Unit-2 Managing Data

1.Cleaning data:

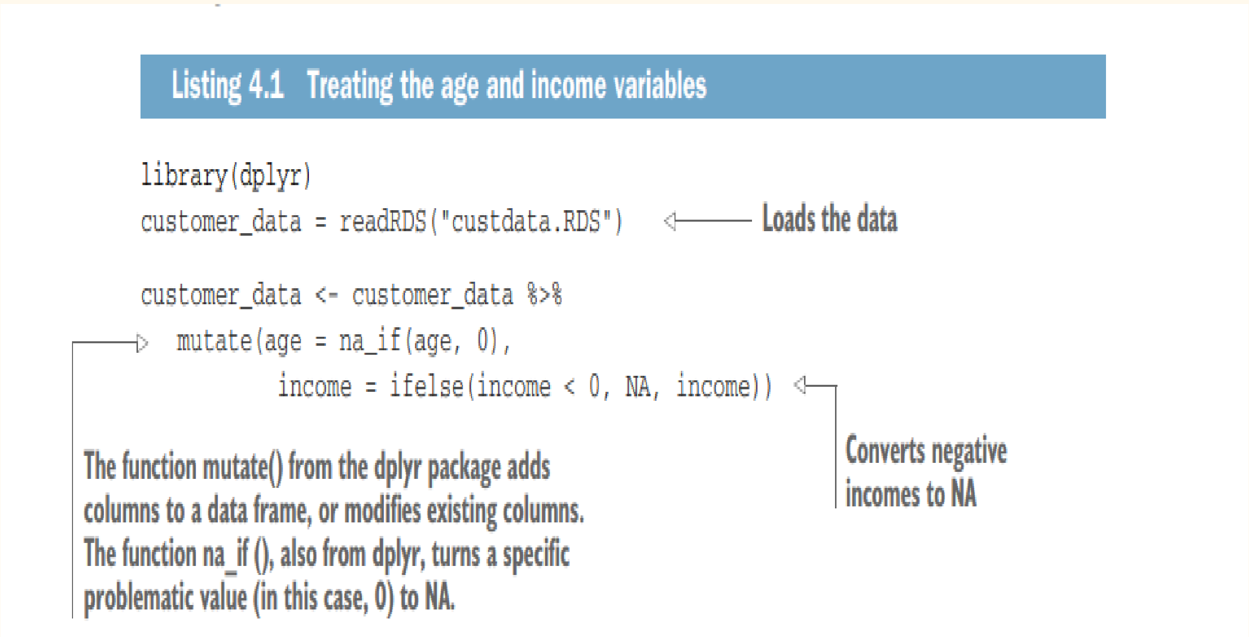
During the data exploration and visualization phase, it's common to encounter issues such as invalid and missing values in the dataset.

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.

Example Suppose you have a numeric variable called credit_score. Domain knowledge will tell you what the valid range for that variable should be. If the credit score is supposed to be a customer's "classic FICO score," then any value outside the range 300–850 should be treated as invalid. Other types of credit scores will have different ranges of valid values.

1.1 Domain-specific data cleaning:

A quick way to treat the age(The variable age has the problematic value 0, which probably means that the age is unknown) and income variables(The variable income has negative values) is to convert the invalid values to NA, as if they were missing variables



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))
```

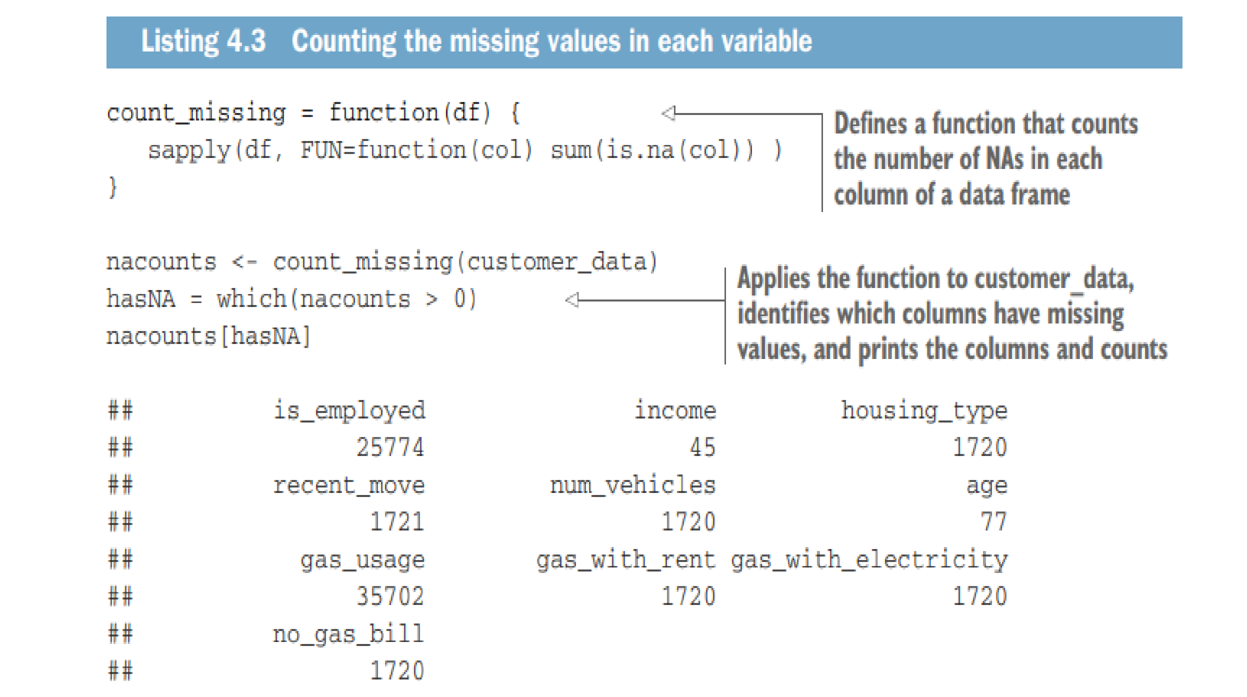
Loads the data

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.

1.2 Treating missing values:

One way to find these variables programmatically is to count how many missing values are in each column of the customer data frame, and look for the columns where that count is greater than zero.



Listing 4.3 Counting the missing values in each variable

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

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

##          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
```

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

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

2 ways to handle missing values:

1. drop the rows with missing values,
2. 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.

How to convert all the missing values to meaningful values.?

MISSING DATA IN CATEGORICAL VARIABLES

When the variable with missing values is categorical, an easy solution is to create a new category for the variable, called, for instance, missing or

invalid. This is shown schematically for the variable housing_type in figure 4.3.

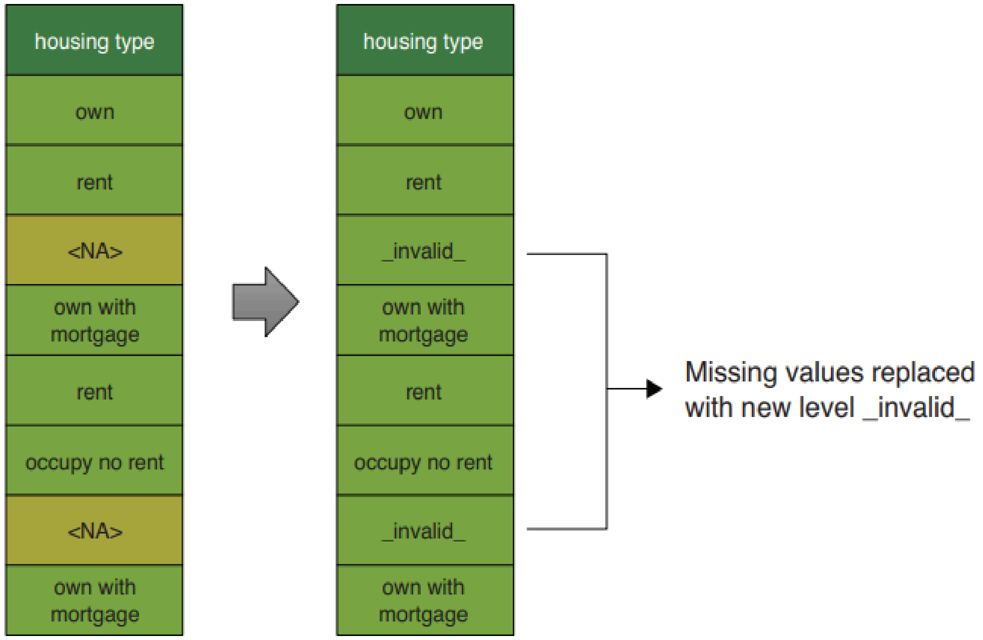
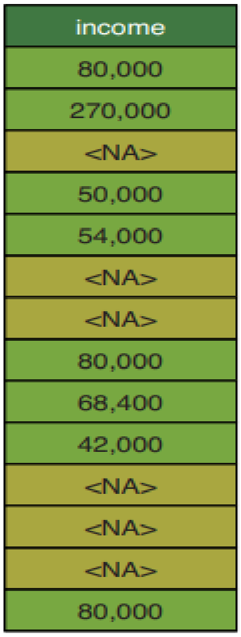


Figure 4.3 Creating a new level for missing categorical values

MISSING VALUES IN NUMERIC OR LOGICAL VARIABLES



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

Figure 4.4 Income data with missing values

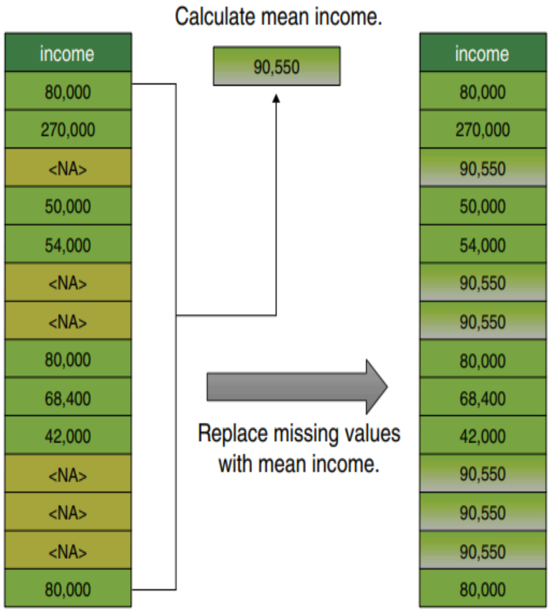
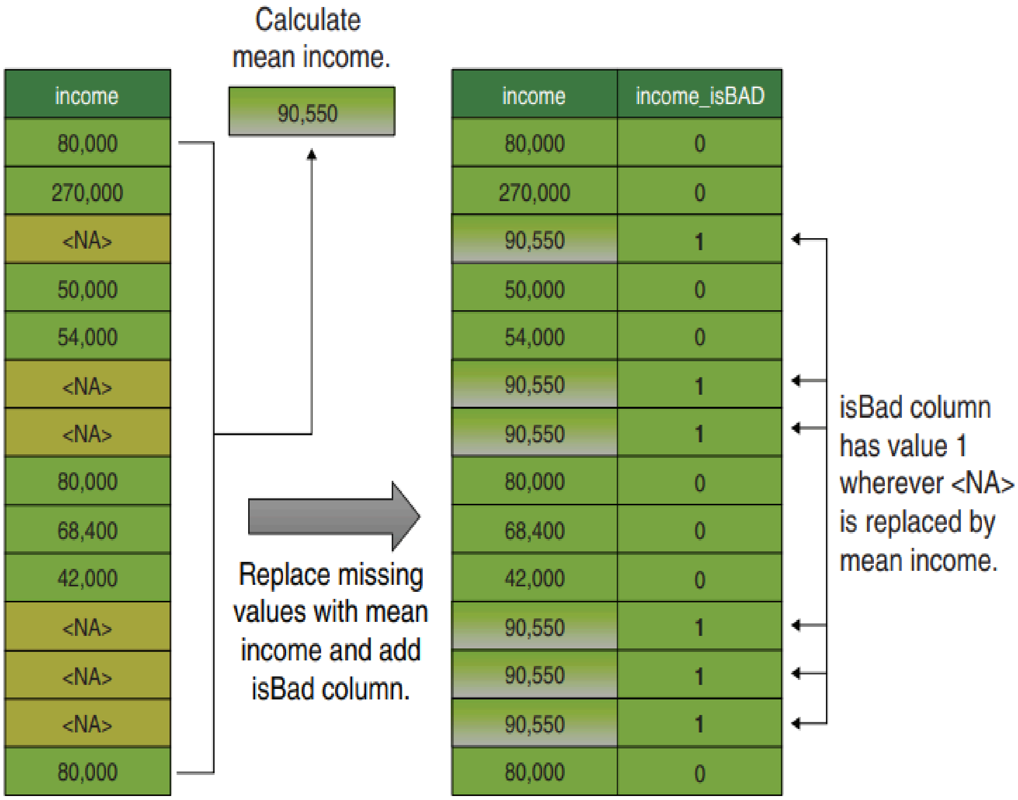


Figure 4.5 Replacing missing values with the mean

TREATING MISSING VALUES AS INFORMATION:

You still need to replace the NAs with a stand-in value, perhaps the mean. But the modeling algorithm should know that these values are possibly different from the others. A trick that has worked well for us is to replace the NAs with the mean, and add an additional indicator variable to keep track of which data points have been altered. This is shown in figure 4.6.


Calculate mean income.
90,550
Replace missing values with mean income and add isBad column.
isBad column has value 1 wherever <NA> is replaced by mean income.

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

The income_isBAD variable lets you differentiate the two kinds of values in the data: the ones that you are about to add, and the ones that were already there

1.3 The vtreat package for automatically treating missing variables:

First, you have to designate which columns of the data are the input variables: all of them except health_ins (which is the outcome to be predicted) and custid:

```
varlist <- setdiff(colnames(customer_data), c("custid", "health_ins"))
```

Then, you create the treatment plan, and “prepare” the data.

Listing 4.4 Creating and applying a treatment plan

```
library(vtreat)
treatment_plan <-
      design_missingness_treatment(customer_data, varlist = varlist)
training_prepared <- prepare(treatment_plan, customer_data)
```

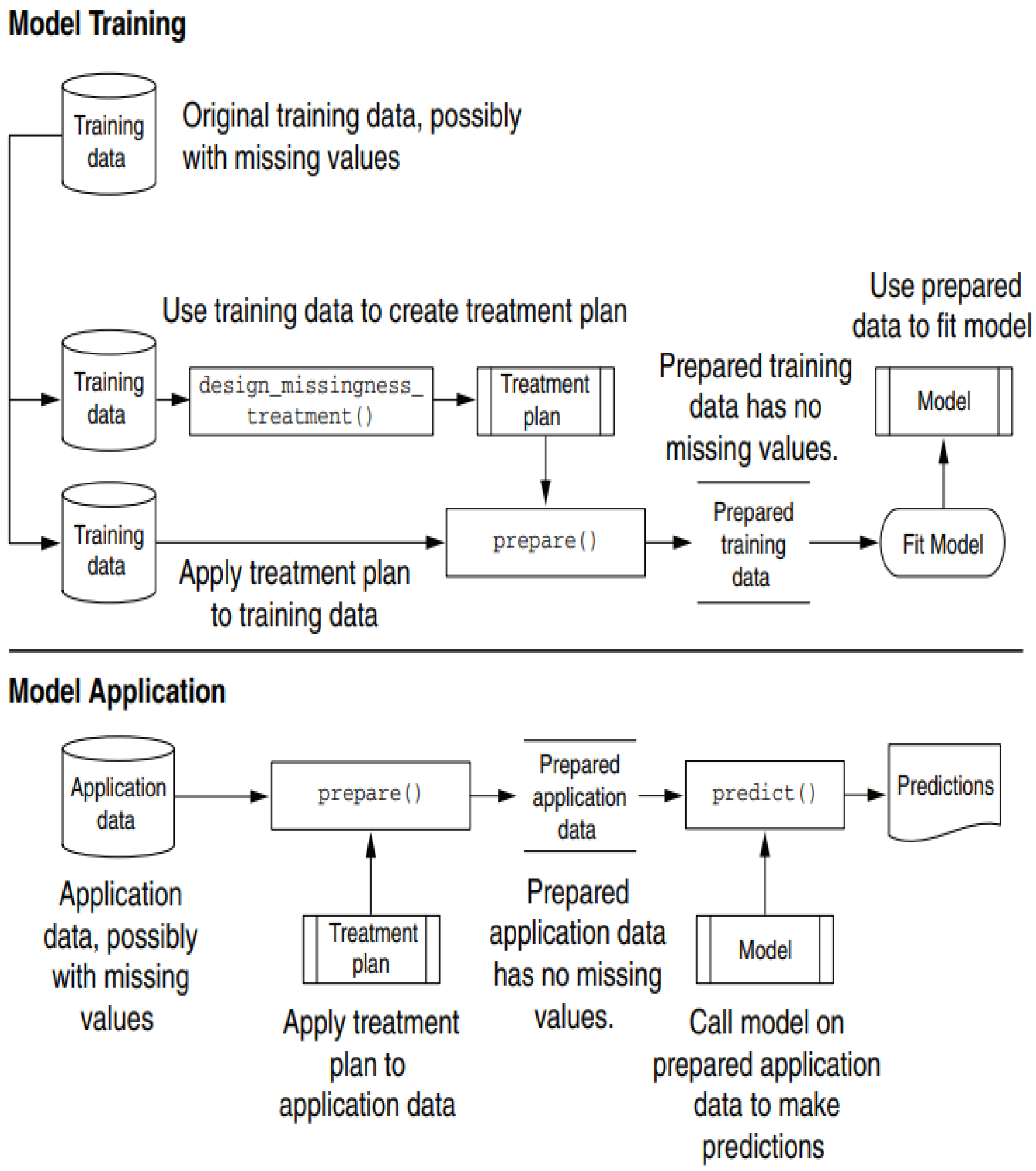
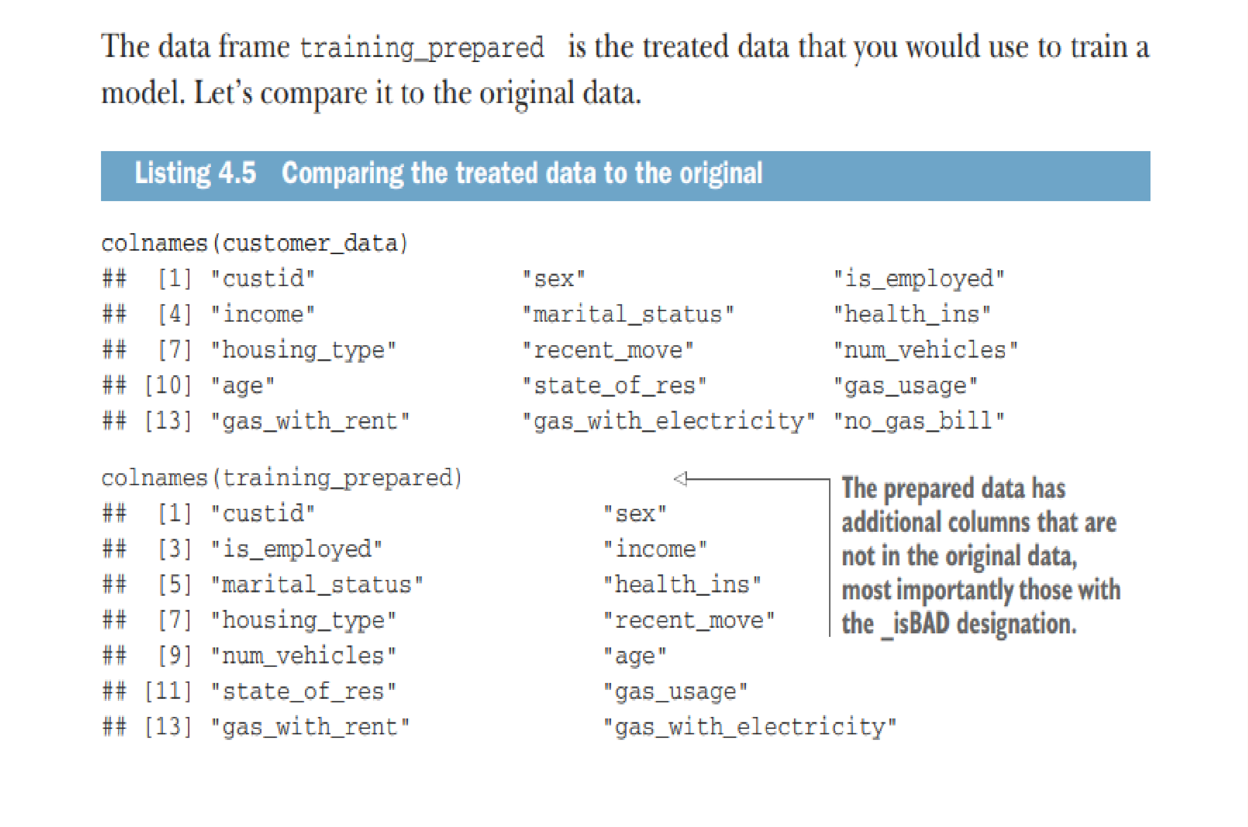

Model Training
Training data
Original training data, possibly with missing values
Use training data to create treatment plan
Training data
design_missingness_treatment()
Treatment plan
Prepared training data has no missing values.
Use prepared data to fit model
Model
Training data
prepare()
Prepared training data
Fit Model
Apply treatment plan to training data
Model Application
Application data
prepare()
Prepared application data
predict()
Predictions
Application data, possibly with missing values
Treatment plan
Apply treatment plan to application data
Prepared application data has no missing values.
Model
Call model on prepared application data to make predictions

Figure 4.7 Creating and applying a simple treatment plan

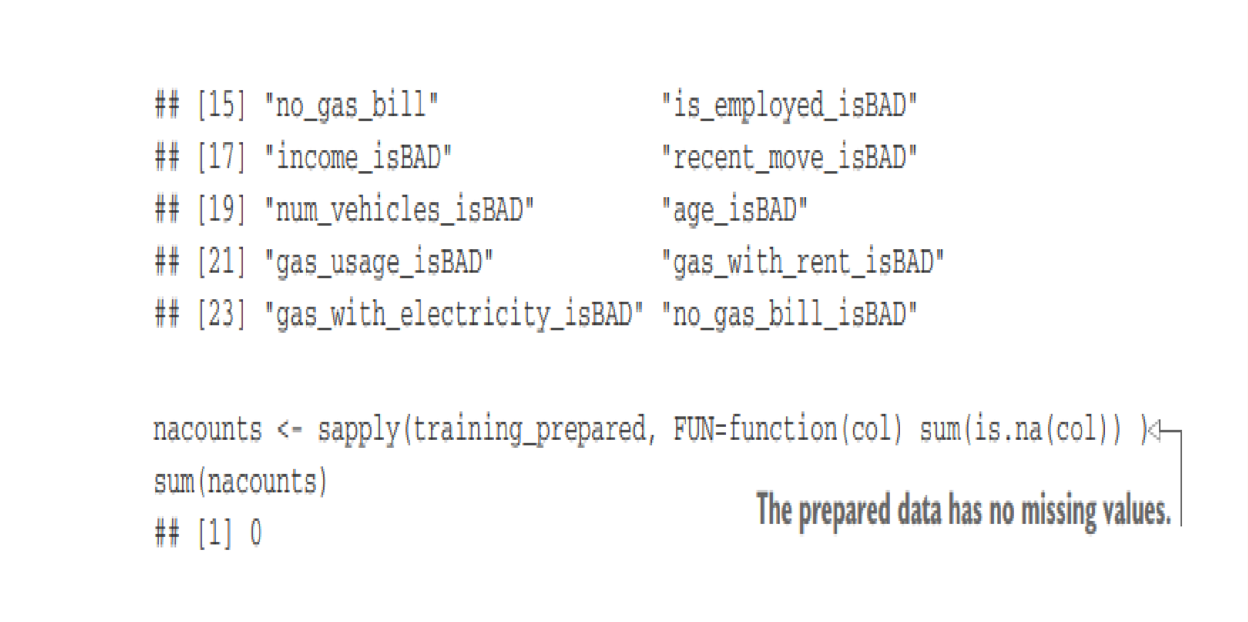


The data frame training_prepared is the treated data that you would use to train a model. Let's compare it to the original data.

Listing 4.5 Comparing the treated data to the original

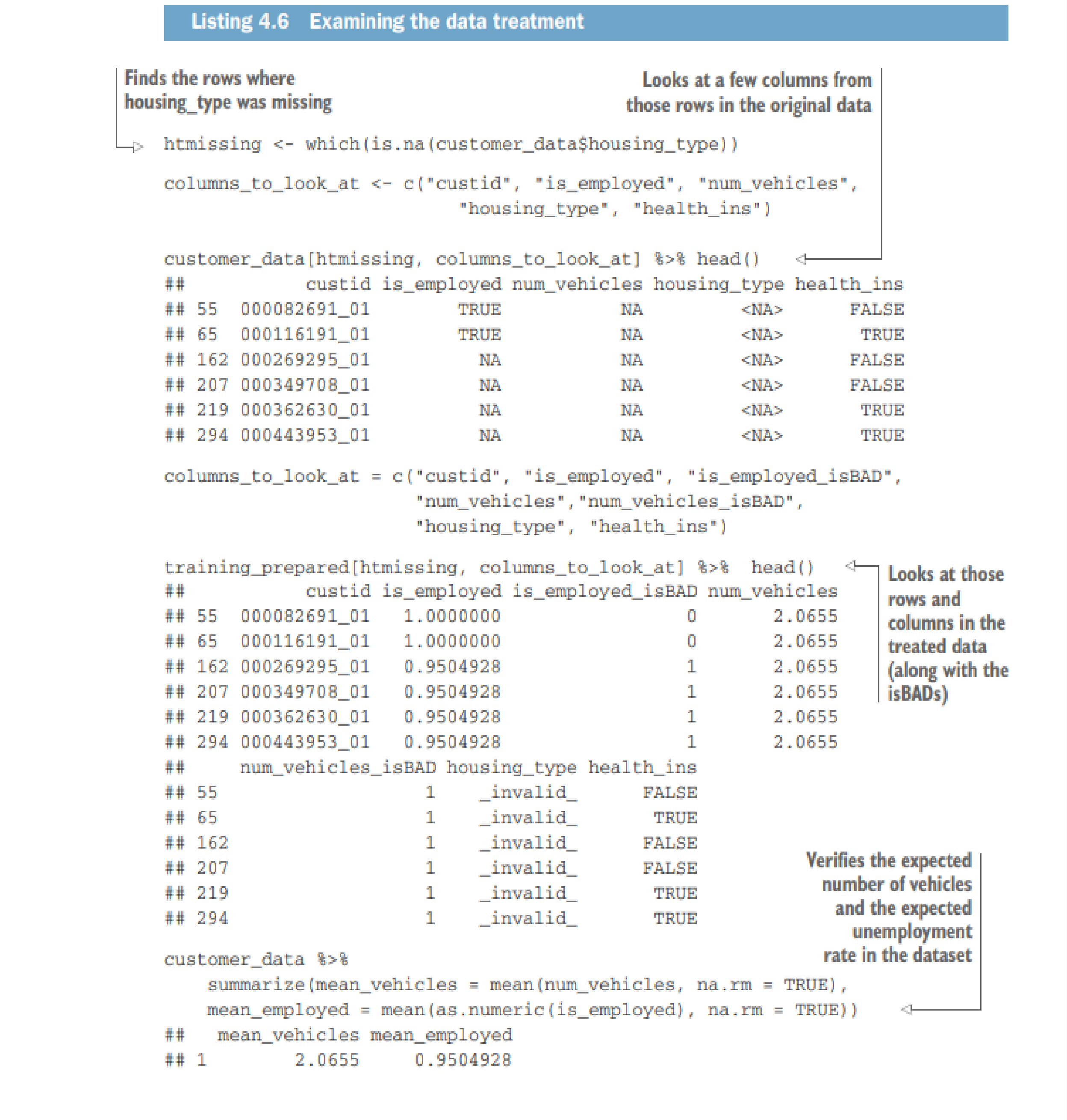
```
colnames(customer_data)
##  [1] "custid"               "sex"                  "is_employed"
##  [4] "income"               "marital_status"       "health_ins"
##  [7] "housing_type"         "recent_move"          "num_vehicles"
## [10] "age"                  "state_of_res"         "gas_usage"
## [13] "gas_with_rent"        "gas_with_electricity" "no_gas_bill"

colnames(training_prepared)
##  [1] "custid"                     "sex"
##  [3] "is_employed"                "income"
##  [5] "marital_status"             "health_ins"
##  [7] "housing_type"               "recent_move"
##  [9] "num_vehicles"               "age"
## [11] "state_of_res"               "gas_usage"
## [13] "gas_with_rent"              "gas_with_electricity"
```

The prepared data has additional columns that are not in the original data, most importantly those with the _isBAD designation.



```
## [15] "no_gas_bill"                "is_employed_isBAD"
## [17] "income_isBAD"               "recent_move_isBAD"
## [19] "num_vehicles_isBAD"         "age_isBAD"
## [21] "gas_usage_isBAD"            "gas_with_rent_isBAD"
## [23] "gas_with_electricity_isBAD" "no_gas_bill_isBAD"

nacounts <- sapply(training_prepared, FUN=function(col) sum(is.na(col)) )
sum(nacounts)
## [1] 0
```

The prepared data has no missing values.



Listing 4.6 Examining the data treatment

Finds the rows where housing_type was missing

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

```
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

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

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
```

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

2. Data Transformations:

1. **Normalization**
2. **Centering and scaling**
3. **Log transformations**

2.1 Normalization:

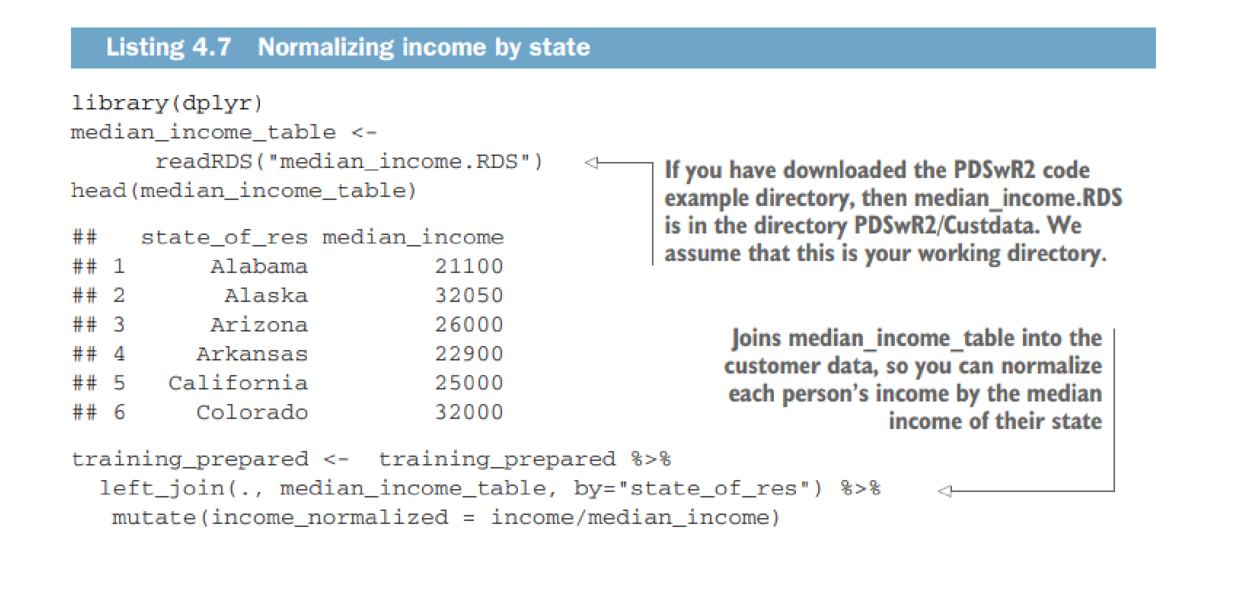
Normalization scales the data to a fixed range, typically between 0 and 1, or sometimes between -1 and 1. It's useful when you want to bound your data within a specific range.

Formula for Normalization:

$$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

When to Use Normalization:

- **Algorithms Sensitive to the Scale of Data:**
  - **K-Nearest Neighbors (K-NN):** Distance-based algorithms can be biased toward features with larger scales.
  - **Neural Networks:** Activation functions like sigmoid can be sensitive to the scale of input features.
- **Features with Different Scales:**
  - When features have different units or scales, normalization helps to bring them to a common scale without distorting differences in the ranges of values.

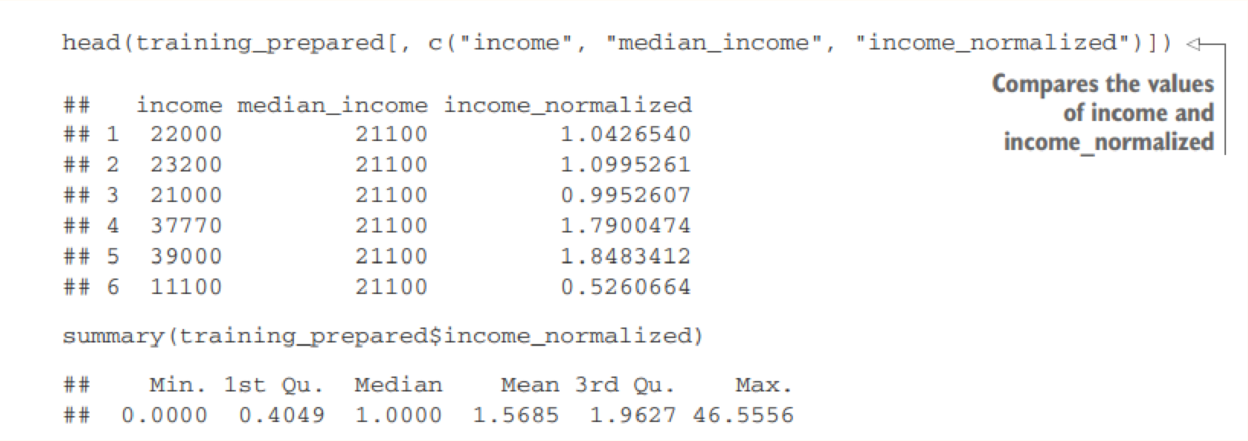


Listing 4.7 Normalizing income by state

```
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)
```

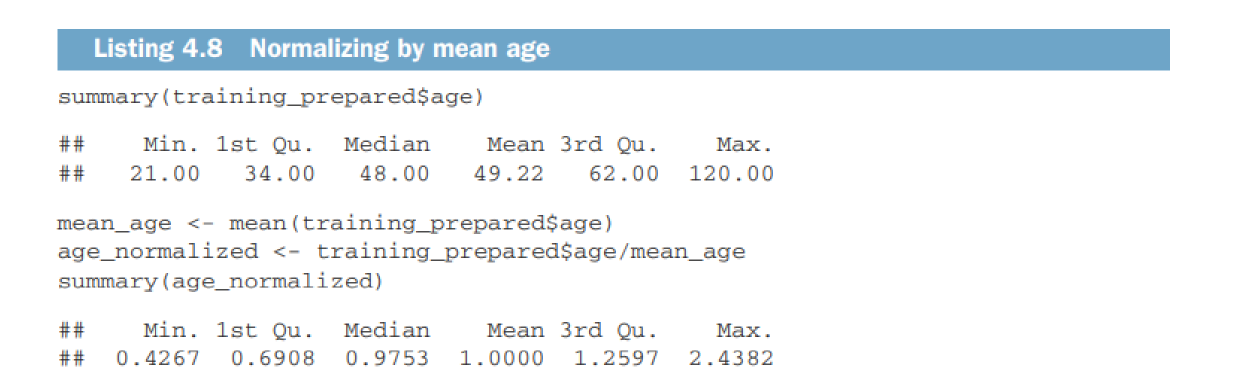
If you have downloaded the PDSwR2 code example directory, then median_income.RDS is in the directory PDSwR2/Custdata. We assume that this is your working directory.

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
```

2.2 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

Scenario:

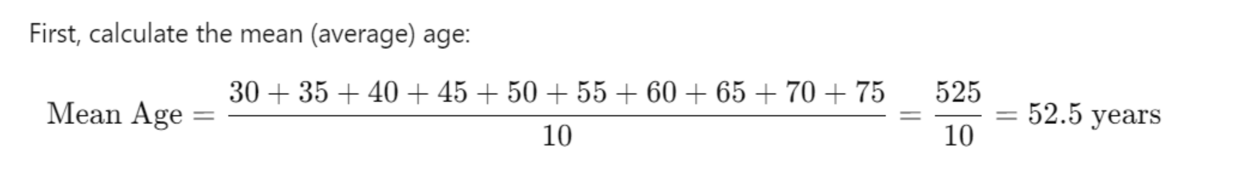
Suppose you have a dataset of customer ages, and you want to determine how old or young each customer is relative to the average (mean) age of all customers in the dataset.

Step 1: Calculate the Mean and Standard Deviation

Assume you have the following ages of 10 customers:

- **Ages: 30, 35, 40, 45, 50, 55, 60, 65, 70, 75**

First, calculate the mean (average) age:



First, calculate the mean (average) age:

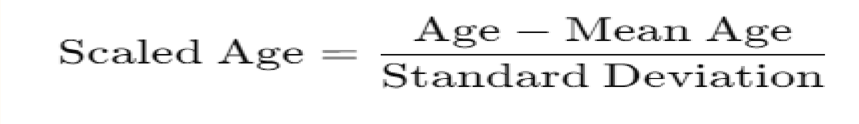
$$\text{Mean Age} = \frac{30+35+40+45+50+55+60+65+70+75}{10} = \frac{525}{10} = 52.5 \text{ years}$$

Next, calculate the standard deviation (which measures the spread of the ages around the mean). For simplicity, let's assume the standard deviation is 15 years.

Step 2: Center and Scale the Ages

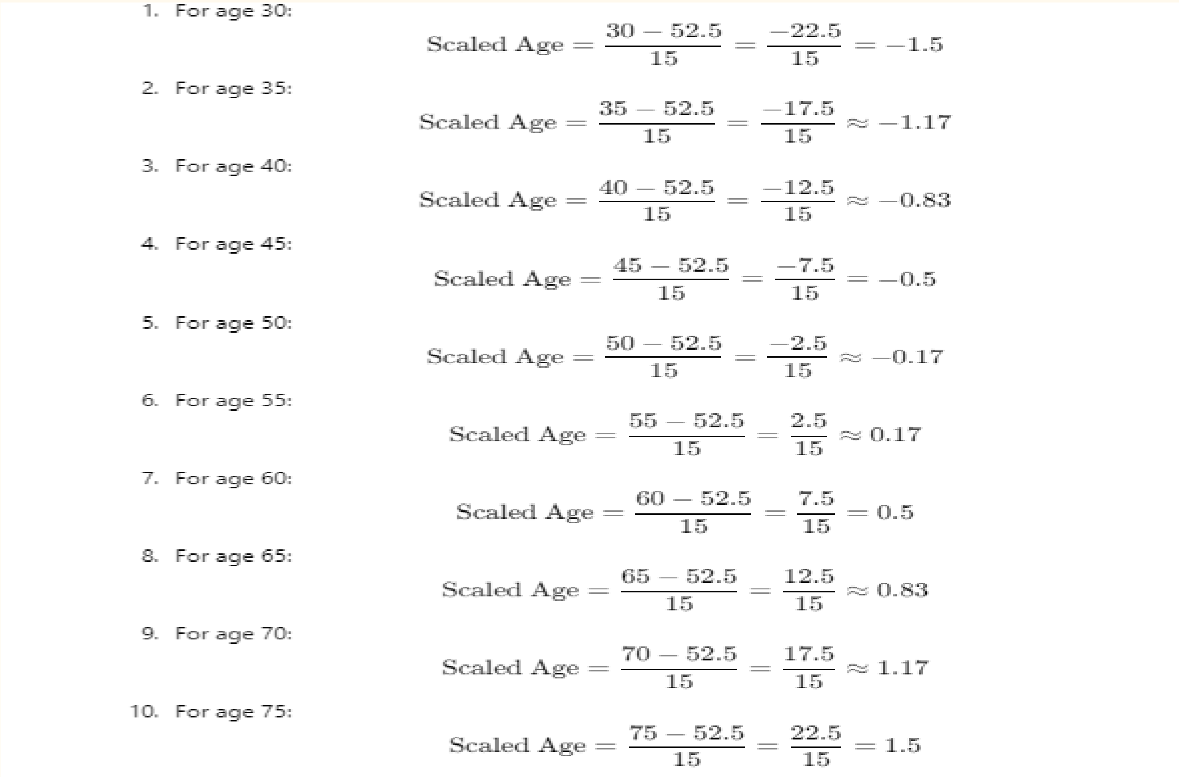
To rescale the ages, subtract the mean from each age and then divide by the standard deviation.

The formula is:



$$\text{Scaled Age} = \frac{\text{Age} - \text{Mean Age}}{\text{Standard Deviation}}$$

Let's calculate this for each customer:



1. For age 30:

$$\text{Scaled Age} = \frac{30 - 52.5}{15} = \frac{-22.5}{15} = -1.5$$

2. For age 35:

$$\text{Scaled Age} = \frac{35 - 52.5}{15} = \frac{-17.5}{15} \approx -1.17$$

3. For age 40:

$$\text{Scaled Age} = \frac{40 - 52.5}{15} = \frac{-12.5}{15} \approx -0.83$$

4. For age 45:

$$\text{Scaled Age} = \frac{45 - 52.5}{15} = \frac{-7.5}{15} = -0.5$$

5. For age 50:

$$\text{Scaled Age} = \frac{50 - 52.5}{15} = \frac{-2.5}{15} \approx -0.17$$

6. For age 55:

$$\text{Scaled Age} = \frac{55 - 52.5}{15} = \frac{2.5}{15} \approx 0.17$$

7. For age 60:

$$\text{Scaled Age} = \frac{60 - 52.5}{15} = \frac{7.5}{15} = 0.5$$

8. For age 65:

$$\text{Scaled Age} = \frac{65 - 52.5}{15} = \frac{12.5}{15} \approx 0.83$$

9. For age 70:

$$\text{Scaled Age} = \frac{70 - 52.5}{15} = \frac{17.5}{15} \approx 1.17$$

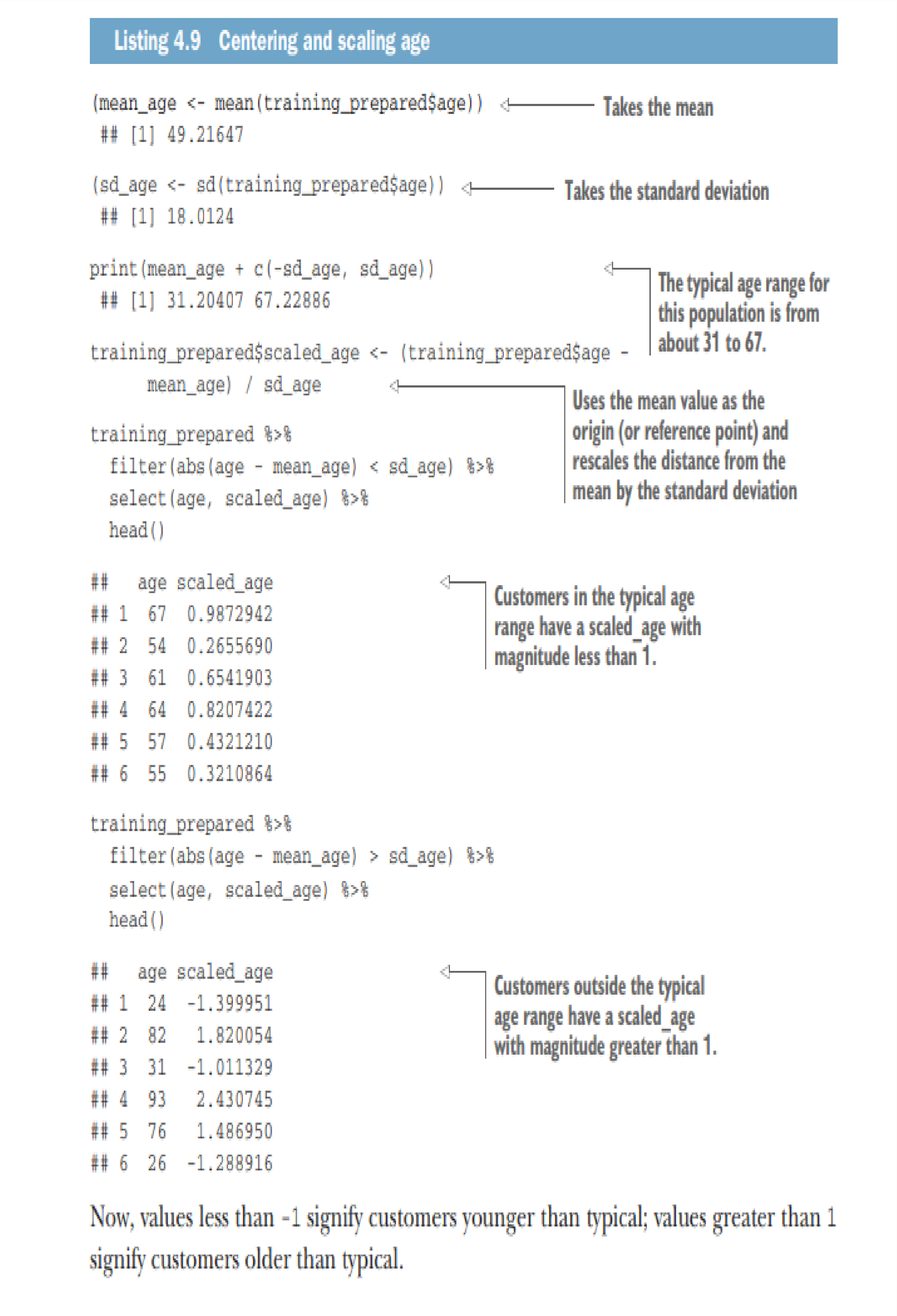
10. For age 75:

$$\text{Scaled Age} = \frac{75 - 52.5}{15} = \frac{22.5}{15} = 1.5$$

Step 3: Interpret the Scaled Ages

- **Centered Age: After centering by the mean, a customer with the mean age (52.5 years) would have a scaled age of 0, meaning they are "typical" in terms of age.**
- **Within One Standard Deviation:**
  -
  - **Customers with ages close to the mean, like 45 or 55 years, have scaled ages between -0.5 and 0.5, indicating they are not much older or younger than the average.**
- **More Than One Standard Deviation:**
  - **A customer who is 30 years old has a scaled age of -1.5, meaning they are 1.5 standard deviations younger than the mean, which is relatively young.**
  - **A customer who is 75 years old has a scaled age of 1.5, meaning they are 1.5 standard deviations older than the mean, which is relatively old.**

Conclusion: ***By rescaling using the standard deviation, you can easily see how much older or younger a customer is compared to the average. This method standardizes the age data, making it easier to compare and analyze.***



Listing 4.9 Centering and scaling age

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

Takes the mean

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

Takes the standard deviation

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

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

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

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

```
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
```

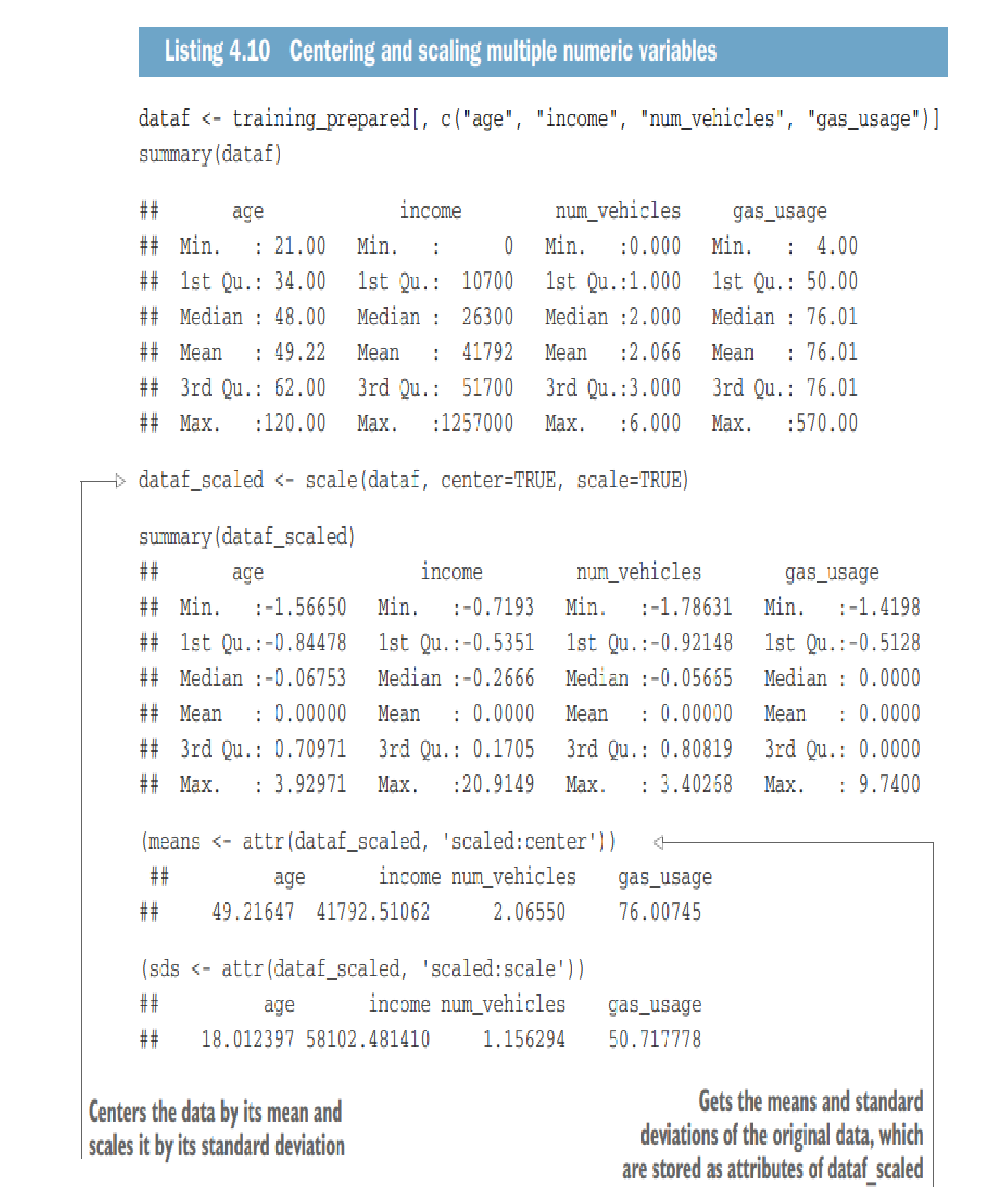
Customers in the typical age range have a scaled_age with magnitude less than 1.

```
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
```

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.

When you have multiple numeric variables, you can use the scale() function to center and scale all of them simultaneously



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

scale():

- **The scale() function in R is used to standardize or normalize numeric data. It can both center the data (subtract the mean) and scale it (divide by the standard deviation). This is useful in many machine learning algorithms where feature scaling is important, such as when features have different units or ranges.**
- **The scale() function can handle both centering and scaling tasks:**
    - **Centering: Subtracts the mean of the data, so the data has a mean of 0.**

- Scaling: Divides by the standard deviation, so the data has a standard deviation of 1.

3. `dataf`:

- This is the input data (a data frame or matrix) to be scaled. The `scale()` function will operate on all the numeric columns of `dataf`.

4. `center = TRUE`:

- When `center = TRUE`, the mean of each feature (column) is subtracted from its values, so each feature will have a mean of 0.
- If `center = FALSE`, the data won't be centered.

5. `scale = TRUE`:

- When `scale = TRUE`, each feature (column) is divided by its standard deviation, so that each feature has a standard deviation of 1.
- If `scale = FALSE`, the data won't be scaled (only centered if `center = TRUE`).

2.3 Log transformations for skewed and wide distributions

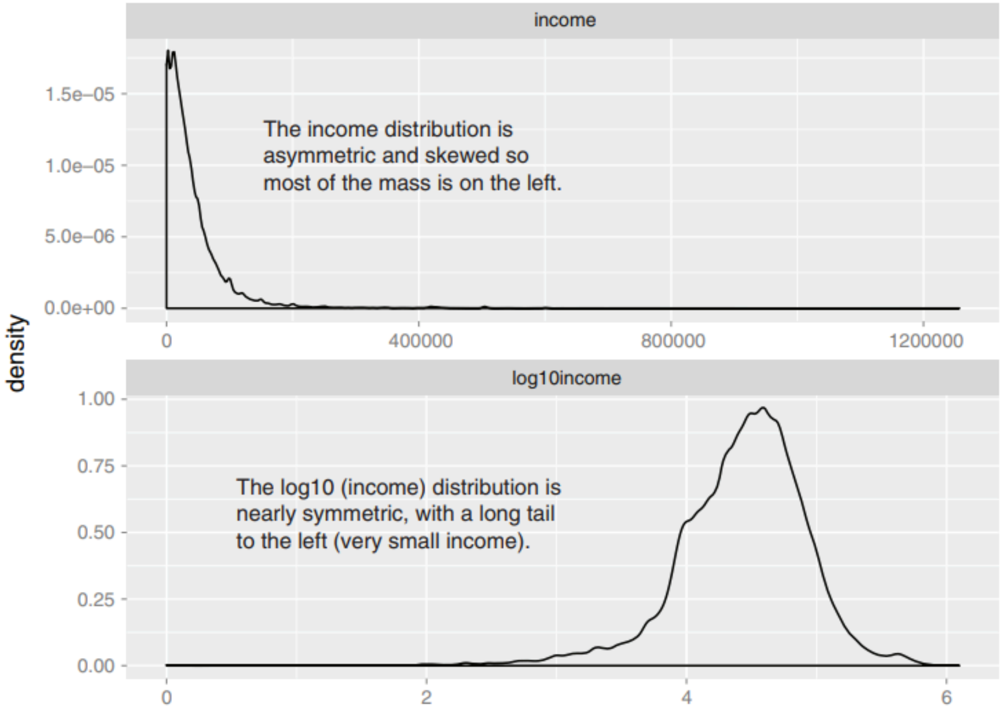


Figure 4.10 A nearly lognormal distribution and its log

Log transformations are a common technique used in data analysis to address issues related to skewed and wide distributions of data. These transformations are particularly useful when the data has a long tail or when there are large differences in the magnitude of the values.

1. Skewed Distributions:

- A skewed distribution is one where the data is not symmetrically distributed. In a right-skewed (positive skew) distribution, the tail on the right side is longer, meaning there are some very high values. In a left-skewed (negative skew) distribution, the tail on the left side is longer, meaning there are some very low values.

- Skewed data can complicate statistical analyses because many statistical techniques assume the data is normally distributed (symmetric around the mean). Skewness can also affect the interpretation of the mean and standard deviation.
- Log transformation can reduce skewness, making the data more symmetric. This is particularly effective for right-skewed data, as it compresses the long right tail and spreads out the lower end of the distribution, leading to a more normal distribution.

2. Wide Distributions:

- A wide distribution means that the data spans a large range of values, often several orders of magnitude. For example, in income data, some people might earn thousands, while others earn millions.
- Wide distributions can make it difficult to analyze or visualize data because the large range of values can dominate the analysis, overshadowing smaller, but still important, differences.
- Log transformation helps compress the wide range of values into a more manageable scale. By applying the logarithm, larger values are reduced more significantly than smaller values, leading to a more compact distribution.

3. Sampling for modeling and validation

Sampling is the process of selecting a subset of a population to represent the whole during analysis and modeling.

Why sampling is needed?

1. 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
2. Visualization can be easier with a subsample of the data; ggplot runs faster on smaller datasets, and too much data will often obscure the patterns in a graph
3. Sampling for Test and Training Splits: Training and Testing: Another common reason for sampling is to create training and test datasets. The training set is used to build the model, while the test set is used to evaluate its performance. Proper sampling ensures that both sets are representative of the overall population, which is crucial for the model's success

3.1 Test and training splits

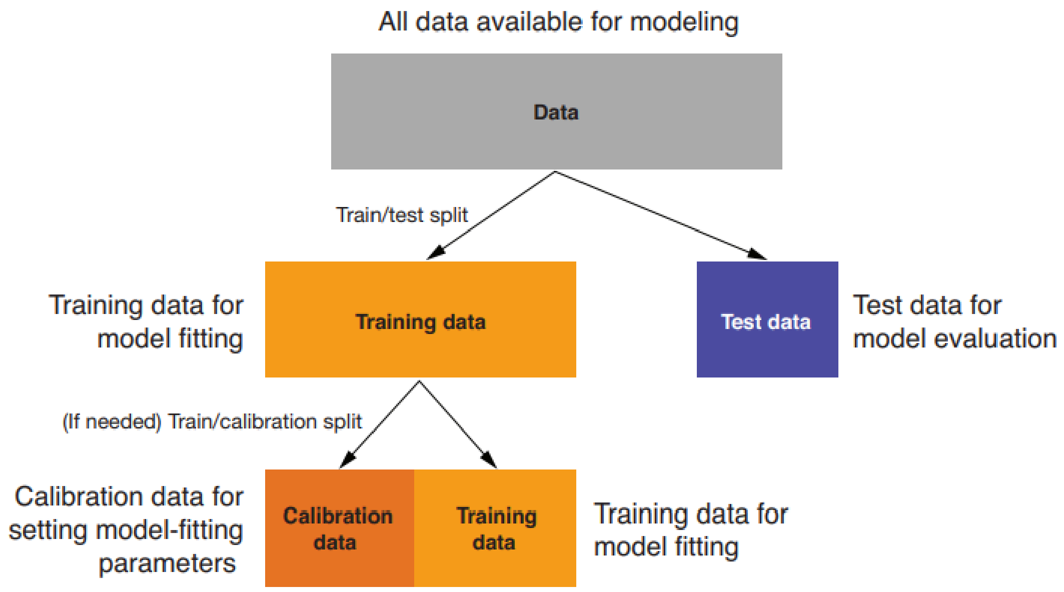

All data available for modeling
Data
Train/test split
Training data for model fitting
Training data
Test data
Test data for model evaluation
(If needed) Train/calibration split
Calibration data for setting model-fitting parameters
Calibration data
Training data
Training data for model fitting

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

When building a predictive model, the dataset is typically split into two main parts:

1. Training Set: Used to build and train the model.
2. Test (or Holdout) Set: Used to evaluate how well the model predicts on new, unseen data.

Optionally, a third split called the Calibration Set can be used

3.2 Creating a sample group column

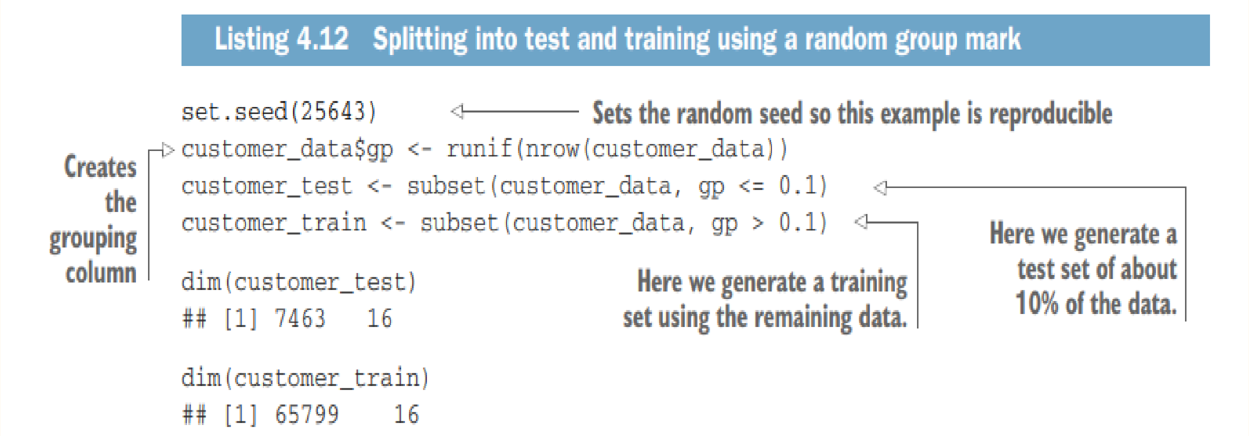
Sample Group Column: Add a column to the data frame with values generated uniformly between 0 and 1 using `runif()`.

Thresholds for Sampling: Use thresholds on the sample group column to select random subsets of data.

Example Thresholds:

- `gp < 0.4` selects about 40% of the data.
- `gp between 0.55 and 0.70` selects about 15% of the data.

Consistent Sampling: This method allows for repeatable random sampling of any desired size from the data.



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.

When sampling data for analysis, especially in cases where the unit of interest is a group (e.g., households) rather than individual elements (e.g., customers), the sampling approach needs to be adapted accordingly.

Unique Rows Sampling

- Basic Method: The standard sampling technique involves adding a sample group column with values generated using `runif()`, and then applying thresholds to this column to select subsets of data. This works well when each row in the dataset corresponds to a unique individual or object (e.g., each customer is a unique row).

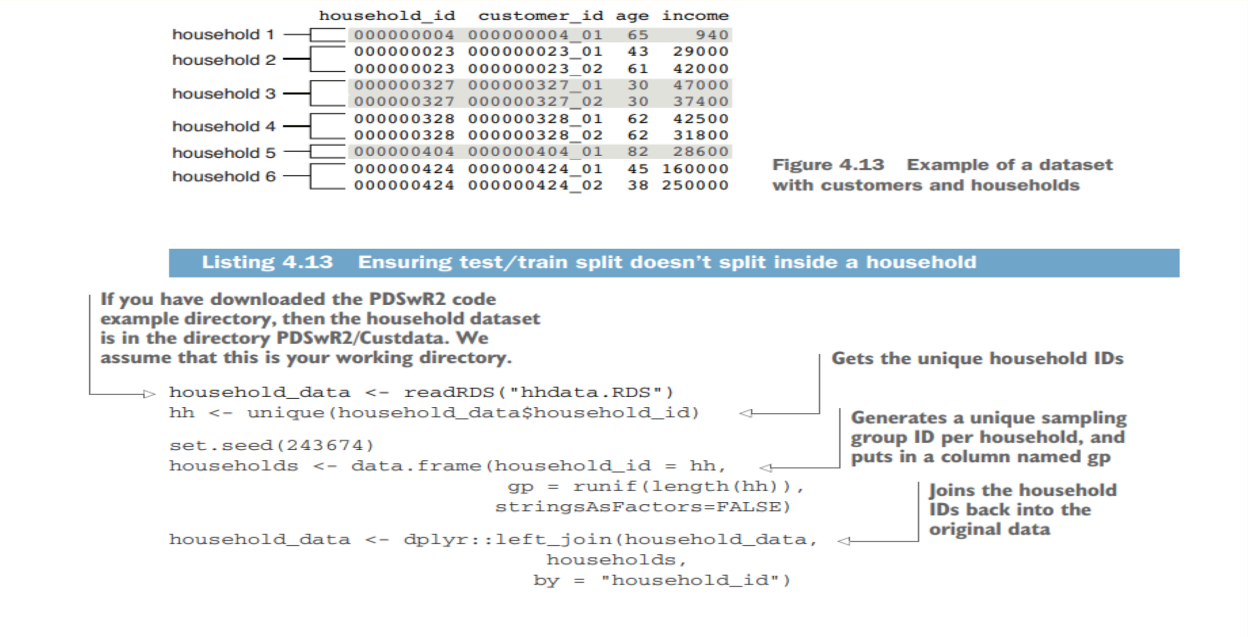
Group-Level Sampling

- Group Analysis: If your analysis is at the household level (e.g., you want to understand which households have uninsured members), you need to ensure that the sampling is done at the household level rather than at the individual level. This is because each household should be treated as a single unit in the sampling process.

Example Scenario

1. Data Structure: Suppose you have a dataset where each row represents an individual customer, but each customer is associated with a household ID. In this scenario, you are interested in the household level, so you need to ensure that all customers from a particular household are consistently assigned to either the training set or the test set.
2. Household IDs and Customer IDs: In your dataset, you might have:
   - Household ID: Identifies the household to which a customer belongs.
   - Customer ID: Identifies individual customers within a household.
3. Splitting Households: To split the data appropriately:
   - Generate Sample Groups at Household Level: First, assign a random sample group to each household based on the household ID.
   - Assign Groups to Customers: Once households are assigned to groups (training or test), ensure all customers within the same household are

included in the same group. This means that all rows with the same household ID will either be in the training set or the test set.



```
             household_id  customer_id  age  income
household 1   000000004   000000004_01   65     940
household 2   000000023   000000023_01   43   29000
              000000023   000000023_02   61   42000
household 3   000000327   000000327_01   30   47000
              000000327   000000327_02   30   37400
household 4   000000328   000000328_01   62   42500
              000000328   000000328_02   62   31800
household 5   000000404   000000404_01   82   28600
household 6   000000424   000000424_01   45  160000
              000000424   000000424_02   38  250000
```

Figure 4.13 Example of a dataset with customers and households

Listing 4.13 Ensuring test/train split doesn't split inside a household

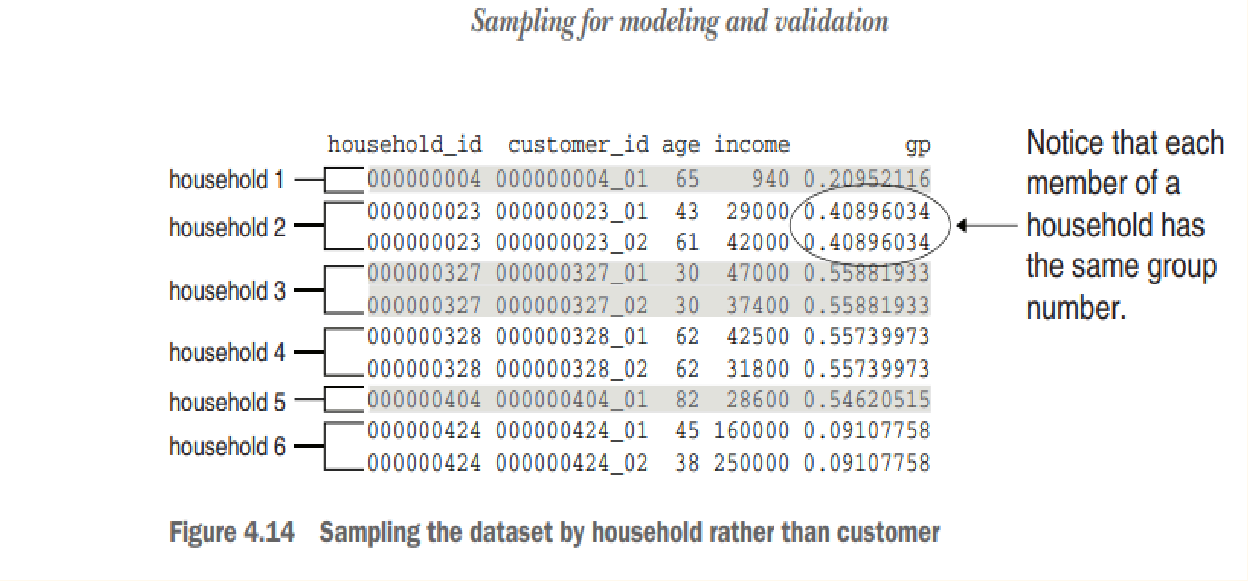
```
household_data <- readRDS("hhdata.RDS")
hh <- unique(household_data$household_id)

set.seed(243674)
households <- data.frame(household_id = hh,
                         gp = runif(length(hh)),
                        stringsAsFactors=FALSE)

household_data <- dplyr::left_join(household_data,
                                   households,
                                  by = "household_id")
```

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

Gets the unique household IDs

Generates a unique sampling group ID per household, and puts in a column named gp

Joins the household IDs back into the original data



Sampling for modeling and validation

```
             household_id  customer_id  age  income         gp
household 1   000000004  000000004_01   65     940 0.20952116
household 2   000000023  000000023_01   43   29000 0.40896034
              000000023  000000023_02   61   42000 0.40896034
household 3   000000327  000000327_01   30   47000 0.55881933
              000000327  000000327_02   30   37400 0.55881933
household 4   000000328  000000328_01   62   42500 0.55739973
              000000328  000000328_02   62   31800 0.55739973
household 5   000000404  000000404_01   82   28600 0.54620515
household 6   000000424  000000424_01   45  160000 0.09107758
              000000424  000000424_02   38  250000 0.09107758
```

Notice that each member of a household has the same group number.

Figure 4.14 Sampling the dataset by household rather than customer

Choosing and evaluating models

1. Evaluating models
    - Estimating model quality is crucial to ensure good real-world performance.
    - Data is often split into training and test sets for this purpose.
    - **Training Data:** Used to build the model.
    - **Test Data:** Not used during training; helps assess performance on new data.
    - Test data is useful for identifying overfitting.
    - **Overfitting:** When a model memorizes training data and fails to generalize to new data.
    - Detecting overfitting is a key step in diagnosing and improving models.

1.1 Overfitting:

- An overfit model performs well on training data but poorly on new data.
- **Training Error:** Prediction error on the data the model was trained on.
- **Generalization Error:** Prediction error on new, unseen data.
- Typically, training error is smaller than generalization error.
- Ideally, these two error rates should be close.
- A large generalization error and poor test performance indicate overfitting.
- Overfitting occurs when a model memorizes training data rather than discovering generalizable patterns.

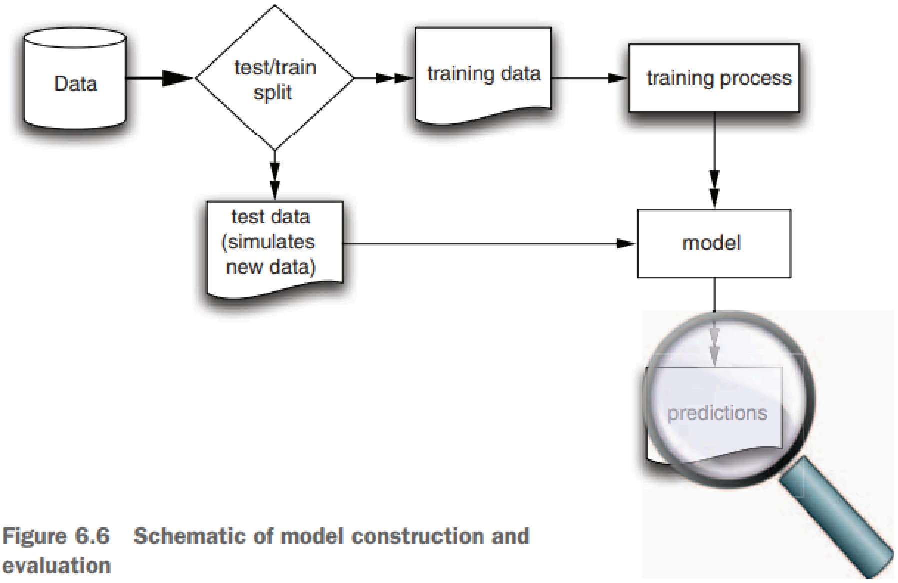
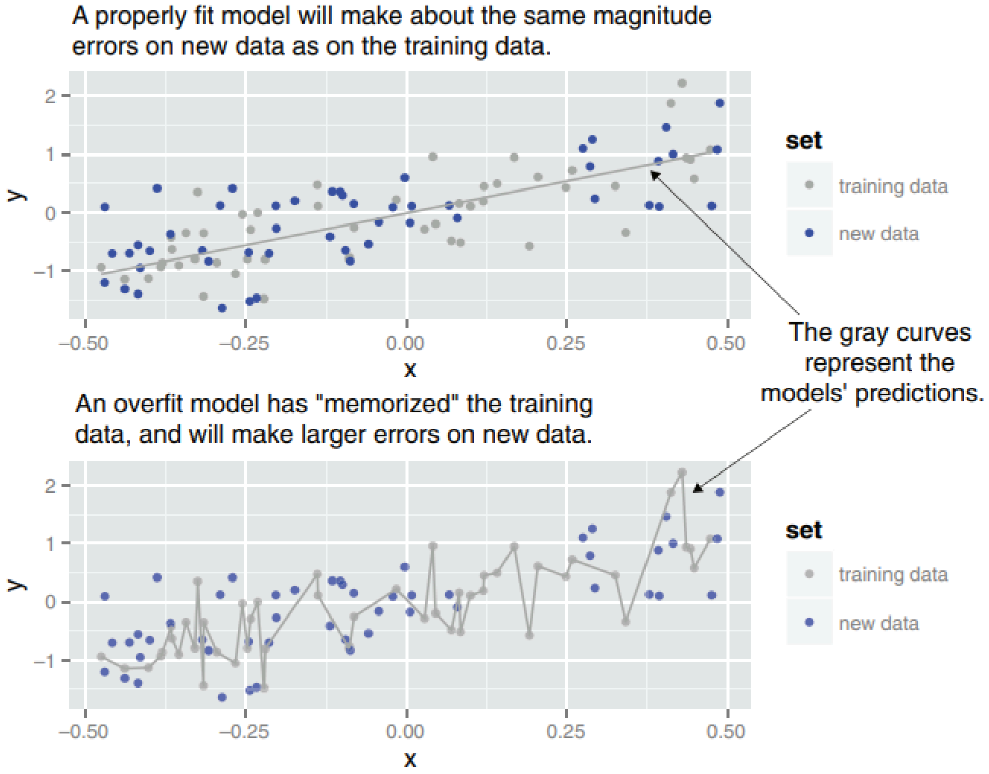


Figure 6.6 Schematic of model construction and evaluation


A properly fit model will make about the same magnitude errors on new data as on the training data.

An overfit model has "memorized" the training data, and will make larger errors on new data.

The gray curves represent the models' predictions.

Figure 6.7 A notional illustration of overfitting

TESTING ON HELD-OUT DATA

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

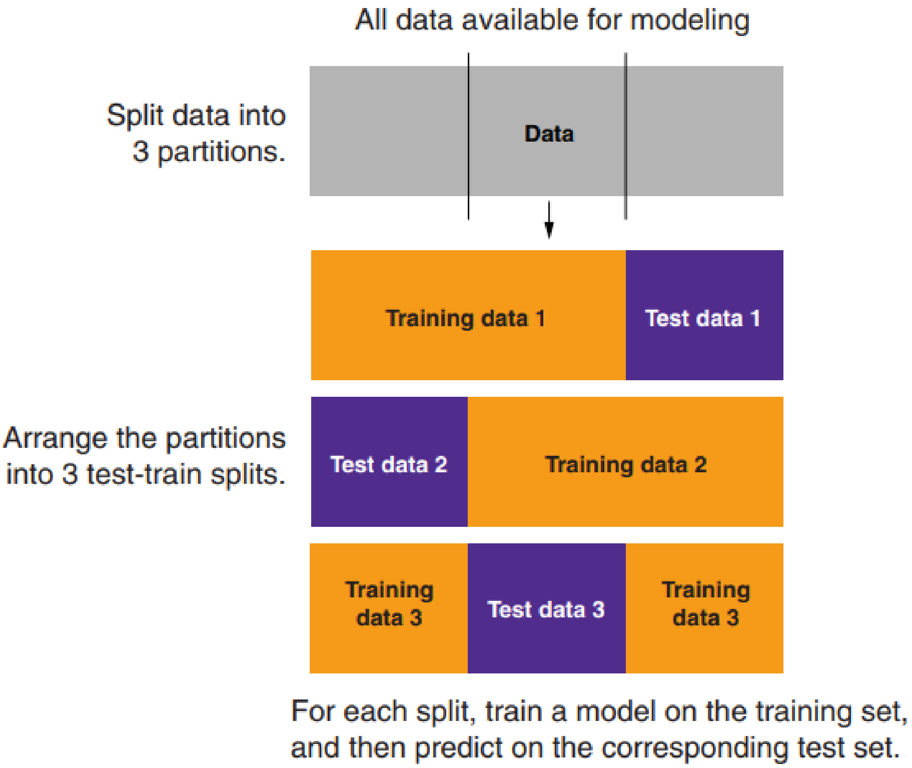
- Testing on holdout data is useful but not statistically efficient because **it uses each example only once**, either for model construction or evaluation.
- The test set is often **smaller than** the entire dataset, leading to less precise estimates of model performance.
- In cases where **the dataset is small**, splitting into training and test sets might not provide enough data for both building and evaluating models effectively.
- To address this, a more thorough partitioning method, **k-fold cross-validation**, can be used.

In **k-fold cross-validation**, the dataset is divided into *k* equal-sized subsets or "folds." The model is trained *k* times, each time using *k-1* folds for training and the remaining fold for evaluation. This process ensures that every data point is used for both training and evaluation, but never in both roles simultaneously.

Example for k = 3:

1. **First Iteration:** Train the model on folds 2 and 3, and evaluate on fold 1.
2. **Second Iteration:** Train on folds 1 and 3, and evaluate on fold 2.
3. **Third Iteration:** Train on folds 1 and 2, and evaluate on fold 3.

This method provides a more robust estimate of model performance by averaging the results from each fold.



All data available for modeling

Split data into 3 partitions.

Arrange the partitions into 3 test-train splits.

For each split, train a model on the training set, and then predict on the corresponding test set.

Figure 6.9 Partitioning data for 3-fold cross-validation

1.2 Measures of model performance

From an evaluation point of view, we group model types this way:

1. Classification

2. Scoring
3. Probability estimation
4. Clustering

To decide if a given score is high or low, we generally compare our model’s performance to a few baseline models.

THE NULL MODEL

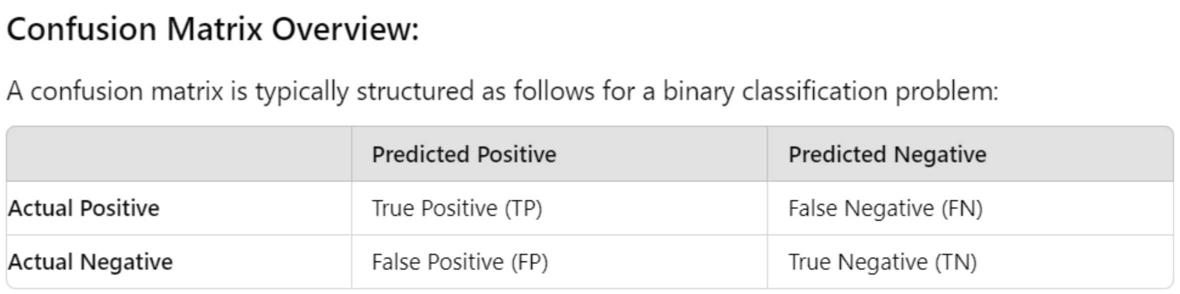
- The **null model** is a very simple model used as a baseline to outperform.
- The most common null model is a constant model that provides the same answer for all situations.
- **Categorical Problem:** The null model returns the most popular category (the easiest guess with the least errors).
- **Score Model:** The null model returns the average of all outcomes (minimizing the square deviation from outcomes).
- The null model serves as a lower bound for desired performance.
- If a model doesn't outperform the null model, it isn't delivering value.
- Despite being simple, the null model is effective because it knows the overall distribution of the data.

SINGLE-VARIABLE MODELS

- Compare any complex model against the best single-variable model available.
- A complex model isn't justified if it doesn't outperform the best single-variable model from the training data.
- Business analysts often use tools like pivot tables for effective single-variable models, so clients may expect better performance from complex models.

1.3 Evaluating classification models

To measure the performance of a classification model, the **confusion matrix** is a key tool. It provides a **summary of the prediction results by showing the number of correct and incorrect predictions for each class**. The confusion matrix allows us to calculate various evaluation metrics that help assess the model's effectiveness.



Confusion Matrix Overview:

A confusion matrix is typically structured as follows for a binary classification problem:

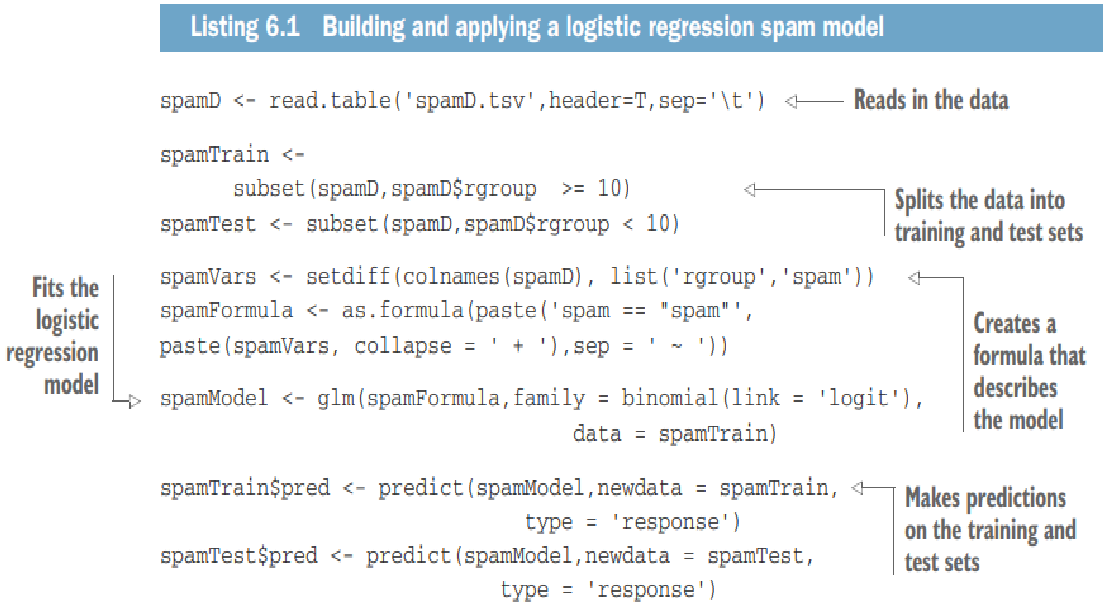
	Predicted Positive	Predicted Negative
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

True Positive (TP): Correctly predicted positive cases.

False Negative (FN): Actual positive cases that were incorrectly predicted as negative.

False Positive (FP): Actual negative cases that were incorrectly predicted as positive.

True Negative (TN): Correctly predicted negative cases.



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')
```

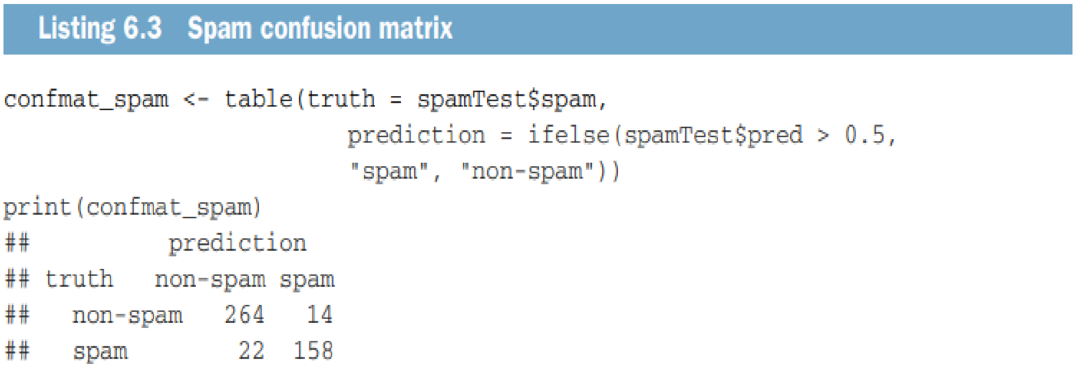
Reads in the data

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



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
```

ACCURACY Accuracy answers the question, “When the spam filter says this email is or is not spam, what’s the probability that it’s correct?” For a classifier, accuracy is defined as the number of items categorized correctly divided by the total number of items. It’s simply what fraction of classifications the classifier makes is correct

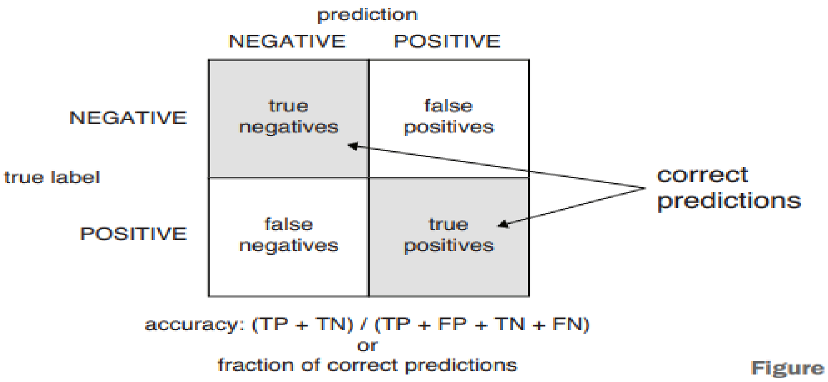


Figure 6.10 Accuracy

ACCURACY IS AN INAPPROPRIATE MEASURE FOR UNBALANCED CLASSES

PRECISION AND RECALL Another evaluation measure used by machine learning researchers is a pair of numbers called precision and recall.

Precision answers the question, “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. This is shown in figure 6.11.

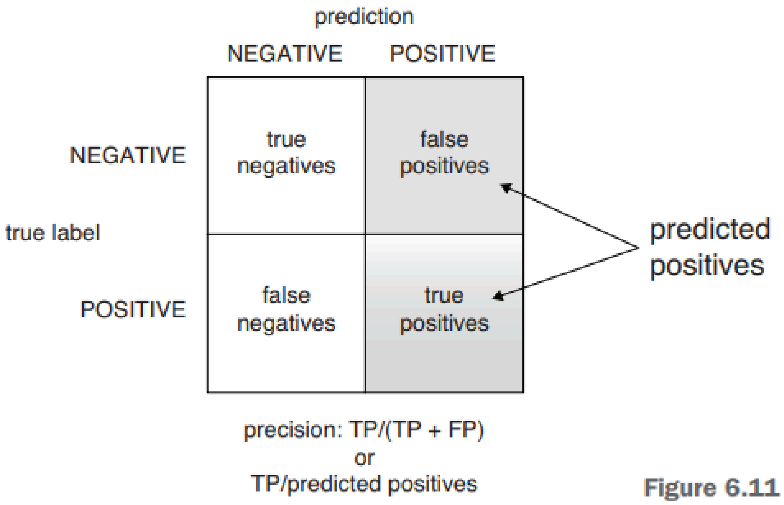


Figure 6.11 Precision

Recall answers the question, “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, as shown in figure 6.12.

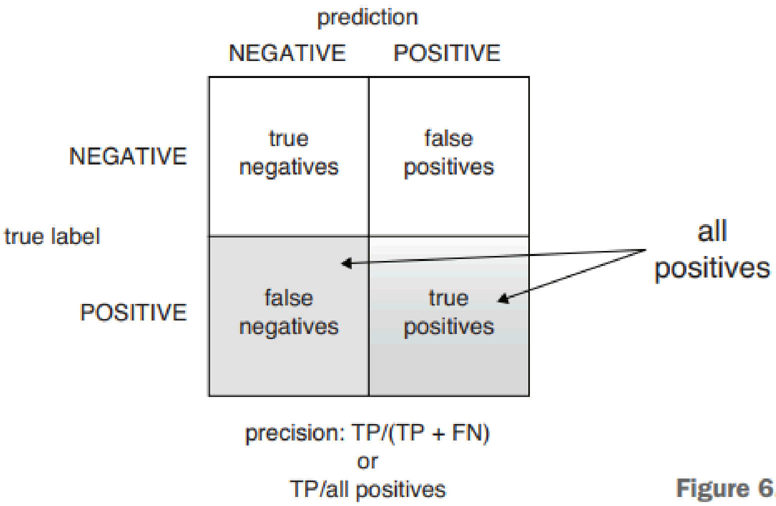


Figure 6.12 Recall

- **Precision:** Measures how often the classifier is correct when it indicates a positive result.
- **Recall:** Measures how much the classifier finds of what actually needs to be found.

In our 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.

For the email spam filter example:

- **Spam Filter Recall:** 88%, meaning 12% of spam still reaches the inbox.
- **Akismet Recall:** 99.88%, indicating most spam is tagged correctly.
- In spam filtering, precision is prioritized over recall to avoid losing important non-spam emails.

The F1 score measures a trade-off between precision and recall. It is defined as the harmonic mean of the precision and recall.

SENSITIVITY AND SPECIFICITY

Example Suppose that you have successfully trained a spam filter with acceptable precision and recall, using your work email as training data. Now you want to use that same spam filter on a personal email account that you use primarily for your photography hobby. Will the filter work as well?

The spam filter may work well on personal email if spam characteristics (such as email length, words used, and number of links) are similar to those on the work email.

However, the proportion of spam in personal versus work email can differ, affecting the filter's performance on personal email. This can change the performance of the spam filter on your personal email.

When the performance of a spam filter in varying conditions of spam prevalence and the importance of metrics that are independent of class prevalence and the recall of the filter is consistently around 88%, the precision changes based on the proportion of spam in the data compared to what the filter was trained on. Specifically:

- **Higher Precision**: When the data contains more spam than expected, precision increases, meaning fewer non-spam emails are incorrectly marked as spam. This is favorable.
- **Lower Precision**: When the data contains less spam than expected, precision decreases, leading to a higher proportion of non-spam emails being misclassified as spam. This is less desirable.

These metrics are particularly valuable in scenarios where the prevalence of the positive class (e.g., spam or a medical condition) can vary, making them common in fields like medical research for evaluating the effectiveness of diagnostic tests across different populations.

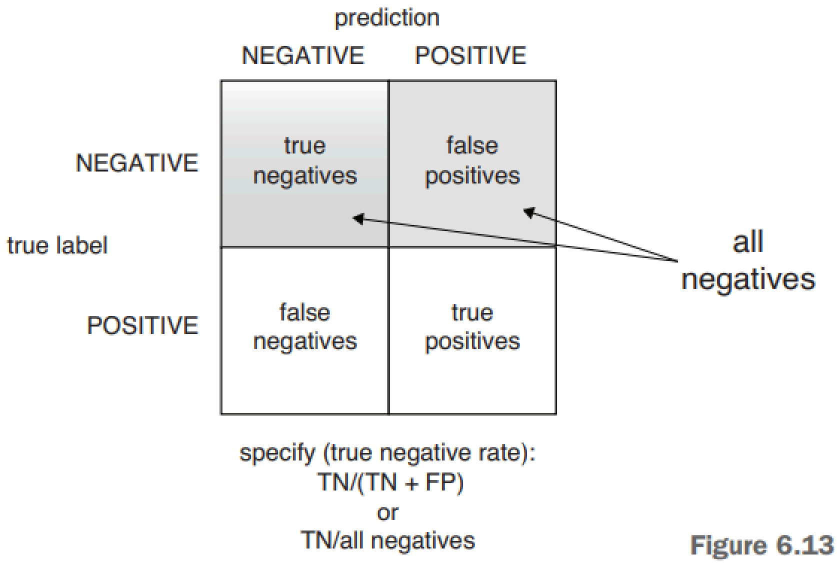


Figure 6.13 Specificity

Sensitivity/Recall and Specificity:

- Sensitivity (true positive rate) and recall measure the fraction of actual spam correctly identified by the spam filter.
- Specificity (true negative rate) measures the fraction of non-spam emails correctly identified as non-spam.

Calculating Specificity:

- The formula for specificity is: $\text{specificity} = \frac{\text{true negatives}}{\text{true negatives}+\text{false positives}}$.
- In this example, the specificity of the spam filter is approximately 0.95, meaning about 5% of non-spam emails are incorrectly marked as spam.

False Positive Rate:

- One minus the specificity is the false positive rate, which indicates the proportion of non-spam emails wrongly classified as spam. A low false positive rate (or high specificity) is desirable.

1.4 Evaluating scoring models

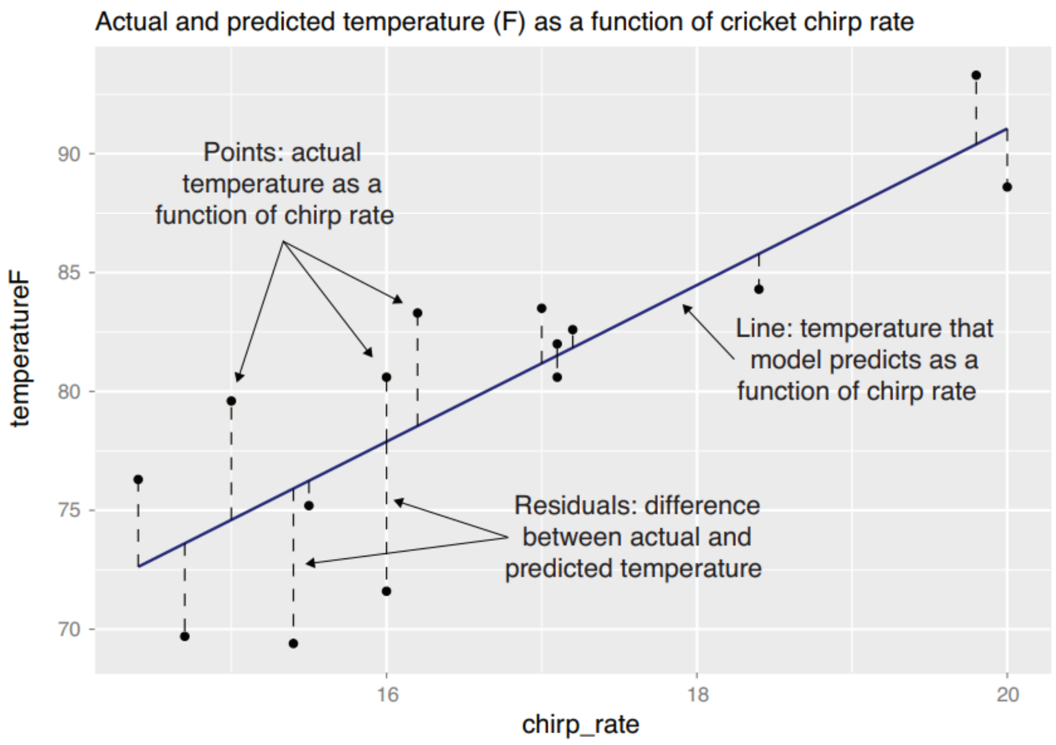
Let’s demonstrate evaluation on a simple example.

Example *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) of a striped ground cricket. Now you want to evaluate this model.*

You can fit a linear regression model to this data, and then make predictions, using the following listing

Listing 6.6 Fitting the cricket model and making predictions

```
crickets <- read.csv("cricketchirps/crickets.csv")

cricket_model <- lm(temperatureF ~ chirp_rate, data=crickets)
crickets$temp_pred <- predict(cricket_model, newdata=crickets)
```

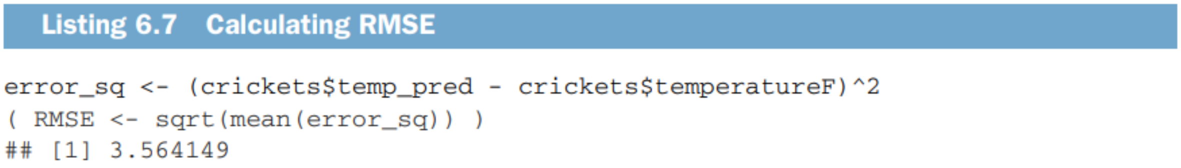


The residuals to calculate some common performance metrics for scoring models

ROOT MEAN SQUARE ERROR

Definition: RMSE is a common measure of goodness-of-fit, assessing how well a model's predictions match the actual observed values. It is calculated as the square root of the average of the squared residuals (differences between predicted and actual values).

RMSE answers the question, “How much is the predicted temperature typically off?”



Listing 6.7 Calculating RMSE

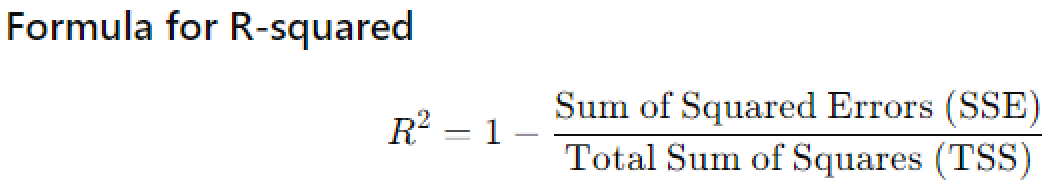
```
error_sq <- (crickets$temp_pred - crickets$temperatureF)^2
( RMSE <- sqrt(mean(error_sq)) )
## [1] 3.564149
```

Example Result: If the RMSE is 3.6 degrees Fahrenheit, this means the model's temperature predictions are typically about 3.6 degrees off from the actual temperatures.

Goal Evaluation: If the target was to have predictions within 5 degrees Fahrenheit, an RMSE of 3.6 degrees indicates that the model meets this goal, suggesting it is a good fit for predicting temperature based on the given data.

RMSE provides a single value that represents the typical prediction error of the model in the units of the actual data (in this case, degrees Fahrenheit). The lower the RMSE, the closer the predictions are to the actual values, indicating a better model fit.

R-squared (R^2), also known as the **coefficient of determination**, is a statistical measure that represents the **proportion of the variance in the dependent variable (outcome) that is predictable from the independent variable(s) (predictors)** in a regression model. It provides a measure of how well the observed outcomes are replicated by the model's predictions.



Formula for R-squared

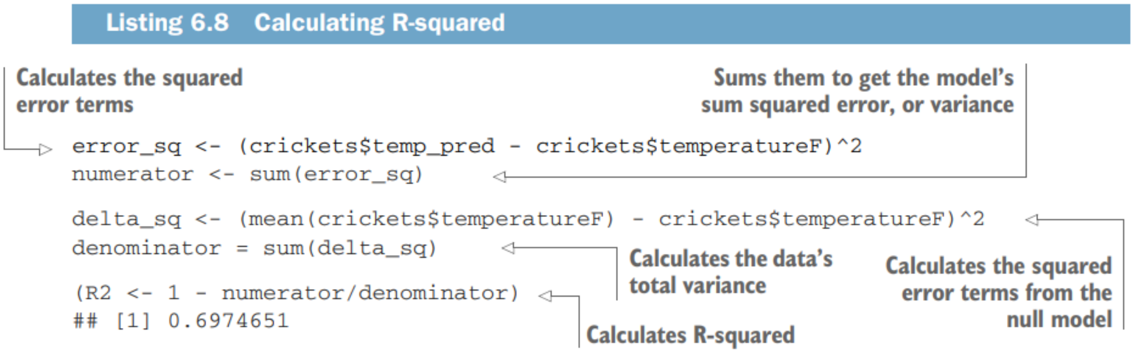
$$R^2 = 1 - \frac{\text{Sum of Squared Errors (SSE)}}{\text{Total Sum of Squares (TSS)}}$$

Where:

- **SSE (Sum of Squared Errors)**: The sum of the squared differences between the actual values and the predicted values from the model. This measures the model's error.
- **TSS (Total Sum of Squares)**: The sum of the squared differences between the actual values and the mean of the actual values. This represents the total variability in the data.

Intuition Behind R-squared

1. **Null Model**: The simplest model is the null model, which predicts the outcome variable as its average value. This model does not use any predictors.
2. **Model Comparison**: R-squared compares how much better the actual model (which uses predictors) is at predicting the outcome compared to the null model. A higher R-squared indicates that the model explains a larger portion of the variance in the outcome variable.
3. **Interpretation**:
   - If R2=0 it means the model does not explain any of the variability in the outcome; it performs no better than the null model.
   - If R2=1, it means the model explains all the variability in the outcome; the predictions are perfect.
   - A value between 0 and 1 indicates the percentage of the variance in the outcome that is explained by the model.



Listing 6.8 Calculating R-squared

```
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
```

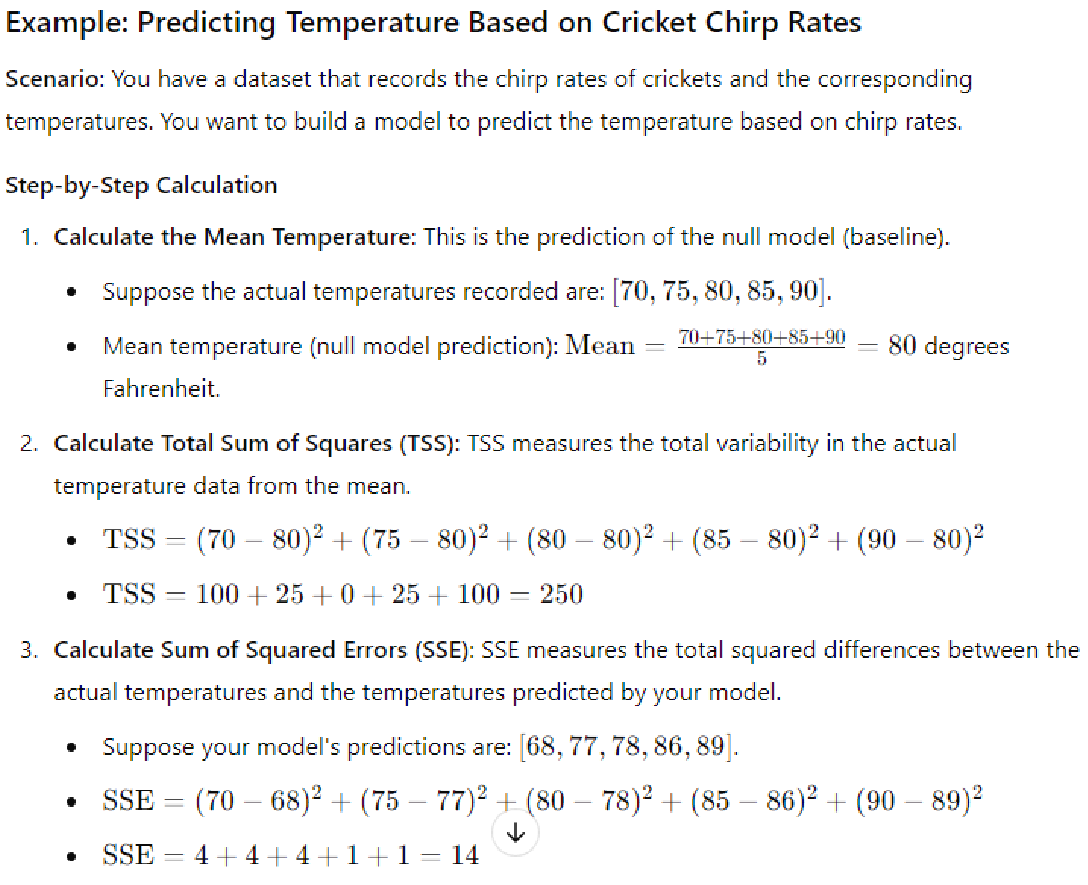
Calculates the squared error terms

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

Calculates the squared error terms from the null model

Calculates the data's total variance

Calculates R-squared

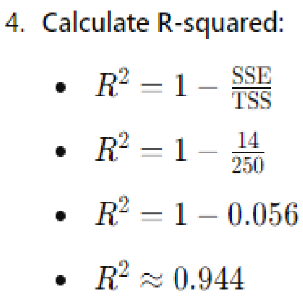


Example: Predicting Temperature Based on Cricket Chirp Rates

Scenario: You have a dataset that records the chirp rates of crickets and the corresponding temperatures. You want to build a model to predict the temperature based on chirp rates.

Step-by-Step Calculation

1. **Calculate the Mean Temperature:** This is the prediction of the null model (baseline).
   - Suppose the actual temperatures recorded are: $[70, 75, 80, 85, 90]$.
   - Mean temperature (null model prediction): $\text{Mean} = \frac{70+75+80+85+90}{5} = 80$ degrees Fahrenheit.
2. **Calculate Total Sum of Squares (TSS):** TSS measures the total variability in the actual temperature data from the mean.
   - $\text{TSS} = (70-80)^2 + (75-80)^2 + (80-80)^2 + (85-80)^2 + (90-80)^2$
   - $\text{TSS} = 100 + 25 + 0 + 25 + 100 = 250$
3. **Calculate Sum of Squared Errors (SSE):** SSE measures the total squared differences between the actual temperatures and the temperatures predicted by your model.
   - Suppose your model's predictions are: $[68, 77, 78, 86, 89]$.
   - $\text{SSE} = (70-68)^2 + (75-77)^2 + (80-78)^2 + (85-86)^2 + (90-89)^2$
   - $\text{SSE} = 4 + 4 + 4 + 1 + 1 = 14$



4. **Calculate R-squared:**
   - $R^2 = 1 - \frac{\text{SSE}}{\text{TSS}}$
   - $R^2 = 1 - \frac{14}{250}$
   - $R^2 = 1 - 0.056$
   - $R^2 \approx 0.944$

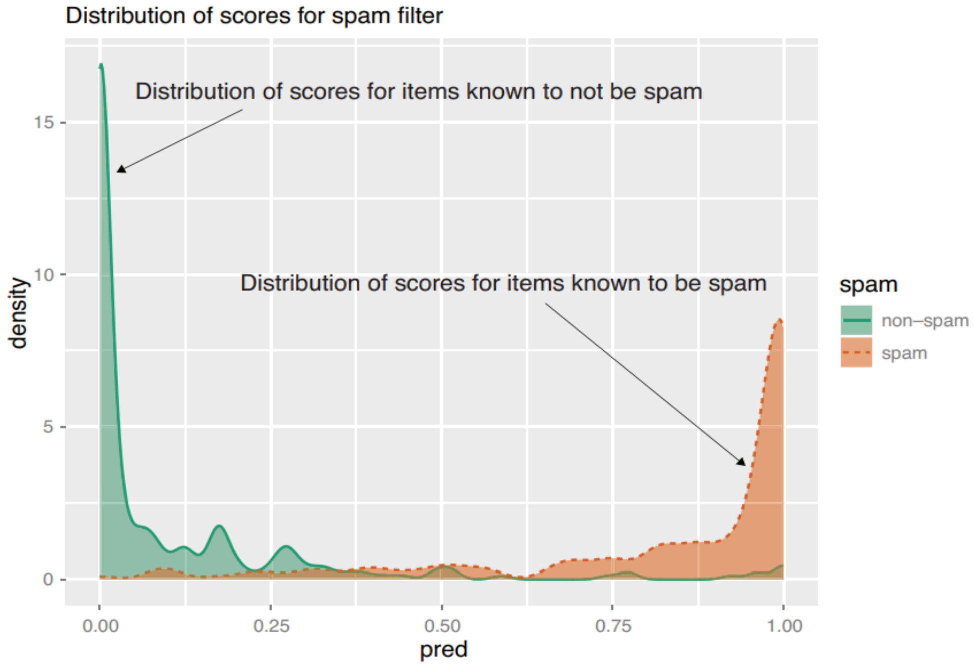
Interpretation

- An R-squared value of approximately 0.944 indicates that 94.4% of the variance in the temperature can be explained by the model based on cricket chirp rates.
- This high R-squared value suggests that the model fits the data well, significantly improving predictions compared to just using the mean temperature (null model).

1.5 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 DOUBLE DENSITY PLOT



1. Axes and Labels:

- **X-axis (pred)**: This axis represents the prediction scores returned by the spam filter model. The scores range from 0 to 1. A score close to 1 indicates a high probability of an email being spam, while a score close to 0 indicates a low probability of spam (or that the email is likely non-spam).

- **Y-axis (density)**: This axis represents the density (or frequency) of emails with particular prediction scores. Higher peaks indicate more emails falling within that score range.

2. Color Coding:

- **Green (solid area)**: This color represents the distribution of scores for emails known to be **non-spam**. These are legitimate emails that the model predicted.
- **Orange (dashed area)**: This color represents the distribution of scores for emails known to be **spam**.

3. Interpretation of Density Peaks:

- **High Peak Near 0 for Non-Spam (Green)**: Most non-spam emails have prediction scores close to 0. This indicates that the spam filter is correctly assigning low spam probabilities to legitimate emails. This high peak near zero suggests the filter effectively identifies non-spam emails.
- **High Peak Near 1 for Spam (Orange)**: Most spam emails have prediction scores close to 1. This high peak near one shows that the filter effectively assigns high spam probabilities to actual spam emails, which is a desirable behavior.

4. Overlapping Region:

- There is some overlap between the green and orange areas, particularly between 0 and 0.25 and around the middle of the range (0.5). This overlap indicates some uncertainty in the model:
  - **False Positives**: Non-spam emails (green) might have higher scores, falling under spam classification if a threshold is set low.
  - **False Negatives**: Spam emails (orange) might have lower scores, falling under non-spam classification if a threshold is set high.

5. Choosing a Threshold:

- **Threshold of 0.5** (the most common threshold): If used, it would mean labeling an email as spam if its score is above 0.5. This threshold would classify most of the non-spam emails correctly but might miss some spam emails that fall below this threshold.

- **Higher Threshold (e.g., 0.75)**: This would reduce false positives (incorrectly marking non-spam as spam) but might lower the recall (failing to catch all spam).
- **Lower Threshold (e.g., 0.25)**: This would increase recall (catching more spam) but might also increase the false positive rate (marking more non-spam as spam).

6. What This Means for Tuning the Spam Filter:

- **High Specificity (Less Overlap on Left)**: The plot indicates the model is good at identifying non-spam (most legitimate emails are correctly classified with low scores).
- **High Sensitivity (High Peak for Spam Near 1)**: The high density of scores near 1 for spam emails shows that the model effectively catches spam when the score is high.

In summary:

This double density plot helps visualize the performance of the spam filter model by comparing how it scores spam versus non-spam emails. By looking at the distribution of prediction scores, you can **decide on an appropriate threshold** to balance between minimizing false positives (non-spam marked as spam) and maximizing true positives (correctly identifying spam). Adjusting the threshold will help achieve the desired balance of precision and recall depending on the specific business needs or user experience goals.

THE RECEIVER OPERATING CHARACTERISTIC CURVE AND THE AUC

ROC Curve Definition: The ROC curve is a graphical representation that shows the performance of a binary classification model by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) at various threshold settings.

True Positive Rate (TPR): Also known as sensitivity, it represents the proportion of actual positives (e.g., spam emails) that are correctly identified by the model.

False Positive Rate (FPR): It represents the proportion of actual negatives (e.g., non-spam emails) that are incorrectly classified as positives by the model.

Purpose of the ROC Curve: The ROC curve demonstrates the trade-offs between the sensitivity (true positive rate) and specificity (related to the false positive rate) of the model across different classification thresholds.

Interpretation of the ROC Curve:

- A point on the ROC curve indicates a specific TPR and FPR for a chosen threshold.
- The curve moves closer to the top-left corner as the model becomes better, indicating high TPR and low FPR.
- A diagonal line (45-degree line) represents a random classifier with no discriminative ability. An effective model's ROC curve will bow towards the top-left corner.

Area Under the Curve (AUC): The AUC measures the overall performance of the model. It is the area under the ROC curve:

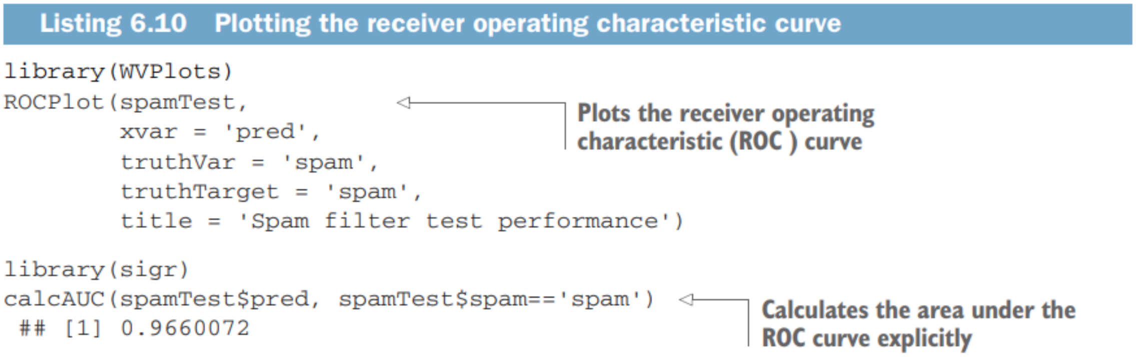
- **AUC Value Range**: The AUC ranges from 0 to 1.
- **AUC Interpretation**: An AUC of 0.5 indicates a model with no discrimination ability (same as random guessing). A higher AUC indicates better model performance. An AUC of 1.0 represents a perfect model.

Comparison of Models: Models can be compared using the AUC value. A model with a higher AUC is considered to have better distinguishing power between positive and negative classes.

Practical Use: The ROC curve helps in selecting the best threshold for classifying items as positive or negative, providing insight into the balance between correctly identifying positives and avoiding false positives.

Example in Context: For a spam filter, the ROC curve shows how changing the threshold for classifying an email as spam (versus non-spam) impacts the rates of true positives (correctly catching spam) and false positives (incorrectly classifying non-spam as spam).

In summary, the ROC curve and AUC provide a comprehensive method to evaluate the trade-offs and overall effectiveness of binary classification models, helping in the selection and tuning of the model for optimal performance.



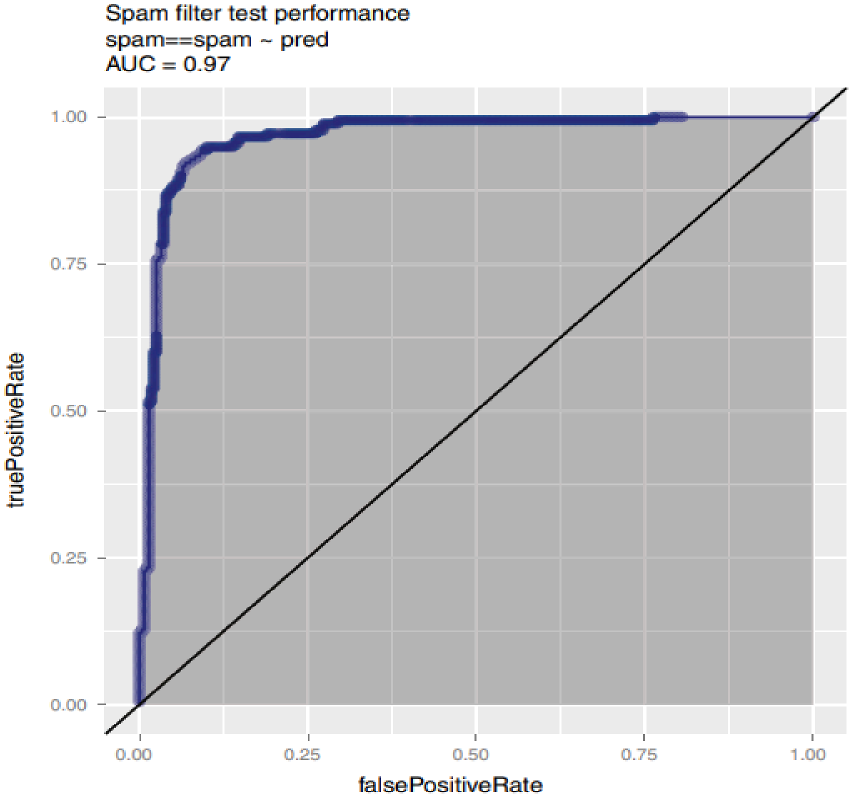
Listing 6.10 Plotting the receiver operating characteristic curve

```
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

Spam filter test performance
spam==spam ~ pred
AUC = 0.97



ROC Curve:

- **X-axis (False Positive Rate - FPR):** The proportion of negative instances (e.g., non-spam emails) incorrectly classified as positive (spam).

- **Y-axis (True Positive Rate - TPR):** The proportion of positive instances (e.g., spam emails) correctly classified as positive (spam).

Ideal Model:

- **Thresholds:**
  - **p = 1:** If the threshold is set at 1, no email is classified as spam (both TPR and FPR are 0), resulting in the point (0,0) on the ROC curve.

When the threshold is set to 1, it means that only instances with a predicted probability of 1 are classified as positive (spam, in this case).

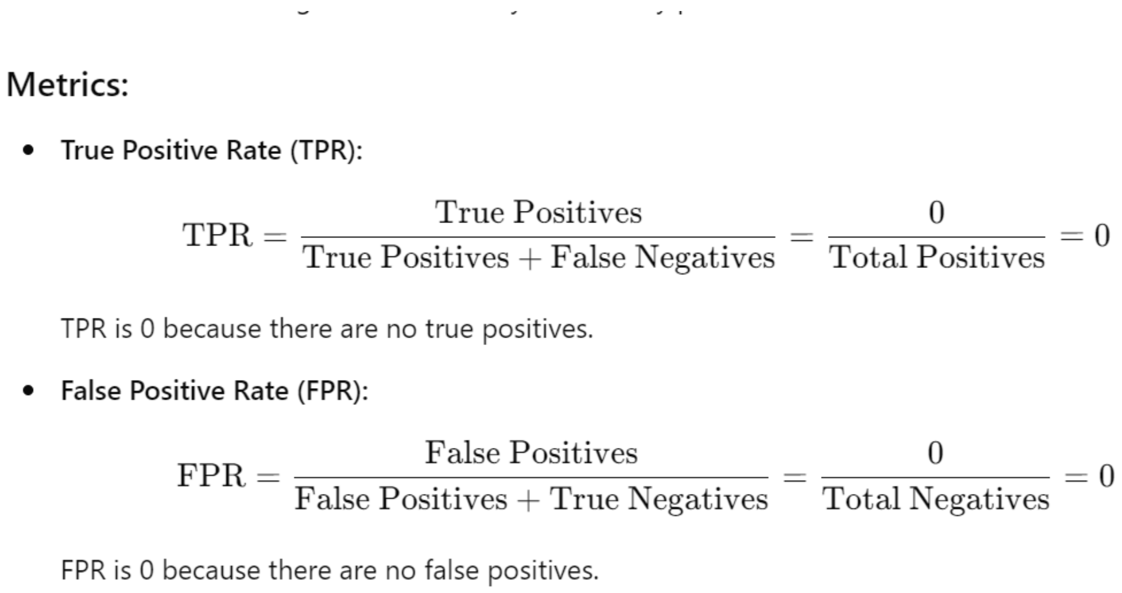
Threshold = 1

Definition: The classification threshold is set to 1, which is the maximum value of predicted probabilities (assuming probabilities are between 0 and 1).

Classification Rule: An instance is classified as positive only if its predicted probability is exactly 1. If the predicted probability is less than 1, the instance is classified as negative.

Outcome:

- **True Positives (TP):** None, because no instance has a predicted probability of exactly 1.
- **False Positives (FP):** None, because no instance is classified as positive.
- **True Negatives (TN):** All instances with predicted probabilities less than 1 are classified as negative.
- **False Negatives (FN):** All instances with predicted probabilities less than 1 are incorrectly classified as negative, even if they are actually positive.



Metrics:

- **True Positive Rate (TPR):**

$$\text{TPR} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} = \frac{0}{\text{Total Positives}} = 0$$

TPR is 0 because there are no true positives.

- **False Positive Rate (FPR):**

$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}} = \frac{0}{\text{Total Negatives}} = 0$$

FPR is 0 because there are no false positives.

- **p = 0:** If the threshold is set at 0, all emails are classified as spam (both TPR and FPR are 1), resulting in the point (1,1) on the ROC curve.

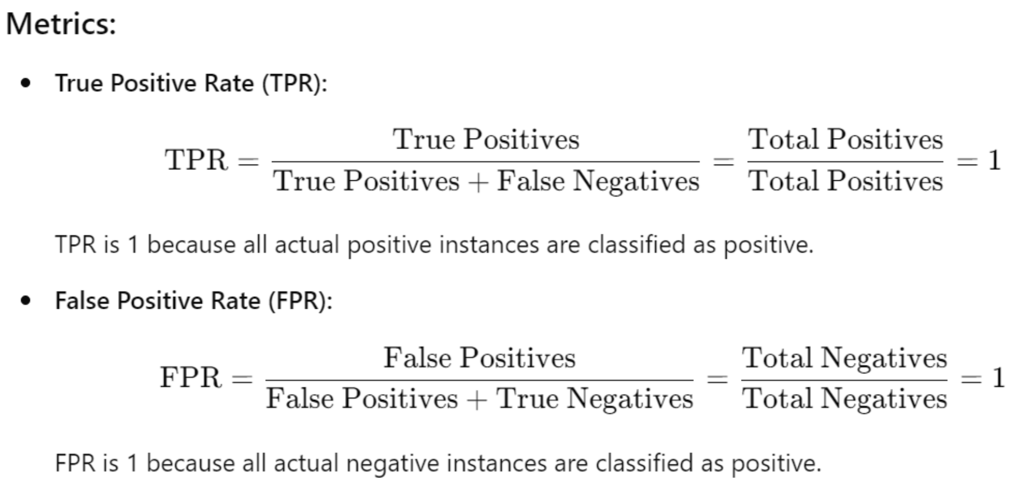
Threshold = 0

Definition: The classification threshold is set to 0, which is the minimum value of predicted probabilities (assuming probabilities range from 0 to 1).

Classification Rule: Every instance is classified as positive if its predicted probability is greater than or equal to 0. In practice, this means all instances are classified as positive because all predicted probabilities are ≥ 0.

Outcome:

- **True Positives (TP):** All instances with actual positive labels (spam emails) are classified as positive.
- **False Positives (FP):** All instances with actual negative labels (non-spam emails) are classified as positive.
- **True Negatives (TN):** None, because all instances are classified as positive.
- **False Negatives (FN):** None, because no positive instances are misclassified as negative.



Metrics:

- True Positive Rate (TPR):

$$\text{TPR} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} = \frac{\text{Total Positives}}{\text{Total Positives}} = 1$$

TPR is 1 because all actual positive instances are classified as positive.

- False Positive Rate (FPR):

$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}} = \frac{\text{Total Negatives}}{\text{Total Negatives}} = 1$$

FPR is 1 because all actual negative instances are classified as positive.

- ■ **$0 < p < 1$:** If the threshold is set between 0 and 1, the ideal model correctly classifies all emails (TPR = 1, FPR = 0), resulting in the point (0,1) on the ROC curve.

- ○ **ROC Shape for Ideal Model:**
  - ■ The curve for an ideal model would go from (0,0) to (0,1) and then from (0,1) to (1,1), forming a right-angle corner at (0,1). This is because the ideal model perfectly distinguishes between spam and non-spam at any threshold between 0 and 1.
  - ■ The area under this curve (AUC) is 1, which represents a perfect classifier.

Random Model:

- ○ For a random classifier, the ROC curve would be a diagonal line from (0,0) to (1,1). The AUC for such a model would be 0.5, indicating no better performance than random guessing.

Interpreting AUC:

- ○ AUC values range from 0.5 (random guessing) to 1 (perfect classifier). A higher AUC indicates a better-performing model.
- ○ When comparing models, those with a higher AUC are generally preferred, but the shape of the ROC curve should also be considered, as it reflects

the trade-offs between the true positive rate and the false positive rate at different thresholds.

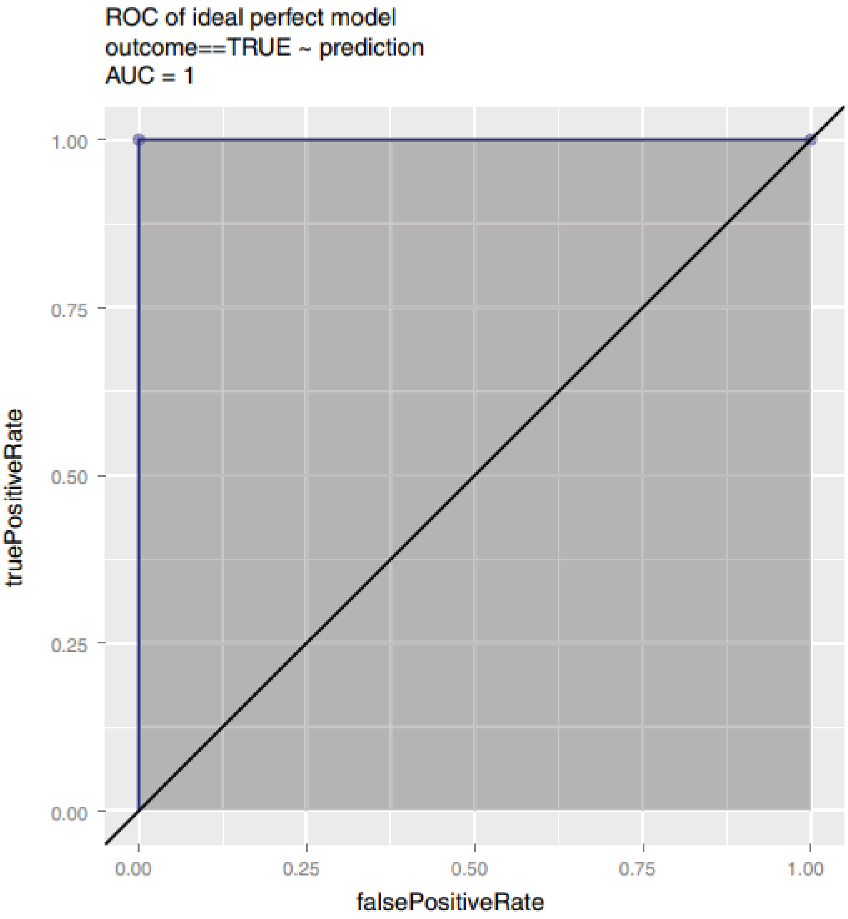


Figure 6.17 ROC curve for an ideal model that classifies perfectly

Summary

The ROC curve in the provided figure shows the ideal performance of a perfect model with an AUC of 1. It perfectly separates spam from non-spam at any reasonable threshold, making it the best possible scenario for a binary classifier. The diagonal line represents random performance with an AUC of 0.5. When evaluating real models, aiming for an AUC close to 1 is ideal, but the specific trade-offs between true positive and false positive rates should also be considered based on the project's goals.

LOG LIKELIHOOD

Log Likelihood as a Measure:

- Log likelihood is used to evaluate how well a model's predictions align with the actual class labels. It quantifies the fit of the model in probabilistic terms.

Non-Positive Value:

- The log likelihood is always a non-positive number (i.e., ≤ 0). A value closer to 0 indicates a better fit, while values further from 0 (more negative) indicate a worse fit.

Perfect Match (Log Likelihood = 0):

- When the log likelihood is 0, it implies a perfect match. In this scenario, the model predicts the correct class with complete certainty (e.g., predicting spam as spam with a probability of 1 and non-spam as non-spam with a probability of 1).

Magnitude and Model Fit:

- The larger the magnitude of the log likelihood (i.e., the more negative it is), the worse the model's predictions match the actual class labels. A larger negative value indicates that the model is less confident or incorrect in its predictions.

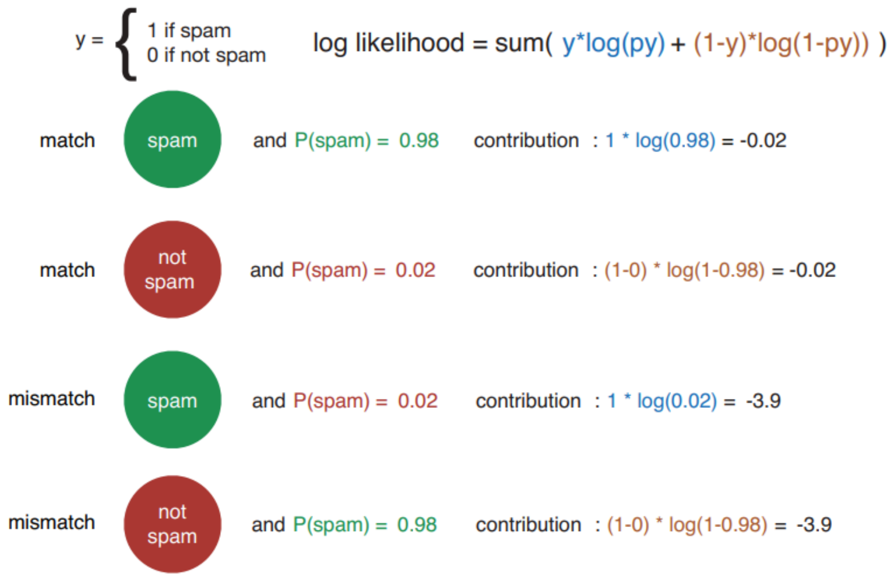

y = 1 if spam
0 if not spam
log likelihood = sum(y*log(py) + (1-y)*log(1-py)))
match
spam
and P(spam) = 0.98
contribution : 1 * log(0.98) = -0.02
match
not spam
and P(spam) = 0.02
contribution : (1-0) * log(1-0.98) = -0.02
mismatch
spam
and P(spam) = 0.02
contribution : 1 * log(0.02) = -3.9
mismatch
not spam
and P(spam) = 0.98
contribution : (1-0) * log(1-0.98) = -3.9

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

DEVIANCE

Deviance is a key concept in statistical modeling, particularly in the context of probability models such as logistic regression. It is used to assess the goodness of fit of a model by comparing the fit of the model under consideration to a perfect model.

Deviance is defined as:

$$\mathrm{Deviance} = -2 \times (\mathrm{logLikelihood} - S)$$

Where:

- logLikelihood is the log likelihood of the model you're evaluating.
- S is the log likelihood of the saturated model (a hypothetical model that perfectly predicts the observed data, with probability 1 for the correct class and 0 for the incorrect class).

Simplified Formula: In many practical cases, S is considered to be 0 (especially for generalized linear models), so the deviance simplifies to:

$$\text{Deviance} = -2 \times \text{logLikelihood}$$

Interpreting Deviance

- Deviance as a Measure of Fit:
  - Deviance is essentially a measure of how much worse the fitted model is compared to the perfect model (saturated model).
  - **The lower the deviance, the better the model fits the data.** A deviance of 0 indicates a perfect fit (where the model's predictions perfectly match the observed outcomes).
- Comparing Models Using Deviance:
  - In practice, the deviance of a model is often compared to the null deviance. The null deviance is the deviance of a model with no predictors, only an intercept. It represents the fit of a model that just predicts the mean outcome for all observations.

Null Deviance vs. Model Deviance

- Null Deviance:
  - This represents the deviance of the simplest possible model, which includes no predictors and just predicts the overall mean of the response variable.
  - It serves as a baseline for model comparison.
- Model Deviance:
  - This is the deviance of the model with predictors. It tells us how much better the current model is compared to the null model.
- Deviance Ratio:
  - The ratio of the null deviance to the model deviance gives an indication of how much the model has improved by including predictors.
  - **A large reduction in deviance from the null model to the fitted model indicates that the predictors explain a significant portion of the variance in the data.**

$$\text{Deviance Ratio} = \frac{\text{Null Deviance}}{\text{Model Deviance}}$$

The Akaike Information Criterion (AIC) is a widely used measure for comparing the goodness of fit of statistical models **while penalizing for model complexity.** It helps in selecting a model that fits the data well without being overly complex.

1. Definition of AIC

AIC is defined as:

$$\text{AIC} = \text{Deviance} + 2 \times \text{Number of Parameters}$$

Where:

- Deviance: As discussed earlier, deviance measures how well a model fits the data. Lower deviance indicates a better fit.
- Number of Parameters: This includes the number of predictors (including the intercept) in the model. More parameters mean a more complex model.

2. Purpose of AIC

- Model Fit vs. Complexity:
  - AIC balances two competing aspects: model fit and model complexity.
  - Deviance alone focuses on how well the model fits the data, but a model with more parameters can always be made to fit the data better, potentially leading to overfitting.
  - AIC adds a **penalty for the number of parameters** to account for the fact that more complex models might fit the training data well but may not generalize to new data.
- Model Comparison:
  - AIC is particularly useful when comparing multiple models. When you have several models, you can compute the AIC for each one, and generally, you prefer the model with the lowest AIC.

3. Interpreting AIC

- **Lower AIC is Better**:
  - **A lower AIC value indicates a better balance between model fit and complexity**. However, it doesn't give an absolute measure of model quality, only relative to other models being compared.
- Comparing Different Models:
  - You can compare models with different numbers of parameters and different predictors. The model with the smallest AIC is usually preferred because it provides the best trade-off between goodness of fit and simplicity.

What is the LIME ?

LIME (Local Interpretable Model-agnostic Explanations) is a framework designed to explain individual predictions made by machine learning models, especially black-box models like neural networks, gradient boosting machines, or support vector machines. LIME **helps to interpret how features contribute to the model's decision-making process for specific predictions**.

The main goal of LIME is to offer **local explanations** for predictions, meaning it generates explanations that are relevant to a particular instance (a single prediction) rather than trying to explain the entire model globally. This makes it very useful when dealing with complex models that are otherwise difficult to interpret.

How LIME Predicts and Explains Individual Samples:

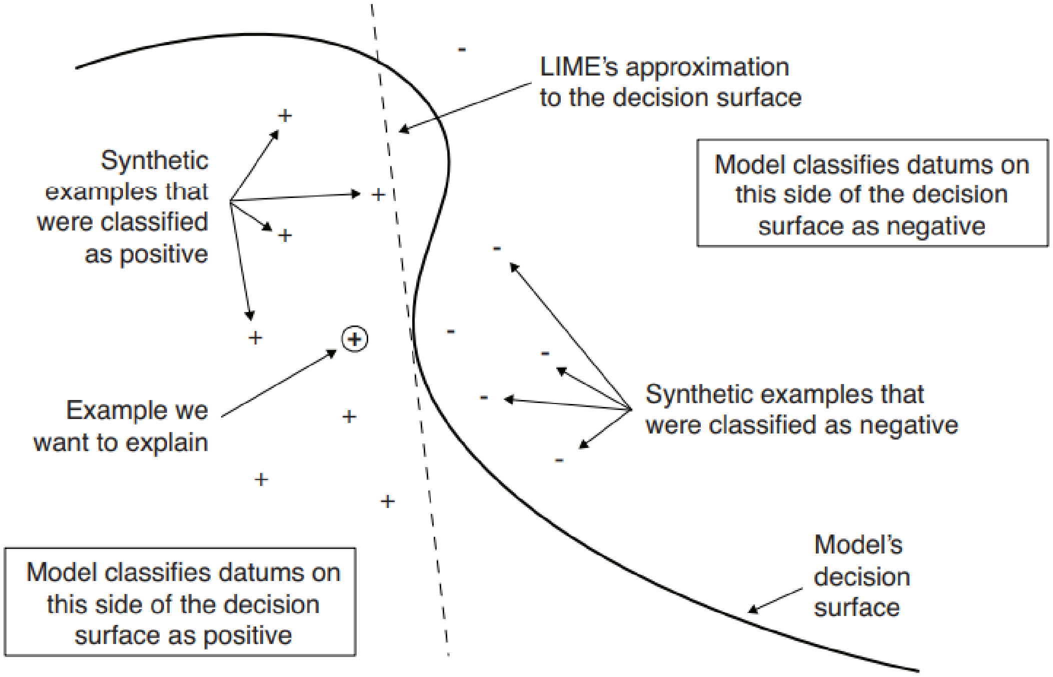
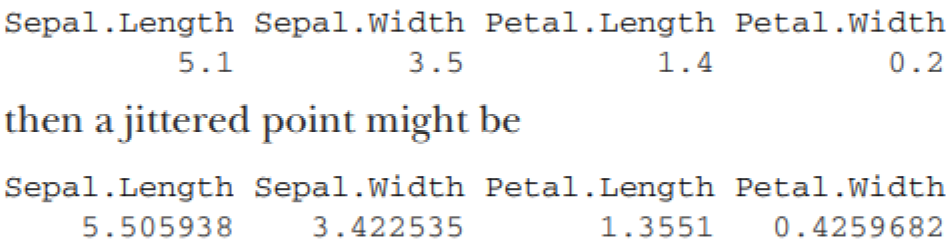

LIME's approximation to the decision surface
Synthetic examples that were classified as positive
Model classifies datums on this side of the decision surface as negative
Example we want to explain
Synthetic examples that were classified as negative
Model classifies datums on this side of the decision surface as positive
Model's decision surface
+
-

Figure 6.23 Notional sketch of how LIME works

1. **Training a Machine Learning Model:** You first train a machine learning model (such as XGBoost or a neural network) on your dataset.
2. **Select a Sample to Explain:** After making predictions, you select a specific sample (an individual data point) whose prediction you want to understand.
3. **Perturb the Sample:** LIME creates a new dataset by perturbing the selected sample's feature values slightly, generating synthetic data points around it.



```
Sepal.Length Sepal.Width Petal.Length Petal.Width
         5.1         3.5          1.4         0.2
```

then a jittered point might be

```
Sepal.Length Sepal.Width Petal.Length Petal.Width
    5.505938    3.422535       1.3551   0.4259682
```

To make sure that the synthetic examples are plausible, LIME uses the distributions of the data in the training set to generate the jittered data.

4. **Get Predictions for Synthetic Data:** The model is then asked to predict the labels (or probabilities) for these synthetic data points. The idea is to see how changes in the feature values influence the model's predictions.
5. **Fit a Local Linear Model:** LIME fits a simple linear model (like logistic regression) to these synthetic data points, with the model's predictions as the dependent variable and the feature values as independent variables. LIME assumes that the model's decision surface is locally linear (flat) in a small neighborhood around example. You can think of LIME's estimate as the flat surface (in the figure, it's a line) that separates the positive synthetic examples from the negative synthetic examples most accurately
6. **Generate Explanations:** The coefficients of this linear model provide the explanation. Features with high positive weights strongly support the original prediction, while features with negative weights suggest the opposite.
7. **Visualization:** LIME also provides visualizations, like feature importance plots, to make the explanation more accessible to users.

R code:

```
# Install required packages if you don't have them
# install.packages("xgboost")
# install.packages("lime")
# install.packages("caret")

# Load libraries
library(xgboost)
library(lime)
library(caret)

# Load Iris dataset
data(iris)

# Convert species to a binary problem (setosa vs. non-setosa)
iris$class <- as.numeric(iris$Species == "setosa")

# Split data into training and test sets
set.seed(2345)
intrain <- createDataPartition(iris$class, p = 0.75, list = FALSE)
```

`createDataPartition()`:

This function comes from the `caret` package. It is used to create data partitions, commonly for splitting a dataset into training and test sets in machine learning tasks.

`iris$class`:

This references the `class` column in the `iris` dataset. The `iris` dataset is a built-in dataset in R, but it does not have a `class` column by default. Usually, the target variable in the `iris` dataset is the `Species` column. So, this would work if there was a column named `class` (or it could be a typo for `iris$Species`).

`p = 0.75`:

This sets the proportion of data that should go into the training set. In this case, 75% of the data will be selected for training.

`list = FALSE`:

This argument ensures that the output is returned as an index matrix instead of a list. When `list = FALSE`, the function returns a matrix of row indices indicating which rows belong to the training set.

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

# Prepare data for xgboost
train_matrix <- as.matrix(train[, 1:4])
test_matrix <- as.matrix(test[, 1:4])
train_labels <- train$class
test_labels <- test$class

# Train an XGBoost model
model <- xgboost(data = train_matrix, label = train_labels,  objective = "binary:logistic",
nrounds = 50,  verbose = 0)
```

`xgboost()`:

`xgboost` is a function from the **`xgboost`** package, used for training a model using the eXtreme Gradient Boosting algorithm, which is a powerful machine learning technique.

`data = train_matrix`:

This argument provides the **training data** in matrix form (i.e., `train_matrix`). XGBoost expects the input data to be in matrix format (or a `DMatrix` object).

`label = train_labels`:

This provides the **target labels** (or responses) for the training data. In this case, `train_labels` contains the binary class labels for each observation in `train_matrix`.

`objective = "binary:logistic"`:

This specifies the **objective function** to be used during training. The objective `"binary:logistic"` is used for **binary classification** problems. It instructs XGBoost to model the probability that an instance belongs to class 1 (logistic regression for binary classification). The output will be probabilities, and a threshold (e.g., 0.5) can be applied to classify instances.

`nrounds = 50`:

This specifies the number of **boosting rounds** (or iterations) that the model will go through. Each round builds a new tree and refines the model. In this case, the model will go through 50 iterations.

`verbose = 0`:

This controls the level of output printed during training. A value of `0` means no output (silent mode). If you wanted more feedback during training (such as progress or evaluation metrics), you could set this to `1` or higher.

```
# Make predictions on test data
predictions <- predict(model, newdata = test_matrix)
predicted_class <- ifelse(predictions > 0.5, 1, 0)

# Create a confusion matrix to evaluate model performance
confusionMatrix(factor(predicted_class), factor(test_labels))

# LIME explanation
explainer <- lime(train[, 1:4], model = model, bin_continuous = TRUE, n_bins = 5)
```

`lime()`:

`lime()` is a function from the **`lime`** package, which is used to explain the predictions of any black-box machine learning model (e.g., XGBoost in your case). The goal of LIME is to provide local, interpretable explanations for individual predictions by approximating the complex model with simpler, interpretable models.

`train[, 1:4]`:

- This argument specifies the **training data** that is used as the reference for explanations.
- Here, `train[, 1:4]` indicates that only the first four columns of the `train` dataset are used. This is commonly the feature set (input variables), excluding the target or label column.

`model = model`:

This specifies the **machine learning model** that you want to explain. In this case, the model trained earlier (assigned to `model`) is used. This model was likely trained using the XGBoost algorithm as indicated in your previous code.

`bin_continuous = TRUE`:

- This argument controls whether continuous (numeric) features should be **binned** into discrete intervals or not.
- When `bin_continuous = TRUE`, LIME will divide continuous variables into a set of discrete bins to simplify the explanation process, as it's often easier to interpret categorical data. This makes the explanation model (like linear models or decision trees) simpler and more interpretable.

`n_bins = 5`:

This specifies the **number of bins** for continuous features. Here, it divides the continuous features into 5 discrete intervals (or bins). Each bin represents a range of values for the continuous features.

```
# Pick an example from the test set for explanation
example <- test[5, 1:4, drop = FALSE]
actual_class <- test$class[5]

# Generate explanations using LIME
explanation <- lime::explain(example, explainer, n_labels = 1, n_features = 4,kernel_width = 0.5)
```

`lime::explain()`:

`lime::explain()` is a function from the `lime` package that generates explanations for a set of predictions. The function takes a sample of data and an explainer object (created using `lime()`), and it returns a local explanation of how the features contributed to the prediction.

`example`:

This is the new data point or set of data points for which you want to explain the model's predictions. It should have the same structure as the training data used in the explainer (e.g., it should have 4 features in this case, corresponding to `train[, 1:4]` from earlier).

`explainer`:

This is the explainer object created previously using the `lime()` function. The explainer knows how to explain the predictions of the given model using the training data structure.

`n_labels = 1`:

This specifies the number of labels to explain for each prediction.

`n_features = 4`:

This specifies the number of features to include in the explanation. Here, the top 4 features (out of the available features) that contribute most to the prediction will be selected and explained.

`kernel_width = 0.5`:

- The `kernel_width` parameter controls the sensitivity of the LIME algorithm when fitting the local surrogate model (the simpler model used to approximate the original model in a local neighborhood).
- A smaller `kernel_width` (like 0.5) means that the explainer focuses more on data points that are closer to the instance being explained, giving more weight to nearby points when fitting the local surrogate model. This makes the explanation more local and sensitive to small changes near the data point being explained.

```
# Visualize the explanation
plot_features(explanation)
```

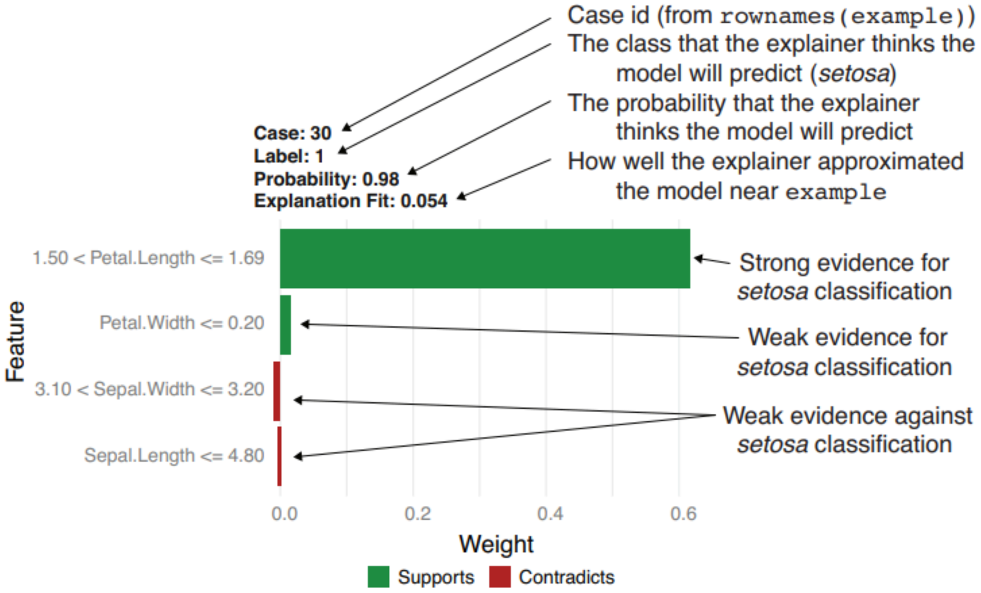


Figure 6.22 Visualize the explanation of the model's prediction.

Limitations of LIME:

1. **Local Approximation May Not Be Accurate:** LIME uses a simple linear model to approximate the decision boundary of a complex model around a particular sample. While this works well locally, it may not fully capture the complexity of the global decision boundary, especially for highly non-linear models.

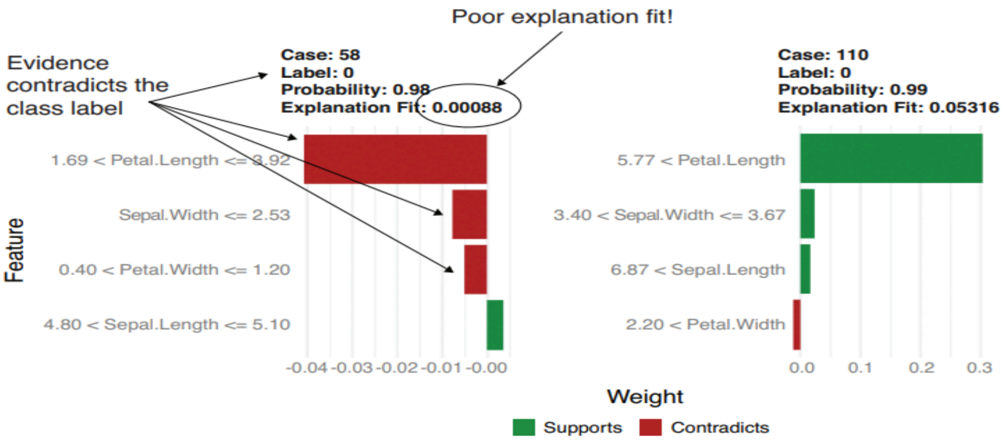


Figure 6.24 Explanations of the two iris examples

2. **Randomness in Perturbation:** The explanations generated by LIME may vary because the process of generating synthetic data points is randomized. Different runs of LIME might yield slightly different explanations for the same sample.

LIME for text classification:

1. Installing and Loading Packages

```
install.packages("zeallot")
library(zeallot)
```

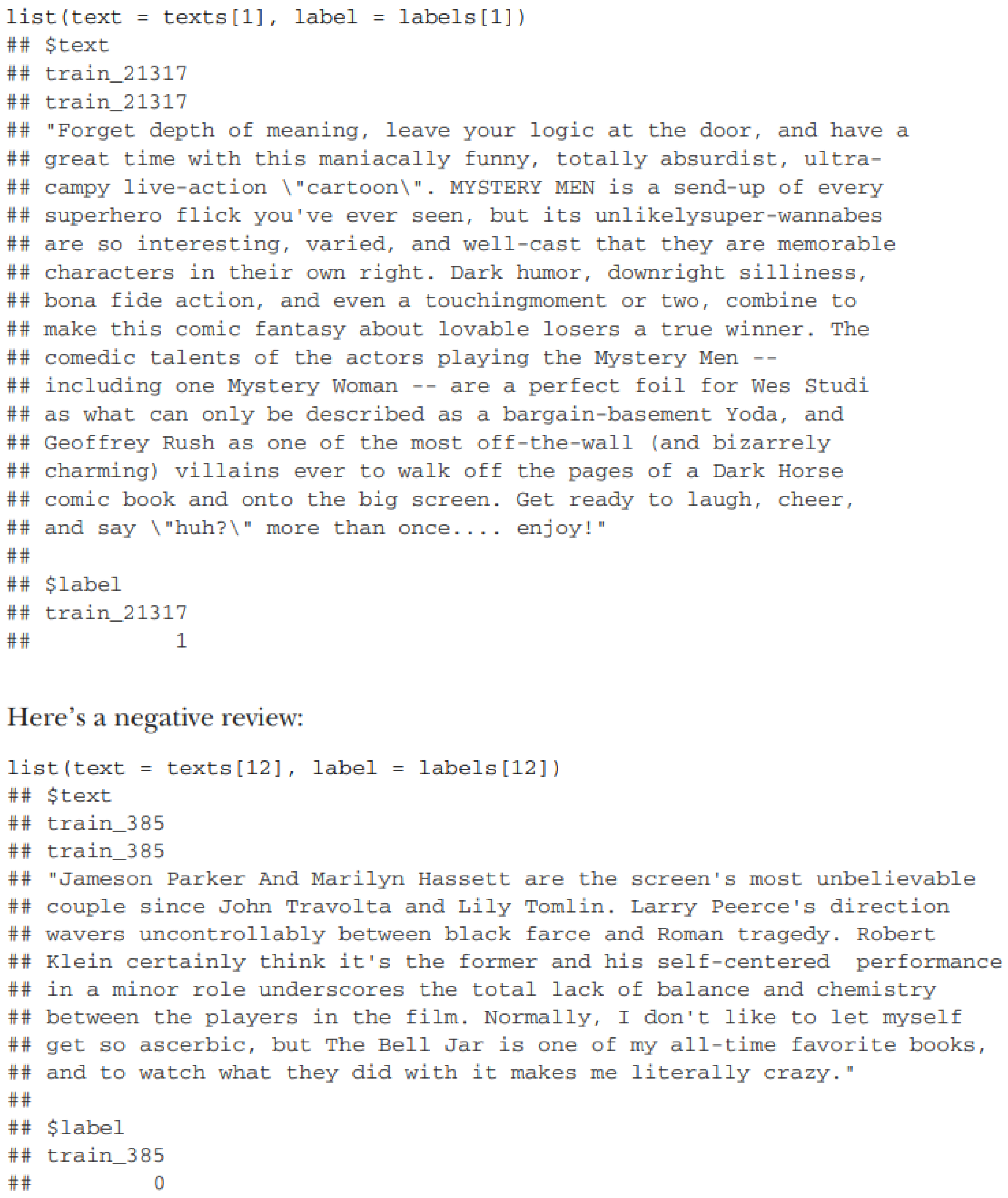
- **`install.packages("zeallot")`: Installs the `zeallot` package, which provides a convenient way to unpack objects in R using the `%<-%` operator.**
- **`library(zeallot)`: Loads the `zeallot` package into the R session.**

2. Loading Data

```
file_path <- file.choose()
c(texts, labels) %<-% readRDS(file_path)
```

3. Displaying Sample Data

```
list(text = texts[1], label = labels[1])
list(text = texts[12], label = labels[12])
```



```
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
```

4. Loading Additional Packages

```
library(wrapr)
library(xgboost)
library(text2vec)
```

- **`library(wrapr)`: Loads the `wrapr` package, which provides utilities for more readable and concise code.**
- **`library(xgboost)`: Loads the `xgboost` package, used for building gradient boosting models.**
- **`library(text2vec)`: Loads the `text2vec` package, which is used for text vectorization and modeling.**

5. Creating a Pruned Vocabulary

```
create_pruned_vocabulary <- function(texts) {
  it_train <- itoken(texts,
                     preprocessor = tolower,
                     tokenizer = word_tokenizer,
                     ids = names(texts),
                     progressbar = FALSE)

  stop_words <- qc(the, a, an, this, that, those, i, you)
  vocab <- create_vocabulary(it_train, stopwords = stop_words)

  pruned_vocab <- prune_vocabulary(
    vocab,
    doc_proportion_max = 0.5,
    doc_proportion_min = 0.001,
    vocab_term_max = 10000
  )

  pruned_vocab
}
```

- `itoken(texts, ...)`: **Creates an iterator for tokenizing the text data. Converts text to lowercase and uses word tokens.**
- `stop_words`: **Defines a small list of stop words to exclude from the vocabulary.**
- `create_vocabulary(it_train, stopwords = stop_words)`: **Creates a vocabulary from the tokenized texts while excluding stop words.**
- `prune_vocabulary(...)`: **Prunes the vocabulary by removing very common and very rare terms and limits the vocabulary size to 10,000 words.**

6. Creating a Document-Term Matrix

```
make_matrix <- function(texts, vocab) {
  iter <- itoken(texts,
                 preprocessor = tolower,
                 tokenizer = word_tokenizer,
                 ids = names(texts),
                 progressbar = FALSE)
  create_dtm(iter, vocab_vectorizer(vocab))
}
```

- `create_dtm(iter, vocab_vectorizer(vocab))`: **Converts the tokenized texts into a Document-Term Matrix (DTM) using the pruned vocabulary.**

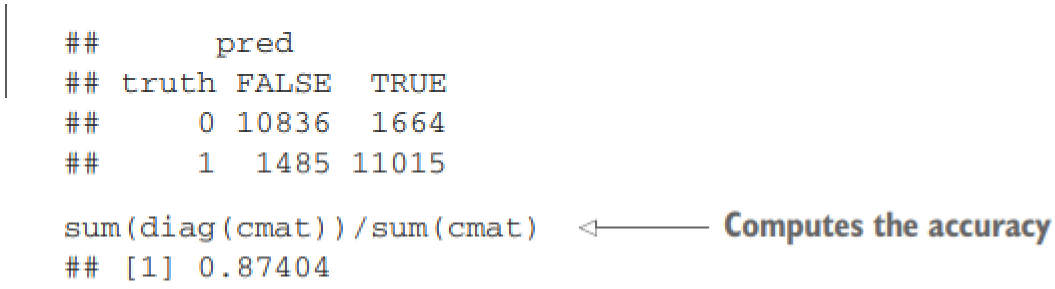
7. Fitting the Model

```
fit_imdb_model <- function(dtm_train, labels) {
  NROUNDS <- 371

  model <- xgboost(data=dtm_train, label=labels,
                   params=list(
                     objective="binary:logistic"
                   ),
                   nrounds=NROUNDS,
                   verbose=FALSE)

  model
}
```

- `xgboost(...)`: **Trains an XGBoost model on the document-term matrix with binary logistic regression.**

8. Loading Test Data and Making Predictions

```
file_path1 <- file.choose()
c(test_txt, test_labels) %<-% readRDS(file_path1)
dtm_test <- make_matrix(test_txt, vocab)
predicted <- predict(model, newdata=dtm_test)
teframe <- data.frame(true_label = test_labels,
                      pred = predicted)
(cmat <- table(truth = teframe$true_label, pred = teframe$pred > 0.5)
sum(diag(cmat))/sum(cmat)
```



```
##       pred
## truth FALSE  TRUE
##     0 10836  1664
##     1  1485 11015

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

9. Explaining Predictions with LIME

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

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

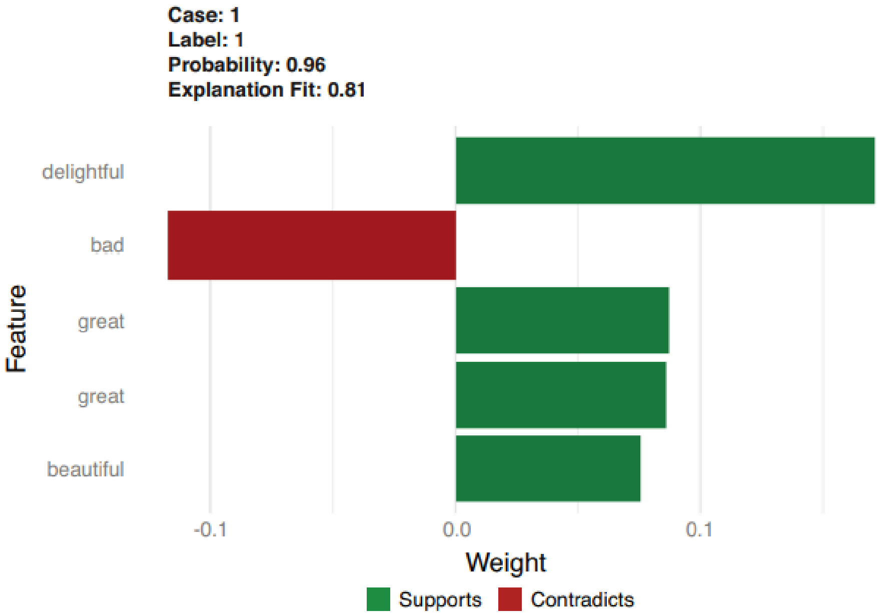


Figure 6.28 Explanation of the prediction on the sample review