# Hands-On Machine Learning with R, 2ed

Brad Boehmke & Brandon Greenwell

2023-09-06

# Table of contents

**Welcome** **4**
Who should read this . . . 4
Why R . . . 5
Conventions used in this book . . . 5
Additional resources . . . 6
Acknowledgments . . . 6
Software information . . . 7

**Preface to the second edition** **11**

**1 Introduction to Machine Learning** **12**
1.1 Supervised learning . . . 12
1.1.1 Regression problems . . . 13
1.1.2 Classification problems . . . 13
1.1.3 Knowledge check . . . 16
1.2 Unsupervised learning . . . 16
1.2.1 Knowledge check . . . 17
1.3 Machine Learning in . . . 18
1.3.1 Knowledge check . . . 19
1.4 Roadmap . . . 19
1.5 Data sets . . . 20
1.6 Exercises . . . 20

**I Supervised Learning** **21**

**2 First model with Tidymodels** **22**
2.1 Prerequisites . . . 23
2.2 Data splitting . . . 23
2.2.1 Simple random sampling . . . 25
2.2.2 Stratified sampling . . . 26
2.2.3 Class imbalances . . . 27
2.2.4 Knowledge check . . . 28
2.3 Building models . . . 28
2.3.1 Knowledge check . . . 32

2.4 Making predictions . . . . . 32
2.4.1 Knowledge check . . . . . 34
2.5 Model evaluation . . . . . 34
2.5.1 Regression models . . . . . 35
2.5.2 Classification models . . . . . 37
2.5.3 Knowledge check . . . . . 42
2.6 Exercises . . . . . 42

**References** **43**

# Welcome

Welcome to the second edition of *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.

> **Note**
>
> This book is undergoing heavy restructuring and may be confusing or incomplete; however, we hope for the first draft to be completed by the fall of 2023. You can find the complete first edition at bradleyboehmke.github.io/HOML.

## 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.

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 (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 language, we would refer you to Advanced R (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 (Hastie, Tibshirani, and Friedman 2009), Computer Age Statistical Inference (Efron and Hastie 2016), Deep Learning (Goodfellow, Bengio, and Courville 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 the **tidymodels** ecosystem of packages for model development, **vip** and **pdp** for model interpretation, TODO (add others as we develop) 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

In this book we focus on implementing machine learning tasks with 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. We are not ignoring other languages such as Python or Julia because we think these tools are inferior. They're not! And in practice, most organizations and data science teams use a mix of languages. In fact, throughout this book we may reference methods or implementations in other languages and we may even provide a few examples in Python. However, we strongly believe that it's best to master one tool at a time, and R is a great place to start.

## 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:

> **Tip**
>
> Signifies a tip or suggestion

> **Note**
>
> Signifies a general note

> **Warning**
>
> 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://koalaverse.github.io/homlr/. 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.

## Acknowledgments

We'd like to thank everyone who contributed feedback, typo corrections, and discussions while the book was being written. GitHub contributors included @agailloty, @asimumba, @benprew, @bfgray3, @bragks, @cunningjames, @DesmondChoy, @erickeniuk, @j-ryanhart, @lcreteig, @liangwu82, @Lianta, @mccurcio, @mmelcher76, @MMonterosso89, @nsharkey, @raycblai, @schoonees, @tpristavec and @william3031. We'd also like to thank folks such as Alex Gutman, Greg Anderson, Jay Cunningham, Joe Keller, Mike Pane, Scott Crawford, and several other co-workers who provided great input around much of this machine learning content.

## Software information

This book was built with the following packages and R version. All code was executed on 2019 MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor, 16 GB of memory, 2667 MHz speed, and double data rate synchronous dynamic random access memory (DDR4).

```
# packages used
pkgs <- c(
    "modeldata",
    "tidymodels",
    "vip"
)

# package & session info
sessioninfo::session_info(pkgs)
#> - Session info ---------------------------------------------------------------
#>  setting  value
#>  version  R version 4.2.0 (2022-04-22)
#>  os       Ubuntu 22.04.3 LTS
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  C.UTF-8
#>  ctype    C.UTF-8
#>  tz       UTC
#>  date     2023-09-06
#>  pandoc   2.9.2.1 @ /usr/bin/ (via rmarkdown)
#>
#> - Packages -------------------------------------------------------------------
#>  package      * version    date (UTC) lib source
#>  backports      1.4.1      2021-12-13 [1] CRAN (R 4.2.0)
#>  broom          1.0.5      2023-06-09 [1] CRAN (R 4.2.0)
#>  cachem         1.0.8      2023-05-01 [1] CRAN (R 4.2.0)
#>  class          7.3-20     2022-01-16 [3] CRAN (R 4.2.0)
#>  cli            3.6.1      2023-03-23 [1] CRAN (R 4.2.0)
#>  clock          0.7.0      2023-05-15 [1] CRAN (R 4.2.0)
#>  codetools      0.2-18     2020-11-04 [3] CRAN (R 4.2.0)
#>  colorspace     2.1-0      2023-01-23 [1] CRAN (R 4.2.0)
#>  conflicted     1.2.0      2023-02-01 [1] CRAN (R 4.2.0)
#>  cpp11          0.4.6      2023-08-10 [1] CRAN (R 4.2.0)
#>  data.table     1.14.8     2023-02-17 [1] CRAN (R 4.2.0)
```

```
#>  diagram        1.6.5      2020-09-30 [1] CRAN (R 4.2.0)
#>  dials          1.2.0      2023-04-03 [1] CRAN (R 4.2.0)
#>  DiceDesign     1.9        2021-02-13 [1] CRAN (R 4.2.0)
#>  digest         0.6.33     2023-07-07 [1] CRAN (R 4.2.0)
#>  dplyr          1.1.3      2023-09-03 [1] CRAN (R 4.2.0)
#>  ellipsis       0.3.2      2021-04-29 [1] CRAN (R 4.2.0)
#>  fansi          1.0.4      2023-01-22 [1] CRAN (R 4.2.0)
#>  farver         2.1.1      2022-07-06 [1] CRAN (R 4.2.0)
#>  fastmap        1.1.1      2023-02-24 [1] CRAN (R 4.2.0)
#>  foreach        1.5.2      2022-02-02 [1] CRAN (R 4.2.0)
#>  furrr          0.3.1      2022-08-15 [1] CRAN (R 4.2.0)
#>  future         1.33.0     2023-07-01 [1] CRAN (R 4.2.0)
#>  future.apply   1.11.0     2023-05-21 [1] CRAN (R 4.2.0)
#>  generics       0.1.3      2022-07-05 [1] CRAN (R 4.2.0)
#>  ggplot2        3.4.3      2023-08-14 [1] CRAN (R 4.2.0)
#>  globals        0.16.2     2022-11-21 [1] CRAN (R 4.2.0)
#>  glue           1.6.2      2022-02-24 [1] CRAN (R 4.2.0)
#>  gower          1.0.1      2022-12-22 [1] CRAN (R 4.2.0)
#>  GPfit          1.0-8      2019-02-08 [1] CRAN (R 4.2.0)
#>  gtable         0.3.4      2023-08-21 [1] CRAN (R 4.2.0)
#>  hardhat        1.3.0      2023-03-30 [1] CRAN (R 4.2.0)
#>  infer          1.0.5      2023-09-06 [1] CRAN (R 4.2.0)
#>  ipred          0.9-14     2023-03-09 [1] CRAN (R 4.2.0)
#>  isoband        0.2.7      2022-12-20 [1] CRAN (R 4.2.0)
#>  iterators      1.0.14     2022-02-05 [1] CRAN (R 4.2.0)
#>  KernSmooth     2.23-20    2021-05-03 [3] CRAN (R 4.2.0)
#>  labeling       0.4.3      2023-08-29 [1] CRAN (R 4.2.0)
#>  lattice        0.20-45    2021-09-22 [3] CRAN (R 4.2.0)
#>  lava           1.7.2.1    2023-02-27 [1] CRAN (R 4.2.0)
#>  lhs            1.1.6      2022-12-17 [1] CRAN (R 4.2.0)
#>  lifecycle      1.0.3      2022-10-07 [1] CRAN (R 4.2.0)
#>  listenv        0.9.0      2022-12-16 [1] CRAN (R 4.2.0)
#>  lubridate      1.9.2      2023-02-10 [1] CRAN (R 4.2.0)
#>  magrittr       2.0.3      2022-03-30 [1] CRAN (R 4.2.0)
#>  MASS           7.3-56     2022-03-23 [3] CRAN (R 4.2.0)
#>  Matrix         1.4-1      2022-03-23 [3] CRAN (R 4.2.0)
#>  memoise        2.0.1      2021-11-26 [1] CRAN (R 4.2.0)
#>  mgcv           1.8-40     2022-03-29 [3] CRAN (R 4.2.0)
#>  modeldata      1.2.0      2023-08-09 [1] CRAN (R 4.2.0)
#>  modelenv       0.1.1      2023-03-08 [1] CRAN (R 4.2.0)
#>  munsell        0.5.0      2018-06-12 [1] CRAN (R 4.2.0)
```

```
#>  nlme           3.1-157    2022-03-25 [3] CRAN (R 4.2.0)
#>  nnet           7.3-17     2022-01-16 [3] CRAN (R 4.2.0)
#>  numDeriv       2016.8-1.1 2019-06-06 [1] CRAN (R 4.2.0)
#>  parallelly     1.36.0     2023-05-26 [1] CRAN (R 4.2.0)
#>  parsnip        1.1.1      2023-08-17 [1] CRAN (R 4.2.0)
#>  patchwork      1.1.3      2023-08-14 [1] CRAN (R 4.2.0)
#>  pillar         1.9.0      2023-03-22 [1] CRAN (R 4.2.0)
#>  pkgconfig      2.0.3      2019-09-22 [1] CRAN (R 4.2.0)
#>  prettyunits    1.1.1      2020-01-24 [1] CRAN (R 4.2.0)
#>  prodlim        2023.08.28 2023-08-28 [1] CRAN (R 4.2.0)
#>  progressr      0.14.0     2023-08-10 [1] CRAN (R 4.2.0)
#>  purrr          1.0.2      2023-08-10 [1] CRAN (R 4.2.0)
#>  R6             2.5.1      2021-08-19 [1] CRAN (R 4.2.0)
#>  RColorBrewer   1.1-3      2022-04-03 [1] CRAN (R 4.2.0)
#>  Rcpp           1.0.11     2023-07-06 [1] CRAN (R 4.2.0)
#>  recipes        1.0.8      2023-08-25 [1] CRAN (R 4.2.0)
#>  rlang          1.1.1      2023-04-28 [1] CRAN (R 4.2.0)
#>  rpart          4.1.16     2022-01-24 [3] CRAN (R 4.2.0)
#>  rsample        1.2.0      2023-08-23 [1] CRAN (R 4.2.0)
#>  rstudioapi     0.15.0     2023-07-07 [1] CRAN (R 4.2.0)
#>  scales         1.2.1      2022-08-20 [1] CRAN (R 4.2.0)
#>  shape          1.4.6      2021-05-19 [1] CRAN (R 4.2.0)
#>  slider         0.3.0      2022-11-16 [1] CRAN (R 4.2.0)
#>  SQUAREM        2021.1     2021-01-13 [1] CRAN (R 4.2.0)
#>  stringi        1.7.12     2023-01-11 [1] CRAN (R 4.2.0)
#>  stringr        1.5.0      2022-12-02 [1] CRAN (R 4.2.0)
#>  survival       3.3-1      2022-03-03 [3] CRAN (R 4.2.0)
#>  tibble         3.2.1      2023-03-20 [1] CRAN (R 4.2.0)
#>  tidymodels     1.1.1      2023-08-24 [1] CRAN (R 4.2.0)
#>  tidyr          1.3.0      2023-01-24 [1] CRAN (R 4.2.0)
#>  tidyselect     1.2.0      2022-10-10 [1] CRAN (R 4.2.0)
#>  timechange     0.2.0      2023-01-11 [1] CRAN (R 4.2.0)
#>  timeDate       4022.108   2023-01-07 [1] CRAN (R 4.2.0)
#>  tune           1.1.2      2023-08-23 [1] CRAN (R 4.2.0)
#>  tzdb           0.4.0      2023-05-12 [1] CRAN (R 4.2.0)
#>  utf8           1.2.3      2023-01-31 [1] CRAN (R 4.2.0)
#>  vctrs          0.6.3      2023-06-14 [1] CRAN (R 4.2.0)
#>  vip            0.4.1      2023-08-21 [1] CRAN (R 4.2.0)
#>  viridisLite    0.4.2      2023-05-02 [1] CRAN (R 4.2.0)
#>  warp           0.2.0      2020-10-21 [1] CRAN (R 4.2.0)
#>  withr          2.5.0      2022-03-03 [1] CRAN (R 4.2.0)
```

```
#>  workflows      1.1.3      2023-02-22 [1] CRAN (R 4.2.0)
#>  workflowsets   1.0.1      2023-04-06 [1] CRAN (R 4.2.0)
#>  yardstick      1.2.0      2023-04-21 [1] CRAN (R 4.2.0)
#>
#>  [1] /home/runner/work/_temp/Library
#>  [2] /opt/R/4.2.0/lib/R/site-library
#>  [3] /opt/R/4.2.0/lib/R/library
#>
#> ------------------------------------------------------------------------------
```

# Preface to the second edition

Welcome to the second edition of *Hands-On Machine Learning with R*! This is a major reworking of the first edition, removing material we no longer think is useful, adding material we wish we included in the first edition, and generally updating the text and code to reflect changes in best practices.

A brief summary of the biggest changes follows:

- TBD
- TBD
- TBD

# 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 and draw inferences from patterns in 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, 26:2), 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 has a defined learning task; they each intend to use attributes ($X$) to predict an outcome measurement ($Y$).

> **Note**
>
> Throughout this text we'll use various terms interchangeably for
>
> - $X$: "predictor variable", "independent variable", "attribute", "feature", "predictor"
> - $Y$: "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.

> **Note**
>
> 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.

### 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

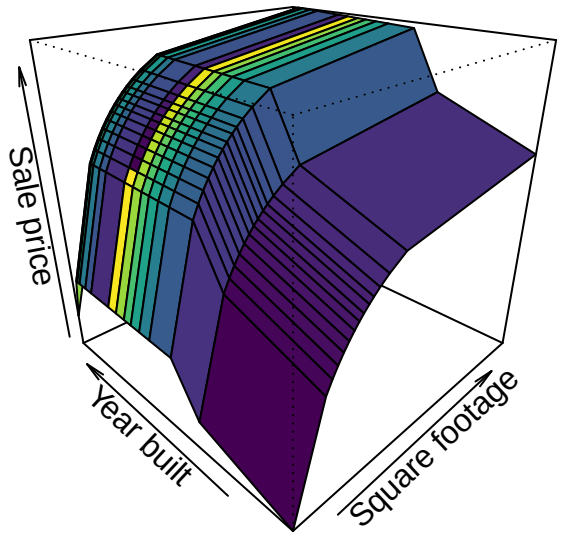



Figure 1.1: Average home sales price as a function of year built and total square footage.

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.

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.

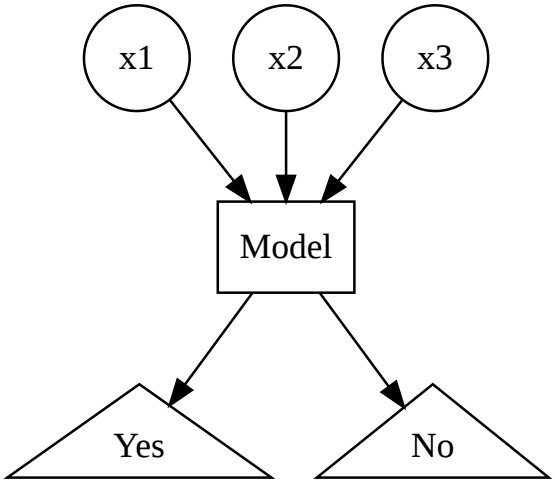



Figure 1.2: Classification problem modeling ‘Yes’/‘No’ response based on three features.

### 1.1.3 Knowledge check

Caution

Identify the features, response variable, and the type of supervised model required for the following tasks:

- There is an online retailer that wants to predict whether you will click on a certain featured product given your demographics, the current products in your online basket, and the time since your previous purchase.
- A bank wants to use a customers historical data such as the number of loans they've had, the time it took to payoff those loans, previous loan defaults, the number of new loans within the past two years, along with the customers income and level of education to determine if they should issue a new loan for a car.
- If the bank above does issue a new loan, they want to use the same information to determine the interest rate of the new loan issued.
- To better plan incoming and outgoing flights, an airline wants to use flight information such as scheduled flight time, day/month of year, number of passengers, airport departing from, airport arriving to, distance to travel, and weather warnings to determine if a flight will be delayed.
- What if the above airline wants to use the same information to predict the number of minutes a flight will arrive late or early?

## 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!

Despite 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 unsupervised learning models can be used as inputs to downstream supervised learning models.

### 1.2.1 Knowledge check

Caution

Identify the type of unsupervised model required for the following tasks:

- Say you have a YouTube channel. You may have a lot of data about the subscribers of your channel. What if you want to use that data to detect groups of similar subscribers?
- Say you'd like to group Ohio counties together based on the demographics of their residents.
- A retailer has collected hundreds of attributes about all their customers; however, many of those features are highly correlated. They'd like to reduce the number of features down by combining all those highly correlated features into groups.

## 1.3 Machine Learning in 

Historically, the R ecosystem provides a wide variety of ML algorithm implementations. This has its benefits; however, this also has drawbacks as it requires the users to learn many different formula interfaces and syntax nuances.

More recently, development on a group of packages called **Tidymodels** has helped to make implementation easier. The **tidymodels** collection allows you to perform discrete parts of the ML workflow with discrete packages:

- rsample for data splitting and resampling
- recipes for data pre-processing and feature engineering
- parsnip for applying algorithms
- tune for hyperparameter tuning
- yardstick for measuring model performance
- and several others!

Throughout this book you'll be exposed to several of these packages. Go ahead and make sure you have the following packages installed.

> **Note**
>
> The **tidymodels** package is a meta package, or a package of packages, that will install several packages that exist in the **tidymodels** ecosystem.

```
# data wrangling
install.packages(c("here", "tidyverse"))

# modeling
install.packages("tidymodels")

# model interpretability
install.packages(c("pdp", "vip"))
```

```
packageVersion("tidymodels")
## [1] '1.1.1'

library(tidymodels)
## -- Attaching packages ---------------------------------------- tidymodels 1.1.1 --
## v broom        1.0.5     v rsample      1.2.0
## v dials        1.2.0     v tibble       3.2.1
## v dplyr        1.1.3     v tidyr        1.3.0
```

```
## v infer        1.0.5     v tune         1.1.2
## v modeldata    1.2.0     v workflows    1.1.3
## v parsnip      1.1.1     v workflowsets 1.0.1
## v purrr        1.0.2     v yardstick    1.2.0
## v recipes      1.0.8
## -- Conflicts --------------------------------------- tidymodels_conflicts() --
## x purrr::discard() masks scales::discard()
## x dplyr::filter()  masks plotly::filter(), stats::filter()
## x dplyr::lag()     masks stats::lag()
## x recipes::step()  masks stats::step()
## * Search for functions across packages at https://www.tidymodels.org/find/
```

### 1.3.1 Knowledge check

> **Caution**
>
> Check out the Tidymodels website: https://www.tidymodels.org/. Identify which packages can be used for:
>
> 1. Efficiently splitting your data
> 2. Optimizing hyperparameters
> 3. Measuring the effectiveness of your model
> 4. Working with correlation matrices

## 1.4 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 ...

> **⚠ TODO**
>
> Fill out roadmap as we progress

## 1.5 Data sets

TBD

## 1.6 Exercises

1. Identify four real-life applications of supervised and unsupervised problems.
    - Explain what makes these problems supervised versus unsupervised.
    - For each problem identify the target variable (if applicable) and potential features.
2. Identify and contrast a regression problem with a classification problem.
    - What is the target variable in each problem and why would being able to accurately predict this target be beneficial to society?
    - What are potential features and where could you collect this information?
    - What is determining if the problem is a regression or a classification problem?
3. Identify three open source data sets suitable for machine learning (e.g., https://bit.ly/35wKu5c).
    - Explain the type of machine learning models that could be constructed from the data (e.g., supervised versus unsupervised and regression versus classification).
    - What are the dimensions of the data?
    - Is there a code book that explains who collected the data, why it was originally collected, and what each variable represents?
    - If the data set is suitable for supervised learning, which variable(s) could be considered as a useful target? Which variable(s) could be considered as features?
4. Identify examples of misuse of machine learning in society. What was the ethical concern?

# Part I

# Supervised Learning

# 2 First model with Tidymodels

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

Approaching ML modeling correctly means approaching it strategically by spending our data wisely on learning and validation procedures, properly pre-processing the feature and target variables, minimizing *data leakage*, 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.

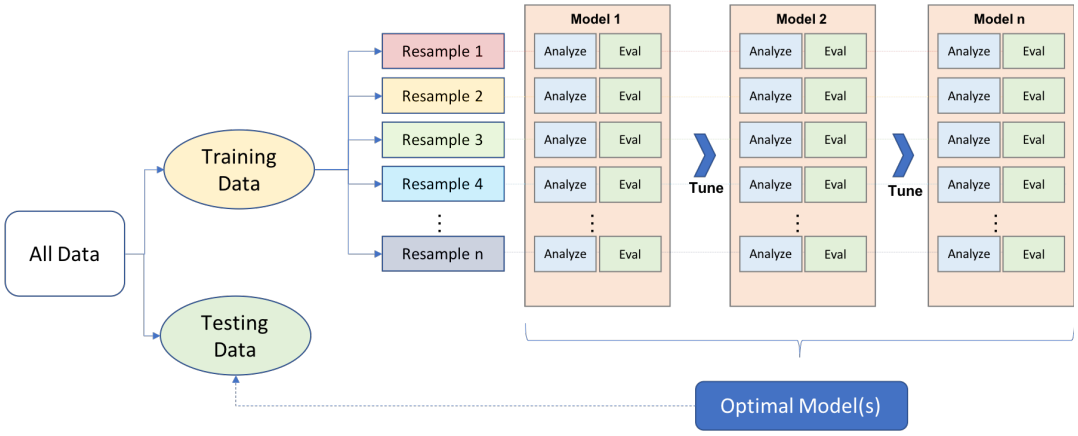



Figure 2.1: General predictive machine learning process.

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. More specifically, this chapter is designed to get you acquainted with building predictive models using the Tidymodels construct. We'll focus on the process of splitting our data for improved generalizability, using Tidymodel's parsnip package for constructing

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

our models, along with yardstick to measure model performance. Future chapters will build upon these concepts by focusing on other parts of the machine learning process illustrated above such as applying resampling procedures to give you a more robust assessment of model performance and performing hyperparameter tuning to control the complexity of machine learning algorithms.

## 2.1 Prerequisites

This chapter leverages the following packages.

```
# Helper packages
library(tidyverse)  # for data manipulation & plotting

# Modeling process packages
library(modeldata)  # for accessing data
library(tidymodels) # for modeling procedures
```

To illustrate some of the concepts, we'll use the Ames Housing and employee attrition data sets introduced in Section 1.5.

```
# Ames housing data
ames <- modeldata::ames

# Job attrition data
attrition <- modeldata::attrition %>%
   mutate(Attrition = fct_relevel(Attrition, "Yes"))
```

## 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*.

> ⚠ Warning
>
> 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.

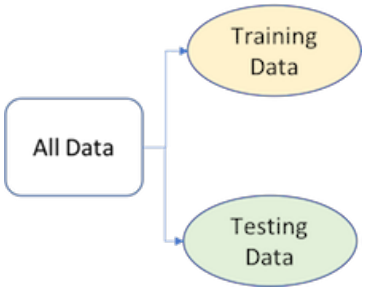



Figure 2.2: Splitting data into training and test sets..

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 of your response variable ($Y$).

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

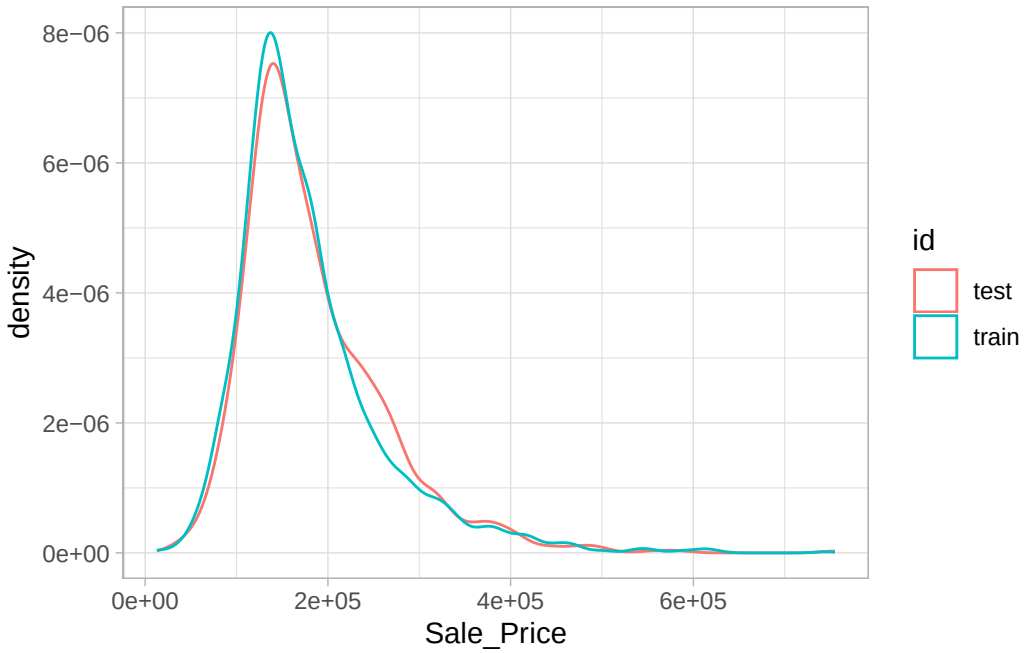
```
# create train/test split
set.seed(123)  # for reproducibility
split  <- initial_split(ames, prop = 0.7)
train  <- training(split)
test   <- testing(split)

# dimensions of training data
dim(train)
## [1] 2051   74

# dimensions of test data
dim(test)
## [1] 879  74
```

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.

```
train %>%
  mutate(id = 'train') %>%
  bind_rows(test %>% mutate(id = 'test')) %>%
  ggplot(aes(Sale_Price, color = id)) +
  geom_density()
```




### 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.

To perform stratified sampling we simply apply the `strata` argument in `initial_split`.

```
set.seed(123)
split_strat <- initial_split(attrition, prop = 0.7, strata = "Attrition")
train_strat <- training(split_strat)
test_strat  <- testing(split_strat)
```

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.

```
# original response distribution
table(attrition$Attrition) %>% prop.table()
##
##       Yes        No
## 0.1612245 0.8387755

# response distribution for training data
table(train_strat$Attrition) %>% prop.table()
##
##       Yes        No
## 0.1605058 0.8394942

# response distribution for test data
table(test_strat$Attrition) %>% prop.table()
##
##       Yes        No
## 0.1628959 0.8371041
```

Tip

There is very little downside to using stratified sampling so when trying to decide if you should use random sampling versus stratified sampling, error on the side of safety with stratified sampling.

### 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 abun-

dant samples, new rare samples are generated by using repetition or bootstrapping (described further in **?@sec-bootstrapping**).

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 with the **themis** package[2], which provides additional sampling procedures on top of the **rsample** package.

### 2.2.4 Knowledge check

Caution

1. Import the penguins data from the **modeldata** package
2. Create a 70-30 stratified train-test split (`species` is the target variable).
3. What are the response variable proportions for the train and test data sets?

## 2.3 Building models

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. 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.

Fortunately, the tidymodels ecosystem simplifies this and, in particular, the **parsnip** package[3] provides one common interface to train many different models supplied by other packages. Consequently, we'll focus on building models the tidymodels way.

To create and fit a model with parsnip we follow 3 steps:

1. Create a model type
2. Choose an "engine"
3. Fit our model

[2]https://themis.tidymodels.org
[3]https://parsnip.tidymodels.org

Let's illustrate by building a linear regression model. For our first model we will simply use two features from our training data - total square feet of the home (`Gr_Liv_Area`) and year built (`Year_Built`) to predict the sale price (`Sale_Price`).

> **Tip**
>
> We can use `tidy()` to get results of our model's parameter estimates and their statistical properties. Although the `summary()` function can provide this output, it gives the results back in an unwieldy format. Go ahead, and run `summary(lm_ols)` to compare the results to what we see below.
> Many models have a `tidy()` method that provides the summary results in a more predictable and useful format (e.g. a data frame with standard column names)

```
lm_ols <- linear_reg() %>%
   fit(Sale_Price ~ Gr_Liv_Area + Year_Built, data = train)

tidy(lm_ols)
## # A tibble: 3 x 5
##   term          estimate std.error statistic   p.value
##   <chr>            <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -2157423.    69234.       -31.2 8.09e-175
## 2 Gr_Liv_Area       94.4      2.12       44.4 2.54e-302
## 3 Year_Built      1114.      35.5        31.4 5.30e-177
```

> **Note**
>
> Don't worry about what these parameters mean at this point; we'll cover these details in a future chapter.

Now, you may have noticed that we only applied two of the three steps mentioned previously:

1. Create a model type
2. ~~Choose an "engine"~~
3. Fit our model

The reason is because most model objects (`linear_reg()` in this example) have a default engine. `linear_reg()` by default uses `stats::lm` for ordinary least squares.[4] But we can always change the engine. For example, say you wanted to use **keras** to perform gradient descent linear regression, then you could change the engine to **keras** but use the same code workflow.

[4] `lm()` is the built in function provided by R to perform ordinary least squares regression. You can learn more about it by checking out the help docs with `?lm`.

> **Warning**
>
> For this code to run successfully on your end you need to have the **keras** and **tensorflow** packages installed on your machine. Depending on your current setup this could be an easy process or you could run into problems. If you run into problems don't fret, this is primarily just to illustrate how we can change engines.

```
lm_sgd <- linear_reg() %>%
   set_engine('keras') %>%
   fit(Sale_Price ~ Gr_Liv_Area + Year_Built, data = train)
## Epoch 1/20
## 65/65 - 1s - loss: 39843188736.0000 - 526ms/epoch - 8ms/step
## Epoch 2/20
## 65/65 - 0s - loss: 39655796736.0000 - 103ms/epoch - 2ms/step
## Epoch 3/20
## 65/65 - 0s - loss: 39483834368.0000 - 81ms/epoch - 1ms/step
## Epoch 4/20
## 65/65 - 0s - loss: 39327260672.0000 - 84ms/epoch - 1ms/step
## Epoch 5/20
## 65/65 - 0s - loss: 39184064512.0000 - 80ms/epoch - 1ms/step
## Epoch 6/20
## 65/65 - 0s - loss: 39052566528.0000 - 82ms/epoch - 1ms/step
## Epoch 7/20
## 65/65 - 0s - loss: 38930870272.0000 - 80ms/epoch - 1ms/step
## Epoch 8/20
## 65/65 - 0s - loss: 38818603008.0000 - 82ms/epoch - 1ms/step
## Epoch 9/20
## 65/65 - 0s - loss: 38714707968.0000 - 79ms/epoch - 1ms/step
## Epoch 10/20
## 65/65 - 0s - loss: 38615781376.0000 - 81ms/epoch - 1ms/step
## Epoch 11/20
## 65/65 - 0s - loss: 38520942592.0000 - 81ms/epoch - 1ms/step
## Epoch 12/20
## 65/65 - 0s - loss: 38426435584.0000 - 81ms/epoch - 1ms/step
## Epoch 13/20
## 65/65 - 0s - loss: 38330400768.0000 - 82ms/epoch - 1ms/step
## Epoch 14/20
## 65/65 - 0s - loss: 38228209664.0000 - 80ms/epoch - 1ms/step
## Epoch 15/20
## 65/65 - 0s - loss: 38116585472.0000 - 80ms/epoch - 1ms/step
## Epoch 16/20
```

```
## 65/65 - 0s - loss: 37991170048.0000 - 84ms/epoch - 1ms/step
## Epoch 17/20
## 65/65 - 0s - loss: 37846781952.0000 - 86ms/epoch - 1ms/step
## Epoch 18/20
## 65/65 - 0s - loss: 37679575040.0000 - 79ms/epoch - 1ms/step
## Epoch 19/20
## 65/65 - 0s - loss: 37486362624.0000 - 85ms/epoch - 1ms/step
## Epoch 20/20
## 65/65 - 0s - loss: 37263183872.0000 - 86ms/epoch - 1ms/step
```

> **Tip**
>
> When we talk about 'engines' we're really just referring to packages that provide the desired algorithm. Each model object has different engines available to use and they are all documented. For example check out the help file for `linear_reg` (`?linear_reg`) and you'll see the different engines available (lm, brulee, glm, glmnet, etc.)

The beauty of this workflow is that if we want to explore different models we can simply change the model object. For example, say we wanted to run a K-nearest neighbor model. We can just use `nearest_neighbor()`.

In this example we have pretty much the same code as above except we added the line of code `set_mode()`. This is because most algorithms require you to specify if you are building a regression model or a classification model.

> **Note**
>
> When you run this code you'll probably get an error message saying that *"This engine requires some package installs: 'kknn'."* This just means you need to `install.packages('kknn')` and then you should be able to successfully run this code.

```
knn <- nearest_neighbor() %>%
   set_engine("kknn") %>%
   set_mode("regression") %>%
   fit(Sale_Price ~ Gr_Liv_Area + Year_Built, data = train)
```

> **Tip**
>
> You can see all the different model objects available at https://parsnip.tidymodels.org/reference/index.html

### 2.3.1 Knowledge check

Caution

1. If you haven't already done so, create a 70-30 stratified train-test split on the `attrition` data (note: `Attrition` is the response variable).
2. Using the `logistic_reg()` model object, fit a model using `Age`, `DistanceFromHome`, and `JobLevel` as the features.
3. Now train a K-nearest neighbor model using the 'kknn' engine and be sure to set the mode to be a classification model.

## 2.4 Making predictions

We have fit a few different models. Now, if we want to see our predictions we can simply apply `predict()` and feed it the data set we want to make predictions on. Here, we can see the predictions made on our training data for our ordinary least square linear regression model.

```
lm_ols %>% predict(train)
## # A tibble: 2,051 x 1
##      .pred
##      <dbl>
##  1 217657.
##  2 214276.
##  3 223425.
##  4 260324.
##  5 109338.
##  6 195106.
##  7 222217.
##  8 126175.
##  9  98550.
## 10 120811.
## # i 2,041 more rows
```

And here we get the predicted values for our KNN model.

```
knn %>% predict(train)
## # A tibble: 2,051 x 1
##      .pred
##      <dbl>
##  1 194967.
```

```
##  2 192240
##  3 174220
##  4 269760
##  5 113617.
##  6 173672
##  7 174820
##  8 120796
##  9 114560
## 10 121346
## # i 2,041 more rows
```

A similar process can be applied to make predictions for a classification model. For example, the following trains a classification model that predicts whether an employee will attrit based on their age. When we make predictions, the output is the predicted class (employee attrition is `Yes` or `No`).

```
simple_logit <- logistic_reg() %>%
   fit(Attrition ~ Age, data = train_strat)

simple_logit %>% predict(train_strat)
## # A tibble: 1,028 x 1
##    .pred_class
##    <fct>
##  1 No
##  2 No
##  3 No
##  4 No
##  5 No
##  6 No
##  7 No
##  8 No
##  9 No
## 10 No
## # i 1,018 more rows
```

In general, machine learning classifiers don't just give binary predictions, but instead provide some numerical value between 0 and 1 for their predictions. This number, sometimes called the model score or confidence, is a way for the model to express their certainty about what class the input data belongs to. In most applications, the exact probability is ignored and we use a threshold (typically $\geq 0.5$) to round the score to a binary answer, yes or no, employee attrition or not attrition. But in some cases we do want the prediction probabilities and we can get those by adding `type = "prob"` to our `predict` call.

```
simple_logit %>% predict(train_strat, type = "prob")
## # A tibble: 1,028 x 2
##    .pred_Yes .pred_No
##        <dbl>    <dbl>
##  1    0.178     0.822
##  2    0.0485    0.952
##  3    0.204     0.796
##  4    0.155     0.845
##  5    0.162     0.838
##  6    0.213     0.787
##  7    0.195     0.805
##  8    0.213     0.787
##  9    0.0664    0.934
## 10    0.141     0.859
## # i 1,018 more rows
```

### 2.4.1 Knowledge check

Caution

1. Using the logistic regression model you trained in the previous exercise, make predictions on the attrition training data.
2. Now make predictions using the K-nearest neighbor model.

## 2.5 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 output of a loss function is often referred to as the *error* or pseudo *residual*).

If we look at our predicted outputs for our ordinary least squares model, we can see that the predicted home value (`.pred`) was $149,091 for the first observation and the actual home value was $172,000, resulting in an error of nearly $23,000. The objective of the loss function is to aggregate the prediction errors for all the observations into a meaningful single value metric.

```
lm_ols %>%
  predict(test) %>%
  bind_cols(test %>% select(Sale_Price)) %>%
  mutate(prediction_error = Sale_Price - .pred)
## # A tibble: 879 x 3
##      .pred Sale_Price prediction_error
##      <dbl>      <int>            <dbl>
##  1 149091.     172000           22909.
##  2 219596.     195500          -24096.
##  3 195491.     212000           16509.
##  4  97418.     141000           43582.
##  5 152195.     170000           17805.
##  6 134471.     142000            7529.
##  7 119697.     115000           -4697.
##  8 195517.     184000          -11517.
##  9 141210.      88000          -53210.
## 10 239057.     306000           66943.
## # i 869 more rows
```

There are many loss functions to choose from 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.

### 2.5.1 Regression models

The most common loss functions for regression models include:

- **MSE**: Mean squared error is the average of the squared error ($MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$)[5]. The squared component results in larger errors having larger penalties. **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 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**

[5]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 $n$.

- $R^2$: 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**

Let's compute the RMSE of our OLS regression model. Remember, we want to assess our model's performance on the test data not the training data since that gives us a better idea of how our model generalizes. To do so, the following:

1. Makes predictions with our test data,
2. Adds the actual `Sale_Price` values from our test data,
3. Computes the RMSE.

```
lm_ols %>%
   predict(test) %>%
   bind_cols(test %>% select(Sale_Price)) %>%
   rmse(truth = Sale_Price, estimate = .pred)
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      45445.
```

The RMSE value suggests that, on average, our model mispredicts the expected sale price of a home by about $45K.

Other common loss functions for regression models include:

- **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 compares 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. [6] **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**

[6] 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.

- **RMSLE**: Root mean squared logarithmic error. Similar 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 response 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**

### 2.5.2 Classification models

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).

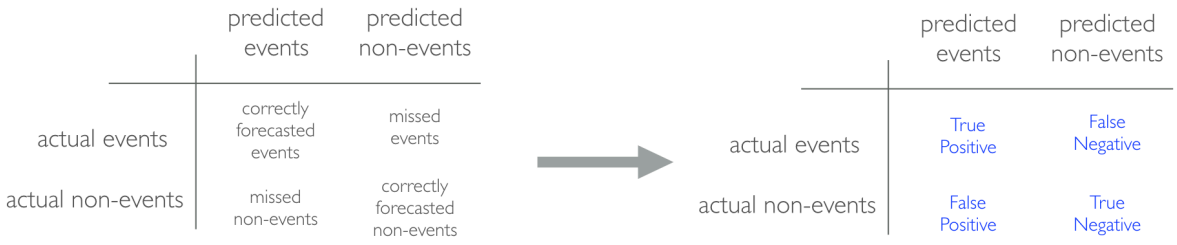



Figure 2.3: Confusion matrix and relationships to terms such as true-positive and false-negative.

Let's go ahead and create a logistic regression classification model with the attrition data.

> **Tip**
>
> In R, using a "." as in `Attrition ~ .` is a shortcut for saying use all available features to predict `Attrition`.

```
logit <- logistic_reg() %>%
   fit(Attrition ~ ., data = train_strat)
```

We can use `conf_mat()` to view the confusion matrix for this model. In essence, this confusion matrix shows that our model has 34 true positive predictions, 353 true negative predictions,

17 false negative predictions, and 38 false predictions.

```
logit %>%
   predict(test_strat) %>%
   bind_cols(test_strat %>% select(Attrition)) %>%
   conf_mat(truth = Attrition, estimate = .pred_class, dnn = c("Truth", "Prediction"))
##      Prediction
## Truth Yes  No
##   Yes  34  17
##   No   38 353
```

> **Note**
>
> Depending on the software and libraries used, you may see the prediction summaries on the rows and the actual value summaries in the columns or vice versa. `conf_mat` allows us to control that with the `dnn` argument to control the table dimension names.

This confusion matrix allows us to extract different levels of performance for our classification model. For example, we can assess:

- **Accuracy**: Overall, how often is the classifier correct? Accuracy is the proportion of the data that are predicted correctly. Example: $\frac{TP+TN}{total} = \frac{34+353}{442} = 0.867$. **Objective: maximize**
- **Precision**: How accurately does the classifier predict events (or positive 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? This characterizes the "purity in retrieval performance" (Buckland and Gey 1994). Example: $\frac{TP}{TP+FP} = \frac{34}{34+17} = 0.667$. **Objective: maximize**
- **Sensitivity (aka recall)**: How accurately does the classifier classify actual events? The sensitivity is defined as the proportion of positive results out of the number of samples which were actually positive. 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: $\frac{TP}{TP+FN} = \frac{34}{34+38} = 0.472$. **Objective: maximize**
- **Specificity**: How accurately does the classifier classify actual non-events? The specificity measures the proportion of negatives that are correctly identified as negatives. Example: $\frac{TN}{TN+FP} = \frac{353}{353+17} = 0.954$. **Objective: maximize**

```
predict_and_actuals <- logit %>%
   predict(test_strat) %>%
   bind_cols(test_strat %>% select(Attrition))
```

```
# accuracy
predict_and_actuals %>% accuracy(truth = Attrition, estimate = .pred_class)
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.876

# precision
predict_and_actuals %>% precision(truth = Attrition, estimate = .pred_class)
## # A tibble: 1 x 3
##   .metric   .estimator .estimate
##   <chr>     <chr>          <dbl>
## 1 precision binary         0.667

# recall
predict_and_actuals %>% sensitivity(truth = Attrition, estimate = .pred_class)
## # A tibble: 1 x 3
##   .metric     .estimator .estimate
##   <chr>       <chr>          <dbl>
## 1 sensitivity binary         0.472

# specificity
predict_and_actuals %>% specificity(truth = Attrition, estimate = .pred_class)
## # A tibble: 1 x 3
##   .metric     .estimator .estimate
##   <chr>       <chr>          <dbl>
## 1 specificity binary         0.954
```

Our results show that our model has high accuracy, which is mainly driven by our model's ability to predict non-events (employees that do not attrit) accurately. However, our model does not do a very good job of predicting positive events (employees that do attrit), represented by the low precision and sensitivity values.

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 ***receiver operator curve*** (ROC) that plots the sensitivity on the y-axis and 1-specificity on the x-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.

To plot the ROC curve we actually need to predict the probability of our classification model's prediction. We then pass the predicted probabilities for the class we care about (here we are concerned with the probability of employees actually attriting) and the truth values to

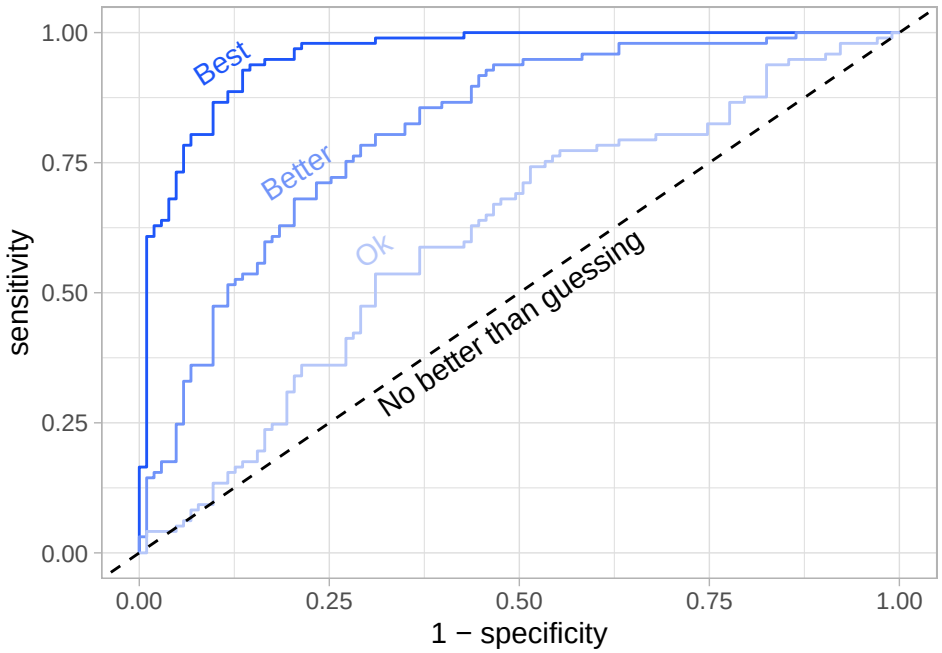
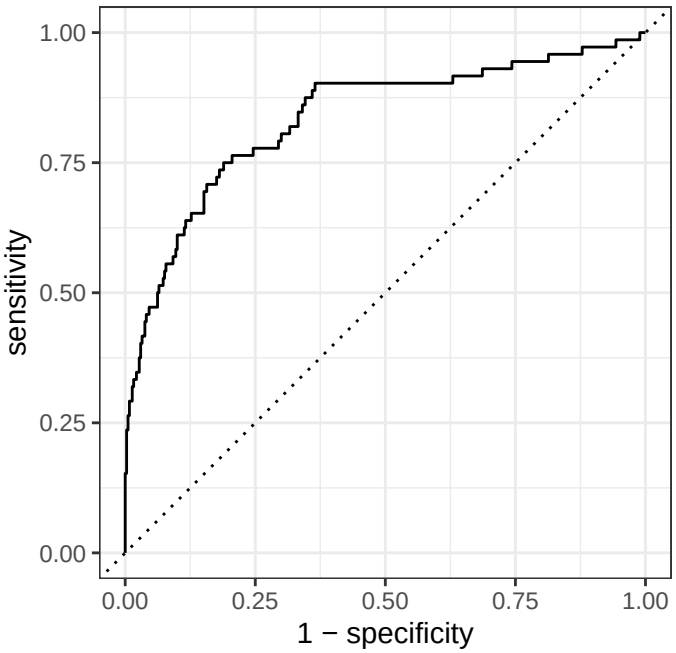



Figure 2.4: ROC curve.

`roc_curve`.

```
logit %>%
   predict(test_strat, type = "prob") %>%
   bind_cols(test_strat %>% select(Attrition)) %>%
   roc_curve(truth = Attrition, .pred_Yes) %>%
   autoplot()
```




Another common metric is the ***area under the curve*** (AUC). Generally, an ROC AUC value is between 0.5 and 1, with 1 being a perfect prediction model. If your value is between 0 and 0.5, then this implies that you have meaningful information in your model, but it is being applied incorrectly because doing the opposite of what the model predicts would result in an AUC $> 0.5$. The benefit of the AUC metric is that it gives us a single metric value that incorporates both sensitivity and specificity of our model. The higher the AUC value, the more balanced our model is.

```
logit %>%
   predict(test_strat, type = "prob") %>%
   bind_cols(test_strat %>% select(Attrition)) %>%
   roc_auc(truth = Attrition, .pred_Yes)
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 roc_auc binary         0.835
```

### 2.5.3 Knowledge check

Caution

1. Compute and compare the $R^2$ of the `lm_ols` and `knn` models trained in Section 2.3.
2. Now compute the accuracy rate and AUC of the `simple_logit` model trained in Section 2.3 and compare it to the `logit` model trained in Section 2.5.2.

## 2.6 Exercises

Caution

For this exercise use the Chicago ridership data set provided by the modeldata library.[7]. This data set is derived from Kuhn and Johnson (2019) and contains an abbreviated training set for modeling the number of people (in thousands) who enter the Clark and Lake L station. The objective is to use the available features (i.e. `temp` (temparature), `wind` (wind speed), `Bulls_Home` (is there a Chicago Bulls game at home), etc. to predict the the number of people (in thousands) represented by the `ridership` column.
Modeling tasks:

1. Load the Chicago ridership data set and remove the `date` column.
2. Split the data into a training set and test set using a 70-30% split.
3. How many observations are in the training set and test set?
4. Compare the distribution of `ridership` between the training set and test set.
5. Fit a linear regression model using all available features to predict `ridership` and compute the RMSE on the test data.
6. Fit a K-nearest neighbor model that uses all available features to predict `ridership` and compute the RMSE on the test data.
7. How do these models compare?

[7]See more details at https://modeldata.tidymodels.org/reference/Chicago.html

# References


Breiman, Leo et al. 2001. "Statistical Modeling: The Two Cultures (with Comments and a Rejoinder by the Author)." *Statistical Science* 16 (3): 199–231.

Buckland, Michael, and Fredric Gey. 1994. "The Relationship Between Recall and Precision." *Journal of the American Society for Information Science* 45 (1): 12–19.

Chawla, Nitesh V, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. "SMOTE: Synthetic Minority over-Sampling Technique." *Journal of Artificial Intelligence Research* 16: 321–57.

Efron, Bradley, and Trevor Hastie. 2016. *Computer Age Statistical Inference*. Vol. 5. Cambridge University Press.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. Vol. 1. MIT Press Cambridge.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Vol. 2. Springer Science+ Business Media.

Kuhn, Max, and Kjell Johnson. 2013. *Applied Predictive Modeling*. Vol. 26. Springer.

———. 2019. *Feature Engineering and Selection: A Practical Approach for Predictive Models*. Chapman; Hall/CRC.

Wickham, Hadley. 2014. *Advanced r*. CRC Press.

Wickham, Hadley, and Garrett Grolemund. 2016. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, Inc.

Wolpert, David H. 1996. "The Lack of a Priori Distinctions Between Learning Algorithms." *Neural Computation* 8 (7): 1341–90.