Tidyverse Packages in R

Howard Nguyen

2023-03-14

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.4.0      v purrr   0.3.5
## v tibble  3.1.8      v dplyr   1.0.10
## v tidyr   1.2.1      v stringr 1.4.1
## v readr   2.1.3      v forcats 0.5.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(tibble) # not necessary if tidyverse is already loaded
```

Use tidy table structures with tibbles

```
titanic_csv <- read.csv("titanic_train.csv")
titanic_tbl <- as_tibble(titanic_csv)
titanic_tbl
```

```
## # A tibble: 891 x 12
##    PassengerId Survived Pclass Name   Sex     Age SibSp Parch Ticket  Fare Cabin
##          <int>    <int>  <int> <chr>  <chr> <dbl> <int> <int> <chr>  <dbl> <chr>
##  1           1        0      3 Braun~ male     22     1     0 A/5 2~  7.25 ""
##  2           2        1      1 Cumin~ fema~    38     1     0 PC 17~ 71.3  "C85"
##  3           3        1      3 Heikk~ fema~    26     0     0 STON/~  7.92 ""
##  4           4        1      1 Futre~ fema~    35     1     0 113803 53.1  "C12~
##  5           5        0      3 Allen~ male     35     0     0 373450  8.05 ""
##  6           6        0      3 Moran~ male     NA     0     0 330877  8.46 ""
##  7           7        0      1 McCar~ male     54     0     0 17463  51.9  "E46"
##  8           8        0      3 Palss~ male      2     3     1 349909 21.1  ""
##  9           9        1      3 Johns~ fema~    27     0     2 347742 11.1  ""
## 10          10        1      2 Nasse~ fema~    14     1     0 237736 30.1  ""
## # ... with 881 more rows, and 1 more variable: Embarked <chr>
```

Use tibble for filter

```
titanic_tbl %>% filter(Sex == "female")
```

```
## # A tibble: 314 x 12
##    PassengerId Survived Pclass Name   Sex     Age SibSp Parch Ticket  Fare Cabin
##          <int>    <int>  <int> <chr>  <chr> <dbl> <int> <int> <chr>  <dbl> <chr>
##  1           2        1      1 "Cumi~ fema~    38     1     0 PC 17~ 71.3  "C85"
##  2           3        1      3 "Heik~ fema~    26     0     0 STON/~  7.92 ""
##  3           4        1      1 "Futr~ fema~    35     1     0 113803 53.1  "C12~
##  4           9        1      3 "John~ fema~    27     0     2 347742 11.1  ""
##  5          10        1      2 "Nass~ fema~    14     1     0 237736 30.1  ""
##  6          11        1      3 "Sand~ fema~     4     1     1 PP 95~ 16.7  "G6"
##  7          12        1      1 "Bonn~ fema~    58     0     0 113783 26.6  "C10~
##  8          15        0      3 "Vest~ fema~    14     0     0 350406  7.85 ""
##  9          16        1      2 "Hewl~ fema~    55     0     0 248706 16    ""
## 10          19        0      3 "Vand~ fema~    31     1     0 345763 18    ""
## # ... with 304 more rows, and 1 more variable: Embarked <chr>
```

Use tibble to select

```
titanic_train <- read_csv("titanic_train.csv")
```

```
## Rows: 891 Columns: 12
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (5): Name, Sex, Ticket, Cabin, Embarked
## dbl (7): PassengerId, Survived, Pclass, Age, SibSp, Parch, Fare
##
## i Use ‘spec()‘ to retrieve the full column specification for this data.
## i Specify the column types or set ‘show_col_types = FALSE‘ to quiet this message.
```

```
titanic_train %>% select(Name, Sex, Age)
```

```
## # A tibble: 891 x 3
##    Name                                                Sex      Age
##    <chr>                                               <chr>  <dbl>
##  1 Braund, Mr. Owen Harris                             male      22
##  2 Cumings, Mrs. John Bradley (Florence Briggs Thayer) female    38
##  3 Heikkinen, Miss. Laina                              female    26
##  4 Futrelle, Mrs. Jacques Heath (Lily May Peel)        female    35
##  5 Allen, Mr. William Henry                            male      35
##  6 Moran, Mr. James                                    male      NA
##  7 McCarthy, Mr. Timothy J                             male      54
##  8 Palsson, Master. Gosta Leonard                      male       2
##  9 Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)   female    27
## 10 Nasser, Mrs. Nicholas (Adele Achem)                 female    14
## # ... with 881 more rows
```

Use tibble with filter, select, and arrange like order in sql

```
titanic_women <- titanic_train %>%
  filter(Sex == "female") %>%
  select(Name, Sex, Age) %>%
  arrange(Name)
titanic_women
```

```
## # A tibble: 314 x 3
##    Name                                            Sex      Age
##    <chr>                                           <chr>  <dbl>
##  1 Abbott, Mrs. Stanton (Rosa Hunt)                female    35
##  2 Abelson, Mrs. Samuel (Hannah Wizosky)           female    28
##  3 Ahlin, Mrs. Johan (Johanna Persdotter Larsson)  female    40
##  4 Aks, Mrs. Sam (Leah Rosen)                      female    18
##  5 Allen, Miss. Elisabeth Walton                   female    29
##  6 Allison, Miss. Helen Loraine                    female     2
##  7 Allison, Mrs. Hudson J C (Bessie Waldo Daniels) female    25
##  8 Andersen-Jensen, Miss. Carla Christine Nielsine female    19
##  9 Andersson, Miss. Ebba Iris Alfrida              female     6
## 10 Andersson, Miss. Ellis Anna Maria               female     2
## # ... with 304 more rows
```

Use mutate() to add additional variable

we might create a binary elderly feature that indicates whether or not a passenger is at least 65 years old. This uses the dplyr package's if_else() function to assign a value of 1 if the passenger is elderly, and 0 otherwise:

```
titanic_train %>%
  mutate(elderly = if_else(Age >= 65, 1, 0))
```

```
## # A tibble: 891 x 13
##    PassengerId Survived Pclass Name   Sex     Age SibSp Parch Ticket  Fare Cabin
##          <dbl>    <dbl>  <dbl> <chr>  <chr> <dbl> <dbl> <dbl> <chr>  <dbl> <chr>
##  1           1        0      3 Braun~ male     22     1     0 A/5 2~  7.25 <NA>
##  2           2        1      1 Cumin~ fema~    38     1     0 PC 17~ 71.3  C85
##  3           3        1      3 Heikk~ fema~    26     0     0 STON/~  7.92 <NA>
##  4           4        1      1 Futre~ fema~    35     1     0 113803 53.1  C123
##  5           5        0      3 Allen~ male     35     0     0 373450  8.05 <NA>
##  6           6        0      3 Moran~ male     NA     0     0 330877  8.46 <NA>
##  7           7        0      1 McCar~ male     54     0     0 17463  51.9  E46
##  8           8        0      3 Palss~ male      2     3     1 349909 21.1  <NA>
##  9           9        1      3 Johns~ fema~    27     0     2 347742 11.1  <NA>
## 10          10        1      2 Nasse~ fema~    14     1     0 237736 30.1  <NA>
## # ... with 881 more rows, and 2 more variables: Embarked <chr>, elderly <dbl>
```

By separating the statements by commas, multiple columns can be created within a single mutate statement. This is demonstrated here to create an additional child feature that indicates whether the passenger is less than 18 years old:

```
titanic_train %>%
  mutate(
    elderly = if_else(Age >= 65, 1, 0),
    child = if_else(Age < 18, 1, 0)
  )
```

```
## # A tibble: 891 x 14
##    PassengerId Survived Pclass Name   Sex     Age SibSp Parch Ticket  Fare Cabin
##          <dbl>    <dbl>  <dbl> <chr>  <chr> <dbl> <dbl> <dbl> <chr>  <dbl> <chr>
##  1           1        0      3 Braun~ male     22     1     0 A/5 2~  7.25 <NA>
##  2           2        1      1 Cumin~ fema~    38     1     0 PC 17~ 71.3  C85
##  3           3        1      3 Heikk~ fema~    26     0     0 STON/~  7.92 <NA>
##  4           4        1      1 Futre~ fema~    35     1     0 113803 53.1  C123
##  5           5        0      3 Allen~ male     35     0     0 373450  8.05 <NA>
##  6           6        0      3 Moran~ male     NA     0     0 330877  8.46 <NA>
##  7           7        0      1 McCar~ male     54     0     0 17463  51.9  E46
##  8           8        0      3 Palss~ male      2     3     1 349909 21.1  <NA>
##  9           9        1      3 Johns~ fema~    27     0     2 347742 11.1  <NA>
## 10          10        1      2 Nasse~ fema~    14     1     0 237736 30.1  <NA>
## # ... with 881 more rows, and 3 more variables: Embarked <chr>, elderly <dbl>,
## #   child <dbl>
```

The remaining summarize() verb, allows us to create aggregated or summarized metrics by grouping rows in the tibble. For example, suppose we would like to compute the survival rate by age or gender. We'll begin with gender, as it is the easier of the two cases. We simply pipe the data into the group_by(Sex) function to create the male and female groups, then follow this with a summarize() statement to create a survival_rate feature that computes the average survival by group:

```
titanic_train %>%
  group_by(Sex) %>%
  summarize(survival_rate = mean(Survived))
```

```
## # A tibble: 2 x 2
##   Sex    survival_rate
##   <chr>          <dbl>
## 1 female         0.742
## 2 male           0.189
```

Computing survival by age, things are slightly more complicated due to the missing age values. We'll need to filter out these rows and use the group_by() function to compare children to non-children as follows:

```
titanic_train %>%
  filter(!is.na(Age)) %>%
  mutate(child = if_else(Age < 18, 1, 0)) %>%
  group_by(child) %>%
  summarize(survival_rate = mean(Survived))
```

```
## # A tibble: 2 x 2
##   child survival_rate
##   <dbl>         <dbl>
## 1     0         0.381
## 2     1         0.540
```

We'll filter() missing age values, use mutate() to create a new AgeGroup feature, and select() only the columns of interest for the decision tree model. The resulting dataset is piped to the rpart() decision tree algorithm, which illustrates the ability to pipe data to functions outside of the tidyverse:

```r
library(rpart)
m_titanic <- titanic_train %>%
    filter(!is.na(Age)) %>%
    mutate(AgeGroup = if_else(Age < 18, "Child", "Adult")) %>%
    select(Survived, Pclass, Sex, AgeGroup) %>%
    rpart(formula = Survived ~ ., data = .)
m_titanic
```

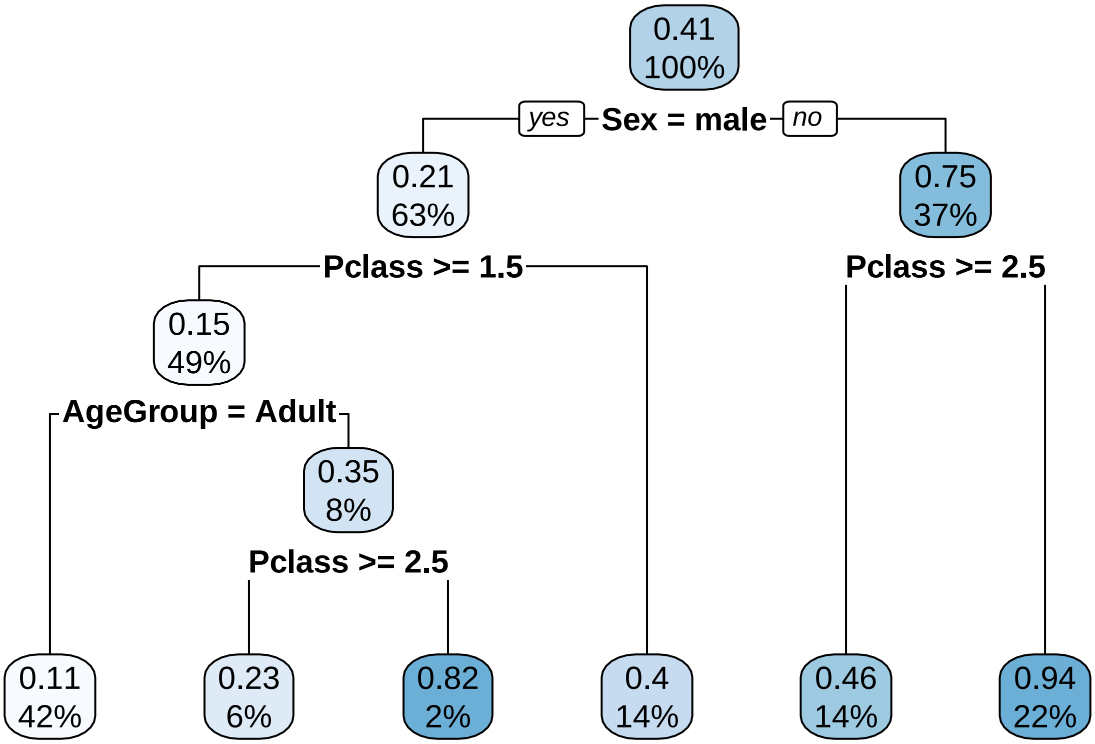
```
## n= 714
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 714 172.212900 0.4061625
##    2) Sex=male 453  73.907280 0.2052980
##      4) Pclass>=1.5 352  45.019890 0.1505682
##        8) AgeGroup=Adult 298  30.120810 0.1140940 *
##        9) AgeGroup=Child 54  12.314810 0.3518519
##         18) Pclass>=2.5 43   7.674419 0.2325581 *
##         19) Pclass< 2.5 11   1.636364 0.8181818 *
##      5) Pclass< 1.5 101  24.158420 0.3960396 *
##    3) Sex=female 261  48.306510 0.7547893
##      6) Pclass>=2.5 102  25.343140 0.4607843 *
##      7) Pclass< 2.5 159   8.490566 0.9433962 *
```

Note that the series of steps reads almost like plain-language pseudocode. It is also worth noting the arguments within the rpart() function call. The formula = Survived ~ . argument uses R's formula interface to model survival as a function of all predictors; the dot here represents the other features in the dataset not explicitly listed. The data = . argument uses the dot in a different way; here, the dot acts a placeholder to represent the data being fed to rpart() by the dplyr pipe. The dot can be used in this way to indicate the parameter to which the data should be piped. This is usually unnecessary for dplyr's built-in functions, because they look for the piped data as the first parameter by default, but functions outside the tidyverse may require the pipe to be targeted in this way.

For fun, we can visualize the resulting decision tree, which shows that women and children are more likely to survive than adults, men, and those in 3rd passenger class:

```r
library(rpart.plot)
rpart.plot(m_titanic)
```

```
## Warning: Cannot retrieve the data used to build the model (so cannot determine roundint and is.binary
## To silence this warning:
##     Call rpart.plot with roundint=FALSE,
##     or rebuild the rpart model with model=TRUE.
```



A decision tree predicting titanic survival, which was built using a series of dplyr pipes.

These are just a few small examples of how sequences of dplyr commands can make complex data manipulation tasks simpler. This is on top of the fact that, due to dplyr's more efficient code, the steps often execute more quickly than the equivalent commands in base R!

Transformaing text with stringr

The stringr package (https://stringr.tidyverse.org) adds functions for analyzing and transforming character strings. Base R, of course, can do this too, but the functions are inconsistent in how they work on vectors and are relatively slow; stringr implements these functions in a form more attuned to the tidyverse workflow. The free resource R for Data Science has a tutorial to introduce the package's complete set of capabilities at https://r4ds.had.co.nz/strings.html, but here we'll examine some of the aspects most relevant to feature engineering. If you'd like to follow along, be sure to read in the Titanic dataset and install and load the stringr package before proceeding.

Earlier in this section, the second hint for feature engineering was to "find insights hidden in text." The stringr package can assist with this effort by providing functions to slice and dice strings and detect patterns within text. All stringr functions begin with the prefix str_, and a few relevant examples are as follows: + str_detect() determines whether a search term is found in a string + str_sub() slices a string by position and returns a substring + str_extract() searches for a string and returns the matching pattern + str_replace() replaces characters in a string with something else

```
titanic_train <- titanic_train %>%
    mutate(CabinCode = str_sub(Cabin, start = 1, end = 1))
```

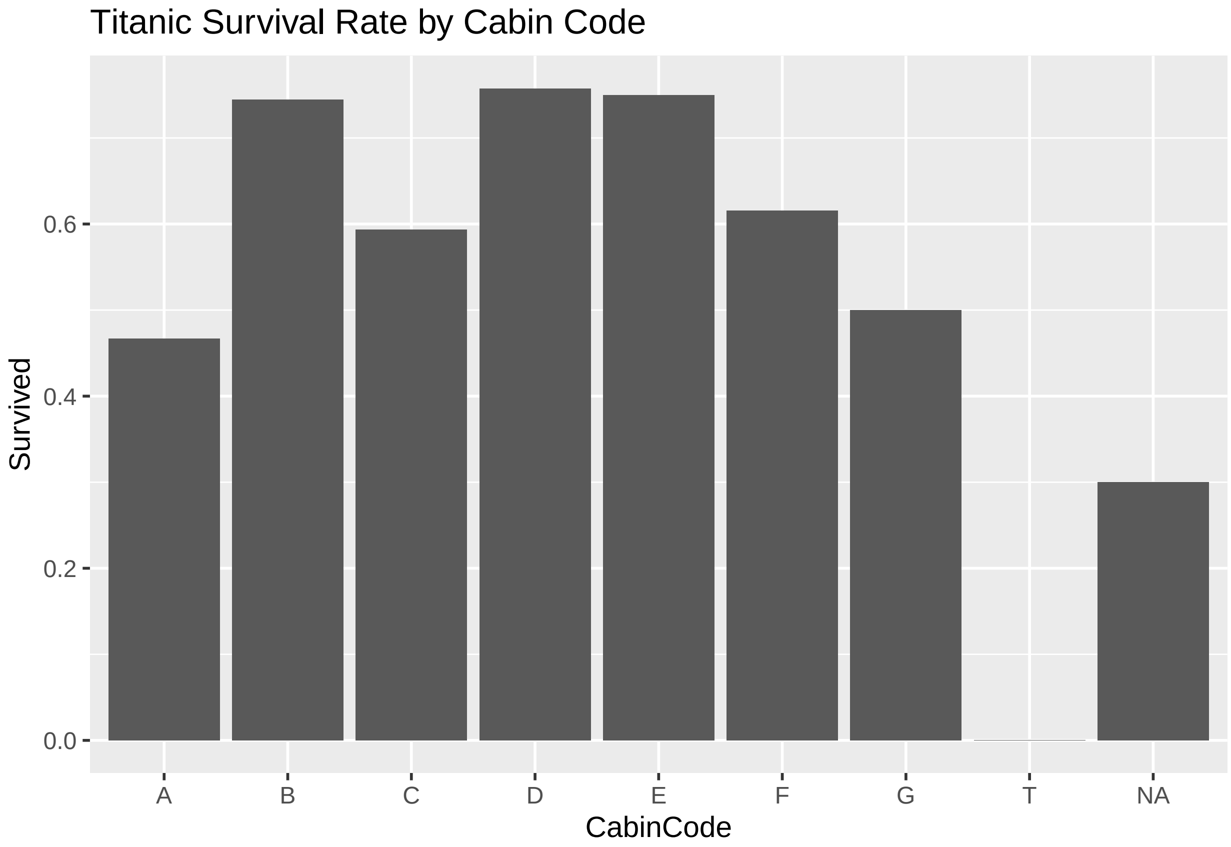
To confirm that the cabin code is meaningful, we use the table() function to see a clear relationship between it and the passenger class:

```
table(titanic_train$Pclass, titanic_train$CabinCode)
```

```
##
##       A  B  C  D  E  F  G  T
##   1 15 47 59 29 25  0  0  1
##   2  0  0  0  4  4  8  0  0
##   3  0  0  0  0  3  5  4  0
```

We can also plot the survival probability by cabin code by piping the file into a ggplot() function:

```
library(ggplot2)
titanic_train %>% ggplot() +
    geom_bar(aes(x = CabinCode, y = Survived),
               stat = "summary", fun = "mean") +
    ggtitle("Titanic Survival Rate by Cabin Code")
```



The cabin code feature seems related to survival, even within first-class cabins (A, B, C).

Without processing the Cabin feature in this way, a learning algorithm would be unable to use the text data as the cabins were tied uniquely to individuals. With this success in mind, let's examine another potential source of insight: the Name column. One might assume that this is unusable in a model, because the name is a unique identifier per row and training a model on this data will inevitably lead to overfitting. Although this is true, there is useful information hiding within the names. Looking at the first few rows reveals some potentially useful pieces hiding among the names:

```r
head(titanic_train$Name)
```

```
## [1] "Braund, Mr. Owen Harris"
## [2] "Cumings, Mrs. John Bradley (Florence Briggs Thayer)"
## [3] "Heikkinen, Miss. Laina"
## [4] "Futrelle, Mrs. Jacques Heath (Lily May Peel)"
## [5] "Allen, Mr. William Henry"
## [6] "Moran, Mr. James"
```

For one, the salutation (Mr., Mrs., and Miss.) might be helpful for prediction. The problem is that these titles are located at different positions within the name strings, so we cannot simply use the str_sub() function to extract them. The correct tool for this job is str_extract(), which is used to match and extract shorter patterns from longer strings. The trick with working with this function is knowing how to express the patterns rather than typing each potential salutation separately.

The shorthand used to express a text search pattern is called a regular expression, or regex for short. Knowing how to create regular expressions is an incredibly useful skill, as they are used for the advanced find-and-replace features in many text editors, in addition to being useful for feature engineering in R. We'll create a simple regex to extract the salutations from the name strings.

The first step in using regular expressions is to identify the common elements across all of the desired target strings. In the case of the Titanic names, it looks like each salutation is preceded by a comma followed by a blank space, then has a series of letters before ending with a period. This can be coded as the following regex string:

```r
", [A-z]+\\."
```

```
## [1] ", [A-z]+\\."
```

This seems to be nonsense but can be understood as a sequence that attempts to match a pattern character-by-character. The matching process begins with a comma and a blank space, as expected. Next, the square brackets tell the search function to look for any of the characters inside the brackets. For instance, [AB] would search for A or B and [ABC] would search for A, B, or C. In our usage, the dash is used to search for any characters within the range between A and z. Note that the capitalization is important; that is, [A-Z] is different from [A-z]. The former will search 26 characters comprising the uppercase alphabet while the latter will search 52 characters including uppercase and lowercase. Keep in mind that [A-z] only matches a single character.

To have the expression match more characters, we follow the brackets with a + symbol to tell the algorithm to continue consuming characters until it reaches something not inside the brackets. Then, it checks to see whether the remaining part of the regex matches. The remaining piece is the \. sequence, which is three characters representing the necessary period at the end of our search pattern. Because the dot is a special term in regular expressions that represents any arbitrary character, we must escape the dot by prefixing it with a slash. Unfortunately, the slash is a special character in R, so we must escape it as well by prefixing it with yet another slash.

Regular expressions can be tricky to learn but are well worth the effort. For a deep dive into understanding how they work, see https://www.regular-expressions.info. Alternatively, there are many text editors and web applications that demonstrate matching in real time. These can be hugely helpful to understand how to develop the regex search patterns and diagnose errors. One of the best such tools is found at: https://regexr.com.

We can put this expression to work on the Titanic name data by combining it in a mutate() function with str_extract() as follows:

```
titanic_train <- titanic_train %>%
    mutate(Title = str_extract(Name, ", [A-z]+\\."))
```

Looking at the first few examples, it looks like these need to be cleaned up a bit:

```
head(titanic_train$Title)
```

```
## [1] ", Mr."   ", Mrs."  ", Miss." ", Mrs."  ", Mr."   ", Mr."
```

Let's use the str_replace() function to replace the punctuation and blank space with an empty (null) string. We begin by constructing a regex to match the punctuation and empty space. One way to do this is to match the comma, blank space, and period using the "[, \.]" search string. Proving that there are often many different ways to accomplish the same task with regular expressions, it is possible to match everything except the alphabet using the "[^A-z]" expression, which includes the not operator denoted by the ^ character. Either one of these will work with str_replace() as shown here:

```
titanic_train <- titanic_train %>%
    mutate(Title = str_replace_all(Title, "[, \\.]", ""))
```

```
head(titanic_train$Title)
```

```
## [1] "Mr"   "Mrs"  "Miss" "Mrs"  "Mr"   "Mr"
```

Note that the str_replace_all() variant of the replace function was used due to the fact that multiple characters needed replacement; the basic str_replace() would have only replaced the first instance of a matching character. Many of stringr's functions have "all" variants for this use case. Let's see the result of our effort:

```
table(titanic_train$Title)
```

```
##
##     Capt      Col      Don       Dr Jonkheer     Lady    Major   Master
##        1        2        1        7        1        1        2       40
##     Miss     Mlle      Mme       Mr      Mrs       Ms      Rev      Sir
##      182        2        1      517      125        1        6        1
```

Given the small counts for some of these titles and salutations, it may make sense to group them together. To this end, we can use dplyr's recode() function to change the categories. We'll keep several of the high-count levels the same, while grouping the rest into variants of "Miss" and a catch-all "Other" bucket:

```
titanic_train <- titanic_train %>%
  mutate(TitleGroup = recode(Title,
    # the first few stay the same
    "Mr" = "Mr", "Mrs" = "Mrs", "Master" = "Master",
    "Miss" = "Miss",
    # combine other variants of "Miss"
   "Ms" = "Miss", "Mlle" = "Miss", "Mme" = "Miss",
    # anything else will be "Other"
    .default = "Other"
    )
  )
```

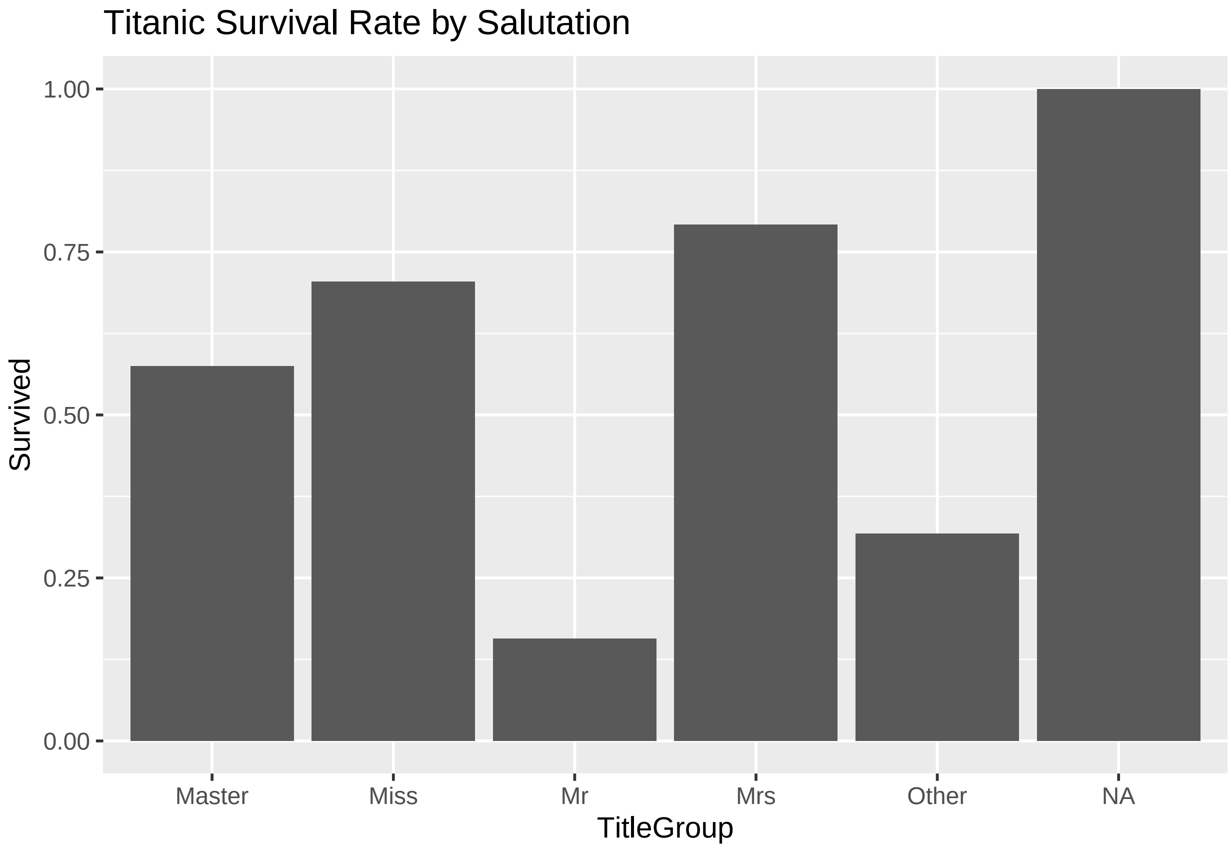
Checking our work, we see that our cleanup worked as planned:

```
table(titanic_train$TitleGroup)
```

```
##
## Master   Miss     Mr    Mrs  Other
##     40    186    517    125     22
```

We can also see that the title is meaningful by examining a plot of survival rates by title:

```
titanic_train %>% ggplot() +
    geom_bar(aes(x = TitleGroup, y = Survived),
             stat = "summary", fun = "mean") +
    ggtitle("Titanic Survival Rate by Salutation")
```



The constructed salutation captures the impact of both age and gender on survival likelihood.

The creation of CabinCode and TitleGroup features exemplifies the feature engineering technique of finding hidden information in text data. These features are likely to provide additional information beyond the given features in the Titanic dataset, which learning algorithms can use to improve performance. A bit of creativity combined with stringr and knowledge of regular expressions may provide the edge needed to surpass the competition.

Cleaning dates with lubridate

The lubridate package (https://lubridate.tidyverse.org) is an important tool for working with date and time data. It is needed only rarely, but when it is needed, it can save much grief. With dates and times, seemingly simple tasks can quickly turn into adventures due to unforeseen subtleties like leap years and time zones—just ask someone who has worked on birthday calculations, billing cycles, or similar date-sensitive tasks. As with the other tidyverse packages, the R for Data Science resource has an in-depth lubridate tutorial at https://r4ds.had.co.nz/dates-and-times.html, but we'll briefly cover three of its most important feature engineering strengths here:

- Ensuring date and time data is loaded into R correctly while accounting for regional differences in how dates and times are expressed
- Accurately calculating differences between dates and times while accounting for time zones and leap years
- Accounting for differences in how increments in time are understood in the real world, such as the fact that people become "one year older" on their birthday

Reading dates into R is challenge number one, because dates are presented as different forms of text strings. For example, the publication date of the first edition of Machine Learning with R can be expressed as:

- October 25, 2013 (a common longhand format in the United States)
- 10/25/13 (a common shorthand format in the United States)
- 25 October 2013 (a common longhand format in Europe)
- 25.10.13 (a common shorthand format in Europe)
- 2013-10-25 (the international standard)

Unfortunately, even lubridate is incapable of always guessing the correct format due to the fact that months, days, and years can all fall on the range from 1 to 12. Instead, assuming the expressed format is known, we simply match the input data to the correct date constructor—either mdy(), dmy(), or ymd(), depending on the order of the month (m), day (d), and year (y) components of the input data. Given the order of the date components, the functions will automatically parse longhand and shorthand variants, as well as handle leading zeros and two- or four-digit years. To demonstrate this, the dates expressed previously can be handled with the appropriate lubridate function as follows:

```
library(lubridate)
```

```
##
## Attaching package: 'lubridate'

## The following objects are masked from 'package:base':
##
##     date, intersect, setdiff, union
```

```
mdy(c("October 25, 2013", "10/25/2013"))
```

```
## [1] "2013-10-25" "2013-10-25"
```

```
dmy(c("25 October 2013", "25.10.13"))
```

```
## [1] "2013-10-25" "2013-10-25"
```

```
ymd("2013-10-25")
```

```
## [1] "2013-10-25"
```

Notice that in each case, the resulting Date object is exactly the same. Let's create a similar object for each of the three previous editions of this book:

```
MLwR_1stEd <- mdy("October 25, 2013")
MLwR_2ndEd <- mdy("July 31, 2015")
MLwR_3rdEd <- mdy("April 15, 2019")
```

We can do a simple math to compute the difference between two dates:

```
MLwR_2ndEd - MLwR_1stEd
```

```
## Time difference of 644 days
```

```
MLwR_3rdEd - MLwR_2ndEd
```

```
## Time difference of 1354 days
```

Notice that by default, the difference between two dates is returned as days. What if we hope to have an answer in years? Unfortunately, because these differences are a special lubridate difftime object, we cannot simply divide these numbers by 365 days to perform the obvious calculation. One option is to convert to a duration, which is one of the ways lubridate computes date differences, and in particular, tracks the passage of physical time—imagine it acting much like a stopwatch. The as.duration() function performs the needed conversion:

```
as.duration(MLwR_2ndEd - MLwR_1stEd)
```

```
## [1] "55641600s (~1.76 years)"
```

```
as.duration(MLwR_3rdEd - MLwR_2ndEd)
```

```
## [1] "116985600s (~3.71 years)"
```

We can see here that the gap between the 2nd and 3rd editions of Machine Learning with R was almost twice as long as the difference between the 1st and 2nd editions. We can also see that the duration seems to default to seconds in spite of the fact that it displays the total in years as well. To obtain only years, we can divide the duration by the duration of one year, which lubridate provides as a dyears() function:

```
dyears()
```

```
## [1] "31557600s (~1 years)"
```

```
as.duration(MLwR_2ndEd - MLwR_1stEd) / dyears()
```

```
## [1] 1.763176
```

```
as.duration(MLwR_3rdEd - MLwR_2ndEd) / dyears()
```

```
## [1] 3.70705
```

Generalizing this work, we can create a function to compute the calendar-based for a given date of birth as of today:

```
age <- function(birthdate) {
    birthdate %--% today() %/% years()
  }
```

To prove that it works, we'll check the ages of a few celebrities:

```
age(mdy("February 24, 1955")) # Jeff Bezos
```

```
## [1] 68
```

```
age(mdy("June 28, 1971")) # Elon Musk
```

```
## [1] 51
```

```
age(mdy("Oct 28, 1955")) # Bill Gates
```

```
## [1] 67
```

```
age(mdy("November 27, 1970"))
```

```
## [1] 52
```

If you are following along in R, be aware that your results may vary depending on when you run the code—we're all, unfortunately, still getting older by the day!