```
#https://raw.githubusercontent.com/WinVector/PDSwR2/main/UCICar/car.data.csv
#https://raw.githubusercontent.com/charlesmutai/kdjha/main/custdata.csv
#https://raw.githubusercontent.com/charlesmutai/statlog/main/creditdata.csv
#https://raw.githubusercontent.com/WinVector/PDSwR2/main/Statlog/mapping.R
# https://raw.githubusercontent.com/WinVector/PDSwR2/main/Statlog/german.data
```

```
customer_data<-read.csv("https://raw.githubusercontent.com/charlesmutai/kdjha/main/custdata.csv")
str(customer_data)
```

```
'data.frame':   73262 obs. of  13 variables:
 $ X             : int  7 8 9 10 11 15 17 19 20 21 ...
 $ custid        : chr  "000006646_03" "000007827_01" "000008359_04" "000008529_01" ...
 $ sex           : chr  "Male" "Female" "Female" "Female" ...
 $ is_employed   : logi  TRUE NA TRUE NA TRUE NA ...
 $ income        : num  22000 23200 21000 37770 39000 ...
 $ marital_status: chr  "Never married" "Divorced/Separated" "Never married" "Widowed" ...
 $ health_ins    : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
 $ housing_type  : chr  "Homeowner free and clear" "Rented" "Homeowner with mortgage/loan" "Homeowner free and clear" ...
 $ recent_move   : logi  FALSE TRUE FALSE FALSE FALSE FALSE ...
 $ num_vehicles  : int  0 0 2 1 2 2 2 2 5 3 ...
 $ age           : int  24 82 31 93 67 76 26 73 27 54 ...
 $ state_of_res  : chr  "Alabama" "Alabama" "Alabama" "Alabama" ...
 $ gas_usage     : int  210 3 40 120 3 200 3 50 3 20 ...
```

```
install.packages("dplyr")
```

```
Installing package into ‘/usr/local/lib/R/site-library’
(as ‘lib’ is unspecified)
```

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

```
Attaching package: ‘dplyr’


The following objects are masked from ‘package:stats’:

    filter, lag


The following objects are masked from ‘package:base’:

    intersect, setdiff, setequal, union
```

```
library(dplyr)
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))
#glimpse(customer_data)
```

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

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

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

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

```
install.packages("vtreat")
```

```
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)

also installing the dependency 'wrapr'
```

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

```
Loading required package: wrapr


Attaching package: 'wrapr'


The following object is masked from 'package:dplyr':

    coalesce
```

```
training_prepared<-prepare(treatment_plan, customer_data)
```

```
glimpse(training_prepared)
```

```
Rows: 73,262
Columns: 25
$ custid                    <chr> "000006646_03", "000007827_01", "000008359_…
$ health_ins                <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, …
$ X                         <dbl> 7, 8, 9, 10, 11, 15, 17, 19, 20, 21, 23, 24…
$ sex                       <chr> "Male", "Female", "Female", "Female", "Male…
$ is_employed               <dbl> 1.0000000, 0.9504928, 1.0000000, 0.9504928,…
$ is_employed_isBAD         <dbl> 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1…
$ income                    <dbl> 22000, 23200, 21000, 37770, 39000, 11100, 2…
$ income_isBAD              <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
$ marital_status            <chr> "Never married", "Divorced/Separated", "Nev…
$ housing_type              <chr> "Homeowner free and clear", "Rented", "Home…
$ recent_move               <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
$ recent_move_isBAD         <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
$ num_vehicles              <dbl> 0, 0, 2, 1, 2, 2, 2, 2, 5, 3, 2, 2, 5, 1, 1…
$ num_vehicles_isBAD        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
$ age                       <dbl> 24, 82, 31, 93, 67, 76, 26, 73, 27, 54, 61,…
$ age_isBAD                 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
$ state_of_res              <chr> "Alabama", "Alabama", "Alabama", "Alabama",…
$ gas_usage                 <dbl> 210.00000, 76.00745, 40.00000, 120.00000, 7…
$ gas_usage_isBAD           <dbl> 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0…
$ gas_with_rent             <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
$ gas_with_rent_isBAD       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
$ gas_with_electricity      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0…
$ gas_with_electricity_isBAD <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
$ no_gas_bill               <dbl> 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0…
$ no_gas_bill_isBAD         <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
```

```
glimpse(customer_data)
```

```
Rows: 73,262
Columns: 16
$ X                    <int> 7, 8, 9, 10, 11, 15, 17, 19, 20, 21, 23, 24, 26, …
$ custid               <chr> "000006646_03", "000007827_01", "000008359_04", "…
$ sex                  <chr> "Male", "Female", "Female", "Female", "Male", "Ma…
$ is_employed          <lgl> TRUE, NA, TRUE, NA, TRUE, NA, TRUE, NA, TRUE, TRU…
$ income               <dbl> 22000, 23200, 21000, 37770, 39000, 11100, 25800, …
$ marital_status       <chr> "Never married", "Divorced/Separated", "Never mar…
$ health_ins           <lgl> TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, TRUE, …
$ housing_type         <chr> "Homeowner free and clear", "Rented", "Homeowner …
$ recent_move          <lgl> FALSE, TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, F…
$ num_vehicles         <int> 0, 0, 2, 1, 2, 2, 2, 2, 5, 3, 2, 2, 5, 1, 1, 2, 1…
$ age                  <int> 24, 82, 31, 93, 67, 76, 26, 73, 27, 54, 61, 64, 5…
$ state_of_res         <chr> "Alabama", "Alabama", "Alabama", "Alabama", "Alab…
$ gas_usage            <int> 210, NA, 40, 120, NA, 200, NA, 50, NA, 20, NA, NA…
$ gas_with_rent        <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, …
$ gas_with_electricity <lgl> FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, …
$ no_gas_bill          <lgl> FALSE, TRUE, FALSE, FALSE, TRUE, FALSE, TRUE, FAL…
```

```
na_counts<-sapply(training_prepared,
FUN=function(col) sum(is.na(col)))
```

```
sum(na_counts)
```

```
0
```

In addition to fixing missing data, there are other ways that you can transform the data to address issues that you found during the exploration phase.

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. Therefore, a data transformation that makes it easier to match patterns in the training data to patterns in new data can be a benefit.

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

For this example, you have external information about the median income in each state, in a file called median_income_table. Use this information to normalize the incomes. The code uses a join operation to match the information from median_income to the existing customer data. We will discuss joining tables in the next topic, but for now, you should understand joining as copying data into a data frame from another data frame with matching rows.

```
library(dplyr)
median_income_table <-read.csv("https://raw.githubusercontent.com/charlesmutai/statlog/main/median_income_table.csv")
glimpse(median_income_table)
```

```
Rows: 51
Columns: 3
$ X             <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1…
$ state_of_res  <chr> "Alabama", "Alaska", "Arizona", "Arkansas", "California"…
$ median_income <int> 21100, 32050, 26000, 22900, 25000, 32000, 36000, 29400, …
```

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

```
library(dplyr)
training_prepared <- training_prepared %>%
left_join(., median_income_table, by="state_of_res") %>%
mutate(income_normalized = income/median_income_table$median_income)
head(training_prepared[, c("income", "median_income", "income_normalized")])
```


```
library(dplyr)
training_prepared <- training_prepared %>%
left_join(., median_income_table, by="state_of_res") %>%
mutate(income_normalized = income/median_income_table$median_income)
head(training_prepared[, c("income","median_income","income_normalized")])
```

```
Warning message:
“There was 1 warning in `mutate()`.
i In argument: `income_normalized = income/median_income_table$median_income`.
Caused by warning in `income / median_income_table$median_income`:
! longer object length is not a multiple of shorter object length”
```

Compares the values of income and income_normalized

```
[71] summary(training_prepared$income_normalized)
```

```
summary (training_prepared$income_normalized )

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 0.0000  0.3882  0.9647  1.5453  1.9231 50.2800 

5   30000          31100           1.5600000
```

Looking at the results above, you see that customers with an income higher than the median income of their state have an income_normalized value larger than 1, and customers with an income lower than the median income of their state have an income_normalized value less than 1. 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. The need for data transformation can also depend on which modeling method you plan to use. For linear and logistic regression, for example, you ideally want to make sure that the relationship between the input variables and the output variable is approximately linear, and that the output variable is constant variance (the variance of the output variable is independent of the input variables). You may need to transform some of your input variables to better meet these assumptions.

Let us look at some useful data transformations and when to use them:

- Normalization
- Centering and scaling
- Log transformations

Normalization

Normalization (or rescaling) is useful when absolute quantities are less meaningful than relative ones. You’ve already seen an example of normalizing income relative to another meaningful quantity (median income). In that case, the meaningful quantity was external (it came from outside information), but it can also be internal (derived from the data itself).

For example, you might be less interested in a customer’s absolute age than you are in how old or young they are relative to a “typical” customer. Let’s take the mean age of your customers to be the typical age. You can normalize by that, as shown in the following code

```
[72] summary(training_prepared$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 
```

```
[73] mean_age <- mean(training_prepared$age)
     age_normalized <- training_prepared$age/mean_age
     summary(age_normalized)
```

```
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.

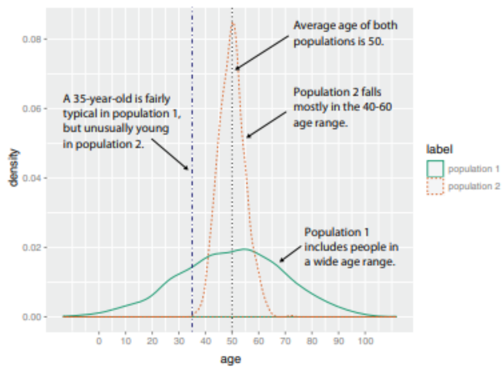


Figure 4.8 Is a 35-year-old young?

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.

```
[74] (mean_age <- mean(training_prepared$age)) #takes the mean
```

```
(mean_age <- mean(training_prepared$age))

    49.2164651226344
```

```
[75] (sd_age <- sd(training_prepared$age))  #Takes the standard deviation
```

```
(sd_age<-sd(training_prepared$age))

    18.0123968871458
```

```
[76] #The typical age range for this population is from about 31 to 67.
     print(mean_age + c(-sd_age, sd_age))
```

```
print(mean_age + c(-sd_age, sd_age))

    [1] 31.20407 67.22886
```

```
[77] #Uses the mean value as the origin (or reference point)
    #  and rescales the distance from the
    #mean by the standard deviation
    training_prepared$scaled_age <- (training_prepared$age -mean_age) / sd_age
```

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

       Min.  1st Qu.   Median     Mean  3rd Qu.     Max. 
   -1.56650 -0.84478 -0.06753  0.00000  0.70971  3.92971 
```

```
[78] #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()
```

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

A data.frame: 6 × 2

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

```
[79] #Customers outside the typical
     #age range have a scaled_age
     #with magnitude greater than 1.

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

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

A data.frame: 6 × 2

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

```
#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 twothirds 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.

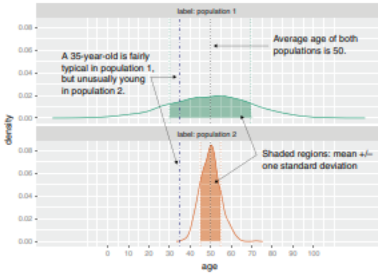


Figure 4.9 Faceted graph: is a 35-year-old young?

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.

The following codes demonstrates centering and scaling four numerical variables from the data with scale().

```
[81] # Centering and scaling multiple numeric variables
     dataf <- training_prepared[, c("age", "income", "num_vehicles", "gas_usage")]
     summary(dataf)
```

```
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
```

```
[92] dataf_scaled <- scale(dataf, center=TRUE, scale=TRUE)
     summary(dataf_scaled)
```

```
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
```

```
[83] #Gets the means and standard
     #deviations of the original data, which
     #are stored as attributes of dataf_scaled
     (means <- attr(dataf_scaled, 'scaled:center'))
```

```
(means<-attr(dataf_scaled,'scaled:center'))
```

age: 49.2164651226344 income: 41792.5106191185 num_vehicles: 2.0654999860222 gas_usage: 76.0074547390841

```
[84] #Centers the data by its mean and
     #scales it by its standard deviation
     (sds <- attr(dataf_scaled, 'scaled:scale'))
```

```
(sds <-attr(dataf_scaled, 'scaled:scale'))
```

age: 18.0123968871458 income: 58102.4814102411 num_vehicles:
1.15629371363957 gas_usage: 50.7177780073365

Because the scale() transformation puts all the numeric variables in compatible units, it's a recommended preprocessing step for some data analysis and machine learning techniques like principal component analysis and deep learning.

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. When you used the scale() function in code, you kept the values of the scaled:center and scaled:scale attributes as the variables means and sds, respectively. This is so that you can use these values to scale new data, as shown in codes before. This makes sure that the new scaled data is in the same units as the training data.

The same principle applies when cleaning missing values using the design_ missingness_treatment() function from the vtreat package, as you did in section before. The resulting treatment plan (called treatment_plan in code 4.1.3) keeps the information from the training data in order to clean missing values from new data, ascyou saw in code 4.5.

```
[85] # code 4.11: Treating new data before feeding it to a model
     newdata <- customer_data  #Simulates having a new customer dataset
     library(vtreat)
     # Cleans it using the treatment
     # plan from the original dataset
     newdata_treated <- prepare(treatment_plan, newdata)
     # Scales age, income, num_vehicles, and
     # gas_usage using the means and standard
     # deviations from the original dataset
     new_dataf <- newdata_treated[, c("age", "income",
     "num_vehicles", "gas_usage")]
     dataf_scaled <- scale(new_dataf, center=means, scale=sds)
```

```
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)
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
```

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. The implication here is that the characteristics of someone who earns three times the median income will be different from those of someone who earns less than the median income, and that these differences are the same independent of the actual dollar amount of the income.

4.2.3 Log transformations for skewed and wide distributions

Normalizing by mean and standard deviation, as you did in before, is most meaningful when the data distribution is roughly symmetric. Next, we'll look at a transformation that can make some distributions more symmetric.

Monetary amounts—incomes, customer value, account values, or purchase sizes— are some of the most commonly encountered sources of skewed distributions in data science applications.Monetary amounts are often lognormally distributed : the log of the data is normally distributed. This leads us to the idea that taking the log of monetary data can restore symmetry and scale to the data, by making it look "more normal." We demonstrate this in figure 4.11.

For the purposes of modeling, it's generally not too critical which logarithm you use, whether the natural logarithm, log base 10, or log base 2. In regression, for example, the choice of logarithm affects the magnitude of the coefficient that corresponds to the logged variable, but it doesn't affect the structure of the model. We like to use log base 10 for monetary amounts, because orders of ten seem natural for money: $100,1000$, $10,000, and so on. The transformed data is easy to read.

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.

As an example of an additive process, suppose you are studying weight loss. 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.

As an example of a multiplicative process, consider salary increases. If management gives everyone in the department a raise, it probably isn't giving everyone $5,000 extra. Instead, everyone gets a 2% raise: how much extra money ends up in your paycheck depends on your initial salary. In this situation, the natural unit of measurement is percentage, not absolute dollars. Other examples of multiplicative processes:

- A change to an online retail site increases conversion (purchases) for each item by 2% (not by exactly two purchases).
- A change to a restaurant menu increases patronage every night by 5% (not by exactly five customers every night).

When the process is multiplicative, log transforming the process data can make modeling easier. Unfortunately, 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). There are other transforms, such as arcsinh, that you can use to decrease data range if you have zero or negative values. We don't always use arcsinh, because we don't find the values of the transformed data to be meaningful. 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.

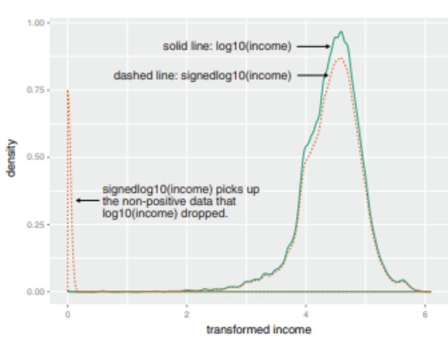


Figure 4.11 Signed log lets you visualize non-positive data on a logarithmic scale.

```
# Here's how to calculate signed log base 10 in R:
signedlog10 <- function(x) {
ifelse(abs(x) <= 1, 0, sign(x)*log10(abs(x)))
}
```

Sampling for modeling and validation

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

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.

- The training set is the data that you feed to the model-building algorithm so that the algorithm can fit the correct structure to best predict the outcome variable.
- The test set is the data that you feed into the resulting model, to verify that the model's predictions will be accurate on new data.

[87]
```
set.seed(25643) #Sets the random seed so this example is reproducible
customer_data$gp <- runif(nrow(customer_data))
#Here we generate a test set of about 10% of the data.
customer_test <- subset(customer_data, gp <= 0.1)
# Here we generate a training set using the remaining data.
customer_train <- subset(customer_data, gp > 0.1)
```

```
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)
summary(customer_train)
```

```
       X              custid              sex             is_employed    
 Min.   :     7   Length:65799       Length:65799       Mode :logical   
 1st Qu.: 24874   Class :character   Class :character   FALSE:2127      
 Median : 49805   Mode  :character   Mode  :character   TRUE :40518     
 Mean   : 49882                                         NA's :23154     
 3rd Qu.: 74790                                                         
 Max.   :100000                                                         
                                                                        
     income        marital_status      health_ins      housing_type      
 Min.   :      0   Length:65799       Mode :logical   Length:65799      
 1st Qu.:  10600   Class :character   FALSE:6575      Class :character  
 Median :  26030   Mode  :character   TRUE :59224     Mode  :character  
 Mean   :  41652                                                        
 3rd Qu.:  51100                                                        
 Max.   :1257000                                                        
 NA's   :43                                                             
 recent_move      num_vehicles        age         state_of_res      
 Mode :logical   Min.   :0.000   Min.   : 21.00   Length:65799      
 FALSE:56105     1st Qu.:1.000   1st Qu.: 34.00   Class :character  
 TRUE :8148      Median :2.000   Median : 48.00   Mode  :character  
 NA's :1546      Mean   :2.065   Mean   : 49.19                     
                 3rd Qu.:3.000   3rd Qu.: 62.00                     
                 Max.   :6.000   Max.   :120.00                     
                 NA's   :1545    NA's   :67                         
   gas_usage      gas_with_rent   gas_with_electricity no_gas_bill    
 Min.   :  4.00   Mode :logical   Mode :logical        Mode :logical  
 1st Qu.: 30.00   FALSE:62121     FALSE:58302          FALSE:41782    
 Median : 50.00   TRUE :2133      TRUE :5952           TRUE :22472    
 Mean   : 75.88   NA's :1545      NA's :1545           NA's :1545     
 3rd Qu.:100.00                                                       
 Max.   :570.00                                                       
 NA's   :32102                                                        
       gp        
 Min.   :0.1000  
 1st Qu.:0.3261  
 Median :0.5513  
 Mean   :0.5512  
 3rd Qu.:0.7756  
 Max.   :1.0000  
```

```
[88] dim(customer_test)
```

```
dim(customer_test)
```

7463 · 17

```
[89] dim(customer_train)
```

```
dim(customer_train)
```

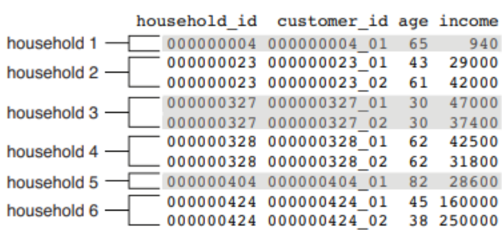
65799 · 17

Record grouping

One caveat is that the preceding trick works if every object of interest (every customer, in this case) corresponds to a unique row.

- But what if you're interested less in which customers don't have health insurance, and more in which households have uninsured members?
- If you're modeling a question at the household level rather than the customer level, then every member of a household should be in the same group (test or training). In other words, the random sampling also has to be at the household level.

Suppose your customers are marked both by a household ID and a customer ID. This is shown in figure 4.13. We want to split the households into a training set and a test set. Listing 4.13 shows one way to generate an appropriate sample group column.



	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

```
# Code 4.13 Ensuring test/train split doesn't split inside a household
household_data <- read.csv("https://raw.githubusercontent.com/charlesmutai/statlog/main/hhdata.csv")
```

```
[90] hh <- unique(household_data$household_id)#Gets the unique household IDs
     set.seed(243674)
     # Generates a unique sampling group ID per household, and
     # puts in a column named gp
     households <- data.frame(household_id = hh,
     gp = runif(length(hh)),
     stringsAsFactors=FALSE)
     # Joins the household IDs back into the original data
     household_data <- dplyr::left_join(household_data,
     households,
     by = "household_id")
     head(household_data)
```

```
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")
head(household_data)
```

```
head(household_data)
```

A data.frame: 6 × 7

	X	household_id	customer_id	age	income	gp	stringAsFactors
	<int>	<int>	<chr>	<int>	<dbl>	<dbl>	<lgl>
1	1	8385	000008385_01	74	45600	0.2063638	FALSE
2	2	12408	000012408_01	54	16300	0.4543296	FALSE
3	3	13288	000013288_01	59	622000	0.9931105	FALSE
4	4	13288	000013288_02	67	43000	0.9931105	FALSE
5	5	17554	000017554_01	47	98000	0.6279021	FALSE
6	6	17554	000017554_02	54	31200	0.6279021	FALSE

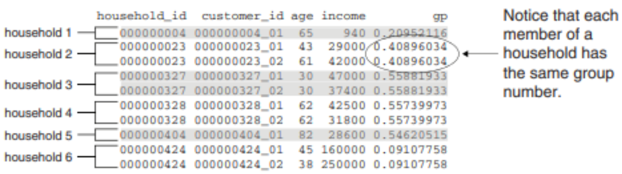


Figure 4.14 Sampling the dataset by household rather than customer

Everyone in a household has the same sampling group number. Now we can generate the test and training sets as before. This time, however, the threshold 0.1 doesn't represent 10% of the data rows, but 10% of the households, which may be more or less than 10% of the data, depending on the sizes of the households.

In this topic you have learned

- Different ways of handling missing values may be more suitable for a one purpose or another.
- You can use the vtreat package to manage missing values automatically.
- How to normalize or rescale data, and when normalization/rescaling are appropriate.
- How to log transform data, and when log transformations are appropriate.
- How to implement a reproducible sampling scheme for creating test/train splits of your data

✓ 0s completed at 11:26 PM