Brad Boehmke & Brandon Greenwell

Hands-on Machine Learning with R

Dedication TBD

Contents

List of Tables

List of Figures

Preface

Welcome to *Hands-on Machine Learning with R*. This book provides hands-on modules for many of the most common machine learning methods to include:

- Generalized low rank models
- Clustering algorithms
- Autoencoders
- Regularized models
- Random forests
- Gradient boosting machines
- Deep neural networks
- Stacking / super learners
- and more!

You will learn how to build and tune these various models with R packages that have been tested and approved due to their ability to scale well. However, our motivation in almost every case is to describe the techniques in a way that helps develop intuition for its strengths and weaknesses. For the most part, we minimize mathematical complexity when possible but also provide resources to get deeper into the details if desired.

Who should read this

We intend this work to be a practitioner's guide to the machine learning process and a place where one can come to learn about the approach and to gain intuition about the many commonly used, modern, and powerful methods accepted in the machine learning community. If you are familiar with the analytic methodologies, this book may still serve as a reference for how to work with the various R packages for implementation. While an abundance of videos, blog posts, and tutorials exist online, we have long been frustrated by the lack of consistency, completeness, and bias towards singular packages for implementation. This is what inspired this book.

This book is not meant to be an introduction to R or to programming in general; as we assume the reader has familiarity with the R language to include defining functions, managing R objects, controlling the flow of a program, and other basic tasks. If not, we would refer you to R for Data Science[1] (Wickham and Grolemund, 2016) to learn the fundamentals of data science with R such as importing, cleaning, transforming, visualizing, and exploring your data. For those looking to advance their R programming skills and knowledge of the languge, we would refer you to Advanced R[2] (Wickham, 2014). Nor is this book designed to be a deep dive into the theory and math underpinning machine learning algorithms. Several books already exist that do great justice in this arena (i.e. Elements of Statistical Learning[3] (Friedman et al., 2001), Computer Age Statistical Inference[4] (Efron and Hastie, 2016), Deep Learning[5] (Goodfellow et al., 2016)).

Instead, this book is meant to help R users learn to use the machine learning stack within R, which includes using various R packages such as **glmnet**, **h2o**, **ranger**, **xgboost**, **lime**, and others to effectively model and gain insight from your data. The book favors a hands-on approach, growing an intuitive understanding of machine learning through concrete examples and just a little bit of theory. While you can read this book without opening R, we highly recommend you experiment with the code examples provided throughout.

Why R

R has emerged over the last couple decades as a first-class tool for scientific computing tasks, and has been a consistent leader in implementing statistical methodologies for analyzing data. The usefulness of R for data science stems from the large, active, and growing ecosystem of third-party packages: **tidyverse** for common data analysis activities; **h2o**, **ranger**, **xgboost**, and others for fast and scalable machine learning; **iml**, **pdp**, **vip**, and others for machine learning interpretability; and many more tools will be mentioned throughout the pages that follow.

[1] http://r4ds.had.co.nz/index.html
[2] http://adv-r.had.co.nz/
[3] https://web.stanford.edu/~hastie/ElemStatLearn/
[4] https://web.stanford.edu/~hastie/CASI/
[5] http://www.deeplearningbook.org/

Conventions used in this book

The following typographical conventions are used in this book:

- ***strong italic***: indicates new terms,
- **bold**: indicates package & file names,
- `inline code`: monospaced highlighted text indicates functions or other commands that could be typed literally by the user,
- code chunk: indicates commands or other text that could be typed literally by the user

```
1 + 2
## [1] 3
```

In addition to the general text used throughout, you will notice the following code chunks with images, which signify:

Signifies a tip or suggestion

Signifies a general note

Signifies a warning or caution

Additional resources

There are many great resources available to learn about machine learning. Throughout the chapters we try to include many of the resources that we have found extremely useful for digging deeper into the methodology and applying with code. However, due to print restrictions, the hard copy version of this book limits the concepts and methods discussed. Online supplementary material exists at `https://github.com/koalaverse/hands-on-machine-learning-with-r`. The additional material will accumulate over time and include extended chapter material (i.e., random forest package benchmarking) along with brand new content we couldn't fit in (i.e., random hyperparameter search). In addition, you can download the data used throughout the book, find teaching resources (i.e., slides and exercises), and more.

Feedback

Reader comments are greatly appreciated. To report errors or bugs please post an issue at `https://github.com/koalaverse/hands-on-machine-learning-with-r/issues`.

Acknowledgments

TBD

Software information

An online version of this book is available at `http://bit.ly/HOML_with_R`. The source of the book along with additional content is available at `https://github.com/koalaverse/hands-on-machine-learning-with-r`. The book is powered by `https://bookdown.org` which makes it easy to turn R markdown files into HTML, PDF, and EPUB.

This book was built with the following packages and R version. All code was executed on 2017 MacBook Pro with a 2.9 GHz Intel Core i7 processor, 16 GB of memory, 2133 MHz speed, and double data rate synchronous dynamic random access memory (DDR3).

```r
# packages used
pkgs <- c(
  "AmesHousing",
  "bookdown",
  "caret",
  "cluster",
  "DALEX",
  "data.table",
  "dplyr",
  "dslabs",
  "e1071",
  "earth",
  "emo",
  "extracat",
  "factoextra",
  "ggplot2",
  "gbm",
  "glmnet",
  "h2o",
  "iml",
  "ipred",
  "keras",
  "kernlab",
  "MASS",
  "mclust",
  "mlbench",
  "pBrackets",
  "pdp",
  "pls",
  "pROC",
  "purrr",
  "ranger",
  "recipes",
  "reshape2",
  "ROCR",
  "rpart",
  "rpart.plot",
  "rsample",
  "tfruns",
  "tfestimators",
  "vip",
  "xgboost"
)
```

```
# package & session info
sessioninfo::session_info(pkgs)
#> - Session info ----------------------------------------
#>  setting  value
#>  version  R version 3.6.0 (2019-04-26)
#>  os       macOS Sierra 10.12.6
#>  system   x86_64, darwin15.6.0
#>  ui       RStudio
#>  language (EN)
#>  collate  en_US.UTF-8
#>  ctype    en_US.UTF-8
#>  tz       America/New_York
#>  date     2019-06-25
#>
#> - Packages --------------------------------------------
#>  ! package       * version    date       lib
#>    abind           1.4-5      2016-07-21 [1]
#>    AmesHousing     0.0.3      2017-12-17 [1]
#>    assertthat      0.2.1      2019-03-21 [1]
#>    backports       1.1.4      2019-04-10 [1]
#>    base64enc       0.1-3      2015-07-28 [1]
#>    BH              1.69.0-1   2019-01-07 [1]
#>    bitops          1.0-6      2013-08-17 [1]
#>    bookdown        0.11       2019-05-28 [1]
#>    boot            1.3-22     2019-04-02 [1]
#>    car             3.0-3      2019-05-27 [1]
#>    carData         3.0-2      2018-09-30 [1]
#>    caret         * 6.0-84     2019-04-27 [1]
#>    caTools         1.17.1.2   2019-03-06 [1]
#>    cellranger      1.1.0      2016-07-27 [1]
#>    checkmate       1.9.3      2019-05-03 [1]
#>    class           7.3-15     2019-01-01 [1]
#>    cli             1.1.0      2019-03-19 [1]
#>    clipr           0.6.0      2019-04-15 [1]
#>    cluster         2.0.8      2019-04-05 [1]
#>    codetools       0.2-16     2018-12-24 [1]
#>    colorspace      1.4-1      2019-03-18 [1]
#>    config          0.3        2018-03-27 [1]
#>    cowplot         0.9.4      2019-01-08 [1]
#>    crayon          1.3.4      2017-09-16 [1]
#>    curl            3.3        2019-01-10 [1]
#>    DALEX           0.3.0      2019-03-25 [1]
#>    data.table      1.12.2     2019-04-07 [1]
#>    dendextend      1.12.0     2019-05-11 [1]
#>    digest           0.6.19     2019-05-20 [1]
#>    dplyr          * 0.8.1      2019-05-14 [1]
#>    dslabs           0.5.2      2018-12-19 [1]
#>    e1071            1.7-1      2019-03-19 [1]
#>    earth            5.1.1      2019-04-12 [1]
#>    ellipse          0.4.1      2018-01-05 [1]
#>    ellipsis         0.1.0      2019-02-19 [1]
#>    emo              0.0.0.9000 2019-05-03 [1]
#>    evaluate         0.14       2019-05-28 [1]
#>  R extracat         <NA>       <NA>       [?]
#>    factoextra       1.0.5      2017-08-22 [1]
#>    FactoMineR       1.41       2018-05-04 [1]
#>    fansi            0.4.0      2018-10-05 [1]
#>    flashClust       1.01-2     2012-08-21 [1]
#>    forcats        * 0.4.0      2019-02-17 [1]
#>    foreach          1.4.4      2017-12-12 [1]
#>    foreign          0.8-71     2018-07-20 [1]
#>    forge            0.2.0      2019-02-26 [1]
#>    Formula          1.2-3      2018-05-03 [1]
#>    gbm              2.1.5      2019-01-14 [1]
#>    gdata            2.18.0     2017-06-06 [1]
#>    generics         0.0.2      2018-11-29 [1]
#>    ggplot2        * 3.1.1      2019-04-07 [1]
#>    ggpubr           0.2        2018-11-15 [1]
#>    ggrepel          0.8.1      2019-05-07 [1]
#>    ggsci            2.9        2018-05-14 [1]
#>    ggsignif         0.5.0      2019-02-20 [1]
#>    glmnet           2.0-16     2018-04-02 [1]
#>    glue             1.3.1.9000 2019-05-03 [1]
#>    gower            0.2.0      2019-03-07 [1]
#>    gplots           3.0.1.1    2019-01-27 [1]
#>    gridExtra        2.3        2017-09-09 [1]
#>    gtable           0.3.0      2019-03-25 [1]
#>    gtools           3.8.1      2018-06-26 [1]
#>    h2o            * 3.22.1.1   2019-01-10 [1]
#>    haven            2.1.0      2019-02-19 [1]
#>    highr            0.8        2019-03-20 [1]
#>    hms              0.4.2      2018-03-10 [1]
#>    htmltools        0.3.6      2017-04-28 [1]
#>    iml              0.9.0      2019-02-05 [1]
#>    inum             1.0-1      2019-04-25 [1]
#>    ipred            0.9-9      2019-04-28 [1]
#>    iterators        1.0.10     2018-07-13 [1]
#>    jsonlite         1.6        2018-12-07 [1]
#>    keras            2.2.4.1    2019-04-05 [1]
#>    kernlab          0.9-27     2018-08-10 [1]
#>    KernSmooth       2.23-15    2015-06-29 [1]
#>    knitr            1.23       2019-05-18 [1]
#>    labeling         0.3        2014-08-23 [1]
#>    lattice        * 0.20-38    2018-11-04 [1]
#>    lava             1.6.5      2019-02-12 [1]
#>    lazyeval         0.2.2      2019-03-15 [1]
#>    leaps            3.0        2017-01-10 [1]
#>    libcoin          1.0-4      2019-02-28 [1]
#>    lme4             1.1-21     2019-03-05 [1]
#>    lubridate        1.7.4      2018-04-11 [1]
#>    magrittr         1.5        2014-11-22 [1]
#>    maptools         0.9-5      2019-02-18 [1]
#>    markdown         1.0        2019-06-07 [1]
#>    MASS             7.3-51.4   2019-03-31 [1]
#>    Matrix           1.2-17     2019-03-22 [1]
#>    MatrixModels     0.4-1      2015-08-22 [1]
#>    mclust           5.4.3      2019-03-14 [1]
#>    Metrics          0.1.4      2018-07-09 [1]
#>    mgcv             1.8-28     2019-03-21 [1]
#>    mime             0.7        2019-06-11 [1]
#>    minqa            1.2.4      2014-10-09 [1]
#>    mlbench          2.1-1      2012-07-10 [1]
#>    ModelMetrics     1.2.2      2018-11-03 [1]
#>    munsell          0.5.0      2018-06-12 [1]
#>    mvtnorm          1.0-10     2019-03-05 [1]
#>    nlme             3.1-139    2019-04-09 [1]
#>    nloptr           1.2.1      2018-10-03 [1]
#>    nnet             7.3-12     2016-02-02 [1]
#>    numDeriv         2016.8-1   2016-08-27 [1]
#>    openxlsx         4.1.0.1    2019-05-28 [1]
#>    partykit         1.2-3      2019-01-31 [1]
#>    pbkrtest         0.4-7      2017-03-15 [1]
#>    pBrackets        1.0        2014-10-17 [1]
#>    pdp              0.7.0      2018-08-27 [1]
#>    pillar           1.4.1      2019-05-28 [1]
#>    pkgconfig        2.0.2      2018-08-16 [1]
#>    plogr            0.2.0      2018-03-25 [1]
#>    plotmo           3.5.4      2019-04-06 [1]
#>    plotrix          3.7-5      2019-04-07 [1]
#>    pls              2.7-1      2019-03-23 [1]
#>    plyr             1.8.4      2016-06-08 [1]
#>    polynom          1.4-0      2019-03-22 [1]
#>    prediction       0.3.6.2    2019-01-31 [1]
#>    prettyunits      1.0.2      2015-07-13 [1]
#>    pROC             1.14.0     2019-03-12 [1]
#>    processx         3.3.0      2019-03-10 [1]
#>    prodlim          2018.04.18 2018-04-18 [1]
#>    progress         1.2.2      2019-05-16 [1]
#>    ps               1.3.0      2018-12-21 [1]
#>    purrr          * 0.3.2      2019-03-15 [1]
#>    quantreg         5.38       2018-12-18 [1]
#>    R6               2.4.0      2019-02-14 [1]
#>    ranger           0.11.2     2019-03-07 [1]
#>    RColorBrewer     1.1-2      2014-12-07 [1]
#>    Rcpp             1.0.1      2019-03-17 [1]
#>    RcppEigen        0.3.3.5.0  2018-11-24 [1]
#>    RcppRoll         0.3.0      2018-06-05 [1]
#>    RCurl            1.95-4.12  2019-03-04 [1]
#>    readr          * 1.3.1      2018-12-21 [1]
#>    readxl           1.3.1      2019-03-13 [1]
#>    recipes          0.1.5      2019-03-21 [1]
#>    rematch          1.0.1      2016-04-21 [1]
#>    reshape2         1.4.3      2017-12-11 [1]
#>    reticulate       1.12       2019-04-12 [1]
#>    rio              0.5.16     2018-11-26 [1]
#>    rlang            0.3.4      2019-04-07 [1]
#>    rmarkdown        1.13       2019-05-22 [1]
#>    ROCR             1.0-7      2015-03-26 [1]
#>    rpart            4.1-15     2019-04-12 [1]
#>    rpart.plot       3.0.7      2019-04-12 [1]
#>    rsample        * 0.0.4      2019-01-07 [1]
#>    rstudioapi       0.10       2019-03-19 [1]
#>    scales           1.0.0      2018-08-09 [1]
#>    scatterplot3d    0.3-41     2018-03-14 [1]
#>    sp               1.3-1      2018-06-05 [1]
#>    SparseM          1.77       2017-04-23 [1]
#>    SQUAREM          2017.10-1  2017-10-07 [1]
#>    stringi          1.4.3      2019-03-12 [1]
#>    stringr        * 1.4.0      2019-02-10 [1]
#>    survival         2.44-1.1   2019-04-01 [1]
#>    TeachingDemos    2.10       2016-02-12 [1]
#>    tensorflow       1.13.1     2019-04-05 [1]
#>    tfestimators     1.9.1      2018-11-07 [1]
#>    tfruns           1.4        2018-08-25 [1]
#>    tibble         * 2.1.2      2019-05-29 [1]
#>    tidyr          * 0.8.3      2019-03-01 [1]
#>    tidyselect     0.2.5      2018-10-11 [1]
#>    timeDate       3043.102   2018-02-21 [1]
#>    tinytex        0.13       2019-05-14 [1]
#>    utf8           1.1.4      2018-05-24 [1]
#>    vctrs          0.1.0      2018-11-29 [1]
#>    vip            0.1.2.9000 2019-06-04 [1]
#>    viridis        0.5.1      2018-03-29 [1]
#>    viridisLite    0.3.0      2018-02-01 [1]
#>    whisker        0.3-2      2013-04-28 [1]
#>    withr          2.1.2      2018-03-15 [1]
#>    xfun           0.7        2019-05-14 [1]
#>    xgboost        0.82.1     2019-03-11 [1]
#>    yaImpute       1.0-31     2019-01-09 [1]
#>    yaml           2.2.0      2018-07-25 [1]
#>    zeallot        0.1.0      2018-01-28 [1]
#>    zip            2.0.2      2019-05-13 [1]
#>  source
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  Github (hadley/emo@02a5206)
#>  CRAN (R 3.6.0)
#>  <NA>
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  Github (tidyverse/glue@ea0edcb)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  Github (koalaverse/vip@9d537bb)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>  CRAN (R 3.6.0)
#>
#> [1] /Library/Frameworks/R.framework/Versions/3.6/Resources/library
#>
#>  R -- Package was removed from disk.
```

Part I

Fundamentals

1 Introduction to Machine Learning

Machine learning (ML) continues to grow in importance for many organizations across nearly all domains. Some example applications of machine learning in practice include:

- Predicting the likelihood of a patient returning to the hospital (*readmission*) within 30 days of discharge.
- Segmenting customers based on common attributes or purchasing behavior for targeted marketing.
- Predicting coupon redemption rates for a given marketing campaign.
- Predicting customer churn so an organization can perform preventative intervention.
- And many more!

In essence, these tasks all seek to learn from data. To address each scenario, we can use a given set of *features* to train an algorithm and extract insights. These algorithms, or *learners*, can be classified according to the amount and type of supervision needed during training. The two main groups this book focuses on are: ***supervised learners*** which construct predictive models, and ***unsupervised learners*** which build descriptive models. Which type you will need to use depends on the learning task you hope to accomplish.

1.1 Supervised learning

A ***predictive model*** is used for tasks that involve the prediction of a given output (or target) using other variables (or features) in the data set. Or, as stated by Kuhn and Johnson (2013, p. 2), predictive modeling is "…the process of developing a mathematical tool or model that generates an accurate prediction." The learning algorithm in a predictive model attempts to discover and model the relationships among the target variable (the variable being predicted) and the other features (aka predictor variables). Examples of predictive modeling include:

- using customer attributes to predict the probability of the customer churning in the next 6 weeks;
- using home attributes to predict the sales price;
- using employee attributes to predict the likelihood of attrition;
- using patient attributes and symptoms to predict the risk of readmission;
- using production attributes to predict time to market.

Each of these examples have a defined learning task; they each intend to use attributes (X) to predict an outcome measurement (Y).

Throughout this text we'll use various terms interchangeably for:

- X: "predictor variables", "independent variables", "attributes", "features", "predictors"
- Y: "target variable", "dependent variable", "response", "outcome measurement"

The predictive modeling examples above describe what is known as *supervised learning*. The supervision refers to the fact that the target values provide a supervisory role, which indicates to the learner the task it needs to learn. Specifically, given a set of data, the learning algorithm attempts to optimize a function (the algorithmic steps) to find the combination of feature values that results in a predicted value that is as close to the actual target output as possible.

In supervised learning, the training data you feed the algorithm includes the target values. Consequently, the solutions can be used to help *supervise* the training process to find the optimal algorithm parameters.

Most supervised learning problems can be bucketed into one of two categories: *regression* or *classification*, which we discuss next.

1.1.1 Regression problems

When the objective of our supervised learning is to predict a numeric outcome, we refer to this as a ***regression problem*** (not to be confused with linear regression modeling). Regression problems revolve around predicting output that falls on a continuum. In the examples above, predicting home sales prices and time to market reflect a regression problem because the output is numeric and continuous. This means, given the combination of predictor values, the response value could fall anywhere along some continuous spectrum (e.g., the predicted sales price of a particular home could be between $80,000 and $755,000). Figure 1.1 illustrates average home sales prices as a function of two home features: year built and total square footage. Depending on the combination of these two features, the expected home sales price could fall anywhere along a plane.

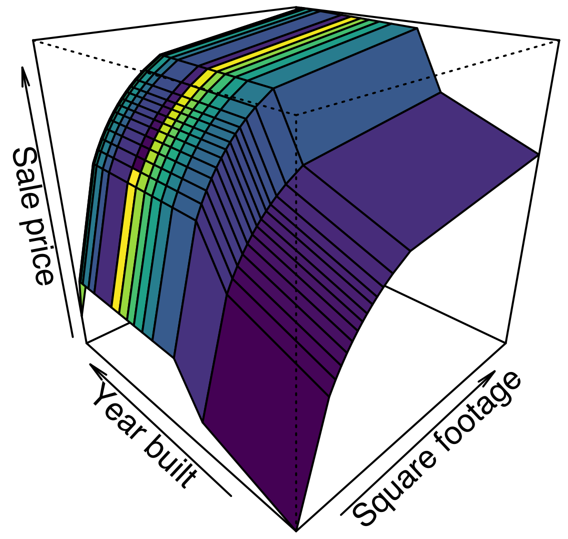


FIGURE 1.1: Average home sales price as a function of year built and total square footage.

1.1.2 Classification problems

When the objective of our supervised learning is to predict a categorical outcome, we refer to this as a ***classification problem***. Classification problems most commonly revolve around predicting a binary or multinomial response measure such as:

- Did a customer redeem a coupon (coded as yes/no or 1/0).
- Did a customer churn (coded as yes/no or 1/0).
- Did a customer click on our online ad (coded as yes/no or 1/0).
- Classifying customer reviews:
  - Binary: positive vs. negative.
  - Multinomial: extremely negative to extremely positive on a 0–5 Likert scale.

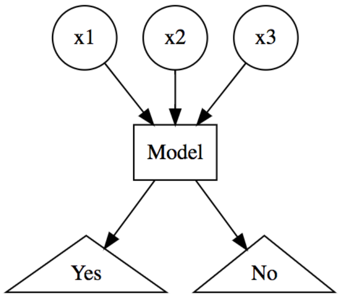


FIGURE 1.2: Classification problem modeling 'Yes'/'No' response based on three features.

However, when we apply machine learning models for classification problems, rather than predict a particular class (i.e., “yes” or “no”), we often want to predict the *probability* of a particular class (i.e., yes: 0.65, no: 0.35). By default, the class with the highest predicted probability becomes the predicted class. Consequently, even though we are performing a classification problem, we are still predicting a numeric output (probability). However, the essence of the problem still makes it a classification problem.

Although there are machine learning algorithms that can be applied to regression problems but not classification and vice versa, most of the supervised learning algorithms we cover in this book can be applied to both. These algorithms have become the most popular machine learning applications in recent years.

1.2 Unsupervised learning

Unsupervised learning, in contrast to supervised learning, includes a set of statistical tools to better understand and describe your data, but performs the analysis without a target variable. In essence, unsupervised learning is concerned with identifying groups in a data set. The groups may be defined by the rows (i.e., *clustering*) or the columns (i.e., *dimension reduction*); however, the motive in each case is quite different.

The goal of ***clustering*** is to segment observations into similar groups based on the observed variables; for example, to divide consumers into different homogeneous groups, a process known as market segmentation. In **dimension reduction**, we are often concerned with reducing the number of variables in a data set. For example, classical linear regression models break down in the presence of highly correlated features. Some dimension reduction techniques can be used to reduce the feature set to a potentially smaller set of uncorrelated variables. Such a reduced feature set is often used as input to downstream supervised learning models (e.g., principal component regression).

Unsupervised learning is often performed as part of an exploratory data analysis (EDA). However, the exercise tends to be more subjective, and there is no simple goal for the analysis, such as prediction of a response. Furthermore, it can be hard to assess the quality of results obtained from unsupervised learning methods. The reason for this is simple. If we fit a predictive model using a supervised learning technique (i.e., linear regression), then it is possible to check our work by seeing how well our model predicts the response Y on observations not used in fitting the model. However, in unsupervised learning, there is no way to check our work because we don't know the true answer—the problem is unsupervised!

Despide its subjectivity, the importance of unsupervised learning should not be overlooked and such techniques are often used in organizations to:

- Divide consumers into different homogeneous groups so that tailored marketing strategies can be developed and deployed for each segment.
- Identify groups of online shoppers with similar browsing and purchase histories, as well as items that are of particular interest to the shoppers within each group. Then an individual shopper can be preferentially shown the items in which he or she is particularly likely to be interested, based on the purchase histories of similar shoppers.
- Identify products that have similar purchasing behavior so that managers can manage them as product groups.

These questions, and many more, can be addressed with unsupervised learning. Moreover, the outputs of an unsupervised learning models can be used as inputs to downstream supervised learning models.

1.3 Roadmap

The goal of this book is to provide effective tools for uncovering relevant and useful patterns in your data by using R's ML stack. We begin by providing an overview of the ML modeling process and discussing fundamental concepts that will carry through the rest of the book. These include feature engineering, data splitting, model validation and tuning, and performance measurement. These concepts will be discussed in Chapters 2-**??**.

Chapters **??**-**??** focus on common supervised learners ranging from simpler linear regression models to the more complicated gradient boosting machines and deep neural networks. Here we will illustrate the fundamental concepts of each base learning algorithm and how to tune its hyperparameters to maximize predictive performance.

Chapters **??**-**??** delve into more advanced approaches to maximize effectiveness, efficiency, and interpretation of your ML models. We discuss how to combine multiple models to create a stacked model (aka *super learner*), which allows you to combine the strengths from each base learner and further maximize predictive accuracy. We then illustrate how to make the training and validation process more efficient with automated ML (aka AutoML). Finally, we illustrate many ways to extract insight from your "black box" models with various ML interpretation techniques.

The latter part of the book focuses on unsupervised techniques aimed at reducing the dimensions of your data for more effective data representation (Chapters **??**-**??**) and identifying common groups among your observations with clustering techniques (Chapters **??**-**??**).

1.4 The data sets

The data sets chosen for this book allow us to illustrate the different features of the presented machine learning algorithms. Since the goal of this book is to demonstrate how to implement R's ML stack, we make the assumption that you have already spent significant time cleaning and getting to know your data via EDA. This would allow you to perform many necessary tasks prior to the ML tasks outlined in this book such as:

- Feature selection (i.e., removing unnecessary variables and retaining only those variables you wish to include in your modeling process).
- Recoding variable names and values so that they are meaningful and more interpretable.
- Recoding, removing, or some other approach to handling missing values.

Consequently, the exemplar data sets we use throughout this book have, for the most part, gone through the necessary cleaning processes. In some cases we illustrate concepts with stereotypical data sets (i.e. `mtcars`, `iris`, `geyser`); however, we tend to focus most of our discussion around the following data sets:

- Property sales information as described in De Cock (2011).

  - **problem type**: supervised regression
  - **response variable**: `Sale_Price` (i.e., $195,000, $215,000)
  - **features**: 80
  - **observations**: 2,930
  - **objective**: use property attributes to predict the sale price of a home
  - **access**: provided by the **AmesHousing** package (Kuhn, 2017a)
  - **more details**: See `?AmesHousing::ames_raw`

```r
# access data
ames <- AmesHousing::make_ames()

# initial dimension
dim(ames)
## [1] 2930   81

# response variable
head(ames$Sale_Price)
## [1] 215000 105000 172000 244000 189900 195500
```

  - You can see the entire data cleaning process to transform the raw Ames housing data (`AmesHousing::ames_raw`) to the final clean data (`AmesHousing::make_ames`) that we will use in machine learning algorithms throughout this book at:
    https://github.com/topepo/AmesHousing/blob/master/R/make_ames.R

- Employee attrition information originally provided by IBM Watson Analytics Lab[1].

  - **problem type**: supervised binomial classification
  - **response variable**: `Attrition` (i.e., "Yes", "No")
  - **features**: 30
  - **observations**: 1,470
  - **objective**: use employee attributes to predict if they will attrit (leave the company)
  - **access**: provided by the **rsample** package (Kuhn and Wickham, 2017)
  - **more details**: See `?rsample::attrition`

[1] https://www.ibm.com/communities/analytics/watson-analytics-blog/hr-employee-attrition/

```
# access data
attrition <- rsample::attrition

# initial dimension
dim(attrition)
## [1] 1470   31

# response variable
head(attrition$Attrition)
## [1] Yes No  Yes No  No  No
## Levels: No Yes
```

- Image information for handwritten numbers originally presented to AT&T Bell Lab's to help build automatic mail-sorting machines for the USPS. Has been used since early 1990s to compare machine learning performance on pattern recognition (i.e., LeCun et al. (1990); LeCun et al. (1998); Cireşan et al. (2012)).
  - **Problem type**: supervised multinomial classification
  - **response variable**: `V785` (i.e., numbers to predict: 0, 1, …, 9)
  - **features**: 784
  - **observations**: 60,000 (train) / 10,000 (test)
  - **objective**: use attributes about the "darkness" of each of the 784 pixels in images of handwritten numbers to predict if the number is 0, 1, …, or 9.
  - **access**: provided by the `dslabs` package (Irizarry, 2018)
  - **more details**: See `?dslabs::read_mnist()` and online MNIST documentation[2]

```
#access data
mnist <- dslabs::read_mnist()
names(mnist)
## [1] "train" "test"

# initial feature dimensions
dim(mnist$train$images)
## [1] 60000   784

# response variable
head(mnist$train$labels)
## [1] 5 0 4 1 9 2
```

[2]http://yann.lecun.com/exdb/mnist/

- Grocery items and quantities purchased. Each observation represents a single basket of goods that were purchased together.
  - **Problem type**: unsupervised basket analysis
  - **response variable**: NA
  - **features**: 42
  - **observations**: 2,000
  - **objective**: use attributes of each basket to identify common groupings of items purchased together.
  - **access**: available via additional online material

```
# access data
my_basket <- readr::read_csv("data/my_basket.csv")

# initial dimension
dim(my_basket)
## [1] 2000   42

# response variable
my_basket
## # A tibble: 2,000 x 42
##    `7up` lasagna pepsi   yop `red-wine` cheese   bbq
##    <dbl>   <dbl> <dbl> <dbl>      <dbl>  <dbl> <dbl>
##  1     0       0     0     0          0      0     0
##  2     0       0     0     0          0      0     0
##  3     0       0     0     0          0      0     0
##  4     0       0     0     2          1      0     0
##  5     0       0     0     0          0      0     0
##  6     0       0     0     0          0      0     0
##  7     1       1     0     0          0      0     1
##  8     0       0     0     0          0      0     0
##  9     0       1     0     0          0      0     0
## 10     0       0     0     0          0      0     0
## # ... with 1,990 more rows, and 35 more variables:
## #   bulmers <dbl>, mayonnaise <dbl>, horlics <dbl>,
## #   `chicken-tikka` <dbl>, milk <dbl>, mars <dbl>,
## #   coke <dbl>, lottery <dbl>, bread <dbl>,
## #   pizza <dbl>, `sunny-delight` <dbl>, ham <dbl>,
## #   lettuce <dbl>, kronenbourg <dbl>, leeks <dbl>,
## #   fanta <dbl>, tea <dbl>, whiskey <dbl>, peas <dbl>,
## #   newspaper <dbl>, muesli <dbl>, `white-wine` <dbl>,
## #   carrots <dbl>, spinach <dbl>, pate <dbl>,
## #   `instant-coffee` <dbl>, twix <dbl>,
## #   potatoes <dbl>, fosters <dbl>, soup <dbl>,
## #   `toad-in-hole` <dbl>, `coco-pops` <dbl>,
## #   kitkat <dbl>, broccoli <dbl>, cigarettes <dbl>
```

2 Modeling Process

Much like EDA, the ML process is very iterative and heurstic-based. With minimal knowledge of the problem or data at hand, it is difficult to know which ML method will perform best. This is known as the *no free lunch* theorem for ML (Wolpert, 1996). Consequently, it is common for many ML approaches to be applied, evaluated, and modified before a final, optimal model can be determined. Performing this process correctly provides great confidence in our outcomes. If not, the results will be useless and, potentially, damaging [1].

Approaching ML modeling correctly means approaching it strategically by spending our data wisely on learning and validation procedures, properly pre-processing the feature and target variables, minimizing *data leakage* (Section **??**), tuning hyperparameters, and assessing model performance. Many books and courses portray the modeling process as a short sprint. A better analogy would be a marathon where many iterations of these steps are repeated before eventually finding the final optimal model. This process is illustrated in Figure 2.1. Before introducing specific algorithms, this chapter, and the next, introduce concepts that are fundamental to the ML modeling process and that you'll see briskly covered in future modeling chapters.

Although the discussions in this chapter focuses on supervised ML modeling, many of the topics also apply to unsupervised methods.

2.1 Prerequisites

This chapter leverages the following packages.

[1]See `https://www.fatml.org/resources/relevant-scholarship` for many discussions regarding implications of poorly applied and interpreted ML.

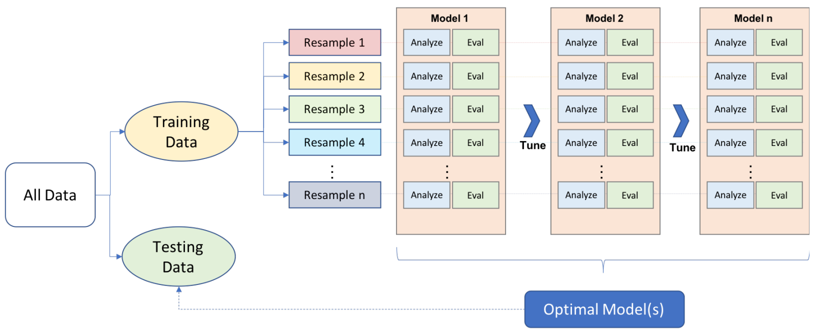


FIGURE 2.1: General predictive machine learning process.

```
# Helper packages
library(dplyr)      # for data manipulation
library(ggplot2)    # for awesome graphics

# Modeling process packages
library(rsample)    # for resampling procedures
library(caret)      # for resampling and model training
library(h2o)        # for resampling and model training

# h2o set-up
h2o.no_progress()   # turn off h2o progress bars
h2o.init()          # launch h2o
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         17 minutes 19 seconds
##     H2O cluster timezone:       America/New_York
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.22.1.1
##     H2O cluster version age:    5 months and 28 days !!!
##     H2O cluster name:           H2O_started_from_R_b294776_vmp196
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   3.28 GB
##     H2O cluster total cores:    8
##     H2O cluster allowed cores:  8
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
##     H2O Connection proxy:       NA
##     H2O Internal Security:      FALSE
##     H2O API Extensions:         XGBoost, Algos, AutoML, Core V3, Core V4
##     R Version:                  R version 3.6.0 (2019-04-26)
```

To illustrate some of the concepts, we'll use the Ames Housing and employee attrition data sets introduced in Chapter 1. Throughout this book, we'll demonstrate approaches with ordinary R data frames. However, since many of the supervised machine learning chapters leverage the **h2o** package, we'll also show how to do some of the tasks with H2O objects. You can convert any R data frame to an H2O object (i.e., import it to the H2O cloud) easily with `as.h2o(<my-data-frame>)`.

If you try to convert the original `rsample::attrition` data set to an H2O object an error will occur. This is because several variables are *ordered factors* and H2O has no way of handling this data type. Consequently, you must convert any ordered factors to unordered; see `?base::ordered` for details.

```
# ames data
ames <- AmesHousing::make_ames()
ames.h2o <- as.h2o(ames)

# attrition data
churn <- rsample::attrition %>%
  mutate_if(is.ordered, factor, ordered = FALSE)
churn.h2o <- as.h2o(churn)
```

2.2 Data splitting

A major goal of the machine learning process is to find an algorithm $f(X)$ that most accurately predicts future values ($\hat{Y}$) based on a set of features (X). In other words, we want an algorithm that not only fits well to our past data, but more importantly, one that predicts a future outcome accurately. This is called the ***generalizability*** of our algorithm. How we "spend" our data will help us understand how well our algorithm generalizes to unseen data.

To provide an accurate understanding of the generalizability of our final optimal model, we can split our data into training and test data sets:

- **Training set**: these data are used to develop feature sets, train our algorithms, tune hyperparameters, compare models, and all of the other activities required to choose a final model (e.g., the model we want to put into production).
- **Test set**: having chosen a final model, these data are used to estimate an unbiased assessment of the model's performance, which we refer to as the *generalization error*.

It is critical that the test set not be used prior to selecting your final model. Assessing results on the test set prior to final model selection biases the model selection process since the testing data will have become part of the model development process.

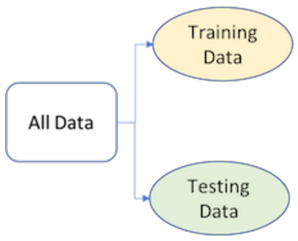


FIGURE 2.2: Splitting data into training and test sets.

Given a fixed amount of data, typical recommendations for splitting your data into training-test splits include 60% (training)–40% (testing), 70%–30%, or 80%–20%. Generally speaking, these are appropriate guidelines to follow; however, it is good to keep the following points in mind:

- Spending too much in training (e.g., $> 80\%$) won't allow us to get a good assessment of predictive performance. We may find a model that fits the training data very well, but is not generalizable (*overfitting*).
- Sometimes too much spent in testing ($> 40\%$) won't allow us to get a good assessment of model parameters.

Other factors should also influence the allocation proportions. For example, very large training sets (e.g., $n > 100\text{K}$) often result in only marginal gains compared to smaller sample sizes. Consequently, you may use a smaller training sample to increase computation speed (e.g., models built on larger training sets often take longer to score new data sets in production). In contrast, as $p \geq n$ (where p represents the number of features), larger samples sizes are often required to identify consistent signals in the features.

The two most common ways of splitting data include ***simple random sampling*** and ***stratified sampling***.

2.2.1 Simple random sampling

The simplest way to split the data into training and test sets is to take a simple random sample. This does not control for any data attributes, such as the distribution your response variable (Y). There are multiple ways to split our data in R. Here we show four options to produce a 70–30 split in the Ames housing data:

Sampling is a random process so setting the random number generator with a common seed allows for reproducible results. Throughout this book we'11 often use the seed `123` for reproducibility but the number itself has no special meaning.

```
# base R
set.seed(123)
index_1 <- sample(1:nrow(ames), round(nrow(ames) * 0.7))
train_1 <- ames[index_1, ]
test_1  <- ames[-index_1, ]

# caret package
set.seed(123)
index_2 <- createDataPartition(ames$Sale_Price, p = 0.7, list = FALSE)
train_2 <- ames[index_2, ]
test_2  <- ames[-index_2, ]

# rsample package
set.seed(123)
split_1  <- initial_split(ames, prop = 0.7)
train_3  <- training(split_1)
test_3   <- testing(split_1)

# h2o package
split_2 <- h2o.splitFrame(ames.h2o, ratios = 0.7, seed = 123)
train_4 <- split_2[[1]]
test_4  <- split_2[[2]]
```

With sufficient sample size, this sampling approach will typically result in a similar distribution of Y (e.g., `Sale_Price` in the `ames` data) between your training and test sets, as illustrated below.

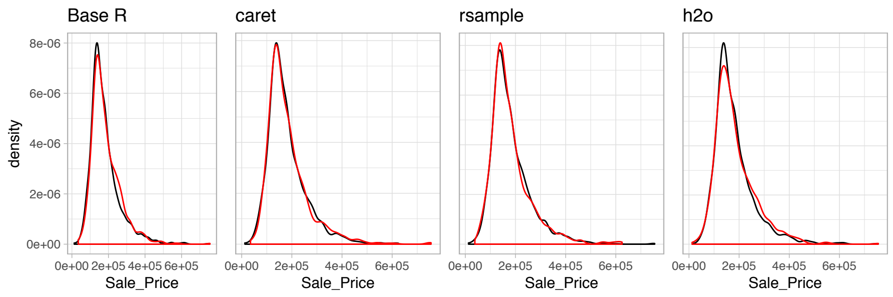


FIGURE 2.3: Training (black) vs. test (red) response distribution.

2.2.2 Stratified sampling

If we want to explicitly control the sampling so that our training and test sets have similar Y distributions, we can use stratified sampling. This is more common with classification problems where the response variable may be severely imbalanced (e.g., 90% of observations with response "Yes" and 10% with response "No"). However, we can also apply stratified sampling to regression problems for data sets that have a small sample size and where the response variable deviates strongly from normality (i.e., positively skewed like `Sale_Price`). With a continuous response variable, stratified sampling will segment Y into quantiles and randomly sample from each. Consequently, this will help ensure a balanced representation of the response distribution in both the training and test sets.

The easiest way to perform stratified sampling on a response variable is to use the **rsample** package, where you specify the response variable to `strata`fy. The following illustrates that in our original employee attrition data we have an imbalanced response (No: 84%, Yes: 16%). By enforcing stratified sampling, both our training and testing sets have approximately equal response distributions.

```
# orginal response distribution
table(churn$Attrition) %>% prop.table()
##
##     No    Yes
## 0.8388 0.1612

# stratified sampling with the rsample package
set.seed(123)
split_strat  <- initial_split(churn, prop = 0.7, strata = "Attrition")
train_strat  <- training(split_strat)
test_strat   <- testing(split_strat)
```

```
# consistent response ratio between train & test
table(train_strat$Attrition) %>% prop.table()
##
##     No    Yes
## 0.8388 0.1612
table(test_strat$Attrition) %>% prop.table()
##
##     No    Yes
## 0.8386 0.1614
```

2.2.3 Class imbalances

Imbalanced data can have a significant impact on model predictions and performance (Kuhn and Johnson, 2013). Most often this involves classification problems where one class has a very small proportion of observations (e.g., defaults - 5% versus nondefaults - 95%). Several sampling methods have been developed to help remedy class imbalance and most of them can be categorized as either *up-sampling* or *down-sampling*.

Down-sampling balances the dataset by reducing the size of the abundant class(es) to match the frequencies in the least prevalent class. This method is used when the quantity of data is sufficient. By keeping all samples in the rare class and randomly selecting an equal number of samples in the abundant class, a balanced new dataset can be retrieved for further modeling. Furthermore, the reduced sample size reduces the computation burden imposed by further steps in the ML process.

On the contrary, up-sampling is used when the quantity of data is insufficient. It tries to balance the dataset by increasing the size of rarer samples. Rather than getting rid of abundant samples, new rare samples are generated by using repetition or bootstrapping (described further in Section 2.4.2).

Note that there is no absolute advantage of one sampling method over another. Application of these two methods depends on the use case it applies to and the data set itself. A combination of over- and under-sampling is often successful and a common approach is known as Synthetic Minority Over-Sampling Technique, or SMOTE (Chawla et al., 2002). This alternative sampling approach, as well as others, can be implemented in R (see the `sampling` argument in `?caret::trainControl()`). Furthermore, many ML algorithms implemented in R have class weighting schemes to remedy imbalances internally (e.g., most **h2o** algorithms have a `weights_column` and `balance_classes` argument).

2.3 Creating models in R

The R ecosystem provides a wide variety of ML algorithm implementations. This makes many powerful algorithms available at your fingertips. Moreover, there are almost always more than one package to perform each algorithm (e.g., there are over 20 packages for fitting random forests). There are pros and cons to this wide selection; some implementations may be more computationally efficient while others may be more flexible (i.e., have more hyper-parameter tuning options). Future chapters will expose you to many of the packages and algorithms that perform and scale best to the kinds of tabular data and problems encountered by most organizations.

However, this also has resulted in some drawbacks as there are inconsistencies in how algorithms allow you to define the formula of interest and how the results and predictions are supplied. In addition to illustrating the more popular and powerful packages, we'll also show you how to use implementations that provide more consistency.

2.3.1 Many formula interfaces

To fit a model to our data, the model terms must be specified. Historically, there are two main interfaces for doing this. The formula interface using R formula rules to specify a symbolic representation of the terms. For example, `Y ~ X` where we say "Y is a function of X". To illustrate, suppose we have some generic modeling function called `model_fn()` which accepts an R formula, as in the following examples:

```r
# sale price as a function of neighborhood and year sold
model_fn(Sale_Price ~ Neighborhood + Year_Sold, data = ames)

# Variables + interactions
model_fn(Sale_Price ~ Neighborhood + Year_Sold + Neighborhood:Year_Sold, data = ames)

# Shorthand for all predictors
model_fn(Sale_Price ~ ., data = ames)

# Inline functions / transformations
model_fn(log10(Sale_Price) ~ ns(Longitude, df = 3) + ns(Latitude, df = 3), data = ames)
```

This is very convenient but it has some disadvantages. For example:

- You can't nest in-line functions such as performing principal components analysis on the feature set prior to executing the model (`model_fn(y ~ pca(scale(x1), scale(x2), scale(x3)), data = df)`).
- All the model matrix calculations happen at once and can't be recycled when used in a model function.
- For very wide data sets, the formula method can be extremely inefficient (Kuhn, 2017b).
- There are limited roles that variables can take which has led to several re-implementations of formulas.
- Specifying multivariate outcomes is clunky and inelegant.
- Not all modeling functions have a formula method (lack of consistency!).

Some modeling functions have a non-formula (XY) interface. These functions have separate arguments for the predictors and the outcome(s):

```r
# use separate inputs for X and Y
features <- c("Year_Sold", "Longitude", "Latitude")
model_fn(x = ames[, features], y = ames$Sale_Price)
```

This provides more efficient calculations but can be inconvenient if you have transformations, factor variables, interactions, or any other operations to apply to the data prior to modeling.

Overall, it is difficult to determine if a package has one or both of these interfaces. For example, the `lm()` function, which performs linear regression, only has the formula method. Consequently, until you are familiar with a particular implementation you will need to continue referencing the corresponding help documentation.

A third interface, is to use *variable name specification* where we provide all the data combined in one training frame but we specify the features and response with character strings. This is the interface used by the **h2o** package.

```r
model_fn(
  x = c("Year_Sold", "Longitude", "Latitude"),
  y = "Sale_Price",
  data = ames.h2o
  )
```

One approach to get around these inconsistencies is to use a meta engine, which we discuss next.

2.3.2 Many engines

Although there are many individual ML packages available, there is also an abundance of meta engines that can be used to help provide consistency. For example, the following all produce the same linear regression model output:

```r
lm_lm    <- lm(Sale_Price ~ ., data = ames)
lm_glm   <- glm(Sale_Price ~ ., data = ames, family = gaussian)
lm_caret <- train(Sale_Price ~ ., data = ames, method = "lm")
```

Here, `lm()` and `glm()` are two different algorithm engines that can be used to fit the linear model and `caret::train()` is a meta engine (aggregator) that allows you to apply almost any direct engine with `method = "<method-name>"`. There are trade-offs to consider when using direct versus meta engines. For example, using direct engines can allow for extreme flexibility but also requires you to familiarize yourself with the unique differences of each implementation. For example, the following highlights the various syntax nuances required to compute and extract predicted class probabilities across different direct engines.[2]

TABLE 2.1: Table 1: Syntax for computing predicted class probabilities with direct engines.

Algorithm	Package	Code
Linear discriminant analysis	**MASS**	`predict(obj)`
Generalized linear model	**stats**	`predict(obj, type = "response")`
Mixture discriminant analysis	**mda**	`predict(obj, type = "posterior")`
Decision tree	**rpart**	`predict(obj, type = "prob")`
Random Forest	**ranger**	`predict(obj)$predictions`
Gradient boosting machine	**gbm**	`predict(obj, type = "response", n.trees)`

Meta engines provide you with more consistency in how you specify inputs and extract outputs but can be less flexible than direct engines. Future chapters will illustrate both approaches. For meta engines, we'll focus on the **caret** package in the hardcopy of the book while also demonstrating the newer **parsnip** package in the additional online resources.[3]

[2]This table was modified from Kuhn (2019)

[3]The **caret** package has been the preferred meta engine over the years; however, the author is now transitioning to fulltime development on **parsnip**, which is designed to be a more robust and tidy meta engine.

2.4 Resampling methods

In section 2.2 we split our data into training and testing sets. Furthermore, we were very explicit about the fact that we ***do not*** use the test set to assess model performance during the training phase. So how do we assess the generalization performance of the model?

One option is to assess an error metric based on the training data. Unfortunately, this leads to biased results as some models can perform very well on the training data but not generalize well to a new data set (we'll illustrate this in Section **??**).

A second method is to use a *validation* approach, which involves splitting the training set further to create two parts (as in Section 2.2): a training set and a validation set (or *holdout set*). We can then train our model(s) on the new training set and estimate the performance on the validation set. Unfortunately, validation using a single holdout set can be highly variable and unreliable unless you are working with very large data sets (Molinaro et al., 2005; Hawkins et al., 2003). As the size of your data set reduces, this concern increases.

Although we stick to our definitions of test, validation, and holdout sets, these terms are sometimes used interchangeably in other literature and software. What's important to remember is to always put a portion of the data under lock and key until a final model has been selected (we refer to this as the test data, but others refer to it as the holdout set).

Resampling methods provide an alternative approach by allowing us to repeatedly fit a model of interest to parts of the training data and testing the performance on other parts. The two most commonly used resampling methods include *k-fold cross validation* and *bootstrapping*.

2.4.1 *k*-fold cross validation

k-fold cross-validation (aka k-fold CV) is a resampling method that randomly divides the training data into k groups (aka folds) of approximately equal size. The model is fit on $k-1$ folds and then the remaining fold is used to compute model performance. This procedure is repeated k times; each time, a different fold is treated as the validation set. This process results in k estimates of the generalization error (say $\epsilon_1, \epsilon_2, \dots, \epsilon_k$). Thus, the k-fold CV estimate is

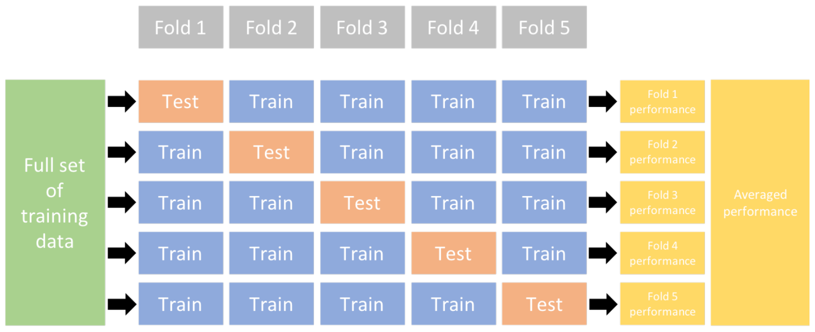


FIGURE 2.4: Illustration of the k-fold cross validation process.

computed by averaging the k test errors, providing us with an approximation of the error we might expect on unseen data.

Consequently, with k-fold CV, every observation in the training data will be held out one time to be included in the test set as illustrated in Figure 2.5. In practice, one typically uses $k = 5$ or $k = 10$. There is no formal rule as to the size of k; however, as k gets larger, the difference between the estimated performance and the true performance to be seen on the test set will decrease. On the other hand, using too large of k can introduce computational burdens. Moreover, Molinaro et al. (2005) found that $k = 10$ performed similarly to leave-one-out cross validation (LOOCV) which is the most extreme approach (i.e., setting $k = n$).

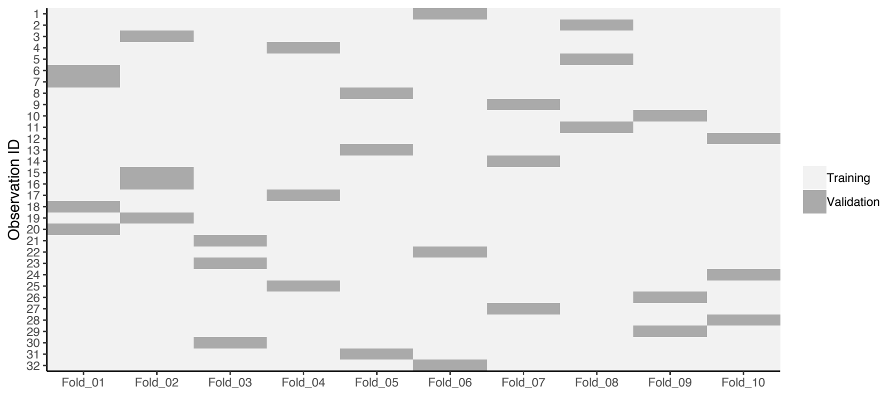


FIGURE 2.5: 10-fold cross validation on 32 observations. Each observation is used once for validation and nine times for training.

Although using $k \geq 10$ helps to minimize the variability in the estimated performance, k-fold CV still tends to have higher variability than bootstrapping (discussed next). Kim (2009) showed that repeating k-fold CV can help to increase the precision of the estimated generalization error. Consequently, for smaller data sets (say $n < 10,000$), 10-fold CV repeated 5 or 10 times will improve the accuracy of your estimated performance and also provide an estimate of its variability.

Throughout this book we'll cover multiple ways to incorporate CV as you can often perform CV directly within certain ML functions:

```
# example in h2o
h2o.cv <- h2o.glm(
  x = x,
  y = y,
  training_frame = ames.h2o,
  nfolds = 10  # perform 10-fold CV
)
```

Or externally as in the below chunk[4]. When applying it externally to an ML algorithm as below, we'll need a process to apply the ML model to each resample, which we'll also cover.

```
vfold_cv(ames, v = 10)
## #  10-fold cross-validation
## # A tibble: 10 x 2
##    splits             id
##    <list>             <chr>
##  1 <split [2.6K/293]> Fold01
##  2 <split [2.6K/293]> Fold02
##  3 <split [2.6K/293]> Fold03
##  4 <split [2.6K/293]> Fold04
##  5 <split [2.6K/293]> Fold05
##  6 <split [2.6K/293]> Fold06
##  7 <split [2.6K/293]> Fold07
##  8 <split [2.6K/293]> Fold08
##  9 <split [2.6K/293]> Fold09
## 10 <split [2.6K/293]> Fold10
```

[4]`rsample::vfold_cv()` results in a nested data frame where each element in `splits` is a list containing the training data frame and the observation IDs that will be used for training the model vs. model validation.

2.4.2 Bootstrapping

A bootstrap sample is a random sample of the data taken *with replacement* (Efron and Tibshirani, 1986). This means that, after a data point is selected for inclusion in the subset, it's still available for further selection. A bootstrap sample is the same size as the original data set from which it was constructed. Figure 2.6 provides a schematic of bootstrap sampling where each bootstrap sample contains 12 observations just as in the original data set. Furthermore, bootstrap sampling will contain approximately the same distribution of values (represented by colors) as the original data set.

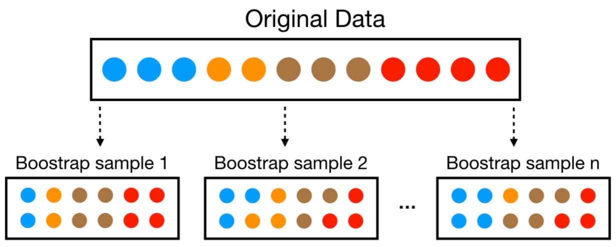


FIGURE 2.6: Illustration of the bootstrapping process.

Since samples are drawn with replacement, each bootstrap sample is likely to contain duplicate values. In fact, on average, $\approx 63.21\%$ of the original sample ends up in any particular bootstrap sample. The original observations not contained in a particular bootstrap sample are considered *out-of-bag* (OOB). When bootstrapping, a model can be built on the selected samples and validated on the OOB samples; this is often done, for example, in random forests (**??**).

Since observations are replicated in bootstrapping, there tends to be less variability in the error measure compared with k-fold CV (Efron, 1983). However, this can also increase the bias of your error estimate. This can be problematic with smaller data sets; however, for most average-to-large data sets (say $n \geq 1,000$) this concern is often negligable.

Figure 2.7 compares bootstrapping to 10-fold CV on a small data set with $n = 32$ observations. A thorough introduction to the bootstrap and its use in R is provided in Davison et al. (1997).

We can create bootstrap samples easily with `rsample::bootstraps()`;

```
bootstraps(ames, times = 10)
## # Bootstrap sampling
## # A tibble: 10 x 2
##    splits               id
```

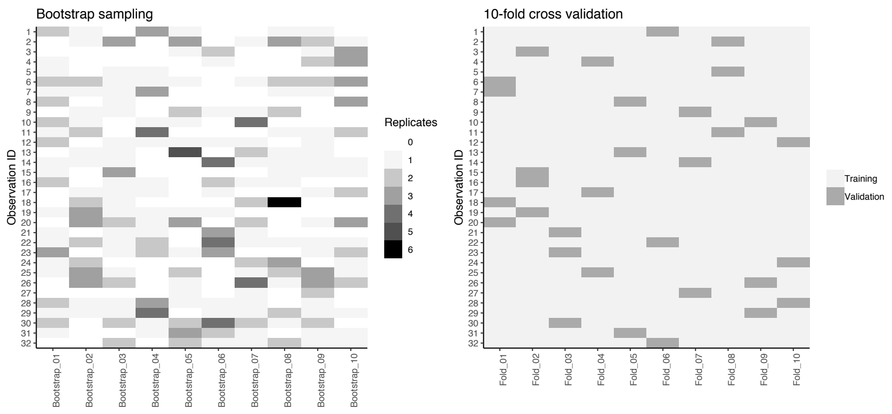


FIGURE 2.7: Bootstrap sampling (left) versus 10-fold cross validation (right) on 32 observations. For bootstrap sampling, the observations that have zero replications (white) are the out-of-bag observations used for validation.

```
##    <list>               <chr>
##  1 <split [2.9K/1.1K]> Bootstrap01
##  2 <split [2.9K/1.1K]> Bootstrap02
##  3 <split [2.9K/1.1K]> Bootstrap03
##  4 <split [2.9K/1K]>   Bootstrap04
##  5 <split [2.9K/1.1K]> Bootstrap05
##  6 <split [2.9K/1.1K]> Bootstrap06
##  7 <split [2.9K/1.1K]> Bootstrap07
##  8 <split [2.9K/1.1K]> Bootstrap08
##  9 <split [2.9K/1.1K]> Bootstrap09
## 10 <split [2.9K/1K]>   Bootstrap10
```

Bootstrapping is, typically, more of an internal resampling procedure that is naturally built into certain ML algorithms. This will become more apparent in the bagging and random forest chapters (**??**-**??**).

2.4.3 Alternatives

Its important to note that there are other useful resampling procedures. If you're working with time-series specific data then you will want to incorporate rolling origin and other time series resampling procedures. Hyndman and Athanasopoulos (2018) is the dominant, R-focused, time series resource[5].

[5]See their open source book at `https://www.otexts.org/fpp2`

Additionally, Efron (1983) developed the "632 method" and Efron and Tibshirani (1997) discuss the "632+ method"; both approaches seek to minimize biases experienced with bootstrapping on smaller data sets and are available via **caret** (see `?caret::trainControl` for details).

2.5 Bias variance trade-off {#bias-var}

Prediction errors can be decomposed into two important subcomponents: error due to "bias" and error due to "variance". There is often a tradeoff between a model's ability to minimize bias and variance. Understanding how different sources of error lead to bias and variance helps us improve the data fitting process resulting in more accurate models.

2.5.1 Bias

Bias is the difference between the expected (or average) prediction of our model and the correct value which we are trying to predict. It measures how far off in general a model's predictions are from the correct value, which provides a sense of how well a model can conform to the underlying structure of the data. Figure 2.8 illustrates an example where the polynomial model does not capture the underlying structure well. Linear models are classical examples of high bias models as they are less flexible and rarely capture non-linear, non-monotonic relationships.

We also need to think of bias-variance in relation to resampling. Models with high bias are rarely effected by the noise introduced by resampling. If a model has high bias, it will have consistency in its resampling performance as illustrated by Figure 2.8.

2.5.2 Variance

On the other hand, error due to *variance* is defined as the variability of a model prediction for a given data point. Many models (e.g., k-nearest neighbor, decision trees, gradient boosting machines) are very adaptable and offer extreme flexibility in the patterns that they can fit to. However, these models offer their own problems as they run the risk of overfitting to the training data. Although you may achieve very good performance on your training data, the model will not automatically generalize well to unseen data.

Since high variance models are more prone to overfitting, using resampling

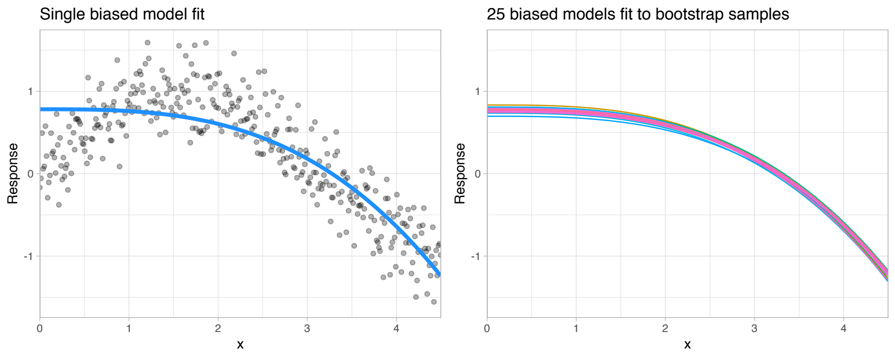


FIGURE 2.8: A biased polynomial model fit to a single data set does not capture the underlying non-linear, non-monotonic data structure (left). Models fit to 25 bootstrapped replicates of the data are underterred by the noise and generates similar, yet still biased, predictions (right).

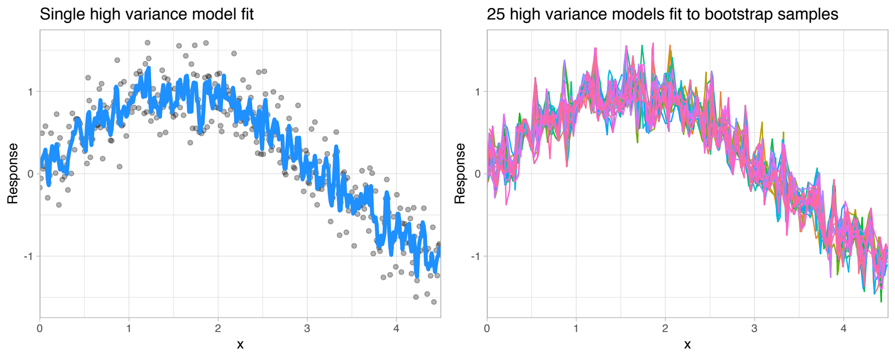


FIGURE 2.9: A high variance k-nearest neighbor model fit to a single data set captures the underlying non-linear, non-monotonic data structure well but also overfits to individual data points (left). Models fit to 25 bootstrapped replicates of the data are deterred by the noise and generate highly variable predictions (right).

procedures are critical to reduce this risk. Moreover, many algorithms that are capable of achieving high generalization performance have lots of *hyperparameters* that control the level of model complexity (i.e., the tradeoff between bias and variance).

2.5.3 Hyperparameter tuning

Hyperparameters (aka *tuning parameters*) are the "knobs to twiddle"[6] to control the complexity of machine learning algorithms and, therefore, the bias-variance trade-off. Not all algorithms have hyperparameters (e.g., ordinary least squares[7]); however, most have at least one or more.

The proper setting of these hyperparameters are often dependent on the data and problem at hand and cannot always be estimated by the training data alone. Consequently, we need a method of identifying the optimal setting. For example, in the high variance example in the previous section, we illustrated a high variance k-nearest neighbor model (we'll discuss k-nearest neighbor in Chapter **??**). k-nearest neighbor models have a single hyperparameter (k) that determines the predicted value to be made based on the k nearest observations in the training data to the one being predicted. If k is small (e.g., $k = 3$), the model will make a prediction for a given observation based on the average of the response values for the 3 observations in the training data most similar to the observation being predicted. This often results in highly variable predicted values because we are basing the prediction (in this case, an average) on a very small subset of the training data. As k gets bigger, we base our predictions on an average of a larger subset of the training data, which naturally reduces the variance in our predicted values (remember this for later, averaging reduces variance!). Figure 2.10 illustrates this point. Smaller k values (e.g., 2, 5, or 10) lead to high variance (but lower bias) and larger values (e.g., 150) lead to high bias (but lower variance). The optimal k value might exist somewhere between 20–50, but how do we know which value of k to use?

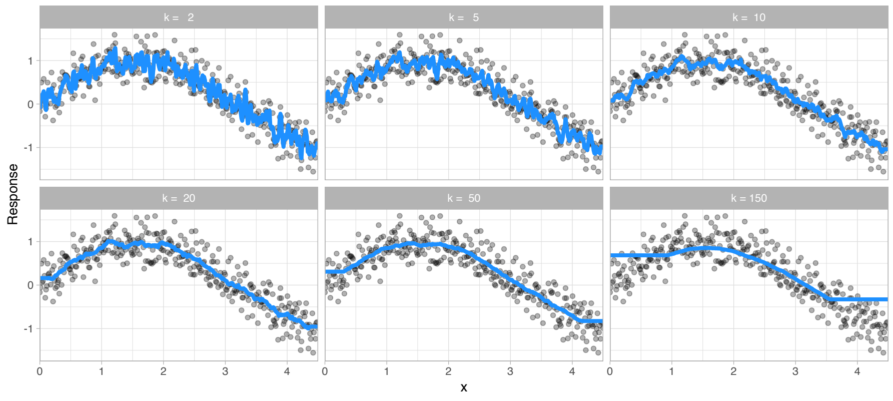


FIGURE 2.10: k-nearest neighbor model with differing values for k.

[6]This phrase comes from Brad Efron's comments in Breiman et al. (2001)

[7]At least in the ordinary sense. You could think of polynomial regression as having a single hyperparamer: the degree of the polynomial.

One way to perform hyperparameter tuning is to fiddle with hyperparameters manually until you find a great combination of hyperparameter values that result in high predictive accuracy (as measured using k-fold CV, for instance). However, this can be very tedious work depending on the number of hyperparameters. An alternative approach is to perform a *grid search*. A grid search is an automated approach to searching across many combinations of hyperparameter values.

For our k-nearest neighbor example, a grid search would predefine a candidate set of values for k (e.g., $k = 1, 2, \ldots, j$) and perform a resampling method (e.g., k-fold CV) to estimate which k value generalizes the best to unseen data. Figure 2.11 illustrates the results from a grid search to assess $k = 2, 12, 14, \ldots, 150$ using repeated 10-fold CV. The error rate displayed represents the average error for each value of k across all the repeated CV folds. On average, $k = 46$ was the optimal hyperparameter value to minimize error (in this case, RMSE which is discussed in Section 2.6)) on unseen data.

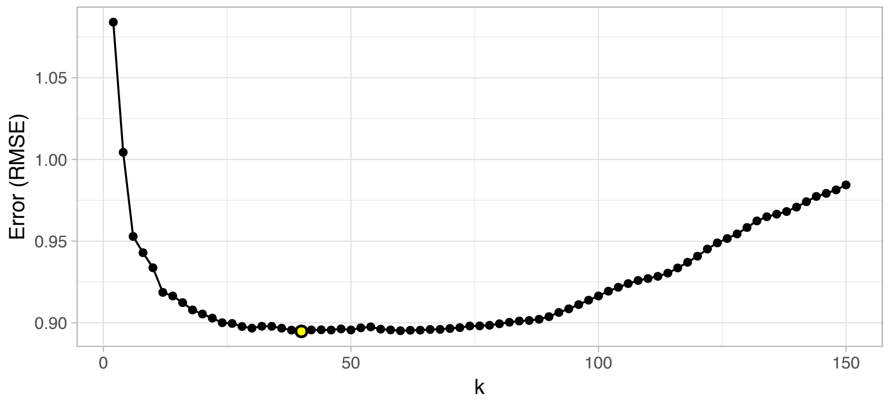


FIGURE 2.11: Results from a grid search for a k-nearest neighbor model assessing values for k ranging from 2-150. We see high error values due to high model variance when k is small and we also see high errors values due to high model bias when k is large. The optimal model is found at k = 46.

Throughout this book you'll be exposed to different approaches to performing grid searches. In the above example, we used a *full cartesian grid search*, which assesses every hyperparameter value manually defined. However, as models get more complex and offer more hyperparameters, this approach can become computationally burdensome and requires you to define the optimal hyperparameter grid settings to explore. Additional approaches we'll illustrate include *random grid searches* (Bergstra and Bengio, 2012) which explores randomly selected hyperparameter values from a range of possible values, *early stopping* which allows you to stop a grid search once reduction in the error stops marginally improving, *adaptive resampling* via futility analysis (Kuhn,

2014) which adaptively resamples candidate hyperparameter values based on approximately optimal performance, and more.

2.6 Model evaluation

Historically, the performance of statistical models was largely based on goodness-of-fit tests and assessment of residuals. Unfortunately, misleading conclusions may follow from predictive models that pass these kinds of assessments (Breiman et al., 2001). Today, it has become widely accepted that a more sound approach to assessing model performance is to assess the predictive accuracy via *loss functions*. Loss functions are metrics that compare the predicted values to the actual value (the outpuf of a loss function is often referred to as the *error* or pseudo *residual*). When performing resampling methods, we assess the predicted values for a validation set compared to the actual target value. For example, in regression, one way to measure error is to take the difference between the actual and predicted value for a given observation (this is the usual definition of a residual in ordinary linear regression). The overall validation error of the model is computed by aggregating the errors across the entire validation data set.

There are many loss functions to choose when assessing the performance of a predictive model; each providing a unique understanding of the predictive accuracy and differing between regression and classification models. Furthermore, the way a loss function is computed will tend to emphasize certain types of errors over others and can lead to drastic differences in how we interpret the "optimal model". Its important to consider the problem context when identifying the preferred performance metric to use. And when comparing multiple models, we need to compare them across the same metric.

2.6.1 Regression models

- **MSE**: Mean squared error is the average of the squared error ($MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2)$[8]. The squared component results in larger errors having larger penalties. This (along with RMSE) is the most common error metric to use. **Objective: minimize**
- **RMSE**: Root mean squared error. This simply takes the square root of the MSE metric ($RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$) so that your error is in

[8]This deviates slightly from the usual definition of MSE in ordinary linear regression, where we divide by $n - p$ (to adjust for bias) as opposed to n.

the same units as your response variable. If your response variable units are dollars, the units of MSE are dollars-squared, but the RMSE will be in dollars. **Objective: minimize**

- **Deviance**: Short for mean residual deviance. In essence, it provides a degree to which a model explains the variation in a set of data when using maximum likelihood estimation. Essentially this computes a saturated model (i.e. fully featured model) to an unsaturated model (i.e. intercept only or average). If the response variable distribution is Gaussian, then it will be approximately equal to MSE. When not, it usually gives a more useful estimate of error. Deviance is often used with classification models. [9] **Objective: minimize**

- **MAE**: Mean absolute error. Similar to MSE but rather than squaring, it just takes the mean absolute difference between the actual and predicted values ($MAE = \frac{1}{n}\sum_{i=1}^{n}(|y_i - \hat{y}_i|)$). This results in less emphasis on larger errors than MSE. **Objective: minimize**

- **RMSLE**: Root mean squared logarithmic error. Similiar to RMSE but it performs a `log()` on the actual and predicted values prior to computing the difference ($RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(log(y_i + 1) - log(\hat{y}_i + 1))^2}$). When your response variable has a wide range of values, large repsonse values with large errors can dominate the MSE/RMSE metric. RMSLE minimizes this impact so that small response values with large errors can have just as meaningful of an impact as large response values with large errors. **Objective: minimize**

- R^2: This is a popular metric that represents the proportion of the variance in the dependent variable that is predictable from the independent variable(s). Unfortunately, it has several limitations. For example, two models built from two different data sets could have the exact same RMSE but if one has less variability in the response variable then it would have a lower R^2 than the other. You should not place too much emphasis on this metric. **Objective: maximize**

Most models we assess in this book will report most, if not all, of these metrics. We will emphasize MSE and RMSE but its important to realize that certain situations warrant emphasis on some metrics more than others.

2.6.2 Classification models

- **Misclassification**: This is the overall error. For example, say you are predicting 3 classes (*high*, *medium*, *low*) and each class has 25, 30, 35

[9] See this StackExchange thread (`http://bit.ly/what-is-deviance`) for a good overview of deviance for different models and in the context of regression versus classification.

observations respectively (90 observations total). If you misclassify 3 observations of class *high*, 6 of class *medium*, and 4 of class *low*, then you misclassified 13 out of 90 observations resulting in a 14% misclassification rate. **Objective: minimize**

- **Mean per class error**: This is the average error rate for each class. For the above example, this would be the mean of $\frac{3}{25}, \frac{6}{30}, \frac{4}{35}$, which is 12%. If your classes are balanced this will be identical to misclassification. **Objective: minimize**
- **MSE**: Mean squared error. Computes the distance from 1.0 to the probability suggested. So, say we have three classes, A, B, and C, and your model predicts a probability of 0.91 for A, 0.07 for B, and 0.02 for C. If the correct answer was A the $MSE = 0.09^2 = 0.0081$, if it is B $MSE = 0.93^2 = 0.8649$, if it is C $MSE = 0.98^2 = 0.9604$. The squared component results in large differences in probabilities for the true class having larger penalties. **Objective: minimize**
- **Cross-entropy (aka Log Loss or Deviance)**: Similar to MSE but it incorporates a log of the predicted probability multiplied by the true class. Consequently, this metric disproportionately punishes predictions where we predict a small probability for the true class, which is another way of saying having high confidence in the wrong answer is really bad. **Objective: minimize**
- **Gini index**: Mainly used with tree-based methods and commonly referred to as a measure of *purity* where a small value indicates that a node contains predominantly observations from a single class. **Objective: minimize**

When applying classification models, we often use a *confusion matrix* to evaluate certain performance measures. A confusion matrix is simply a matrix that compares actual categorical levels (or events) to the predicted categorical levels. When we predict the right level, we refer to this as a *true positive*. However, if we predict a level or event that did not happen this is called a *false positive* (i.e. we predicted a customer would redeem a coupon and they did not). Alternatively, when we do not predict a level or event and it does happen that this is called a *false negative* (i.e. a customer that we did not predict to redeem a coupon does).

We can extract different levels of performance for binary classifiers. For example, given the classification (or confusion) matrix illustrated in Figure 2.13 we can assess the following:

- **Accuracy**: Overall, how often is the classifier correct? Opposite of misclassification above. Example: $\frac{TP+TN}{total} = \frac{100+50}{165} = 0.91$. **Objective: maximize**

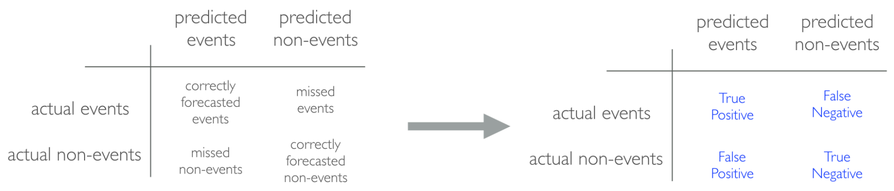
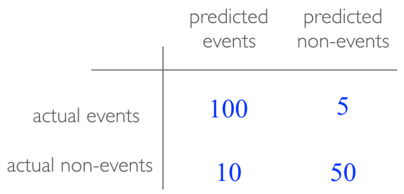


FIGURE 2.12: Confusion matrix and relationships to terms such as true-positive and false-negative.

- **Precision**: How accurately does the classifier predict events? This metric is concerned with maximizing the true positives to false positive ratio. In other words, for the number of predictions that we made, how many were correct? Example: $\frac{TP}{TP+FP} = \frac{100}{100+10} = 0.91$. **Objective: maximize**
- **Sensitivity (aka recall)**: How accurately does the classifier classify actual events? This metric is concerned with maximizing the true positives to false negatives ratio. In other words, for the events that occurred, how many did we predict? Example: $\frac{TP}{TP+FN} = \frac{100}{100+5} = 0.95$. **Objective: maximize**
- **Specificity**: How accurately does the classifier classify actual non-events? Example: $\frac{TN}{TN+FP} = \frac{50}{50+10} = 0.83$. **Objective: maximize**



	predicted events	predicted non-events
actual events	100	5
actual non-events	10	50

FIGURE 2.13: Example confusion matrix.

- **AUC**: Area under the curve. A good binary classifier will have high precision and sensitivity. This means the classifier does well when it predicts an event will and will not occur, which minimizes false positives and false negatives. To capture this balance, we often use a ROC curve that plots the false positive rate along the x-axis and the true positive rate along the y-axis. A line that is diagonal from the lower left corner to the upper right corner represents a random guess. The higher the line is in the upper left-hand corner, the better. AUC computes the area under this curve. **Objective: maximize**

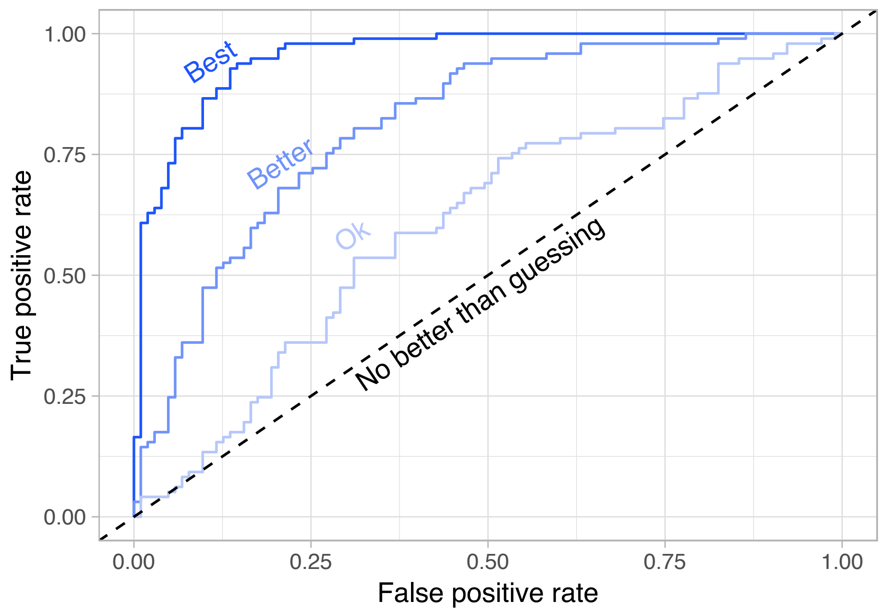


FIGURE 2.14: ROC curve.

2.7 Putting the processes together

To illustrate how this process works together via R code, let's do a simple assessment on the `ames` housing data. First, we perform stratified sampling as illustrated in Section 2.2.2 to break our data into training vs. test data while ensuring we have consistent distributions between the training and test sets.

```
# stratified sampling with the rsample package
set.seed(123)
split  <- initial_split(ames, prop = 0.7, strata = "Sale_Price")
ames_train  <- training(split)
ames_test   <- testing(split)
```

Next, we're going to apply a k-nearest neighbor regressor to our data. To do so, we'll use **caret**, which is a meta-engine to simplify the resampling, grid search, and model application processes. The following defines:

1. **Resampling method**: we use 10-fold CV repeated 5 times.

2. **Grid search**: we specify the hyperparameter values to assess ($k = 2, 4, 6, \dots, 25$).
3. **Model training & Validation**: we train a k-nearest neighbor (`method = "knn"`) model using our pre-specified resampling procedure (`trControl = cv`), grid search (`tuneGrid = hyper_grid`), and preferred loss function (`metric = "RMSE"`).

This grid search takes approximately 3.5 minutes

```r
# create a resampling method
cv <- trainControl(
  method = "repeatedcv",
  number = 10,
  repeats = 5
  )

# create a hyperparameter grid search
hyper_grid <- expand.grid(k = seq(2, 25, by = 1))

# fit knn model and perform grid search
knn_fit <- train(
  Sale_Price ~ .,
  data = ames_train,
  method = "knn",
  trControl = cv,
  tuneGrid = hyper_grid,
  metric = "RMSE"
  )
```

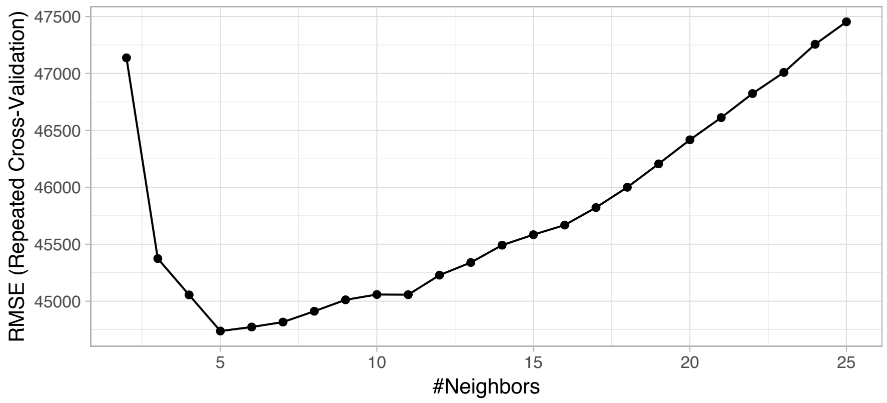
Looking at our results we see that the best model coincided with $k = 5$, which resulted in an RMSE of 44738. This implies that, on average, our model mispredicts the expected sale price of a home by $44,738. Figure 2.7 illustrates the cross-validated error rate across the spectrum of hyperparameter values that we specified.

```r
# print model results
knn_fit
## k-Nearest Neighbors
##
```

```
## 2054 samples
##   80 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 1849, 1848, 1848, 1849, 1849, 1847, ...
## Resampling results across tuning parameters:
##
##   k   RMSE   Rsquared  MAE
##    2  47138  0.6592    30432
##    3  45374  0.6806    29403
##    4  45055  0.6847    29194
##    5  44738  0.6898    28966
##    6  44773  0.6908    28926
##    7  44816  0.6918    28970
##    8  44911  0.6921    29022
##    9  45012  0.6929    29047
##   10  45058  0.6945    28972
##   11  45057  0.6967    28908
##   12  45229  0.6962    28952
##   13  45339  0.6961    29031
##   14  45492  0.6958    29124
##   15  45584  0.6961    29188
##   16  45668  0.6964    29277
##   17  45822  0.6959    29410
##   18  46000  0.6943    29543
##   19  46206  0.6927    29722
##   20  46417  0.6911    29845
##   21  46612  0.6895    29955
##   22  46824  0.6877    30120
##   23  47009  0.6863    30257
##   24  47256  0.6837    30413
##   25  47454  0.6819    30555
##
## RMSE was used to select the optimal model using
##  the smallest value.
## The final value used for the model was k = 5.

# plot cross validation results
ggplot(knn_fit)
```

\begin{figure}



{

}

\caption{Results from a grid search for a k-nearest neighbor model on the Ames housing data assessing values for *k* ranging from 2-25.} \end{figure}

The question remains: "Is this the best predictive model we can find?" We may have identified the optimal *k*-nearest neighbor model for our given data set, but this doesn't mean we've found the best possible overall model. Nor have we considered potential feature and target engineering options. The remainder of this book will walk you through the journey of identifying alternative solutions and, hopefully, a much more optimal model.

Bibliography

Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305.

Breiman, L. et al. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical science*, 16(3):199–231.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357.

Cireşan, D., Meier, U., and Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. *arXiv preprint arXiv:1202.2745*.

Davison, A. C., Hinkley, D. V., et al. (1997). *Bootstrap methods and their application*, volume 1. Cambridge university press.

De Cock, D. (2011). Ames, iowa: Alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, 19(3).

Efron, B. (1983). Estimating the error rate of a prediction rule: improvement on cross-validation. *Journal of the American statistical association*, 78(382):316–331.

Efron, B. and Hastie, T. (2016). *Computer age statistical inference*, volume 5. Cambridge University Press.

Efron, B. and Tibshirani, R. (1986). Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical science*, pages 54–75.

Efron, B. and Tibshirani, R. (1997). Improvements on cross-validation: the 632+ bootstrap method. *Journal of the American Statistical Association*, 92(438):548–560.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics New York, NY, USA:.

Goodfellow, I., Bengio, Y., Courville, A., and Bengio, Y. (2016). *Deep learning*, volume 1. MIT press Cambridge.

Hawkins, D. M., Basak, S. C., and Mills, D. (2003). Assessing model fit by cross-validation. *Journal of chemical information and computer sciences*, 43(2):579–586.

Hyndman, R. J. and Athanasopoulos, G. (2018). *Forecasting: principles and practice*. OTexts.

Irizarry, R. A. (2018). *dslabs: Data Science Labs*. R package version 0.5.2.

Kim, J.-H. (2009). Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. *Computational statistics & data analysis*, 53(11):3735–3745.

Kuhn, M. (2014). Futility analysis in the cross-validation of machine learning models. *arXiv preprint arXiv:1405.6974*.

Kuhn, M. (2017a). *AmesHousing: The Ames Iowa Housing Data*. R package version 0.0.3.

Kuhn, M. (2017b). The r formula method: The bad parts.

Kuhn, M. (2019). Applied machine learning. RStudio Conference.

Kuhn, M. and Johnson, K. (2013). *Applied predictive modeling*, volume 26. Springer.

Kuhn, M. and Wickham, H. (2017). *rsample: General Resampling Infrastructure*. R package version 0.0.2.

LeCun, Y., Boser, B. E., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. E., and Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*, pages 396–404.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Molinaro, A. M., Simon, R., and Pfeiffer, R. M. (2005). Prediction error estimation: a comparison of resampling methods. *Bioinformatics*, 21(15):3301–3307.

Wickham, H. (2014). *Advanced R*. Chapman and Hall/CRC.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*. O'Reilly Media, Inc.

Wolpert, D. H. (1996). The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390.

Index