UNIT-2

Dr T Prathima, Dept. of IT, CBIT(A)

Managing data: cleaning data, Data transformations, Sampling for modeling and validation.

Choosing and evaluating models: Mapping problems to machine learning tasks, evaluating models, Local interpretable model-agnostic explanations (LIME) for explaining model predictions

Zumel, N., Mount, J., &Porzak, J., "Practical data science with R", 2nd edition. Shelter Island, NY: Manning, 2019.

Managing Data

- Cleaning data
- Data transformations
- Sampling for Modeling and Validation

Cleaning Data

- Invalid Values : Handling invalid values is often *domain-specific*: which values are invalid, and what you do about them, depends on the problem that you are trying to solve.
  - The variable age has the problematic value 0, which probably means that the age is unknown. In addition, there are a few customers with ages older than 100 years, which may also be an error. However, for this project, we'll treat the value 0 as invalid, and assume ages older than 100 years are valid.
  - The variable income has negative values. We'll assume for this discussion that those values are invalid.
- Missing Values: Find null entries
- Many modeling algorithms in R (and in other languages) will quietly drop rows that have missing values. So if you have wide data, and many columns have missing values, it may not be safe to drop rows with missing values.
- This is because the fraction of rows with at least one missing value can be high in that situation, and you can lose most of your data

Listing 4.1 Treating the age and income variables

```
library(dplyr)
customer_data = readRDS("custdata.RDS")

customer_data <- customer_data %>%
  mutate(age = na_if(age, 0),
         income = ifelse(income < 0, NA, income))
```

Loads the data

Converts negative incomes to NA

The function mutate() from the dplyr package adds columns to a data frame, or modifies existing columns. The function na_if (), also from dplyr, turns a specific problematic value (in this case, 0) to NA.

- The variable `age` has the problematic value `0`, which probably means that the age is unknown.
- In addition, there are a few customers with ages older than 100 years, which may also be an error.
- However, for this project, we'll treat the value `0` as invalid, and assume ages older than 100 years are valid.

- The value 1 means "Gas bill included in rent or condo fee."
- The value 2 means "Gas bill included in electricity payment."
- The value 3 means "No charge or gas not used."

One way to treat gas_usage is to convert all the special codes (1, 2, 3) to NA, and to add three new *indicator variables*, one for each code. For example, the indicator variable gas_with_electricity will have the value 1 (or TRUE) whenever the original gas_usage variable had the value 2, and the value 0 otherwise. In the following listing, you will create the three new indicator variables, gas_with_rent, gas_with_electricity, and no_gas_bill.

Listing 4.2 Treating the gas_usage variable

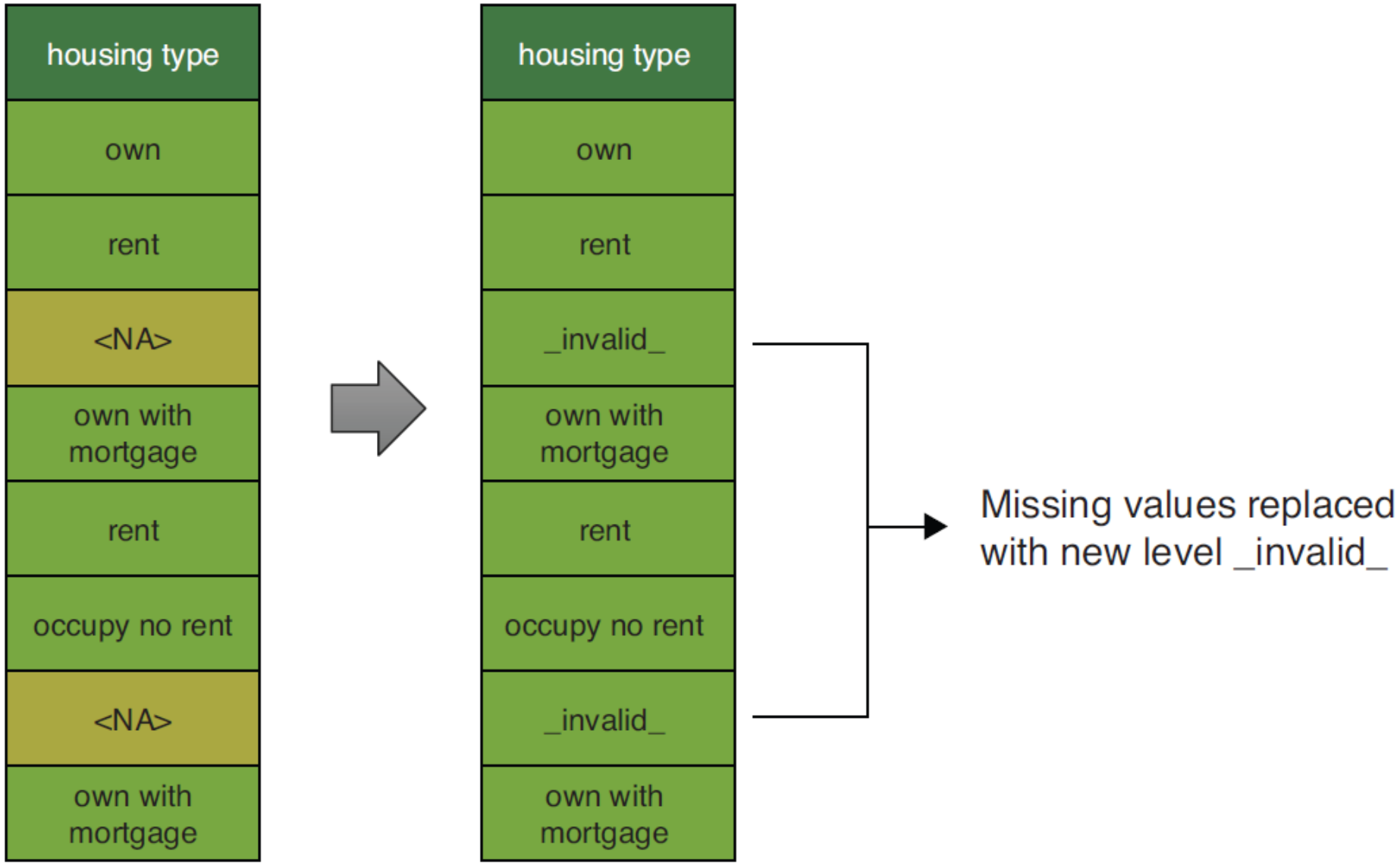
```
customer_data <- customer_data %>%
  mutate(gas_with_rent = (gas_usage == 1),
         gas_with_electricity = (gas_usage == 2),
         no_gas_bill = (gas_usage == 3) ) %>%
  mutate(gas_usage = ifelse(gas_usage < 4, NA, gas_usage))
```

Creates the three indicator variables

Converts the special codes in the gas_usage column to NA

- The variable gas_usage mixes numeric and symbolic data: values greater than 3 are monthly gas_bills, but values from 1 to 3 are special codes.
- In addition, gas_usage has some missing values.

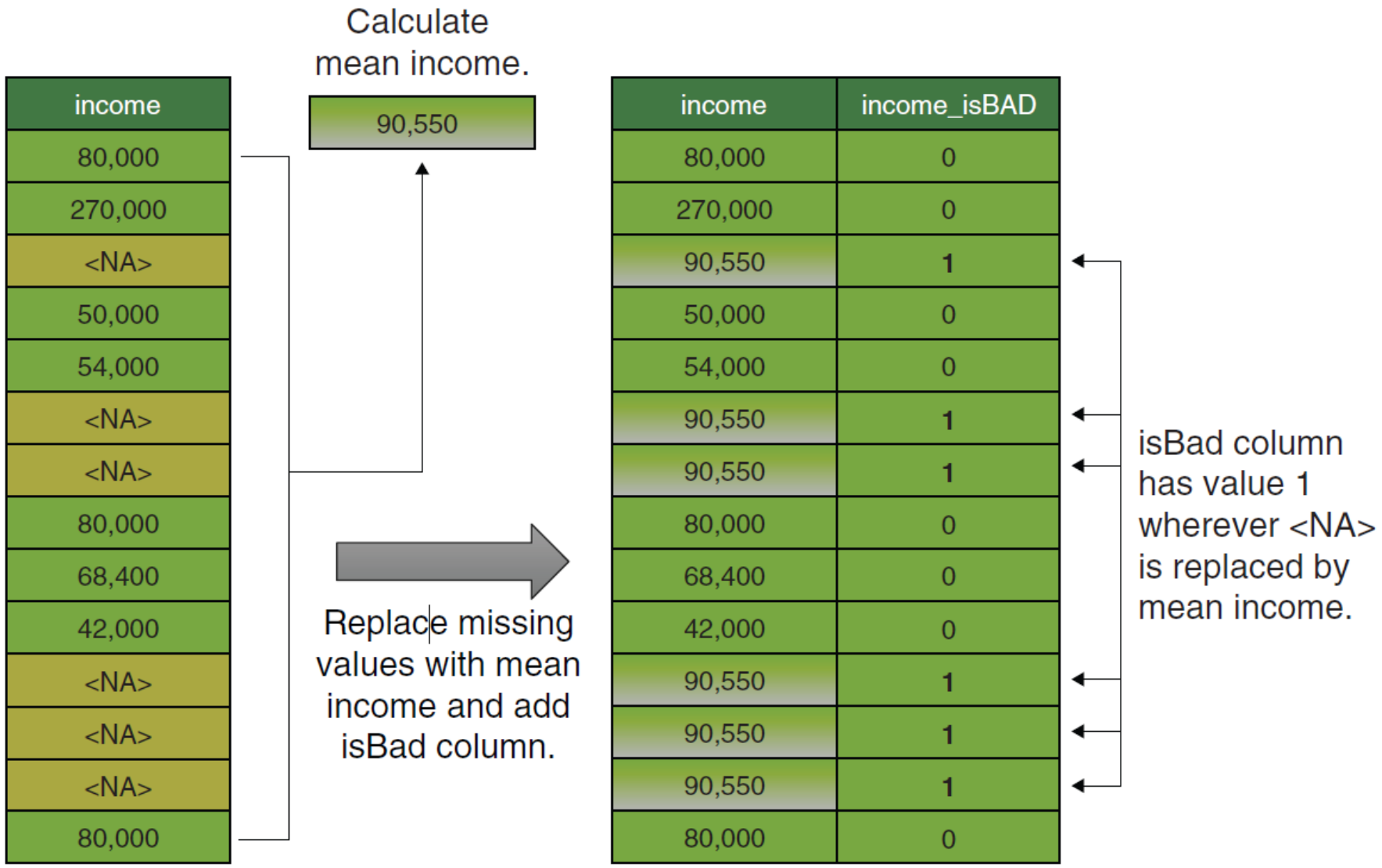
MISSING DATA IN CATEGORICAL VARIABLES



MISSING VALUES IN NUMERIC OR LOGICAL VARIABLES

- Replace with “reasonable estimate,” or *imputed value*. Statistically, one commonly used estimate is the expected, or mean, income
- You can improve this estimate when you remember that income is related to other variables in your data
- the customers with missing income information truly have no income, because they are full-time students or stay-at-home spouses or otherwise not in the active workforce. If this is so, then “filling in” their income information by using one of the preceding methods is an insufficient treatment, and may lead to false conclusions.
- The vtreat package for automatically treating missing variables

TREATING MISSING VALUES AS INFORMATION



income
80,000
270,000
<NA>
50,000
54,000
<NA>
<NA>
80,000
68,400
42,000
<NA>
<NA>
<NA>
80,000

income	income_isBAD
80,000	0
270,000	0
90,550	1
50,000	0
54,000	0
90,550	1
90,550	1
80,000	0
68,400	0
42,000	0
90,550	1
90,550	1
90,550	1
80,000	0

Figure 4.6 Replacing missing values with the mean and adding an indicator column to track the altered values

Listing 4.3 Counting the missing values in each variable

```
count_missing = function(df) {
   sapply(df, FUN=function(col) sum(is.na(col)) )
}
```

Defines a function that counts the number of NAs in each column of a data frame

```
nacounts <- count_missing(customer_data)
hasNA = which(nacounts > 0)
nacounts[hasNA]
```

Applies the function to customer_data, identifies which columns have missing values, and prints the columns and counts

```
##          is_employed               income        housing_type
##                25774                   45                1720
##          recent_move          num_vehicles                 age
##                 1721                 1720                  77
##            gas_usage       gas_with_rent gas_with_electricity
##                35702                 1720                1720
##          no_gas_bill
##                 1720
```

Fundamentally, there are two things you can do with these variables: drop the rows with missing values, or convert the missing values to a meaningful value. For variables like income or age that have very few missing values relative to the size of the data (customer_data has 73,262 rows), it could be safe to drop the rows. It wouldn't be safe to drop rows from variables like is_employed or gas_usage, where a large fraction of the values is missing.

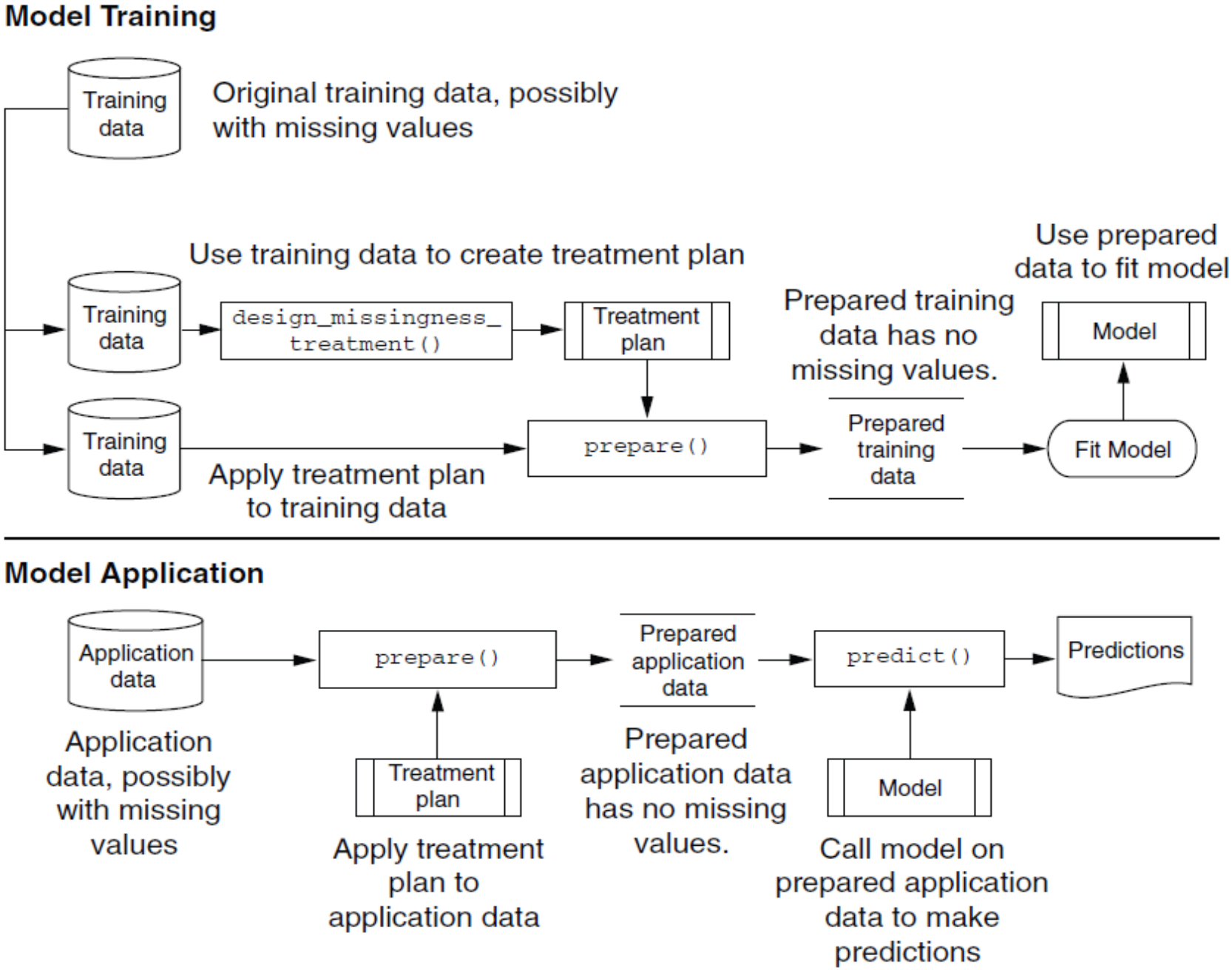


Figure 4.7 Creating and applying a simple treatment plan

Listing 4.6 Examining the data treatment

Finds the rows where housing_type was missing

```
htmissing <- which(is.na(customer_data$housing_type))

columns_to_look_at <- c("custid", "is_employed", "num_vehicles",
                        "housing_type", "health_ins")

customer_data[htmissing, columns_to_look_at] %>% head()
##           custid is_employed num_vehicles housing_type health_ins
## 55  000082691_01        TRUE           NA          <NA>      FALSE
## 65  000116191_01        TRUE           NA          <NA>       TRUE
## 162 000269295_01          NA           NA          <NA>      FALSE
## 207 000349708_01          NA           NA          <NA>      FALSE
## 219 000362630_01          NA           NA          <NA>       TRUE
## 294 000443953_01          NA           NA          <NA>       TRUE
```

Looks at a few columns from those rows in the original data

```
columns_to_look_at = c("custid", "is_employed", "is_employed_isBAD",
                       "num_vehicles","num_vehicles_isBAD",
                       "housing_type", "health_ins")

training_prepared[htmissing, columns_to_look_at] %>%  head()
##           custid is_employed is_employed_isBAD num_vehicles
## 55  000082691_01   1.0000000                 0       2.0655
## 65  000116191_01   1.0000000                 0       2.0655
## 162 000269295_01   0.9504928                 1       2.0655
## 207 000349708_01   0.9504928                 1       2.0655
## 219 000362630_01   0.9504928                 1       2.0655
## 294 000443953_01   0.9504928                 1       2.0655
##     num_vehicles_isBAD housing_type health_ins
## 55                   1    _invalid_      FALSE
## 65                   1    _invalid_       TRUE
## 162                  1    _invalid_      FALSE
## 207                  1    _invalid_      FALSE
## 219                  1    _invalid_       TRUE
## 294                  1    _invalid_       TRUE
```

Looks at those rows and columns in the treated data (along with the isBADs)

Verifies the expected number of vehicles and the expected unemployment rate in the dataset

```
customer_data %>%
    summarize(mean_vehicles = mean(num_vehicles, na.rm = TRUE),
    mean_employed = mean(as.numeric(is_employed), na.rm = TRUE))
##   mean_vehicles mean_employed
## 1        2.0655     0.9504928
```

Data transformations

- The purpose of data transformation is to make data easier to model, and easier to understand. Machine learning works by learning meaningful patterns in training data, and then making predictions by exploiting those patterns in new data.
- **Example** *Suppose you are considering the use of income as an input to your insurance model. The cost of living will vary from state to state, so what would be a high salary in one region could be barely enough to scrape by in another. Because of this, it might be more meaningful to normalize a customer's income by the typical income in the area where they live. This is an example of a relatively simple (and common) transformation.*
- Because customers in different states get a different normalization, we call this a *conditional* transform. A long way to say this is that "the normalization is conditioned on the customer's state of residence." We would call scaling all the customers by the same value an *unconditioned* transform.
- Data Transformations:
  - Normalization
  - Centering and scaling
  - Log transformations

Listing 4.7 Normalizing income by state

```
library(dplyr)
median_income_table <-
      readRDS("median_income.RDS")
head(median_income_table)

##   state_of_res median_income
## 1      Alabama         21100
## 2       Alaska         32050
## 3      Arizona         26000
## 4     Arkansas         22900
## 5   California         25000
## 6     Colorado         32000

training_prepared <-  training_prepared %>%
  left_join(., median_income_table, by="state_of_res") %>%
   mutate(income_normalized = income/median_income)
```

If you have downloaded the PDSwR2 code example directory, then median_income.RDS is in the directory PDSwR2/Custdata. We assume that this is your working directory.

Joins median_income_table into the customer data, so you can normalize each person's income by the median income of their state

```
head(training_prepared[, c("income", "median_income", "income_normalized")])

##   income median_income income_normalized
## 1  22000         21100          1.0426540
## 2  23200         21100          1.0995261
## 3  21000         21100          0.9952607
## 4  37770         21100          1.7900474
## 5  39000         21100          1.8483412
## 6  11100         21100          0.5260664

summary(training_prepared$income_normalized)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.4049  1.0000  1.5685  1.9627 46.5556
```

Compares the values of income and income_normalized

Listing 4.8 Normalizing by mean age

```
summary(training_prepared$age)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   21.00   34.00   48.00   49.22   62.00  120.00

mean_age <- mean(training_prepared$age)
age_normalized <- training_prepared$age/mean_age
summary(age_normalized)

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.4267  0.6908  0.9753  1.0000  1.2597  2.4382
```

A value for `age_normalized` that is much less than 1 signifies an unusually young customer; much greater than 1 signifies an unusually old customer. But what constitutes "much less" or "much greater" than 1? That depends on how wide an age spread your customers tend to have. See figure 4.8 for an example.

Normalisation

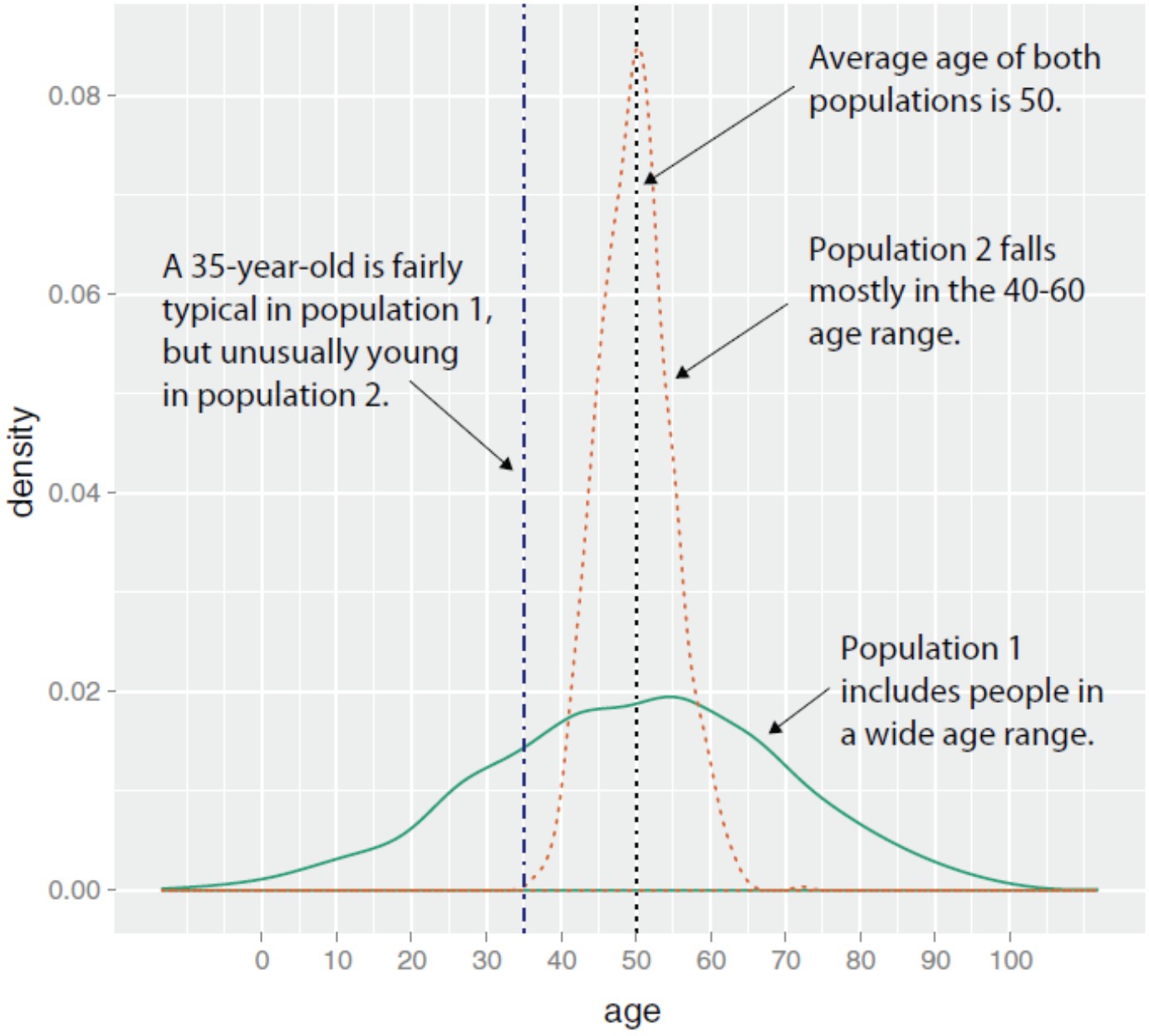


Figure 4.8 Is a 35-year-old young?

- Normalization (or rescaling) is useful when absolute quantities are less meaningful than relative ones.
- The average customer in both populations is 50. The *population 1* group has a fairly wide age spread, so a 35-year-old still seems fairly typical (perhaps a little young).
- That same 35-year-old seems unusually young in *population 2*, which has a narrow age spread.
- The typical age spread of your customers is summarized by the standard deviation.
- This leads to another way of expressing the relative ages of your customers.

Centering and Scaling

- You can rescale your data by using the standard deviation as a unit of distance.
- A customer who is within one standard deviation of the mean age is considered not much older or younger than typical.
- A customer who is more than one or two standard deviations from the mean can be considered much older, or much younger.
- To make the relative ages even easier to understand, you can also center the data by the mean, so a customer of "typical age" has a centered age of 0.

- When you have multiple numeric variables, you can use the scale() function to center and scale all of them simultaneously. This has the advantage that the numeric variables now all have similar and more-compatible ranges. To make this concrete, compare the variable age in years to the variable income in dollars. A 10-year difference in age between two customers could be a lot, but a 10-dollar difference in income is quite small.
- If you center and scale both variables, then the value 0 means the same thing for both scaled variables: the mean age or mean income. And the value 1.5 also means the same thing: a person who is 1.5 standard deviations older than the mean age, or who makes 1.5 standard deviations more than the mean income. In both situations, the value 1.5 can be considered a big difference from the average.

Listing 4.9 Centering and scaling age

```
(mean_age <- mean(training_prepared$age))
 ## [1] 49.21647

(sd_age <- sd(training_prepared$age))
 ## [1] 18.0124

print(mean_age + c(-sd_age, sd_age))
 ## [1] 31.20407 67.22886

training_prepared$scaled_age <- (training_prepared$age -
      mean_age) / sd_age

training_prepared %>%
  filter(abs(age - mean_age) < sd_age) %>%
  select(age, scaled_age) %>%
  head()

##   age scaled_age
## 1  67  0.9872942
## 2  54  0.2655690
## 3  61  0.6541903
## 4  64  0.8207422
## 5  57  0.4321210
## 6  55  0.3210864

training_prepared %>%
  filter(abs(age - mean_age) > sd_age) %>%
  select(age, scaled_age) %>%
  head()

##   age scaled_age
## 1  24  -1.399951
## 2  82   1.820054
## 3  31  -1.011329
## 4  93   2.430745
## 5  76   1.486950
## 6  26  -1.288916
```

Takes the mean

Takes the standard deviation

The typical age range for this population is from about 31 to 67.

Uses the mean value as the origin (or reference point) and rescales the distance from the mean by the standard deviation

Customers in the typical age range have a scaled_age with magnitude less than 1.

Customers outside the typical age range have a scaled_age with magnitude greater than 1.

Now, values less than `-1` signify customers younger than typical; values greater than `1` signify customers older than typical.

A technicality

- The common interpretation of standard deviation as a unit of distance implicitly assumes that the data is distributed normally.
- For a normal distribution, roughly two thirds of the data (about 68%) is within plus/minus one standard deviation from the mean.
- About 95% of the data is within plus/minus two standard deviations from the mean.
- In figure 4.8 (reproduced as a faceted graph in figure 4.9), a 35-year-old is within one standard deviation from the mean in *population 1*, but more than one (in fact, more than two) standard deviations from the mean in *population 2*.
- You can still use this transformation if the data isn't normally distributed, but the standard deviation is most meaningful as a unit of distance if the data is unimodal and roughly symmetric around the mean.

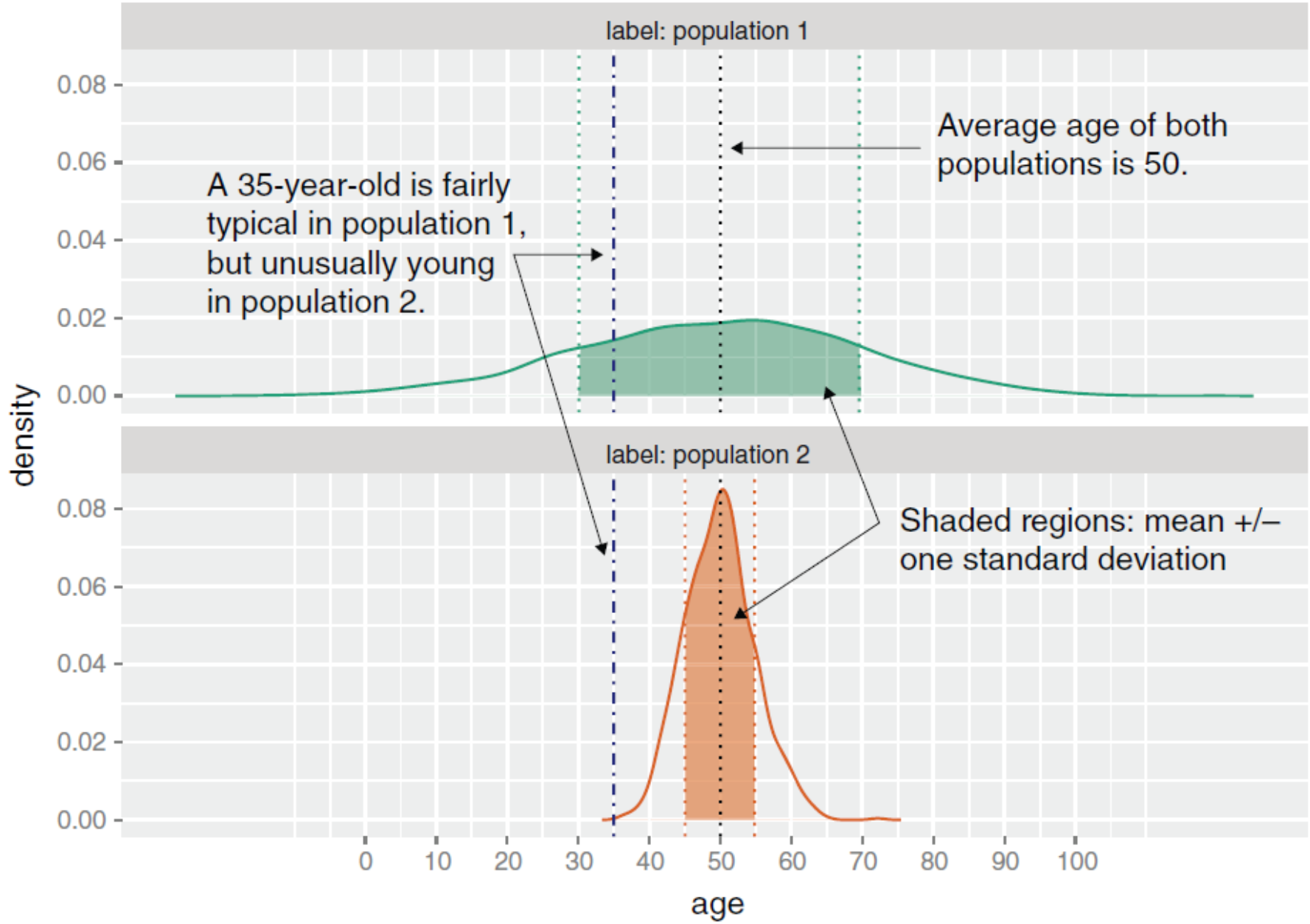


Figure 4.9 Faceted graph: is a 35-year-old young?

Listing 4.10 Centering and scaling multiple numeric variables

```
dataf <- training_prepared[, c("age", "income", "num_vehicles", "gas_usage")
summary(dataf)

##       age              income         num_vehicles     gas_usage
##  Min.   : 21.00   Min.   :      0   Min.   :0.000   Min.   :  4.00
##  1st Qu.: 34.00   1st Qu.:  10700   1st Qu.:1.000   1st Qu.: 50.00
##  Median : 48.00   Median :  26300   Median :2.000   Median : 76.01
##  Mean   : 49.22   Mean   :  41792   Mean   :2.066   Mean   : 76.01
##  3rd Qu.: 62.00   3rd Qu.:  51700   3rd Qu.:3.000   3rd Qu.: 76.01
##  Max.   :120.00   Max.   :1257000   Max.   :6.000   Max.   :570.00

dataf_scaled <- scale(dataf, center=TRUE, scale=TRUE)

summary(dataf_scaled)
##       age                income          num_vehicles        gas_usage
##  Min.   :-1.56650   Min.   :-0.7193   Min.   :-1.78631   Min.   :-1.4198
##  1st Qu.:-0.84478   1st Qu.:-0.5351   1st Qu.:-0.92148   1st Qu.:-0.5128
##  Median :-0.06753   Median :-0.2666   Median :-0.05665   Median : 0.0000
##  Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.00000   Mean   : 0.0000
##  3rd Qu.: 0.70971   3rd Qu.: 0.1705   3rd Qu.: 0.80819   3rd Qu.: 0.0000
##  Max.   : 3.92971   Max.   :20.9149   Max.   : 3.40268   Max.   : 9.7400

(means <- attr(dataf_scaled, 'scaled:center'))
 ##          age       income num_vehicles    gas_usage
##     49.21647  41792.51062      2.06550     76.00745

(sds <- attr(dataf_scaled, 'scaled:scale'))
##          age       income num_vehicles    gas_usage
##    18.012397 58102.481410     1.156294    50.717778
```

Centers the data by its mean and scales it by its standard deviation

Gets the means and standard deviations of the original data, which are stored as attributes of dataf_scaled

Listing 4.11 Treating new data before feeding it to a model

```
newdata <- customer_data

library(vtreat)
newdata_treated <- prepare(treatment_plan, newdata)

new_dataf <- newdata_treated[, c("age", "income",
     "num_vehicles", "gas_usage")]

dataf_scaled <- scale(new_dataf, center=means, scale=sds)
```

Simulates having a new customer dataset

Cleans it using the treatment plan from the original dataset

Scales age, income, num_vehicles, and gas_usage using the means and standard deviations from the original dataset

KEEP THE TRAINING TRANSFORMATION

- When you use parameters derived from the data (like means, medians, or standard deviations) to transform the data before modeling, you generally should keep those parameters and use them when transforming new data that will be input to the model.
- The same principle applies when cleaning missing values using the function from the `vtreat` package, the resulting treatment plan keeps the information from the training data in order to clean missing values from new data
- However, there are some situations when you may wish to use new parameters. For example, if the important information in the model is how a subject's income relates to the *current* median income, then when preparing new data for modeling, you would want to normalize income by the current median income, rather than the median income from the time when the model was trained.

Log transformations for skewed and wide distributions

- Normalizing by mean and standard deviation is most meaningful when the data distribution is roughly symmetric.
- The *lognormal distribution* is the distribution of a random variable `X` whose natural log `log(X)` is normally distributed.
- The distribution of highly skewed positive data, like the value of profitable customers, incomes, sales, or stock prices, can often be modeled as a lognormal distribution.
- A lognormal distribution is defined over all nonnegative real numbers; it's asymmetric, with a long tail out toward positive infinity.
- The distribution of `log(X)` is a normal distribution centered at `mean(log(X))`.
- For lognormal populations, the mean is generally much higher than the median, and the bulk of the contribution toward the mean value is due to a small population of highest-valued data points.

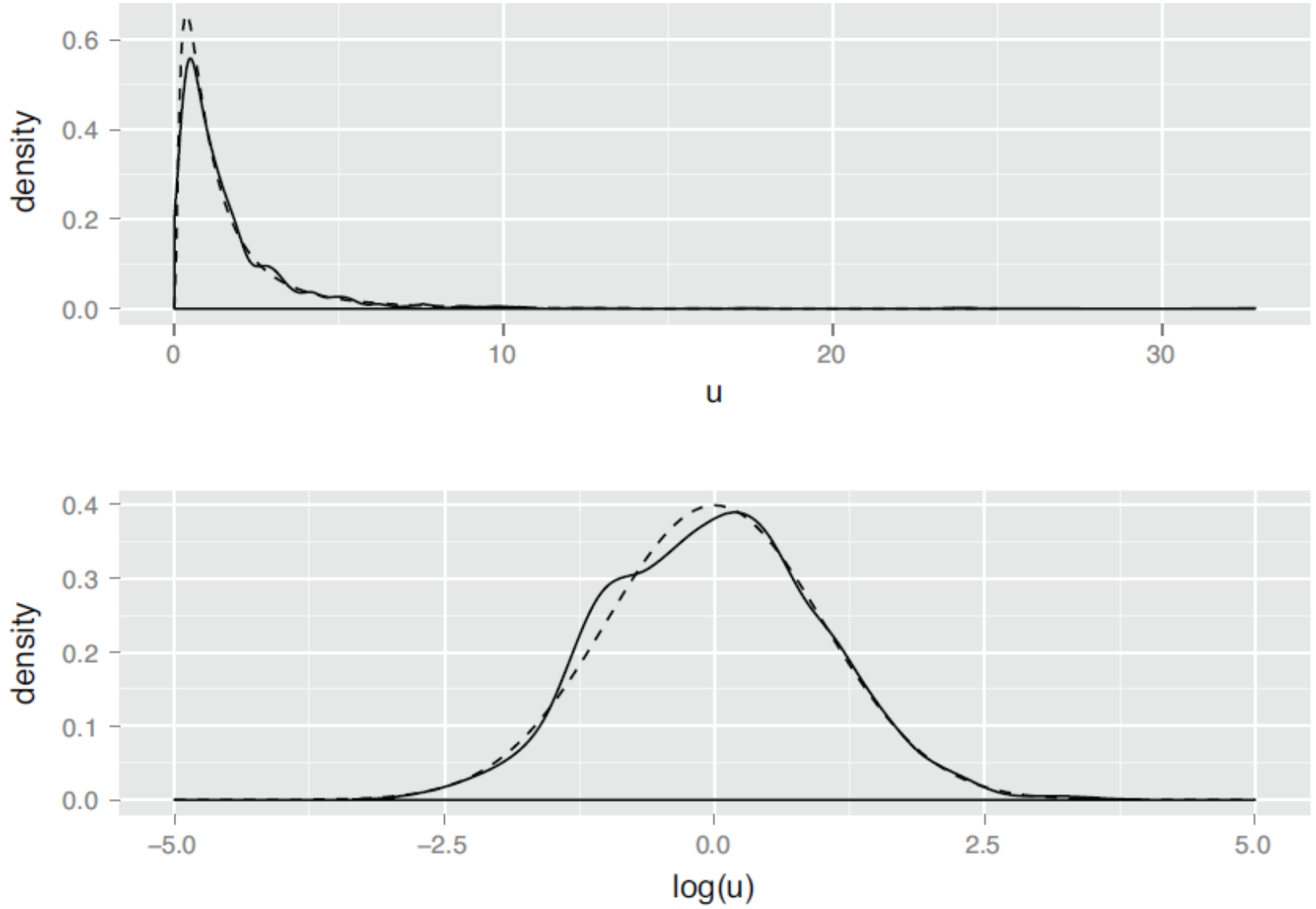


Figure B.4 Top: The lognormal distribution `X` such that `mean(log(X)) = 0` and `sd(log(X)) = 1`. The dashed line is the theoretical distribution, and the solid line is the distribution of a random lognormal sample. Bottom: The solid line is the distribution of `log(X)`.

Signed Logarithm

- Taking the logarithm only works if the data is non-negative, because the log of zero is –Infinity and the log of negative values isn't defined (R marks the log of negative numbers as `NaN`: not a number).
- In applications where the skewed data is monetary (like account balances or customer value), we instead use what we call a *signed logarithm*.
- A signed logarithm takes the logarithm of the absolute value of the variable and multiplies by the appropriate sign. Values strictly between `-1` and `1` are mapped to zero.
- The difference between log and signed log is shown in figure 4.11.
- This maps all datums between -1 and 1 to zero, so clearly this transformation isn't useful if values with magnitude less than 1 are important. But with many monetary variables (in US currency), values less than a dollar aren't much different from zero (or 1), for all practical purposes.
- So, for example, mapping account balances that are less than or equal to $1 (the equivalent of every account always having a minimum balance of $1) is probably okay

Here's how to calculate signed log base 10 in R:

```
signedlog10 <- function(x) {
    ifelse(abs(x) <= 1, 0, sign(x)*log10(abs(x)))
}
```

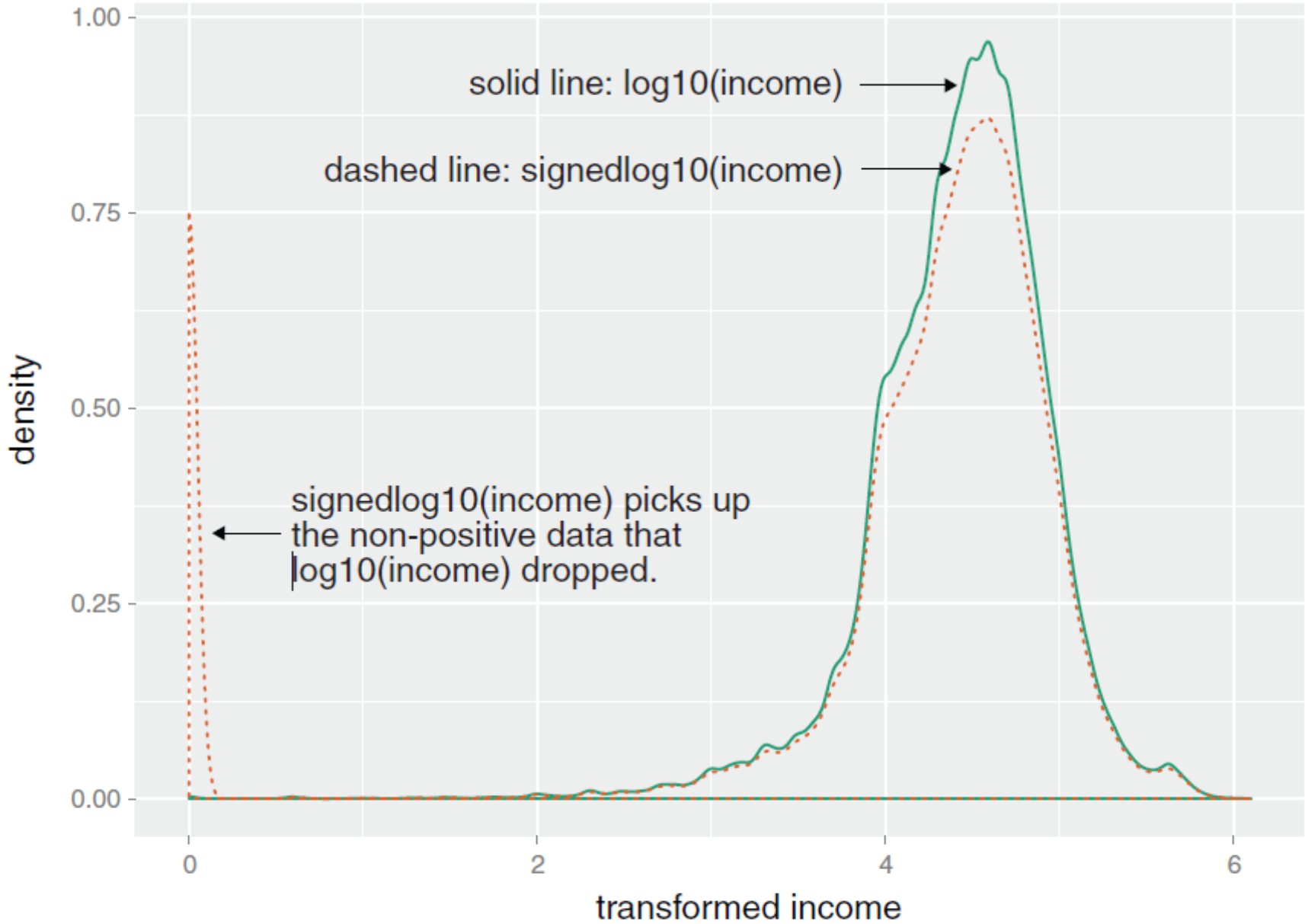


Figure 4.11 Signed log lets you visualize non-positive data on a logarithmic scale.

- It’s also generally a good idea to log transform data containing values that range over several orders of magnitude, for example, the population of towns and cities, which may range from a few hundred to several million.
- One reason for this is that modeling techniques often have a difficult time with very wide data ranges.
- Another reason is because such data often comes from multiplicative processes rather than from an additive one, so log units are in some sense more natural.
- Additive process:
  - If you weigh 150 pounds and your friend weighs 200, you’re equally active, and you both go on the exact same restricted-calorie diet, then you’ll probably both lose about the same number of pounds. How much weight you lose doesn’t depend on how much you weighed in the first place, only on calorie intake.
  - The natural unit of measurement in this situation is absolute pounds (or kilograms) lost.
- Multiplicative process:
  - If management gives everyone in the department a raise, it probably isn’t giving everyone $5,000 extra. Instead, everyone gets a 2% raise
  - When the process is multiplicative, log transforming the process data can make modeling easier.

Sampling for modeling and validation

- Sampling is the process of selecting a subset of a population to represent the whole during analysis and modeling.
- some understanding of sampling is always needed to work with data.
- We can certainly analyze larger datasets than we could before, but sampling is still a useful tool.
- When you're in the middle of developing or refining a modeling procedure, it's easier to test and debug the code on small subsamples before training the model on the entire dataset
- It's important that the dataset that you do use is an accurate representation of your population as a whole.

Test and training splits

- When you're building a model to make predictions, like our model to predict the probability of health insurance coverage, you need data to build the model.
- You also need data to test whether the model makes correct predictions on new data.
- The first set is called the *training set*, and the second set is called the *test* (or *holdout*) set.
- Many writers recommend train/calibration/ test splits, where the *calibration set* is used to set parameters that the model-fitting algorithm needs, and the training set is used to fit the model.
- This is also good advice. Our philosophy is this: split the data into train/test early, don't look at test until final evaluation, and if you need calibration data, resplit it from your training subset

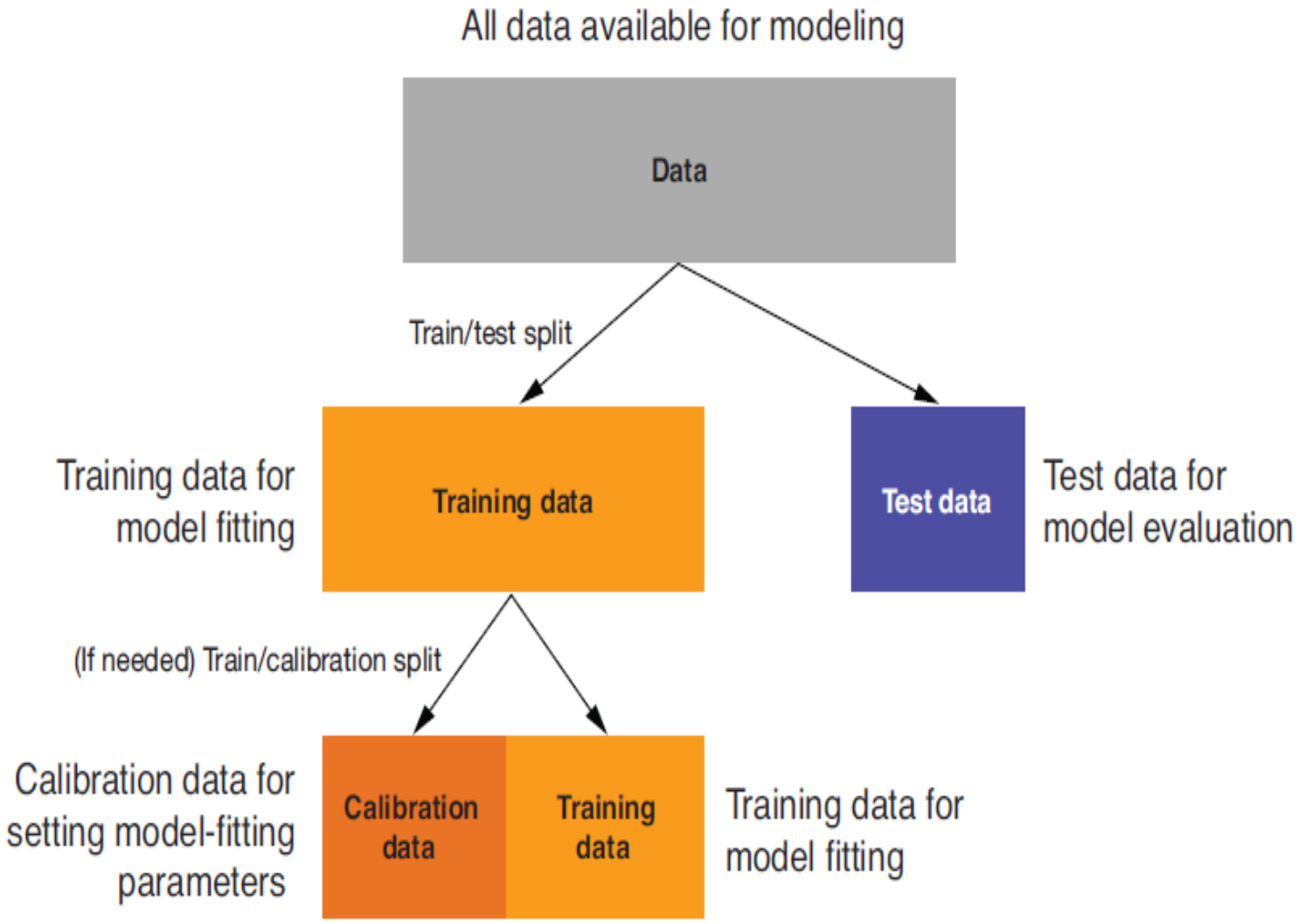


Figure 4.12 Splitting data into training and test (or training, calibration, and test) sets

- Creating a Sample group column:
  - A convenient way to manage random sampling is to add a sample group column to the data frame.
  - The sample group column contains a number generated uniformly from zero to one, using the `runif()` function. You can draw a random sample of arbitrary size from the data frame by using the appropriate threshold on the sample group column.
- Record Group: grouping based on household id or customer ID
- Data provenance:
  - You'll also want to add a column (or columns) to record data provenance: when your dataset was collected, perhaps what version of your data-cleaning procedure was used on the data before modeling, and so on.
  - This metadata is akin to version control for data. It's handy information to have, to make sure that you're comparing apples to apples when you're in the process of improving your model, or comparing different models or different versions of a model.

Listing 4.12 Splitting into test and training using a random group mark

```
set.seed(25643)
customer_data$gp <- runif(nrow(customer_data))
customer_test <- subset(customer_data, gp <= 0.1)
customer_train <- subset(customer_data, gp > 0.1)

dim(customer_test)
## [1] 7463   16

dim(customer_train)
## [1] 65799    16
```

Sets the random seed so this example is reproducible

Creates the grouping column

Here we generate a test set of about 10% of the data.

Here we generate a training set using the remaining data.

Listing 4.12 generates a test set of approximately 10% of the data and allocates the remaining 90% of the data to the training set.

Choosing and Evaluating Models

- Mapping problems to machine learning tasks
  - Mapping business problems to machine learning tasks
  - Evaluating model quality
  - Explaining model predictions
- Evaluating models
- Local interpretable model-agnostic explanations (LIME) for explaining model predictions

Mapping problems to machine learning tasks

- Predicting what customers might buy, based on past transactions
- Identifying fraudulent transactions
- Determining price elasticity (the rate at which a price increase will decrease sales, and vice versa) of various products or product classes
- Determining the best way to present product listings when a customer searches for an item
- Customer segmentation: grouping customers with similar purchasing behavior
- AdWord valuation: how much the company should spend to buy certain AdWords on search engines
- Evaluation of marketing campaigns
- Organizing new products into a product catalog

Different kinds of problems solved by DS

- *Classification*—Assigning labels to datums
- *Scoring*—Assigning numerical values to datums
- *Grouping*—Discovering patterns and commonalities in data

Classification Problems

- **Automate the assignment of new products to your company's product categories**
- Products that come from different sources may have their own product classification that doesn't coincide with the one that you use on your retail site, or they may come without any classification at all.
- Many large online retailers use teams of human taggers to hand categorize their products.
- This is not only labor intensive, but inconsistent and error prone.
- Automation is an attractive option; it's labor saving, and can improve the quality of the retail site.
- Product categorization based on product attributes and/or text descriptions of the product is an example of *classification*: deciding how to assign (known) labels to an object.
- Classification itself is an example of what is called *supervised learning*: in order to learn how to classify objects, you need a dataset of objects that have already been classified (called the *training set*).
- Building training data is the major expense for most classification tasks, especially text-related ones.

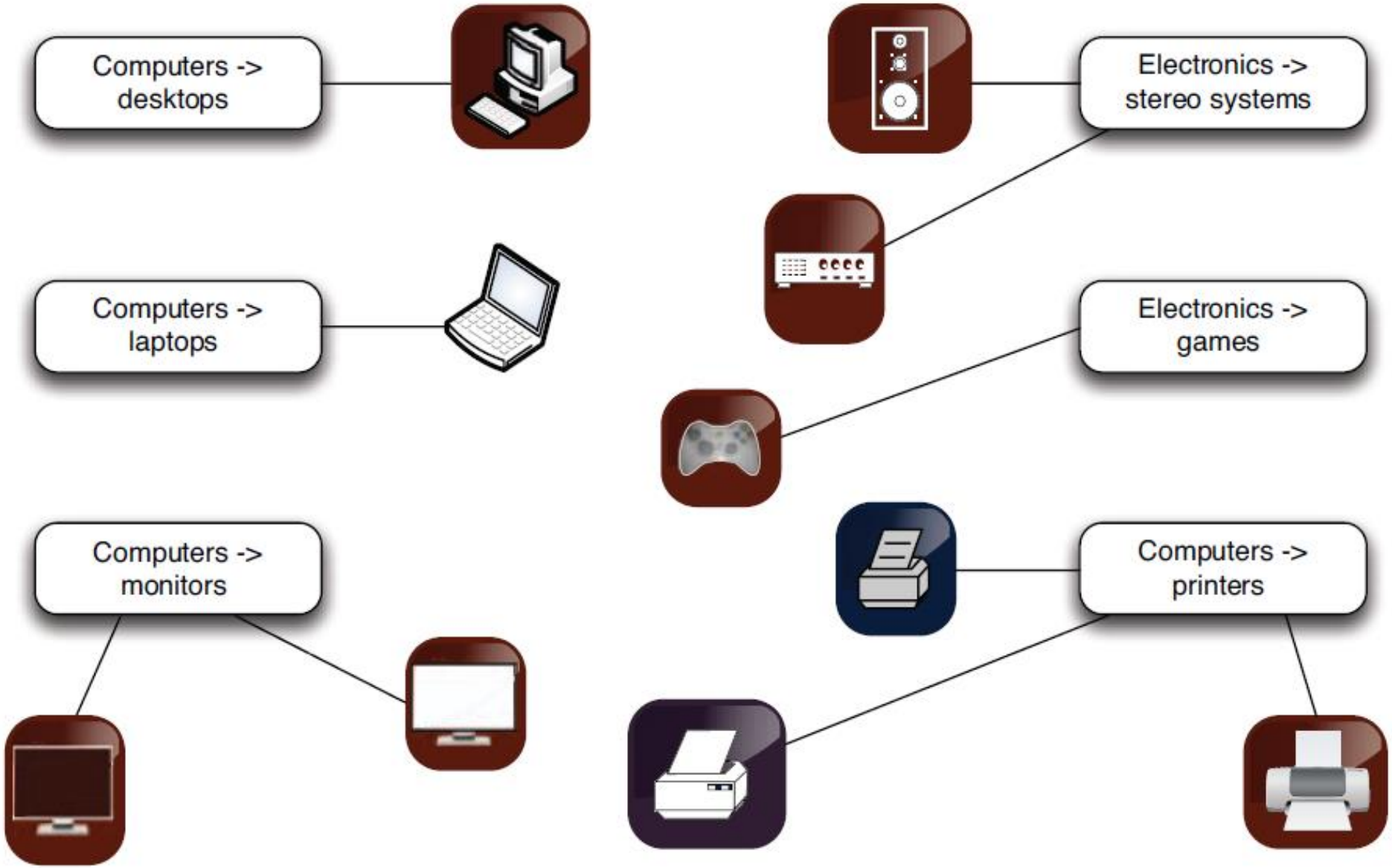


Figure 6.2 Assigning products to product categories

Multicategory vs. two-category classification

- Product classification is an example of *multicategory* or *multinomial* classification.
- Most classification problems and most classification algorithms are specialized for two-category, or binomial, classification.
- There are tricks to using binary classifiers to solve multicategory problems (for example, building one classifier for each category, called a *one-versus-rest* classifier).
- But in most cases it's worth the effort to find a suitable multiple-category implementation, as they tend to work better than multiple binary classifiers

Scoring Problems

- ***Suppose that your task is to help evaluate how different marketing campaigns can increase valuable traffic to the website. The goal is not only to bring more people to the site, but to bring more people who buy.***
- Consider a number of different factors: the communication channel (ads on websites, YouTube videos, print media, email, and so on); the traffic source (Facebook, Google, radio stations, and so on); the demographic targeted; the time of year, and so on.
- You want to measure if these factors increase sales, and by how much.
- Predicting the increase in sales from a particular marketing campaign based on factors such as these is an example of *regression*, or *scoring*.
- In this case, a regression model would map the different factors being measured into a numerical value: sales, or the increase in sales from some baseline.
- Predicting the probability of an event (like belonging to a given class) can also be considered scoring. For example, you might think of fraud detection as classification:
- is this event fraud or not? However, if you are trying to estimate the probability that an event is fraud, this can be considered scoring.
- Scoring is also an instance of supervised learning.

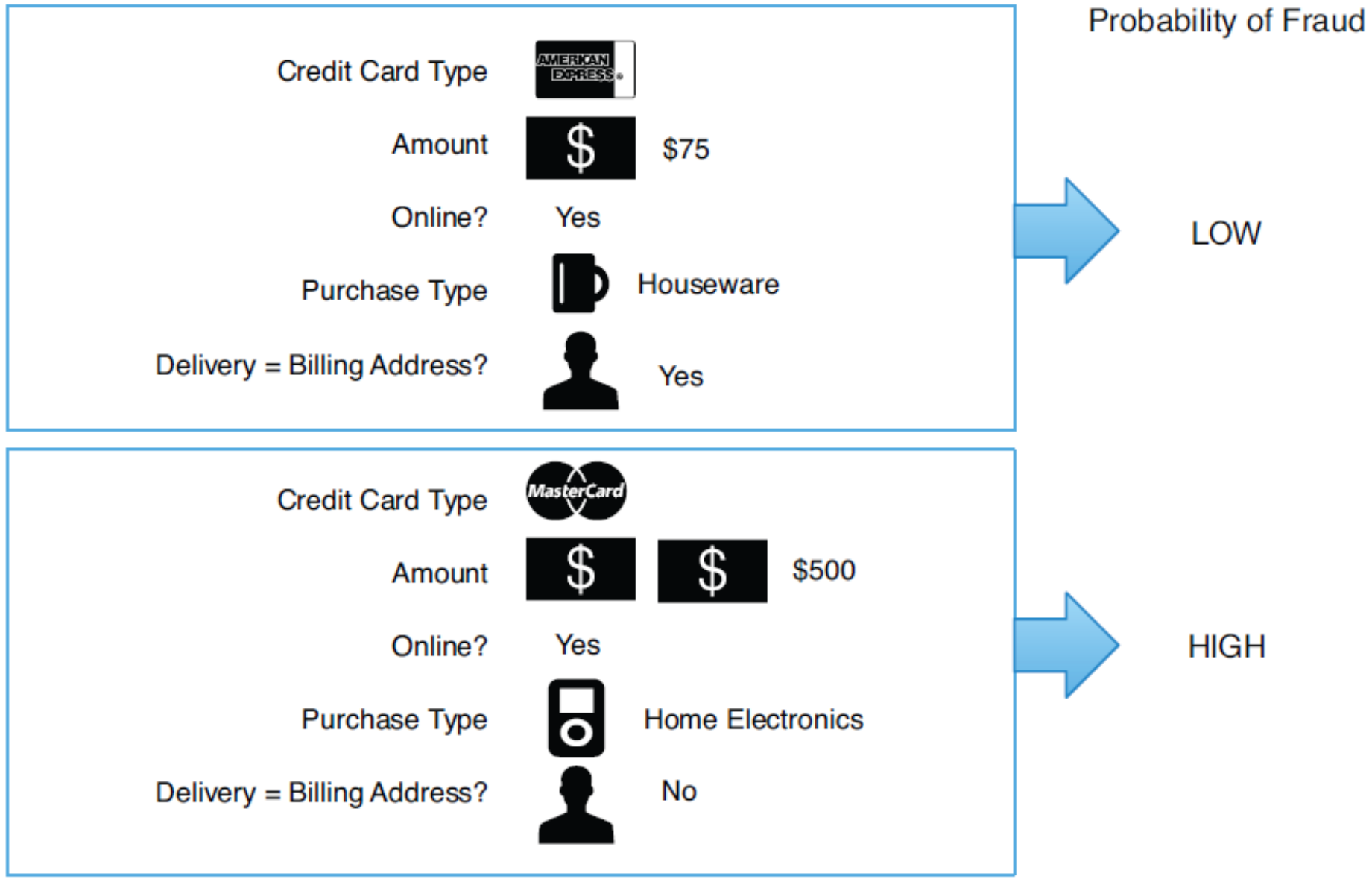


Figure 6.3 Notional example of determining the probability that a transaction is fraudulent

Grouping: working without known targets

- The preceding methods require that you have a training dataset of situations with known outcomes. In some situations, there's not (yet) a specific outcome that you want to predict.
- Instead, you may be looking for patterns and relationships in the data that will help you understand your customers or your business better.
- These situations correspond to a class of approaches called *unsupervised learning*: rather than predicting outputs based on inputs, the objective of unsupervised learning is to discover similarities and relationships in the data.
- Some common unsupervised tasks include these:
- *Clustering*—Grouping similar objects together
- *Association rules*—Discovering common behavior patterns, for example, items that are always bought together, or library books that are always checked out together
- ***Suppose you want to segment your customers into general categories of people with similar buying patterns. You might not know in advance what these groups should be.***

k-means clustering

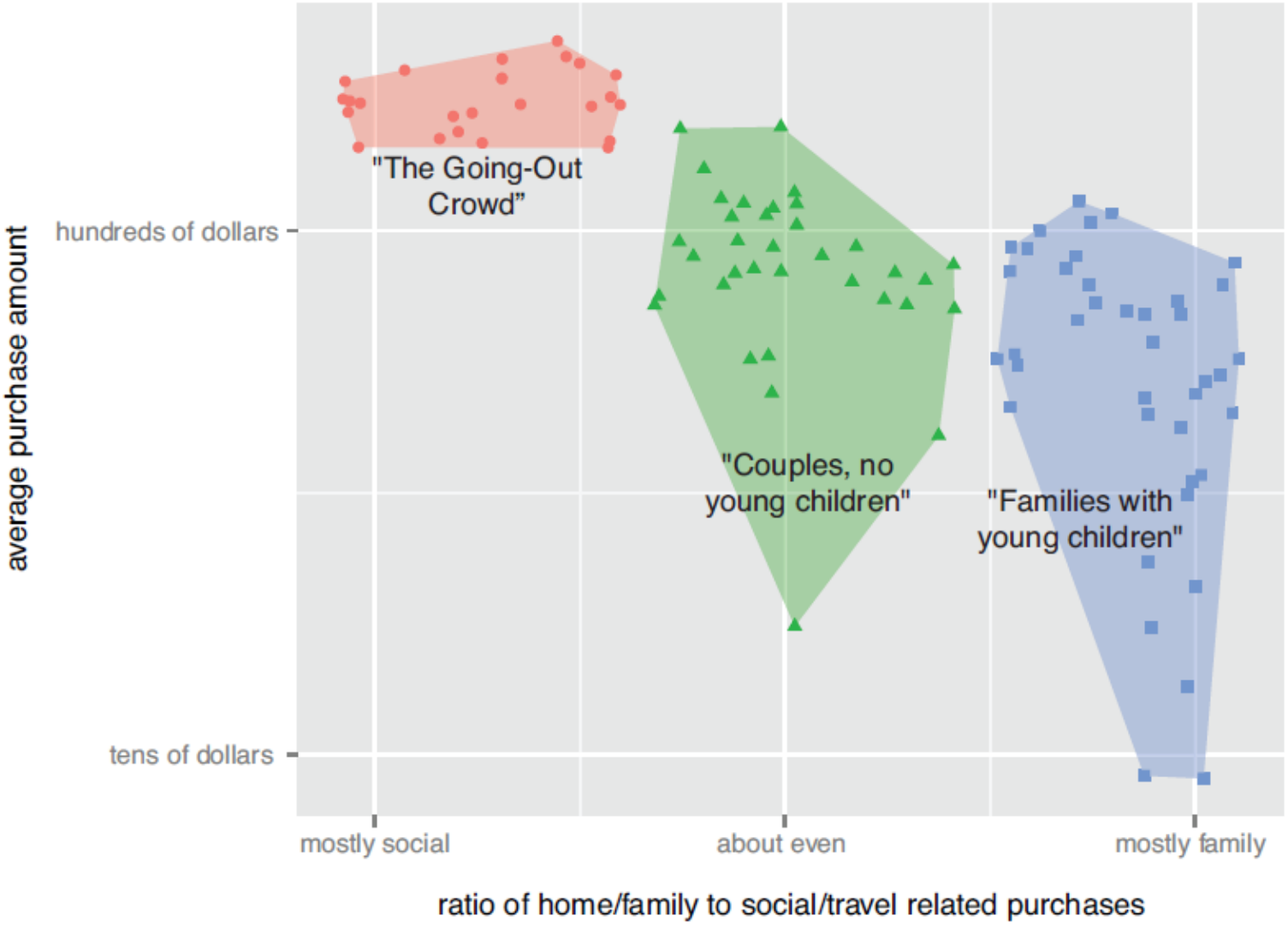
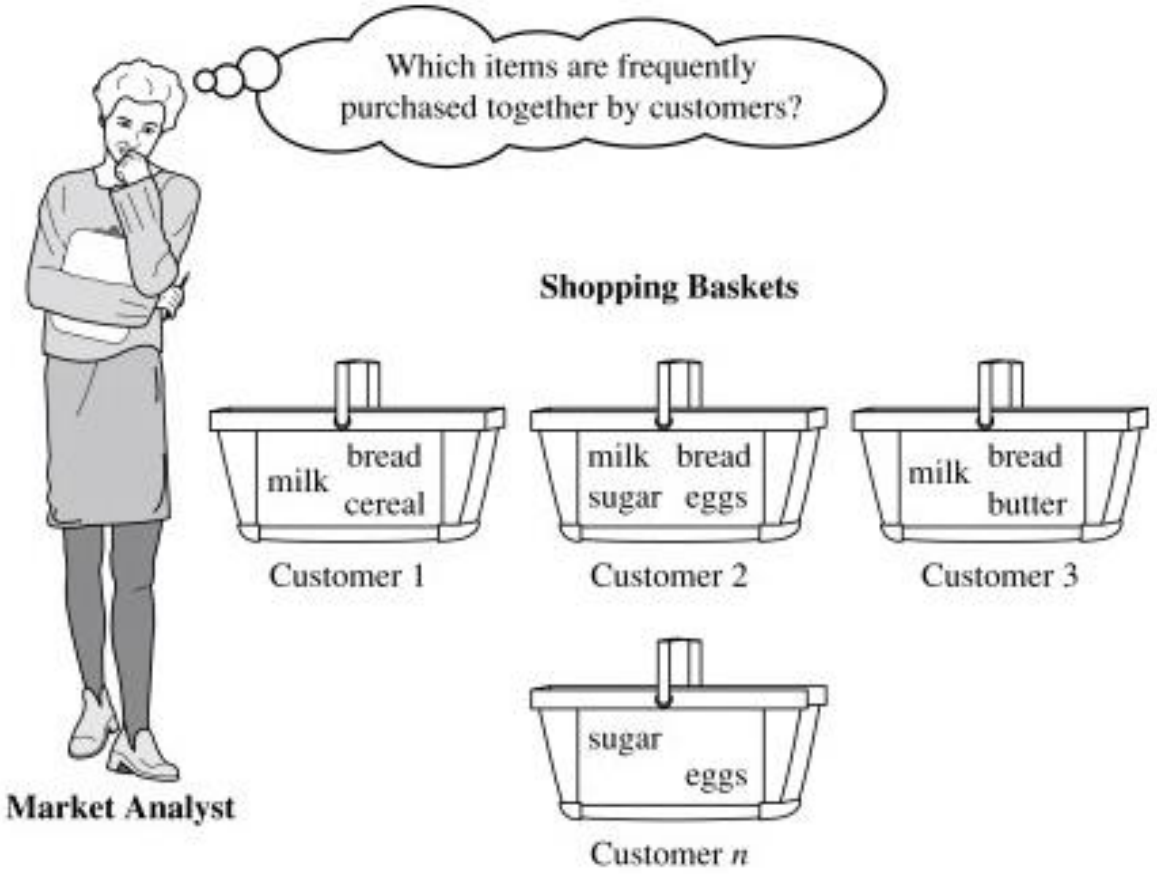


Figure 6.4 Notional example of clustering your customers by purchase pattern and purchase amount

ASSOCIATION RULES

- Bathing suits and sunglasses are frequently purchased at the same time
- This is a good application for association rules (or even recommendation systems).
- You can mine useful product recommendations: whenever you observe that someone has put a bathing suit into their shopping cart, you can recommend suntan lotion, as well.



Problem to method mapping

Table 6.1 From problem to approach

Example tasks	Machine learning terminology
Identifying spam email Sorting products in a product catalog Identifying loans that are about to default Assigning customers to preexisting customer clusters	*Classification* —Assigning known labels to objects. Classification is a supervised method, so you need preclassified data in order to train a model.
Predicting the value of AdWords Estimating the probability that a loan will default Predicting how much a marketing campaign will increase traffic or sales Predicting the final price of an auction item based on the final prices of similar products that have been auctioned in the past	*Regression* —Predicting or forecasting numerical values. Regression is also a supervised method, so you need data where the output is known, in order to train a model.
Finding products that are purchased together Identifying web pages that are often visited in the same session Identifying successful (often-clicked) combinations of web pages and AdWords	*Association rules* —Finding objects that tend to appear in the data together. Association rules are an unsupervised method; you do not need data where you already know the relationships, but are trying to discover the relationships within your data.

Example tasks	Machine learning terminology
Identifying groups of customers with the same buying patterns Identifying groups of products that are popular in the same regions or with the same customer clusters Identifying news items that are all discussing similar events	*Clustering* —Finding groups of objects that are more similar to each other than to objects in other groups. Clustering is also an unsupervised method; you do not need pregrouped data, but are trying to discover the groupings within your data.

Prediction Vs Forecasting

- In everyday language, we tend to use the terms *prediction* and *forecasting* interchangeably.
- Technically, to predict is to pick an outcome, such as “It will rain tomorrow,” and to forecast is to assign a probability: “There’s an 80% chance it will rain tomorrow.”
- For unbalanced class applications (such as predicting credit default), the difference is important.
- Consider the case of modeling loan defaults, and assume the overall default rate is 5%.
- Identifying a group that has a 30% default rate is an inaccurate prediction (you don’t know who in the group will default, and most people in the group won’t default), but potentially a very useful forecast (this group defaults at six times the overall rate).

Evaluating Models

- *Test data* is data not used during training, and is intended to give us some experience with how the model will perform on new data.
- One of the things the test set can help you identify is *overfitting*: building a model that memorizes the training data, and does not generalize well to new data.
- A lot of modeling problems are related to overfitting, and looking for signs of overfit is a good first step in diagnosing models.
- Overfitting: An overfit model looks great on the training data and then performs poorly on new data.
- A model's prediction error on the data that it trained from is called *training error*.
- A model's prediction error on new data is called *generalization error*.
- Usually, training error will be smaller than generalization error (no big surprise).
- Ideally, though, the two error rates should be close.
- If generalization error is large, and your model's test performance is poor, then your model has probably *overfit*—it's memorized the training data instead of discovering generalizable rules or patterns

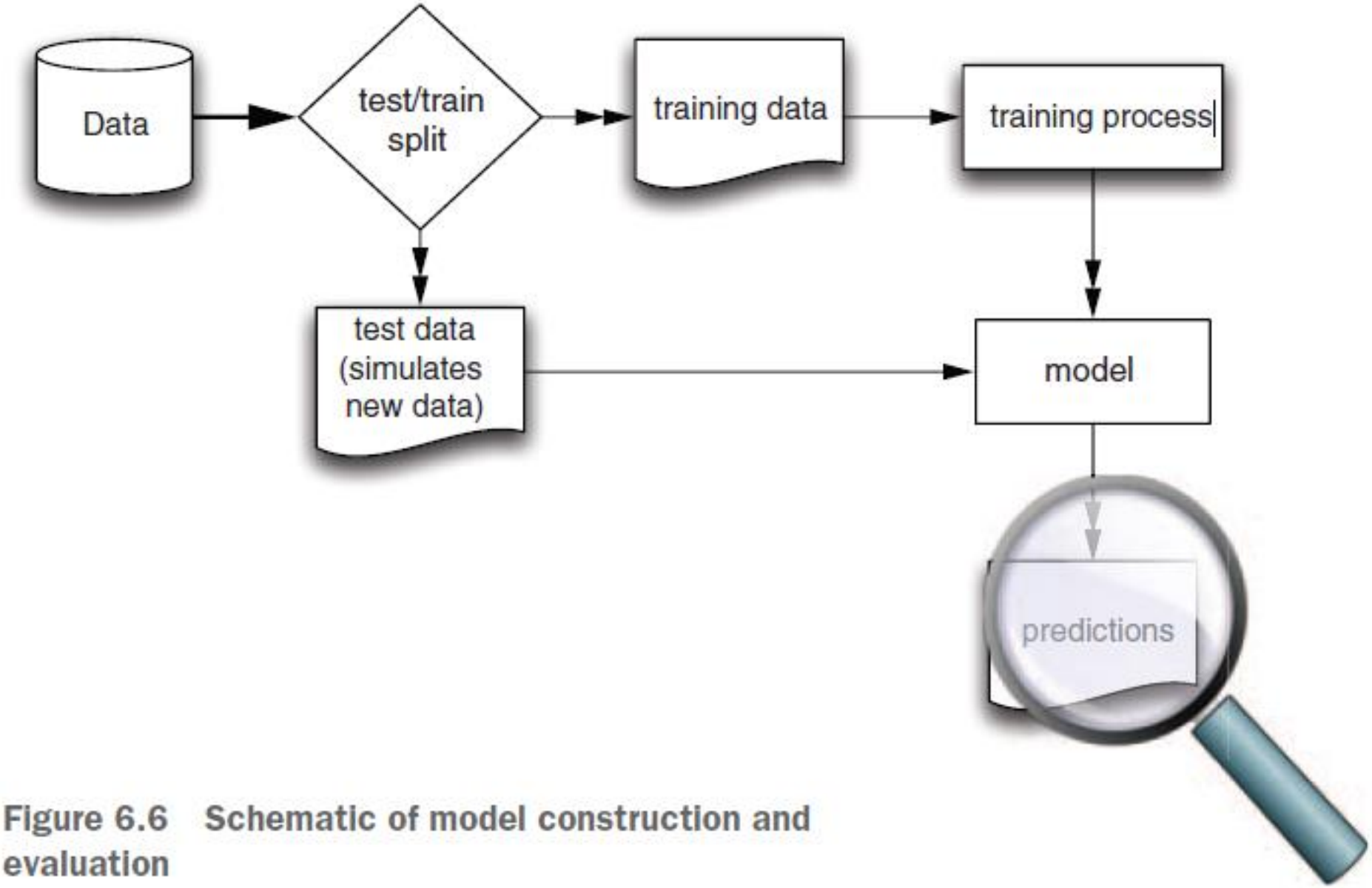


Figure 6.6 Schematic of model construction and evaluation

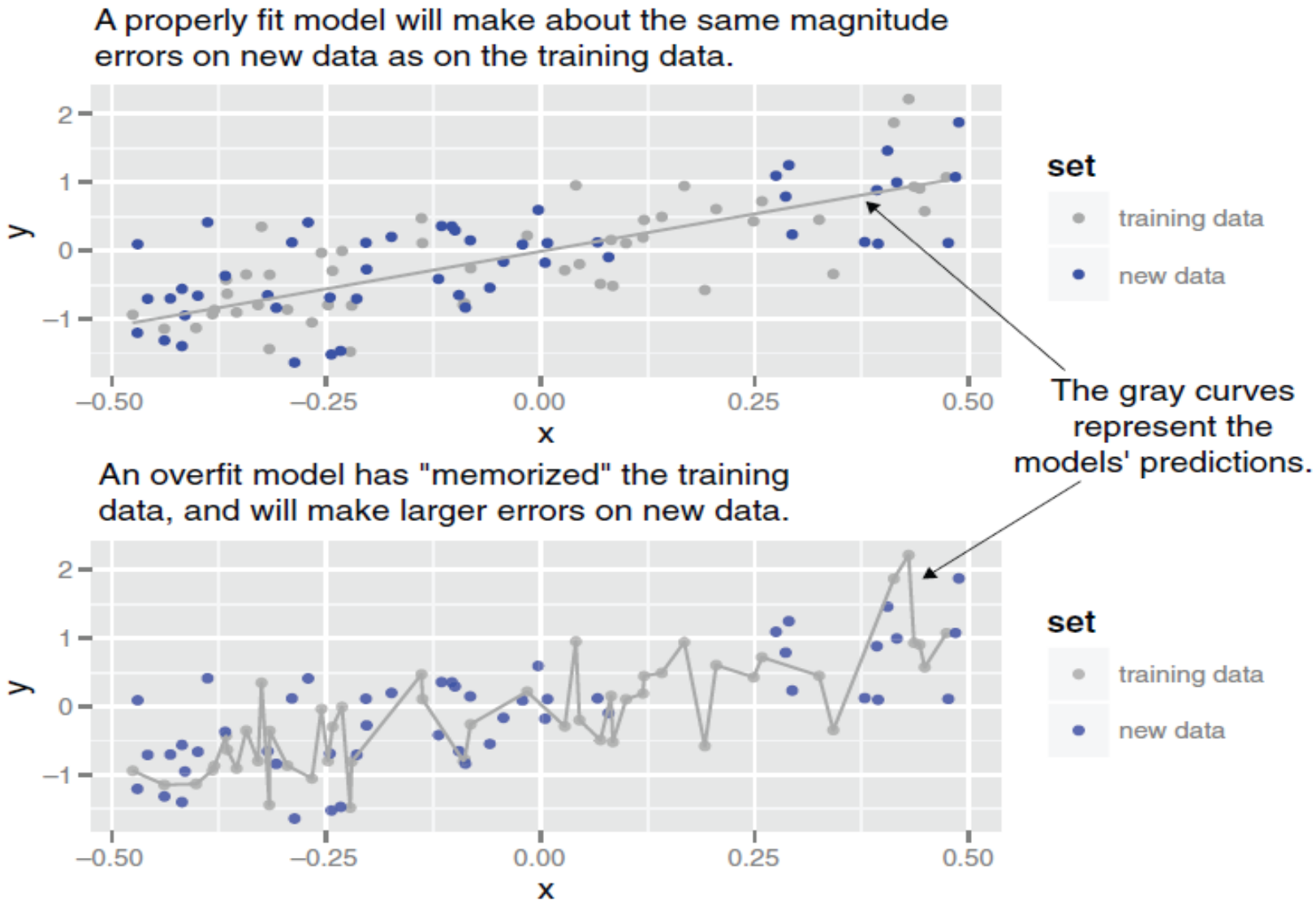


Figure 6.7 A notional illustration of overfitting

Other techniques to prevent overfitting include regularization (preferring small effects from model variables) and bagging (averaging different models to reduce variance).

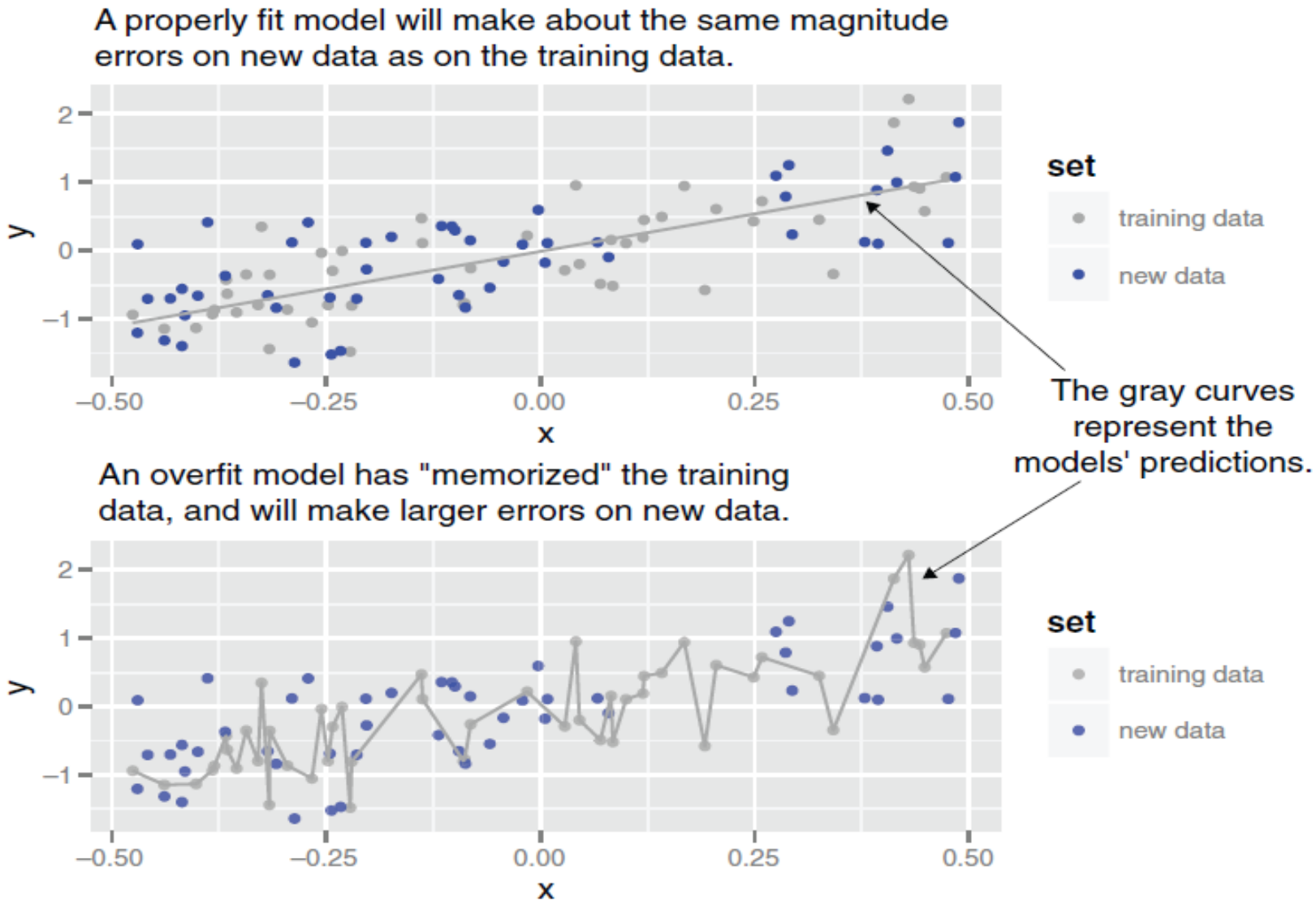


Figure 6.7 A notional illustration of overfitting

Other techniques to prevent overfitting include regularization (preferring small effects from model variables) and bagging (averaging different models to reduce variance).

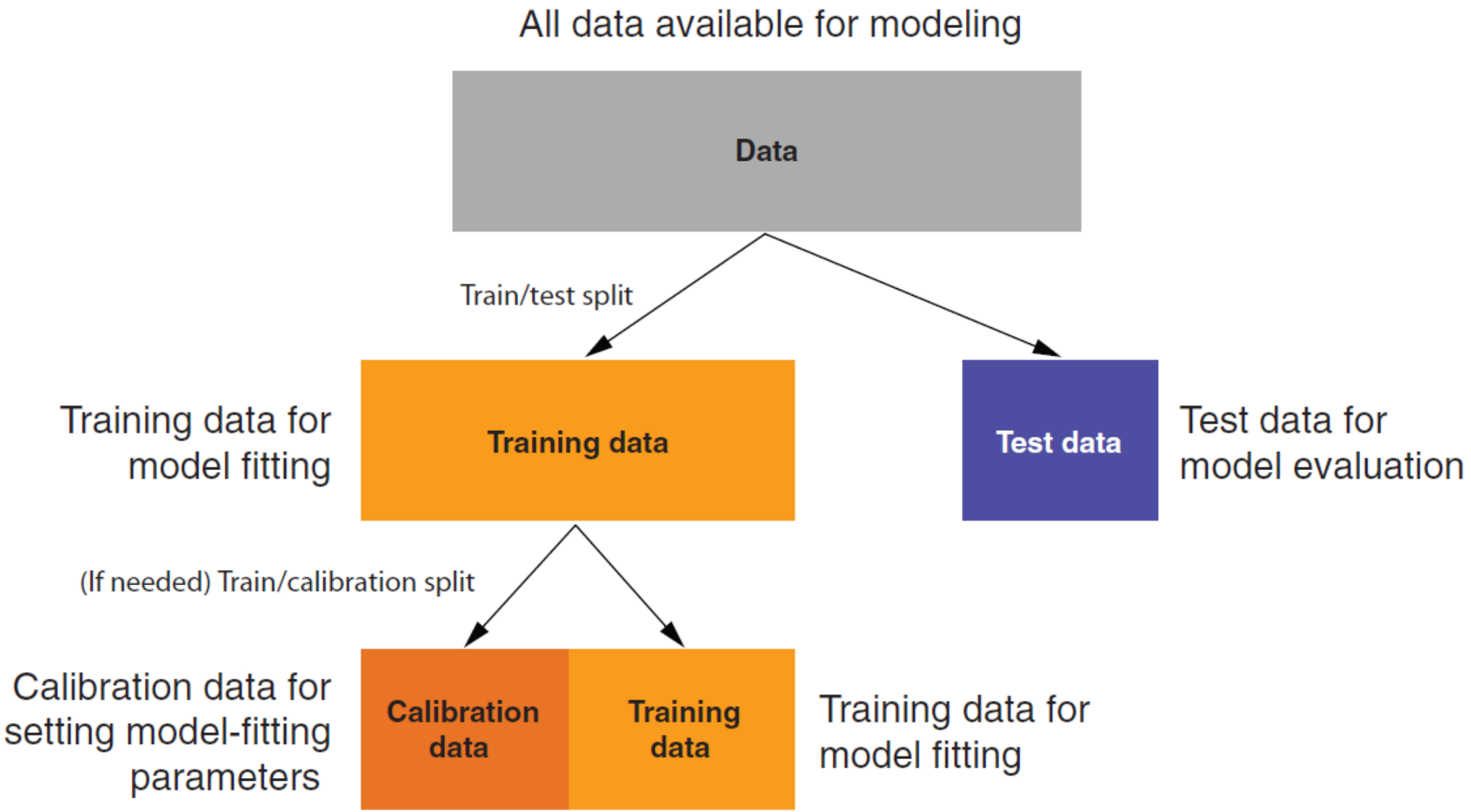


Figure 6.8 Splitting data into training and test (or training, calibration, and test) sets

Splitting data into training and test

- If you do not split your data, but instead use all available data to both train and evaluate each model, then you might think that you will pick the better model, because the model evaluation has seen more data. However, the data used to build a model is not the best data for evaluating the model's performance.
- This is because there's an optimistic *measurement bias* in this data, because this data was seen during model construction.
- Model construction is optimizing your performance measure (or at least something related to your performance measure), so you tend to get exaggerated estimates of performance on your training data.
- A recommended precaution for this optimistic bias is to split your available data into test and training.
- The desire to keep the test data secret for as long as possible is why we often actually split data into training, calibration, and test sets
- When partitioning your data, you want to balance the trade-off between keeping enough data to fit a good model, and holding out enough data to make good estimates of the model's performance.
- Some common splits are 70% training to 30% test, or 80% training to 20% test. For large datasets, you may even sometimes see a 50–50 split

K-Fold Cross Validation

- Testing on holdout data, while useful, uses each example only once: either as part of the model construction or as part of the held-out model evaluation set.
- This is not *statistically efficient*, because the test set is often much smaller than our whole dataset.
- This means we are losing some precision in our estimate of model performance by partitioning our data so simply.
- use a more thorough partitioning scheme called *k-fold cross-validation*.

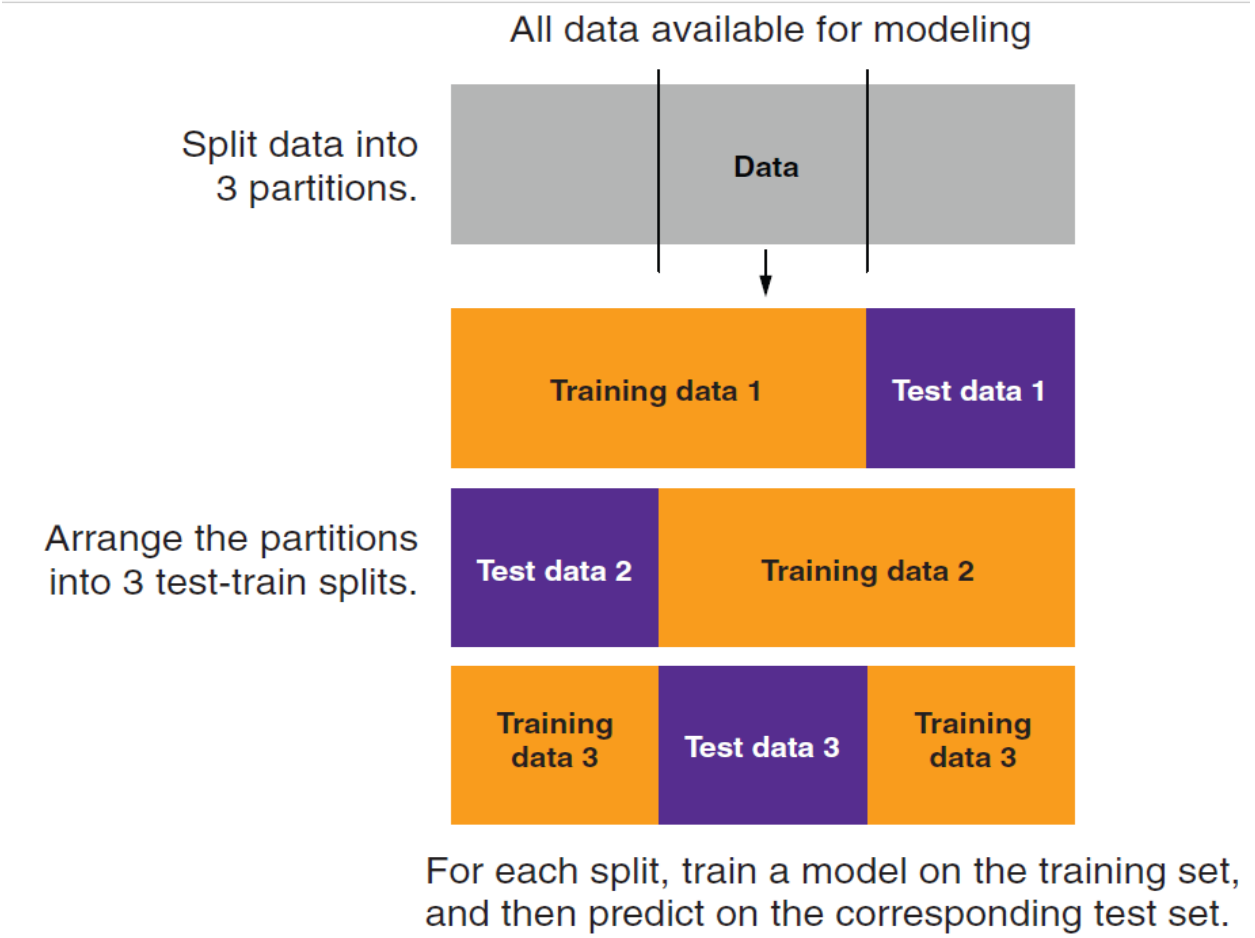


Figure 6.9 Partitioning data for 3-fold cross-validation

Parameters and Hyperparameters

Measures of model performance

- Classification
  - Confusion Matrix
  - Precision and Recall
  - Sensitivity and Specificity
- Scoring
  - Root Mean Square Error
  - R-Squared
- Probability estimation
  - Double Density Plot
  - ROC and the AUC
  - Log Likelihood
  - Deviance
  - AIC
- Clustering

Listing 6.1 Building and applying a logistic regression spam model

```
spamD <- read.table('spamD.tsv',header=T,sep='\t')

spamTrain <-
      subset(spamD,spamD$rgroup  >= 10)
spamTest <- subset(spamD,spamD$rgroup < 10)

spamVars <- setdiff(colnames(spamD), list('rgroup','spam'))
spamFormula <- as.formula(paste('spam == "spam"',
paste(spamVars, collapse = ' + '),sep = ' ~ '))

spamModel <- glm(spamFormula,family = binomial(link = 'logit'),
                                 data = spamTrain)

spamTrain$pred <- predict(spamModel,newdata = spamTrain,
                                 type = 'response')
spamTest$pred <- predict(spamModel,newdata = spamTest,
                                type = 'response')
```

Reads in the data

Splits the data into training and test sets

Creates a formula that describes the model

Fits the logistic regression model

Makes predictions on the training and test sets

The spam model predicts the probability that a given email is spam. A sample of the results of our simple spam classifier is shown in the next listing.

Listing 6.2 Spam classifications

```
sample <- spamTest[c(7,35,224,327), c('spam','pred')]
print(sample)
##           spam         pred
## 115       spam 0.9903246227
## 361       spam 0.4800498077
## 2300 non-spam 0.0006846551
## 3428 non-spam 0.0001434345
```

The first column gives the actual class label (spam or non-spam). The second column gives the predicted probability that an email is spam. If the probability > 0.5, the email is labeled "spam;" otherwise, it is "non-spam."

Evaluating Classification models

- **Confusion Matrix:** The absolute most interesting summary of classifier performance is the confusion matrix. This matrix is just a table that summarizes the classifier's predictions against the actual known data categories.

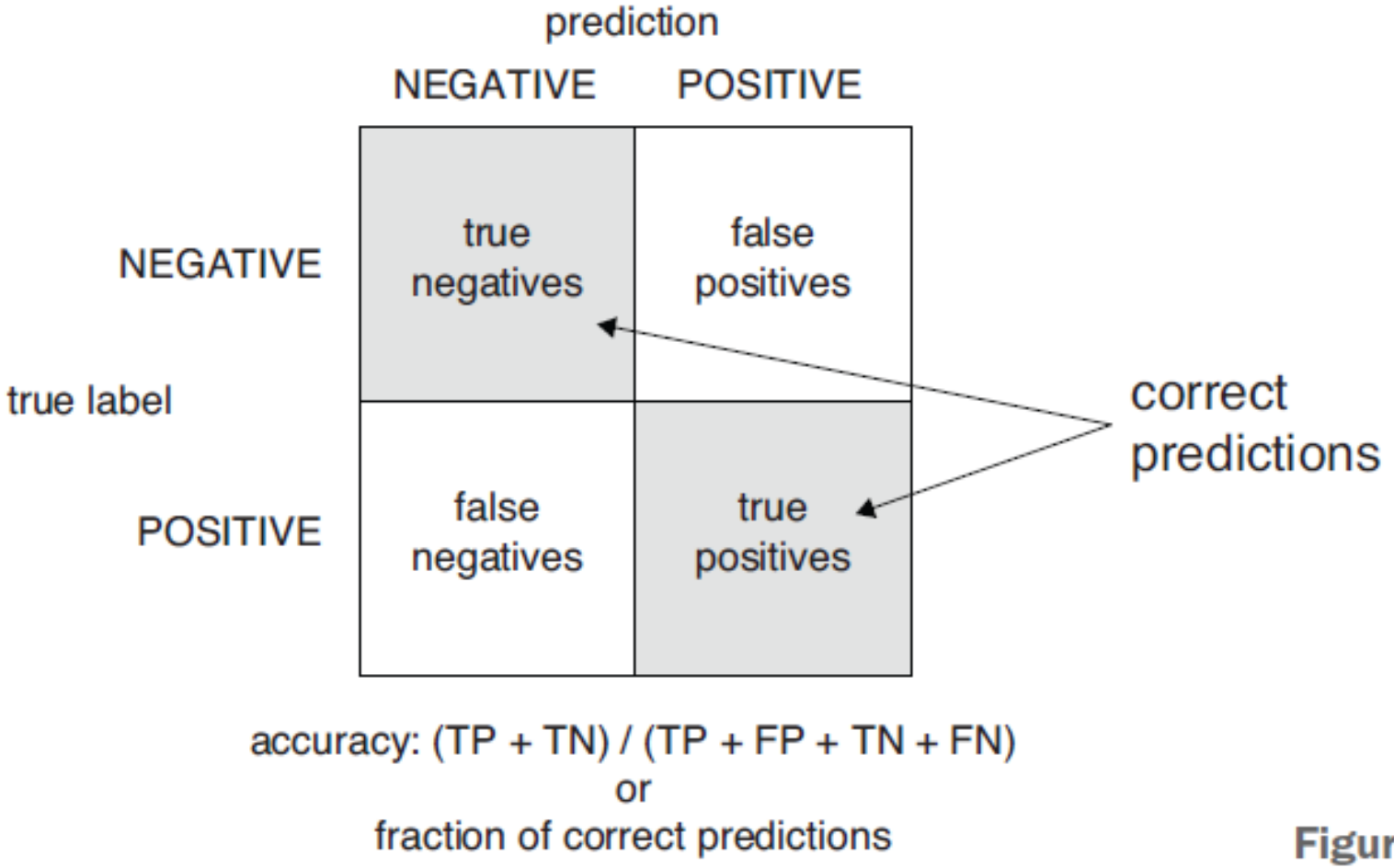


Figure 6.10 Accuracy

- Accuracy is an inappropriate measure for unbalanced classes
- PRECISION : If the spam filter says this email is spam, what's the probability that it's really spam?" Precision is defined as the ratio of true positives to predicted positives.
- **Precision: TP/(TP + FP)**
- precision is a measure of confirmation (when the classifier indicates positive, how often it is in fact correct)
- RECALL: "Of all the spam in the email set, what fraction did the spam filter detect?" Recall is the ratio of true positives over all actual positives
- **Recall: TP/(TP + FN)**
- recall is a measure of utility (how much the classifier finds of what there actually is to find)

F1 Score

- *Suppose that you had multiple spam filters to choose from, each with different values of precision and recall. How do you pick which spam filter to use?*
- The **F1 SCORE** measures a trade-off between precision and recall. It is defined as the harmonic mean of the precision and recall.
- **F1 <- 2 * precision * recall / (precision + recall)**
- F1 is 1.00 when a classifier has perfect precision and recall, and goes to 0.00 for classifiers that have either very low precision or recall (or both).

SENSITIVITY AND SPECIFICITY

- There are situations where a classifier or filter may be used on populations where the prevalence of the positive class (in this example, spam) varies, it's useful to have performance metrics that are independent of the class prevalence.
- One such pair of metrics is ***sensitivity* and *specificity*.**
- This pair of metrics is common in medical research, because tests for diseases and other conditions will be used on different populations, with different prevalence of a given disease or condition.
- *Sensitivity* is also called the *true positive rate* and is exactly equal to recall. **TP/(TP + FN)**
- *Specificity* is also called the *true negative rate*: it is the ratio of true negatives to all negatives. **TN/(TN + FP)**
- Sensitivity and recall answer the question, "What fraction of spam does the spam filter find?"
- Specificity answers the question, "What fraction of non-spam does the spam filter find?"

- One minus the specificity is also called **the *false positive rate***. False positive rate answers the question, “What fraction of non-spam will the model classify as spam?”
- You want the false positive rate to be low (or the specificity to be high), and the sensitivity to also be high.
- Our spam filter has a specificity of about 0.95, which means that it will mark about 5% of non-spam email as spam.
- Sensitivity/specificity is good for fields, like medicine, where it’s important to have an idea how well a classifier, test, or filter separates positive from negative instances independently of the distribution of the different classes in the population. But precision/recall gives you an idea how well a classifier or filter will work on a specific population.
- If you want to know the probability that an email identified as spam is really spam, you have to know how common spam is in that person’s email box, and the appropriate measure is precision.

Listing 6.3 Spam confusion matrix

```
confmat_spam <- table(truth = spamTest$spam,
                      prediction = ifelse(spamTest$pred > 0.5,
                      "spam", "non-spam"))
print(confmat_spam)
##           prediction
## truth     non-spam spam
##   non-spam      264   14
##   spam           22  158
```

Listing 6.4 Entering the Akismet confusion matrix by hand

```
confmat_akismet <- as.table(matrix(data=c(288-1,17,1,13882-17),nrow=2,ncol=2))
rownames(confmat_akismet) <- rownames(confmat_spam)
colnames(confmat_akismet) <- colnames(confmat_spam)
print(confmat_akismet)
##          non-spam  spam
## non-spam      287     1
## spam           17 13865
```

Because the Akismet filter uses link destination clues and determination from other websites (in addition to text features), it achieves a more acceptable accuracy:

```
(confmat_akismet[1,1] + confmat_akismet[2,2]) / sum(confmat_akismet)
## [1] 0.9987297
```

We can calculate the precision of our spam filter as follows:

```
confmat_spam[2,2] / (confmat_spam[2,2]+ confmat_spam[1,2])
## [1] 0.9186047
```

- In email spam example, 92% precision means 8% of what was flagged as spam was in fact not spam.
- This is an unacceptable rate for losing possibly important messages.
- Akismet, on the other hand, had a precision of over 99.99%, so it throws out very little non-spam email

Let's compare the recall of the two spam filters.

```
confmat_spam[2,2] / (confmat_spam[2,2] + confmat_spam[2,1])
## [1] 0.8777778

confmat_akismet[2,2] / (confmat_akismet[2,2] + confmat_akismet[2,1])
## [1] 0.9987754
```

For our email spam filter, this is 88%, which means about 12% of the spam email we receive will still make it into our inbox. Akismet has a recall of 99.88%. In both cases, most spam is in fact tagged (we have high recall) and precision is emphasized over recall. This is appropriate for a spam filter, because it's more important to not lose non-spam email than it is to filter every single piece of spam out of our inbox.

Example *Suppose that you had multiple spam filters to choose from, each with different values of precision and recall. How do you pick which spam filter to use?*

In situations like this, some people prefer to have just one number to compare all the different choices by. One such score is the *F1 score*. The F1 score measures a trade-off between precision and recall. It is defined as the harmonic mean of the precision and recall. This is most easily shown with an explicit calculation:

```
precision <- confmat_spam[2,2] / (confmat_spam[2,2]+ confmat_spam[1,2])
recall <- confmat_spam[2,2] / (confmat_spam[2,2] + confmat_spam[2,1])

(F1 <- 2 * precision * recall / (precision + recall) )
## [1] 0.8977273
```

We can calculate specificity for our spam filter:

```
confmat_spam[1,1] / (confmat_spam[1,1] + confmat_spam[1,2])
## [1] 0.9496403
```

Classifier performance measures business stories.

Measure	Typical business need	Follow-up question
Accuracy	"We need most of our decisions to be correct."	"Can we tolerate being wrong 5% of the time? And do users see mistakes like spam marked as non-spam or non-spam marked as spam as being equivalent?"
Precision	"Most of what we marked as spam had darn well better be spam."	"That would guarantee that most of what is in the spam folder is in fact spam, but it isn't the best way to measure what fraction of the user's legitimate email is lost. We could cheat on this goal by sending all our users a bunch of easy-to-identify spam that we correctly identify. Maybe we really want good specificity."
Recall	"We want to cut down on the amount of spam a user sees by a factor of 10 (eliminate 90% of the spam)."	"If 10% of the spam gets through, will the user see mostly non-spam mail or mostly spam? Will this result in a good user experience?"
Sensitivity	"We have to cut a lot of spam; otherwise, the user won't see a benefit."	"If we cut spam down to 1% of what it is now, would that be a good user experience?"
Specificity	"We must be at least *three nines* on legitimate email; the user must see at least 99.9% of their non-spam email."	"Will the user tolerate missing 0.1% of their legitimate email, and should we keep a spam folder the user can look at?"

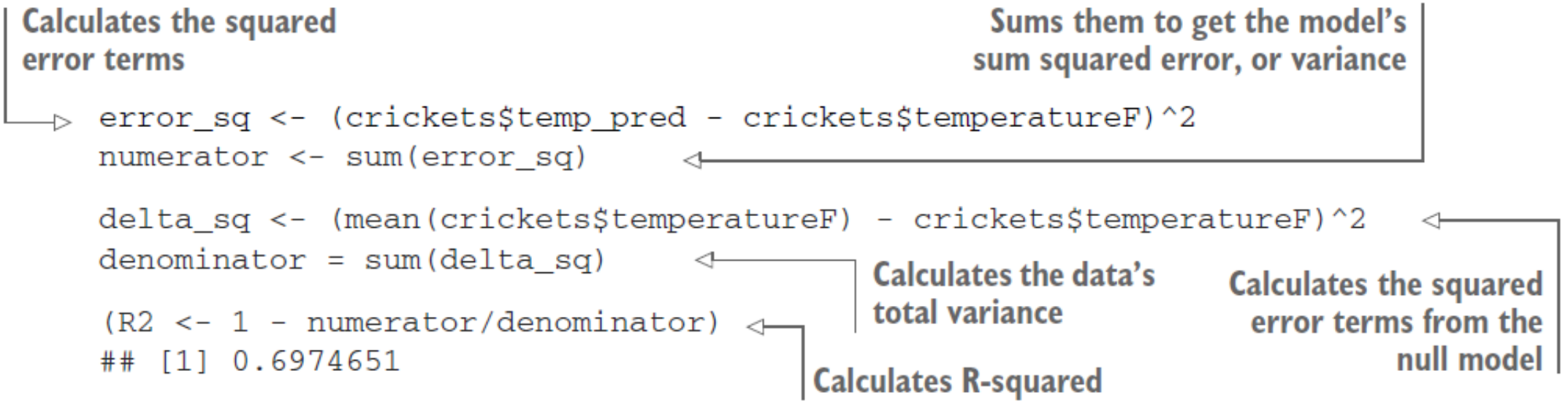
Evaluating Scoring models

- *Suppose you've read that the rate at which crickets chirp is proportional to the temperature, so you have gathered some data and fit a model that predicts temperature (in Fahrenheit) from the chirp rate (chirps/sec)*
- fit a linear regression model to this data, and then make predictions
- The differences between the predictions of `temperatureF` and `temp_pred` are called the *residuals* or *error* of the model on the data. We will use the residuals to calculate some common performance metrics for scoring models
- **Root Mean Square Error:** The most common goodness-of-fit measure is called *root mean square error (RMSE)*. The RMSE is the square root of the average squared residuals (also called the mean squared error). RMSE answers the question, "How much is the predicted temperature typically off?"
- The RMSE is in the same units as the outcome: since the outcome (temperature) is in degrees Fahrenheit, the RMSE is also in degrees Fahrenheit.
- Here the RMSE tells you that the model's predictions will typically (that is, on average) be about 3.6 degrees off from the actual temperature. Suppose that you consider a model that typically predicts the temperature to within 5 degrees to be "good."
- `error_sq <- (crickets$temp_pred - crickets$temperatureF)^2`
- `( RMSE <- sqrt(mean(error_sq)) )`
- The quantity `mean(error_sq)` is called the *mean squared error*.
- We will call the quantity `sum(error_sq)` the *sum squared error*, and also refer to it as the model's *variance*.

Listing 6.7 Calculating RMSE

```
error_sq <- (crickets$temp_pred - crickets$temperatureF)^2
( RMSE <- sqrt(mean(error_sq)) )
## [1] 3.564149
```

Listing 6.8 Calculating R-squared



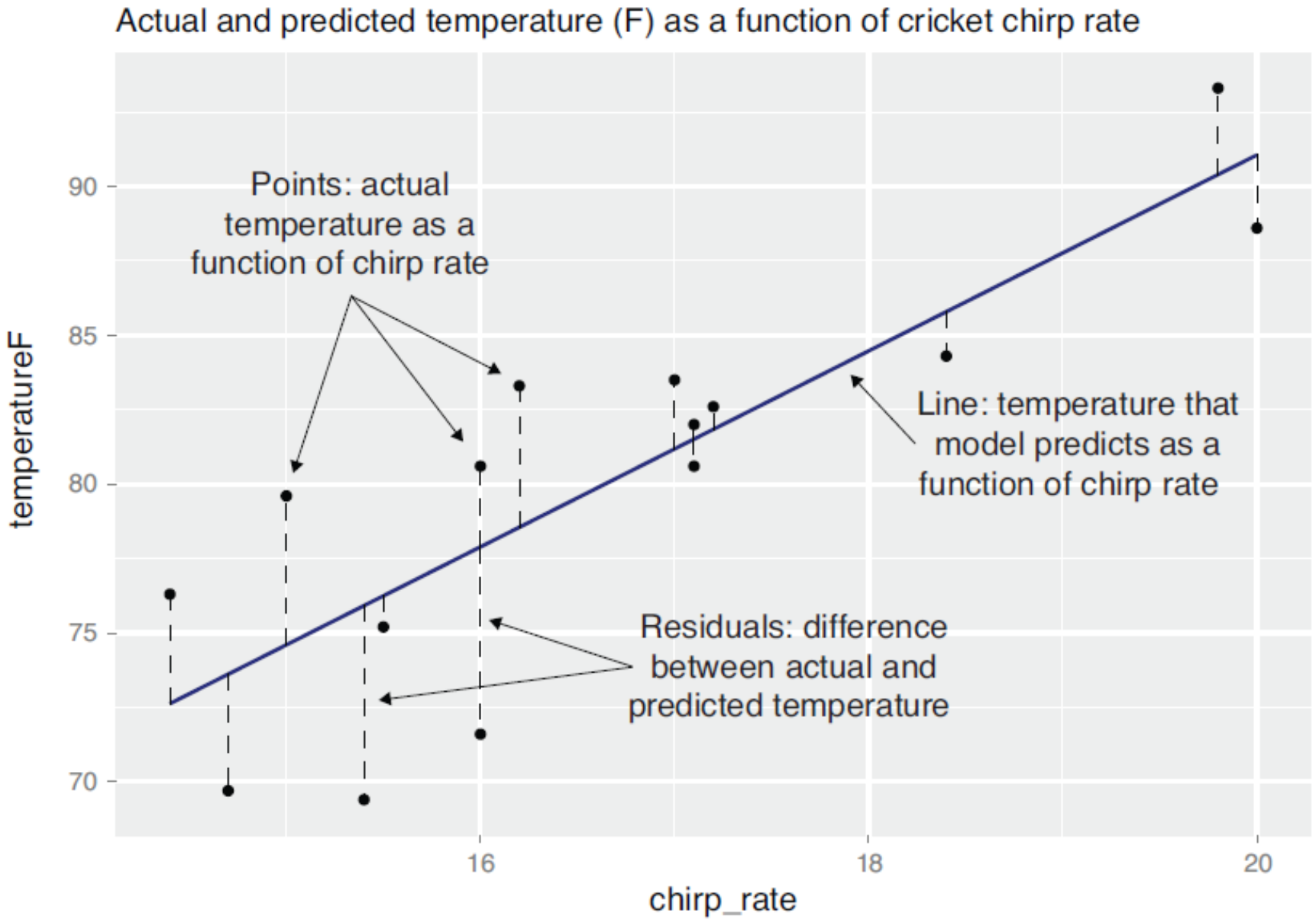


Figure 6.14 Scoring residuals

R-SQUARED

- Another important measure of fit is called *R-squared* (or *R2*, or the *coefficient of determination*).
- Simplest baseline prediction of the temperature is simply the average temperature in the dataset. This is the *null model*;
- it's not a very good model, but you have to perform at least better than it does.
- The data's *total variance* is the sum squared error of the null model. You want the sum squared error of your actual model to be much smaller than the data's variance—that is, you want the ratio of your model's sum squared error to the total variance to be near zero.
- R-squared is defined as one minus this ratio, so we want R-squared to be close to one.
- As R-squared is formed from a ratio comparing your model's variance to the total variance, you can think of R-squared as a measure of how much variance your model "explains."
- R-squared is also sometimes referred to as a measure of how well the model "fits" the data, or its "goodness of fit."
- The best possible R-squared is 1.0, with near-zero or negative R-squareds being horrible.
- Some other models (such as logistic regression) use deviance to report an analogous quantity called *pseudo R-squared*.
- Under certain circumstances, R-squared is equal to the square of another measure called the *correlation*
- A good statement of a R-squared business goal would be "We want the model to explain at least 70% of variation in account value."

Listing 6.8 Calculating R-squared

Calculates the squared error terms

Sums them to get the model's sum squared error, or variance

```
error_sq <- (crickets$temp_pred - crickets$temperatureF)^2
numerator <- sum(error_sq)

delta_sq <- (mean(crickets$temperatureF) - crickets$temperatureF)^2
denominator = sum(delta_sq)

(R2 <- 1 - numerator/denominator)
## [1] 0.6974651
```

Calculates the data's total variance

Calculates the squared error terms from the null model

Calculates R-squared

Evaluating Probability models

- Probability models are models that both decide if an item is in a given class and return an estimated probability (or confidence) of the item being in the class.
- The modelling techniques of logistic regression and decision trees are fairly famous for being able to return good probability estimates
- *Suppose that, while building your spam filter, you try several different algorithms and modeling approaches and come up with several models, all of which return the probability that a given email is spam. You want to compare these different models quickly and identify the one that will make the best spam filter.*
- In order to turn a probability model into a classifier, you need to select a threshold: items that score higher than that threshold will be classified as spam; otherwise, they are classified as non-spam.
- The easiest (and probably the most common) threshold for a probability model is 0.5, but the “best possible” classifier for a given probability model may require a different threshold.
- This optimal threshold can vary from model to model. The metrics in this section compare probability models directly, without having turned them into classifiers.
- If you make the reasonable assumption that the best probability model will make the best classifier, then you can use these metrics to quickly select the most appropriate probability model, and then spend some time tuning the threshold to build the best classifier for your needs.

THE DOUBLE DENSITY PLOT:

Double density plots can be useful when picking classifier thresholds, or the threshold score where the classifier switches from labeling an email as non-spam to spam.

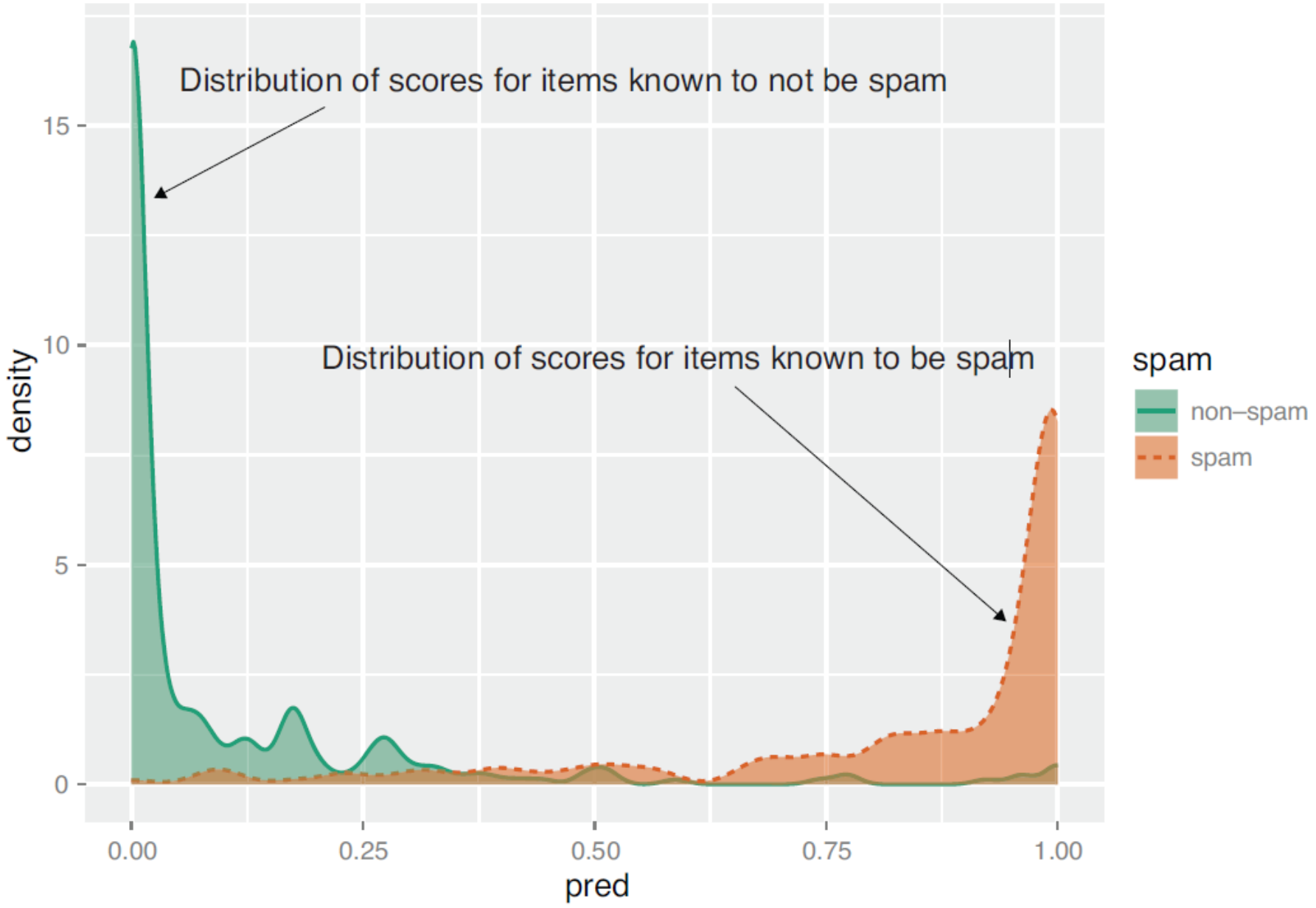


Figure 6.15 Distribution of scores broken up by known classes

Listing 6.9 Making a double density plot

```
library(WVPlots)
DoubleDensityPlot(spamTest,
                  xvar = "pred",
                  truthVar = "spam",
                  title = "Distribution of scores for spam filter")
```

The x-axis in the figure corresponds to the prediction scores returned by the spam filter. Figure 6.15 illustrates what we're going to try to check when evaluating estimated probability models: examples in the class should mostly have high scores, and examples not in the class should mostly have low scores.

THE RECEIVER OPERATING CHARACTERISTIC CURVE AND THE AUC

- The *receiver operating characteristic curve* (or *ROC curve*) is a popular alternative to the double density plot.
- For each different classifier we'd get by picking a different score threshold between spam and not-spam, we plot both the true positive (TP) rate and the false positive (FP) rate.
- The resulting curve represents every possible trade-off between true positive rate and false positive rate that is available for classifiers derived from this model.
- In the last line of the listing, we compute the *AUC* or *area under the curve*, which is another measure of the quality of the model.

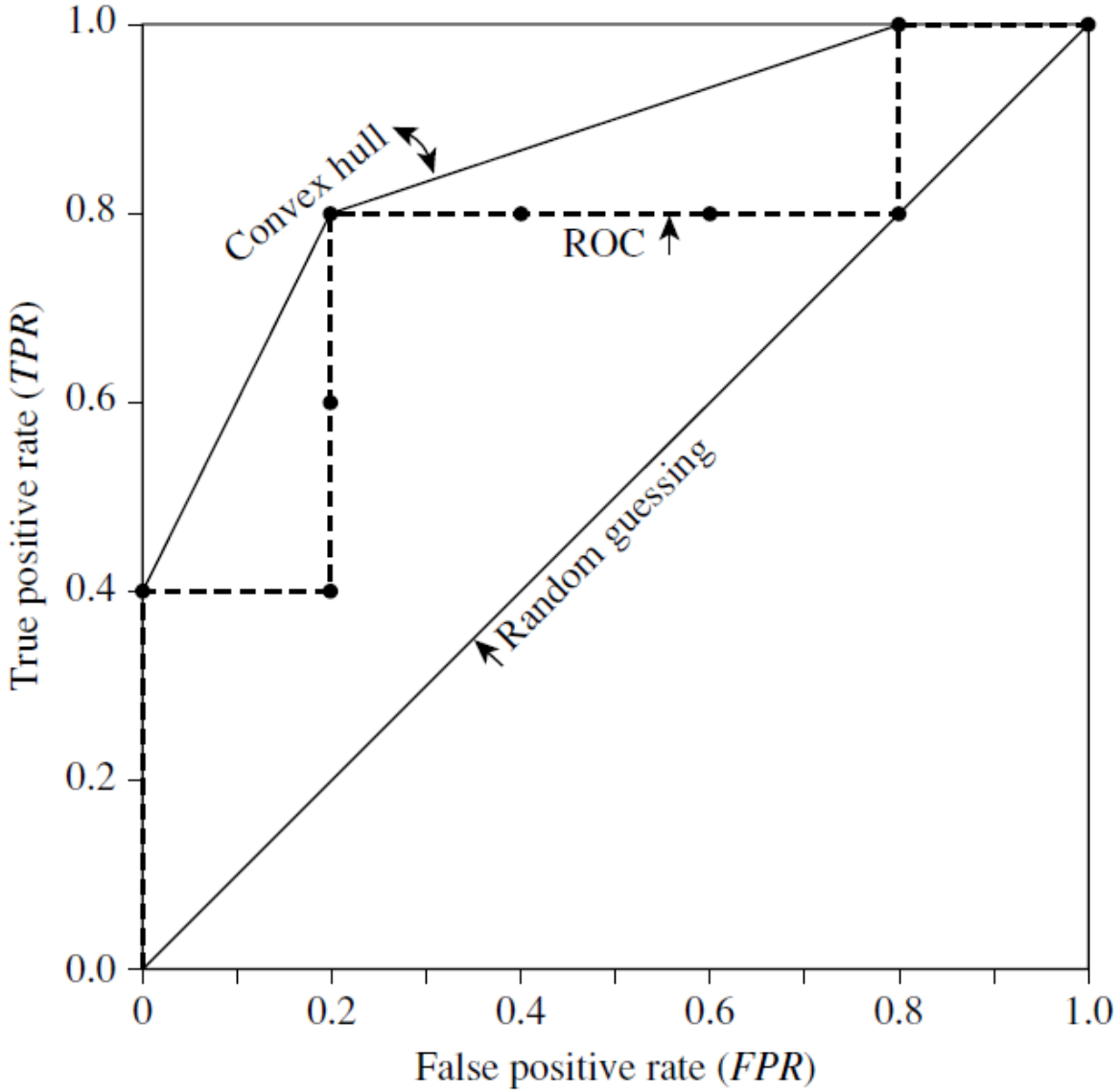
Plotting an ROC curve. Figure 8.18 shows the probability value (column 3) returned by a probabilistic classifier for each of the 10 tuples in a test set, sorted by decreasing probability order. Column 1 is merely a tuple identification number, which aids in our explanation. Column 2 is the actual class label of the tuple. There are five positive tuples and five negative tuples, thus $P = 5$ and $N = 5$. As we examine the known class label of each tuple, we can determine the values of the remaining columns, *TP*, *FP*, *TN*, *FN*, *TPR*, and *FPR*. We start with tuple 1, which has the highest probability score, and take that score as our threshold, that is, $t = 0.9$. Thus, the classifier considers tuple 1 to be positive, and all the other tuples are considered negative. Since the actual class label of tuple 1 is positive, we have a true positive, hence $TP = 1$ and $FP = 0$. Among the

Tuple #	Class	Prob.	TP	FP	TN	FN	TPR	FPR
1	*P*	0.90	1	0	5	4	0.2	0
2	*P*	0.80	2	0	5	3	0.4	0
3	*N*	0.70	2	1	4	3	0.4	0.2
4	*P*	0.60	3	1	4	2	0.6	0.2
5	*P*	0.55	4	1	4	1	0.8	0.2
6	*N*	0.54	4	2	3	1	0.8	0.4
7	*N*	0.53	4	3	2	1	0.8	0.6
8	*N*	0.51	4	4	1	1	0.8	0.8
9	*P*	0.50	5	4	0	1	1.0	0.8
10	*N*	0.40	5	5	0	0	1.0	1.0

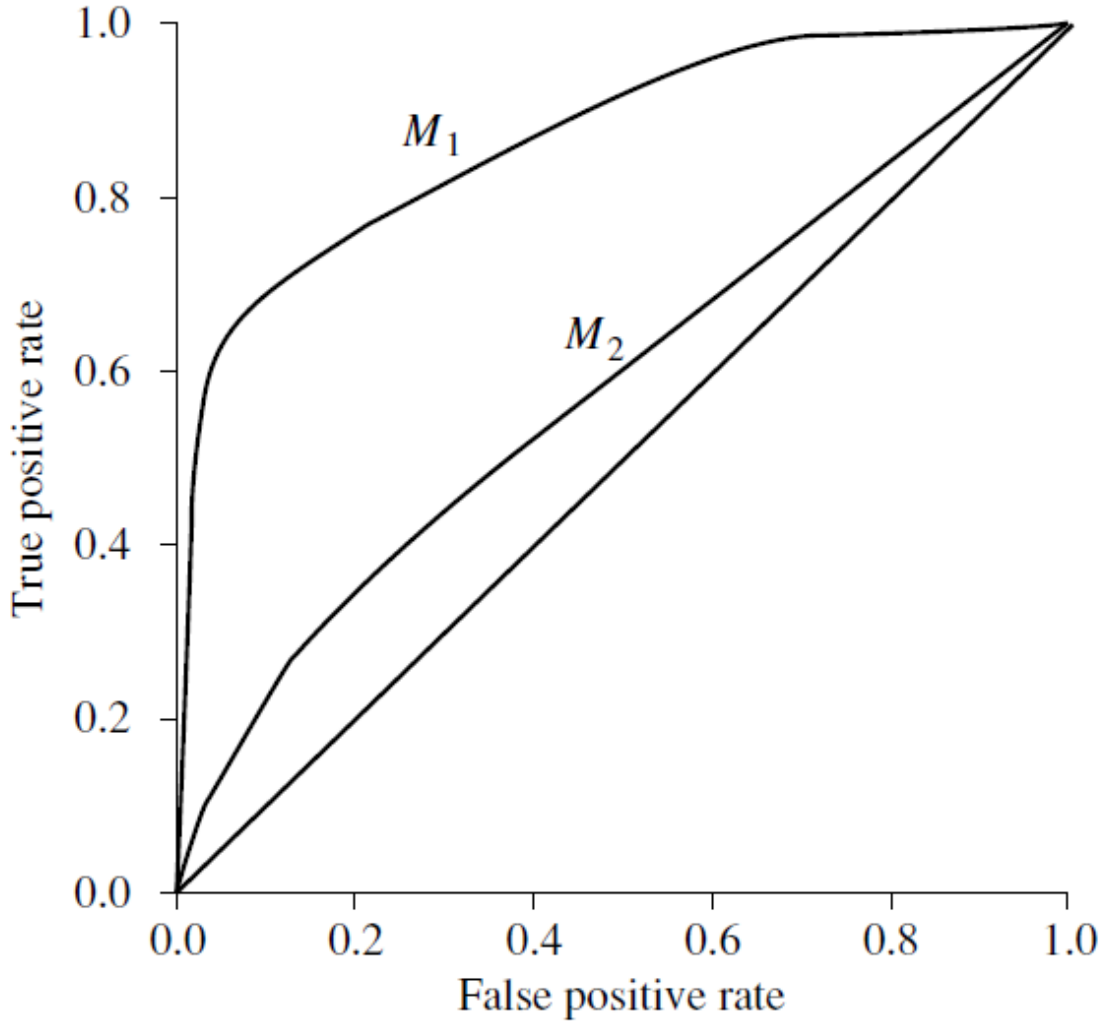
Tuples sorted by decreasing score, where the score is the value returned by a probabilistic classifier.

ROC-AUC

- Among the remaining nine tuples, which are all classified as negative, five actually are negative (thus, *TN* = 5).
- The remaining four are all actually positive, thus, *FN* D 4. We can therefore compute *TPR* = *TP/P* 1/5=0.2, while *FPR* = 0. Thus, we have the point (0.2,0) for the ROC curve.
- Next, threshold *t* is set to 0.8, the probability value for tuple 2, so this tuple is now also considered positive, while tuples 3 through 10 are considered negative.
- The actual class label of tuple 2 is positive, thus now *TP* = 2. The rest of the row can easily be computed, resulting in the point (0.4,0).
- Next, we examine the class label of tuple 3 and let *t* be 0.7, the probability value returned by the classifier for that tuple. Thus, tuple 3 is considered positive, yet its actual label is negative, and so it is a false positive.
- Thus, *TP* stays the same and *FP* increments so that *FP* = 1. The rest of the values in the row can also be easily computed, yielding the point (0.4, 0.2).



ROC curve for the data in Figure 8.18.



ROC curves of two classification models, M_1 and M_2. The diagonal shows where, for every true positive, we are equally likely to encounter a false positive. The closer an ROC curve is to the diagonal line, the less accurate the model is. Thus, $M1$ is more accurate here.

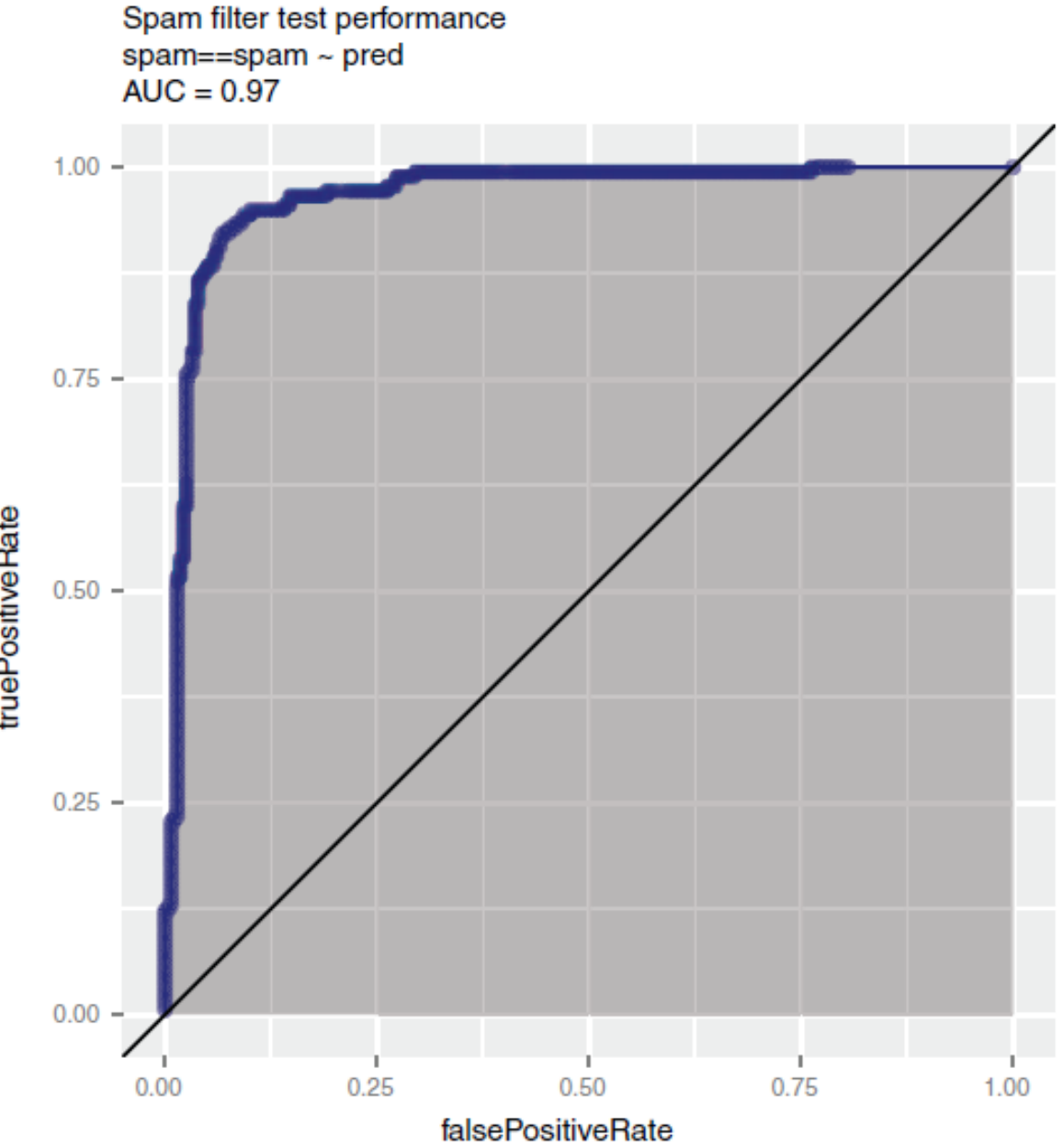


Figure 6.16 ROC curve for the email spam example

Listing 6.10 Plotting the receiver operating characteristic curve

```
library(WVPlots)
ROCPlot(spamTest,
        xvar = 'pred',
        truthVar = 'spam',
        truthTarget = 'spam',
        title = 'Spam filter test performance')

library(sigr)
calcAUC(spamTest$pred, spamTest$spam=='spam')
 ## [1] 0.9660072
```

Plots the receiver operating characteristic (ROC) curve

Calculates the area under the ROC curve explicitly

How Does the AUC-ROC Curve Work?

- In a ROC curve, a higher X-axis value indicates a higher number of False positives than True negatives.
- While a higher Y-axis value indicates a higher number of True positives than False negatives.
- So, the choice of the threshold depends on the ability to balance False positives and False negatives.
- Thus, the closer the ROC curve of a model is to the diagonal line, the less accurate the model.
- To assess the accuracy of a model, we can measure the area under the curve.

The reasoning behind the AUC

- At one end of the spectrum of models is the ideal perfect model that would return a score of 1 for spam emails and a score of 0 for non-spam. This ideal model would form an ROC with three points:
- (0,0)—Corresponding to a classifier defined by the threshold p = 1: nothing gets classified as spam, so this classifier has a zero false positive rate and a zero true positive rate.
- (1,1)—Corresponding to a classifier defined by the threshold p = 0: everything gets classified as spam, so this classifier has a false positive rate of 1 and a true positive rate of 1.
- (0,1)—Corresponding to any classifier defined by a threshold between 0 and 1: everything is classified correctly, so this classifier has a false positive rate of 0 and a true positive rate of 1.

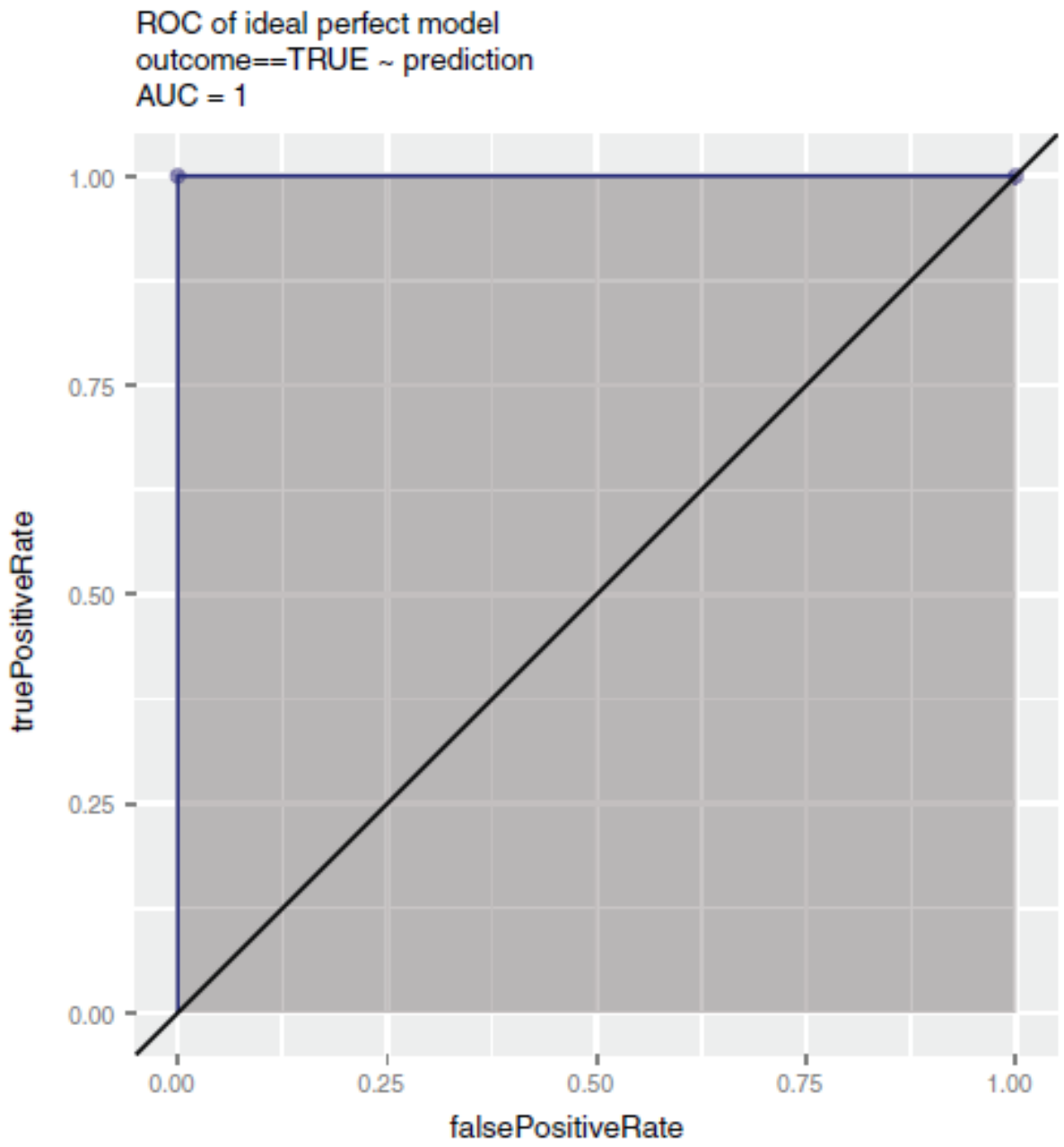


Figure 6.17 ROC curve for an ideal model that classifies perfectly

LOG LIKELIHOOD

- *Log likelihood* is a measure of how well the model's predictions "match" the true class labels.
- It is a non-positive number, where a log likelihood of 0 means a perfect match: the model scores all the spam as being spam with a probability of 1, and all the nonspam as having a probability 0 of being spam.
- The larger the magnitude of the log likelihood, the worse the match.

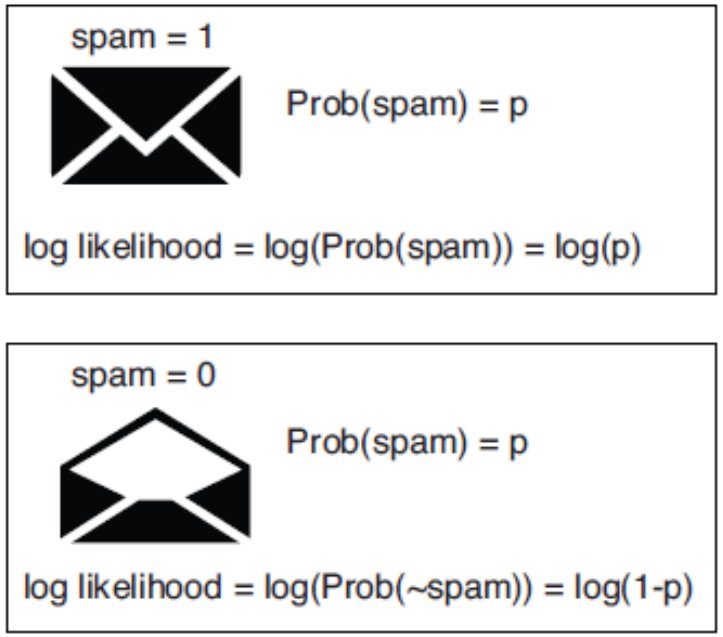


Figure 6.18 Log likelihood of a spam filter prediction

The log likelihood of a model's predictions on an entire dataset is the sum of the individual log likelihoods:

```
log_likelihood = sum(y * log(py) + (1-y) * log(1 - py))
```

Here `y` is the true class label (0 for non-spam and 1 for spam) and `py` is the probability that an instance is of class 1 (spam). We are using multiplication to select the correct logarithm. We also use the convention that `0 * log(0) = 0` (though for simplicity, this isn't shown in the code).

Listing 6.11 Calculating log likelihood

```
ylogpy <- function(y, py) {
   logpy = ifelse(py > 0, log(py), 0)
  y*logpy
}

y <- spamTest$spam == 'spam'

sum(ylogpy(y, spamTest$pred) +
      ylogpy(1-y, 1-spamTest$pred))
## [1] -134.9478
```

A function to calculate y * log(py), with the convention that 0 * log(0) = 0

Gets the class labels of the test set as TRUE/FALSE, which R treats as 1/0 in arithmetic operations

Calculates the log likelihood of the model's predictions on the test set

- The log likelihood is useful for comparing multiple probability models *on the same test dataset*---because the log likelihood is an unnormalized sum, its magnitude implicitly depends on the size of the dataset, so you can't directly compare log likelihoods that were computed on different datasets.
- When comparing multiple models, you generally want to prefer models with a larger (that is, smaller magnitude) log likelihood.

Deviance

- Another common measure when fitting probability models is the *deviance*.
- The deviance is defined as `-2*(logLikelihood-S)`, where `S` is a technical constant called "the log likelihood of the saturated model."
- In most cases, the saturated model is a perfect model that returns probability 1 for items in the class and probability 0 for items not in the class (so `S=0`).
- The lower the deviance, the better the model.
- We're most concerned with ratios of deviance, such as the ratio between the null deviance and the model deviance
- Like the log likelihood, deviance is unnormalized, so you should only compare deviances that are computed over the same dataset.
- When comparing multiple models, you will generally prefer models with smaller deviance.
- The pseudo R-squared is normalized (it's a function of a ratio of deviances), so in principle you can compare pseudo R-squareds even when they were computed over different test sets.
- When comparing multiple models, you will generally prefer models with larger pseudo R-squareds.

Listing 6.13 Computing the deviance and pseudo R-squared

```
library(sigr)

(deviance <- calcDeviance(spamTest$pred, spamTest$spam == 'spam'))
## [1] 253.8598
(nullDeviance <- calcDeviance(pNull, spamTest$spam == 'spam'))
## [1] 613.7929

(pseudoR2 <- 1 - deviance/nullDeviance)
## [1] 0.586408
```

Like the log likelihood, deviance is unnormalized, so you should only compare deviances that are computed over the same dataset. When comparing multiple models, you will generally prefer models with smaller deviance. The pseudo R-squared is normalized (it's a function of a ratio of deviances), so in principle you can compare pseudo R-squareds even when they were computed over different test sets. When comparing multiple models, you will generally prefer models with larger pseudo R-squareds.

AIC: Akaike information criterion

- An important variant of deviance is the *Akaike information criterion* (*AIC*).
- This is equivalent to `deviance + 2*numberOfParameters` used in the model.
- The more parameters in the model, the more complex the model is; the more complex a model is, the more likely it is to overfit.
- Thus, AIC is deviance penalized for model complexity.
- When comparing models (on the same test set), you will generally prefer the model with the smaller AIC.
- The AIC is useful for comparing models with different measures of complexity and modeling variables with differing numbers of levels

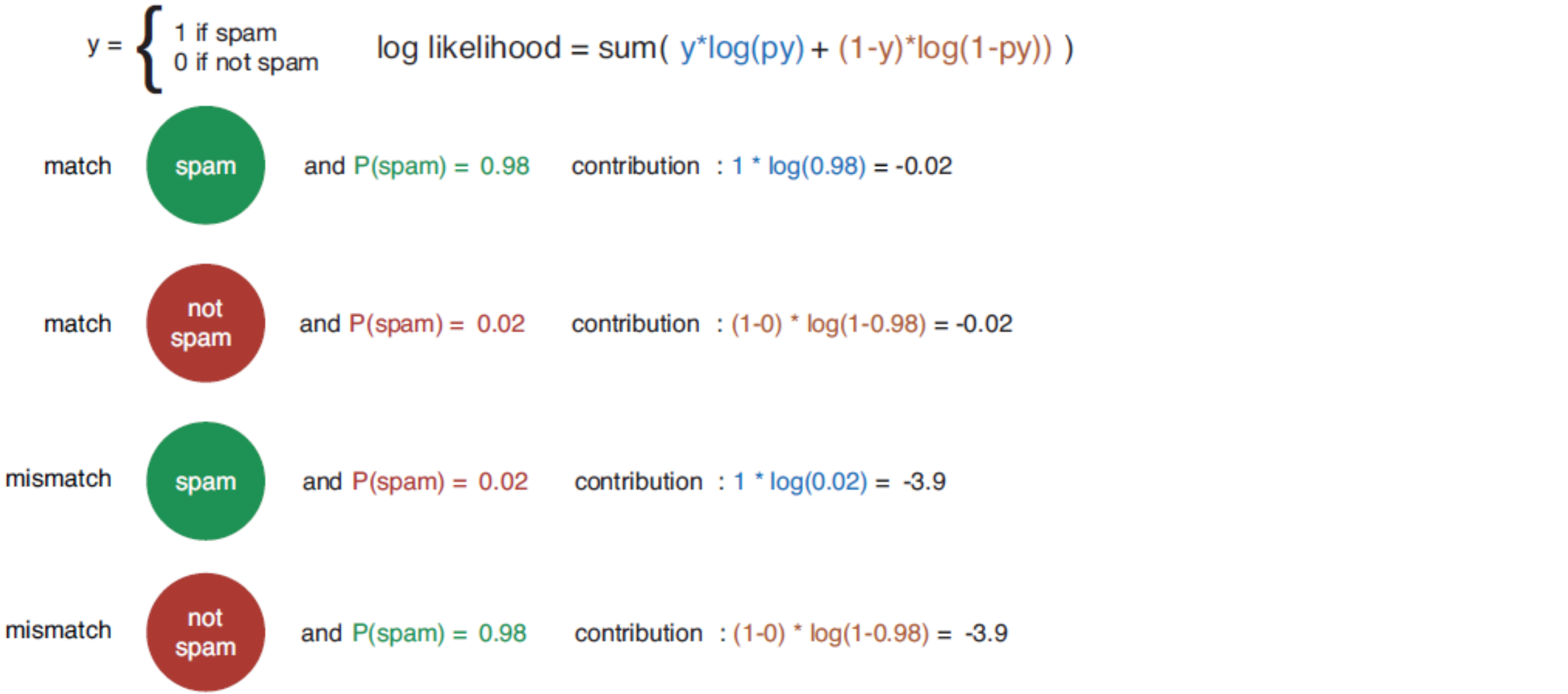


Figure 6.19 Log likelihood penalizes mismatches between the prediction and the true class label.

Local Interpretable Model-agnostic Explanations (LIME) for explaining model predictions

- In many people's opinion, the improved prediction performance of modern machine learning methods like deep learning or gradient boosted trees comes at the cost of decreased explanation
- A human domain expert can review the if-then structure of a decision tree and compare it to their own decision-making processes to decide if the decision tree will make reasonable decisions.
- However, other methods have far more complex structures that are difficult for a human to evaluate.
- Examples include the multiple individual trees of a random forest

LIME

- real-world example is Amazon's recent attempt to automate resume reviews, using the resumes of people hired by Amazon over a 10-year period as training data.
- As Reuters reported, the company discovered that their model was discriminating against women.
- It penalized resumes that included words like "women's," and downvoted applicants who had graduated from two particular all-women's colleges.
- Researchers also discovered that the algorithm ignored common terms that referred to specific skills (such as the names of computer programming languages), and favored words like *executed* or *captured* that were disproportionately used by male applicants.
- the flaw was not in the machine learning algorithm, but in the training data, which had apparently captured existing biases in Amazon's hiring practices— which the model then codified.
- Prediction explanation techniques like LIME can potentially discover such issues.
- LIME produces an "explanation" of a model's prediction on a specific datum.
- That is, LIME tries to determine which features of that datum contributed the most to the model's decision about it.
- This helps data scientists attempt to understand the behavior of black-box machine learning models

LIME: IRIS setosa- *The object is to predict whether a given iris is a setosa based on its petal and sepal dimensions.*

Listing 6.14 Loading the iris dataset

```
iris <- iris

iris$class <- as.numeric(iris$Species == "setosa")

set.seed(2345)
intrain <- runif(nrow(iris)) < 0.75

train <- iris[intrain,]
test <- iris[!intrain,]

head(train)
```

Setosa is the positive class.

Uses 75% of the data for training, the remainder as holdout (test data)

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species class
## 1           5.1         3.5          1.4         0.2  setosa     1
## 2           4.9         3.0          1.4         0.2  setosa     1
## 3           4.7         3.2          1.3         0.2  setosa     1
## 4           4.6         3.1          1.5         0.2  setosa     1
## 5           5.0         3.6          1.4         0.2  setosa     1
```

Listing 6.15 Fitting a model to the iris training data

```
source("lime_iris_example.R")
```

Loads the convenience function

```
input <- as.matrix(train[, 1:4])
model <- fit_iris_example(input, train$class)
```

The input to the model is the first four columns of the training data, converted to a matrix.

After you fit the model, you can evaluate the model on the test data. The model's predictions are the probability that a given iris is *setosa*.

Listing 6.16 Evaluating the iris model

```
predictions <- predict(model, newdata=as.matrix(test[,1:4]))

teframe <- data.frame(isSetosa = ifelse(test$class == 1,
                                        "setosa",
                                        "not setosa"),
                      pred = ifelse(predictions > 0.5,
                                    "setosa",
                                    "not setosa"))
with(teframe, table(truth=isSetosa, pred=pred))
```

A data frame of predictions and actual outcome

Makes predictions on the test data. The predictions are the probability that an iris is a setosa.

Examines the confusion matrix

```
##              pred
## truth         not setosa setosa
##   not setosa          25      0
##   setosa               0     11
```

Listing 6.17 Building a LIME explainer from the model and training data

```
library(lime)
explainer <- lime(train[,1:4],
                  model = model,
                  bin_continuous = TRUE,
                  n_bins = 10)
```

Builds the explainer from the training data

Bins the continuous variables when making explanations

Uses 10 bins

Now pick a specific example from the test set.

Listing 6.18 An example iris datum

```
(example <- test[5, 1:4, drop=FALSE])
##    Sepal.Length Sepal.Width Petal.Length Petal.Width
## 30          4.7         3.2          1.6         0.2

test$class[5]
## [1] 1
```

A single row data frame

This example is a setosa.

```
round(predict(model, newdata = as.matrix(example)))
## [1] 1
```

And the model predicts that it is a setosa.

Now explain the model's prediction on `example`. Note that the `dplyr` package also has a function called `explain()`, so if you have `dplyr` in your namespace, you may get a conflict trying to call `lime`'s `explain()` function. To prevent this ambiguity, specify the function using namespace notation: `lime::explain(...)`.

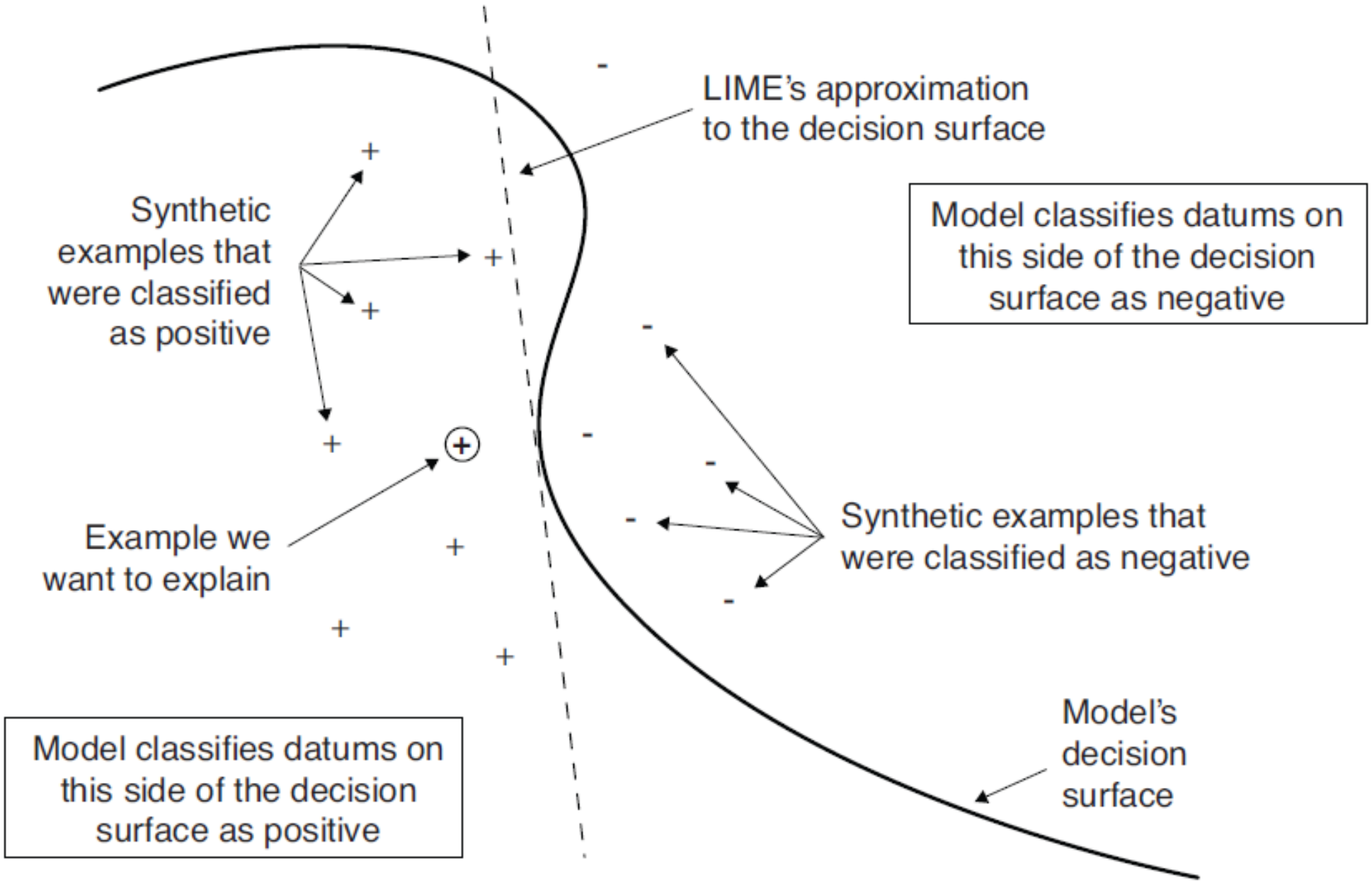


Figure 6.23 Notional sketch of how LIME works

Listing 6.19 Explaining the iris example

```
explanation <- lime::explain(example,
                             explainer,
                             n_labels = 1,
                             n_features = 4)
```

The number of labels to explain; use 1 for binary classification.

The number of features to use when fitting the explanation

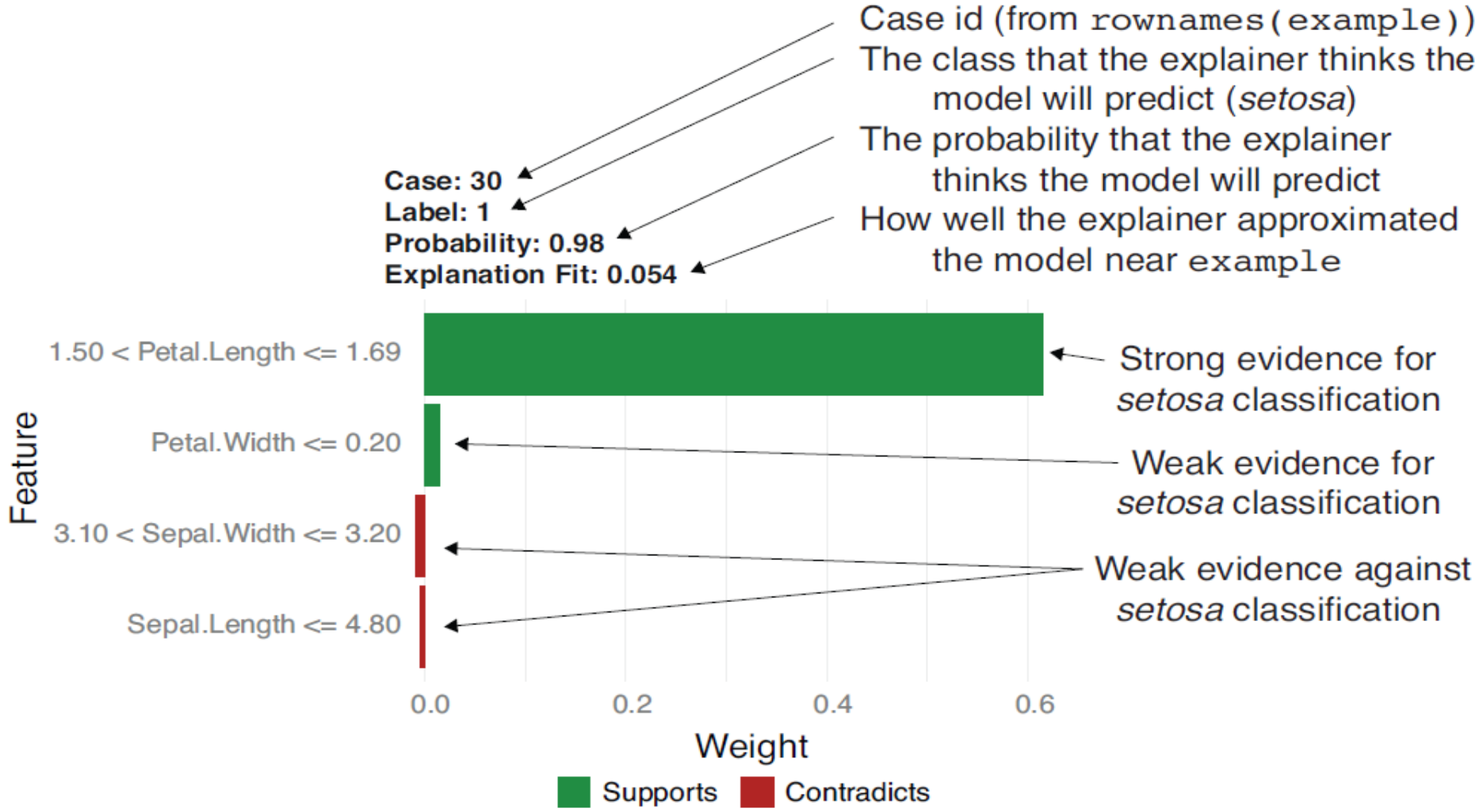


Figure 6.22 Visualize the explanation of the model's prediction.

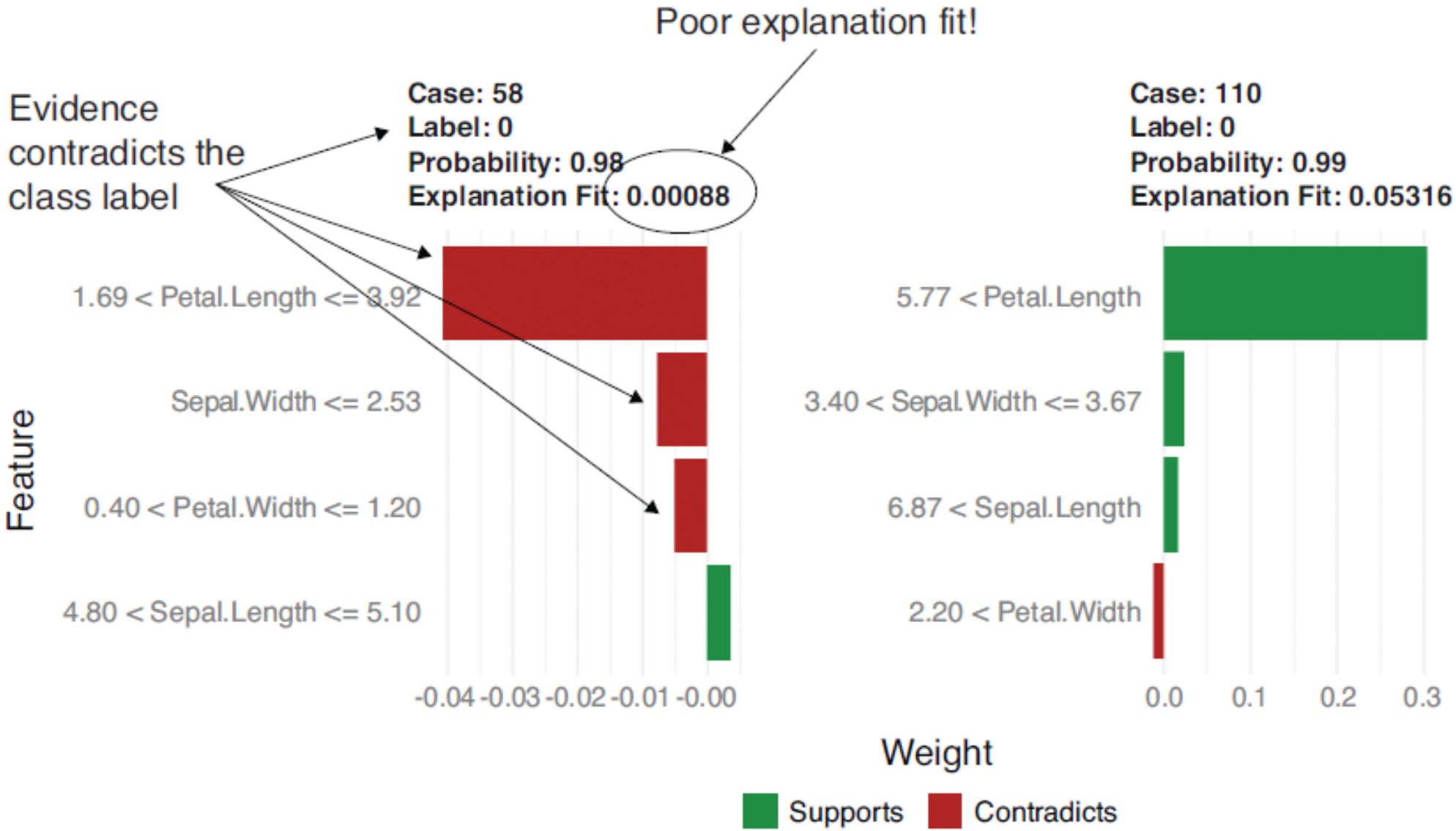


Figure 6.24 Explanations of the two iris examples

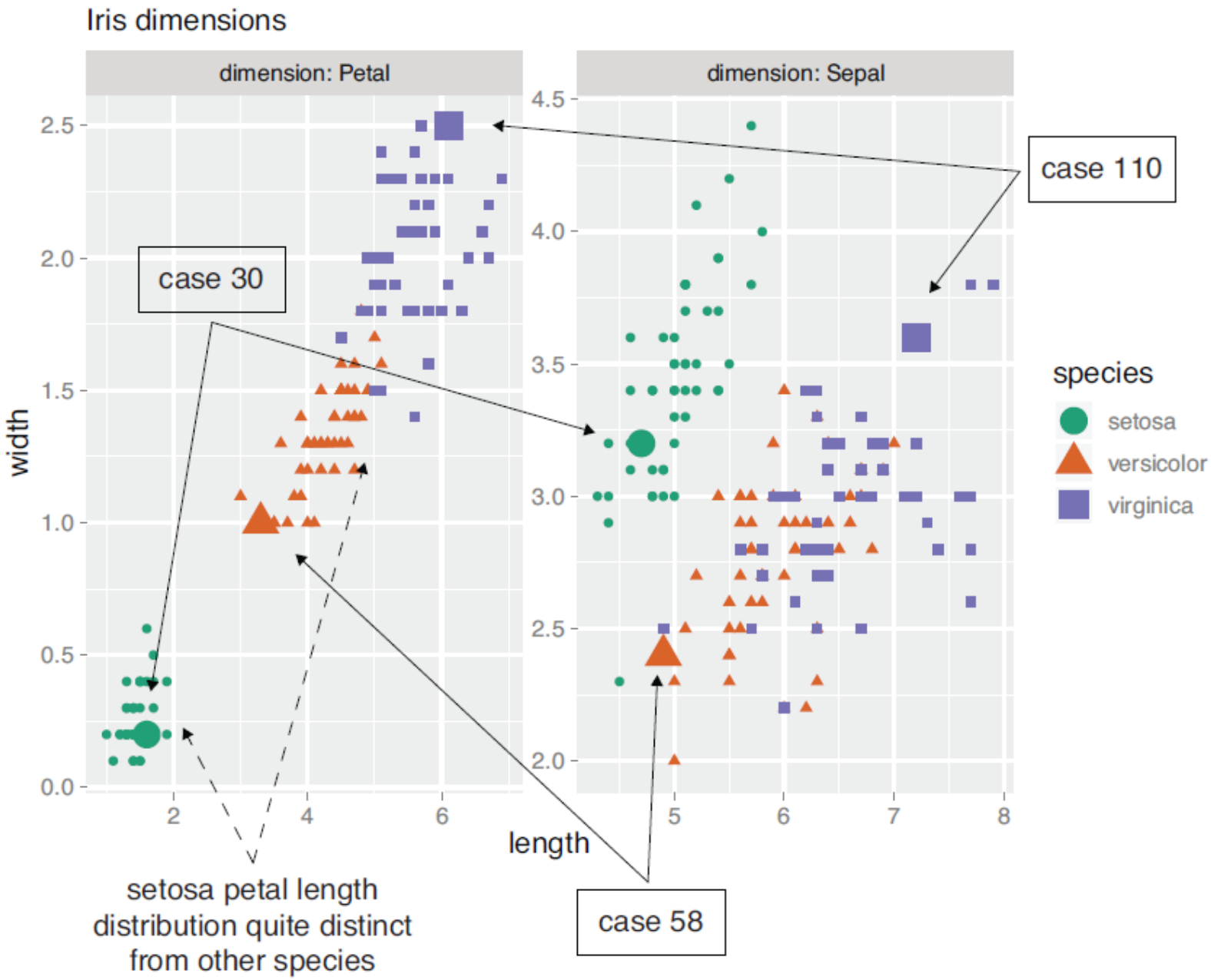


Figure 6.25 Distributions of petal and sepal dimensions by species

LIME for text classification

- *For this example, you will classify movie reviews from the Internet Movie Database (IMDB). The task is to identify positive reviews.*
- the original archive is split into two RDS files, `IMDBtrain.RDS` and `IMDBtest.RDS`
- Each RDS object is a list with two elements: a character vector representing 25,000 reviews, and a vector of numeric labels where 1 means a positive review and 0 a negative review
- For our text model, the features are the individual words, and there are a lot of them.
- To use `xgboost` to fit a model on texts, we have to build a finite feature set, or the *vocabulary*.
- The words in the vocabulary are the only features that the model will consider
- Once we have the vocabulary, we have to convert the texts (again using `text2vec`) into a numeric representation that `xgboost` can use.
- This representation is called a *document-term matrix*, where the rows represent each document in the corpus, and each column represents a word in the vocabulary. For a document-term matrix `dtm`, the entry `dtm[i, j]` is the number of times that the vocabulary word `w[j]` appeared in document `texts[i]`.
- The document-term matrix will be quite large: 25,000 rows by 10,000 columns.
- Luckily, most words in the vocabulary won't show up in a given document, so each row will be mostly zeros

Listing 6.21 Loading the IMDB training data

```
library(zeallot)

c(texts, labels) %<-% readRDS("IMDBtrain.RDS")
```

Loads the zeallot library. Calls install.packages("zeallot") if this fails.

The command read(IMDBtrain.RDS) returns a list object. The zeallot assignment arrow %<-% unpacks the list into two elements: texts is a character vector of reviews, and labels is a 0/1 vector of class labels. The label 1 designates a positive review.

You can examine the reviews and their corresponding labels. Here's a positive review:

```
list(text = texts[1], label = labels[1])
## $text
## train_21317
## train_21317
## "Forget depth of meaning, leave your logic at the door, and have a
## great time with this maniacally funny, totally absurdist, ultra-
## campy live-action \"cartoon\". MYSTERY MEN is a send-up of every
## superhero flick you've ever seen, but its unlikelysuper-wannabes
## are so interesting, varied, and well-cast that they are memorable
## characters in their own right. Dark humor, downright silliness,
## bona fide action, and even a touchingmoment or two, combine to
## make this comic fantasy about lovable losers a true winner. The
## comedic talents of the actors playing the Mystery Men --
## including one Mystery Woman -- are a perfect foil for Wes Studi
## as what can only be described as a bargain-basement Yoda, and
## Geoffrey Rush as one of the most off-the-wall (and bizarrely
## charming) villains ever to walk off the pages of a Dark Horse
## comic book and onto the big screen. Get ready to laugh, cheer,
## and say \"huh?\" more than once.... enjoy!"
##
## $label
## train_21317
##           1
```

Here's a negative review:

```
list(text = texts[12], label = labels[12])
## $text
## train_385
## train_385
## "Jameson Parker And Marilyn Hassett are the screen's most unbelievable
## couple since John Travolta and Lily Tomlin. Larry Peerce's direction
## wavers uncontrollably between black farce and Roman tragedy. Robert
## Klein certainly think it's the former and his self-centered  performance
## in a minor role underscores the total lack of balance and chemistry
## between the players in the film. Normally, I don't like to let myself
## get so ascerbic, but The Bell Jar is one of my all-time favorite books,
## and to watch what they did with it makes me literally crazy."
##
## $label
## train_385
##         0
```

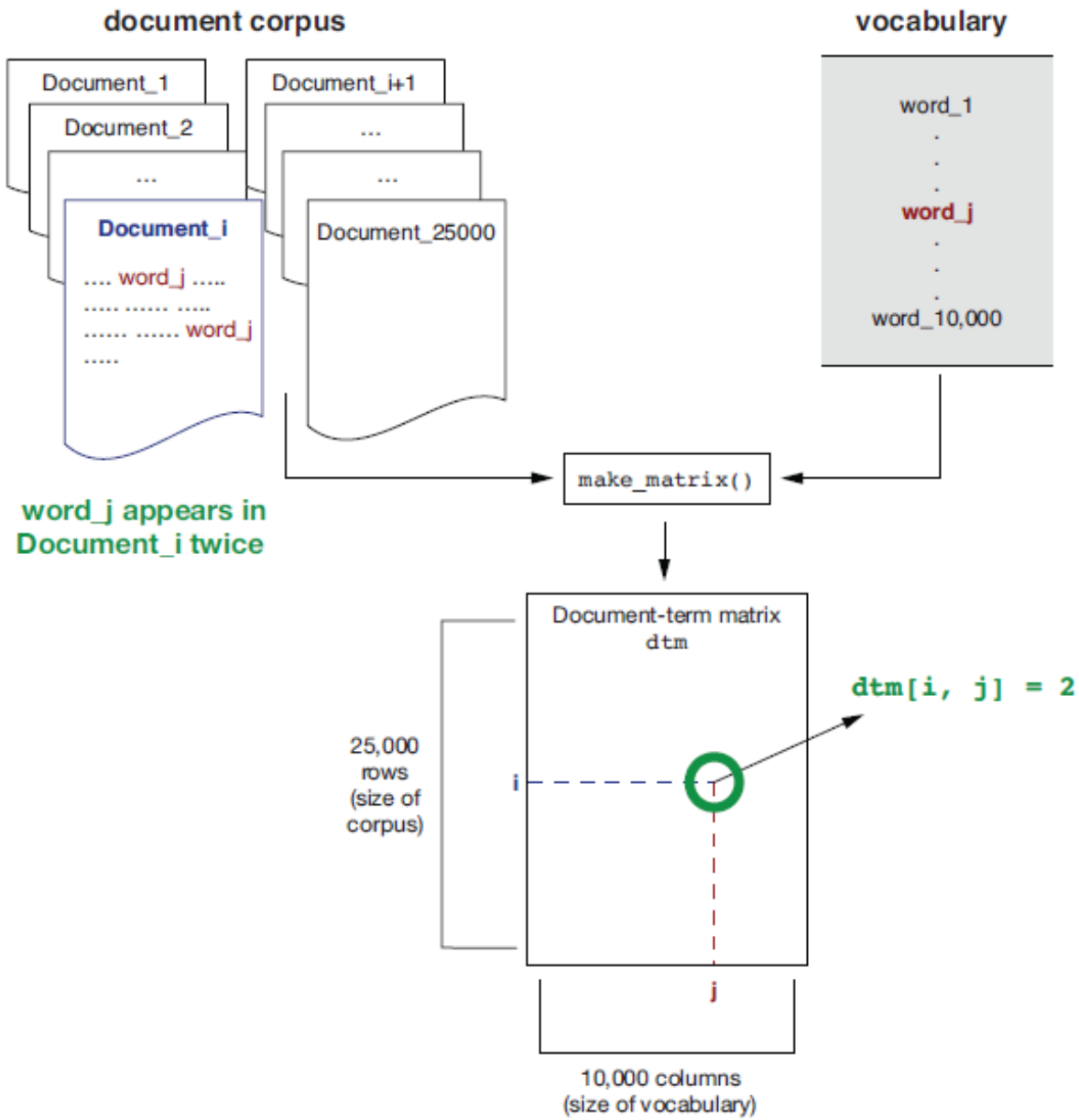


Figure 6.26 Creating a document-term matrix

Listing 6.22 Converting the texts and fitting the model

```
source("lime_imdb_example.R")

vocab <- create_pruned_vocabulary(texts)
dtm_train <- make_matrix(texts, vocab)
model <- fit_imdb_model(dtm_train, labels)
```

Creates the vocabulary from the training data

Creates the document-term matrix of the training corpus

Trains the model

Now load the test corpus and evaluate the model.

Listing 6.23 Evaluate the review classifier

```
c(test_txt, test_labels) %<-%  readRDS("IMDBtest.RDS")
dtm_test <- make_matrix(test_txt, vocab)

predicted <- predict(model, newdata=dtm_test)

teframe <- data.frame(true_label = test_labels,
                      pred = predicted)

(cmat <- with(teframe, table(truth=true_label, pred=pred > 0.5)))
##       pred
## truth FALSE  TRUE
##     0 10836  1664
##     1  1485 11015

sum(diag(cmat))/sum(cmat)
## [1] 0.87404

library(WVPlots)
DoubleDensityPlot(teframe, "pred", "true_label",
                  "Distribution of test prediction scores")
```

Reads in the test corpus

Converts the corpus to a document-term matrix

Makes predictions (probabilities) on the test corpus

Creates a frame with true and predicted labels

Computes the confusion matrix

Computes the accuracy

Plots the distribution of predictions

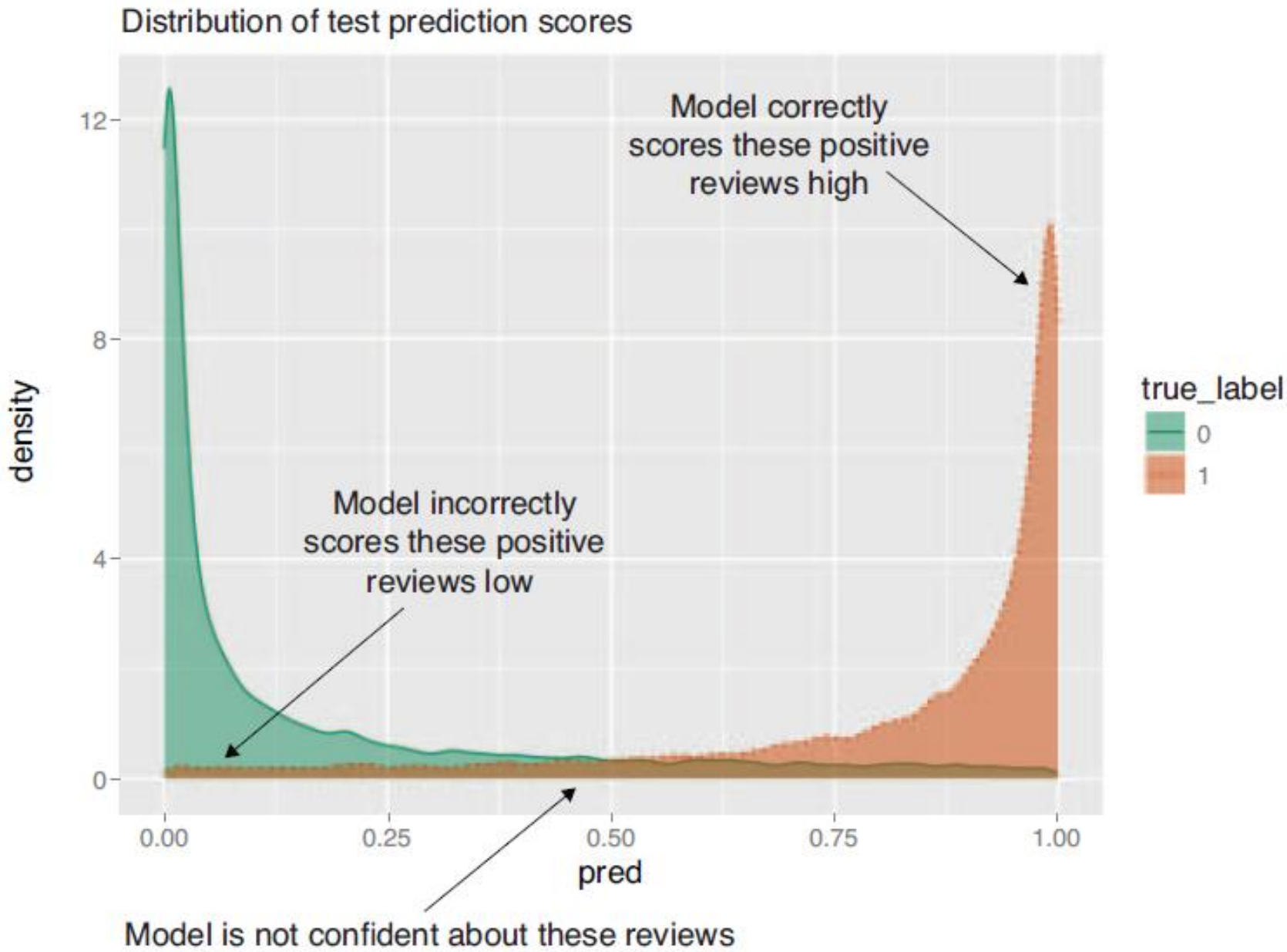


Figure 6.27 Distribution of test prediction scores

Listing 6.24 Building an explainer for a text classifier

```
explainer <- lime(texts, model = model,
                  preprocess = function(x) make_matrix(x, vocab))
```

Now take a short sample text from the test corpus. This review is positive, and the model predicts that it is positive.

Listing 6.25 Explaining the model's prediction on a review

```
casename <- "test_19552";
sample_case <- test_txt[casename]
pred_prob <- predict(model, make_matrix(sample_case, vocab))
list(text = sample_case,
     label = test_labels[casename],
     prediction = round(pred_prob) )
```

```
## $text
## test_19552
## "Great story, great music. A heartwarming love story that's beautiful to
## watch and delightful to listen to. Too bad there is no soundtrack CD."
##
## $label
## test_19552
##          1
##
## $prediction
## [1] 1
```

Listing 6.26 Explaining the model's prediction

```
explanation <- lime::explain(sample_case,
                             explainer,
                             n_labels = 1,
                             n_features = 5)

plot_features(explanation)
```

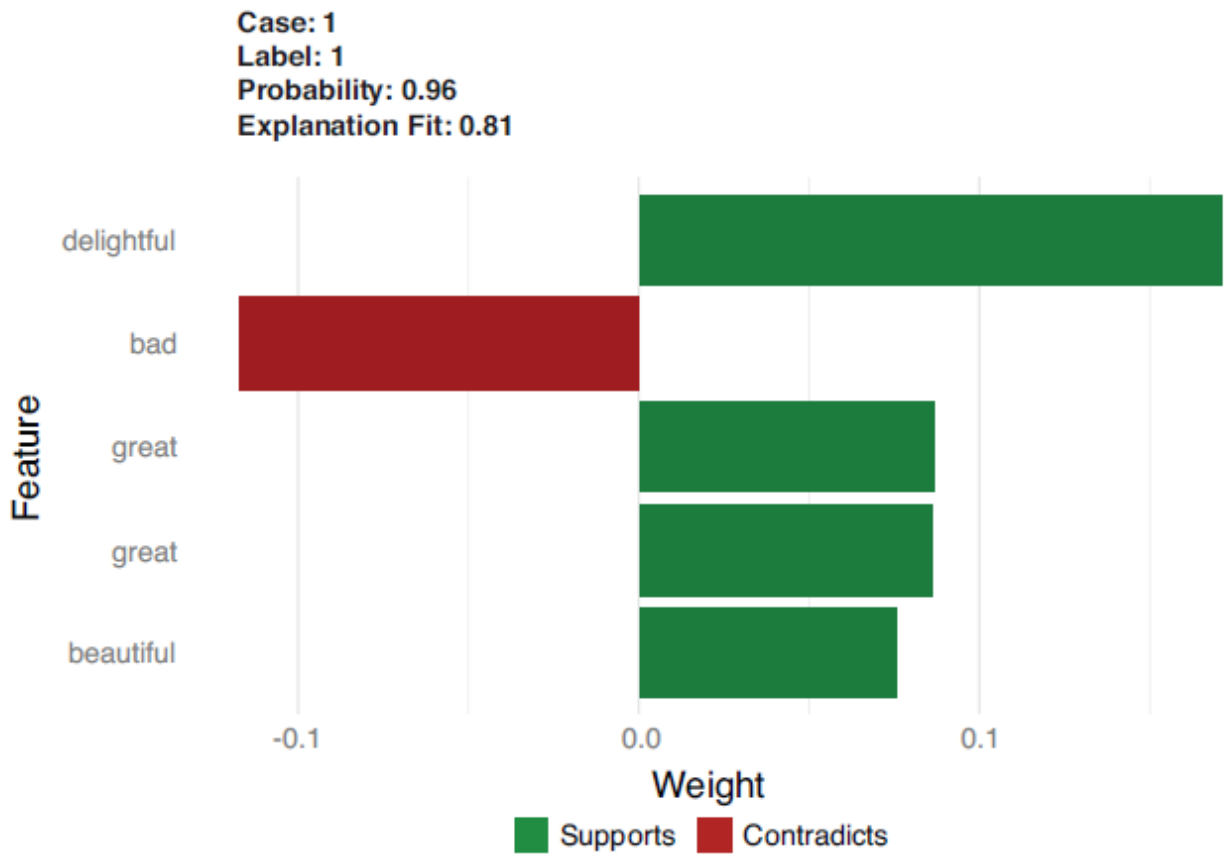


Figure 6.28 Explanation of the prediction on the sample review

Listing 6.27 Examining two more reviews

```
casenames <-  c("test_12034", "test_10294")
sample_cases <- test_txt[casenames]
pred_probs <- predict(model, newdata=make_matrix(sample_cases, vocab))
list(texts = sample_cases,
     labels = test_labels[casenames],
     predictions = round(pred_probs))

## $texts
## test_12034
## "I don't know why I even watched this film. I think it was because
## I liked the idea of the scenery and was hoping the film would be
## as good. Very boring and pointless."
##
## test_10294
## "To anyone who likes the TV series: forget the movie. The jokes
## are bad and some topics are much too sensitive to laugh about it.
## <br /><br />We have seen much better acting by R. Dueringer in
## \"Hinterholz 8\"".
##
## $labels
## test_12034 test_10294
##          0          0
##
## $predictions
## [1] 0 1

explanation <- lime::explain(sample_cases,
                             explainer,
                             n_labels = 1,
                             n_features = 5)

plot_features(explanation)
plot_text_explanations(explanation)
```

Both these reviews are negative.

The model misclassified the second review.

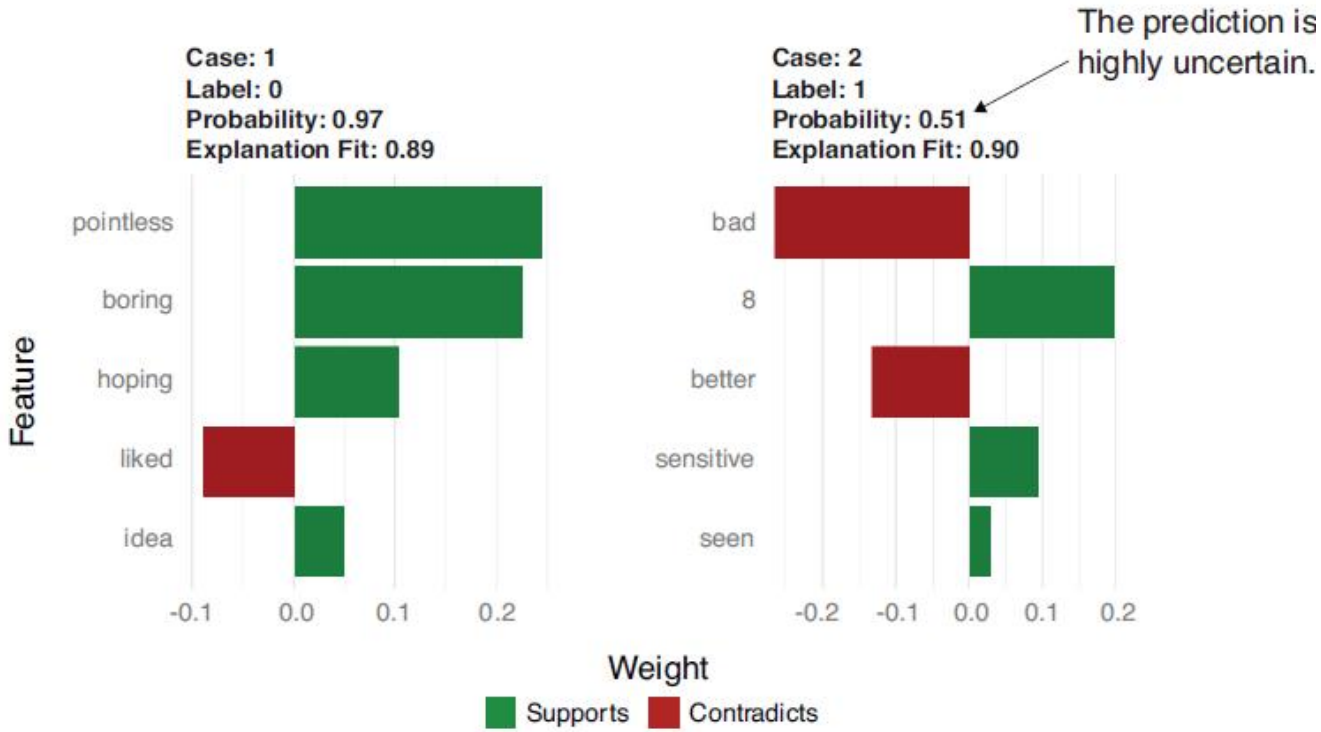


Figure 6.30 Explanation visualizations for the two sample reviews in listing 6.27

As a simple experiment, you can try removing the numbers 1 through 10 from the vocabulary,[14] and then refitting the model. The new model correctly classifies `test_10294` and returns a more reasonable explanation, as shown in figure 6.31.

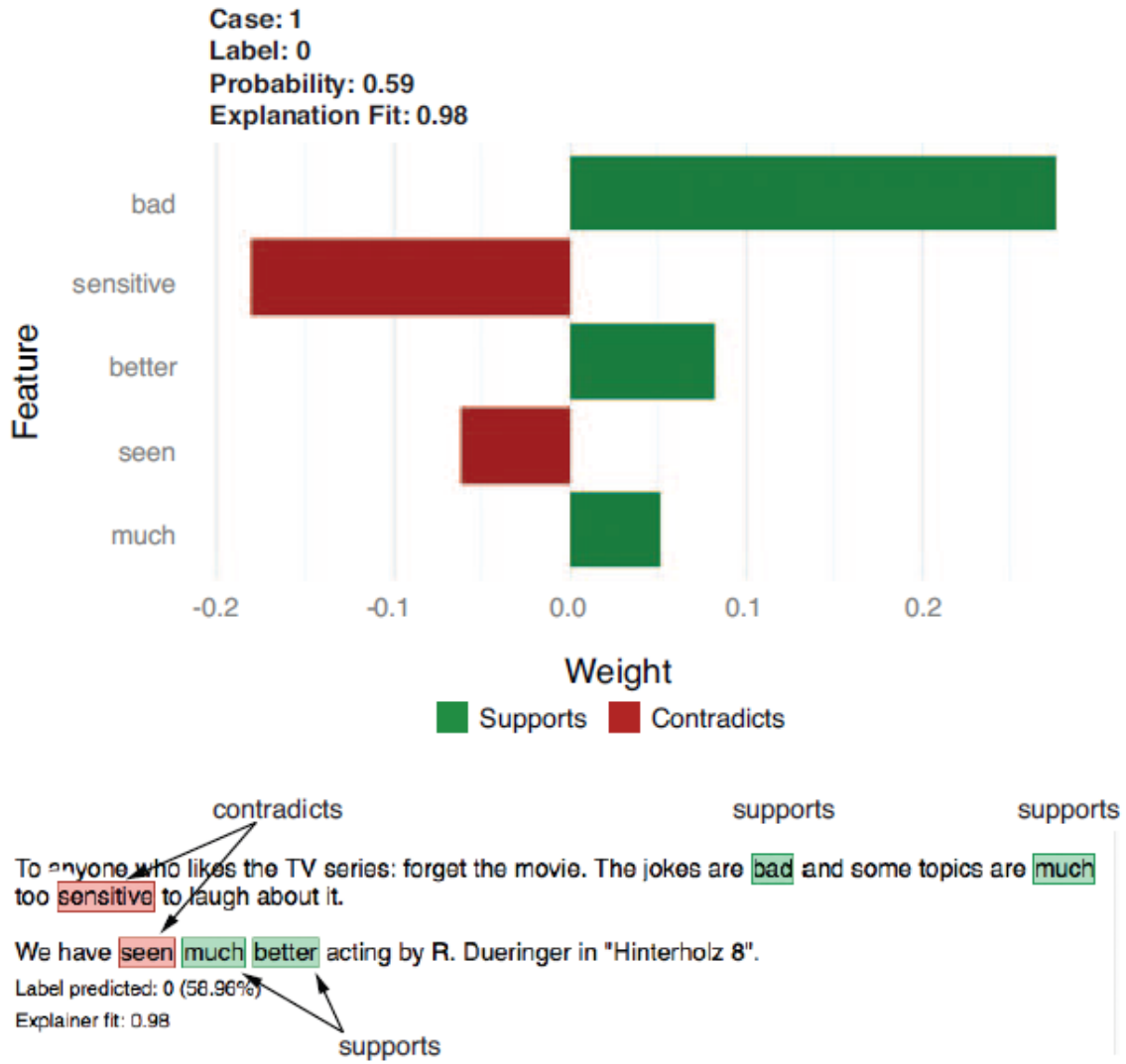


Figure 6.31 Explanation visualizations for `test_10294`