Hands-On Machine Learning with R, 2ed

Brad Boehmke & Brandon Greenwell

2024-02-17

Table of contents

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

> ℹ 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

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     2024-02-17
#>  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            3.6.2     2023-12-11 [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          0.4.7     2023-12-02 [1] CRAN (R 4.2.0)
#>  data.table     1.15.0    2024-01-30 [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     1.10      2023-12-07 [1] CRAN (R 4.2.0)
#> digest         0.6.34    2024-01-11 [1] CRAN (R 4.2.0)
#> dplyr          1.1.4     2023-11-17 [1] CRAN (R 4.2.0)
#> ellipsis       0.3.2     2021-04-29 [1] CRAN (R 4.2.0)
#> fansi          1.0.6     2023-12-08 [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         1.33.1    2023-12-22 [1] CRAN (R 4.2.0)
#> future.apply   1.11.1    2023-12-21 [1] CRAN (R 4.2.0)
#> generics       0.1.3     2022-07-05 [1] CRAN (R 4.2.0)
#> ggplot2        3.4.4     2023-10-12 [1] CRAN (R 4.2.0)
#> globals        0.16.2    2022-11-21 [1] CRAN (R 4.2.0)
#> glue           1.7.0     2024-01-09 [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        1.3.1     2024-02-02 [1] CRAN (R 4.2.0)
#> infer          1.0.6     2024-01-31 [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           1.7.3     2023-11-04 [1] CRAN (R 4.2.0)
#> lhs            1.1.6     2022-12-17 [1] CRAN (R 4.2.0)
#> lifecycle      1.0.4     2023-11-07 [1] CRAN (R 4.2.0)
#> listenv        0.9.1     2024-01-29 [1] CRAN (R 4.2.0)
#> lubridate      1.9.3     2023-09-27 [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      1.3.0     2024-01-21 [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    1.37.0     2024-02-14 [1] CRAN (R 4.2.0)
#> parsnip       1.2.0      2024-02-16 [1] CRAN (R 4.2.0)
#> patchwork     1.2.0      2024-01-08 [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   1.2.0      2023-09-24 [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          1.0.12     2024-01-09 [1] CRAN (R 4.2.0)
#> recipes       1.0.9      2023-12-13 [1] CRAN (R 4.2.0)
#> rlang         1.1.3      2024-01-10 [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        1.3.0      2023-11-28 [1] CRAN (R 4.2.0)
#> shape         1.4.6      2021-05-19 [1] CRAN (R 4.2.0)
#> slider        0.3.1      2023-10-12 [1] CRAN (R 4.2.0)
#> SQUAREM       2021.1     2021-01-13 [1] CRAN (R 4.2.0)
#> stringi       1.8.3      2023-12-11 [1] CRAN (R 4.2.0)
#> stringr       1.5.1      2023-11-14 [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         1.3.1      2024-01-24 [1] CRAN (R 4.2.0)
#> tidyselect    1.2.0      2022-10-10 [1] CRAN (R 4.2.0)
#> timechange    0.3.0      2024-01-18 [1] CRAN (R 4.2.0)
#> timeDate      4032.109   2023-12-14 [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          1.2.4      2023-10-22 [1] CRAN (R 4.2.0)
#> vctrs         0.6.5      2023-12-01 [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          0.2.1      2023-11-02 [1] CRAN (R 4.2.0)
#> withr         3.0.0      2024-01-16 [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     1.3.0      2024-01-19 [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. There's no shortage of definitions for the term machine learning. For the purposes of this book, we can think of it as a blended field with a focus on using algorithms to help *learn from data.* This process of learning from data results in a model, which we can use to make predictions.

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.
- Predicting songs to recommend to listeners on a music app.
- Predicting the estimated travel time to drive from your home to work.
- Predicting a segmentation that a customer aligns to based on common attributes or purchasing behavior for targeted marketing.
- Predicting coupon redemption rates for a given marketing campaign.
- Predicting customer (or employee) 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 how they learn to make predictions and the four main groups of learners are:

- Supervised learning
- Unsupervised learning
- Reinforcement learning
- Generative artificial intelligence (AI)

Which type you will need to use depends on the learning task you hope to accomplish; and the primary focus of this book is on the first two groups of learners - supervised and unsupervised learning.

1.1 Supervised learning

Supervised learning is a set of ML learners that learn the relationship between inputs (often referred to as features or predictors) and output(s) (often referred to as the target variable). Or, as stated by Kuhn and Johnson (2013, 26:2), supervised learning is "…the process of developing a mathematical tool or model that generates an accurate prediction." The learning algorithm in a supervised learning 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 the following:

- Using customer attributes to predict the probability of the customer churning in the next 6 weeks.
- Using various home attributes to predict the sales price.
- Using employee attributes to predict the likelihood of attrition within the next six months.
- Using patient attributes and symptoms to predict the risk of being readmitted to the hospital within 30 days after release.
- Using product attributes to predict time to market.
- Using weather conditions and relevant historical information to predict the number of bikes that will be rented out on a given day.

Each of these examples has a defined learning task; they each intend to use various features (X) to predict a well-defined target (Y). Think about the hospital readmission example. Predicting the liklihood of readmittance is not specific enough and will make pulling together relevant data a challenge, so we need to think carefully aabout how we define the features and response for modeling. Defining the target as *whether or not a patient was readmitted within 30 days after release* is something that can easily be measured, assuming it's relevant to the stakeholders.

The scenarios listed above are examples of 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.

> **i** 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, called *hyperparameters*.

Most supervised learning problems can be bucketed into one of two general categories, *regression* or *classification*, depending on the type of response variable. We'll briefly discuss both cases over the next two sections.

> **ℹ Note**
>
> Throughout this text we'll use various terms and notation interchangeably. In particular,
>
> - we'll use X to denote a feature, predictor, or attribute (we may even use the more classic term *independent variable*);
> - bold notation may be used to denote a set of features, for example $X = (X_1, X_2)$, where, in the case of the predicting sale price example, X_1 might represent square footage and X_2 represent the overall quality of the home;
> - we'll use y when referring to response or target variable (again, we may sometimes use the more classic term *dependent variable*).

1.1.1 Regression problems

When the objective is to predict a quantitative outcome, we generally refer to this as a *regression problem* (not to be confused with linear regression modeling, which is a special case). Regression problems revolve around numeric output where both order and distance matters (e.g., sales or a discrete count, like the number of bike rentals in a given day). In the examples above, predicting home sale prices based home attributes is a regression problem because the output is ordered and values closer to each other are closer in nature (e.g., the closer two sale prices are to each other the more similar the homes are in terms of sale value).

Figure 1.1 shows a regression model's predicted sale price of homes in Ames, Iowa (from 2006–2010) as a function of two attributes: year built and total square footage. Depending on the combination of these two features, the expected home sales price could fall anywhere along the surface.

See Table 1.1 for a few more examples of regression models:

Table 1.1: Example regression problems

Scenario	Potential features	Numeric prediction
Predict home prices	Square footage, zip code, number of bedrooms and bathrooms, lot size, mortgage interest rate, property tax rate, construction costs, and number of homes for sale in the area.	The home price in dollars.
Predict ride time	Historical traffic conditions (gathered from smartphones, traffic sensors, ride-hailing and other navigation applications), distance from destination, and weather conditions.	The time in minutes and seconds to arrive at a destination.

Scenario	Potential features	Numeric prediction
Predict loan interest rate	Customer credit score, number of loans outstanding, historical repayment history, size of loan requested, current inflation and treasury rates.	The interest rate to be applied to a loan.

We'll learn various ways of building such a model in Part II of this book (TODO: cross-reference "Part II").

1.1.2 Classification problems

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

- Predicting if a customer will redeem a coupon (coded as yes/no or 1/0)?
- Predicting if a customer will churn (coded as yes/no or 1/0)?
- Predicting if a customer will click on our online ad (coded as yes/no or 1/0)?
- Predicting if a customer review is:
  - Binary: positive vs. negative.
  - Multinomial: extremely negative to extremely positive on a 0–5 Likert scale.

However, when we apply ML models to classification problems, rather than predict a particular class (i.e., "yes" or "no"), we often want to predict the *conditional probability* of a particular class (i.e., yes: 0.65, no: 0.35).[1] By default, the class with the highest predicted probability becomes the predicted class (called a classification). Consequently, even though we classify it as a classification problem (pun intended), we're often still predicting a numeric output (i.e., a probability). However, the nature of the target variable is what makes it a classification problem.

Table 1.2 illustrates some example classification predictions where the model predicts the conditional probability of "Yes" and "No" classes. A threshold of 0.5 probability is used to determine if the predicted class is "Yes" or "No".

Table 1.2: Example classification predictions

Predicted "Yes" probability	Predicted "No" probability	Predicted class
0.65	0.35	Yes
0.15	0.85	No

---

[1]Conditional on the set of input feature values.

Predicted "Yes" probability	Predicted "No" probability	Predicted class
0.43	0.57	No
0.72	0.28	Yes

Throughout this book we will commonly use the term classification for brevity; however, the distinction between predicting the probability of an output and classifying that prediction into a particular class is important and should not be overlooked. Frank Harrell's discussion on classification versus prediction (Harrell 2017) is an excellent read to delve deeper into this distinction, along with why and when we should be focusing on probability prediction over classification and vice versa.

Although there are ML algorithms that can be applied to regression problems but not classification and vice versa, most of the supervised learning algorithms we'll cover in this book can be applied to both. These algorithms have become some of the most popular machine learning models in recent years (often driven by their availability in both proprietary and open-source software).

1.1.3 Knowledge check

> **Caution**
>
> Identify the features, response variable, and the type of supervised model (regression or classification) 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 them 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?

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

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, dividing consumers into different homogeneous groups, a process known as market segmentation.

Clustering differs from classification because the categories aren't defined by you. For example, Figure 1.3 shows how an unsupervised model might cluster a weather dataset based on temperature, revealing segmentations that define the seasons. You would then have to name those clusters based on your understanding of the dataset.

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, a situation known as *multicollinearity*.[2] Some dimension reduction techniques can be used to reduce the feature set to a potentially smaller set of uncorrelated

---

[2]To be fair, and as we'll see later in the book, the interpretation of most fitted ML models becomes problematic in the presence of correlated or (otherwise dependent) fetures.

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

(a) Data containing similar weather patterns.

(b) Clusters of weather patterns labeled as snow, sleet, rain, and no rain.

Figure 1.3: Clustering weather patterns which we would label the clusters based on our understanding of the data.

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 (e.g., linear regression), then it is possible to check our work by seeing how well our model predicts the response y on new observations not used in fitting the model. However, in unsupervised learning, there's 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 Reinforcement learning

Reinforcement learning (RL) refers to a family of algorithms that learn to make predictions by getting rewards or penalties based on actions performed within an environment. A reinforcement learning system generates a policy that defines the best strategy for getting the most rewards.

This best strategy is learned through interactions with the environment and observations of how it responds. In the absence of a supervisor, the learner must independently discover the sequence of actions that maximize the reward. This discovery process is akin to a trial-and-error search. The quality of actions is measured by not just the immediate reward they return, but also the delayed reward they might fetch. As it can learn the actions that result in eventual success in an unseen environment without the help of a supervisor, reinforcement learning is a very powerful algorithm.

A few examples of RL include:

- **Robotics**. Robots with pre-programmed behavior are useful in structured environments, such as the assembly line of an automobile manufacturing plant, where the task is repetitive in nature. However, in the unpredictable real world, where the interaction between a robot's actions and the environment is uncertain, achieving precise pre-programmed actions becomes exceedingly challenging. In such situations, Reinforcement Learning (RL) offers an effective approach to develop versatile robots. RL has demonstrated success in the context of robotic path planning, where robots need to autonomously discover optimal, obstacle-free, and dynamically compatible paths between two locations.

- **AlphaGo**. Go, a Chinese board game dating back 3,000 years, stands out as one of the most intricate strategic games known. Its complexity is attributed to the staggering number of possible board configurations, estimated at 10^270, surpassing the complexity of chess by several orders of magnitude. In 2016, AlphaGo, an artificial intelligence agent based on Reinforcement Learning (RL), achieved victory against the world's top human Go player. Much like a human player, AlphaGo learned through experience, engaging in thousands of games against professional opponents. Notably, the most recent RL-based Go agent possesses the unique ability to enhance its skills by playing against itself, a capability not available to human players.
- **Autonomous Driving**. An autonomous driving system faces the challenge of executing numerous perception and planning functions within an environment characterized by uncertainty. Reinforcement Learning (RL) is employed in various specific tasks, including vehicle path planning and motion prediction. Vehicle path planning entails the use of multiple low and high-level policies to make decisions that span different temporal and spatial scales. On the other hand, motion prediction involves the anticipation of pedestrian and other vehicle movements, thereby enabling an understanding of how the current environmental state might evolve.

Reinforcement learning as field has existed for quite some time but it really gained mainstream popularity after DeepMind's break through application of Q-learning to play Atari games (Mnih et al. 2013). Since then reinforcement learning has experienced several major breakthroughs and gains in popularity. Unfortunately, this topic is beyond the scope of this book but we recommend Sutton and Barto (2018), Powell (2021), and Szepesvári (2022) to learn more about RL.

1.4 Generative AI

Generative AI represents a category of models that generate content based on user input. It possesses the capability to generate various forms of content, including original images, musical compositions, and humorous jokes. Additionally, it can summarize articles, provide instructions for performing tasks, or even enhance and modify photographs.

Generative AI is versatile in its ability to work with a wide range of inputs and produce diverse outputs, encompassing text, images, audio, and video. Furthermore, it can handle combinations of these input and output types. For instance, a model can take an image as input and generate both an image and accompanying text as output, or it can accept input in the form of an image and text and produce a video as the output.

We can discuss generative models by their inputs and outputs, typically written as "type of input"-to-"type of output." For example, the following is a partial list of some inputs and outputs for generative models:

- Text-to-text

- Text-to-image
- Text-to-video
- Text-to-code
- Text-to-speech
- Image and text-to-image

Table 1.3 lists example use-cases of generative models.

Table 1.3: Examples of generative models, their input, and an example of their possible output.

Model	Input	Example output
Text-to-text	Who came up with the michelin star rating for restaurants?	The Michelin star rating system for restaurants was introduced by the French tire company Michelin. The system was first developed in the early 20th century as a means to promote travel and, by extension, the use of automobiles, in order to increase the demand for tires. Michelin began publishing restaurant and hotel guides in 1900, and in 1926, they started awarding stars to restaurants in recognition of their culinary excellence. The Michelin Guide is highly regarded in the culinary world, and Michelin stars have become a prestigious mark of quality for restaurants. [source: ChatGPT]
Text-to-image	A brain riding a rocketship heading towards the moon.	[source: Imagen]
Text-to-code	Write a hello world statement in R	`print("Hello, World!")` [source: Bard]

Model	Input	Example output
Image-to-text		This is a flamingo. They are found in the Caribbean. [Source: Google DeepMind]

How does generative AI work? At a high-level, generative models learn patterns in data with the goal to produce new but similar data. To produce unique and creative outputs, generative models are initially trained using an unsupervised approach, where the model learns to mimic the data it's trained on. The model is sometimes trained further using supervised or reinforcement learning on specific data related to tasks the model might be asked to perform, for example, summarize an article or edit a photo.

Generative AI is a quickly evolving technology with new use cases constantly being discovered. For example, generative models are helping businesses refine their ecommerce product images by automatically removing distracting backgrounds or improving the quality of low-resolution images.

Although this book does not delve into generative AI directly, the deep learning chapters do provide the foundation that many generative AI models are built upon.

1.5 Machine learning in R

The ML open-source ecosystem is a vibrant and rapidly evolving collection of software tools, libraries, frameworks, and platforms that are made freely available to the public for building, training, and deploying ML. This ecosystem has played a crucial role in democratizing ML and making ML accessible to a wide range of data scientists, researchers, and organizations.

Although this ecosystem expands multiple programming languages, our focus will predominately be with the R programming language. 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).

This book will expose you to many of the R packages and algorithms that perform and scale best to the kinds of data and problems encountered by most organizations while also showing you how to use implementations that provide more consistency.

For example, 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!

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

Throughout this book you'll be exposed to several of these packages and more. Moreover, in some cases, ML algorithms are available in one language but not another. As data scientists, we need to be comfortable in finding alternative solutions to those available in our native programming language of choice. Consequently, we may even provide examples of implementations using other languages such as Python or Julia.

Prior to moving on, let's take the time to make sure you have the required packages installed.

> **⚠ TODO**
>
> Once book is complete provide link to DESCRIPTION file or alternative approach for an easy way for readers to install all requirements. Maybe discuss `renv`??

```r
# 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        1.1.4     v tidyr        1.3.1
## v infer        1.0.6     v tune         1.1.2
## v modeldata    1.3.0     v workflows    1.1.3
## v parsnip      1.2.0     v workflowsets 1.0.1
## v purrr        1.0.2     v yardstick    1.3.0
## v recipes      1.0.9
## -- 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/
```

1.5.1 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

1.6 Roadmap

The goal of this book is to provide effective methods and 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

assessing model performance. These concepts will be discussed more thoroughly in Chapters
...

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

1.7 Data sets

> ⚠ **TODO**
>
> Revisit as we progress

1.8 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 ML (e.g., https://bit.ly/35wKu5c).

   - Explain the type of ML 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 ML 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 1.7.

```
# 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).

29

- **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).

> **i** 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
```

> ℹ 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`.

```r
lm_sgd <- linear_reg() %>%
   set_engine('keras') %>%
   fit(Sale_Price ~ Gr_Liv_Area + Year_Built, data = train)
## Epoch 1/20
## 65/65 - 0s - loss: 39842058240.0000 - 383ms/epoch - 6ms/step
## Epoch 2/20
## 65/65 - 0s - loss: 39654916096.0000 - 71ms/epoch - 1ms/step
## Epoch 3/20
## 65/65 - 0s - loss: 39484092416.0000 - 57ms/epoch - 876us/step
## Epoch 4/20
## 65/65 - 0s - loss: 39326892032.0000 - 55ms/epoch - 851us/step
## Epoch 5/20
## 65/65 - 0s - loss: 39183216640.0000 - 57ms/epoch - 874us/step
## Epoch 6/20
## 65/65 - 0s - loss: 39050776576.0000 - 56ms/epoch - 865us/step
## Epoch 7/20
## 65/65 - 0s - loss: 38929813504.0000 - 56ms/epoch - 865us/step
## Epoch 8/20
## 65/65 - 0s - loss: 38818160640.0000 - 55ms/epoch - 853us/step
## Epoch 9/20
## 65/65 - 0s - loss: 38713389056.0000 - 56ms/epoch - 864us/step
## Epoch 10/20
## 65/65 - 0s - loss: 38614712320.0000 - 56ms/epoch - 861us/step
## Epoch 11/20
## 65/65 - 0s - loss: 38519271424.0000 - 56ms/epoch - 863us/step
## Epoch 12/20
## 65/65 - 0s - loss: 38424731648.0000 - 55ms/epoch - 851us/step
## Epoch 13/20
## 65/65 - 0s - loss: 38327779328.0000 - 54ms/epoch - 835us/step
## Epoch 14/20
## 65/65 - 0s - loss: 38225723392.0000 - 57ms/epoch - 875us/step
## Epoch 15/20
## 65/65 - 0s - loss: 38113865728.0000 - 56ms/epoch - 857us/step
## Epoch 16/20
## 65/65 - 0s - loss: 37987278848.0000 - 55ms/epoch - 852us/step
```

```
## Epoch 17/20
## 65/65 - 0s - loss: 37840420864.0000 - 54ms/epoch - 834us/step
## Epoch 18/20
## 65/65 - 0s - loss: 37670699008.0000 - 56ms/epoch - 854us/step
## Epoch 19/20
## 65/65 - 0s - loss: 37475581952.0000 - 56ms/epoch - 855us/step
## Epoch 20/20
## 65/65 - 0s - loss: 37250654208.0000 - 56ms/epoch - 860us/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.

> ℹ 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

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 ≥ 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
```

> **ℹ 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 speci-
  ficity 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

Figure 2.4: ROC curve.

are concerned with the probability of employees actually attriting) and the truth values to `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

3 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.

> **i** Note
>
> This chapter introduces linear regression with an emphasis on prediction, rather than inference.
>
> 1. **Modeling for inference**: When you want to explicitly describe and quantify the relationship between the outcome variable Y and a set of explanatory variables X, determine the significance of any relationships, have measures summarizing these relationships, and possibly identify any causal relationships between the variables.
> 2. **Modeling for prediction**: When you want to predict an outcome variable Y based on the information contained in a set of predictor variables X. Unlike modeling for explanation, however, you don't care so much about understanding how all the variables relate and interact with one another, but rather only whether you can make good predictions about Y using the information in X.
>
> 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 and Lipovetsky (2020) for a great introduction to linear regression for inference and explanation rather than prediction.

3.1 Prerequisites

This chapter leverages the following packages:

```
# Data wrangling & visualization packages
library(tidyverse)

# Modeling packages
library(tidymodels)
```

We'll also continue working with the Ames housing data:

```
# stratified sampling with the rsample package
ames <- AmesHousing::make_ames()

set.seed(123)
split  <- initial_split(ames, prop = 0.7, strata = "Sale_Price")
ames_train  <- training(split)
ames_test   <- testing(split)
```

3.2 Correlation

Correlation is a single-number statistic that measures the extent that two variables are related ("co-related") to one another. For example, say we want to understand the relationship between the total above ground living space of a home (`Gr_Liv_Area`) and the home's sale price (`Sale_Price`).

Looking at the following scatter plot we can see that some relationship does exist. It appears that as `Gr_Liv_Area` increases the `Sale_Price` of a home increases as well.

```
ggplot(ames_train, aes(Gr_Liv_Area, Sale_Price)) +
   geom_point(size = 1.5, alpha = .25)
```

Correlation allows us to quantify this relationship. We can compute the correlation with the following:

```
ames_train %>%
    summarize(correlation = cor(Gr_Liv_Area, Sale_Price))
## # A tibble: 1 x 1
##   correlation
##         <dbl>
## 1       0.708
```

The value of a correlation coefficient will always vary between +1 and -1. In our example, the correlation coefficient is 0.71. When the value of the correlation coefficient is +1 or -1, then it is said to be a perfect degree of association between the two variables (near +1 implies a strong positive association and near -1 implies a strong negative association). This simply means that when there is a one unit change in one variable we will always see a certain change change in units in the other variable. As the correlation coefficient nears 0, the relationship between the two variables weakens with a near 0 value implying no association between the two variables (a one unit change in one variable has no relationship to any level of change in the other variable).

So, in our case we could say we have a moderate positive correlation between `Gr_Liv_Area` and `Sale_Price`. Let's look at another relationship. In the following we look at the relationship between the unfinished basement square footage of homes (`Bsmt_Unf_SF`) and the `Sale_Price`.

```
ggplot(ames_train, aes(Bsmt_Unf_SF, Sale_Price)) +
   geom_point(size = 1.5, alpha = .25)
```


In this example, we don't see much of a relationship. Basically, as `Bsmt_Unf_SF` gets larger or smaller, we really don't see a strong pattern with `Sale_Price`.

If we look at the correlation for this relationship, we see that the correlation coefficient is much closer to zero than to 1. This confirms our visual assessment that there does not seem to be much of a relationship between these two variables.

```
ames_train %>%
   summarize(correlation = cor(Bsmt_Unf_SF, Sale_Price))
## # A tibble: 1 x 1
##   correlation
##         <dbl>
## 1       0.186
```

Although a useful measure, correlation can be hard to imagine exactly what the association is between two variables based on this single statistic. Moreover, its important to realize that correlation typically assumes a **linear** relationship between two variables.

For example, let's check out the **anscombe** data, which is a built-in data set provided in R. If we look at each `x` and `y` relationship visually, we can see significant differences:

```
p1 <- qplot(x = x1, y = y1, data = anscombe)
p2 <- qplot(x = x2, y = y2, data = anscombe)
p3 <- qplot(x = x3, y = y3, data = anscombe)
p4 <- qplot(x = x4, y = y4, data = anscombe)

gridExtra::grid.arrange(p1, p2, p3, p4, ncol = 2)
```


However, if we compute the correlation between each of these relationships we see that they all have nearly equal correlation coefficients!

> ⚠️ **Warning**
>
> Never take a correlation coefficient at face value! You should always compare the visual relationship with the computed correlation value.

```
anscombe %>%
  summarize(
    corr_x1_y1 = cor(x1, y1),
    corr_x2_y2 = cor(x2, y2),
    corr_x3_y3 = cor(x3, y3),
    corr_x4_y4 = cor(x4, y4)
    )
##   corr_x1_y1 corr_x2_y2 corr_x3_y3 corr_x4_y4
## 1  0.8164205  0.8162365  0.8162867  0.8165214
```

There are actually several different ways to measure correlation. The most common, and the one we've been using here, is Pearson's correlation, represented by r_{xy}. Given paired data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ consisting of n pairs, r_{xy} is defined as

$$r_{XY} = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

where X_i and Y_i represent the individual sample points and $\bar{X}$ and $\bar{Y}$ are the sample mean. There are alternative methods that allow us to loosen some assumptions such as assuming a linear relationship; however, in the next section we'll see how simple linear regression fully characterizes this Pearson's correlation.

3.2.1 Knowledge check

> **Caution**
>
> 1. Interpreting coefficients that are not close to the extreme values of -1, 0, and 1 can be somewhat subjective. To help develop your sense of correlation coefficients, we suggest you play the 80s-style video game called, "Guess the Correlation", at http://guessthecorrelation.com/
> 2. Using the `ames_train` data, visualize the relationship between `Year_Built` and `Sale_Price`.
> 3. Guess what the correlation is between these two variables?
> 4. Now compute the correlation between these two variables.

3.3 Simple linear regression

As discussed in the last section, correlation is often used to quantify the strength of the linear association between two continuous variables. However, this statistic alone does not provide

us with a lot of actionable insights. But we can build on the concept of correlation to provide us with more useful information.

In this section, we seek to fully characterize the linear relationship we measured with correlation using a method called *simple linear regression* (SLR).

3.3.1 Best fit line

Let's go back to our plot illustrating the relationship between `Gr_Liv_Area` and `Sale_Price`. We can characterize this relationship with a linear line that we consider is the "best-fitting" line (we'll define "best-fitting" in a little bit). We do this by adding overplotting with `geom_smooth(method = "lm", se = FALSE)`

```
ggplot(ames_train, aes(Gr_Liv_Area, Sale_Price)) +
   geom_point(size = 1.5, alpha = .25) +
   geom_smooth(method = "lm", se = FALSE)
```


The line in the above plot is called a "regression line." The regression line is a visual summary of the linear relationship between two numerical variables, in our case the outcome variable `Sale_Price` and the explanatory variable `Gr_Liv_Area`. The positive slope of the blue line is consistent with our earlier observed correlation coefficient of 0.71 suggesting that there is a positive relationship between these two variables.

Mathematically, we can express this regression line as

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i, \quad \text{for } i = 1, 2, \dots, n, \tag{3.1}$$

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 distributed with mean zero and constant variance σ^2, denoted $\stackrel{iid}{\sim} (0, \sigma^2)$. Since the random errors are centered around zero (i.e., $E(\epsilon) = 0$), linear regression is really a problem of estimating a *conditional mean*:

$$E(Y_i|X_i) = \beta_0 + \beta_1 X_i.$$

For brevity, we often drop the conditional piece and write $E(Y|X) = E(Y)$. Consequently, the interpretation of the coefficients is 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 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*).

So what are the coefficients of our best fit line that characterizes the relationship between `Gr_Liv_Area` and `Sale_Price`? We can get that by fitting an SLR model where `Sale_Price` is our response variable and `Gr_Liv_Area` is our single predictor variable.

Once our model is fit we can extract our fitted model results with `tidy()`:

```
model1 <- linear_reg() %>%
   fit(Sale_Price ~ Gr_Liv_Area, data = ames_train)

tidy(model1)
## # A tibble: 2 x 5
##   term         estimate std.error statistic   p.value
##   <chr>           <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)   15938.     3852.      4.14 3.65e-  5
## 2 Gr_Liv_Area     110.      2.42     45.3  5.17e-311
```

The estimated coefficients from our model are $b_0 = 15938.17$ and $b_1 = 109.67$. To interpret, we estimate that the mean selling price increases by 109.67 for each additional one square foot of above ground living space.

With these coefficients, we can look at our scatter plot again (this time with the x & y axes formatted) and compare the characterization of our regression line with the coefficients. 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.

```
ggplot(ames_train, aes(Gr_Liv_Area, Sale_Price)) +
  geom_point(size = 1.5, alpha = .25) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_x_continuous(labels = scales::comma) +
  scale_y_continuous(labels = scales::dollar)
```


3.3.2 Estimation

This is great but you may still be asking how we are estimating the coefficients? Ideally, we want estimates of b_0 and b_1 that give us the "best fitting" line represented in the previous plot. 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).

Before we mathematically define RSS, let's first define what a residual is. Let's look at a single home. This home has 3,608 square feet of living space and sold for $475,000. In other words, $x = 3608$ and $y = 475000$.

```
## # A tibble: 1 x 2
##   Gr_Liv_Area Sale_Price
##         <int>      <int>
## 1        3608     475000
```

Based on our linear regression model (or the intercept and slope we identified from our model) our best fit line estimates that this house's sale price is

$$\widehat{Y_i} = b_0 + b_1 \times X_i = 15938.1733 + 109.6675 \times 3608 = 411618.5$$

We can visualize this in our plot where we have the actual `Sale_Price` (orange) and the estimated `Sale_Price` based on our fitted line. The difference between these two values $(Y_i - \widehat{Y_i} = 475000 - 411618.5 = 63381.5)$ is what we call our residual. It is considered the error for this observation, which we can visualize with the red line.

Now, if we look across all our data points you will see that each one has a residual associated with it. In the right plot, the vertical lines represent the individual residuals/errors associated with each observation.

Figure 3.1: The least squares fit from regressing sale price on living space for the the Ames housing data. Left: Fitted regression line. Right: Fitted regression line with vertical grey bars representing the residuals.

The OLS criterion identifies the "best fitting" line that minimizes the sum of squares of these residuals. Mathematically, this is computed by taking the sum of the squared residuals (or as stated before the residual sum of squares –> "RSS").

$$RSS = \sum_{i=1}^{n} \left(Y_i - \widehat{Y_i} \right)^2$$

where Y_i and $\widehat{Y_i}$ just mean the actual and predicted response values for the ith observation.

One drawback of the LS procedure in linear regression is that it only provides estimates of the coefficients; 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) Section 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):

$$\hat{\sigma}^2 = \frac{1}{n-p} \sum_{i=1}^{n} r_i^2, \tag{3.2}$$

where $r_i = \left(Y_i - \widehat{Y_i}\right)$ is referred to as the ith residual (i.e., the difference between the ith observed and predicted response value). The quantity $\hat{\sigma}^2$ is also referred to as the *mean square error* (MSE) and its square root is denoted RMSE (see Section Section 2.5.1 for discussion on these metrics).

For our SLR model, we can extract the RMSE metric and others using the `glance()` function:

```
glance(model1) %>%
    select(sigma) %>%
    mutate(RMSE = sigma, MSE = RMSE^2)
## # A tibble: 1 x 3
##    sigma    RMSE          MSE
##    <dbl>   <dbl>        <dbl>
## 1 56788. 56788. 3224869786.
```

> **ⓘ Note**
>
> Typically, these error metrics are computed on a separate validation set discussed in Section Section 2.5 or using cross-validation as will be discussed in a future chapter; however, they can also be computed on the same training data the model was trained on as illustrated here.

3.3.3 Inference

Let's go back to our `model1` results that show the b_0 and b_1 coefficient values:

```
tidy(model1)
## # A tibble: 2 x 5
##    term        estimate std.error statistic   p.value
##    <chr>          <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)   15938.    3852.      4.14 3.65e-  5
## 2 Gr_Liv_Area     110.      2.42     45.3  5.17e-311
```

Note that we call these coefficient values "estimates." Due to various reasons we should always assume that there is some variability in our estimated coefficient values. The variability of an estimate is often measured by its *standard error* (SE)—the square root of its variance. Since 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 were computed for us and displayed in the column labeled `std.error` in the output from `tidy()`.

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 `tidy()`, `t value (aka statistic) = 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 `tidy()` in the column labeled `p.value`.

> **i** Note
>
> This may seem quite complicated but don't worry, R will do the heavy lifting for us. Just realize we can use these additional statistics provided in our model summary to tell us if the predictor variable (`Gr_Liv_Area` in our example) has a statistically significant relationship with our response variable.

When the `p.value` for a given coefficient is quite small (i.e. `p.value` < 0.005), that is a good indication that the estimate for that coefficient is statistically different than zero. For example, the `p.value` for the `Gr_Liv_Area` coefficient is 5.17e-311 (basically zero). This means that the estimated coefficient value of 109.6675 is statistically different than zero.

Let's look at this from another perspective. Under the same assumptions, we can also derive confidence intervals for the coefficients. The formula for the traditional $100 \left(1 - \alpha\right)\%$ confidence interval for β_j is

$$\hat{\beta}_j \pm t_{1-\alpha/2,n-p}\widehat{SE}\left(\hat{\beta}_j\right). \tag{3.3}$$

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$fit, level = 0.95)
##                  2.5 %     97.5 %
## (Intercept) 8384.213 23492.1336
## Gr_Liv_Area  104.920   114.4149
```

To interpret, we estimate with 95% confidence that the mean selling price increases between 104.92 and 114.41 for each additional one square foot of above ground living space. We can also conclude that the slope b_1 is significantly different from zero (or any other pre-specified value not included in the interval) at the $\alpha = 0.05$ level ($\alpha = 0.05$ because we just take 1 - confidence level we are computing so $1 - 0.95 = 0.05$).

> **i** Note
>
> 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 regression 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 remedial 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.

3.3.4 Making predictions

We've created a simple linear regression model to describe the relationship between `Gr_Liv_Area` and `Sale_Price`. As we saw in the last chapter, we can make predictions with this model.

```
model1 %>%
    predict(ames_train)
## # A tibble: 2,049 x 1
##       .pred
##       <dbl>
##   1 135695.
##   2 135695.
##   3 107620.
##   4  98408.
##   5 126922.
##   6 224526.
##   7 114639.
##   8 129992.
##   9 205444.
## 10 132515.
## # i 2,039 more rows
```

And we can always add these predictions back to our training data if we want to look at how the predicted values differ from the actual values.

```
model1 %>%
   predict(ames_train) %>%
   bind_cols(ames_train) %>%
   select(Gr_Liv_Area, Sale_Price, .pred)
## # A tibble: 2,049 x 3
##    Gr_Liv_Area Sale_Price    .pred
##          <int>      <int>    <dbl>
##  1        1092     105500 135695.
##  2        1092      88000 135695.
##  3         836     120000 107620.
##  4         752     125000  98408.
##  5        1012      67500 126922.
##  6        1902     112000 224526.
##  7         900     122000 114639.
##  8        1040     127000 129992.
##  9        1728      84900 205444.
## 10        1063     128000 132515.
## # i 2,039 more rows
```

3.3.5 Assessing model accuracy

This allows us to assess the accuracy of our model. Recall from the last module that for regression models we often use mean squared error (MSE) and root mean squared error (RMSE) to quantify the accuracy of our model. These two values are directly correlated to the RSS we discussed above, which determines the best fit line. Let's illustrate.

3.3.5.1 Training data accuracy

Recall that the residuals are the differences between the actual y and the estimated $\hat{y}$ based on the best fit line.

```
residuals <- model1 %>%
   predict(ames_train) %>%
   bind_cols(ames_train) %>%
   select(Gr_Liv_Area, Sale_Price, .pred) %>%
   mutate(residual = Sale_Price - .pred)

head(residuals, 5)
## # A tibble: 5 x 4
##    Gr_Liv_Area Sale_Price    .pred residual
```

```
##            <int>         <int>    <dbl>      <dbl>
## 1          1092        105500 135695.    -30195.
## 2          1092         88000 135695.    -47695.
## 3           836        120000 107620.     12380.
## 4           752        125000  98408.     26592.
## 5          1012         67500 126922.    -59422.
```

The RSS squares these values, sums them, and multiples by 1 divided by the number of observations minus the number of coefficients in our model, which is 2.

```
residuals %>%
   mutate(squared_residuals = residual^2) %>%
   summarize(sum_of_squared_residuals = sum(squared_residuals), n = n()) %>%
   mutate(RSS = (1 / (n-2)) * sum_of_squared_residuals)
## # A tibble: 1 x 3
##    sum_of_squared_residuals       n           RSS
##                       <dbl> <int>         <dbl>
## 1                   6.60e12   2049 3224869786.
```

> **i** Note
>
> Why do we square the residuals? So that both positive and negative deviations of the same amount are treated equally. While taking the absolute value of the residuals would also treat both positive and negative deviations of the same amount equally, squaring the residuals is used for reasons related to calculus: taking derivatives and minimizing functions.

However, when expressing the performance of a model we rarely state the RSS. Instead it is more common to state the *average of the squared error*, or the MSE as discussed here. Unfortunately, both the RSS and MSE are not very intuitive because the units the metrics are expressed in do have much meaning. So, we usually use the RMSE metric, which simply takes the square root of the MSE metric so that your error metric is in the same units as your response variable.

We can manually compute this with the following, which tells us that on average, our linear regression model mispredicts the expected sale price of a home by about $56,760.

```
residuals %>%
   mutate(squared_residuals = residual^2) %>%
   summarize(
      MSE = mean(squared_residuals),
      RMSE = sqrt(MSE)
```

```
      )
## # A tibble: 1 x 2
##            MSE    RMSE
##          <dbl>   <dbl>
## 1 3221722037. 56760.
```

We could also compute this using the `rmse()` function we saw in the last module:

```
model1 %>%
   predict(ames_train) %>%
   bind_cols(ames_train) %>%
   rmse(truth = Sale_Price, estimate = .pred)
## # A tibble: 1 x 3
##    .metric .estimator .estimate
##    <chr>    <chr>          <dbl>
## 1 rmse     standard       56760.
```

3.3.5.2 Test data accuracy

Recall that a major goal of the machine learning process is to find a model that most accurately predicts future values based on a set of features. 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. In the last chapter we called this our generalization error.

So, ultimately, we want to understand how well our model will generalize to *unseen* data. To do this we need to compute the RMSE of our model on our test set.

> **i** Note
>
> Here, we see that our test RMSE is right around the same as our training data. As we'll see in later chapters, this is not always the case.

```
model1 %>%
   predict(ames_test) %>%
   bind_cols(ames_test) %>%
   rmse(truth = Sale_Price, estimate = .pred)
## # A tibble: 1 x 3
##    .metric .estimator .estimate
##    <chr>    <chr>          <dbl>
## 1 rmse     standard       55942.
```

3.3.6 Knowledge check

> **Caution**
>
> Let's revisit the relationship between `Year_Built` and `Sale_Price`. Using the `ames_train` data:
>
> 1. Visualize the relationship between these two variables.
> 2. Compute their correlation.
> 3. Create a simple linear regression model where `Sale_Price` is a function of `Year_Built`.
> 4. Interpret the coefficient for `Year_Built`.
> 5. What is the 95% confidence interval for this coefficient and can we confidently say it is statistically different than zero?
> 6. Using this model, make predictions using the test data. What is the predicted value for the first home in the test data?
> 7. Compute and interpret the generalization RMSE for this model. How does this model compare to the model based on `Gr_Liv_Area`?

3.4 Multiple linear regression

TBD

3.5 Exercises

> **Caution**
>
> We'll continue working with the Chicago ridership data we used for the exercises in the previous chapter.
>
> 1. Load the Chicago ridership data set and split the data into a training set and test set using a 70-30% split.
> 2. Using the training data
>
>   1. Visualize the relationship between the `Ashland` and `ridership` variables.
>   2. Compute the correlation between these two features.
>   3. Create a simple linear regression model with `ridership` as the response variable and `Ashland` as the feature variable.
>   4. Interpret the feature's coefficient.
>   5. What is the model's generalization error?

3. Now pick one of the `weather_` feature variables and repeat the process in #2.

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.

Faraway, Julian J. 2016. *Linear Models with r.* Chapman; Hall/CRC.

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

Harrell, Frank. 2017. "Classification Vs. Prediction." 2017. https://www.fharrell.com/post/classification/.

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.

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

Lipovetsky, Stan. 2020. Taylor & Francis.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning." *CoRR* abs/1312.5602. http://arxiv.org/abs/1312.5602.

Powell, Warren B. 2021. "From Reinforcement Learning to Optimal Control: A Unified Framework for Sequential Decisions." In *Handbook of Reinforcement Learning and Control*, 29–74. Springer.

Sutton, Richard S, and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction.* MIT press.

Szepesvári, Csaba. 2022. *Algorithms for Reinforcement Learning.* Springer Nature.

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.