



Machine Learning:

An Introductory Handbook Using R

Dr. Karen Mazidi



Copyright © 2018 Karen Mazidi

WWW.KARENMAZIDI.COM

This work is released under Creative Commons License CC BY-NC-ND 4.0

You are free to use this work for your personal educational purposes. You may not redistribute any portion of this work without prior permission of the author.

FIRST EDITION, JULY 2018

Created with a LaTeX template by Mathias Legrand <http://www.LaTeXTemplates.com>

Contents

I Part One: Introduction to Machine Learning and R

1	The Craft of Machine Learning	17
1.1	Machine Learning	18
1.1.1	Data	18
1.1.2	Patterns	19
1.1.3	Predictions from Data	19
1.1.4	Accuracy	19
1.1.5	Actions	19
1.2	Machine Learning Scenarios	20
1.2.1	Informative v. Active	20
1.2.2	Supervised v. unsupervised learning	20
1.2.3	Regression v. classification	20
1.3	Machine learning v. traditional algorithms	20
1.4	Terminology	21
1.5	Notation	22
2	Learning R	23
2.1	Installing R, RStudio	23
2.1.1	Getting Started in R	24
2.2	R Data	25
2.2.1	Vector	26
2.2.2	Lists	27

2.2.3	Matrix	28
2.2.4	Arrays	28
2.2.5	Data Frames	28
2.3	Data Exploration	29
2.4	Visual Data Exploration	30
2.5	Factors	32
2.5.1	Adding a Factor Column to a Data Frame	33
2.6	R Notebooks	34
2.7	Control structures	36
2.7.1	if-else	37
2.7.2	Defining and Calling Functions	37
2.7.3	Loops	39
2.8	R Style	40
2.9	Summary	41

II

Part Two: Linear Models

3	Linear Regression	47
3.1	Overview	47
3.2	Linear Regression in R	47
3.3	Metrics	49
3.3.1	Metrics for Data Analysis	50
3.3.2	Linear Regression Metrics for Model Fit	50
3.3.3	Metrics for Test Set Evaluation	52
3.4	The Algorithm	54
3.5	Mathematical Foundations	55
3.5.1	Gradient descent	58
3.6	Multiple Linear Regression	58
3.6.1	Interpreting summary() output for a linear model	60
3.6.2	Residuals	61
3.6.3	Dummy variables	63
3.6.4	The anova0 function	63
3.7	Polynomial Linear Regression	64
3.8	Model Fitting and Assumptions	66
3.8.1	Overfitting v. underfitting	66
3.8.2	Bias and variance	66
3.8.3	Linear Model Assumptions	67
3.9	Advanced Topic: Regularization	67
3.10	Summary	69
3.10.1	Terminology	69
3.10.2	Quick Reference	70

3.10.3	Lab	70
3.10.4	Exploring Concepts	71
3.10.5	Going Further	71
4	Logistic Regression	73
4.1	Overview	73
4.2	Logistic Regression in R	74
4.2.1	Plotting factor data	74
4.2.2	Train and Test on the Plasma Data	75
4.2.3	Interpreting summary() in Logistical Regression	76
4.2.4	Evaluation on the Test Data	77
4.2.5	Learning (or Not) From Data	77
4.3	Metrics	78
4.3.1	Accuracy, sensitivity and specificity	78
4.3.2	Kappa	79
4.3.3	ROC Curves and AUC	80
4.3.4	Probability, odds, and log odds	80
4.4	The Algorithm	82
4.5	Mathematical Foundations	83
4.6	Logistic Regression with Multiple Predictors	85
4.7	Advanced Topic: Optimization Methods	87
4.7.1	Gradient Descent	88
4.7.2	Stochastic Gradient Descent	88
4.7.3	Newton's Method	88
4.7.4	Optimization from Scratch	89
4.8	Multiclass Classification	91
4.9	Generalized Linear Models	93
4.10	Summary	93
4.10.1	New Terminology in this Chapter	94
4.10.2	Quick Reference	94
4.10.3	Lab	95
4.10.4	Exploring Concepts	95
4.10.5	Going Further	96
5	Naive Bayes	97
5.1	Overview	97
5.2	Naive Bayes in R	97
5.3	Probability Foundations	100
5.4	Probability Distributions	101
5.4.1	Bernouli, Binomial, and Beta Distributions	101
5.4.2	Multinomial and Dirichlet Distributions	103
5.4.3	Gaussian	105

5.5	The Algorithm	106
5.6	Applying the Bayes Theorem	107
5.7	Handling Text Data	108
5.8	Naive Bayes v. Logistic Regression	108
5.8.1	Compare to Logistic Regression	108
5.8.2	Balanced or Unbalanced Data	108
5.8.3	Accuracy, Sensitivity and Specificity	109
5.8.4	Naive Bayes Model	110
5.8.5	Generative v. Discriminative Classifiers	112
5.9	Naive Bayes from Scratch	112
5.9.1	Probability Tables	112
5.9.2	Conditional Probability for Discrete Data	113
5.9.3	Likelihood for Continuous Data	114
5.9.4	Putting it All Together	114
5.9.5	Results	115
5.10	Summary	115
5.10.1	New Terminology in this Chapter	116
5.10.2	Quick Reference	116
5.10.3	Labs	116
5.10.4	Exploring Concepts	117
5.10.5	Going Further	117
6	Support Vector Machines	119
6.1	Overview	120
6.2	SVM in R	120
6.3	The Algorithm	121
6.4	Mathematical Foundations	122
6.4.1	Slack Variables and the C Hyperparameter	123
6.5	Kernel Methods and the Gamma Hyperparameter	124
6.6	SVM Regression	125
6.6.1	Radial Kernel SVM	127
6.7	Summary	128
6.7.1	New Terminology in this Chapter	128
6.7.2	Quick Reference	128
6.7.3	Labs	129
6.7.4	Going Further	129
7	The Craft 2: Data Wrangling	131
7.1	Data Organization	131
7.1.1	Reproducible Research	131

7.2	Data Standardization	132
7.2.1	Dealing with NAs	132
7.2.2	Outliers	133
7.3	Time Data	133
7.3.1	Lubridate	133
7.4	Text Data	134
7.4.1	Text Mining Packages tm	134
7.4.2	RTextTools	135

III

Part Three: Searching for Similarity

8	Instance-Based Learning with kNN	141
8.1	Overview	141
8.2	kNN in R	142
8.3	The Algorithm	143
8.3.1	Choosing K	143
8.3.2	Curse of Dimensionality	144
8.4	Mathematical Foundations	144
8.4.1	Scaling Data	144
8.5	kNN Regression	145
8.6	Find the Best K	146
8.7	k-fold Cross Validation	147
8.7.1	Creating k Folds	148
8.7.2	Run knnreg() on each Fold	149
8.7.3	Cross Validation with Various K	149
8.7.4	Applying CV to Other Algorithms	150
8.8	Summary	151
8.8.1	Quick Reference	152
8.8.2	Lab	152
8.8.3	Exploring Concepts	153
8.8.4	Going Further	153
9	Clustering	155
9.1	Overview	156
9.2	Clustering in R with k-Means	156
9.3	Metrics	158
9.4	Algorithmic Foundations of k-means	158
9.5	Finding k	159
9.5.1	NbClust	161

9.6	Hierarchical Clustering	161
9.6.1	The Algorithm	162
9.6.2	Cutting the Dendogram	163
9.6.3	Advantages and Disadvantages of Hierarchical Clustering	164
9.7	Summary	164
9.7.1	Quick Reference	164
9.7.2	Lab	165
9.7.3	Exploring Concepts	165
9.7.4	Going Further	165
10	Decision Trees and Random Forests	167
10.1	Overview	167
10.2	Decision Trees in R	168
10.3	The Algorithm	169
10.3.1	Regression	169
10.3.2	Classification	169
10.3.3	Qualitative data	170
10.4	Mathematical Foundations	170
10.4.1	Entropy	170
10.4.2	Information Gain	171
10.4.3	Gini index	172
10.5	Tree Pruning	172
10.6	Random Forests	174
10.7	Cross validation, Bagging, Random Forests in R	174
10.8	Summary	175
10.8.1	Quick Reference	175
10.8.2	Lab	176
10.8.3	Exploring Concepts	177
10.8.4	Going Further	177
11	The Craft 3: Feature Engineering	179
11.1	Feature Selection	179
11.1.1	Feature Selection with caret	179
11.1.2	Ranking Feature Importance	180
11.1.3	Recursive Feature Selection	181
11.2	Feature Engineering	182
11.2.1	Principal Components Analysis	182
11.2.2	Linear Discriminant Analysis	183

12	Neural Networks	189
12.1	Neural Network Regression Example	190
12.1.1	Results	191
12.1.2	Hidden Nodes and Layers	192
12.2	The Algorithm	192
12.3	Mathematical Foundations	193
12.4	Neural Network Classification Example	194
12.4.1	Preprocessing for Classification	194
12.5	Neural Network Architecture	196
12.5.1	Training	196
12.5.2	The Backpropagation Algorithm	197
12.5.3	Model Output	197
12.5.4	Visualizing Results	198
12.6	Summary	198
12.6.1	New Terminology	198
12.6.2	Quick Reference	198
12.6.3	Going Further	199
13	Deep Learning	201
13.1	Keras	201
13.1.1	Installing Keras and Tensorflow	201
13.2	Data Representation in TensorFlow	202
13.3	Keras in R Example	202
13.3.1	Data Preparation	203
13.3.2	Build the Model	203
13.3.3	Train and Evaluate	204
13.4	Deep Learning Basics	204
13.4.1	Layers and Units	205
13.4.2	Activation Functions	206
13.4.3	Optimization Schemes and Loss Functions	206
13.5	Bring Your Own Data	206
13.6	Deep Architecture	209
13.6.1	Convolutional Neural Networks	209
13.6.2	Recurrent Neural Networks	209
13.7	Summary	210
13.7.1	Going Further	210
14	The Craft 4: Never Stop Learning	211
14.1	Machine Learning Trends	212

14.2	Inside the Black Box	212
------	----------------------	-----

V

Part Five: Modeling the World

15	Bayes Nets	217
15.1	Bayes Nets in R	218
15.1.1	Querying the Network	219
15.2	Bayesian Net Semantics	219
15.2.1	Review of Graph Terminology	219
15.2.2	Graph Structure	219
15.2.3	Reasoning with Graphs	220
15.3	The Algorithm	220
15.4	Example: Coronary Data	221
15.5	Summary	221
15.5.1	Going Further	222
16	Markov Models	223
16.1	Overview	223
16.2	Markov Model	223
16.3	Hidden Markov Model	225
16.4	HMM in R	225
16.5	Metrics	227
16.6	The Algorithm	227
16.7	Another HMM in R	227
16.8	Summary	229
16.8.1	Going Further	230
17	Reinforcement Learning	231
17.1	Overview	231
17.2	The Markov Decision Process	232
17.3	MDPtoolbox	233
17.4	Reinforcement Learning	234
17.5	The ReinforcementLearning Package	234
17.6	Summary	235
17.6.1	Going Further	235
18	The Craft 5: Algorithms	237
18.1	The Five Tribes	237
18.2	Choosing an Algorithm	238

19	Modern R	241
19.1	Package dplyr	242
19.1.1	Tibble	242
19.1.2	Function glimpse()	242
19.1.3	Select	243
19.1.4	Mutate	243
19.1.5	Filter	244
19.1.6	Arrange	244
19.1.7	Summarize	244
19.1.8	Pipes	244
19.1.9	Going Further with dplyr	245
19.2	Package ggplot2	245
19.2.1	Improving Appearance	246
19.2.2	Facet Grid	246
19.2.3	Histogram	247
19.2.4	Boxplot and Rug	247
19.2.5	Density Plot	248
19.2.6	Bubble Chart	248
19.2.7	Grid	249
19.2.8	Going Further with Graphs	249
19.3	Shiny	249
20	Big Data with R	253
20.1	Memory, Data and R	253
20.1.1	Limit Data Size	254
20.2	Subset Data Base with dplyr	254
20.3	The ff Package	255
20.3.1	Read in the data	255
20.4	Amazon Cloud Services	257
20.5	Google Cloud Services	257
20.6	Big Data: A Sham?	258
21	Sampling Big Data	259
21.1	What is Sampling?	259
21.2	Sampling Example	260
21.3	Current Practices	261
	Index	267

Preface to the Book

I accidentally started writing this book during Spring Break 2018. I was driving down a long Texas road out in the country, thinking about a problem with the undergraduate machine learning class I had been teaching. Chiefly, there was no book suitable for my course. There are several excellent machine learning books for graduate students. There are a few books for undergraduate students, but they didn't cover all the topics I was required to teach for my course at The University of Texas at Dallas. The solution I came up with was to format my own notes in latex. I scribbled an outline on a piece of paper as I drove down the road. When I came home I started looking for a suitable latex template and found this one which I really liked. It was designed for a book and I thought, why not just make this a book? And so the book began.

Many years ago I authored several textbooks on microprocessors and microcontrollers with my husband. These books were quite successful and provided a nice second income for our family. I was able to write the books at home while taking care of our two rambunctious sons. However, the days when one can write textbooks as a job are gone. As soon as a book is published, it is pirated, and the pdf is available for free to anyone in the world. That is not entirely a bad thing in my opinion. Yes it is stealing intellectual property from the owner but on the other hand it makes information available to people who want to learn but perhaps can't pay the exorbitant price that publishing companies charge for textbooks. And so my goals are not monetary. The pdf of this book will be free for my students, although there will be a print version available on Amazon for those who prefer to read from paper.

My goals in writing this book are the following:

- To create an undergraduate book that covers the material needed for my course.
- To clarify how I want to teach the material as I write.
- To provide an accessible introduction to the field to those who want to get started with machine learning.
- To increase the number of practitioners in the field. I believe that machine learning is to the present and future what computer programming was in the 1970s when I started: a revolutionary new way of solving problems.
- To help people enhance their skill set. If I can introduce people to these skills it will help them in their career goals.

I hope you enjoy the book. This first edition covers what I teach in a one-semester undergraduate Introduction to Machine Learning course. However, I consider it to be a work in progress and I will include additional topics in the future. You can find out about future editions on my website: www.karenmazidi.com

The book has a companion github site for code samples: https://github.com/kjmazidi/Machine_Learning/



Part One: Introduction to Machine Learning and R

1 The Craft of Machine Learning 17

- 1.1 Machine Learning
- 1.2 Machine Learning Scenarios
- 1.3 Machine learning v. traditional algorithms
- 1.4 Terminology
- 1.5 Notation

2 Learning R 23

- 2.1 Installing R, RStudio
- 2.2 R Data
- 2.3 Data Exploration
- 2.4 Visual Data Exploration
- 2.5 Factors
- 2.6 R Notebooks
- 2.7 Control structures
- 2.8 R Style
- 2.9 Summary

1. The Craft of Machine Learning

This handbook provides an introduction to machine learning using R. It assumes no prior experience with either machine learning or the R language. The book is designed for upper level undergraduate or postgraduate students in computer science as well as other disciplines, since machine learning has found application in a wide variety of fields from the social sciences, biological sciences, economics, and more. The handbook will also be a good resource for professionals wanting to get started with machine learning. Prior courses in linear algebra, probability and statistics are assumed; however, the book attempts to fill in as much detail as needed for the subject at hand. This is a handbook¹, not a textbook. A few practice problems are provided with the assumption that more problems will be provided by your instructor. In these days where solutions to textbook problems are found online, for a price, there is not much educational value to be had in end-of-chapter problems. Readers can find on the web many sample R notebooks and learn from those as well. Two particularly good resources are <https://www.r-bloggers.com/> and <https://www.kaggle.com>.

The aim of this handbook is to provide an introduction to the craft of machine learning through conceptual explanations of the algorithms and small examples of running the algorithms in R. Sample notebooks are provided on the author's github at: https://github.com/kjmazidi/Machine_Learning.

Figure 1.1 shows fields related to machine learning. Machine learning would not be possible without the fields surrounding it in the figure. Statistics and probability form the mathematical foundations of many of the algorithms we will learn in this book. AI and computer science pushed the frontiers of what computers could do which made machine learning possible.

¹A reference work providing guidance for a technical application or art.

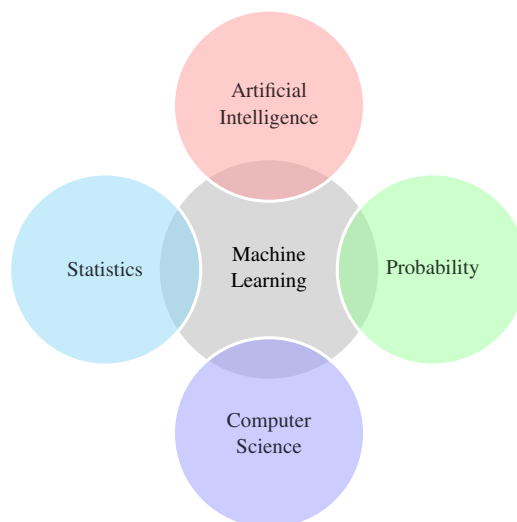


Figure 1.1: Fields Related to Machine Learning

1.1 Machine Learning

We will use *machine learning* as an umbrella term for many closely related terms. Some statisticians call machine learning *statistical learning*. The field of *data science* and the task of *data mining* often involve machine learning techniques, coupled with more data exploration and analysis.

Machine learning has received varied definitions as the field developed. This book proposes the following definition:

Definition 1.1.1 — Machine Learning. Machine learning trains computers to accurately recognize patterns in data for purposes of data analysis, prediction, and/or action selection by autonomous agents.

The key words in this definition are: data, patterns, predictions, and actions, along with the caveat: accurate. Let's examine these in detail.

1.1.1 Data

Nothing can be learned without data. The data and what we wish to learn from the data go hand in hand. For many learning scenarios, data takes the form of a table of values where each row represents one data example, and columns represent attributes or features of the examples. One learning scenario, clustering, seeks to group like instances. Another learning scenario, supervised learning, seeks to learn about one feature based on combinations of the other features. Data can take other forms besides tables, such as a set of actions weighted by features that represent the current environment.

The data we will use in this handbook is small and neat compared to data you will encounter in real-world machine learning problems. This is by design, so that focus can be placed on the algorithms and how they learn from data. Just be aware that in real-world scenarios, more of your time will be spent in data gathering and data cleaning than in the actual machine learning.

When gathering data or using data from other sources, ethical considerations must

always guide our actions. Who owns the data? Who are the subjects of the data? Has the data been anonymized? Did the subjects give consent for the use of the data? How will the analysis of this data be used? How might it impact the subjects in the data as well as the larger community?

1.1.2 Patterns

The best *general* pattern recognition machine is the human mind, but computers can in many cases beat human performance on narrowly defined tasks. The ability to recognize patterns in data enables algorithms to learn things like whether someone is a good credit risk, whether two people might be compatible, whether the object outlined in sensors is a human or a dark spot on the pavement.

When beginning a real-world machine learning project, how do you know what to look for? From raw data, organized data must be built. Once the data is organized, decisions must be made about what could be learned and what is important to learn from the data. These decisions often need to be in concert with domain experts and/or the owners and users of the data who wish to learn from it.

1.1.3 Predictions from Data

Learning patterns in data enables us to predict outcomes on future data – data the algorithm has never before seen. Predictions may simply involve finding like instances in the data, or predicting a target value which may be quantitative or qualitative.

In the examples in this book, generally we train algorithms on a portion of the data and use the remaining data to test and evaluate how well the trained model can perform on previously unseen data. This is a common situation in machine learning, sometimes called batch learning because the data is fed into the algorithm in one batch. There are other approaches, however, such as online learning in which the algorithm is continually learning from newly available data and being evaluated in real time. Online learning techniques can also be used when the available data is too large to be stored in memory. An alternate approach to handle big data is to do parallel distributed machine learning, often in the cloud with specialized software.

1.1.4 Accuracy

Predictions must be accurate or they are not predictions but random guesses. Typically, we want our predictions to beat some predetermined baseline approach. For example if 99% of the observations in a credit data set did not default on their loans, we want to beat a simple baseline that always guesses "not default" and has 99% accuracy. Machine learning makes use of many measurement techniques to gauge accuracy and evaluate performance of the algorithms. Many of these metrics are used to evaluate the training model itself and others will be used to evaluate performance on a held-out test set.

1.1.5 Actions

Every day more autonomous agents enter our lives, from smart thermostats, to automated assistants, to self-driving vehicles. These agents take actions based on what they have learned, and most continue to learn over time, usually by uploading data to a central learning repository. Some actions taken by autonomous agents will be controversial in

the coming decades as ethical and legal issues evolve in response to humans co-existing with autonomous agents. The big players in AI are at the forefront of autonomous agents because they have the resources, expertise, and data to pursue big projects. In the future, we expect the development of autonomous agents to be available to more developers.

1.2 Machine Learning Scenarios

There are scores of machine learning algorithms with countless variations each. This book describes the most common algorithms, while providing a foundation for students to learn more algorithms on their own. There are many ways to classify machine learning algorithms, and not all algorithms fit neatly into our categories but the following general categorizations serve as a helpful overview of the field.

1.2.1 Informative v. Active

Most machine learning algorithms covered in this book are **informative**. They provide information to us in the form of data analysis or prediction. These informative algorithms input data observations and output a model of the data that can then be used to predict outcomes for new data fed into the model. In contrast, the field of Reinforcement Learning teaches **active** agents to identify optimal actions given the current environment and what has been learned in past experience. The input to these algorithms for initial training comes in the form of data but some agents may continue to learn with sensors and other input methods that let them learn from the environment.

1.2.2 Supervised v. unsupervised learning

Informative algorithms are of two main types. The term **supervised learning** refers to scenarios where each data instance has a label. This label is used to train the algorithm so that labels can be predicted for future data items. The term **unsupervised learning** refers to scenarios where data does not have labels and the goal is simply to learn more about the data.

1.2.3 Regression v. classification

Supervised learning algorithms fall into two major groups. Regression and classification occur in supervised learning scenarios with labeled data. In **regression**, our target (the label) is a real-numbered value, like trying to predict the market value of a home given its square footage and other data. In **classification**, our target is a class, like predicting if a borrower is a good credit risk or not, given their income, outstanding credit balance, and other predictors.

1.3 Machine learning v. traditional algorithms

Machine learning algorithms are different from traditional algorithms encountered in computer programming. In Figure 1.2 we see the traditional computer programming paradigm on the left: data is fed into code that processes it and outputs the results of the processing. In Figure 1.2 we see the machine learning paradigm on the right: we feed data into an algorithm which builds and outputs a model of the data. In traditional programming,

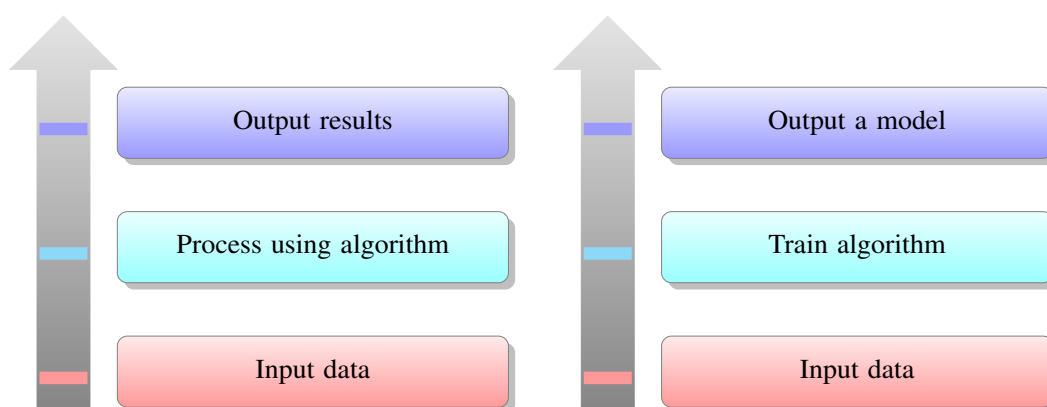


Figure 1.2: Traditional Programming (Left) v. Machine Learning (Right)

all knowledge is explicitly encoded in the algorithm by programmers using code statements, loops, and conditional statements. Therefore, knowledge must be known beforehand. In machine learning, knowledge is inferred from data. Knowledge is discovered.

Why do we need machine learning? Can't we just explicitly code algorithms for problems? There are two typical situations in which traditional programming cannot be used to solve problems. The first type is when it is not possible to encode all the rules needed to solve a problem. How would you encode rules for recognizing faces in photos? We don't even know the rules we use in our minds to recognize faces so it would not be possible to encode rules. However, we can train computers to recognize key edges and regions of photos that are likely to be faces. The second type of situation in which traditional programming cannot be used to solve a problem is when the scale of the problem is too large. If a company has huge amounts of customer data it would take millions of human hours to find useful patterns in the data that could be then extracted programmatically. Machine learning algorithms can find patterns quickly in large amounts of data.

As we go through the material in this handbook you will learn several machine learning algorithms. These algorithms typically have statistical and probabilistic foundations which we will explore. However, beyond the theory and technique, machine learning is also a craft as well as a science. Each major Part of the book devotes a chapter pointing out some innovations, ideas, and techniques from this evolving craft.

1.4 Terminology

Machine learning grew out of statistics and probability, computer science, as well as other fields. For this reason there are often multiple terms for the same thing. Let's start with names for data. The table below contains a sample data set (with headings).

We have only 3 rows of data. Each row is a sample data point, also called an **example**, **instance**, or an **observation**. Each column in the table is an **attribute**, also called a **feature** or a **predictor**. We have 4 features: GPA, average number of hours studied per week, SAT score, and classification. The first 3 are **quantitative**, or numeric, features while Class is a **qualitative** feature because it can only take on one of a finite set of values. Qualitative features are also called **factors** or **categorical data**.

GPA	Hours	SAT	Class
3.2	15	1450	Junior
3.8	21	1420	Sophomore
2.5	9	1367	Freshman

Table 1.1: Student GPA, Average Hours Studied/Week, SAT, Class

If we want to learn GPA as a function of the other 3 features, we say that GPA is our **target**, or **response**, variable while the other 3 are **features** or **predictors**. If we want to predict SAT then SAT would be our target and all other columns our predictors.

1.5 Notation

This book uses the following notation conventions for data:

- x_i subscript i indexes observations in a data set; i ranges from 1 to N , the number of observations (rows) in the data set
- $x_{i,j}$ subscript j indexes predictors in a data set; j ranges from 1 to P , the number of predictors (columns) in the data set. Here we are referencing predictor j from observation i .
- In matrix notation, lower case letters like x represent scalars, bold face lower case letters like \mathbf{x} represent vectors, and upper case bold face letters like \mathbf{X} represent matrices.

2. Learning R

Why R?

R is simple enough for programming novices to learn and a breeze for computer scientists. Everything in R is built in. You can switch seamlessly from data cleaning, to data analysis, to creating plots, to running machine learning algorithms, to performing statistical analysis on your results. R is open source, supported by a core group of contributors. There is substantial online help at sites such as <https://stackoverflow.com> and <https://stats.stackexchange.com/>. R gives us the ability to get work done quickly with minimal learning curves. R is the *lingua franca* in academia among data scientists, statisticians, and machine learning specialists. R is also used widely in industry by companies with a lot of data like Google, Microsoft, Amazon, as well as traditional industries. R also gives you great tools for easily sharing and communicating your results with html or pdf reports as well as interactive web sites.

In the next few sections you will learn:

- Where to find R and RStudio installation links and instructions
- R data structures and how to manipulate them in R syntax
- R data exploration functions and techniques
- R data visualization functions and techniques
- How to install and use R packages
- How to use R notebooks which combine text and code
- R control structures for more complex code
- Recommendations for R style

2.1 Installing R, RStudio

You will need to first install R and then install RStudio, a beautiful and powerful IDE for R. Both are available for Windows, Mac and linux.

Link to download and install R: <https://www.r-project.org/>

Link to download and install RStudio: <https://www.rstudio.com/>

Choose RStudio's free community edition. While you are on the site take a look around at some demos and tutorials.

2.1.1 Getting Started in R

If you open up RStudio you will see a couple of panes on the right, and one large pane on the left, similar to Figure 2.1. This pane on the left is the console. At the top of the console screen you will see some information about your version of R, then the interactive prompt `>` waiting for you to do something. In the figure, we've typed in a simple expression: `3+4`.

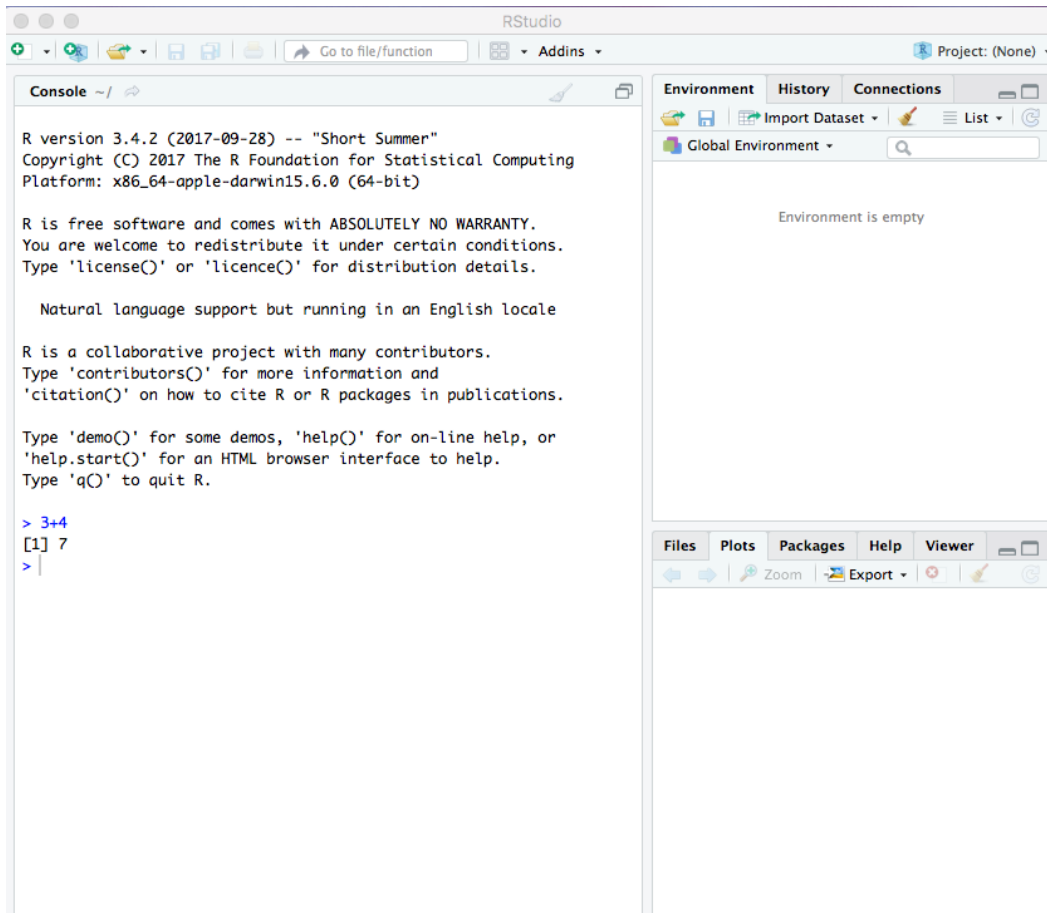


Figure 2.1: RStudio IDE

R is an interpreted language which makes it easy to experiment with ideas at the console. You can also save your code in notebooks or scripts, to run again and again. Although R is an interpreted language, most of the code is in C/C++ so it is fast. For now we will get to know R at the console.

Type along with these examples in the console window as we go. Below, we typed an arithmetic expression at the prompt, hit enter and we see the results on the next line. There should be no surprises here, it's the usual operators with the usual order of operations.


```
> 3 + 4 - 1 * 8 / 2
[1] 3
```

But what is that [1] doing there in front of the output 3? That's just telling you that your result is a one-dimensional object. It's actually a vector of length 1. In R, pretty much everything is an object. Let's type in the same expression as above but this time save it to variable `x`. By the way, the up arrow displays the previous command, just like in a unix console. The less-than-dash, `<-`, is the assignment operator in R. We did not have to declare variable `x`. R figured out what type it is by what was stored in it. R has dynamic typing; if we change what type of object is in `x` later, R will not complain.

```
> x <- 3 + 4 - 1 * 8 / 2
> x
[1] 3
```

Now when we type in `x` at the console, R echoes back its contents. What is `x`? Let's ask R about it. There are many functions we can use to inquire about an object:

```
> is.numeric(x)
[1] TRUE
> class(x)
[1] "numeric"
> typeof(x)
[1] "double"
```

Note that numbers are stored as doubles by default. If we specifically want to store an integer we use the `L` indicator:

```
> x <- c(1L, 2L, 3L)
> typeof(x)
[1] "integer"
```

The `c()` function combines elements into a vector. A vector is a sequence of objects of the same type. The `c()` function will coerce its arguments to a common type, the most inclusive type in the vector:

```
> x <- c(1L, 1.2, "hi")
> typeof(x)
[1] "character"
```

2.2 R Data

We have seen in the examples above that data in R can be logical, numeric (integer or floating point), or character. Data is organized into objects of varying types. In this section we learn more about R objects that hold and organize data. The following table organizes the R data structures by their dimensions and whether or not all elements have to be of the same type.

Dimension	Homogeneous	Heterogeneous
1d	atomic vector	list
2d	matrix	data frame
nd	array	

Table 2.1: R Data Structures

2.2.1 Vector

A vector is a sequential structure with one or more elements of the same type. There isn't really a scalar in R, it's just a vector of length 1. Follow along with this code:

```
> x <- 1:10
> x*5
[1] 5 10 15 20 25 30 35 40 45 50
> length(x)
[1] 10
> sum(x) # try mean(), median(), range(), max(), min()
[1] 55
```

In the first line, `x` is assigned to be a vector from 1 to 10 with the sequence operator `:`. In the second line we multiply each element by 5. In most programming languages you would need a loop to do this but note the simple syntax that allows us to do this in one line. There are many built-in functions in R. The next line shows `length()`. The last line returns the sum of each element of `x`. You may be wondering why it is 55 since we multiplied each element by 5 above. The reason is that we did not assign it back to `x` as in `x <- x*5`.

The vector `x` is numeric but we can have other types of vectors such as character or logical. When we type `"x > 5"` at the console, a logical vector returns with `TRUE` or `FALSE` for each element. If we `sum(x>5)`, the `TRUE` values count as 1 and the `FALSE` values as 0 so we get a count of how many elements are greater than 5. Notice in the last line below that we can select those instances greater than 5 and replace only those with 0. Such power with such simple syntax!

```
> x
[1] 1 2 3 4 5 6 7 8 9 10
> x > 5
[1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
> sum(x>5)
[1] 5
> x[x>5] <- 0
> x
[1] 1 2 3 4 5 0 0 0 0 0
```

Subsetting vectors requires first learning one strange thing about R. It starts indexing at 1, not 0 like other programming languages. If you type `x[0]` at the console you won't get an error but some information about the vector, so it looks like that space is put to good use. Here are some indexing examples:

```

> x <- 1:10
> x[0]
integer(0)
> x[2:4]      # use : for a range of elements
[1] 2 3 4
> x[c(2:4,8)] # use c() for noncontiguous elements
[1] 2 3 4 8
> x[11]
[1] NA

```

Notice above that we did not get an error when we had an index out of bounds, it just gave us an NA, for "not available".

To check the type of a vector you can use:

- `typeof(x)` # returns type
- `is.integer(x)` # returns boolean
- `is.double(x)` # returns boolean
- `is.numeric(x)` # returns TRUE for integer or double
- `is.character(x)` # returns boolean
- `is.logical(x)` # returns boolean


2.2.2 Lists

A list is an ordered collection of objects not necessarily of the same type. Lists can contain other lists as elements. Lists are often used as holders for things returned from functions. List elements are selected with double square brackets:

```

z <- list(1,2,3)
z[[1]]
[1] 1
y <- list('a', TRUE, z)
> typeof(y)
[1] "list"
> length(y)      # y is a list of length 3
[1] 3
> length(y[[3]]) # the 3rd element is also a list of length 3
[1] 3

```

 **Lists v. Vectors.** A list is a generic vector where elements do not have to be of the same type. For this reason, to check if an object is a vector, don't use `is.vector()` but the following:

```

x <- 1:5
is.atomic(x) # check if x is a vector
[1] TRUE

```

You can check if a list has lists as elements with the `is.recursive()` function. The `unlist()` function converts a list into a 1-dimensional atomic vector, coercing all elements into the most general type.

```
> w <- unlist(y)
> w
[1] "a"      "TRUE" "1"      "2"      "3"
> typeof(w)
[1] "character"
```

2.2.3 Matrix

A matrix is a 2-dimensional object with elements of the same type. The code below creates a sequence from 1:10 and stores it in a matrix of 2 rows.

```
> m <- matrix(1:10, nrow=2)
> m
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Notice the handy reminders showing how to index at the top of the columns and the left of the rows. Practice indexing matrix `m`. Indices are `[row, col]`.

We will mainly use vectors and data frames (discussed below) in machine learning but we will use matrices from time to time.

2.2.4 Arrays

Arrays are similar to matrices but can have more than 2 dimensions. Like matrices, all elements must be of the same type.

2.2.5 Data Frames

A data frame is a 2-dimensional structure where each column can be of a different type. Most of the time we will load a data frame from disk but we can create one manually by creating 3 vectors that we combine with `cbind` (column bind):

```
> x <- c(1,2,3)
> y <- c(1.1, 2.2, 3.3)
> z <- c('a', 'b', 'c')
> df <- data.frame(cbind(x,y,z))
> df
   x  y z
1 1 1.1 a
2 2 2.2 b
3 3 3.3 c
```

In a data frame, each column is a vector of a specific type. All the columns are of the same length. Observe in the code below that we can access these column vectors with the dollar operator and that we can also change the column names.

```
> df$x
[1] 1 2 3
```

```

Levels: 1 2 3
> colnames(df) <- c('Ticket', 'Discount', 'Section')
> df
  Ticket Discount Section
1      1       1.1      a
2      2       2.2      b
3      3       3.3      c

```

We can read in a text file or csv file from disk as shown below. If you want to know more about the options for read.csv use the command at the bottom of the following code example. The str() functions tells you about the structure of the data frame, its columns, and what type of data it contains, as well as how many observations and variables.

```

> df <- read.csv("titanic3.csv")
> str(df)
'data.frame': 1310 obs. of  14 variables:
 $ pclass   : int  1 1 1 1 1 1 1 1 1 1 ...
 $ survived : int  1 1 0 0 0 1 1 0 1 0 ...
 $ name     : Factor w/ 1308 levels "", "Abbing, Mr. Anthony", ...
 $ sex      : Factor w/ 3 levels "", "female", "male": 2 3 ...
 $ age      : num  29 0.917 2 30 25 ...
 $ sibsp    : int  0 1 1 1 1 0 1 0 2 0 ...
 $ parch    : int  0 2 2 2 2 0 0 0 0 0 ...
 $ ticket   : Factor w/ 930 levels "", "110152", "110413", ...
 $ fare     : num  211 152 152 152 152 ...
 $ cabin    : Factor w/ 187 levels "", "A10", "A11", ...
 $ embarked : Factor w/ 4 levels "", "C", "Q", "S": 4 4 4 2 ...
 $ boat     : Factor w/ 28 levels "", "1", "10", "11", ...
 $ body     : int  NA NA NA 135 NA NA NA NA NA 22 ...
 $ home.dest: Factor w/ 370 levels "", "?Havana, Cuba", ...
> ?read.csv # ? is the help command

```

There are also many built-in data sets in R. You can find out about them by typing data() at the console.

2.3 Data Exploration

You can load a built-in data set with the data() function and then look at a few rows with head() or tail(). Comments start with # so you don't need to type those in when you are working at the console.

```

> data(airquality)
> head(airquality, n=2) # see the first two rows
  Ozone Solar.R Wind Temp Month Day
1   41    190  7.4   67     5    1
2   36    118  8.0   72     5    2

```

Here are several data exploration functions you should explore at the console, replacing "df" with the name of your data frame, such as "airquality":

- `names(df)` lists the column names
- `dim(df)` gives the row, col dimensions
- `summary(df)` gives summary statistics for each column
- `str(df)` gives row and column counts, information for each column
- `head(df)` and `tail(df)` give the first and last 6 rows by default

Missing items in a data frame are typically encoded as NA. This can cause some problems for built-in functions as shown below. Notice the `sum()` function would not work until we told it to ignore NAs.

```
> sum(is.na(airquality$Ozone)) # count the NAs
[1] 37
> mean(airquality$Ozone)
[1] NA
> mean(airquality$Ozone, na.rm=TRUE)
[1] 42.12931
```

A trickier problem is deciding what to do with NAs before we run data sets through machine learning algorithms. Some algorithms may not work and others may give less than optimal results with NAs. One option is to delete rows that have NAs but this could be a problem if your data set is already small. Another option is to replace NAs with a mean or median value as shown next. First we copy the data set to another variable so we won't alter the original.

```
> df <- airquality[]
> df$Ozone[is.na(df$Ozone)] <- mean(df$Ozone, na.rm=TRUE)
> mean(df$Ozone)
[1] 42.12931
```

R syntax can be quite compact and nested; it takes a little getting used to. The syntax `df$Ozone[is.na(df$Ozone)]` above selects only those rows in `df` in which Ozone is NA. Only these will be replaced by the mean of the column.

2.4 Visual Data Exploration

In addition to exploring data with the R commands described in the last section, and exploring the data with built-in statistical functions, we can also explore data with graphs. Type `demo(graphics)` at the console to see the variety of graphs you can create in R. Next we continue looking at `airquality` with R graphical functions.

The `plot()` function is most often used to plot points as in `plot(x, y)`, where the first argument is plotted on the x axis and the second on the y axis. If only one argument is given, as in `plot(y)`, the x axis will have a numbering vector. Type the following at the console to see the graphs:

```
hist(airquality$Temp)
plot(airquality$Temp)
plot(airquality$Temp, airquality$Ozone)
```

Here are some useful arguments for modifying graphs:

- `pch` - specifies the symbol to use for points; 1 is the default
- `cex` - specifies symbol size, 1 is the default, 1.5 is larger, 0.5 is smaller
- `lty` - specifies line type, default is 1
- `lwd` - specifies line width, default is 1
- `col` - color of graph element; colors can be specified by name, number, hex; `col=2` and `col="red"` are the same
- `xlab` - label for x axis
- `ylab` - label for y axis
- `main` - main label

We will learn more about graphs in R as we go. The following link provides a good overview of R graph parameters: <https://www.statmethods.net/advgraphs/parameters.html> Below is an example of using some of these parameters in code. Figure 2.2 shows the resulting plot. The `plot()` code says to plot Ozone on the x axis and Temp on the y axis. These plot points are shown with `pch=16` (see the statmethods link above for other options), color blue, and 1.5 times the regular size. We also specified a main label as well as x and y axis labels.

```
plot(airquality$Ozone, airquality$Temp, pch=16, col="blue", cex=1.5,  
     main="Airquality", xlab="Ozone", ylab="Temperature")
```

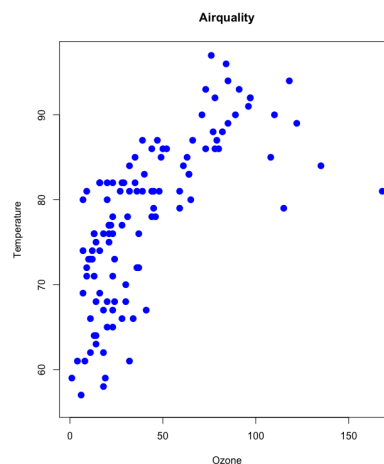


Figure 2.2: Airquality

The `plot()` example above says to plot Ozone on the x axis and Temp on the y axis. These plot points are shown with `pch=16` (see the statmethods link above for other options), color blue, and 1.5 times the regular size. We also specified a main label as well as x and y axis labels.

Often we want to know if columns are correlated with other columns. A high positive correlation between x and y would look like a line of slope 1. A high negative correlation would look like a line of slope -1. Correlation is quantified on a scale of -1 to $+1$, with values closer to the 1s indicating strong positive or negative correlation and values closer to

0 indicating weak or no correlation. The code below shows how to generate a correlation matrix or a graph. See if you can identify correlations in the graphs.

```
> cor(airquality[1:4], use="complete")
      Ozone  Solar.R   Wind   Temp
Ozone  1.0000000  0.3483417 -0.6124966  0.6985414
Solar.R 0.3483417  1.0000000 -0.1271835  0.2940876
Wind   -0.6124966 -0.1271835  1.0000000 -0.4971897
Temp    0.6985414  0.2940876 -0.4971897  1.0000000
> pairs(airquality[1:4])
```

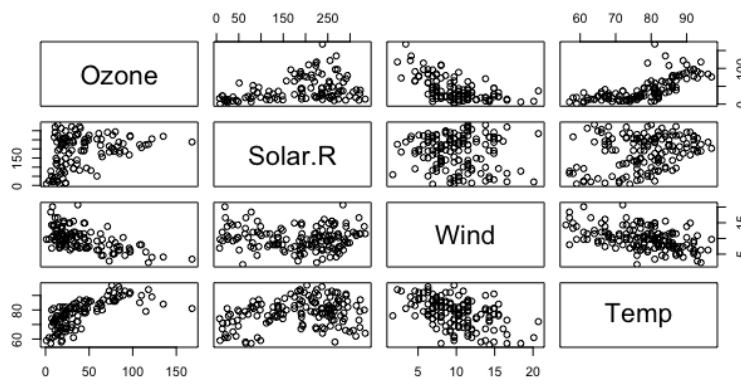


Figure 2.3: Correlation Pairs for Airquality

In the `cor()` code above, we requested it to only use data that is not NA with the `use=` parameter. Remove that extra parameter and compare your results to the output above. The `pairs()` command requested to look at only columns 1:4.

2.5 Factors

We have seen vectors of various types: integer, numeric, character, and logical. There is one more type of vector that will be important to us. Factors are used to encode qualitative data. A factor vector is a vector of integers with an associated vector of labels. Internally it is stored as integers but for our benefit they display as labels. Look at the Titanic data.

```
> df <- read.csv("data/titanic3.csv", na.strings="NA",
  stringsAsFactors=FALSE, header=TRUE)
> str(df)
'data.frame': 1310 obs. of 14 variables:
 $ pclass : int  1 1 1 1 1 1 1 1 1 1 ...
 $ survived : int  1 1 0 0 0 1 1 0 1 0 ...
 $ name : chr  "Allen, Miss. Elisabeth Walton" ...
 $ sex : chr  "female" "male" "female" "male" ...
 . . .
```


The above code assumes that you have the data/csv file in the same directory you are working in. We used the parameter `na.strings="NA"` to tell R to fill missing cells with NA. This data is a bit messy because some things that should be factors, like `pclass` or `survived`, are not. If we had not used the `stringsAsFactors=FALSE` parameter, strings like `name` or `ticket` would be encoded as factors. We can convert to another data type as shown below. Notice we can tell R about the factor levels we want if we use `factor()` instead of `as.factor()`.

```
df$pclass <- as.factor(df$pclass)
df$sex <- factor(df$sex, levels=c("male", "female"))
```

If you rerun the `str()` function on the data you will see that `pclass` and `sex` are now factors. We can see how many levels a factor has with the `levels()` function, and see the encoding with the `contrasts()` function as shown below.

```
contrasts(df$sex)
      female
male      0
female    1

> contrasts(df$pclass)
  2 3
1 0 0
2 1 0
3 0 1
```

For `sex` we only need one variable to encode the two genders in the data. The contrasts for `pclass` shows that we need 2 variables to encode 3 classes. The base case will be class 1. R will create 2 *dummy variables* for classes 2 and 3. We will see the importance of these when we get to machine learning.

2.5.1 Adding a Factor Column to a Data Frame

The following code adds a new column to our data frame for `airquality`. First it makes all items = FALSE, then makes those with a temperature over 80 to be TRUE. Finally we coerce it to be a factor. The first line has a comment which starts with a hash tag.

```
> df <- airquality[] # copy the data set
> df$Hot <- FALSE
> df$Hot[df$Temp>89] <- TRUE
> df$Hot <- factor(df$Hot)
> df$Hot[40:46]
[1] TRUE FALSE TRUE TRUE FALSE FALSE FALSE
Levels: FALSE TRUE
> plot(df$Hot)
```

Here are some plots of the `Hot` column, arranged in a 1x3 grid with the `par()` function. Plotting the column by itself just gives us a visual of the distribution of `Hot` and not `Hot`.

The `cdplot()` gives a conditional density of Hot (light grey) and not (black) across the x axis which represents temperature. The third plot is a box plot in which the heavy middle line in the box denotes the median, the box itself indicates the IQR and the horizontal lines at either end of the dashed line indicate min and max, excluding suspected outliers which are dots beyond that line.

```
> par(mfrow=c(1,3))
> plot(df$Hot)
> cdplot(df$Temp, df$Hot)
> plot(df$Hot, df$Temp)
```

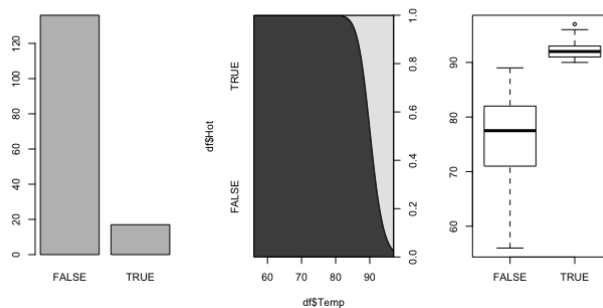


Figure 2.4: Airquality

The graphs shown in this chapter are rather plain. We can add headings, labels, color and more. Read more about graphical parameters here: <https://www.statmethods.net/advgraphs/parameters.html>

2.6 R Notebooks

Here are a few things to keep in mind about R:

- R is case sensitive
- The period has no special meaning unlike many other languages
- The dollar sign acts somewhat like the period in other languages
- The hashtag starts comments
- White space is ignored
- The R workspace stores all the objects you create in a session. You can save it, but for now you should always choose No when it asks you to save the workspace upon exiting RStudio.

If you end up with items in your workspace that you don't want, you can list them with `ls()` and remove things with `rm()`. The second command below shows how to remove everything from the workspace environment.

```
> ls()
[1] "airquality"
> rm(list=ls())
```

R is a powerful language in its own right. However it becomes truly awesome when we load packages built for specific purposes. These packages are available on CRAN, the Comprehensive R Archival Network. We install packages at the console with the `install.packages()` command. You only have to install once. We load packages into our working environment only once per script with either the `library()` or `require()` functions.

In RStudio, go to File, New File, then R Notebook. Save your file. I recommend creating a folder to keep all your notebooks in, then always loading from that folder by double-clicking on it.

Before we get started, we should note that you can also create a simple R script. These are text files that end in `.R` and can be run from the console or from within RStudio. We are going to focus on R notebooks which are similar to Python notebooks, interspersing text commentary and code.

Once you create the R notebook you will see some YAML code at the top. You can also add an `author:` line and you should change the title. YAML is a recursive acronym for `YAML Ain't Markup Language`. This YAML tells RStudio to what kind of output to create, in this case it will make an html notebook that you can upload to your website or github.

```
title: "R Notebook"
output: html_notebook
```

You'll notice white portions for text and grey portions for code in the boiler plate notebook that is created for you. Read through the standard text which provides some useful tips on things like how to add new code chunks and get started with markdown text. You can type regular text in the white portions but markdown is worth learning because you will encounter it in many situations beyond R. There is a nice cheat sheet for markdown in RStudio's web site. The really nice thing about markdown is that you can format as you type without taking your hands off the keyboard. In the figure below the three hashtags denote a level 3 heading. When you create a pdf or html file from this notebook, you won't see the hashtags, just the text rendered as a level-3 heading.

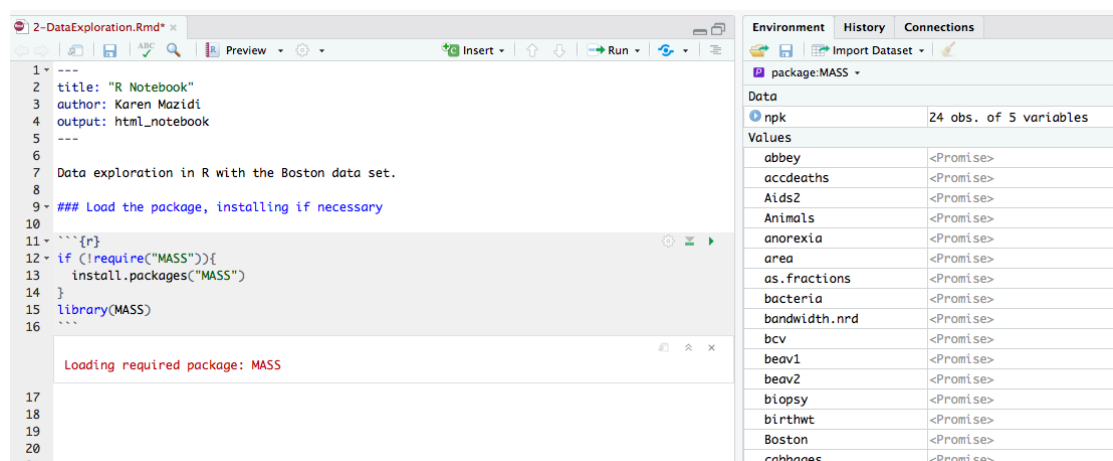


Figure 2.5: RStudio Environment

Figure 2.5 shows the RStudio environment with the white portions for text markdown and the grey portions for code. The full notebook is available on the github site for this

book. You should practice making your own notebook similar to the sample. Let's point out a couple of things before we move on. First, once the MASS package was loaded, you can see it in the environment in the upper right. This environment is a great way to see your variables as you go. It is a great help in debugging. Second, notice the Preview button above the notebook. You can use this to "knit" or render your file into html or pdf or other formats. This option is also available under the File menu.

2.7 Control structures

We have seen earlier that a lot of things we might use a loop for in other languages can be done in simple R syntax. However there will be times when we want to code loops, conditionals, and functions. We describe those in this section, in an example getting you started using an R notebook.

This section uses more advanced R code so it is fine to skip this section and come back to it when you are more familiar with R. The notebook is in the github site so you can refer to it as you need it.

If you wish to go through this section now, open RStudio and go to File->New File->R Notebook. Get rid of the text in the white areas. Change the title at the top of the file. Now we are going to load package `mlbench`, which contains several benchmark ML data sets. In the grey code box, remove the code that is there and type "`library(mlbench)`". If you run this chunk of code either with the green arrow on the right side of the box or just hitting ctrl-enter on the line, you will probably get an error since you probably don't have this package installed. To install the package, type the following down at the console, not in your file:

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

This should just take a minute. If R asks you for a mirror site in a pop-up, just pick one. Once the package is installed, from then on you just load the package into memory with `library()` or `require()`. By the way, if you look back at Figure 2.5 you will see some code that first installs a package only if it is not (!) present, and then loads the package. The difference between `library()` and `require()` is that `require()` returns a TRUE-FALSE value that was used to good benefit in that sample code. Now type the following at the console:

```
data(package='mlbench')
```

This will pop up a tab listing the data in the package. Once you have installed `mlbench`, add the following commands to the code chunk so that it looks like the following and rerun the code.

```
```{r}
library(mlbench)
data(PimaIndiansDiabetes2)
str(PimaIndiansDiabetes2)
```
```

Figure 2.6: First Code Chunk

In the upper right pane of RStudio, click on Environment then Global Environment. You should see that the data has been loaded into memory. You should see that it has 768 observations and 9 variables. The `str()` command you ran above will output information about each variable. You can learn more about this data set by typing `?PimaIndiansDiabetes2` at the console.

From the `str()` function above, you can see that there are a lot of missing values denoted by NA. Create a new code chunk at the bottom of your notebook by clicking on Insert->R at the top of the notebook window. Type in the following code and see the results. The `sapply` function applies a function to data. Here we have an anonymous function coded on the fly to sum NAs. The `sapply()` function will apply this to each column. We have a few NAs for glucose, mass, and pressure, but a lot for triceps and insulin. If we just omit all rows with NAs that will cut our data in half. An alternative to just deleting them is to fill them with either the mean or the median of the column. In the code section 2.7.1 shown below, we use an if-else to either calculate the mean or median of a vector. Then we write a function to fill NAs of a vector with either the mean or the median. Whether or not it is a good idea to fill missing values in this way may depend on how you are using the data. Always document any changes you made to your data.

```
```{r}
sapply(PimaIndiansDiabetes2, function(x) sum(is.na(x))==TRUE))
```
```

| pregnant | glucose | pressure | triceps | insulin | mass | pedigree | age | diabetes |
|----------|---------|----------|---------|---------|------|----------|-----|----------|
| 0 | 5 | 35 | 227 | 374 | 11 | 0 | 0 | 0 |

Figure 2.7: Second Code Chunk and Results

2.7.1 if-else

Within the function is an if statement. The if statement in R has this form:

```
if (condition) {statements if true} else {statements if false}
```

The `()` around the condition is required, as are curly braces around the statements. Statements should occur on individual lines for readability, although R will allow multiple statements separated by a semicolon. We will discuss R style preferences below, but note that the opening `{` is at the end of a line and the closing `}` is on a line by itself. The if-else statement below lets us use either mean or median, depending on the choice sent to the function.

We will see examples of the ifelse shortcut throughout the book. Here is the format:

```
ifelse(cond, true, false)
```

2.7.2 Defining and Calling Functions

The code below shows a user-defined function. The definition format is:

```
name <- function(args) {statements}
```

Code 2.7.1 — Function Example. Third Code Chunk

```

fill_NA <- function(mean_med, v){
  # fill missing values with either 1=mean or 2=median
  if (mean_med == 1){
    m <- mean(v, na.rm=TRUE)
  } else {
    m <- median(v, na.rm=TRUE)
  }
  v[is.na(v)] <- m
  v
}

# make a new data set with NA's filled
df <- PimaIndiansDiabetes2
df$triceps <- fill_NA(1, df$triceps)
df$insulin <- fill_NA(1, df$insulin)
df <- df[complete.cases(df),] # omit rows with NAs

```

The next-to-last line of our function replaced all NAs in the vector with the mean (or median). Notice that there is no "return" statement. The result of the last statement in the function is what is returned, which in this case is our updated vector. Notice again that the opening curly brace for the function is at the end of the line and the closing curly brace is on a line by itself.

Our calling statements simply have the name of the function and the arguments. The result of the function replaces the original values of those vectors. The last line of the third code chunk handles the remaining missing value which are few. Our data set df will have 724 observations.

For an additional example of a function in R, let's look at a recursive function for the familiar Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, ...

Code 2.7.2 — A Recursive Function. Fibonacci Sequence

```

fib <- function(n){
  if (n <= 1){
    return(n)
  } else {
    return (fib(n-1) + fib(n-2))
  }
}

# print a sequence
for (i in 1:8)
  print(fib(i))

```

Try this out on your computer. Note in the above code that we used return(). However, the function would work correctly without the surrounding return(). For example, instead of return(n) just n.

2.7.3 Loops

R has for-loops and while-loops. The formats are:

```
for (condition) {statements}
while (condition) {statements}
```

The next code chunk shows an example for-loop. The loop creates 3 linear models. The linear model is the first algorithm we will learn, in the next chapter. The formula "glucose~df[,cols[i]]" will learn 3 models: glucose as a function of mass, glucose as a function of age, and glucose as a function of the number of pregnancies, because these are the columns selected in variable cols. These three models are stored in a list.

First we plot mass on the x axis and glucose on the y axis. Then we plot 3 ablines, one for each stored model. The abline function can be used to plot the regression line as we are doing here, but can also be used to plot straight lines. We make each line a different color by just letting color =i, our index. Notice that we had to use the double square bracket to index the list items. Finally we add a legend using the same color codes as the ablines.

```
```{r}
cols <- c(6,8,1)
plot(df$mass, df$glucose, main="PimaIndianDiabetes2")
for (i in 1:3){
 model <- lm(glucose~df[,cols[i]], data=df)[1]
 abline(model, col=i)
}
legend("topright", title="Predictors", c("Mass", "Age", "Pregnant"), fill=c(1,2,3))
```
```

Figure 2.8: Fourth Code Chunk

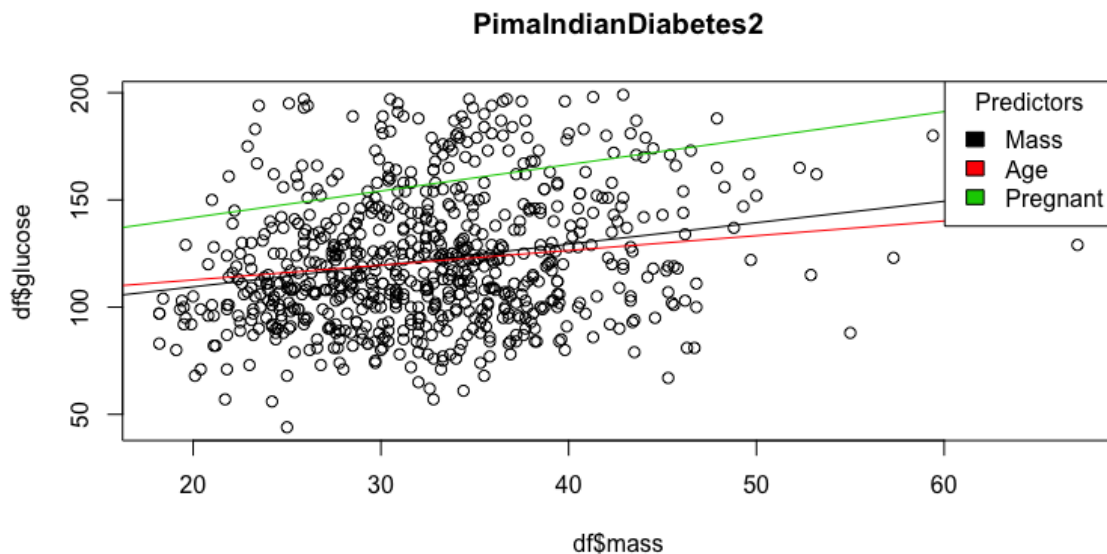


Figure 2.9: Graph with Regression Lines

Next, we demonstrate the `ifelse()` shortcut. The `ifelse()` below checks if insulin is greater than 155. If this is true, the element will be 1 otherwise it will be 0. The `ifelse()` creates an integer vector, which is converted to a factor vector by `factor()`. In the first line of the 5th code chunk we are adding a new column to the data frame. In the second line, we plot the observations again, this time color coding those with high insulin as red, and those that do not have high insulin as blue. The `bg=` argument indicates what fill color is used for the points. We have a vector of "blue" and "red". How these colors are selected is controlled by the `unclass()` function which converts the "large" column factors to 1 or 2. So observations with large insulin values display as red and those with low values display as blue.

```
```{r}
df$large <- factor(ifelse(df$insulin>155,1,0))
plot(df$mass, df$glucose, pch=21, bg=c("blue","red")[unclass(df$large)])
```
```

Figure 2.10: Fifth Code Chunk

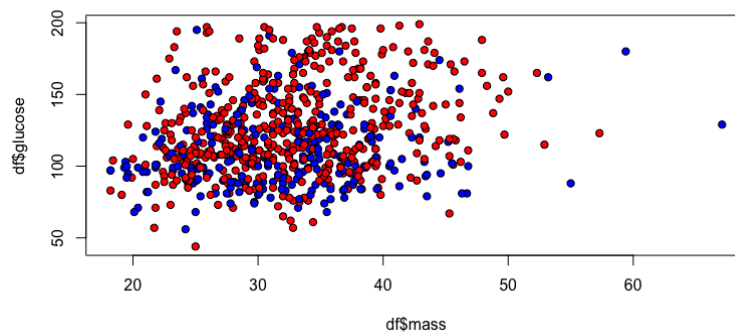


Figure 2.11: Graph with Color Coding

2.8 R Style

Google has its own R style guide available here: <https://google.github.io/styleguide/Rguide.xml>. Another style guide, written by Hadley Wickham, is available here: <http://adv-r.had.co.nz/Style.html>. Hadley Wickham is the Chief Scientist at RStudio, and an influential R developer, having created several amazing open-source packages such as `ggplot2`, `dplyr` and more. The point is not to follow one style guide rather than the other, but to be consistent in your code. It makes it easier to read, even for you.

Here are a few recommendations from Hadley Wickham's style guide:

- variable and function names should be lowercase, using underscore to separate names
- variable names generally should be nouns and function names should be verbs
- try to use names that are concise but meaningful
- don't use names of existing functions; R will let you override them

- put a space around operators, except :
- otherwise don't add unnecessary spaces
- opening curly braces should not be on their own lines
- closing curly braces should go on a new line unless followed by else
- indent with 2 spaces, exception: function arguments

2.9 Summary

This chapter provided a fast-paced introduction to R. Don't expect to retain everything in the chapter right now, you can refer back to it as you need it. You will learn more R as we go, but now you already know enough to get started with machine learning in the next chapter. The github site has an R cheat sheet that will be helpful as you start coding your own scripts.

Also in the github are full notebooks of the data exploration and R control structure examples in this chapter. Make sure you understand the R code in these notebooks. Link: https://github.com/kjmazidi/Machine_Learning

Part Two: Linear Models

3 Linear Regression 47

- 3.1 Overview
- 3.2 Linear Regression in R
- 3.3 Metrics
- 3.4 The Algorithm
- 3.5 Mathematical Foundations
- 3.6 Multiple Linear Regression
- 3.7 Polynomial Linear Regression
- 3.8 Model Fitting and Assumptions
- 3.9 Advanced Topic: Regularization
- 3.10 Summary

4 Logistic Regression 73

- 4.1 Overview
- 4.2 Logistic Regression in R
- 4.3 Metrics
- 4.4 The Algorithm
- 4.5 Mathematical Foundations
- 4.6 Logistic Regression with Multiple Predictors
- 4.7 Advanced Topic: Optimization Methods
- 4.8 Multiclass Classification
- 4.9 Generalized Linear Models
- 4.10 Summary

5 Naive Bayes 97

- 5.1 Overview
- 5.2 Naive Bayes in R
- 5.3 Probability Foundations
- 5.4 Probability Distributions
- 5.5 The Algorithm
- 5.6 Applying the Bayes Theorem
- 5.7 Handling Text Data
- 5.8 Naive Bayes v. Logistic Regression
- 5.9 Naive Bayes from Scratch
- 5.10 Summary

6 Support Vector Machines 119

- 6.1 Overview
- 6.2 SVM in R
- 6.3 The Algorithm
- 6.4 Mathematical Foundations
- 6.5 Kernel Methods and the Gamma Hyperparameter
- 6.6 SVM Regression
- 6.7 Summary

7 The Craft 2: Data Wrangling 131

- 7.1 Data Organization
- 7.2 Data Standardization
- 7.3 Time Data
- 7.4 Text Data

Preface to Part 2

In Part 2 we explore a category of machine learning algorithms called *linear models*, including:

- linear regression
- logistic regression
- naive Bayes
- support vector machines

These algorithms are grouped together in Part 2 because they take a linear combination of inputs to estimate either a target output variable or a linear decision boundary for a given input data set $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$:

$$y = \mathbf{w}\mathbf{X} + b. \tag{2.1}$$

where \mathbf{w} represents weights that are multiplied by the \mathbf{X} input features plus the fitting parameter b . The parameters w and b are learned from the data D .

Throughout Part 2 of the book, keep in mind that *linear* does not always mean a straight line, but any line that can be created with a linear combination of inputs. For example, $y = x^2$ is a line, but not a straight line.

In linear regression, the target output y will be a real-number value. We call this kind of machine learning *regression*. In support vector machines (SVM), the target output y will be an indicator for membership in one of a finite number of classes. We call this kind of machine learning *classification*. In logistic regression and naive Bayes, the output is the *probability* of membership in one of a finite number of classes.

This book begins with linear models because they have visual, intuitive explanations and are commonly used in machine learning tasks.

3. Linear Regression

3.1 Overview

Many machine learning algorithms have their origin in the field of statistics, and linear regression is a prime example. In fact, the fundamentals of linear regression can be traced back to mathematicians in the early 1800s. We begin with linear regression because it is relatively simple, yet powerful, and introduces foundational concepts and terminology we will use for other algorithms throughout the book. In linear regression, our data consists of predictor values, x , and target values y . We wish to find the relationship between x and y . This linear relationship can be defined by parameters w and b , with w , the slope of the line, quantifying the amount that y changes with changes in x , and b serving as an intercept.

3.2 Linear Regression in R

Let's take a look at the `women` data set, one of the built-in data sets in R.

Code 3.2.1 — women data. Height in inches and weight in pounds for 15 women.

```
> str(women)
'data.frame': 15 obs. of 2 variables:
 $ height: num  58 59 60 61 62 63 64 65 66 67 ...
 $ weight: num  115 117 120 123 126 129 132 135 139 142 ...
> plot(women$weight~women$height)
```

This data set is from the mid-1970s, when Americans were considerably thinner than we are now. Figure 3.1 reveals a linear trend in the data: taller women weigh more than shorter women.

We will use the built-in `lm()` function to build a linear regression model. There are many parameters we can send to the `lm()` function, here we only send two: the formula,

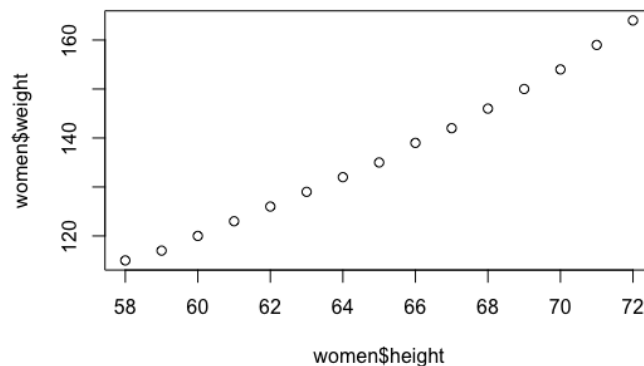


Figure 3.1: Women Data Set

and the data. The formula *weight ~ height* says to model weight, our target, as a function of one predictor: height.

Code 3.2.2 — Build a linear regression model. You can type `?lm` at the console to learn more about the function.

```
> lm1 <- lm(weight~height, data=women)
> lm1

Call:
lm(formula = weight ~ height, data = women)

Coefficients:
(Intercept)      height
    -87.52         3.45
```

When we built the model, we saved it to variable `lm1`. If we type the model name at the console as we see in the code above, we get basic information about the model. Our parameters (coefficients) are the w and b that the algorithm learned from the data. In this model, they are $w = 3.45$, and $b = -87.52$. The intercept parameter, b , is generally not of interest to us as it is just used to fit the data. The parameter w is of more interest. In linear regression, it tells us how much we can expect the y -value to change for every one-unit change in the x -value. So for every inch taller a woman is, we expect her to weight 3.45 pounds more. We can use these values for prediction. Let's say we have a woman who is 65 inches tall. Her weight should be: $65 * 3.45 - 87.52 = 136.7$. Check if that value makes sense given the data in Figure 3.1.

We have a model but we don't know if it is a good model. We can use the `summary()` function to get a glimpse of the goodness of fit achieved by this model. This is shown in the next code block. We will delve deeper into the `summary()` output later but now we will just point out two key elements of the output. Notice the three asterisks at the end of the line for height under Coefficients. This indicates that height was a good predictor. Notice

also that the intercept received three asterisks, but again, we are not really interested in the intercept. The second thing to notice is the R-squared is about 0.99. The R-squared is a measure of goodness of fit that ranges from 0 to 1, the closer to 1 the better. This provides evidence that we have a good model for this data.

Code 3.2.3 — Model summary. Use the `summary()` function to learn more about the model.

```
> summary(lm1)

Call:
lm(formula = weight ~ height, data = women)

Residuals:
    Min       1Q   Median       3Q      Max
-1.7333 -1.1333 -0.3833  0.7417  3.1167

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept) -87.51667     5.93694  -14.74 1.71e-09 ***
height       3.45000     0.09114   37.85 1.09e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.525 on 13 degrees of freedom
Multiple R-squared:  0.991, Adjusted R-squared:  0.9903
F-statistic: 1433 on 1 and 13 DF,  p-value: 1.091e-14
```

Check Your Understanding 3.1 — Building a Simple Linear Regression Model. Using the built-in data set `swiss`. Try the following in an R script:

- Learn more about the data by typing `?swiss` at the console.
- Load the data and look at it with R data exploration functions and at least one plot.
- Build a linear regression model predicting Fertility based on Education.
- What is the mean of Education? of Fertility?
- What is the predicted Fertility, given the mean of Education? Use the coefficients from your model.
- Do you think Education is a good predictor? Why or why not?
- Do you think this is a good model? Why or why not?

3.3 Metrics

There are a lot of metrics associated with linear regression and we will start with metrics we can use in data analysis before we begin applying a machine learning algorithm.

3.3.1 Metrics for Data Analysis

In linear regression we often want to know if variables are correlated. We can use the `cor()` function or `plot.pairs()` as shown in Section 2.4 earlier. For example, to see if x and y are correlated we can use this code: `cor(x, y)`.

The default method is Pearson's, which measures the linear correlation between two variables. It ranges from -1 to $+1$ where the former is a perfect negative correlation, the latter is a perfect positive correlation, and values close to 0 indicate little correlation. The formula for Pearson's correlation is:

$$\rho_{x,y} = \text{Corr}(x,y) = \frac{\text{Cov}(x,y)}{\sigma_x \sigma_y} \quad (3.1)$$

We see that correlation is actually covariance, scaled to $[-1, 1]$. Covariance measures how changes in one variable are associated with changes in a second variable. The numbers can range wildly which is why the scaled correlation is often preferred. Here is the formula for covariance, where n is the number of data points:

$$\text{cov}(x,y) = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{n-1} \quad (3.2)$$

3.3.2 Linear Regression Metrics for Model Fit

The `summary()` output of a linear model gives a lot of useful metrics concerning the model fit. Looking back at the output of `summary()` for the linear model at the Coefficients section:

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|-------------|-----------|------------|---------|--------------|
| (Intercept) | -87.51667 | 5.93694 | -14.74 | 1.71e-09 *** |
| height | 3.45000 | 0.09114 | 37.85 | 1.09e-14 *** |

The estimated coefficient for height and the intercept are given, along with standard error, t value and p value. The standard error gives us an estimate of variation in the coefficient estimate and can be used to predict a confidence interval for the coefficient. So the confidence interval for w_1 would be: $w_1 \pm 2 \text{SE}(w_1)$. Standard errors are used for the hypothesis test on the coefficient, where the null hypothesis is that there is no relationship between the predictor variable and the target variable. In other words, the true $w = 0$. This is computed using the t-statistic:

$$t = \frac{\hat{w}_1 - 0}{\text{SE}(\hat{w}_1)} \quad (3.3)$$

which measures the number of standard deviations our estimate coefficient \hat{w}_1 is from 0. Notice we put the hat symbol, $\hat{\cdot}$, over w to remind us that it is an estimate. The distribution of the t-statistic has a bell shape which makes it easy to compute the probability of observing a t-statistic larger in absolute value than what was computed, if the null hypothesis were true. This is the p-value. If the p-value is small we can reject the

null hypothesis. Typical cut-off points for the p-value are 0.05 and 0.01. One caveat about p-values is that generally you will have more confidence in them if your data size is greater than 30. That is not the case for the women data because it has only 15 observations.

The final part of the summary() output of the women linear regression model is:

```
Residual standard error: 1.525 on 13 degrees of freedom
Multiple R-squared:  0.991, Adjusted R-squared:  0.9903
F-statistic: 1433 on 1 and 13 DF, p-value: 1.091e-14
```

Whereas the metrics for the coefficients indicate how well each coefficient modeled the true data, these statistics tell us how well the model as a whole fit the training data. The RSE, residual standard error, is computed from the RSS, residual sum of squares.

$$RSS = e_1^2 + e_2^2 + \dots + e_n^2 = \sum_i (y_i - \hat{y}_i)^2 \quad (3.4)$$

The RSS is just the sum of squared errors. We square them because some will be errors in the positive direction and some will be in the negative direction. The RSE is computed from the RSS:

$$RSE = \sqrt{\frac{1}{n-2}RSS} = \sqrt{\frac{1}{n-2} \sum_i (y_i - \hat{y}_i)^2} \quad (3.5)$$

Why is RSS scaled by $\frac{1}{n-2}$? The value n is the number of observations and the 2 is because we have 2 estimated variables. This gives us $15 - 2 = 13$ degrees of freedom mentioned on the line with the RSE. The RSE measures how off our model was from the data, the lack of fit of the model. It is measured in units of y so in this case our RSE of 1.525 is about 1.5 pounds.

Since RSE is in terms of Y it can be hard to interpret. For this reason the R^2 statistic is also provided:

$$R^2 = 1 - \frac{RSS}{TSS} \quad (3.6)$$

where TSS, total sum of squares, is a measure of how far off y values tend to be from the mean:

$$TSS = \sum (y_i - \bar{y})^2 \quad (3.7)$$

The R^2 statistic will always be between 0 and 1, the closer to 1 the more variance in the model is explained by the predictors. The R^2 is the summary above was 0.991, which is very high. This means that almost all the variation in weight is predicted by height. For linear regression models such as this one where there is only one predictor, R^2 will be the same as the squared correlation between the X and Y values.

The final statistic listed is the F-statistic:

$$F = \frac{(TSS - RSS)/p}{RSS/(n - p - 1)} \quad (3.8)$$

where n is the number of observations and p is the number of predictors. The F statistic takes into account all of the predictors to determine if they are significant predictors of Y . It provides evidence against the null hypothesis that the predictors are not really predictors. The advantage of the F-statistic over R^2 is that R^2 does not tell us whether it is statistically significant or not but the F-statistic does. It has an associated p-value. So we check for a F-statistic greater than 1 and a low p-value to indicate confidence in the model.

3.3.3 Metrics for Test Set Evaluation

Earlier we ran the women data set through the linear regression algorithm and looked at some metrics for judging the quality of the model. In practice, we will not run entire data sets through the algorithm, but divide the data into a training set and a test set. The model should be built on the training data and should not see the test data until evaluation time. Then the model is used to predict values for the test data, and we can use various metrics to see how far off the predictions were from the true values. We will do that in future examples. For now, we are going to just make up some test data out of the blue, making sure that the data does not fit well to the regression line so it shows up well on the graph in Figure 3.2. Again, we are only making up data for illustration purposes.

What are some metrics we can use to evaluate the prediction accuracy? For regression tasks, common metrics include mean-squared error and correlation. Imagine that we randomly selected a few data observations to hold out from the training data. These observations would not have been seen by the algorithm during training and so we can use them to test the model. It is very important that the algorithm be tested on unseen data, otherwise we don't know if the algorithm learned anything or just memorized data. Now using our test data to predict weight, given height, we can compute the correlation of the actual y values with the predicted y values with the `R cor()` function: `cor(predicted, actual)`. Correlation gives us a general idea about our model. A correlation close to +1 would mean that as height went up, weight went up as well. However, this would not tell us how much our estimates were off for the individual observations. These errors are called residuals: `residuals <- predicted - actual`

In Figure 3.2 these residuals are represented as vertical lines drawn from the data points to the regression line. Some errors may be in the positive direction and some may be in the negative direction. For this reason they are squared to find mean squared error:

$$mse = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (3.9)$$

This averages the squared difference between the actual values (y) and the predicted values (\hat{y}) over all elements in the test set. Sometimes the square root of the above is used, `rmse` (root mean squared error), since it will be in units of y . The metrics `mse` or `rmse` are useful in comparing two models built on the same training data.

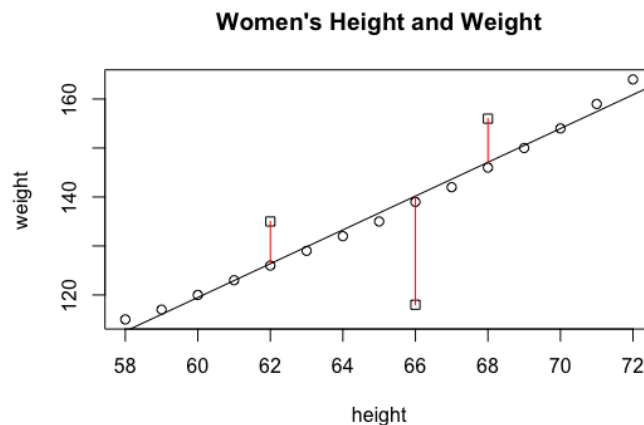


Figure 3.2: Test Set Errors

We see in Code 3.3.1 that our correlation is 0.38 which is not that good, unsurprisingly, since we purposely made up data that was far from the regression line. As you can see in Figure 3.2, the 3 test examples, shown as squares, are not that close to the regression line. Two are above the line and one is below. Red lines show the residuals, the errors. We quantify the amount of error with mse and rmse. The mse value can be hard to interpret in isolation; it is most useful in comparing models. The rmse however is in units of y. In this case, we see that our test data was off by 14.67 pounds on average. Again, it should be stressed that we normally don't hallucinate test data. We normally take our data and divide it into train and test sets.

Code 3.3.1 — Hallucinated Test Data. Women Height-Weight Model.

```
# fake test data
test <- women[c(5, 9, 11),]
test[1, 2] <- 135
test[2, 2] <- 118
test[3, 2] <- 156
# predictions on test data
pred <- predict(lm1, newdata=test)
# metrics
correlation <- cor(pred, test$weight)
print(paste("correlation: ", correlation))
mse <- mean((pred - test$weight)^2)
print(paste("mse: ", mse))
rmse <- sqrt(mse)
print(paste("rmse: ", rmse))
[1] "correlation: 0.38404402702441"
[1] "mse: 215.284722222223"
[1] "rmse: 14.6725840335717"
```

Check Your Understanding 3.2 — Exploring Metrics. Using the linear regression model you built on the swiss data, try the following. For many of the following steps you will need to refer back to the formulas in this section.

- Run the R covariance and correlation functions on Education and Fertility and discuss your observations.
- Does the order of the operands matter for these functions?
- Using R, compute the t-value for the estimated Education coefficient, and compare to the summary output for the model.
- Using R, compute rss, rse, and r-squared for the model, and compare to the summary output for the model.
- Make a test set of 5 randomly chosen observations from the swiss data set. Note: This is cheating! We never let the algorithm see the test data before evaluation. However, we are just using the "test" set to compute some more metrics.
- Make predictions on Fertility for this "test" data.
- Print the test and predicted values side-by-side using print() and cbind(). Are they as close in value as you would expect?
- Run cor() and compute mse on the predicted versus actual data.
- Compute the rmse and compare this to the mean of the residuals. Are they close in absolute value? Is this surprising? Why or why not?

3.4 The Algorithm

The example above is called *simple linear regression* because it had only one predictor variable, height, for the target variable weight. When we have more than one predictor variable as we will see below, this is called *multiple linear regression*. In general, the more predictors are added to a model, the better it will be, but that does not mean you should just add all the predictors. We will learn techniques for determining which predictors should be included as we go.

Recall that the coefficients for the simple linear regression model above were $w = 3.45$ and $b = -87.511667$. Therefore, looking at an observation where height is 64 inches, we would expect weight to be:

$$3.45 * 64 - 87.511667 = 133.28$$

and we see from Figure 3.2 that this is a reasonable estimate of weight for this height. These parameters w and b were learned from the training data, but how? We will dig deeper into the math below, but the general idea is that we want a line that minimizes the residuals, the errors, designated as e :

$$\operatorname{argmin}_{w,b} \sum_{i=1}^n e_i^2 = \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (3.10)$$

The above equation states that we want the parameters w , b that minimize the squared errors over all n examples in the training data. The estimated values of w and b are:

$$\hat{b} = \bar{y} - \hat{w}\bar{x} \quad \hat{w} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2} \quad (3.11)$$

As seen above, these equations rely on mean values of x and y to compute w and b . To prove to ourselves that these equations find the parameters w , b , let's find the mean of x and y in the women's data and plug them into the equations.

Code 3.4.1 — Verifying the equations. Manually Computing Coefficients.

```
x <- women$height
y <- women$weight
x_mean <- mean(women$height)
y_mean <- mean(women$weight)

w_hat <- sum((x-x_mean)*(y-y_mean)) / sum((x-x_mean)^2)
b_hat <- y_mean - w_hat * x_mean
print(paste("w and b estimates = ", w_hat, b_hat))

[1] "w and b estimates = 3.45 -87.5166666666667"
```

Check Your Understanding 3.3 — Verify the equations. Using the formulas in this section in R, verify the coefficients for the linear model you built on the swiss data. Comment on the role of means in linear regression. ■

The algorithm described above for linear regression is called the **ordinary least squares (OLS) method**. Next we explore how these equations are derived.

3.5 Mathematical Foundations

Our goal in the OLS method is to reduce the errors over all the training data. These errors are quantified in the residual sum of errors, RSS:

$$RSS = e_1^2 + e_2^2 + \dots + e_n^2 \quad (3.12)$$

Each error, e , in turn is the difference between the actual y value and the predicted value:

$$RSS = (y_1 - b - wx_1)^2 + (y_2 - b - wx_2)^2 + \dots + (y_n - b - wx_n)^2 \quad (3.13)$$

The **loss function** describes how much accuracy we lose in our model. The first equation below specifies the loss for one example, the second averages the loss over all the examples. By the way, you will also see this called a **cost function**. The terms loss function, cost function, and error function are used somewhat synonymously, but unfortunately inconsistently across the literature. In the equations below we see the loss function with subscript i to indicate the loss for one instance, and the loss function without the subscript indicates the loss averaged over all examples.

$$\mathcal{L}_i = (y_i - f(x_i))^2 \quad (3.14)$$

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^n (y_i - f(x_i))^2 \quad (3.15)$$

Our goal is to find the parameters (coefficients) that minimize these errors on the training data:

$$\operatorname{argmin}_{w,b} \sum_{i=1}^n e_1^2 = \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (3.16)$$

One method for finding the coefficients is to take the partial derivatives of the loss function, set them to zero, and solve. This produces the normal equations given in the last section. Here we show the details of how they are derived. The equations above express the loss function in algebraic notation. We could find the derivative with this notation but it is more concise if we use matrix notation. We are now going to consider parameter b to be one of possibly many parameters (in this case only two) in a vector. Specifically, we will refer to parameter b as w_0 . Additionally we will express x as a vector, making the first element 1 so that it multiplies by w_0 , the intercept.

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \end{bmatrix}$$

$$f(x) = \mathbf{w}^T \mathbf{x} = w_0 + w_1 x_1$$

In the notation above, only one predictor is used. For multiple predictors, the w and x vectors would expand accordingly. Let's visualize this matrix representation of $f(x)$ for the women data set.

$$\mathbf{y} \begin{bmatrix} 115 \\ 117 \\ 120 \\ 123 \\ 126 \\ \dots \\ 154 \\ 159 \\ 164 \end{bmatrix} = \mathbf{w} \begin{bmatrix} -87.5167 \\ 3.45 \end{bmatrix}^T \mathbf{X} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & \dots & 1 & 1 & 1 \\ 58 & 59 & 60 & 61 & 62 & \dots & 70 & 71 & 72 \end{bmatrix}$$

Our loss function expressed in matrix notation, with i indexing individual observations:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^n (\mathbf{y}_i - \mathbf{w}^T \mathbf{x}_i)^2 \quad (3.17)$$

Rewriting the square term above into the form shown below uses the property that $(\mathbf{X}\mathbf{w})^T = \mathbf{w}^T \mathbf{X}^T$:

$$\mathcal{L} = \frac{1}{N}(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w}) \quad (3.18)$$

Next we go through a few steps to multiply out and collect terms. First, bring the transpose inside the parenthesis.

$$\mathcal{L} = \frac{1}{N}(\mathbf{y}^T - (\mathbf{X}\mathbf{w})^T)(\mathbf{y} - \mathbf{X}\mathbf{w}) \quad (3.19)$$

Multiply out:

$$\mathcal{L} = \frac{1}{N}\mathbf{y}^T\mathbf{y} - \frac{1}{N}\mathbf{X}\mathbf{w}\mathbf{y}^T - \frac{1}{N}(\mathbf{X}\mathbf{w})^T\mathbf{y} + \frac{1}{N}(\mathbf{X}\mathbf{w})^T\mathbf{X}\mathbf{w} \quad (3.20)$$

We can combine the middle two terms because $\mathbf{w}^T \mathbf{X}^T \mathbf{y}$ and $\mathbf{y}^T \mathbf{X} \mathbf{w}$ are transposes of one another and scalars.

$$\mathcal{L} = \frac{1}{N}\mathbf{y}^T\mathbf{y} - \frac{2}{N}\mathbf{w}^T \mathbf{X}^T \mathbf{y} + \frac{1}{N}\mathbf{w}^T \mathbf{X}^T \mathbf{X} \mathbf{w} \quad (3.21)$$

Before we take the partial derivative wrt \mathbf{w} , we can get rid of the first term since it doesn't involve \mathbf{w} .

$$\mathcal{L} = -\frac{2}{N}\mathbf{w}^T \mathbf{X}^T \mathbf{y} + \frac{1}{N}\mathbf{w}^T \mathbf{X}^T \mathbf{X} \mathbf{w} \quad (3.22)$$

Now use the rules for matrix partial derivatives to get:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = -\frac{2}{N}\mathbf{X}^T \mathbf{y} + \frac{2}{N}\mathbf{X}^T \mathbf{X} \mathbf{w} \quad (3.23)$$

Above we used the fact that the partial derivative of $\mathbf{w}^T \mathbf{x} = x$ for the first term and $\mathbf{w}^T \mathbf{w} = 2\mathbf{w}$ for the second term. We simplify the equation to:

$$\mathbf{X}^T \mathbf{y} = \mathbf{X}^T \mathbf{X} \mathbf{w} \quad (3.24)$$

And so our estimated parameters must be:

$$\hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad (3.25)$$

In practice, finding the inverse matrix above is computationally intense, $O(n^3)$, and therefore very slow for large data sets. For this reason, R uses mathematical techniques to find the parameters.

3.5.1 Gradient descent

Unlike directly finding the inverse in the normal equation above, gradient descent will not get bogged down for large data sets. The algorithm starts with some value for the parameters \mathbf{w} and keeps changing them in an iterative loop until they find a minimum. Figure 3.3 visualizes a convex function with a random starting point symbolized by the higher red dot. One iteration may move the red dot to the second location. This is one step.

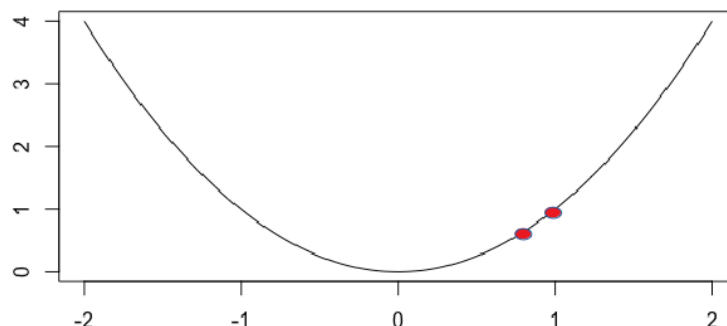


Figure 3.3: Searching for Error Minimum

The gradient descent algorithm repeats the parameter update until convergence:

$$w_j := w_j - \alpha \frac{\partial \mathcal{L}}{\partial w_j} \quad (3.26)$$

Note that all parameters in \mathbf{w} are updated at every iteration. The derivative gives us the slope and the alpha determines our step size. If alpha is too small, the algorithm will be slow. On the other hand, if it is too large we could overshoot the minimum and fail to converge.

Check Your Understanding 3.4 — Mathematical Foundations. Use what you have learned in this section to answer the following questions. A good reference for R matrix operations is: <https://www.statmethods.net/advstats/matrix.html>

- What is the purpose of a loss or cost function?
- Is gradient descent guaranteed to find the optimal parameter? Why or why not?
- Using R matrix operations, compute the \mathbf{w} matrix of coefficients using formula 3.25 above.

3.6 Multiple Linear Regression

Next we look at another example of linear regression in R. We will use the R build-in data set `ChickWeight`, which has 578 rows and 4 columns of data resulting from an experiment on the effect of different types of feed on chick weight. We will use `weight` as our target, with the following predictors:

- `Time` - number of days since birth
- `Diet` - a factor representing 4 different diets

We will ignore the Chick column which identifies the chicken. In **simple linear regression** we use only one predictor. In **multiple linear regression** we use more than one predictor. We will do both on this data set. First, let's make a couple of plots to visualize the data.

Code 3.6.1 — Plots. Use `par()` to set up a 1x2 grid for the plots.

```
par(mfrow=c(1,2))
plot(ChickWeight$Time, ChickWeight$weight,
     xlab="Time", ylab="Weight")
plot(ChickWeight$Diet, ChickWeight$weight,
     xlab="Diet", ylab="Weight")
```

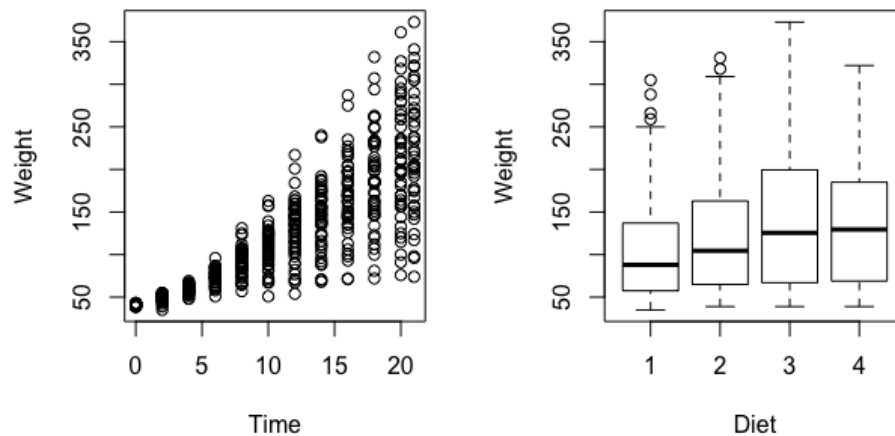


Figure 3.4: ChickWeight Weight Impacted by Time, Diet

In Figure 3.4 we see in the plot on the left that chicks gain weight over time. That is not surprising but what the plot also shows is that chicks start off at near identical weights and diverge over time. The plot on the right shows the impact of diet on weights. Box plots show the median value as the horizontal line through the box, the box itself shows the IQR (inter-quartile range from the first to the third quartile), and the horizontal lines at the end of the dashed lines show minimum and maximum values, not including suspected outliers which appear as dots beyond the horizontal lines. In this example it appears that diet 4 is slightly better than diet 3 and diet 1 appears to result in the lowest weights for chicks.

Next we divide the data into train and test sets by randomly sampling the rows. We set a seed so that each time we run this code we should get the same results. Vector `i` will contain the row numbers so we can subset the data frame with indices `i` to get a train set and *not* `i`, or `-i` to get the test set. After this 75/25 split, the train set has 433 observations and the test set has 145.

Code 3.6.2 — Divide data into train and test sets. Made a 75/25 split.

```
set.seed(1234)
i <- sample(1:nrow(ChickWeight), nrow(ChickWeight)*0.75,
            replace=FALSE)
train <- ChickWeight[i,]
test <- ChickWeight[-i,]
```

The `sample()` function as we used it above has this form: `sample(x, size, replace = FALSE)`. The first argument is a vector of elements from which to choose, in our case from 1:578, the row numbers of the data frame. The second argument specifies the size; in our case we want 75% of the row numbers. The last argument indicates that sampling should be done without replacement. To learn more about this function type `?sample()` at the console.

3.6.1 Interpreting `summary()` output for a linear model

Next we create a linear model on the train set, using only Time as a predictor. Let's look at the output of `summary()` in more detail. First it echoes back the code used to build `lm1`.

Next we see the distribution of the residuals, the errors. We want to see that the residuals are symmetrically distributed around the mean or median. There is a wide range here, from the minimum of -140 to the maximum of 158. That is not encouraging. The wide range of residuals confirms what we saw in Figure 3.4, that chick weight diverges widely as time goes on.

The Coefficients section lists for each predictor and the intercept, the estimated value, standard error, t value and p value, followed by significance codes. If you recall from an earlier statistics class, a t-value measures variation in the data, and the associated p-value estimates confidence in that value. A low p-value indicates evidence for rejecting the null hypothesis that the predictor does not influence the target variable. We want to see low p-values and in this case we do. Time has a low p-value and correspondingly, 3 asterisks. The estimate for our one predictor, Time, is 8.952. This means we would expect chicks to gain an average of almost 9 gm a day. The standard error measures the average amount the coefficient estimate varies from the actual values. The t-value is a measure of how many standard deviations the coefficient estimate was from 0. The further it is from 0, the more confidence we have in rejecting the null hypothesis, in this case that Time has no effect on chick weight. The p-value gives the probability of observing a similar or larger t-value due to chance, given the data. A small p-value gives us confidence that there really is a relationship between our predictor(s) and the target variable. The chart for the significance codes is given at the bottom of this section.

The last section gives some statistics on the model. The residual standard error, RSE, is in units of y. In this case our RSE was 41.4, so the average error of the model was about 41 gm. This statistic was calculated on 431 degrees of freedom: we had 433 data points minus 2 predictors. Multiple R-squared is scaled from 0 to 1 and so is easier to interpret than RSE. The adjusted R-squared is 0.6863 which is not bad. This means that 68% of the variance in the model can be explain by our predictor. The adjusted R-squared takes into account the number of predictors. This is important because R-squared tends to increase as we add predictors, and the adjusted R-squared accounts for this. Finally the F-statistic also measures whether our predictors and target are related. We want to see the F-statistic

be far away from zero and its associated p-value to be low. The more data observations we have, the lower the F-statistic can be to confirm the relationship. This is why the p-value is important.

Code 3.6.3 — Linear model 1. Using only Time as a predictor.

```
lm1 <- lm(weight~Time, data=train)
summary(lm1)
```

Call:

```
lm(formula = weight ~ Time, data = train)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|----------|---------|--------|--------|---------|
| | -140.314 | -16.648 | 0.778 | 14.682 | 158.686 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|-------------|----------|------------|---------|--------------|
| (Intercept) | 26.318 | 3.743 | 7.031 | 8.06e-12 *** |
| Time | 8.952 | 0.291 | 30.760 | < 2e-16 *** |

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 41.4 on 431 degrees of freedom

Multiple R-squared: 0.687, Adjusted R-squared: 0.6863

F-statistic: 946.2 on 1 and 431 DF, p-value: < 2.2e-16

3.6.2 Residuals

Plotting the residuals results in 4 plots, which we have arranged in a 2x2 grid in Figure 3.5. How do we interpret these plots? A comprehensive explanation is given here: <http://data.library.virginia.edu/diagnostic-plots/>.

Code 3.6.4 — Plot the residuals. The residuals give us information about how well the model fits the data.

```
par(mfrow=c(2,2))
plot(lm1)
```

Below we provide a brief overview of the 4 plots:

1. Plot 1 Residuals vs Fitted: This plots the residuals (errors) with a red trend line. You want to see a fairly horizontal red line. Otherwise, the plot is showing you some variation in the data that your model did not capture.
2. Plot 2 Normal Q-Q: If the residuals are normally distributed, you will see a fairly straight diagonal line following the dashed line.
3. Plot 3 Scale-Location: You want to see a fairly horizontal line with points distributed equally around it. If not, your data may not be homoscedastic (means "same variance").

4. Plot 4 Residuals vs Leverage: This plot will indicate leverage points which are influencing the regression line. They may or may not be outliers, but further investigation is warranted. An **outlier** is a data point with an unusual y value whereas a **leverage point** is a data point with an unusual x value.

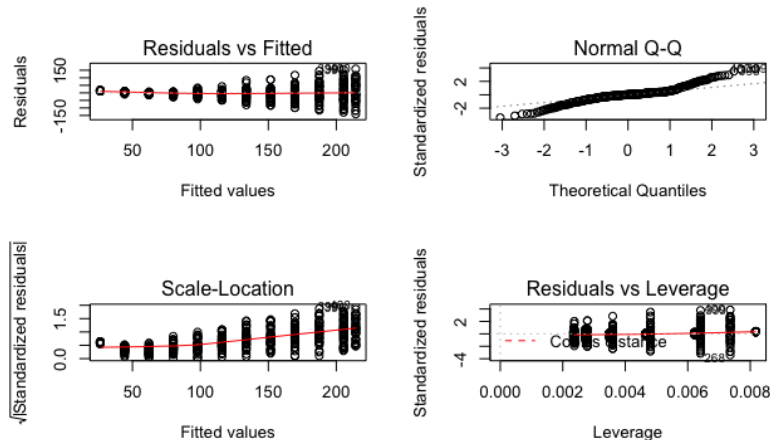


Figure 3.5: Residuals Plot

Looking at Figure 3.5 we see some problems with our model. In the first graph, the red line is horizontal but notice that the residuals vary more as we go to the right. This confirms our very first observation of the data, that chicks vary greatly in how much weight they gain. Time and even diet cannot account for this. What is not accounted for in our model is the genetic contribution. We humans start off life at an average of about 7 pounds but end up as adults in a wide range of weights. We could consider genetics a hidden or unseen variable in this experiment. Also chick gender was not included in the data, so we have no way of knowing if this influenced the weight variation. The second graph indicates that most of the residuals are normally distributed except for those in the lower range. So there is variation in the data that our model does not capture.

Let's build another model using Time and Diet as predictors. The adjusted R-squared for `lm2` is 0.7338 which is higher than the adjusted R-squared for `lm1`, which was 0.6863. Also, RSE has decreased to 38.13 from 41.1 in `lm1`. Adding Diet seems to have improved our model.

Code 3.6.5 — Linear model 2. Using Time and Diet as predictors.

```
lm2 <- lm(weight~Time+Diet, data=train)
summary(lm2)
```

Call:

```
lm(formula = weight ~ Time + Diet, data = train)
```

Residuals:

| | Min | 1Q | Median | 3Q | Max |
|--|----------|---------|--------|--------|---------|
| | -137.857 | -20.492 | -1.685 | 16.955 | 137.365 |

Coefficients:

| | Estimate | Std. Error | t value | Pr(> t) |
|-------------|----------|------------|---------|--------------|
| (Intercept) | 8.4109 | 4.1372 | 2.033 | 0.042670 * |
| Time | 8.9086 | 0.2682 | 33.218 | < 2e-16 *** |
| Diet2 | 16.3645 | 4.9235 | 3.324 | 0.000965 *** |
| Diet3 | 40.1424 | 4.8907 | 8.208 | 2.67e-15 *** |
| Diet4 | 32.1873 | 5.2503 | 6.131 | 1.99e-09 *** |

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 38.13 on 428 degrees of freedom

Multiple R-squared: 0.7363, Adjusted R-squared: 0.7338

F-statistic: 298.8 on 4 and 428 DF, p-value: < 2.2e-16

3.6.3 Dummy variables

Looking at the summary output for `lm2`, we see that `Time` and `Diet` were good predictors. Why do we have `Diet2`, `Diet3`, `Diet4`? Since `Diet` is a factor with 4 levels, R made 3 **dummy variables** for us. The base model represents `Diet1`, the dummy variables `Diet2` - `Diet 4` tell us how much each diet impacted the model compared to `Diet 1`. Recall from the boxplot above, that `Diet1` resulted in the lowest weights and we see that confirmed in this summary, because `Diet2` - `Diet4` each have positive coefficients. For each data observation, only one of the dummy variables will be active with the others being zero. So for a chick on `Diet 1`, the dummy variables for Diets 2 through 4 would be zero.

3.6.4 The `anova0` function

We can compare the `summary()` statistics of models to gauge their relative value. Another way to compare them is to run `anova()` on the two models. The `anova()` function lists each model and provides similar statistics as the `summary()` function for each model. We see that the RSS is lower for model 2, and model 2 is given a low p-value. This is confirmation that `lm2` outperformed `lm1`.

Code 3.6.6 — The `anova()` function. Analysis of Variance.

```
anova(lm1, lm2)
```

Analysis of Variance Table

Model 1: `weight ~ Time`

Model 2: `weight ~ Time + Diet`

| | Res.Df | RSS | Df | Sum of Sq | F | Pr(>F) |
|---|--------|--------|----|-----------|--------|---------------|
| 1 | 431 | 738546 | | | | |
| 2 | 428 | 622323 | 3 | 116222 | 26.644 | 8.107e-16 *** |

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Let's try one more thing. Recall from Figure 3.5 that there was a funnel shape in the residuals in that they became more spread out from left to right. The `log` function damps down `x` values across the axis. Perhaps this could squish the chick weights closer about the linear regression line.

Code 3.6.7 — Linear model 3. Linear models are not always a straight line.

```
lm3 <- lm(log(weight)~Time+Diet, data=ChickWeight)
summary(lm3)
```

```
Residual standard error: 0.2281 on 573 degrees of freedom
Multiple R-squared: 0.8484, Adjusted R-squared: 0.8474
F-statistic: 802 on 4 and 573 DF, p-value: < 2.2e-16
```

Above we show only the statistics portion of the `summary()` output. Now the R-squared has increased to 0.8474, indicating that `lm3` may be better than `lm1` and `lm2`. We cannot run `anova` on the 3 models because `lm3` has a different response: `log(weight)` instead of `weight`. But we could look at the residuals plots to look for improvement.

Check Your Understanding 3.5 — Multiple Linear Regression. Using the same swiss data, try the following activities:

- Run `cor()` and `pairs()` on `swiss` and discuss any patterns you see.
- Build a linear regression model predicting Fertility from all predictors.
- Build another model just using 2 or 3 predictors of your choice.
- Compare RSE and R-squared for the simple linear regression model and the two models you just created. Which appears best?
- Run `anova()` on the 3 models and discuss the results.
- Plot the residuals on your best model and discuss what you see.

3.7 Polynomial Linear Regression

To emphasize the point that linear regression is not always a straight line, we next look at polynomial linear regression. We will perform polynomial regression on the `cars` dataset, included in R. The data set has 50 observations and 2 variables: speed and stopping distance. Using the `range()` function on the columns we see that speed ranges from 4 to 25 mph, and dist ranges from 2 to 120 feet. The data was collected in the 1920s. The code example is from the R documentation. The `plot()` call at the top of the code sets up the plot. The `seq()` call sets up a sequence for the `s` values we want to plot across the horizontal axis. The `for` loop plots models of degree 1 through 4 in different colors. Colors 1-4 correspond to black, red, green3, blue. The `poly()` function is used to create orthogonal (not correlated) polynomials. Finally an `anova()` is run on the 4 models.

Code 3.7.1 — Polynomial Regression.

```
plot(cars, xlab = "Speed (mph)",
     ylab = "Stopping distance (ft)", xlim = c(0, 25))
s <- seq(0, 25, length.out = 200)
for(degree in 1:4) {
  fm <- lm(dist ~ poly(speed, degree), data = cars)
  assign(paste("cars", degree, sep = "."), fm)
  lines(d, predict(fm, data.frame(speed = s)), col = degree)
}
anova(cars.1, cars.2, cars.3, cars.4)
```


Analysis of Variance Table

```

Model 1: dist ~ poly(speed, degree)
Model 2: dist ~ poly(speed, degree)
Model 3: dist ~ poly(speed, degree)
Model 4: dist ~ poly(speed, degree)

```

| | Res.Df | RSS | Df | Sum of Sq | F | Pr(>F) |
|---|--------|-------|----|-----------|--------|--------|
| 1 | 48 | 11354 | | | | |
| 2 | 47 | 10825 | 1 | 528.81 | 2.3108 | 0.1355 |
| 3 | 46 | 10634 | 1 | 190.35 | 0.8318 | 0.3666 |
| 4 | 45 | 10298 | 1 | 336.55 | 1.4707 | 0.2316 |

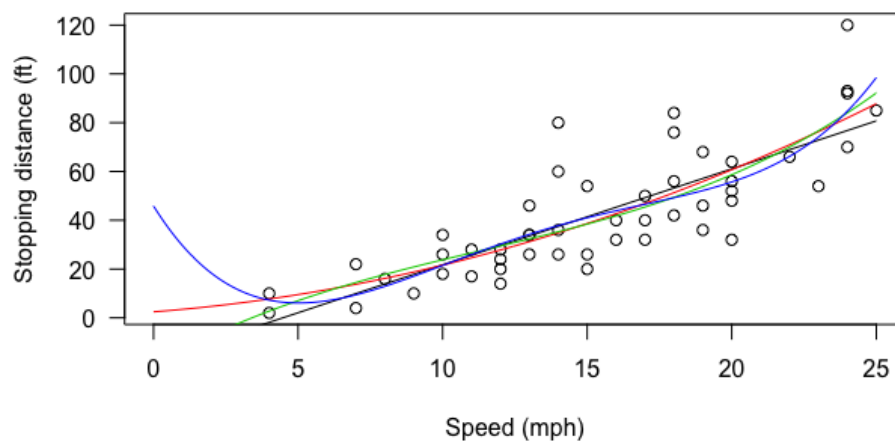


Figure 3.6: Cars with Polynomial Degree 1:4

When we look at the results above, we see that the residuals of Model 2 are less than the degree-1 model. This is an indication that model 2 might be better than model 1. Residuals for models 3 and 4 are lower still, but at some point we worry about overfitting the training data, which we will discuss in the next section.

Check Your Understanding 3.6 — Polynomial Linear Regression. Using the same swiss data, try the following activities:

- Rerun the degree-1 linear regression model predicting Fertility from Education if necessary.
- Build a degree-2 regression model predicting Fertility from Education.
- Build a degree-3 regression model predicting Fertility from Education.
- Run `anova()` on the 3 models and discuss the results.

3.8 Model Fitting and Assumptions

In this section we explore important concepts in machine learning that relate to how well a model fits the data. Overfitting is a common problem in many machine learning algorithms we will learn. And we will discuss the bias-variance tradeoff of various algorithms as we learn them.

3.8.1 Overfitting v. underfitting

The `anova()` results from the polynomial regression indicate the smallest RSS with the degree 4 model. However, none of the p-values are significant, and it is difficult to draw firm conclusions from such a small set of data points. However, the graph in Figure 3.6 gives us an opportunity to talk about underfitting versus overfitting. The linear degree=1 model probably underfits. In contrast the degree 3 and 4 models might be overfitting the data. When you underfit the data, your model does not have sufficient complexity to explain the data. That is what we see with the degree=1 model. The straight line is not capturing some of the complexity in the data. On the other hand, overfitting is when the model has too much complexity. The principle of **Occam's razor** tells us that when choosing between two likely explanations, choose the simpler one. In this case we might choose degree=2 since it did have a lower p-value. In a scenario where you have a train and test set, if the data performs well on the training data but poorly on the test data you may have overfit. Your model tuned itself too much to variation in the training data and this limited its ability to generalize to new data. On the other hand, if your model does poorly on the training set, you may have underfit. Figure 3.7 illustrates overfitting versus underfitting.

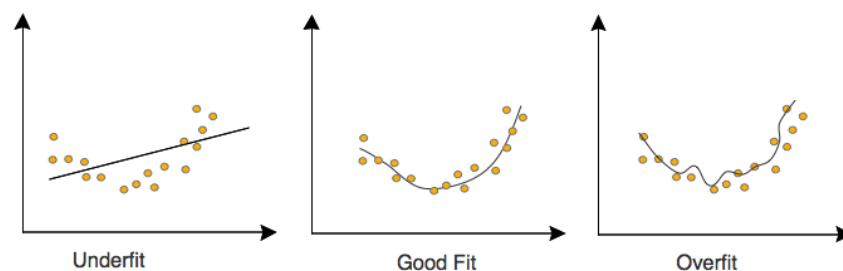


Figure 3.7: Fitting a model

3.8.2 Bias and variance

A common theme throughout this handbook will be the bias-variance tradeoff which is related to underfitting and overfitting. Each algorithm that we learn will have tendencies one way or the other. In the underfitting illustration in Figure 3.7, we see that the straight line underfit. It had a strong bias that the data was truly linear and is too simple a model for the data. On the other extreme, the overfitting example showed what can happen if variance is too high: the model learned random things from the data.

So bias-variance and underfitting-overfitting are related but it is important to distinguish their separate meanings. A high bias, low variance model is likely to underfit and not capture the true shape of the data. This tends to happen more with simpler models like

linear regression or logistic regression. In contrast, a low bias, high variance model captures too much complexity and noise in the data and will not generalize well to new data. This can happen with more complex models like decision trees, SVM, or neural networks.

So if you suspect your model has high bias, what can you do? We could try different algorithms. Right now, that's not helpful because we have only learned one algorithm. As we go we will learn more algorithms and learn their tendencies towards either bias or variance. We will see one tool in this chapter however that can help, and that is adding a regularization term that will help linear regression pull back its tendency towards bias. Also in this chapter we learned that a linear model doesn't necessarily mean a straight line. It can be a polynomial line or any mathematical transformation of the linear equation. One more technique we can try to reduce bias is to add more features, if available.

If you suspect your model is suffering from high variance, what can you do? More data should help. Algorithms that tend toward high variance are overly sensitive to noise in the data. Adding more data can quiet the effect of a few noisy observations. Another approach is to try fewer features if you are using a lot of features. Some features may be noisier than others so this could help.

Check Your Understanding 3.7 — Interaction Effects. In R, we can add interaction effects to formulas using the `*` operator. For example, `y~x1+x2+x1*x2` has three predictors: `x1`, `x2`, and the interaction of `x1` and `x2`. Again using the swiss data, create a linear model with all predictors plus an interaction between Education and Infant.Mortality. Is this interaction a good predictor? Is the model better? ■

3.8.3 Linear Model Assumptions

A linear model first and foremost assumes some linear shape in the data. Beyond that, the linear model also has an **additive assumption**, that each predictor contributes to the model independently of the other predictors. In reality, some predictors may be correlated, in which case we might consider removing one of them, since it will be hard for the model to assess the effect of them independently and thus the coefficient estimates may be erroneous. Other predictors may have an **interaction effect**, a synergy between them. Yet another concern is **confounding** variables, which are variables that correlate with both the target and a predictor. How can we detect these situations? We can use the `cor()` and `pairs()` methods in R to quantify and visualize correlations in the data set.

3.9 Advanced Topic: Regularization

An extension of the least squares approach is to add a regularization term to the RSS, with its importance controlled by another parameter, `lambda`. This extra term penalizes large coefficients. When `lambda=0` it is the same as the least squares estimate. As `lambda` gets larger, the coefficients will shrink. The intercept does not shrink because the goal is to reduce the coefficients associated with predictors. The notation $\|w\|^2$ denotes the l_2 norm, which is $\sqrt{(\sum_{j=1}^p)w_j^2}$. Regularization can help prevent overfitting when you have relatively complex models on a small data set.

$$\text{RSS} = \sum_{i=1}^n (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \lambda \|\mathbf{w}\|^2 \quad (3.27)$$

We can implement ridge regression with the R package `glmnet`. We will use the `airquality` data set. Since `airquality` has a lot of NAs we will omit observations with NAs in the columns we care about with the `complete.cases()` function. Before performing ridge regression, we build a multiple linear regression model as usual.

Code 3.9.1 — Linear Regression. Multiple Linear Regression on Airquality

```
df <- airquality[complete.cases(airquality[, 1:5]),]
df <- df[,-6]
set.seed(1234)
i <- sample(1:nrow(df), .75*nrow(df), replace=FALSE)
train <- df[i,]
test <- df[-i,]
lm1 <- lm(Ozone~., data=train)
pred <- predict(lm1, newdata=test)
mse1 <- mean((pred-test$Ozone)^2)
```

The output above for the mse is 409.3799. Let's see if we can beat that with ridge regression. First we use the `model.matrix()` function to create a matrix of the predictors. Then we split into the same train and test observations as for `lm1`.

Code 3.9.2 — Regularization. Ridge Regression On Airquality

```
library(glmnet)
x <- model.matrix(Ozone~., df)[,-1]
y <- df$Ozone
train_x <- x[i,]
train_y <- y[i]
test_x <- x[-i,]
test_y <- y[-i]

# build a ridge regression model
rm <- glmnet(train_x, train_y, alpha=0)

# use cv to see which lambda is best
set.seed(1)
cv_results <- cv.glmnet(train_x, train_y, alpha=0)
l <- cv_results$lambda.min

# get data for best lambda, which is the 99th
pred2 <- predict(rm, s=l, newx=test_x)
mse2 <- mean((pred2-test_y)^2)
coef2 <- coef(rm)[,99]
```

The mse for the ridge regression was 371.0138, which is about 10% lower than for the regular multiple regression. Let's confirm that the ridge regression shrunk our coefficients. It appears that all the coefficients shrunk a bit.

```
> lm1$coefficients
(Intercept)      Solar.R      Wind      Temp      Month
-66.85709002    0.08314323  -3.75229006   1.98524049  -3.27749222

> coef2
(Intercept)      Solar.R      Wind      Temp      Month
-60.80449134    0.08165752  -3.61256523   1.83183505  -2.60738344
```

3.10 Summary

In this chapter we learned the supervised regression technique of linear regression, where our target was a real number variable and our predictors could be any combination of quantitative or qualitative variables. Linear regression has a strong bias in that it assumes that the relationship between the target and the predictors is linear. Keep in mind that *linear* does not always mean a straight line as we saw in the examples.

When we run the linear regression algorithm on training data, we create a *model* of the data that can then be used for predictions on new data. Our model gives us the coefficients which quantify the effect of each predictor on the target variable. As we explore many more algorithms for regression in the book, we will typically use linear regression as a baseline algorithm to see if other algorithms can beat it. If the data is linear, linear regression is quite hard to beat. Linear regression strengths and weaknesses:

Strengths:

- Relatively simple algorithm with an intuitive explanation because the coefficients quantify the effect of predictors on the target variable.
- Works well when the data follows a linear pattern.
- Has low variance.

Weaknesses:

- High bias because it assumes a linear shape to the data.

3.10.1 Terminology

This chapter introduces a lot of terminology as we explored simple linear regression, multiple linear regression, and polynomial linear regression. There is a glossary at the end of the book but you might want to read back through the chapter again if you are unsure of the meaning of any of the following terms.

Terms related to the data:

- outlier
- leverage point
- dummy variables
- confounding variables

Terms related to the algorithm:

- coefficients
- residuals
- loss function, or cost function
- gradient descent
- additive assumption of linear regression
- interaction effect in linear predictors

Terms related to metrics:

- correlation
- covariance
- mse mean squared error
- rmse root mean squared error
- rss residual sum of squared errors
- rse residual standard error
- R^2 and adjusted R squared
- F-statistic
- p-value

Terms relevant to all machine learning algorithms:

- overfitting
- underfitting
- bias
- variance
- regularization

3.10.2 Quick Reference

Reference 3.10.1 Create Train and Test Sets

```
set.seed(...)
i <- sample(1:nrow(df), nrow(df)*0.8, replace=FALSE)
train <- df[i,]
test <- df[-i,]
```

Reference 3.10.2 Build and Examine a Linear Regression Model

```
lmName <- lm(formula, data=train)
summary(lmName)
```

Reference 3.10.3 Predict on Test Data

```
pred <- predict(lmName, newdata=test)
mse1 <- mean((pred - test$y)^2)
cor1 <- cor(pred, test$y)
```

Reference 3.10.4 Comparing Models with anova()

```
anova(model1, model2, ..., modeln)
```

3.10.3 Lab

Problem 3.1 — Practice on the Abalone Data. Try the following:

1. Download the Abalone data from the UCI Machine Learning Repository, <http://archive.ics.uci.edu/ml/datasets/Abalone>.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Check if there are missing values.
4. Run `cor()` and `pairs()` to see if there are any columns you might consider getting rid of.

5. Divide the data into 80-20 train-test, setting a seed for reproducibility.
6. Create a linear regression model with all predictors. What is the correlation of the predicted and actual values? What is the mse?
7. Create at least 2 more models, trying different features and combinations of features to see if you can improve these results.

3.10.4 Exploring Concepts

Problem 3.2 Based on your experience with the linear regression algorithm in R, does removing predictors with low p-values necessarily improve performance? Discuss possible reasons for your answer.

Problem 3.3 If you found that some predictors had high p-values, what reasons might you give for leaving them in? What reasons might you give for taking them out?

3.10.5 Going Further

There are entire statistics courses devoted to linear regression and scores of books if you have time for a deep exploration. Here are some recommendations:

- Free online linear regression tutorial here: <https://onlinecourses.science.psu.edu/stat501/node/250>
- *Linear Models in Statistics* by Rencher and Schaalje. This book is available online or physical book at the UTD library.
- *Linear Models with R* by Faraway. This book is available online at the UTD library site.

4. Logistic Regression

4.1 Overview

Despite its name, when we use logistic regression, we are performing **classification**, not regression. Whereas in linear regression, our target variable was a *quantitative* variable, in logistic regression, our target variable is *qualitative*: we want to know what class an observation is in. In the most common classification scenario, the target variable is a binary output so that we classify into one class or the other. As we will see later in the chapter, there are techniques that allow classification into more than two classes.

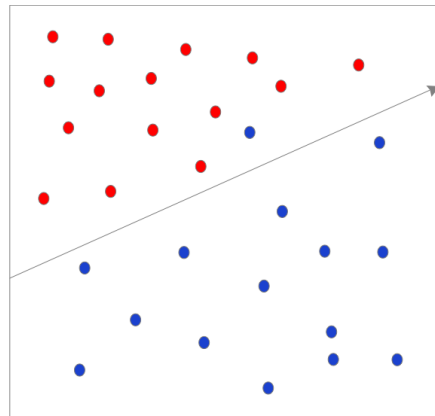


Figure 4.1: Decision Boundary for Binary Classification

Linear models for classification create decision boundaries to separate the observations into regions in which most observations are of the same class. The decision boundary is a linear combination of the X parameters. As we see in Figure 4.1 the two classes are almost perfectly separated by the decision boundary. There is one misclassified observation.

4.2 Logistic Regression in R

Let's take a look at the plasma data set in package HSAUR. This is a blood screening data set for 32 patients that gives measures of two plasma proteins, fibrinogen and globulin, and a binary indicator associated with the two protein levels. The fibrinogen and globulin variables are quantitative. The qualitative variable $\text{ESR} > 20$ indicates whether the erythrocyte sedimentation rate, the rate at which red blood cells settle in blood plasma, is over 20 or not. In logistic regression our target needs to be a qualitative variable. In this data set, $\text{ESR} > 20$ is our target. The $\text{ESR} > 20$ variable has been coded as a factor in R so that $\text{ESR} > 20 = 2$ means that ESR is over 20 and 1 means otherwise. We want to learn to predict whether $\text{ESR} > 20$ or not, based on the levels of the plasma proteins fibrinogen and globulin. Values > 20 indicate some possible associations with various health conditions.

4.2.1 Plotting factor data

Code 4.2.1 shows how the plots in Figure 4.2 were generated. First, we specify a 1x2 layout for the plots, then use the `plot(x, y)` command. The parameter `varwidth=TRUE` makes the boxplot widths proportional to the square root of the samples sizes. This easily lets us see that $\text{ESR} < 20$ is more common than $\text{ESR} > 20$. More importantly, the box plots show that $\text{ESR} > 20$ observations are associated with slightly higher levels of globulin and significantly higher levels of fibrinogen.

Code 4.2.1 — Plotting Factors. Boxplots.

```
par(mfrow=c(1,2))
plot(ESR, fibrinogen, data=plasma, main="Fibrinogen",
     varwidth=TRUE)
plot(ESR, globulin, data=plasma, main="Globulin", varwidth=TRUE)
```

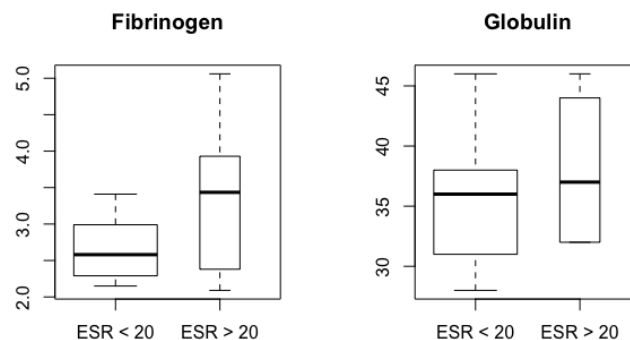


Figure 4.2: Box Plots

Plotting Qualitative Data

If you use the R `plot()` function with command:

```
plot(globulin, ESR, data=plasma)
```

you will see a row of points at $y=1$ and another row of points at $y=0$ on the y axis. This should be your first clue that you need to rethink your plot. The plot function expects the data in (x, y) order. In Code 4.2.1 above we have globulin as x and ESR as y so we get the box plot as shown in figure 4.2.

In Code 4.2.2 we make two conditional density (CD) plots, shown in Figure 4.3. We can make the same observations as we did when looking at the box plots. Here they are just visualized differently. The total probability space is the rectangle, with the lighter grey indicating $\text{ESR} > 20$.

Code 4.2.2 — Plotting Factors. CD Plots.

```
par(mfrow=c(1,2))
cdplot(ESR~fibrinogen)
cdplot(ESR~globulin)
```

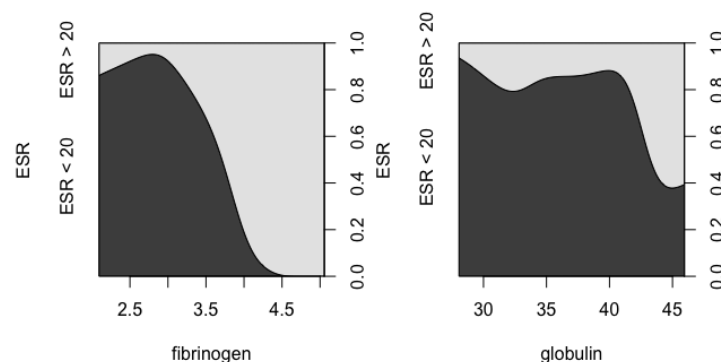


Figure 4.3: Conditional Density Plots

4.2.2 Train and Test on the Plasma Data

Even though this data set is very small, only 32 observations, we divided it into train and test sets, then created a logistic regression model using fibrinogen to predict $\text{ESR} > 20$. The full notebook is available online.

Code 4.2.3 — Logistic Regression. Using glm().

```
set.seed(1234)
i <- sample(1:nrow(plasma), 0.75*nrow(plasma), replace=FALSE)
train <- plasma[i,]
test <- plasma[-i,]
glm1 <- glm(ESR~fibrinogen, data=train, family=binomial)
summary(glm1)
```

For logistic regression we use the `glm()` *generalized* linear function instead of `lm()` that we used for linear regression. Also we need the parameter `family=binomial` for

logistic regression. Otherwise the `glm()` function call looks similar to what we have done previously for `lm()`. The first argument is the formula, which seeks to learn the target ESR with one predictor, fibrinogen.

You will notice that the output of the `summary()` function is very similar to the output we saw for linear regression, with 4 sections:

- the `glm()` call
- the residual distribution
- the coefficients with statistical significance metrics
- metrics for the model

Here is the summary output:

Call:

```
glm(formula = ESR ~ fibrinogen, family = binomial, data = train)
```

Deviance Residuals:

| Min | 1Q | Median | 3Q | Max |
|---------|---------|---------|---------|--------|
| -1.0112 | -0.4648 | -0.3517 | -0.2692 | 2.6543 |

Coefficients:

| | Estimate | Std. Error | z value | Pr(> z) |
|-------------|----------|------------|---------|----------|
| (Intercept) | -8.383 | 3.627 | -2.311 | 0.0208 * |
| fibrinogen | 2.340 | 1.151 | 2.033 | 0.0421 * |

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 24.564 on 23 degrees of freedom
 Residual deviance: 17.107 on 22 degrees of freedom
 AIC: 21.107

Number of Fisher Scoring iterations: 5

4.2.3 Interpreting `summary()` in Logistical Regression

For the logistic regression output, it is important to note that the residuals are *deviance residuals*. What does that mean? The deviance residual is a mathematical transformation of the loss function (discussed in a later section) and quantifies a given point's contribution to the overall likelihood. These deviance residuals can then be used to form RSS-like statistics.

At the bottom of the output, the statistics section is quite different. We have null deviance and residual deviance for metrics. The null deviance measures the lack of fit of the model, considering only the intercept. The residual deviance measures the lack of fit of the entire model. We want to see that the Residual deviance is much lower than the Null deviance. The AIC is most useful in comparing models. AIC stands for Akaike Information Criterion and is based on deviance. AIC penalizes more complex models. The Fisher scoring algorithm is a modified form of Newton's method of solving a maximum likelihood problem.

The most significant difference is the interpretation of the coefficient estimates. Whereas the coefficient of a linear regression predictor quantifies the difference in the target variable as the predictor changes. This is not true for logistic regression. Instead of quantifying the difference in the target variable, it quantifies the difference in the log odds of the target variable. We will discuss odds, log odds, and probability in the Metrics section.

4.2.4 Evaluation on the Test Data

Next, we look at the output of `predict()` for logistic regression created in Code 4.2.4. Notice the parameter `type="response"`. This is important to get probabilities out of the model. The model outputs log-odds but by requesting "response" we get these numbers converted to probabilities for us. The probabilities for the first few test observations are:

```

              7              8              9              10              11              18
0.29967653 0.03871679 0.26646707 0.09116524 0.04631732 0.10790621

```

The `ifelse()` function is needed to convert these probabilities to 1 or 2, the internal coding for our target variable. Once we have these predictions in variable `pred`, we can compare them to the actual values in the test observations to get accuracy, the percentage of observations that were classified correctly. We also output a table of predictions and actual values. All the predictions were of class 1 and all but 7 of the 8 actual values were of class 1.

Code 4.2.4 — Logistic Regression. Evaluating the output.

```

probs <- predict(glm1, newdata=test, type="response")
pred <- ifelse(probs>0.5, 2, 1)
acc1 <- mean(pred==as.integer(test$ESR))
print(paste("glm1 accuracy = ", acc1))
table(pred, as.integer(test$ESR))

```

```
[1] "glm1 accuracy = 0.875"
```

```

pred 1 2
 1 7 1

```

4.2.5 Learning (or Not) From Data

Note one odd thing about the table above: the model always predicted class 1 ($ESR < 20$). Even though 87.5% accuracy sounds good, we cannot be impressed. In our test sample, 7 were of one class and only 1 of the other, therefore if the model just guesses the majority class, as it did in this case, it will be right 87.5% of the time. There are two problems here: (1) a small amount of data, (2) an unbalanced data set.

The first problem is that we have a very small data set. In order to create a stronger model, much more data would have to be collected. Data collection can be expensive and require domain expert assistance. In the case of this data set, more blood samples would have to be drawn from a random population, analyzed by technicians, and supervised by a hematologist or other clinician.

When additional data cannot be obtained, another but less preferable option is to use sampling techniques. Consider our example above, with 8 test cases randomly sampled

from the data. What if we randomly sampled the data multiple times? By randomly sampling the data multiple times, each time we would have a different test set. We could average our test accuracy over all these samples and get a better idea of the accuracy of our model. One such sampling technique is cross-validation which we will explore later in the book.

The second problem with the data set is that it is unbalanced. Of the 32 observations, only 6 have $ESR > 20$. This is a 81% $ESR < 20$ to 19% $ESR > 20$ ratio. Unbalanced data sets can pose problems for some classification algorithms while others can rise above it. Again, more data would be helpful. If that additional data is still unbalanced, sampling techniques may be of help. The idea is to oversample from the minority class and undersample from the majority class to come up with a data set that is more balanced. We will explore techniques for doing this in future chapters, and discuss when it might be helpful. For now, we will take our model as created, but take it with a grain of salt.

Check Your Understanding 4.1 — Logistic Regression. Practice on the PimaIndians-Diabetes2 data set from package `mlbench`. First, create an 80-20 split into train and test sets. What is your accuracy predicting diabetes from glucose?

By the way, this data set has a lot of missing NA values. If you get NA for your accuracy, it means that one or more of the predictions was NA. Check this with code: `sum(is.na(pred))`. If you have NAs, compute your mean accuracy using parameter `na.rm=TRUE` in the `mean()` function. ■

4.3 Metrics

Classification can be evaluated by many measures. In this section we will look at accuracy, sensitivity and specificity, Kappa, AUC, and ROC curves.

4.3.1 Accuracy, sensitivity and specificity

The most common metric to evaluate results in classification is accuracy:

$$acc = \frac{C}{N} \quad (4.1)$$

where C is the number of correct predictions, and N is the total number of test observations. The output of the `table()` command above was limited because the model predicted all test cases to be in one class. Normally it should look something like this:

```
pred 1 2
  1 7 1
  2 1 5
```

Such a table is also called a confusion matrix. The diagonal values from the upper left to the lower right are the correctly classified instances, 7 and 5 in this case. The other values are errors, 1 and 1. The flip side of accuracy is the error rate, calculated by subtracting accuracy from 1.

We can break down each component of the confusion matrix as follows:

```

pred T  F
   T TP FP
   F FN TN

```

- TP - true positive: these items are true and were classified as true
- FP - false positive: these items are false and were classified as true
- TN - true negative: these items are false and were classified as false
- FN - false negative: these items are true and were classified as false

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (4.2)$$

$$error\ rate = \frac{FP + FN}{TP + TN + FP + FN} = 1 - accuracy \quad (4.3)$$

The **sensitivity** measures the true positive rate:

$$sensitivity = \frac{TP}{TP + FN} \quad (4.4)$$

The **specificity** measures the true negative rate:

$$specificity = \frac{TN}{TN + FP} \quad (4.5)$$

Sensitivity and specificity range from 0 to 1, just as accuracy does, with values closer to 1 being better. They help to quantify the extent to which a given class was misclassified.

4.3.2 Kappa

Cohen's Kappa is a statistic that attempts to adjust accuracy by accounting for the possibility of a correct prediction by chance alone. Kappa is often used to quantify agreement between two annotators of data. Here we are quantifying agreement between our predictions and the actual values. Kappa is computed as follows:

$$\kappa = \frac{Pr(a) - Pr(e)}{1 - Pr(e)} \quad (4.6)$$

where $Pr(a)$ is the actual agreement and $Pr(e)$ is the expected agreement based on the distribution of the classes. The following interpretation of Kappa is often used but there is not universal agreement on this. Kappa scores:

- <0.2 poor agreement
- 0.2 to 0.4 fair agreement
- 0.4 to 0.6 moderate agreement
- 0.6 to 0.8 good agreement
- 0.8 to 1.0 very good agreement

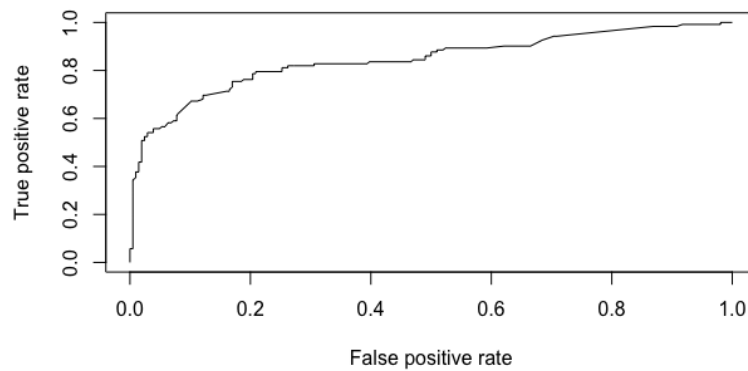


Figure 4.4: ROC Curve

4.3.3 ROC Curves and AUC

The ROC curve is a visualization of the performance of machine learning algorithms. The name ROC stands for Receiver Operating Characteristics which reflects its origin in communication technology in WWII in detecting between true signals and false alarms. The ROC curve shows the tradeoff between prediction true positives while avoiding false positives.

Figure 4.4 shows an ROC curve. This figure is taken from the Titanic logistic regression notebook on the github. The y axis is the true positive rate while the x axis is the false positive rate. The predictions are first sorted according to the estimation probability of the positive class. A perfect classifier would shoot straight up from the origin since it classified all correctly. We want to see the classifier shoot up and leave little space at the top left. Starting at the origin, each prediction's impact on the curve is vertical for correct predictions and horizontal for incorrect ones. If we see a diagonal line from the lower left to the upper right, then our classifier had no predictive value.

A related metric is AUC, the area under the curve. AUC values range from 0.5 for a classifier with no predictive value to 1.0 for a perfect classifier.

4.3.4 Probability, odds, and log odds

What is the difference between odds and probability? Let's look at this using a sports outcome example. Imagine we played 10 games with a friend and won 7. That means we lost 3 of course. Assuming we will do as well next time, our odds are 7 to 3:

$$\text{odds} = \frac{\text{number of wins}}{\text{number of losses}} = \frac{7}{3} \quad (4.7)$$

If we want to express the same data as a probability:

$$\text{probability} = \frac{\text{number of wins}}{\text{number of games}} = \frac{7}{10} \quad (4.8)$$

Notice that probability will always range from 0 to 1, whereas odds will not. Recall that the `glm()` algorithm coefficients represent a change in log odds. Just as it sounds, log odds are the log of the odds: $\log(\text{odds})$. So to find the odds, we use the inverse of log, the `exp()` function. And to convert odds to probability, we use the following:

$$\text{probability} = \frac{\text{odds}}{1 + \text{odds}} \quad (4.9)$$

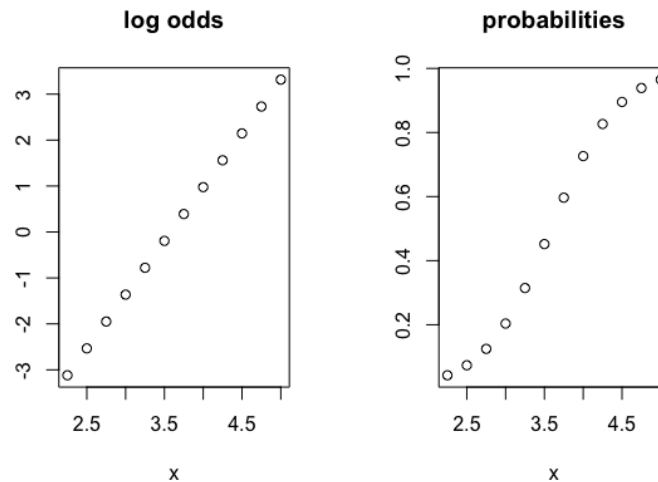


Figure 4.5: Log Odds versus Probability

Let's see what this means in terms of the plasma data logistic regression model above. The predictor is fibrinogen, which ranges from around 2.09 to 5.06 in this data set. Let's compare the effect of fibrinogen across values from 2.25 to 5.0. Figure 4.5 plots the log odds across this range of values on the left, and on the right, the associated probabilities across the same range of values. Observe that the results of the logistic regression model (the log odd) are linear in the parameters (w , b) but that the associated probabilities are not linear.

The coefficient for fibrinogen in the linear regression model was 2.34. For every one-unit increase in x , the probability of $\text{ESR} > 20$ changes by $\exp(2.34)/[1 + \exp(2.34)]$. Let's look at some sample values in Table 4.1. The X column is fibrinogen for a range of values. The log odds is found by plugging in the value of x for the logistic regression formula given by: $2.34 * x - 8.383$.

As you see in Table 4.1 and more clearly in Figure 4.5, the log odds are linear in the parameters but the probability is not linear, we can discern a subtle S-shape in the probabilities.

R Logistic Regression Coefficients

In linear regression we interpret a predictor coefficient as the amount of change in y for a 1-unit change in x . We cannot make this interpretation in logistic regression. The predictor coefficient in a logistic regression model specifies the change in log-odds for a 1-unit change in x .

| X | Log Odds | Probability |
|-----|----------|-------------|
| 2.5 | -2.53 | 0.07 |
| 3.0 | -1.36 | 0.20 |
| 3.5 | -0.19 | 0.45 |
| 4.0 | 0.977 | 0.73 |
| 4.5 | 2.147 | 0.89 |

Table 4.1: Log Odds and Probability for Plasma Data

Check Your Understanding 4.2 — Metrics for Classification. Using your model created for the Pima Indians data, compute the following in R using the formulas above:

- TP, TN, FP and FN
- accuracy using TP, TN, FP, FN
- error rate using accuracy
- sensitivity
- specificity

Next, use the `confusionMatrix()` function in package `caret` to confirm your results. Notice that `caret` displays Kappa as well as other statistics. ■

4.4 The Algorithm

The linear regression output was a quantitative value that could range over all the real numbers. What we need for logistic regression classification is a function that will output probabilities in the range $[0, 1]$. The sigmoid, or logistic, function is used for this purpose and of course is where the algorithm gets its name. When real numbers are input to the logistic function, the output is squashed into the range $[0, 1]$ as seen in Figure 4.6. The logistic function is:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (4.10)$$

The logistic regression algorithm computes the log odds from the estimated parameters. The log odds is just $\log(\text{odds})$. The odds are the probability of the positive class, $p(x)$, over the probability of the negative class $(1 - p(x))$. If we have a single predictor w_1 and an intercept w_0 , the log odds are:

$$\log \frac{p(x)}{1 - p(x)} = w_0 + w_1 x \quad (4.11)$$

Solving for p gives us the logistic function:

$$p(x) = \frac{e^{-(w_0 + w_1 x)}}{1 + e^{-(w_0 + w_1 x)}} = \frac{1}{1 + e^{-(w_0 + w_1 x)}} \quad (4.12)$$

We can see in Equation 4.11 why logistic regression is considered a linear classifier. It creates a linear boundary between classes in which the distance from the boundary

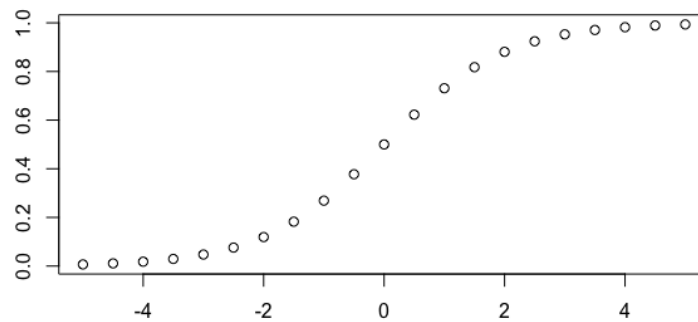


Figure 4.6: The Logistic Function

determines the probability. When we use the logistic function for classification, the cut-off point is usually 0.5. Notice in Figure 4.6 that this is the inflection point of the S-curve. Probabilities greater than 0.5 are classified as the positive class and probabilities less than 0.5 are classified in the other class.

Check Your Understanding 4.3 — Logistic Regression Algorithm. Using the coefficients and data from the logistic regression Check-Your-Understanding above, try the following:

- Compute the probabilities in R using only the glucose column from the test data and the coefficients of the model.
- Compare these to the output of `predict()` you did earlier. Are they the same?
- Create a plot with test glucose on the x axis and the probabilities you calculated for the test set on the y axis. What do you observe?
- Create a vector of x values: 60, 100, 140, 180, 220
- Compute probabilities for these values using the same formula as above.
- Are these probabilities consistent with the graph you just created?

4.5 Mathematical Foundations

How are the parameters, \mathbf{w} , found for logistic regression? First an appropriate loss function is established, then an optimization technique such as gradient descent is used to find optimal parameters.

Recall the loss function for linear regression:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^n (y_i - f(x_i))^2 \quad (4.13)$$

If we plug in the logistic function for $f(x)$, it will not be a convex function. That is a problem because gradient descent works only for convex functions. A suitable loss

function for logistic regression can be found by starting with the likelihood function, L :

$$L(w_0, w_1) = \prod_{i=1}^n f(x_i)^{y_i} (1 - f(x_i))^{1-y_i} \quad (4.14)$$

Notice in the likelihood equation that one of the terms in the product will always reduce to 1 because the y values are either 0 or 1. To simplify the likelihood equation we will take the log of it to find the log-likelihood, ℓ :

$$\ell = \sum_{i=1}^n y_i \log f(x_i) + (1 - y_i) \log(1 - f(x_i)) \quad (4.15)$$

The log-likelihood equation for each instance:

$$\ell = y \log f(x) + (1 - y) \log(1 - f(x)) \quad (4.16)$$

In training the classifier, we want to penalize it for wrong classifications. This is our loss function. The penalties follow directly from Equation 4.16. Our Loss is:

$$\mathcal{L} = -\log(f(x)) \text{ if } y = 1 \quad \mathcal{L} = -\log(1 - f(x)) \text{ if } y = 0 \quad (4.17)$$

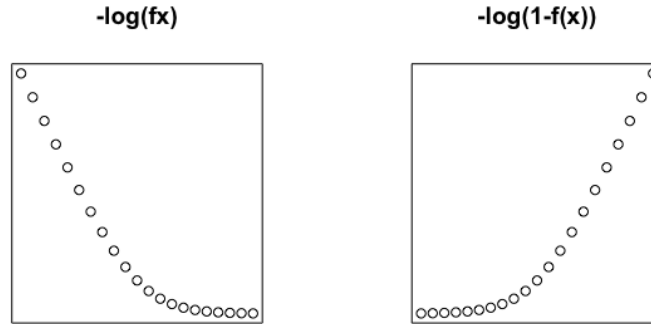


Figure 4.7: Loss Function for Logistic Regression

These two loss functions are visualized in Figure 4.7. In the leftmost graph which represents $-\log(fx)$, the closer the function gets to 1, the smaller the penalty but the closer it gets to 0, the higher. We want to penalize $-\log(f(x))$ severely if it classifies as 0 when the true target is 1. The reverse is true when the true target is 0. This is shown in the rightmost graph. Here we penalize $-\log(1-f(x))$ severely as it moves toward 1 because the true target is 0.

We can put the two loss functions into one equation as shown below, where they are summed over all observations. Notice that one of the terms will always be zero because y will be either 0 or 1.

$$\mathcal{L} = - \left[\sum_{i=1}^N y_i \log(f(x_i)) + (1 - y_i) \log(1 - f(x_i)) \right] \quad (4.18)$$

where $f(x) =$

$$f(x) = \frac{1}{1 + e^{-(w^T x)}} \quad (4.19)$$

The parameters w are then determined using gradient descent. You can see from Figure 4.7 that when you put together the two loss functions you will have a convex function suitable for gradient descent. Once the parameters are known, classifying new instances is done by plugging in the instance into the logistic function shown in Equation 4.19. This returns a probability in the range $[0, 1]$. Typically we establish a threshold such as 0.5. Values over that threshold are classified as 1, values below are classified as 0.

4.6 Logistic Regression with Multiple Predictors

The github repo for the book has a logistic regression example on a Titanic data set. We will just cover a few points here. The first is that the data had a lot of NA values. NA values can cause a lot of problems for many classifiers and it is a good idea to remove them. There are a couple of approaches to take. One is to simply remove rows with NAs which is a good choice if you have lots of remaining data. We saw an example of that using R's `complete.cases()` function in the linear regression chapter. Another approach is to replace NA values with either the mean or median of the variable for quantitative variables, or the most common factor for qualitative variables. The code below shows the detection of NAs with the `apply()` function.

Code 4.6.1 — Detecting NAs. Using `apply()`.

```
apply(df, function(x) sum(is.na(x)==TRUE))
pclass survived      sex      age
1           1         0      264
```

Since there are single NAs for `pclass` and `survived`, we can just remove those rows, for `age`, we will replace the NA with the median value. That is shown in the code below.

Code 4.6.2 — Removing NAs. By deleting rows or replacing with the median.

```
df <- df[!is.na(df$pclass),]
df <- df[!is.na(df$survived),]
df$age[is.na(df$age)] <- median(df$age, na.rm=T)
```

One concern about replace age NAs with median values is that there were so many NAs, almost one-third of the data. By replacing so many NAs with the mean or median, we are diminishing the predictive power of age. An alternative is to replace age NAs for observations that survived with the mean or median for age of those who survived, and then do something similar for NAs for observations that did not survive. This approach is somewhat problematic as well. We are manipulating our data. Whatever decisions you make should be thoroughly documented in your RStudio notebook and any other reporting you do on your results.

In the online notebook you will see that we divided the data into train and test sets and build a logistic regression model which achieved 79% accuracy. The code also gives an example of using the `caret` package to output the confusion matrix and other statistics including the Kappa which was 0.5563. The ROC curve for our predictions is the ROC curve shown in the Metrics section above. The AUC for that curve was 0.84. The following is the key output of `summary()` for the model predicting survival with all predictors.

```
Call:
glm(formula = survived ~ ., family = "binomial", data = train)
Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.5650  -0.6931  -0.4586   0.6847   2.3496
Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  3.313570   0.342936   9.662 < 2e-16 ***
pclass2     -1.234362   0.244633  -5.046 4.52e-07 ***
pclass3     -2.253769   0.225046 -10.015 < 2e-16 ***
sexmale      -2.391903   0.169846 -14.083 < 2e-16 ***
age          -0.030974   0.006996  -4.427 9.54e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
. . .
Null deviance: 1307.9 on 980 degrees of freedom
Residual deviance: 932.5 on 976 degrees of freedom
AIC: 942.5
```

Consider the summary shown above. All of the predictors seem to be good predictors. Notice that `pclass` has dummy variables for class 2 and class 3. How do we interpret this? If a person is `pclass 2` instead of 1, the log odds of their survival decreases by 1.23 and if they are `pclass 3` instead of 1, the log odds of their survival decreases by 2.25. A similar interpretation can be made for male over female. In this example, age is our only quantitative predictor. The coefficient is telling us that log odds of survival decrease a little for every year. Notice also the large drop in deviance from the null deviance which considers the intercept alone, and the residual deviance, which considers all predictors. This drop indicates that our predictors are good predictors.

Below we have the code for generating the ROC plot and computing AUC. The AUC is .84.

Code 4.6.3 — ROC and AUC. On the Titanic Data.

```
library(ROCR)
p <- predict(glm1, newdata=test, type="response")
pr <- prediction(p, test$survived)
# TPR = sensitivity, FPR=specificity
prf <- performance(pr, measure = "tpr", x.measure = "fpr")
plot(prf)
# compute AUC
auc <- performance(pr, measure = "auc")
auc <- auc@y.values[[1]]
```

Check Your Understanding 4.4 — Logistic Regression with Multiple Predictors. In this example, you will build a logistic regression model using multiple predictors on the Pima Indian data. Try the following:

- Build a model with all predictors, using the same train and test set as you used with only glucose as a predictor. Is your accuracy higher or lower?
- Use R functions to count how many NAs are in each column.
- Replace NAs in the triceps and insulin columns with the mean of each column in both train and test. Now replace train and test with rows that have complete data using `complete.cases()`.
- Build a second model with all predictors on the train and test that have been cleaned up.
- Can you run `anova()` on these two models? Why or why not?
- Compare the accuracies of the two models.
- Compare the residual deviance and degrees of freedom for each model. Which model do you think is best, and why?

4.7 Advanced Topic: Optimization Methods

Earlier we specified our log-loss function for logistic regression as follows:

$$\ell = \sum_{i=1}^n y_i \log f(x_i) + (1 - y_i) \log(1 - f(x_i)) \quad (4.20)$$

where $f(x) =$

$$f(x) = \frac{1}{1 + e^{-(w^T x)}} \quad (4.21)$$

Next we find the gradient of the log likelihood. The gradient, g , is the partial derivative with respect to the parameters w . The gradient is the slope which is going to tell the algorithm which direction to move to find the minimum. The gradient equation is given

below. Notice that it is really the \mathbf{X} matrix multiplied by how wrong $f(\mathbf{x})$ is at this point in predicting the true y .

$$\mathbf{g} = \frac{\partial \ell}{\partial \mathbf{w}} = \sum_i (f(x_i) - y_i) \mathbf{x}_i = \mathbf{X}^T (f(\mathbf{x}) - \mathbf{y}) \quad (4.22)$$

All we need for gradient descent is the first derivative, the gradient. For other optimization methods we will also need the second derivative, the Hessian, \mathbf{H} . The Hessian is a square matrix of second partial derivatives that gives the local curvature of a function. Note that in the following equation for the Hessian, the \mathbf{S} represents $\mathbf{S} \triangleq \text{diag}(p_i(1 - p_i))$

$$\mathbf{H} = \frac{\partial}{\partial \mathbf{w}} \mathbf{g}(\mathbf{w})^T = \sum_i (\nabla_{\mathbf{w}} f(x_i)) \mathbf{x}_i^T = \sum_i f(x_i) (1 - f(x_i)) \mathbf{x}_i \mathbf{x}_i^T = \mathbf{X}^T \mathbf{S} \mathbf{X} \quad (4.23)$$

There are many algorithms to find the optimal parameters, \mathbf{w} . The first one we examine is gradient descent.

4.7.1 Gradient Descent

Gradient descent is an iterative approach where at each step the estimated parameters, θ get closer to the optimal values. We can express this as follows:

$$\theta_{k+1} = \theta_k - \eta_k \mathbf{g}_k \quad (4.24)$$

where η is the learning rate, which specifies how big of a step to take at each iteration. If the eta (step size) is too slow the algorithm will take a long time to converge. If eta is too large, the true minimum can be stepped over and then the algorithm will not be able to converge.

4.7.2 Stochastic Gradient Descent

For large data sets, gradient descent can bog down. An alternative is *stochastic gradient descent* which processes the data either one at a time or in small batches instead of all at once. It is stochastic because the observations are chosen randomly. If the data is processed one at a time it means that the gradient has to be computed at each step. It turns out that the gradient will reflect the underlying function better if it is computed from a small batch. This also improves computation time.

4.7.3 Newton's Method

Gradient descent finds the optimal parameters using the first derivative. Newton's method is another approach; it uses the second derivative, the Hessian defined above. Newton's method (also called Newton-Raphson) is also an iterative method. At each step either the full Hessian is recalculated, or it is updated in which we call the method quasi-Newton. In a well-behaved convex function, Newton's method will converge faster.

A key insight in Newton's method is that if it is computationally difficult to compute a minimum for a given function, then come up with a function that shares important

properties with the original function but is easier to minimize. At each iteration, Newton's method constructs a quadratic approximation to the objective function in which the first and second derivatives are the same. The approximate function is minimized instead of the original.

How is this approximate function found? A Taylor series about the point is used, but ignores derivatives past the second. A Taylor series converts a function into a power function and the first few terms can be used to get an approximate value for a function.

4.7.4 Optimization from Scratch

We are going to take a closer look at gradient descent by finding our optimal coefficients in the plasma data set from scratch. The R Notebook for this is available online. First we recreate the logistic regression model from earlier in the chapter. Our coefficients were $w_0 = -8.38$ and $w_1 = 2.34$.

The first thing we need to do is define our sigmoid/logistic function that will take an input matrix and return a vector of sigmoid values for each observation. We initialize w_0 and w_1 to 1. We make a data matrix where column 1 is all 1s that will be multiplied by the intercept, and column 2 is fibrinogen which will be multiplied by w_1 . We will also need the labels but since they were coded as 1-2 instead of 0-1 we subtract 1.

Code 4.7.1 — Logistic Regression from Scratch. Set up code.

```
sigmoid <- function(z){
  1.0 / (1+exp(-z))
}
# set up weight vector, label vector, and data matrix
weights <- c(1, 1)
data_matrix <- cbind(rep(1, nrow(train)), train$fibrinogen)
labels <- as.integer(train$ESR) - 1
```

Now we are ready to iterate. The code below iterates 500,000 times. In each iteration it does the following:

1. multiplies the data by the weights to get the log likelihood, then runs these values through the sigmoid() function to get a vector of probabilities
2. computes the error: the true values (0 or 1) minus the probabilities
3. updates the weights by weights plus the learning rate times the gradient; Recall that the gradient is the X values times the errors as shown in Equation 4.22; Notice also the operator for matrix multiplication is an asterisk surrounded by percent signs.

Code 4.7.2 — Gradient Descent from Scratch. Three steps per iteration.

```
learning_rate <- 0.001
for (i in 1:500000){
  prob_vector <- sigmoid(data_matrix %*% weights)
  error <- labels - prob_vector
  weights <- weights + learning_rate * t(data_matrix) %*% error
}
weights
```

Try running this code several times, changing the number of iterations. The following table shows the weights at various numbers of iterations.

| No. Iterations | (w_0) | (w_1) |
|----------------|---------|---------|
| 50 | 0.45 | -0.36 |
| 500 | -0.25 | -0.27 |
| 5000 | -4.15 | 1.00 |
| 50000 | -8.26 | 2.30 |
| 500000 | -8.38 | 2.34 |

Table 4.2: Optimized Weights by Number of Iterations

The 500,000 iterations gives use the same coefficients as R's `glm()`. However, it was slow compared to R's optimized code. R functions are heavily optimized and will reliably give good performance.

Finally, we create a plot that confirms Equation 4.11. The linear combination of the weights we calculated in the code above and X values give us the log odds. In the graph the $ESR > 20$ are color coded.

Code 4.7.3 — Log Odds. Linear combination of $w_0 + w_1 x$

```
plasma_log_odds <- cbind(rep(1, 32), plasma$fibrinogen) %*% weights
plot(plasma$fibrinogen, plasma_log_odds, col=plasma$ESR)
abline(weights[1], weights[2])
```

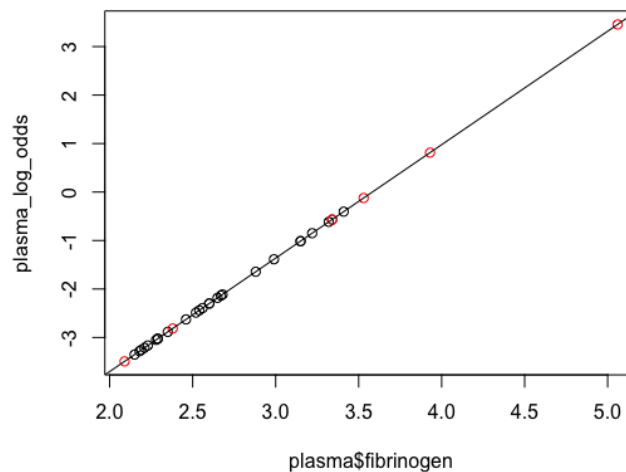


Figure 4.8: LogOdds is a Linear Combination of the Parameters

4.8 Multiclass Classification

The classification examples we have seen so far have been binary, classifying into one of two possible classes. What if we want to classify in a scenario where there are more than two classes? One technique that can be used is **one-versus-all** classification in which we perform multiple binary classifications. Let's look at the iris data set as an example. The notebook for this is in github as usual. The iris data set contains 150 observations of flower measurements. There are 50 observations each of 3 species: virginica, setosa, and versicolor. First we look at a couple of graphs for data exploration. Figure 4.9 shows the pairs() output for the predictor columns with the color of each observation representing one of the 3 classes. Figure 4.10 plots petal width and length, color coded as well. The code for these plots is given below. The as.integer() function was used to make Species an integer 1, 2, or 3, which in turn is used to match colors red, yellow and blue.

Code 4.8.1 — Code to Generate Plots. Using as.integer()

```
pairs(iris[1:4], pch = 21,
      bg = c("red", "yellow", "blue")[as.integer(Species)])
plot(Petal.Length, Petal.Width, pch=21,
     bg=c("red","yellow","blue")as.integer(Species))
```

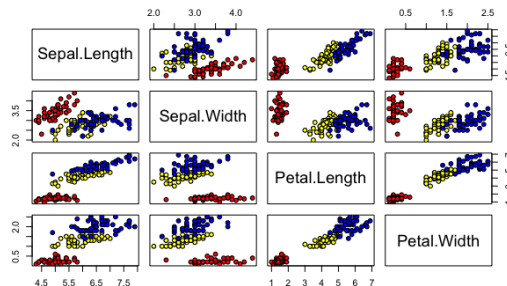


Figure 4.9: Iris Pairs

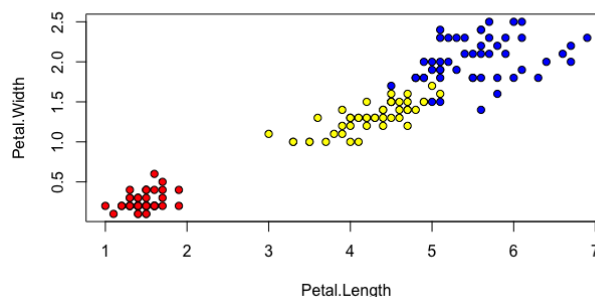


Figure 4.10: Iris Petal Length and Width

To perform one-versus-all classification for 3 classes, we have to create 3 data sets, one for each class as shown next. Each time we copy the original data set, then set the Species column to be a binary factor for that species or not.

Code 4.8.2 — Make a data set for each class. With a binary target.

```
# recode as virginica or not
iris_virginica <- iris
iris_virginica$Species <-
  as.factor(ifelse (iris_virginica$Species=="virginica",1,0))
# recode as setosa or not
iris_setosa <- iris
iris_setosa$Species <-
  as.factor(ifelse (iris_setosa$Species=="setosa",1,0))
# recode as versicolor or not
iris_versicolor <- iris
iris_versicolor$Species <-
  as.factor(ifelse (iris_versicolor$Species=="versicolor",1,0))
```

We next write a function to enable us to run the same code 3 times.

Code 4.8.3 — Function Definition. For repeated code.

```
fun <- function(df, i){
  train <- df[i,]
  test <- df[-i,]
  glm1 <- glm(Species~., data=train, family="binomial")
  probs <- predict(glm1, newdata=test)
  pred <- ifelse(probs>0.5, 1, 0)
  acc <- mean(pred==test$Species)
  print(paste("accuracy = ", acc))
  table(pred, test$Species)
}
```

Next we make one set of indices to divide the train and test sets, and run the function on each data set.

Code 4.8.4 — Run the function. On each data set.

```
set.seed(1234)
i <- sample(1:150, 100, replace=FALSE)
fun(iris_virginica, i)
fun(iris_setosa, i)
fun(iris_versicolor, i)
```

The accuracies for the 3 runs were: 0.98, 1.0, and 0.62. The average gives 87% overall accuracy. From the figures above it is clear that separating setosa (yellow) and versicolor (blue) is challenging and it looks like the versicolor classifier is the weakest one.

How well does this one-versus-all approach scale up? You can imagine that it would be troublesome for classifying 10 classes, as in digit recognition. You would have to build

10 classifiers. As we will see in the deep learning chapter, neural networks can be built to output probabilities for 10 classes, as in this problem scenario. The kNN algorithm is another example of an algorithm that can handle multi-class problems.

Warnings in glm()

For logistic regression, if your training data is perfectly or nearly perfectly linearly separable, R will throw out several warning messages. This is due to the inability to maximize the likelihood which already has separated the data perfectly. For example, for the iris data which is too easy to classify, the sample notebook on github shows the error messages:

```
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

Since there are just warnings, they do not stop the algorithm from producing a model.

4.9 Generalized Linear Models

Logistic regression is part of the GLM (generalized linear model) family because its response is determined by a predictor in which the parameters are linear. In R's glm() function we used the family parameter to specify the family and link function.

```
glm1 <- glm(ESR~fibrinogen, data=train, family=binomial)
```

For binomial, the link function is "logit". Other available families include gaussian, gamma, poisson and more, as indicated in the R documentation.

Why do we need a link function? In linear regression the algorithm assumes that the target variable is normally distributed over the real numbers. This is not the case for logistic regression and other generalized linear models. The link function links the mean of the target $\mu_i = E(Y_i)$ to the linear term $x_i^T w$. The link function "links" the linear predictors to the response. The canonical link for mapping real numbers to $[0,1]$ is the logit.

4.10 Summary

Logistic regression is something of a misnomer because we use it for classification, not regression. It is considered a linear model because it is linear in the parameters. The sigmoid function shapes the output to be in range $[0, 1]$ for probabilities. Here are the strengths and weaknesses of logistic regression:

Strengths:

- Separates classes well if they are linearly separable
- Computationally inexpensive
- Nice probabilistic output

Weaknesses:

- Prone to underfitting; Not flexible enough to capture complex non-linear decision boundaries

In this chapter we showed how to perform logistic regression with one predictor or multiple predictors. In addition, we discussed how to use the one-versus-all technique for multi-class classification.

4.10.1 New Terminology in this Chapter

Refer back to the chapter or look at the glossary if you are unsure of the meaning of these terms.

New metrics:

- accuracy
- error rate
- sensitivity
- specificity
- Kappa
- ROC Curves
- AUC

Mathematical terms:

- probability
- odds
- log odds

4.10.2 Quick Reference

Reference 4.10.1 Build and Examine a Logistic Regression Model

```
glmName <- glm(formula, data=train, family=binomial)
summary(glmName)
```

Reference 4.10.2 Predict on Test Data

```
probs <- predict(glmName, newdata=test, type="response")
pred <- ifelse(probs>0.5, 1, 0)
table(pred, test$target)
acc <- mean(pred==test$target)
```

Reference 4.10.3 ROC and AUC

```
library(ROCR)
p <- predict(glm1, newdata=test, type="response")
pr <- prediction(p, test$survived)
# TPR = sensitivity, FPR=specificity
prf <- performance(pr, measure = "tpr", x.measure = "fpr")
plot(prf)
# compute AUC
auc <- performance(pr, measure = "auc")
auc <- auc@y.values[[1]]
```

Reference 4.10.4 Counting NAs with `sapply()`. The first argument to `sapply` is a list, and a data frame can be considered a list of vectors. The second argument in our example is an anonymous function that sums the number of NAs. The `sapply` function applies this anonymous function to every column in the data frame.

```
sapply(df, function(x) sum(is.na(x))==TRUE)
```

Reference 4.10.5 Fixing NAs with `lapply()`. In the code below we first define a function to replace NAs in a vector with the average of the vector. Notice we have to use the parameter `na.rm=TRUE` to get the mean value. As a reminder, in R the last statement evaluated is returned. There is no need for a `return()` statement in this simple function. The last statement below uses `lapply()` to run the `fix_NA` function on every vector. Use `lapply` instead of `sapply` when you want a vector back instead of a list.

```
fix_NA <- function(x){
  ifelse(!is.na(x),x,mean(x, na.rm=TRUE))
}

df[] <- lapply(df, fix_NA)
```

4.10.3 Lab

Problem 4.1 — Practice on the Abalone Data. Try the following:

1. Download the Abalone data from the UCI Machine Learning Repository, <http://archive.ics.uci.edu/ml/datasets/Abalone>.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Check if there are missing values.
4. Examine the rings column with `range()`, `median()`, and `hist()` to determine where you would like to split the data into two classes: large and small.
5. Create a new factor column for binary large/small based on the rings column and your cut-off decision.
6. Divide the data into 80-20 train-test, setting a seed for reproducibility.
7. Create a logistic regression model with all predictors except rings. What is the accuracy of the model? Do you think this is a good model? Why or why not?
8. Create at least 2 more models, trying different features, and combinations of features to see if you can improve these results.

Problem 4.2 — Practice on the Heart Data. Try the following:

1. Download the Heart data from the UCI Machine Learning Repository, <http://archive.ics.uci.edu/ml/datasets/Heart+Disease>.
2. The data does not have column names so you will have to create them. Use meaningful names based on your reading about the data on the UCI site.
3. Make sure that columns are of the correct type: factors, integer, etc.
4. Check if there are missing values, and how many per column.
5. Divide the data into 80-20 train-test, setting a seed for reproducibility.
6. Create a logistic regression model with all predictors. What is the accuracy of the model? Do you think this is a good model? Why or why not?
7. Create at least one more model, trying different features, and combinations of features to see if you can improve these results.

4.10.4 Exploring Concepts

Problem 4.3 Based on your experience with the logistic regression algorithm in R, does removing predictors with low p-values necessarily improve performance? Discuss possible reasons for your answer.

Problem 4.4 If you found that some predictors were not adding to the performance of the model, what reasons might you give for leaving them in? What reasons might you give for taking them out?

4.10.5 Going Further

A free online lesson on Logistic Regression is available here: <https://onlinecourses.science.psu.edu/stat504/node/149>