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).

```
# 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     2019-06-26
#>
#> - 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)
#>  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)
#>  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)
#>
#> [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.

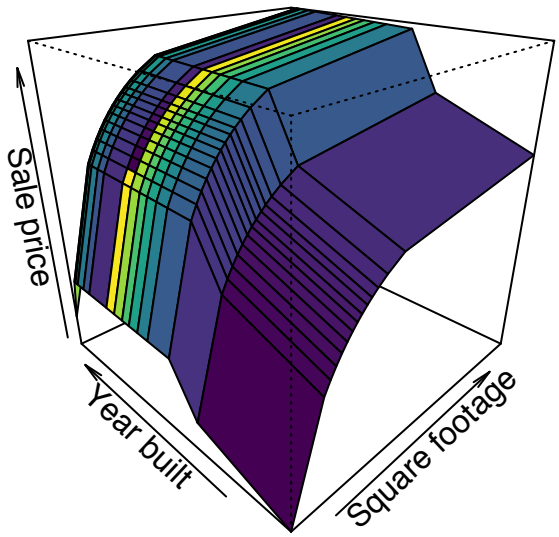

FIGURE 1.1: Average home sales price as a function of year built and total square footage.

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.

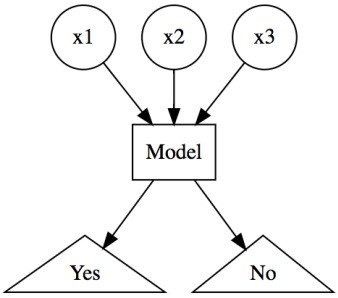

FIGURE 1.2: Classification problem modeling 'Yes'/'No' response based on three features.

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 2-3.

Chapters 4-?? 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`

```
# 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 preprocessing the feature and target variables, minimizing *data leakage* (Section 3.8.2), 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.

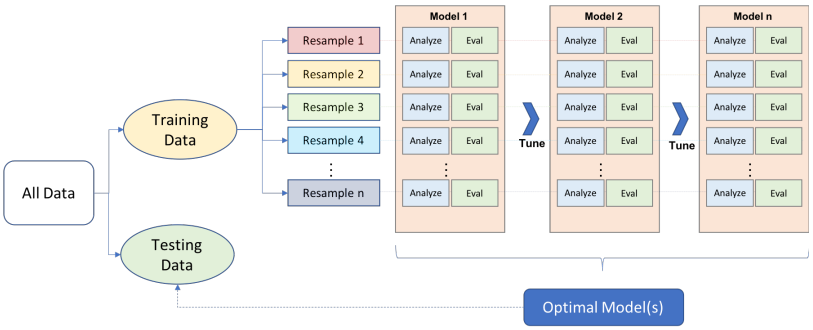

FIGURE 2.1: General predictive machine learning process.

```
# 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.

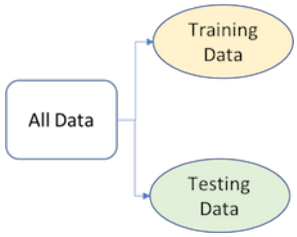

FIGURE 2.2: Splitting data into training and test sets.

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'll 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.

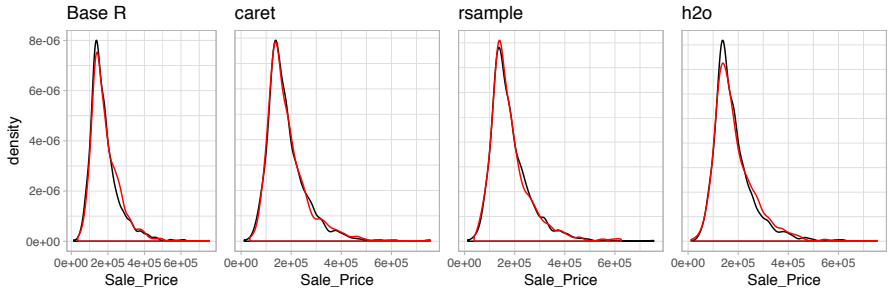

FIGURE 2.3: Training (black) vs. test (red) response distribution.

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 hyperparameter 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:

```
# 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):

```
# 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.

```
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:

```
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 2.5).

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

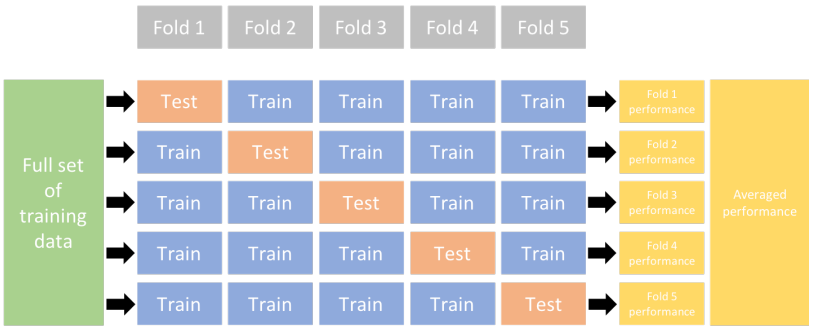

FIGURE 2.4: Illustration of the k-fold cross validation process.

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 $k = n$).

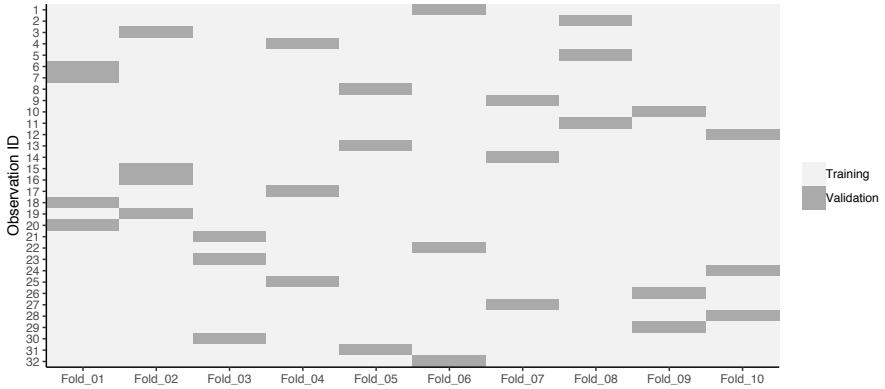

FIGURE 2.5: 10-fold cross validation on 32 observations. Each observation is used once for validation and nine times for training.

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.

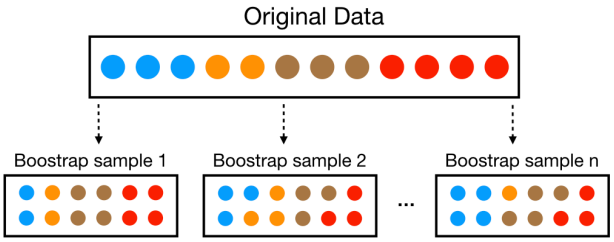

FIGURE 2.6: Illustration of the bootstrapping process.

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
```

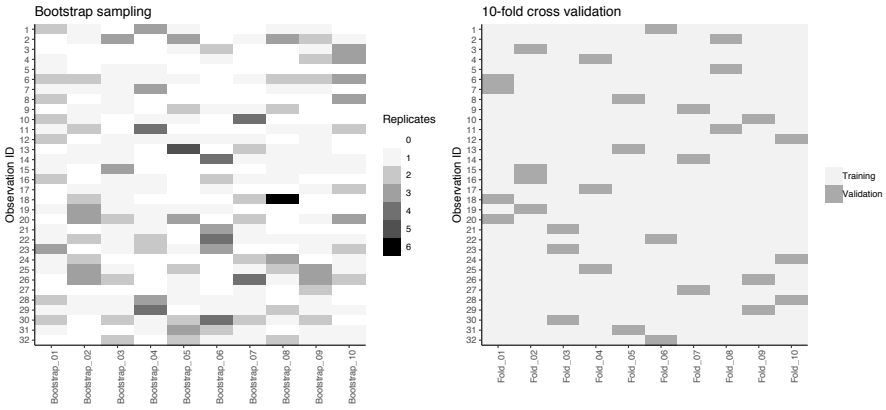

FIGURE 2.7: Bootstrap sampling (left) versus 10-fold cross validation (right) on 32 observations. For bootstrap sampling, the observations that have zero replications (white) are the out-of-bag observations used for validation.

```
##    <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

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

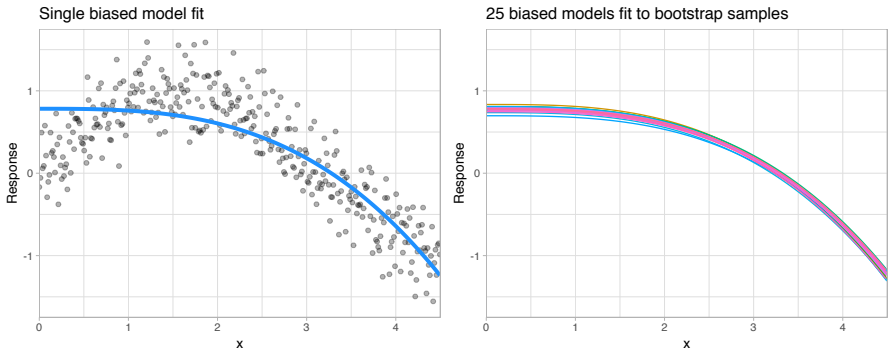

FIGURE 2.8: A biased polynomial model fit to a single data set does not capture the underlying non-linear, non-monotonic data structure (left). Models fit to 25 bootstrapped replicates of the data are underterred by the noise and generates similar, yet still biased, predictions (right).

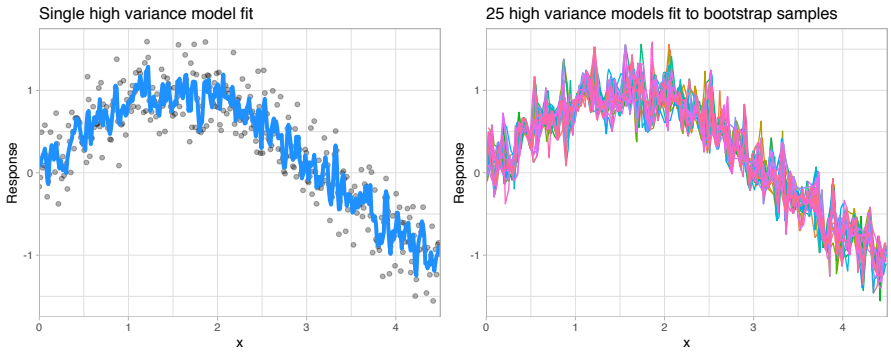

FIGURE 2.9: A high variance k-nearest neighbor model fit to a single data set captures the underlying non-linear, non-monotonic data structure well but also overfits to individual data points (left). Models fit to 25 bootstrapped replicates of the data are deterred by the noise and generate highly variable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

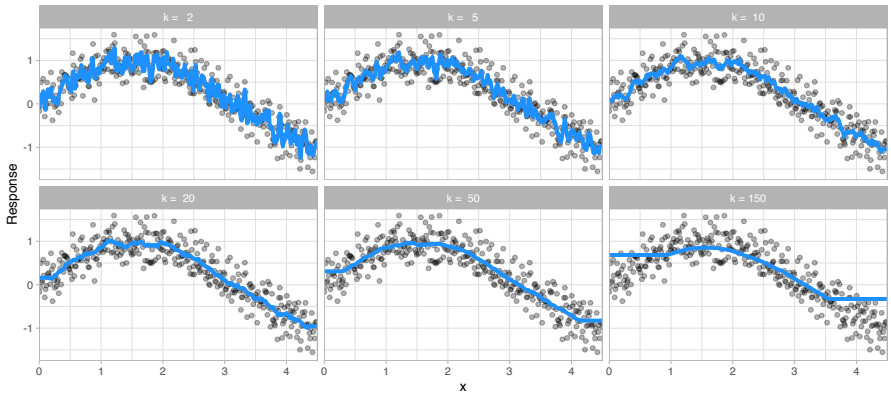

FIGURE 2.10: k-nearest neighbor model with differing values for k.

[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., $k = 1, 2, \dots, j$) 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 $k = 2, 12, 14, \dots, 150$ 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.

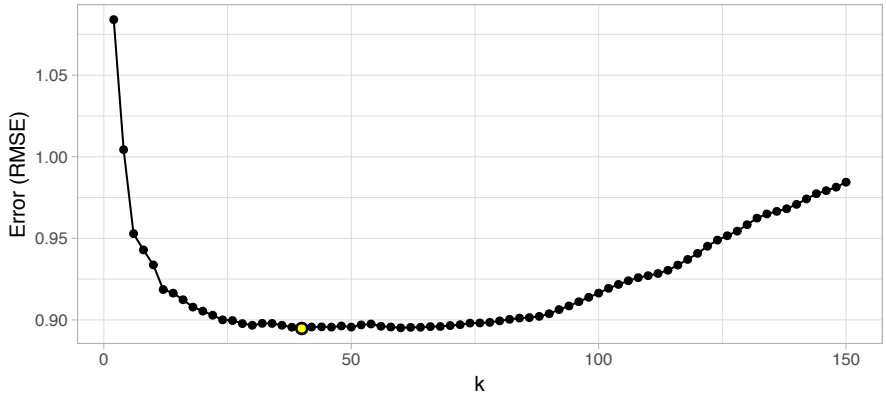

FIGURE 2.11: Results from a grid search for a k-nearest neighbor model assessing values for k ranging from 2-150. We see high error values due to high model variance when k is small and we also see high errors values due to high model bias when k is large. The optimal model is found at k = 46.

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**

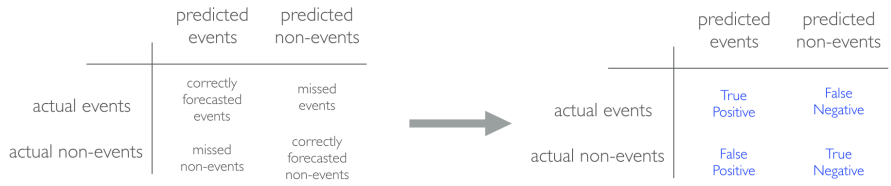

FIGURE 2.12: Confusion matrix and relationships to terms such as true-positive and false-negative.

- **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**

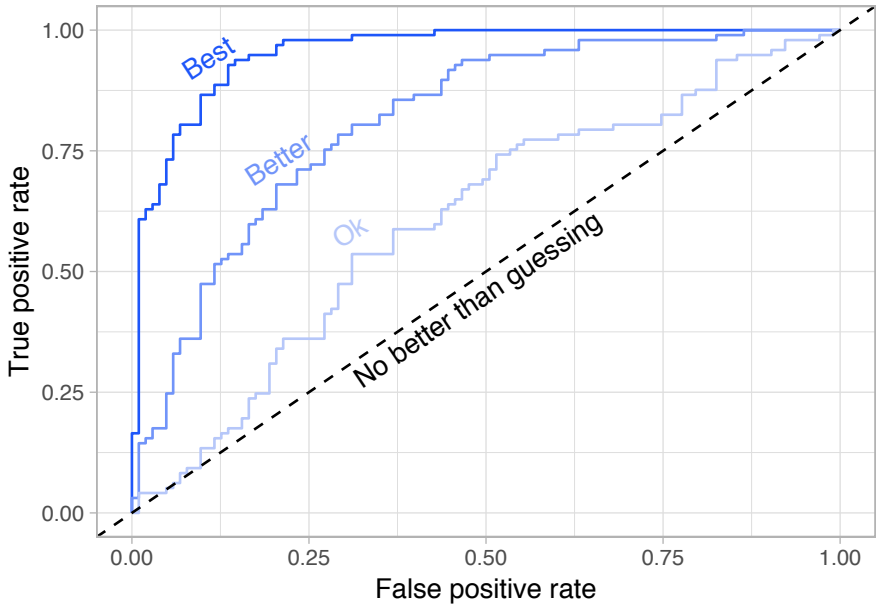

FIGURE 2.14: ROC curve.

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 ($k = 2, 4, 6, \ldots, 25$).
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

```
# 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 2.15 illustrates the cross-validated error rate across the spectrum of hyperparameter values that we specified.

```
# 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)
```

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

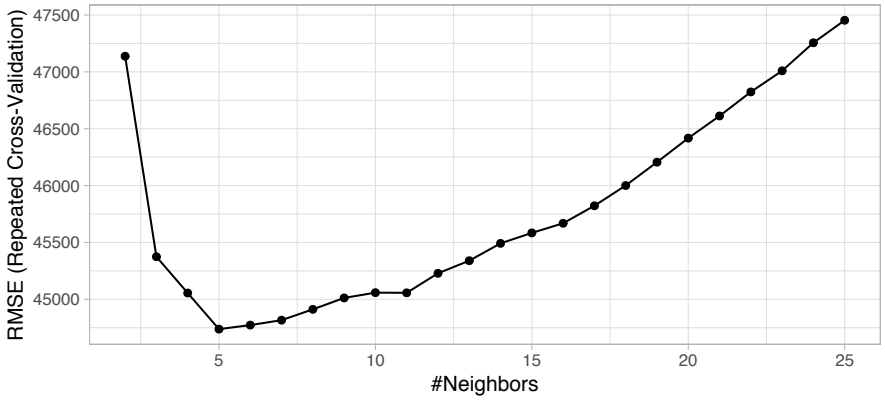

FIGURE 2.15: Results from a grid search for a k-nearest neighbor model on the Ames housing data assessing values for k ranging from 2-25.

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.

3

Feature & Target Engineering

Data pre-processing and engineering techniques generally refer to the addition, deletion, or transformation of data. The time spent on identifying data engineering needs can be significant and requires you to spend substantial time understanding your data...or as Leo Breiman said "live with your data before you plunge into modeling" (Breiman et al., 2001, p. 201). Although this book primarily focuses on applying machine learning algorithms, feature engineering can make or break an algorithm's predictive ability and deserves your continued focus and education.

We will not cover all the potential ways of implementing feature engineering; however, we'll cover several fundamental pre-processing tasks that can potentially significantly improve modeling performance. Moreover, different models have different sensitivities to the type of target and feature values in the model and we will try to highlight some of these concerns. For more indepth coverage of feature engineering, please refer to Kuhn and Johnson (2019) and Zheng and Casari (2018).

3.1 Prerequisites

This chapter leverages the following packages:

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

# Feature engineering packages
library(caret)    # for various feature engineering tasks
library(recipes)  # for various feature engineering tasks
```

We'll also continue working with the `ames_train` data set created in Section 2.7:

3.2 Target engineering

Although not always a requirement, transforming the response variable can lead to predictive improvement, especially with parametric models (where require that certain assumptions about the model be met). For instance, ordinary linear regression models assume that the prediction errors (and hence the response) are normally distributed. This is usually fine, except when the prediction target has heavy tails (i.e., *outliers*) or is skewed in one direction or the other. In these cases, the normality assumption likely does not hold. For example, as we saw in the data splitting section (2.2), the response variable for the Ames housing data (`Sale_Price`) is right (or positively) skewed as illustrated in Figure 2.3 (ranging from \$12,789 to \$755,000). A simple linear model, say Sale_Price $= \beta_0 + \beta_1$ Year_Built $+ \epsilon$, often assumes the error term ϵ (and hence `Sale_Price`) is normally distributed; fortunately, a simple log (or similar) transformation of the response can often help alleviate this concern as Figure 3.1 illustrates.

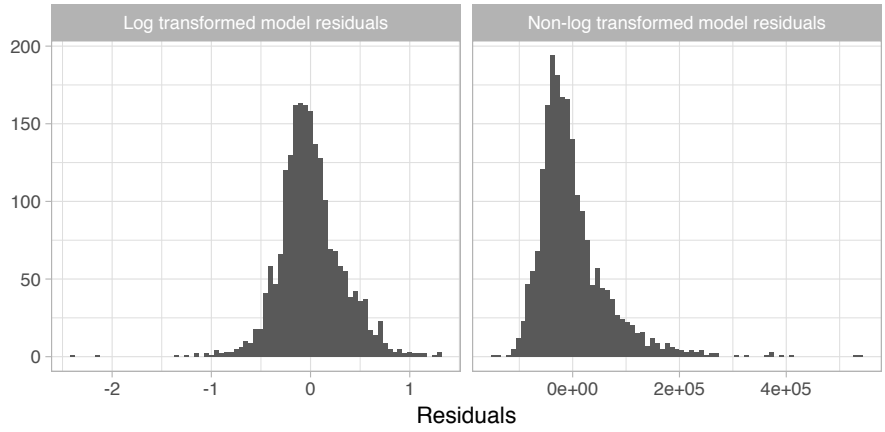

FIGURE 3.1: Transforming the response variable to minimize skewness can resolve concerns with non-normally distributed errors.

Furthermore, using a log (or other) transformation to minimize the response skewness can be used for shaping the business problem as well. For example,

in the House Prices: Advanced Regression Techniques Kaggle competition[1], which used the Ames housing data, the competition focused on using a log transformed Sale Price response because "...taking logs means that errors in predicting expensive houses and cheap houses will affect the result equally." This would be an alternative to using the root mean squared logarithmic error (RMSLE) loss function as discussed in Section 2.6.

There are two main approaches to help correct for postively skewed target variables:

Option 1: normalize with a log transformation. This will transform most right skewed distributions to be approximately normal. One way to do this is to simply log transform the training and test set in a manual, single step manner similar to:

```
transformed_response <- log(ames_train$Sale_Price)
```

However, we should think of the pre-processing as creating a blueprint to be re-applied strategically. For this, you can use the **recipe** package or something similar (e.g., `caret::preProcess()`). This will not return the actual log transformed values but, rather, a blueprint to be applied later.

```
# log transformation
ames_recipe <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_log(all_outcomes())

ames_recipe
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Log transformation on all_outcomes()
```

If your reponse has negative values or zeros then a log transformation will produce `NaN`s and `-Inf`s, respectively (you cannot take the logarithm of a

[1] https://www.kaggle.com/c/house-prices-advanced-regression-techniques

negative number). If the nonpositive response values values are small (say between -0.99 and 0) then you can apply a small offset such as in `log1p()` which adds 1 to the value prior to applying a log transformation (you can do the same within `step_log()` by using the `offset` argument). If your data consists of values ≤ -1, use the Yeo-Johnson transformation mentioned next.

```
log(-0.5)
## [1] NaN
log1p(-0.5)
## [1] -0.6931
```

Option 2: use a *Box Cox transformation*. A Box Cox transformation is more flexible than (but also includes as a special case) the log transformation and will find an appropriate transformation from a family of power transforms that will transform the variable as close as possible to a normal distribution (Box and Cox, 1964; Carroll and Ruppert, 1981). At the core of the Box Cox transformation is an exponent, lambda (λ), which varies from -5 to 5. All values of λ are considered and the optimal value for the given data is estimated from the training data; The "optimal value" is the one which results in the best transofrmation to an approximate normal distribution. The transformation of the response Y has the form:

$$y(\lambda) = \begin{cases} \frac{Y^{\lambda}-1}{\lambda}, & \text{if } \lambda \neq 0 \\ \log(Y), & \text{if } \lambda = 0. \end{cases} \tag{3.1}$$

Be sure to compute the `lambda` on the training set and apply that same `lambda` to both the training and test set to minimize *data leakage*. The **recipes** package automates this process for you.

If your response has negative values, the Yeo-Johnson transformation is very similar to the Box-Cox but does not require the input variables to be strictly positive. To apply, use `step_YeoJohnson()`.

Figure 3.2 illustrates that the log transformation and Box Cox transformation both do about equally well in transforming `Sale_Price` to look more normally distributed.

Note that when you model with a transformed response variable, your predictions will also be on the transformed scale. You will likely want to re-transform your predicted values back to their normal scale so that decision-makers can more easily interpret the results. This is illustrated in the following code chunk:

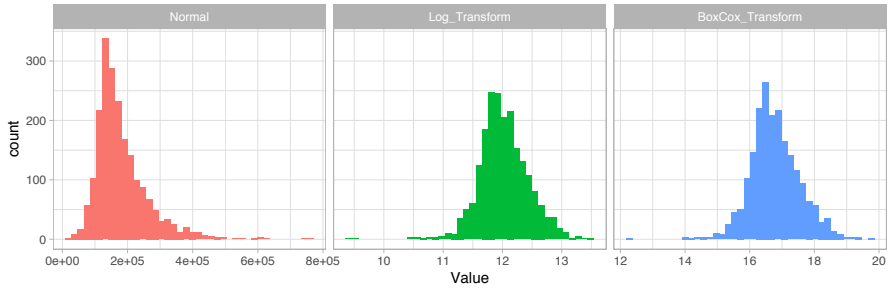

FIGURE 3.2: Response variable transformations.

```
# log transform a value
y <- log(10)

# re-transforming the log-transformed value
exp(y)
## [1] 10

# Box Cox transform a value
y <- forecast::BoxCox(10, lambda)

# Inverse Box Cox function
inv_box_cox <- function(x, lambda) {
  # for Box-Cox, lambda = 0 is equivalent to log transform
  if (lambda == 0) exp(x) else (lambda*x + 1)^(1/lambda)
}

# re-transforming the Box Cox-transformed value
inv_box_cox(y, lambda)
## [1] 10
## attr(,"lambda")
## [1] 0.0526
```

3.3 Dealing with missingness

Data quality is an important issue for any project involving analyzing data. Data quality issues deserve an entire book in their own right, and a good

reference is the The Quartz guide to bad data.[2] One of the most common data quality concerns you will run into is missing values.

Data can be missing for many different reasons; however, these reasons are usually lumped into two categories: *informative missingess* (Kuhn and Johnson, 2013) and *missingness at random* (Little and Rubin, 2014). Informative missinginess implies a structural cause for the missing value that can provide insight in its own right; whether this be definciencies in how the data was collected or abnormalities in the oberservational environment. Missingness at random implies that missing values occur independent of the data collection process[3].

The category that drives missing values will determine how you handle them. For example, we may give values that are driven by informative missingness their own category (e.g., `"None"`) as their unique value may affect predictive performance. Whereas values that are missing at random may deserve deletion[4] or imputation.

Furthermore, different machine learning models handle missingness differently. Most algorithms cannot handle missingness (e.g., generalized linear models and their cousins, neural networks, and support vector machines) and, therefore, require them to be dealt with before hand. A few models (mainly tree-based), have built-in procedures to deal with missing values. However, since the modeling process involves comparing and contrasting multiple models to identify the optimal one, you will want to handle missing values prior to applying any models so that your algorithms are based on the same data quality assumptions.

3.3.1 Visualizing missing values

It is important to understand the distribution of missing values (i.e., `NA`) in any data set. So far, we have been using a pre-processed version of the Ames housing data set (via the `AmesHousing::make_ames()` function). However, if we use the raw Ames housing data (via `AmesHousing::ames_raw`), there are actually 13,997 missing values—there is at least one missing values in each row of the original data!

[2] `https://github.com/Quartz/bad-data-guide`

[3] Little and Rubin (2014) discuss two different kinds of missingness at random; however, we combine them for simplicity as their nuanced differences are distinguished between the two in practice.

[4] If your data set is large, deleting missing observations that have missing values at random rarely impacts predictive performance. However, as your data sets get smaller, preserving observations is critical and alternative solutions should be explored.

```
sum(is.na(AmesHousing::ames_raw))
## [1] 13997
```

It is important to understand the distribution of missing values in a data set in order to determine the best approach for pre-processing. Heat maps are an efficient way to visualize the distribution of missing values for small- to medium-sized data sets. The code `is.na(<data-frame-name>)` will return a matrix of the same dimension as the given data frame, but each cell will contain either `TRUE` (if the corresponding value is missing) or `FALSE` (if the corresponding value is not missing). To construct such a plot, we can use R's built-in `heatmap()` or `image()` functions, or **ggplot2**'s `geom_raster()` function, among others; Figure 3.3 illustrates `geom_raster()`. This allows us to easily see where the majority of missing values occur (i.e., in the variables `Alley`, `Fireplace Qual`, `Pool QC`, `Fence`, and `Misc Feature`). Due to their high frequency of missingness, these variables would likely need to be removed prior to statiscial analysis, or imputed. We can also spot obvious patterns of missingness. For example, missing values appear to occur within the same observations across all garage variables.

```
AmesHousing::ames_raw %>%
  is.na() %>%
  reshape2::melt() %>%
  ggplot(aes(Var2, Var1, fill=value)) +
    geom_raster() +
    coord_flip() +
    scale_y_continuous(NULL, expand = c(0, 0)) +
    scale_fill_grey(name = "", labels = c("Present", "Missing")) +
    xlab("Observation") +
    theme(axis.text.y  = element_text(size = 4))
```

Digging a little deeper into these variables, we might notice that `Garage_Cars` and `Garage_Area` contain the value `0` whenever the other `Garage_xx` variables have missing values (i.e. a value of `NA`). This might be because they did not have a way to identify houses with no garages when the data were originally collected, and therefore, all houses with no garage were identified by including nothing. Since this missingness is informative, it would be appropriate to impute `NA` with a new category level (e.g., `"None"`) for these garage variables. Circumstances like this tend to only become apparent upon careful descriptive and visual examination of the data!

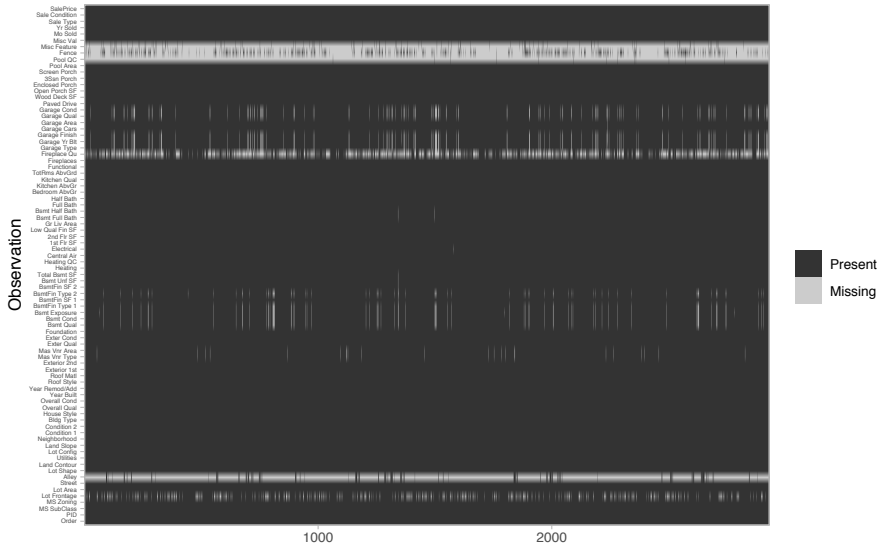

FIGURE 3.3: Heat map of missing values in the raw Ames housing data.

```
AmesHousing::ames_raw %>%
  filter(is.na(`Garage Type`)) %>%
  select(`Garage Type`, `Garage Cars`, `Garage Area`)
## # A tibble: 157 x 3
##    `Garage Type` `Garage Cars` `Garage Area`
##    <chr>                 <int>         <int>
##  1 <NA>                      0             0
##  2 <NA>                      0             0
##  3 <NA>                      0             0
##  4 <NA>                      0             0
##  5 <NA>                      0             0
##  6 <NA>                      0             0
##  7 <NA>                      0             0
##  8 <NA>                      0             0
##  9 <NA>                      0             0
## 10 <NA>                      0             0
## # ... with 147 more rows
```

The `vis_miss()` function in R package `visdat` (Tierney, 2017) also allows for easy visualization of missing data patterns (with sorting and clustering options). We illustrate this functionality below using the raw Ames housing data (Figure 3.4). The columns of the heat map represent the 82 variables of the raw data and the rows represent the observations. Missing values (i.e.,

`NA`) are inidcated via a black cell. The variables and `NA` patterns have been clustered by rows (i.e., `cluster = TRUE`).

```
vis_miss(AmesHousing::ames_raw, cluster = TRUE)
```

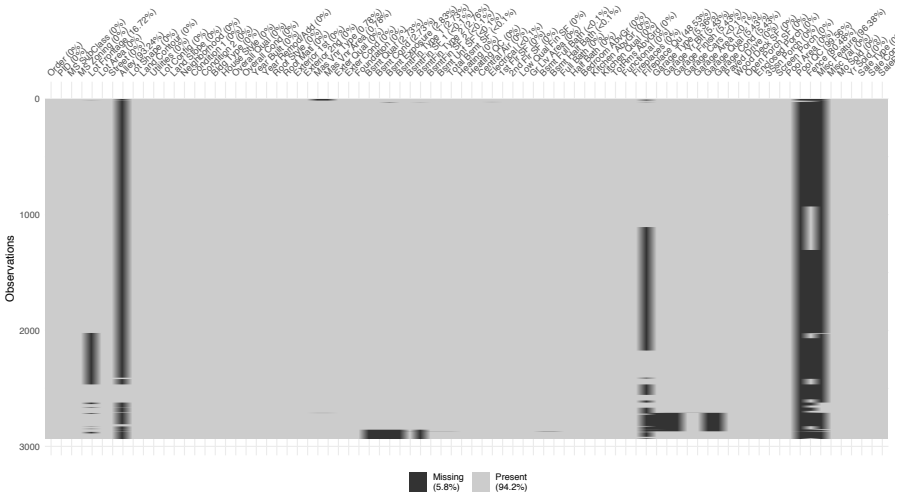

FIGURE 3.4: Visualizing missing patterns in the raw Ames housing data.

Data can be missing for different reasons. Perhaps the values were never recoded (or lost in translation), or it was recorded in error (a common feature of data enetered by hand). Regardless, it is important to identify and attempt to understand how missing values are distributed across a data set as it can provide insight into how to deal with these observations.

3.3.2 Imputation

Imputation is the process of replacing a missing value with a substituted, "best guess" value. Imputation should be one of the first feature engineering steps yo take as it will effect any downstream pre-processing[5].

3.3.2.1 Estimated statistic

An elementary approach to imputing missing values for a feature is to compute descriptive statistics such as the mean, median, or mode (for categorical)

[5]For example, standardizing numeric features will include the imputed numeric values in the calculation and one-hot encoding will include the imputated categorical value.

and use that value to replace NAs. Although computationally efficient, this approach does not consider any other attributes for a given observation when imputing (e.g., a female patient that is 63 inches tall may have her weight imputed as 175 lbs since that is the average weight across all observations which contains 65% males that average a height of 70 inches).

An alternative is to try use grouped statistics to capture expected values for observations that fall into similar groups. However, this becomes infeasable for larger data sets. Modeling imputation can automate this process for you and the two most common methods include K-nearest neighbor and tree-based imputation, which are discussed next.

However, it is important to remember that imputation should be performed **within the resampling process** and as your data set gets larger, repeated model-based imputation can compound the computational demands. Thus, you must weigh the pros and cons of the two approaches. The following would build onto our `ames_recipe` and impute all missing values for the `Gr_Liv_Area` variable with the median value:

```
ames_recipe %>%
  step_medianimpute(Gr_Liv_Area)
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Box-Cox transformation on all_outcomes()
## Median Imputation for Gr_Liv_Area
```

Use `step_modeimpute()` to impute categorical features with the most common value.

3.3.2.2 *K*-nearest neighbor

K-nearest neighbor (KNN) imputes values by identifying observations with missing values, then identifying other observations that are most similar based

on the other available features, and using the values from these nearest neighbor observations to impute missing values.

We discuss KNN for predictive modeling in Chapter **??**; the imputation application works in a similar manner. In KNN imputation, the missing value for a given observation is treated as the targeted response and is predicted based on the average (for quantitative values) or the mode (for qualitative values) of the k nearest neighbors.

As discussed in Chapter **??**, if all features are quantitative then standard Euclidean distance is commonly used as the distance metric to identify the k neighbors and when there is a mixture of quantitative and qualitative features then Gower's distance (Gower, 1971) can be used. KNN imputation is best used on small to moderate sized data sets as it becomes computationally burdomesome with larger data sets (Kuhn and Johnson, 2019).

As we saw in Section 2.7, k is a tunable hyperparameter. Suggested values for imputation are 5–10 [@kuhn2019feature]. By default, `step_knnimpute()` will use 5 but can be adjusted with the `neighbors` argument.

```
ames_recipe %>%
  step_knnimpute(all_predictors(), neighbors = 6)
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Box-Cox transformation on all_outcomes()
## 6-nearest neighbor imputation for all_predictors()
```

3.3.2.3 Tree-based

As previously discussed, several implementations of decision trees (Chapter **??**) and their derivatives can be constructed in the presence of missing values. Thus, they provide a good alternative for imputation. As discussed in Chapters **??**-**??**, single trees have high variance but aggregating across many trees creates a robust, low variance predictor. Random forest imputation procedures

have been studied (Shah et al., 2014; Stekhoven, 2015); however, they require significant computational demands in a resampling environment (Kuhn and Johnson, 2019). Bagged trees (Chapter **??**) offer a compromise between predictive accuracy and computational burden.

Similar to KNN imputation, observations with missing values are identified and the feature containing the missing value is treated as the target and predicted using bagged decision trees.

```
ames_recipe %>%
  step_bagimpute(all_predictors())
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Box-Cox transformation on all_outcomes()
## Bagged tree imputation for all_predictors()
```

Figure 3.5 illustrates the differences between mean, KNN, and tree-based imputation on the raw Ames housing data. It is apparent how descriptive statistic methods (e.g., using the mean and median) are inferior to the KNN and tree-based imputation methods.

3.4 Feature filtering

In many data analyses and modeling projects we end up with hundreds or even thousands of collected features. From a practical perspective, a model with more features often becomes harder to interpret and is costly to compute. Some models are more resistant to non-informative predictors (e.g., the Lasso and tree-based methods) than others as illustrated in Figure 3.6.[6]

Although the performance of some of our models are not significantly affected

[6] See Kuhn and Johnson (2013) section 19.1 for data set generation.

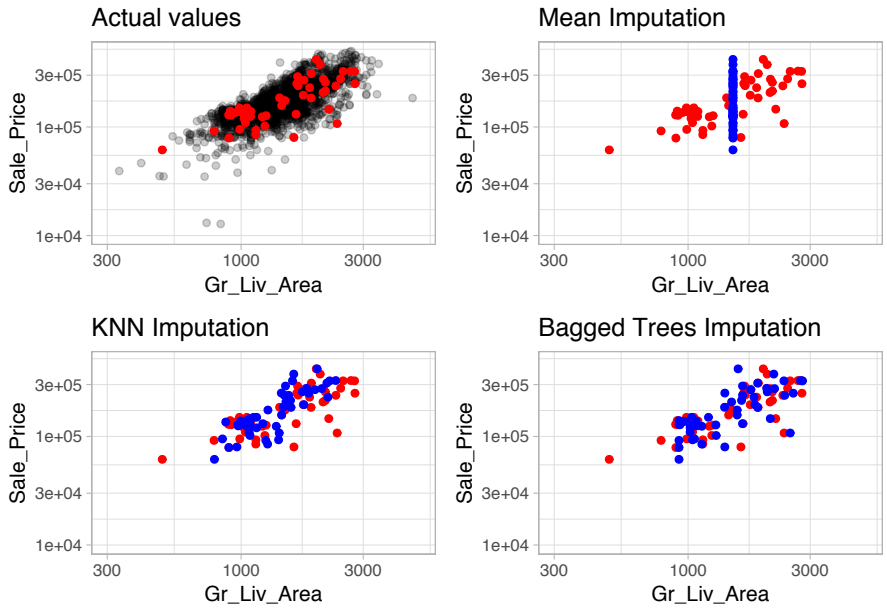

FIGURE 3.5: Comparison of three different imputation methods. The red points represent actual values which were removed and made missing and the blue points represent the imputed values. Estimated statistic imputation methods (i.e. mean, median) merely predict the same value for each observation and can reduce the signal between a feature and the response; whereas KNN and tree-based procedures tend to maintain the feature distribution and relationship.

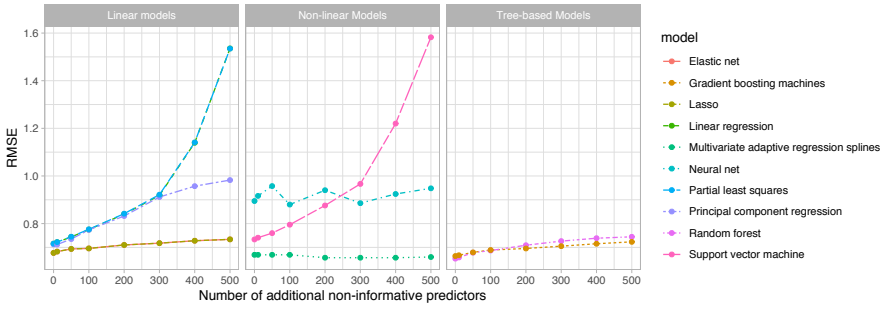

FIGURE 3.6: Test set RMSE profiles when non-informative predictors are added.

by non-informative predictors, the time to train these models can be negatively impacted as more features are added. Figure 3.7 shows the increase in time to perform 10-fold CV on the exemplar data, which consists of 10,000 observations. We see that many algorithms (e.g., elastic nets, random forests, and gradient boosting machines) become extremely time intensive the more predictors we add. Consequently, filtering or reducing features prior to modeling may significantly speed up training time.

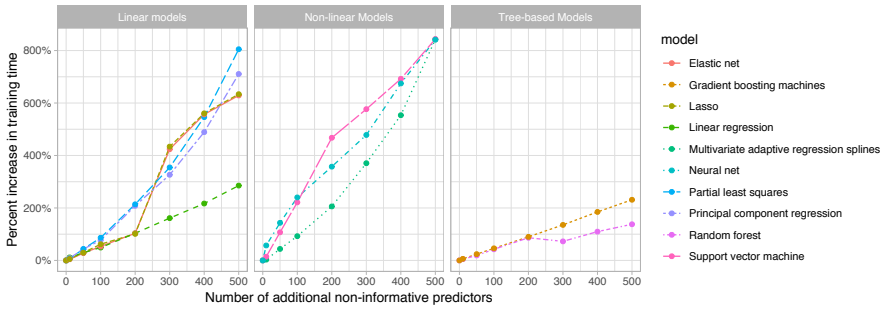

FIGURE 3.7: Impact in model training time as non-informative predictors are added.

Zero and near-zero variance variables are low-hanging fruit to eliminate. Zero variance variables, meaning the feature only contains a single uniquen value, provides no useful information to a model. Some algorithms are unaffected by zero variance features. However, features that have near-zero variance also offer very little, if any, information to a model. Furthermore, they can cause problems during resampling as there is a high probability that a given sample will only contain a single unique value (the dominant value) for that feature. A rule of thumb for detecting near-zero variance features is:

- The fraction of unique values over the sample size is low (say $ 10$%).
- The ratio of the frequency of the most prevalent value to the frequency of the second most prevalent value is large (say $ 20$%).

If both of these criteria are true then it is often adventageous to remove the variable from the model. For the Ames data, we do not have any zero variance predictors but there are 20 features that meet the near-zero threshold.

```
caret::nearZeroVar(ames_train, saveMetrics= TRUE) %>%
  rownames_to_column() %>%
  filter(nzv)
##               rowname freqRatio percentUnique zeroVar
## 1              Street    255.75       0.09737   FALSE
```

```
## 2              Alley     24.69       0.14606   FALSE
## 3        Land_Contour     21.88       0.19474   FALSE
## 4           Utilities   2052.00       0.14606   FALSE
## 5          Land_Slope     21.68       0.14606   FALSE
## 6         Condition_2    184.73       0.34080   FALSE
## 7           Roof_Matl    112.50       0.29211   FALSE
## 8           Bsmt_Cond     24.03       0.29211   FALSE
## 9      BsmtFin_Type_2     23.13       0.34080   FALSE
## 10            Heating     91.82       0.29211   FALSE
## 11    Low_Qual_Fin_SF    674.33       1.41188   FALSE
## 12      Kitchen_AbvGr     22.85       0.19474   FALSE
## 13         Functional     35.30       0.38948   FALSE
## 14     Enclosed_Porch    109.06       7.20545   FALSE
## 15 Three_season_porch    674.67       1.26582   FALSE
## 16       Screen_Porch    186.90       4.81986   FALSE
## 17          Pool_Area   2046.00       0.43817   FALSE
## 18            Pool_QC    682.00       0.24343   FALSE
## 19       Misc_Feature     29.58       0.29211   FALSE
## 20           Misc_Val    124.00       1.26582   FALSE
##     nzv
## 1  TRUE
## 2  TRUE
## 3  TRUE
## 4  TRUE
## 5  TRUE
## 6  TRUE
## 7  TRUE
## 8  TRUE
## 9  TRUE
## 10 TRUE
## 11 TRUE
## 12 TRUE
## 13 TRUE
## 14 TRUE
## 15 TRUE
## 16 TRUE
## 17 TRUE
## 18 TRUE
## 19 TRUE
## 20 TRUE
```

We can add `step_zv()` and `step_nzv()` to our `ames_recipe` to remove zero or near-zero variance features.

Other feature filtering methods exist; see Saeys et al. (2007) for a thorough review. Furthermore, several wrapper methods exist that evaluate multiple models using procedures that add or remove predictors to find the optimal combination of features that maximizes model performance (see, for example, Kursa et al. (2010), Granitto et al. (2006), Maldonado and Weber (2009)). However, this topic is beyond the scope of this book.

3.5 Numeric feature engineering

Numeric features can create a host of problems for certain models when their distributions are skewed, contain outliers, or have a wide range in magnitudes. Tree-based models are quite immune to these types of problems in the feature space, but many other models (e.g., GLMs, regularized regression, KNN, support vector machines, neural networks) can be greatly hampered by these issues. Normalizing and standardizing heavily skewed features can help minimize these concerns.

3.5.1 Skewness

Similar to the process discussed to normalize target variables, parametric models that have distributional assumptions (e.g., GLMs, and regularized models) can benefit from minimizing the skewness of numeric features. When normalizing many variables, its best to use the Box-Cox (when feature values are strictly positive) or Yeo-Johnson (when feature values are not strictly positive) procedures as these methods will identify if a transformation is required and what the optimal transformation will be.

Non-parametric models are rarely affected by skewed features; however, normalizing features will not have a negative affect on these models' performance. For example, normalizing features will only shift the optimal split points in tree-based algorithms. Consequently, when in doubt, normalize.

```
# we can normalize all numeric features, including the response
# at the same time
recipe(Sale_Price ~ ., data = ames_train) %>%
  step_YeoJohnson(all_numeric())
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Yeo-Johnson transformation on all_numeric()
```

3.5.2 Standardization

We must also consider the scale on which the individual features are measured. What are the largest and smallest values across all features and do they span several orders of magnitude? Models that incorporate smooth functions of input features are sensitive to the scale of the inputs. For example, $5X + 2$ is a simple linear function of the input X, and the scale of its output depends directly on the scale of the input. Many algorithms use linear functions within their algorithms, some more obvious (e.g., GLMs and regularized regression) than others (e.g., neural networks, support vector machines, and principal components analysis). Other examples include algorithms that use distance measures such as the Euclidean distance (e.g., k nearest neighbor, k-means clustering, and hierarchical clustering).

For these models and modeling components, it is often a good idea to *standardize* the features. Standardizing features includes *centering* and *scaling* so that numeric variables have zero mean and unit variance, which provides a common comparable unit of measure across all the variables.

Some packages (e.g., **glmnet**, and **caret**) have built-in options to standardize and some do not (e.g., **keras** for neural networks). However, you should standardize your variables within the recipe blueprint so that both training and test data standardization are based on the same mean and variance. This helps to minimize data leakage.

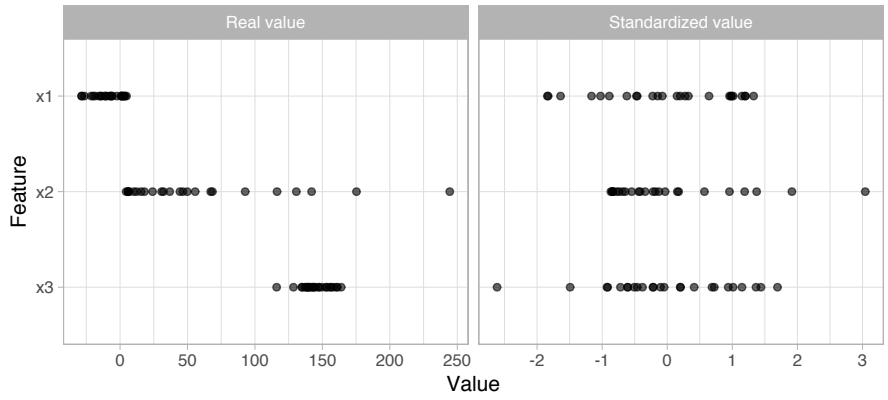

FIGURE 3.8: Standardizing features allows all features to be compared on a common value scale regardless of their real value differences.

```
ames_recipe %>%
  step_center(all_numeric(), -all_outcomes()) %>%
  step_scale(all_numeric(), -all_outcomes())
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Box-Cox transformation on all_outcomes()
## Centering for 2 items
## Scaling for 2 items
```

3.6 Categorical feature engineering

Most models require that the predictors take numeric form. There are exceptions, for example, tree-based models naturally handle numeric or categorical features. However, even tree-based models can benefit from pre-processing cat-

egorical features. The following sections will discuss a few of the more common approaches to engineer categorical features.

3.6.1 Lumping

Sometimes features will contain levels that have very few observations. For example, there are 28 unique neighborhoods represented in the Ames housing data but several of them only have a few observations.

```
count(ames_train, Neighborhood) %>% arrange(n)
## # A tibble: 28 x 2
##    Neighborhood           n
##    <fct>              <int>
##  1 Green_Hills            1
##  2 Landmark               1
##  3 Blueste                5
##  4 Greens                 7
##  5 Veenker               16
##  6 Northpark_Villa       17
##  7 Briardale             22
##  8 Bloomington_Heights   23
##  9 Meadow_Village        27
## 10 Clear_Creek           30
## # ... with 18 more rows
```

Even numeric features can have similar distributions. For example, `Screen_Porch` has 92% values recorded as zero (zero square footage meaning no screen porch) and the remaining 8% have unique dispersed values.

```
count(ames_train, Screen_Porch) %>% arrange(n)
## # A tibble: 99 x 2
##    Screen_Porch     n
##           <int> <int>
##  1           40     1
##  2           53     1
##  3           60     1
##  4           63     1
##  5           80     1
##  6           84     1
##  7           88     1
##  8           92     1
```

```
##  9           94     1
## 10           95     1
## # ... with 89 more rows
```

Sometimes we can benefit from collapsing, or "lumping" these into a lesser number of categories. In the above examples, we may want to collapse all levels that are observed in less than 10% of the training sample into an "other" category. We can use `step_other()` to do so. However, lumping should be used sparingly as there is often a loss in model performance (Kuhn and Johnson, 2013).

Tree-based models often perform exceptionally well with high cardinality features and are not as impacted by levels with small representation.

```
# lump levels for two features
lumping <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_other(Neighborhood, threshold = .01, other = "other") %>%
   step_other(Screen_Porch, threshold = .1, other = ">0")

# apply this blue print --> you will learn about this at
# the end of the chapter
apply_2_training <- prep(lumping, training = ames_train) %>%
  bake(ames_train)

# new distribution of Neighborhood
count(apply_2_training, Neighborhood) %>% arrange(n)
## # A tibble: 23 x 2
##    Neighborhood                            n
##    <fct>                               <int>
##  1 Briardale                              22
##  2 Bloomington_Heights                    23
##  3 Meadow_Village                         27
##  4 Clear_Creek                            30
##  5 South_and_West_of_Iowa_State_University    33
##  6 Stone_Brook                            36
##  7 Timberland                             47
##  8 other                                  47
##  9 Northridge                             55
## 10 Iowa_DOT_and_Rail_Road                 59
## # ... with 13 more rows
```

```
# new distribution of Screen_Porch
count(apply_2_training, Screen_Porch) %>% arrange(n)
## # A tibble: 2 x 2
##   Screen_Porch     n
##   <fct>        <int>
## 1 >0             185
## 2 0             1869
```

3.6.2 One-hot & dummy encoding

Many models require that all predictor variables be numeric. Consequently, we need to intelligently transform any categorical variables into numeric representations so that these algorithms can compute. Some packages automate this process (e.g., **h2o** and **caret**) while others do not (e.g., **glmnet** and **keras**). There are many ways to re, say,code categorical variables as numeric (e.g., one-hot, ordinal, binary, sum, and Helmert).

The most common is referred to as one-hot encoding, where we transpose our categorical variables so that each level of the feature is represented as a boolean value. For example, one-hot encoding `x` in the following

id	x
1	a
2	c
3	a
4	b
5	a
6	c
7	c
8	b

results in the following representation:

id	x.a	x.b	x.c
1	1	0	0
2	0	0	1
3	1	0	0
4	0	1	0
5	1	0	0
6	0	0	1
7	0	0	1
8	0	1	0

This is called less than *full rank* encoding where we retain all variables for

each level of `x`. However, this creates perfect collinearity which causes problems with some predictive modeling algorithms (e.g., ordinary linear regression and neural networks). Alternatively, we can create a full-rank encoding by dropping one of the levels (level `a` has been dropped). This is referred to as *dummy* encoding.

id	x.b	x.c
1	0	0
2	0	1
3	0	0
4	1	0
5	0	0
6	0	1
7	0	1
8	1	0

We can one-hot or dummy encode with the same function (`step_dummy()`). By default, `step_dummy()` will create a full rank encoding but you can change this by setting `one_hot = TRUE`.

```
# lump levels for two features
recipe(Sale_Price ~ ., data = ames_train) %>%
  step_dummy(all_nominal(), one_hot = TRUE)
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Dummy variables from all_nominal()
```

Since one-hot encoding adds new features it can significantly increase the dimensionality of our data. If you have a data set with many categorical variables and those categorical variables in turn have many unique levels, the number of features can explode. In these cases you may want to explore label/ordinal encoding or some other alternative.

3.6.3 Label encoding

Label encoding is a pure numeric conversion of the levels of a categorical variable. If a categorical variable is a factor and it has pre-specified levels then the numeric conversion will be in level order. If no levels are specified, the encoding will be based on alphabetical order. For example, the `MS_SubClass` variable has 16 levels, which we can recode numerically with `step_integer()`.

```
# original categories
count(ames_train, MS_SubClass)
## # A tibble: 16 x 2
##    MS_SubClass                                 n
##    <fct>                                   <int>
##  1 One_Story_1946_and_Newer_All_Styles       749
##  2 One_Story_1945_and_Older                   97
##  3 One_Story_with_Finished_Attic_All_Ages      4
##  4 One_and_Half_Story_Unfinished_All_Ages      14
##  5 One_and_Half_Story_Finished_All_Ages       192
##  6 Two_Story_1946_and_Newer                   401
##  7 Two_Story_1945_and_Older                    94
##  8 Two_and_Half_Story_All_Ages                 16
##  9 Split_or_Multilevel                         87
## 10 Split_Foyer                                 31
## 11 Duplex_All_Styles_and_Ages                  73
## 12 One_Story_PUD_1946_and_Newer                147
## 13 One_and_Half_Story_PUD_All_Ages               1
## 14 Two_Story_PUD_1946_and_Newer                 94
## 15 PUD_Multilevel_Split_Level_Foyer             12
## 16 Two_Family_conversion_All_Styles_and_Ages    42

# label encoded
recipe(Sale_Price ~ ., data = ames_train) %>%
  step_integer(MS_SubClass) %>%
  prep(ames_train) %>%
  bake(ames_train) %>%
  count(MS_SubClass)
## # A tibble: 16 x 2
##    MS_SubClass     n
##          <dbl> <int>
##  1           1   749
##  2           2    97
##  3           3     4
##  4           4    14
##  5           5   192
```

```
##  6           6   401
##  7           7    94
##  8           8    16
##  9           9    87
## 10          10    31
## 11          11    73
## 12          12   147
## 13          13     1
## 14          14    94
## 15          15    12
## 16          16    42
```

We should be careful with label encoding unordered categorical features because most models will treat them as ordered numeric features. If a categorical feature is naturally ordered then label encoding is a natural choice (most commonly referred to as ordinal encoding). For example, the various quality features in the Ames housing data are ordinal in nature (ranging from `Very_Poor` to `Very_Excellent`).

```
ames_train %>% select(contains("Qual"))
## # A tibble: 2,054 x 6
##    Overall_Qual Exter_Qual Bsmt_Qual Low_Qual_Fin_SF
##    <fct>        <fct>      <fct>               <int>
##  1 Above_Avera~ Typical    Typical                 0
##  2 Average      Typical    Typical                 0
##  3 Above_Avera~ Typical    Typical                 0
##  4 Good         Good       Typical                 0
##  5 Above_Avera~ Typical    Typical                 0
##  6 Very_Good    Good       Good                    0
##  7 Very_Good    Good       Good                    0
##  8 Good         Typical    Typical                 0
##  9 Above_Avera~ Typical    Good                    0
## 10 Above_Avera~ Typical    Good                    0
## # ... with 2,044 more rows, and 2 more variables:
## #   Kitchen_Qual <fct>, Garage_Qual <fct>
```

Ordinal encoding these features provides a natural and intuitive interpretation and can logically be applied to all models.

The various `xxx_Qual` features in the Ames housing are not ordered factors. For ordered factors you could also use `step_ordinalscore()`.

```
# original categories
count(ames_train, Overall_Qual)
## # A tibble: 10 x 2
##    Overall_Qual       n
##    <fct>          <int>
##  1 Very_Poor          4
##  2 Poor               8
##  3 Fair              23
##  4 Below_Average    169
##  5 Average          582
##  6 Above_Average    497
##  7 Good             425
##  8 Very_Good        249
##  9 Excellent         75
## 10 Very_Excellent    22

# label encoded
recipe(Sale_Price ~ ., data = ames_train) %>%
  step_integer(Overall_Qual) %>%
  prep(ames_train) %>%
  bake(ames_train) %>%
  count(Overall_Qual)
## # A tibble: 10 x 2
##    Overall_Qual     n
##           <dbl> <int>
##  1            1     4
##  2            2     8
##  3            3    23
##  4            4   169
##  5            5   582
##  6            6   497
##  7            7   425
##  8            8   249
##  9            9    75
## 10           10    22
```

3.6.4 Alternatives

There are several alternative categorical encodings that are implemented in various R machine learning engines and are worth exploring. For example, target encoding is the process of replacing a categorical value with the mean (regression) or proportion (classification) of the target variable. For exam-

ple, target encoding the `Neighborhood` feature would change `North_Ames` to 144617.

Neighborhood	Avg Sale_Price
North_Ames	147040
College_Creek	202438
Old_Town	121815
Edwards	124297
Somerset	232394
Northridge_Heights	320174
Gilbert	191095
Sawyer	137405
Northwest_Ames	186082
Sawyer_West	183062

Target encoding runs the risk of *data leakage* since you are using the reponse variable to encode a feature. An alternative to this is to change the feature value to represent the proportion a particular level represents for a given feature. In this case, `North_Ames` would be changed to 0.153.

In Chapter 9, we discuss how tree-based models use this approach to order categorical features when choosing a split point.

Neighborhood	Proportion
North_Ames	0.1543
College_Creek	0.0930
Old_Town	0.0808
Edwards	0.0638
Somerset	0.0618
Northridge_Heights	0.0579
Gilbert	0.0560
Sawyer	0.0521
Northwest_Ames	0.0399
Sawyer_West	0.0448

Several alternative approaches include effect or likelihood encoding (Micci-Barreca, 2001; Zumel and Mount, 2016), empirical Bayes methods (West et al., 2014), word and entity embeddings (Guo and Berkhahn, 2016; Chollet and Allaire, 2018), and more. For more indepth coverage of categorical encodings we highly recommend Kuhn and Johnson (2019).

3.7 Dimension reduction

Dimension reduction is an alternative approach to filter out non-informative features without manually removing them. We discuss dimension reduction topics in depth later in the book (Chapters **??**-**??**) so please refer to those chapters for details.

However, we wanted to highlight that it is very common to include these types of dimension reduction approaches during the feature engineering process. For example, we may wish to reduce the dimension of our features with principal components analysis (Chapter **??**) and retain the number of components required to explain, say, 95% of the variance and use these components as features in downstream modeling.

```
recipe(Sale_Price ~ ., data = ames_train) %>%
  step_center(all_numeric()) %>%
  step_scale(all_numeric()) %>%
  step_pca(all_numeric(), threshold = .95)
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Centering for all_numeric()
## Scaling for all_numeric()
## PCA extraction with all_numeric()
```

3.8 Proper implementation

We stated at the beginning of this chapter that we should think of feature engineering as creating a blueprint rather than manually performing each task individually. This helps us in two ways: (1) thinking sequentially and (2) to apply appropriately within the resampling process.

3.8.1 Sequential steps

Thinking of feature engineering as a blueprint forces us to think of the ordering of our pre-processing steps. Although each particular problem requires you to think of the effects of sequential pre-processing, there are some general suggestions that you should consider:

- If using a log or Box-Cox transformation, don't center the data first or do any operations that might make the data non-positive. Alternatively, use the Yeo-Johnson transformation so you don't have to worry about this.
- One-hot or dummy encoding typically results in sparse data which many algorithms can operate efficiently on. If you standardize sparse data you will create dense data and you loose the computational efficiency. Consequently, its often preferred to standardize your numeric features and then one-hot/dummy encode.
- If you are lumping infrequently categories together, do so before one-hot/dummy encoding.
- Although you can perform dimension reduction procedures on categorical features, it is common to primarily do so on numeric features when doing so for feature engineering purposes.

While your project's needs may vary, here is a suggested order of potential steps that should work for most problems:

1. Filter out zero or near-zero variance features.
2. Perform imputation if required.
3. Normalize to resolve numeric feature skewness.
4. Standardize (center and scale) numeric features.
5. Perform dimension reduction (e.g., PCA) on numeric features.
6. One-hot or dummy encode categorical features.

3.8.2 Data leakage

Data leakage is when information from outside the training data set is used to create the model. Data leakage often occurs during the data preprocessing period. To minimize this, feature engineering should be done in isolation of each resampling iteration. Recall that resampling allows us to estimate the generalizable prediction error. Therefore, we should apply our feature engineering blueprint to each resample independently as illustrated in Figure 3.9. That way we are not leaking information from one data set to another (each resample is designed to act as isolated training and test data).

For example, when standardizing numeric features, each resampled training data should use its own mean and variance estimates and these specific values

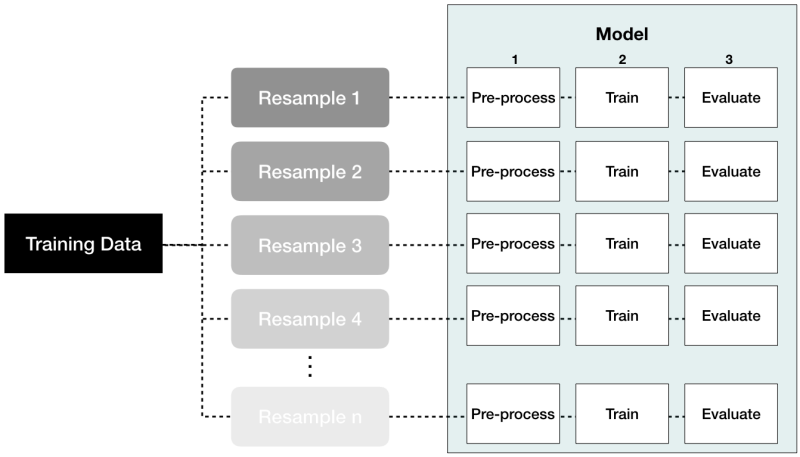

FIGURE 3.9: Performing feature engineering pre-processing within each resample helps to minimize data leakage.

should be applied to the same resampled test set. This imitates how real-life prediction occurs where we only know our current data's mean and variance estimates; therefore, on new data that comes in where we need to predict we assume the feature values follow the same distribution of what we've seen in the past.

3.8.3 Putting the process together

To illustrate how this process works together via R code, let's do a simple re-assessment on the `ames` data set that we did at the end of the last chapter (section 2.7) and see if some simple feature engineering improves our prediction error. But first, we'll formally introduce the **recipes** package, which we've been implicitly illustrating throughout.

The **recipes** package allows us to develop our feature engineering blueprint in a sequential nature. The idea behind **recipes** is similar to `caret::preProcess()` where we want to create the pre-processing blueprint but apply it later and within each resample.[7]

There are three main steps in creating and applying feature engineering with **recipes**:

1. `recipe`: where you define your feature engineering steps to create your blueprint.

[7] In fact, most of the feature engineering capabilities found in **resample** can also be found in `caret::preProcess()`.

2. `prep`are: estimate feature engineering parameters based on training data.
3. `bake`: apply the blueprint to new data.

The first step is where you define your blueprint (aka recipe). With this process, you supply the formula of interest (the target variable, features, and the data these are based on) with `recipe()` and then you sequentially add feature engineering steps with `step_xxx()`. For example, the following defines `Sale_Price` as the target variable and then uses all the remaining columns as features based on `ames_train`. We then:

1. Remove near-zero variance features that are categorical (aka nominal).
2. Ordinally encode our quality-based features (which are inherently ordinal).
3. Center and scale (i.e., standardize) all numeric fetures.
4. Perform dimension reduction by applying PCA to all numeric features.

```
blueprint <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_nzv(all_nominal())  %>%
  step_integer(matches("Qual|Cond|QC|Qu")) %>%
  step_center(all_numeric(), -all_outcomes()) %>%
  step_scale(all_numeric(), -all_outcomes()) %>%
  step_pca(all_numeric(), -all_outcomes())

blueprint
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Sparse, unbalanced variable filter on all_nominal()
## Integer encoding for matches("Qual|Cond|QC|Qu")
## Centering for 2 items
## Scaling for 2 items
## PCA extraction with 2 items
```

Next, we need to train this blueprint on some training data. Remember, there are many feature engineering steps that we do not want to train on the test data (e.g., standardize and PCA) as this would create data leakage. So in this step we estimate these parameters based on the training data of interest.

```
prepare <- prep(blueprint, training = ames_train)
prepare
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Training data contained 2054 data points and no missing data.
##
## Operations:
##
## Sparse, unbalanced variable filter removed Street, ... [trained]
## Integer encoding for Condition_1, ... [trained]
## Centering for Lot_Frontage, ... [trained]
## Scaling for Lot_Frontage, ... [trained]
## PCA extraction with Lot_Frontage, ... [trained]
```

Lastly, we can apply our blueprint to new data (e.g., the training data or future test data) with `bake()`.

```
baked_train <- bake(prepare, new_data = ames_train)
baked_test <- bake(prepare, new_data = ames_test)
baked_train
## # A tibble: 2,054 x 27
##    MS_SubClass MS_Zoning Lot_Shape Lot_Config
##    <fct>       <fct>     <fct>     <fct>
##  1 One_Story_~ Resident~ Slightly~ Corner
##  2 One_Story_~ Resident~ Regular   Inside
##  3 One_Story_~ Resident~ Slightly~ Corner
##  4 One_Story_~ Resident~ Regular   Corner
##  5 Two_Story_~ Resident~ Slightly~ Inside
##  6 One_Story_~ Resident~ Slightly~ Inside
##  7 One_Story_~ Resident~ Slightly~ Inside
##  8 Two_Story_~ Resident~ Regular   Inside
```

```
##  9 Two_Story_~ Resident~ Slightly~ Corner
## 10 One_Story_~ Resident~ Slightly~ Inside
## # ... with 2,044 more rows, and 23 more variables:
## #   Neighborhood <fct>, Bldg_Type <fct>,
## #   House_Style <fct>, Roof_Style <fct>,
## #   Exterior_1st <fct>, Exterior_2nd <fct>,
## #   Mas_Vnr_Type <fct>, Foundation <fct>,
## #   Bsmt_Exposure <fct>, BsmtFin_Type_1 <fct>,
## #   Central_Air <fct>, Electrical <fct>,
## #   Garage_Type <fct>, Garage_Finish <fct>,
## #   Paved_Drive <fct>, Fence <fct>, Sale_Type <fct>,
## #   Sale_Price <int>, PC1 <dbl>, PC2 <dbl>, PC3 <dbl>,
## #   PC4 <dbl>, PC5 <dbl>
```

Consequently, the goal is to develop our blueprint, then within each resample iteration we want to apply `prep()` and `bake()` to our resample training and validation data. Luckily, the **caret** package simplifies this process. We only need to specify the blueprint and **caret** will automatically prepare and bake within each resample. We illustrate with the `ames` housing example.

First, we create our feature engineering blueprint to perform the following tasks:

1. Filter out near-zero variance features for categorical features.
2. Ordinally encode all quality features, which are on a 1–10 likert scale.
3. Standardize (center and scale) all numeric features.
4. One-hot encode our remaining categorical features.

```
blueprint <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_nzv(all_nominal()) %>%
  step_integer(matches("Qual|Cond|QC|Qu")) %>%
  step_center(all_numeric(), -all_outcomes()) %>%
  step_scale(all_numeric(), -all_outcomes()) %>%
  step_dummy(all_nominal(), -all_outcomes(), one_hot = TRUE)
```

Next, we apply the same resampling method and hyperparameter search grid as we did in Section 2.7. The only difference is when we train our resample models with `train()`, we supply our blueprint as the first argument and then **caret** takes care of the rest.

```
# 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_fit2 <- train(
  blueprint,
  data = ames_train,
  method = "knn",
  trControl = cv,
  tuneGrid = hyper_grid,
  metric = "RMSE"
  )
```

Looking at our results we see that the best model was associated with $k = 12$, which resulted in a cross-validated RMSE of 32,991. Figure 3.10 illustrates the cross-validated error rate across the spectrum of hyperparameter values that we specified.

```
# print model results
knn_fit2
## k-Nearest Neighbors
##
## 2054 samples
##   80 predictor
##
## Recipe steps: nzv, integer, center, scale, dummy
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 1848, 1850, 1848, 1850, 1850, 1848, ...
## Resampling results across tuning parameters:
##
##   k   RMSE   Rsquared  MAE
##    2  36949  0.7931    22711
##    3  35724  0.8071    21985
##    4  34731  0.8194    21332
##    5  34127  0.8271    20948
```

```
##    6  33910  0.8314    20777
##    7  33414  0.8389    20670
##    8  33237  0.8428    20568
##    9  33145  0.8452    20541
##   10  33043  0.8476    20499
##   11  33006  0.8485    20508
##   12  32991  0.8495    20541
##   13  33095  0.8491    20610
##   14  33116  0.8496    20631
##   15  33075  0.8508    20657
##   16  33105  0.8510    20696
##   17  33194  0.8509    20738
##   18  33289  0.8504    20828
##   19  33373  0.8500    20889
##   20  33468  0.8497    20963
##   21  33529  0.8498    21020
##   22  33581  0.8498    21065
##   23  33652  0.8496    21133
##   24  33702  0.8498    21202
##   25  33710  0.8504    21236
##
## RMSE was used to select the optimal model using
##  the smallest value.
## The final value used for the model was k = 12.

# plot cross validation results
ggplot(knn_fit2)
```

By applying a handful of the preprocessing techniques discussed throughout this chapter, we were able to reduce our prediction error by over $10,000. The chapters that follow will look to see if we can continue reducing our error by applying different algorithms and feature engineering blueprints.

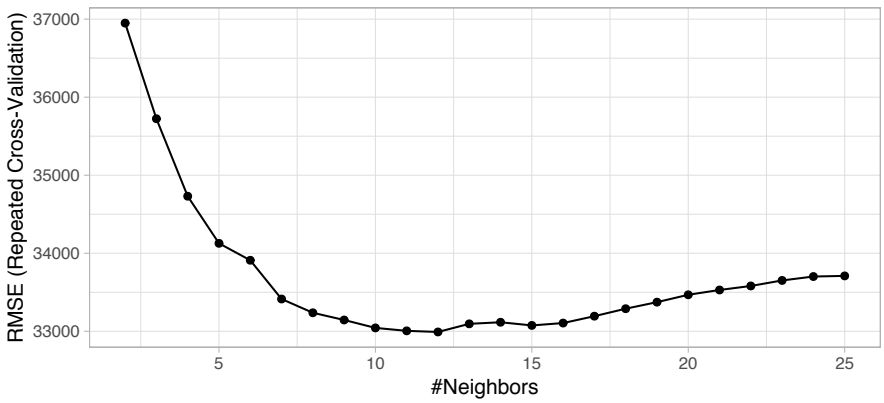

FIGURE 3.10: Results from the same grid search performed in Section 2.7 but with feature engineering performed within each resample.

Part II

Supervised Learning

4

Linear Regression

Linear regression, a staple of classical statistical modeling, is one of the simplest algorithms for doing supervised learning. Though it may seem somewhat dull compared to some of the more modern statistical learning approaches described in later chapters, linear regression is still a useful and widely applied statistical learning method. Moreover, it serves as a good starting point for more advanced approaches; as we will see in later chapters, many of the more sophisticated statistical learning approaches can be seen as generalizations to or extensions of ordinary linear regression. Consequently, it is important to have a good understanding of linear regression before studying more complex learning methods. This chapter introduces linear regression with an emphasis on prediction, rather than inference. An excellent and comprehensive overview of linear regression is provided in Kutner et al. (2005). See Faraway (2016) for a discussion of linear regression in R (the book's website also provides Python scripts).

4.1 Prerequisites

This chapter leverages the following packages:

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

# Modeling packages
library(caret)    # for logistic regression modeling

# Model interpretability packages
library(vip)      # variable importance
```

We'll also continue working with the `ames_train` data set created in Section 2.7.

4.2 Simple linear regression

Pearson's correlation coefficient is often used to quantify the strength of the linear association between two continuous variables. In this section, we seek to fully characterize that linear relationship. *Simple linear regression* (SLR) assumes that the statistical relationship between two continuous variables (say X and Y) is (at least approximately) linear:

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i, \quad \text{for } i = 1, 2, \ldots, n, \tag{4.1}$$

where Y_i represents the i-th response value, X_i represents the i-th feature value, β_0 and β_1 are fixed, but unknown constants (commonly referred to as coefficients or parameters) that represent the intercept and slope of the regression line, respectively, and ϵ_i represents noise or random error. In this Chapter, we'll assume that the errors are normally distirbuted with mean zero and constant variance σ^2, denoted $\overset{iid}{\sim} (0, \sigma^2)$. Since the random errors are centered around zero (i.e., $E(\epsilon) = 0$), linear regression is really a problem of estimating a *conditional mean*:

$$E(Y_i|X_i) = \beta_0 + \beta_1 X_i. \tag{4.2}$$

For brevity, we often drop the conditional piece and write $E(Y|X) = E(Y)$. Consequently, the interpretation of the coefficients are in terms of the average, or mean response. For example, the intercept β_0 represents the average response value when $X = 0$ (it is often not meaningful or of interest and is is sometimes referred to as a *bias term*). The slope β_1 represents the increase in the average response per one-unit increase in X (i.e., it is a *rate of change*).

4.2.1 Estimation

Ideally, we want estimates of β_0 and β_1 that give us the "best fitting" line. But what is meant by "best fitting"? The most common approach is to use the method of *least squares* (LS) estimation; this form of linear regression is often referred to as ordinary least squares (OLS) regression. There are multiple ways to measure "best fitting", but the LS criterion finds the "best fitting" line by minimizing the *residual sum of squares* (RSS):

$$RSS(\beta_0, \beta_1) = \sum_{i=1}^{n} [Y_i - (\beta_0 + \beta_1 X_i)]^2 = \sum_{i=1}^{n} (Y_i - \beta_0 - \beta_1 X_i)^2. \tag{4.3}$$

The LS estimates of β_0 and β_1 are denoted as $\hat{\beta}_0$ and $\hat{\beta}_1$, respectively. Once

obtained, we can generate predicted values, say at $X = X_{new}$, using the estimated regression equation:

$$\widehat{Y}_{new} = \hat{\beta}_0 + \hat{\beta}_1 X_{new}, \tag{4.4}$$

where $\widehat{Y}_{new} = E\left(Y_{new}|\widehat{X = X_{new}}\right)$ is the estimated mean response at $X = X_{new}$.

With the Ames housing data, suppose we wanted to model a linear relationship between the total above ground living space of a home (`Gr_Liv_Area`) and sale price (`Sale_Price`). To perform an OLS regression model in R we can use the `lm()` function:

```
model1 <- lm(Sale_Price ~ Gr_Liv_Area, data = ames_train)
```

The fitted model (`model1`) is displayed in the left plot in Figure 4.1 where the points represent the values of `Sale_Price` in the training data. In the right plot of Figure 4.1, the vertical lines represent the individual errors, called *residuals*, associated with each observation. The OLS criterion (4.3) identifies the "best fitting" line that minimizes the sum of squares of these residuals.

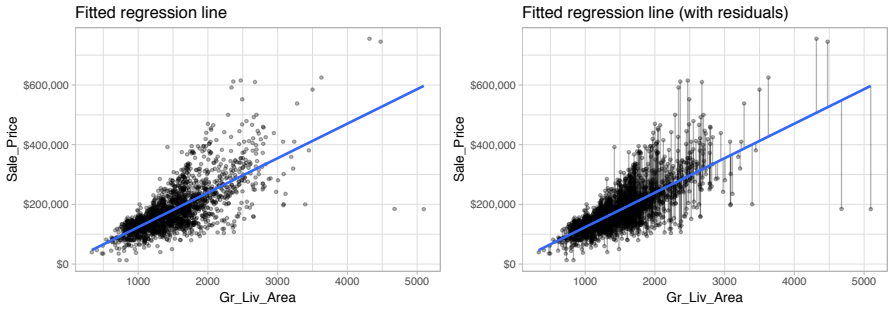

FIGURE 4.1: The least squares fit from regressing sale price on living space for the the Ames housing data. Left: Fitted regresison line. Right: Fitted regression line with vertical grey bars representing the residuals.

The `coef()` function extracts the estimated coefficients from the model. We can also use `summary()` to get a more detailed report of the model results.

```
summary(model1)
##
## Call:
## lm(formula = Sale_Price ~ Gr_Liv_Area, data = ames_train)
```

```
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -413052  -30218   -1612   23383  330421
##
## Coefficients:
##             Estimate Std. Error t value
## (Intercept)  7989.35    3892.40    2.05
## Gr_Liv_Area   115.59       2.46   46.90
##                        Pr(>|t|)
## (Intercept)                0.04 *
## Gr_Liv_Area <0.0000000000000002 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 55800 on 2052 degrees of freedom
## Multiple R-squared:  0.517,  Adjusted R-squared:  0.517
## F-statistic: 2.2e+03 on 1 and 2052 DF,  p-value: <0.0000000000000002
```

The estimated coefficients from our model are $\hat{\beta}_0 = 7989.35$ and $\hat{\beta}_1 = 115.59$. To interpret, we estimate that the mean selling price increases by 115.59 for each additional one square foot of above ground living space. This simple description of the relationship between the sale price and square footage using a single number (i.e., the slope) is what makes linear regression such an intuitive and popular modeling tool.

One drawback of the LS procedure in linear regression is that it only provides estimates of the coefficents; it does not provide an estimate of the error variance σ^2! LS also makes no assumptions about the random errors. These assumptions are important for inference and in estimating the error variance which we're assuming is a constant value σ^2. One way to estimate σ^2 (which is required for characterizing the variability of our fitted model), is to use the method of *maximum likelihood* (ML) estimation (see Kutner et al. (2005) sec 1.7 for details). The ML procedure requires that we assume a particular distribution for the random errors. Most often, we assume the errors to be normally distributed. In practice, under the usual assumptions stated above, an unbiased estimate of the error variance is given as the sum of the squared residuals divided by $n - p$ (where p is the number of regression coefficients or parameters in the model):

$$\hat{\sigma}^2 = \frac{1}{n-p} \sum_{i=1}^{n} r_i^2, \tag{4.5}$$

where $r_i = \left(Y_i - \widehat{Y}_i\right)$ is referred to as the i-th residual (i.e., the difference between the i-th observed and predicted response value). The quantity $\hat{\sigma}^2$ is also referred to as the *mean square error* (MSE) and it's square root is denoted RMSE (see Section 2.6 for discussion on these metrics). In R, the RMSE of a linear model can be extracted using the `sigma()` function:

Typically, these error metrics are computed on a separate validation set or using cross-validation as discussed in Section 2.4; however, they can also be computed on the same training data the model was trained on as illustrated here.

```
sigma(model1)    # RMSE
## [1] 55753
sigma(model1)^2  # MSE
## [1] 3108447546
```

Note that the RMSE is also reported as the `Residual standard error` in the output from `summary()`.

4.2.2 Inference

How accurate are the LS of β_0 and β_1? Point estimates by themselves are not very useful. It is often desirable to associate some measure of an estimates variability. The variability of an estimate is often measured by its *standard error* (SE)—the square root of its variance. If we assume that the errors in the linear regression model are $\overset{iid}{\sim} (0, \sigma^2)$, then simple expressions for the SEs of the estimated coefficients exist and are displayed in the column labeled `Std. Error` in the output from `summary()`. From this, we can also derive simple t-tests to understand if the individual coefficients are statistically significant from zero. The t-statistics for such a test are nothing more than the estimated coefficients divided by their corresponding estimated standard errors (i.e., in the output from `summary()`, `t value` = `Estimate` / `Std. Error`). The reported t-statistics measure the number of standard deviations each coefficient is away from 0. Thus, large t-statistics (greater than two in absolute value, say) roughly indicate statistical significance at the $\alpha = 0.05$ level. The p-values for these tests are also reported by `summary()` in the column labeled `Pr(>|t|)`.

Under the same assumptions, we can also derive confidence intervals for the coefficients. The formula for the traditional $100\,(1-\alpha)\%$ confidence interval for β_j is

$$\hat{\beta}_j \pm t_{1-\alpha/2,n-p}\widehat{SE}\left(\hat{\beta}_j\right), \tag{4.6}$$

In R, we can construct such (one-at-a-time) confidence intervals for each coefficient using `confint()`. For example, a 95% confidence intervals for the coefficients in our SLR example can be computed using

```
confint(model1, level = 0.95)
##                2.5 %   97.5 %
## (Intercept)  355.9  15622.8
## Gr_Liv_Area 110.8    120.4
```

To interpret, we estimate with 95% confidence that the mean selling price increases between 110.75 and 120.42 for each additional one square foot of above ground living space. We can also conclude that the slope β_1 is significantly different from zero (or any other pre-specified value not included in the interval) at the $\alpha = 0.05$ level. This is also supported by the output from `summary()`.

Most statistical software, including R, will include estimated standard errors, t-statistics, etc. as part of its regression output. However, it is important to remember that such quantities depend on three major assumptions of the linear regresion model:

1. Independent observations
2. The random errors have mean zero, and constant variance
3. The random errors are normally distributed

If any or all of these assumptions are violated, then remdial measures need to be taken. For instance, *weighted least squares* (and other procedures) can be used when the constant variance assumption is violated. Transformations (of both the response and features) can also help to correct departures from these assumptions. The residuals are extremely useful in helping to identify how parametric models depart from such assumptions.

4.3 Multiple linear regression

In practice, we often have more than one predictor. For example, with the Ames housing data, we may wish to understand if above ground square footage

(`Gr_Liv_Area`) and the year the house was built (`Year_Built`) are (linearly) related to sale price (`Sale_Price`). We can extend the SLR model so that it can directly accommodate multiple predictors; this is referred to as the *multiple linear regression* (MLR) model. With two predictors, the MLR model becomes:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \epsilon, \tag{4.7}$$

where X_1 and X_2 are features of interest. In our Ames housing example, X_1 represents `Gr_Liv_Area` and X_2 represents `Year_Built`.

In R, multiple linear regression models can be fit by separating all the features of interest with a `+`:

```
(model2 <- lm(Sale_Price ~ Gr_Liv_Area + Year_Built, data = ames_train))
##
## Call:
## lm(formula = Sale_Price ~ Gr_Liv_Area + Year_Built, data = ames_train)
##
## Coefficients:
## (Intercept)  Gr_Liv_Area   Year_Built
##  -2071358.4         99.2       1067.1
```

Alternatively, we can use `update()` to update the model formula used in `model1`. The new formula can use a `.` as short hand for keep everything on either the left or right hand side of the formula, and a `+` or `-` can be used to add or remove terms from the original model, respectively. In the case of adding `Year_Built` to to `model1`, we could've used:

```
(model2 <- update(model1, . ~ . + Year_Built))
##
## Call:
## lm(formula = Sale_Price ~ Gr_Liv_Area + Year_Built, data = ames_train)
##
## Coefficients:
## (Intercept)  Gr_Liv_Area   Year_Built
##  -2071358.4         99.2       1067.1
```

The LS estimates of the regression coefficients are $\hat{\beta}_1 = 99.169$ and $\hat{\beta}_2 = 1067.108$ (the estimated intercept is -2071358.426. In other words, every one square foot increase to above ground square footage is associated with an additional $99.17 in **mean selling price** when holding the year the house was

built constant. Likewise, for every year newer a home is there is approximately an increase of $1,067.11 in selling price when holding the above ground square footage constant.

A contour plot of the fitted regression surface is displayed in the left side of Figure 4.2 below. Note how the fitted regression surface is flat (i.e., it does not twist or bend). This is true for all linear models that include only *main effects* (i.e., terms involving only a single predictor). One way to model curvature is to include *interaction effects*. An interaction occurs when the effect of one predictor on the response depends on the values of other predictors. In linear regression, interactions can be captured via products of features (i.e., $X1 \times X_2$). A model with two main effects can also include a two-way interaction. For example, to include an interaction between X_1 = `Gr_Liv_Area` and X_2 = `Year_Built`, we introduce an additional product term:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_1 X_2 + \epsilon. \tag{4.8}$$

Note that in R, we use the `:` operator to include an interaction (technically, we could use `*` as well, but `x1 * x2` is shorthand for `x1 + x2 + x1:x2` so is slightly redundant):

```
lm(Sale_Price ~ Gr_Liv_Area + Year_Built + Gr_Liv_Area : Year_Built,
   data = ames_train)
##
## Call:
## lm(formula = Sale_Price ~ Gr_Liv_Area + Year_Built + Gr_Liv_Area:Year_Built,
##     data = ames_train)
##
## Coefficients:
##             (Intercept)             Gr_Liv_Area
##               30353.414               -1243.040
##              Year_Built  Gr_Liv_Area:Year_Built
##                   0.149                   0.681
```

A contour plot of the fitted regression surface with interaction is displayed in the right side of Figure 4.2. Note the curvature in the contour lines.

Interaction effects are quite prevelant in predictive modeling. Since linear models are an example of parametric modeling, it is up to the analyst to decide if and when to include interaction effects. In later chapters, we'll discuss algorithms that can automatically detect and incorporate interaction effects (albeit in different ways). It is also important to understand a concept

called the *hierarchy principle*—which demands that all lower-order terms corresponding to an interaction be retained in the model—when concidering interaction effects in linear regression models.

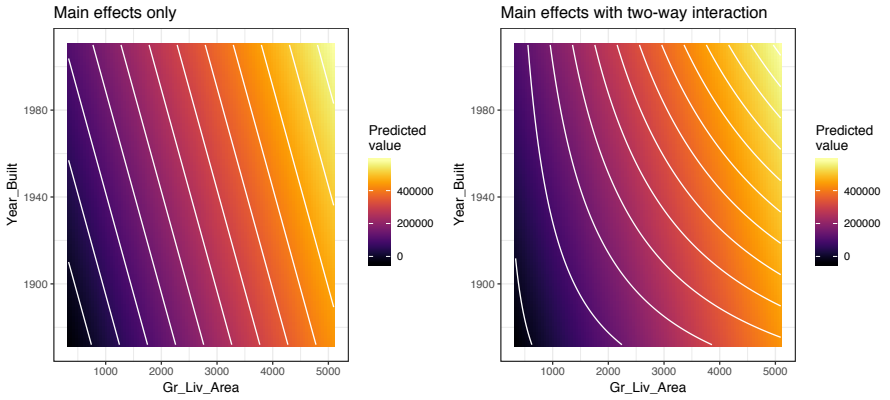

FIGURE 4.2: In a three-dimensional setting, with two predictors and one response, the least squares regression line becomes a plane. The 'best-fit' plane minimizes the sum of squared errors between the actual sales price (individual dots) and the predicted sales price (plane).

In general, we can include as many predictors as we want, as long as we have more rows than parameters! The general multiple linear regression model with p distinct predictors is

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \epsilon, \tag{4.9}$$

where X_i for $i = 1, 2, \ldots, p$ are the predictors of interest. Note some of these may represent interactions (e.g., $X_3 = X_1 \times X_2$) between or transformations[1] (e.g., $X_4 = \sqrt{X_1}$) of the original features. Unfortunately, visualizing beyond three dimensions is not practical as our best-fit plane becomes a hyperplane. However, the motivation remains the same where the best-fit hyperplane is identified by minimizing the RSS. The code below creates a third model where we use all features in our data set as main effects (i.e., no interaction terms) to predict `Sale_Price`.

[1] Transformations of the features serve a number of purposes (e.g., modeling nonlinear relationships or alleviating departures from common regression assumptions). See Kutner et al. (2005) for details.

```
# include all possible main effects
model3 <- lm(Sale_Price ~ ., data = ames_train)

# print estimated coefficients in a tidy data frame
broom::tidy(model3)
## # A tibble: 292 x 5
##    term            estimate std.error statistic p.value
##    <chr>              <dbl>     <dbl>     <dbl>   <dbl>
##  1 (Intercept)      -1.20e7 10949313.     -1.09    0.274
##  2 MS_SubClassOne~   3.37e3     3655.      0.921   0.357
##  3 MS_SubClassOne~   1.21e4    11926.      1.02    0.309
##  4 MS_SubClassOne~   1.16e4    12833.      0.902   0.367
##  5 MS_SubClassOne~   6.67e3     6552.      1.02    0.309
##  6 MS_SubClassTwo~  -1.81e3     6018.     -0.301   0.763
##  7 MS_SubClassTwo~   1.02e4     6612.      1.54    0.124
##  8 MS_SubClassTwo~  -1.62e4    10468.     -1.54    0.123
##  9 MS_SubClassSpl~  -1.03e4    11585.     -0.888   0.375
## 10 MS_SubClassSpl~  -3.13e3     7577.     -0.413   0.680
## # ... with 282 more rows
```

4.4 Assessing model accuracy

We've fit three main effects models to the Ames housing data: a single predictor, two predictors, and all possible predictors. But the question remains, which model is "best"? To answer this question we have to define what we mean by "best". In our case, we'll use the RMSE metric and cross-validation (Section 2.4) to determine the "best" model. We can use the `caret::train()` function to train a linear model (i.e., `method = "lm"`) using cross-validation (or a variety of other validation methods). In practice, a number of factors should be considered in determining a "best" model (e.g., time constraints, model production cost, predictive accuracy, etc.). The benefit of **caret** is that it provides built-in cross-validation capabilities, whereas the `lm()` function does not[2]. The following code chunk uses `caret::train()` to refit `model1` using 10-fold cross-validation:

[2]Although general cross-validation is not avilable in `lm()` alone, a simple metric called the *PRESS* statistic, for **PRE**dictive **S**um of **S**quare, (equivalent to a *leave-one-out* cross-validated RMSE) can be computed by summing the PRESS residuals which are available using `rstandard(<lm-model-name>, type = "predictive")`. See `?rstandard` for details.

```
# Use caret package to train model using 10-fold cross-validation
set.seed(123)  # for reproducibility
(cv_model1 <- train(
  form = Sale_Price ~ Gr_Liv_Area,
  data = ames_train,
  method = "lm",
  trControl = trainControl(method = "cv", number = 10)
))
## Linear Regression
##
## 2054 samples
##    1 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 1848, 1847, 1848, 1849, 1848, 1850, ...
## Resampling results:
##
##   RMSE   Rsquared  MAE
##   55670  0.5214    38380
##
## Tuning parameter 'intercept' was held constant at
##  a value of TRUE
```

The resulting cross-validated RMSE is $55,670.37 (this is the average RMSE across the 10 CV folds). How should we interpret this? When applied to unseen data, the predictions this model makes are, on average, about $55,670.37 off from the actual sale price.

We can perform cross-validation on the other two models in a similar fashion, which we do in the code chunk below.

```
# model 2 CV
set.seed(123)
cv_model2 <- train(
  Sale_Price ~ Gr_Liv_Area + Year_Built,
  data = ames_train,
  method = "lm",
  trControl = trainControl(method = "cv", number = 10)
)

# model 3 CV
set.seed(123)
```

```
cv_model3 <- train(
  Sale_Price ~ .,
  data = ames_train,
  method = "lm",
  trControl = trainControl(method = "cv", number = 10)
)

# Extract out of sample performance measures
summary(resamples(list(
  model1 = cv_model1,
  model2 = cv_model2,
  model3 = cv_model3
)))
##
## Call:
## summary.resamples(object = resamples(list(model1
##  = cv_model1, model2 = cv_model2, model3 = cv_model3)))
##
## Models: model1, model2, model3
## Number of resamples: 10
##
## MAE
##         Min. 1st Qu. Median  Mean 3rd Qu.  Max. NA's
## model1 36295   36806  37005 38380   40034 42096    0
## model2 28076   30690  31325 31479   32620 34536    0
## model3 14257   15855  16131 17080   16689 25677    0
##
## RMSE
##         Min. 1st Qu. Median  Mean 3rd Qu.  Max. NA's
## model1 50003   52413  54193 55670   60344 62415    0
## model2 40456   42957  45597 46133   49114 53745    0
## model3 20945   25674  33769 37304   42967 80339    0
##
## Rsquared
##          Min. 1st Qu. Median   Mean 3rd Qu.   Max.
## model1 0.3805  0.4832 0.5277 0.5214  0.5883 0.6403
## model2 0.5304  0.6666 0.6860 0.6716  0.7084 0.7371
## model3 0.4204  0.7462 0.8287 0.7967  0.9099 0.9224
##        NA's
## model1    0
## model2    0
## model3    0
```

Extracting the results for each model, we see that by adding more informa-

tion via more predictors, we are able to improve the out-of-sample cross validation performance metrics. Specifically, our cross-validated RMSE reduces from $46,132.74 (the model with two predictors) down to $37,304.33 (for our full model). In this case, the model with all possible main effects performs the "best" (compared with the other two).

4.5 Model concerns

As previously stated, linear regression has been a popular modeling tool due to the ease of interpreting the coefficients. However, linear regression makes several strong assumptions that are often violated as we include more predictors in our model. Violation of these assumptions can lead to flawed interpretation of the coefficients and prediction results.

1. Linear relationship: Linear regression assumes a linear relationship between the predictor and the response variable. However, as discussed in Chapter 3, non-linear relationships can be made linear (or near-linear) by applying transformations to the response and/or predictors. For example, Figure 4.3 illustrates the relationship between sale price and the year a home was built. The left plot illustrates the non-linear relationship that exists. However, we can achieve a near-linear relationship by log transforming sale price; although some non-linearity still exists for older homes.

```
p1 <- ggplot(ames_train, aes(Year_Built, Sale_Price)) +
  geom_point(size = 1, alpha = .4) +
  geom_smooth(se = FALSE) +
  scale_y_continuous("Sale price", labels = scales::dollar) +
  xlab("Year built") +
  ggtitle("Non-transformed variables with a \nnon-linear relationship.")

p2 <- ggplot(ames_train, aes(Year_Built, Sale_Price)) +
  geom_point(size = 1, alpha = .4) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_y_log10("Sale price", labels = scales::dollar,
                breaks = seq(0, 400000, by = 100000)) +
  xlab("Year built") +
  ggtitle("Transforming variables can provide a \nnear-linear relationship.")

gridExtra::grid.arrange(p1, p2, nrow = 1)
```

2. Constant variance among residuals: Linear regression assumes the

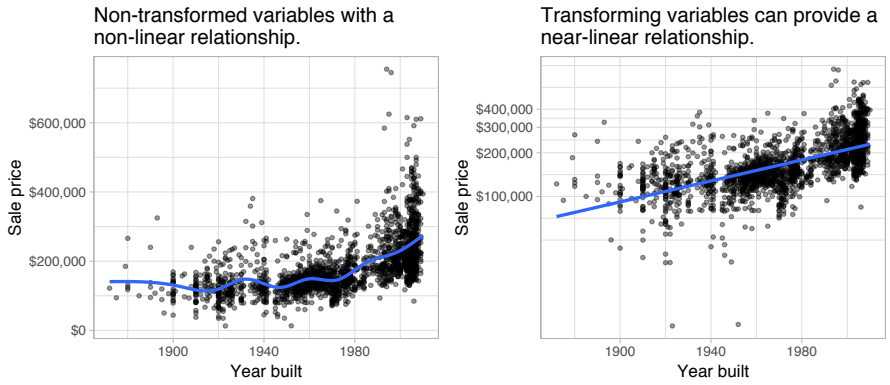

FIGURE 4.3: Linear regression assumes a linear relationship between the predictor(s) and the response variable; however, non-linear relationships can often be altered to be near-linear by appying a transformation to the variable(s).

variance among error terms $(\epsilon_1, \epsilon_2, \dots, \epsilon_p)$ are constant (this assumption is referred to as homoscedasticity). If the error variance is not constant, the p-values and confidence intervals for the coefficients will be invalid. Similar to the linear relationship assumption, non-constant variance can often be resolved with variable transformations or by including additional predictors. For example, Figure 4.4 shows the residuals vs. predicted values for `model1` and `model3`. `model1` displays a classic violation of constant variance as indicated by the cone-shaped pattern. However, `model3` appears to have near-constant variance.

The `broom::augment` function is an easy way to add model results to each observation (i.e. predicted values, residuals).

```
df1 <- broom::augment(cv_model1$finalModel, data = ames_train)

p1 <- ggplot(df1, aes(.fitted, .resid)) +
  geom_point(size = 1, alpha = .4) +
  xlab("Predicted values") +
  ylab("Residuals") +
  ggtitle("Model 1", subtitle = "Sale_Price ~ Gr_Liv_Area")

df2 <- broom::augment(cv_model3$finalModel, data = ames_train)
```

```
p2 <- ggplot(df2, aes(.fitted, .resid)) +
  geom_point(size = 1, alpha = .4)  +
  xlab("Predicted values") +
  ylab("Residuals") +
  ggtitle("Model 3", subtitle = "Sale_Price ~ .")

gridExtra::grid.arrange(p1, p2, nrow = 1)
```

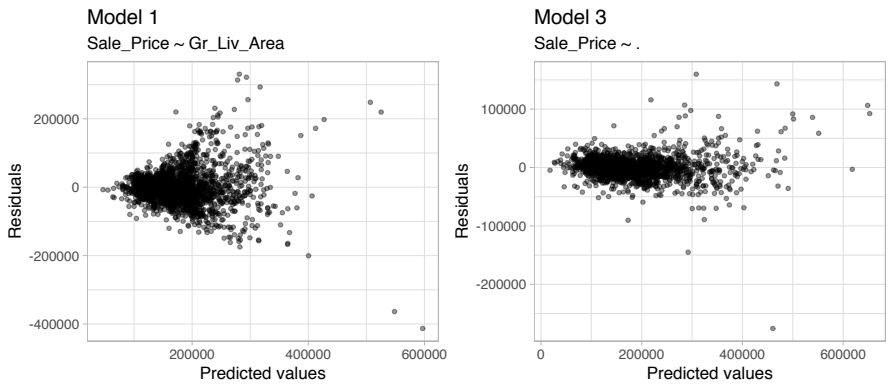

FIGURE 4.4: Linear regression assumes constant variance among the residuals. 'model1' (left) shows definitive signs of heteroskedasticity whereas 'model3' (right) appears to have constant variance.

3. No autocorrelation: Linear regression assumes the errors are independent and uncorrelated. If in fact, there is correlation among the errors, then the estimated standard errors of the coefficients will be biased leading to prediction intervals being narrower than they should be. For example, the left plot in Figure 4.5 displays the residuals (y-axis) vs. the observation ID (x-axis) for `model1`. A clear pattern exists suggesting that information about ϵ_1 provides information about ϵ_2.

This pattern is a result of the data being ordered by neighborhood, which we have not accounted for in this model. Consequently, the residuals for homes in the same neighborhood are correlated (homes within a neighborhood are typically the same size and can often contain similar features). Since the `Neighborhood` predictor is included in `model3` (right plot), the correlation in the errors is reduced.

```
df1 <- mutate(df1, id = row_number())
```

```
df2 <- mutate(df2, id = row_number())

p1 <- ggplot(df1, aes(id, .resid)) +
  geom_point(size = 1, alpha = .4) +
  xlab("Row ID") +
  ylab("Residuals") +
  ggtitle("Model 1",
    subtitle = "Correlated residuals.")

p2 <- ggplot(df2, aes(id, .resid)) +
  geom_point(size = 1, alpha = .4) +
  xlab("Row ID") +
  ylab("Residuals") +
  ggtitle("Model 3",
    subtitle = "Uncorrelated residuals.")

gridExtra::grid.arrange(p1, p2, nrow = 1)
```

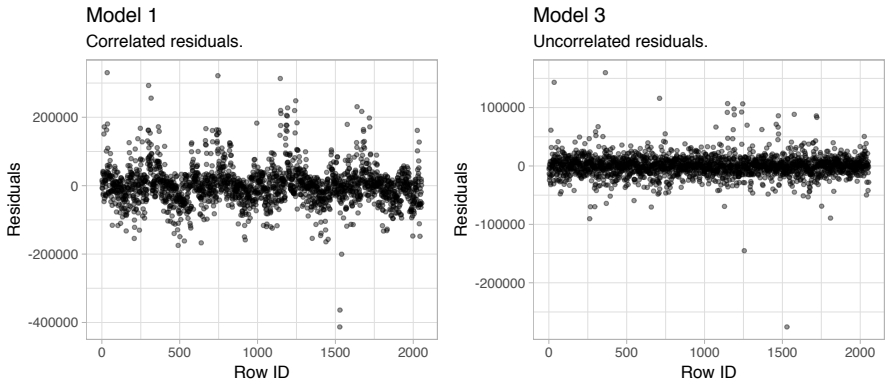

FIGURE 4.5: Linear regression assumes uncorrelated errors. The residuals in 'model1' (left) have a distinct pattern suggesting that information about ϵ_1 provides information about ϵ_2. Whereas 'model3' has no signs of autocorrelation.

4. More observations than predictors: Although not an issue with the Ames housing data, when the number of features exceeds the number of observations ($p > n$), the OLS estimates are not obtainable. To resolve this issue an analyst can remove variables one-at-a-time until $p < n$. Although pre-processing tools can be used to guide this manual approach (Kuhn and Johnson, 2013, 43-47), it can be cumbersome and prone to errors. In Chapter

?? we'll introduce regularized regression which provides an alternative to OLS that can be used when $p > n$.

5. No or little multicollinearity: *Collinearity* refers to the situation in which two or more predictor variables are closely related to one another. The presence of collinearity can pose problems in the OLS, since it can be difficult to separate out the individual effects of collinear variables on the response. In fact, collinearity can cause predictor variables to appear as statistically insignificant when in fact they are significant. This obviously leads to an inaccurate interpretation of coefficients and makes it difficult to identify influential predictors.

In `ames`, for example, `Garage_Area` and `Garage_Cars` are two variables that have a correlation of 0.89 and both variables are strongly related to our response variable (`Sale_Price`). Looking at our full model where both of these variables are included, we see that `Garage_Cars` is found to be statistically significant but `Garage_Area` is not:

```
# fit with two strongly correlated variables
summary(cv_model3) %>%
  broom::tidy() %>%
  filter(term %in% c("Garage_Area", "Garage_Cars"))
## # A tibble: 2 x 5
##   term        estimate std.error statistic p.value
##   <chr>          <dbl>     <dbl>     <dbl>   <dbl>
## 1 Garage_Cars  4962.     1803.        2.75 0.00599
## 2 Garage_Area     9.47      5.97      1.58 0.113
```

However, if we refit the full model without `Garage_Cars`, the coefficient estimate for `Garage_Area` increases two fold and becomes statistically significant.

```
# model without Garage_Area
set.seed(123)
mod_wo_Garage_Cars <- train(
  Sale_Price ~ .,
  data = select(ames_train, -Garage_Cars),
  method = "lm",
  trControl = trainControl(method = "cv", number = 10)
)

summary(mod_wo_Garage_Cars) %>%
  broom::tidy() %>%
  filter(term == "Garage_Area")
```

```
## # A tibble: 1 x 5
##   term        estimate std.error statistic      p.value
##   <chr>          <dbl>     <dbl>     <dbl>        <dbl>
## 1 Garage_Area     21.6      4.02      5.38 0.0000000846
```

This reflects the instability in the linear regression model caused by between-predictor relationships; this instability also gets propagated directly to the model predictions. Considering 16 of our 34 numeric predictors have a medium to strong correlation (Section **??**), the biased coefficients of these predictors are likely restricting the predictive accuracy of our model. How can we control for this problem? One option is to manually remove the offending predictors (one-at-a-time) until all pariwise correlations are below some pre-determined threshold. However, when the number of predictors is large such as in our case, this becomes tedious. Moreover, multicollinearity can arise when one feature is linearly related to two or more features (which is more difficult to detect[3]). In these cases, manual removal of specific predictors may not be possible. Consequently, the following sections offers two simple extensions of linear regression where dimension reduction is applied prior to performing linear regression. Chapter **??** offers a modified regression approach that helps to deal with the problem. And future chapters provide alternative methods that are less effected by multicollinearity.

4.6 Principal component regression

As mentioned in Section 3.7 and fully discussed in Chapter **??**, principal components analysis can be used to represent correlated variables with a smaller number of uncorrelated features (called principle components) and the resulting components can be used as predictors in a linear regression model. This two-step process is known as *principal component regression* (PCR) (Massy, 1965) and is illustrated in Figure 4.6.

Performing PCR with **caret** is an easy extension from our previous model. We simply specify `method = "pcr"` within `train()` to perform PCA on all our numeric predictors prior to fitting the model. Often, we can greatly improve performance by only using a small subset of all principal components as predictors. Consequently, you can think of the number of principal components as a tuning parameter (see Section 2.5.3). The following performs cross-validated PCR with $1, 2, \ldots, 20$ principal components, and Figure 4.7

[3] In such cases we can use a statistic called the *variance inflation factor* which tries to capture how strongly each feature is linearly related to all the others predictors in a model.

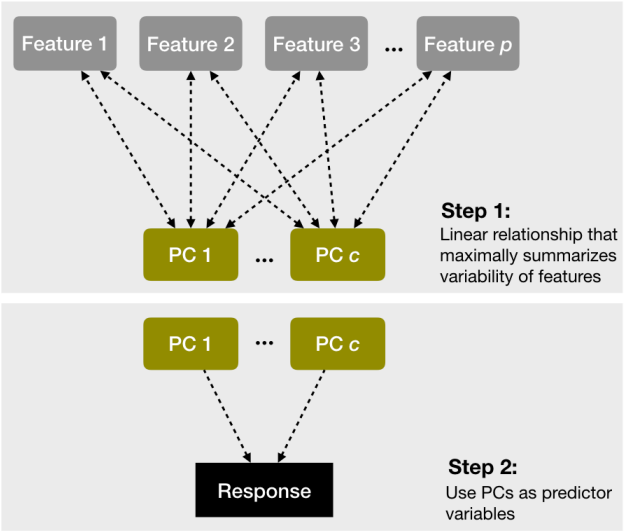

FIGURE 4.6: A depiction of the steps involved in performing principal component regression.

illustrates the cross-validated RMSE. You can see a significant drop in prediction error from our previous linear models using just five principal components followed by a gradual decrease thereafter. Using 17 principal components corresponds to the lowest RMSE (see `cv_model_pcr` for a comparison of the cross-validated results).

Note in the below example we use `preProcess` to remove near-zero variance features and center/scale the numeric features. We then use `method = "pcr"`. This is equivalent to creating a blueprint as illustrated in Section 3.8.3 to remove near-zero variance features, center/scale the numeric features, perform PCA on the nuemric features, then feeding that blueprint into `train()` with `method = "lm"`.

```
# perform 10-fold cross validation on a PCR model tuning the
# number of principal components to use as predictors from 1-20
set.seed(123)
cv_model_pcr <- train(
  Sale_Price ~ .,
  data = ames_train,
  method = "pcr",
```

```
  trControl = trainControl(method = "cv", number = 10),
  preProcess = c("zv", "center", "scale"),
  tuneLength = 20
  )

# model with lowest RMSE
cv_model_pcr$bestTune
##    ncomp
## 19    19

# plot cross-validated RMSE
ggplot(cv_model_pcr)
```

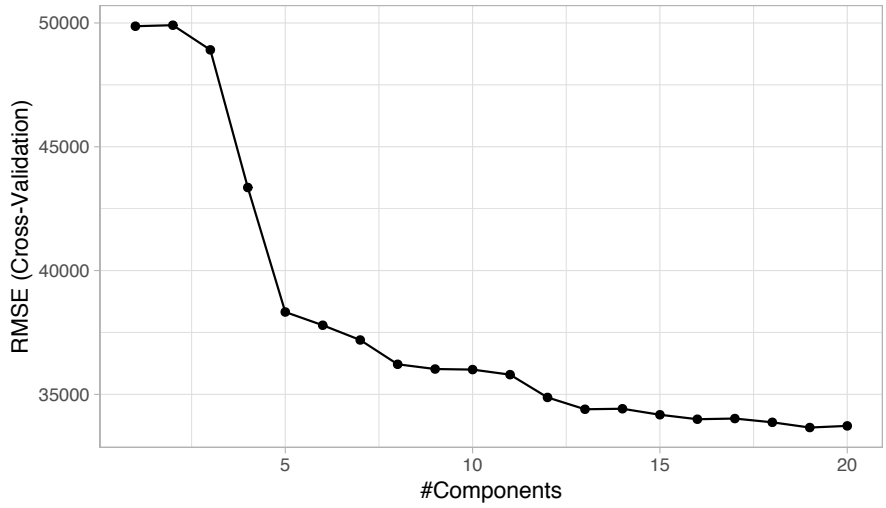

FIGURE 4.7: The 10-fold cross valdation RMSE obtained using PCR with 1-20 principal components.

By controlling for multicollinearity with PCR, we can experience significant improvement in our predictive accuary compared to the previously obtained linear models (reducing the cross-validated RMSE from about $37,000 to below $35,000); however, we still do not improve upon the k-nearest neighbor model illustrated in Section 3.8.3. It's important to note that since PCR is a two step process, the PCA step does not consider any aspects of the response when it selects the components. Consequently, the new predictors produced by the PCA step are not designed to maximize the relationship with the response. Instead, it simply seeks to reduce the variability present throughout the predictor space. If that variability happens to be related to the response

variability, then PCR has a good chance to identify a predictive relationship, as in our case. If, however, the variability in the predictor space is not related to the variability of the response, then PCR can have difficulty identifying a predictive relationship when one might actually exists (i.e., we may actually experience a decrease in our predictive accuracy). An alternative approach to reduce the impact of multicollinearity is partial least squares.

4.7 Partial least squares

Partial least squares (PLS) can be viewed as a supervised dimension reduction procedure (Kuhn and Johnson, 2013). Similar to PCR, this technique also constructs a set of linear combinations of the inputs for regression, but unlike PCR it uses the response variable to aid the construction of the principal components as illustrated in Figure 4.8[4]. Thus, we can think of PLS as a supervised dimension reduction procedure that finds new features that not only captures most of the information in the original features, but also are related to the response.

We illustrate PLS with some exemplar data[5]. Figure 4.9 illustrates that the first two PCs when using PCR have very little relationship to the response variable; however, the first two PCs when using PLS have a much stronger association to the response.

Referring to Equation (**??**) in Chapter **??**, PLS will compute the first principal (z_1) by setting each ϕ_{j1} to the coefficient from a SLR model of y onto that respective x_j. One can show that this coefficient is proportional to the correlation between y and x_j. Hence, in computing $z_1 = \sum_{j=1}^{p} \phi_{j1} x_j$, PLS places the highest weight on the variables that are most strongly related to the response.

To compute the second PC (z_2), we first regress each variable on z_1. The residuals from this regression capture the remaining signal that has not been explained by the first PC. We substitute these residual values for the predictor values in Equation (**??**) in Chapter **??**. This process continues until all m components have been computed and then we use OLS to regress the response on $z_1, \ldots, z_m$.

[4] Figure 4.8 was inspired by, and modified from, Chapter 6 in Kuhn and Johnson (2013).

[5] This is actually using the solubility data that is provided by the **AppliedPredictiveModeling** package (Kuhn and Johnson, 2018)

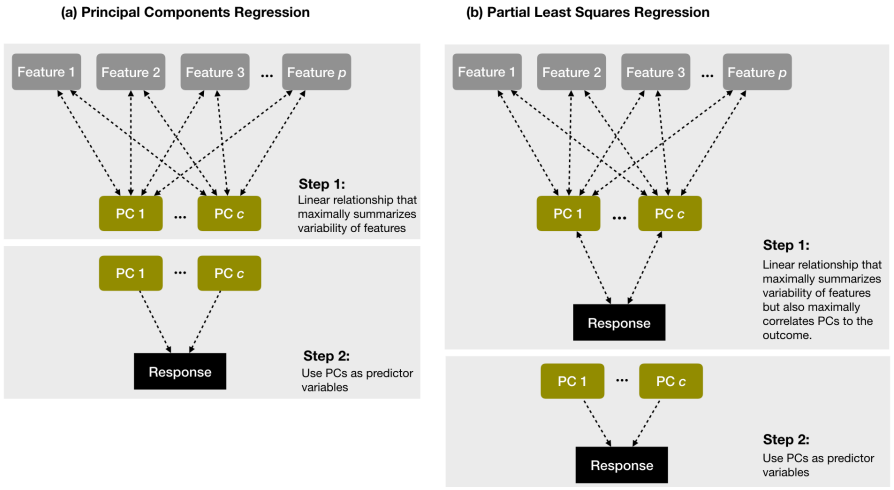

FIGURE 4.8: A diagram depicting the differences between PCR (left) and PLS (right). PCR finds principal components (PCs) that maximally summarize the features independent of the response variable and then uses those PCs as predictor variables. PLS finds components that simultaneously summarize variation of the predictors while being optimally correlated with the outcome and then uses those PCs as predictors.

See @esl and @geladi1986partial for a thorough discussion of PLS.

Similar to PCR, we can easily fit a PLS model by changing the `method` argument in `train()`. As with PCR, the number of principal components to use is a tuning parameter that is determined by the model that maximizes predictive accuracy (minimizes RMSE in this case). The following performs cross-validated PLS with $1, 2, \ldots, 20$ PCs, and Figure 4.10 shows the cross-validated RMSEs. You can see a greater drop in prediction error compared to PCR. Using PLS with $m = 3$ principal components corresponded with the lowest cross-validated RMSE of $29,970.

```
# perform 10-fold cross validation on a PLS model tuning the number of
# principal components to use as predictors from 1-20
set.seed(123)
cv_model_pls <- train(
  Sale_Price ~ .,
  data = ames_train,
  method = "pls",
```

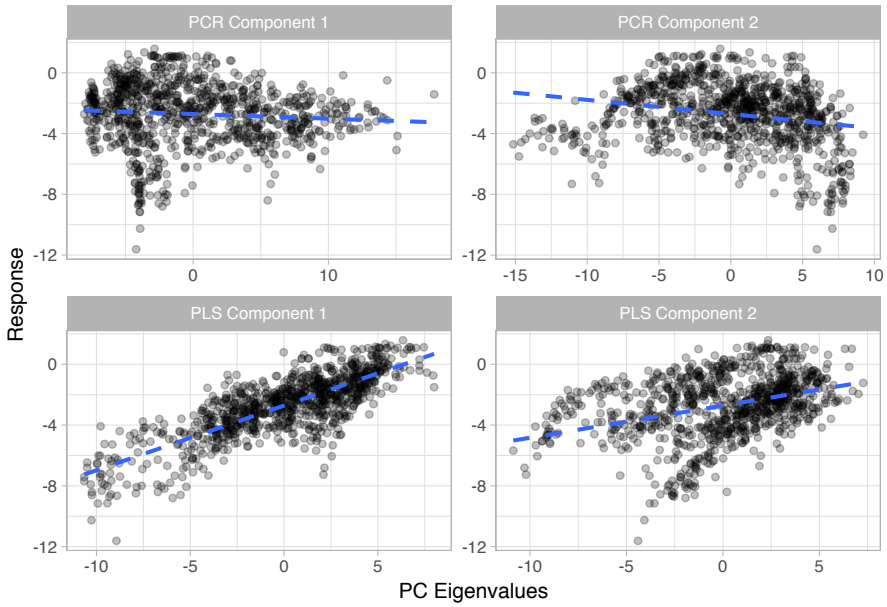

FIGURE 4.9: Illustration showing that the first two PCs when using PCR have very little relationship to the response variable (top row); however, the first two PCs when using PLS have a much stronger association to the response (bottom row).

```
  trControl = trainControl(method = "cv", number = 10),
  preProcess = c("zv", "center", "scale"),
  tuneLength = 20
)

# model with lowest RMSE
cv_model_pls$bestTune
##   ncomp
## 3     3

# plot cross-validated RMSE
ggplot(cv_model_pls)
```

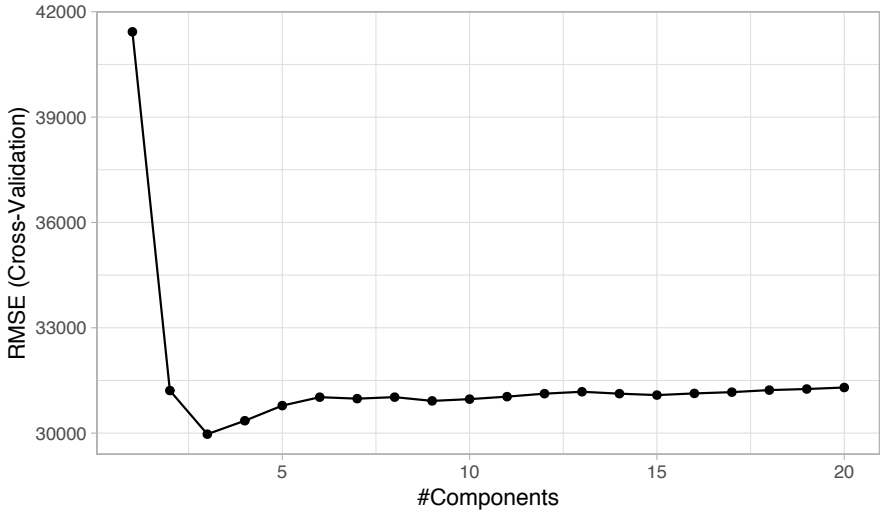

FIGURE 4.10: The 10-fold cross valdation RMSE obtained using PLS with 1-20 principal components.

4.8 Feature interpretation

Once we've found the model that minimizes the predictive accuracy, our next goal is to interpret the model structure. Linear regression models provide a very intuitive model structure as they assume a *monotonic linear relationship* between the predictor variables and the response. The *linear* relationship part of that statement just means, for a given predictor variable, it assumes for every one unit change in a given predictor variable there is a constant change in the response. As discussed earlier in the chapter, this constant rate of change is provided by the coefficient for a predictor. The *monotonic* relationship means that a given predictor variable will always have a positive or negative relationship. But how do we determine the most influential variables?

Variable importance seeks to identify those variables that are most influential in our model. For linear regression models, this is most often measured by the absolute value of the t-statistic for each model parameter used; though simple, the results can be hard to interpret when the model includes interaction effects and complex transformations (in Chapter **??** we'll discuss *model-agnostic* approaches that don't have this issue). For a PLS model, variable importance can be computed using the weighted sums of the absolute regression coefficients. The weights are a function of the reduction of the RSS across the number of PLS components and are computed separately for each outcome.

Therefore, the contribution of the coefficients are weighted proportionally to the reduction in the RSS.

We can use `vip::vip()` to extract and plot the most important variables. The importance measure is normalized from 100 (most important) to 0 (least important). Figure 4.11 illustrates that the top 4 most important variables are `Gr_liv_Area`, `First_Flr_SF`, `Total_Bsmt_SF`, and `Garage_Cars` respectively.

```
vip(cv_model_pls, num_features = 20, method = "model")
```

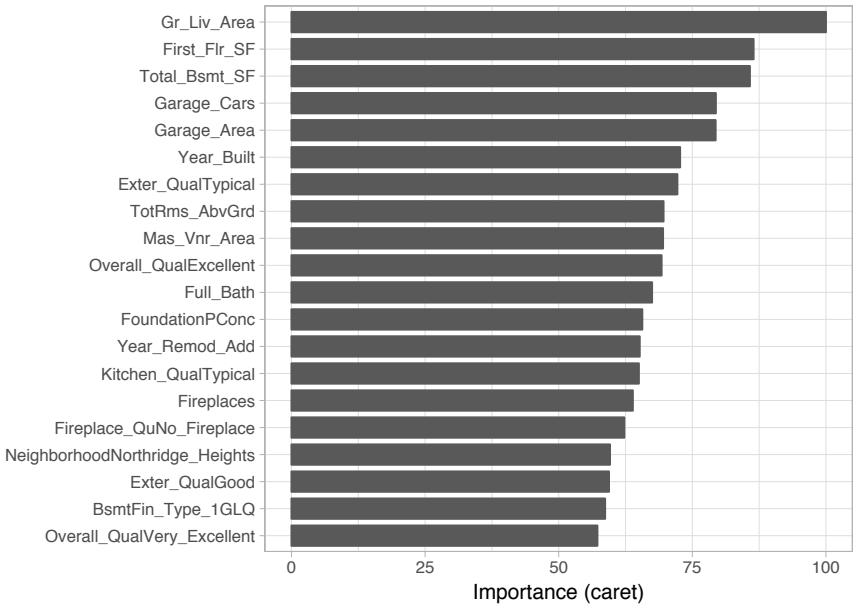

FIGURE 4.11: Top 20 most important variables for the PLS model.

As stated earlier, linear regression models assume a monotonic linear relationship. To illustrate this, we can construct partial dependence plots (PDPs). PDPs plot the change in the average predicted value ($\widehat{y}$) as specified feature(s) vary over their marginal distribution. As you will see in later chapters, PDPs become more useful when non-linear relationships are present (we discuss PDPs and other ML interpretation techniques in Chapter **??**). However, PDPs of linear models help illustrate how a fixed change in x_i relates to a fixed linear change in $\widehat{y}_i$ while taking into account the average effect of all the other features in the model (for linear models, the slope of the PDP is equal to the corresponding features LS coefficient).

The **pdp** package [@R-pdp] provides convenient functions for computing and plotting PDPs. For example, the following code chunk would plot the PDP for the `Gr_Liv_Area` predictor.

```
pdp::partial(cv_model_pls, "Gr_Liv_Area", grid.resolution =
20, plot = TRUE)
```

All four of the most important predictors have a positive relationship with sale price; however, we see that the slope ($\widehat{beta}_i$) is steepest for the most important predictor and gradually decreases for lessor important variables.

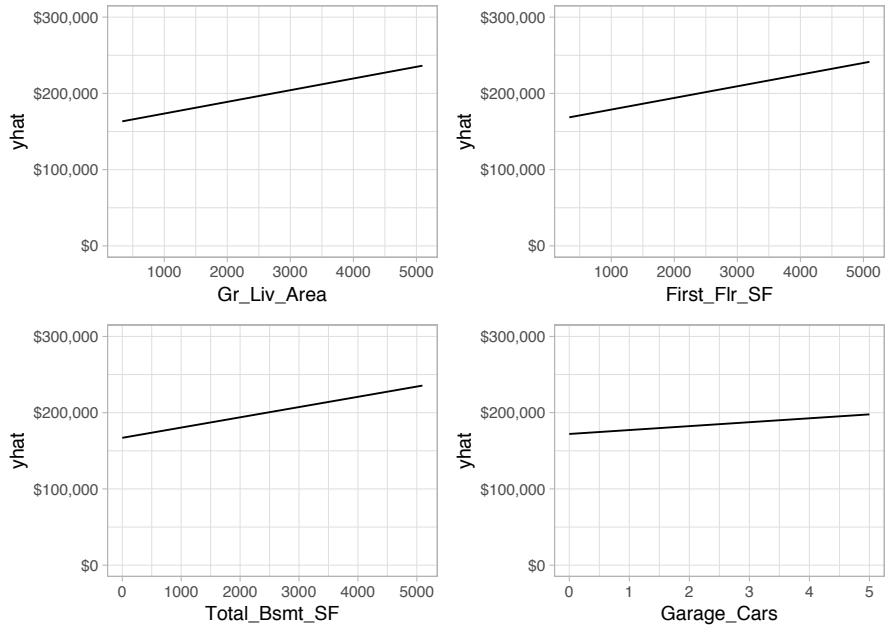

FIGURE 4.12: Partial dependence plots for the first four most important variables.

4.9 Final thoughts

Linear regression is usually the first supervised learning algorithm you will learn. The approach provides a solid fundamental understanding of the supervised learning task; however, as we've discussed there are several concerns

that result from the assumptions required. Although extensions of linear regression that integrate dimension reduction steps into the algorithm can help address some of the problems with linear regression, more advanced supervised algorithms typically provide greater flexibility and improved accuracy. Nonetheless, understanding linear regression provides a foundation that will serve you well in learning these more advanced methods.

Bibliography

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

Box, G. E. and Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 211–252.

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

Carroll, R. J. and Ruppert, D. (1981). On prediction and the power transformation family. *Biometrika*, 68(3):609–615.

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.

Chollet, F. and Allaire, J. J. (2018). *Deep Learning with R*. Manning Publications Company.

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.

Faraway, J. J. (2016). *Linear models with R*. Chapman and Hall/CRC.

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.

Gower, J. C. (1971). A general coefficient of similarity and some of its properties. *Biometrics*, pages 857–871.

Granitto, P. M., Furlanello, C., Biasioli, F., and Gasperi, F. (2006). Recursive feature elimination with random forest for ptr-ms analysis of agroindustrial products. *Chemometrics and Intelligent Laboratory Systems*, 83(2):83–90.

Guo, C. and Berkhahn, F. (2016). Entity embeddings of categorical variables. *arXiv preprint arXiv:1604.06737*.

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 Johnson, K. (2018). *AppliedPredictiveModeling: Functions and Data Sets for 'Applied Predictive Modeling'*. R package version 1.1-7.

Kuhn, M. and Johnson, K. (2019). *Feature Engineering and Selection: A Practical Approach for Predictive Models*. "Chapman & Hall/CRC".

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

Kursa, M. B., Rudnicki, W. R., et al. (2010). Feature selection with the boruta package. *J Stat Softw*, 36(11):1–13.

Kutner, M. H., Nachtsheim, C. J., Neter, J., and Li, W. (2005). *Applied Linear Statistical Models*. McGraw Hill, 5th edition.

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.

Little, R. J. and Rubin, D. B. (2014). *Statistical analysis with missing data*, volume 333. John Wiley & Sons.

Maldonado, S. and Weber, R. (2009). A wrapper method for feature selection using support vector machines. *Information Sciences*, 179(13):2208–2217.

Massy, W. F. (1965). Principal components regression in exploratory statistical research. *Journal of the American Statistical Association*, 60(309):234–256.

Micci-Barreca, D. (2001). A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems. *ACM SIGKDD Explorations Newsletter*, 3(1):27–32.

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

Saeys, Y., Inza, I., and Larrañaga, P. (2007). A review of feature selection techniques in bioinformatics. *bioinformatics*, 23(19):2507–2517.

Shah, A. D., Bartlett, J. W., Carpenter, J., Nicholas, O., and Hemingway, H. (2014). Comparison of random forest and parametric imputation models for imputing missing data using mice: a caliber study. *American journal of epidemiology*, 179(6):764–774.

Stekhoven, D. J. (2015). missforest: Nonparametric missing value imputation using random forest. *Astrophysics Source Code Library*.

Tierney, N. (2017). visdat: Visualising whole data frames. *JOSS*, 2(16):355.

West, B. T., Welch, K. B., and Galecki, A. T. (2014). *Linear mixed models: a practical guide using statistical software*. Chapman and Hall/CRC.

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.

Zheng, A. and Casari, A. (2018). *Feature Engineering for Machine Learning: Principles and Techniques for Data Scientists*. "O'Reilly Media, Inc.".

Zumel, N. and Mount, J. (2016). vtreat: a data. frame processor for predictive modeling. *arXiv preprint arXiv:1611.09477*.

Index