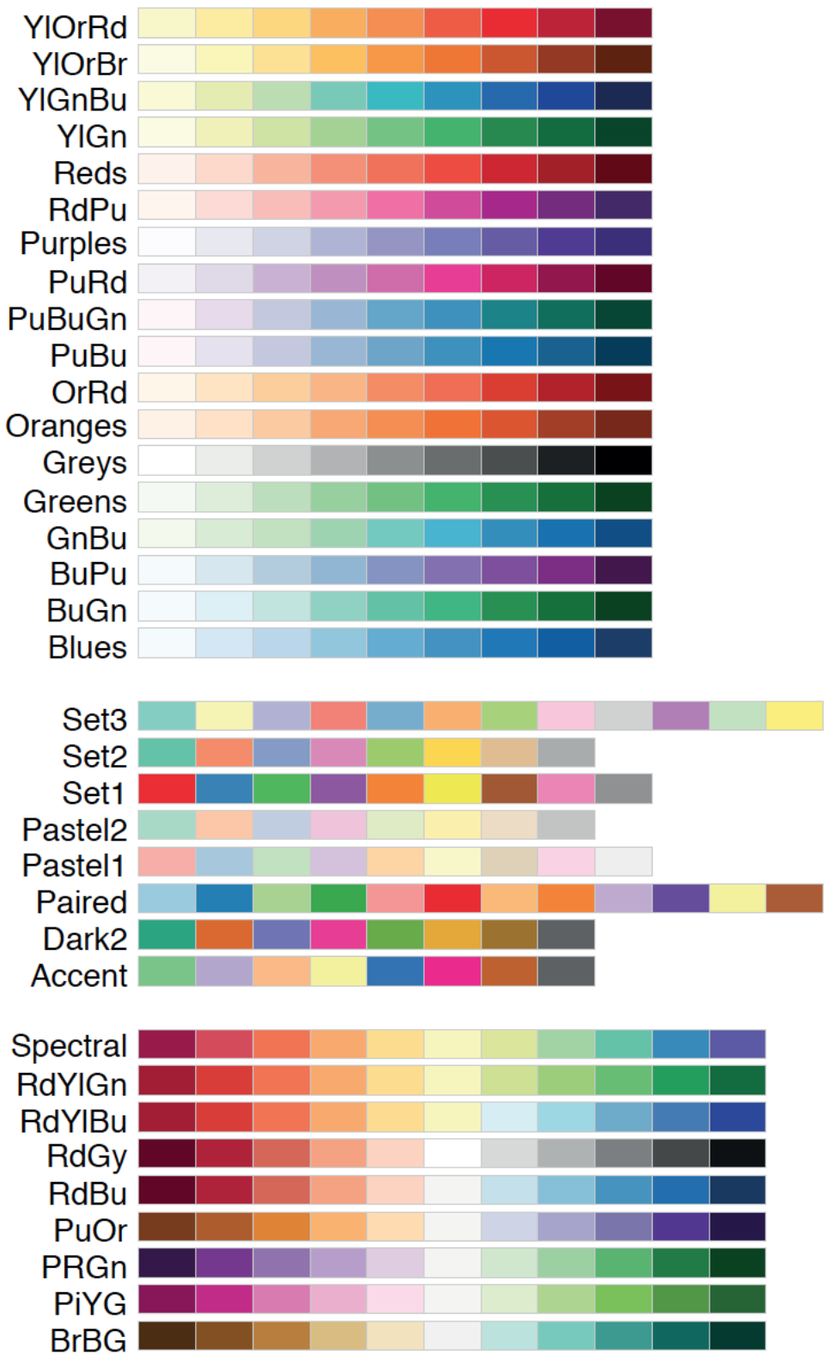
Przemysław Biecek

Przewodnik po pakiecie

Oficyna Wydawnicza GiS

palegreen4 gray100
palegreen3 gray99
palegreen2 gray98
palegreen1 gray97
palegreen gray96
palegoldenrod gray95
orchid4 gray94
orchid3 gray93
orchid2 gray92
orchid1 gray91
orchid gray90
orangered4 gray89
orangered3 gray88
orangered2 gray87
orangered1 gray86
orangered gray85
orange4 gray84
orange3 gray83
orange2 gray82
orange1 gray81
orange gray52
grey gray51
greenyellow gray50
green4 gray49
green3 gray48
green2 gray47
green1 gray46
green gray45
gray gray44
goldenrod4 gray43
goldenrod3 gray42
goldenrod2 gray41
goldenrod1 gray40
goldenrod gray39
gold4 gray38
gold3 gray37
gold2 gray23
gold1 gray22
gold gray21
ghostwhite gray20
gainsboro gray19
forestgreen gray18
floralwhite gray17
firebrick4 gray16
firebrick3 gray15
firebrick2 gray14
firebrick1 gray13
firebrick gray12
dodgerblue4 gray11
dodgerblue3 gray10
dodgerblue2 gray9
dodgerblue1 gray8
dodgerblue gray7
dimgrey gray6
dimgray gray5
deepskyblue4 gray4
deepskyblue3 gray3
deepskyblue2 gray2
deepskyblue1 gray1
deepskyblue gray0

RColorBrewer



coral3 deeppink4 lightpink1 mistyrose1 pink4 slategray1
coral2 deeppink3 lightpink mistyrose pink3 slategray
coral1 deeppink2 lightgrey mintcream pink2 slateblue4
coral deeppink1 lightgreen midnightblue pink1 slateblue3 yellowgreen
chocolate4 deeppink lightgray mediumvioletred pink slateblue2 yellow4
chocolate3 darkviolet lightgoldenrodyello mediumturquoise peru slateblue1 yellow3
chocolate2 darkturquoise lightgoldenrod4 mediumspringgree peachpuff4 slateblue yellow2
chocolate1 darkslategrey lightgoldenrod3 mediumslateblue peachpuff3 skyblue4 yellow1
chocolate darkslategray4 lightgoldenrod2 mediumseagreen peachpuff2 skyblue3 yellow
chartreuse4 darkslategray3 lightgoldenrod1 mediumpurple4 peachpuff1 skyblue2 whitesmoke
chartreuse3 darkslategray2 lightgoldenrod mediumpurple3 peachpuff skyblue1 wheat4
chartreuse2 darkslategray1 lightcyan4 mediumpurple2 papayawhip skyblue wheat3
chartreuse1 darkslategray lightcyan3 mediumpurple1 palevioletred4 sienna4 wheat2
chartreuse darkslateblue lightcyan2 mediumpurple palevioletred3 sienna3 wheat1
cadetblue4 darkseagreen4 lightcyan1 mediumorchid4 palevioletred2 sienna2 wheat
cadetblue3 darkseagreen3 lightcyan mediumorchid3 palevioletred1 sienna1 violetred4
cadetblue2 darkseagreen2 lightcoral mediumorchid2 palevioletred sienna violetred3
cadetblue1 darkseagreen1 lightblue4 mediumorchid1 paleturquoise4 seashell4 violetred2
cadetblue darkseagreen lightblue3 mediumorchid paleturquoise3 seashell3 violetred1
burlywood4 darksalmon lightblue2 mediumblue paleturquoise2 seashell2 violetred
burlywood3 lightblue1 mediumaquamarin paleturquoise1 seashell1 violet
burlywood2 darkorchid4 lightblue maroon4 paleturquoise seashell turquoise4
burlywood1 darkorchid3 lemonchiffon4 maroon3 palegreen4 seagreen4 turquoise3
burlywood darkorchid2 lemonchiffon3 maroon2 palegreen3 seagreen3 turquoise2
brown4 darkorchid1 lemonchiffon2 maroon1 palegreen2 seagreen2 turquoise1
brown3 darkorchid lemonchiffon1 maroon palegreen1 seagreen1 turquoise
brown2 darkorange4 lemonchiffon magenta4 palegreen seagreen tomato4
brown1 darkorange3 lawngreen magenta3 palegoldenrod sandybrown tomato3
brown darkorange2 lavenderblush4 magenta2 orchid4 salmon4 tomato2
blueviolet darkorange1 lavenderblush3 magenta1 orchid3 salmon3 tomato1
blue4 darkorange lavenderblush2 magenta orchid2 salmon2 tomato
blue3 darkolivegreen4 lavenderblush1 linen orchid1 salmon1 thistle4
blue2 darkolivegreen3 lavenderblush limegreen orchid salmon thistle3
blue1 darkolivegreen2 lavender lightyellow4 orangered4 saddlebrown thistle2
blue darkolivegreen1 khaki4 lightyellow3 orangered3 royalblue4 thistle1
blanchedalmond darkolivegreen khaki3 lightyellow2 orangered2 royalblue3 thistle
black darkmagenta khaki2 lightyellow1 orangered1 royalblue2 tan4
bisque4 darkkhaki khaki1 lightyellow orangered royalblue1 tan3
bisque3 darkgrey khaki lightsteelblue4 orange4 royalblue tan2
bisque2 darkgreen ivory4 lightsteelblue3 orange3 rosybrown4 tan1
bisque1 darkgray ivory3 lightsteelblue2 orange2 rosybrown3 tan
bisque darkgoldenrod4 ivory2 lightsteelblue1 orange1 rosybrown2 steelblue4
beige darkgoldenrod3 ivory1 lightsteelblue orange rosybrown1 steelblue3
azure4 darkgoldenrod2 ivory lightslategrey olivedrab4 rosybrown steelblue2
azure3 darkgoldenrod1 indianred4 lightslategray olivedrab3 red4 steelblue1
azure2 darkgoldenrod indianred3 lightslateblue olivedrab2 red3 steelblue
azure1 darkcyan indianred2 lightskyblue4 olivedrab1 red2 springgreen4
azure darkblue indianred1 lightskyblue3 olivedrab red1 springgreen3
aquamarine4 cyan4 indianred lightskyblue2 oldlace red springgreen2
aquamarine3 cyan3 hotpink4 lightskyblue1 navyblue purple4 springgreen1
aquamarine2 cyan2 hotpink3 lightskyblue navy purple3 springgreen
aquamarine1 cyan1 hotpink2 lightseagreen navajowhite4 purple2 snow4
aquamarine cyan hotpink1 lightsalmon4 navajowhite3 purple1 snow3
antiquewhite4 cornsilk4 hotpink lightsalmon3 navajowhite2 purple snow2
antiquewhite3 cornsilk3 honeydew4 lightsalmon2 navajowhite1 powderblue snow1
antiquewhite2 cornsilk2 honeydew3 lightsalmon1 navajowhite plum4 snow
antiquewhite1 cornsilk1 honeydew2 lightsalmon moccasin plum3 slategrey
antiquewhite cornsilk honeydew1 lightpink4 mistyrose4 plum2 slategray4
aliceblue cornflowerblue honeydew lightpink3 mistyrose3 plum1 slategray3
white coral4 grey100 lightpink2 mistyrose2 plum slategray2

Przemysław Biecek

Przewodnik po pakiecie

Przemysław Biecek

Przewodnik po pakiecie

Wnioski, skargi i zażalenia należy kierować na adres
dr hab. inż. Przemysław Biecek
Uniwersytet Warszawski, Politechnika Warszawska
e-mail: przemyslaw.biecek@gmail.com
strona www: http://www.biecek.pl

Redakcja i korekta
Katarzyna Wróbel

Projekt okładki:
Jakub Rakusa-Suszczewski

Skład komputerowy książki w systemie LaTeX:
Przemysław Biecek

Przygotowałem tę książkę, aby przedstawić możliwości programu R, filozofię pracy z tym programem, pokazać dobre praktyki dotyczące codziennej pracy z danymi w R. Liczę, że książka ta przyda się tym wszystkim, którzy w pracy lub w szkole zajmują się analizą danych. Książka może być wykorzystana, jako pomoc w nauce programu R lub jako atlas funkcjonalności i możliwości tego programu. W latach 2010 – 2016 używałem jej z powodzeniem jako główny podręcznik do przedmiotu „Programowanie w R". Mam nadzieje, że pomoże ona odkryć przyjemność w pracy z danymi i z programem R.

Pod adresem http://www.biecek.pl/R/ czytelnik znajdzie dodatkowe informacje, treści przykładów oraz odnośniki do innych materiałów pomagających w nauce programu R.

Dziękuję za pomoc przy przygotowaniu czwartego wydania, wiele cennych uwag i korekt (w kolejności alfabetycznej): Bartoszowi Jabłońskiemu, Katarzynie Pękała, Pawłowi Sakowskiemu, Bartoszowi Sękiewiczowi, Małgorzacie Szatkowskiej-Żak i Krzysztofowi Trajkowskiemu.

Dziękuję mojej żonie i dzieciom za cierpliwość, wsparcie i wyrozumiałość.

Copyright © 2008, 2011, 2014, 2017 by Przemysław Biecek

Wszelkie prawa zastrzeżone. Żadna część niniejszej publikacji, zarówno w całości, jak i we fragmentach, nie może być reprodukowana w sposób elektroniczny, fotograficzny i inny bez pisemnej zgody posiadaczy praw autorskich.

Wydanie czwarte rozszerzone, Wrocław 2017
ISBN: 978-83-62780-44-0
Oficyna Wydawnicza GiS, s.c., http://www.gis.wroc.pl
Druk i oprawa: I-BiS Usługi Komputerowe Wydawnictwo s.c.

Spis treści

1. Wprowadzenie 1
1.1. Data science, czyli dlaczego warto poznać R 1
1.2. Jak wygląda praca z programem R 4
1.2.1. Przykład: Pozycja Polski w rankingu FIFA 5
1.3. Jak przygotować środowisko pracy 7
1.3.1. Instalacja podstawowego środowiska R 8
1.3.2. Instalacja edytora RStudio 9
1.3.3. Instalacja dodatkowych pakietów 11
1.4. Gdzie szukać dalszych informacji 13

2. Podstawy pracy z R 16
2.1. Wczytywanie danych 16
2.2. Struktury danych 18
2.2.1. Wektory 18
2.2.2. Ramki danych 20
2.2.3. Listy 22
2.3. Statystyki opisowe 23
2.3.1. Zmienne ilościowe 25
2.3.2. Zmienne jakościowe 26
2.4. Statystyki graficzne 28
2.4.1. Wykres słupkowy 28
2.4.2. Histogram 29
2.4.3. Wykres pudełkowy 31
2.4.4. Jądrowy estymator gęstości 32
2.4.5. Wykres kropkowy 35
2.4.6. Wykres mozaikowy 36
2.5. Jak przetwarzać dane z pakietem `dplyr` 38
2.5.1. Jak filtrować wiersze 39
2.5.2. Jak wybierać kolumny 40
2.5.3. Jak tworzyć i transformować zmienne 40
2.5.4. Jak sortować wiersze 41
2.5.5. Jak pracować z potokami 42

2.5.6. Jak wyznaczać agregaty/statystyki w grupach 45
2.5.7. Postać szeroka/postać wąska 47
2.5.8. Sklejanie/rozcinanie kolumn 49
2.6. Jak wczytać i zapisać dane w różnych formatach 50
2.6.1. Wczytywanie danych z pakietów 50
2.6.2. Wczytywanie danych z plików tekstowych 51
2.6.3. Zapisywanie danych do pliku tekstowego 57
2.6.4. Wczytywanie i zapisywanie z zastosowaniem formatu JSON 58
2.6.5. Wczytywanie danych w formatach HTML i XML 60
2.6.6. Inne formaty plików tekstowych 60
2.6.7. Wczytywanie danych z plików Excela 62
2.6.8. Wczytywanie danych z SPSS'a 64
2.6.9. Wczytywanie danych z programu Matlab 65
2.6.10. Wczytywanie danych z SAS 66
2.6.11. Inne funkcje do importu danych 66
2.7. Automatyczne raporty, powtarzalne badania 67
2.7.1. Pakiet `knitr`, markdown a raporty w HTML 68
2.7.2. Prezentacje w HTML5 72
2.7.3. Pakiet `Sweave` a raporty w języku LaTeX 74

3. Niezbędnik programisty 79
3.1. Instrukcje sterujące 80
3.1.1. Jak tworzyć funkcje 80
3.1.2. Jak stosować instrukcje warunkowe 85
3.1.3. Jak budować pętle 88
3.2. Jak pracować z bazami danych 92
3.2.1. Jak pracować z bazą danych SQLite 93
3.2.2. Jak pracować z większymi relacyjnymi bazami danych . . 94
3.2.3. Jak używać pakietu `dplyr` w pracy z bazami danych . . . 96
3.3. Budowa aplikacji WWW z pakietem `shiny` 98
3.3.1. Jak wygląda model akcja–reakcja 99
3.3.2. Jak opisać interfejs użytkownika 99
3.3.3. Jak opisać przetwarzanie na serwerze 100
3.3.4. Jak dodawać kontrolki sterujące 102
3.4. Budowanie własnych pakietów 105
3.4.1. Niezbędne oprogramowanie 106
3.4.2. Jak wygląda struktura pakietu 106
3.4.3. Jak stworzyć nowy pakiet 107
3.4.4. Plik `DESCRIPTION` 108
3.4.5. Jak dodawać funkcje do pakietu 109
3.4.6. Jak dodawać zbiory danych do pakietu 111

3.4.7. Jak dodawać testy do pakietu . . . 113
3.4.8. Jak budować stworzony pakiet . . . 115
3.5. Git, GitHub i kontrola wersji . . . 117
3.5.1. Jak kopiować repozytorium – `clone` . . . 117
3.5.2. Jak dodawać zmiany – `commit` . . . 119
3.6. Debugger . . . 119
3.6.1. Co mogłoby pójść źle? . . . 119
3.6.2. Błędy i ostrzeżenia . . . 120
3.6.3. Co można zrobić post-mortem – funkcja `traceback()` . . . 120
3.6.4. Jak zastawić pułapkę – funkcja `recover()` . . . 121
3.6.5. Jak śledzić krok po kroku – funkcja `debug()` . . . 122
3.6.6. Jak ignorować błędy – funkcja `try()` . . . 123
3.6.7. Zaokrąglenia numeryczne – studium przypadku . . . 124
3.7. Profiler . . . 125
3.7.1. Jak mierzyć czas działania bloku instrukcji . . . 125
3.7.2. Jak mierzyć czas co do milisekundy . . . 126
3.7.3. Jak szukać wąskich gardeł . . . 127
3.7.4. Jak przedstawiać graficznie wyniki profilowania . . . 130
3.7.5. Jak zrównoleglać obliczenia . . . 130
3.7.6. Jak zwiększać wydajność R . . . 135
3.8. Więcej o obiektach w R . . . 138
3.8.1. Funkcje polimorficzne i klasy S3 . . . 138
3.8.2. Tworzenie własnych operatorów . . . 140
3.8.3. Obiekty wywołań funkcji . . . 140
3.8.4. Leniwa ewaluacja . . . 141
3.8.5. Zasięg symboli w przestrzeniach nazw . . . 143
3.8.6. Współdzielona przestrzeń nazw . . . 145
3.8.7. Obiekty . . . 146
3.8.8. Klasy S4 . . . 148
3.8.9. Formuły . . . 151
3.9. Inne przydatne funkcje . . . 153
3.9.1. Rodzina funkcji `*apply` . . . 154
3.9.2. Pakiety `plyr` i `reshape2` . . . 157
3.9.3. Funkcje systemowe . . . 161
3.9.4. Operacje na plikach i katalogach . . . 162

4. Niezbędnik statystyka . . . 165
4.1. Generatory liczb losowych . . . 166
4.1.1. Wprowadzenie do generatorów liczb pseudolosowych . . . 166
4.1.2. Popularne rozkłady zmiennych losowych . . . 168
4.1.3. Wybrane metody generowania zmiennych losowych . . . 175

4.1.4. Estymacja parametrów rozkładu . . . 186
4.2. Wstępne przetwarzanie danych . . . 186
4.2.1. Brakujące obserwacje . . . 187
4.2.2. Normalizacja, skalowanie i transformacje nieliniowe . . . 191
4.3. Analiza wariancji, regresja liniowa i logistyczna . . . 195
4.3.1. Wprowadzenie do analizy wariancji . . . 196
4.3.2. Analiza jednoczynnikowa . . . 197
4.3.3. Analiza wieloczynnikowa . . . 206
4.3.4. Regresja . . . 211
4.3.5. Wprowadzenie do regresji logistycznej . . . 226
4.4. Testowanie . . . 243
4.4.1. Testowanie zgodności . . . 244
4.4.2. Testowanie hipotezy o równości parametrów położenia . 251
4.4.3. Testowanie hipotezy o równości parametrów skali . . . 256
4.4.4. Testowanie hipotez dotyczących wskaźnika struktury . . 258
4.4.5. Testy istotności zależności pomiędzy dwoma zmiennymi 260
4.4.6. Testowanie zbioru hipotez . . . 269
4.4.7. Rozkład statystyki testowej . . . 272
4.5. Bootstrap . . . 276
4.5.1. Rozkład i obciążenie estymatorów . . . 277
4.5.2. Testy bootstrapowe . . . 280
4.6. Analiza przeżycia . . . 282
4.6.1. Krzywa przeżycia . . . 282
4.6.2. Model Coxa . . . 284
4.7. Wybrane funkcje matematyczne . . . 287
4.7.1. Operacje na zbiorach . . . 287
4.7.2. Operacje arytmetyczne . . . 288
4.7.3. Wielomiany . . . 290
4.7.4. Bazy wielomianów ortogonalnych . . . 291
4.7.5. Szukanie maksimum/minimum/zer funkcji . . . 293
4.7.6. Rachunek różniczkowo–całkowy . . . 294

5. Graficzna prezentacja danych . . . 296
5.1. Znajdź siedem różnic . . . 297
5.2. Jak zapisać wykres do pliku . . . 298
5.3. Pakiet `lattice` . . . 301
5.3.1. Wprowadzenie . . . 301
5.3.2. Szablony dla wykresów . . . 301
5.3.3. Formuły i wybór zmiennych . . . 305
5.3.4. Panele i mechanizm warunkowania . . . 305
5.3.5. Mechanizm grupowania . . . 306

5.3.6. Legenda wykresu . . . 306
5.3.7. Atlas funkcji graficznych z pakietu `lattice` . . . 308
5.3.8. Więcej o panelach . . . 318
5.3.9. Motywy i parametry graficzne . . . 321
5.3.10. Zaawansowane opcje . . . 321
5.4. Pakiet `ggplot2` . . . 326
5.4.1. Wprowadzenie . . . 326
5.4.2. Warstwy wykresu . . . 327
5.4.3. Mapowanie zmiennych na atrybuty wykresu . . . 329
5.4.4. Geometria warstwy . . . 332
5.4.5. Statystyki i agregacje . . . 334
5.4.6. Mechanizm warunkowania . . . 335
5.4.7. Kontrola skal . . . 337
5.4.8. Układ współrzędnych i osie wykresu . . . 339
5.4.9. Motywy i kompozycje graficzne . . . 340
5.4.10. Pozycjonowanie wykresów na rysunku . . . 341
5.4.11. Obiekt klasy `gg` . . . 342
5.5. Pakiet graphics . . . 344
5.5.1. Wprowadzenie . . . 344
5.5.2. Funkcja `plot()` . . . 347
5.5.3. Funkcja `matplot()` . . . 348
5.5.4. Osie wykresu . . . 348
5.5.5. Legenda wykresu . . . 349
5.5.6. Funkcja `image()` . . . 350
5.5.7. Wyrażenia matematyczne . . . 351
5.5.8. Kolory . . . 353
5.5.9. Właściwości linii . . . 354
5.5.10. Właściwości punktów/symboli . . . 355
5.5.11. Atomowe, niskopoziomowe funkcje graficzne . . . 355
5.5.12. Tytuł, podtytuł i opisy osi wykresu . . . 356
5.5.13. Pozycjonowanie wykresu, wiele wykresów na rysunku . . . 360
5.5.14. Wykres słonecznikowy . . . 361
5.5.15. Wykresy kropkowe, dwu- i wielowymiarowe . . . 362
5.5.16. Wykres macierzy korelacji . . . 364
5.5.17. Wykres konturowy . . . 364
5.5.18. Wykres koniczyny . . . 366
5.5.19. Wielowymiarowy, jądrowy estymator gęstości . . . 366
5.5.20. Wykresy konturowe . . . 368
5.5.21. Wykres mapa ciepła . . . 368
5.5.22. Wykres profili obserwacji . . . 369
5.5.23. Parametry graficzne . . . 370

5.6. Pakiet `rCharts` 374
5.6.1. Wprowadzenie 374
5.6.2. Biblioteka `NVD3` 374
5.6.3. Biblioteka `Leaflet` 376
5.6.4. Inne szablony 378

Opis zbiorów danych 379

Skorowidz 384

Rozdział 1

Wprowadzenie

One of the greatest advantages of R: getting your work done better and in less time.
Frank Harrell, Biostatistics, Vanderbilt University

W tym rozdziale przybliżymy czytelnikowi podstawowe informacje związane z programem R. Rozdział jest przygotowany z myślą o osobach, które jeszcze nie zetknęły się z R, nie wiedzą, jak wygląda praca w tym języku, i być może nie są pewne, czy warto się go uczyć.

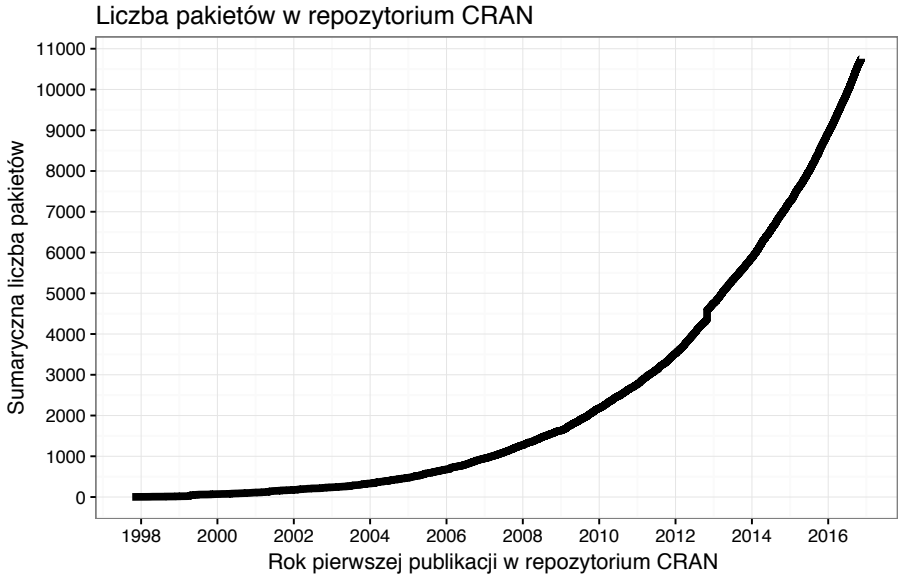
Zaczniemy więc od uzasadnienia, dlaczego warto. Następnie przedstawimy historię tego języka i kierunki rozwoju. Druga część tego rozdziału ilustruje, jak wygląda przykładowa praca z R. Na prostym przykładzie zaprezentujemy, jak pobierać dane, przetwarzać je i tworzyć wykresy. W trzecim rozdziale pokażemy, w jaki sposób przygotować sobie środowisko pracy, co i jak należy zainstalować. W ostatnim podrozdziale wymienimy miejsca, w których można znaleźć więcej informacji o tym języku.

1.1. Data science, czyli dlaczego warto poznać R

Data science to dynamicznie rosnąca dyscyplina, łącząca zagadnienia z obszaru analizy danych i programowania. Szybko wzrasta zapotrzebowanie na osoby potrafiące analizować nowo pojawiające się strumienie danych. Aby nadążyć za tym cyfrowym tsunami, potrzebne są dobre narzędzia do analizy danych. Takim narzędziem jest właśnie język R.

Języków programowania jest wiele, ale jeżeli chodzi o analizę danych, to z R możesz szybciej i lepiej zrobić więcej niż w jakimkolwiek innym języku. Pod adresem `http://bit.ly/2fXfWV2` (*Video Introduction to R* przygotowane przez Microsoft) znajduje się 90-sekundowe video-wprowadzenie do potencjału programu R.

Dlaczego? R powstał po to, by ułatwić profesjonalistom pracę z danymi. Język R jest dialektem języka S, który opracował John Chambers w Bell Labs

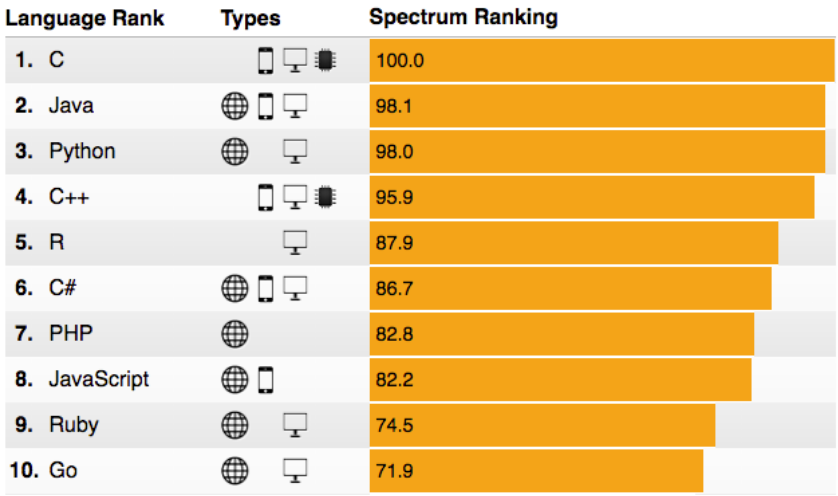


Rysunek 1.1. Liczba pakietów R w repozytorium CRAN, stan na listopad 2016

w okolicach roku 1976. Język S stworzono z myślą o interaktywnej analizie danych. Wiele rozwiązań zostało wprowadzonych do tego języka po to, by ułatwić pracę z danymi.

Pierwsza wersja R została napisana w roku 1991 przez Roberta Gentlemana i Rossa Ihakę (znanych jako R&R), pracujących na Wydziale Statystyki Uniwersytetu w Auckland. Pakiet R początkowo służył jako pomoc dydaktyczna do uczenia statystyki na tym uniwersytecie. Jednocześnie, ponieważ był to projekt otwarty, bardzo szybko zyskiwał na popularności. Od roku 1997 rozwojem R kierował zespół ponad dwudziestu osób, tworzących tzw. *core team*. W zespole tym znaleźli się eksperci z różnych dziedzin (statystyki, matematyki, metod numerycznych oraz szeroko pojętej informatyki) z całego świata. Liczba osób rozwijających R szybko rosła, a obecnie rozwojem projektu kieruje fundacja „The R Foundation for Statistical Computing", licząca setki aktywnych uczestników. Ponadto w rozwój R mają wkład tysiące osób z całego świata udostępniające własne pakiety z funkcjami do najróżniejszych zastosowań. Liczba bibliotek dostępnych dla użytkowników R szybko rośnie. W roku 2016 przekroczyła liczbę 10 000 pakietów w głównym repozytorium CRAN (zobacz rysunek 1.1). Do tej liczby trzeba doliczyć tysiące pakietów prywatnych lub umieszczonych w innych repozytoriach.

Program R jest projektem GNU opartym na licencji GNU GPL, co oznacza, iż jest zupełnie darmowy zarówno do zastosowań edukacyjnych, jak i biznesowych. Więcej o licencji GNU GPL można przeczytać pod adresem [53]. Platforma R wyposażona jest w świetną dokumentację, dostępną w postaci



Rysunek 1.2. Ranking IEEE dotyczący popularności języków, rok 2016

dokumentów PDF lub stron HTML. Dokumentacja w większości jest angielskojęzyczna, ale niektóre pliki pomocy mają swoje lokalizacje, w tym polską.

Od początku język R był tworzony i rozwijany z myślą o osobach zajmujących się analizą danych. Z tego też powodu przez informatyków często nie był traktowany jak pełnowartościowy język, ale raczej jak język domenowy (DSL, ang. *domain-specific language*). Z czasem jednak możliwości R rosły, pojawiło się coraz więcej rozwiązań wykraczających poza analizę danych i dziś R jest jednym z popularniejszych języków programowania.

Świadczyć może o tym ranking przygotowany na podstawie głosów członków komitetu IEEE (Institute of Electrical and Electronics Engineers). W tym rankingu R został w roku 2016 wymieniony na 5. miejscu wśród najpopularniejszych języków, wyprzedzając takie tuzy jak PHP czy JavaScript.

Jest wiele powodów takiej popularności języka R. Poniżej przedstawiamy te subiektywnie uznane za najważniejsze.

- Z uwagi na swoją elastyczność dzisiaj jest to podstawowy język wykorzystywany w nauczaniu statystyki i analizy danych, obecny na praktycznie każdej dobrej uczelni.
- Program R pozwala na tworzenie i upowszechnianie pakietów zawierających nowe funkcjonalności. Obecnie dostępnych jest przeszło 10 000 pakietów do różnorodnych zastosowań, np. `rgl` – do grafiki trójwymiarowej, `lima` – do analizy danych mikromacierzowych, `seqinr` – do analizy danych genomicznych, `psy` – z funkcjami statystycznymi powszechnie wykorzystywanymi w psychometrii, `geoR` – z funkcjami geostatystycznymi, `Sweave` – do generowania raportów w języku LaTeX i wiele, wiele innych. Każdy może napisać swój własny pakiet i udostępnić go innym osobom.

- Program R pozwala na wykonywanie funkcji z bibliotek napisanych w innych językach (C, C++, Fortran), a także istnieje możliwość wykonywania funkcji dostępnych w programie R z poziomu innych języków (Java, C, C++, pakiety Statistica, SAS i wiele innych). Dzięki temu możemy np. znaczną część programu napisać w Javie, a R wykorzystywać jako dużą zewnętrzną bibliotekę funkcji statystycznych.
- Program R jest w zupełności darmowy do wszelkich zastosowań zarówno prywatnych, naukowych, jak i komercyjnych. Również większość pakietów napisanych dla R jest darmowa i dostępna w ramach licencji GNU GPL lub GNU GPL 2.0.
- W programie R można wykonać wykresy o wysokiej jakości, co jest bardzo istotne przy prezentacji wyników. Wykresy te nawet przy domyślnych ustawieniach wyglądają znacznie lepiej od podobnych wykresów przygotowanych w innych pakietach.

Więcej o historii i kierunkach rozwoju języka R można przeczytać na stronie `http://bit.ly/2domniT` (Kurs „R Programming" na `coursera.org`).

1.2. Jak wygląda praca z programem R

Praca z programem R jest najczęściej interaktywna. Sam program R jest często kojarzony z konsolą, w której mruga kursor, służącą do wprowadzania poleceń i odczytywania wyników.

Aby oddać ten styl pracy, w tej książce prezentujemy przykłady instrukcji R wraz z wynikami, które te instrukcje generują. Zarówno instrukcje R, jak i ich wyniki są otoczone szarą ramką, aby łatwiej było je odnaleźć. Wyniki znajdują się w liniach oznaczonych podwójnym komentarzem `##`.

Przykładowo, funkcja `substr()` wycina z napisu podciąg o wskazanych współrzędnych. Po uruchomieniu programu R wystarczy wpisać do konsoli poniższą instrukcję, aby w tej samej konsoli pojawił się wynik. Tutaj wynikiem jest napis składający się z jednej litery `R`.

```
substr("Co jest supeR?", start = 13, stop = 13)
## [1] "R"
```

Pisząc o funkcjach, pakietach, elementach języka, stosujemy `pismo o stałej szerokości znaków`. Angielskie nazwy oznaczymy z kolei *kursywą*. Przy pierwszym wymienieniu nazwy funkcji z niestandardowego pakietu zaznaczamy również, w jakim pakiecie ta funkcja jest dostępna.

Zapis `ggplot2::qplot()` oznacza, że funkcja `qplot()` znajduje się w pakiecie `ggplot2`. Czasem funkcja o tej samej nazwie dostępna jest w różnych pakietach. Zatem wskazanie w jawny sposób, w którym pakiecie dana funkcja się znajduje, pozwala na jednoznaczne określenie, o którą funkcję chodzi. W sytuacji, gdy do użycia danej funkcji potrzeba zainstalować lub włączyć

niestandardowy pakiet, warto wiedzieć, w jakim pakiecie danej funkcji szukać. Na końcu tej książki znajduje się indeks funkcji, zarówno alfabetyczny, jak i w podziale na pakiety.

1.2.1. Przykład: Pozycja Polski w rankingu FIFA

Przyjrzyjmy się, jak wygląda przykładowa „poważna" praca z R na poniższym przykładzie.

- Wykorzystamy funkcje z pakietu `rvest`, aby odczytać bezpośrednio z Wikipedii dane o pozycji Polski w rankingu FIFA.
- Wykorzystamy funkcje z pakietów `tidyr` i `dplyr`, aby przekształcić wczytane dane do odpowiedniej postaci.
- Wykorzystamy funkcje z pakietu `ggplot2`, aby przedstawić graficznie wczytane dane.

Cały kod R, który wykonuje te trzy etapy, jest dosyć krótki i łatwo go prześledzić krok po kroku. Niektóre elementy mogą wydawać się znajome, inne mogą być zaskakujące. W kolejnych rozdziałach wyjaśnimy każdy z zaprezentowanych tutaj elementów.

Wszystkie instrukcje R przedstawiane w tej książce można pobrać ze strony `http://biecek.pl/R`.
Przedstawiane instrukcje R pozwalają na odtworzenie przedstawianych wyników. Jedynym odstępstwem od tej reguły są niektóre wykresy, które zostały zmodyfikowane aby zwiększyć ich czytelność w druku.

1.2.1.1. Wczytanie danych

Poniższy fragment kodu wczytuje do R tabelę z danymi ze strony *Reprezentacja Polski w piłce nożnej* z Wikipedii. Wykorzystujemy tutaj funkcje z pakietu `rvest`, który jest bardzo przydatny w pobieraniu danych ze stron internetowych.

Na interesującej nas stronie jest wiele tabel. Pobieramy wszystkie, a następnie wybieramy tę o 14 kolumnach. Po wczytaniu danych wyświetlamy pierwszych 6 wierszy.

```
library("rvest")
wikiPL <- "https://pl.wikipedia.org/wiki/Reprezentacja_Polski_w_pi%C5
    %82ce_no%C5%BCnej"
webpage <- read_html(wikiPL)
table_links <- html_nodes(webpage, '.wikitable')
tables <- html_table(table_links, fill=TRUE)
FIFA_table <- which(sapply(tables, ncol) == 14)
tab <- tables[[FIFA_table]]
```

```
head(tab)
##    Rok  I II III IV  V VI VII VIII IX  X XI XII Sr.
## 1 1993  -  -   -  -  -  -   -   20 22 23 26  28  24
## 2 1994 28 24  24 28 27 32  32   32 33 36 33  29  30
## 3 1995 29 32  32 34 36 29  32   28 33 27 33  33  32
## 4 1996 35 37  37 40 42 42  50   55 56 55 52  53  48
## 5 1997 53 56  56 52 57 57  51   53 50 45 47  48  52
## 6 1998 48 60  61 55 54 54  48   40 34 34 29  31  45
```

1.2.1.2. Transformacja danych

Po wczytaniu danych zazwyczaj kolejnym krokiem jest ich wyczyszczenie i przetransformowanie. Do tego celu mogą przydać się funkcje z pakietów `tidyr` i `dplyr`, omówimy je szczegółowo już niedługo, poniżej krótka ilustracja.

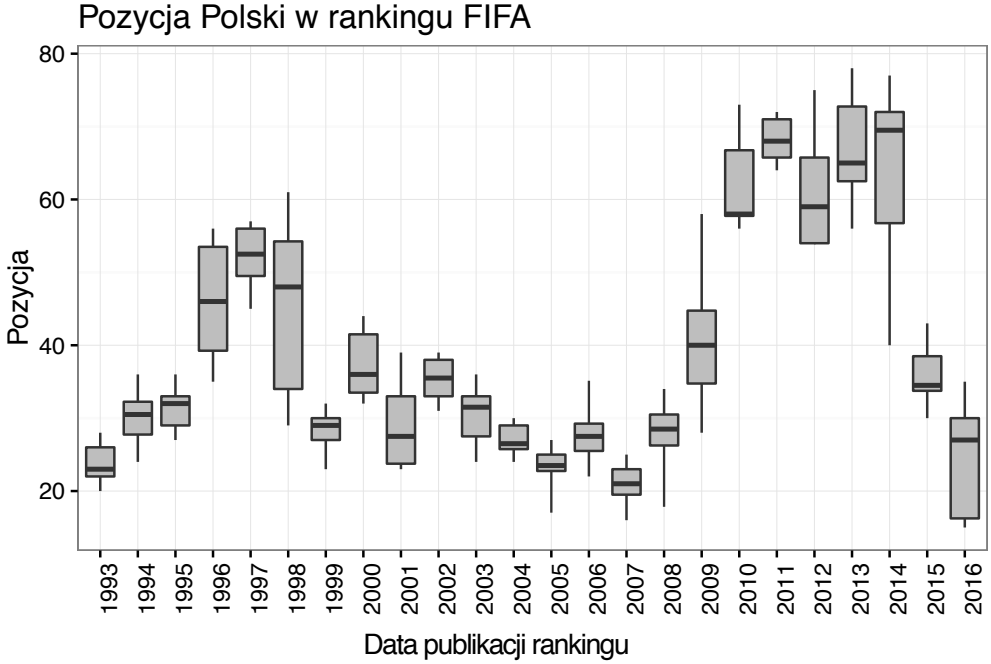
Zamieniamy dane z formatu szerokiego, w którym miesiące są w różnych kolumnach, na format wąski, w którym miesiące są w jednej kolumnie. Następnie zamieniamy rzymskie nazwy miesięcy na arabskie. Teraz dane są gotowe do prezentacji graficznej.

```
library("tidyr")
library("dplyr")
colnames(tab)[2:13] <- 1:12
data_long <- gather(tab[,1:13], Miesiac, Pozycja, -Rok)
data_long <- mutate(data_long,
                    Pozycja = as.numeric(Pozycja),
                    Miesiac = as.numeric(Miesiac))
head(data_long, 3)
##    Rok Miesiac Pozycja
## 1 1993       1      NA
## 2 1994       1      28
## 3 1995       1      29
```

1.2.1.3. Prezentacja graficzna danych

Wczytaliśmy dane, czas się im przyjrzeć. Do graficznej prezentacji danych wykorzystać można pakiet `ggplot2`. Poniżej przedstawiamy instrukcje, które wygenerują wykres z rysunku 1.3, tak zwany wykres pudełkowy. Przedstawia on minimalną, maksymalną, połówkową i ćwiartkową pozycję Polski w rankingu w każdym roku. Widzimy, w których latach były największe zmiany oraz kiedy byliśmy na najlepszych pozycjach.

```
library("ggplot2")
ggplot(data_long, aes(factor(Rok), Pozycja)) +
  geom_boxplot() + ggtitle("Pozycja Polski w rankingu FIFA")
```



Rysunek 1.3. Przykładowy wykres wykonany przy użyciu pakietu `ggplot2`. Przedstawiona jest pozycja Polski w rankingu FIFA w różnych latach. W ostatnich dwóch latach odnotowano znaczącą poprawę

1.3. Jak przygotować środowisko pracy

Należy wykonać trzy kroki, aby wygodnie pracować z programem R.

- Zainstalować podstawowe środowisko R, na które składa się interpreter i podstawowy zbiór pakietów. Już ten zestaw ma potężne możliwości. W większości przypadków wystarczające do analizy danych, rysowania wykresów i wykonywania innych typowych zadań.
- Zainstalować edytor RStudio, narzędzie, które bardzo ułatwia pracę z programem R. Nie jest to jedyny edytor dla R, ale z obecnych dzisiaj na rynku wydaje się najlepszym rozwiązaniem. Podobnie jak R również ten edytor w wersji podstawowej jest dostępny bezpłatnie.
- Doinstalować dodatkowe pakiety z przydatnymi funkcjami. Obecnie dostępnych jest ponad siedem tysięcy pakietów! Nie ma jednak potrzeby instalowania ich wszystkich od razu. Z reguły w miarę używania okazuje się, że przydałaby się nam jakaś dodatkowa funkcja z nowego pakietu. Dopiero wtedy warto taki pakiet doinstalować.

Poniżej znajduje się krótka informacja ułatwiająca przejście przez poszczególne etapy instalacji.

1.3.1. Instalacja podstawowego środowiska R

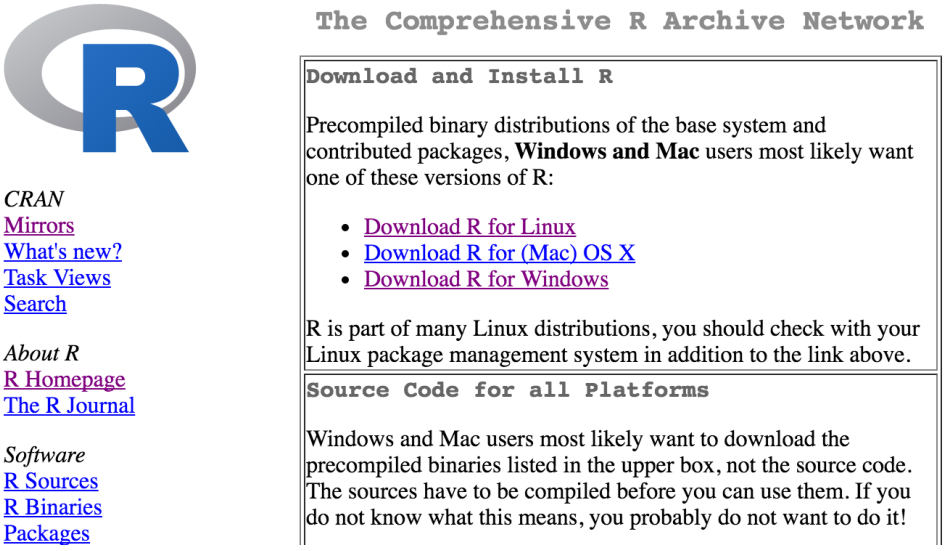
Dla większości systemów operacyjnych, w tym wszystkich dystrybucji Linuxa, Unixa, dla wersji Windows począwszy od Windows 95 i dla Mac OS, pakiet R jest dostępny w postaci źródłowej oraz binarnej.

W chwili pisania tego tekstu aktualna wersja R to 3.3.2. Co roku w kwietniu ukazuje się kolejna duża wersja 3.x, czyli niedługo będziemy cieszyć się wersją 3.4.0. W trakcie roku w miarę potrzeb wydawane są podwersje. Program R szybko się rozwija i warto przynajmniej raz do roku aktualizować go do najnowszej dostępnej wersji.

Aby zainstalować R, należy otworzyć stronę `https://cran.r-project.org/`, wybrać system operacyjny i pobrać plik binarny do instalacji. Instalacja sprowadza się do kliknięcia kilka razy przycisku *Next*. Można mieć zainstalowanych jednocześnie kilka różnych wersji R. Przechowywanie starszych wersji programu R może być przydatne, jeżeli chcemy mieć prostą możliwość odtworzenia w przyszłości dokładnie tych samych wyników.

W przypadku instalacji na nietypowych platformach można posiłkować się szczegółową instrukcją instalacji dostępną na stronie [49].

Wygodną właściwością środowiska R jest to, że można je uruchamiać bez instalowania. Można więc skopiować środowisko R na płytę CD, na pendrive lub dysk przenośny i uruchamiać na dowolnym komputerze bez potrzeby instalacji.



Rysunek 1.4. Strona `https://cran.r-project.org/`. Można z niej pobrać program R dla najpopularniejszych systemów operacyjnych

Trudno jest podać minimalne wymagania sprzętowe niezbędne do działania R. Wersję podstawową można bez problemu zainstalować na wieloletnich maszynach z 256 MB RAM, procesorami klasy Pentium lub wyższej i kilkudziesięcioma MB miejsca na dysku twardym. O ile jednak wersja podstawowa jest „lekka", to instalacja dodatkowych pakietów może wymagać kilku GB pamięci RAM lub miejsca na dysku. Szczególnie pakiety z dużymi danymi genomicznymi potrafią dużo „ważyć".

Osoby wykorzystujące program R do bardzo wymagających obliczeniowo analiz powinny raczej używać OS X-owej, linuksowej lub uniksowej wersji R. W tych systemach operacyjnych zarządzanie pamięcią jest wydajniejsze, przez co R działa (odrobinę) szybciej. W przypadku Uniksów dostępne są też dodatkowe narzędzia pozwalające na wykorzystanie wielowątkowości i innych mechanizmów systemowych (np. funkcji `fork()`).

1.3.2. Instalacja edytora RStudio

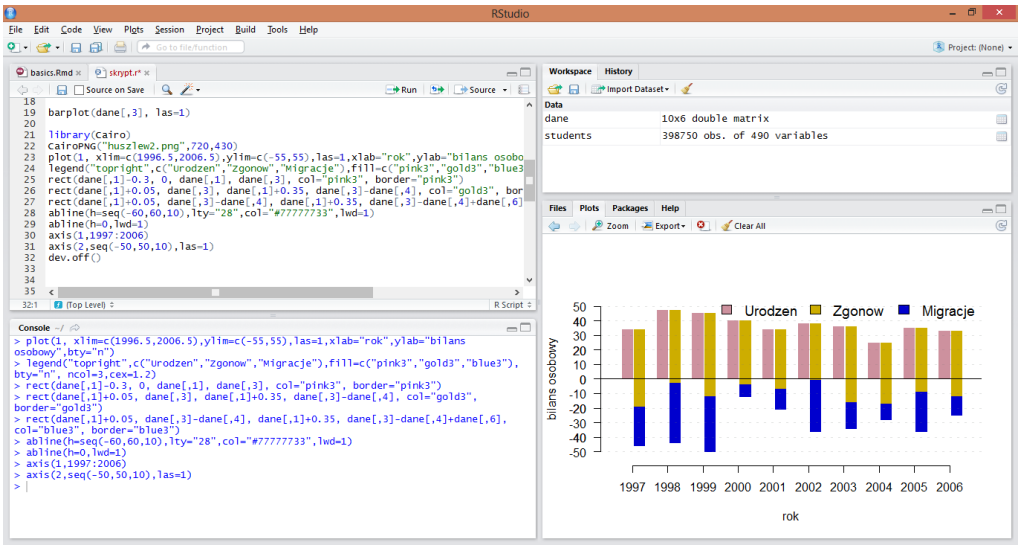
Można pracować z podstawowym programem R bez dodatkowego edytora, jednak praca z edytorem jest znacznie wygodniejsza, szczególnie jeżeli R to nasze podstawowe narzędzie pracy. Obecnie najlepszym edytorem do programu R jest RStudio – narzędzie opracowane przez firmę o tej samej nazwie. Ten edytor dostępny jest zarówno na platformę Windows, Linux, jak i OS X, wspiera on wiele zaawansowanych pakietów i funkcjonalności R.

RStudio jest narzędziem ułatwiającym pracę z R. Jest to edytor, manager wersji, narzędzie wspierające debugowanie, tworzenie pakietów, aplikacji czy raportów. Można żyć bez tej nakładki, ale co to za życie. Najnowszą wersję RStudio Desktop można pobrać ze strony `http://www.rstudio.com/`.

Przykładowy wygląd tego programu jest przedstawiony na rysunku 1.5.

Korzystanie z omawianego edytora jest bardzo intuicyjne. Do ciekawszych funkcji tego narzędzia należą następujące:

- możliwość automatycznego wysyłania całego skryptu lub fragmentu kodu do konsoli R. Po zaznaczeniu określonego fragmentu kodu i naciśnięciu `Ctrl-Enter` zadany fragment będzie wykonany w konsoli R,
- zarządzanie wieloma plikami/projektami,
- wsparcie dla pakietów `knitr`, `Sweave` i `shiny`, łatwa nawigacja po wstawkach kodu,
- praca z R Notebooks, interaktywna praca z raportami i możliwość uruchamiania i aktualizacji poszczególnych wstawek kodu,
- wyświetlanie obiektów (nazwa, klasa, wymiary) obecnych w przestrzeni nazw,



Rysunek 1.5. Przykładowy wygląd paneli edytora RStudio. Na czterech panelach widoczne są: okno pozwalające na edycję kodu, okno pozwalające na podgląd zmiennych załadowanych do globalnej przestrzeni nazw R, konsola programu R, w której możemy wpisywać bezpośrednio polecenia, oraz okno, w którym domyślnie wyświetlane są pliki pomocy oraz wykresy

- edytor danych, funkcji i innych obiektów R, po kliknięciu na nazwę zbioru danych obecnego w przestrzeni nazw mamy możliwość edycji tego obiektu, jest to często wygodniejszy sposób niż użycie funkcji `fix()` czy `edit()`,
- uproszczony sposób wczytywania danych z plików poprzez interfejs graficzny,
- podświetlanie słów kluczowych i funkcji,
- kontekstowe uzupełnianie nazw funkcji, zmiennych, właściwości, argumentów funkcji (gdy rozpoczynamy pisanie nazwy funkcji, zmiennej lub argumentu, pojawia się lista wszystkich słów, które pasują w danym kontekście),
- zwijanie/rozwijanie kodu funkcji i pętli,
- wsparcie dla tworzenia i testowania nowych pakietów,
- domykanie otwartych nawiasów, cudzysłowów, wraz z inteligentnym zaznaczaniem zawartości,
- inteligentne wstawianie wcięć połączone z rozpoznawaniem składni (czyli nowe wcięcie dodawane jest w pętlach, funkcjach itp.),
- interaktywny debugger.

Interesującym rozszerzeniem programu RStudio Desktop jest wersja serwerowa – RStudio Server. Po zainstalowaniu na serwerze można z niej korzystać zdalnie przez przeglądarkę internetową. W wersji płatnej możne z niej korzystać jednocześnie kilka osób.

Pracę z narzędziem RStudio bardzo ułatwi poznanie kilku podstawowych skrótów klawiszowych. Poniżej przedstawiam te moim zdaniem najciekawsze. Bardzo dobry opis programu RStudio wraz z listą wszystkich skrótów klawiszowych można pobrać ze strony `http://bit.ly/2d4B0Ix` (tzw. *RStudio IDE cheatsheets*). W poniższym opisie zapis `Ctrl/Cmd` oznacza `Ctrl` dla systemu Windows i Linux oraz `Cmd` dla systemu OS X.

- `Ctrl/Cmd+Shift+C`, komentowanie linii lub zbioru linii kodu,
- `Ctrl/Cmd+Enter`, wykonanie zaznaczonego kodu w konsoli,
- `Ctrl/Cmd+Up`, wyświetlanie historii komend,
- `Ctrl+L`, czyszczenie konsoli,
- `Ctrl+1-9`, przełączanie pomiędzy oknami aplikacji,
- `Tab`, uzupełnianie kodu, bardzo, bardzo użyteczne,
- `Ctrl/Cmd+Shift+K`, kompilacja obecnego dokumentu.

1.3.3. Instalacja dodatkowych pakietów

Po zainstalowaniu podstawowego środowiska R mamy już dostęp do wielu użytecznych funkcjonalności. Prawdziwa potęga kryje się jednak w tysiącach dodatkowych pakietów, w których znajdują się kolejne tysiące funkcji.

Aby skorzystać z dodatkowych pakietów, należy wykonać dwa kroki:

1. Zainstalować nowy pakiet na dysku. Ten krok wystarczy wykonać raz, skopiuje on niezbędne pliki do katalogu z pakietami. Ścieżkę do katalogu z pakietami można sprawdzić, używając polecenia `.libPaths()`.
2. Włączyć zainstalowany pakiet w działającej sesji `R`. Ten krok należy wykonać po każdym uruchomieniu środowiska `R`. Spowoduje on, że w aktualnej sesji będzie można korzystać z funkcji i zbiorów danych dostępnych w pakiecie.

Instalować pakiety można na kilka sposobów. Najczęściej pakiety instaluje się z oficjalnego repozytorium CRAN (The Comprehensive R Archive Network). Jeżeli pakiet, który nas interesuje, jest w tym repozytorium, to można go zainstalować funkcją `install.packages()`.

Przykładowo, poniższe polecenie zainstaluje pakiet `Przewodnik` wraz z pakietami zależnymi, bezpośrednio z repozytorium CRAN. Pakiet `Przewodnik` zawiera jako zależności wszystkie pakiety i zbiory danych, z których korzystamy w tej książce. Aby go zainstalować, wystarczy wpisać w konsoli programu R:

```
install.packages("Przewodnik")
```

lub wybrać z menu RStudio opcję `Tools/Install packages...`

Niektórych pakietów nie ma w repozytorium CRAN lub są dostępne, ale w starszej wersji. Przykładowo, jednym z wymogów stawianych pakietom w CRAN jest ograniczenie wielkości pakietu do 10 MB. Z uwagi na to ogra-

niczenie większe rozmiarowo pakiety są przechowywane na innych repozytoriach, najczęściej w repozytorium GitHub. Do instalacji pakietów z GitHuba można wykorzystać funkcję `devtools::install_github()`.

Na przykład, aby zainstalować najnowszą wersję pakietu `Przewodnik` z repozytorium użytkownika `pbiecek` z repozytorium GitHub, można użyć instrukcji:

```
devtools::install_github("pbiecek/PrzewodnikPakiet")
```

Poza tymi dwoma repozytoriami istnieje też wiele innych specjalistycznych. Przykładowo wiele pakietów wykorzystywanych w analizach bioinformatycznych znajduje się w repozytorium `Bioconductor`. Więcej o tym repozytorium na stronie `https://www.bioconductor.org/`.

Po zainstalowaniu nowego pakietu pliki z danymi, funkcjami i plikami pomocy znajdą się na dysku twardym komputera. Aby jednak móc skorzystać z wybranych funkcji, należy przed pierwszym użyciem załadować (włączyć) odpowiedni pakiet. Po każdym uruchomieniu platformy R ładowane są pakiety podstawowe, takie jak: `base`, `graphics`, `stats` itp. Aby skorzystać z dodatkowych funkcji lub zbiorów danych, należy załadować (włączyć) pakiet, w którym się one znajdują (zakładamy, że pakiety te zostały już zainstalowane). Pakiety włącza się poleceniem `library()`.

Poniższa instrukcja włącza pakiet `devtools`. Po włączeniu pakietu można używać funkcji z tego pakietu bez konieczności dodawania prefixu `devtools::`.

```
library("devtools")
```

Jak już pisaliśmy, w repozytorium CRAN dostępnych jest obecnie ponad 10 000 pakietów. W tak dużym zbiorze trudno czasem odnaleźć pakiet z interesującą nas funkcjonalnością. Dlatego przedstawiając nowe funkcje, będziemy korzystać z notacji `nazwaPakietu::nazwaFunkcji()`. Zapis `ggplot2::qplot()` oznacza, że funkcja `qplot()` znajduje się w pakiecie `ggplot2`. W skorowidzu, znajdującym się na końcu książki, funkcje są wymienione zarówno w kolejności alfabetycznej, jak i po pogrupowaniu w pakiety.

Jeżeli znamy nazwę funkcji i chcemy dowiedzieć się, w jakim pakiecie się ona znajduje, a nie mamy tej książki pod ręką, to możemy wykorzystać funkcję `help.search()`. Przeszuka ona wszystkie zainstalowane pakiety w poszukiwaniu funkcji o wskazanej nazwie lub funkcji, w których opisie wystąpiło zadane słowo kluczowe. Więcej o tej funkcji i innych sposobach wyszukiwania informacji o funkcjach piszemy w podrozdziale 1.4.

Po załadowaniu odpowiedniego pakietu możemy korzystać z dostępnych w nim funkcji, podając ich nazwę. Możemy też wprost wskazać, z którego pakietu funkcję chcemy uruchomić, co jest przydatne, gdy funkcje o identycznych nazwach znajdują się w kilku załadowanych pakietach. Przykładowo zarówno w pakiecie `epitools`, jak i `vcd` znajduje się funkcja `oddsratio()`, przy

czym każda z nich ma inne działanie. Aby wskazać, z którego pakietu chcemy wybrać funkcję, należy użyć operatora :: lub :::. Operator :: pozwala na odnoszenie się wyłącznie do publicznych funkcji z pakietu, podczas gdy operator ::: umożliwia odwoływanie się również do funkcji prywatnych.

Oba poniższe wywołania dotyczą funkcji `seq()` z pakietu `base`. Drugi sposób jest przydatny, gdy występuje kolizja nazw funkcji z różnych pakietów.

```
seq(10)
base::seq(10)
```

Jeżeli nie zastosujemy operatora `::`, a funkcja o danej nazwie występuje w kilku różnych pakietach, to program R użyje funkcji z ostatnio załadowanego pakietu.

1.4. Gdzie szukać dalszych informacji

W dalszej części tej książki omawiamy biblioteki do transformacji danych, modelowania statystycznego oraz wizualizacji. Z pewnością jednak pojawią się pytania lub wątki, które nie będą tutaj opisane. Gdzie szukać dodatkowych informacji?

Najłatwiej i najszybciej jest zapytać kogoś, kto to wie i chce nam podpowiedzieć.

Tak się składa, że wokół programu R działa duże środowisko entuzjastów, organizujących cykliczne spotkania z referatami. Obecnie trzy największe w Polsce środowiska pasjonatów R działają w trzech dużych miastach. Spotkania odbywają się mniej więcej raz w miesiącu.

- SER: Warszawskie Spotkania Entuzjastów R, `http://bit.ly/2epbNb9`,
- eRka: Entuzjastów R Krakowska Alternatywa, `http://bit.ly/2fJ5N9o`,
- PAZUR: Poznań R User Group, `http://bit.ly/2fKnTJE`.

Silne grupy użytkowników R działają również we Wrocławiu, Łodzi i innych miastach, wymieniając się informacjami w ramach spotkań data science.

W sytuacji, gdy nie mamy takiej osoby pod ręką, program R oferuje bogaty system pomocy. Pierwszym jej źródłem są wbudowane funkcje R ułatwiające wyszukiwanie informacji. Oto lista najbardziej przydatnych:

- Funkcja `help()` wyświetla stronę powitalną systemu pomocy R. Na tej stronie opisane są szczegółowo wymienione poniżej funkcje.
- Funkcje `help("sort")` i `?sort` wyświetlają stronę z pomocą dotyczącą funkcji o nazwie `sort`. Zobacz też przykład przedstawiony poniżej. Format opisów funkcji jest ujednolicony, tak aby łatwo było z nich korzystać. W RStudio można też użyć przycisku `F1` aby aktywować pomoc dla funkcji.
  Kolejne sekcje pomocy zawierają: zwięzły opis funkcji (sekcja `Description`), deklaracje funkcji (sekcja `Usage`), objaśnienie poszczególnych argumentów

(sekcja `Arguments`), szczegółowy opis funkcji (sekcja `Details`), literaturę (sekcja `References`), odnośniki do innych funkcji (sekcja `See Also`) oraz przykłady użycia (sekcja `Examples`). Jeżeli podamy argument `package`, to uzyskamy pomoc dotyczącą wskazanego pakietu.
Przykładowo polecenie `help(package=MASS)` wyświetla opis pakietu `MASS`.

- Funkcja `args("nazwaFunkcji")` wyświetla listę argumentów `nazwaFunkcji`.
- Funkcje `apropos("regression")` i `find("regression")` wypisują listę funkcji (oraz obiektów), które w swojej nazwie mają podciąg `regression`.
- Funkcja `example("plot")` uruchamia skrypt z przykładami użycia funkcji `plot`. Dzięki przykładom można szybko zobaczyć, jak korzystać z danej funkcji, a także jakich wyników należy się spodziewać.
- Funkcja `help.search("slowoKluczowe")` przegląda opisy funkcji znajdujących się w zainstalowanych pakietach i wyświetla te funkcje, w których znaleziono wskazane `slowoKluczowe`. W tym przypadku `slowoKluczowe` może oznaczać również kilka słów lub zwrot. Lista wyników zawiera też informację, w którym pakiecie znajdują się znalezione funkcje.

Poniżej przedstawiona jest przykładowa sesja w programie R. Szukamy informacji o funkcji `plot()` oraz o funkcjach do testowania hipotez. W pierwszej linii wyświetlamy pomoc dotyczącą funkcji plot, następnie przykłady jej użycia. Kolejna linia wyświetla funkcje ze słowem `"test"` w nazwie, a ostatnia wyświetla nazwy funkcji ze zwrotem `"normality test"` w opisie.

```
?plot
example(plot)
apropos("test")
help.search("normality test")
```

Funkcje przedstawione powyżej wyszukują informacje na zadany temat, przeglądając pakiety, które są już zainstalowane na komputerze. Jeżeli to okaże się niewystarczające (a może się zdarzyć, że nie mamy zainstalowanego pakietu, w którym znajduje się potencjalnie interesująca nas funkcja), to możemy skorzystać z zasobów dostępnych w Internecie. W szczególności warto wiedzieć, gdzie znaleźć poniższe pomoce:

- Internetowy kurs w języku polskim **Pogromcy Danych**, dostępny pod adresem `http://pogromcydanych.icm.edu.pl`. Kurs podzielony jest na kilkanaście krótkich bloków z zadaniami pozwalającymi na sprawdzenie poziomu opanowania omawianego materiału.
- Forum *StackExchange*, z licznymi pytaniami i setkami osób chętnych, by na te pytania odpowiedzieć. Forum często pojawia się w wynikach wyszukiwarki Google. Jest dostępne pod adresem `http://stackexchange.com/`.
- Poradniki (podręczniki, ang. *manuals*) poświęcone różnym aspektom programowania w R lub analizy danych w programie R. Dostępne są bezpośrednio z menu `Help` w programie R oraz w Internecie pod adresem

http://cran.r-project.org/manuals.html.

- R-bloggers, czyli agregator blogów poświęconych programowi R, często źródło wielu ciekawych informacji: http://www.r-bloggers.com.
- Książki poświęcone pakietowi R oraz analizie danych z użyciem tego pakietu. Aktualizowana lista książek na ten temat znajduje się pod adresem http://www.r-project.org/doc/bib/R-books.html.
- Portale z pytaniami i odpowiedziami dotyczącymi statystyki, programowania. Np. http://stats.stackexchange.com (pytania dotyczące statystyki) lub http://stackoverflow.com/questions/tagged/r (pytania dotyczące programowania).

W Internecie można znaleźć też wiele materiałów w języku polskim. Jednym z pierwszych jest *Wprowadzenie do środowiska R* Łukasza Komsty [17], a co roku przybywa wiele kolejnych. Aktualna lista polskojęzycznych pozycji znajduje się na stronie Wikipedii https://pl.wikipedia.org/wiki/R.

Bardzo dobrą książką do nauki języka R jest książka Johna Chambersa [5], a do nauki funkcji statystycznych dostępnych w programie R polecamy pozycję Briana Everitta [8]. Obie one dotyczą języka S, jednak z punktu widzenia użytkownika jest on prawie identyczny z R. Uzupełnieniem licznych pozycji książkowych jest olbrzymia liczba stron internetowych oraz dokumentów elektronicznych szczegółowo przedstawiających rozmaite aspekty programu i języka R. Pod koniec roku 2007 ukazała się bardzo obszerna i godna polecenia książka Michaela Crawleya [6] przedstawiająca zarówno język R, jak i wiele procedur statystycznych zaimplementowanych w programie R. Pojawiają się też i będą się pojawiały liczne książki poświęcone specjalistycznym zastosowaniom programu R, jak np. świetna pozycja przygotowana przez Paula Murrella poświęcona grafice [21], książka autorstwa Juliana Farawaya poświęcona modelom liniowym [10] czy kolejna pozycja Briana Everitta przedstawiająca podstawy statystyki [9]. Wydawnictwo Springer ma w ofercie ponad 45 książek o programie R wydanych w serii Use R!, każda z tych pozycji traktuje o jednym wybranym zagadnieniu, takim jak analiza danych przestrzennych, wizualizacja danych, analiza danych socjologicznych, genetycznych itp.

Rozdział 2

Podstawy pracy z R

W tym rozdziale przedstawiamy podstawowe informacje związane z pracą w programie R. Pierwsze cztery podrozdziały wprowadzają do tematu wczytywania danych z plików tekstowych, omawiają kluczowe struktury danych w programie R, przedstawiają wybrane statystyki opisowe, zarówno liczbowe, jak i graficzne. Temat wczytywania danych wraca ponownie w podrozdziale 2.6, w którym pokazujemy, jak wczytywać i zapisywać dane, wykorzystując najróżniejsze formaty.

Następnie omawiane są pakiet `dplyr` i `tidyr`, dwa pakiety pozwalające na przeprowadzenie większości transformacji i agregacji danych, takich jak filtrowanie, grupowanie, wyznaczanie agregatów.

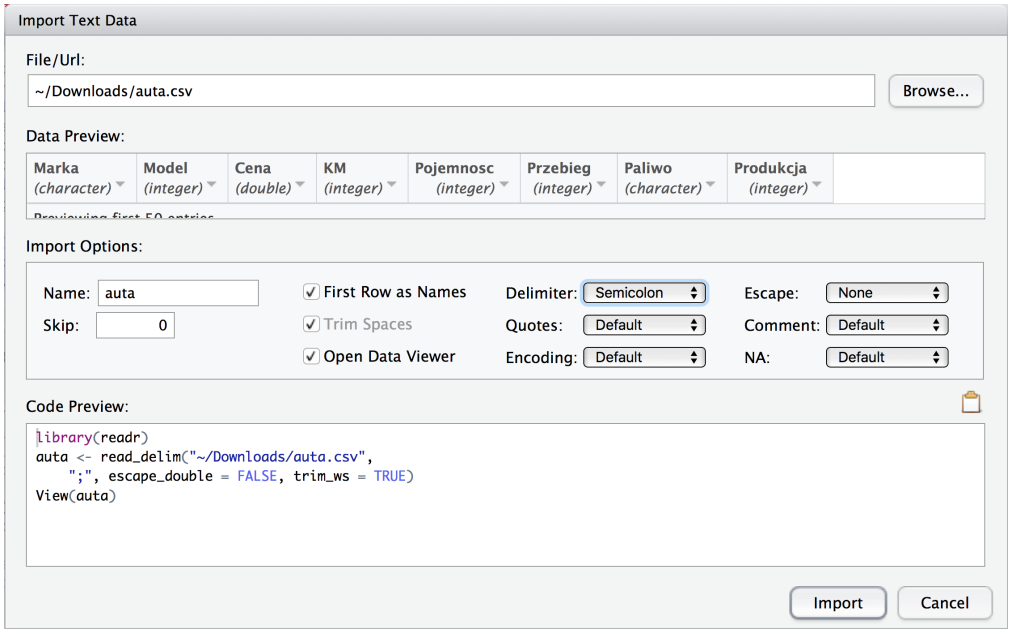
Ostatnim podrozdziałem jest omówienie zagadnień związanych z tworzeniem automatycznych raportów. Pokazujemy, jak prosto jest wykorzystać pakiet `knitr` lub program RStudio, aby błyskawicznie wykonać czytelny i łatwy do odtworzenia raport.

2.1. Wczytywanie danych

Dwa najpopularniejsze formaty przechowywania danych to pliki tekstowe i pliki w formacie Excel/OpenOffice. Korzystając z programu RStudio, oba te formaty można wczytać zaledwie kilkoma kliknięciami. Wystarczy wybrać opcję `File/Import Dataset`, aby otworzyć graficzny interfejs do importu danych – zobacz rysunek 2.1.

W przypadku niestandardowego formatowania pliku do dyspozycji mamy kilka kontrolek pozwalających na dookreślenie, co jest separatorem, czy w pierwszej linii są nazwy zmiennych itp.

Jeżeli nie korzystamy z programu RStudio, to do wczytywania danych z plików tekstowych możemy użyć funkcji `read.table()`. Zilustrujmy użycie tej funkcji na poniższym przykładzie.



Rysunek 2.1. Przykładowe okno programu RStudio do importu danych za pomocą graficznego interfejsu

W pliku `http://www.biecek.pl/R/auta.csv` znajdują się dane dotyczące 2400 ofert sprzedaży samochodów różnych marek. Dane te są zapisane w pliku o rozszerzeniu `csv`. Z każdą ofertą związanych jest 8 parametrów, w pliku tekstowym są one rozdzielone średnikami.

Fragment tego pliku przedstawiony jest poniżej.

```
"Marka";"Model";"Cena";"KM";"Pojemnosc";"Przebieg";"Paliwo";"
    Produkcja"
"Peugeot";"206";8799;60;1100;85000;"benzyna";2003
"Peugeot";"206";15500;60;1124;114000;"benzyna";2005
"Peugeot";"206";11900;90;1997;215000;"diesel";2003
"Peugeot";"206";10999;70;1868;165000;"diesel";2003
"Peugeot";"206";11900;70;1398;146000;"diesel";2005
"Peugeot";"206";19900;70;1398;86400;"diesel";2006
...
```

Pierwszym argumentem funkcji `read.table()` jest ścieżka do pliku z danymi. Plik może znajdować się na dysku lub w Internecie.

Następne dwa argumenty w poniższym wywołaniu określają, że wartości w kolejnych kolumnach są rozdzielone średnikami, a pierwszy wiersz zawiera nazwy zmiennych.

Wynik funkcji `read.table()`, czyli tabela z danymi, jest przypisany do symbolu `auta` za pomocą operatora `<-`.

```
auta <- read.table("http://www.biecek.pl/R/auta.csv",
                   sep = ";", header = TRUE)
head(auta)
##     Marka Model  Cena KM Pojemnosc Przebieg  Paliwo Produkcja
## 1 Peugeot   206  8799 60      1100    85000 benzyna      2003
## 2 Peugeot   206 15500 60      1124   114000 benzyna      2005
## 3 Peugeot   206 11900 90      1997   215000  diesel      2003
## 4 Peugeot   206 10999 70      1868   165000  diesel      2003
## 5 Peugeot   206 11900 70      1398   146000  diesel      2005
## 6 Peugeot   206 19900 70      1398    86400  diesel      2006
```

Funkcja `head()` wyświetla pierwsze 6 wierszy ze zbioru danych. Wyświetlenie nagłówka przydaje się, by zorientować się, co jest w tabeli.

W rozdziale 2.6 omawiamy więcej metod wczytywania danych z najróżniejszych formatów.

Gdy już uda nam się wczytać dane, czas, aby z nimi popracować.

2.2. Struktury danych

Dane, z którymi będziemy pracować, mają najczęściej format tabeli lub wektora wartości. Jedną z podstawowych operacji na tabelach oraz wektorach jest wybieranie podzbioru wierszy, kolumn lub wartości. W tym rozdziale opisujemy, w jaki sposób odwoływać się do elementów tabeli lub wektorów. Więcej o technikach przetwarzania danych można przeczytać w rozdziale 2.5.

2.2.1. Wektory

Podstawową strukturą danych w programie R są wektory, a więc sekwencje wartości tego samego typu. Sekwencje liczb, napisów, wartości logicznych. Dla programu R nawet jedna liczba jest wektorem, tyle że jednoelementowym.

Wektory można budować z mniejszych wektorów przy użyciu funkcji `c()`.

```
c(2, 3, 5, 7, 11, 13, 17)
## [1]  2  3  5  7 11 13 17
```

Szczególną grupą wektorów są sekwencje kolejnych liczb. Takie sekwencje najłatwiej tworzyć operatorem `:`.

```
-3:3
## [1] -3 -2 -1  0  1  2  3
```

Jeżeli potrzebujemy sekwencji liczb, ale z krokiem innym niż 1, to najwygodniej jest użyć funkcji `seq()`.

```
seq(from = 0, to = 100, by = 11)
## [1]  0 11 22 33 44 55 66 77 88 99
```

Wiele użytecznych wektorów jest dostępnych w podstawowych pakietach. Przykładami są nazwy miesięcy czy kolejne litery.

```
month.name
##  [1] "January"   "February"  "March"     "April"     "May"
##  [6] "June"      "July"      "August"    "September" "October"
## [11] "November"  "December"
LETTERS
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
## [15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

2.2.1.1. Indeksowanie wektorów

Aby odwołać się do elementów wektora, należy użyć operatora `[]`, wewnątrz którego podaje się wektor indeksów do odczytania. Elementy wektora numerowane są od liczby `1`. Najłatwiej będzie nam przedstawić indeksowanie na przykładzie wybierania części liter z wektora `LETTERS`.

```
LETTERS[ 5:10 ]
## [1] "E" "F" "G" "H" "I" "J"
LETTERS[ c(1, 2, 9:14) ]
## [1] "A" "B" "I" "J" "K" "L" "M" "N"
```

Funkcja `length()` wyznacza liczbę elementów w wektorze. Jest ona przydatna, np. gdy chcemy wybrać co drugi element wektora. Poniżej korzystamy z funkcji `seq()`, aby zbudować sekwencję indeksów złożoną z co drugiej litery wektora `LETTERS`.

```
co_drugi <- seq(from = 1, to = length(LETTERS), by = 2)
LETTERS[ co_drugi ]
## [1] "A" "C" "E" "G" "I" "K" "M" "O" "Q" "S" "U" "W" "Y"
```

Jako indeksy można podawać też ujemne liczby. W ten sposób wskazuje się wszystkie wartości *poza* określonymi indeksami.

```
month.name[ -(5:9) ]
## [1] "January"  "February" "March"    "April"    "October"
## [6] "November" "December"
```

Elementy wektora mogą być nazywane. Nazwy są wskazywane przy konstrukcji wektora funkcją `c()`, lub mogą być dodawane funkcją `names()`. Jeżeli wektor zawiera nazwy, to do jego elementów można się odwoływać, podając zamiast indeksów nazwy.

```
wartosc <- c(pion = 1, skoczek = 3, goniec = 3,
             wieza = 5, hetman = 9, krol = Inf)
wartosc[ c("goniec", "wieza") ]
## goniec  wieza
##      3      5
```

Celem indeksowania nie zawsze jest wybieranie podwektora. Równie często chodzi o zmianę kolejności elementów w wektorze. Przykładowo, poniżej odwracamy kolejność elementów, podając nowy zbiór indeksów.

```
wartosc[ 6:1 ]
## krol  hetman   wieza  goniec skoczek    pion
##  Inf       9       5       3       3       1
```

Innym przydatnym sposobem wskazywania podciągu elementów wektora jest korzystanie z wartości logicznych. Na poniższym przykładzie wybieramy wszystkie elementy wektora, które mają wartość mniejszą od 6.

```
wartosc[ wartosc < 6 ]
## pion skoczek  goniec   wieza
##    1       3       3       5
```

2.2.1.2. Zmiana elementów wektora

Indeksowanie wektorów jest też przydatne, gdy chcemy zmienić tylko niektóre elementy. Można wybrać podzbiór elementów wektora, a następnie przypisać do niego wartości operatorem <-.

W poniższym przykładzie zmieniamy wartość czwartego i piątego elementu wektora `wartosc`.

```
wartosc[ c(4,5) ] <- c(6,7)
wartosc
## pion skoczek  goniec   wieza  hetman    krol
##    1       3       3       6       7     Inf
```

Aby dodać nowe elementy do wektora, można wykorzystać funkcję `c()`.

```
(wartosc <- c(wartosc, nowa_bierka = 5))
## pion skoczek  goniec   wieza  hetman    krol nowa_bierka
##    1       3       3       6       7     Inf           5
```

2.2.2. Ramki danych

W analizie danych najczęściej wykorzystywaną strukturą danych jest ramka danych. Każda ramka to zbiór kolumn (czyli zmiennych), każda kolumna to wektor o tej samej długości. Różne kolumny mogą mieć wartości różnych typów.

Omówimy pracę z ramkami na przykładzie niewielkiego zbioru danych `koty_ptaki`. W siedmiu kolumnach przedstawione są różne charakterystyki wybranych kotów i ptaków.

```
library("Przewodnik")
head(koty_ptaki)
```

```
##   gatunek waga dlugosc predkosc habitat zywotnosc druzyna
## 1  Tygrys  300     2.5       60    Azja        25     Kot
## 2     Lew  200     2.0       80  Afryka        29     Kot
## 3  Jaguar  100     1.7       90 Ameryka        15     Kot
## 4    Puma   80     1.7       70 Ameryka        13     Kot
## 5 Leopard   70     1.4       85    Azja        21     Kot
## 6  Gepard   60     1.4      115  Afryka        12     Kot
```

2.2.2.1. Indeksowanie ramek danych

Ramka danych ma strukturę dwuwymiarowej tabeli. Aby określać jej elementy, należy podać współrzędne i wierszy, i kolumn. W tym celu można wykorzystać operatory `[]` lub `$` (ten omawiamy pod koniec podrozdziału).

Używając operatora `[]`, należy wskazać osobno indeksy wierszy i indeksy kolumn, rozdzielając je znakiem `","` (przecinka). Jeżeli indeksy wierszy lub kolumn nie zostaną wskazane, to wybrane będą wszystkie wiersze/kolumny.

W poniższym przykładzie wybieramy trzy wiersze z ramki `koty_ptaki`.

```
koty_ptaki[ c(1, 3, 5) , ]
##   gatunek waga dlugosc predkosc habitat zywotnosc druzyna
## 1  Tygrys  300     2.5       60    Azja        25     Kot
## 3  Jaguar  100     1.7       90 Ameryka        15     Kot
## 5 Leopard   70     1.4       85    Azja        21     Kot
```

Możemy jednocześnie indeksować i wiersze, i kolumny.

```
koty_ptaki[ c(1, 3, 5) , 2:5]
##   waga dlugosc predkosc habitat
## 1  300     2.5       60    Azja
## 3  100     1.7       90 Ameryka
## 5   70     1.4       85    Azja
```

Każdy wymiar można indeksować zgodnie z tymi samymi zasadami co indeksowanie wektorów. A więc indeksami mogą być warunki logiczne, można odwoływać się do nazw lub używać ujemnych indeksów.

W poniższym przykładzie wybieramy tylko zwierzęta rozwijające większą prędkość niż 100 km/h. Dla każdego wiersza spełniającego ten warunek wyświetlamy gatunek, prędkość i długość zwierzaka.

```
koty_ptaki[ koty_ptaki,"predkosc"] > 100,
            c("gatunek", "predkosc", "dlugosc")]
##           gatunek predkosc dlugosc
## 6          Gepard      115     1.4
## 8          Jerzyk      170     0.2
## 10  Orzel przedni      160     0.9
## 11 Sokol wedrowny      110     0.5
## 13       Albatros      120     0.8
```

Poszczególne kolumny ramki danych to wektory. Jeżeli odwołamy się do jednej kolumny, to jako wynik otrzymamy już nie ramkę danych, ale wektor. Jest to bardzo użyteczna właściwość, skracająca w wielu miejscach zapis. Są jednak sytuacje, w których jest to przyczyną błędów, więc należy zachować ostrożność gdy wybiera się tylko jedną kolumnę.

Wybrać kolumnę możemy przez wskazanie jej numeru lub nazwy. Poniższe polecenie ma ten sam efekt co `koty_ptaki[,4]`.

```
koty_ptaki[, "predkosc"]
## [1]  60  80  90  70  85 115  65 170  70 160 110 100 120
```

Gdy chcemy wybrać tylko jedną kolumnę z ramki, to możemy również skorzystać z operatora `$`. Skraca on zapis o całe 4 znaki, a jednocześnie łatwiej zauważyć, że wynik jest z pewnością wektorem.

Po operatorze `$` możemy podać całą nazwę zmiennej lub jej część. Jeżeli podamy część, to wynikiem będzie kolumna o nazwie rozpoczynającej się od wskazanego napisu.

```
koty_ptaki$predkosc
## [1]  60  80  90  70  85 115  65 170  70 160 110 100 120
```

Operator `$` jest też przydatny, gdy chcemy do ramki danych dodać nową kolumnę. Można za jego pomocą dokleić na koniec ramki danych nową kolumnę o wskazanej nazwie. Zmieniamy jednostkę, pamiętając, że 1 mila/h = 1.6 km/h.

```
koty_ptaki$predkosc_mile <- koty_ptaki$predkosc * 1.6
head(koty_ptaki, 2)
##   gatunek waga dlugosc predkosc habitat druzyna predkosc_mile
## 1  Tygrys  300     2.5       60    Azja     Kot            96
## 2     Lew  200     2.0       80  Afryka     Kot           128
```

Innym sposobem na dodanie kolumn jest skorzystanie z funkcji `cbind()` lub `data.frame()`.

Funkcja `rbind()` skleja wierszami dwie wskazane ramki danych, a tym samym pozwala na dodanie nowych wierszy. W pakiecie `dplyr` dostępne są szybsze wersje obu tych funkcji, o nazwach `bind_cols()` i `bind_rows()`.

2.2.3. Listy

Trzecią z najpopularniejszych struktur danych wykorzystywanych w programie R są listy. Lista, podobnie jak wektor, to sekwencja wartości. W przeciwieństwie do wektora elementy listy mogą być różnych typów. Elementem listy może być inna złożona struktura, taka jak ramka danych czy inna lista.

Listy tworzy się funkcją `list()`. Na poniższym przykładzie tworzymy listę wektora liczb, liter i wartości logicznych.

```
trojka <- list(liczby = 1:5, litery = LETTERS, logika = c(TRUE, TRUE,
    TRUE, FALSE))
trojka
## $liczby
## [1] 1 2 3 4 5
##
## $litery
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"
## [15] "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
##
## $logika
## [1]  TRUE  TRUE  TRUE FALSE
```

Do elementów listy można odwołać się operatorem `[[]]`. Wskazać można tylko jeden element listy.

Jeżeli lista jest nazwana, to można się do jej elementów odwoływać przez nazwy operatorem `$`, podobnie jak dla ramek danych. Ramka danych jest wewnętrznie reprezentowana jako lista wektorów o tej samej długości.

```
trojka[[2]]
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "
##  P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
trojka[["logika"]]
## [1]  TRUE  TRUE  TRUE FALSE
trojka$liczby
## [1] 1 2 3 4 5
```

Listy są bardzo przydatne, gdy mamy do przetworzenia wiele zbiorów danych lub chcemy wygenerować wiele modeli. Zarówno zbiory danych, jak i modele mogą być elementami listy.

Program R posiada wiele przydatnych funkcji do pracy z listami. Omawiamy je w rozdziale 3.9.1.

2.3. Statystyki opisowe

Program R jest głównie wykorzystywany do przetwarzania, analizy i wizualizacji danych. Kolejne części tej książki poświęcone są tym trzem typowym zastosowaniom.

Ale zanim omówione będą bardziej złożone funkcjonalności, w tym podrozdziale przeglądamy te podstawowe.

Zmienne w analizie danych zazwyczaj charakteryzuje się zgodnie z klasyfikacją Stanleya Stevensa:

- Zmienne jakościowe (nazywane również wyliczeniowymi, czynnikami lub kategorycznymi) to zmienne przyjmujące określoną liczbę wartości (najczęściej nieliczbowych). Zmienne te można podzielić na:
  - binarne (nazywane również dwumianowymi, dychotomicznymi), np. płeć (poziomy: kobieta/mężczyzna),
  - nominalne (nazywane również zmiennymi jakościowymi nieuporządkowanymi), np. marka samochodu, pomiędzy wartościami tych zmiennych nie ma żadnego porządku,
  - porządkowe (nazywane też jakościowymi uporządkowanymi), np. wykształcenie (poziomy: podstawowe/średnie/wyższe).
- Zmienne ilościowe, które można dodatkowo podzielić na:
  - licznikowe (liczba wystąpień pewnego zjawiska, opisywana liczbą naturalną), np. liczba lat nauki,
  - przedziałowe (nazywane też interwałowymi), mierzone w skali, w której odejmowanie wartości ma sens, ale iloraz już nie, np. temperatura w stopniach Celsjusza lub rok naszej ery,
  - ilorazowe, czyli zmienne przedziałowe, mierzone w skali, w której dodatkowo zachowane są proporcje, wartości można dzielić, skala ma zero absolutne, np. temperatura w stopniach Kelvina, wzrost w centymetrach itp.

W programie R zmienne ilościowe reprezentowane są przez typ liczbowy `numeric`. Nie ma osobnych typów do opisywania liczb w skali ilorazowej i interwałowej.

Zmienne jakościowe w R są reprezentowane przez typ czynnikowy `factor`. Zmienne typu `factor` mogą dodatkowo być oznaczone jako uporządkowane, mają wtedy dodatkowo klasę `ordered`.

Zmienne binarne mogą być też reprezentowane przez typ logiczny `logical`.

W tabeli 2.1 przedstawiamy funkcje wyznaczające najpopularniejsze statystyki opisowe. Przykładowe wywołania statystyk opisowych przećwiczymy na danych `daneSoc` z pakietu `Przewodnik`.

```
library("Przewodnik")
head(daneSoc, 4)
##   wiek wyksztalcenie st.cywilny      plec             praca
## 1   70      zawodowe  w zwiazku mezczyzna uczen lub pracuje
## 2   66      zawodowe  w zwiazku   kobieta uczen lub pracuje
## 3   71      zawodowe    singiel   kobieta uczen lub pracuje
## 4   57       srednie  w zwiazku mezczyzna uczen lub pracuje
##   cisnienie.skurczowe cisnienie.rozkurczowe
## 1                 143                    83
## 2                 123                    80
## 3                 167                    80
## 4                 150                    87
```

2.3.1. Zmienne ilościowe

Przyjrzyjmy się wartościom w kolumnie `wiek`. Możemy się do niej odwoływać przez instrukcję `daneSoc$wiek`.

Wiek to zmienna ilościowa w skali ilorazowej (ilorazy mają w jej przypadku sens, można np. powiedzieć, że ktoś jest dwa razy starszy niż inna osoba).

Pierwsze pytanie: jaka jest najmniejsza i największa wartość przyjmowana przez zmienną `wiek`? Zawsze warto sprawdzić wartości skrajne, mogą uwidocznić jakieś błędy w danych.

```
range(daneSoc$wiek)
## [1] 22 75
```

Jaka jest średnia dla zmiennej `wiek`?

```
mean(daneSoc$wiek)
## [1] 43.16176
```

A średnia obcięta wyznaczona na podstawie 60% środkowych obserwacji?

```
mean(daneSoc$wiek, trim=0.2)
## [1] 42.58065
```

Mediana okazuje się bliska średniej – czyżby brak skośności?

```
median(daneSoc$wiek)
## [1] 45
```

Do przedstawiania najważniejszych charakterystyk zmiennych można wykorzystać funkcję `summary()`. W przypadku zmiennych ilościowych wynikiem jest wektor z wartościami minimum, maksimum, średniej, mediany i kwartyli pierwszego i trzeciego (nazywanych też dolnym i górnym).

Wszystkie te wartości poza średnią są również wyznaczane przez funkcje `fivenum()` (tzw. pięć liczb Tukeya, dzielących obserwowane wartości na cztery równe części). Jeżeli w zmiennej występują brakujące obserwacje, to w podsumowaniu wypisywana jest ich liczba.

```
summary(daneSoc$wiek)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   22.00   30.00   45.00   43.16   53.00   75.00
```

Odchylenie standardowe:

```
sd(daneSoc$wiek)
## [1] 13.8471
```

Kurtoza / miara spłaszczenia:

```
kurtosis(daneSoc$wiek)
## [1] -0.955848
```

Skośność:

```
skewness(daneSoc$wiek)
## [1] 0.233151
```

Wybrane kwantyle zmiennej `wiek`:

```
quantile(daneSoc$wiek, c(0.1, 0.25, 0.5, 0.75, 0.9))
##  10%  25%  50%  75%  90%
## 26.0 30.0 45.0 53.0 62.4
```

Dla kilku zmiennych ilościowych popularnie wyznaczaną statystyką jest korelacja. Można ją obliczyć funkcją `cor()`. Macierz korelacji dla trzech wybranych kolumn:

```
cor(daneSoc[,c(1,6,7)])
##                          wiek cisnienie.skurcz cisnienie.rozkurcz
## wiek                  1.00000         -0.02765           -0.08313
## cisnienie.skurcz     -0.02765          1.00000            0.67852
## cisnienie.rozkurcz   -0.08313          0.67852            1.00000
```

2.3.2. Zmienne jakościowe

Przyjrzyjmy się teraz wartościom w kolumnie `wyksztalcenie`. Możemy się do niej odwoływać przez instrukcję `daneSoc$wyksztalcenie`.

Wykształcenie to zmienna jakościowa uporządkowana. Przyjmuje cztery różne wartości i mają one naturalnie określony porządek.

Najczęściej wykorzystywaną statystyką w przypadku zmiennych jakościowych jest tablica liczebności. Funkcja `table()`:

```
table(daneSoc$wyksztalcenie)
## podstawowe    srednie     wyzsze   zawodowe
##         93         55         34         22
```

Funkcja ta wyznacza tablicę liczebności dla jednej, dwóch lub większej liczby zmiennych wyliczeniowych. Tablice kontyngencji można również obliczyć funkcjami `xtabs()` i `ftable()`.

```
table(daneSoc$wyksztalcenie, daneSoc$praca)
##              nie pracuje uczen lub pracuje
##   podstawowe           8                14
##   srednie              6                28
##   wyzsze              22                71
##   zawodowe            16                39
```

W przypadku zmiennych jakościowych podobny efekt co funkcja `table()` ma funkcja `summary()`. Różnica polega na tym, że w razie występowania danych `NA` funkcja `table()` je ignoruje, a funkcja `summary()` wypisuje ich liczbę.

Tablica 2.1. Statystyki opisowe dla wektora lub macierzy

Z pakietu `base`	
`max()/min()`	Wartość maksymalna/minimalna w próbie.
`mean()`	Średnia arytmetyczna, $\bar{x} = \sum_i x_i / n$. Opcjonalnym argumentem jest `trim`, jeżeli jest różny od zera, to wyznaczana jest średnia ucięta. Średnią uciętą oblicza się tak jak arytmetyczną, po usunięciu $200\% * trim$ skrajnych obserwacji.
`length()`	Liczba elementów w próbie.
`range()`	Przedział zmienności próby, wyznaczony jako $[\min_i x_i, \max_i x_i]$.
Z pakietu `stats`	
`weighted.mean()`	Średnia ważona, wyznaczona jako $\frac{1}{n}\sum_i w_i x_i$. Wektor wag w_i jest drugim argumentem.
`median()`	Mediana (wartość środkowa).
`quantile()`	Kwantyl wybranego rzędu. Drugim argumentem funkcji `quantile()` jest wektor kwantyli do wyznaczenia. W tej funkcji zaimplementowano 9 różnych algorytmów do wyliczania kwantyli, zobacz opis argumentu `type`.
`IQR()`	Rozstęp międzykwartylowy, czyli różnica pomiędzy górnym a dolnym kwartylem, $IQR = q_{0.75} - q_{0.25}$.
`var()`	Wariancja w próbie. Wyznaczana jest nieobciążona ocena wariancji $S^2 = \frac{1}{n-1}\sum_i (x - \bar{x})^2$. Dla dwóch wektorów obliczona będzie kowariancja dla tych wektorów, a dla macierzy wynikiem będzie macierz kowariancji kolumn.
`sd()`	Odchylenie standardowe wyznaczone jako $\sqrt{S^2}$, gdzie S^2 to ocena wariancji.
`cor()`, `cov()`	Macierz korelacji i kowariancji. Argumentami może być para wektorów lub macierz.
`mad()`	Medianowe odchylenie bezwzględne, wyznaczone jako $1.4826 * median(\lvert x_i - median(x_i)\rvert)$.
Z innych pakietów	
`kurtosis()`	Kurtoza, miara koncentracji, $\frac{n\sum_i (x_i-\bar{x})^4}{\left(\sum_i (x_i-\bar{x})^2\right)^2} - 3$. Rozkład normalny ma kurtozę 0. Funkcja z pakietu `e1071`.
`skewness()`	Skośność, miara asymetryczności, $\frac{\sqrt{n}\sum_i (x_i-\bar{x})^3}{\left(\sum_i (x_i-\bar{x})^2\right)^{3/2}}$. Rozkład symetryczny ma skośność 0. Funkcja z pakietu `e1071`.
`geometric.mean()`	Średnia geometryczna, wyznaczona jako $(\Pi_i x_i)^{1/n}$. Funkcja z pakietu `psych`.
`harmonic.mean()`	Średnia harmoniczna, wyznaczona jako $n / \sum_i x_i^{-1}$. Funkcja z pakietu `psych`.
`moda()`	Moda lub dominanta, czyli wartość występująca najczęściej w próbie. Funkcja z pakietu `dprep`. W Linuxie można też użyć funkcji `mod()` z pakietu `RVAideMemoire`.

```
summary(daneSoc$wyksztalcenie)
## podstawowe    srednie     wyzsze   zawodowe
##         22         34         93         55
```

Argumentem funkcji `summary()` może być również obiekt typu `data.frame`. W tym przypadku podsumowania wyznaczane są dla każdej kolumny wskazanej ramki danych.

```
summary(daneSoc[,1:4])
##       wiek          wyksztalcenie     st.cywilny         plec
##  Min.   :22.00   podstawowe:93    singiel  :120   kobieta  : 55
##  1st Qu.:30.00   srednie   :55    w zwiazku: 84   mezczyzna:149
##  Median :45.00   wyzsze    :34
##  Mean   :43.16   zawodowe  :22
##  3rd Qu.:53.00
##  Max.   :75.00
```

2.4. Statystyki graficzne

Poniżej prezentujemy najpopularniejsze wykresy wykorzystywane do graficznej reprezentacji zmiennych. Mniej popularne, ale równie ciekawe i użyteczne wykresy przedstawione są w rozdziale 5.5.

W poniższych przykładach wykorzystano zbiór danych `daneSoc` z pakietu `Przewodnik`.

2.4.1. Wykres słupkowy, funkcja: barplot()

Wykresy słupkowe są jedną z najpopularniejszych form graficznego przedstawiania danych. Patrząc na słupki, zazwyczaj porównujemy ich względne długości, dlatego ta forma prezentacji nadaje się do zmiennych w skali ilorazowej.

Funkcja `barplot()` (patrz rysunki 2.2 i 2.3) służy do przedstawiania danych w postaci słupków, zarówno poziomych, jak i pionowych. Argumentem może być wektor wartości do przedstawienia lub dwuwymiarowa macierz.

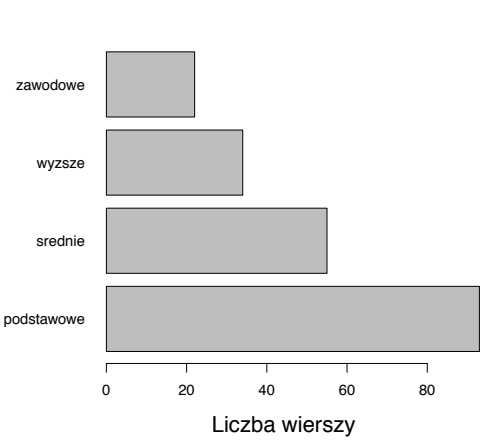
Argument `horiz` określa, czy słupki mają być rysowane poziomo, czy pionowo. Argument `las` wskazuje natomiast, w jakim kierunku mają być rysowane opisy przy osiach. Funkcją `legend()` można dorysować do wykresu legendę, o tej funkcji można przeczytać więcej w rozdziale 5.5.5.

Argument `col` pozwala na wybranie kolorów dla słupków i legendy, a zastosowanie argumentu `add=TRUE` powoduje, że nie jest tworzony nowy wykres, ale słupki dorysowane są do poprzedniego.

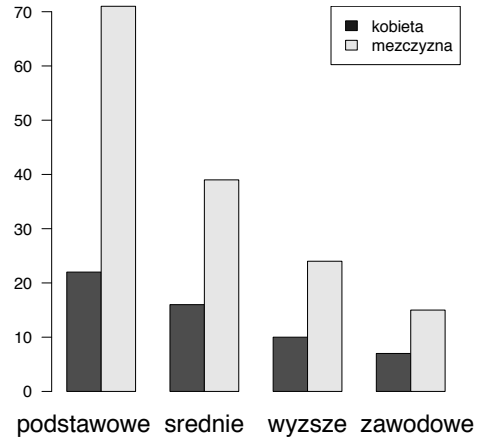
Poniższy kod generuje wykresy przedstawione na rysunkach 2.2 i 2.3. Funkcją `table()` wyznaczana jest macierz częstości, która jest następnie rysowana funkcją `barplot()`.

```
library("Przewodnik")
tab <- table( daneSoc$wyksztalcenie )
barplot(tab, horiz = TRUE, las = 1)

tab2 <- table( daneSoc$plec, daneSoc$wyksztalcenie )
barplot(tab2, las=1, beside = TRUE)
legend("topright",c("kobieta","mezczyzna"),fill=c("grey","black"))
```



Rysunek 2.2. Poziomy wykres słupkowy dla wektora liczb.



Rysunek 2.3. Pionowy wykres słupkowy dla macierzy liczb.

2.4.2. Histogram, funkcja: hist()

Histogram jest bez wątpienia jedną z najpopularniejszych statystyk graficznych, przedstawiającą rozkład wartości zmiennych ilościowych.

Funkcja `hist()` (patrz rysunki 2.4 i 2.5) służy do prezentowania za pomocą długości słupków liczby obserwacji w zadanych przedziałach. Pierwszym argumentem tej funkcji jest wektor liczb.

Argument `breaks` określa sposób podział zakresu zmienności przedstawianych danych na przedziały, może być to jedna liczba opisująca liczbę przedziałów do wyznaczenia lub napis oznaczający algorytm wyznaczania liczby podziałów (patrz opis niżej). Argumenty `freq` i `probability` określają, czy przy rysowaniu histogramu mają być zaznaczane frakcje, czy liczebności elementów w przedziałach. Argument `right` wskazuje, czy przedziały mają być traktowane jako domknięte prawostronnie, czy lewostronnie.

Jeżeli nie podamy liczby przedziałów, to zostanie ona dobrana w zależności od liczby obserwacji oraz zmienności danej zmiennej. Do określania liczby i szerokości przedziałów służy argument `breaks`. Jeżeli podamy jako war-

tość tego argumentu jedną liczbę, to będzie ona potraktowana jako sugestia oczekiwanej liczby automatycznie wyznaczonych przedziałów (tylko sugestia, ponieważ funkcja `hist()` może liczbę przedziałów nieznacznie zwiększyć lub zmniejszyć). Jeżeli podany będzie wektor liczb, to będzie on uznany za wektor punktów rozdzielających przedziały (przedziały nie muszą mieć równych szerokości). Jeżeli argumentem będzie napis, to zostanie on zinterpretowany jako nazwa algorytmu do wyznaczenia liczby przedziałów (możliwe wartości to: `"Sturges"`, `"Scott"`, `"FD"` i `"Freedman-Diaconis"`).

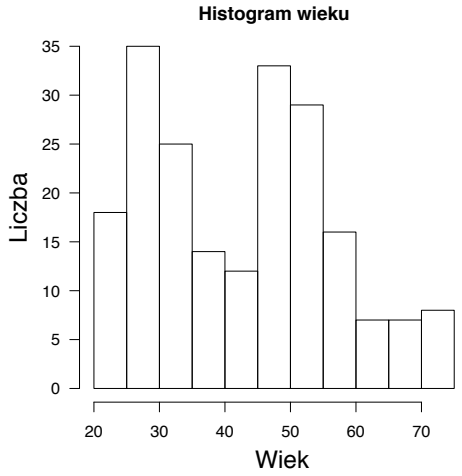
Poniżej przedstawiamy dwa przykładowe wywołania funkcji `hist()`, wyniki ich działania można zobaczyć na rysunkach 2.4 i 2.5.

Zaczniemy od histogramu dla zmiennej `wiek`. Ponieważ w danych mamy wiele wierszy, argumentem `breaks` zasugerujemy podział poziomej osi na 15 przedziałów/kubełków. Pozostałe argumenty poniższego wywołania to określenia opisów osi i tytułu wykresu.

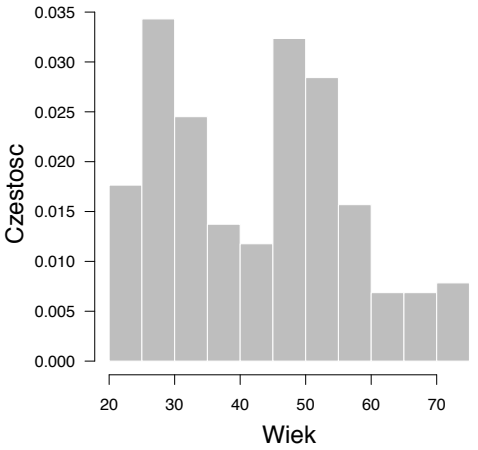
```
hist(daneSoc$wiek, breaks = 15, main="Histogram wieku", las=1,
     ylab="Liczba", xlab="Wiek")
```

Zmieniając kolory obramowania histogramu i słupków, możemy uzyskać bardziej estetyczny histogram. Dodatkowo zmieniamy oś pionową, by przedstawiać częstości zamiast liczebności.

```
hist(daneSoc$wiek, breaks = 15, col="grey", border="white", las=1,
     probability = TRUE, ylab="Czestosc", xlab="Wiek")
```



Rysunek 2.4. Histogram dla zmiennej wiek.



Rysunek 2.5. Zmiana kolorów wypełniania i obramowania słupków.

2.4.3. Wykres pudełkowy: boxplot()

Główną zaletą histogramu jest to, że łatwo na nim zauważyć regiony dużego zagęszczenia/rozrzedzenia wartości. Wadę stanowi to, że trudno z jego pomocą porównać grupy obserwacji, ponieważ dla każdej grupy przedstawianych jest wiele informacji.

Do porównywania rozkładów wartości w grupach bardzo popularną metodą jest wykres pudełkowy, nazywany też boxplotem.

Funkcja `boxplot()` (patrz rysunki 2.6 i 2.7) służy do przedstawiania rozkładu zmiennej liczbowej za pomocą pudełek. Na jednym wykresie można zaprezentować wyniki dla kilku zmiennych (by porównać rozkłady tych zmiennych) lub dla pojedynczej zmiennej w rozbiciu na grupy obserwacji (by porównać rozkład zmiennej w podgrupach).

Pierwszy argument funkcji `boxplot()` określa wektor lub listę wektorów, lub ramkę danych, lub formułę ze zmiennymi, które mają znaleźć się na wykresie pudełkowym. Można też podać kilka wektorów zmiennych jako kolejne argumenty. Jeżeli argumentem będzie wektor liczb, to narysowane zostanie jedno pudełko, jeżeli będzie to kilka wektorów liczb, to narysowanych zostanie kilka pudełek, a jeżeli argumentem będzie formuła opisująca zależność pomiędzy zmienną ilościową a zmienną jakościową, to powstanie osobne pudełko dla każdego poziomu zmiennej jakościowej.

Pozostałe argumenty funkcji `boxplot()` to: `range` – określający szerokość przedziału, poza którym obserwacje są traktowane jako odstające, argument `varwidth` – określający czy grubość pudełka ma być proporcjonalna do pierwiastka z liczby obserwacji w wektorze, argument `outline` – określający, czy mają być rysowane wartości odstające.

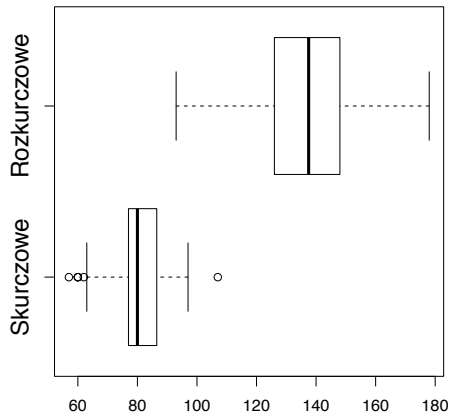
Poniżej zamieszczamy dwa przykłady wywołania funkcji `boxplot()`. Wynik tych wywołań prezentujemy na rysunkach 2.6 i 2.7.

Jako kolejne argumenty funkcji `boxplot()` można podawać następne wektory liczb, a argumentem `names` można określić etykiety na osi wykresu. Argument `horizontal` umożliwia obrót wykresu pudełkowego.

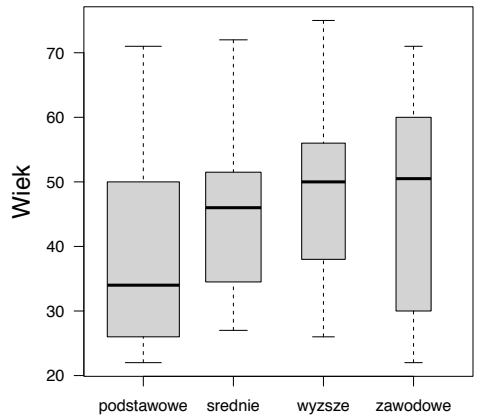
```
boxplot(daneSoc$cisnienie.rozk, daneSoc$cisnienie.skur,
        horizontal = TRUE, names = c("Skurczowe","Rozkurczowe"))
```

Pierwszym argumentem może być formuła, w wygodny sposób określa ona, jakie grupy mają być porównywane. Poniżej porównujemy wiek w grupach wykształcenia. Dodatkowo, argumentem `varwidth` powodujemy, że szerokość pudełka odpowiadać będzie liczebności porównywanej grupy.

```
boxplot(wiek~wyksztalcenie, data = daneSoc, varwidth=TRUE,
        col="lightgrey", ylab="Wiek", las=1)
```

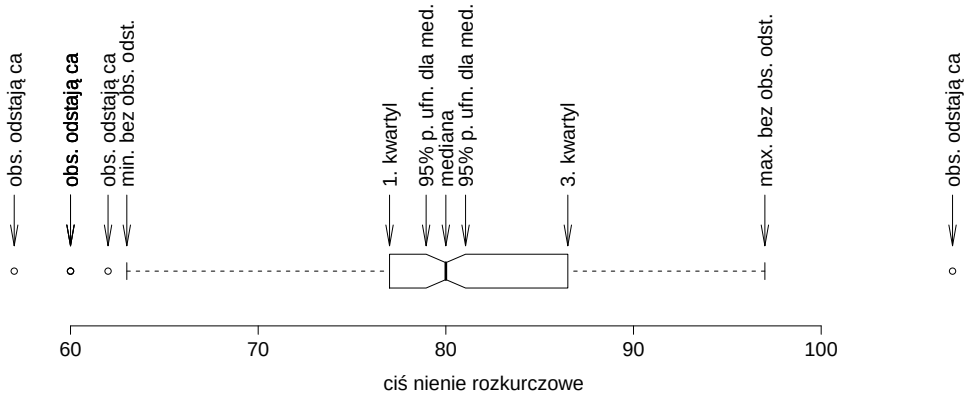


Rysunek 2.6. Wykresy pudełkowe dla dwóch wektorów



Rysunek 2.7. Wykres pudełkowy przedstawiający zmienną `wiek`

Wykres pudełkowy przypomina ramkę z wąsami, stąd też inna jego nazwa: wykres ramka–wąsy. Pomiędzy wąsami znajdują się prawie wszystkie obserwacje (poza odstającymi). Za odstające obserwacje uznawane są te, które odstają od kwartyli bardziej niż o 1,5 IQR (1,5 to domyślna wartość argumentu `range`). Wewnątrz pudełka wyznaczonego przez kwartyle znajduje się 50% obserwacji. Znaczenie poszczególnych elementów jest przedstawione na poniższym schemacie.



2.4.4. Jądrowy estymator gęstości, funkcja: density()

Zarówno histogram, jak i wykres pudełkowy przedstawiają rozkład wartości wektora liczb. Dwie inne statystyki, które są często wykorzystywane w tym celu, to jądrowy estymator gęstości (gładka wersja histogramu wyznaczana funkcją `density()`) i dystrybuanta empiryczna (obliczana funkcją `ecdf()`).

Idea jądrowego estymatora gęstości polega na wyznaczeniu oceny gęstości w danym punkcie na podstawie koncentracji obserwacji w okolicy tego punktu. Obserwacje położone bliżej interesującego punktu wpływają na oceny gęstości z większą wagą niż obserwacje bardziej oddalone. Szablon tych wag określony jest przez parametr nazywany jądrem. Za to, które obserwacje są uznawane za bliskie, odpowiada parametr nazywany szerokością okna, szerokością pasma lub też stopniem wygładzenia.

Funkcja `density()` (patrz rysunki 2.8 i 2.9) służy do wyznaczania jądrowego estymatora gęstości dla wektora liczb. Pierwszym argumentem jest wektor wartości, dla których chcemy wyliczyć ocenę gęstości.

Argumenty `from` i `to` określają początek i koniec przedziału, w którym wyznaczona ma być gęstość, argument `n` wskazuje liczbę punktów, w których wartość gęstości ma być wyznaczona (gęstość wyliczana jest dla regularnej siatki punktów). Parametry `kernel` i `bw` służą do określenia rodzaju jądra i szerokości okna. Wynikiem funkcji `density()` jest obiekt klasy `density`, którego składowe przechowują wartości ocen gęstości we wskazanych punktach. Obiekty tej klasy można przedstawiać graficznie przeciążoną funkcją `plot()`.

Domyślnie ocena gęstości wyznaczana jest z użyciem jądra gaussowskiego. Przeglądając plik pomocy dla funkcji `density()`, zainteresowany czytelnik odkryje, jak wykorzystywać inne jądra i czym one się różnią. Wybór jest spory, dostępne są jądra prostokątne, trójkątne i wiele innych, patrz parametr `kernel`.

Szerokość pasma (szerokość okna) można ustalić ręcznie lub wskazać regułę jej wyboru, która automatycznie określi najodpowiedniejszą szerokość. W pakiecie `stats` zaimplementowanych jest pięć różnych metod automatycznego wyboru szerokości pasma.

Domyślnie stosowana jest „reguła kciuka” (używna gdy `bw="nrd0"`) zaproponowana przez Silvermana. Zgodnie z tą regułą szerokość okna h wyznaczana jest ze wzoru:

$$h_{bw.nrd0} = 0{,}9 \min(\hat{\sigma}, IQR/1{,}34) n^{-1/5}, \tag{2.1}$$

gdzie $\hat{\sigma}$ to ocena odchylenia standardowego, IQR to rozstęp kwartylowy z próby, a n to liczba obserwacji. „Magiczna” stała 1,34 bierze się stąd, że dla rozkładu normalnego $IQR/1{,}34 \approx \sigma$. Inną popularną regułą kciuka jest reguła Scota, stosowana gdy `bw="nrd"`. Szerokość okna wyznacza się ze wzoru:

$$h_{bw.nrd} = 1{,}06 \hat{\sigma} n^{-1/5}. \tag{2.2}$$

Można wykorzystać też inne reguły wyboru szerokości pasma, np. oparte na metodzie walidacji krzyżowej (ang. *cross validation*): nieobciążoną – dla argumentu `bw="ucv"` i obciążoną – dla argumentu `bw="bcv"`, lub estymator typu `plug-in` Sheathera-Jonesa (dla argumentu `bw="SJ"`). W większości przypad-

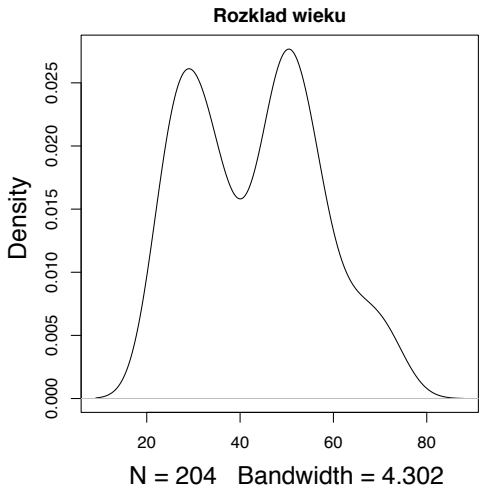
ków najlepsze wyniki oceny szerokości pasma otrzymuje się, wykorzystując metodę Sheathera-Jonesa.

Poniżej przedstawiamy przykłady dwóch ocen gęstości wyznaczonych dla różnych parametrów szerokości pasma (pierwszy wybrany automatyczny, drugi „ręcznie"). Efekt wykonania tych instrukcji znajduje się na rysunkach 2.8 i 2.9.

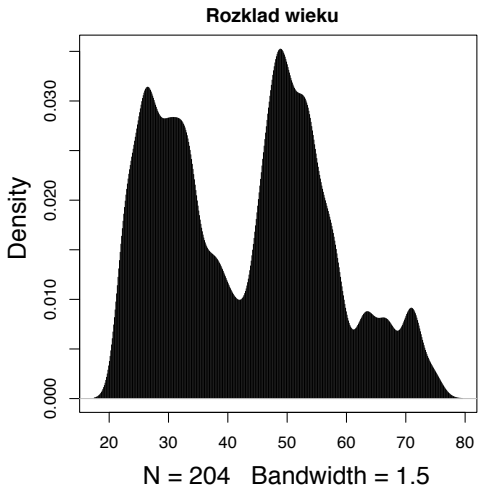
```
plot(density(daneSoc$wiek), main="Rozklad wieku")
```

Jeżeli uznamy, że okno wygładzania jest zbyt szerokie, można je ręcznie zmodyfikować. Na przykładzie poniżej redukujemy je do wartości 1,5.

```
plot(density(daneSoc$wiek, bw=1.5), main="Rozklad wieku",type="h")
```



Rysunek 2.8. Jądrowy estymator gęstości z automatycznie wybraną szerokością okna.



Rysunek 2.9. Jądrowy estymator z szerokością okna ustawioną na 1,5.

Kolejną przydatną statystyką do opisu rozkładu wektora obserwacji jest dystrybuanta empiryczna. Można ją wyznaczyć używając komendy `ecdf()`.

Dystrybuanta empiryczna to funkcja opisująca, jaki procent obserwacji w wektorze przyjmuje wartości mniejsze niż x.

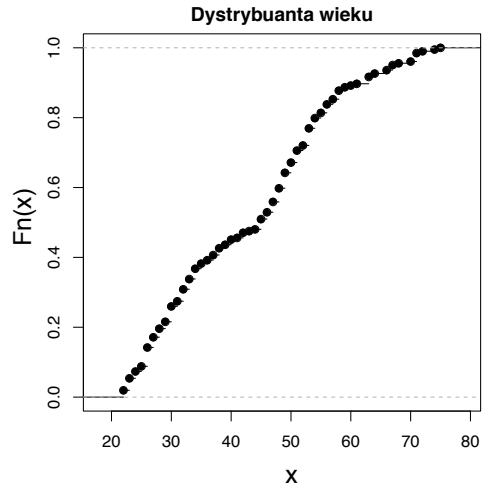
Wynik funkcji `ecdf()` jest obiektem klasy `ecdf`. Dla tej klasy zaimplementowano przeciążoną wersję funkcji `plot()` prezentującą graficznie dystrybuantę empiryczną (zobacz przykład na rysunku 2.11).

```
plot(ecdf(daneSoc$wiek), main="Dystrybuanta wieku")
```

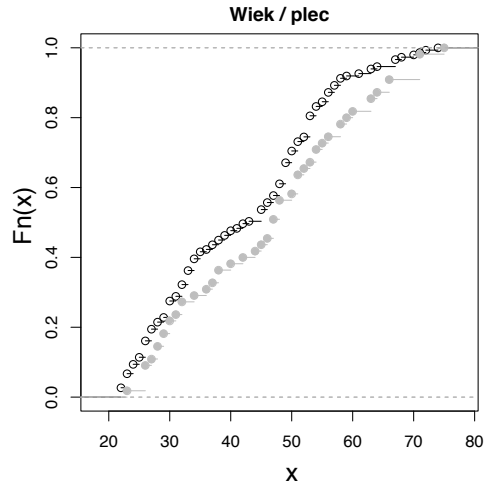
Dystrybuantę można wykorzystać, aby porównać kilka grup obserwacji. Wystarczy na jednym wykresie przedstawić dystrybuanty dla różnych grup (argument `add`).

Na poniższym przykładzie rysujemy dwie dystrybuanty na jednym wykresie, jedną dla mężczyzn, a drugą dla kobiet.

```
library("dplyr")
mezczyzni <- filter(daneSoc, plec == "mezczyzna")
kobiety   <- filter(daneSoc, plec == "kobieta")
plot(ecdf(mezczyzni$wiek), main="Wiek / plec", pch=21)
plot(ecdf(kobiety$wiek), add=TRUE, col = "grey")
```



Rysunek 2.10. Dystrybuanta empiryczna dla wieku



Rysunek 2.11. Dwie dystrybuanty empiryczne pozwalają na porównania populacji

2.4.5. Wykres kropkowy, funkcja: scatterplot()

Aby pokazać zależność pomiędzy dwoma zmiennymi liczbowymi, często wykorzystuje się wykres kropkowy, nazywany też punktowym lub wykresem rozrzutu (przedstawia rozrzut obserwacji).

W programie R jest wiele sposobów narysowania tego popularnego wykresu. Najprostszym jest użycie funkcji `plot()`, ale poniżej prezentujemy funkcję `car::scatterplot()`, która ma kilka interesujących dodatkowych opcji. Wśród nich: prezentację danych w podziale na grupy lub dodanie trendu liniowego lub nieliniowego do danych.

Pierwszym argumentem mogą być albo dwa wektory wskazujące zmienne, które mają być zaprezentowane na wykresie, albo formuła wskazująca dwie zmienne z określonego zbioru danych.

Domyślnie na osiach rysowane są również wykresy pudełkowe przedstawiające rozkład poszczególnych zmiennych. Można to zmienić, modyfikując argument `boxplots`, np. podając `boxplots="x"`, powodujemy, że wykres pudełkowy rysowany jest tylko na osi OX. W bibliotece `car` znajduje się kopia funkcji `scatterplot()` o krótszej nazwie `sp()`, ale identycznym działaniu.

Do wykresu rozrzutu można dorysować prostą regresji liniowej (prosta jest rysowana w zależności od wartości argumentu `reg.line`, wartość `FALSE` wyłącza rysowanie prostej regresji, domyślnie `reg.line=TRUE`) oraz krzywą regresji wygładzonej (krzywa jest rysowana w zależności od wartości argumentu `smooth`, wartość `FALSE` wyłącza rysowanie tej krzywej, domyślna wartość to `smooth=TRUE`). W prezentowanym przykładzie (rysunek 2.13) wykres rozrzutu dla dwóch zmiennych (ciśnienia skurczowego i rozkurczowego) został przedstawiony w dwóch podpopulacjach określonych przez zmienną `plec`. Na wykresie rozrzutu możemy zobaczyć, czy istnieje wyraźna zależność pomiędzy badanymi zmiennymi i czy ta zależność jest taka sama, czy różna w podpopulacjach.

Efekty wykonania poniżej przedstawionych instrukcji znajdują się na rysunkach 2.12 i 2.13. Pierwszy argument funkcji `sp()` to formuła określająca, która zmienna ma być przedstawiona na osi OX, a która na osi OY wykresu. Argument `pch` wskazuje kształt rysowanych punktów.

```
library("car")
sp(cisnienie.rozkurczowe~cisnienie.skurczowe, data=daneSoc,
   smooth=FALSE, reg.line=FALSE, pch=19)
```

W formule, po znaku `|`, możemy również określić zmienną warunkującą. W przykładzie poniżej każda płeć jest rysowana na wykresie za pomocą innych kształtów punktów i kolorów. Dodatkowo do każdego zbioru punktów dodawany jest trend liniowy.

```
sp(cisnienie.rozkurczowe~cisnienie.skurczowe|plec, data=daneSoc,
   smooth=FALSE, lwd=3, pch=c(19,17))
```

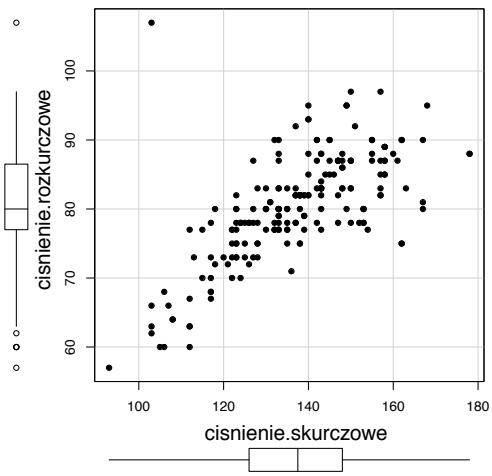
Wykresy rozrzutu są bardzo przydatne do wykrywania i opisywania zależności liniowych lub monotonicznych pomiędzy parą zmiennych.

W przypadku dużej liczby zmiennych można wykorzystać funkcje `pairs()`, `scatterplot.matrix()` lub `YaleToolkit::gpairs()`. Rysują one macierz zależności pomiędzy każdą parą analizowanych zmiennych.

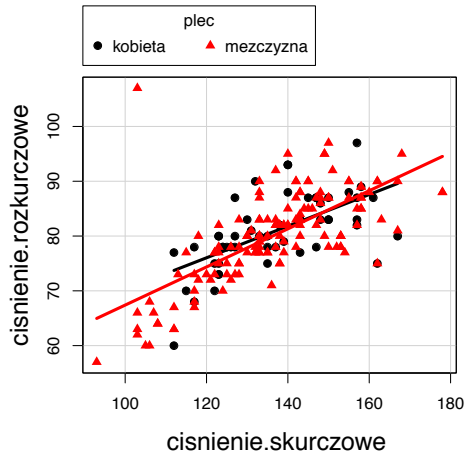
2.4.6. Wykres mozaikowy, funkcja: mosaicplot()

Jak pokazać zależność pomiędzy dwoma lub większą liczbą zmiennych jakościowych? Jedną z możliwości jest użycie wykresu słupkowego `barplot()`. Inną jest zastosowanie jego mniej znanego, ale bardzo potężnego brata – wykresu mozaikowego.

Funkcja `mosaicplot()` (patrz rysunki 2.14 i 2.15) przedstawia liczebność wartości w zbiorze danych za pomocą pola prostokątów. Prostokąty te są tak ułożone, by można było łatwo zauważyć: (1) czy dwie zmienne od siebie zależą oraz (2) jakie jakie są częstości występowania pojedynczych wartości i par wartości.



Rysunek 2.12. Wykres kropkowy dla dwóch zmiennych.



Rysunek 2.13. Dodajemy linie trendu i zaznaczamy dane dla różnych płci.

Przedstawmy ten wykres na dwóch przykładach. Wyniki poniższych instrukcji są zaprezentowane na rysunkach 2.14 i 2.15.

Pierwszy wykres przestawia wyłącznie częstość występowania różnych grup wykształcenia. Prostokąty mają stałą wysokość, ale różnią się szerokością. Im prostokąt jest szerszy (pole większe) tym dana grupa jest częstsza.

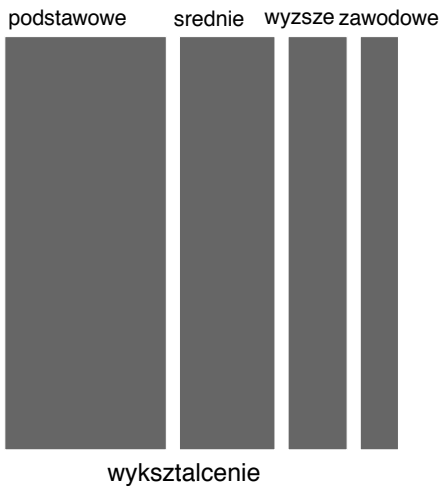
```
mosaicplot(~wyksztalcenie, data=daneSoc, main="", border="white")
```

Drugi wykres przedstawia zależność pomiędzy zmiennymi `wyksztalcenie` a `praca`. Tak jak w poprzednim przykładzie szerokość prostokątów odpowiada częstościom grup wykształcenia. Wysokości odpowiadają względnym częstościom osób pracujących w różnych grupach wykształcenia.

Co ważne, jeżeli dwie zmienne są niezależne, to mozaika przypomina regularną kratę. Jeżeli występuje jakaś zależność, to łatwo zauważyć, jakiego jest ona rodzaju. Na poniższym wykresie widać, że najwięcej osób pracujących występuje wśród osób z wykształceniem wyższym.

```
mosaicplot(~wyksztalcenie+praca, data=daneSoc, border="white",
       col=c("grey40", "grey70"))
```

Na jednej mozaice można zaprezentować wyniki dla więcej niż dwóch zmiennych typu wyliczeniowego. Należy jednak uważać, bowiem umieszczenie bardzo wielu zmiennych lub zmiennych o bardzo wielu poziomach znacząco zmniejsza czytelność i przydatność tego wykresu.



Rysunek 2.14. Wykres mozaikowy przedstawiający częstość zmiennej `wyksztalcenie`.

Rysunek 2.15. Wykres mozaikowy przedstawiający zależność pomiędzy zatrudnieniem a wykształceniem.

2.5. Jak przetwarzać dane z pakietem `dplyr`

Program R jest wyposażony w tysiące (tak, dosłownie tysiące!) funkcji do przetwarzania danych. Na szczęście, aby wygodnie pracować z większością danych, wystarczy poznać na początek kilka najważniejszych.

W przypadku przetwarzania danych obowiązuje zasada 80/20 (tzw. zasada Pareta, od nazwiska Wilfreda Pareto). Oznacza ona, że aby efektywnie przetwarzać większość (nie wszystkie, ale np. 80% możliwych) danych, wystarczy znać jedynie niewielki ułamek z wszystkich dostępnych funkcji.

Hadley Wickham przygotował dwa pakiety, `dplyr` i `tidyr`, które udostępniają dokładnie tę niewielką część funkcji. Każda z nich wykonuje tylko jedną elementarną operację. Ale można je elastycznie łączyć, przez co w sumie pozwalają one na przeprowadzenie większości typowych operacji na danych.

Wickham nazywa funkcje w tych pakietach czasownikami, a proces analizy danych przyrównuje do konstrukcji zdania. Podstawowymi czasownikami analizy danych są:

- `filter()` – wybierz wskazane wiersze ze zbioru danych,
- `select()` – wybierz tylko wskazane kolumny ze zbioru danych,
- `arrange()` – posortuj wiersze względem wskazanych kolumn,
- `mutate()` – dodaj nową kolumnę z danymi lub zmień istniejącą,
- `group_by()` / `ungroup()` - pogrupuj dane względem wskazanych czynników / usuń informację o pogrupowaniu,
- `summarise()` – wyznacz określone agregaty w każdej grupie,

- gather() / spread() - przekształć dane z postaci szerokiej na wąską / z wąskiej na szeroką.

Te podstawowe czasowniki są opisane w kolejnych podrozdziałach.

Więcej funkcji pozwalających na eksplorację danych przedstawionych jest w ściągawce opracowanej przez RStudio, dostępnej na ich stronie [26] lub na http://bit.ly/1LaYWBd.

2.5.1. Jak filtrować wiersze

Jedną z najczęstszych operacji na danych jest filtrowanie wierszy, które spełniają określony warunek lub warunki.

Funkcja filter() z pakietu dplyr wykonuje filtrowanie. Jej pierwszym argumentem jest zbiór danych, na których ma pracować, a kolejne argumenty to warunki logiczne.

Wynikiem tej funkcji są wiersze spełniające wszystkie wskazane warunki logiczne. Określając warunki, można wykorzystywać nazwy kolumn ze zbioru danych bez podawania dodatkowych odnośników.

Pokażmy to na przykładzie. Instrukcja, która ze zbioru danych auta wybierze tylko oferty z polem Model == "Corsa", jest następująca:

```
library("dplyr")
tylkoCorsa <- filter(auta, Model == "Corsa")
head(tylkoCorsa)
##   Marka Model  Cena  KM Pojemnosc Przebieg  Paliwo Produkcja
## 1  Opel Corsa 13450  70      1248   190000  diesel      2004
## 2  Opel Corsa 25990  75      1300    84000  diesel      2008
## 3  Opel Corsa 17990  80      1229    48606 benzyna      2007
## 4  Opel Corsa 23999  60       998    63000 benzyna      2009
## 5  Opel Corsa 16500 101      1700   118000  diesel      2006
## 6  Opel Corsa 27900  80      1229    73500 benzyna      2007
```

Możemy określać jednocześnie więcej warunków. Poniższy przykład jako wynik wyznaczy zbiór danych obejmujący auta marki Corsa, z silnikiem Diesla wyprodukowane w roku 2010.

```
tylkoCorsa <- filter(auta, Model == "Corsa", Produkcja == 2010,
    Paliwo == "diesel")
head(tylkoCorsa)
##   Marka Model     Cena KM Pojemnosc Przebieg Paliwo Produkcja
## 1  Opel Corsa 49050.00 75      1300      100 diesel      2010
## 2  Opel Corsa 47202.34 76      1300    53000 diesel      2010
## 3  Opel Corsa 37900.00 95      1248     8300 diesel      2010
## 4  Opel Corsa 12200.00 75      1248    11378 diesel      2010
## 5  Opel Corsa 34900.00 95      1300    24000 diesel      2010
## 6  Opel Corsa 37900.00 75      1248    18500 diesel      2010
```

2.5.2. Jak wybierać kolumny

Dane zawierają niekiedy dużą liczbę kolumn, ale często pracujemy tylko na kilku z nich. Usunięcie pozostałych spowoduje, że dane będą mniejsze i operacje na nich będą szybsze.

Inną zaletą wybierania jedynie potrzebnych kolumn jest łatwiejsze wyświetlanie danych. Zamiast pokazywać wszystkie, nawet nieistotne kolumny, często lepiej zaprezentować jedynie te istotne.

Funkcja `select()` z pakietu `dplyr` pozwala na wybór jednej lub wielu zmiennych ze źródła danych. Pierwszym argumentem jest źródło danych, a kolejnymi są nazwy kolumn, które mają być wybrane.

Przykładowo poniższa instrukcja wybiera jedynie kolumny z marką, modelem i ceną auta.

```
trzyKolumny <- select(auta, Marka, Model, Cena)
head(trzyKolumny)
##     Marka Model  Cena
## 1 Peugeot   206  8799
## 2 Peugeot   206 15500
## 3 Peugeot   206 11900
## 4 Peugeot   206 10999
## 5 Peugeot   206 11900
## 6 Peugeot   206 19900
```

Poza wybraniem kolumn przez wskazanie ich nazw – można również skorzystać z operatora negacji `'-'` (*znak minus*), który wybiera wszystkie kolumny poza wskazanymi, lub z funkcji `matches()`, `starts_with()`, `ends_with()`, które wybierają kolumny o nazwach spełniających określone warunki.

W poniższym przykładzie wybieramy kolumny o nazwach rozpoczynających się na literę `M`.

```
head(select(auta, starts_with("M")))
##     Marka Model
## 1 Peugeot   206
## 2 Peugeot   206
## 3 Peugeot   206
## 4 Peugeot   206
## 5 Peugeot   206
## 6 Peugeot   206
```

2.5.3. Jak tworzyć i transformować zmienne

Modelowanie czy przetwarzanie danych często wymaga utworzenia nowych zmiennych na bazie istniejących. Czasem na podstawie ceny buduje się logarytm ceny (jedna zmienna z jednej zmiennej), czasem na podstawie wagi i wzrostu liczy się BMI (jedna zmienna z kilku zmiennych).

Funkcja mutate() z pakietu dplyr pozwala na wygodne tworzenie dodatkowych kolumn w zbiorze danych na podstawie transformacji istniejących już kolumn.

Przedstawmy działanie tej funkcji na przykładzie danych auta, dostępnych w pakiecie Przewodnik. Jest to zbiór danych o ogłoszeniach sprzedaży samochodów z portalu otomoto.pl z roku 2012.

Przykład rozpoczniemy od wyznaczenia wieku oferowanych aut. Ponieważ przedstawione ogłoszenia pochodzą z roku 2012, a rok produkcji oferowanego samochodu znajduje się w kolumnie Produkcja, to wiek będzie równy 2013 - Produkcja.

W kolejnym kroku liczymy średni przebieg na rok użytkowania. Do tej operacji potrzebujemy jednocześnie danych z dwóch kolumn.

```
autaZWiekiem <- mutate(auta,
                       Wiek = 2013 - Produkcja,
                       PrzebiegNaRok = round(Przebieg/Wiek))
head(select(autaZWiekiem, Marka, Model, Cena, Paliwo, Wiek,
    PrzebiegNaRok))
##     Marka Model  Cena  Paliwo Wiek PrzebiegNaRok
## 1 Peugeot   206  8799 benzyna   10          8500
## 2 Peugeot   206 15500 benzyna    8         14250
## 3 Peugeot   206 11900  diesel   10         21500
## 4 Peugeot   206 10999  diesel   10         16500
## 5 Peugeot   206 11900  diesel    8         18250
## 6 Peugeot   206 19900  diesel    7         12343
```

Przetwarzanie kolumn może być bardziej złożone niż proste przekształcenia arytmetyczne i może dotyczyć dowolnych zmiennych, nie tylko liczbowych. W dalszych rozdziałach omówimy operacje, które można wykonywać na napisach, wartościach logicznych oraz czynnikach.

2.5.4. Jak sortować wiersze

Sortowanie danych po określonej kolumnie znacznie ułatwia analizę wartości w tej kolumnie. Przede wszystkim natychmiast widać wartości skrajne, sortowanie jest więc wygodne podczas eksploracji danych.

Za pomocą funkcji arrange() z pakietu dplyr możemy wykonać sortowanie po jednej lub większej liczbie zmiennych. W przypadku remisów ze względu na pierwsze kryteria porządek rozstrzygają kolejne kryteria.

W przykładzie poniżej w pierwszej kolejności dane są sortowane po wartościach z kolumny Model. Dopiero w przypadku, gdy w tej kolumnie znajdują się takie same wartości, o kolejności decyduje następna zmienna – Cena.

```
posortowaneAuta <- arrange(auta, Model, Cena)
head(posortowaneAuta)
```

```
##     Marka Model    Cena KM Pojemnosc Przebieg  Paliwo Produkcja
## 1 Peugeot   206 6330.70 90      1997    90000  diesel      2004
## 2 Peugeot   206 6599.00 70      1398   277000  diesel      2004
## 3 Peugeot   206 6900.00 90      1997   132000  diesel      2004
## 4 Peugeot   206 7900.00 88      1360   114000 benzyna      2004
## 5 Peugeot   206 8469.45 60      1124    77800 benzyna      2005
## 6 Peugeot   206 8500.00 60      1124   117000 benzyna      2004
```

Aby odwrócić kolejność sortowania, należy wskazaną zmienną otoczyć wywołaniem funkcji `desc()`.

```
posortowaneAuta <- arrange(auta, Model, desc(Cena))
head(posortowaneAuta, 2)
##     Marka Model  Cena KM Pojemnosc Przebieg  Paliwo Produkcja
## 1 Peugeot   206 36115 60      1100      100 benzyna      2010
## 2 Peugeot   206 33600 65      1100      380 benzyna      2011
```

2.5.5. Jak pracować z potokami

Przetwarzanie danych składa się najczęściej z wielu kroków. Wykonujemy operację A, później B, później C itp.

Ważne jest, aby patrząc na kod R, można było łatwo zrozumieć, jakie operacje są wykonywane. Ułatwi to późniejszą analizę kodu – czy to w przypadku poszukiwania błędów, czy też na wypadek, gdybyśmy chcieli nasz kod pokazać innym osobom.

Potoki to mechanizm wprowadzony do R niedawno, ale który szybko zdobył wielu zwolenników. Jego główną zaletą jest to, że skraca kod R, zwiększa jego czytelność i ułatwia dodawanie kolejnych kroków przetwarzania danych.

Problem cebulki

Aby przedstawić potoki, rozpatrzmy najpierw taką serię czterech instrukcji:

```
tylkoKia <- filter(auta, Marka == "Kia")
posortowane <- arrange(tylkoKia, Cena)
czteryKolumny <- select(posortowane, Model, Cena, Przebieg,Produkcja)
head(czteryKolumny, 4)
##   Model     Cena Przebieg Produkcja
## 1 Cee'd  1026.60    28000      2008
## 2 Cee'd 13900.00   129000      2009
## 3 Cee'd 14700.00    34000      2009
## 4 Cee'd 14900.00   158500      2005
```

W języku R można wynik jednej funkcji przekazać bezpośrednio jako argument kolejnej funkcji. Z tego powodu, aby skrócić zapis, często stosuje się mało czytelny zapis „na wielką cebulkę”.

```
head(
  select(
    arrange(
      filter(auta,
             Marka == "Kia"),
      Cena),
    Model, Cena, Przebieg, Produkcja)
  , 4)
##   Model     Cena Przebieg Produkcja
## 1 Cee'd  1026.60    28000      2008
## 2 Cee'd 13900.00   129000      2009
## 3 Cee'd 14700.00    34000      2009
## 4 Cee'd 14900.00   158500      2005
```

Taką „cebulkę" czyta się od środka na zewnątrz. Najpierw wykonywane jest filtrowanie, później sortowanie, następnie wybieranie czterech kolumn i wybieranie pierwszych czterech wierszy.

Jednak taki zapis nie jest czytelny, szczególnie w przypadku tych bardziej zewnętrznych funkcji. W powyższym przykładzie funkcja `head()` przyjmuje dwa argumenty, ale pierwszy zajmuje 6 linii kodu, a drugi jest wymieniony dopiero w siódmej linii. Przy dłuższych „cebulkach" łatwo pomylić, które argumenty są argumentami których funkcji.

Jak działa operator potoku

Rozwiązaniem problemu „cebulki" jest stosowanie specjalnego operatora do przetwarzania potokowego %>%. Operator ten pochodzi z pakietu `magrittr`, ale jest też dostępny po włączeniu pakietu `dplyr`.

Najłatwiej zapamiętać, co robi ten operator, na prostym przykładzie dwuargumentowej funkcji. Zapis:

```
a %>% funkcja(b)
```

jest równoważny instrukcji:

```
funkcja(a, b)
```

Można go też stosować do funkcji o większej liczbie argumentów. Operator %>% przekazuje lewą stronę jako pierwszy argument funkcji wskazanej z prawej strony.

Jeżeli chcemy przekazać wartość do drugiego lub dalszego argumentu, to można ten argument wskazać znakiem `"."` (kropki), tak jak na poniższym przykładzie:

```
a %>% funkcja(b, data=.)
```

Potoki w akcji

Przedstawiona powyżej „cebulka" może być zapisana równoważnie w następujący, znacznie bardziej czytelny sposób.

```
auta %>%
  filter(Marka == "Kia") %>%
  arrange(Cena) %>%
  select(Model, Cena, Przebieg, Produkcja) ->
  posortowane
head(posortowane)
##   Model     Cena Przebieg Produkcja
## 1 Cee'd  1026.60    28000      2008
## 2 Cee'd 13900.00   129000      2009
## 3 Cee'd 14700.00    34000      2009
## 4 Cee'd 14900.00   158500      2005
## 5 Cee'd 16900.00     9900      2010
## 6 Cee'd 18803.89    15000      2010
```

W powyższym przykładzie użyliśmy też operatora przypisania ->. Dzięki temu składowe potoku czytamy konsekwentnie od lewej do prawej.

Operator -> działa jak <-, z tą tylko różnicą, że przypisuje wynik do zmiennej z prawej strony, a nie z lewej.

Funkcje z pakietu `dplyr` są tak zdefiniowane, by pierwszym argumentem był zawsze zbiór danych. Dzięki takiej konstrukcji domyślne zachowanie operatora %>% pozwala na skrótowy i czytelny zapis nawet dużej sekwencji wywołań funkcji.

W przypadku funkcji, które przyjmują zbiór danych jako drugi lub kolejny argument, trzeba posłużyć się dodatkowo symbolem `"."` (kropki). Symbol ten wskazuje, gdzie należy wstawić lewą stronę operatora %>%.

Zobaczmy przykład z użyciem funkcji `lm()`, która buduje model liniowy. Ta funkcja oczekuje, że dane bedą przekazane jako drugi argument. Musimy więc użyć symbolu `"."`.

```
auta %>%
        lm(Cena~Przebieg, data = .) %>%
        summary()
## Call:
## lm(formula = Cena ~ Przebieg, data = .)
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.388e+04  3.693e+03  11.883  < 2e-16 ***
## Przebieg    -9.166e-02  2.856e-02  -3.209  0.00135 **
```

W tym przykładzie nie musimy wskazywać argumentu przez nazwę `data=`, ale zrobiliśmy to dla zwiększenia czytelności.

Zauważmy, że potoki nie tylko łatwiej się czyta, ale też łatwiej opatrywać je komentarzami. Linię komentarza można umieścić przed każdą linią potoku, opisując, co się akurat dzieje w tym konkretnym miejscu.

2.5.6. Jak wyznaczać agregaty/statystyki w grupach

Częstą operacją, którą wykonuje się na danych, szczególnie tych dużych, jest wyznaczanie statystyk/podsumowań/agregatów dla podgrup danych.

Aby takie agregaty wyznaczać przy użyciu pakietu dplyr możemy wykorzystać funkcje summarise() i group_by(). Pierwsza określa, jakie statystyki chcemy policzyć, a druga wskazuje, w jakich grupach.

Przedstawiamy te funkcje poniżej jedna po drugiej.

Agregaty

Funkcją summarise() można wyznaczyć agregaty w danych.

Przykładowo poniższa instrukcja dla zbioru danych auta liczy średnią cenę, medianę przebiegu aut i średni wiek aut.

```
auta %>%
  summarise(sredniaCena = mean(Cena),
            medianaPrzebiegu = median(Przebieg),
            sredniWiek = mean(2013 - Produkcja),
            liczba = n())
##   sredniaCena medianaPrzebiegu sredniWiek liczba
## 1    33340.38           122000   6.638333   2400
```

Nie zawsze agregat związany jest z przekształceniem wartości w jakiejś kolumnie. Na przykład dosyć przydatną statystykę stanowi liczba wierszy, która nie zależy od wartości w danych. Taki agregat jest w powyższym przykładzie wyznaczany funkcją n().

Grupowanie

Funkcja group_by() pozwala określić, w jakich grupach ma być przeprowadzane dalsze przetwarzanie. Sama funkcja nie powoduje żadnych modyfikacji w danych, a jedynie dodaje znacznik wskazujący, co jest teraz zmienną grupującą. Kolejne funkcje w potoku będą dzięki temu znacznikowi wiedziały, że przetwarzanie należy prowadzić w grupach.

Funkcja group_by wpływa na działanie innych funkcji z pakietu dplyr w intuicyjny sposób.

- W przypadku funkcji summarise(), liczącej statystyki, powoduje wyznaczenie statystyk dla każdej możliwej kombinacji zmiennych grupujących.
- W przypadku funkcji arrange(), sortującej wiersze, powoduje, że wiersze są sortowane najpierw po zmiennej grupującej.

- W przypadku funkcji `sample_n()`, filtrującej wiersze, powoduje, że próbkowanie zostaje wykonane w każdej grupie niezależnie.
- W przypadku funkcji `mutate()` powoduje, że każdy agregat jest wyznaczany w ramach grupy, dzięki czemu można łatwo normować zmienne w grupach.

Poniższy przykład liczy trzy statystyki (średnią cenę, medianę przebiegu, średni wiek aut i liczbę ofert) w grupach opisanych przez rodzaj paliwa.

```
auta %>%
  group_by(Marka) %>%
  summarise(sredniaCena = mean(Cena),
            medianaPrzebiegu = median(Przebieg),
            sredniWiek = mean(2013 - Produkcja),
            liczba = n())
##        Marka sredniaCena medianaPrzebiegu sredniWiek liczba
## 1       Audi    59891.57         150452.5      6.550    200
## 2       Fiat    15772.33          75000.0      6.330    200
## 3        Kia    36982.64          42850.0      3.980    200
## 4       Opel    33590.74         120550.0      6.615    800
```

Agregaty są zwykłą ramką danych, można wykonywać na nich kolejne operacje, np. sortowanie.

```
auta %>%
  group_by(Marka) %>%
  summarise(sredniaCena = mean(Cena),
            medianaPrzebiegu = median(Przebieg),
            sredniWiek = mean(2013 - Produkcja),
            liczba = n()) %>%
  arrange(sredniaCena)
##        Marka sredniaCena medianaPrzebiegu sredniWiek liczba
##       <fctr>       <dbl>            <dbl>      <dbl>  <int>
## 1       Fiat    15772.33          75000.0      6.330    200
## 2    Peugeot    17388.51         127750.0      8.165    400
## 3       Opel    33590.74         120550.0      6.615    800
## 4        Kia    36982.64          42850.0      3.980    200
```

Poniższy przykład modyfikuje zmienną `Przebieg`, dzieląc ją przez średni przebieg danej marki.

```
auta %>%
  select(Marka, Cena, Przebieg, Model) %>%
  group_by(Marka) %>%
  mutate(Przebieg = Przebieg/mean(Przebieg, na.rm=TRUE))
##      Marka  Cena  Przebieg  Model
## 1  Peugeot  8799 0.6712017    206
## 2  Peugeot 15500 0.9001999    206
## 3  Peugeot 11900 1.6977455    206
## 4  Peugeot 10999 1.3029209    206
```

2.5.7. Postać szeroka/postać wąska

Wiele funkcji zakłada, że dane mają strukturę tabeli, w której wiersze to kolejne obserwacje, a kolumny to cechy kolejnych obserwacji. Takiego formatu danych oczekują między innymi pakiety dplyr, ggplot2 i wiele innych.

Jednak czasem dane są dostępne w innym formacie. Czasem ta sama zmienna jest opisana przez wiele kolumn lub wiele zmiennych jest opisanych w jednej kolumnie. Funkcje z pakietu tidyr pozwalają na transformacje różnych formatów danych.

Przedstawimy te formaty danych i funkcje do ich przekształcania na przykładzie danych z Eurostatu. Pobierzemy te dane z pomocą pakietu eurostat. Poniżej wczytujemy dane z tabeli tsdtr210, w której Eurostat gromadzi dane o popularności różnych form transportu w różnych krajach.

```
library("eurostat")
tsdtr210 <- get_eurostat("tsdtr210", time_format="num")
head(tsdtr210, 4)
##                     unit vehicle geo time values
## PC_BUS_TOT_AT_1990   PC BUS_TOT  AT 1990  11.0
## PC_BUS_TOT_BE_1990   PC BUS_TOT  BE 1990  10.6
## PC_BUS_TOT_BG_1990   PC BUS_TOT  BG 1990    NA
## PC_BUS_TOT_CH_1990   PC BUS_TOT  CH 1990   3.7
```

Pobrane dane są w postaci wąskiej. Co dokładnie charakteryzuje tę postać, stanie się jasne, gdy tylko przedstawimy postać szeroką. W tym przykładzie kolumna geo określa kraj, kolumna time określa rok, kolumna vehicle rodzaj transportu, a kolumna values popularność danego rodzaju transportu w podanym kraju i roku.

Rozciągnij na kolumny

Aby przejść z postaci wąskiej do postaci szerokiej, można użyć funkcji spread(). Nazwa *postać szeroka* bierze się stąd, że jest w niej więcej kolumn. Nowe kolumny w postaci szerokiej danych odpowiadają wartościom w jednej kolumnie w postaci wąskiej danych.

Funkcja spread() poza pierwszym argumentem – zbiorem danych – oczekuje jeszcze dwóch innych argumentów. Jeden będzie kluczem odpowiadającym nazwom kolumn, a drugi będzie wartościami, które zostaną wpisane do poszczególnych kolumn. Wiersze nowej tabeli z danymi są wybierane jako unikalne wartości w pozostałych kolumnach.

W przykładzie poniżej kolumna time z postaci wąskiej jest zamieniana na wiele kolumn, których nazwy odpowiadają kolejnym latom – wartościom z kolumny time. Wartości w nowych kolumnach są kopiowane z kolumny values wąskiego zbioru danych.

```
library("tidyr")
szeroka <- spread(tsdtr210, time, values)
szeroka %>% filter(geo == "PL")
##   unit vehicle geo 1990 1991 1992 1993 1994 1995 1996 1997 1998
## 1   PC BUS_TOT  PL 28.1 25.6 24.4 23.2 20.9 19.9 19.4 17.9 17.4
## 2   PC     CAR  PL 41.3 49.8 55.3 57.9 62.3 64.6 69.3 71.3 72.1
## 3   PC     TRN  PL 30.6 24.6 20.3 18.9 16.8 15.5 11.3 10.8 10.5
```

Dane w tabeli `tsdtr210` opisują wartości w przekrojach przez cztery wymiary (jednostka, typ transportu, kraj, rok). Każdy z tych wymiarów może być użyty do stworzenia nowych kolumn w postaci szerokiej.

Przykładowo poniżej kolejne kolumny są tworzone na bazie kolumny `geo` z danych w postaci wąskiej.

```
szeroka2 <- spread(tsdtr210, geo, values)
```

Wyświetlmy wiersze dla roku 2010 (pominięto część kolumn).

```
szeroka2 %>% filter(time == "2010")
##   unit vehicle time   AT   BE   BG   CH   CY   CZ DE   DK   EE
## 1   PC BUS_TOT 2010 10.3 12.7 16.4  5.1 18.1 19.5  6 10.5 14.4
## 2   PC     CAR 2010 78.7 79.7 80.0 77.3 81.9 73.0 86 79.7 83.6
## 3   PC     TRN 2010 11.0  7.7  3.6 17.6   NA  7.5  8  9.8  2.0
```

Zbierz w kolumny

Operacją odwrotną do rozciągnięcia jednej kolumny na wiele kolumn jest zbieranie wielu kolumn w jedną. Można to zrobić funkcją `gather()`.

Ta funkcja jako pierwszy argument przyjmuje zbiór danych. Kolejne dwa argumenty określają nazwy kolumn z kluczami i wartościami, a pozostałe argumenty wskazują kolumny starego zbioru danych, które mają być zebrane razem w nowym zbiorze danych. Można stosować notację ze znakiem `-"` (minus), czyli „wszystko poza...".

Na poniższym przykładzie przekształcamy ramkę `szeroka` na postać wąską w taki sposób, że wszystkie kolumny poza `geo` i `vehicle` są przekształcane w kolumnę `wartosc`. Dlatego w wyniku występują kolumny `geo` i `vehicle` (pominięte przy zbieraniu) oraz `rok` i `wartosc` (wynik zbierania).

```
szeroka %>%
  gather(rok, wartosc, -geo, -vehicle) %>%
  tail()
##      vehicle geo  rok wartosc
## 2622     TRN  SI 2013     2.3
## 2623     TRN  SK 2013     7.1
## 2624     TRN  TR 2013     1.4
## 2625     TRN  UK 2013     8.4
```

Aby wyświetlić przykładowe 4 wiersze, użyto tutaj funkcji tail() (wyświetla ostatnie sześć wierszy), ponieważ w pierwszych sześciu wierszach są wartości NA.

2.5.8. Sklejanie/rozcinanie kolumn

Pracując z danymi, często potrzebujemy zamienić kilka kolumn w jedną kolumnę lub jedną w kilka.

Sklejanie kolumn

Zdarza się, że wartości z kilku kolumn chcemy skleić ze sobą w jedną kolumnę. Można to zrobić funkcją unite().

Poniższy przykład skleja wartości z kolumnach geo i time w jedną kolumnę geo_time klasy napis.

```
unite(tsdtr210, geo_time, geo, time, sep=":") %>%
    head(4)
##                     unit vehicle geo_time values
## PC_BUS_TOT_AT_1990   PC BUS_TOT  AT:1990  11.0
## PC_BUS_TOT_BE_1990   PC BUS_TOT  BE:1990  10.6
## PC_BUS_TOT_BG_1990   PC BUS_TOT  BG:1990    NA
## PC_BUS_TOT_CH_1990   PC BUS_TOT  CH:1990   3.7
```

Operacja sklejenia jest często przydatna, jeżeli chcemy oprzeć grupowanie na kilku zmiennych. Wtedy te zmienne możemy posklejać i taki „zlepek" traktować jako nową zmienną grupującą.

Rozcinanie kolumn

Operacją odwrotną do sklejania jest rozcinanie kolumn. Można je wykonać funkcją separate().

Działanie tej funkcji przedstawimy na przykładzie sztucznego zbioru danych z dwoma kolumnami – datą i identfikatorem.

W poniższym przykładzie funkcja separate() na podstawie kolumny daty tworzy trzy nowe kolumny. Wypełnia je wartościami po rozbiciu daty na części rozdzielone znakiem -.

```
df <- data.frame(daty = c("2004-01-01", "2012-04-15", "2006-10-29",
    "2010-03-03"), id = 1:4)
df
##         daty id
## 1 2004-01-01  1
## 2 2012-04-15  2
## 3 2006-10-29  3
## 4 2010-03-03  4
```

```
separate(df, col=daty, into=c("rok", "miesiac", "dzien"), sep="-")
##    rok miesiac dzien id
## 1 2004      01    01  1
## 2 2012      04    15  2
## 3 2006      10    29  3
## 4 2010      03    03  4
```

Długość wektora `into` określa, ile kolumn ma powstać po rozcięciu. Jeżeli w kolumnie, która jest rozcinana, będzie zbyt mało lub zbyt dużo wartości (np. tylko dwa elementy rozdzielane separatorem), to domyślnie funkcja `separate()` będzie sygnalizowała ostrzeżenia. Dodatkowymi argumentami `extra` i `fill` można określić, jak w takiej sytuacji mają być uzupełniane brakujące wartości.

2.6. Jak wczytać i zapisać dane w różnych formatach

W tym rozdziale pokazujemy, jak wczytywać dane tabelaryczne z różnych źródeł. Przedstawiamy najróżniejsze formaty, mając na uwadze dużą różnorodność potrzeb osób korzystających z programu R. W większości przypadków lekturę tego rozdziału można ograniczyć do trzech podrozdziałów, pokazujących, jak wczytywać dane z pakietów, plików tekstowych i plików Excela.

W ogólności dane mogą mieć najróżniejszą strukturę. Mogą przyjmować strukturę grafu, drzewa, mogą opisywać zdjęcia czy pliki multimedialne. Wszystkie te struktury można do programu R wczytać i je analizować. Najczęściej jednak dane, z którymi pracujemy, mają strukturę tabeli z wyróżnionymi kolumnami i wierszami. Na tego rodzaju strukturze skupimy się w tym rozdziale.

2.6.1. Wczytywanie danych z pakietów

Najłatwiej wczytać dane, które są umieszczone w pakiecie. W takiej sytuacji wystarczy włączyć pakiet funkcją `library()` oraz użyć funkcji `data()`, aby wczytać dane o określonej nazwie. W jednym pakiecie można umieścić wiele zbiorów danych, przez co często pakiety wykorzystywane są jako nośniki użytecznych danych.

Takim pakietem – nośnikiem danych jest pakiet `PogromcyDanych`. Zawiera on różne zbiory danych, z których korzystamy w tej książce, między innymi zbiór danych `koty_ptaki`. Po zainstalowaniu pakietu `PogromcyDanych`, aby skorzystać z tego zbioru danych, wystarczą dwa polecenia:

```
library("PogromcyDanych")
data("koty_ptaki")
```

W okienku Environment programu RStudio powinien pojawić się nowy symbol wskazujący na dodany zbiór danych. Używając funkcji head(), wyświetlmy pierwsze wiersze z ww. zbioru danych.

```
head(koty_ptaki)
##   gatunek waga dlugosc predkosc habitat zywotnosc druzyna
## 1  Tygrys  300     2.5       60    Azja        25     Kot
## 2     Lew  200     2.0       80  Afryka        29     Kot
## 3  Jaguar  100     1.7       90 Ameryka        15     Kot
## 4    Puma   80     1.7       70 Ameryka        13     Kot
## 5 Leopard   70     1.4       85    Azja        21     Kot
## 6  Gepard   60     1.4      115  Afryka        12     Kot
```

Zbiór danych koty_ptaki składa się z 7 kolumn, opisujących różne parametry przykładowych zwierząt.

Aby wyświetlić listę wszystkich zbiorów danych, które znajdują się w określonym pakiecie, również można wykorzystać funkcję data():

```
data(package="PogromcyDanych")
```

2.6.2. Wczytywanie danych z plików tekstowych

Bardzo popularnym formatem przechowywania danych są pliki tekstowe. Ich zaletą jest to, że zawartość pliku można podejrzeć w dowolnym edytorze tekstu, np. w RStudio. Najczęściej mają one rozszerzenia txt, csv (od ang. *comma separated values*), tsv (od ang. *tab separated values*) itp. W takich plikach kolejne wartości kolumn są rozdzielane za pomocą jakiegoś znaku separującego, np. tabulacji, przecinka, średnika czy innej wyróżnionej wartości.

Przyjrzyjmy się takim przykładowym danym umieszczonym w pliku http://biecek.pl/R/dane/koty_ptaki.csv. Poniżej znajduje się treść tego pliku:

```
gatunek;waga;dlugosc;predkosc;habitat;zywotnosc;druzyna
Tygrys;300;2,5;60;Azja;25;Kot
Lew;200;2;80;Afryka;29;Kot
Jaguar;100;1,7;90;Ameryka;15;Kot
Puma;80;1,7;70;Ameryka;13;Kot
Leopard;70;1,4;85;Azja;21;Kot
Gepard;60;1,4;115;Afryka;12;Kot
Irbis;50;1,3;65;Azja;18;Kot
Jerzyk;0,05;0,2;170;Euroazja;20;Ptak
Strus;150;2,5;70;Afryka;45;Ptak
Orzel przedni;5;0,9;160;Polnoc;20;Ptak
Sokol wedrowny;0,7;0,5;110;Polnoc;15;Ptak
Sokol norweski;2;0,7;100;Polnoc;20;Ptak
Albatros;4;0,8;120;Poludnie;50;Ptak
```

Widzimy, że plik zawiera tabelę wartości rozdzielanych średnikami. Pierwszy wiersz to nagłówek z nazwami kolumn – zmiennych:

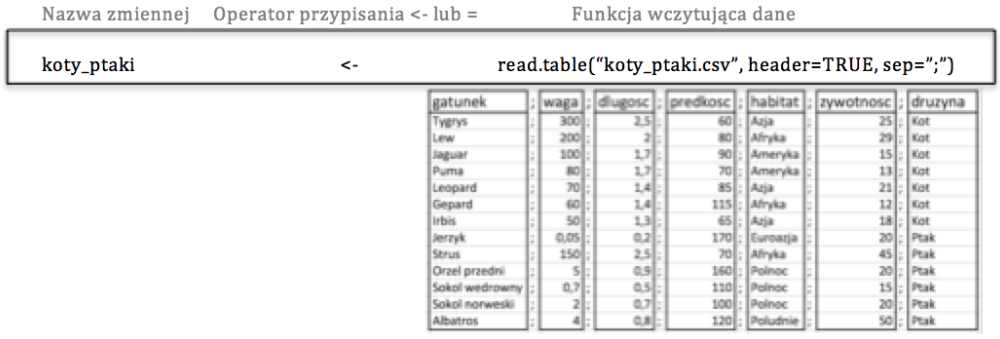
```
gatunek;waga;dlugosc;predkosc;habitat;zywotnosc;druzyna
```

Kolejne wiersze zawierają wartości tych zmiennych. Opis struktury logicznej tego pliku przedstawiamy na rysunku 2.16.

gatunek	;	waga	;	dlugosc	;	predkosc	;	habitat	;	zywotnosc	;	druzyna
Tygrys	;	300	;	2,5	;	60	;	Azja	;	25	;	Kot
Lew	;	200	;	2	;	80	;	Afryka	;	29	;	Kot
Jaguar	;	100	;	1,7	;	90	;	Ameryka	;	15	;	Kot
Puma	;	80	;	1,7	;	70	;	Ameryka	;	13	;	Kot
Leopard	;	70	;	1,4	;	85	;	Azja	;	21	;	Kot
Gepard	;	60	;	1,4	;	115	;	Afryka	;	12	;	Kot
Irbis	;	50	;	1,3	;	65	;	Azja	;	18	;	Kot
Jerzyk	;	0,05	;	0,2	;	170	;	Euroazja	;	20	;	Ptak
Strus	;	150	;	2,5	;	70	;	Afryka	;	45	;	Ptak
Orzel przedni	;	5	;	0,9	;	160	;	Polnoc	;	20	;	Ptak
Sokol wedrowny	;	0,7	;	0,5	;	110	;	Polnoc	;	15	;	Ptak
Sokol norweski	;	2	;	0,7	;	100	;	Polnoc	;	20	;	Ptak
Albatros	;	4	;	0,8	;	120	;	Poludnie	;	50	;	Ptak

Rysunek 2.16. Logiczna struktura pliku `koty_ptaki.csv`

Aby dane w tym formacie wczytać do programu R, można wykorzystać funkcję `read.table()`. Ma ona wiele parametrów, które opisujemy szczegółowo w dalszej części. Na poniższym przykładzie wczytana tabela wartości jest przypisywana za pomocą operatora <- do symbolu `koty_ptaki`.



Rysunek 2.17. Logiczna struktura wczytania danych z pliku

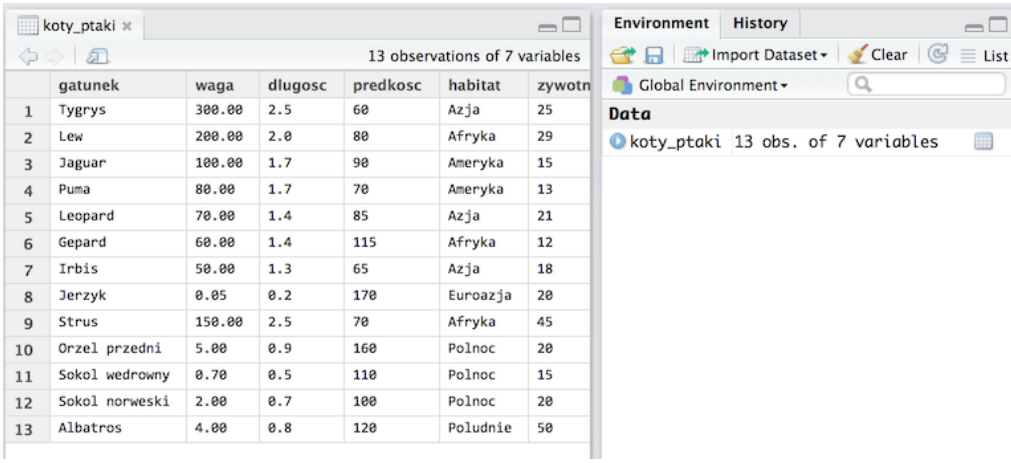
```
koty_ptaki <- read.table(file="http://biecek.pl/R/koty_ptaki.csv",
          sep=";", dec=",", header=TRUE)
```

Dane możemy wczytać z pliku znajdującego się na dysku, lub bezpośrednio z Internetu. W przypadku rozważanego pliku musimy określić argumenty:

- `file = "http://biecek.pl/R/koty_ptaki.csv"`, czyli ścieżka do pliku tekstowego,
- `sep=";"`, czyli informacja, co jest separatorem kolejnych pól, tutaj jest to średnik,
- `dec=","`, czyli informacja, co jest kropką dziesiętną, zazwyczaj jest to znak kropki, ale akurat w tym pliku kropką dziesiętną jest przecinek,
- `header=TRUE`, czyli informacja, że pierwszy wiersz z danymi to nazwy kolumn, a nie wartości.

Aby przekonać się, że dane zostały poprawnie wczytane, wykorzystajmy ponownie funkcję `head()`. Innym sposobem jest użycie zakładki `Environment` w programie RStudio, tak jak to przedstawiliśmy na rysunku 2.18.

```
head(koty_ptaki)
##   gatunek waga dlugosc predkosc habitat zywotnosc druzyna
## 1  Tygrys  300     2.5       60    Azja         25     Kot
## 2     Lew  200     2.0       80  Afryka         29     Kot
## 3  Jaguar  100     1.7       90 Ameryka         15     Kot
## 4    Puma   80     1.7       70 Ameryka         13     Kot
## 5 Leopard   70     1.4       85    Azja         21     Kot
## 6  Gepard   60     1.4      115  Afryka         12     Kot
```



	gatunek	waga	dlugosc	predkosc	habitat	zywotn
1	Tygrys	300.00	2.5	60	Azja	25
2	Lew	200.00	2.0	80	Afryka	29
3	Jaguar	100.00	1.7	90	Ameryka	15
4	Puma	80.00	1.7	70	Ameryka	13
5	Leopard	70.00	1.4	85	Azja	21
6	Gepard	60.00	1.4	115	Afryka	12
7	Irbis	50.00	1.3	65	Azja	18
8	Jerzyk	0.05	0.2	170	Euroazja	20
9	Strus	150.00	2.5	70	Afryka	45
10	Orzel przedni	5.00	0.9	160	Polnoc	20
11	Sokol wedrowny	0.70	0.5	110	Polnoc	15
12	Sokol norweski	2.00	0.7	100	Polnoc	20
13	Albatros	4.00	0.8	120	Poludnie	50

Rysunek 2.18. Podgląd tabeli z danymi w programie RStudio

Wczytywanie danych z dużych plików funkcją `read.table()` może być czasochłonne. W takich sytuacjach warto rozważyć użycie znaczenie szybszej funkcji `data.table::fread()`, z niewiele mniejszą liczbą parametrów.

2.6.2.1. Funkcja read.table() od podszewki

Niestety nie ma jednego oficjalnego formatu przechowywania danych w plikach tekstowych. Separatorami, kropkami dziesiętnymi, znakami komentarza czy znakami oznaczającymi napisy może być wiele różnych znaków.

Funkcja `read.table()` potrafi wczytać dane z praktycznie każdym wariantem zapisu. Jest to jednak możliwe wyłącznie za cenę olbrzymiej liczby dodatkowych opcji. Deklaracja funkcji `read.table()` jest następująca:

```
read.table(file, header=FALSE, sep="", quote="\"'", dec=".",
        row.names, col.names, as.is=!stringsAsFactors,
        na.strings="NA", colClasses=NA, nrows=-1, skip=0,
        check.names=T, fill=!blank.lines.skip, strip.white=F,
        blank.lines.skip=T, comment.char="#", allowEscapes=F,
        flush=FALSE, stringsAsFactors, encoding = "unknown")
```

W pakiecie `utils` znajdują się również cztery inne funkcje o takim samym działaniu, ale innych wartościach domyślnych.

```
read.csv(file, header=TRUE, sep=",", quote="\"", dec=".",
         fill=TRUE, comment.char="", ...)
read.csv2(file, header=TRUE, sep=";", quote="\"", dec=",",
         fill=TRUE, comment.char="", ...)
read.delim(file, header=TRUE, sep="\t", quote="\"", dec=".",
         fill=TRUE, comment.char="", ...)
read.delim2(file, header=TRUE, sep="\t", quote="\"", dec=",",
         fill=TRUE, comment.char="", ...)
```

Jeżeli dane zapisaliśmy w formacie CSV przy użyciu polskiej wersji programu Excel, separatorem kolumn będzie średnik, a znakiem dziesiętnym będzie przecinek. W tym przypadku najodpowiedniejszą funkcją do odczytu takich danych jest funkcja `read.csv2()`. Jeżeli korzystamy z angielskiej wersji Excela, to znakiem dziesiętnym jest kropka, a najodpowiedniejszą funkcją do odczytu tych danych jest funkcja `read.csv()`.

W tabeli 2.2 przedstawiamy opis argumentów funkcji `read.table()`. Angielską wersję tych opisów znaleźć można w pliku pomocy `?read.table`. Jak widzimy, argumentów jest wiele, warto choć raz o wszystkich przeczytać.

Tablica 2.2. Argumenty funkcji `read.table()`

`file`	Pole o wartości typu znakowego. Jest to jedyny argument obligatoryjny, tzn. musi być jawnie wskazany. Jego wartość to ścieżka do pliku, w którym znajdują się dane. Można wskazać plik na dysku lokalnym lub z Internetu, podając adres URL określający, na którym serwerze i w jakim katalogu ten plik się znajduje. To pole może również określać połączenie lub gniazdo (ang. *socket*). Jeżeli zamiast nazwy pliku podamy wartość `"clipboard"`, to dane będą czytane ze schowka systemowego, co jest wygodnym sposobem przenoszenia danych z różnych „office'ów".
`header`	Pole o wartości typu logicznego. Jeżeli wartość w tym polu wynosi `TRUE`, to pierwszy wiersz będzie traktowany jako lista nazw kolumn. Jeżeli ten argument nie zostanie podany, a w pierwszej linii w pliku jest o jedno pole mniej pól niż pól w kolejnych liniach, to R automatycznie potraktuje tę linię jako nagłówek.
`sep`	Pole o wartości typu znakowego. Wskazany łańcuch znaków będzie traktowany jako separator kolejnych pól. Wartość `""` (domyślna) powoduje, że każdy biały znak (spacja lub ich ciąg, tabulator, znak nowej linii) jest traktowany jako separator.
`quote`	Pole o wartości typu znakowego. Każdy znak występujący w tym łańcuchu jest traktowany jako znak cytowania. Tekst pomiędzy dwoma znakami cytowania traktowany jest jako jedno pole nawet wtedy, gdy występują w nim separatory.
`dec`	Pole o wartości typu znakowego. Określa, jaki znak reprezentuje kropkę dziesiętną. Jeżeli dane zapisane są zgodnie z polskimi standardami, to kropką dziesiętną jest znak `","` (przecinek).
`row.names`	Pole o wartości typu wektor napisów lub pojedyncza liczba. Jeżeli jest to wektor napisów, to będzie on traktowany jako nazwy kolejnych wierszy. Jeżeli tym argumentem jest liczba, to jest ona traktowana jako numer kolumny w odczytywanym pliku, która zawiera nazwy wierszy. Jeżeli w danych występuje nagłówek (a więc parametr `header=TRUE`) i pierwsza linia ma o jedno pole mniej niż pozostałe, to pierwsza kolumna automatycznie traktowana jest jako wektor z nazwami wierszy.
`col.names`	Wektor napisów. Tym argumentem można określić nazwy kolumn. Jeżeli ten argument nie jest podany, a w pliku nie ma nagłówka, to nazwy kolejnych zmiennych konstruowane są przez złączenie litery `"V"` z numerem kolumny.
`nrows`	Wartość typu liczbowego. Określa, jaka maksymalna liczba wierszy ma być odczytana.

`skip`	Wartość typu liczbowego. Określa, ile początkowych linii pliku ma zostać pominiętych przed rozpoczęciem czytania danych.
`check.names`	Wartość typu logicznego. Czy wymuszać sprawdzanie, czy nazwy zmiennych są poprawne, bez niedozwolonych znaków i powtórzeń.
`as.is`	Wektor zmiennych typu logicznego lub liczbowego. Domyślnie każde pole, które nie jest konwertowane na liczbę rzeczywistą lub wartość logiczną, jest konwertowane na zmienną typu `factor`. Jeżeli parametr `as.is` jest wektorem wartości logicznych, to jest on traktowany jako wskazanie, które kolumny mogą być konwertowane na typ `factor`, a które nie. Wektor liczb jest traktowany jako indeksy kolumn, które nie powinny być konwertowane.
`na.strings`	Wektor napisów. Wskazuje, jakie wartości mają być traktowane jako `NA`, czyli jakie wartości określają brakujące obserwacje. Dodatkowo jako brakujące obserwacje oznaczane są też puste pola w kolumnach o wartościach typu liczbowego lub logicznego.
`colClasses`	Wektor znaków. Każdy znak określa typ kolejnej zmiennej (kolumny). Dopuszczalne wartości to: `NA` – domyślna wartość, oznacza automatyczną konwersję, `NULL` – ta kolumna będzie pominięta, jedna z klas atomowych (logical, integer, numeric, complex, character, raw, factor, Date lub POSIXct).
`fill`	Wartość typu logicznego. Jeżeli kolejne linie mają różną liczbę wartości, a argument `fill=FALSE` (domyślnie), to działanie funkcji `read.table()` zakończy się błędem. Jeżeli argument `fill=TRUE`, to do krótszych wierszy dodawane będą na koniec wartości `NA`, tak aby uzupełnić wiersze do równej długości.
`blank.lines.skip`	Wartość typu logicznego. Jeżeli wynosi `TRUE`, to przy wczytywaniu danych pomijane będą puste linie.
`comment.char`	Wartość typu znakowego. Ten znak traktowany jest jako znak komentarza. Po wystąpieniu tego znaku zawartość do końca linii jest ignorowana.
`allowEscapes`	Wartość typu logicznego. Jeżeli jest równa `TRUE`, to tekst odczytywany jest zgodnie z zasadami oznaczania znaków specjalnych. Przykładowo napisy $\backslash t$ lub $\backslash n$ są zamieniane na odpowiednio znak tabulacji oraz znak nowej linii. Wartość `FALSE` (domyślna) powoduje, że tekst czytany jest dosłownie, bez żadnych interpretacji.
`stringsAsFactors`	Wartość typu logicznego. Jeżeli jest równa `TRUE`, to łańcuchy znaków będą konwertowane na typ wyliczeniowy `factor`. Jeżeli `FALSE`, to będą reprezentowane jako łańcuchy znaków. Domyślna wartość to `TRUE`.

2.6.3. Zapisywanie danych do pliku tekstowego

2.6.3.1. Funkcja write.table()

Funkcja `write.table()` zapisuje wartość zmiennej (atomowej, wektora, macierzy lub ramki danych) w określonym formatowaniu do pliku tekstowego. Pełna deklaracja tej funkcji jest następująca:

```
write.table(x, file="", append=FALSE, quote=TRUE, sep=" ",
            eol="\n", na="NA", dec=".", row.names=TRUE,
            col.names=TRUE, qmethod=c("escape","double"))
```

Podobnie jak w przypadku funkcji `read.table()` w pakiecie `utils` dostępne są dwie funkcje o identycznym działaniu co `write.table()`, ale przy innej parametryzacji argumentów domyślnych. Obie funkcje przeznaczone są do zapisywania danych w formacie `CSV`.

```
write.csv(x, file = "", sep = ",", dec=".", ...)
write.csv2(x, file = "", sep = ";", dec=",", ...)
```

W tabeli 2.3 przedstawiamy opis wszystkich parametrów funkcji `write.table()`, przygotowany na podstawie pliku pomocy `?write.table`. Poniżej na kilku przykładach pokażemy, jak zapisać wektor, liczbę lub ramkę danych do pliku.

Ta instrukcja wypisze wektor liczb na ekranie zamiast do pliku:

```
write.table(runif(100),"")
```

Ta instrukcja wpisze wektor do schowka systemowego:

```
write.table(1:100,"clipboard",col.names=FALSE, row.names=FALSE)
```

Ta instrukcja wpisze napis do pliku tekstowego:

```
write.table("poniedzialek","c:/najgorszyDzienWTygodniu.txt")
```

Ta instrukcja zapisze dane do pliku w formacie `CSV`:

```
write.csv2(iris, file="doExcela.csv")
```

Zapisywanie dużych obiektów typu `data.frame` do pliku w formacie tekstowym może być czasochłonne. Jest to spowodowane przełączaniem się formatowania dla każdej kolejnej kolumny, która może być innego typu. Program R sprawdza typ dla każdej kolumny i każdego wiersza i formatuje wartości w zależności od ich typu. Aby przyśpieszyć zapis, czasem warto dane przekonwertować na macierz funkcją `as.matrix()`.

Tablica 2.3. Argumenty funkcji `write.table()`

`x`	Obiekt do zapisania. Pole obligatoryjne. Ten obiekt to najczęściej macierz lub obiekt typu `data.frame`.
`file`	Wartość typu znakowego. Jest to ścieżka do pliku, w którym mają być zapisane dane. To pole może określać również połączenie lub gniazdo, tak jak w przypadku funkcji `read.table()`. Podobnie, jeżeli zamiast nazwy pliku podamy wartość `"clipboard"`, to dane będą zapisane do schowka systemowego. Wartość `""` spowoduje wyświetlenie obiektu na konsoli.
`qmethod`	Wartość typu znakowego. Wskazuje, jak oznaczone mają być znaki specjalne. Wartość `"escape"` poleca zastosować oznaczenie w stylu C (z dodaniem znaku backslash), wartość `"double"` nakazuje zastosować podwójny znak backslash.
`sep`	Wartość typu znakowego. Wskazany łańcuch znaków będzie traktowany jako separator kolejnych pól (separatorem nie musi być pojedynczy znak).
`eol`	Wartość typu znakowego. Ten łańcuch będzie wstawiany jako koniec linii. W przypadku systemu Windows domyślne zakończenie linii to `eol="\r\n"`.
`quote`	Wartość typu logicznego. Wartość `TRUE` powoduje, że pola typu `factor` oraz łańcuchy znaków będą otoczone znakiem `'"'` (cudzysłów).
`dec`	Wartość typu znakowego. To pole wskazuje, jaki ma być znak dziesiętny. Domyślnie jest to `"."` (kropka). Jeżeli dane mają być zapisane zgodnie z polskimi standardami, to separatorem dziesiętnym powinien być znak `","` (przecinek).
`row.names`	Wektor napisów lub wartość typu logicznego. Jeżeli jest to wektor napisów, to będzie on traktowany jako nazwy kolejnych wierszy. Jeżeli jest to wartość logiczna, to będzie ona określała, czy nazwy wierszy powinny być zapisane, czy nie.
`col.names`	Wektor napisów lub wartość typu logicznego. Jeżeli jest to wektor napisów, to będzie on traktowany jako nazwy kolejnych kolumn. Jeżeli jest to wartość logiczna, to będzie ona określała, czy nazwy kolumn powinny być zapisane, czy nie.

2.6.4. Wczytywanie i zapisywanie z zastosowaniem formatu JSON

Powyżej pisaliśmy głównie o danych tabelarycznych. Jednak nie wszystkie dane można zapisać w postaci tabeli.

Uniwersalnym i bardzo elastycznym formatem stosowanym do przenoszenia różnych danych jest JSON (ang. *JavaScript Object Notation*). Jest on szczególnie często wykorzystywany w rozwiązaniach opartych na komunikacji przez Interenet, między innymi w rozmaitych serwisach REST.

JSON to format tekstowy (a więc czytelny dla człowieka), opisuje dane w hierarchicznej strukturze (co jest bardzo elastycznym rozwiązaniem) i jest lekki (czyli nie zajmuje dużo miejsca).

Poniżej przedstawiamy przykładowe dane w formacie JSON. Opisują one dwuelementową listę, której pierwszą wartością jest napis, a drugą wartością tabela.

```
{ "zrodlo": "Przewodnik",
  "wskazniki": [
     {"rok": "2010", "wartosc": 15.5},
     {"rok": "2011", "wartosc": 10.1},
     {"rok": "2012", "wartosc": 12.2}
 ] }
```

W programie R jest kilka pakietów do wczytywania danych z formatu JSON i zapisywania danych do tego formatu. Trzy najpopularniejsze to `RJSONIO`, `rjson` i `jsonlite`. Każdy z nich ma funkcję `toJSON()`, konwertującą dane do formatu JSON, i funkcję `fromJSON()`, wczytującą dane zapisane w tym formacie.

Najbardziej stabilny w użyciu jest pakiet `jsonlite` i poniżej przedstawiamy przykład tylko dla niego.

Z formatu JSON do R: Wczytane dane mają początkowo strukturę listy, jeżeli jest to możliwe, to struktura jest upraszczana do ramki danych/wektora.

```
fromJson({"zrodlo": "Przewodnik","wskazniki": [{"rok": "2010", "
    wartosc": 15.5},{"rok": "2011", "wartosc": 10.1},{"rok": "2012", "
    wartosc": 12.2}]})
## $zrodlo
## [1] "Przewodnik"
##
## $wskazniki
##    rok wartosc
## 1 2010    15.5
## 2 2011    10.1
## 3 2012    12.2
```

Z R do formatu JSON: Poniżej przedstawiamy przykład konwersji dwóch pierwszych wierszy ramki danych `koty_ptaki`:

```
toJSON(koty_ptaki[1:2,])
## [{"gatunek":"Tygrys","waga":300,"dlugosc":2.5,"predkosc":60,"
    habitat":"Azja","zywotnosc":25,"druzyna":"Kot"},
## {"gatunek":"Lew","waga":200,"dlugosc":2,"predkosc":80,"habitat":"
    Afryka","zywotnosc":29,"druzyna":"Kot"}]
```

2.6.5. Wczytywanie danych w formatach HTML i XML

Źródłem danych mogą być też strony WWW. Treść takich stron jest opisana w formacie HTML.

Format HTML jest formatem tekstowym opartym na znacznikach, podobnie jak format XML. Oba te formaty są do siebie podobne. Główną różnicą jest to, że pierwszy wykorzystuje się do opisu wyglądu (strony internetowej), a drugi do opisu treści (dokumentu).

Aby wygodnie pracować z danymi w formacie XML, można użyć pakietu `xml2`. Aby wygodnie pracować z danymi w formacie HTML, można użyć pakietu `rvest`, który w dużej części korzysta z `xml2`.

Dane w formatach HTML i XML mają strukturę drzewiastą. Zazwyczaj praca z takimi danymi sprowadza się do określenia wzorca, który zlokalizuje wszystkie pasujące węzły dokumentu. Następnie na takiej kolekcji wykonuje się dalsze przetwarzanie, np. wyłuskiwanie atrybutów/treści z węzłów.

Poniżej przedstawiamy przykład. Używając pakietu `rvest`, wczytujemy stronę w formacie HTML. Funkcją `html_nodes()` wybieramy tylko te węzły, które mają klasę `.gwt-filmPage`. W końcu ze znalezionych węzłów wyciągamy treść funkcją `html_text()`.

```
library("rvest")
premiery <- read_html("http://www.filmweb.pl/premiere")
filmy <- html_nodes(premiery, ".gwt-filmPage")
html_text(filmy)
##  [1] "Carol"
##  [2] "Londyn w ogniu"
##  [3] "Historia Roja"
##  [4] "Zoolander 2 "
```

2.6.6. Inne formaty plików tekstowych

Bardziej elastyczną funkcją do odczytywania danych (niekoniecznie tabelarycznych) jest funkcja `scan(utils)`. Dane nie muszą być w postaci tabelarycznej, w każdym wierszu może być różna liczba wartości. Wynikiem funkcji `scan()` jest wektor odczytanych wartości.

```
wektor.wartosci <- scan("nazwa.pliku")
```

Funkcja `scan()` ma równie wiele parametrów co `read.table()`. Najprzydatniejsze z nich to `what` (określający typ odczytywanych wartości), `sep` określający separator dla kolejnych wartości oraz `nmax` umożliwiający określenie maksymalnej liczby wartości do odczytania. Poniższy przykład pozwala odczytać wektor łańcuchów znaków rozdzielonych przecinkami, bez ograniczeń na maksymalną liczbę odczytanych wartości.

```
wektor.lancuchow <- scan("nazwa.pliku", what="character", sep=",")
```

Domyślnie dane odczytywane są ze standardowego strumienia wejściowego, czyli bezpośrednio z konsoli (domyślna wartość parametru `file = ""`). Dzięki temu możemy w skryptach R, wraz z kodem źródłowym umieszczać nieduże zbiory danych. Poniższy skrypt inicjuje macierz `mat` wskazanymi 20 liczbami.

```
dane <- scan()
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4

## Read 20 items
mat <- matrix(dane, 4, 5, byrow=TRUE)
mat[1,]
## [1] 1 2 3 4 1
```

Funkcja `scan()` umożliwia wprowadzanie wartości bezpośrednio z konsoli. Kopiując powyższy kod do programu R, do wektora dane zostanie wprowadzony wektor dwudziestu liczb. Dane wczytane przez funkcję `scan()` są odczytywane jako wektor, który zamieniamy na macierz.

Jeżeli dane zapisane są w pliku tekstowym w blokach o stałych szerokościach pól, to można je odczytywać funkcją `read.fwf()` (gdzie `fwf` to skrót od ang. *fixed width field*). Pierwszy argument tej funkcji (argument `file`) wskazuje nazwę pliku do odczytania, drugi (argument `widths`) jest specyfikacją szerokości kolejnych pól. Tę specyfikację możemy zadać jedną dla całego pliku lub osobną dla każdej linii.

Poniżej dwa przykłady wywołania funkcji `read.fwf()`. W zależności od podanych parametrów, odczytamy z tego samego pliku różne ramki danych. W pierwszym przykładzie specyfikujemy jeden wzorzec dla całego pliku, w drugim osobny wzorzec dla każdej z linii. Załóżmy, że pracujemy z plikiem tekstowym `daneFWF.txt` o następującej zawartości.

```
110 ALA STOP13
111 OLA STOP 5
```

Podajemy format szerokości kolumn wspólny dla wszystkich linii.

```
dane <- read.fwf("http://www.biecek.pl/R/dane/daneFWF.txt",
                 widths=c(1,2,4,5,2))
dane
##   V1 V2   V3    V4 V5
## 1  1 10  ALA  STOP 13
## 2  1 11  OLA  STOP  5
```

Podajemy format dla każdej linii osobno.

```
(dane <- read.fwf("http://www.biecek.pl/R/dane/daneFWF.txt",
                  widths=list(c(1,2,11), c(2,1,11))))
##   V1 V2          V3 V4 V5          V6
## 1  1 10  ALA STOP13 11  1  OLA STOP 5
```

2.6.7. Wczytywanie danych z plików Excela

Jedną z najbardziej popularnych form przechowywania danych tabelarycznych są pliki Excela. Poniżej pokażemy jak takie dane odczytać z pliku Excela bezpośrednio do programu R.

Jako przykład wykorzystamy dane udostępniane przez Główny Urząd Statystyczny (GUS) w Banku Danych Lokalnych (BDL) [55]. Przez interface www można wyklinać jakie wskaźniki w jakich grupach chcemy pobrać, a następnie możemy pobrać je w formacie pliku Excela. Za pomocą strony BDL pobrano dane dotyczące ludności Polsce w latach 1995:2015 w podziale na grupy wiekowe i płeć. Dane ze strony `http://biecek.pl/R/dane/GUS_LUDN.xlsx`.

Pliki Excela pobrane z BDL mają trzy zakładki. W pierwszej opisane są główne informacje dotyczące danych. Druga zakładka to dane w postaci wąskiej (patrz rozdział 2.5.7). Trzecia zakładka to dane w postaci szerokiej, gdzie kolejne kolumny odpowiadają kolejnym latom. Na rysunku 2.19 umieszczone są zrzuty ekranu dla zakładek z danymi.

W programie R dostępnych jest kilka rożnych pakietów pozwalających na wczytywanie danych z Excela. Na dziś dzień, dla plików w formacie `xlsx` najbardziej uniwersalnym rozwiązaniem jest pakiet `openxlsx`.

Funkcja `read.xlsx()` pozwala na wczytanie danych z określonego pliku. Możemy dodatkowo określić z której zakładki chcemy wczytać dane.

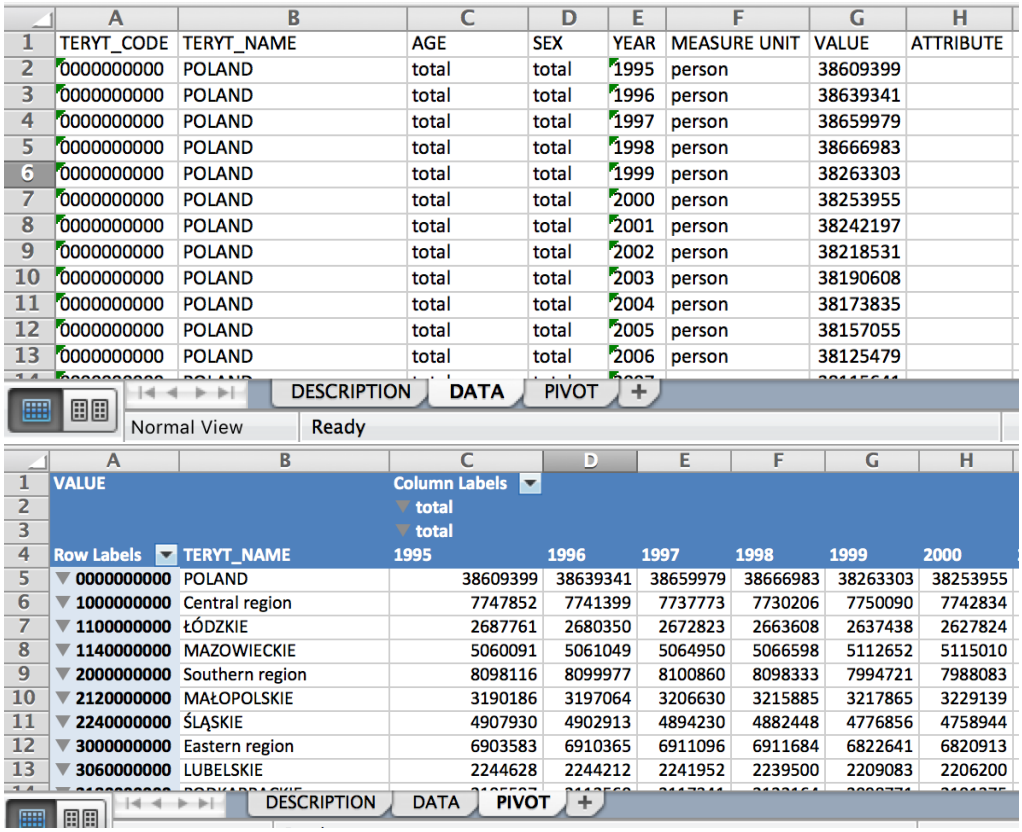
Zacznijmy od wczytania danych w postaci wąskiej z drugiej zakładki.

```
library("openxlsx")
ludnosc <- read.xlsx("GUS_LUDN.xlsx", sheet = 2)
```

Możemy teraz podejrzeć jakie dane się wczytały.

```
head(ludnosc)
##   TERYT_CODE TERYT_NAME   AGE   SEX YEAR MEASURE.UNIT    VALUE
    ATTRIBUTE
## 1 0000000000     POLAND total total 1995       person 38609399
## 2 0000000000     POLAND total total 1996       person 38639341
## 3 0000000000     POLAND total total 1997       person 38659979
## 4 0000000000     POLAND total total 1998       person 38666983
## 5 0000000000     POLAND total total 1999       person 38263303
```

W podobny sposób możemy odczytać dane z zakładki 3. Poniżej wyświetlamy 6 pierwszych kolumn.



	A	B	C	D	E	F	G	H
1	TERYT_CODE	TERYT_NAME	AGE	SEX	YEAR	MEASURE UNIT	VALUE	ATTRIBUTE
2	0000000000	POLAND	total	total	1995	person	38609399	
3	0000000000	POLAND	total	total	1996	person	38639341	
4	0000000000	POLAND	total	total	1997	person	38659979	
5	0000000000	POLAND	total	total	1998	person	38666983	
6	0000000000	POLAND	total	total	1999	person	38263303	
7	0000000000	POLAND	total	total	2000	person	38253955	
8	0000000000	POLAND	total	total	2001	person	38242197	
9	0000000000	POLAND	total	total	2002	person	38218531	
10	0000000000	POLAND	total	total	2003	person	38190608	
11	0000000000	POLAND	total	total	2004	person	38173835	
12	0000000000	POLAND	total	total	2005	person	38157055	
13	0000000000	POLAND	total	total	2006	person	38125479	

	A	B	C	D	E	F	G	H
1	VALUE		Column Labels					
2			total					
3			total					
4	Row Labels	TERYT_NAME	1995	1996	1997	1998	1999	2000
5	0000000000	POLAND	38609399	38639341	38659979	38666983	38263303	38253955
6	1000000000	Central region	7747852	7741399	7737773	7730206	7750090	7742834
7	1100000000	ŁÓDZKIE	2687761	2680350	2672823	2663608	2637438	2627824
8	1140000000	MAZOWIECKIE	5060091	5061049	5064950	5066598	5112652	5115010
9	2000000000	Southern region	8098116	8099977	8100860	8098333	7994721	7988083
10	2120000000	MAŁOPOLSKIE	3190186	3197064	3206630	3215885	3217865	3229139
11	2240000000	ŚLĄSKIE	4907930	4902913	4894230	4882448	4776856	4758944
12	3000000000	Eastern region	6903583	6910365	6911096	6911684	6822641	6820913
13	3060000000	LUBELSKIE	2244628	2244212	2241952	2239500	2209083	2206200

Rysunek 2.19. Podgląd zakładek w pliku Excela `GUS_LUDN.xlsx` ze strony `http://biecek.pl/R/dane/GUS_LUDN.xlsx`

```
ludnoscSzeroka <- read.xlsx("GUS_LUDN.xlsx", sheet = 3)
ludnoscSzeroka[3:8,1:6]
##         VALUE               X2 Column.Labels        X4        X5
## 3 Row Labels        TERYT_NAME          1995      1996      1997
## 4 0000000000           POLAND      38609399 38639341 38659979
## 5 1000000000   Central region       7747852  7741399  7737773
## 6 1100000000          LODZKIE       2687761  2680350  2672823
## 7 1140000000      MAZOWIECKIE       5060091  5061049  5064950
```

2.6.7.1. Inne pakiety do wczytywanie danych z plików Excela

Powyżej przedstawiliśmy pakiet `openxlsx`, ale w programie R dostępnych jest więcej pakietów pozwalających na wczytywanie danych z plików Excela. Może się zdarzyć, że z uwagi na dostępność zewnętrznych bibliotek, na któryś systemie operacyjnym pewne pakiety nie będą działały, lub nie będą w stanie poprawnie wczytać określonych danych.

Tablica 2.4. Wybrane funkcje do wczytywania danych z plików Excela

pakiet	funkcja_odczyt	funkcja_zapis	formaty	wymaga
readxl	read_excel()	—	.xlsx, .xls	
openxlsx	read.xlsx()	write.xlsx()	.xlsx	
xlsx	read.xlsx()	write.xlsx()	.xlsx, .xls	Java
XLConnect	readWorksheet()	writeWorksheet()	.xlsx, .xls	Java
gdata	read.xls()	–	.xlsx, .xls	Perl
WriteXLS	–	WriteXLS()	.xlsx, .xls	Perl

Zestawienie najpopularniejszych prezentuje tabela 2.4. Część z pakietów wymaga dodatkowych programów.

Wczytywanie danych z innych programów

Trzy najpopularniejsze źródła danych to pliki tekstowe, pliki Excela i bazy danych. Jednak w pracy z danymi często przychodzi korzystać z danych zapisanych w innych formatach takich jak SAS, Matlab czy SPSS. W kolejnych punktach przedstawimy instrukcje, pozwalające wczytać dane również z takich programów.

2.6.8. Wczytywanie danych z SPSS'a

Jeżeli dane są zapisane w formacie programu SPSS, to aby je wczytać potrzebny jest pakiet `Hmisc`. W pakiecie `Hmisc` znajduje się wiele funkcji do importu danych zapisanych w plikach binarnych w formacie różnych programów. Do importu formatu SPSS służy funkcja `Hmisc::spss.get()`.

Poniżej odczytujemy i wyświetlamy dane z formatu programu SPSS.

```
library("Hmisc")
dane  <- spss.get("http://www.biecek.pl/R/dane/daneSPSS.sav",
    datevars="urodzony")
head(dane, 4)
##   wiek waga diagnoza   urodzony
## 1   18   62 zdrowy   1989-10-07
## 2   19   66 zdrowy   1988-01-01
## 3   21   98 chory    1986-11-06
## 4   28   74 chory    1979-12-01
```

W wyniku wykonania tego polecenia do ramki danych zostaną wczytane dane (dokładniej informacje z widoku `Data View` programu SPSS). Aby mieć dostęp do danych z widoku `Variable View` należy użyć atrybutów poszczególnych kolumn. Przykładowo, aby odczytać informacje o opisie danej kolumny (pole `Label` w widoku `Variable View`) możemy wykonać następujące polecenia:

Opis kolumny `wiek` w ramce danych.

```
attributes(dane$wiek)$label
##                          wiek
## "Wiek osoby w chwili badania"
```

Jak wyżej, tyle że funkcją label z pakietu `Hmisc`.

```
label(dane$wiek)
##                          wiek
## "Wiek osoby w chwili badania"
```

Rozbudowane podsumowanie zmiennej `wiek`.

```
describe(dane$wiek)
## dane$wiek : Wiek osoby w chwili badania
##       n missing  unique    Mean
##      11       0       9   22.09
##
##            16 18 19 20 21 24 25 27 28
## Frequency   1  1  1  1  2  2  1  1  1
## %           9  9  9  9 18 18  9  9  9
```

Wiele niskopoziomowych funkcji do odczytu plików w formatach innych programów, znajduje się w pakiecie `foreign`. Jest tam np. funkcja `read.spss()`, również odczytująca dane z formatu SPSS. Funkcje z pakietu `foreign` są wykorzystywane przez omawiany w tym rozdziale pakiet `Hmisc`.

2.6.9. Wczytywanie danych z programu Matlab

Pakiety programu R związane z programem Matlab to `matlab` (emulator Matlaba) i `R.matlab` (pakiet do komunikacji pomiędzy programem R a Matlabem, również do odczytywania i zapisywania plików w formacie MAT). Szczegółowej dokumentacji tych pakietów należy szukać na stronach CRAN. Poniżej podamy przykład jak odczytać pliki zapisane w formacie MAT. Wykorzystamy do tego funkcję `R.matlab::readMat()`. Odczytuje ona pliki MAT zapisane w wersjach V4, V5 i V6. Dane są odczytywane jako lista, której kolejne pola odpowiadają kolejnym zmiennym zapisanym w pliku MAT.

Odczytujemy i wyświetlamy strukturę pliku w formacie Matlaba.

```
library("R.matlab")
daneMat <- readMat("http://www.biecek.pl/R/dane/daneMatlab.MAT")
str(daneMat)
## List of 2
##  $ normalny   : num [1:10, 1:10]  0.1352 -0.1390 -1.1634 ...
##  $ wykladniczy: num [1:10, 1:10] 0.540 1.097 0.837 ...
##  - attr(*, "header")=List of 3
##   ..$ description: chr "MATLAB 5.0 MAT-file, Platform: PCWIN,
    Created on: Sat Nov 03 12:59:22 2007 "
```

W tym pliku zapisane są zmienne o nazwach `normalny` i `wykladniczy`.

```
dim(daneMat$normalny)
## [1] 10 10
```

Osoby korzystające z Matlaba zainteresują się interfejsem `RMatlab`, pozwalającym na wywoływanie komend R z poziomu Matlaba i komend Matlaba z poziomu R. Więcej informacji na ten temat można znaleźć w [45].

2.6.10. Wczytywanie danych z SAS

Jest kilka sposobów na odczytanie danych zapisanych w formacie SAS. Najłatwiejszym jest skorzystanie z funkcji `foreign::read.ssd()`, np. w następujący sposób:

```
library("foreign")
dane <- read.ssd("http://www.biecek.pl/R/dane","daneSAS.sas7bdat")
```

Minusem tego rozwiązania jest konieczność posiadania zainstalowanego SASa. Funkcja `read.ssd()` uruchamia skrypt SASa, korzystający z procedury `PROC COPY` konwertującej dane na bardziej przenośny format, który następnie jest wczytywany do programu R. Aby tę procedurę wywołać, trzeba mieć zainstalowanego SASa.

Inne możliwe sposoby odczytywania danych z SASa to użycie jednej z funkcji `Hmisc::sas.get()`, funkcji `foregin::read.xport()`, lub funkcji `sas7bdat::read.sas7bdat()` (nie wymaga instalacji SASa). Można też wykorzystać instrukcje programu SAS opisujące formatowanie pliku z danymi, tak by wykorzystując SASową definicję tzw. DATA STEP wczytać dane do R. Do tego służą funkcje `SAScii::read.SAScii()` (wczytuje dane) i `SAScii::parse.SAScii()` (parsuje skrypt SAS).

2.6.11. Inne funkcje do importu danych

W tabeli 2.5 znajduje się lista funkcji umożliwiających import danych zapisanych w formatach innych pakietów statystycznych. Nie ma potrzeby omawiania tych funkcji szczegółowiej, bowiem korzysta się z nich w sposób podobny do funkcji `read.table()`.

Tablica 2.5. Lista wybranych funkcji do importu danych z popularnych programów statystycznych. Za wyjątkiem ostatnich trzech funkcji, pozostałe pochodzą z pakietu `foreign`

`read.S()`	Wczytywanie danych w formacie S.
`read.arff()`	Wczytywanie danych w formacie Attribute-Relation File Format wykorzystywanym przez program Weka.
`read.dbf()`	Wczytywanie danych z pliku bazy danych DBF.

`read.dta()`	Wczytywanie danych z pliku w formacie Stata.
`read.epiinfo()`	Wczytywanie danych z pliku w formacie Epi (pliki .REC).
`read.mtp()`	Wczytywanie danych z pliku w formacie Minitab.
`read.octave()`	Wczytywanie danych z pliku w formacie Octave.
`read.spss()`	Wczytywanie danych z pliku w formacie SPSS.
`read.ssd()`	Wczytywanie danych z pliku w formacie SAS.
`read.systat()`	Wczytywanie danych z pliku w formacie Systat.
`read.xport()`	Wczytywanie danych w formacie SAS XPORT.
`sas.get()`	Wczytywanie plików w formacie SAS. Funkcja z pakietu `Hmisc`.
`spss.get()`	Wczytywanie plików w formacie SPSS. Funkcja z pakietu `Hmisc`.
`readMat()`	Wczytywanie plików Matlaba .MAT. Funkcja z pakietu `R.matlab`.

2.7. Automatyczne raporty, powtarzalne badania

W tym rozdziale pokażemy, jak w prosty sposób wygenerować raport z analizami do pliku HTML, PDF lub DOC. Takie raporty można wykorzystać w biznesie, aby tworzyć dla klientów zautomatyzowane cykliczne zestawienia, lub na własny użytek, aby razem z kodem R trzymać komentarze do tworzonego programu.

Logika opisanych poniżej rozwiązań jest trzykrokowa.

1. Tworzymy plik (najczęściej o rozszerzeniu `*.Rmd` lub `*.Rnw`). W tym pliku umieszczamy zarówno instrukcje w programie R, jak i opisy sformatowane w języku `markdown` lub LaTeX.
2. Plik jest przetwarzany przez parser pakietów `knitr` lub `Sweave`. Parser wykonuje instrukcje R i w dokumencie umieszcza również ich wynik. Wyniki kodu R i ich opis zapisywane są w pliku `*.md` lub `*.tex`.
3. Ostatnim krokiem jest użycie programów `pandoc` lub `pdflatex`, które budują raport w formacie HTML, PDF lub DOC.

Wszystkie te kroki wykonać można praktycznie jednym przyciskiem i poniżej pokażemy, jak to zrobić. Ale najpierw kilka słów o historii tego rozwiązania.

Angielski termin *literate programming* (przetłumaczony tutaj jako „programowanie objaśniające") został zaproponowany przez Donalda Knutha, autora między innymi języka TeX. Za tym terminem kryje się idea, którą można opisać w trzech punktach (zobacz również artykuły [20] i [42]):

- programy są bezużyteczne bez opisów/dokumentacji,
- opisy powinny wyjaśniać program, zwykłe komentarze w kodzie nie są wystarczające,
- przykładowe kody wykorzystane w opisach powinny być wykonywalne i kompletne.

W ostatnich latach programowanie objaśniające zyskało na popularności z uwagi na problem powtarzalnych badań (ang. *reproducible research*). W czasach gdy publikuje się coraz więcej i coraz mniejsza jest kontrola recenzenta nad pracą, coraz ważniejsze staje się zapewnienie, że wyniki z naukowych prac można powtórzyć w przyszłości.

Poniżej przedstawiamy zarówno pakiet `knitr`, jak i `Sweave`. Pakiet `knitr` uważany jest za następcę `Sweave`. Jest łatwiejszy w użyciu, pozwala na wyprodukowanie plików w formatach PDF, HTML raz w wielu innych (pośrednio dzięki wykorzystaniu programu `pandoc`). Zaletą pakietu `Sweave` jest większa kontrola nad procesem przetwarzania. Poniżej prezentujemy najpierw proste zastosowanie pakietu `knitr`, następnie pakietu `slidify`, później napiszemy więcej o pakiecie `Sweave`.

2.7.1. Pakiet `knitr`, markdown a raporty w HTML

Punktem wyjściowym jest przygotowanie pliku, w którym znajdą się fragmenty kodu w języku R wraz z opisami w języku naturalnym. Opisy te chcemy formatować, wytłuszczać, powiększać, tworzyć na ich podstawie rozdziały, podrozdziały, dodawać tabele czy rysunki.

Jednym z najprostszych do nauczenia się języków opisujących formatowanie tekstu jest `markdown`. Z uwagi na prostą składnię jest często wykorzystywany na forach internetowych. Do formatowania wystarczy użyć kilku prostych schematów.

- Aby wyróżnić jakiś element, należy otoczyć go z obu stron znakami `*`.
- Aby dodać tytuł rozdziału lub podrozdziału, wystarczy rozpocząć linię od znaków `#`, `##` lub `###`. In więcej znaków `#`, tym niższy poziom w hierarchii.
- Aby dodać wyliczenie, należy linię z każdym elementem wyliczenia rozpocząć od znaku `*`.
- Aby dodać link do strony WWW, wystarczy ten link wkleić z prefixem `http://` lub `https://`.

W procesie kompilacji tego pliku odpowiednio oznaczone fragmenty kodu R zostaną wykonane, a ich wynik zostanie dodany do opisu kodu. W ten sposób czytelnik ma 100% gwarancji, że rezultat, który jest prezentowany w opisie, powstał tylko i wyłącznie w wyniku wywołania wymienionych poleceń R.

Zwyczajowo pliki w których łączy się kod R i markdown, oznacza się rozszerzeniem `.Rmd`. Mając taki plik, przetwarza się go następnie, używając funkcji `knit()`. Ta funkcja wykonuje instrukcje programu R (można dołączać również kod w innych językach, np. python), a następnie zarówno instrukcje jak i ich wynik zapisuje z użyciem formatu markdown. Zwyczajowo pliki w formacie markdown mają rozszerzenie `.md`. W wyniku przetwarzania plik o takim rozszerzeniu będzie dodany do katalogu roboczego. Plik w formacie markdown jest transformowany do pliku z rozszerzeniem `.html`.

Jeżeli korzystamy z programu `RStudio`, to w oknie edytora dostępny jest przycisk `Knit HTML` (o ile rozszerzenie pliku to `*.Rmd`). Kliknięcie tego przycisku powoduje automatyczne skompilowanie raportu i wyświetlenie wynikowej strony w przeglądarce.

Na rysunku 2.20 przedstawiamy przykładowy kod źródłowy raportu. Rozpoczyna go niewielki nagłówek opisany w formacie YAML (ang. *YAML Ain't Markup Language*). Określa on tytuł dokumentu, autora, datę powstania i format pliku wyjściowego.

raport.Rmd

Knit HTML

```
---
title: "Przykładowy skrypt łączący markdown i R"
author: "Przemyslaw Biecek"
date: "2 March 2016"
output: html_document
---

**Wykres pudełkowy** jest bardzo popularną metodą prezentacji
zmienności pojedynczej zmiennej. Przedstawimy go na przykładzie
zbioru danych _iris_, który ma `r nrow(iris)` wierszy.

```{r fig.width=7, fig.height=6}
summary(iris$Petal.Length)
boxplot(Petal.Length ~ Species, data = iris)
```
```

Rysunek 2.20. Dokument źródłowy to mieszanina kodu R i markdown

raport.Rmd raport.md

Preview HTML

```
---
title: "Przykładowy skrypt łączący markdown i R"
author: "Przemyslaw Biecek"
date: "2 March 2016"
output: html_document
---

**Wykres pudełkowy** jest bardzo popularną metodą prezentacji
zmienności pojedynczej zmiennej. Przedstawimy go na przykładzie
zbioru danych _iris_, który ma 150 wierszy.
```r
summary(iris$Petal.Length)
```
```
```
Min. 1st Qu. Median   Mean 3rd Qu.   Max.
1.000   1.600   4.350   3.758   5.100   6.900
```
```r
boxplot(Petal.Length ~ Species, data = iris)
```

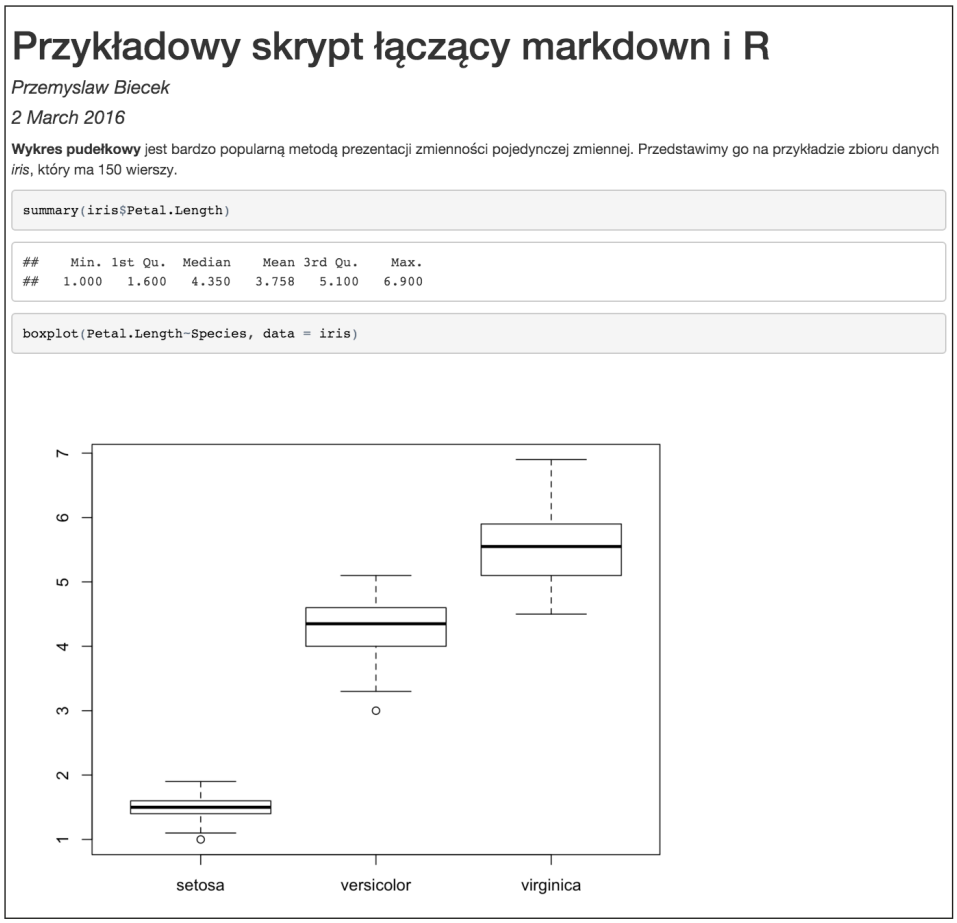
![plot of chunk unnamed-chunk-1](figure/unnamed-chunk-1-1.png)
```

Rysunek 2.21. Dokument pośredni zawiera tylko kod w formacie markdown
```

W tym przypadku format wyjściowy to `"output: html_document"`, dlatego plik zostanie przekonwertowany do formatu HTML. Aby otrzymać plik Worda jako wynik, wystarczy zmienić ten napis na `"output: word_document"`.

W wyniku przetworzenia pliku przez funkcję `knitr()` wygenerowany będzie plik `raport.md` o zawartości przedstawionej na rysunku 2.21.

W kolejnym kroku plik w formacie markdown jest konwertowany do pliku HTML za pomocą programu `pandoc`. Wynik przetworzenia przedstawiliśmy na rysunku 2.22. Format pliku wynikowego był umieszczony w nagłówku YAML. Jeżeli go zmienimy na `"output: pdf_document"` to przycisk `Knit HTML` w programie RStudio zamieni się na `Knit PDF`, a w wyniku kompilacji otrzymamy plik PDF'.



Rysunek 2.22. Wynikowy dokument w postaci strony HTML utworzony na bazie skryptu 2.20. Ten sam plik można przekonwertować do formatu Word lub PDF

2.7.1.1. Wstawki

Fragmenty kodu R, które mają być zamienione na wynik, będziemy nazywali wstawkami, ponieważ wyniki będą wstawione w wynikowy plik.

Aby parser wiedział, który fragment pliku jest kodem R, należy zastosować odpowiednie oznaczenia. Wyróżniamy tutaj dwie możliwości, nazywane krótką lub długą wstawką.

Krótka wstawka to instrukcje R pomiędzy ogranicznikami `` `r `` (dwa znaki) a `` ` `` (jeden znak). Zwróćmy uwagę, że znak `` ` `` to *grawis*, a nie prosty pojedynczy apostrof. W zależności od klawiatury znak `` ` `` znajduje się pod znakiem `Esc` lub obok lewego znaku `Shift`.

Instrukcje R w krótkiej wstawce zostaną wykonane, a ich wynik bezpośrednio wstawiony w miejsce tych instrukcji. Dlatego napis `` `r nrow(iris)` `` widoczny w przykładowym pliku `Rmd` został przekształcony w liczbę `150` w pliku `md`.

Długa wstawka to ciąg instrukcji otoczony dwoma liniami. Rozpoczęcie wstawki to trzy znaki `` ` `` i literka `r`. Zakończenie wstawki to trzy znaki `` ` ``.

```
```{r ...}
```
```

Po przetworzeniu długich wstawek w pliku wynikowym pojawią się bloki odpowiadające zarówno wejściowemu kodowi R, jak i rezultatom jego wykonania – tekstowym (te są poprzedzone znakami `##`) oraz graficznym (te umieszczane są w osobnych plikach PNG).

W linii rozpoczynającej długą wstawkę można dodać dodatkowe parametry. W przykładzie powyżej te parametry określały szerokość i wysokość wygenerowanego wykresu. Takich parametrów wykonania, które można zmieniać, jest znacznie więcej. Np. można wyłączyć raportowanie ostrzeżeń lub błędów. Dokładną listę znaleźć można w pozycji [42]. Najczęściej używane z parametrów opisaliśmy poniżej:

- Parametr `warning=TRUE|FALSE`, `message=TRUE|FALSE` określa, czy w wynikowym pliku `md` umieszczać komunikaty ostrzeżeń lub powiadomień (`TRUE`), czy nie (`FALSE`).
- Parametr `cache=TRUE|FALSE` określa, czy dana wstawka ma być za każdym razem wykonywana, czy też jej wyniki mają być zapisane i odtwarzane z cache'a. Domyślnie każda wstawka jest kompilowana za każdym razem. Jeżeli jednak obliczenia określonej wstawki trwają długo, a zawartość wstawki się nie zmienia, to warto włączyć korzystanie z cache'a.
- Parametr `echo=TRUE|FALSE` określa, czy w wynikowym pliku mają znaleźć się instrukcje z programu `R`. Ustawienie tego parametru na wartość `FALSE` powoduje, że w pliku HTML pojawią się tylko wyniki, bez kodu `R`. Wygodne, gdy chcemy ten kod z jakichś powodów ukryć przed końcowym odbiorcą.

- Parametr `results='asis'|'markup'|'hide'` określa w jakim formacie wynik ma być umieszczony w pliku `md`. Opcja `'markup'` (domyślna) powoduje przekształcenie wyniku do formatu markdown, `'asis'` powoduje wklejenie wyniku z kodu R bez żadnego dodatkowego formatowania, `'hide'` powoduje uruchomienie kodu R, ale wynik nie zostanie zawarty w pliku `md`.
- Parametr `tidy=TRUE|FALSE` określa, czy kod wejściowy powinien być dodatkowo „uporządkowany" przed umieszczeniem go w pliku wynikowym. Wartość `TRUE` dodaje wcięcia, spacje i dba o to, by kod wyglądał czytelniej.
- Parametr `highlight=TRUE|FALSE` określa, czy w wynikowym pliku kod R powinien być dodatkowo „kolorowany" (`TRUE`), czy nie (`FALSE`).
- Parametr `dev=pdf|png|...` określa, w jakim formacie mają być przechowywane wykresy wygenerowane przez kod R.
- Parametry `fig.width`, `fig.height`, `dpi` określają wymiary wykresów (w calach) i rozdzielczość (liczba punktów na cal).

Krótkich, jak i długich wstawek może być w plikach `.Rmd` dowolnie dużo.

2.7.2. Prezentacje w HTML5

Powyżej pokazaliśmy jak wykorzystać pakiet `knitr` do tworzenia raportów, które są jedną długą stroną html. Co jeżeli zamiast raportu chcemy przygotować prezentację składającą się z serii slajdów? Można użyć LaTeXowego pakietu `beamer` (wymaga to jednak znajomości LaTeX) lub `slidify` opisanego na stronie `http://slidify.org`. Od roku można też bardzo prosto robić prezentacje w programie RStudio. Poniżej opiszemy jak.

Treść slajdów prezentacji opisuje się w języku markdown, a same slajdy są generowane do formatu HTML5. Dokument źródłowy powinien być zapisany w pliku o rozszerzeniu `*.Pres`.

Aby oznaczyć, w którym miejscu ma rozpocząć się kolejny slajd należy umieścić tytuł slajdu, a po nim linię składającą się wyłącznie ze znaków

`==========`

Pracując w programie `RStudio` po utworzeniu pliku o rozszerzeniu `Rpres` z definicją prezentacji wystarczy przycisnąć przycisk `Preview` aby po kilku sekundach otrzymać prezentację.

Na rysunku 2.23 przedstawiony jest kod definiujący te cztery slajdy: tytułowy, ze spisem treści i dwa slajdy integrujący program R i opisy. Z jednej strony mamy proste instrukcje pozwalające na wskazanie nagłówków, treści w języku naturalnym, a z drugiej strony możemy dołączyć wyniki wygenerowane „na żywo" przez program R.

Ten kod źródłowy, należy zapisać w pliku z rozszerzeniem `Rpres`. Dynamiczne generowanie prezentacji pozwala na dużą kontrolę spójności wyników. Za każdym razem mamy możliwość sprawdzenia, jaki kod R został wykorzystany do otrzymania prezentowanych wyników. Unika się dzięki temu pro-

blemu z pamiętaniem, jaka wersja danych, jaka wersja parametrów doprowadziła akurat do tego konkretnego wyniku.

Rpres.Rpres

Preview Insert Run

```
Przykładowa prezentacja w HTML5
========================================================
author: Przemysław Biecek
date: 7 X 2016
autosize: true

Plan prezentacji
========================================================

1. Pakiet Rpres - tekst
2. Pakiet Rpres - grafika

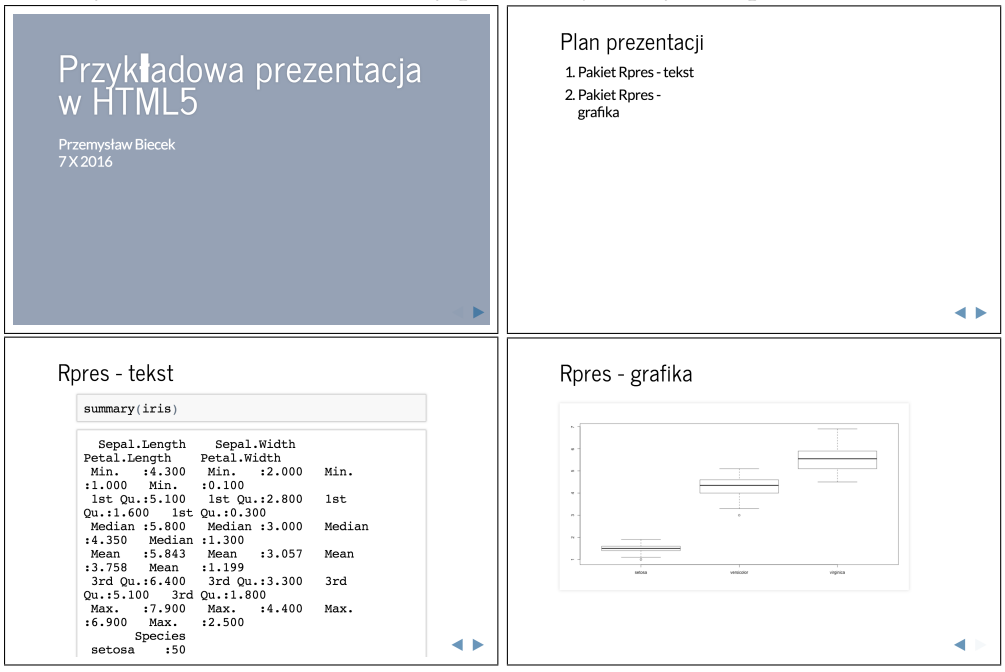
Rpres - tekst
========================================================

```{r}
summary(iris)
```

Rpres - grafika
==

```{r fig.width=14, fig.height=7.5, echo=FALSE, warning=FALSE}
boxplot(Petal.Length ~ Species, data = iris)
```
```

Rysunek 2.23. Kod źródłowy prezentacji z użyciem pakietu `Rpres`



Rysunek 2.24. Prezentacja wygenerowana z użyciem pakietu `Rpres` na podstawie kodu przedstawionego na rysunku 2.23

Pewną wadą prezentacji w formacie HTML5 jest to, że trudno je zamienić na `pdf` czy `pptx`. Zaletą jest szybkość ich tworzenia i integracja z kodem R produkującym wyniki umieszczone w prezentacji.

2.7.3. Pakiet `Sweave` a raporty w języku LaTeX

Aby ten podrozdział był zrozumiały wymagana jest przynajmniej podstawowa znajomość programu do składu dokumentów LaTeX.

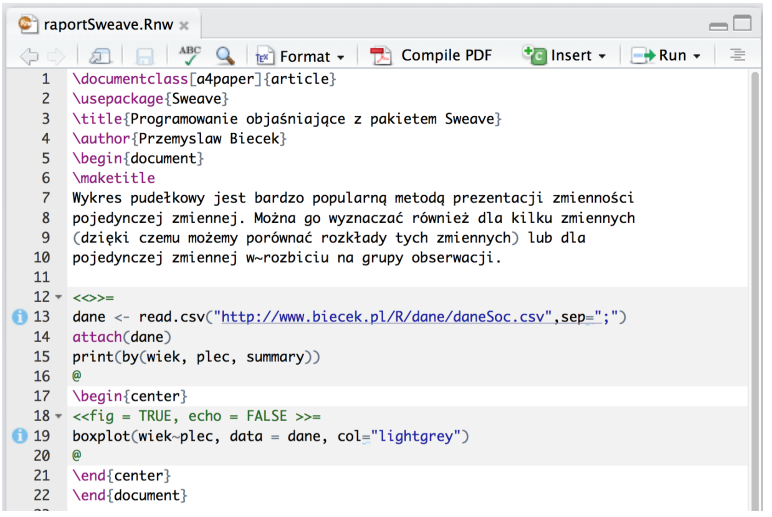
Zacznijmy od przykładu. Na rysunku 2.25 przedstawiona jest zawartość pliku `raportSweave.Rnw`. Jest to plik przypominający dokument w formacie LaTeX, ale w tym pliku znajdują się również instrukcje języka R.

Taki plik można przekształcić na raport na dwa sposoby, Korzystając z funkcji `Sweave` lub klikając przycisk *Compile PDF* w programie RStudio.

Wynikiem tego polecenia będzie plik `raportSweave.tex`. Fragmenty kodu R otoczone znacznikami `<<...>>=` i `@` zostały wykonane. Wyniki zostały wklejone w kod dokumentu LaTeX.

Następnie ten plik zostanie skompilowany poleceniem `pdflatex` do formatu `pdf`. Wynik przykładowego docelowego dokumentu jest przedstawiony na rysunku 2.26. Po uruchomieniu procesu kompilacji powinniśmy w konsoli zobaczyć następujące komunikaty.

```
Sweave("raportSweave.Rnw")
## Writing to file raportSweave.Rnw
## Processing code chunks ...
##  1 : echo term verbatim
##  2 : term verbatim eps pdf
```



Rysunek 2.25. Kod źródłowy dokumentu Sweave

Programowanie objaśniające ze Sweave

Przemyslaw Biecek

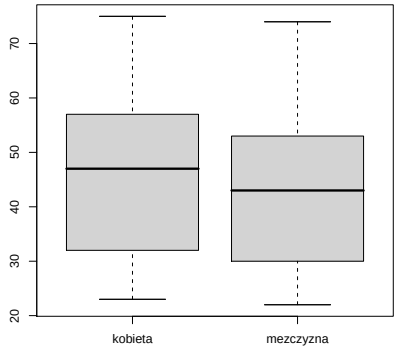
Wykres pudełkowy jest bardzo popularną metodą prezentacji zmienności pojedynczej zmiennej. Można go wyznaczać również dla kilku zmiennych (dzięki czemu możemy porównać rozkłady tych zmiennych) lub dla pojedynczej zmiennej w rozbiciu na grupy obserwacji.

Kształt tego wykresu jest bardzo charakterystyczny, przypominający pudełko z wąsami. Poszczególne elementy wykresu przedstawiają różne charakterystyki obserwowanej zmiennej. Środek pudełka przedstawia medianę, dolna i górna granica pudełka odpowiada kwartylom z próby (odpowiednio dolnemu i górnemu kwartylowi), kropki przedstawiają obserwacje odstające. Zakres zmienności (minimum i maksimum) danej zmiennej (po ominięciu wartości odstających) zaznaczony jest za pomocą wąsów wykresu.

```
> dane <- read.csv("http://www.biecek.pl/R/dane/daneSoc.csv", sep = ";")
> attach(dane)
> print(by(wiek, plec, summary))

INDICES: kobieta
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  23.00   32.00   47.00   46.38   57.00   75.00
------------------------------------------------------------
INDICES: mezczyzna
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  22.00   30.00   43.00   41.97   53.00   74.00

> boxplot(wiek ~ plec, data = dane, col = "lightgrey")
```



1

Rysunek 2.26. Strona dokumentu w formacie pdf otrzymana w sposób zautomatyzowany za pomocą pakietu Sweave

Ostatnia linia informuje o tym, że przetwarzanie pliku `raportSweave.Rnw` przebiegło bez problemów. W katalogu roboczym utworzonych zostało kilka nowych plików między innymi plik `raportSweave.tex`. To plik wynikowy, zapisany w języku LaTeX. W tym pliku został umieszczony opis oraz wyniki wykonania komend R znajdujących się w pliku `raportSweave.Rnw`.

Przyjrzyjmy się bliżej zawartości pliku `raportSweave.Rnw`. Wygląda on jak zwykły plik z kodem w LaTeXu, w którym występują fragmenty o składni:

```
<< argumenty >>=
kod R
@
```

Określając odpowiednie argumenty wstawek, możemy określać, w jaki sposób wyniki kodu R zostaną wklejone do pliku `tex`.

Jeżeli wynikiem poleceń R jest wykres, to aby znalazł się on w pliku wynikowym, należy za argument wstawki podać `fig=true`. W jednej wstawce możemy wyświetlić dowolnie wiele wykresów. Wynik każdego z nich zostanie przetworzony do dwóch plików: jednego w formacie `pdf` i jednego w formacie `eps`. Do pliku wynikowego automatycznie zostanie dodana odpowiednia instrukcja LaTeX dołączająca plik z rysunkiem.

Jeżeli spodziewamy się, że wykonywana we wstawce funkcja R wyprodukuje kod w formacie LaTeX (a więc nie jest potrzebna dodatkowa transformacja wyniku), to w danej wstawce powinniśmy podać argument `results=tex`. Taki argument powinien być podany np. jeżeli korzystamy z funkcji `xtable::xtable()`, służącej do wyświetlania tabel w formacie LaTeX.

Np. następująca wstawka

```
<< results=tex,echo=false >>=
library("xtable")
mat = matrix(1:25,5,5, dimnames=list(LETTERS[1:5], LETTERS[6:10]))
xtable(mat, caption="Tabela kolejnych liczb");
@
```

zostanie przetworzona na

```
% latex table generated in R 3.0.1 by xtable 1.5-2 package
% Thu Jul 10 11:13:41 2010
\begin{table}[ht]
\begin{center}
\begin{tabular}{rrrrrr}
    & F   & G   & H   & I   & J   \\  \hline
  \hline
  A &   1 &   6 &  11 &  16 &  21 \\
  B &   2 &   7 &  12 &  17 &  22 \\
  C &   3 &   8 &  13 &  18 &  23 \\
  D &   4 &   9 &  14 &  19 &  24 \\
  E &   5 &  10 &  15 &  20 &  25 \\   \hline
```

```
\end{tabular}
\caption{Tabela kolejnych liczb}
\end{center}
\end{table}
```

Do konwersji obiektów R do języka LaTeX można wykorzystać funkcję `toLatex()` lub funkcję `toBibtex()` przygotowującą odpowiednią tekstową reprezentacje w języku LaTeX dla obiektu będącego jej argumentem.

Jeżeli chcemy w kodzie LaTeXa wykonać wyrażenie napisane z użyciem składni programu R, to zamiast używać wstawek można posłużyć się poleceniem `\Sexpr`. Poniżej prezentujemy przykład kodu, w którym po przetworzeniu przez `Sweave()` prawa strona wyrażenia (fragment w poleceniu `\Sexpr{...}`) zostanie zastąpiona wynikiem wyliczonym w programie R. Poniższy skrypt

```
$$\frac{1-log(3)}{\sqrt 5} = \Sexpr{(1-log(3))/5^.5}$$
```

w wyniku przetwarzania zostanie zamieniony na

```
$$\frac{1-log(3)}{\sqrt 5} = -0.0441007561757451$$
```

Jeżeli nasz wynikowy plik ma być bardzo duży, to możemy nie chcieć, by wyniki wszystkich wstawek umieszczane były w jednym, tym samym pliku.

Ustawienie we wstawce argumentu `split=TRUE` spowoduje, że wynik wykonanego kodu R zostanie zapisany w osobnym pliku o nazwie w postaci `plik-etykieta.tex`, gdzie etykieta powinna być określona w liście argumentów.

```
<<split=TRUE, label=komentarz.S>>=
Ten tekst trafi do innego pliku, ktory zostanie wlaczony do glownego
    pliku LaTeX.
@
```

Wynik z wykonania skryptów R umieszczany jest w pliku wynikowym w jednym z dwóch środowisk `Sinput` (dla instrukcji R) i `Soutput` (dla wyników tych instrukcji). Aby LaTeX potrafił rozpoznać te środowiska musi znać ich definicje. Znajdują się one w pliku `Sweave.sty`, który jest w katalogu pakietu `Sweave`.

2.7.3.1. Ilustracje / winietki

Programowanie objaśniające nadaje się świetnie do przedstawiania konkretnych rozwiązań/poleceń/funkcji w R. Opis rozwiązania połączony jest z kodem, który to rozwiązanie generuje, a również z wynikiem wykonania tego kodu. Czytelnik może więc szczegółowo porównać wynik z argumentami wywoływanych poleceń, mając pewność, że autor nie pominął jakiegoś ważnego parametru.

Takie przykładowe rozwiązania są dostępne w programie R w postaci ilustracji (ang. *vignette*). Do pakietów R, poza funkcjami, zbiorami danych i opisami, dołączane są również ilustracje rozwiązań określonych problemów. Ilustracja składa się z kodu R prezentującego rozwiązanie oraz pliku przedstawiającego wynik tego rozwiązania. Dostęp do ilustracji rozwiązań odbywa się z użyciem funkcji `vignette()`. Poniżej przedstawiamy przykładowe wywołania tej funkcji.

Wyświetlmy listę ilustracji dostępnych w zainstalowanych pakietach. Zobaczmy jedną konkretną ilustrację, dotyczącą wielowymiarowej oceny gęstości. Poniższe polecenie otworzy plik PDF z opisem funkcjonalności funkcji `ks::kde()` oraz opisem przykładów wykorzystania tej funkcji w programie R. Kolejne polecenie otworzy okno edytora z kodami w języku R, których wyniki zostały przedstawione w powyższej ilustracji.

```
vignette(all = TRUE)
library("ks")
ilustracja <- vignette("kde")
print(ilustracja)
edit(ilustracja, editor="internal")
```

Rozdział 3

Niezbędnik programisty

The point is, ladies and gentleman, that r programming – for lack of a better word – is good. R Programming is right. R Programming works. R Programming clarifies, cuts through, and captures the essence of the evolutionary spirit. R Programming, in all of its forms – R Programming for life, for money, for love, knowledge – has marked the upward surge of mankind. – Random R Hacker

```
rfoaas::greed('R Programming', 'Random R Hacker')
```

Dane są coraz większe i coraz bardziej złożone. Trudno jest dzisiaj być biegłym w analizie danych bez przyzwoitej umiejętności programowania. Tylko ono daje wystarczająco dużą elastyczność w przetwarzaniu danych.

Program R jest wyposażony w pełnoprawny język programowania, który umożliwia tworzenie bardzo rozbudowanych profesjonalnych rozwiązań programistycznych. W tym rozdziale przedstawiamy wybrane zaawansowane możliwości tego języka programowania. Po lekturze tego rozdziału czytelnik będzie gotów do tworzenia aplikacji internetowych, własnych pakietów oraz rozbudowanych funkcji. Kolejnym krokiem może być lektura zaawansowanych pozycji, takich jak [36].

W podrozdziale 3.1 omawiamy, jak tworzyć funkcje, pętle i instrukcje warunkowe w programie R. Opanowanie tego materiału jest niezbędne, by budować bardziej złożone rozwiązania. W podrozdziale 3.2 omawiamy, w jaki sposób korzystać z danych dostępnych w bazach danych. Pozwalają one wyjść poza największe ograniczenie R – konieczność przechowywania danych roboczych w pamięci RAM. W podrozdziale 3.3 opisujemy, jak budować aplikacje internetowe oparte na programie R. Omawiany pakiet `shiny` dosłownie zrewolucjonizował sposób, w jaki przygotowuje się obecnie rozwiązania wykorzystujące dane.

W drodze do poznawania coraz bardziej zaawansowanych możliwości programu R w podrozdziale 3.4 wskazujemy, jak tworzyć własne pakiety w programie R. W podrozdziale 3.6 omawiamy narzędzia do identyfikacji oraz na-

prawiania błędów. W podrozdziale 3.7 opisujemy natomiast narzędzia do przyśpieszania kodu. Kolejne podrozdziały poświęcone są innym przydatnym funkcjom.

3.1. Instrukcje sterujące

W trzech kolejnych podrozdziałach omawiamy, jak tworzyć funkcje w programie R, jak sterować kodem, a wiec jak budować instrukcje warunkowe, oraz przedstawiamy kilka różnych podejść do tworzenia pętli w programie R.

Po lekturze tego podrozdziału czytelnik będzie umiał tworzyć całkiem złożone programy.

3.1.1. Jak tworzyć funkcje

W większych programach pewne fragmenty kodu często powtarzają się wielokrotnie i/lub są używane w różnych miejscach. Czasem należy wykonać pewien schemat instrukcji, być może z niewielką różnicą w argumentach. W takich sytuacjach wygodnie jest napisać funkcję, która uprości program i zwiększy jego czytelność.

Generalnie rzecz biorąc, zamykanie kodu w funkcjach jest dobrym zwyczajem, zwiększa to czytelność i zazwyczaj skraca kod, a tym samym wpływa na zmniejszenie liczby błędów, które zawsze gdzieś w kodzie się znajdą.

Istnieje wiele wskazówek, jak pisać dobre funkcje, elastyczne, z możliwością ponownego wykorzystania, robiące jedną rzecz, a więc łatwe do opisania i zapamiętania, do czego dana funkcja służy. Ponieważ jednak nie jest to podręcznik do programowania, poprzestanę na dwóch uwagach. Funkcje powinny być tak tworzone, by nie korzystały ze zmiennych globalnych (parametry do działania powinny być przekazane poprzez argumenty, używanie zmiennych globalnych najczęściej prowadzi do trudnych w wykryciu błędów), dzięki temu stają się izolowane i łatwiej je wykorzystywać w różnych kontekstach, różnych programach. Funkcje powinny być możliwie krótkie, ułatwi to ich modyfikacje i śledzenie poprawności, łatwiej też zrozumieć, co dana funkcja robi. Jeżeli jakaś funkcja znacznie się rozrosła, to z pewnością można i warto podzielić ją na mniejsze funkcje.

Z punktu widzenia użytkownika każda funkcja posiada listę argumentów oraz tak zwane *ciało funkcji*. Ciałem nazywa się zbiór instrukcji, które mają być wykonane przy każdym wywołaniu funkcji.

Przyjrzyjmy się obu elementom na przykładzie funkcji dodającej pierwszy argument do kwadratu drugiego.

```
function(a, b) a + b^2
```

W nawiasach po słowie kluczowym `function` wymienione są argumenty funkcji. Po nawiasach znajduje się ciało, czyli sekwencja instrukcji określających, co funkcja ma robić. W ogólnym przypadku sekwencja instrukcji powinna być zamknięta w blok za pomocą nawiasów `{}`. Gdy mamy do czynienia tylko z jedną instrukcją, to te nawiasy nie są niezbędne i można je pominąć.

W programie R każda funkcja jest zwykłym obiektem, takim jak liczba czy napis. Jako obiekt może istnieć bez nazwy. Nawet gdy nie ma nazwy, można ją wywołać za pomocą operatora `()`, tak jak w poniższym przykładzie:

```
(function(a, b) a + b^2 )(1, 2)
## [1] 5
```

Gdy po definicji funkcji pojawił się napis `(1, 2)`, wtedy interpreter R wykonał tę funkcję z parametrami `a=1` i `b=2`.

Funkcje mogą przekazywać (potocznie mówiąc: zwracać) wartości. Za wynik funkcji przyjmowana jest wartość wyznaczona w ostatniej linii ciała funkcji. Innym sposobem przekazywania wartości jest wykorzystanie instrukcji `return()`. Powoduje ona przerwanie wykonywania funkcji oraz przekazanie jako wyniku wartości będącej argumentem polecenia `return()`.

3.1.1.1. Nazywanie funkcji

Tak jak widzieliśmy, funkcja nie musi mieć nazwy. Znacznie prościej się jednak z niej korzysta, gdy ma nazwę. W tym celu wystarczy przypisać funkcję do dowolnego symbolu.

Nazwa funkcji, podobnie jak nazwa dowolnej zmiennej, musi rozpoczynać się literą, może zawierać litery, cyfry lub inne znaki. Najlepiej też, by nazwa funkcji wyjaśniała, co funkcja robi.

```
ab2 <- function(a, b) a + b^2
ab2(1, 2)
## [1] 5
```

Gdy ciało funkcji jest dłuższe niż jedna instrukcja, to ciąg instrukcji należy zamknąć w blok nawiasów klamrowych. Wynikiem funkcji jest wynik wskazany przez polecenie `return()` lub wynik ostatnio wykonanej operacji.

```
ab2 <- function(a, b) {
  a + b^2
}
```

Z punktu widzenia R funkcja jest takim samym obiektem jak każdy inny obiekt. Nazwa funkcji nie jest związana z jej definicją, a jedynie z nazwą zmiennej, w której ta funkcja jest zapamiętana.

3.1.1.2. Argumenty funkcji

Funkcja nie musi mieć argumentów. Przykładem użytecznej funkcji niepotrzebującej żadnego argumentu jest `Sys.time()`. Jako wynik zwraca ona aktualny czas.

```
Sys.time()
## [1] "2016-10-20 13:27:09 CEST"
```

Funkcje mogą mieć dowolną liczbę argumentów. Każdy argument musi mieć unikalną nazwę. Dla każdego argumentu możemy też wskazać wartość domyślną, czyli wartość, która będzie zastosowana, jeżeli dany argument nie zostanie użyty przy wywołaniu funkcji.

Przy wywołaniu funkcji można przez nazwę wskazać, który argument jest określany. Jeżeli wywołujemy funkcję z wartościami, ale bez podawania nazw argumentów, to wartości będą przypisane do kolejnych argumentów.

```
ab2 <- function(a = 1, b = 2) a + b^2
ab2(2)
## [1] 6
ab2(b = 1)
## [1] 2
ab2(b = 1, 2)
## [1] 3
```

Specjalnym argumentem, który można określić w funkcji, jest ...(trzy kropki). Jest on używany do opisu *wszystkich pozostałych argumentów*.

Zazwyczaj wykorzystuje się go, aby nowo tworzona funkcja mogła przekazać argumenty wywołania do jednej z funkcji w swoim ciele.

Przykładem jest poniższa wersja funkcji `paste0()`, która wywołuje funkcję `paste()` i przekazuje do niej wszystkie argumenty, a dodatkowo określa argument `sep`.

Funkcja `paste()` skleja argumenty w napis, stosując domyślnie jako separator spację. Jeżeli często wykonujemy funkcję `paste` z argumentem `sep=""`, to aby skrócić zapis, wygodnie jest opakować funkcję `paste` z ustawionym argumentem `sep=""` i przekazanymi wszystkimi pozostałymi argumentami.

```
paste0 <- function(...) paste(..., sep="")
paste0("Ala", "ma", "kota")
## [1] "Alamakota"
```

Poniżej deklarujemy funkcję z dwoma argumentami. Wywołujemy ją jawnie, wskazując wartości obu argumentów albo używając skrótów nazw.

```
funkcja <- function(liczba = 5, poziom = "sredni")
             cat(paste(liczba, poziom,"\n"))
funkcja(3, "duzy")
## 3 duzy
```

Nazwy argumentów funkcji potrafią być długie, jednak nie trzeba ich całych podawać! Zamiast pełnej nazwy argumentu wystarczy podać fragment nazwy, który jednoznacznie identyfikuje argument.

```
funkcja(po = "duzy")
## 5 duzy
funkcja(p = "maly", li = 1313)
## 1313 maly
```

3.1.1.3. Leniwe argumenty

W programie R argumenty są przypisywane jako wyrażenia, a nie przez wartość. Wartość przypisanego wyrażenia jest wyznaczana dopiero, gdy jest potrzebna. Mechanizm ten pozwala pominąć obliczenia, które nie są niezbędne.

Przyjrzyjmy się poniższemu przykładowi. Funkcja `skaluj` przyjmuje argumenty `center` i `scale`, które są wyznaczane jako wartość średnia i standardowe odchylenie z próby. Jednak te argumenty w ciele funkcji są wykorzystywane tylko, gdy argument `trans` ma wartość `TRUE`.

W przeciwnym wypadku nie są one używane i nawet nie są wyznaczane ich wartości (ponieważ nie są potrzebne). Dzięki temu skracamy czas obliczeń, nie wyznaczając wartości zbędnych argumentów.

```
skaluj <- function(x, trans=FALSE, center=mean(x), scale=sd(x)) {
  if (trans)   x <- (x - center)/scale
  x
}
```

Jednak osoby przyzwyczajone do pracy z językami stosującymi gorliwe wartościowanie, mogą być zaskoczone różnymi efektami ubocznymi leniwego wartościowania. Przyjrzyjmy się poniższemu przykładowi.

```
zagadkowa <- function(x = y) {
  y <- 2
  cat(x)
}
y <- 1
zagadkowa()
## 2
```

Funkcja `zagadkowa` z powyższego przykładu wyświetla wartość argumentu `x`. Argument ten jest określony domyślnie na wartość symbolu `y`.

W chwili gdy funkcja `zagadkowa` jest wykonywana, symbol `y` wskazuje na wartość `1`. Jednak wartość zmiennej `x` jest wyznaczana dopiero wtedy, gdy jest ona potrzebna, już wewnątrz funkcji `zagadkowa`. A w tej chwili zmienna `y` wskazuje już na wartość `2`. Dlatego wynikiem funkcji `zagadkowa` jest napis `2`.

3.1.1.4. Funkcja zwracająca jako wynik inną funkcję

Ponieważ w programie R funkcja jest zwykłą wartością, więc może być zarówno argumentem funkcji, jak i jej wynikiem.

Przyjrzyjmy się obu sytuacjom oddzielnie. Często wykorzystywaną funkcją jest `ecdf()`, czyli funkcja wyznaczająca empiryczną dystrybuantę. Ale empiryczna dystrybuanta jest funkcją!

```
library("Przewodnik")
dystrybuantaCen <- ecdf(mieszkania$cena)
class(dystrybuantaCen)
## [1] "ecdf"     "stepfun"  "function"
```

Skoro jest funkcją, to możemy ją wywołać, aby dowiedzieć się, jaka część mieszkań jest tańsza niż np. 200 000 PLN (ceny mieszkań w zbiorze danych `mieszkania` są z 2007 roku).

```
dystrybuantaCen(200000)
## [1] 0.665
```

Na poniższym przykładzie pokazujemy, jak stworzyć funkcje zwracającą funkcję jako wynik. W ciele funkcji `generatorPoteg` tworzona jest nowa funkcja. Można ją przypisać do symbolu i wywołać tak, jak każdą inną funkcję w programie R.

```
generatorPoteg <- function(wykladnik) {
  function(x) x^wykladnik
}
kwadrat <- generatorPoteg(2)
kwadrat(3)
## 9
```

3.1.1.5. Funkcja przyjmująca inną funkcję jako argument

Funkcja może być argumentem innej funkcji. Najczęstszym zastosowaniem tej możliwości jest prawdopodobnie instrukcja `lapply`, która jako argumenty przyjmuje listę i funkcję `FUN` oraz stosuje funkcję `FUN` do każdego elementu listy. Na poniższym przykładzie uruchamiamy funkcję `mean()` na każdej z pierwszych trzech kolumn ramki `mieszkania`.

```
lapply(mieszkania[,1:3], mean)
##
## $cena
## [1] 175934
## $pokoi
## [1] 2.55
## $powierzchnia
## [1] 46.197
```

Możemy sami stworzyć funkcję, która przyjmie inną funkcję jako argument. Przykładowo poniższa funkcja jako pierwszy argument przyjmuje generator liczb losowych i wywołuje go z wartością 10 oraz ewentualnymi dodatkowymi argumentami.

```
losuj10 <- function(FUN = rnorm, ...) {
  FUN(10, ...)
}
losuj10()
## [1]  0.65135429  0.28681211 -0.54293185  0.45658142
## [5] -0.68000616  1.05359267  0.93484140 -0.78962715
## [9]  0.08774471  0.32790392
losuj10(rpois, 1)
## [1] 0 2 1 1 2 1 0 1 1 1
```

3.1.2. Jak stosować instrukcje warunkowe

Poniżej przedstawione są instrukcje warunkowe, czyli polecenia pozwalające na wykonanie określonej listy instrukcji w zależności od tego, czy określony warunek jest prawdziwy, czy nie. Następnie opisane są pętle, czyli polecenia pozwalające na powtórzenie określonej listy instrukcji wielokrotnie.

3.1.2.1. Instrukcja warunkowa if... else...

W języku R, tak jak w większości języków programowania, mamy możliwość korzystania z instrukcji `if... else...`.

Umożliwia ona warunkowe wykonanie fragmentu kodu w zależności od prawdziwości pewnego warunku logicznego. Składnia instrukcji `if... else...` bez bloku `else` jest następująca:

```
if (warunek_logiczny) {
  blok_instrukcji_1
}
```

a z blokiem `else`:

```
if (warunek_logiczny) {
  blok_instrukcji_1
} else {
  blok_instrukcji_2
}
```

Jeżeli `warunek_logiczny` jest prawdziwy (wartość logiczna równa `TRUE` lub liczbowa różna od 0), to wykonany zostanie `blok_instrukcji_1`. W przeciwnym wypadku wykonany zostanie `blok_instrukcji_2` (o ile zastosowano wariant z `else`).

Jeżeli `blok_instrukcji_1` lub `blok_instrukcji_2` składa się tylko z jednej instrukcji, to można pominąć nawiasy klamrowe `{}`.

Zobaczmy, jak to wygląda na poniższym przykładzie.

Zapis: `liczba %% 2 == 0` oznacza sprawdzenie, czy reszta z dzielenia przez 2 ma wartość 0, jeżeli tak jest, to to wyrażenie przyjmuje wartość `TRUE`, w przeciwnym razie wartość `FALSE`.

```
liczba <- 1313
if (liczba %% 2 == 0) {
   cat("ta liczba jest parzysta\n")
} else {
   cat("ta liczba jest nieparzysta\n")
}
## ta liczba jest nieparzysta
```

Należy uważać, by słowo kluczowe `else` nie rozpoczynało nowej linii. Błędem zakończy się następujący ciąg poleceń:

```
if (1 == 0)
    cat("to nie moze byc prawda")
  else
    cat("wszystko ok")
```

Dlaczego? Jak pamiętamy, R to język interpretowany. Po zakończeniu przetwarzania drugiej linii tego przykładu interpreter nie spodziewa się dalszego ciągu instrukcji, dlatego wykona (w jego mniemaniu już kompletną) instrukcję warunkową. Przechodząc do trzeciej linii, o instrukcji `if` już nie pamięta, dlatego zostanie zgłoszony błąd składni. Poprawne użycie instrukcji `if` z wariantem `else` jest następujące:

```
if (1 == 0) {
    cat("to nie moze byc prawda")
  } else {
    cat("wszystko ok")
  }
```

3.1.2.2. Funkcja ifelse()

Powyżej omówiona instrukcja warunkowa bierze pod uwagę wartość tylko jednego warunku logicznego. Funkcja `ifelse()` pozwala na wykonanie ciągu działań w zależności od wektora warunków logicznych. Schemat użycia jest następujący:

```
ifelse(warunek_logiczny, instrukcja_1, instrukcja_2)
```

Warunek `warunek_logiczny` może być jedną wartością logiczną lub wektorem wartości logicznych. W wyniku wykonania funkcji `ifelse()` zwrócone zostaną wartość lub wektor.

Wynik będzie miał wartości opisane przez `instrukcja_1` w pozycjach odpowiadających wartości `TRUE` wektora `warunek_logiczny` oraz wartości opisane przez `instrukcja_2` w pozycjach odpowiadających wartości `FALSE` wektora `warunek_logiczny`.

Prześledźmy wyniki poniższych przykładów. Najpierw przykład, w którym tylko pierwszy argument jest wektorem:

```
ifelse(1:7 < 4, "mniej", "wiecej")
## [1] "mniej" "mniej" "mniej" "wiecej" "wiecej" "wiecej" "wiecej"
```

Tutaj wszystkie argumenty są wektorami:

```
ifelse(sin(1:5)>0, (1:5)^2, (1:5)^3)
## [1]   1   4   9  64 125
```

A tutaj wszystkie argumenty to pojedyncze wartości.

```
ifelse(1==2, "cos jest nie tak", "uff")
## [1] "uff"
```

3.1.2.3. Funkcja switch()

W przypadku omówionych powyżej instrukcji warunkowych mieliśmy do czynienia z warunkiem logicznym, który mógł być prawdziwy lub fałszywy. Jednak w pewnych sytuacjach zbiór możliwych akcji, które chcemy wykonać, jest większy. W takich sytuacjach sprawdza się instrukcja warunkowa `switch()`, o następującej składni:

```
switch(klucz, wartosc1 = akcja1, wartosc2 = akcja2, ...)
```

Pierwszy argument powinien być typu znakowego lub typu wyliczeniowego `factor`. W zależności od wartości tego argumentu jako wynik zostanie zwrócona wartość otrzymana w wyniku wykonania odpowiedniej akcji.

W poniższym przykładzie sprawdzamy, jaka jest klasa danej zmiennej, i w zależności od tego wykonujemy jedną z wielu możliwych akcji. W bloku `klucz` sprawdzana jest klasa zmiennej `liczba`. Wiemy, że ta klasa to `'numeric'`, ponieważ zmienna `liczba` przechowuje wartość 1313. Dlatego też zostanie wykonana wyłącznie druga akcja.

```
liczba <- 1313
switch(class(liczba),
   logical = ,
   numeric = cat("typ liczbowy lub logiczny"),
   factor = cat("typ czynnikowy"),
   cat("trudno okreslic")
)
## typ liczbowy lub logiczny
```

Jeżeli wartość klucza (pierwszego argumentu funkcji `switch()`) nie pasuje do etykiety żadnego z kolejnych argumentów, to wynikiem instrukcji `switch()` jest wartość argumentu nienazwanego (czyli domyślną akcją w powyższym przykładzie jest wyświetlenie napisu `"trudno okreslic"`). Jeżeli wartość klucza zostanie dopasowana do etykiety jednego z kolejnych argumentów, ale nie jest podana żadna związana z nim akcja, to wykonana zostanie akcja dla kolejnego argumentu. Innymi słowy, jeżeli klucz miałby wartość `"logical"`, to ponieważ nie jest wskazana wartość dla argumentu `logical`, zostanie wykonana akcja wskazana przy kolejnej z etykiet, czyli `"numeric"`.

Argumenty funkcji `switch()` korzystają z mechanizmu leniwego wartościowania (ang. *lazy evaluation*, więcej informacji nt. tego mechanizmu przedstawimy w podrozdziale 3.8.4). W językach z gorliwą ewaluacją, takich jak Java czy C++, przekazywanie argumentów funkcji polega na wyznaczeniu wartości kolejnych argumentów i na przekazaniu do funkcji wyłącznie wyznaczonych wartości. Gdyby tak było w powyższym przykładzie, to przed wywołaniem funkcji `switch()` określone byłyby wartości wszystkich argumentów, czyli wykonane byłyby wszystkie funkcje `cat()`.

3.1.3. Jak budować pętle

W tym rozdziale omawiamy różne podejścia do powtarzania określonych instrukcji kilkukrotnie, czyli do tworzenia tak zwanych *pętli*.

Bardzo często zachodzi potrzeba wykonania jakiejś operacji dla każdego modelu, każdego kraju lub określoną liczbę razy. W zależności jednak od tego, ilukrotnie chcemy wykonać zadaną operację, wygodniej będzie nam wykorzystać inny mechanizm R do tworzenia pętli.

Omawiamy te różne podejścia poniżej jedno po drugim. Do ilustracji, które podejście jest bardziej przydatne, wykorzystujemy przykład ubezpieczeniowy.

Poniżej przedstawiamy definicję funkcji `strata()`, która symuluje wysokość straty w określonym czsie. Najpierw z rozkładu Poissona wyznacza liczbę strat, a następnie zadaną liczbę strat losuje z rozkładu log-normalnego. Wynikiem funkcji jest suma strat.

```
strata <- function(lambda = 10, mu = 10) {
  l_strat <- rpois(1, lambda)
  straty <- rlnorm(l_strat, mu)
  sum(straty)
}
strata()
# [1] 356397.9
```

3.1.3.1. Funkcja `replicate()`

Funkcja `replicate(n, expr)` może być wykorzystana, by powtórzyć określony blok operacji (`expr`) zadaną liczbę razy (`n`).

Przykładowo, aby poznać rozkład strat, możemy chcieć powtórzyć proces losowania wysokości straty 10 000 razy. Z pomocą funkcji `replicate` można to zrobić w jednej linii. Wynikiem jest wektor wartości wyznaczonych przez kolejne wywołania funkcji `strata`.

```
straty <- replicate(10000, strata())
summary(straty)
## Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0  228600  332500  361800  459900 2678000
```

Funkcja `replicate` jest często wykorzystywana w przypadku symulacji, w prosty sposób pozwala na powtórzenie określonych instrukcji wskazaną liczbę razy. Jeżeli wykonywany kod jako wynik zwraca więcej niż jedną liczbę, to wynikiem funkcji `replicate` będzie macierz (jeżeli wyniki są tego samego typu) lub lista.

3.1.3.2. Pętla for

Funkcja `replicate` ma wiele zastosowań, ale nie można z jej pomocą wykonać wszystkiego. W jej przypadku każde powtórzenie przebiega w identycznych warunkach.

Często potrzebujemy pętli, by wykonać jakąś operację dla każdego elementu jakiegoś wektora. Na przykład dla każdego pliku na dysku lub dla każdej tabeli w bazie danych, lub dla każdej zmiennej w ramce.

W takich przypadkach wygodniej jest użyć pętli `for`. Ogólny schemat użycia tej pętli to:

```
for (iterator in kolekcja) {
  blok_instrukcji
}
```

Przy użyciu tej pętli `blok_instrukcji` będzie wykonany tyle razy, ile elementów znajduje się w obiekcie `kolekcja` (może być to wektor lub lista). Zmienna `iterator` w każdym okrążeniu pętli przyjmować będzie kolejną wartość z wektora lub listy `kolekcja`. Jeżeli `blok_instrukcji` składa się tylko z jednej instrukcji, to możemy pominąć nawiasy klamrowe.

Elementy wektora lub listy `kolekcja` mogą być dowolne, nie muszą być różne, mogą się powtarzać.

Na przykładzie poniżej wykorzystujemy funkcję `ls()`, aby wyznaczyć listę wszystkich zmiennych widzianych przez R. Następnie pętlą `for` dla każdego obiektu wyznaczamy jego wielkość (funkcją `object.size`) i wypisujemy ją na ekranie.

```
lista_obiektow <- ls()
for (obiekt in lista_obiektow) {
   wielkosc <- object.size(get(obiekt))
   cat("Obiekt ", obiekt ," zajmuje ", wielkosc, "bajtow.\n")
}
## Obiekt  auta2012  zajmuje  47101296 bajtow.
## Obiekt  lista_obiektow  zajmuje  224 bajtow.
## Obiekt  strata  zajmuje  7272 bajtow.
## Obiekt  straty  zajmuje  1600200 bajtow.
```

Możliwość iterowania po dowolnej kolekcji pętlą `for` bardzo często pozwala bardzo uprościć kod. Często jednak wygodnie jest mieć indeks poszczególnych kroków pętli, choćby po to, by zapisywać wyniki po drodze.

W takiej sytuacji najłatwiej jest wykorzystać funkcję `seq_along()` do wyznaczania indeksów określonego wektora lub listy:

```
lista_obiektow <- ls()
wielkosci <- numeric(length(lista_obiektow))
for (i in seq_along(lista_obiektow)) {
   obiekt <- lista_obiektow[i]
   wielkosci[i] <- object.size(get(obiekt))
   cat("Obiekt ", obiekt ," zajmuje ", wielkosci[i], "bajtow.\n")
}
wielkosci
## [1] 47101296     224   7272 1600200
```

Jeżeli wewnątrz pętli wystąpi błąd, przerywa on wykonywanie pętli. Aby sprawdzić, w którym kroku pojawił się błąd należy sprawdzić wartość zmiennej `iterator`.

Zazwyczaj podobny efekt do pętli `for` możemy uzyskać, stosując funkcje z rodziny `*apply()` (np. `lapply()`). Funkcje te są przedstawione w podrozdziale 3.9.1. Żadne z tych rozwiązań nie jest absolutnie lepsze, można więc korzystać z dowolnego z nich w zależności od tego, które jest łatwiej w konkretnej sytuacji zastosować (zapisać). Najczęściej jednak używając funkcji z rodziny `*apply()`, otrzymuje się bardziej elegancki zapis, w wielu przypadkach też wynik wyznaczony będzie szybciej, ponieważ R jest optymalizowany do pracy na wektorach.

W przykładzie powyżej używaliśmy funkcji `seq_along(x)`, by wyznaczyć sekwencję indeksów od 1 do długości wektora. Alternatywny zapis `1:length(x)` źle zadziała, jeżeli argument x ma zerową długość, zamiast pustego wektora otrzymujemy wektor o dwóch wartościach `c(1, 0)`. Lepiej więc używać funkcji `seq_along()`.

```
x <- NULL
seq_along(x)
## integer(0)
1:length(x)
## [1] 1 0
```

3.1.3.3. Pętla `while`

Zarówno w przypadku pętli `for`, jak i w przypadku funkcji `replicate()` liczba powtórzeń pętli jest ustalona. Zdarzają się jednak sytuacje, w których potrzebujemy bardziej elastycznego rozwiązania. Często można wtedy wykorzystać pętlę `while`, o szablonie przedstawionym poniżej:

```
while (warunek_logiczny) {
  blok_instrukcji
}
```

W tej pętli `blok_instrukcji` będzie wykonywany tak długo, jak długo prawdziwy pozostanie `warunek_logiczny`. Oczywiście należy zadbać o to, by taka sytuacja kiedykolwiek zaistniała, a więc by pętla kiedyś się zakończyła.

W poniższym przykładzie ponownie wykorzystujemy funkcję `strata()`. Będziemy ją wywoływać tak długo, aż nie otrzymamy straty wyższej niż 500 000. Dodatkowo w zmiennej `i` będziemy zliczać kroki.

```
x <- 0
i <- 1
while (x < 500000) {
   x <- strata()
   i <- i + 1
}
x
## [1] 640346.9
i
## [1] 7
```

Instrukcja `while()` może być bardzo przydatna, jeżeli mamy do czynienia z losowością i chcemy by zdarzył się pewien określony warunek.

3.1.3.4. Pętla `repeat`

Swoistą wadą powyższego wywołania instrukcji `while` jest konieczność zainicjowania zmiennej `x`, zanim warunek w pętli `while` będzie po raz pierwszy spełniony. Często można tego uniknąć, stosując pętlę `repeat`, o składni przedstawionej poniżej:

```
repeat {
    blok_instrukcji
}
```

Działanie tej pętli polega na powtarzaniu `blok_instrukcji` tak długo aż .... Właśnie, nie ma tu żadnego warunku stop! Działanie zostanie przerwane wyłącznie w wyniku wygenerowania błędu lub użycia instrukcji `break` (ta instrukcja przerywa wykonywanie wszystkich rodzajów pętli, również pętli `for` i `while`).

Przyjrzyjmy się poniższemu przykładowi:

```
repeat {
  x <- strata()
  if (x > 500000)
    break
}
x
## [1] 835052.8
```

Instrukcję `break` można wykorzystywać w każdym rodzaju pętli, podobnie w każdej pętli można wykorzystywać instrukcję `next`. Pierwsza przerywa działanie pętli, druga powoduje przerwanie wykonywania aktualnej iteracji pętli oraz przejście do kolejnej iteracji.

3.2. Jak pracować z bazami danych

Zdecydowana większość funkcji w programie R wymaga, aby dane na których pracujemy, były w pamięci RAM. Ale jeżeli chcemy korzystać z dużych zbiorów danych, które zajmują dużo miejsca, to mamy dwie możliwości:

- pracować na „tłustych" komputerach, z dużą ilością RAM (jednak obecnie trudno wyjść poza 1 TB RAM, chyba że ma się baaardzo duży budżet),
- trzymać dane poza R, możliwie dużo przetwarzania wykonywać poza R, a do R wczytywać tylko takie dane, które są naprawdę niezbędne.

W tym rozdziale przyjrzymy się tej drugiej możliwości. Trzymanie danych poza R oznaczać będzie przechowywanie ich w bazie danych. W takiej bazie możemy szybko wyznaczyć niezbędne agregaty lub pobrać losową próbkę do dalszych analiz.

Nawet jeżeli danych nie jest bardzo dużo, to warto rozważyć korzystanie z bazy. Pozwala to na jednolity sposób dostępu do danych z różnych narzędzi przez różne osoby. Umożliwia zarządzanie prawami dostępu do danych oraz zapewnia skalowalność operacji na danych.

Poniżej przedstawiamy dwa przykłady komunikacji R z bazami danych. Pierwszy przykład dotyczy prostej bazy danych SQLite. Cała baza jest jednym plikiem. Prezentujemy elementarne przykłady pozwalające na przećwiczenie podstawowych operacji. Drugi przykład dotyczy pracy z popularną bazą PostgreSQL. Sposób pracy z taką bazą jest bardzo podobny do większości popularnie stosowanych baz relacyjnych.

3.2.1. Jak pracować z bazą danych SQLite

SQLite (patrz [54]) to lekka baza danych przechowująca całe dane w jednym pliku. Ma ona dosyć ograniczone funkcjonalności, ale jej główną zaletą jest prostota użycia. Tak łatwo ją zainstalować, że wręcz trudno powiedzieć, w którym momencie się to robi. Ponieważ jest to jeden plik, łatwo tę bazę skopiować czy komuś wysłać. Dlatego pomimo ograniczonych możliwości ma ona sporo zastosowań, w których ważniejsza jest prostota niż skalowalność.

Aby użyć tej bazy danych w programie `R`, potrzebny jest pakiet `RSQLite`. Korzystanie z tej bazy składa się z poniższych kroków

- Należy wczytać sterownik do łączenia się z bazą danych. To można zrobić funkcją `dbDriver()`.
- Należy nawiązać połączenie z bazą danych, najczęściej funkcją `dbConnect()`. W przypadku bazy SQLite wystarczy wskazać ścieżkę do pliku z bazą danych.
- Do bazy danych wysyła się instrukcje oparte na języku SQL. W wyniku otrzymuje się tabele z danymi.
- Po zakończeniu pracy z bazą danych należy zwolnić połączenie (funkcja `dbDisconnect()`) i sterownik.

Przykładowa sesja z bazą danych jest następująca. Ładujemy sterownik do bazy danych i inicjujemy połączenie z serwerem. Jeżeli wskazany plik nie istnieje, to zostanie stworzona pusta baza danych. Plik wykorzystany w tych przykładach można pobrać ze strony `http://biecek.pl/R/zabawka.db`.

```
library("RSQLite")
sterownik <- dbDriver("SQLite")
polaczenie <- dbConnect(sterownik, "zabawka.db")
```

Wyświetlamy tabele widoczne w bazie danych pod wskazanym połączeniem, a następnie wypisujemy nazwy kolumn w określonej tabeli:

```
dbListTables(polaczenie)
## [1] "auta2012"     "sqlite_stat1" "wynik"
dbListFields(polaczenie, "auta2012")
##  [1] "Cena"                          "Waluta"
##  [3] "Cena.w.PLN"                    "Brutto.netto"
##  [5] "KM"                            "kW"
##  [7] "Marka"                         "Model"
##  [9] "Wersja"                        "Liczba.drzwi"
## [11] "Pojemnosc.skokowa"             "Przebieg.w.km"
## [13] "Rodzaj.paliwa"                 "Rok.produkcji"
## [15] "Kolor"                         "Kraj.aktualnej.rejestracji"
## [17] "Kraj.pochodzenia"              "Pojazd.uszkodzony"
## [19] "Skrzynia.biegow"               "Status.pojazdu"
## [21] "Wyposazenie.dodatkowe"
```

Używając funkcji `dbGetQuery()`, możemy wykonać na bazie zapytanie SQL:

```
pierwsze5 <- dbGetQuery(polaczenie,
    "select Cena, Waluta, Marka, Model from auta2012 limit 5")
pierwsze5
##    Cena Waluta         Marka     Model
## 1 49900    PLN           Kia    Carens
## 2 88000    PLN    Mitsubishi Outlander
## 3 86000    PLN     Chevrolet   Captiva
## 4 25900    PLN         Volvo       S80
## 5 55900    PLN Mercedes-Benz  Sprinter
agregat <- dbGetQuery(polaczenie,
    "select count(*) as liczba, avg(‘Cena.w.PLN‘) as cena, Marka from
         auta2012 group by Marka limit 10")
agregat
##    liczba      cena       Marka
## 1      24  32735.33
## 2      59  68140.39       Acura
## 3      50  27037.04       Aixam
## 4    2142  21403.58   AlfaRomeo
## 5       4  15424.42         Aro
## 6      35 505359.61 AstonMartin
## 7   12851  64608.61        Audi
## 8      17  46396.60      Austin
## 9   10126  72385.68         BMW
## 10     39 478483.82     Bentley
```

Używając funkcji `dbDisconnect()`, możemy się z bazą danych rozłączyć. Ważne jest, by po sobie posprzątać na wypadek, gdyby dane z pamięci nie zostały zapisane do pliku.

```
dbDisconnect(polaczenie)
## [1] TRUE
```

Funkcja `dbGetQuery()` tworzy zapytanie, wykonuje je i pobiera jego wyniki. Tę operację można rozbić na części. Funkcja `dbSendQuery()` tworzy i wysyła zapytanie SQL, a funkcja `fetch()` pobiera kolejne porcje danych.

Funkcja `dbWriteTable()` zapisuje wskazany obiekt `data.table` jako tabelę w bazie danych. Dwa użyteczne argumenty tej funkcji to `append` (dodaj dane do tabeli jako kolejne wiersze) i `overwrite` (nadpisz istniejącą tabelę).

3.2.2. Jak pracować z większymi relacyjnymi bazami danych

SQLite to baza zabawka. Do przechowywania większych danych w produkcyjnych rozwiązaniach wykorzystywać można otwarte rozwiązania typu PostgreSQL czy MySQL lub duże rozwiązania *big data* takie jak Oracle, RedShift, Teradata, Netezza i inne.

O ile te bazy różnią się funkcjonalnością, skalowalnością i prędkością, to z perspektywy użytkownika R używanie ich jest dosyć podobne. Poniżej pokażemy, jak korzystać z bazy PostgreSQL.

Na potrzeby przykładu użyjemy działającej bazy PostgreSQL dostępnej na serwerze `services.mini.pw.edu.pl`. PostgreSQL pozwala na zarządzanie wieloma użytkownikami i wieloma bazami danych, tutaj wykorzystamy bazę `sejmrp`, przechowującą dane z Sejmu RP 7. i 8. kadencji (głosowania i stenogramy). Dane te są uzupełniane za pomocą pakietu `sejmRP` [47].

Aby korzystać z bazy danych, potrzebny jest użytkownik i hasło. Poniżej prezentujemy przykład dla użytkownika `reader` i hasła `qux94874`. Ten użytkownik ma wyłącznie uprawnienia do czytania, można więc pracować z nim bez obaw, że coś zostanie usunięte.

Aby połączyć się z bazą `PostgreSQL` potrzebujemy sterownika, który jest dostępny w pakiecie `RPostgreSQL`. Wczytajmy ten pakiet i nawiążmy połączenie z bazą danych:

```
library("RPostgreSQL")
dbname     <- "sejmrp"
user       <- "reader"
password   <- "qux94874"
host       <- "services.mini.pw.edu.pl"
sterownik  <- dbDriver("PostgreSQL")
polaczenie <- dbConnect(sterownik, dbname = dbname, user = user,
    password = password, host = host)
```

Możemy teraz zadawać dowolne zapytania SQL, pobierać i wysyłać całe tabele z danymi:

```
(gadki <- dbGetQuery(polaczenie, "SELECT * FROM statements ORDER BY
    nr_term_of_office, id_statement limit 1"))
##   id_statement     surname_name  date_statement
## 1    100.1.001  Marek Poznanski      2015-09-16
## statement
## 1 Informuje, ze w dniu dzisiejszym o godz. 16 odbedzie sie
    posiedzenie Komisji  Zdrowia. Dziekuje.
glosy <- dbGetQuery(polaczenie, "SELECT club, vote, count(*) FROM
    votes GROUP BY club, vote limit 7")
glosy
##       club          vote  count
## 1       ZP       Przeciw   8448
## 2       PO       Przeciw 827472
## 3     KPSP       Przeciw   5659
## 4    niez.     Nieobecny  12418
## 5       PO Wstrzymal sie   3291
## 6       ZP Wstrzymal sie   3472
## 7  Kukiz15            Za   4495
```

Na koniec pracy należy rozłączyć się z bazą danych i zwolnić połączenie.

```
dbDisconnect(polaczenie)
## [1] TRUE
```

3.2.3. Jak używać pakietu `dplyr` w pracy z bazami danych

Standardów i implementacji SQL-a jest tak wiele, że zastanawiające jest, dlaczego nazywane są standardami. Praktycznie każda baza danych różni się listą zaimplementowanych funkcjonalności czy agregatów. Jeżeli pracujemy z jedną bazą danych, to ta różnorodność nie będzie nam doskwierać. Ale jeżeli przyjdzie nam jednocześnie korzystać z baz MSSQL, MySQL, RedShift i Postgres? Lub gdy okaże się, że dane zostały zmigrowane do nowej bazy danych?

Wielu problemów można sobie oszczędzić, używając pośrednika do komunikacji z bazą danych. Takim pośrednikiem może być pakiet `dplyr`, omówiony w poprzednim rozdziale. Pozwala on na pracę z danymi bez zastanawiania się, gdzie te dane aktualnie są i jak nazywa się w tym systemie bazodanowym potrzebna nam funkcja.

W pakiecie `dplyr` połączenia ze źródłem danych (tabelą z liczbami lub bazą danych) tworzy się dzięki funkcjom `src_*`. Poniżej korzystamy z funkcji `src_sqlite()` i `src_postgres()`.

Zainicjujmy połączenie z bazą SQLite i pobierzmy kilka pierwszych wierszy z tabeli `auta2012`:

```
library("dplyr")
polaczenie <- src_sqlite(path = 'zabawka.db')
auta1 <- tbl(polaczenie, "auta2012")
```

Mając taki obiekt, reszta operacji wygląda tak jak praca z pakietem `dplyr` na lokalnych ramkach danych.

```
auta1 %>% head(2)
##    Cena Waluta Cena.w.PLN Brutto.netto  KM  kW      Marka
## 1 49900    PLN      49900       brutto 140 103        Kia
## 2 88000    PLN      88000       brutto 156 115 Mitsubishi
```

A teraz pokażmy przykład dla bazy PostgreSQL. Połączymy się z bazą, wyświetlimy dostępne tabele i stworzymy połączenie do tabeli `votes`.

```
polaczenie <- src_postgres(dbname = dbname, host = host,
    user = user, password = password)
src_tbls(polaczenie)
## [1] "votes"            "counter"          "statements"
## [4] "deputies"         "votings"          "votes_copy_27_08"
## [7] "db"               "test_statements"
glosy <- tbl(polaczenie, "votes")
```

Zdefiniowawszy połączenie, możemy pracować na tej tabeli, używając funkcji z pakietu `dplyr`.

```
glosy %>%
  group_by(club, vote) %>%
  summarise(liczba = n()) ->
  liczba_glosow
class(liczba_glosow)
## [1] "tbl_postgres" "tbl_sql"      "tbl"
```

Wynikiem tych operacji jest obiekt klasy `tbl_sql`. Nie przechowuje on danych, ale instrukcje pozwalające na dostęp do danych (zapytanie SQL). Można to zapytanie i plan zapytania wyłuskać:

```
liczba_glosow$query
## <Query> SELECT "club", "vote", "liczba"
## FROM (SELECT "club", "vote", count(*) AS "liczba"
## FROM "votes"
## GROUP BY "club", "vote") AS "zzz3"
## <PostgreSQLConnection:(42999,5)>
explain(liczba_glosow)
## <SQL>
## SELECT "club", "vote", "liczba"
## FROM (SELECT "club", "vote", count(*) AS "liczba"
## FROM "votes"
## GROUP BY "club", "vote") AS "zzz3"
## <PLAN>
## HashAggregate  (cost=74577.20..74577.72 rows=52 width=9)
##   ->  Seq Scan on votes  (cost=0.00..51780.54 rows=3039554 width
    =9)
```

Dane wskazywane przez to połączenie nie są materializowane, o ile nie muszą być materializowane (użytkownik wprost tego nie zażąda). Pozwala to jak najwięcej operacji przeprowadzić zdalnie, jeszcze w bazie danych, i do R pobrać jedynie końcowe agregaty. Zazwyczaj jest to dużo szybsze rozwiązanie niż pobieranie całych danych i praca z nimi lokalnie.

Gdy już jest jasne, co użytkownik chce zrobić, i gdy jawnie zażąda wyniku, wszystkie operacje są wykonywane w możliwie małej liczbie kroków (jednym).

Materializować wyniki można na dwa sposoby:

- `collect()` – wyznacza wynik oraz pobiera go do R,
- `compute()` – wyznacza wynik i zapisuje go w tymczasowej tabeli w bazie danych.

Poniższa instrukcja pobiera wyliczone agregaty ze zbioru danych:

```
collect(liczba_glosow)
## Source: local data frame [60 x 3]
## Groups: club [15]
```

```
##       club           vote liczba
##      (chr)          (chr)  (dbl)
## 1       ZP       Przeciw   8448
## 2       PO       Przeciw 827472
## 3     KPSP       Przeciw   5659
## 4    niez.     Nieobecny  12418
## 5       PO Wstrzymal sie   3291
## 6       ZP Wstrzymal sie   3472
## 7  Kukiz15            Za   4495
## 8    niez.            Za  27274
## 9     KPSP Wstrzymal sie   2699
## 10 Kukiz15     Nieobecny    893
## ..     ...           ...    ...
```

Więcej informacji o funkcjach z pakietu `dplyr`, które można stosować do baz danych, znaleźć można na stronie [35].

3.3. Budowa aplikacji WWW z pakietem `shiny`

W poprzednim rozdziale pokazaliśmy, jak tworzyć automatyzowalne raporty z użyciem pakietów `knitr` i `Sweave`. Jednym z częstych zastosowań raportów jest przygotowanie zbioru zestawień statystycznych dla innej osoby (klienta, przełożonego, recenzenta, kolegi z zespołu). Wadą raportu jest jego statyczność i liniowość. Raz wygenerowany raport ma określoną treść (statyczną), którą zazwyczaj przegląda się strona po stronie (liniowość).

Co jednak, gdybyśmy chcieli w interaktywny sposób eksplorować dane? Dać odbiorcy możliwość wyboru interesujących go parametrów i wyznaczenia opisu dla zadanych parametrów? Obecnie najlepszym sposobem zrobienia tego w programie R jest przygotowanie aplikacji internetowej z użyciem pakietu `shiny` (patrz też [25]).

Aplikacje Shiny to strony internetowe tworzone i zasilane przez działający na serwerze program R. Użytkownicy aplikacji mogą poprzez stronę HTML wybierać parametry przetwarzania. Przetwarzanie danych i parametrów ma miejsce na serwerze, a jego wynik jest przedstawiany ponownie na stronie internetowej.

Galeria interesujących przykładów aplikacji wykonanych z użyciem biblioteki `shiny` znajduje się na stronie `http://shiny.rstudio.com/gallery/`. Zastosowania są najróżniejsze, od hobbystycznych aplikacji do oglądania małych zbiorów danych po złożone serwisy internetowe dostarczające informacji rzeszom użytkowników.

Poniżej pokażemy, jak zbudować aplikację `shiny` krok po kroku.

3.3.1. Jak wygląda model akcja–reakcja

Aplikacje Shiny działają zgodnie z modelem *akcja–reakcja* (ang. *reactive programming*). Jest to model znany np. z arkuszy kalkulacyjnych. Zazwyczaj takie arkusze poza danymi zawierają również formuły określające, jak ma być wyznaczona wartość danej komórki w zależności od innych komórek. Jeżeli zmienimy wartość w jednej komórce (akcja) to automatycznie zmienią się wartości w komórkach zależnych (reakcja).

Podobnie wygląda reakcja w aplikacji zbudowanej z wykorzystaniem biblioteki `shiny`. Na stronie WWW umieszczamy kontrolki wskazujące pewne wartości oraz wyniki działania funkcji na tych wartościach. Gdy zmienimy kontrolkę, to odświeżone zostaną wszystkie wyniki, które od danej kontrolki zależą.

Takie „inteligentne odświeżanie" wartości, które mogą się zmienić, pozwala na budowę szybkich i jednocześnie złożonych aplikacji. Model *akcja–reakcja* wymaga oprogramowania sposobu, w jaki elementy wejściowe wpływają na wygląd/stan/wartość elementów wyjściowych. Jeżeli w trakcie pracy zmieniona zostanie wartość jakiegoś elementu wejściowego (kliknięty przycisk, wpisana wartość liczbowa, przesunięty suwak), to przeliczone zostaną odpowiednie elementy wyjściowe.

3.3.2. Jak opisać interfejs użytkownika

Aplikacja Shiny składa się z dwóch części, które zazwyczaj opisuje się w dwóch osobnych plikach. Jedna część zawiera opis interfejsu użytkownika, układu elementów graficznych (zapisuje się ją w pliku `ui.R`), druga opisuje sposób przetwarzania danych (zapisuje się ją w pliku `server.R`).

Pierwsza aplikacja, którą przedstawiamy, jest dostępna pod adresem `http://bit.ly/2e0KRcO`. Nie ma w niej jeszcze żadnej interakcji, ale pokazana jest budowa elementów interfejsu graficznego. Ekran tej aplikacji zaprezentowany jest na rysunku 3.1.

Plik `ui.R` opisuje strukturę wyglądu ekranu. Zagnieżdżona struktura funkcji odpowiada zagnieżdżeniu elementów HTML w aplikacji.

Rysunek 3.1. Ekran pierwszej z prezentowanych aplikacji Shiny

Funkcja `titlePanel()` odpowiada za określenie tytułu, a `sidebarLayout()` odpowiada za stworzenie podziału strony na lewy i prawy panel. Funkcje `p()` i `br()` tworzą następnie HTML-owe tagi oraz <br/>. W pakiecie `shiny` znajdują się funkcje odpowiadające wszystkim tagom HTML-a.

```
# Plik ui.R -- interfejs
library(shiny)
shinyUI(fluidPage(
  titlePanel("Moja pierwsza aplikacja"),
  sidebarLayout(
    sidebarPanel(
      p("Tutaj jest lewy panel")
    ),
    mainPanel(
      br(),
      p("Tutaj jest prawy panel")
    )
  )
))
```

Każda aplikacja potrzebuje też instrukcji odpowiadających za przetwarzanie po stronie serwera. Zazwyczaj umieszcza się je w pliku `server.R`. W tej aplikacji nie ma żadnej interakcji, więc i opis strony serwerowej jest pusty:

```
# Plik server.R -- logika biznesowa
shinyServer(function(input, output) {

})
```

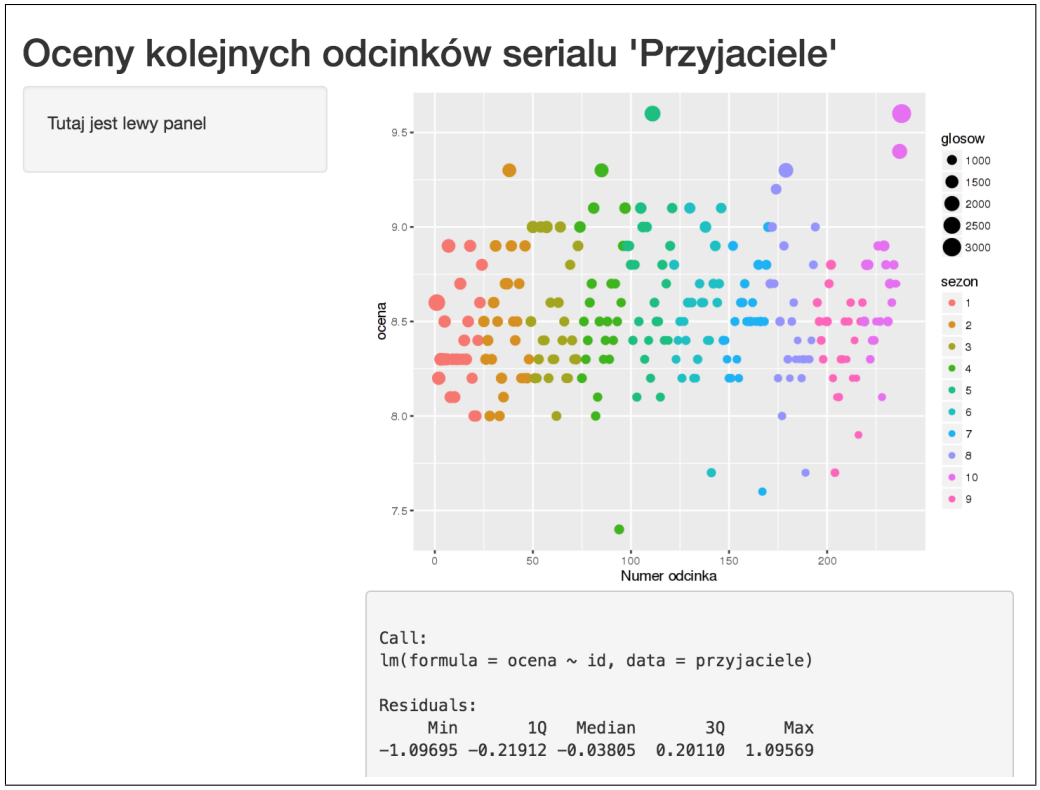
Aby odtworzyć tę aplikacje na własnym komputerze, należy w katalogu stworzyć dwa pliki: `ui.R` i `server.R`, a następnie przekopiować do nich powyżej przedstawione treści. Po otwarciu tych plików w programie RStudio w prawym górnym rogu powinna pojawić się ikonka `Run App`, której kliknięcie uruchomi aplikację. Inną możliwością jest wywołanie funkcji `shiny::runApp()` ze wskazaniem katalogu, w którym znajdują się oba pliki.

Uruchomienie aplikacji przez program RStudio spowoduje, że działa ona na lokalnym komputerze. Wgranie obu plików na serwer, na którym zainstalowana jest aplikacja Shiny Server, spowoduje, że będziemy mogli naszą aplikację udostępnić również innym użytkownikom.

3.3.3. Jak opisać przetwarzanie na serwerze

Zbudujmy teraz aplikację, która coś liczy i wyświetla efekt swej pracy na ekranie. W przypadku poniższej aplikacji wynikiem będą dwa elementy: wykres i opis modelu liniowego.

Pod adresem http://bit.ly/2enFK7F można obejrzeć opisaną poniżej aplikację. Dodanie wyników do aplikacji wymaga zmiany obu jej części. W pliku ui.R należy określić, gdzie ma zostać umieszczony wynik, a w pliku server.R należy podać sposób jego wyliczenia. Ekran tej aplikacji przedstawiony jest na rysunku 3.2.



Rysunek 3.2. Ekran aplikacji Shiny wyświetlającej wykres. Wykres jest za każdym razem generowany przez R umieszczony na serwerze. Cała aplikacja jest dostępna na stronie http://bit.ly/2enFK7F

Zacznijmy od pliku ui.R. W miejscu, gdzie tworzony jest prawy panel, znalazły się wywołania dwóch nowych funkcji:

- plotOutput(), wskazującej, gdzie ma zostać umieszczony wykres zapisany w obiekcie o nazwie trend,
- verbatimTextOutput(), wskazującej, gdzie ma zostać umieszczony opis zapisany w obiekcie o nazwie model.

Funkcje osadzające obiekty w interfejsie HTML mają nazwy określające, co jest osadzane (wykres, tabela, opis obiektu, strona HTML), i kończące się słowem Output.

```
# Plik ui.R -- interfejs uzytkownika
library(shiny)
shinyUI(fluidPage(
  titlePanel("Oceny kolejnych odcinkow serialu 'Przyjaciele'"),
  sidebarLayout(
    sidebarPanel(
      p("Tutaj jest lewy panel")
    ),
    mainPanel(
      plotOutput("trend"),
      verbatimTextOutput("model")
    )
  )
))
```

W pliku `server.R` muszą znaleźć się instrukcje tworzące obiekty `trend` i `model`. Treść tego pliku przedstawiona jest poniżej. Istotne deklaracje doklejane są do listy `output$`. W przypadku obiektu `trend` jest on produkowany przez wyrażenie przekazane do funkcji `renderPlot()`. To wyrażenie tworzy wykres z użyciem pakietu `ggplot2`.

Model liniowy tworzony wewnątrz funkcji `renderPrint()`.

```
# Plik server.R -- logika biznesowa
library(PogromcyDanych)
library(ggplot2)
shinyServer(function(input, output) {
  output$trend = renderPlot({
    przyjaciele <- serialeIMDB[serialeIMDB$serial == "Friends", ]
    ggplot(przyjaciele, aes(id, ocena, size=glosow, color=sezon)) +
      geom_point() + xlab("Numer odcinka")
  })

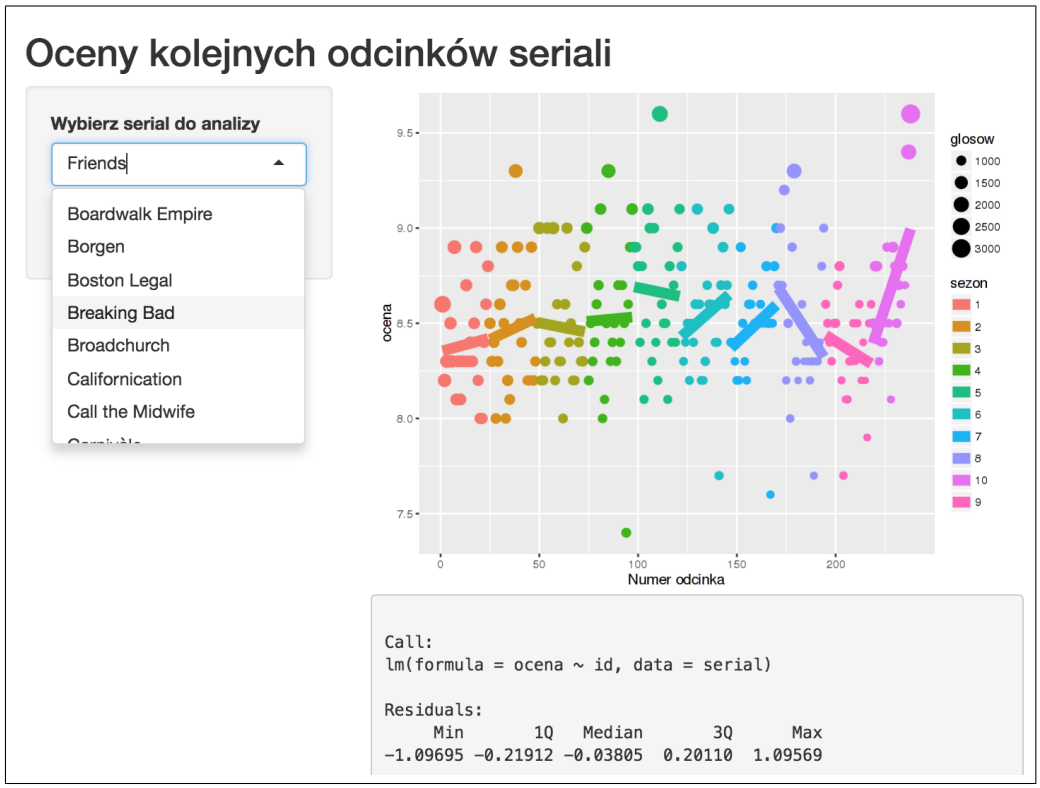
  output$model = renderPrint({
    przyjaciele <- serialeIMDB[serialeIMDB$serial == "Friends", ]
    summary(lm(ocena~id, przyjaciele))
  })
})
```

3.3.4. Jak dodawać kontrolki sterujące

Największą zaletą aplikacji Shiny jest ich interaktywność. Nie tylko możemy oglądać wyniki raz zadeklarowanych analiz, ale ustawiając określone parametry aplikacji, możemy oglądać wyniki dla różnych wariantów parametrów.

Przedstawiamy to na przykładzie dwóch dodatkowych kontrolek w naszej aplikacji. Jedna kontrolka opisuje, który serial ma być analizowany, a druga określa, czy ma być dorysowywana linia trendu dla poszczególnych sezonów.

Pod adresem http://bit.ly/2eR50Up można zobaczyć opisaną poniżej aplikację. Początkowy ekran tej aplikacji jest przedstawiony na rysunku 3.3.



Rysunek 3.3. Ekran aplikacji Shiny z kontrolkami umożliwiającymi wybór serialu oraz dodanie trendu. Każda zmiana stanu kontrolek powoduje narysowanie nowego wykresu. Cała aplikacja jest dostępna na stronie http://bit.ly/2eR50Up

Dodanie obsługi kontrolek również wymaga zmian w obu częściach opisujących aplikację. W pliku ui.R należy wskazać, gdzie mają znaleźć się kontrolki i jakie wartości mogą one przyjmować. W pliku server.R należy określić, jak zachowanie aplikacji ma zależeć od wskazań użytkownika.

Zacznijmy do pliku ui.R. W poniższej deklaracji do lewego panelu dodano wywołania dwóch funkcji.

- Funkcja selectInput() tworzy rozwijaną listę wyboru. W takiej liście należy określić, jakie wartości są możliwe do wskazania (argument choices), jaka wartość powinna być domyślnie wybrana (argument selected) oraz do jakiego obiektu ma być zapisany wybór użytkownika (argument inputId).
- Funkcja checkboxInput() tworzy przycisk wyboru. Należy w niej określić, jaka ma być początkowa wartość przycisku (argument value) oraz do jakiego obiektu ma być zapisany wybór użytkownika (argument inputId).

```
# Plik ui.R
library("shiny")
library("PogromcyDanych")
nazwySeriali <- sort(levels(serialeIMDB$serial))
shinyUI(fluidPage(
  titlePanel("Oceny kolejnych odcinkow seriali'"),
  sidebarLayout(
    sidebarPanel(
      selectInput(inputId = "wybranySerial",
                  label = "Wybierz serial do analizy",
                  choices = nazwySeriali,
                  selected = "Friends"),
      checkboxInput(inputId = "liniaTrendu",
                    label = "Czy zaznaczyc linie trendu?",
                    value = TRUE)
    ),
    mainPanel(
      plotOutput("trend"),
      verbatimTextOutput("model")
    )
  )
))
```

W pliku `server.R` wewnątrz funkcji `*Output` można odwoływać się do wartości kontrolki poprzez listę `input`.

Zmienna `input$wybranySerial` odpowiada nazwie wybranego serialu. Zmienna `input$liniaTrendu` będzie wartością logiczną `TRUE/FALSE`, określającą, czy przycisk jest zaznaczony, czy odznaczony.

Wewnątrz funkcji `*Output` można dowolnie odwoływać się do wartości kontrolek. Za każdym razem, gdy użytkownik zmieni wartość kontrolki, zostaną wykonane wszystkie wyrażenia zawierające w swojej treści odwołania do zmienionej kontrolki. Wyniki tych wyrażeń zostaną przeliczone na nowo i uaktualnione w oknie aplikacji.

```
library("PogromcyDanych")
library("ggplot2")
library("dplyr")
shinyServer(function(input, output) {
  output$trend = renderPlot({
        wybranySerial <- input$wybranySerial
    serial <- filter(serialeIMDB, serial == wybranySerial)
    pl <- ggplot(serial, aes(id,ocena,size=glosow, color=sezon)) +
      geom_point() + xlab("Numer odcinka")
    if (input$liniaTrendu)
      pl <- pl + geom_smooth(se=FALSE, method="lm", size=3)
    pl
  })
```

```
  output$model = renderPrint({
        wybranySerial <- input$wybranySerial
    serial <- filter(serialeIMDB, serial == wybranySerial)
    summary(lm(ocena~id, serial))
  })
})
```

Przedstawioną powyżej aplikację można dowolnie rozbudowywać, dodając więcej kontrolek i elementów wynikowych.

W pakiecie `shiny` dostępnych jest wiele kontrolek, obsługujących daty, napisy, przedziały liczb, załączniki i inne możliwe argumenty wejściowe. Dostępnych jest również wiele możliwych formatów wynikowych, takich jak panele, tabele, interaktywne wykresy itp.

Opracowane aplikacje możemy umieszczać na własnych serwerach. Gdy nie mamy własnego serwera, możemy skorzystać z bezpłatnego hostingu aplikacji na stronie `http://shiny.rstudio.com/`.

Bardzo dobry tutorial, pozwalający na poznanie dodatkowych możliwości pakietu `shiny`, jest dostępny na stronie `http://shiny.rstudio.com/tutorial/`. Na uwagę zasługuje też dwustronicowa ściągawka podsumowująca kluczowe opcje pakietu `shiny` dostępna na stronie `http://bit.ly/2f1Ztuy`.

3.4. Budowanie własnych pakietów

W programie R mamy dostęp do olbrzymiej kolekcji funkcji i zbiorów danych. Aby móc zapanować nad tym oceanem możliwości, funkcje i zbiory danych są pogrupowane w pakiety.

Pracując z R, na początku korzystamy z pakietów opracowanych przez inne osoby. Ale szybko okazuje się, że mamy nasze własne funkcje i zbiory danych, które chcielibyśmy zachować dla innych użytkowników R. A może tylko dla grupy współpracowników z firmy, z którymi pracujemy nad jakimś projektem. A może tylko dla naszego „przyszłego ja". W każdym z tych przypadków okazuje się, że wygodnie jest pozamykać część funkcjonalności w grupy pakietów.

Zasadniczo pakiet pozwala na złożenie sześciu elementów.

- Funkcji czy też funkcjonalności, które chcemy udostępnić.
- Zbiorów danych, które mają być dostępne po wczytaniu pakietu.
- Opisów i dokumentacji tych funkcjonalności i zbiorów danych pokazujących, kiedy i jak z nich korzystać.
- Opisu zależności od innych pakietów, które powinny być wcześniej wczytane i zainstalowane.
- Testów, pozwalających na weryfikację czy funkcjonalność pakietu wciąż działa poprawnie.
- Większych ilustracji działania całego zbioru funkcji.

Stworzony pakiet możemy umieścić na naszej stronie WWW, wysłać zainteresowanym osobom mailem lub, jeżeli chcemy, by był dostępny szerokiemu gronu, możemy go umieścić w publicznym repozytorium pakietów R, np. repozytorium CRAN lub Bioconductor. W ostatnich latach najpopularniejsze miejsce trzymania pakietu to repozytorium GitHub.com. Bardzo wiele pakietów, zanim ich oficjalna wersja trafi na serwery CRAN, jest roboczo rozwijanych na serwerach GitHub.

Czy tworzenie pakietów jest dla Ciebie? Tak! Nawet jeżeli w chwili czytania tej książki opracowaliśmy zbiór funkcji tylko do naszego użytku, to dobrym pomysłem jest dodać do nich opisy i złożyć je w pakiet. Korzystanie z pakietów pozwala na łatwe przenoszenie kodu, automatyczne testowanie opracowanych funkcji, łatwe zarządzanie dokumentacją kodu, oferuje przenośność funkcjonalności pomiędzy różnymi systemami operacyjnymi, komputerami lub wersjami programu R.

Podsumowując: warto tworzyć pakiety! Tym bardziej że obecnie jest to bardzo proste dzięki pakietowi `devtools`. Nie wymaga to tak dużego nakładu pracy jak chociażby trzy lata temu.

Poniżej krok po kroku pokazujemy, jak zbudować szkielet pakietu, a także jak wypełnić go funkcjami, dokumentacją, zależnościami, testami czy zbiorami danych. Szczegółowo proces budowy pakietów jest opisany w dokumencie [22], a także w osobnej książce poświęconej wyłącznie budowie pakietów [34].

Bardzo przydatna może być dwustronicowa ściągawka z procesu budowania pakietów przygotowana przez firmę RStudio w serii *Cheatsheets*. Jest ona dostępna na stronie `http://bit.ly/2eT1DhL`.

3.4.1. Niezbędne oprogramowanie

Aby budować pakiety z użyciem narzędzi omówionych w dalszej części tego rozdziału, potrzebujemy pakietu `devtools`. Można go zainstalować instrukcją:

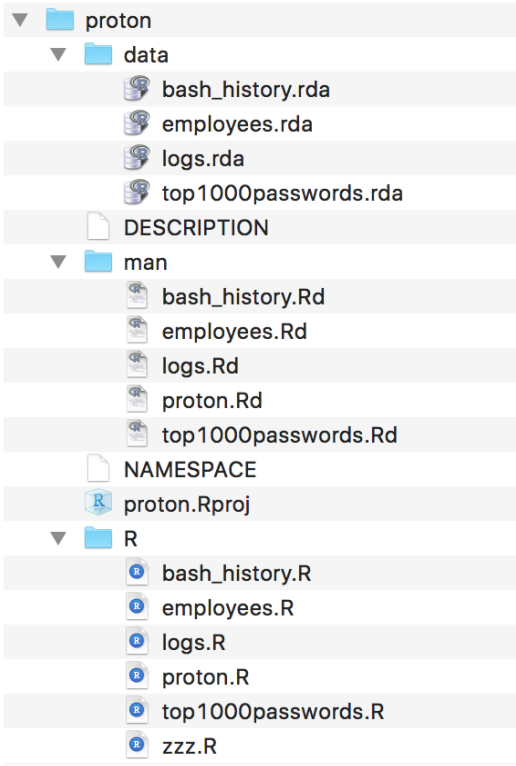
```
install.packages("devtools")
```

Jeżeli pracujemy w systemie Linux/Unix/OS X, to wszystkie inne elementy niezbędne do budowy pakietów są już zainstalowane.

Jeżeli pracujemy w systemie Windows, to musimy dodatkowo doinstalować tzw. R tools, czyli zbiór kompilatorów niezbędnych do budowy pakietu. Można je pobrać ze strony `https://cran.r-project.org/bin/windows/Rtools/` lub wpisując w dowolną wyszukiwarkę frazę *R tools*. Dla wersji R 3.x zestaw ten zawiera MinGW, Perla i kilka innych dodatków.

3.4.2. Jak wygląda struktura pakietu

Na rysunku 3.4 przedstawiona jest struktura przykładowego pakietu. Pakiet to katalog, wewnątrz którego znajdują się określone pliki i podkatalogi:



Rysunek 3.4. Struktura przykładowego pakietu `proton`

- plik `DESCRIPTION`, z opisem podstawowych informacji o pakiecie (nazwa pakietu, krótki opis, autor, zależności),
- plik `NAMESPACE`, z opisem funkcji udostępnionych przez pakiet,
- katalog `R`, z listą funkcji w programie R,
- katalog `data`, z listą zbiorów danych udostępnionych przez pakiet,
- katalog `man`, z plikami dokumentacji Rd do udostępnionych funkcji i zbiorów danych,
- katalog `vignettes`, z listą przewodników opisujących funkcjonalności pakietu,
- katalog `tests`, z listą testów weryfikujących poprawność funkcji.

W kolejnych sekcjach omawiamy zawartość poszczególnych plików i katalogów.

3.4.3. Jak stworzyć nowy pakiet

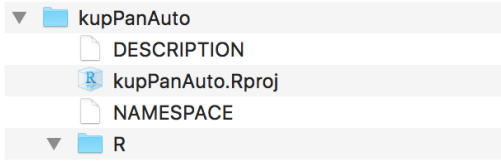
Nowy pakiet najwygodniej zbudować, używając funkcji `create()` z biblioteki `devtools`. Pierwszym argumentem tej funkcji jest ścieżka do katalogu, w którym ma być założony pakiet. Ostatni człon tej nazwy jest traktowany jako nazwa pakietu.

Na potrzeby tego rozdziału stworzymy nowy pakiet `kupPanAuto` i zbudujemy go w obecnym roboczym katalogu.

```
library(devtools)
create("kupPanAuto")
## Creating package kupPanAuto in .
## No DESCRIPTION found. Creating with values:
## Package: kupPanAuto
## Title: What the Package Does (one line, title case)
## Version: 0.0.0.9000
## Authors@R: person("First", "Last", email = "first.last@example.com
    ", role = c("aut", "cre"))
```

```
## Description: What the package does (one paragraph).
## Depends: R (>= 3.3.2)
## License: What license is it under?
## LazyData: true
## Adding RStudio project file to kupPanAuto
```

Powyższa instrukcja w aktualnym katalogu tworzy podkatalog o nazwie `kupPanAuto` z minimalną liczbą niezbędnych plików. Początkowa zawartość katalogu jest przedstawiona na rysunku 3.5.



Rysunek 3.5. Struktura plików i katalogów w nowo utworzonym pakiecie `kupPanAuto`

Funkcja `create()` tworzy automatycznie plik konfiguracyjny projektu RStudio. Jest on zapisany w pliku `*.proj`. Otwarcie takiego pliku w programie RStudio spowoduje otwarcie projektu z katalogiem roboczym automatycznie ustawionym na katalog pakietu.

3.4.4. Plik `DESCRIPTION`

Każdy pakiet musi mieć plik `DESCRIPTION`, który zawiera podstawowe informacje o pakiecie. Do informacji tych należą: nazwa, opis, informacja o autorach i lista zależności od innych pakietów.

Nowo stworzony przez funkcję `create()` pakiet ma plik `DESCRIPTION`. Pierwszą rzeczą, którą należy zrobić po zbudowaniu pakietu, jest uzupełnienie informacji w tym pliku.

W sekcji `Package` wpisujemy nazwę pakietu, w sekcji `Title` jednozdaniowy opis pakietu, w sekcji `Authors` informację o autorach oraz ich roli przy tworzeniu pakietów. W sekcji `Description` uzupełniamy szerszy opis pakietu.

Szczególnie istotne są sekcje `Imports` i `Suggests`, które opisują zależności od innych pakietów.

- Sekcja `Imports` wskazuje pakiety niezbędne do działania Twojego pakietu. Program R będzie je instalował przy instalacji Twojego pakietu.
- Sekcja `Suggests` wskazuje pakiety dodatkowe, które mogą rozszerzać możliwości Twojego pakietu, ale nie są niezbędne do działania kluczowych funkcjonalności.

Na rysunku 3.6 przedstawiamy przykładowy uzupełniony plik `DESCRIPTION`.

Jeżeli w naszym pakiecie chcemy korzystać z funkcji lub danych z innych pakietów, powinniśmy dodać odpowiednią zależność do pliku `DESCRIPTION`.

```
Package: proton
Title: The Proton Game
Version: 1.0
Authors@R: c(
    person("Przemysław", "Biecek", email = "przemyslaw.biecek@gmail.com", role = c("aut", "cre")),
    person("Witold", "Chodor", email = "witoldchodor@gmail.com", role = c("trl")),
    person("Foundation", "SmarterPoland.pl", email = "smarterpoland@gmail.com", role = c("cph")))
Description: 'The Proton Game' is a console-based data-crunching game for younger and older data sci
  Act as a data-hacker and find Slawomir Pietraszko's credentials to the Proton server.
  You have to solve four data-based puzzles to find the login and password.
  There are many ways to solve these puzzles. You may use loops, data filtering, ordering, aggregati
  Only basics knowledge of R is required to play the game, yet the more functions you know, the more
  The knowledge of dplyr is not required but may be very helpful.
  This game is linked with the ,,Pietraszko's Cave'' story available at http://biecek.pl/BetaBit/War
  It's a part of Beta and Bit series.
  You will find more about the Beta and Bit series at http://biecek.pl/BetaBit.
Depends:
    R (>= 3.0.0)
Imports:
    digest
License: GPL-2
LazyData: true
Encoding: UTF-8
RoxygenNote: 5.0.1
```

Rysunek 3.6. Uzupełniony plik `DESCRIPTION` przykładowego pakietu `proton`

Możemy w tym celu wykorzystać funkcję `use_package()`, która dodaje informacje o zależnościach od wskazanych pakietów.

```
use_package(package = "PogromcyDanych", pkg = "kupPanAuto")

## Adding PogromcyDanych to Imports
## Refer to functions with PogromcyDanych::fun()

use_package(package = "dplyr", pkg = "kupPanAuto")

## Adding dplyr to Imports
## Refer to functions with dplyr::fun()
```

3.4.5. Jak dodawać funkcje do pakietu

Najczęstszym zastosowaniem pakietów jest udostępnianie zbioru użytecznych funkcji.

Aby dodać funkcję do pakietu, wystarczy jej treść dodać w pliku o rozszerzeniu `R` do podkatalogu o nazwie `R`. Podczas ładowania pakietu wczytywane są wszystkie instrukcje z plików `*.R` umieszczonych w katalogu `R`.

Nazwa pliku `*.R` nie musi być taka sama jak nazwa funkcji. Jednak dobrze jest stosować te same lub podobne nazwy, aby łatwiej było znaleźć miejsce, gdzie znajduje się opis funkcji.

Przykładowo, poniżej przedstawiamy definicję funkcji `jakiPrzebieg()`, liczącej średni przebieg aut z danego rocznika. Definicję tej funkcji należy wkleić do dowolnego pliku z rozszerzeniem `R` w podkatalogu o nazwie `R`.

```
jakiPrzebieg <- function(rok = '', auta) {
  wybrane <- filter(auta, Rok.produkcji == rok)
  mean(wybrane$Przebieg.w.km, na.rm=TRUE)
}
```

Aby można było wygodnie korzystać z funkcji, dobrym zwyczajem jest przygotowanie dokumentacji. Umożliwia ona zarówno innym osobom, jak i „przyszłemu ja" zdobycie informacji, jakie parametry funkcja przyjmuje i co z nimi robi.

Dokumentacja funkcji i zbiorów danych przechowywana jest w plikach `Rd` w katalogu `man`. Można ją tworzyć w dowolnym tekstowym edytorze. Kiedyś trzeba to było robić „ręcznie", teraz jednak zdecydowanie wygodniej korzystać z pakietu `roxygen2`. Umożliwia automatyczne tworzenie dokumentacji na bazie krótkich opisów umieszczanych przed kodem funkcji w plikach `R`.

Opis funkcji umieszcza się w tym samym pliku co kod funkcji, w wierszach rozpoczynających się od znaków `#'`. Pierwsza linia dokumentacji określa jednozdaniowy tytuł funkcji, kolejny akapit to krótki opis funkcji. Następne akapity są traktowane jako rozszerzony opis.

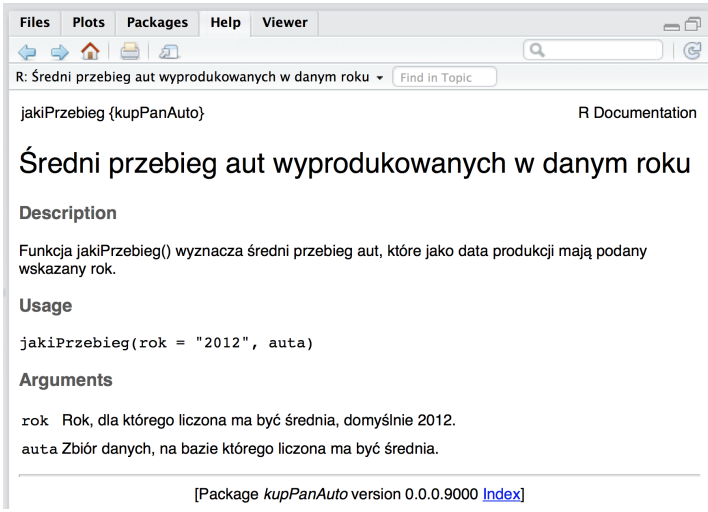
Aby oznaczyć funkcję kolejnych elementów opisu, stosuje się specjalne tagi. Ich nazwy rozpoczynają się od znaku `@`. Najczęściej stosowane tagi to:

- `@param` – opis określonego parametru opisywanej funkcji,
- `@return` – opis wyniku funkcji,
- `@export` – tag określający, że dana funkcja ma być publiczna,
- `@examples` – blok z przykładami,
- `@rdname` – określa nazwę pliku `Rd`, w którym przechowywana będzie dokumentacja.

Minimalna dokumentacja powinna zawierać tytuł oraz opisy argumentów funkcji. Dla przykładowej funkcji `jakiPrzebieg()` zawartość pliku R z kodem funkcji oraz jej dokumentacją mogłaby wyglądać tak:

```
#' Sredni przebieg aut wyprodukowanych w danym roku
#'
#' Funkcja jakiPrzebieg() wyznacza sredni przebieg aut,
#' ktore jako data produkcji maja podany wskazany rok.
#'
#' @param rok Rok, dla ktorego liczona ma byc srednia.
#' @param auta Zbior danych do liczenia sredniej.
#'
#' @export

jakiPrzebieg <- function(rok = '2012', auta) {
  wybrane <- filter(auta, Rok.produkcji == rok)
  mean(wybrane$Przebieg.w.km, na.rm=TRUE)
}
```



Rysunek 3.7. Przykładowa dokumentacja funkcji `jakiPrzebieg`

Gdy już zbudujemy i włączymy pakiet, opis funkcji `jakiPrzebieg()` będzie dostępny po uruchomieniu instrukcji `?jakiPrzebieg`. Powyżej przedstawiony kod zostanie przekształcony na dokumentację przedstawioną na rysunku 3.7.

Na rysunku 3.8 przedstawiliśmy przykładowy plik z opisem dokumentacji bardziej rozbudowanej funkcji.

3.4.6. Jak dodawać zbiory danych do pakietu

W pakietach możemy również przechowywać dane. Jest to wygodne, jeżeli chcemy jeden lub kilka zbiorów danych udostępnić współpracownikom, studentom lub całej reszcie świata. Wiele zbiorów danych stało się bardzo popularnymi właśnie dzięki temu, że łatwo było do nich sięgnąć i coś z nimi zrobić. Dane do pakietu dodać można na dwa sposoby:

- zapisując ten zbiór danych z użyciem funkcji `save()` do pliku z rozszerzeniem `rda` lub `RData`. Następnie ten plik należy wkopiować do katalogu `data` w pakiecie,
- używając funkcji `use_data()` z pakietu `devtools`. Poniższe linijki wyznaczają wektor ze średnimi cenami różnych marek aut, a następnie zapisują ten wektor do pakietu `kupPanAuto`.

```
library("PogromcyDanych")
srednieCeny <- tapply(auta2012$Cena, auta2012$Marka, mean, na.rm=T)
use_data(srednieCeny, pkg = "kupPanAuto")
```

Wszystkie obiekty zapisane w katalogu `data` są wczytywane do R wraz z załadowaniem pakietu funkcją `library()`. Są one widoczne dla wszystkich funkcji pakietu.

Obiekty zapisane w pliku `sysdata.rda` (w jednym pliku można zapisać więcej niż jeden obiekt) są widoczne wyłącznie dla funkcji wewnątrz pakietu. Są na swój sposób prywatne. Pozostałe obiekty będą widoczne globalnie.

Umieszczenie zbioru danych w pakiecie to jedno, opisanie użytkownikowi, jak z niego korzystać to drugie.

Podobnie jak w przypadku funkcji, aby dodać dokumentację zbioru danych, należy przygotować plik o rozszerzeniu `Rd` w katalogu `man`. Ale zamiast robić to ręcznie, lepiej skorzystać z pakietu `roxygen2`, który na bazie metaopisu wygeneruje nam poprawny plik `Rd`.

Pakiet `roxygen2` wymaga, by odpowiedni opis dokumentowanego pakietu umieścić w pliku o rozszerzeniu R w katalogu `R`, a więc tam, gdzie znajdują się opisy funkcji.

```
#' @title The Proton Game
#'
#' @description
#' The \code{proton} function is used for solving problems in the data-based game ,,The Proton Game''.
#' Solve four data-based puzzles in order to crack into Pietraszko's account!
#'
#' @param ... \code{proton} function is called by different arguments, which vary depending
#' on a problem that Bit is trying to solve. See \code{Details} in order to learn more about the list of possible
#'
#' @details Every time when some additional hints are needed one should add
#' \code{hint=TRUE} argument to the \code{proton} function.
#'
#' In order to get more information about a user on the Proton server
#' one should pass \code{action = "login"}, \code{login="XYZ"} arguments
#' to the \code{proton} function.
#'
#' @author
#' Przemyslaw Biecek, \email{przemyslaw.biecek@@gmail.com}, SmarterPoland.pl Foundation.
#'
#' @examples
#' \dontrun{
#' proton()
#' proton(hint=TRUE)
#' }
#' @rdname proton
#' @importFrom digest digest
#' @export
```

Rysunek 3.8. Przykładowy opis funkcji `proton`

```
#' The history of logs into the Proton server
#'
#' The dataset describing the history of logs: who, from where and when logged into the Proton server.
#' The subsequent columns in this dataset describe:
#' \itemize{
#'   \item login. The login of the user which logs into the Proton server.
#'   \item host. The IP address of the computer, from which the log into the Proton server was detected.
#'   \item date. The date of log into the Proton server. Rows are sorted by this column.
#' }
#'
#' @docType data
#' @keywords datasets
#' @name logs
#' @usage data(logs)
#' @format a data frame with 59366 rows and 3 columns.
NULL

```

Rysunek 3.9. Przykładowa dokumentacja zbioru danych

Pierwsza linia opisu traktowana jest jako tytuł pliku pomocy. Aby zaznaczyć, że jest to opis zbioru z danymi, należy dodać parametr `@docType data`.

Parametr `@name` wskazuje, pod jakim hasłem należy szukać danego pliku pomocy, zazwyczaj jest to nazwa zbioru danych. Inne parametry pozwalają na określenie struktury danych i opisanie zawartości poszczególnych składowych.

Przykładową dokumentację zbioru danych przedstawia rysunek 3.9.

3.4.7. Jak dodawać testy do pakietu

Bardzo ważną i jednocześnie zaniedbywaną składową tworzenia pakietów jest przygotowywanie testów weryfikujących, czy funkcjonalności (wciąż) działają.

Dla programu R przygotowano kilka różnych rozwiązań pozwalających na proste budowanie testów. W przypadku budowania pakietów lekkim i wygodnym rozwiązaniem umożliwiającym tworzenie testów jest pakiet `testthat`, który omawiamy poniżej.

Testy dodajemy w plikach R do katalogu `tests/testthat` w pakiecie.

Aby dodać ten katalog do naszego pakietu, wraz z potrzebnymi zależnościami, wystarczy wykonać instrukcję `use_testthat()` z pakietu `devtools`. Przygotuje ona całą niezbędną infrastrukturę dla testów w pakiecie.

```
use_testthat("kupPanAuto")
```

Nazwa „testy jednostkowe" bierze się stąd, że testy weryfikują poprawność poszczególnych jednostek/modułów. Zazwyczaj wykorzystuje się je, by przetestować poszczególne funkcje.

Każdy test to wywołania funkcji `test_that()`, która przyjmuje argumenty:

- Pierwszym jest nazwa testu. Gdyby podczas wykonania pojawił się błąd, ta nazwa ułatwi określenie, co nie działa.
- Drugi to lista oczekiwań, czyli funkcji rozpoczynających się od `expect_...`. Funkcje te sprawdzają, czy wywołanie funkcji kończy się oczekiwanym rezultatem, wartością, błędem, ostrzeżeniem, napisem itp.

Trzy przykładowe testy przestawione są poniżej:

```
test_that("Result is a number",{
  expect_true(is.numeric(jakaCena("Kia", 2010)))
})
test_that("An invalid number of parameters",{
  expect_error(jakaCena())
  expect_error(jakaCena("Kia"))
})
test_that("Results are correct",{
  expect_equal(round(jakaCena("Kia", 2010)), 44544)
  expect_equal(round(jakaCena("Kia", 2011)), 65989)
})
```

Aby uruchomić wszystkie testy z katalogu `tests`, wystarczy wywołać funkcję `test()` z pakietu `devtools`.

Jeżeli test zostanie wykonany bez błędów, wyświetli się jako kropka. Jeżeli w teście wystąpi błąd, wyświetli się informacja o tym, co to za test.

I poniżej przykład takiego działania:

```
library(devtools)
test("kupPanAuto")
Loading kupPanAuto
Testing kupPanAuto
1...
1. Error: Result is a number
    ----------------------------------------------
could not find function "jakaCena"

I believe in you!
```

Swoiste testy wykonywane są również przy przyjęciu pakietu do publicznych repozytoriów. Np. aby pakiet został przyjęty do CRAN, nie wystarczy, że funkcje działają poprawnie. Pakiet musi spełniać listę określonych warunków dotyczących formatowania (tzw. *Camel Case* dla tytułów, niezbyt długie linie, poprawna dokumentacja, wykonywalne przykłady) czy spójności opisu.

Do weryfikacji, czy pakiet jest zgodny z wszystkimi wymogami, można wykorzystać funkcję `check()`.

```
check("kupPanAuto")
## Updating kupPanAuto documentation
## Loading kupPanAuto
## ...
## * checking dependencies in R code ... OK
## * checking S3 generic/method consistency ... OK
## * checking replacement functions ... OK
## * checking foreign function calls ... OK
## * checking R code for possible problems ... NOTE
## * checking Rd files ... WARNING
## * checking Rd metadata ... OK
## * checking Rd line widths ... OK
## * checking Rd cross-references ... OK
## * checking Rd \usage sections ... OK
## * checking Rd contents ... OK
## * checking examples ... NONE
## * DONE
```

Wszystko, co nie jest oznaczone jako OK, należy poprawić.

3.4.8. Jak budować stworzony pakiet

Cykl życia pakietu składa się z kilku kroków:
1. Funkcje i zbiory danych do pakietu są dostępne jako katalog na dysku twórcy.
2. Funkcje, zbiory danych i ich dokumentacja w formacie `Rd` są dostępne jako katalog na dysku twórcy.
3. Pakiet jest dostępny jako plik źródłowy tar.gz lub binarny zip/tgz.
4. Pakiet jest zainstalowany na komputerze, a więc znajduje się w katalogu z bibliotekami.
5. Pakiet włączony w aktualnej sesji R.

Opisy z poprzednich rozdziałów pozwalają na zbudowanie pakietu w stanie z punktu 1. Poniżej pokazujemy, jak przejść do kolejnych kroków.

3.4.8.1. Budowanie dokumentacji (-> 2.)

W rozdziale 3.4.5 pisaliśmy o tym, jak dodawać dokumentację do funkcji i zbiorów danych w pakiecie. Aby zamienić tak przygotowany opis w dokumentację w formacie `Rd`, należy użyć funkcji `document()` z pakietu `devtools`. Tworzy ona pliki `Rd` w katalogu `man` na podstawie opisów roxygen oraz uzupełnia plik `NAMESPACE`.

```
document("kupPanAuto")
## Updating kupPanAuto documentation
## Loading kupPanAuto
## First time using roxygen2 5.0.1. Upgrading automatically...
```

3.4.8.2. Budowanie pakietu do postaci źródłowej (-> 3.)

Aby zbudować katalog z wszystkimi częściami pakietu do jednego pliku – postaci źródłowej pakietu, należy wykorzystać funkcję `build()` z pakietu `devtools`.

Parametrem funkcji `build()` jest katalog, w którym znajduje się pakiet do zbudowania. Domyślnie jest to aktualny katalog roboczy.

Poniższa instrukcja stworzy plik źródłowy `kupPanAuto_0.0.0.9000.tar.gz` (do nazwy pakietu dodawany jest automatycznie numer wersji) z zawartością pakietu:

```
build("kupPanAuto")
## '/Library/Frameworks/R.framework/Resources/bin/R'  \
##   --no-site-file --no-environ --no-save  \
##   --no-restore CMD build  \
##   '/Users/pbiecek/GitHub/Przewodnik/inne/kupPanAuto'  \
##   --no-resave-data --no-manual
##
```

```
## * checking for file ‘/Users/pbiecek/GitHub/Przewodnik/inne/
    kupPanAuto/’DESCRIPTION ... OK
## * preparing ‘’kupPanAuto:
## * checking DESCRIPTION meta-information ... OK
## * checking for LF line-endings in source and make files
## * checking for empty or unneeded directories
## * building ‘kupPanAuto_0.0.0.9000.tar.’gz
##
## [1] "/Users/pbiecek/GitHub/Przewodnik/inne/kupPanAuto_0.0.0.9000.
    tar.gz"
```

3.4.8.3. Instalacja pakietu (-> 4.)

Pakiety instalowane są w katalogu z bibliotekami R. Ścieżkę do tych katalogów można odczytać przy użyciu funkcji `.libPaths()`:

```
.libPaths()
## [1] "/Library/Frameworks/R.framework/Versions/3.2/Resources/
    library"
```

Aby zainstalować pakiet z pliku źródłowego lub z katalogu, można wykorzystać polecenie `install()`.

Parametrem funkcji `install()` jest katalog, w którym znajduje się pakiet do zbudowania. Domyślnie jest to katalog roboczy.

```
install("kupPanAuto")
## Installing kupPanAuto
## ’/Library/Frameworks/R.framework/Resources/bin/R’ --no-site-file
    --no-environ --no-save --no-restore CMD  \
##   INSTALL ’/Users/pbiecek/GitHub/Przewodnik/inne/kupPanAuto’  \
##   --library=’/Library/Frameworks/R.framework/Versions/3.2/
    Resources/library’ --install-tests
##
## * installing *source* package ‘’kupPanAuto ...
## ** R
## ** preparing package for lazy loading
## ** help
## *** installing help indices
## ** building package indices
## ** testing if installed package can be loaded
## * DONE (kupPanAuto)
```

Poleceniem `install.packages()` można instalować pakiety z repozytorium CRAN, a poleceniem `install_github()` – z repozytoriów GitHub.

Poniższa instrukcja zainstaluje pakiet `proton` ze strony `https://github.com/BetaAndBit/PieczaraPietraszki`:

```
install_github("BetaAndBit/PieczaraPietraszki/protonENG")
```

3.4.8.4. Włączanie pakietu (-> 5.)

Aby skorzystać z danych lub funkcji pakietu, należy go włączyć poleceniem `require()` lub `library()`. Ewentualnie można wykorzystać operator dostępu `::` (lub `:::` w przypadku funkcji, które nie są publiczne).

```
library("kupPanAuto")
```

3.5. Git, GitHub i kontrola wersji

W tym podrozdziale omawiamy narzędzia pozwalające na wersjonowanie kodu, w szczególności oparte na systemie Git. Jest wiele powodów, dla których warto je poznać i stosować, wymieńmy cztery.

- System kontroli kodu pozwala na śledzenie wszystkich zmian w kodzie. Jeżeli okaże się, że pewne zmiany trzeba jednak wycofać, to zawsze jest możliwość powrotu stanu kodu do określonego punktu w przeszłości.
- W przypadku pracy w zespole można zawsze sprawdzić, kto wprowadził jakie zmiany. Gdyby trzeba było później ten kod poprawić, wiadomo, kogo o to zapytać.
- W przypadku awarii jednego komputera lub pracy na kilku komputerach dostępność repozytorium pozwala odtworzyć kod z kopii.
- Bardzo często nowe pakiety są umieszczane i instalowane z repozytoriów Git. W porównaniu z repozytorium CRAN daje to większą swobodę w aktualizacji kolejnych wersji pakietów.

Jednym z najpopularniejszych systemów kontroli wersji jest Git. Dzięki serwisowi GitHub można z niego korzystać bezpłatnie, o ile godzimy się na niewielkie ograniczenia.

Na stronie `http://github.com` można utworzyć nowego użytkownika, jak również stworzyć nowe repozytorium z kodem. Po utworzeniu nowego repozytorium będzie ono pełniło funkcję centralnego zdalnego repozytorium.

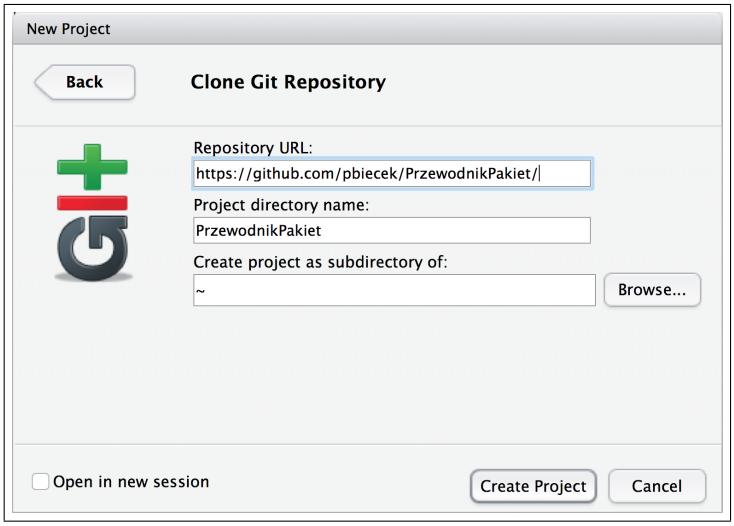
3.5.1. Jak kopiować repozytorium – `clone`

Kod zazwyczaj jest rozwijany na lokalnym komputerze. Pierwszym krokiem, by rozpocząć pracę nad kodem, jest utworzenie na lokalnym komputerze kopii zdalnego repozytorium.

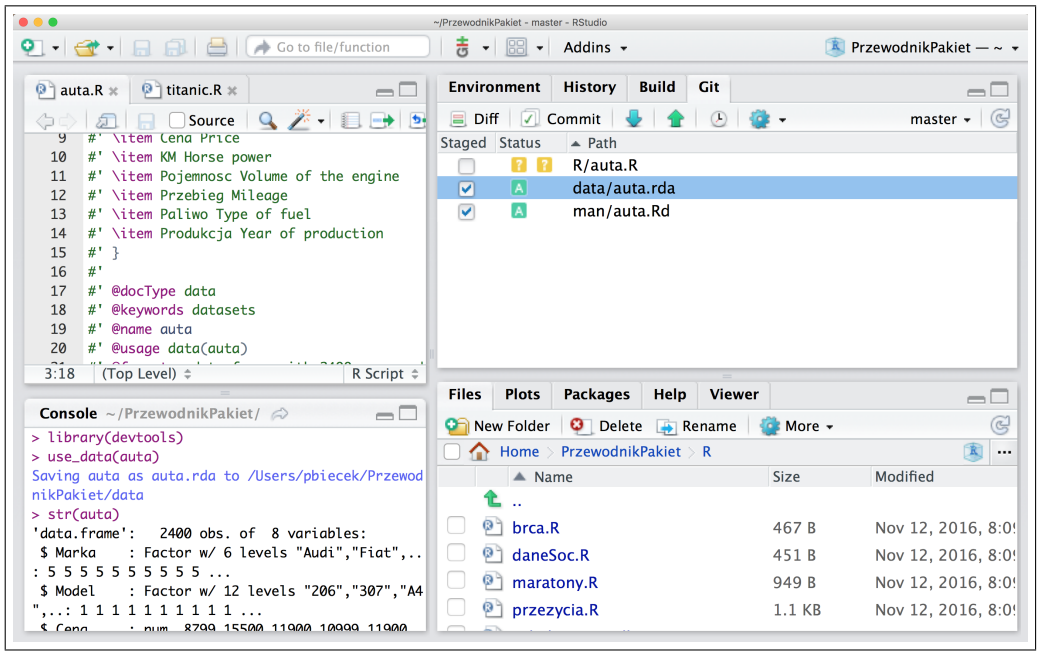
Obsługa systemu Git jest znacznie uproszczona dzięki programowi RStudio. Operację kopiowania repozytorium najłatwiej przeprowadzić, wybierając w menu `New project.../Version control`.

Pojawi się okno jak na rysunku 3.10, w którym można określić parametry globalnego repozytorium do klonowania.

Na dysku zostanie utworzona kopia repozytorium we wskazanym katalogu. Na tej kopii możemy teraz pracować, modyfikując, dodając i usuwając z niej pliki.



Rysunek 3.10. Okno programu RStudio z parametrami do kopiowania repozytorium ze zdalnego serwera git. W tym przypadku kopiowane jest repozytorium `PrzewodnikParkiet` z GitHuba



Rysunek 3.11. Okno programu RStudio z zaznaczonymi plikami do dodania zmian (prawy górny panel)

3.5.2. Jak dodawać zmiany – `commit`

Po wprowadzeniu zmian w katalogu z projektem kolejnym krokiem jest zapisanie zmian w repozytorium Git.

Po prawej stronie w oknie projektu RStudio (patrz rysunek 3.11) umieszczona jest informacja, które pliki na dysku różnią się w stosunku do plików z repozytorium.

Możemy w tym oknie zaznaczyć, które zmiany chcemy zapisać w repozytorium. Po ich wybraniu klikamy przycisk `Commit`, aby zmiana została zapisana.

3.6. Debugger

Bez względu na doświadczenie, umiejętności, wiedzę i wiek, nie da się pisać dużych programów pozbawionych błędów.

Błędy biorą się z sytuacji, których nie przewidzieliśmy. W tym podrozdziale opiszemy źródła najczęstszych błędów oraz techniki pozwalające na ich odkrycie i naprawienie.

3.6.1. Co mogłoby pójść źle?

Spójrzmy na poniższe dwie instrukcje. Pierwsza wyznacza zbiór indeksów, w tym przypadku na konsoli wyświetlają się liczby 7, 8, 9. Druga instrukcja odczytuje element wektora `LETTERS`. Tutaj niespodzianka! Zamiast liter G, H i I wyświetla się podwójne G!

```
(inds <- seq(0.7, 0.9, by = 0.1)*10)
## [1] 7 8 9
LETTERS[inds]
## [1] "G" "G" "I"
```

Widząc wynik można domyślić się, że jego przyczyną są błędy wynikające z zaokrągleń numerycznych. Ale któż mógł je przewidzieć? Innym częstym błędem jest indeksowanie pętli `for` za pomocą sekwencji `1:length(v)`, co miałoby znaczyć od 1 do ostatniego elementu wektora `v`. Ale dla pustych wektorów to nie jest zbyt dobry pomysł.

```
v <- c()
1:length(v)
## [1] 1 0
```

Nawet program, który wydawał się być wolny od błędów, po zainstalowaniu nowej wersji jakiegoś zależnego pakietu może przestać działać lub zgłaszać bardzo dziwne wyniki. Należy więc nauczyć się wyszukiwać i naprawiać błędy.

3.6.2. Błędy i ostrzeżenia

Błędy biorą się z sytuacji, których nie przewidzieliśmy. Z tego też powodu bardzo trudno jest je wykryć patrząc uważnie w kod i analizując go spokojnie linia po linii. Przecież my te linie pisaliśmy, więc je znamy. Problem na tym, że nie przewidzieliśmy do czego mogą one prowadzić.

To, że w kodzie jest błąd daje się odkryć dopiero gdy zauważymy, że wyniki są podejrzane, gdy kod przerwie się niespodziewanie lub zakończy się ostrzeżeniem. Niektóre błędy żyją w ukryciu latami, więc należy doprowadzać do sytuacji, w których błąd (jeżeli jest) ujawni się możliwie szybko.

Sytuacje awaryjne w programie R raportowane są jako *błędy* lub jako *ostrzeżenia*.

Błędy (and. *errors*) przerywają wykonanie pętli, funkcji, bloku kodu i innych instrukcji. Wystąpienie błędu powoduje wywołanie funkcji `stop()`, która z kolei wykonuje akcję określoną przez globalny parametr `error`. Zmieniając wartość tego parametru możemy zażądać, aby R przed przerwaniem wykonywanych instrukcji zrobił coś jeszcze, no. zapisał stan wszystkich aktualnych zmiennych. Umożliwi to analizę *post-mortem* tego, co się stało i ułatwi znalezienie przyczyny wystąpienia błędu.

Ostrzeżenia (ang. *warnings*) też są raportowane, ale nie powodują przerwania wykonywania podprogramu. Po zakończeniu wykonywania funkcji, pod jej wynikiem, wypisywane są ostrzeżenia wygenerowane w trakcie działania funkcji.

Przyjrzyjmy się błędogennej operacji. Na potrzeby poniższych przykładów zdefiniujmy funkcję wyznaczającą logarytm.

```
funkcja <- function(x) {
   log(x)
}
```

Jeżeli wywołamy tę funkcję z argumentami złego typu, to zgłoszony zostanie błąd.

```
funkcja("2")
## log: Using log base e.
## Error in log(x) : Non-numeric argument to mathematical function
## Execution halted
```

3.6.3. Co można zrobić post-mortem – funkcja `traceback()`

Jeżeli wykonywana instrukcja została przerwana, ale nie wiemy gdzie dokładnie wystąpił błąd, to wygodnie jest użyć funkcji `traceback()`. Wypisuje ona listę ostatnio wywołanych funkcji wraz z wartościami argumentów w wywołaniu.

Aby zobaczyć jak ta funkcja działa, zdefiniujmy jeszcze jedną pomocniczą funkcję, której wywołanie doprowadzi do błędu.

```
funkcja2 <- function(x,y) {funkcja(x); funkcja(y)}
funkcja2(1,"jeden")
## Error in log(x) : Non-numeric argument to mathematical function
```

Wywołując funkcję `traceback()` możemy prześledzić, które funkcje i z jakimi argumentami były wywoływane w miejscu gdzie wystąpił błąd.

```
traceback()
## 2: funkcja(y)
## 1: funkcja2(1, "jeden")
```

3.6.4. Jak zastawić pułapkę – funkcja `recover()`

Funkcja `recover()` pozwala na interaktywną eksplorację stanu funkcji, w której wystąpił błąd. Jeżeli chcemy, aby po napotkaniu błędu natychmiast pojawiło się okienko debuggera, należy wskazać funkcję `recover()` jako instrukcje do wykonania w sytuacji błędogennej.

```
options(error = recover)
```

Teraz pojawienie się błędu spowoduje otworzenie się okna debuggera. Poniżej przedstawiamy przykładową sesję po wywołaniu `funkcja("2")`. Wyświetla się lista otwartych środowisk, możemy wybrać jedno i je eksplorować.

```
## Execution halted
## Available environments had calls:
## 1: funkcja("napis")
## 2: log(x)
## 3: .execute(.Primitive("log"), x, envir = sys.parent(1))
## Enter an environment number, or 0 to exit  Selection: 1
## Browsing in the environment with call:
##    funkcja("2")
## Called from: debugger.look(ind)
```

Sprawdźmy wartość zmiennej `x` w tym miejscu

```
Browse[1]> x
## [1] "2"
Browse[1]>
```

Pusta linia powraca do menu wyboru środowisk. Wybranie wartości `0` kończy pracę debuggera.

```
## Available environments had calls:
## 1: funkcja("napis")
## 2: log(x)
## 3: .execute(.Primitive("log"), x, envir = sys.parent(1))
## Enter an environment number, or 0 to exit  Selection: 0
```

Po uruchomieniu funkcji `recover()` wyświetla się lista dostępnych przestrzeni nazw związanych z wywoływaniami kolejnych funkcji.

W rozważanym przypadku, błąd został zgłoszony przez wbudowaną w programie R funkcję oznaczoną `.Primitive("log")`. Ta wbudowana funkcja została wywołana przez funkcję `log()`, a ta z kolei przez funkcję `funkcja()`.

Pierwsze pytanie debuggera dotyczy numeru przestrzeni nazw, którą chcemy prześledzić. W powyższym przykładzie wybieramy `1`, a więc przestrzeń nazw widzianą przez funkcję `funkcja()`. Po tym wyborze linia konsoli poleceń rozpoczyna się prefixem `Browse[1]>`, który oznacza, że jesteśmy w przestrzeni nazw opatrzonej indeksem 1. Możemy teraz sprawdzić wartości zmiennych w tej przestrzeni nazw w chwili, w której pojawił się błąd. W naszym przykładzie sprawdziliśmy wartość zmiennej `x`, czyli argumentu funkcji `funkcja()`. Okazało się, że jest to napis `"2"`. Acha, na tym polegał problem!

Możemy teraz opuścić debugger klikając `ENTER`, a następnie wybierając przestrzeń nazw o numerze `0`.

3.6.5. Jak śledzić krok po kroku – funkcja `debug()`

Jeszcze innym sposobem szukania błędów jest debugowanie funkcji krok po kroku. Do tego celu służy funkcja `debug()`.

Zaznacza ona, że wskazana argumentem funkcja ma być uruchamiana w trybie śledzenia krok po kroku. W trakcie wykonywania tej funkcji można korzystać z poleceń: `n` (wykonaj kolejną linię kodu), `c` (wykonaj całą funkcję) lub `Q` (zamknij debugger). Debugowanie wyłącza się funkcją `undebug()`.

Na poniższym przykładzie prezentujemy działanie funkcji `debug()`.

W pierwszej linii zaznaczamy funkcję `funkcja2()` do debugowania. Następnie wywołujemy tę funkcję, przez co uruchamia się też debugger.

```
debug(funkcja2)
funkcja2(1, "jeden")
## debugging in: funkcja2(1, "jeden")
## debug: {
##     funkcja(x)
##     funkcja(y)
## }
```

W tej chwili mamy dostęp do środowiska sprzed uruchomienia `funkcja2` (ale po zainicjowaniu argumentów). Sprawdźmy wartość `x`

```
Browse[1]> x
## [1] 1
```

Polecenie `n` wykonuje kolejną linię.

```
Browse[1]> n
## debug: funkcja(x)
```

A teraz sprawdźmy zawartość zmiennej y.

```
Browse[1]> y
## [1] "jeden"
```

Idziemy dalej dwa kroki i napotykamy na błąd.

```
Browse[1]> n
## debug: funkcja(y)
Browse[1]> n
## Error in log(x) : Non-numeric argument to mathematical function
```

3.6.6. Jak ignorować błędy – funkcja `try()`

Błędy i ostrzeżenia możemy zgłaszać używając funkcji `simpleError()` i `simpleWarning()`. Obie znajdują się w pakiecie `base` i obie służą do przygotowania komunikatu o błędzie lub ostrzeżeniu. W obu, za pierwszy argument należy podać komunikat błędu. Błąd lub ostrzeżenie można wygenerować funkcjami `stop()` lub `warning()`.

Jeżeli spodziewamy się ostrzeżeń w uruchamianym poleceniu i chcemy te ostrzeżenia zignorować, to możemy użyć funkcji `suppressWarnings()`. Argumentem tej funkcji jest polecenie do wykonania. Wyznaczony zostanie wynik, a dodatkowo zignorowane zostaną wszystkie wygenerowane ostrzeżenia.

Podobnie jest z błędami. Jeżeli spodziewamy się, że dane polecenie może wywołać błąd i na ten błąd chcemy odpowiednio zareagować (inaczej niż przerwaniem podprogramu), to możemy użyć funkcji `try()`. Ta funkcja wykona polecenie, podane jako jej pierwszy argument.

Jeżeli polecenie wykona się bez błędu, to wynikiem funkcji `try()` będzie wyznaczony wynik tego polecenia. Jeżeli podczas wykonywania wystąpi błąd, to wynikiem funkcji `try()` będzie obiekt klasy `try-error` zawierający komunikat o błędzie. Jeżeli nie chcemy, by komunikat o błędach był sygnalizowany, to należy za drugi argument funkcji `try()` podać `silent=TRUE`.

Poniżej przykład bezpiecznego uruchomienia funkcji `cor()`. Nawet, jeżeli funkcja `cor()` wygeneruje błąd, to błąd ten zostanie wychwycony przez funkcję `try()` i nie przerwie działania programu. Wywołanie zwykłej funkcji `cor()` zakończyłoby się błędem. Warunek `class(tmp)=="try-error"` to sprawdzenie czy został wygenerowany błąd i odpowiednia reakcja na ten błąd.

```
bezpieczna.cor <- function(x) {
   tmp <- try(cor(x), silent=TRUE)
   if (class(tmp)=="try-error") {
     cat("Korelacja nie zostala policzona\n")
     tmp <- 0
    }
   tmp
}
```

Przykładowe uruchomienie.

```
cor(1)
## Error in cor(1) : supply both 'x' and 'y' or a matrix-like 'x'
```

Wywołanie bezpiecznej wersji funkcji `cor()` nie sygnalizuje błędu.

```
bezpieczna.cor(1)
## Korelacja nie zostala policzona
## [1] 0
```

3.6.7. Zaokrąglenia numeryczne – studium przypadku

Ten rozdział otwierał przykład błędy wygenerowanego przez ograniczenia arytmetyki zmiennoprzecinkowej. Jest to częste źródło błędów, więc przyjrzyjmy mu się bliżej

```
(v <- seq(0.7, 0.8, by=0.1))
## [1] 0.7 0.8
v == 0.8
## [1] FALSE FALSE
```

Ponieważ różnica pomiędzy `0.8` a `0.7+0.1` jest większa od stałej `double.neg.eps`, dlatego program R traktuje je jako różne liczby. Wartości `double.neg.eps` i `double.eps` określają maszynowy epsilon, czyli maksymalną różnicę pomiędzy liczbami, do której dwie liczby uznawana są za równe.

```
(0.7+0.1) - 0.8
## [1] -1.110223e-16
```

Powyższa różnica jest większa równa niż zmienna `double.neg.eps`. Jeszcze większe błędy numeryczne są widoczne przy potęgowaniu, np. ciekawemu czytelnikowi polecam sprawdzenie ile wynosi $\left(\sqrt{2}\right)^2 - 2$.

Należy być ostrożnym porównując liczby lub wektory liczb rzeczywistych. Problemy, wynikające z zaokrągleń oraz dokładności reprezentacji liczb rzeczywistych w pamięci komputera, mogą pojawić się nawet w bardzo prostych i niewinnie wyglądających sytuacjach.
Jeżeli chcemy mieć pewność, że super-dokładnie wyznaczymy wynik, to należy korzystać z obliczeń symbolicznych lub obliczeń na liczbach całkowitych (np. w banku licząc pieniądze nie możemy pozwolić sobie na żadne niedokładności).

Warto przy okazji wspomnieć o dwóch zmiennych: `.Machine` i `.Platform`, które przechowują informacje o parametrach systemu operacyjnego, w którym pracujemy oraz o parametrach maszyny obliczeniowej, na której pracujemy.

Każda z tych zmiennych jest listą wartości opisujących właściwości maszyny lub platformy, a więc jak reprezentowane są liczby w pamięci maszyny oraz z jaką dokładnością wykonywane są operacje. Dokładny opis poszczególnych pól tych list znaleźć można w pomocy dla obiektu `.Machine`.

```
unlist(.Machine)
##            double.eps        double.neg.eps
##          2.220446e-16          1.110223e-16
##            double.xmax           double.base
##          1.797693e+308          2.000000e+00
##            double.rounding       double.guard
##          5.000000e+00          0.000000e+00
...
```

I zmienna `.Platform`.

```
unlist(.Platform)
##      OS.type     file.sep   dynlib.ext          GUI       endian
##    "windows"          "/"       ".dll"       "Rgui"     "little"
##      pkgType     path.sep       r_arch
##    "win.binary"       ";"            ""
```

3.7. Profiler

Profiler to narzędzie pozwalające analizować zajętość czasu procesora oraz użycie pamięci przy wykonaniu funkcji programu R. Narzędzie to może być bardzo przydatne do sprawdzenia, które elementy programu są czaso- lub pamięciożerne. Wyniki profilera można wykorzystać, aby przepisać wybrany problematyczny fragment kodu w inny sposób, tak by poprawić wydajność całego programu. W przypadku bardzo czasochłonnych obliczeń może być opłacalne napisanie wrażliwego fragmentu w języku kompilowanym, np. w C i wywoływanie takich zewnętrznych funkcji z programu R.

Jednym z częstych błędów w tworzeniu złożonych programów jest optymalizacja na siłę każdego elementu kodu. Często prowadzi to do znacznej komplikacji kodu, co w konsekwencji prowadzi do błędów. Jeżeli nasz program jest wolny, to powinniśmy wykorzystać profiler aby ocenić który jego element jest wąskim gardłem i należy skupić się na przyśpieszeniu tego wąskiego gardła.

3.7.1. Jak mierzyć czas działania bloku instrukcji

Najłatwiejszym sposobem sprawdzenia, jak długo wykonuje się pewien fragment kodu, jest użycie funkcji `system.time()` lub `proc.time()`.

Wynikiem funkcji `proc.time()` jest wektor trzech liczb, określający ile czasu procesora zużyło środowisko R od początku pracy do danej chwili, ile czasu

procesora zużył system oraz ile czasu minęło od uruchomienia środowiska R. Do wyznaczania ile czasu trwało wykonanie określonego polecenia funkcja `system.time()` korzysta z dwóch wywołań funkcji `proc.time()`, przed i po zakończeniu działania interesującego polecenia.

Poniższy przykład pokazuje, jak używać funkcji `system.time()`. Przedstawione są wyniki porównania czasów czterech sposobów wypełnienia wektora liczbami losowymi. Przy okazji zauważmy, że najszybsze są operacje na całych wektorach. Oznacza to, że jeżeli już trzeba używać pętli, to wewnątrz pętli wektory nie powinny być dynamicznie powiększane.

Zaczynamy od przykładu z dynamicznym powiększaniem wektora. Takie rozszerzanie wektora przez doklejanie zajmuje ponad 62 sekundy .

```
system.time({ x=NULL; for (i in 1:10^5) x =c(x, runif(1)) })
##    user  system elapsed
##   62.54    0.16   62.94
```

Ta wersja jest tak samo czasożerna, choć zapis jest nieznacznie krótszy.

```
system.time({ x=NULL; for (i in 1:10^5) x[i] =runif(1) })
##    user  system elapsed
##   64.04    0.20   65.75
```

Jeżeli przed pętlą zainicjujemy wektor `x` o rozmiarze 10^5 elementów, to ponad 50 razy skrócimy czas wykonywania się tego fragmentu kodu. Tylko dzięki uniknięciu dynamicznego rozszerzania wektora.

```
system.time({x=numeric(10^5); for(i in 1:10^5) x[i]=runif(1)})
##    user  system elapsed
##    1.24    0.00    1.26
```

Usuwając pętlę skrócimy czas wykonywania 6000 razy.

```
system.time({ x=NULL; x = runif(10^5) })
##    user  system elapsed
##    0.02    0.00    0.01
```

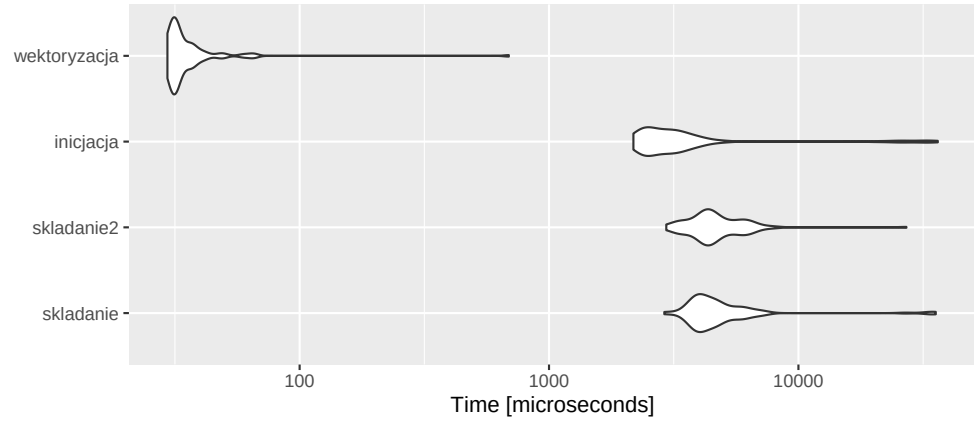
3.7.2. Jak mierzyć czas co do milisekundy

Jeżeli mierzenie czasu działania z dokładnością do sekundy nie jest wystarczające, to warto skorzystać z funkcji `microbenchmark` z biblioteki o tej samej nazwie.

Pozwala ona na porównanie czasów działania określonych instrukcji z dokładnością do milisekundy. Dodatkowo powtarza ona wywoływanie zadanych instrukcji wiele razy (domyślnie 100), aby ocenić na ile stabilne są prezentowane instrukcje. Wyniki pomiarów przestawia graficznie i tekstowo.

W poniższym przykładzie porównujemy ponownie cztery sposoby losowania wektora liczb. Graficzne podsumowanie tych pomiarów przedstawiamy na rysunku 3.12.

```
library("microbenchmark")
library("ggplot2")
pomiary <- microbenchmark(
  skladanie    = { x=NULL; for (i in 1:10^3) x =c(x, runif(1)) },
  skladanie2   = { x=NULL; for (i in 1:10^3) x[i] =runif(1) },
  inicjacja    = { x=numeric(10^3); for(i in 1:10^3) x[i]=runif(1) },
  wektoryzacja = { x=NULL; x = runif(10^3) })
pomiary
## Unit: microseconds
##          expr    min     mean  median     max neval cld
##     skladanie 2899.0 5722.777 4366.66 35529.7   100   b
##    skladanie2 2952.0 4908.663 4446.69 27099.2   100   b
##     inicjacja 2179.3 4759.810 2937.46 36207.7   100   b
##  wektoryzacja   29.5   41.834   32.28   689.8   100  a
autoplot(pomiary)
```



Rysunek 3.12. Graficzna prezentacja pomiarów czasu działania wykonanych przez bibliotekę `microbenchmark`

3.7.3. Jak szukać wąskich gardeł

Gdy bardziej niż czas działania pewnego bloku kodu interesuje nas czas działania jego składowych, wtedy przydaje się funkcja `Rprof()`.

Pierwsze wywołanie tej funkcji powoduje włączenie zapisywania informacji o działaniu podprogramu do pliku określonego przez argument `filename` (domyślnie do pliku `"Rprof.out"`). Aby przerwać zapisywanie informacji dla profilera, należy uruchomić funkcję `Rprof` z argumentem `NULL`. Co określony czas (argument `interval`, domyślnie 0.02 sek) profiler sprawdza jak wygląda stos wywołań R. Informacje o tym stosie wywołań są zapisywane we wskazanym pliku. Wyniki tego profilera można podsumować funkcją `summaryRprof()`.

Poniżej przedstawiamy przykład wykorzystania profilera.

```
Rprof("profiler.out", interval = 0.01, memory.profiling = TRUE)
  N   <- 2000
  df  <- as.data.frame(matrix(runif(2*N*N), 2*N, N))
  tmp <- summary(lm(V1~., data=df))
Rprof()
```

Powyższe instrukcje utworzą w katalogu roboczym plik `profiler.out` z informacjami o wywołaniach. Fragment tego pliku jest przedstawiony poniżej. Pierwsze kolumny opisują zajętość pamięci, ostatnia wymienia funkcje, które aktualnie znajdują się na stosie wywołania.

```
memory profiling: sample.interval=10000
:2707201:50881097:139412560:121:"runif" "matrix" "as.data.frame"
:2707201:50881097:139412560:0:"runif" "matrix" "as.data.frame"
:2707206:58881410:139413624:0:"matrix" "as.data.frame"
:2707206:58881410:139413624:0:"matrix" "as.data.frame"
:2708210:64700854:140252056:0:"as.vector" "as.data.frame.matrix" "as.
    data.frame"
```

Analiza organoleptyczna pliku `profiler.out` nie jest łatwa, ponieważ ten plik jest zazwyczaj duży i trzeba go odpowiednio przetworzyć aby cokolwiek zauważyć.

Dlatego do jego analizy wygodniej jest użyć funkcji `summaryRprof()`. W wyniku zwraca ona listę z czterema elementami:

- `by.self` informacja, która funkcja jak długo działała, uporządkowana zgodnie z czasem działania wyłącznie danej funkcji bez podfunkcji.
- `by.total` informacja, która funkcja jak długo działała wliczając czas działania podfunkcji,
- `sample.interval`, częstość próbkowania kodu,
- `sampling.time`, czas trwania całego analizowanego kodu.

Poniżej przedstawiamy przykład działania

```
summaryRprof("profiler.out")$by.self
##                       self.time self.pct total.time total.pct
## "lm.fit"                   9.84    75.17       9.84     75.17
## "chol2inv"                 1.89    14.44       1.89     14.44
## "runif"                    0.30     2.29       0.30      2.29
## "na.omit.data.frame"       0.15     1.15       0.26      1.99
## "makepredictcall.default" 0.12     0.92       0.12      0.92
## ".External2"               0.11     0.84       0.42      3.21
## "matrix"                   0.08     0.61       0.38      2.90
...
```

Na załączonym przykładzie najwięcej czasu na wykonanie potrzebowała funkcja `lm.fit()` (około 75% całkowitego czasu wywołania).

Poza wymienionymi powyżej narzędziami, do profilowania i wizualizacji informacji z profilowanego kodu można wykorzystać pakiety `proftools` (wykorzystuje pakiet graficzny `Rgraphviz` do graficznego przedstawienia wyników) oraz `profr` (wykorzystuje pakiet graficzny `ggplot2`).

Poniżej przedstawiamy przykładowe użycie pakietu `profr`. Wykres, będący wynikiem tego kodu jest przedstawiony na rysunku 3.13. Wyniki profilera są konwertowane do postaci rozpoznawalnej przez funkcję plot.

Włączamy proces profilowania wykonania kodu. Pierwsza funkcja trwa około 25 sek, druga funkcja trwa około 12 sek.

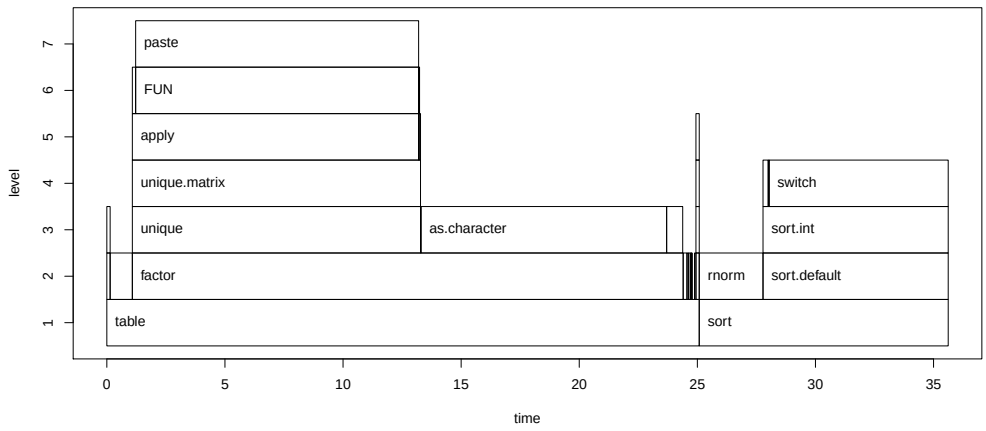
```
Rprof("profiler.out", interval = 0.01)
tmp <- table(outer(1:N, 1:N, "*") %% 10)
tmp <- sort(rnorm(10^7))
Rprof(NULL)
```

Aby użyć pakietu `profr` należy przetransformować wyniki profilera do innej postaci.

```
out <- parse_rprof("profiler.out")
## Read 1782 items
```

Przedstawmy graficznie czasy działania. Wynik poniższej instrukcji jest na rysunku 3.13.

```
plot(out)
```



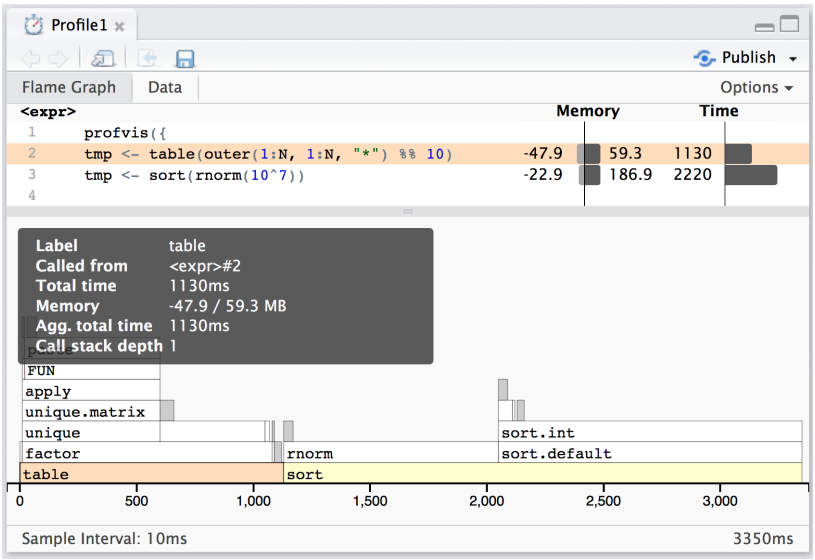
Rysunek 3.13. Przykład użycia pakietu `profr` do wizualizacji danych o czasie działania. Na osi X prezentowany jest czas w sekundach począwszy od rozpoczęcia profilowania. Każdy z prostokątów odpowiada za czas działania kolejnych funkcji w tym również pod-funkcji

3.7.4. Jak przedstawiać graficznie wyniki profilowania

Użytkownicy programu RStudio mogą skorzystać z jeszcze jednej opcji graficznego profilowania kodu, a mianowicie z pakietu `profvis`. Dla zadanego bloku kody uruchamia on pod spodem funkcję `Rprof`, zbiera informacje o wykonaniu zadanego bloku kody a następnie pozwala na interaktywną eksplorację czasów działania.

Poniżej przedstawiamy przykładowe wywołanie. W wyniku otrzymujemy interaktywną aplikację przestawioną na rysunku 3.14.

```
library("profvis")
profvis({
tmp <- table(outer(1:N, 1:N, "*") %% 10)
tmp <- sort(rnorm(10^7))
})
```



Rysunek 3.14. Graficzna prezentacja profilowania kodu z użyciem biblioteki `profvis`

3.7.5. Jak zrównoleglać obliczenia

Jeżeli mamy do wykonania obliczenia (np. symulacje), które trwałyby 10 dni, to mamy do wyboru trzy możliwości:

- poczekać 10 dni, w sumie to nie tak długo. Choć co jeżeli to 10 lat a nie 10 dni?
- poszukać szybszego komputera, jeżeli nasz nie jest najszybszy to może znajdziemy jakiś 2-3 razy szybszy. A może nawet 5 razy szybszy, dzięki czemu z 10 lat zrobią się tyko dwa lata.

Indeks

argument
- alternative, 250
- bg, 355
- cex, 355
- col, 344, 353
- col.main, 353
- col.sub, 353
- error, 120
- header, 54
- lty, 354
- lwd, 354
- main, 344, 356
- mfcol, 360
- mfrow, 360
- pch, 355
- sub, 356
- type, 344, 354
- xlab, 356
- ylab, 356

dane
- alfabetycznie
  - koty_ptaki, 20
- PogromcyDanych
  - koty_ptaki, 20

funkcja
- utils
  - Sweave, 74
- agricolae
  - bar.err, 200
  - bar.group, 200
  - durbin.test, 201
  - HSD.test, 200
  - LSD.test, 201
  - waller.test, 201
- alfabetycznie
  - :, 18
  - [,], 21
  - [[]], 23
  - [], 19
  - a_ply, 159
  - aaply , 159
  - abline, 345, 356
  - abs, 289
  - acast, 157
  - acos, 289
  - ad.test, 247
  - add1.glm, 232
  - addlogo, 351
  - adply , 159
  - aggregate, 155
  - alply , 159
  - anova, 198, 217
  - anova.coxph, 286
  - anova.glm, 232
  - anova.survreg, 286
  - ansari.test, 257
  - aov, 199
  - apply, 135, 191
  - apropos, 13
  - aregImpute, 188
  - Arg, 290
  - args, 13
  - arrows, 356
  - as, 150
  - as.complex, 290
  - as.matrix, 369
  - asin, 289
  - atan, 289
  - atan2, 289
  - attr, 147
  - attributes, 147
  - axis, 348
  - bar.err, 200
  - bar.group, 200
  - barchart, 304, 310
  - barplot, 28
  - bartlett.test, 257
  - beta, 290
  - betareg, 241
  - bind_cols, 22
  - bind_rows, 22
  - binom.test, 260
  - bmp, 298
  - boot, 276
  - boot.ci, 278
  - box.cox, 192
  - boxcox, 192
  - boxplot, 31
  - browseEnv, 162
  - bwplot, 304, 310
  - by, 155
  - c, 18
  - cbind, 22
  - cdplot, 229

ce.mimp, 189
ceiling, 289
chisq.test, 248, 263
chol, 179
choose, 290
choose.files, 164
class, 147
cloud, 304, 316
cm.colors, 353
coef, 217
col2rgb, 354
colors, 353
colours, 353
combn, 290
comment, 147
complete.cases, 187
complex, 290
confint.glm, 232
Conj, 290
contour, 368
contourplot, 304, 316
contr.helmert, 205
contr.poly, 205
contr.SAS, 205
contr.sdif, 205
contr.sum, 205
contr.treatment, 205
cooks.distance, 232
coord_cartesian, 340
coord_equal, 340
coord_flip, 340
coord_map, 340
coord_polar, 340
coord_trans, 340
coplot, 362
cor, 26, 261
cor.test, 261
cos, 289
cov, 26
cox.zph, 285
coxph, 285
ctree, 286
curve, 345
cut, 194
cvm.test, 247
D, 294
d_ply, 159
daply , 159
data, 50
data.ellipse, 364
data.frame, 22
dcast, 157
ddply , 159
debug, 122
density, 32, 229, 312
densityplot, 304
deparse, 142
deriv, 290, 294
dev.off, 298
deviance, 217
digamma, 290
dir, 163
dir.create, 164
dlply , 159
dotplot, 304, 310
drop1.glm, 232
durbin.test, 201
eapply, 155
ec.knnimp, 189
ecdf, 32
ecdfplot, 304
edit, 10
effects.glm, 232
eigen, 178
equal.count, 306
eval, 141
example, 13
exp, 289
expm1, 289
expression, 294, 351
extends, 150
factorial, 290
family, 227
file.append, 164
file.choose, 164
file.copy, 164
file.create, 164
file.exists, 164
file.info, 164
file.remove, 164
file.rename, 164
file.show, 164
file.symlink, 164
file_test, 164
filled.contour, 368
fisher.test, 263
fitdistr, 186
fitted, 217
fivenum, 25
fix, 10, 147
fligner.test, 257
floor, 289
fourfold, 366
fourfoldplot, 366
gam, 241
gamma, 290
gc, 145
GCD, 290
geometric.mean, 26
getEurostatRCV, 157
getwd, 162
glm, 226, 231
glm.diag.plots, 232
gModel, 242
gnls, 239
goodfit, 250
gpairs, 362
harmonic.mean, 26

heat.colors, 353
heatmap, 368
help, 13
help.search, 13
hist, 29
histogram, 304, 312
HSD.test, 200
hsv, 353
identify3d, 364
ifelse, 86
Im, 290
image, 350
impute, 189
initialize, 149
install.packages, 11
install$_{g}ithub$, 12
integral, 290
integrate, 295
interaction.plot, 207
intersect, 288
IQR, 26
is, 150
is.complex, 290
is.element, 288
is.na, 187
isClassUnion, 150
jitter, 361
jpeg, 298
kde, 366
kde2d, 366
knit, 68
kruskal.test, 254
ks.test, 249, 250
kurtosis, 26
l_ply, 159
laply , 159
lapply, 154
layout, 360
lbeta, 290
lchoose, 290
LCM, 290
ldply , 159
legend, 349
legendre.polynomials, 291
length, 19, 26
levelplot, 304, 316
levels, 194, 219
leverage.plot, 232
lfactorial, 290
lgamma, 290
library, 12
lillie.test, 247
lines, 344
list, 22
list.files, 163, 164
llply , 159
lm, 198, 203, 213, 345
lm.fit, 213
lm.ridge, 223, 224
lme, 239
loess, 239
log, 289
log10, 289
log1p, 289
log2, 289
logb, 289
logLik.glm, 232
logtrans, 194
lowess, 239
lqs, 225
lrm, 226
ls, 144
ls.str, 144
LSD.test, 201
lsf.str, 144
lsoda, 295
m_ply, 159
make.contrasts, 204
manova, 210
mantelhaen.test, 265
maply , 159
mapply, 155
matlines, 348
matplot, 348
matpoints, 348
max, 26
mcnemar.exact, 265
mcnemar.test, 265
mdply , 159
mean, 26
median, 26
melt, 157
microbenchmark, 126
min, 26
mlply , 159
Mod, 290
mod, 26
moda, 26
mode, 147
mood.test, 257
mosaicplot, 36
mt.rawp2adjp, 271
mtext, 360
mvr, 238
na.omit, 187
new, 149
nlm, 234, 294
nlme, 239
nlrq, 241
nls, 234
object.size, 147
objects, 144
optim, 294
optimize, 293
order, 137
outer, 135, 368
outlier.test, 232
p.adjust, 271

pageWithSidebar, 99
pairs, 362
pairwise.prop.test, 260
pairwise.t.test, 255
par, 370
parallel, 304, 312
parcoord, 369
parent.env, 144
parse.SAScii, 66
pdf, 298
pearson.test, 247
persp, 368
pictex, 300
plot, 217, 342, 344, 347, 351
plot.design, 207
plot.lm, 218
plotcorr, 364
plotmath, 351
plotOutput, 100
png, 298
points, 356
poly.calc, 290
polygon, 356
polynomial, 290
poscript, 298
pR2, 231
predict, 217, 222, 229
predict.glm, 232
print.summary.lm, 215
proc.time, 125
prop.test, 258
psigamma, 290
qnorm, 167
qq, 304, 312
qq.plot, 245
qq.plot.glm, 232
qqmath, 304, 312
qqnorm, 245
qqplot, 251
quantile, 26
r_ply, 159
rainbow, 353
range, 26
raply , 159
rapply, 155
rbind, 22
rdply , 159
Re, 290
read.arff, 66
read.dbf, 66
read.dta, 66
read.epiinfo, 66
read.fwf, 61
read.mtp, 66
read.octave, 66
read.pnm, 351
read.S, 66
read.sas7bdat, 66
read.SAScii, 66
read.spss, 65, 66
read.ssd, 66
read.systat, 66
read.table, 54
read.xport, 66
readMat, 65, 66
rect, 356
renderPlot, 102
renderPrint, 102
reorder, 219
rep, 368
replicate, 155, 161
resid, 217
residuals.glm, 232, 233
return, 81
rgb, 353
rgb2hsv, 354
rgl.surface, 368
rk4, 295
rlm, 225
rlply , 159
rm, 145
RNGkind, 166, 167
rnmvnorm, 179
round, 289
Rprof, 127
rq, 241
rstandard.glm, 232
rstudent.glm, 232
rug, 30
runApp, 100
sample, 137
sample.int, 137
sapply, 135, 154
sas.get, 66
scan, 60
scatter3d, 362
scatterplot, 239, 361
scatterplot.matrix, 362
scatterplot3d, 362
sd, 26
segmented, 242
segments, 356
selectInput, 102
sem, 242
seq, 18
seq_along, 90
set.seed, 168
setClass, 149
setClassUnion, 150
setdiff, 288
setequal, 288
setwd, 162
sf.test, 247
shapiro.test, 247
shell, 162
shell.exec, 162
shingle, 306
show.settings, 321

sign.test, 255
signif, 289
simpleError, 123
simpleWarning, 123
sin, 289
sink, 163
skewness, 26
slegendre.polynomials, 291
sliderInput, 102
slot, 149
slotNames, 150
smooth.spline, 239
solve, 290
sort, 137
sp, 35, 361
specify.model, 242
split, 161
splom, 304, 308
spss.get, 64, 66
sqrt, 289
step, 220, 228
step.glm, 232
stop, 120, 123
str, 147
stripplot, 304, 310
substitute, 142
summary, 26, 215, 217, 228, 342
summary.glm, 232
summary.manova, 210
suppressWarnings, 123
supsmu, 239
Surv, 282
surv_test, 284
survdiff, 284
survfit, 282
survreg, 286
svd, 178
svdpc.fit, 238
Sweave, 74
sweep, 191
switch, 87
symbols, 356
system, 161
system.time, 125
t.test, 254
table, 26, 248
tan, 289
tapply, 155
tempdir, 163
tempfile, 163
terrain.colors, 353
text, 345, 360
theme, 341
theme_bw, 341
tiff, 298
title, 356
toBibtex, 77
toLatex, 77
topo.color, 353
traceback, 120
transcan, 188
transform, 194
trellis.par.get, 321
trigamma, 290
trunc, 289
try, 123
typeof, 147
unclass, 147
undebug, 122
union, 288
uniroot, 293
unlink, 164
unlist, 154
UseMethod, 139
var, 26
var.test, 257
verbatimTextOutput, 100
vif, 223
vignette, 78
waller.test, 201
warning, 123
weighted.mean, 26
wilcox.test, 254
wireframe, 304, 316, 368
write.table, 57
X11, 300
xfig, 300
xtable, 76
xyplot, 304, 308

base

:, 18
[,], 21
[[]], 23
[], 19
abs, 289
acos, 289
aggregate, 155
apply, 135, 191
Arg, 290
as.complex, 290
as.matrix, 369
asin, 289
atan, 289
atan2, 289
attr, 147
attributes, 147
beta, 290
browseEnv, 162
by, 155
c, 18
cbind, 22
ceiling, 289
chol, 179
choose, 290
choose.files, 164
class, 147
combn, 290
comment, 147

complex, 290
Conj, 290
cos, 289
cut, 194
data.frame, 22
debug, 122
deparse, 142
digamma, 290
dir, 163
dir.create, 164
eapply, 155
edit, 10
eigen, 178
eval, 141
exp, 289
expm1, 289
expression, 294, 351
factorial, 290
file.append, 164
file.choose, 164
file.copy, 164
file.create, 164
file.exists, 164
file.info, 164
file.remove, 164
file.rename, 164
file.show, 164
file.symlink, 164
fix, 10
floor, 289
gamma, 290
gc, 145
getwd, 162
ifelse, 86
Im, 290
intersect, 288
is.complex, 290
is.element, 288
is.na, 187
jitter, 361
lapply, 154
lbeta, 290
lchoose, 290
length, 19, 26
levels, 194, 219
lfactorial, 290
lgamma, 290
library, 12
list, 22
list.files, 163, 164
log, 289
log10, 289
log1p, 289
log2, 289
logb, 289
ls, 144
ls.str, 144
mapply, 155
max, 26
min, 26
Mod, 290
mode, 147
object.size, 147
objects, 144
order, 137
outer, 135, 368
parent.env, 144
proc.time, 125
psigamma, 290
range, 26
rapply, 155
rbind, 22
Re, 290
reorder, 219
rep, 368
replicate, 155, 161
return, 81
rm, 145
RNGkind, 166, 167
round, 289
Rprof, 127
sample, 137
sample.int, 137
sapply, 135, 154
scan, 60
seq, 18
seq_along, 90
set.seed, 168
setClass, 149
setdiff, 288
setequal, 288
setwd, 162
shell, 162
shell.exec, 162
signif, 289
simpleError, 123
simpleWarning, 123
sin, 289
sink, 163
sort, 137
split, 161
sqrt, 289
stop, 120, 123
str, 147
substitute, 142
summary, 26, 215, 217, 228
suppressWarnings, 123
svd, 178
sweep, 191
switch, 87
system, 161
system.time, 125
table, 26, 248
tan, 289
tapply, 155
tempdir, 163
tempfile, 163
traceback, 120

transform, 194
trigamma, 290
trunc, 289
try, 123
typeof, 147
unclass, 147
undebug, 122
union, 288
unlink, 164
unlist, 154
UseMethod, 139
warning, 123
betareg
betareg, 241
boot
boot, 276
boot.ci, 278
glm.diag.plots, 232
BSDA
sign.test, 255
car
box.cox, 192
data.ellipse, 364
qq.plot, 245
scatterplot, 35, 239, 361
scatterplot.matrix, 362
sp, 35, 361
vif, 223
Design
lrm, 226
devtools
install$_{g}ithub$, 12
dplyr
bind_cols, 22
bind_rows, 22
dprep
ce.mimp, 189
ec.knnimp, 189
moda, 26
e1071
impute, 189
kurtosis, 26
skewness, 26
ellipse
plotcorr, 364
exact2x2
mcnemar.exact, 265
foregin
read.xport, 66
foreign
read.arff, 66
read.dbf, 66
read.dta, 66
read.epiinfo, 66
read.mtp, 66
read.octave, 66
read.S, 66
read.spss, 65, 66
read.ssd, 66
read.systat, 66
read.xport, 66
gam
gam, 241
ggplot2
coord_cartesian, 340
coord_equal, 340
coord_flip, 340
coord_map, 340
coord_polar, 340
coord_trans, 340
plot, 342
summary, 342
theme_bw, 341
theme_grey, 341
gmodels
make.contrasts, 204
graphics
abline, 345, 356
arrows, 356
axis, 348
barplot, 28
boxplot, 31
cdplot, 229
colors, 353
colours, 353
contour, 368
coplot, 362
curve, 345
filled.contour, 368
fourfoldplot, 366
hist, 29
image, 350
layout, 360
legend, 349
lines, 344
matlines, 348
matplot, 348
matpoints, 348
mosaicplot, 36
mtext, 360
pairs, 362
par, 370
persp, 368
plot, 344, 347
plot.design, 207
points, 356
polygon, 356
rect, 356
rug, 30
segments, 356
symbols, 356
text, 345, 360
title, 356
gRbase
gModel, 242
grDevices
bmp, 298
cm.colors, 353

col2rgb, 354
dev.off, 298
heat.colors, 353
hsv, 353
jpeg, 298
pdf, 298
pictex, 300
plotmath, 351
png, 298
poscript, 298
rainbow, 353
rgb, 353
rgb2hsv, 354
terrain.colors, 353
tiff, 298
topo.color, 353
X11, 300
xfig, 300
Hmisc
aregImpute, 188
sas.get, 66
spss.get, 64, 66
transcan, 188
knitr
knit, 68
ks
kde, 366
lattice
barchart, 304, 310
bwplot, 304, 310
cloud, 304, 316
contourplot, 304, 316
density, 312
densityplot, 304
dotplot, 304, 310
ecdfplot, 304
equal.count, 306
histogram, 304, 312
levelplot, 304, 316
parallel, 304, 312
qq, 304, 312
qqmath, 304, 312
shingle, 306
show.settings, 321
splom, 304, 308
stripplot, 304, 310
trellis.par.get, 321
wireframe, 304, 316, 368
xyplot, 304, 308
MASS
boxcox, 192
contr.sdif, 205
fitdistr, 186
kde2d, 366
lm.ridge, 223, 224
logtrans, 194
lqs, 225
parcoord, 369
rlm, 225
methods
as, 150
extends, 150
initialize, 149
is, 150
isClassUnion, 150
new, 149
setClassUnion, 150
slot, 149
slotNames, 150
microbenchmark
microbenchmark, 126
multtest
mt.rawp2adjp, 271
mvnorm
rnmvnorm, 179
nlme
gnls, 239
lme, 239
nlme, 239
nortest
ad.test, 247
cvm.test, 247
lillie.test, 247
pearson.test, 247
sf.test, 247
shapiro.test, 247
odesolve
lsoda, 295
rk4, 295
orthopolynom
legendre.polynomials, 291
slegendre.polynomials, 291
party
ctree, 286
pixmap
addlogo, 351
plot, 351
read.pnm, 351
pls
mvr, 238
svdpc.fit, 238
plyr
a_ply, 159
aaply , 159
adply , 159
alply , 159
d_ply, 159
daply , 159
ddply , 159
dlply , 159
l_ply, 159
laply , 159
ldply , 159
llply , 159
m_ply, 159
maply , 159
mdply , 159
mlply , 159

r_ply, 159
raply , 159
rdply , 159
rlply , 159
polynom
deriv, 290
GCD, 290
integral, 290
LCM, 290
poly.calc, 290
polynomial, 290
solve, 290
pscl
pR2, 231
psych
geometric.mean, 26
harmonic.mean, 26
quantreg
nlrq, 241
rq, 241
R.matlab
readMat, 65, 66
Rcmdr
scatter3d, 362
reshape2
acast, 157
dcast, 157
melt, 157
rgl
rgl.surface, 368
RVAideMemoire
mod, 26
sas7bdat
read.sas7bdat, 66
SAScii
parse.SAScii, 66
read.SAScii, 66
scatterplot3d
identify3d, 364
scatterplot3d, 362
segmented
segmented, 242
sem
sem, 242
specify.model, 242
shiny
pageWithSidebar, 99
plotOutput, 100
renderPlot, 102
renderPrint, 102
runApp, 100
selectInput, 102
sliderInput, 102
verbatimTextOutput, 100
SmarterPoland
getEurostatRCV, 157
stats
add1.glm, 232
anova, 198, 217
anova.glm, 232
ansari.test, 257
aov, 199
bartlett.test, 257
binom.test, 260
chisq.test, 248, 263
coef, 217
complete.cases, 187
confint.glm, 232
contr.helmert, 205
contr.poly, 205
contr.SAS, 205
contr.sum, 205
contr.treatment, 205
convolve, 290
cooks.distance, 232
cor, 26, 261
cor.test, 261
cov, 26
D, 294
density, 32, 229
deriv, 294
deviance, 217
drop1.glm, 232
ecdf, 32
effects.glm, 232
family, 227
fisher.test, 263
fitted, 217
fivenum, 25
fligner.test, 257
glm, 226, 231
heatmap, 368
integrate, 295
interaction.plot, 207
IQR, 26
kruskal.test, 254
ks.test, 249, 250
leverage.plot, 232
lm, 198, 203, 213, 345
lm.fit, 213
loess, 239
logLik.glm, 232
lowess, 239
manova, 210
mantelhaen.test, 265
mcnemar.test, 265
mean, 26
median, 26
mood.test, 257
na.omit, 187
nlm, 234, 294
nls, 234
optim, 294
optimize, 293
outlier.test, 232
p.adjust, 271
pairwise.prop.test, 260
pairwise.t.test, 255

plot, 217
plot.lm, 218
predict, 217, 222, 229
predict.glm, 232
print.summary.lm, 215
prop.test, 258
qnorm, 167
qq.plot.glm, 232
qqnorm, 245
qqplot, 251
quantile, 26
resid, 217
residuals.glm, 232, 233
rstandard.glm, 232
rstudent.glm, 232
sd, 26
smooth.spline, 239
step, 220, 228
step.glm, 232
summary.glm, 232
summary.manova, 210
supsmu, 239
t.test, 254
uniroot, 293
var, 26
var.test, 257
weighted.mean, 26
wilcox.test, 254
survival
anova.coxph, 286
anova.survreg, 286
cox.zph, 285
coxph, 285
Surv, 282
survdiff, 284
survfit, 282
survreg, 286
utils
apropos, 13, 14
args, 13, 14
data, 50
example, 13, 14
file_test, 164
find, 14
fix, 147
help, 13
help.search, 13, 14
install.packages, 11
lsf.str, 144
read.fwf, 61
read.table, 54
toBibtex, 77
toLatex, 77
vignette, 78
write.table, 57
vcd
fourfold, 366
goodfit, 250
xtable
xtable, 76
YaleToolkit
gpairs, 362
funkcje
arytmetyczne, 289
do manipulacji właściwościami obiektów, 147
do operacji na plikach, 164
dystrybuanty i gęstości, 173
generatory liczb losowych, 173
importu/eksportu danych, 66
polimorficzne, 138
specjalne, 290
trygonometryczne, 289
tworzące kontrasty, 205
z rodziny apply, 155

instrukcja
for, 89
function, 85
if, 85
if-else, 85
repeat, 91
switch, 87
while, 91

klasa
anova, 198
call, 141, 294
data.table, 138
ecdf, 34
expression, 294, 351
glm, 227, 233
histogram, 30
htest, 244, 247
lm, 217
nls, 235
summary, 144
summary.glm, 232
summary.lm, 217
Surv, 282
survfit, 283
try-error, 123

operator
::, 13
:::, 13
malpka, 149
operatory, 140

pakiet
agricolae, 200
betareg, 241
biopara, 137
bnlearn, 242
boot, 276
bootstrap, 276
BSDA, 50
colorRamps, 353
contrast, 203

copula, 168
cudaBayesreg, 137
debug, 120
dplyr, 96
e1071, 26
evd, 173
evir, 173
fdrtool, 271
foreign, 65, 66
gam, 241
ggm, 242
ggplot2, 344
gmodels, 203
gputools, 137
gRbase, 242
grDevices, 300, 353
grid, 360
Hmisc, 64, 66
ipred, 286
irr, 267
locfdr, 271
matlab, 65
MCMCpack, 173
mehods, 150
microbenchmark, 126
mnormt, 173
multcomp, 203
multicore, 138
multtest, 271
mvnorm, 173
nlme, 239
nortest, 244
orthopolynom, 291
PBImisc, 301
pixmap, 351
pls, 238
plyr, 153
pnmath0, 138
polynom, 290
profr, 129
proftools, 129
psy, 267
psych, 26
quantreg, 241
R.matlab, 65, 66
RColorBrewer, 353
Rcpp, 138
reshape2, 153
rgl, 362
RInside, 138
Rmpi, 137
sem, 242
slurm, 137
SmarterPoland, 157
snow, 137
stats, 26, 173, 203, 239, 271
survival, 282
taskPR, 137
program
ATLAS, 137
test
niezależności
χ^2, 263
Cochrana-Mantela-Haenszela, 265
Fishera, 263
normalności
Andersona Darlinga, 247
Cramera-von Misesa, 247
Lillieforsa (Kołmogorowa-Smirnowa), 247
Pearsona, 247
Shapiro-Francia, 247
Shapiro-Wilka, 247
równości średnich
Kruskala-Wallisa , 254
t Studenta, 254
Wilcoxona (Manna-Whitneya), 254
równości parametrów skali
Ansari-Bradley, 257
Mooda, 257
równości proporcji, 258
równości wariancji
Bartletta, 257
F, 257
Flingera-Killeen, 257
symetrii
McNemary, 265
współczynnika korelacji, 261
zmienna
.GlobalEnv, 144
.Machine, 124
.Platform, 124
.Random.seed, 167

R jest uniwersalnym narzędziem analizy danych. Jest całkowicie bezpłatny, ma olbrzymie możliwości, jest powszechnie używany w akademii i w biznesie. Znajomość tego języka przyda się każdemu, kto chce pracować z danymi.

Przewodnik po pakiecie R to podręcznik do nauki języka programowania oraz kluczowych bibliotek programu R. Skonstruowany na zasadzie *od zera do bohatera*. Rozpoczynamy od wprowadzenia dla osób zaczynających przygodę z R od podstaw. W kolejnych rozdziałach wprowadzamy coraz bardziej złożone wątki z obszaru Data Science, takie jak praca z bazami danych, tworzenie raportów, własnych pakietów, modelowanie statystyczne, oraz narzędzia do tworzenia treściwych i czytelnych wykresów. Łącznie omawiamy ponad 500 funkcji, setki wykresów i przykładów kodów w R.

©Tadeusz Późniak/POLITYKA

Dr hab. inż. Przemysław Biecek prof. PW używa na co dzień programu R w pracy naukowej. Program R jest idealnym narzędziem do przetwarzania terabajtów danych genetycznych w poszukiwaniu cech istotnych z punktu widzenia zastosowań w medycynie.

Od dziesięciu lat prowadzi zajęcia z programu R i analizy danych na wydziałach Matematyki i Informatyki Uniwersytetu Warszawskiego i Politechniki Warszawskiej. Współorganizuje comiesięczne otwarte Spotkania Entuzjastów R. Opracował materiały popularnego internetowego kursu *Pogromcy Danych*. Prowadzi też szkolenia biznesowe związane z programem R. W wolnym czasie pracuje nad opowiadaniami oraz grami dla dzieci i młodzieży z serii Beta Bit http://betabit.wiki.

ISBN 978-83-62780-44-0