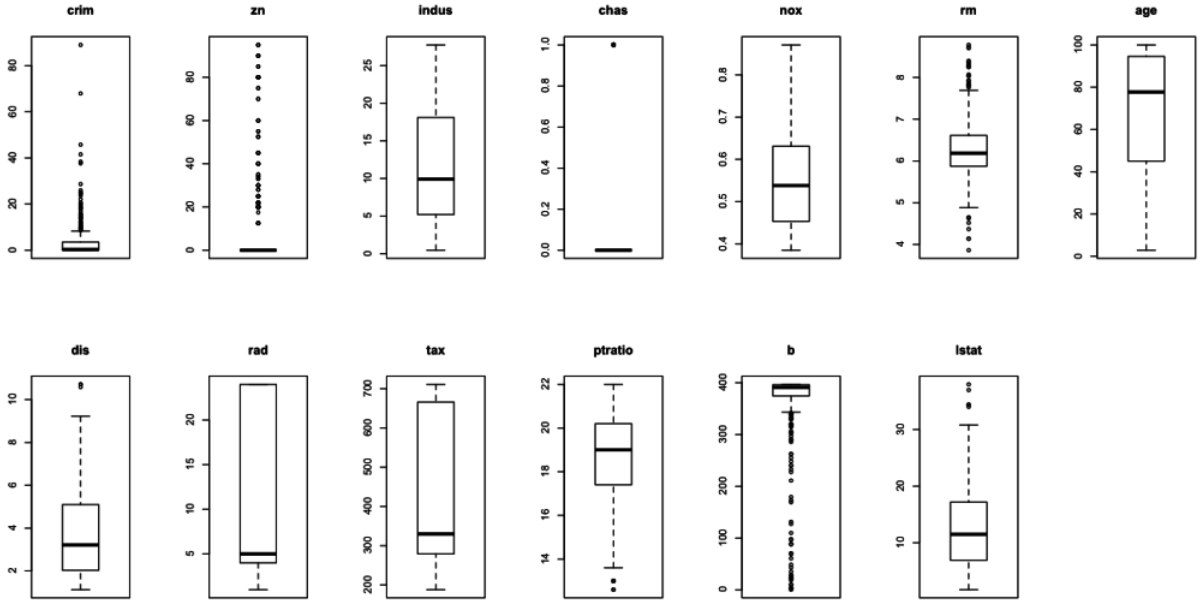

Figure 19.3: Box and Whisker plots of Boston House Dataset Input Attributes

19.2.3 Multimodal Data Visualizations

Let's look at some visualizations of the interactions between variables.

The best place to start is a scatter plot matrix.

```
# scatter plot matrix
pairs(dataset[,1:13])
```

Listing 19.17: Calculate a scatter plot matrix.

We can see that some of the correlated attributes do show good structure in their relationship. Not linear, but nice predictable curved relationships.

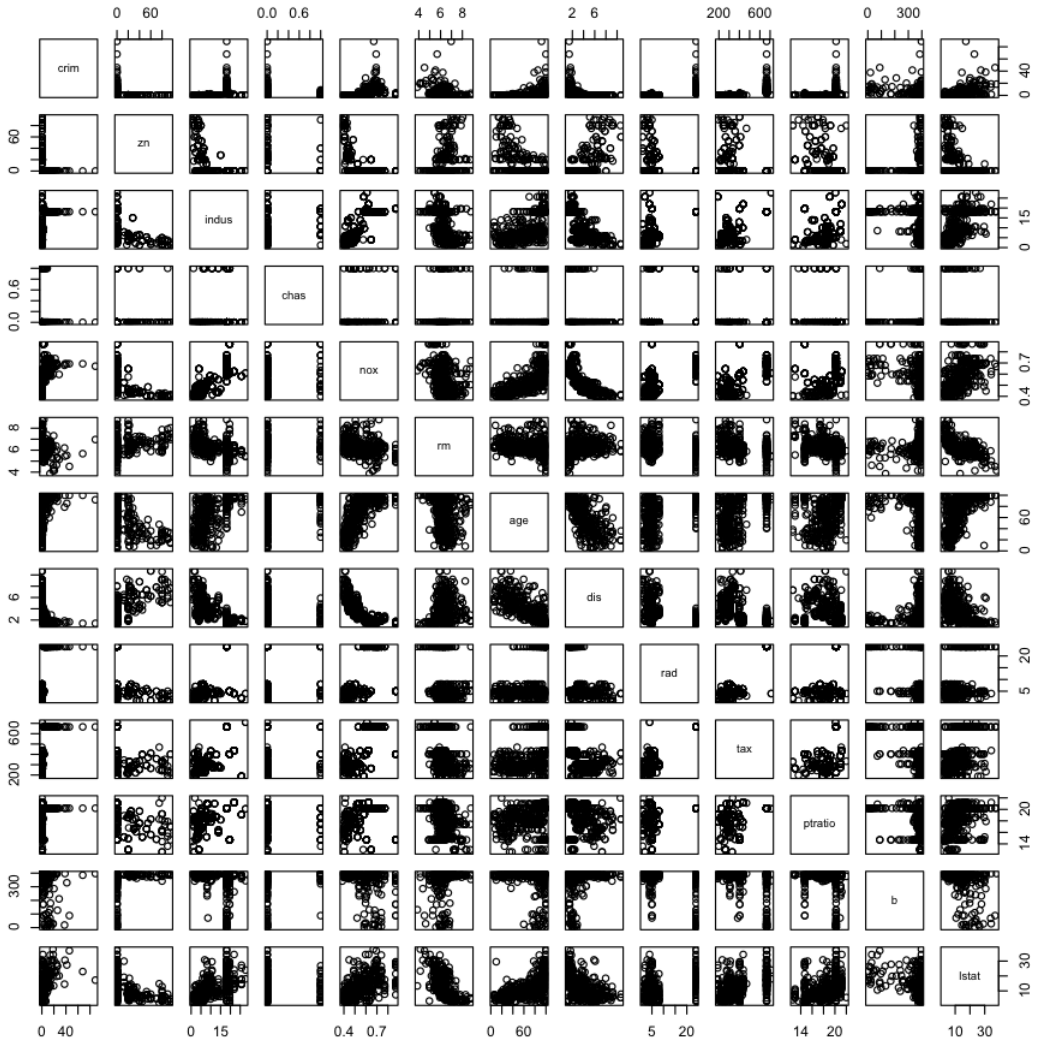

Figure 19.4: Scatterplot Matrix of Boston House Dataset Input Attributes

```
# correlation plot
correlations <- cor(dataset[,1:13])
corrplot(correlations, method="circle")
```

Listing 19.18: Calculate a correlation plot.

The larger darker blue dots confirm the positively correlated attributes we listed early (not the diagonal). We can also see some larger darker red dots that suggest some negatively correlated attributes. For example `tax` and `rad`. These too may be candidates for removal to better improve accuracy of models later on.

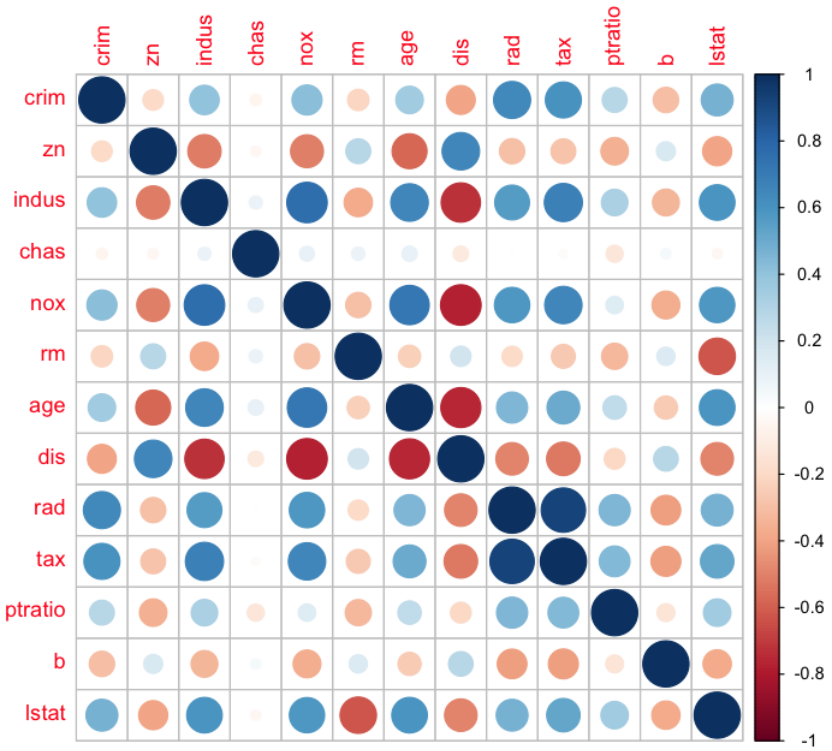

Figure 19.5: Correlation Matrix of Boston House Dataset Input Attributes

19.2.4 Summary of Ideas

There is a lot of structure in this dataset. We need to think about transforms that we could use later to better expose the structure which in turn may improve modeling accuracy. So far it would be worth trying:

- Feature selection and removing the most correlated attributes.
- Normalizing the dataset to reduce the effect of differing scales.
- Standardizing the dataset to reduce the effects of differing distributions.
- Box-Cox transform to see if flattening out some of the distributions improves accuracy.

With lots of additional time I would also explore the possibility of binning (discretization) of the data. This can often improve accuracy for decision tree algorithms.

19.3 Evaluate Algorithms: Baseline

We have no idea what algorithms will do well on this problem. Gut feel suggests regression algorithms like GLM and GLMNET may do well. It is also possible that decision trees and even SVM may do well. I have no idea. Let's design our test harness. We will use 10-fold cross validation (each fold will be about 360 instances for training and 40 for test) with 3 repeats. The dataset is not too small and this is a good standard test harness configuration. We will

evaluate algorithms using the RMSE and R^2 metrics. RMSE will give a gross idea of how wrong all predictions are (0 is perfect) and R^2 will give an idea of how well the model has fit the data (1 is perfect, 0 is worst).

```
# Run algorithms using 10-fold cross validation
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"
```

Listing 19.19: Prepare the test harness for evaluating algorithms.

Let's create a baseline of performance on this problem and spot-check a number of different algorithms. We will select a suite of different algorithms capable of working on this regression problem. The 6 algorithms selected include:

- **Linear Algorithms**: Linear Regression (LR), Generalized Linear Regression (GLM) and Penalized Linear Regression (GLMNET)
- **Nonlinear Algorithms**: Classification and Regression Trees (CART), Support Vector Machines (SVM) with a radial basis function and k-Nearest Neighbors (KNN)

We know the data has differing units of measure so we will standardize the data for this baseline comparison. This will affect those algorithms that prefer data in the same scale (e.g. instance-based methods and some regression algorithms) a chance to do well.

```
# LM
set.seed(7)
fit.lm <- train(medv~., data=dataset, method="lm", metric=metric, preProc=c("center",
    "scale"), trControl=trainControl)
# GLM
set.seed(7)
fit.glm <- train(medv~., data=dataset, method="glm", metric=metric, preProc=c("center",
    "scale"), trControl=trainControl)
# GLMNET
set.seed(7)
fit.glmnet <- train(medv~., data=dataset, method="glmnet", metric=metric,
    preProc=c("center", "scale"), trControl=trainControl)
# SVM
set.seed(7)
fit.svm <- train(medv~., data=dataset, method="svmRadial", metric=metric,
    preProc=c("center", "scale"), trControl=trainControl)
# CART
set.seed(7)
grid <- expand.grid(.cp=c(0, 0.05, 0.1))
fit.cart <- train(medv~., data=dataset, method="rpart", metric=metric, tuneGrid=grid,
    preProc=c("center", "scale"), trControl=trainControl)
# KNN
set.seed(7)
fit.knn <- train(medv~., data=dataset, method="knn", metric=metric, preProc=c("center",
    "scale"), trControl=trainControl)
```

Listing 19.20: Estimate accuracy of machine learning algorithms.

The algorithms all use default tuning parameters, except CART which is fussy on this dataset and has 3 default parameters specified. Let's compare the algorithms. We will use a simple table of results to get a quick idea of what is going on. We will also use a dot plot to show the 95% confidence level for the estimated metrics.

```
# Compare algorithms
results <- resamples(list(LM=fit.lm, GLM=fit.glm, GLMNET=fit.glmnet, SVM=fit.svm,
    CART=fit.cart, KNN=fit.knn))
summary(results)
dotplot(results)
```

Listing 19.21: Collect resample statistics from models and summarize results.

It looks like SVM has the lowest RMSE, followed closely by the other nonlinear algorithms CART and KNN. The linear regression algorithms all appear to be in the same ball park and exhibit slightly worse error.

```
Models: LM, GLM, GLMNET, SVM, CART, KNN
Number of resamples: 30

RMSE
       Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
LM    3.514   4.056 4.773 4.963  5.529 9.448   0
GLM   3.514   4.056 4.773 4.963  5.529 9.448   0
GLMNET 3.484  4.017 4.767 4.955  5.520 9.506   0
SVM   2.377   3.010 3.750 3.952  4.463 8.177   0
CART  2.797   3.434 4.272 4.541  5.437 9.248   0
KNN   2.394   3.509 4.471 4.555  5.089 8.757   0

Rsquared
        Min. 1st Qu. Median Mean 3rd Qu.  Max. NA's
LM    0.3169 0.6682 0.7428 0.7293 0.7984 0.8882 0
GLM   0.3169 0.6682 0.7428 0.7293 0.7984 0.8882 0
GLMNET 0.3092 0.6670 0.7437 0.7296 0.7989 0.8921 0
SVM   0.5229 0.7803 0.8513 0.8290 0.8820 0.9418 0
CART  0.3614 0.6733 0.8197 0.7680 0.8613 0.9026 0
KNN   0.4313 0.7168 0.8024 0.7732 0.8588 0.9146 0
```

Listing 19.22: Summary of estimated model accuracy.

We can also see that SVM and the other nonlinear algorithms have the best fit for the data in their R^2 measures.

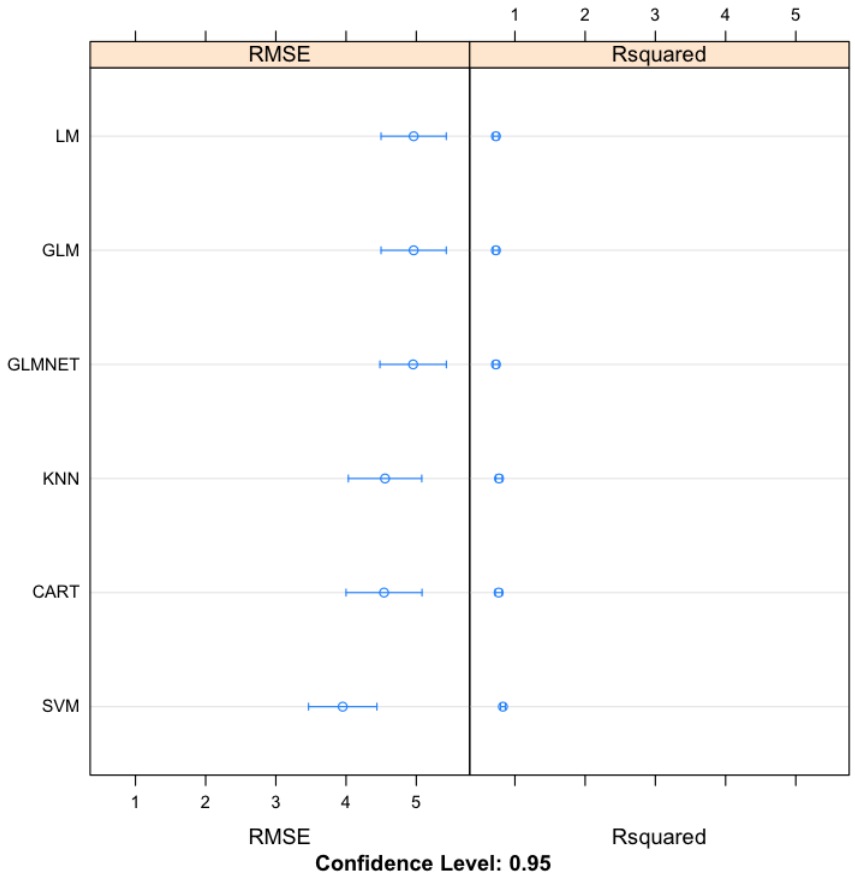

Figure 19.6: Dotplot to compare algorithms on the Boston House Price dataset

Did centering and scaling make a difference to the algorithms other than KNN? I doubt it. But I prefer to hold the data constant at this stage. Perhaps the worse performance of the linear regression algorithms has something to do with the highly correlated attributes. Let's look at that in the next section.

19.4 Evaluate Algorithms: Feature Selection

We have a theory that the correlated attributes are reducing the accuracy of the linear algorithms tried in the baseline spot-check in the last step. In this step we will remove the highly correlated attributes and see what effect that has on the evaluation metrics. We can find and remove the highly correlated attributes using the `findCorrelation()` function from the `caret` package as follows:

```
# remove correlated attributes
# find attributes that are highly correlated
set.seed(7)
cutoff <- 0.70
correlations <- cor(dataset[,1:13])
highlyCorrelated <- findCorrelation(correlations, cutoff=cutoff)
for (value in highlyCorrelated) {
  print(names(dataset)[value])
}
```

```
# create a new dataset without highly correlated features
datasetFeatures <- dataset[,-highlyCorrelated]
dim(datasetFeatures)
```

Listing 19.23: Remove highly correlated attributes from the dataset.

We can see that we have dropped 4 attributes: `indus`, `box`, `tax` and `dis`.

```
[1] "indus"
[1] "nox"
[1] "tax"
[1] "dis"

407  10
```

Listing 19.24: List of highly correlated attributes.

Now let's try the same 6 algorithms from our baseline experiment.

```
# Run algorithms using 10-fold cross validation
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"
# lm
set.seed(7)
fit.lm <- train(medv~., data=datasetFeatures, method="lm", metric=metric,
    preProc=c("center", "scale"), trControl=trainControl)
# GLM
set.seed(7)
fit.glm <- train(medv~., data=datasetFeatures, method="glm", metric=metric,
    preProc=c("center", "scale"), trControl=trainControl)
# GLMNET
set.seed(7)
fit.glmnet <- train(medv~., data=datasetFeatures, method="glmnet", metric=metric,
    preProc=c("center", "scale"), trControl=trainControl)
# SVM
set.seed(7)
fit.svm <- train(medv~., data=datasetFeatures, method="svmRadial", metric=metric,
    preProc=c("center", "scale"), trControl=trainControl)
# CART
set.seed(7)
grid <- expand.grid(.cp=c(0, 0.05, 0.1))
fit.cart <- train(medv~., data=datasetFeatures, method="rpart", metric=metric,
    tuneGrid=grid, preProc=c("center", "scale"), trControl=trainControl)
# KNN
set.seed(7)
fit.knn <- train(medv~., data=datasetFeatures, method="knn", metric=metric,
    preProc=c("center", "scale"), trControl=trainControl)
# Compare algorithms
feature_results <- resamples(list(LM=fit.lm, GLM=fit.glm, GLMNET=fit.glmnet, SVM=fit.svm,
    CART=fit.cart, KNN=fit.knn))
summary(feature_results)
dotplot(feature_results)
```

Listing 19.25: Estimate accuracy of models on modified dataset.

Comparing the results, we can see that this has made the RMSE worse for the linear and the nonlinear algorithms. The correlated attributes we removed are contributing to the accuracy of the models.

```
Models: LM, GLM, GLMNET, SVM, CART, KNN
Number of resamples: 30

RMSE
       Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
LM     3.431   4.439 4.908 5.277  5.998 9.982   0
GLM    3.431   4.439 4.908 5.277  5.998 9.982   0
GLMNET 3.283   4.330 4.950 5.236  5.895 9.869   0
SVM    2.726   3.337 4.100 4.352  5.036 8.503   0
CART   2.661   3.550 4.462 4.618  5.246 9.558   0
KNN    2.488   3.377 4.467 4.453  5.051 8.889   0

Rsquared
        Min. 1st Qu. Median Mean 3rd Qu.  Max. NA's
LM     0.2505  0.6271 0.7125 0.6955 0.7797 0.8877 0
GLM    0.2505  0.6271 0.7125 0.6955 0.7797 0.8877 0
GLMNET 0.2581  0.6274 0.7174 0.7027 0.7783 0.8905 0
SVM    0.4866  0.7522 0.8185 0.7883 0.8673 0.9168 0
CART   0.3310  0.7067 0.7987 0.7607 0.8363 0.9360 0
KNN    0.4105  0.7147 0.7981 0.7759 0.8648 0.9117 0
```

Listing 19.26: Output of estimated model accuracy on modified dataset.

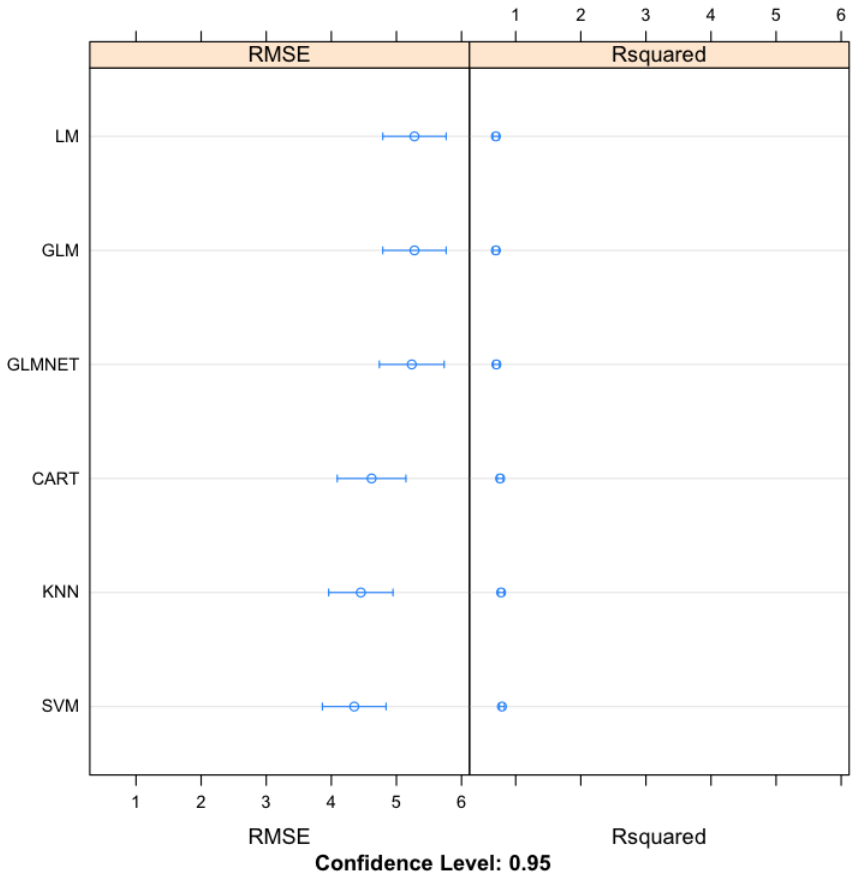

Figure 19.7: Dotplot to compare algorithms on the Boston House Price dataset with feature selection

19.5 Evaluate Algorithms: Box-Cox Transform

We know that some of the attributes have a skew and others perhaps have an exponential distribution. One option would be to explore squaring and log transforms respectively (you could try this!). Another approach would be to use a power transform and let it figure out the amount to correct each attribute. One example is the Box-Cox power transform. Let's try using this transform to rescale the original data and evaluate the effect on the same 6 algorithms. We will also leave in the centering and scaling for the benefit of the instance-based methods.

```
# Run algorithms using 10-fold cross validation
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"
# lm
set.seed(7)
fit.lm <- train(medv~., data=dataset, method="lm", metric=metric, preProc=c("center",
    "scale", "BoxCox"), trControl=trainControl)
# GLM
set.seed(7)
fit.glm <- train(medv~., data=dataset, method="glm", metric=metric, preProc=c("center",
    "scale", "BoxCox"), trControl=trainControl)
# GLMNET
set.seed(7)
fit.glmnet <- train(medv~., data=dataset, method="glmnet", metric=metric,
    preProc=c("center", "scale", "BoxCox"), trControl=trainControl)
# SVM
set.seed(7)
fit.svm <- train(medv~., data=dataset, method="svmRadial", metric=metric,
    preProc=c("center", "scale", "BoxCox"), trControl=trainControl)
# CART
set.seed(7)
grid <- expand.grid(.cp=c(0, 0.05, 0.1))
fit.cart <- train(medv~., data=dataset, method="rpart", metric=metric, tuneGrid=grid,
    preProc=c("center", "scale", "BoxCox"), trControl=trainControl)
# KNN
set.seed(7)
fit.knn <- train(medv~., data=dataset, method="knn", metric=metric, preProc=c("center",
    "scale", "BoxCox"), trControl=trainControl)
# Compare algorithms
transformResults <- resamples(list(LM=fit.lm, GLM=fit.glm, GLMNET=fit.glmnet, SVM=fit.svm,
    CART=fit.cart, KNN=fit.knn))
summary(transformResults)
dotplot(transformResults)
```

Listing 19.27: Estimate accuracy of algorithms on transformed dataset.

We can see that this decreased the RMSE and increased the R^2 on all except the CART algorithms. The RMSE of SVM dropped to an average of 3.761.

```
Models: LM, GLM, GLMNET, SVM, CART, KNN
Number of resamples: 30

RMSE
       Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
LM    3.404   3.811 4.399 4.621  5.167 7.781   0
GLM   3.404   3.811 4.399 4.621  5.167 7.781   0
GLMNET 3.312  3.802 4.429 4.611  5.123 7.976   0
```

```
SVM    2.336   2.937  3.543 3.761  4.216 8.207   0
CART   2.797   3.434  4.272 4.541  5.437 9.248   0
KNN    2.474   3.608  4.308 4.563  5.080 8.922   0

Rsquared
         Min. 1st Qu. Median Mean 3rd Qu.  Max. NA's
LM     0.5439  0.7177 0.7832 0.7627 0.8257 0.8861 0
GLM    0.5439  0.7177 0.7832 0.7627 0.8257 0.8861 0
GLMNET 0.5198  0.7172 0.7808 0.7634 0.8297 0.8909 0
SVM    0.5082  0.8249 0.8760 0.8452 0.8998 0.9450 0
CART   0.3614  0.6733 0.8197 0.7680 0.8613 0.9026 0
KNN    0.4065  0.7562 0.8073 0.7790 0.8594 0.9043 0
```

Listing 19.28: Output of estimated accuracy of models on transformed dataset.

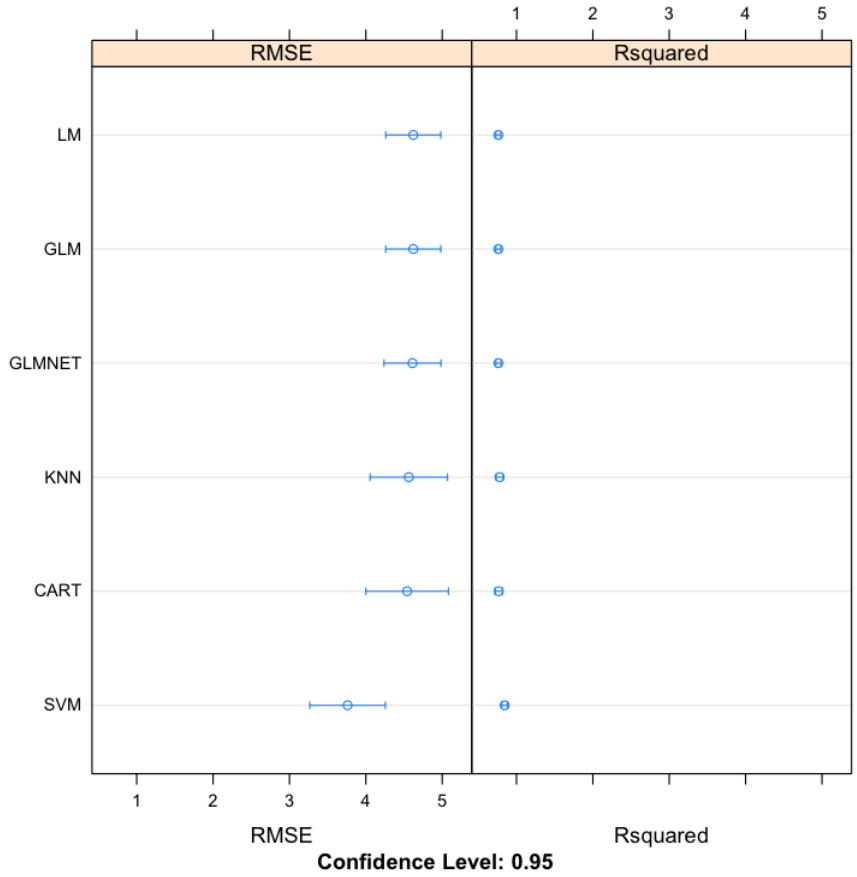

Figure 19.8: Dotplot Compare Machine Learning Algorithms on the Boston House Price Dataset with Box-Cox Power Transform

19.6 Improve Results With Tuning

We can improve the accuracy of the well performing algorithms by tuning their parameters. In this section we will look at tuning the parameters of SVM with a Radial Basis Function (RBF). with more time it might be worth exploring tuning of the parameters for CART and KNN. It

might also be worth exploring other kernels for SVM besides the RBF. Let's look at the default parameters already adopted.

```
print(fit.svm)
```

Listing 19.29: Display estimated accuracy of a model.

The C parameter is the cost constraint used by SVM. Learn more in the help for the `ksvm` function `?ksvm`. We can see from previous results that a C value of 1.0 is a good starting point.

```
Support Vector Machines with Radial Basis Function Kernel

407 samples
 13 predictor

Pre-processing: centered (13), scaled (13), Box-Cox transformation (11)
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 366, 367, 366, 366, 367, 367, ...
Resampling results across tuning parameters:

  C     RMSE     Rsquared  RMSE SD  Rsquared SD
  0.25  4.555338 0.7906921 1.533391 0.11596196
  0.50  4.111564 0.8204520 1.467153 0.10573527
  1.00  3.761245 0.8451964 1.323218 0.09487941

Tuning parameter 'sigma' was held constant at a value of 0.07491936
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were sigma = 0.07491936 and C = 1.
```

Listing 19.30: Output of estimated accuracy of a model.

Let's design a grid search around a C value of 1. We might see a small trend of decreasing RMSE with increasing C, so let's try all integer C values between 1 and 10. Another parameter that `caret` let's us tune is the `sigma` parameter. This is a smoothing parameter. Good `sigma` values often start around 0.1, so we will try numbers before and after.

```
# tune SVM sigma and C parametres
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"
set.seed(7)
grid <- expand.grid(.sigma=c(0.025, 0.05, 0.1, 0.15), .C=seq(1, 10, by=1))
fit.svm <- train(medv~., data=dataset, method="svmRadial", metric=metric, tuneGrid=grid,
    preProc=c("BoxCox"), trControl=trainControl)
print(fit.svm)
plot(fit.svm)
```

Listing 19.31: Tune the parameters of a model.

We can see that the `sigma` values flatten out with larger C cost constraints. It looks like we might do well with a sigma of 0.05 and a C of 10. This gives us a respectable RMSE of 2.977085.

```
Support Vector Machines with Radial Basis Function Kernel

407 samples
 13 predictor
```

```
Pre-processing: Box-Cox transformation (11)
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 366, 367, 366, 366, 367, 367, ...
Resampling results across tuning parameters:

  sigma  C   RMSE     Rsquared  RMSE SD   Rsquared SD
  0.025   1  3.889703 0.8335201 1.4904294 0.11313650
  0.025   2  3.685009 0.8470869 1.4126374 0.10919207
  0.025   3  3.562851 0.8553298 1.3664097 0.10658536
  0.025   4  3.453041 0.8628558 1.3167032 0.10282201
  0.025   5  3.372501 0.8686287 1.2700128 0.09837303
  0.025   6  3.306693 0.8731149 1.2461425 0.09559507
  0.025   7  3.261471 0.8761873 1.2222133 0.09312101
  0.025   8  3.232191 0.8780827 1.2061263 0.09157306
  0.025   9  3.208426 0.8797434 1.1863449 0.08988573
  0.025  10  3.186740 0.8812147 1.1649693 0.08812914
  0.050   1  3.771428 0.8438368 1.3673050 0.09997011
  0.050   2  3.484116 0.8634056 1.2563140 0.09475264
  0.050   3  3.282230 0.8768963 1.1689480 0.08515589
  0.050   4  3.179856 0.8829293 1.1417952 0.08311244
  0.050   5  3.105290 0.8873315 1.1139808 0.08053864
  0.050   6  3.054516 0.8907211 1.0797893 0.07759377
  0.050   7  3.024010 0.8925927 1.0630675 0.07622395
  0.050   8  3.003371 0.8936101 1.0533396 0.07544553
  0.050   9  2.984457 0.8944677 1.0475715 0.07501395
  0.050  10  2.977085 0.8948000 1.0411527 0.07437254
  0.100   1  3.762027 0.8453751 1.2904160 0.09047894
...
  0.150  10  3.156134 0.8807506 0.9741032 0.07304302

RMSE was used to select the optimal model using the smallest value.
The final values used for the model were sigma = 0.05 and C = 10.
```

Listing 19.32: Output of tuning the parameters of a model.

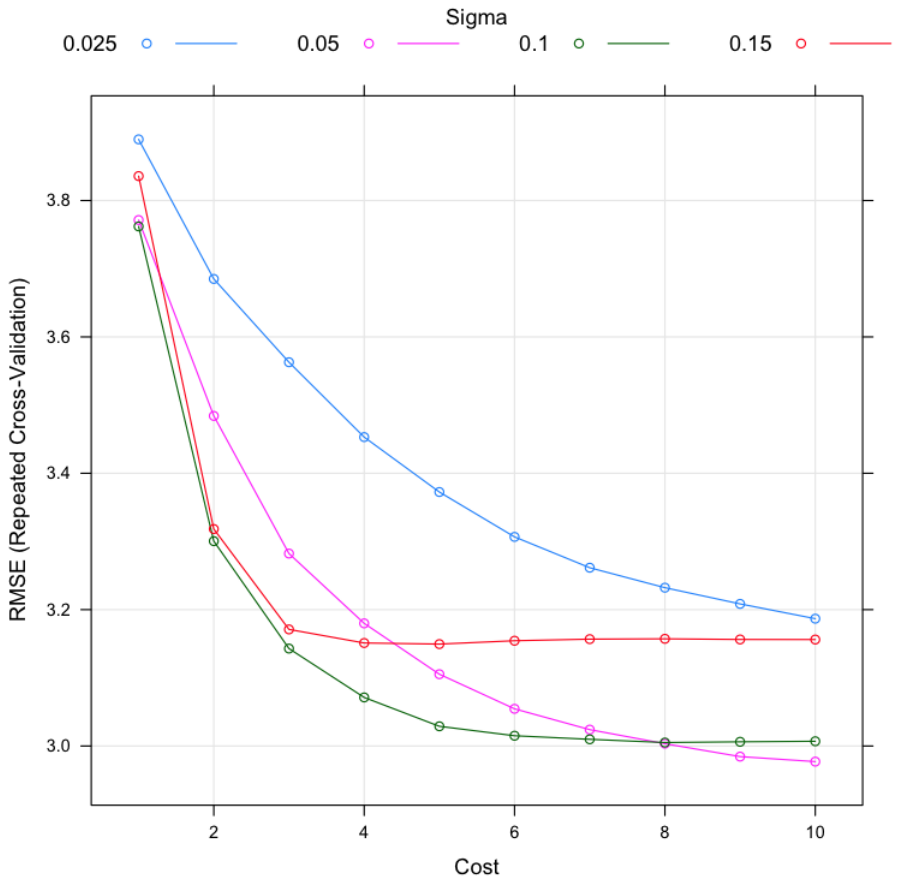

Figure 19.9: Algorithm Tuning Results for SVM on the Boston House Price Dataset

If we wanted to take this further, we could try even more fine tuning with more grid searches. We could also explore trying to tune other parameters of the underlying `ksvm()` function. Finally and as already mentioned, we could perform some grid searches on the other nonlinear regression methods.

19.7 Ensemble Methods

We can try some ensemble methods on the problem and see if we can get a further decrease in our RMSE. In this section we will look at some boosting and bagging techniques for decision trees. Additional approaches you could look into would be blending the predictions of multiple well performing models together, called stacking. Let's take a look at the following ensemble methods:

- Random Forest, bagging (RF).
- Gradient Boosting Machines (GBM).
- Cubist, boosting (CUBIST).

```
# try ensembles
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"
# Random Forest
set.seed(seed)
fit.rf <- train(medv~., data=dataset, method="rf", metric=metric, preProc=c("BoxCox"),
    trControl=trainControl)
# Stochastic Gradient Boosting
set.seed(seed)
fit.gbm <- train(medv~., data=dataset, method="gbm", metric=metric, preProc=c("BoxCox"),
    trControl=trainControl, verbose=FALSE)
# Cubist
set.seed(seed)
fit.cubist <- train(medv~., data=dataset, method="cubist", metric=metric,
    preProc=c("BoxCox"), trControl=trainControl)
# Compare algorithms
ensembleResults <- resamples(list(RF=fit.rf, GBM=fit.gbm, CUBIST=fit.cubist))
summary(ensembleResults)
dotplot(ensembleResults)
```

Listing 19.33: Estiamte accuracy of ensemble methods.

We can see that Cubist was the most accurate with an RMSE that was lower than that achieved by tuning SVM.

```
Models: RF, GBM, CUBIST
Number of resamples: 30

RMSE
        Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
RF     2.072  2.484 2.849 3.199  3.593 7.552   0
GBM    2.349  2.514 2.855 3.314  3.734 7.326   0
CUBIST 1.671  2.325 2.598 2.935  2.862 7.894   0

Rsquared
         Min. 1st Qu. Median Mean 3rd Qu.  Max. NA's
RF     0.5809 0.8736 0.9179 0.8825 0.9323 0.9578 0
GBM    0.6067 0.8425 0.9045 0.8701 0.9264 0.9494 0
CUBIST 0.5312 0.9051 0.9272 0.8945 0.9401 0.9700 0
```

Listing 19.34: Output of estimated accuracy of ensemble methods.

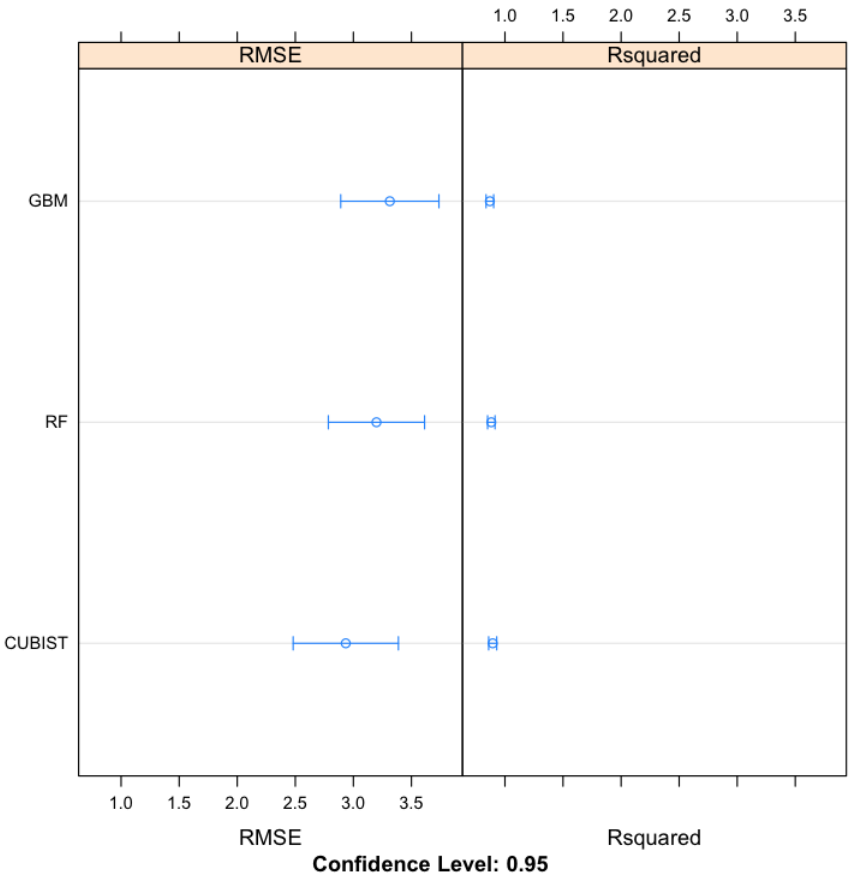

Figure 19.10: Ensemble Methods on the Boston House Price Dataset

Let's dive deeper into Cubist and see if we can tune it further and get more skill out of it. Cubist has two parameters that are tunable with `caret`: `committees` which is the number of boosting operations and `neighbors` which is used during prediction and is the number of instances used to correct the rule-based prediction (although the documentation is perhaps a little ambiguous on this). For more information about Cubist see the help on the function `?cubist`. Let's first look at the default tuning parameter used by `caret` that resulted in our accurate model.

```
# look at parameters used for Cubist
print(fit.cubist)
```

Listing 19.35: Summarize accuracy of a model.

We can see that the best RMSE was achieved with `committees = 20` and `neighbors = 5`.

```
Cubist

407 samples
 13 predictor

Pre-processing: Box-Cox transformation (11)
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 366, 367, 366, 366, 367, 367, ...
Resampling results across tuning parameters:
```

```
  committees neighbors RMSE     Rsquared  RMSE SD  Rsquared SD
   1         0         3.805611 0.8291291 1.446821 0.12468698
   1         5         3.372092 0.8607419 1.441227 0.12021181
   1         9         3.478679 0.8544866 1.461305 0.12201360
  10         0         3.321898 0.8684445 1.331075 0.11184780
  10         5         3.014602 0.8880220 1.310053 0.10505469
  10         9         3.087316 0.8836591 1.343094 0.10781126
  20         0         3.248094 0.8747071 1.252816 0.09983180
  20         5         2.934577 0.8944885 1.213757 0.09319354
  20         9         3.011090 0.8899419 1.253828 0.09601677

RMSE was used to select the optimal model using the smallest value.
The final values used for the model were committees = 20 and neighbors = 5.
```

Listing 19.36: Output of summary of a model.

Let's use a grid search to tune around those values. We'll try all `committees` between 15 and 25 and spot-check a `neighbors` value above and below 5.

```
# Tune the Cubist algorithm
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"
set.seed(7)
grid <- expand.grid(.committees=seq(15, 25, by=1), .neighbors=c(3, 5, 7))
tune.cubist <- train(medv~., data=dataset, method="cubist", metric=metric,
    preProc=c("BoxCox"), tuneGrid=grid, trControl=trainControl)
print(tune.cubist)
plot(tune.cubist)
```

Listing 19.37: Tune the parameters of a model.

We can see that we have achieved a more accurate model again with an RMSE of 2.822 using `committees = 18` and `neighbors = 3`.

```
Cubist

407 samples
 13 predictor

Pre-processing: Box-Cox transformation (11)
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 366, 367, 366, 366, 367, 367, ...
Resampling results across tuning parameters:

  committees neighbors RMSE     Rsquared  RMSE SD  Rsquared SD
  15         3         2.843135 0.9009984 1.153144 0.08574148
  15         5         2.945379 0.8942976 1.196652 0.09127607
  15         7         2.992984 0.8913018 1.229347 0.09364576
  16         3         2.838901 0.9006522 1.191235 0.08975360
  16         5         2.943103 0.8937805 1.238710 0.09565818
  16         7         2.990762 0.8907565 1.273834 0.09828343
  17         3         2.831608 0.9014030 1.159990 0.08674319
  17         5         2.936655 0.8944879 1.205534 0.09255285
  17         7         2.984208 0.8915462 1.238689 0.09490105
  18         3         2.822586 0.9018685 1.160574 0.08691505
  18         5         2.928190 0.8949810 1.204704 0.09243459
```

```
  18         7         2.974838 0.8920938 1.237423 0.09472307
  19         3         2.833172 0.9015738 1.154942 0.08586679
  19         5         2.935970 0.8947952 1.200135 0.09126598
  19         7         2.983656 0.8918658 1.232270 0.09355376
  20         3         2.828846 0.9014772 1.166684 0.08754715
...

RMSE was used to select the optimal model using the smallest value.
The final values used for the model were committees = 18 and neighbors = 3.
```

Listing 19.38: Output estimated accuracy of tuned model.

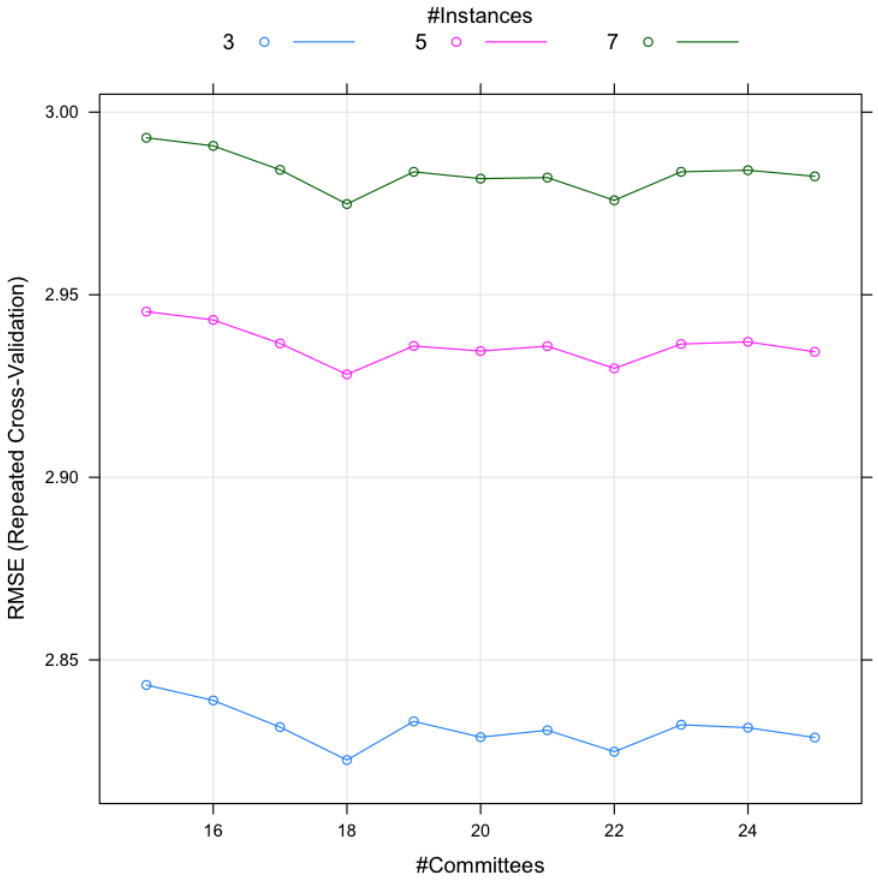

Figure 19.11: Tuning the parameters of the Cubist Algorithm on the Boston House Price Dataset

With more time we could tune the Cubist algorithm further. Also, with results like this, it also suggests that it might be worth investigating whether we can get more out of the GBM or other boosting implementations.

19.8 Finalize Model

It looks like the results for the Cubist algorithm are the most accurate. Let's finalize it by creating a new standalone Cubist model with the parameters above trained using the whole dataset. We must also use the Box-Cox power transform.

```
# prepare the data transform using training data
set.seed(7)
x <- dataset[,1:13]
y <- dataset[,14]
preprocessParams <- preProcess(x, method=c("BoxCox"))
transX <- predict(preprocessParams, x)
# train the final model
finalModel <- cubist(x=transX, y=y, committees=18)
summary(finalModel)
```

Listing 19.39: Prepare the data transform and finalize the model.

We can now use this model to evaluate our held-out validation dataset. Again, we must prepare the input data using the same Box-Cox transform.

```
# transform the validation dataset
set.seed(7)
valX <- validation[,1:13]
trans_valX <- predict(preprocessParams, valX)
valY <- validation[,14]
# use final model to make predictions on the validation dataset
predictions <- predict(finalModel, newdata=trans_valX, neighbors=3)
# calculate RMSE
rmse <- RMSE(predictions, valY)
r2 <- R2(predictions, valY)
print(rmse)
```

Listing 19.40: Make predictions using the finalized model.

We can see that the estimated RMSE on this unseen data is about 2.666, lower but not too dissimilar from our expected RMSE of 2.822.

```
2.666336
```

Listing 19.41: Estimated accuracy on validation dataset.

19.9 Summary

In this lesson you worked through a regression predictive modeling machine learning problem from end-to-end using R. Specifically, the steps covered were:

1. Problem Definition (Boston house price data).
2. Analyze Data (some skewed distributions and correlated attributes).
3. Evaluate Algorithms (SVM with radial basis function looks good).
4. Feature Selection (removing correlated attributes didn't help).
5. Transforms (Box-Cox transform made things better).
6. Algorithm Tuning (getting the most from SVM).
7. Ensemble Methods (Bagging and Boosting and getting the most from Cubist).

8. Finalize Model (use all training data and confirm using validation dataset).

Working through this case study showed you how the recipes for specific machine learning tasks can be pulled together into a complete project. Working through this case study is good practice at applied machine learning using R.

19.9.1 Next Step

You have now completed two predictive modeling machine learning projects end-to-end. The first was a multiclass classification problem and this second project was a regression problem. Next is the third and final case study on a binary classification problem.

Chapter 20

Binary Classification Machine Learning Case Study Project

Working through machine learning problems end-to-end is the best way to practice applied machine learning. In this lesson you will work through a binary classification problem using R from start to finish. After completing this project, you will know:

1. How to work through a binary classification predictive modeling problem end-to-end.
2. How to use data transforms and model tuning to improve model accuracy.
3. How to identify when you have hit an accuracy ceiling and the point of diminishing returns on a project.

Let's get started.

20.1 Problem Definition

For this project we will investigate the Wisconsin Breast Cancer Dataset described in Chapter 5. Each record in the dataset represents one breast cancer tissue sample. The data was collected from University of Wisconsin Hospitals. Below is a summary of the attributes taken from the UCI Machine Learning repository.

- Sample id number.
- Clump Thickness.
- Uniformity of Cell Size.
- Uniformity of Cell Shape.
- Marginal Adhesion.
- Single Epithelial Cell Size.
- Bare Nuclei.
- Bland Chromatin.

- Normal Nucleoli.
- Mitoses.
- Class.

Although the test methodologies differ, the best published results appear to be in the high 90% accuracy such as 96% and 97%. Achieving results in this range would be desirable in this case study.

20.1.1 Load the Dataset

The dataset is available in the `mlbench` package. Let's start off by loading the required packages and loading the dataset.

```
# load packages
library(mlbench)
library(caret)
# Load data
data(BreastCancer)
```

Listing 20.1: Load packages and dataset.

20.1.2 Validation Dataset

Let's create a validation dataset. This is a sample of our raw data that we hold back from the modeling process. We use it right at the end of the project to confirm the accuracy of our final model. It is a smoke test that we can use to see if we messed up and to give us confidence on our estimates of accuracy on unseen data.

```
# Split out validation dataset
# create a list of 80% of the rows in the original dataset we can use for training
set.seed(7)
validationIndex <- createDataPartition(BreastCancer$Class, p=0.80, list=FALSE)
# select 20% of the data for validation
validation <- BreastCancer[-validationIndex,]
# use the remaining 80% of data to training and testing the models
dataset <- BreastCancer[validationIndex,]
```

Listing 20.2: Split dataset into training and validation sets.

20.2 Analyze Data

The objective of this step in the process is to better understand the dataset.

20.2.1 Descriptive Statistics

Let's start off by confirming the dimensions of the dataset.

```
# dimensions of dataaset
dim(dataset)
```

Listing 20.3: Calculate the dimensions of the dataset.

We can see that we have 560 rows and 11 attributes.

```
560  11
```

Listing 20.4: Output of the dimensions of the dataset.

Let's eye-ball some data and see what we are working with. We can preview the first 20 rows.

```
# peek
head(dataset, n=20)
```

Listing 20.5: Display first few rows of the dataset.

We can see that the sample number (`Id`) is probably not going to be useful. We can probably remove it. We can see that all of the inputs are integers. We can also see that we may have some missing data (`NA`).

```
Id Cl.thickness Cell.size Cell.shape Marg.adhesion Epith.c.size Bare.nuclei Bl.cromatin
    Normal.nucleoli Mitoses Class
2  1002945           5         4           4            5            7          10           3
                2       1    benign
4  1016277           6         8           8            1            3           4           3
                7       1    benign
5  1017023           4         1           1            3            2           1           3
                1       1    benign
6  1017122           8        10          10            8            7          10           9
                7       1 malignant
7  1018099           1         1           1            1            2          10           3
                1       1    benign
9  1033078           2         1           1            1            2           1           1
                1       5    benign
10 1033078           4         2           1            1            2           1           2
                1       1    benign
11 1035283           1         1           1            1            1           1           3
                1       1    benign
12 1036172           2         1           1            1            2           1           2
                1       1    benign
14 1043999           1         1           1            1            2           3           3
                1       1    benign
15 1044572           8         7           5           10            7           9           5
                5       4 malignant
16 1047630           7         4           6            4            6           1           4
                3       1 malignant
17 1048672           4         1           1            1            2           1           2
                1       1    benign
18 1049815           4         1           1            1            2           1           3
                1       1    benign
19 1050670          10         7           7            6            4          10           4
                1       2 malignant
20 1050718           6         1           1            1            2           1           3
                1       1    benign
```

```
21 1054590          7          3          2          10          5          10          5
                4          4 malignant
22 1054593         10          5          5           3          6           7          7
               10          1 malignant
23 1056784          3          1          1           1          2           1          2
                1          1    benign
24 1057013          8          4          5           1          2        <NA>          7
                3          1 malignant
```

Listing 20.6: Output of the first few rows of the dataset.

Let's look at the attribute data types.

```
# types
sapply(dataset, class)
```

Listing 20.7: Display the data type of attributes in the dataset.

We can see that besides the `Id`, the attributes are factors. This makes sense. I think for modeling it may be more useful to work with the data as numbers than factors. Factors might make things easier for decision tree algorithms (or not). Given that there is an ordinal relationship between the levels we can expose that structure to other algorithms better if we work directly with the integer numbers.

```
$Id
[1] "character"

$Cl.thickness
[1] "ordered" "factor"

$Cell.size
[1] "ordered" "factor"

$Cell.shape
[1] "ordered" "factor"

$Marg.adhesion
[1] "ordered" "factor"

$Epith.c.size
[1] "ordered" "factor"

$Bare.nuclei
[1] "factor"

$Bl.cromatin
[1] "factor"

$Normal.nucleoli
[1] "factor"

$Mitoses
[1] "factor"

$Class
[1] "factor"
```

Listing 20.8: Output of the data type of attributes in the dataset.

Let's go ahead and remove the `Id` column and convert the dataset to numeric values.

```
# Remove redundant variable Id
dataset <- dataset[,-1]
# convert input values to numeric
for(i in 1:9) {
 dataset[,i] <- as.numeric(as.character(dataset[,i]))
}
```

Listing 20.9: Remove a column and covert the dataset to numeric types.

Now, let's take a summary of the data and see what we have.

```
# summary
summary(dataset)
```

Listing 20.10: Calculate a summary of attributes in the dataset.

Interestingly, we can see we have 13 `NA` values for the `Bare.nuclei` attribute. This suggests we may need to remove the records (or impute values) with `NA` values for some analysis and modeling techniques. We can also see that all attributes have integer values in the range `[1,10]`. This suggests that we may not see much benefit from normalizing attributes for instance-based methods like KNN.

```
 Cl.thickness     Cell.size      Cell.shape    Marg.adhesion   Epith.c.size    Bare.nuclei
        Bl.cromatin    Normal.nucleoli   Mitoses           Class
 Min.   : 1.000  Min.   : 1.000  Min.   : 1.000  Min.   : 1.000  Min.   : 1.000  Min.   : 1.000
     Min.   : 1.000  Min.   : 1.000  Min.   : 1.000  benign   :367
 1st Qu.: 2.000  1st Qu.: 1.000  1st Qu.: 1.000  1st Qu.: 1.000  1st Qu.: 2.000  1st Qu.:
    1.000  1st Qu.: 2.000  1st Qu.: 1.000  1st Qu.: 1.000  malignant:193
 Median : 4.000  Median : 1.000  Median : 2.000  Median : 1.000  Median : 2.000  Median :
    1.000  Median : 3.000  Median : 1.000  Median : 1.000
 Mean   : 4.384  Mean   : 3.116  Mean   : 3.198  Mean   : 2.875  Mean   : 3.232  Mean   : 3.468
     Mean   : 3.405  Mean   : 2.877  Mean   : 1.611
 3rd Qu.: 6.000  3rd Qu.: 5.000  3rd Qu.: 5.000  3rd Qu.: 4.000  3rd Qu.: 4.000  3rd Qu.:
    5.000  3rd Qu.: 4.250  3rd Qu.: 4.000  3rd Qu.: 1.000
 Max.   :10.000  Max.   :10.000  Max.   :10.000  Max.   :10.000  Max.   :10.000  Max.   :10.000
     Max.   :10.000  Max.   :10.000  Max.   :10.000
```

Listing 20.11: Output summary of attributes in the dataset.

We can also see that there is some imbalance in the `Class` values. Let's take a closer look at the breakdown of the `Class` values.

```
# class distribution
cbind(freq=table(dataset$Class), percentage=prop.table(table(dataset$Class))*100)
```

Listing 20.12: Summarize the breakdown of the class attribute.

There is indeed a 65% to 35% split for benign-malignant in the class values which is imbalanced, but not so much that we need to be thinking about rebalancing the dataset, at least not yet.

```
         freq percentage
benign    367   65.53571
```

```
malignant 193  34.46429
```

Listing 20.13: Output of the breakdown of the class attribute.

Finally, let's look at the correlation between the attributes. We have to exclude the 13 rows with `NA` values (incomplete cases) when calculating the correlations.

```
# summarize correlations between input variables
complete_cases <- complete.cases(dataset)
cor(dataset[complete_cases,1:9])
```

Listing 20.14: Calculate the correlation between attributes.

We can see some modest to high correlation between some of the attributes. For example between `cell shape` and `cell size` at 0.90 correlation. Some algorithms may benefit from removing the highly correlated attributes.

```
              Cl.thickness Cell.size Cell.shape Marg.adhesion Epith.c.size Bare.nuclei
                  Bl.cromatin Normal.nucleoli Mitoses
Cl.thickness     1.0000000 0.6200884 0.6302917   0.4741733   0.5089557  0.5600770  0.5290733
        0.5143933 0.3426018
Cell.size        0.6200884 1.0000000 0.9011340   0.7141150   0.7404824  0.6687226  0.7502700
        0.7072182 0.4506532
Cell.shape       0.6302917 0.9011340 1.0000000   0.6846206   0.7043423  0.6896724  0.7276114
        0.7127155 0.4345125
Marg.adhesion    0.4741733 0.7141150 0.6846206   1.0000000   0.5860660  0.6660165  0.6660533
        0.6031036 0.4314910
Epith.c.size     0.5089557 0.7404824 0.7043423   0.5860660   1.0000000  0.5568406  0.6102032
        0.6433364 0.4775271
Bare.nuclei      0.5600770 0.6687226 0.6896724   0.6660165   0.5568406  1.0000000  0.6668483
        0.5795794 0.3539473
Bl.cromatin      0.5290733 0.7502700 0.7276114   0.6660533   0.6102032  0.6668483  1.0000000
        0.6838547 0.3545122
Normal.nucleoli  0.5143933 0.7072182 0.7127155   0.6031036   0.6433364  0.5795794  0.6838547
        1.0000000 0.4084127
Mitoses          0.3426018 0.4506532 0.4345125   0.4314910   0.4775271  0.3539473  0.3545122
        0.4084127 1.0000000
```

Listing 20.15: Output of the correlation between attributes.

20.2.2 Unimodal Data Visualizations

Let's look at the distribution of individual attributes in the dataset. We'll start with histograms of all of the attributes.

```
# histograms each attribute
par(mfrow=c(3,3))
for(i in 1:9) {
  hist(dataset[,i], main=names(dataset)[i])
}
```

Listing 20.16: Calculate histograms.

We can see that almost all of the distributions have an exponential or bimodal shape to them. We may benefit from log transforms or other power transforms later on.

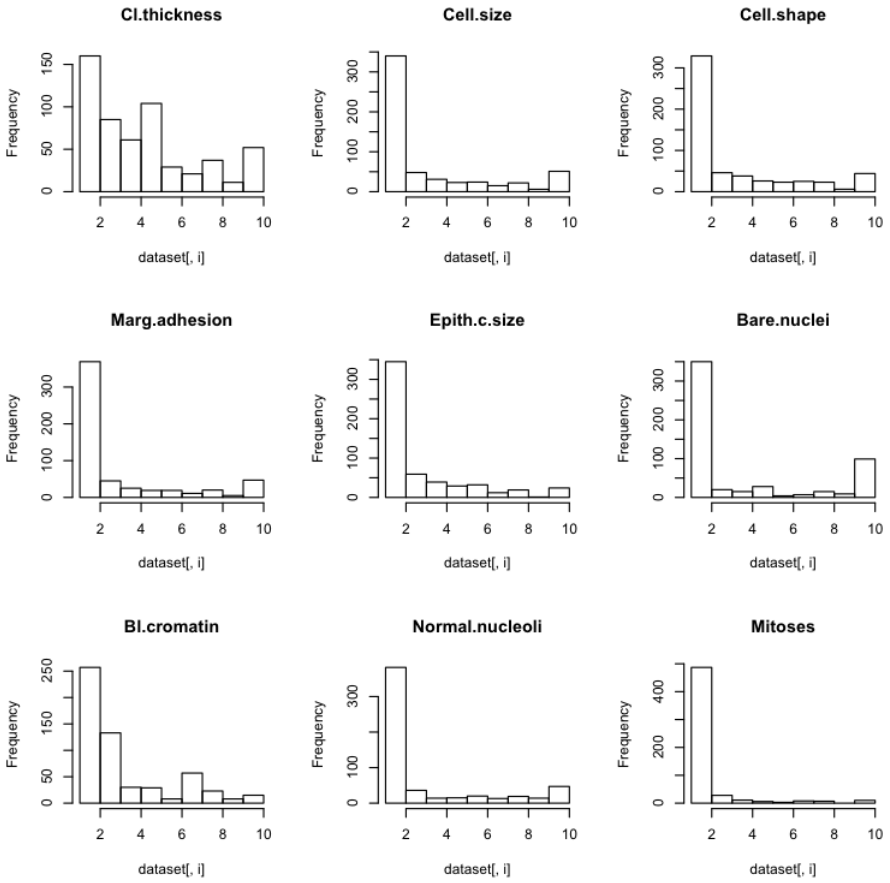

Figure 20.1: Histograms of Input Attributes for the Wisconsin Breast Cancer Dataset

Let's use density plots to get a more smoothed look at the distributions.

```
# density plot for each attribute
par(mfrow=c(3,3))
complete_cases <- complete.cases(dataset)
for(i in 1:9) {
  plot(density(dataset[complete_cases,i]), main=names(dataset)[i])
}
```

Listing 20.17: Calculate density plots.

These plots add more support to our initial ideas. We can see bimodal distributions (two bumps) and exponential-looking distributions.

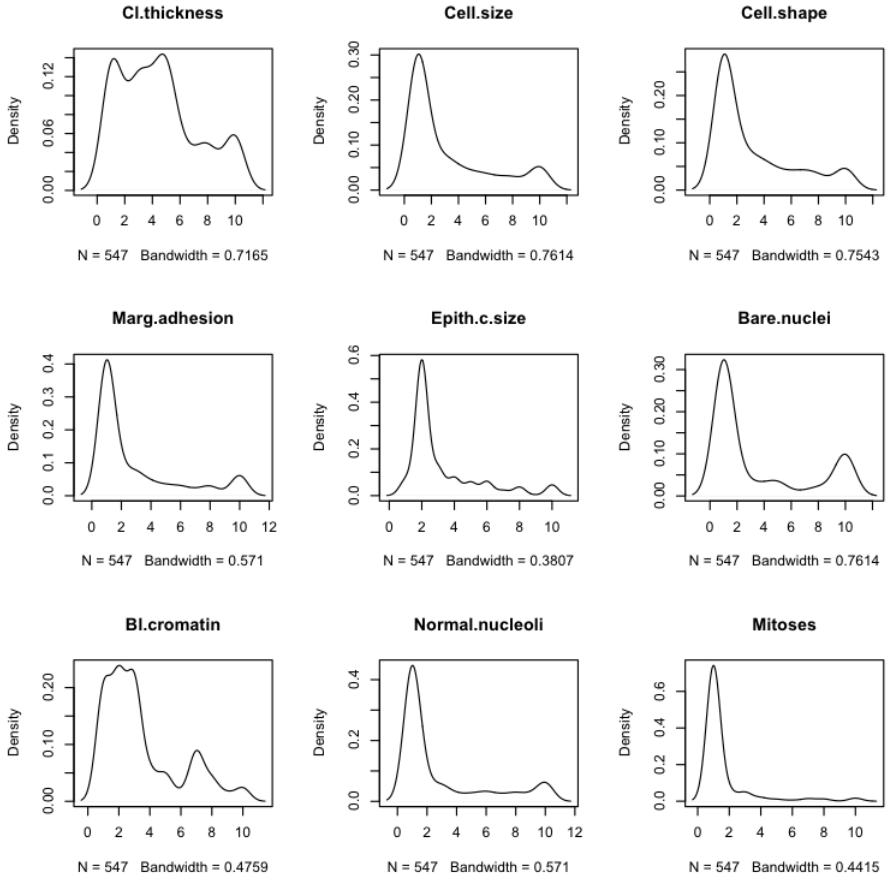

Figure 20.2: Density Plots of Input Attributes for the Wisconsin Breast Cancer Dataset

Let's take a look at the distributions from another perspective using box and whisker plots.

```
# boxplots for each attribute
par(mfrow=c(3,3))
for(i in 1:9) {
  boxplot(dataset[,i], main=names(dataset)[i])
}
```

Listing 20.18: Calculate box and whisker plots.

We see squashed distributions given the exponential shapes we've already observed. Also, because the attributes are scorings of some kind, the scale is limited to `[1,10]` for all inputs.

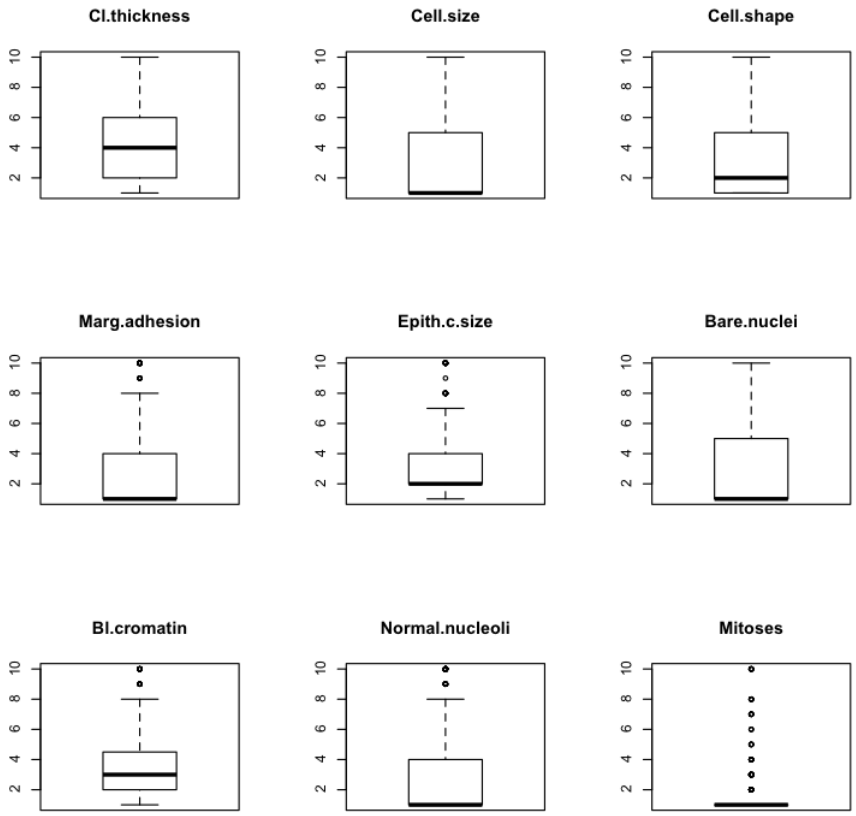

Figure 20.3: Box and Whisker Plots of Input Attributes for the Wisconsin Breast Cancer Dataset

20.2.3 Multimodal Data Visualizations

Now, let's take a look at the interactions between the attributes. Let's start with a scatter plot matrix of the attributes colored by the class values. Because the data is discrete (integer values) we need to add some jitter to make the scatter plots useful, otherwise the dots will all be on top of each other.

```
# scatter plot matrix
jittered_x <- sapply(dataset[,1:9], jitter)
pairs(jittered_x, names(dataset[,1:9]), col=dataset$Class)
```

Listing 20.19: Calculate scatter plot matrix.

We can see that the black (benign) a part to be clustered around the bottom-right corner (smaller values) and red (malignant) are all over the place.

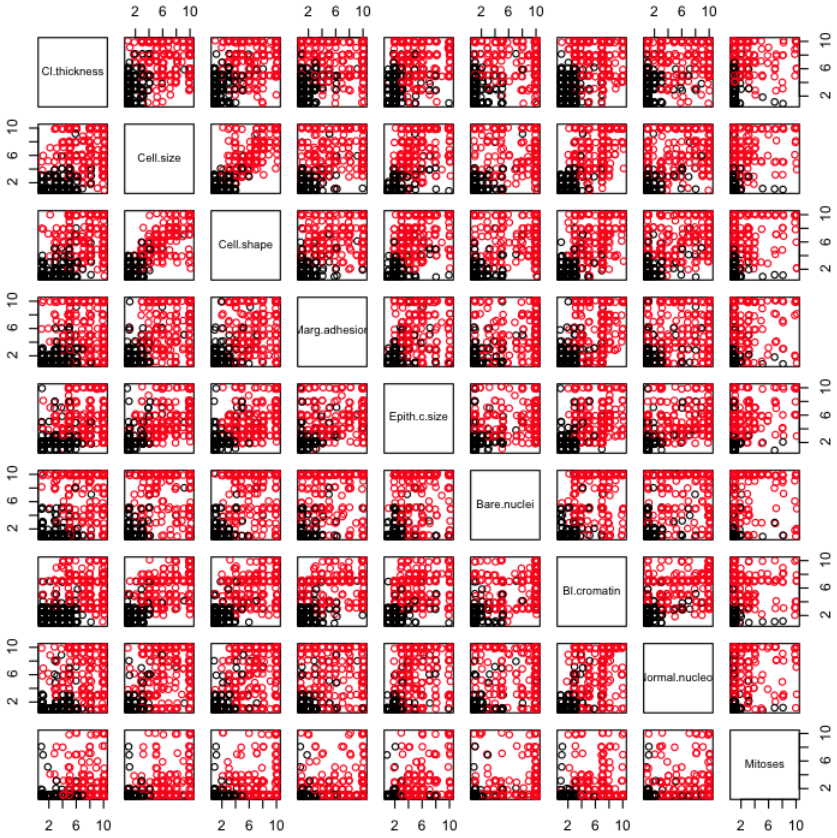

Figure 20.4: Scatterplot Matrix Plots of Input Attributes for the Wisconsin Breast Cancer Dataset

Because the data is discrete, we can use bar plots to get an idea of the interaction of the distribution of each attribute and how they breakdown by class value.

```
# bar plots of each variable by class
par(mfrow=c(3,3))
for(i in 1:9) {
 barplot(table(dataset$Class,dataset[,i]), main=names(dataset)[i],
     legend.text=unique(dataset$Class))
}
```

Listing 20.20: Calculate bar plots.

This gives us a more nuanced idea of how the benign values clustered at the left (smaller values) of each distribution and malignant values are all over the place.

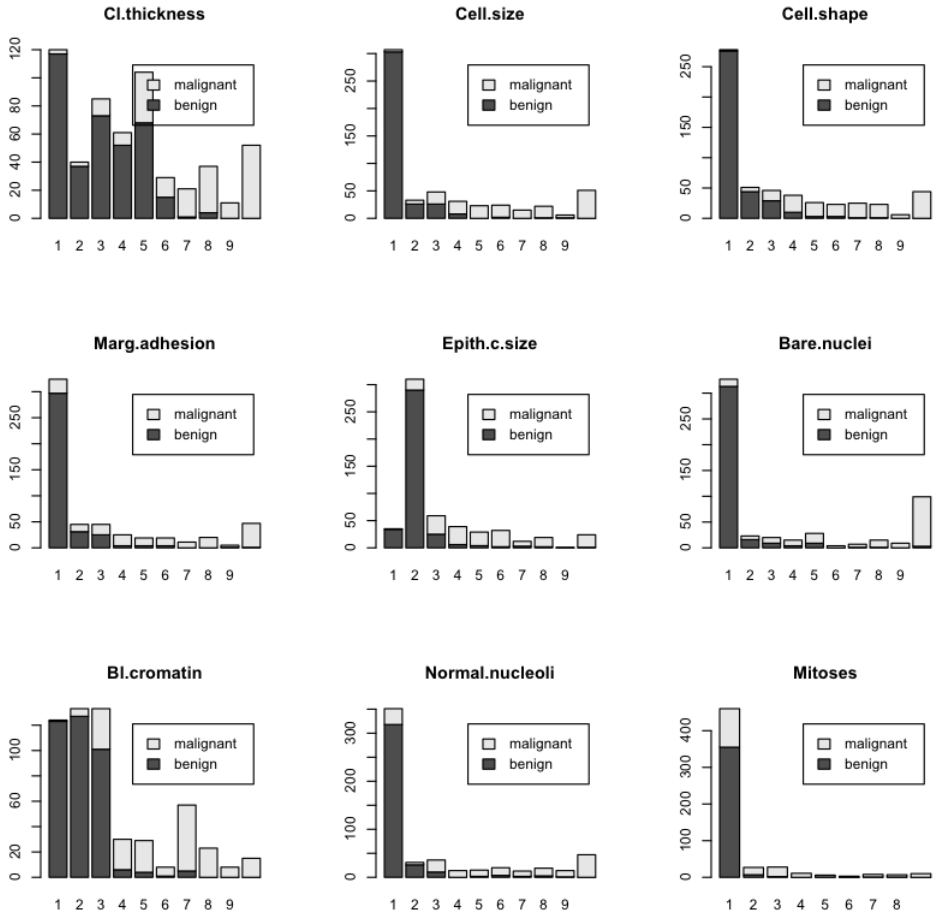

Figure 20.5: Bar Plots by Class Value of Input Attributes for the Wisconsin Breast Cancer Dataset

20.3 Evaluate Algorithms: Baseline

We don't know what algorithms will perform well on this data beforehand. We have to spot-check various different methods and see what looks good then double down on those methods. Given that the data is discrete, I would generally expect decision tree and rule-based methods to do well. I would expect regression and instance-based methods to not do so well. This is just intuition, and it could very well be wrong. Let's try a smattering of linear and nonlinear algorithms:

- **Linear Algorithms**: Logistic Regression (LG), Linear Discriminate Analysis (LDA) and Regularized Logistic Regression (GLMNET).
- **Nonlinear Algorithms**: k-Nearest Neighbors (KNN), Classification and Regression Trees (CART), Naive Bayes (NB) and Support Vector Machines with Radial Basis Functions (SVM).

Let's start off by defining the test harness. We have a good amount of data so we will use 10-fold cross validation with 3 repeats. This is a good standard test harness configuration. It is

a binary classification problem. For simplicity, we will use Accuracy and Kappa metrics. Given that it is a medical test, we could have gone with the Area Under ROC Curve (AUC) and looked at the sensitivity and specificity to select the best algorithms.

```
# 10-fold cross validation with 3 repeats
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "Accuracy"
```

Listing 20.21: Prepare algorithm evaluation test harness.

Let's create our models. We will use the default parameters for each algorithm for now. Algorithm tuning is considered later. We need to reset the random number seed before training each algorithm to ensure that each algorithm is evaluated on exactly the same splits of data. This will make later comparisons simpler (e.g. apples to apples).

```
# LG
set.seed(7)
fit.glm <- train(Class~., data=dataset, method="glm", metric=metric, trControl=trainControl)
# LDA
set.seed(7)
fit.lda <- train(Class~., data=dataset, method="lda", metric=metric, trControl=trainControl)
# GLMNET
set.seed(7)
fit.glmnet <- train(Class~., data=dataset, method="glmnet", metric=metric,
    trControl=trainControl)
# KNN
set.seed(7)
fit.knn <- train(Class~., data=dataset, method="knn", metric=metric, trControl=trainControl)
# CART
set.seed(7)
fit.cart <- train(Class~., data=dataset, method="rpart", metric=metric,
    trControl=trainControl)
# Naive Bayes
set.seed(7)
fit.nb <- train(Class~., data=dataset, method="nb", metric=metric, trControl=trainControl)
# SVM
set.seed(7)
fit.svm <- train(Class~., data=dataset, method="svmRadial", metric=metric,
    trControl=trainControl)
# Compare algorithms
results <- resamples(list(LG=fit.glm, LDA=fit.lda, GLMNET=fit.glmnet, KNN=fit.knn,
    CART=fit.cart, NB=fit.nb, SVM=fit.svm))
summary(results)
dotplot(results)
```

Listing 20.22: Estimate the accuracy of a suite of machine learning algorithms.

We can see good accuracy across the board. All algorithms have a mean accuracy above 90%, well above the baseline of 65% if we just predicted benign. The problem is learnable. We can see that KNN (97.08%) and logistic regression (LG was 96.35% and GLMNET was 96.47%) had the highest accuracy on the problem.

```
Models: LG, LDA, GLMNET, KNN, CART, NB, SVM
Number of resamples: 30

Accuracy
```

```
        Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
LG     0.9091 0.9455 0.9636 0.9635 0.9815 1.0000 0
LDA    0.9091 0.9447 0.9633 0.9592 0.9815 0.9821 0
GLMNET 0.9091 0.9455 0.9640 0.9647 0.9815 1.0000 0
KNN    0.9273 0.9630 0.9815 0.9708 0.9818 1.0000 0
CART   0.8571 0.9259 0.9444 0.9349 0.9455 0.9818 0
NB     0.9259 0.9444 0.9633 0.9616 0.9818 1.0000 0
SVM    0.9074 0.9325 0.9630 0.9519 0.9636 0.9818 0

Kappa
        Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
LG     0.8014 0.8786 0.9206 0.9192 0.9599 1.0000 0
LDA    0.7989 0.8752 0.9175 0.9089 0.9599 0.9607 0
GLMNET 0.8014 0.8786 0.9219 0.9219 0.9599 1.0000 0
KNN    0.8431 0.9175 0.9589 0.9356 0.9603 1.0000 0
CART   0.6957 0.8346 0.8758 0.8573 0.8796 0.9603 0
NB     0.8336 0.8802 0.9207 0.9159 0.9593 1.0000 0
SVM    0.8041 0.8544 0.9201 0.8969 0.9215 0.9611 0
```

Listing 20.23: Output estimated accuracy of algorithms.

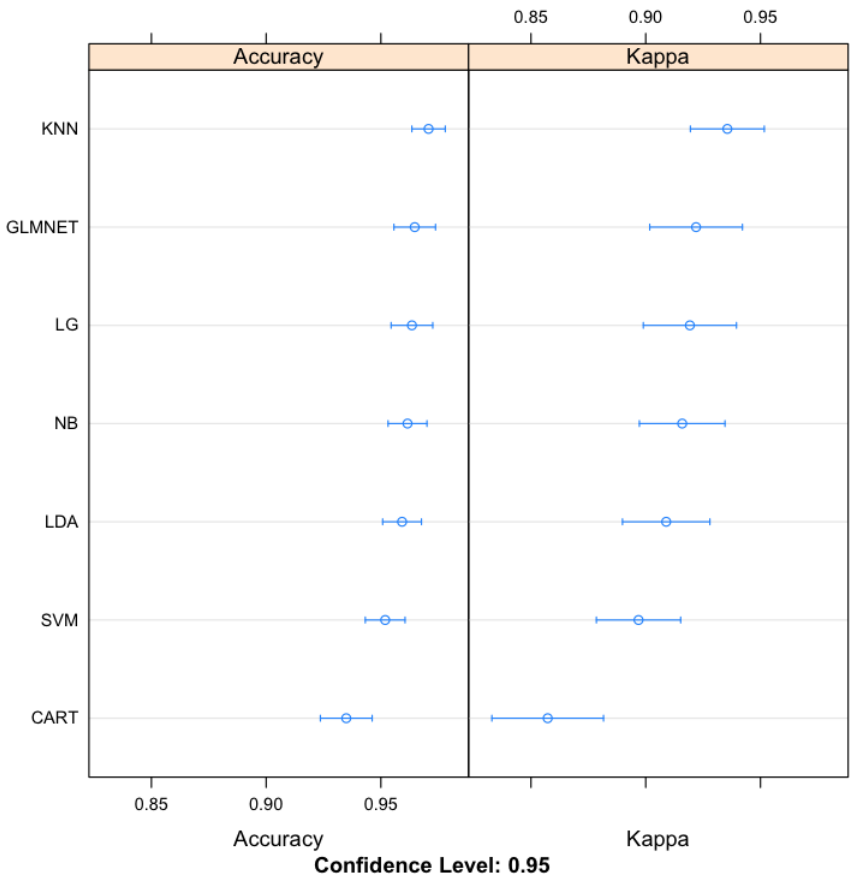

Figure 20.6: Baseline Accuracy on the Wisconsin Breast Cancer Dataset

20.4 Evaluate Algorithms: Transform

We know we have some skewed distributions. There are transform methods that we can use to adjust and normalize these distributions. A favorite for positive input attributes (which we have in this case) is the Box-Cox transform. In this section we evaluate the same 7 algorithms as above except this time the data is transformed using a Box-Cox power transform to flatten out the distributions.

```
# 10-fold cross validation with 3 repeats
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "Accuracy"
# LG
set.seed(7)
fit.glm <- train(Class~., data=dataset, method="glm", metric=metric, preProc=c("BoxCox"),
    trControl=trainControl)
# LDA
set.seed(7)
fit.lda <- train(Class~., data=dataset, method="lda", metric=metric, preProc=c("BoxCox"),
    trControl=trainControl)
# GLMNET
set.seed(7)
fit.glmnet <- train(Class~., data=dataset, method="glmnet", metric=metric,
    preProc=c("BoxCox"), trControl=trainControl)
# KNN
set.seed(7)
fit.knn <- train(Class~., data=dataset, method="knn", metric=metric, preProc=c("BoxCox"),
    trControl=trainControl)
# CART
set.seed(7)
fit.cart <- train(Class~., data=dataset, method="rpart", metric=metric,
    preProc=c("BoxCox"), trControl=trainControl)
# Naive Bayes
set.seed(7)
fit.nb <- train(Class~., data=dataset, method="nb", metric=metric, preProc=c("BoxCox"),
    trControl=trainControl)
# SVM
set.seed(7)
fit.svm <- train(Class~., data=dataset, method="svmRadial", metric=metric,
    preProc=c("BoxCox"), trControl=trainControl)
# Compare algorithms
transformResults <- resamples(list(LG=fit.glm, LDA=fit.lda, GLMNET=fit.glmnet, KNN=fit.knn,
    CART=fit.cart, NB=fit.nb, SVM=fit.svm))
summary(transformResults)
dotplot(transformResults)
```

Listing 20.24: Estimate accuracy on transformed dataset.

We can see that the accuracy of the previous best algorithm KNN was elevated to 97.14%. We have a new ranking, showing SVM with the most accurate mean accuracy at 97.20%.

```
Models: LG, LDA, GLMNET, KNN, CART, NB, SVM
Number of resamples: 30

Accuracy
        Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
LG    0.9091  0.9498 0.9636 0.9629 0.9815 1.0000 0
```

```
LDA    0.9091  0.9457 0.9729 0.9684 0.9818 1.0000 0
GLMNET 0.9273  0.9630 0.9636 0.9690 0.9818 1.0000 0
KNN    0.9091  0.9636 0.9815 0.9714 0.9818 1.0000 0
CART   0.8571  0.9259 0.9444 0.9349 0.9455 0.9818 0
NB     0.9273  0.9630 0.9636 0.9690 0.9818 1.0000 0
SVM    0.9273  0.9630 0.9729 0.9720 0.9818 1.0000 0

Kappa
         Min. 1st Qu. Median Mean 3rd Qu.  Max. NA's
LG     0.8062  0.8889 0.9207 0.9184 0.9599 1.0000 0
LDA    0.8062  0.8839 0.9411 0.9317 0.9603 1.0000 0
GLMNET 0.8431  0.9190 0.9215 0.9324 0.9599 1.0000 0
KNN    0.8062  0.9199 0.9591 0.9380 0.9603 1.0000 0
CART   0.6957  0.8346 0.8758 0.8573 0.8796 0.9603 0
NB     0.8392  0.9190 0.9215 0.9324 0.9602 1.0000 0
SVM    0.8468  0.9207 0.9411 0.9397 0.9603 1.0000 0
```

Listing 20.25: Output estimated accuracy on transformed dataset.

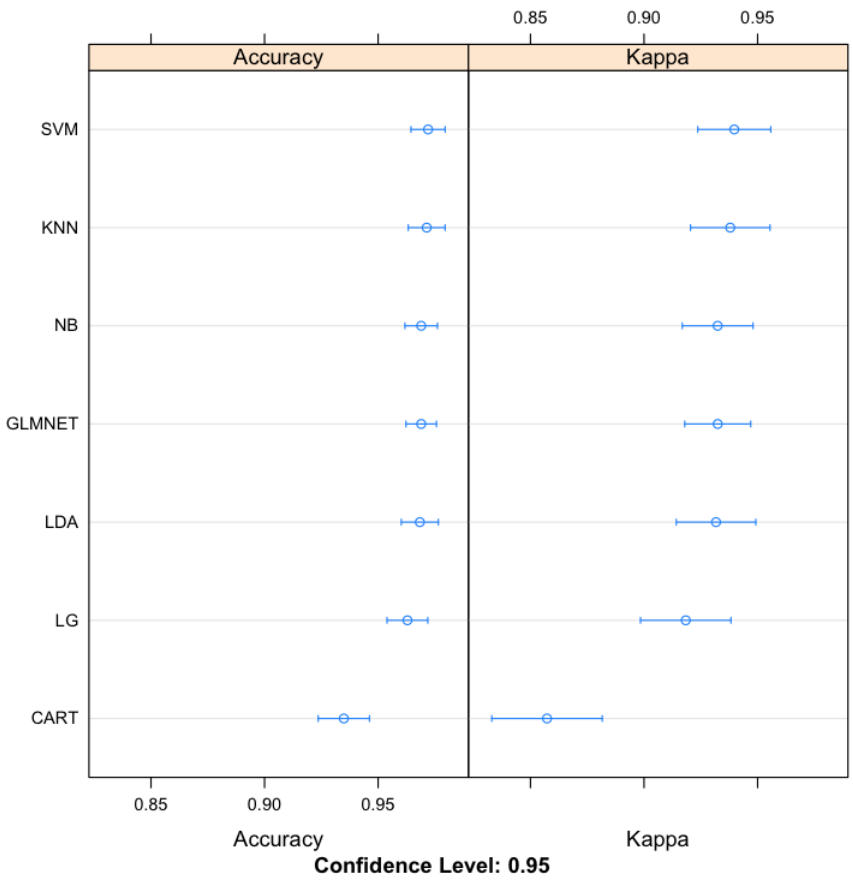

Figure 20.7: Accuracy With Data Transforms on the Wisconsin Breast Cancer Dataset

20.5 Algorithm Tuning

Let's try some tuning of the top algorithms, specifically SVM and see if we can lift the accuracy.

20.5.1 Tuning SVM

The SVM implementation has two parameters that we can tune with the `caret` package: `sigma` which is a smoothing term and C which is a cost constraint. You can learn more about these parameters in the help for the `ksvm()` function `?ksvm`. Let's try a range of values for C between 1 and 10 and a few small values for sigma around the default of 0.1.

```
# 10-fold cross validation with 3 repeats
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "Accuracy"
set.seed(7)
grid <- expand.grid(.sigma=c(0.025, 0.05, 0.1, 0.15), .C=seq(1, 10, by=1))
fit.svm <- train(Class~., data=dataset, method="svmRadial", metric=metric, tuneGrid=grid,
    preProc=c("BoxCox"), trControl=trainControl)
print(fit.svm)
plot(fit.svm)
```

Listing 20.26: Tune the parameters of the SVM algorithm.

We can see that we have made very little difference to the results. The most accurate model had a score of 97.19% (the same as our previously rounded score of 97.20%) using a `sigma = 0.15` and `C = 1`. We could tune further, but I don't expect a payoff.

```
Support Vector Machines with Radial Basis Function Kernel

560 samples
  9 predictor
  2 classes: 'benign', 'malignant'

Pre-processing: Box-Cox transformation (9)
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 492, 492, 493, 492, 493, 493, ...
Resampling results across tuning parameters:

  sigma  C   Accuracy  Kappa      Accuracy SD  Kappa SD
...
  0.100   7  0.9641278 0.9223610 0.02261232  0.04868379
  0.100   8  0.9622872 0.9181073 0.02180600  0.04732571
  0.100   9  0.9622872 0.9181073 0.02180600  0.04732571
  0.100  10  0.9622872 0.9181073 0.02180600  0.04732571
  0.150   1  0.9719958 0.9397291 0.02015647  0.04306497
  0.150   2  0.9707945 0.9370321 0.02003909  0.04288659
  0.150   3  0.9665408 0.9278046 0.02083330  0.04457734
  0.150   4  0.9647226 0.9236680 0.02128219  0.04578654
  0.150   5  0.9634993 0.9208189 0.02132057  0.04611720
  0.150   6  0.9629044 0.9191889 0.02255859  0.04930297
  0.150   7  0.9628932 0.9192086 0.02097758  0.04571537
  0.150   8  0.9622759 0.9177768 0.02014458  0.04394943
  0.150   9  0.9622872 0.9178560 0.01896851  0.04135621
  0.150  10  0.9616586 0.9164069 0.01925515  0.04194362

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were sigma = 0.15 and C = 1.
```

Listing 20.27: Output estimated accuracy of tuning SVM parameters.

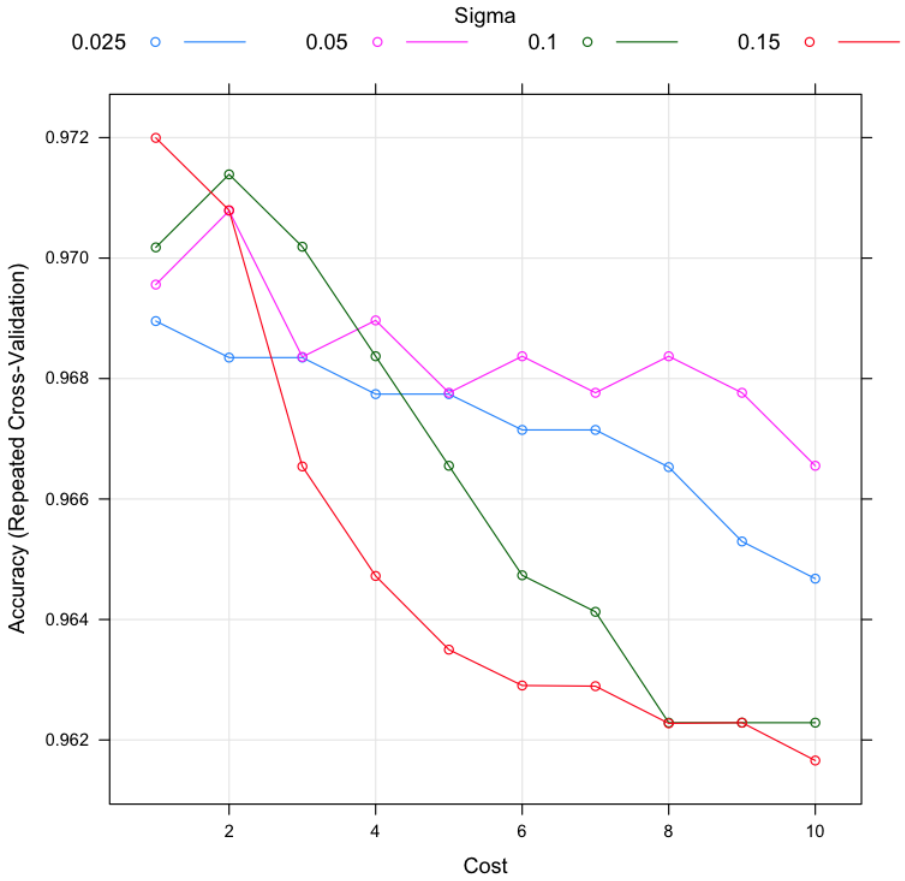

Figure 20.8: Tuning SVM parameters on the Wisconsin Breast Cancer Dataset

20.5.2 Tuning KNN

The KNN implementation has one parameter that we can tune with `caret`: k, the number of closest instances to collect in order to make a prediction. Let's try all k values between 1 and 20.

```
# 10-fold cross validation with 3 repeats
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "Accuracy"
set.seed(7)
grid <- expand.grid(.k=seq(1,20,by=1))
fit.knn <- train(Class~., data=dataset, method="knn", metric=metric, tuneGrid=grid,
    preProc=c("BoxCox"), trControl=trainControl)
print(fit.knn)
plot(fit.knn)
```

Listing 20.28: Tune the parameters of the KNN algorithm.

We can see again that tuning has made little difference, settling on a value of `k = 8` with an accuracy of 97.19%. This is higher than the previous 97.14%, but very similar (or perhaps identical!) to the result achieved by the tuned SVM.

```
k-Nearest Neighbors
```

```
560 samples
  9 predictor
  2 classes: 'benign', 'malignant'

Pre-processing: Box-Cox transformation (9)
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 492, 492, 493, 492, 493, 493, ...
Resampling results across tuning parameters:

  k   Accuracy  Kappa     Accuracy SD Kappa SD
   1  0.9524667 0.8953398 0.02651015  0.05883977
   2  0.9482023 0.8857023 0.02411603  0.05345626
   3  0.9665079 0.9271012 0.02397303  0.05188289
   4  0.9683482 0.9310991 0.02237821  0.04843864
   5  0.9707728 0.9365613 0.02324286  0.05022458
   6  0.9701331 0.9352947 0.02173624  0.04666792
   7  0.9713676 0.9379343 0.02127377  0.04572732
   8  0.9719962 0.9393206 0.02282626  0.04915446
   9  0.9713897 0.9379635 0.02173774  0.04684891
  10  0.9707945 0.9367099 0.02217898  0.04777854
  11  0.9714117 0.9380798 0.02273735  0.04901007
  12  0.9714117 0.9380798 0.02273735  0.04901007
  13  0.9701996 0.9354319 0.02263113  0.04877625
  14  0.9707949 0.9366855 0.02221537  0.04790061
  15  0.9701776 0.9353156 0.02162533  0.04659741
  16  0.9707836 0.9367691 0.02117843  0.04543484
  17  0.9707724 0.9366901 0.02006679  0.04296231
  18  0.9719849 0.9393670 0.02129015  0.04575184
  19  0.9719849 0.9393670 0.02129015  0.04575184
  20  0.9708057 0.9367228 0.02218466  0.04778476

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 8
```

Listing 20.29: Output estimated accuracy of tuning KNN parameters.

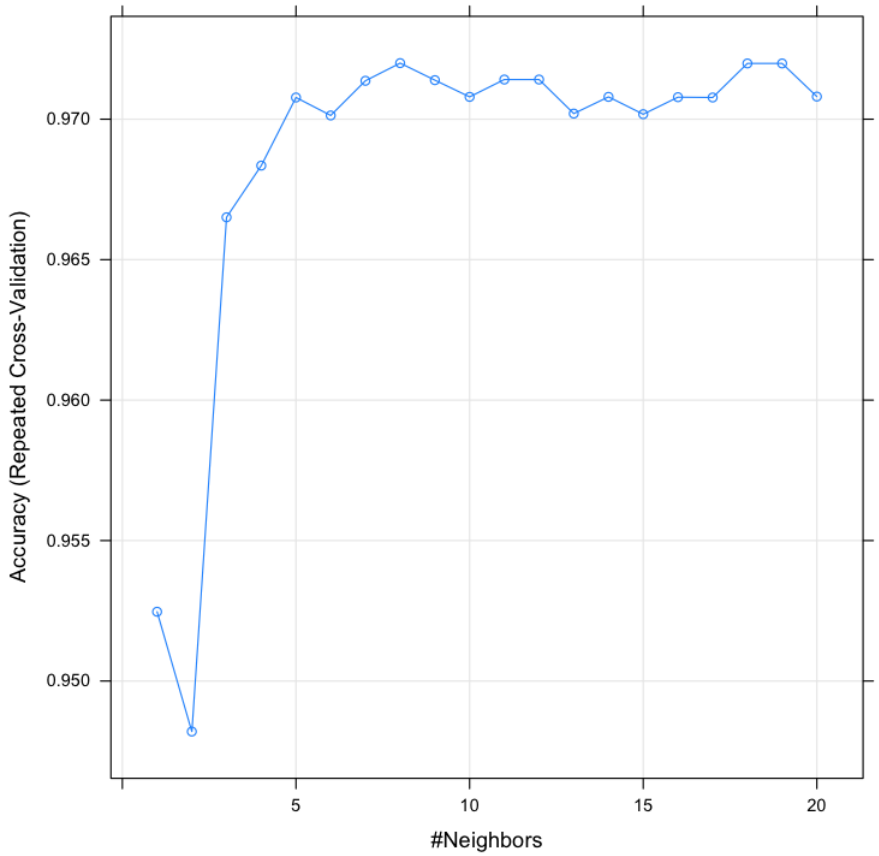

Figure 20.9: Tuning KNN parameters on the Wisconsin Breast Cancer Dataset

20.6 Ensemble Methods

As a final check, let's look at some boosting and bagging ensemble algorithms on the dataset. I expect them to do quite well given the decision trees that underlie these methods. If our guess about hitting the accuracy ceiling is true, we may also see these methods top out around 97.20%. Let's look at 4 ensemble methods:

- `Bagging`: Bagged CART (BAG) and Random Forest (RF).
- `Boosting`: Stochastic Gradient Boosting (GBM) and C5.0 (C50).

We will use the same test harness as before including the Box-Cox transform that flattens out the distributions.

```
# 10-fold cross validation with 3 repeats
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "Accuracy"
# Bagged CART
set.seed(7)
fit.treebag <- train(Class~., data=dataset, method="treebag", metric=metric,
    trControl=trainControl)
# Random Forest
```

```
set.seed(7)
fit.rf <- train(Class~., data=dataset, method="rf", metric=metric, preProc=c("BoxCox"),
    trControl=trainControl)
# Stochastic Gradient Boosting
set.seed(7)
fit.gbm <- train(Class~., data=dataset, method="gbm", metric=metric, preProc=c("BoxCox"),
    trControl=trainControl, verbose=FALSE)
# C5.0
set.seed(7)
fit.c50 <- train(Class~., data=dataset, method="C5.0", metric=metric, preProc=c("BoxCox"),
    trControl=trainControl)
# Compare results
ensembleResults <- resamples(list(BAG=fit.treebag, RF=fit.rf, GBM=fit.gbm, C50=fit.c50))
summary(ensembleResults)
dotplot(ensembleResults)
```

Listing 20.30: Estimate accuracy of ensemble methods.

We see that Random Forest was the most accurate with a score of 97.26%. Very similar to our tuned models above. We could spend time tuning the parameters of Random Forest (e.g. increasing the number of trees) and the other ensemble methods, but I don't expect to see better accuracy scores other than random statistical fluctuations.

```
Models: BAG, RF, GBM, C50
Number of resamples: 30

Accuracy
     Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
BAG 0.8909 0.9325 0.9633 0.9550 0.9815 0.9818 0
RF  0.9273 0.9631 0.9726 0.9726 0.9818 1.0000 0
GBM 0.9107 0.9447 0.9636 0.9629 0.9818 1.0000 0
C50 0.9273 0.9630 0.9636 0.9666 0.9815 1.0000 0

Kappa
     Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
BAG 0.7462 0.8499 0.9196 0.9005 0.9589 0.9611 0
RF  0.8431 0.9196 0.9407 0.9405 0.9603 1.0000 0
GBM 0.7989 0.8799 0.9215 0.9181 0.9599 1.0000 0
C50 0.8392 0.9190 0.9211 0.9268 0.9599 1.0000 0
```

Listing 20.31: Output estimated accuracy of ensemble methods.

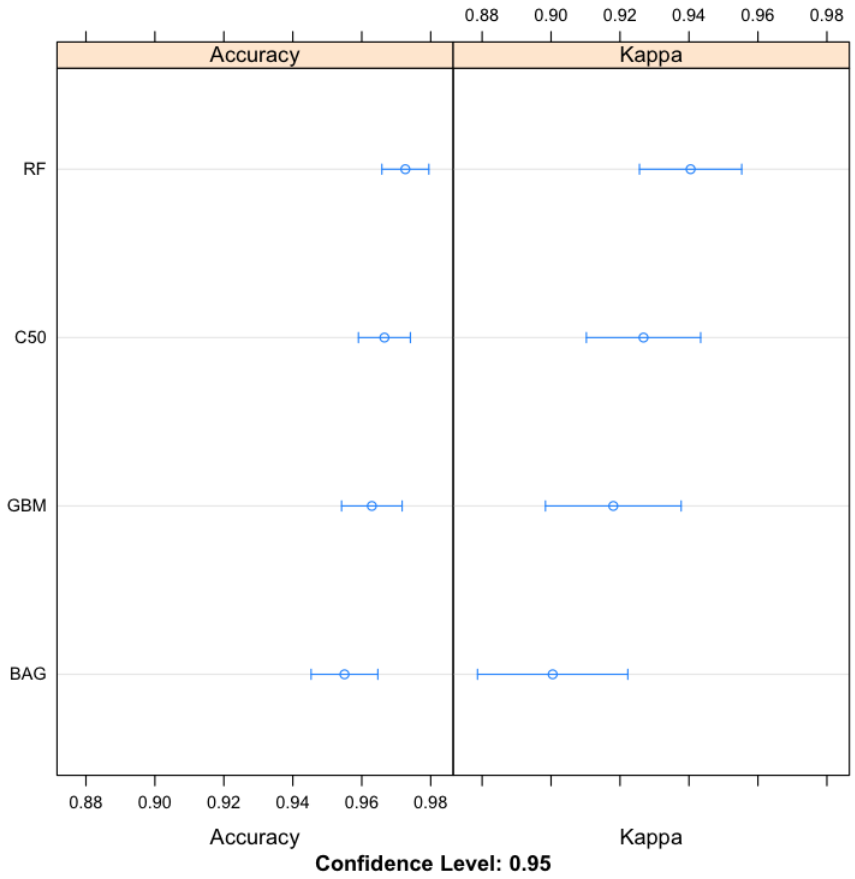

Figure 20.10: Bagging and Boosting Ensemble Methods on the Wisconsin Breast Cancer Dataset

20.7 Finalize Model

We now need to finalize the model, which really means choose which model we would like to use. For simplicity I would probably select the KNN method, at the expense of the memory required to store the training dataset. SVM would be a good choice to trade-off space and time complexity. I probably would not select the Random Forest algorithm given the complexity of the model. It seems overkill for this dataset, lots of trees with little benefit in Accuracy.

Let's go with the KNN algorithm. This is really simple, as we do not need to store a model. We do need to capture the parameters of the Box-Cox transform though. And we also need to prepare the data by removing the unused Id attribute and converting all of the inputs to numeric format. The implementation of KNN (`knn3()`) belongs to the `caret` package and does not support missing values. We will have to remove the rows with missing values from the training dataset as well as the validation dataset. The code below shows the preparation of the pre-processing parameters using the training dataset.

```
# prepare parameters for data transform
set.seed(7)
datasetNoMissing <- dataset[complete.cases(dataset),]
x <- datasetNoMissing[,1:9]
preprocessParams <- preProcess(x, method=c("BoxCox"))
x <- predict(preprocessParams, x)
```

Listing 20.32: Prepare the data transform for finalizing the model.

Next we need to prepare the validation dataset for making a prediction. We must:

1. Remove the `Id` attribute.
2. Remove those rows with missing data.
3. Convert all input attributes to numeric.
4. Apply the Box-Cox transform to the input attributes using parameters prepared on the training dataset.

```
# prepare the validation dataset
set.seed(7)
# remove id column
validation <- validation[,-1]
# remove missing values (not allowed in this implementation of knn)
validation <- validation[complete.cases(validation),]
# convert to numeric
for(i in 1:9) {
  validation[,i] <- as.numeric(as.character(validation[,i]))
}
# transform the validation dataset
validationX <- predict(preprocessParams, validation[,1:9])
```

Listing 20.33: Prepare the validation dataset for making new predictions.

Now we are ready to actually make a prediction in the training dataset.

```
# make predictions
set.seed(7)
predictions <- knn3Train(x, validationX, datasetNoMissing$Class, k=9, prob=FALSE)
confusionMatrix(predictions, validation$Class)
```

Listing 20.34: Estimate accuracy of KNN on the unseen validation dataset.

We can see that the accuracy of the final model on the validation dataset is 99.26%. This is optimistic because there is only 136 rows, but it does show that we have an accurate standalone model that we could use on other unclassified data.

```
Confusion Matrix and Statistics

          Reference
Prediction benign malignant
  benign       87         0
  malignant     1        48

               Accuracy : 0.9926
                 95% CI : (0.9597, 0.9998)
    No Information Rate : 0.6471
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.984
 Mcnemar's Test P-Value : 1
```

```
            Sensitivity : 0.9886
            Specificity : 1.0000
         Pos Pred Value : 1.0000
         Neg Pred Value : 0.9796
             Prevalence : 0.6471
         Detection Rate : 0.6397
   Detection Prevalence : 0.6397
      Balanced Accuracy : 0.9943

       'Positive' Class : benign
```

Listing 20.35: Output estimated accuracy of KNN on the unseen validation dataset.

20.8 Summary

In this lesson you have worked through a case study binary classification machine learning problem. We pulled together a number of lessons for machine learning tasks in R into an end-to-end project using the R platform. We covered the following steps:

1. Problem definition (predict the type of cancer from details of tissue samples).
2. Analyzed Data (noted the exponential and bimodal shapes of many of the attributes).
3. Evaluated Algorithms (baseline accuracy suggesting KNN is accurate on the problem).
4. Evaluate Algorithms with Transform (improved accuracy with a Box-Cox transform and also noted SVM does well).
5. Tuned Algorithms (small improvements to SVM and KNN, but nothing too exciting).
6. Ensemble Methods (hitting the prediction accuracy ceiling with the dataset).
7. Finalized Model (accurate on the validation dataset, perhaps optimistically so).

Working through this case study will give you confidence in the steps and techniques that you can use to in R to complete your own predictive modeling project.

20.8.1 Next Step

This was the third and final predictive modeling project case study. Well done! You now have experience and skills in working through predictive modeling machine learning projects end-to-end. In the next section you will discover ideas for additional small case study projects that you could work on for further practice.

Chapter 21

More Predictive Modeling Projects

You can now work through predictive modeling machine learning projects using R. Now what? In this chapter, we look at ways that you can practice and refine your new found skills.

21.1 Build And Maintain Recipes

Throughout this book you have worked through many machine learning lessons using R. Taken together, this is the start of your own private code base that you can use to jump-start your current or next machine learning project. These recipes are a beginning, not an end. The larger and more sophisticated that your catalog of machine learning recipes is, the faster you can get started on new projects and the more accurate the models that you can develope.

As you apply your machine learning skills using the R platform, you will develop experience and skills with new and different techniques with R. You can pull out or abstract snippets and recipes as you go along and add them to your own collection of recipes, building upon the code that you can use on future machine learning projects. With time, you will amass your own mature and highly-tailored catalog of machine learning code for R.

21.2 Small Projects on Small Datasets

Keep practicing your skills using R. Datasets from the UCI Machine Learning Repository were used throughout this book to demonstrate how to achieve specific tasks in a machine learning project. They were also used in the longer case study projects. They are standardized, relatively clean, well understood and excellent for you to use as practice datasets.

You can use the datasets on the UCI Machine Learning repository as the focus of small (5-to-10 hours of effort) focused machine learning projects using the R platform. Once completed, you can write-up your findings and share them online as part of your expanding portfolio of machine learning projects.

These can be used by you later as a repository of knowledge on which you can build and further develop your skills. They can also be used to demonstrate to bosses or future employers that you are capable of delivering results on predictive modeling machine learning projects using the R platform. Here is a process that you can use to practice machine learning on R:

1. Browse the list of free datasets on the repository and download some that look interesting to you.

2. Use the project template and recipes in this book to work through the dataset and develop an accurate model.

3. Write up your work-flow and findings in a way that you can refer to later or perhaps share it publicly on a website.

Keep the project short, limit your projects to 5-to-10 hours, say a week worth of nights and spare time.

21.3 Competitive Machine Learning

Use competitive machine learning to push your skills. Working on small projects in the previous section is a good way to practice the fundamentals. At some point the problems will become easy for you. You also need to be pushed out of your comfort zone to help you grow your skills further.

An excellent way to develop your machine learning skills with R further is to start participating in competitions. In a competition, the organizer provides you with a training dataset, a test dataset on which you are to make predictions, a performance measure and a time limit. You and your competitors then work to create the most accurate model possible. Winners often get prize money.

These competitions often last weeks to months and can be a lot of fun. They also offer a great opportunity to test your skills with machine learning tools on datasets that often require a lot of cleaning and preparation. The premier website for machine learning competitions is Kaggle: http://www.kaggle.com.

Competitions are stratified into different classes such as research, recruitment and 101 for beginners. A good place to start would be the beginner competitions as they are often less challenging and have a lot of help in the form of tutorials to get you started.

21.4 Summary

In this chapter you have discovered 3 areas where you could practice your new found machine learning skills with R. They were:

1. To continue to build up and maintain your catalog of machine learning recipes starting with the catalog of recipes provided as a bonus with this book.

2. To continue to work on the standard machine learning datasets described in Chapter 5 as well as the broader set of datasets on the UCI Machine Learning Repository.

3. To start work through the larger datasets from competitive machine learning and even start participating in machine learning competitions.

21.4.1 Next Step

This concludes Part III of this book on machine learning projects. Up next we finish off the book with a summary of how far you have come and where you can look if you need additional help with R.

Part IV

Conclusions

Chapter 22

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come.

1. You started off with an interest in machine learning and a strong desire to be able to practice and apply machine learning using R.

2. You downloaded, installed and started R, perhaps for the first time, and started to get familiar with the syntax of the language.

3. Slowly and steadily over the course of a number of lessons you learned how the standard tasks of a predictive modeling machine learning project map onto the R platform.

4. Building upon the recipes for common machine learning tasks you worked through your first machine learning problems end-to-end using R.

5. Using a standard template, the recipes and experience you have gathered, you are now capable of working through new and different predictive modeling machine learning problems on your own.

Don't make light of this - you have come a long way in a short amount of time. You have developed the important and valuable skill of being able to work through machine learning problems end-to-end using R. This is a platform that is used by a majority of data scientist professionals and some of the best data scientists on the planet. The sky is the limit for you.

I want to take a moment and sincerely thank you for letting me help you start your machine learning journey with R. I hope you keep learning and have fun as you continue to master machine learning.

Chapter 23

Getting More Help

This is just the beginning of your machine learning journey with R. As you start to work on your own machine learning projects you may need help. This chapter points out some of the best sources of R and machine learning help that you can find.

23.1 CRAN

The Comprehensive R Archive Network (or CRAN for short) provides a point of distribution for R packages that you can download.

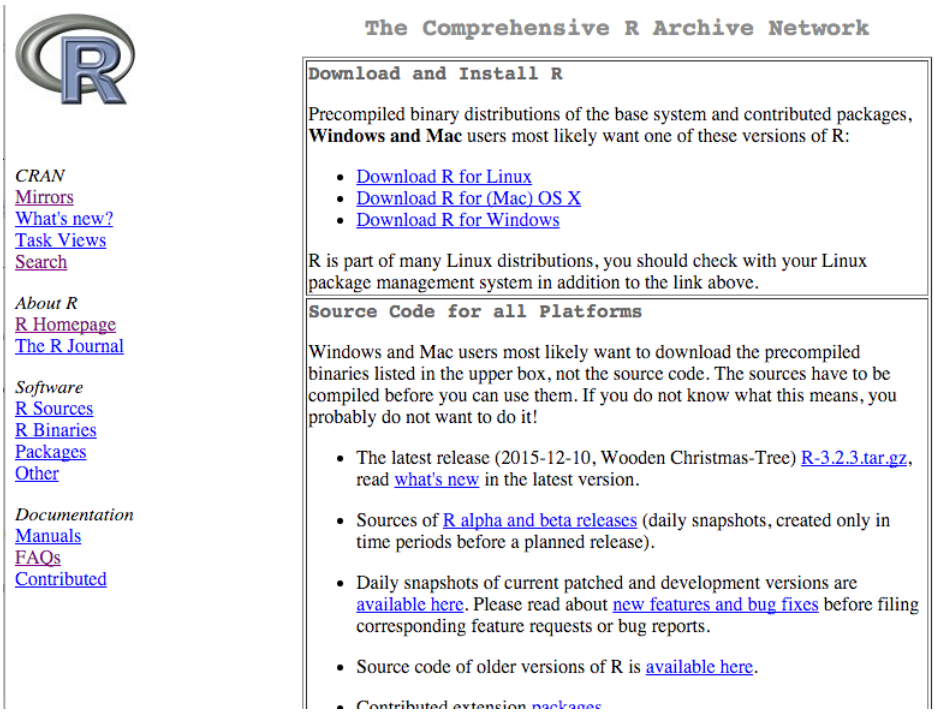

Figure 23.1: The Comprehensive R Archive Network Homepage

You can search CRAN for a package that provides a specific technique: https://cran.r-project.org/search.html.

You can also browse Machine Learning packages in the *Machine Learning & Statistical Learning* task view:
https://cran.r-project.org/web/views/MachineLearning.html.

Each package has a homepage that can give you a good summary. I find this a useful way to locate and view documentation for a package such as manuals and vignettes. This information is also available within R once you install the package. For example, here is the CRAN home page for the `caret` package:
https://cran.r-project.org/web/packages/caret/index.html

23.2 Q&A Websites

Question and answer sites are perhaps the best way to get answers to your specific technical questions about R. You can search them for similar questions, browse through topics to learn about solutions to common problems and ask your own technical questions.

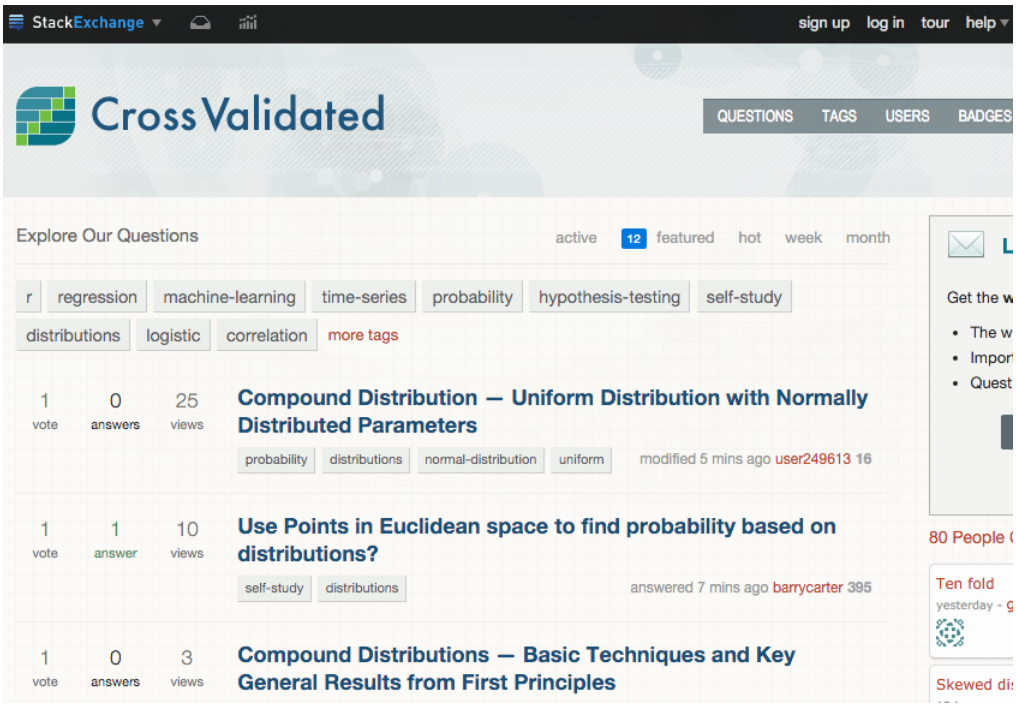

Figure 23.2: Cross Validated Q&A Homepage

The best Q&A sites I would recommend for your R Machine Learning questions are:

- Cross Validated: http://stats.stackexchange.com/
- Stack Overflow: http://stackoverflow.com/questions/tagged/r
- Data Science: http://datascience.stackexchange.com/questions/tagged/r

Make heavy use of the search feature on these sites. Also note the list of *Related* questions in the right-hand navigation bar when viewing a specific question. These are often relevant and useful.

23.3 Mailing Lists

A well established and perhaps slightly out-dated way of getting help would be to use an R email list. There are a number of R mailing lists available:
https://www.r-project.org/mail.html

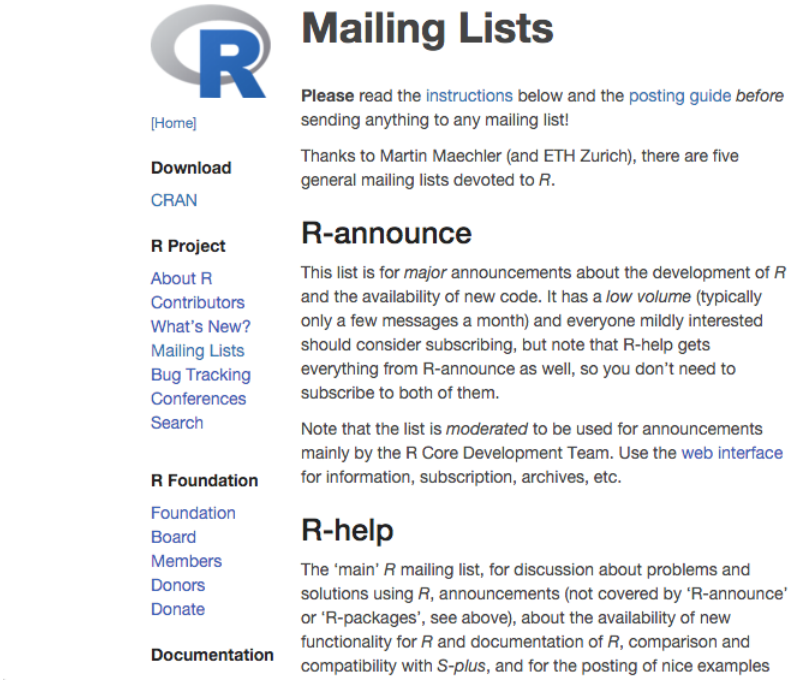

Figure 23.3: R Help Mailing Lists Homepage

The most appropriate list for getting help to technical questions is the R-help mailing list: https://stat.ethz.ch/mailman/listinfo/r-help.

Users on the list are known to be a little prickly, so read the posting guide well before emailing the list:
https://www.r-project.org/posting-guide.html.

My best advice is to search through the web archive of the mailing list to see if someone has asked or answered a similar question to yours in the past:
http://tolstoy.newcastle.edu.au/R/.

23.4 Package Websites

Many R packages have their own website that provide even more information about the package. For example, the `caret` package has its own website:
http://topepo.github.io/caret/index.html.

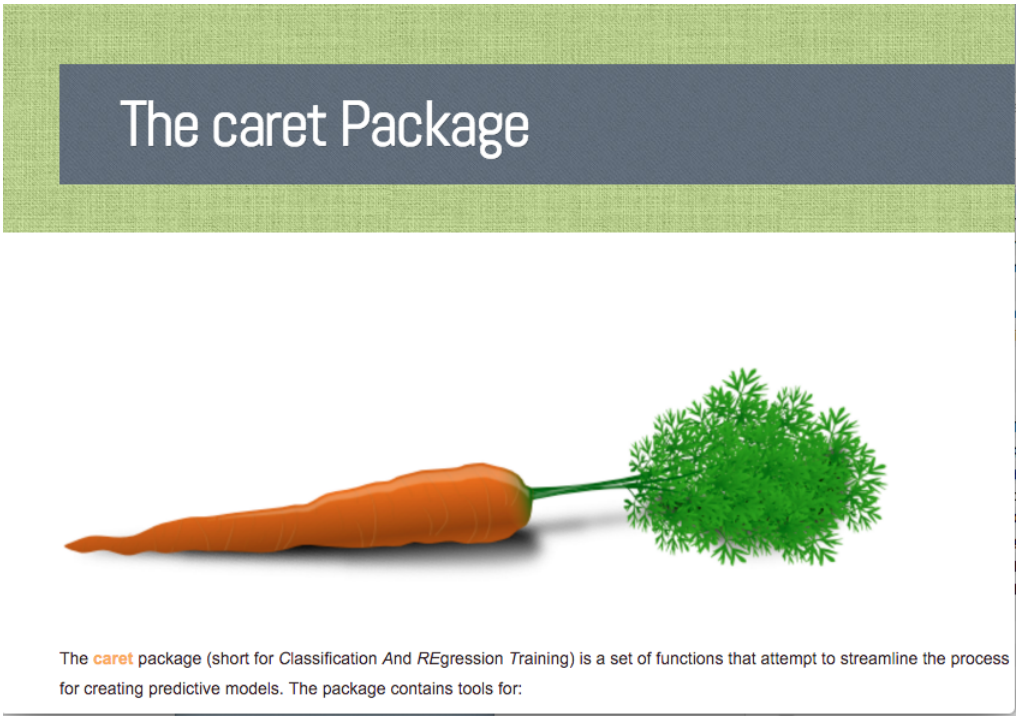

Figure 23.4: R Caret Package Homepage

The `caret` package software project is also hosted on GitHub which provides direct access to the source code and an issue tracking system where you can report bugs: https://github.com/topepo/caret.

23.5 Books

This book contains everything that you need to get started and complete predictive modeling machine learning projects in R, but if you are like me, then you love books. This section lists some additional books on R and machine learning that I recommend.

- **R in a Nutshell**. An excellent reference book to the R platform that even covers some machine learning algorithms.
  http://www.amazon.com/dp/144931208X?tag=inspiredalgor-20
- **Applied Predictive Modeling**. An excellent introduction to predictive modeling, co-written by the author of the `caret` package in R.
  http://www.amazon.com/dp/1461468485?tag=inspiredalgor-20
- **An Introduction to Statistical Learning**. A more theoretical introduction to the field with examples in R.
  http://www.amazon.com/dp/1461471397?tag=inspiredalgor-20
- **Data Science with R**. An introduction to data science using the R platform.
  http://www.amazon.com/dp/1617291560?tag=inspiredalgor-20

- **Machine Learning with R**. Introduction to machine learning using the R platform with a strong focus on algorithms.
  http://www.amazon.com/dp/1782162143?tag=inspiredalgor-20