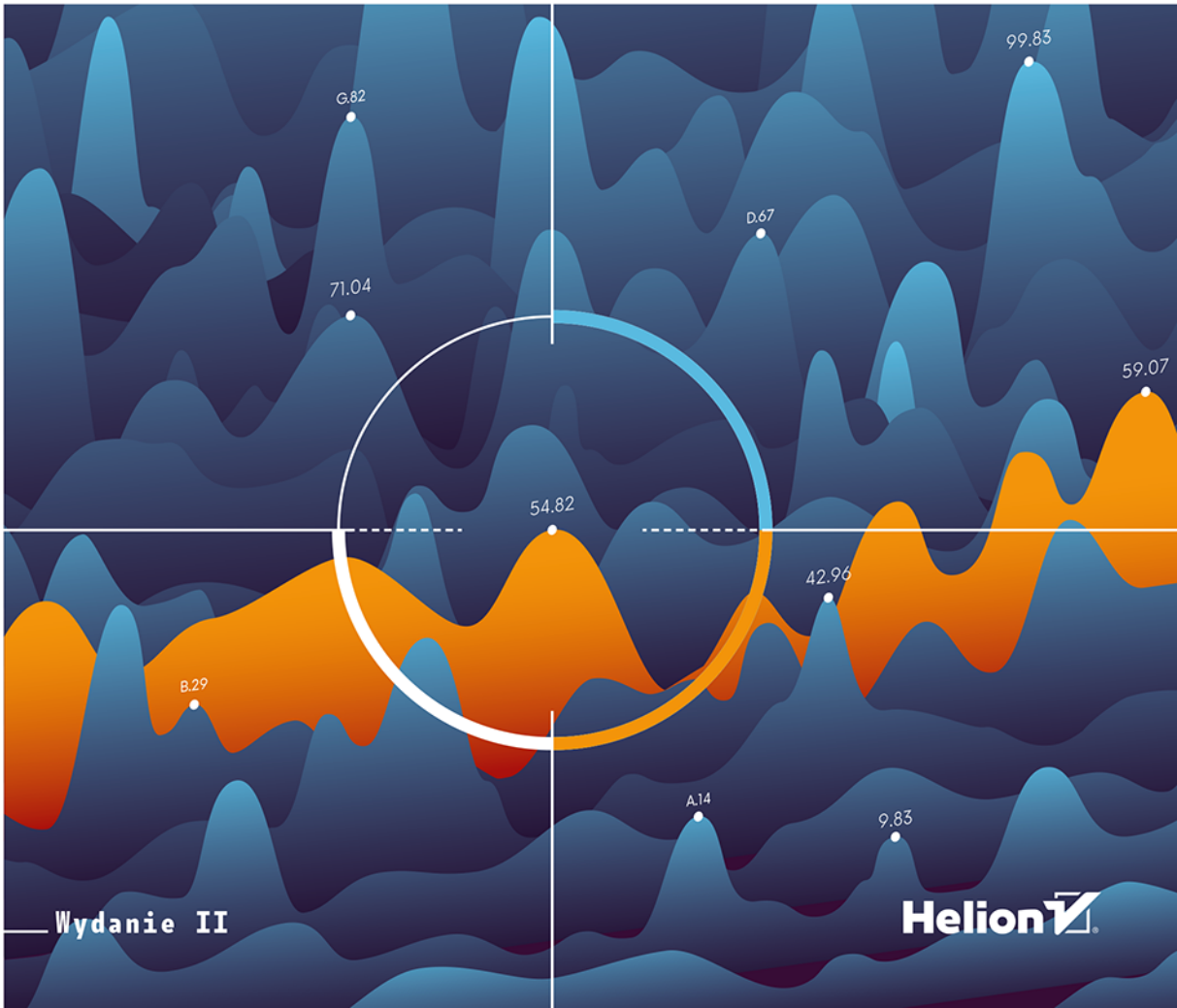
Nina Zumel, John Mount

JĘZYK R I ANALIZA DANYCH w praktyce



Wydanie II

Helion

Tytuł oryginału: Practical Data Science with R, 2nd Edition

Tłumaczenie: Krzysztof Sawka

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-6816-3

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/jrand2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/jrand2.zip

Printed in Poland.

Spis treści

2 Wprowadzenie do języka R i danych

Zawartość rozdziału:

- podstawowe informacje na temat pracy z R i danymi,
- omówienie ramki danych R,
- wczytywanie danych do R,
- przygotowywanie danych do dalszej analizy.

W tym rozdziale uzyskasz podstawowe informacje na temat języka R i nauczysz się importować dane do R z różnych źródeł. Przygotujemy Cię w ten sposób do pracy z przykładami omawianymi w dalszej części książki.

Zaprezentowany na rysunku 2.1 diagram przedstawia model mentalny, w którym zaakcentowaliśmy zadanie niniejszego rozdziału: rozpoczęcie pracy z R i importowanie danych. Cały diagram stanowi połączenie procesu analizy danych omówionego w rozdziale 1. z rebusem, którego odpowiedź stanowi tytuł książki. W każdym rozdziale będziemy zaznaczać poszczególne elementy diagramu, na które będziemy kładli szczególny nacisk. Na przykład w tym rozdziale skoncentrujemy się na początkowych etapach zdobywania danych i zarządzania nimi, a także poruszymy kwestie praktyczności, danych i R (ale jeszcze nie zagadnienia naukowe).

Rysunek 2.1. Model mentalny dla rozdziału 2.

Wiele projektów analizy danych rozpoczyna się w momencie, gdy ktoś przekazuje analitykowi jakiś zbiór danych i każe mu je zinterpretować[1]. Być może najpierw stwierdzisz, że warto skorzystać z doraźnych narzędzi i arkuszy kalkulacyjnych, przekonasz się jednak całkiem szybko, że więcej czasu poświęcasz na dostosowywanie tych narzędzi niż na właściwą analizę danych. Na szczęście istnieje lepszy sposób: język R. Po przeczytaniu niniejszego rozdziału będziesz w stanie całkiem pewnie używać R do wydobywania, przekształcania i wczytywania danych do analizy.

> *Korzystanie z języka R bez danych przypomina pójście do teatru po to, aby oglądać podnoszenie i opuszczanie kurtyny.*
>
> — Zaadaptowane z książki Bena Katchora
> *Julius Knipl, Real Estate Photographer: Stories*

2.1. Początki z R

R jest oprogramowaniem o otwartym kodzie źródłowym, dostępnym w Uniksie, Linuksie, a także systemach macOS i Windows. W tej książce będziemy koncentrować się na pracy jako analityk danych. Żeby jednak Czytelnik mógł realizować omawiane przykłady, musi być zaznajomiony z programowaniem w R. Jeżeli potrzebujesz jakichś wstępnych informacji, zalecamy zajrzenie do bezpłatnych instrukcji w serwisie CRAN

[1] Zakładamy, że jesteś zainteresowany pracą jako analityk, statystyk lub badacz danych, dlatego w dalszej części książki będziemy naprzemiennie korzystać z tych terminów.

(jest to główne repozytorium pakietów R: *https://cran.r-project.org/manuals.html*) i innych materiałów dostępnych w internecie. Warto także zajrzeć do następujących książek poświęconych R:

- *R in Action, Second Edition*, Robert Kabacoff, Manning, 2015,
- *Beyond Spreadsheets with R*, Jonathan Carroll, Manning, 2018,
- *The Art of R Programming*, Norman Matloff, No Starch Press, 2011,
- *R dla każdego. Zaawansowane analizy i grafika statystyczna*, Jared P. Lander, Addison-Wesley, 2017 (wydanie polskie: APN Promise, 2018).

Każda z tych książek przekazuje wiedzę w inny sposób, a niektóre z nich zawierają dodatkowe informacje na temat statystyki, uczenia maszynowego i inżynierii danych. Wystarczy poświęcić chwilę, aby sprawdzić, profil której książki odpowiada Ci najbardziej. W niniejszej książce skoncentrujemy się na realizowaniu ważnych przykładów z zakresu analizy danych, dzięki którym poznasz etapy wymagane do przezwyciężenia najczęściej spotykanych problemów w rzeczywistych projektach.

Naszym zdaniem analiza danych jest powtarzalna: to samo zadanie zrealizowane na tych samych danych powinno dawać podobny wynik (różnice mogą wynikać z precyzji numerycznej, synchronizacji, zrównoleglania obliczeń, a także liczb pseudolosowych). W istocie powinniśmy dążyć do takiej powtarzalności. Z tego właśnie powodu zajmujemy się programowaniem w książce poświęconej analizie danych. Programowanie stanowi łatwy sposób określania wielokrotnie używanych sekwencji operacji. Mając to na uwadze, powinniśmy zawsze traktować odświeżanie danych (uzyskiwanie nowszych, poprawionych lub większej ilości danych) jako coś dobrego, ponieważ wielokrotne przeprowadzanie analizy powinno być bardzo łatwe. Analiza, której wiele etapów należy realizować własnoręcznie, nigdy nie będzie łatwa do powtarzania.

2.1.1. Instalowanie R, narzędzi i przykładów

W celu zainstalowania środowiska R, pakietów, narzędzi i przykładów omawianych w książce zalecamy zapoznanie się z instrukcjami zawartymi w podrozdziale A.1 w dodatku A.

NIE BÓJ SIĘ SZUKAĆ POMOCY. R zawiera bardzo przydatny system pomocy. Aby uzyskać informacje na temat jakiegoś polecenia R, wystarczy użyć komendy `help()`. Na przykład, jeżeli chcesz się dowiedzieć, jak można zmieniać katalogi, wpisz polecenie `help(setwd)`. Musisz znać nazwę funkcji, na temat której chcesz uzyskać informacje, dlatego usilnie zalecamy sporządzanie notatek. Niektórych prostych funkcji nie będziemy tłumaczyć i pozostawimy Czytelnikowi możliwość wywołania polecenia `help()`.

2.1.2. Programowanie w R

W tym podrozdziale omówimy pokrótce niektóre konwencje pisania kodu, semantykę i problemy ze stylami w R. Szczegóły znajdziesz w dokumentacji danego pakietu, w systemie pomocy R (`help()`), a także w różnych wariantach omówionych w tym rozdziale przykładów. Skoncentrujemy się na aspektach odróżniających R od innych języków programowania, a także na konwencjach wykorzystywanych w dalszej części książki. Dzięki temu powinieneś łatwiej „wczuć się" w R.

Istnieje wiele popularnych stylów pisania kodu w R. **Styl pisania kodu** (ang. *coding style*) stanowi próbę zwiększenia spójności, przejrzystości i czytelności listingów. W tej książce będziemy stosować wariant stylu, który uważamy za bardzo skuteczny do nauki oraz do utrzymywania kodu. Nasz styl, oczywiście, jest zaledwie jednym z wielu i w żadnym wypadku nie należy go traktować jako obowiązkowego. Kwestia stylów pisania kodu została dobrze opisana w następujących materiałach:

- *Google's R Style Guide* (*https://google.github.io/styleguide/Rguide.html*),
- przewodnik po stylach z książki Hadleya Wickhama *Advanced R* (*http://adv-r.had.co.nz/Style.html*).

Postaramy się zminimalizować różnice pomiędzy notacjami i będziemy zwracać uwagę wszędzie tam, gdzie będą się pojawiać jakieś odstępstwa. Polecamy również dział *R tips*, dostępny na blogu jednego z autorów niniejszej książki[2].

R jest bogatym i rozbudowanym językiem, dzięki któremu wiele zadań można nieraz wykonać na różne sposoby. Oznacza to dość stromą początkową krzywą uczenia, gdyż dopóki nie przywykniesz do notacji, działanie programów może być trudne do zrozumienia. Jednak czas spędzony na zrozumieniu podstawowej notacji jest dobrze spożytkowany, ponieważ dzięki temu będzie Ci o wiele łatwiej pracować z przykładami omówionymi w tej książce. Rozumiemy, że sama w sobie gramatyka języka R nie stanowi obiektu zainteresowania użytkownika chcącego poznać metody i praktyki analizy danych (pamiętaj, że należysz do naszej docelowej grupy odbiorców!), ale dzięki temu małemu, początkowemu wysiłkowi unikniesz później wielu nieporozumień. W tym podrozdziale zajmiemy się notacją języka R i jej znaczeniem, przy czym skoncentrujemy się na elementach szczególnie użytecznych lub zaskakujących. Wszystkie omówione poniżej kwestie są podstawowe, ale część z nich zawiera wiele niuansów, które można poznać poprzez eksperymentowanie.

KONCENTRUJ SIĘ NA DZIAŁAJĄCYM KODZIE. Skupiaj uwagę na programach, skryptach i kodzie, które już działają, ale być może nie realizują wszystkich wymaganych zadań. Zamiast pisać rozbudowany, nieprzetestowany program lub skrypt zawierający wszystkie potrzebne etapy analizy, stwórz aplikację, która we właściwy sposób wykonuje określoną fazę, a następnie iteracyjnie ją usprawniaj, tak aby kod stopniowo realizował pozostałe etapy. Taki sposób aktualizowania kolejnych wersji kodu zazwyczaj pozwala uzyskać pożądane wyniki znacznie szybciej niż mozolne usuwanie błędów z dużego, niesprawnego systemu.

PRZYKŁADY I SYMBOL KOMENTARZA (#)

W naszych przykładach polecenia R będą prezentowane jako zwykły tekst, natomiast wyniki będziemy oznaczać symbolem komentarza R (#). W wielu listingach będziemy umieszczać rezultaty tuż po poleceniach; dzięki symbolom komentarza będą one łatwo rozpoznawane. W wyświetlanych wynikach często występują indeksy tablicy umieszczone w nawiasach kwadratowych, a wiersze nieraz są zawijane. Na przykład wyświetlanie liczb stałoprzecinkowych w przedziale od 1 do 25 wygląda następująco:

[2] Zobacz *http://www.win-vector.com/blog/tag/r-tips/*.

```
print(seq_len(25))
# [1]  1  2  3  4  5  6  7  8  9  10 11 12
# [13] 13 14 15 16 17 18 19 20 21 22 23 24
# [25] 25
```

Zwróć uwagę, że wartości zostały rozmieszczone w trzech wierszach, a na początku każdego z nich występuje zamknięty w nawiasie kwadratowym indeks pierwszej komórki wyświetlonej w tym wierszu. Czasami nie będziemy prezentować wyników; powinno to stanowić dodatkowy bodziec do wykonania ćwiczenia.

WYŚWIETLANIE

R zawiera dużą liczbę reguł włączających i wyłączających wyświetlanie niejawne/automatyczne. W niektórych pakietach, takich jak `ggplot2`, wyświetlanie uruchamia określone operacje. Wpisanie wartości zazwyczaj powoduje jej wyświetlenie. Pamiętaj, że w przypadku funkcji lub pętli `for` automatyczne wyświetlanie wyników jest wyłączone. Problem może stanowić wyświetlanie bardzo dużych obiektów, dlatego najlepiej unikać wyświetlania elementów o nieznanym rozmiarze. Często można wymusić wyświetlanie niejawne poprzez wprowadzenie dodatkowych nawiasów, np. `(x <- 5)`.

WEKTORY I LISTY

Wektory (tablice sekwencyjne wartości) są podstawowymi strukturami danych w R. W każdej pozycji listy mogą być przechowywane różne typy danych, natomiast wektory mogą zawierać wyłącznie ten sam typ prymitywny (atomowy) w poszczególnych polach. Zarówno wektory, jak i listy obsługą indeksowanie numeryczne, a także pary nazwa-klucz. Dostęp do elementów listy lub wektora uzyskujemy za pomocą operatorów zaprezentowanych poniżej.

INDEKSOWANIE WEKTORÓW. W przeciwieństwie do wielu innych języków programowania wektory i listy R są indeksowane od wartości 1, nie 0.

Tworzy przykładowy wektor. Funkcja `c()` jest w R operatorem konkatenacji: łączy krótsze wektory i listy w ich dłuższe odpowiedniki bez zagnieżdżania. Na przykład `c(1)` oznacza wyłącznie wartość 1, natomiast `c(1, c(2, 3))` jest równoznaczna operacji `c(1, 2, 3)`, dzięki której otrzymujemy wartości stałoprzecinkowe od 1 do 3 (ale przechowywane w formacie zmiennoprzecinkowym).

```
przykladowy_wektor <- c(10, 20, 30)
przykladowa_lista <- list(a = 10, b = 20, c = 30)
```

Tworzy przykładową listę.

```
przykladowy_wektor[1]
 ## [1] 10
przykladowa_lista[1]
## $a
## [1] 10
```

Demonstruje przeznaczenie pojedynczego nawiasu kwadratowego (zarówno w wektorach, jak i listach). Zwróć uwagę, że w przypadku listy zapis `[]` zwraca nową krótką listę, a nie element.

```
przykladowy_wektor[[2]]
 ## [1] 20
przykladowa_lista[[2]]
## [1] 20
```

Demonstruje przeznaczenie podwójnego nawiasu kwadratowego (zarówno w wektorach, jak i listach). Najczęściej konstrukcja ta wymusza zwrócenie pojedynczego elementu, chociaż w listach zagnieżdżonych złożonego typu taki pojedynczy element może być sam w sobie listą.

```
przykladowy_wektor[c(FALSE, TRUE, TRUE)]
 ## [1] 20 30
przykladowa_lista[c(FALSE, TRUE, TRUE)]
## $b
## [1] 20
##
## $c
## [1] 30
```

Wektory i listy mogą być indeksowane za pomocą wektorów przechowujących wartości logiczne, stałoprzecinkowe i (jeżeli wektor lub lista zawiera nazwy) znaki.

```
przykladowa_lista$b
 ## [1] 20

przykladowa_lista[["b"]]
## [1] 20
```

W przypadkach list określonych nazwą składnia `przykladowa_lista$b` **stanowi w istocie skróconą wersję składni** `przykladowa_lista[["b"]]` **(to samo dotyczy wektorów określonych nazwą).**

Nie wszystkie przykłady będziemy tak bogato opatrywać uwagami, ale równie przydatne okazuje się korzystanie z funkcji `help()` przy okazji natrafiania na nową funkcję lub polecenie. Usilnie zachęcamy również do testowania wariantów. W R „błędy" oznaczają jedynie informację, że R w bezpieczny sposób odmówił realizacji źle przygotowanej operacji (nie oznacza to „zawieszenia" aplikacji ani otrzymania nieprawidłowych wyników). Z tego powodu strach przed błędami nie powinien ograniczać eksperymentowania.

```
x <- 1:5
print(x)
# [1] 1 2 3 4 5
```

Definiuje wartość, którą jesteśmy zainteresowani, i przechowuje ją w zmiennej x.

```
x <- cumsumMISSPELLED(x)
# Error in cumsumMISSPELLED(x) : could not find function "cumsumMISSPELLED"
```

Stara się, bez powodzenia, przypisać nowy rezultat do x.

```
print(x)
# [1] 1 2 3 4 5
```

Zwróć uwagę, że otrzymujemy nie tylko użyteczny komunikat o błędzie, ale R zachowuje oryginalną wartość x.

```
x <- cumsum(x)
print(x)
# [1] 1 3 6 10 15
```

Ponownie próbuje przeprowadzić wcześniejszą operację, ale tym razem funkcja `cumsum()` **została zapisana poprawnie. Pełna nazwa tej funkcji to ang. cumulative sum, czyli suma wyników; funkcja ta przydaje się do szybkiego obliczania sumy bieżącej.**

Kolejną własnością R jest **wektoryzacja** większości operacji. Dane funkcja lub operator są wektoryzowane wtedy, gdy ich zastosowanie wobec jakiegoś wektora jest równoznaczne ich zastosowaniu wobec każdego elementu tego wektora niezależnie od pozostałych. Na przykład funkcja `nchar()` określa liczbę znaków tworzących dany łańcuch znaków. W R funkcji tej można użyć na pojedynczym łańcuchu znaków lub na wektorze składającym się z łańcuchów znaków.

LISTY I WEKTORY SĄ W R STRUKTURAMI MAPUJĄCYMI. Mogą one odwzorowywać łańcuchy znaków na dowolne obiekty. Główne operacje `list`, `[]`, `match ()` i `%in%` są **wektorowe**. Oznacza to, że użyte na wektorze wartości zwracają wektor wyników poprzez realizację jednego przeszukiwania na każdy wpis. Aby wydobywać pojedyncze elementy z listy, skorzystaj z podwójnego nawiasu `[[]]`.

```
nchar("łańcuch znaków")
# [1] 14
nchar(c("a", "aa", "aaa", "aaaa"))
# [1] 1 2 3 4
```

OPERACJE LOGICZNE. W R występują dwa rodzaje operatorów logicznych. Znajdziemy tu standardowe, skalarne operatory wrostkowe, które oczekują tylko jednej wartości i cechują się takim samym działaniem oraz nazewnictwem, jak analogiczne wektory w językach C czy Java: `&&`, a także `||`. R zawiera również wektorowe operatory wrostkowe, działające na wektorach przechowujących wartości logiczne: `&` i `|`. Zawsze korzystaj z wersji skalarnych (`&&` i `||`) w takich sytuacjach, jak korzystanie z instrukcji `if`, a z wersji wektorowych (`&` i `|`) podczas przetwarzania wektorów logicznych.

WARTOŚCI NULL I NA

W R wartość `NULL` symbolizuje po prostu pusty wektor, utworzony za pomocą operatora konkatenacji `c()` niezawierającego argumentów. Na przykład po wpisaniu operatora `c()` w konsoli R zostanie zwrócona wartość `NULL`. W przeciwieństwie do większości języków z rodziny C czy Javy tutaj `NULL` nie jest nieprawidłowym wskaźnikiem, lecz jedynie pustym wektorem. Operacja konkatenacji dająca w wyniku wartość `NULL` jest bezpieczna i dobrze zdefiniowana (zasadniczo jest to instrukcja pusta, czyli operacja, która niczego nie realizuje). Na przykład zapis `c(c(), 1, NULL)` jest całkowicie prawidłowy i zwraca wartość `1`.

Z kolei rozwinięcie skrótu `NA` to w języku angielskim *not available*, czyli „niedostępna"; wartość ta jest specyficzna dla R. Większość prostych typów danych może przyjmować wartość `NA`. Na przykład wektor `c("a", NA, "c")` zawiera trzy łańcuchy znaków i nie znamy wartości drugiego elementu. Wartość `NA` stanowi olbrzymie udogodnienie, ponieważ za jej pomocą możemy oznaczać brakujące lub niedostępne dane, co jest kluczowym aspektem przetwarzania danych. Wartość `NA` przypomina nieco działaniem wartość `NaN` występującą w arytmetyce zmiennoprzecinkowej[3], z tym że nie jesteśmy tu ograniczeni jedynie do zmiennoprzecinkowych typów danych. Ponadto wartość `NA` oznacza „niedostępna", a nie nieprawidłowa (na co z kolei wskazuje wartość `NaN`), dlatego cechują ją pewne wygodne reguły (na przykład uproszczenie wyrażenia `FALSE & NA` do postaci `FALSE`).

IDENTYFIKATORY

Identyfikatory, czyli nazwy symboliczne, stanowią mechanizm odwoływania się do zmiennych i funkcji w R. *Google's R Style Guide* sugeruje zapisywanie nazw symbolicznych w formacie zwanym CamelCase (początek każdego wyrazu zostaje oznaczony dużą literą, jak w samej nazwie CamelCase). Z kolei w zaleceniach z książki *Advanced R* autor proponuje stosowanie podkreślników rozdzielających poszczególne wyrazy (na przykład „godzina_pierwsza" zamiast „GodzinaPierwsza"). Wielu użytkowników do rozdzielania wyrazów wykorzystuje kropkę (np. „godzina.pierwsza"). W szczególności istotne, wbudowane struktury danych R, takie jak `data.frame` czy pakiety (np. `data.table`), stosują nomenklaturę kropkową.

---

[3] Ograniczanie **arytmetyki zmiennoprzecinkowej**, czyli sposobu **aproksymowania** liczb rzeczywistych przez komputery, od dawna stanowi źródło utrapienia i problemów podczas pracy z danymi numerycznymi. Aby docenić kwestie pracy z danymi numerycznymi, polecamy artykuł Davida Goldberga z 1991 r. pt. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, „Computing Surveys", opublikowany pod adresem *https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html*.

Zalecamy korzystanie z podkreślników, zauważyliśmy jednak, że często musimy dostosowywać się do konwencji stosowanej przez inne osoby. W miarę możliwości staraj się unikać notacji kropkowej, ponieważ ma ona inne zastosowania w językach obiektowych oraz bazach danych, dlatego może niepotrzebnie wprowadzać innych w błąd[4].

ŁAMANIE WIERSZA

Zasadniczo zalecane jest ograniczanie długości wiersza kodu do 80 znaków. R akceptuje instrukcje wielowierszowe, jeżeli koniec takiej instrukcji zostanie jednoznacznie określony. Na przykład, jeżeli chcesz podzielić prostą instrukcję `1 + 2` na wiele wierszy, zapisz kod w następujący sposób:

```
1 +
  2
```

Nie zapisuj tej instrukcji w poniższy sposób, ponieważ pierwsza instrukcja jest sama w sobie prawidłowa, co wprowadza niejednoznaczność:

```
1
  + 2
```

Reguła jest następująca: zostaje wyświetlony błąd składni za każdym razem, gdy odczytywanie instrukcji wielowierszowej zostanie zakończone przedwcześnie.

ŚREDNIKI

R umożliwia wprowadzanie średników jako znaczników zakończenia instrukcji, ale nie są one wymagane. Większość przewodników po stylach pisania kodu zaleca zrezygnowanie z używania średników w R, a zdecydowanie unikanie ich wstawiania na końcu wiersza.

OPERATORY PRZYPISYWANIA

W R występuje wiele operatorów przypisywania (zobacz tabela 2.1); zalecanym operatorem jest <-, operator = także służy do przypisywania zmiennych, ale za jego pomocą możemy wiązać wartości argumentu z nazwami argumentów podczas wywoływania funkcji (zatem z operatorem = wiąże się pewna niejednoznaczność).

Tabela 2.1. Główne operatory przypisywania w R

Operator	Zadanie	Przykład
<-	Przypisuje wartość z prawej strony do symbolu umieszczonego po lewej stronie.	`x <- 5` *# przypisuje wartość 5 do symbolu x*
=	Przypisuje wartość z prawej strony do symbolu umieszczonego po lewej stronie.	`x = 5` *# przypisuje wartość 5 do symbolu x*
->	Przypisuje wartość z lewej strony do symbolu umieszczonego po prawej stronie (czyli odwrotnie niż w pozostałych przypadkach).	`5 -> x` *# przypisuje wartość 5 do symbolu x*

[4] Notacja kropkowa prawdopodobnie wywodzi się z języka Lisp (mającego duży wpływ na kształt R), a niechęć do podkreślników stanowi relikt czasów, gdy symbol _ był jednym z przydatnych operatorów przypisania w R (obecnie już nie pełni tej funkcji).

LEWA STRONA W OPERACJACH PRZYPISYWANIA

Wiele popularnych języków programowania umożliwia wyłącznie przypisywanie wartości do nazw zmiennych lub symboli. R pozwala wprowadzać wyrażenia fragmentaryczne po lewej stronie operacji przypisywania, a także umożliwia indeksowanie numeryczne oraz logiczne tablic. Zyskujemy w ten sposób dostęp do potężnych poleceń dzielenia tablic i stylów pisania kodu. Na przykład możemy zastąpić wszystkie brakujące wartości (oznaczone jako `NA`) w wektorze zerami w sposób zaprezentowany poniżej:

```
d <- data.frame(x = c(1, NA, 3))
print(d)
#    x
# 1  1
# 2 NA
# 3  3

d$x[is.na(d$x)] <- 0
print(d)
#   x
# 1 1
# 2 0
# 3 3
```

Typ `data.frame` jest tabelarycznym typem danych w R i jednocześnie najważniejszą strukturą danych. Obiekt `data.frame` przechowuje dane zorganizowane w rzędy i kolumny.

Podczas wyświetlania obiektów `data.frame` najpierw prezentowane są numery rzędów w pierwszej (nienazwanej) kolumnie, a wartości mieszczące się w kolumnach występują pod odpowiednimi nazwami kolumn.

Możemy umieścić fragment kolumny x z ramki danych d po lewej stronie operacji przypisywania, aby z łatwością zastąpić wszystkie wartości NA zerami.

WEKTORY CZYNNIKOWE

R obsługuje wiele typów danych: numeryczne, logiczne, stałoprzecinkowe, łańcuchy znaków (zwane typami **znakowymi**) i **wektory czynnikowe** (ang. *factors*). W tym ostatnim przypadku mamy do czynienia z ustalonym zbiorem łańcuchów znaków w postaci stałoprzecinkowej. Wektory czynnikowe pozwalają zaoszczędzić mnóstwo miejsca, a jednocześnie wykazywać własności łańcuchów znaków. Mogą one jednak w zgoła niespodziewany sposób oddziaływać z poleceniem `as.numeric()` (dla wektorów czynnikowych zwracają one kod wektorów czynnikowych, ale dla typów znakowych przeprowadzają analizę składniową tekstu). Wektory czynnikowe kodują także cały zbiór dopuszczalnych wartości, co jest przydatne, ale może sprawiać, że łączenie danych z różnych źródeł (mających różne zestawy wartości) będzie nieco uciążliwe. Proponujemy, aby w celu uniknięcia problemów opóźniać przekształcanie łańcuchów znaków w wektory czynnikowe aż do późnego stadium analizy. Osiągamy to zazwyczaj poprzez dodanie argumentu `stringsAsFactors = FALSE` do takich funkcji jak `data.frame()` czy `read.table()`. Zachęcamy jednak, aby korzystać z wektorów czynnikowych, gdy masz dobry powód do ich użycia, na przykład jeśli chcesz użyć funkcji `summary()` albo przygotować wskaźniki fikcyjne (więcej informacji na temat wskaźników fikcyjnych i ich związków z wektorami czynnikowymi znajdziesz w podpunkcie „Dodatkowe informacje na temat kodowania wektorów czynnikowych”).

ARGUMENTY NAZWANE

Istotą R jest przetwarzanie danych za pomocą funkcji. Funkcje przyjmujące dużą liczbę argumentów szybko stają się mało zrozumiałe i nieczytelne. Z tego właśnie powodu wprowadzono w R argumenty nazwane. Na przykład, gdybyśmy chcieli wyznaczyć */tmp* jako katalog roboczy, moglibyśmy użyć polecenia `setwd()` w następujący sposób: `setwd("/tmp")`. Jednak zgodnie z dokumentacją wyświetlaną po wpisaniu `help(setwd)`

pierwszy argument funkcji `setwd()` ma nazwę `dir`, dlatego moglibyśmy napisać również `setwd(dir = "/tmp")`. Rozwiązanie to okazuje się przydatne w przypadku występowania dużej liczby argumentów, a także do wyznaczania dodatkowych argumentów funkcji. Uwaga: argumenty nazwane muszą być wyznaczane za pomocą operatora =, a nie <-.

> *Jeżeli dysponujesz procedurą zawierającą 10 parametrów, to prawdopodobnie jakieś przegapiłeś.*
>
> — Alan Perlis, *Epigrams on Programming*, „ACM SIGPLAN Notices" 17

NOTACJA STOSOWANA W PRZYPADKU PAKIETÓW

W R można wyznaczać funkcję z pakietu na dwa sposoby. W pierwszym z nich należy najpierw wczytać dany pakiet za pomocą polecenia `library()`, a następnie podać nazwę funkcji. Drugim rozwiązaniem jest podanie nazw pakietu i funkcji, które przedzielamy dwoma dwukropkami (`::`). Przykład tej drugiej metody wygląda następująco: `stats::sd(1:5)`. Zapis wykorzystujący dwukropki pozwala uniknąć niejednoznaczności, a także pozwala szybko ustalić podczas sprawdzania kodu, z którego pakietu pochodzi dana funkcja.

SEMANTYKA WARTOŚCI

R jest o tyle wyjątkowym językiem, że skutecznie symuluje semantykę „kopiowania przez wartość". Jeżeli użytkownik dysponuje dwoma odniesieniami do danych, każde z nich rozwija się niezależnie od drugiego: zmiany wprowadzane w jednym odniesieniu nie wpływają na drugie odniesienie. Jest to bardzo przydatne dla dorywczych programistów, gdyż pozwala wyeliminować olbrzymią rodzinę możliwych błędów związanych z aliasami podczas pisania kodu. Poniżej prezentujemy prosty przykład:

```
d <- data.frame(x = 1, y = 2)
d2 <- d
d$x <- 5

print(d)
#   x y
# 1 5 2

print(d2)
#   x y
# 1 1 2
```

Tworzy przykładowe dane i odniesienie do nich o nazwie `d`

Tworzy dodatkowe odniesienie `d2` **do tych samych danych**

Modyfikuje wartość, do której odnosi się `d`

Zwróć uwagę, że odniesienie `d2` przechowuje dla `x` starą wartość `1`. Własność ta umożliwia bardzo wygodne i bezpieczne pisanie kodu. Wiele języków programowania chroni w ten sposób odniesienia lub wskaźniki w wywołaniach funkcji; R chroni jednak także wartości złożone i robi to we wszystkich sytuacjach (nie tylko w wywołaniach funkcji). Trzeba poświęcić trochę uwagi w przypadku współdzielenia zmian, na przykład podczas wywoływania ostatecznej przypisanej wartości, takiej jak `d2 <- d`, po wprowadzeniu wszystkich pożądanych zmian. Na podstawie własnego doświadczenia możemy stwierdzić, że semantyka izolowania wartości w R zapobiega znacznie większej liczbie problemów, niż jest uciążliwości związanych z kopiowaniem zmian.

ORGANIZOWANIE WARTOŚCI POŚREDNICH

Długie sekwencje obliczeń mogą być trudne do odczytu, utrzymywania i usuwania z nich błędów. Aby tego uniknąć, sugerujemy zarezerwowanie zmiennej o nazwie . (kropka), w której będziemy przechowywać wartości pośrednie. Koncepcja jest następująca: pracuj powoli, aby czynić szybkie postępy. Na przykład popularnym zagadnieniem w analizie danych jest uszeregowanie rejestrów przychodów i obliczenie, jaki ułamek przychodów całkowitych jest uzyskiwany dla danego klucza sortującego. Bardzo łatwo to osiągnąć w R poprzez rozdzielenie tego zadania na podetapy:

```
dane <- data.frame(przychody = c(2, 1, 2),
                   klucz_sortowania = c("b", "c", "a"),
                   stringsAsFactors = FALSE)
print(dane)

#   przychody klucz_sortowania
# 1         2               b
# 2         1               c
# 3         2               a

. <- dane
. <- .[order(.$klucz_sortowania), , drop = FALSE]
.$uporzadkowana_suma_przychodow <- cumsum(.$przychody)
.$ulamek_widzianych_przychodow <- .$uporzadkowana_suma_przychodow/sum(.$przychody)
wynik <- .
print(wynik)

#   przychody klucz_sortowania uporzadkowana_suma_przychodow
# 3         2               a                             2
# 1         2               b                             4
# 2         1               c                             5
#  ulamek_widzianych_przychodow
# 3                          0.4
# 1                          0.8
# 2                          1.0
```

Nasze dane referencyjne (przykładowe).

Przypisuje nasze dane do zmiennej tymczasowej „.". Pierwotne wartości pozostaną w zmiennej data, dzięki czemu w razie potrzeby łatwo będzie powrócić do początku obliczeń.

Przypisuje wynik ze zmiennej . do zmiennej o nazwie łatwiejszej do zapamiętania.

Polecenie order służy do szeregowania rzędów. Argument `drop = FALSE` **nie jest konieczny, ale dobrze wyrobić sobie nawyk jego wstawiania. W przypadku jednokolumnowych obiektów** `data.frame` **bez zdefiniowanego argumentu** `drop = FALSE` **operator indeksowania** `[,]` **przekształci wynik w wektor, co niemal zawsze stanowi problem dla programisty. Argument** `drop = FALSE` **uniemożliwia to przekształcenie, dlatego warto go wstawiać „tak na wszelki wypadek", a do tego okazuje się niezbędny wtedy, gdy obiekt** `data.frame` **zawiera tylko jedną kolumnę lub gdy nie wiemy, czy obiekt ten zawiera więcej kolumn (może on pochodzić z nieznanego źródła).**

Pakiet `dplyr` zastępuje notację kropkową tak zwaną **notacją potokową** (ang. *piped notation*; zapewnianą przez inny pakiet R o nazwie `magrittr` i przypominającą operacje łańcuchowe w języku JavaScript). Pakiet `dplyr` jest bardzo popularny i bardzo możliwe, że często będziesz spotykać się z kodem pisanym w tej notacji, dlatego będziemy z niej korzystać od czasu do czasu, abyś do niej przywykł.

Nie zapominaj jednak, że pakiet `dplyr` stanowi jedynie popularną alternatywę dla standardowego kodu R, a nie jego doskonalszą wersję.

```
library("dplyr")

wynik <- dane %>%
  arrange(., klucz_sortowania) %>%
```

```
  mutate(., uporzadkowana_suma_przychodow = cumsum(przychody)) %>%
  mutate(., ulamek_widzianych_przychodow = uporzadkowana_suma_przychodow/sum(przychody))
```

Każdy etap z poprzedniego przykładu zapisaliśmy tu w notacji pakietu `dplyr`. Funkcja `arrange()` stanowi odpowiednik funkcji `order()`, a polecenie `mutate()` zastępuje operacje przypisywania. Oprócz faktu, że operacja przypisania została umieszczona jako pierwsza (chociaż jest realizowana na sam koniec), to kolejność pozostałych etapów pozostała niezmieniona. Kolejność obliczeń zostaje ustalona za pomocą operatora potoku `%>%`.

Potok `magrittr` akceptuje każdy z następujących zapisów w miejsce `f(x)`: `x %>% f`, `x %>% f()` lub `x %>% f(.)`. Najczęściej stosowana jest notacja `x %>% f`, uważamy jednak, że zapis `x %>% f(.)` jest najbardziej przejrzysty pod względem działania kodu[5].

Szczegóły notacji używanej w pakiecie `dplyr` znajdziesz na stronie *https://dplyr.tidyverse.org/articles/dplyr.html*. Pamiętaj, że usuwanie błędów z długich potoków `dplyr` jest uciążliwe, a w trakcie projektowania i eksperymentowania warto dzielić takie potoki na podetapy oraz przechowywać wartości pośrednie w zmiennych tymczasowych.

Notacja wykorzystująca wynik pośredni pozwala na szybkie rozpoczynanie od nowa, a także na wieloetapowe usuwanie błędów. W niniejszej książce będziemy korzystać z różnych rodzajów notacji.

KLASA DATA.FRAME

Klasa `data.frame` została zaprojektowana pod kątem przechowywania danych w bardzo dobrym formacie umożliwiającym natychmiastową analizę. Obiekty `data.frame` to dwuwymiarowe tablice, w których każda kolumna reprezentuje zmienną, pomiar lub fakt, natomiast każdy wiersz wyznacza przykład (próbkę). W takim formacie pojedyncza komórka przechowuje informacje na temat danego faktu (zmiennej) dla pojedynczego przykładu. Obiekty `data.frame` są implementowane jako nazwane listy wektorów kolumnowych (istnieją również listy kolumnowe, ale stanowią one raczej wyjątek niż regułę w obiektach `data.frame`). W obiekcie `data.frame` wszystkie kolumny mają taką samą długość, a to oznacza, że możemy traktować *k*-ty wpis we wszystkich kolumnach jako tworzenie wiersza.

Operacje na kolumnach `data.frame` są zarówno skuteczne, jak i wektorowe. Dodawanie, przeglądanie i usuwanie kolumn przebiega bardzo szybko. W obiekcie `data.frame` operacje na rzędach mogą być kosztowne, dlatego podczas przetwarzania dużych obiektów `data.frame` lepiej korzystać z wektorowych notacji kolumn.

Obiekt `data.frame` przypomina trochę tabelę bazodanową, gdyż informacje przechowywane są na zasadzie podobnej do schematów: jako jawna lista nazw i typów kolumn. Większość rodzajów analiz najłatwiej wyrażać w postaci przekształceń kolumn `data.frame`.

POZWÓL, ŻEBY R WYKONYWAŁ PRACĘ ZA CIEBIE

Większość popularnych operacji statystycznych lub przetwarzania danych została już bardzo wydajnie zaimplementowana albo w „bazowym” języku R (czyli samym R i jego pakietach podstawowych, takich jak `utils` czy `stats`), albo w pakietach dodatkowych.

[5] We własnych projektach wolimy w rzeczywistości korzystać z „potoków kropkowych” `%.>%` z pakietu `wrapr`, gdyż wymuszają one większą spójność notacyjną.

Jeżeli nie poświęcisz czasu R, to Twoje kontakty z nim będą bardzo burzliwe. Na przykład programista wywodzący się ze środowiska Java może używać pętli for, aby sumować wartości każdego rzędu z dwóch kolumn danych. W R dodawanie dwóch kolumn danych jest jedną z podstawowych czynności i można ją wykonać następująco:

```
d <- data.frame(kol1 = c(1, 2, 3), kol2 = c(-1, 0, 1))
d$kol3 <- d$kol1 + d$kol2
print(d)
#   kol1 kol2 kol3
# 1    1   -1    0
# 2    2    0    2
# 3    3    1    4
```

Obiekty `data.frame` są w istocie nazwanymi listami kolumn. Będziemy korzystać z nich w dalszej części książki. W R pracujemy na kolumnach i pozwalamy, aby mechanizmy wektoryzacji przeprowadzały określoną operację na każdym rzędzie jednocześnie. Jeżeli zamierzasz iterować po rzędach, to będziesz walczył z językiem R.

SZUKAJ GOTOWYCH ROZWIĄZAŃ. Wyszukiwanie odpowiedniej funkcji w R może być nużące, ale warto poświęcić na to czas (zwłaszcza jeżeli sporządzasz notatki). R został zaprojektowany do analizowania danych, zatem większość etapów wymaganych w tym celu została już zaimplementowana, chociaż być może ich nazwa może okazać się nie całkiem zrozumiała albo ich ustawienia domyślne mogą być nietypowe. Zgodnie ze słowami chemika Franka Westheimera: „Dwa miesiące w laboratorium często pozwalają zaoszczędzić dwie godziny w bibliotece”[6]. Jest to celowo ironiczne przeformułowanie zasady szybkich postępów poprzez powolną pracę: analizowanie potencjalnych rozwiązań jest czasochłonne, ale często w ten sposób możesz zaoszczędzić znacznie więcej czasu przeznaczonego na programowanie.

2.2. Praca z danymi przechowywanymi w plikach

Najpopularniejszy, gotowy do użytku format danych należy w istocie do rodziny formatów tablicowych zwanych **wartościami ustrukturyzowanymi** (ang. *structured values*). Większość z wykorzystywanych danych przynależy (lub niemal przynależy) do któregoś z tych formatów. Po wczytaniu takich plików do R zyskujesz możliwość analizowania danych z niewiarygodnie bogatego zakresu publicznych i prywatnych źródeł. W tym podrozdziale przyjrzymy się dwóm przykładom wczytywania danych z plików ustrukturyzowanych oraz przykładowi ich wczytywania bezpośrednio z relacyjnej bazy danych. Naszym zadaniem będzie szybkie wprowadzenie danych do R po to, aby móc przeprowadzać na nich interesującą analizę.

2.2.1. Praca z danymi ustrukturyzowanymi z poziomu plików lub adresów URL

Najprostszym do odczytu formatem są dane tablicowe z nagłówkami. Jak widać na rysunku 2.2, dane te są ułożone w rzędy i kolumny, a nagłówek zawiera nazwy kolumn. Każda kolumna reprezentuje inny fakt lub pomiar; z kolei rzędy symbolizują przykłady

[6] Zobacz *https://en.wikiquote.org/wiki/Frank_Westheimer*.

lub próbki, dla których dysponujemy znanym zbiorem faktów. Wiele danych publicznych występuje w tym formacie, dlatego umiejętność ich odczytu otwiera przed nami mnóstwo możliwości.

Zanim wczytamy omawiany w poprzednim rozdziale zestaw danych German credit, przyjrzyjmy się podstawowym poleceniom wczytywania na przykładzie prostego zestawu danych, dostępnego w repozytorium uczenia maszynowego Uniwersytetu Kalifornijskiego w Irvine (*http://archive.ics.uci.edu/ml/index.php*). Pliki z repozytorium UCI zazwyczaj nie zawierają nagłówków, dlatego przygotowaliśmy pierwszy przykład na podstawie zestawu danych UCI car: *http://archive.ics.uci.edu/ml/machine-learning-databases/car/*. Przygotowany przez nas plik znajdziesz w materiałach dodatkowych w katalogu *PDSwR2/UCICar* (instrukcje znajdziesz w sekcji FM.5.6) i wygląda on tak:

```
buying,maint,doors,persons,lug_boot,safety,rating
vhigh,vhigh,2,2,small,low,unacc
vhigh,vhigh,2,2,small,med,unacc
vhigh,vhigh,2,2,small,high,unacc
vhigh,vhigh,2,2,med,low,unacc
...
```

Nagłówek zawiera nazwy kolumn danych, w tym przypadku rozdzielane przecinkami. Gdy elementami rozdzielającymi są przecinki, mamy do czynienia z formatem CSV (ang. *comma-separated values* — wartości rozdzielane przecinkami).

Rzędy z danymi mają taki sam format jak nagłówek, jednak w każdym rzędzie występują rzeczywiste wartości. W tym przypadku pierwszy rząd reprezentuje zbiór par nazwa/wartość: `buying=vhigh`, `maintenenace=vhigh`, `doors=2`, `persons=2` **itd.**

UNIKAJ „RĘCZNEGO" WYKONYWANIA OPERACJI POZA R. Bardzo zalecamy, abyś unikał „ręcznego" wykonywania czynności poza R podczas importowania danych. Możesz czuć pokusę, aby dodać wiersz nagłówka do pliku za pomocą edytora. Lepszą strategią jest stworzenie skryptu R wykonującego niezbędne czynności przygotowawcze. Automatyzacja tych zadań znacznie zmniejsza stres i wysiłek związane z nieuniknionym odświeżaniem danych. Uzyskiwanie nowych, lepszych danych zawsze powinno być dobrą wiadomością, a pisanie zautomatyzowanych, powtarzalnych procedur stanowi olbrzymi krok w tym kierunku.

W punkcie 2.2.2 zaprezentujemy sposób dodawania nagłówków bez konieczności „ręcznego" edytowania plików.

Zwróć uwagę, że dane przypominają strukturą arkusz kalkulacyjny, na którym możemy z łatwością rozpoznawać rzędy i kolumny. Każdy rząd (niebędący nagłówkiem) reprezentuje opis innego modelu samochodu. W kolumnach znajdziemy fakty dotyczące poszczególnych modeli. Większość kolumn zawiera obiektywne pomiary (koszt zakupu, koszt eksploatacji, liczbę drzwi itd.), a ostatnia kolumna (rating) symbolizuje ogólną, obiektywną ocenę (`vgood` — bardzo dobra, `good` — dobra, `acc` — akceptowalna i `unacc` — nieakceptowalna). Takie szczegóły znajdziemy w dokumentacji dołączonej do pierwotnych danych i stanowią one klucz do projektów (dlatego sugerujemy sporządzanie notatek).

WCZYTYWANIE DANYCH O PRZEJRZYSTEJ STRUKTURZE

Do wczytania tego typu danych w R wystarczy jeden wiersz kodu: wpisz polecenie `utils::read.table()` i gotowe[7]. Zakładamy, że pobrałeś i wypakowałeś materiały dodatkowe do tej książki (*ftp://ftp.helion.pl/przyklady/jrand2.zip* lub w wersji oryginalnej

[7] Możesz także skorzystać z funkcji dostępnych w pakiecie `readr`.

https://github.com/WinVector/PDSwR2) i wyznaczyłeś katalog roboczy *PDSwR2/UCICar*, tak jak zostało wyjaśnione w podrozdziale „Korzystanie z niniejszej książki" na początku. (Należy skorzystać z funkcji `setwd()` i wprowadzić pełną nazwę ścieżki do katalogu *PDSwR2*, nie tylko zaprezentowany przez nas fragment). Gdy już znajdziesz się w katalogu *PDSwR2/UCICar*, będziesz mógł wczytać dane za pomocą listingu 2.1.

Listing 2.1. Wczytywanie zestawu danych UCI car

Polecenie odczytujące dane z pliku lub adresu URL i zapisujące rezultat w nowej ramce danych o nazwie `uciCar`.

Nazwa pliku lub adres URL zawierające dane do pobrania.

Określa znak (w tym przypadku przecinek) rozdzielający kolumny lub pola.

Informacja dla R, że ma spodziewać się wiersza nagłówka, w którym będą zdefiniowane nazwy kolumn.

Informacja dla R, że ma przekształcić wartości znakowe w wektory czynnikowe. Jest to domyślne działanie, dlatego umieszczamy ten argument jedynie w celach poglądowych.

Wyświetla te dane we wbudowanej przeglądarce tabel R.

```
uciCar <- read.table(
    'car.data.csv',
    sep = ',',
    header = TRUE,
    stringsAsFactor = TRUE
    )
View(uciCar)
```

Kod z listingu 2.1 wczytuje dane i umieszcza je w nowej ramce danych `uciCar`, której zawartość wyświetlamy za pomocą polecenia `View()` (rysunek 2.2).

uciCar

Filter

	buying	maint	doors	persons	lug_boot	safety	rating
1	vhigh	vhigh	2	2	small	low	unacc
2	vhigh	vhigh	2	2	small	med	unacc
3	vhigh	vhigh	2	2	small	high	unacc
4	vhigh	vhigh	2	2	med	low	unacc
5	vhigh	vhigh	2	2	med	med	unacc
6	vhigh	vhigh	2	2	med	high	unacc
7	vhigh	vhigh	2	2	big	low	unacc
8	vhigh	vhigh	2	2	big	med	unacc
9	vhigh	vhigh	2	2	big	high	unacc
10	vhigh	vhigh	2	4	small	low	unacc
11	vhigh	vhigh	2	4	small	med	unacc

Showing 1 to 12 of 1,728 entries

Rysunek 2.2. Dane UCI car ukazane w formie tabelarycznej

Polecenie `read.table()` jest potężne i elastyczne; akceptuje wiele różnych rodzajów elementów rozdzielających dane (przecinki, tabulatory, odstępy, potoki itd.) i zawiera wiele opcji sterujących cytatami czy znakami ucieczki. Funkcja ta może odczytywać dane z plików lokalnych lub adresów URL. Jeżeli nazwa zasobu zawiera rozszerzenie *.gz*, funkcja `read.table()` zakłada, że jest on skompresowany w formacie gzip i automatycznie rozpakuje takie archiwum w trakcie jego odczytywania.

POZNAWANIE DANYCH

Po wczytaniu danych do R chcemy się z nimi zapoznać. Istnieje kilka poleceń, które zawsze warto wypróbować na samym początku:

- `class()` — wyświetla rodzaj obiektu, z jakim mamy do czynienia. W naszym przypadku po wpisaniu `class(uciCar)` dowiemy się, że `uciCar` jest obiektem `data.frame`. Klasa jest pojęciem obiektowym, dzięki któremu wiemy, jak dany obiekt będzie się zachowywał. Jest również dostępna (nieco mniej przydatna) komenda `typeof()`, określająca sposób implementacji magazynu obiektu.
- `dim()` — w przypadku ramek danych polecenie to ukazuje liczbę rzędów i kolumn w danych.
- `head()` — pokazuje pięć pierwszych rzędów z danymi. Przykład: `head(uciCar)`.
- `help()` — wyświetla dokumentację danej klasy. Dokładniej rzecz biorąc, wpisz polecenie `help(class(uciCar))`.
- `str()` — prezentuje strukturę obiektu. Wypróbuj `str(uciCar)`.
- `summary()` — za pomocą tego polecenia możesz sprawdzić podsumowanie niemal każdego obiektu R. Dzięki komendzie `summary(uciCar)` uzyskasz mnóstwo informacji na temat rozkładu danych UCI car.
- `print()` — wyświetla wszystkie dane. Uwaga: w przypadku dużych zestawów danych może to potrwać bardzo długo, a raczej nie chciałbyś tracić czasu.
- `View()` — wyświetla dane w prostej przeglądarce tabel.

WIELE FUNKCJI R JEST OGÓLNYCH. Wiele funkcji R jest ogólnych, gdyż obsługują w ten sam sposób różne typy danych, nawet obiektowych, ponieważ są w stanie dobrać odpowiednie działanie w zależności od klasy danego obiektu. Jeżeli natrafisz w jakimś przykładzie na funkcję przetwarzającą jakiś obiekt lub klasę, to proponujemy, żebyś sprawdził jej działanie również na innych obiektach/klasach. Wśród powszechnie występujących funkcji, których można używać z różnymi klasami i typami, znajdziemy takie jak `length()`, `print()`, `saveRDS()`, `str()` i `summary()`. Środowisko R jest bardzo wszechstronne i warto w nim eksperymentować. Najpopularniejsze błędy są wyłapywane i nie są w stanie uszkodzić danych ani zawiesić interpretera R. Zatem gorąco zachęcamy do eksperymentowania!

Listing 2.2 przedstawia wyniki kilku z powyższych poleceń (rezultaty te zostały oznaczone podwójnym symbolem komentarza, czyli ##).

Listing 2.2. Eksplorowanie danych UCI car

```
class(uciCar)
## [1] "data.frame"  <-- Wczytany obiekt uciCar jest typu data.frame.
summary(uciCar)
##    buying       maint        doors
## high :432   high :432   2    :432
## low  :432   low  :432   3    :432
## med  :432   med  :432   4    :432
## vhigh:432   vhigh:432   5more:432
##
## persons      lug_boot     safety
## 2   :576   big  :576   high:576
## 4   :576   med  :576   low :576
## more:576   small:576   med :576
```

```
##
##  rating
## acc  : 384
## good :  69
## unacc:1210
## vgood:  65

dim(uciCar)
## [1] 1728 7
```

Zapis [1] to jedynie znacznik sekwencji wyjściowej. Właściwe informacje są następujące: obiekt `uciCar` zawiera 1728 rzędów i 7 kolumn. Zawsze staraj się sprawdzić, czy dane zostały właściwie przeanalizowane składniowo poprzez przynajmniej sprawdzenie, czy liczba rzędów jest dokładnie o 1 mniejsza od liczby wierszy tekstu w pierwotnym pliku. Różnica wynika z faktu, że nagłówek kolumny jest traktowany jako wiersz tekstu, ale nie jako rząd danych.

Polecenie `summary()` ukazuje nam rozkład każdej zmiennej w zestawie danych. Dowiadujemy się na przykład, że auta zostały podzielone na dwuosobowe (*2* w kolumnie `persons`), czteroosobowe (*4*) i mieszczące więcej osób (*more*), a także że w zestawie danych znajduje się *576* pojazdów dwuosobowych. Poznajemy w ten sposób wiele informacji o zestawie danych i nawet nie musieliśmy żmudnie przygotowywać tabel przestawnych, co byłoby konieczne w arkuszu kalkulacyjnym.

PRACA Z INNYMI FORMATAMI DANYCH

Format CSV nie jest jedynym popularnym formatem danym, na który natrafisz. Wśród innych formatów znajdziesz takie jak TSV (wartości rozdzielane tabulatorami), pliki rozdzielane potokami (kreskami pionowymi), skoroszyty Microsoft Excel, dane JSON i znaczniki XML. Polecenie `read.table()` jest w stanie odczytywać większość formatów danych rozdzielanych znakami. Dla wielu bardziej zaawansowanych formatów danych zostały przygotowane odpowiednie pakiety:

- **CSV/TSV/FWF** — pakiet `readr` (*https://readr.tidyverse.org/*) zawiera narzędzia obsługujące formaty „danych rozdzielanych", takie jak wartości rozdzielane przecinkami (CSV), wartości rozdzielane tabulatorami (TSV) i pliki o stałej szerokości (FWF).
- **SQL** — *https://cran.r-project.org/web/packages/DBI/index.html*.
- **XLS/XLSX** — *https://readxl.tidyverse.org/*.
- **.Rdata/.RDS** — R zawiera formaty danych binarnych (co pozwala unikać komplikacji z analizą składni, cytowaniem, ucieczką, a także utratą precyzji spowodowaną odczytywaniem i zapisywaniem danych numerycznych lub zmiennoprzecinkowych jako tekstu). Format .Rdata służy do zapisywania zbiorów obiektów i nazw obiektów, a także jest wykorzystywany w poleceniach `save()` i `load()`. Format .RDS pozwala na zapisywanie pojedynczych obiektów (bez zapisywania pierwotnej nazwy obiektu) i używany jest wraz z komendami `saveRDS()` oraz `loadRDS()`. Do pracy doraźnej bardziej nadaje się format .RData (ponieważ możemy w nim zapisać cały obszar roboczy R), ale jeżeli chcesz wykorzystywać obiekty wielokrotnie, lepiej używać formatu .RDS, gdyż związane z nim funkcje zapisywania i odczytywania są nieco jaśniej sprecyzowane. Jeżeli chcesz zapisywać wiele elementów w formacie .RDS, sugerujemy korzystanie z **listy nazwanej**.
- **JSON** — *https://cran.r-project.org/web/packages/rjson/index.html*.
- **XML** — *https://cran.r-project.org/web/packages/XML/index.html*.
- **MongoDB** — *https://cran.r-project.org/web/packages/mongolite/index.html*.

2.2.2. Praca z mniej ustrukturyzowanymi danymi

Dane nie zawsze są dostępne w formacie umożliwiającym ich natychmiastową analizę. Kuratorzy danych często poprzestają na przygotowaniu danych w formacie zrozumiałym dla komputera. Omówiony w rozdziale 1. zestaw danych German bank credit stanowi dobry przykład takich danych. Są one przechowywane w formie tabelarycznej pozbawionej nagłówków; wartości są zakodowane w taki sposób, że do ich zrozumienia wymagane jest zapoznanie się z dołączoną dokumentacją. Jest to zjawisko dość powszechne i wynika ze zwyczajów lub ograniczeń związanych z innymi narzędziami przetwarzania danych. Tym razem zaprezentujemy alternatywę dla zmiany formatu danych przed wprowadzeniem ich do R, czyli dokonamy tego już w samym środowisku R. Jest to znacznie lepsze rozwiązanie, gdyż możemy zapisywać i wielokrotnie wykorzystywać polecenia R wymagane do przygotowania danych.

Szczegóły omawianego zestawu danych znaleźć można pod adresem *http://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data)*, my zaś umieściliśmy kopię tych danych w katalogu *PDSwR2/Statlog*. Pokażemy, w jaki sposób przekształcić je w coś bardziej zrozumiałego za pomocą R. Po wykonaniu poniższych czynności będziesz w stanie przeprowadzić analizę omówioną w rozdziale 1. Jak widać, pierwotnie dane te stanowią nieczytelny blok kodu:

```
A11 6 A34 A43 1169 A65 A75 4 A93 A101 4 ...
A12 48 A32 A43 5951 A61 A73 2 A92 A101 2 ...
A14 12 A34 A46 2096 A61 A74 2 A93 A101 3 ...
   ...
```

PRZEKSZTAŁCANIE DANYCH W R

Dane trzeba czasem poddać pewnym przekształceniom, żeby stały się zrozumiałe. Aby móc rozszyfrować problematyczne dane, potrzebujesz tzw. **dokumentacji schematu** (ang. *schema documentation*) lub **słownika danych** (ang. *data dictionary*). W tym przypadku dowiadujemy się z dołączonego opisu, że dane składają się z 20 kolumn wejściowych i jednej kolumny wynikowej. W tym przykładzie plik danych nie zawiera nagłówka. Definicje kolumn i tajemnicze kody *A-** zostały wyjaśnione w dokumentacji danych. Wczytajmy najpierw nieprzetworzone dane do R. Uruchom konsolę R albo aplikację RStudio i wprowadź polecenia zaprezentowane w listingu 2.3.

Listing 2.3. Wczytanie zestawu danych German bank credit

```
setwd("PDSwR2/Statlog")          ← Wstaw tu ścieżkę do katalogu PDSwR2.
 d <- read.table('german.data', sep=' ',
   stringsAsFactors = FALSE, header = FALSE)
```

Plik ten nie zawiera nagłówka kolumn, dlatego nasza ramka danych d będzie przechowywać bezużyteczne nazwy kolumn w postaci *V#*. Możemy zmienić nazwy kolumn w coś bardziej zrozumiałego za pomocą polecenia `c()` (listing 2.4).

Listing 2.4. Wyznaczanie nazw kolumn

```
d <- read.table('german.data',
                sep = " ",
                stringsAsFactors = FALSE, header = FALSE)
```

```
colnames(d) <- c('Status_of_existing_checking_account', 'Duration_in_month',
                 'Credit_history', 'Purpose', 'Credit_amount', 'Savings_account_bonds',
                 'Present_employment_since',
                 'Installment_rate_in_percentage_of_disposable_income',
                 'Personal_status_and_sex', 'Other_debtors_guarantors',
                 'Present_residence_since', 'Property', 'Age_in_years',
                 'Other_installment_plans', 'Housing',
                 'Number_of_existing_credits_at_this_bank', 'Job',
                 'Number_of_people_being_liable_to_provide_maintenance_for',
                 'Telephone', 'foreign_worker', 'Good_Loan')
str(d)
## 'data.frame': 1000 obs. of 21 variables:


## $ Status_of_existing_checking_account                          : chr "A11" "A
    12" "A14" "A11" ...
## $ Duration_in_month                                            : int 6 48 12
    42 24 36 24 36 12 30 ...
## $ Credit_history                                               : chr "A34" "A
    32" "A34" "A32" ...
## $ Purpose                                                      : chr "A43" "A
    43" "A46" "A42" ...
## $ Credit_amount                                                : int 1169 595
    1 2096 7882 4870 9055 2835 6948 3059 5234 ...
## $ Savings_account_bonds                                        : chr "A65" "A
    61" "A61" "A61" ...
## $ Present_employment_since                                     : chr "A75" "A
    73" "A74" "A74" ...
## $ Installment_rate_in_percentage_of_disposable_income          : int 4 2 2 2
    3 2 3 2 2 4 ...
## $ Personal_status_and_sex                                      : chr "A93" "A
    92" "A93" "A93" ...
## $ Other_debtors_guarantors                                     : chr "A101" "
    A101" "A101" "A103" ...
## $ Present_residence_since                                      : int 4 2 3 4
    4 4 4 2 4 2 ...
## $ Property                                                     : chr "A121" "
    A121" "A121" "A122" ...
## $ Age_in_years                                                 : int 67 22 49
    45 53 35 53 35 61 28 ...
## $ Other_installment_plans                                      : chr "A143" "
    A143" "A143" "A143" ...
## $ Housing                                                      : chr "A152" "
    A152" "A152" "A153" ...
## $ Number_of_existing_credits_at_this_bank                      : int 2 1 1 1
    2 1 1 1 1 2 ...
## $ Job                                                          : chr "A173" "
    A173" "A172" "A173" ...
## $ Number_of_people_being_liable_to_provide_maintenance_for     : int 1 1 2 2
    2 2 1 1 1 1 ...
## $ Telephone                                                    : chr "A192" "
    A191" "A191" "A191" ...
## $ foreign_worker                                               : chr "A201" "
    A201" "A201" "A201" ...
## $ Good_Loan                                                    : int 1 2 1 1
    2 1 1 1 1 2 ...
```

Polecenie `c()` stanowi metodę tworzenia wektorów w R[8]. Nazwy kolumn skopiowaliśmy bezpośrednio z dokumentacji zestawu danych. Poprzez przypisanie wektora nazw do akcesora `colnames()` naszej ramki danych przekształciliśmy nazwy kolumn w coś bardziej zrozumiałego.

PRZYPISYWANIE DO AKCESORÓW. W R klasa ramek danych zawiera dużą liczbę akcesorów danych, takich jak `colnames()` czy `names()`. Do wielu z nich możemy przypisywać zmienne, tak jak to zrobiliśmy w listingu 2.3 z nowymi nazwami kolumn: `colnames(d) <- c('Status_of_existing_checking_account', ...)`. Taka możliwość przypisywania danych do akcesorów jest dość niecodzienną, ale bardzo przydatną własnością R.

W dokumentacji danych oprócz nazw kolumn znajdziemy także słownik objaśniający wszystkie te tajemnicze kody *A-**. Na przykład dowiadujemy się, że w kolumnie 4. (zwanej obecnie `Purpose`, czyli przeznaczenie kredytu) kod *A40* oznacza kredyt na nowy samochód, *A41* definiuje pożyczkę na samochód używany itd. Możemy wykorzystać możliwości mapujące R do odwzorowania bardziej opisowych wartości. W pliku *PDSwR2/Statlog/GCDEtapy.Rmd* (w oryginale *GCDSteps.Rmd*) znajdziesz kod w języku R Markdown zawierający dotychczas omówione etapy przygotowywania danych, a także przekształcający wszystkie wartości z postaci *A#* w bardziej zrozumiałe nazwy. Kod umieszczony w tym pliku najpierw implementuje odwzorowanie wartości z dokumentacji zestawu danych na wektor nazwany. W ten sposób możemy zmieniać nieczytelne nazwy (takie jak *A11*) w bardziej zrozumiałe opisy (np. `... < 0 DM`, który sam w sobie najprawdopodobniej oznacza „zero albo mniej zgłoszonych marek niemieckich”)[9]. Kilka pierwszych wierszy w tej definicji mapowania wygląda następująco:

```
mapping <- c('A11' = '... < 0 DM',
             'A12' = '0 <= ... < 200 DM',
             'A13' = '... >= 200 DM / salary assignments for at least 1 year',
             ...
                )
```

Uwaga: podczas tworzenia takiej mapy nazw należy korzystać z symbolu wiązania argumentów `=`, a nie z operatorów przypisywania, takich jak `<-`.

Gdy już mamy zdefiniowaną listę mapowania, możemy skorzystać z następującej pętli `for` po to, aby przekształcić wartości typu `character` z pierwotnych tajemniczych kodów *A-** w krótkie opisy wzięte wprost z dokumentacji. Oczywiście pomijamy w tym przypadku wszystkie kolumny zawierające dane numeryczne (listing 2.5).

Listing 2.5. Przekształcanie danych dotyczących samochodów

Plik ten znajdziesz w katalogu *PDSwR2/Statlog* lub (w wersji oryginalnej) pod adresem *https://github.com/WinVector/PDSwR2/blob/master/Statlog/mapping.R*.

```
source("mapowanie.R")
for(ci in colnames(d)) {
    if(is.character(d[[ci]])) {
```

Preferuje korzystanie z nazw kolumn, a nie indeksów.

[8] Metoda `c()` również łączy wektory ze sobą w taki sposób, żeby nie pojawiały się dodatkowe zagnieżdżenia.

[9] W czasie tworzenia zestawu danych obowiązującą walutą w Niemczech były marki niemieckie (DM).

```
    d[[ci]] <- as.factor(mapping[d[[ci]]])
  }
}
```

Dzięki notacji [[]] wykorzystujemy fakt, że ramki danych są nazwanymi listami kolumn. Z tego powodu po kolei działamy na każdej kolumnie. Zwróć uwagę, że przeszukiwanie mapowania jest zwektoryzowane: w jednym kroku zostaje zastosowane wobec wszystkich elementów kolumny.

Jak już wspomnieliśmy, pełny kod przygotowujący kolumny znajduje się w pliku *PDSwR2/Statlog/GCDEtapy.Rmd*. Zachęcamy Cię, abyś zajrzał do tego pliku i spróbował samodzielnie wykonać te czynności. Dla wygody przygotowaliśmy też dane w pliku *PDSwR2/Statlog/danekredytowe.RDS* (w oryginale *creditdata.RDS*).

EKSPLORACJA NOWYCH DANYCH

Możemy teraz z łatwością sprawdzić przeznaczenie trzech pierwszych kredytów za pomocą polecenia `print(d[1:3,'Purpose'])`. Dzięki komendzie `summary(d$Purpose)` jesteśmy w stanie sprawdzić rozkład tychże danych. To właśnie z powodu tego podsumowania przekształciliśmy wartości w wektory czynnikowe, ponieważ `summary()` nie dostarcza wielu informacji w przypadku typów znakowych; moglibyśmy jednak użyć polecenia `table ↪(d$Purpose, useNA = "always")` na typach znakowych. Możemy również przyjrzeć się relacji pomiędzy typem kredytu a jego wynikiem, co zostało zaprezentowane w listingu 2.6.

Listing 2.6. Podsumowanie kolumn Good_Loan i Purpose

```
setwd("PDSwR2/Statlog")
 d <- readRDS("danekredytowe.RDS")
```

Odczytuje przygotowane dane *Statlog*

```
table(d$Purpose, d$Good_Loan)
```

Wyznacza katalog roboczy. Musisz zastąpić *PDSwR2/Statlog* pełną ścieżką do katalogu Statlog na Twoim komputerze.

```
##                       BadLoan GoodLoan
## business                   34       63
## car (new)                  89      145
## car (used)                 17       86
## domestic appliances         4        8
## education                  22       28
## furniture/equipment        58      123
## others                      5        7
## radio/television           62      218
## repairs                     8       14
## retraining                  1        8
```

Wyświetlony wynik mówi nam, że dane zostały prawidłowo wczytane z pliku. Wiemy jednak również, że istnieją różne źródła danych, takie jak arkusze kalkulacyjne Excel (dzięki pakietowi `readxl` można je traktować tak samo, jak dane przechowywane w plikach) czy bazy danych (w tym takie systemy danych wielkoformatowych jak Apache Spark). Nauczymy się teraz pracować z relacyjnymi bazami danych za pomocą języka SQL i pakietu DBI.

2.3. Praca z relacyjnymi bazami danych

W wielu środowiskach produkcyjnych wykorzystywane przez nas dane są przechowywane nie w plikach, lecz w relacyjnych bazach danych (SQL). Dane publiczne nieraz są umieszczane w plikach (ponieważ łatwiej jest je udostępniać), ale najważniejsze dane

klienckie często są przechowywane w bazach danych. Relacyjne bazy danych łatwo dostosować do setek milionów rekordów i mają istotne własności produkcyjne, takie jak zrównoleglanie, spójność, transakcje, dzienniki zdarzeń czy audyty. Bazy relacyjne zostały zaprojektowane pod względem obsługi przetwarzania transakcji na bieżąco (ang. *online transaction processing* — OLTP), zatem prawdopodobnie mieszczą się tam, gdzie zostały wygenerowane interesujące nas transakcje.

Często możemy eksportować dane do pliku ustrukturyzowanego i wykorzystać metody z poprzednich podrozdziałów do wczytania tych danych do R. Zazwyczaj jednak nie jest to dobre rozwiązanie. Eksportowanie zawartości baz danych do plików nieraz okazuje się zawodne i idiosynkratyczne z powodu utraty informacji o schemacie, znaków ucieczki, cytatów, a także występowania problemów z kodowaniem znaków. Najlepszym sposobem pracy z danymi występującymi w bazie danych jest jej bezpośrednie podłączenie do R, czym zajmiemy się w niniejszym podrozdziale.

W ramach demonstracji zaprezentujemy najpierw sposób wczytywania treści do bazy danych. Relacyjne bazy danych stanowią idealne miejsce na wykonywanie takich przekształceń jak złączanie czy losowanie próby (ale takie pakiety R jak `sqldf` czy `splyr` oferują podobne możliwości), którymi zajmiemy się w rozdziale 5. Zacznijmy od przygotowania danych bazodanowych do następnego przykładu.

2.3.1. Przykładowe dane o rozmiarze produkcyjnym

Jako przykład wykorzystamy dane próby mikrodanych użytku publicznego (ang. *Public Use Microdata Sample* — PUMS) uzyskane dzięki Środowiskowym Badaniom Społecznym (ang. *American Community Survey* — ACS) przeprowadzonym przez Biuro Spisu Ludności Stanów Zjednoczonych w 2016r.; dane te są często w skrócie nazywane **ACS PUMS**. Informacje na temat pobrania i przygotowania próby tych danych znajdziesz w katalogu *PDSwR2/PUMS/pobieranie* (w oryginale *download*). Przygotowaliśmy także przekształconą próbę w pliku *PDSwR2/PUMS/PUMSproba.RDS* (w oryginale *PUMSsample.RDS*), dzięki czemu możesz pominąć początkowe etapy pobierania i przetwarzania danych.

Dane PUMS nadają się idealnie do testowania w miarę realistycznych scenariuszy badawczych: podsumowywania danych i tworzenia modeli przewidujących jedną kolumną danych na podstawie pozostałych kolumn. Jeszcze powrócimy do tego zestawu danych w dalszej części książki.

Jest to godny zestaw danych, zawierający około 3 miliony osób i 1,5 miliona rodzin. To jeden z niewielu udostępnionych przez Biuro Spisu Ludności zestawów danych opisujących poszczególne osoby i rodziny, a nie podsumowania regionalne. Jest to ważne, ponieważ większość popularnych zadań analizy danych jest tworzona z myślą o przetwarzaniu szczegółowych rejestrów zindywidualizowanych, dlatego zestaw PUMS zawiera dane publiczne najbardziej przypominające dane prywatne, na jakich pracuje analityk danych. Każdy rząd zawiera ponad 200 faktów opisujących osobę lub rodzinę (dochód, zatrudnienie, wykształcenie, liczbę pomieszczeń itd.). Dane zawierają identyfikatory odsyłające do rodzin, zatem dane poszczególnych osób można dołączyć do danych rodzin, do których one przynależą. Interesujący jest rozmiar tego zestawu danych: skompresowany zajmuje kilka gigabajtów, zatem jest wystarczająco mały, aby

go umieścić w odpowiedniej sieci albo w pamięci USB, ale zasadniczo zbyt duży, aby można było na nim pracować na laptopie z R wczytanym do pamięci (to ostatnie rozwiązanie raczej nadaje się do zestawów danych mieszczących się w zakresie setek tysięcy rzędów).

PODSUMOWANIA LUB ROZKŁADY BRZEGOWE. Przejście od pojedynczych danych do podsumowań lub rozkładów brzegowych to prosty proces zwany **statystyką podsumowującą** (ang. *summary statistics*) lub **analityką podstawową** (ang. *basic analytics*). Proces odwrotny jest albo niemożliwy, albo stanowi bardzo zaawansowany problem statystyczny (wykraczający poza zakres podstawowej analizy danych). Większość danych udostępnianych przez Biuro Spisu Ludności Stanów Zjednoczonych stanowi podsumowania regionalne, zatem uzyskanie przydatnych modeli predykcyjnych na poziomie pojedynczych osób wymaga skomplikowanej metodologii imputowania statystycznego. Dane PUMS są bardzo przydatne, ponieważ są zorientowane na jednostki.

Dziesiątki milionów rzędów pasują idealnie do relacyjnej bazy danych lub analizy wspomaganej językiem SQL na jednym komputerze. Nie jesteśmy jeszcze zmuszeni do korzystania z klastra bazodanowego lub klastra Apache Spark.

EKSTRAKCJA DANYCH

Żelazną zasadą każdej dziedziny nauki jest możliwość odtwarzania rezultatów. W najgorszym przypadku musisz być w stanie powtórzyć własny udany eksperyment zgodnie z zarejestrowanymi czynnościami. Każdy element musi zawierać dokładne instrukcje jego uzyskania albo jasną dokumentację wyjaśniającą jego pochodzenie. Takie podejście nazywamy dyscypliną „pozbawioną artefaktów Obcych". Na przykład, jeżeli stwierdzamy, że korzystamy z zestawu ACS PUMS, stwierdzenie to okazuje się niewystarczająco precyzyjne, aby wiadomo było, który zestaw danych mamy dokładnie na myśli. Listing 2.7 prezentuje nasz rzeczywisty wpis w notatniku (trzymamy go w internecie, więc możemy go łatwo sprawdzać) na temat zestawu danych PUMS.

Listing 2.7. Dokumentacja dotycząca pochodzenia zestawu danych PUMS (PDSwR2/PUMS/pobieranie/WczytajPUMS.Rmd; w oryginale download/LoadPUMS.Rmd)

```
Dane pobrano 21.04.2018 r. z: Reduce Zoom
```

Data pobrania danych.

Adresy dokumentacji. Jest to ważny zapis, ponieważ wiele plików danych nie zawiera odnośników do dokumentacji.

```
  https://www.census.gov/data/developers/data-sets/acs-1year.2016.html
   https://www.census.gov/programs-surveys/acs/technical-documentation/pums.html
  http://www2.census.gov/programssurveys/acs/tech_docs/pums/data_dict/PUMSDataDict16.txt
  https://www2.census.gov/programs-surveys/acs/data/pums/2016/1-Year/


Najpierw wykonaj następujące czynności w powłoce bash:

wget https://www2.census.gov/programs-surveys/acs/data/pums/
2016/1-Year/csv_hus.zip
 md5 csv_hus.zip
# MD5 (csv_hus.zip) = c81d4b96a95d573c1b10fc7f230d5f7a
 wget https://www2.census.gov/programs-surveys/acs/data/pums/2016/1-
     Year/csv_pus.zip
md5 csv_pus.zip
# MD5 (csv_pus.zip) = 06142320c3865620b0630d74d74181db
```

Dokładny opis wykonanych przez nas czynności.

Szyfry (hasze) kryptograficzne zawartości pobranych plików. Są to bardzo krótkie podsumowania, które są niepowtarzalne dla każdego pliku (prawdopodobieństwo, że dwa pliki będą miały taki sam hasz, jest znikome). Dzięki tym podsumowaniom możemy z łatwością sprawdzić, czy inny badacz w naszej organizacji korzysta z tych samych danych.

```
wget http://www2.census.gov/programssurveys/
     acs/tech_docs/pums/data_dict/PUMSDataDict16.txt
md5 PUMSDataDict16.txt
# MD5 (PUMSDataDict16.txt) = 56b4e8fcc7596cc8b69c9e878f2e699aunzip csv_hus.zip
```

SPORZĄDZAJ NOTATKI. Istotnym elementem pracy analityka danych jest możliwość obrony swoich wyników i odtwarzania pracy. Bardzo zalecamy przechowywanie lokalnie kopii danych i spisywanie notatek. Zwróć uwagę, że w listingu 2.7 pokazujemy nie tylko, w jaki sposób i kiedy pozyskaliśmy dane, lecz także prezentujemy szyfry kryptograficzne pobranych plików. Rozwiązanie to bardzo pomaga w odtwarzaniu wyników, a także wyszukiwaniu potencjalnych różnic i zmian. Zalecamy także, abyś przechowywał wszystkie skrypty i kod w systemie kontroli wersji (wrócimy jeszcze do tego tematu w rozdziale 11.). Musisz koniecznie być w stanie wskazać kod i dane użyte do uzyskania wyników zaprezentowanych w zeszłym tygodniu.

Szczególnie istotną formą pilnowania tych kwestii jest korzystanie z systemu kontroli kodu źródłowego Git, do którego powrócimy w rozdziale 11.

PIERWSZE KROKI Z ZESTAWEM DANYCH PUMS

Koniecznie przejrzyj chociaż pobieżnie dokumentację pobranych danych PUMS: *PDSwR2/PUMS/ACS2016_PUMS_README.pdf* (plik znajdował się w pobranym archiwum zip) i *PDSwR2/PUMS/PUMSDataDict16.txt* (jeden z pobranych przez nas plików). Trzy kwestie zwracają uwagę: dane mają postać plików ustrukturyzowanych, rozdzielanych przecinkami i zawierających nagłówki kolumn; wartości są zakodowane jako nieczytelne wartości stałoprzecinkowe (podobnie jak w przykładzie *Statlog*), a pojedyncze osoby są ważone tak, aby reprezentowały różne liczby dodatkowych rodzin. Skrypt R Markdown[10] *PDSwR2/PUMS/pobieranie/WczytajPUMS.Rmd* odczytuje pliki CSV (ze skompresowanego pliku pośredniego), przekształca wartości w bardziej zrozumiałe łańcuchy znaków i pobiera pseudolosową próbę danych o prawdopodobieństwach proporcjonalnych do określonych wag losowania rodzin. Takie losowanie proporcjonalne jednocześnie zmniejsza rozmiar pliku do ok. 10 MB (taki rozmiar łatwo umieścić w serwisie GitHub) i tworzy próbę, z której można korzystać we właściwy statystycznie sposób bez dalszych odniesień do wag przygotowanych przez Biuro Spisu Ludności.

LOSOWANIE. Gdy mówimy o „próbie pseudolosowej", mamy po prostu na myśli próbę wylosowaną za pomocą generatora liczb pseudolosowych stanowiącego część R. Jest to generator „pseudolosowy", ponieważ w istocie wyznacza on deterministyczną sekwencję losowań, które w założeniu mają być trudne do przewidzenia, a więc w dużej mierze przypomina to otrzymywanie prawdziwie losowej próby. Dobrze jest pracować na próbach pseudolosowych, gdyż są one łatwe do odtworzenia: uruchom generator z takim samym ziarnem losowości, a będziesz otrzymywać identyczne sekwencje losowań. Zanim komputery cyfrowe zyskały na popularności, statystycy osiągali taką powtarzalność za pomocą specjalnych tablic, takich jak *A Million Random Digits with 100,000 Normal Deviates*, opublikowana w 1955 r. przez Rand Corporation. Podstawowe założenie jest takie, że próba losowa powinna mieć własności bardzo podobne do ogółu populacji. Im powszechniejsza jest cecha, nad którą pracujesz, tym prawdziwsze jest to stwierdzenie.

Uwaga: należy poświęcić odrobinę uwagi kwestii powtarzalności eksperymentów pseudolosowych. Wiele elementów może wpływać na dokładne odtworzenie prób i wyników pseudolosowych. Na przykład zmiana kolejności wykonywanych operacji może wygenerować

[10] Język R Markdown zostanie omówiony w dalszej części książki. Jest to istotny format, służący do wspólnego przechowywania kodu R i dokumentacji tekstowej.

różne wyniki (zwłaszcza w przypadku algorytmów równoległych), a nawet pewne szczegóły generatora liczb pseudolosowych zostały zmodyfikowany, gdy język R został zaktualizowany z wersji 3.5.* (która była używana w czasie przygotowywania książki) do wersji 3.6.*. W kwestii reprezentacji zmiennoprzecinkowych czasami trzeba zaakceptować wyniki równoważne, a nie identyczne.

Dane ustrukturyzowane o rozmiarach milionów rzędów najlepiej obsługiwać z poziomu bazy danych, ale R i pakiet `data.table` również przetwarzają dane w tej skali. Będziemy symulować pracę z danymi przechowywanymi w bazie danych poprzez skopiowanie naszej próby PUMS do bazy danych umieszczonej w pamięci (listing 2.8).

Listing 2.8. Wczytywanie danych do R z relacyjnej bazy danych

Wczytuje dane z formatu RDS do pamięci R. Uwaga: musisz zmienić ścieżkę do pliku *PUMSproba* (w oryginale *PUMSsample*) na adres, pod którym przechowujesz katalog PUMS.

Importuje pakiety zawierające funkcje i polecenia, z których będziemy korzystać.

Łączy się z nową bazą danych RSQLite przechowywaną w pamięci. Będziemy z niej korzystać w naszych przykładach. W praktyce łączymy się zazwyczaj z istniejącą bazą danych, taką jak PostgreSQL czy Spark, zawierającą istniejące tablice.

Kopiuje dane ze struktury `dlist` przechowywanej w pamięci do bazy danych.

Usuwa kopię lokalną naszych danych, ponieważ symulujemy korzystanie z danych znalezionych w bazie danych.

Wykorzystuje język kwerend SQL do szybkiego przejrzenia pięciu rzędów danych.

Tworzy uchwyty pakietu `dplyr` odnoszące się do zdalnych danych bazodanowych.

Wykorzystuje pakiet `dplyr` do eksplorowania danych i pracy z nimi.

Wykorzystuje pakiet `rquery` do utworzenia podsumowania zdalnych danych.

```
library("DBI")
library("dplyr")
library("rquery")

dlist <- readRDS("PUMSproba.RDS")
db <- dbConnect(RSQLite::SQLite(), ":memory:")
dbWriteTable(db, "dpus", as.data.frame(dlist$ss16pus))
dbWriteTable(db, "dhus", as.data.frame(dlist$ss16hus))
rm(list = "dlist")

dbGetQuery(db, "SELECT * FROM dpus LIMIT 5")

dpus <- tbl(db, "dpus")
dhus <- tbl(db, "dhus")

print(dpus)
glimpse(dpus)

View(rsummary(db, "dpus"))
```

W listingu 2.8 celowo nie ukazaliśmy wyników poleceń, ponieważ chcemy, abyś samodzielnie sprawdził ten przykład.

PRZYKŁADOWY KOD. Wszystkie przykładowe listingi są dostępne w katalogu *PDSwR2/Przyklady* (w oryginale *CodeExamples*). Użycie takiego kodu jest łatwiejsze od jego przepisywania, a także pewniejsze od kopiowania i wklejania z wydania cyfrowego książki (unikniesz dzięki temu problemu z łamaniem wierszy, kodowaniem znaków i formatowaniem takich elementów jak cudzysłowy drukarskie).

Zwróć uwagę, że chociaż te dane są małe, wykraczają poza zakres wygodnego ich używania w arkuszach kalkulacyjnych. Po wpisaniu poleceń `dim(dlist$ss16hus)` i `dim(dlist$ss16pus)` (przed poleceniem `rm()` lub po ponownym wczytaniu danych) przekonamy się, że nasza

próba z danymi rodzin zawiera 50 000 rzędów i 149 kolumn, natomiast w przypadku danych poszczególnych osób mamy do czynienia ze 109 696 rzędami i 203 kolumnami. Wszystkie kolumny i zakodowane wartości zostały opisane w dokumentacji Biura Spisu Ludności. Dokumentacja ta jest kluczowym elementem, dlatego dołączyliśmy ją w katalogu *PDSwR2/PUMS*.

SPRAWDZANIE I DOBIERANIE DANYCH PUMS

Celem wczytywania danych do R jest ułatwienie ich modelowania i analizowania. Analitycy danych powinni mieć zawsze szybki dostęp do danych i móc je pobieżnie przejrzeć po wczytaniu. Zaprezentujemy na naszym przykładzie, jak można w szybki sposób sprawdzić niektóre kolumny i pola danych PUMS.

Każdy rząd w zestawie danych PUMS reprezentuje anonimową osobę lub rodzinę. Wśród dostępnych danych osobistych znajdują się takie jak wykonywany zawód, wykształcenie, dochód osobisty i wiele innych zmiennych demograficznych. W listingu 2.8 wczytaliśmy te dane, ale zanim przejdziemy dalej, przyjrzyjmy się kilku kolumnom znajdującym się w zestawie danych i opisanym w dokumentacji:

- **Wiek** — liczba stałoprzecinkowa umieszczona w kolumnie `AGEP`.
- **Rodzaj zatrudnienia** — na przykład organizacja komercyjna, organizacja typu non profit itd. (kolumna `COW`).
- **Wykształcenie** — na przykład podstawowe, średnie, wyższe itd. (kolumna `SCHL`).
- **Całkowity dochód osoby** — umieszczony w kolumnie `PINCP`.
- **Płeć pracownika** — oznaczona w kolumnie `SEX`.

Nasze hipotetyczne zadanie będzie polegało na znalezieniu relacji pomiędzy dochodami (wyrażane w dolarach amerykańskich) a tymi zmiennymi. Jest to typowe zadanie modelowania predykcyjnego: określanie relacji pomiędzy jakimiś zmiennymi o znanych wartościach (wiek, zatrudnienie itd.) a zmienną, której wartości chcemy poznać (w tym przypadku dochodami). Zadanie to stanowi przykład uczenia nadzorowanego, co oznacza, że korzystamy z zestawu danych, w których jednocześnie są dostępne zarówno zmienne obserwowalne (w statystyce zwane zmiennymi niezależnymi), jak i nieobserwowany wynik (zmienna zależna). Zazwyczaj uzyskujemy takie dane oznakowane poprzez ich zakup, zatrudnienie osób dodających etykiety do danych lub korzystanie ze starszych danych, w których miałeś czas, aby zaobserwować pożądany wynik.

> **LOSOWANIE PRÓBY NIE JEST POWODEM DO WSTYDU.** Wielu analityków danych spędza zbyt wiele czasu na dostosowywaniu algorytmów do pracy z danymi wielkoskalowymi. Często jest to marnowaniem wysiłku, ponieważ w wielu rodzajach modeli otrzymywane rezultaty są i tak porównywalne z wynikami otrzymywanymi dla rozsądnie dobranego rozmiaru próby. Musisz pracować „ze wszystkimi swoimi danymi" jedynie wtedy, gdy modelowanych aspektów nie da się wyjaśnić za pomocą próby, czyli na przykład podczas charakteryzowania rzadkich zjawisk lub przy wyznaczaniu powiązań w sieciach społecznościowych.

Nie chcemy poświęcać zbyt wiele czasu na sztuczne aspekty omawianego problemu; naszym celem jest zobrazowanie procedur modelowania i obsługi danych. Wnioski zależą w olbrzymim stopniu od dobierania danych (wyznaczania podzbioru używanych danych) i ich kodowania (odwzorowywania rekordów na zrozumiałe symbole). Z tego

właśnie powodu publikacje naukowe zawierają niezbędną sekcję poświęconą materiałom i metodom, w której autorzy opisują sposób doboru i przygotowania danych. Naszą metodą będzie wybór podzbioru „typowych pracowników pełnoetatowych" poprzez jego ograniczenie do danych spełniających wszystkie poniższe warunki:

- Pracownicy, którzy sami określili siebie jako pełnoetatowych.
- Pracownicy zgłaszający co najmniej 30 godzin aktywności zawodowej w tygodniu.
- Pracownicy w wieku od 18. do 65. roku życia.
- Pracownicy z rocznym dochodem w przedziale od 1000 do 250 000 USD.

Za pomocą listingu 2.9 ograniczymy dane do podzbioru spełniającego powyższe kryteria. Listing ten stanowi kontynuację kodu zawartego w listingu 2.8. Nasze dane nie są duże (próba z zestawu danych PUMS), dlatego skorzystamy z pakietu DBI, aby przenieść je do R, gdzie będziemy nad nimi pracować.

Listing 2.9. Wczytywanie danych z bazy danych

```
dpus <- dbReadTable(db, "dpus")

dpus <- dpus[, c("AGEP", "COW", "ESR", "PERNP",
                 "PINCP","SCHL", "SEX", "WKHP")]

for(ci in c("AGEP", "PERNP", "PINCP", "WKHP")) {
   dpus[[ci]] <- as.numeric(dpus[[ci]])
}

dpus$COW <- strtrim(dpus$COW, 50)

str(dpus)
```

Wyznacza podzbiór kolumn, z którymi chcemy pracować. Ograniczanie kolumn nie jest wymagane, ale zwiększa czytelność późniejszych rezultatów.

Kopiuje dane z bazy danych do pamięci R. Zakładamy tutaj, że kod ten stanowi kontynuację listingu 2.8, zatem dołączone pakiety są ciągle dostępne, a uchwyt bazy danych db nadal działa.

Wszystkie kolumny w tej kopii danych PUMS są przechowywane jako typ znakowy po to, aby przechowywać takie cechy, jak zera wiodące z danych pierwotnych. Przekształcamy tu kolumny, które chcemy traktować jako kolumny numeryczne, w typ numeryczny. Wartości, które nie są numeryczne (często brakujące wpisy), zostają zakodowane jako wartość NA, czyli niedostępna.

Nazwy typów wykonywanego zawodu w zestawie danych PUMS są bardzo długie (jest to jeden z powodów wyznaczenia w tych kolumnach typu stałoprzecinkowego), dlatego w tym zestawie danych będziemy skracać kody zatrudnienia do co najwyżej 50 znaków.

Sprawdza kilka pierwszych rzędów danych w orientacji kolumnowej.

UWAŻAJ NA WARTOŚCI NA. Wartość *NA* reprezentuje w R puste lub brakujące dane. Niestety, wiele z poleceń R pomija dyskretnie wartości *NA* bez żadnego ostrzeżenia. Polecenie `table(dpus$COW, useNA = 'always')` będzie ukazywać wartość *NA* w sposób zbliżony do komendy `summary(dpus$COW)`.

Przeprowadziliśmy już kilka standardowych etapów analizy danych: wczytywanie danych, przekształcenie kilku kolumn i przejrzenie danych. Zrealizowaliśmy je za pomocą tzw. R bazowego, co oznacza korzystanie z własności i funkcji samego języka R i podstawowych, dołączonych doń pakietów (takich jak `base`, `stats` czy `utils`). R bardzo dobrze nadaje się do zadań przetwarzania danych, ponieważ większość użytkowników korzysta z tego języka właśnie w tym celu. Mamy również do dyspozycji dodatkowe pakiety, takie jak `dplyr`, które zawierają własną notację przetwarzania, a także mogące realizować wiele czynności bezpośrednio na danych przechowywań w bazie danych,

jak również zdolne do pracy z danymi przetrzymywanymi w pamięci. Te same etapy przetwarzania danych umieściliśmy także w trzech wersjach: wykorzystującej język R Markdown (plik *PDSwR2/PUMS/PUMS1.Rmd*), pakiet `dplyr` (plik *PDSwR2/PUMS/PUMS1_dplyr.Rmd*), a także zaawansowany pakiet generowania kwerend `rquery` (plik *PDSwR2/PUMS/PUMS1_rquery.Rmd*).

Możemy teraz przejść do definiowania naszego hipotetycznego problemu w listingu 2.10, czyli charakteryzowania dochodów w stosunku do innych faktów znanych na temat osób. Zaczniemy od pewnych czynności wiążących się z naszym problemem: będziemy mapować nazwy poziomów zatrudnienia i przekształcimy je w wektory czynnikowe, gdzie każda wartość będzie miała swój poziom referencyjny. Wektory czynnikowe to łańcuchy znaków brane z określonego zbioru (podobnie jak typy wyliczeniowe w innych językach programowania). Wektory czynnikowe zawierają także jeden specyficzny poziom, zwany **poziomem referencyjnym** (ang. *reference level*); zgodnie z konwencją każdy poziom jest odejmowany od poziomu referencyjnego. Na przykład wszystkie poziomy wykształcenia niższe od licencjatu będą miały wyznaczony nowy poziom o nazwie *No Advanced Degree*, który stanie się naszym poziomem referencyjnym. Niektóre funkcje modelowania będą następnie oceniać poziom wykształcenia, na przykład magistra w odniesieniu do poziomu referencyjnego *No Advanced Degree*. Stanie się to bardziej zrozumiałe w trakcie realizowania przykładu.

Listing 2.10. Przemapowanie wartości i wybieranie rzędów z danych

```
target_emp_levs <- c(
  "Employee of a private for-profit company or busine",
  "Employee of a private not-for-profit, tax-exempt, ",
  "Federal government employee",
  "Local government employee (city, county, etc.)",
  "Self-employed in own incorporated business, profes",
  "Self-employed in own not incorporated business, pr",
  "State government employee")

complete <- complete.cases(dpus)

stdworker <- with(dpus,
                  (PINCP>1000) &
                   (ESR=="Civilian employed, at work") &
                   (PINCP<=250000) &
                   (PERNP>1000) & (PERNP<=250000) &
                   (WKHP>=30) &
                   (AGEP>=18) & (AGEP<=65) &
                   (COW %in% target_emp_levs))

dpus <- dpus[complete & stdworker, , drop = FALSE]
```

Definiuje wektor definicji zatrudnienia, które uznajemy za „standardowe”.

Tworzy nowy wektor logiczny wskazujący, które rzędy mają właściwe wartości we wszystkich interesujących nas kolumnach. W rzeczywistych zastosowaniach istotna jest kwestia zajmowania się brakującymi wartościami i nie zawsze można ją rozwiązać poprzez pomijanie niepełnych rzędów. Wrócimy do kwestii zajmowania się brakującymi wartościami podczas omawiania tematu zarządzania danymi.

Tworzy nowy wektor logiczny wyznaczający osoby, które możemy uznać za typowych pracowników pełnoetatowych. Nazwy wszystkich wymienionych tu kolumn zostały opisane w tekście. Od podanej tu definicji zależą wyniki dowolnej przeprowadzanej analizy, dlatego w rzeczywistym projekcie należy poświęcić mnóstwo czasu na dobór odpowiednich elementów. W tym miejscu dosłownie określamy, kogo i co chcemy badać. Zwróć uwagę, że w celu zachowania przejrzystości ograniczyliśmy analizę do cywilów, co w pełnoprawnej analizie stanowiłoby niedopuszczalne ograniczenie.

Ogranicza analizę wyłącznie do rzędów lub przykładów, które spełniają naszą definicję typowego pracownika.

```
no_advanced_degree <- is.na(dpus$SCHL) |
   (!(dpus$SCHL %in% c("Associate's degree",
                       "Bachelor's degree",
                       "Doctorate degree",
                       "Master's degree",
                       "Professional degree beyond a bachelor's degree")))
dpus$SCHL[no_advanced_degree] <- "No Advanced Degree"
dpus$SCHL <- relevel(factor(dpus$SCHL),
                     "No Advanced Degree")
dpus$COW <- relevel(factor(dpus$COW),
                    target_emp_levs[[1]])
dpus$ESR <- relevel(factor(dpus$ESR),
                    "Civilian employed, at work")
dpus$SEX <- relevel(factor(dpus$SEX),
                    "Male")

saveRDS(dpus, "dpus_std_pracownik.RDS")

summary(dpus)
```

Przekształca informacje o wykształceniu: scala poziomy wykształcenia niższe od licencjata (ang. *bachelor*) w jeden poziom No Advanced Degree.

Przekształca kolumny przechowujące łańcuchy znaków w wektory czynnikowe i dobiera poziom referencyjny za pomocą funkcji `relevel()`.

Zapisuje te dane do pliku, dzięki czemu będziemy mogli je wykorzystać w innych przykładach. Plik ten jest również dostępny pod adresem *PDSwR2/PUMS/dpus_std_pracownik.RDS*.

Wyświetla informacje o naszych danych. Jedną z zalet wektorów czynnikowych jest fakt, że funkcja `summary()` generuje ich przydatne statystyki. Warto jednak zawsze poczekać z przekształcaniem łańcuchów znaków w wektory czynnikowe aż do zakończenia operacji przemapowania kodów poziomów.

DODATKOWE INFORMACJE NA TEMAT KODOWANIA WEKTORÓW CZYNNIKOWYCH

Wektory czynnikowe kodują łańcuchy znaków w postaci indeksów stałoprzecinkowych jako znany zbiór możliwych łańcuchów znaków. Na przykład nasza kolumna `SCHL` jest reprezentowana w R w następujący sposób:

```
levels(dpus$SCHL)
## [1] "No Advanced Degree" "Associate's degree"
## [3] "Bachelor's degree"  "Doctorate degree"
## [5] "Master's degree"    "Professional degree beyond a bachelor's degree"

head(dpus$SCHL)
## [1] Associate's degree Associate's degree Associate's degree No Advanced D
     egree Doctorate degree Associate's degree
##   6 Levels: No Advanced Degree Associate's degree Bachelor's degree Doctor
     ate degree ... Professional degree beyond a bachelor's degree

str(dpus$SCHL)
## Factor w/ 6 levels "No Advanced Degree",..: 2 2 2 1 4 2 1 5 1 1 ...
```

Wyświetla możliwe poziomy w kolumnie `SCHL`.

Pokazuje, w jaki sposób kilka pierwszych poziomów jest reprezentowanych w postaci kodu.

Wyświetla kilka pierwszych łańcuchów znaków dla kolumny `SCHL`.

Osoby niebędące statystykami często dziwią się, że można korzystać z kolumn nienumerycznych (np. łańcuchów znaków czy wektorów czynnikowych) jako danych wejściowych lub zmiennych w modelach. Można to osiągnąć na wiele sposobów, a najpopularniejszy z nich jest nazywany **wskaźnikami wprowadzającymi** (ang. *introducing indicators*) lub **zmiennymi fikcyjnymi** (ang. *dummy variables*). W R kodowanie takie jest często realizowane automatycznie i niejawnie. W innych systemach (np. w bibliotece Scikit-Learn Pythona) analityk musi wyznaczyć kodowanie (za pomocą nazwy metody, np. „gorącojedynkowej"). W niniejszej książce będziemy korzystać z tego kodowania, a także dodatkowych, bardziej zaawansowanych jego rodzajów, dostępnych w pakiecie `vtreat`. Kolumna `SCHL` może być jawnie przekształcana w podstawowe zmienne fikcyjne,

o czym przekonamy się już niebawem. Taka strategia przekodowania będzie stosowana zarówno jawnie, jak i niejawnie w tej książce, dlatego zaprezentujemy ją teraz:

Operator `cbind` łączy dwie ramki danych kolumnami, ewentualnie każdy rząd jest tworzony poprzez dopasowywanie kolumn z rzędów w każdej ramce danych.

Tworzy obiekt `data.frame` z kolumną `SCHL`, w której wartości zostają zakodowane jako łańcuchy znaków, a nie jako wektor czynnikowy.

Tworzy macierz zawierającą zmienne fikcyjne, wygenerowane z kolumny wektorów czynnikowych `SCHL`.

```
d <- cbind(
   data.frame(SCHL = as.character(dpus$SCHL),
              stringsAsFactors = FALSE),
   model.matrix(~SCHL, dpus)
 )
d$'(Intercept)' <- NULL
 str(d)
```

Usuwa kolumnę o nazwie (`Intercept`) z ramki danych, ponieważ stanowi ona efekt uboczny operacji model.matrix, którym na razie nie jesteśmy zainteresowani.

Ukazuje strukturę przedstawiającą pierwotny łańcuch znaków z kolumny `SCHL` wraz ze wskaźnikami. Metoda `str()` prezentuje kilka pierwszych rzędów w formacie transpozycji (kolumny są teraz ułożone w poziomie, a rzędy w pionie).

```
## 'data.frame': 41305 obs. of 6 variables:
## $ SCHL : chr "Associate's d
    egree" "Associate's degree" "Associate's degree" "No Advanced Degree" ...
## $ SCHLAssociate's degree                          : num 1 1 1 0 0 1 0 0 0 0 ...
## $ SCHLBachelor's degree                           : num 0 0 0 0 0 0 0 0 0 0 ...
## $ SCHLDoctorate degree                            : num 0 0 0 0 1 0 0 0 0 0 ...
## $ SCHLMaster's degree                             : num 0 0 0 0 0 0 0 1 0 0 ...
## $ SCHLProfessional degree beyond a bachelor's degree: num 0 0 0 0 0 0 0 0 0 0 ...
```

Zwróć uwagę, że poziom referencyjny `No Advanced Degree` nie otrzymał kolumny, natomiast nowe kolumny wskaźnikowe mają wartość `1`, co stanowi informację o tym, która wartość znajduje się w pierwotnej kolumnie `SCHL`. Kolumny należące do poziomu `No Advanced Degree` zawierają zmienne fikcyjne wypełnione zerami, możemy więc także stwierdzić, które przykłady miały taką wartość. Takie kodowanie można odczytać jako „rzędy zawierające same zera stanowią przypadek podstawowy/normalny, a pozostałe rzędy różnią się obecnością jednego wskaźnika (pokazującego interesujący nas przypadek)". Zwróć uwagę, że takie kodowanie zawiera wszystkie informacje przechowywane przez pierwotną formę łańcucha znaków, ale wszystkie kolumny są teraz numeryczne (gdyż format ten jest wymagany przez wiele procedur uczenia maszynowego i modelowania). Taki format jest w sposób niejawny wykorzystywany w wielu funkcjach uczenia maszynowego i modelowego w R, a użytkownik może nawet nie wiedzieć, że ma miejsce takie przekształcenie.

PRACA Z DANYMI PUMS

Możemy w końcu poćwiczyć rozwiązywanie problemu z danymi. Jak widzieliśmy, polecenie `summary(dpus)` wyświetla informacje o rozkładzie każdej zmiennej w zestawie danych. Możemy sprawdzić także relacje pomiędzy zmiennymi za pomocą jednego z poleceń tabelarycznych: `tapply()` lub `table()`. Na przykład, aby sprawdzić liczbę przykładów rozdzielonych jednocześnie pod względem wykształcenia i płci, moglibyśmy skorzystać z polecenia `table(schooling = dpus$SCHL, sex = dpus$SEX)`. Aby

poznać średni przychód w kontekście wykształcenia i płci, posłużylibyśmy się komendą `tapply(dpus$PINCP,list(dpus$SCHL, dpus$SEX), FUN = mean)`.

Wykorzystujemy polecenie tabelaryczne do zliczenia wystąpień każdej pary kolumn SCHL i SEX

```
table(schooling = dpus$SCHL, sex = dpus$SEX)
```

```
##                                                    sex
## schooling                                           Male   Female
##   No Advanced Degree                               13178     9350
##   Associate's degree                                1796     2088
##   Bachelor's degree                                 4927     4519
##   Doctorate degree                                   361      269
##   Master's degree                                   1792     2225
##   Professional degree beyond a bachelor's degree     421      379
```

Za pomocą funkcji tapply zestawia częstość występowania każdej pary kolumn SCHL i SEX

Ten argument jest wektorem danych, które będziemy łączyć (podsumowywać) za pomocą funkcji tapply.

Ta lista argumentów określa sposób grupowania danych, w tym przypadku jednocześnie za pomocą kolumn SCHL i SEX.

Argument ten określa sposób łączenia wartości; w tym przypadku przyjmujemy wartość średnią za pomocą funkcji mean.

```
tapply(
    dpus$PINCP,
    list(dpus$SCHL, dpus$SEX),
    FUN = mean
    )
```

```
##                                                     Male      Female
## No Advanced Degree                              44304.21    33117.37
## Associate's degree                              56971.93    42002.06
## Bachelor's degree                               76111.84    57260.44
## Doctorate degree                               104943.33    89336.99
## Master's degree                                 94663.41    69104.54
## Professional degree beyond a bachelor's degree 111047.26    92071.56
```

Te same obliczenia za pomocą pakietu `dplyr` wyglądają następująco:

```
library("dplyr")

dpus %>%
  group_by(., SCHL, SEX) %>%
  summarize(.,
            count = n(),
            mean_income = mean(PINCP)) %>%
  ungroup(.) %>%
  arrange(., SCHL, SEX)
```

```
## # A tibble: 12 x 4
##      SCHL                                           SEX     count mean_income
##      <fct>                                          <fct>   <int>       <dbl>
##  1 No Advanced Degree                               Male    13178      44304.
##  2 No Advanced Degree                               Female   9350      33117.
##  3 Associate's degree                               Male     1796      56972.
##  4 Associate's degree                               Female   2088      42002.
##  5 Bachelor's degree                                Male     4927      76112.
##  6 Bachelor's degree                                Female   4519      57260.
##  7 Doctorate degree                                 Male      361     104943.
##  8 Doctorate degree                                 Female    269      89337.
##  9 Master's degree                                  Male     1792      94663.
## 10 Master's degree                                  Female   2225      69105.
## 11 Professional degree beyond a bachelor's degree   Male      421     111047.
## 12 Professional degree beyond a bachelor's degree   Female    379      92072.
```

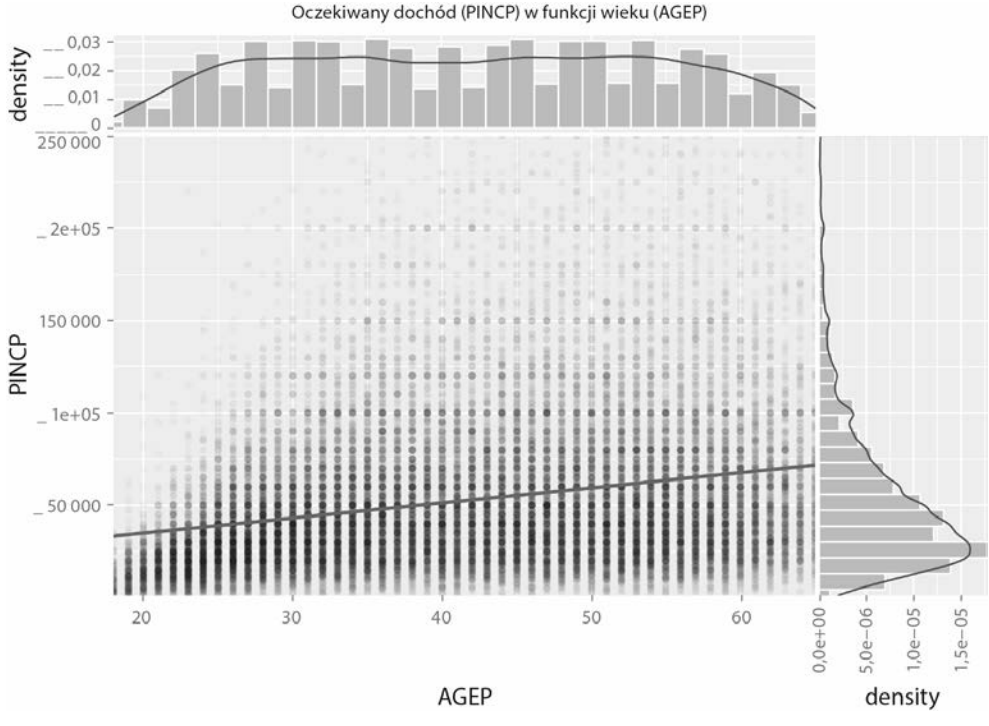
Potoki `dplyr` wyrażają zadania jako sekwencje podstawowych przekształceń danych. Zwróć także uwagę, że wynik funkcji `tapply()` to tzw. format szeroki (komórki danych dopasowywane po rzędzie i kolumnie), natomiast wynik funkcji `dplyr()` występuje w formacie wysokim (komórki danych dopasowywane zgodnie z kluczową kolumną w każdym rzędzie).

Możemy nawet stworzyć wykres relacji za pomocą listingu 2.11. W końcu, gdybyśmy chcieli wymodelować przychód jako funkcję wspólną wszystkich pozostałych zmiennych, moglibyśmy skorzystać z modelu regresji, omówionego w rozdziale 8. Przekształcanie pomiędzy takimi formatami stanowi jeden z motywów przewodnich rozdziału 5.

Listing 2.11. Tworzenie wykresu danych

```
WVPlots::ScatterHist(
  dpus, "AGEP", "PINCP",
  "Oczekiwany dochód (PINCP) w funkcji wieku (AGEP)",
  smoothmethod = "lm",
  point_alpha = 0.025)
```

Oto nadszedł moment, w którym możemy świętować, gdyż w końcu zrealizowaliśmy cel analizy danych. Na rysunku 2.3 widzimy dane i relacje pomiędzy nimi. Objaśnieniem informacji widocznych w podsumowaniu wykresu (*Test summary*) zajmiemy się w rozdziale 8.



Rysunek 2.3. Wykres punktowy dochodu (PINCP) w funkcji wieku (AGEP)

Jeszcze kilkakrotnie powrócimy do danych Biura Spisu Ludności i zaprezentujemy bardziej zaawansowane techniki modelowania. W każdym z tych przypadków celem będzie ukazanie podstawowych wyzwań, jakie napotykają analitycy danych, a także zaprezentowanie narzędzi R, stworzonych z myślą o rozwiązywaniu tych problemów. Bardzo zalecamy realizację omówionych w tym rozdziale ćwiczeń i korzystanie z polecenia `help()` z każdą nową funkcją; dobrym pomysłem jest również poszukiwanie oficjalnej dokumentacji i instrukcji w internecie.

Podsumowanie

W tym rozdziale poznaliśmy podstawy wstępnego wydobywania, przekształcania i wczytywania danych w ramach ich analizy. W przypadku mniejszych zestawów danych dokonywaliśmy przekształceń za pomocą R w pamięci komputera. Zalecamy, aby większe zestawy danych umieszczać w bazie danych SQL, a nawet w systemach danych wielkoskalowych, takich jak Spark (za pomocą kombinacji pakietów `sparklyr` i `SQL`, `dplyr` lub `rquery`). W każdym przypadku zapisujemy ***wszystkie*** etapy przekształceń jako kod (w języku SQL lub R), którego można wielokrotnie używać podczas odświeżania danych. Celem tego rozdziału było przygotowanie podwalin pod interesujące zadania czekające nas w następnych rozdziałach: eksplorowanie, poprawianie, modelowanie danych i zarządzanie nimi.

Język R został stworzony z myślą o przetwarzaniu danych, a celem wczytywania danych jest ich przegląd i praca z nimi. W rozdziale 3. nauczymy się charakteryzować dane za pomocą podsumowań, eksploracji i wykresów. Są to kluczowe elementy we wczesnych fazach modelowania, ponieważ właśnie dzięki nim poznasz szczegóły oraz naturę problemu, z jakim masz do czynienia.

W tym rozdziale dowiedziałeś się, że:

- Ramki danych, czyli obiekty, w których każdy rząd reprezentuje przykład, a kolumna symbolizuje zmienną lub pomiar, są znakomitymi strukturami przeznaczonymi do analizowania danych.
- Za pomocą wyrażenia `utils::read.table()` lub pakietu `readr` możesz wczytywać małe, ustrukturyzowane zestawy danych do R.
- Pakiet `DBI` pozwala na bezpośrednią pracę z bazami danych lub systemem Apache Spark przy użyciu bibliotek `SQL`, `dplyr` lub `rquery`.
- Język R został zaprojektowany do szczegółowego przetwarzania danych i zawiera wiele gotowych poleceń i funkcji transformujących dane. Jeżeli jakieś zadanie staje się zbyt trudne, zazwyczaj wynika to z faktu, że przypadkiem starasz się ponownie odkryć Amerykę i od nowa zaimplementować jakieś wysokopoziomowe przekształcenia danych za pomocą niskopoziomowych operacji programistycznych.

Analiza danych albo nauka o danych jest interdyscyplinarną dziedziną, dzięki której hipotezy i dane przekształca się w zrozumiałe przewidywania. Predykcyjna analiza danych przynosi wymierne korzyści w wielu dziedzinach, od polityki począwszy, a na udzielaniu kredytów skończywszy. Osobą odpowiedzialną za tę magię jest analityk danych — człowiek, który zbiera i przygotowuje dane, wybiera technikę modelowania, pisze kod, weryfikuje wyniki swojej pracy, wreszcie komunikuje je interesariuszom. Jak widać, profesja analityka danych jest wyjątkowo atrakcyjna i wyjątkowo wymagająca. Aby określić umiejętności praktyczne wymagane w zawodzie analityka danych, najlepiej prześledzić realizacje konkretnych projektów z wykorzystaniem rzeczywistych danych.

Ta książka jest samouczkiem prezentującym praktyczne aspekty dziesiątek technik, które wykorzystują profesjonalni analitycy danych. Główny nacisk autorzy położyli na zadania: ich zaplanowanie, przygotowanie, realizację i prezentację wyników. Dzięki praktycznemu podejściu z tej pozycji skorzystają zarówno analitycy biznesowi, jak i badacze danych. Pokazano tu, w jakich przypadkach i w jaki sposób należy stosować techniki statystyczne oraz metody uczenia maszynowego. W każdym rozdziale omówiono nowe narzędzia w kontekście rzeczywistych, praktycznych projektów. W rezultacie powstał potężny zbiór przydatnych ćwiczeń napisanych w języku R, opatrzonych wartościowymi wskazówkami, komentarzami i podpowiedziami.

W książce między innymi:

- zasady zarządzania procesem analizy danych
- zadania analityka danych
- przekształcanie danych w celu przygotowania ich do analizy
- techniki statystyczne i metody uczenia maszynowego w języku R
- zaawansowane metody modelowania
- tajniki skutecznego prezentowania wyników analiz

R: jesteś gotów na właściwe wyniki analizy danych?

Nina Zumel pracowała jako naukowiec w SRI International, niezależnym instytucie badawczym typu non profit. Była głównym naukowcem w przedsiębiorstwie zajmującym się optymalizacją kosztów, a także założyła firmę prowadzącą badania. Obecnie jest głównym doradcą w firmie Win Vector LLC.

John Mount pracował jako badacz obliczeniowy w dziedzinie biotechnologii. Zajmował się również projektowaniem algorytmów giełdowych oraz zarządzał zespołem badawczym w firmie Shopping.com. Obecnie jest głównym doradcą w firmie Win Vector LLC.

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-6816-3

9 788328 368163

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 99,00 zł