Chapter 16
Specialized Machine Learning Topics

This chapter presents some technical details about data formats, streaming, optimization of computation, and distributed deployment of optimized learning algorithms. Chapter 22 provides additional optimization details. We show format conversion and working with XML, SQL, JSON, 15 CSV, SAS and other data objects. In addition, we illustrate SQL server queries, describe protocols for managing, classifying and predicting outcomes from data streams, and demonstrate strategies for optimization, improvement of computational performance, parallel (MPI) and graphics (GPU) computing.

The Internet of Things (IoT) leads to a paradigm shift of scientific inference – from static data interrogated in a batch or distributed environment to on-demand service-based Cloud computing. Here, we will demonstrate how to work with specialized data, data-streams, and SQL databases, as well as develop and assess on-the-fly data modeling, classification, prediction and forecasting methods. Important examples to keep in mind throughout this chapter include high-frequency data delivered real time in hospital ICU's (e.g., microsecond Electroencephalography signals, EEGs), dynamically changing stock market data (e.g., Dow Jones Industrial Average Index, DJI), and weather patterns.

We will present (1) format conversion of XML, SQL, JSON, CSV, SAS and other data objects, (2) visualization of bioinformatics and network data, (3) protocols for managing, classifying and predicting outcomes from data streams, (4) strategies for optimization, improvement of computational performance, parallel (MPI) and graphics (GPU) computing, and (5) processing of very large datasets.

16.1 Working with Specialized Data and Databases

Unlike the case studies we saw in the previous chapters, some real world data may not always be nicely formatted, e.g., as CSV files. We must collect, arrange, wrangle, and harmonize scattered information to generate computable data objects that can be further processed by various techniques. Data wrangling and preprocessing may take

I. D. Dinov, *Data Science and Predictive Analytics*,
https://doi.org/10.1007/978-3-319-72347-1_16

over 80% of the time researchers spend interrogating complex multi-source data archives. The following procedures will enhance your skills in collecting and handling heterogeneous real world data. Multiple examples of handling long-and-wide data, messy and tidy data, and data cleaning strategies can be found in this JSS Tidy Data article by Hadley Wickham.

16.1.1 Data Format Conversion

The R package `rio` imports and exports various types of file formats, e.g., tab-separated (`.tsv`), comma-separated (`.csv`), JSON (`.json`), Stata (`.dta`), SPSS (`.sav` and `.por`), Microsoft Excel (`.xls` and `.xlsx`), Weka (`.arff`), and SAS (`.sas7bdat` and `.xpt`).

`rio` provides three important functions `import()`, `export()` and `convert()`. They are intuitive, easy to understand, and efficient to execute. Take Stata (.dta) files as an example. First, we can download 02_Nof1_Data.dta from our datasets folder.

```
# install.packages("rio")
library(rio)
# Download the SAS .DTA file first locally
# Local data can be loaded by:
#nof1<-import("02_Nof1_Data.dta")
# the data can also be loaded from the server remotely as well:
nof1<-read.csv("https://umich.instructure.com/files/330385/download?download
_frd=1")
str(nof1)

## 'data.frame':    900 obs. of  10 variables:
##  $ ID       : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Day      : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ Tx       : int  1 1 0 0 1 1 0 0 1 1 ...
##  $ SelfEff  : int  33 33 33 33 33 33 33 33 33 33 ...
##  $ SelfEff25: int  8 8 8 8 8 8 8 8 8 8 ...
##  $ WPSS     : num  0.97 -0.17 0.81 -0.41 0.59 -1.16 0.3 -0.34 -0.74 -0.38
...
##  $ SocSuppt : num  5 3.87 4.84 3.62 4.62 2.87 4.33 3.69 3.29 3.66 ...
##  $ PMss     : num  4.03 4.03 4.03 4.03 4.03 4.03 4.03 4.03 4.03 4.03 ...
##  $ PMss3    : num  1.03 1.03 1.03 1.03 1.03 1.03 1.03 1.03 1.03 1.03 ...
##  $ PhyAct   : int  53 73 23 36 21 0 21 0 73 114 ...
```

The data are automatically stored as a data frame. Note that `rio` sets `stingAsFactors=FALSE` as default.

`rio` can help us export files into any other format we choose. To do this we have to use the `export()` function.

```
#Sys.getenv("R_ZIPCMD", "zip")   # Get the C Zip application
Sys.setenv(R_ZIPCMD="E:/Tools/ZIP/bin/zip.exe")
Sys.getenv("R_ZIPCMD", "zip")

## [1] "E:/Tools/ZIP/bin/zip.exe"

export(nof1, "02_Nof1.xlsx")
```

This line of code exports the *Nof1* data in `xlsx` format located in the `R` working directory. Mac users may have a problem exporting `*.xslx` files using `rio` because of a lack of a zip tool, but still can output other formats such as ".csv". An alternative strategy to save an `xlsx` file is to use package `xlsx` with default `row.name=TRUE`.

`rio` also provides a one step process to convert and save data into alternative formats. The following simple code allows us to convert and save the `02_Nof1_Data.dta` file we just downloaded into a CSV file.

```
# convert("02_Nof1_Data.dta", "02_Nof1_Data.csv")
convert("02_Nof1.xlsx",
"02_Nof1_Data.csv")
```

You can see a new CSV file popup in the current working directory. Similar transformations are available for other data formats and types.

16.1.2 Querying Data in SQL Databases

Let's use as an example the CDC Behavioral Risk Factor Surveillance System (BRFSS) Data, 2013-2015. This file for the combined landline and cell phone data set was exported from SAS V9.3 in the XPT transport format. This file contains 330 variables and can be imported into SPSS or STATA. Please note: some of the variable labels get truncated in the process of converting to the XPT format.

Be careful – this compressed (ZIP) file is over 315MB in size!

```
# install.packages("Hmisc")
library(Hmisc)

memory.size(max=T)

## [1] 115.81
pathToZip <- tempfile()
download.file("http://www.socr.umich.edu/data/DSPA/BRFSS_2013_2014_2015.zip"
, pathToZip)
# let's just pull two of the 3 years of data (2013 and 2015)
brfss_2013 <- sasxport.get(unzip(pathToZip)[1])

## Processing SAS dataset LLCP2013   ..

brfss_2015 <- sasxport.get(unzip(pathToZip)[3])

## Processing SAS dataset LLCP2015   ..

dim(brfss_2013); object.size(brfss_2013)

## [1] 491773    336

## 685581232 bytes
```

```
# summary(brfss_2013[1:1000, 1:10])  # subsample the data

# report the summaries for
summary(brfss_2013$has_plan)

## Length  Class   Mode
##      0   NULL   NULL

brfss_2013$x.race <- as.factor(brfss_2013$x.race)
summary(brfss_2013$x.race)

##      1      2      3      4      5      6      7      8      9   NA's
## 376451  39151   7683   9510   1546   2693   9130  37054   8530     25

# clean up
unlink(pathToZip)
```

Let's try to use logistic regression to find out if self-reported race/ethnicity predicts the binary outcome of having a health care plan.

```
brfss_2013$has_plan <- brfss_2013$hlthpln1 == 1

system.time(
  gml1 <- glm(has_plan ~ as.factor(x.race), data=brfss_2013,
              family=binomial)
)   # report execution time

##    user  system elapsed
##    2.20    0.23    2.46

summary(gml1)

##
## Call:
## glm(formula = has_plan ~ as.factor(x.race), family = binomial,
##     data = brfss_2013)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.1862   0.4385   0.4385   0.4385   0.8047
##
## Coefficients:
##                     Estimate Std. Error  z value Pr(>|z|)
## (Intercept)         2.293549   0.005649  406.044   <2e-16 ***
## as.factor(x.race)2 -0.721676   0.014536  -49.647   <2e-16 ***
## as.factor(x.race)3 -0.511776   0.032974  -15.520   <2e-16 ***
## as.factor(x.race)4 -0.329489   0.031726  -10.386   <2e-16 ***
## as.factor(x.race)5 -1.119329   0.060153  -18.608   <2e-16 ***
## as.factor(x.race)6 -0.544458   0.054535   -9.984   <2e-16 ***
## as.factor(x.race)7 -0.510452   0.030346  -16.821   <2e-16 ***
## as.factor(x.race)8 -1.332005   0.012915 -103.138   <2e-16 ***
## as.factor(x.race)9 -0.582204   0.030604  -19.024   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
```

```
##     Null deviance: 353371  on 491747  degrees of freedom
## Residual deviance: 342497  on 491739  degrees of freedom
##   (25 observations deleted due to missingness)
## AIC: 342515
##
## Number of Fisher Scoring iterations: 5
```

Next, we'll examine the odds (rather the log odds ratio, LOR) of having a health care plan (HCP) by race (R). The LORs are calculated for two array dimensions, separately for each *race* level (presence of *health care plan* (HCP) is binary, whereas *race* (R) has 9 levels, $R1$, $R2$, ..., $R9$). For example, the odds ratio of having a HCP for $R1 : R2$ is:

$$OR(R1 : R2) = \frac{\frac{P(HCP|R1)}{1-P(HCP|R1)}}{\frac{P(HCP|R2)}{1-P(HCP|R2)}}.$$

```
#load the vcd package to compute the LOR
library("vcd")

## Loading required package: grid

lor_HCP_by_R <- loddsratio(has_plan ~ as.factor(x.race), data = brfss_2013)
lor_HCP_by_R

## log odds ratios for has_plan and as.factor(x.race)
##
##         1:2         2:3         3:4         4:5         5:6         6:7
## -0.72167619  0.20990061  0.18228646 -0.78984000  0.57487142  0.03400611
##         7:8         8:9
## -0.82155382  0.74980101
```

Now, let's see an example of querying a database containing structured relational collection of data records. A *query* is a machine instruction (typically represented as text) sent by a user to a remote database requesting a specific database operation (e.g., search or summary). One database communication protocol relies on SQL (Structured query language). MySQL is an instance of a database management system that supports SQL communication and is utilized by many web applications, e.g., YouTube, Flickr, Wikipedia, biological databases like GO, ensembl, etc. Below is an example of an SQL query using the package `RMySQL`. An alternative way to interface an SQL database is using the package `RODBC`.

```
# install.packages("DBI"); install.packages("RMySQL")
# install.packages("RODBC"); library(RODBC)
library(DBI)
library(RMySQL)

ucscGenomeConn <- dbConnect(MySQL(),
                user='genome',
                dbname='hg38',
                host='genome-mysql.cse.ucsc.edu')
```

```
result <- dbGetQuery(ucscGenomeConn,"show databases;");

# List the DB tables
allTables <- dbListTables(ucscGenomeConn); length(allTables)

# Get dimensions of a table, read and report the head
dbListFields(ucscGenomeConn, "affyU133Plus2")
affyData <- dbReadTable(ucscGenomeConn, "affyU133Plus2"); head(affyData)

# Select a subset, fetch the data, and report the quantiles
subsetQuery <- dbSendQuery(ucscGenomeConn, "select * from affyU133Plus2
where misMatches between 1 and 3")
affySmall <- fetch(subsetQuery); quantile(affySmall$misMatches)

# Get repeat mask
bedFile <- 'repUCSC.bed'
df <- dbSendQuery(ucscGenomeConn,'select genoName,genoStart,genoEnd,
repName,swScore, strand,repClass, repFamily from rmsk') %>%
        dbFetch(n=-1) %>%
        mutate(genoName = str_replace(genoName,'chr','')) %>%
        tbl_df %>%
        write_tsv(bedFile,col_names=F)
message('written ', bedFile)

# Once done, close the connection
dbDisconnect(ucscGenomeConn)
```

To complete the above database SQL commands, it requires access to the remote UCSC SQL Genome server and user-specific credentials. You can see this functional example on the DSPA website. Below is another example that can be done by all readers, as it relies only on local services.

```
# install.packages("RSQLite")
library("RSQLite")

# generate an empty DB and store it in RAM
myConnection <- dbConnect(RSQLite::SQLite(), ":memory:")
myConnection

## <SQLiteConnection>
##   Path: :memory:
##   Extensions: TRUE

dbListTables(myConnection)

## character(0)

# Add tables to the local SQL DB
data(USArrests); dbWriteTable(myConnection, "USArrests", USArrests)

## [1] TRUE

dbWriteTable(myConnection, "brfss_2013", brfss_2013)

## [1] TRUE

dbWriteTable(myConnection, "brfss_2015", brfss_2015)

## [1] TRUE
```

```
# Check again the DB content
dbListFields(myConnection, "brfss_2013")

##   [1] "x.state"   "fmonth"    "idate"     "imonth"    "iday"
##   [6] "iyear"     "dispcode"  "seqno"     "x.psu"     "ctelenum"
##  [11] "pvtresd1"  "colghous"  "stateres"  "cellfon3"  "ladult"
##  [16] "numadult"  "nummen"    "numwomen"  "genhlth"   "physhlth"
##  [21] "menthlth"  "poorhlth"  "hlthpln1"  "persdoc2"  "medcost"
…
## [331] "rcsbrac1"  "rcsrace1"  "rchisla1"  "rcsbirth"  "typeinds"
## [336] "typework"  "has_plan"

dbListTables(myConnection);

## [1] "USArrests"  "brfss_2013" "brfss_2015"

# Retrieve the entire DB table (for the smaller USArrests table)
dbGetQuery(myConnection, "SELECT * FROM USArrests")

##    Murder Assault UrbanPop Rape
## 1    13.2     236       58 21.2
## 2    10.0     263       48 44.5
## 3     8.1     294       80 31.0
## 4     8.8     190       50 19.5
## 5     9.0     276       91 40.6
## 6     7.9     204       78 38.7
## 7     3.3     110       77 11.1
## 8     5.9     238       72 15.8
## 9    15.4     335       80 31.9
## 10   17.4     211       60 25.8
## 11    5.3      46       83 20.2
## 12    2.6     120       54 14.2
## 13   10.4     249       83 24.0
## 14    7.2     113       65 21.0
## 15    2.2      56       57 11.3
## 16    6.0     115       66 18.0
## 17    9.7     109       52 16.3
## 18   15.4     249       66 22.2
## 19    2.1      83       51  7.8
## 20   11.3     300       67 27.8
## 21    4.4     149       85 16.3
## 22   12.1     255       74 35.1
## 23    2.7      72       66 14.9
## 24   16.1     259       44 17.1
## 25    9.0     178       70 28.2
## 26    6.0     109       53 16.4
## 27    4.3     102       62 16.5
## 28   12.2     252       81 46.0
## 29    2.1      57       56  9.5
## 30    7.4     159       89 18.8
## 31   11.4     285       70 32.1
## 32   11.1     254       86 26.1
## 33   13.0     337       45 16.1
## 34    0.8      45       44  7.3
## 35    7.3     120       75 21.4
## 36    6.6     151       68 20.0
## 37    4.9     159       67 29.3
## 38    6.3     106       72 14.9
## 39    3.4     174        87  8.3
## 40   14.4     279        48 22.5
## 41    3.8      86        45 12.8
## 42   13.2     188        59 26.9
## 43   12.7     201        80 25.5
## 44    3.2     120        80 22.9
## 45    2.2      48        32 11.2
## 46    8.5     156        63 20.7
## 47    4.0     145        73 26.2
## 48    5.7      81        39  9.3
## 49    2.6      53        66 10.8
## 50    6.8     161        60 15.6

# Retrieve just the average of one feature
myQuery <- dbGetQuery(myConnection, "SELECT avg(Assault) FROM USArrests"); m
yQuery

##   avg(Assault)
## 1       170.76

myQuery <- dbGetQuery(myConnection, "SELECT avg(Assault) FROM USArrests GROU
P BY UrbanPop"); myQuery
##    avg(Assault)
## 1         48.00
## 2         81.00
## 3        152.00
## 4        211.50
## 5        271.00
## 6        190.00
## 7         83.00
## 8        109.00
## 9        109.00
## 10       120.00
## 11        57.00
## 12        56.00
## 13       236.00
## 14       188.00
## 15       186.00
## 16       102.00
## 17       156.00
## 18       113.00
## 19       122.25
## 20       229.50
## 21       151.00
## 22       231.50
## 23       172.00
## 24       145.00
## 25       255.00
## 26       120.00
## 27       110.00
## 28       204.00
## 29       237.50
## 30       252.00
## 31       147.50
## 32       149.00
## 33       254.00
## 34       174.00
## 35       159.00
## 36       276.00
```

```
# Or do it in batches (for the much larger brfss_2013 and brfss_2015 tables)
myQuery <- dbGetQuery(myConnection, "SELECT * FROM brfss_2013")

# extract data in chunks of 2 rows, note: dbGetQuery vs. dbSendQuery
# myQuery <- dbSendQuery(myConnection, "SELECT * FROM brfss_2013")
# fetch2 <- dbFetch(myQuery, n = 2); fetch2
# do we have other cases in the DB remaining?
# extract all remaining data
# fetchRemaining <- dbFetch(myQuery, n = -1);fetchRemaining
# We should have all data in DB now
# dbHasCompleted(myQuery)
# compute the average (poorhlth) grouping by Insurance (hlthpln1)
# Try some alternatives: numadult nummen numwomen genhlth physhlth menthlth
poorhlth hlthpln1
myQuery1_13 <- dbGetQuery(myConnection, "SELECT avg(poorhlth) FROM brfss_201
3 GROUP BY hlthpln1"); myQuery1_13

##   avg(poorhlth)
## 1      56.25466
## 2      53.99962
## 3      58.85072
## 4      66.26757

# Compare 2013 vs. 2015: Health grouping by Insurance
myQuery1_15 <- dbGetQuery(myConnection, "SELECT avg(poorhlth) FROM brfss_201
5 GROUP BY hlthpln1"); myQuery1_15

##   avg(poorhlth)
## 1      55.75539
## 2      55.49487
## 3      61.35445
## 4      67.62125

myQuery1_13 - myQuery1_15

##   avg(poorhlth)
## 1     0.4992652
## 2    -1.4952515
## 3    -2.5037326
## 4    -1.3536797

# reset the DB query
# dbClearResult(myQuery)

# clean up
dbDisconnect(myConnection)

## [1] TRUE
```

16.1.3 Real Random Number Generation

We are already familiar with (pseudo) random number generation (e.g., `rnorm (100, 10, 4)` or `runif(100, 10,20)`), which generate *algorithmically* computer values subject to specified distributions. There are also web services, e.g., random.org, that can provide *true random* numbers based on atmospheric noise, rather than using a pseudo random number generation protocol. Below is one example of generating a total of 300 numbers arranged in 3 columns, each of 100 rows of random integers (in decimal format) between 100 and 200.

```
#https://www.random.org/integers/?num=300&min=100&max=200&col=3&base=10&
format=plain&rnd=new
siteURL <- "http://random.org/integers/"  #   base URL
shortQuery<-"num=300&min=100&max=200&col=3&base=10&format=plain&rnd=new"
completeQuery <- paste(siteURL, shortQuery, sep="?")  # concat url and
submit query string
rngNumbers <- read.table(file=completeQuery)    # and read the data
rngNumbers

##      V1  V2  V3
## 1   144 179 131
## 2   127 160 150
## 3   142 169 109
…
## 98  178 103 134
## 99  173 178 156
## 100 117 118 110
```

16.1.4 Downloading the Complete Text of Web Pages

`RCurl` package provides an amazing tool for extracting and scraping information from websites. Let's install it and extract information from a SOCR website.

```
# install.packages("RCurl")
library(RCurl)

## Loading required package: bitops

web<-getURL("http://wiki.socr.umich.edu/index.php/SOCR_Data", followlocation
= TRUE)
str(web, nchar.max = 200)

##  chr "<!DOCTYPE html>\n<html lang=\"en\" dir=\"ltr\" class=\"client-nojs\
">\n<head>\n<meta charset=\"UTF-8\" />\n<title>SOCR Data - SOCR</title>\n<me
ta http-equiv=\"X-UA-Compatible\" content=\"IE=EDGE\" />"| __truncated__
```

The `web` object looks incomprehensible. This is because most websites are wrapped in XML/HTML hypertext or include JSON formatted metadata. `RCurl` deals with special HTML tags and website metadata.

To deal with the web pages only, `httr` package would be a better choice than `RCurl`. It returns a list that makes much more sense.

```
#install.packages("httr")
library(httr)
web<-GET("http://wiki.socr.umich.edu/index.php/SOCR_Data")
str(web[1:3])

## List of 3
##  $ url        : chr "http://wiki.socr.umich.edu/index.php/SOCR_Data"
##  $ status_code: int 200
##  $ headers    :List of 12
##   ..$ date                  : chr "Mon, 03 Jul 2017 19:09:56 GMT"
##   ..$ server                : chr "Apache/2.2.15 (Red Hat)"
##   ..$ x-powered-by          : chr "PHP/5.3.3"
##   ..$ x-content-type-options: chr "nosniff"
##   ..$ content-language      : chr "en"
##   ..$ vary                  : chr "Accept-Encoding,Cookie"
##   ..$ expires               : chr "Thu, 01 Jan 1970 00:00:00 GMT"
##   ..$ cache-control         : chr "private, must-revalidate, max-age=0"
##   ..$ last-modified         : chr "Sat, 22 Oct 2016 21:46:21 GMT"
##   ..$ connection            : chr "close"
##   ..$ transfer-encoding     : chr "chunked"
##   ..$ content-type          : chr "text/html; charset=UTF-8"
##   ..- attr(*, "class")= chr [1:2] "insensitive" "list"
```

16.1.5 Reading and Writing XML with the `XML` Package

A combination of the `RCurl` and the `XML` packages could help us extract only the plain text in our desired webpages. This would be very helpful to get information from heavy text-based websites.

```
web<-getURL("http://wiki.socr.umich.edu/index.php/SOCR_Data", followlocation
= TRUE)
#install.packages("XML")
library(XML)
web.parsed<-htmlParse(web, asText = T)
plain.text<-xpathSApply(web.parsed, "//p", xmlValue)
cat(paste(plain.text, collapse = "\n"))

## The links below contain a number of datasets that may be used for demonst
ration purposes in probability and statistics education. There are two types
of data - simulated (computer-generated using random sampling) and observed
(research, observationally or experimentally acquired).
##
## The SOCR resources provide a number of mechanisms to simulate data using
computer random-number generators. Here are some of the most commonly used S
OCR generators of simulated data:
##
## The following collections include a number of real observed datasets from
different disciplines, acquired using different techniques and applicable in
different situations.
##
## In addition to human interactions with the SOCR Data, we provide several
machine interfaces to consume and process these data.
##
## Translate this page:
##
## (default)
##
## Deutsch
…
## România
##
## Sverige
```

Here we extracted all plain text between the starting and ending *paragraph* HTML tags, `<p>` and `</p>`.

More information about extracting text from XML/HTML to text via XPath is available online.

16.1.6 Web-Page Data Scraping

The process that extracting data from complete web pages and storing it in structured data format is called `scraping`. However, before starting a data scrape from a website, we need to understand the underlying HTML structure for that specific website. Also, we have to check the terms of that website to make sure that scraping from this site is allowed.

The R package `rvest` is a very good place to start "harvesting" data from websites.

To start with, we use `read_html()` to store the SOCR data website into a `xmlnode` object.

```
library(rvest)

SOCR<-read_html("http://wiki.socr.umich.edu/index.php/SOCR_Data")
SOCR

## {xml_document}
## <html lang="en" dir="ltr" class="client-nojs">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=
...
## [2] <body class="mediawiki ltr sitedir-ltr ns-0 ns-subject page-SOCR_Dat
...
```

From the summary structure of `SOCR`, we can discover that there are two important hypertext section markups `<head>` and `<body>`. Also, notice that the SOCR data website uses `<title>` and `</title>` tags to separate title in the `<head>` section. Let's use `html_node()` to extract title information based on this knowledge.

```
SOCR %>% html_node("head title") %>% html_text()

## [1] "SOCR Data - SOCR"
```

Here we used `%>%` operator, or pipe, to connect two functions. The above line of code creates a chain of functions to operate on the `SOCR` object. The first function in the chain `html_node()` extracts the `title` from `head` section. Then, `html_text()` translates HTML formatted hypertext into English. More on R piping can be found in the magrittr package.

Another function, `rvest::html_nodes()` can be very helpful in scraping. Similar to `html_node()`, `html_nodes()` can help us extract multiple nodes in an `xmlnode` object. Assume that we want to obtain the meta elements (usually page description, keywords, author of the document, last modified, and other metadata) from the SOCR data website. We apply `html_nodes()` to the `SOCR` object to extract the hypertext data, e.g., lines starting with `<meta>` in the `<head>` section of the HTML page source. It is optional to use `html_attrs()`, which extracts attributes, text and tag names from HTML, obtain the main text attributes.

```
meta<-SOCR %>% html_nodes("head meta") %>% html_attrs()
meta

## [[1]]
##               http-equiv                      content
##           "Content-Type" "text/html; charset=UTF-8"
##
## [[2]]
## charset
## "UTF-8"
##
## [[3]]
##         http-equiv           content
## "X-UA-Compatible"         "IE=EDGE"
##
## [[4]]
##              name             content
##         "generator" "MediaWiki 1.23.1"
##
## [[5]]
##                          name                       content
## "ResourceLoaderDynamicStyles"                            ""
```

16.1.7 Parsing JSON from Web APIs

Application Programming Interfaces (APIs) allow web-accessible functions to communicate with each other. Today most API is stored in JSON (JavaScript Object Notation) format.

JSON represents a plain text format used for web applications, data structures or objects. Online JSON objects could be retrieved by packages like `RCurl` and `httr`. Let's see a JSON formatted dataset first. We can use 02_Nof1_Data.json in the class file as an example.

```
library(httr)
nof1<-GET("https://umich.instructure.com/files/1760327/download?download_frd
=1")
nof1

## Response [https://instructure-uploads.s3.amazonaws.com/account_1770000000
0000001/attachments/1760327/02_Nof1_Data.json?response-content-disposition=a
ttachment%3B%20filename%3D%2202_Nof1_Data.json%22%3B%20filename%2A%3DUTF-8%2
7%2702%255FNof1%255FData.json&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credent
ial=AKIAJFNFXH2V2O7RPCAA%2F20170703%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Da
te=20170703T190959Z&X-Amz-Expires=86400&X-Amz-SignedHeaders=host&X-Amz-Signa
ture=ceb3be3e71d9c370239bab558fcb0191bc829b98a7ba61ac86e27a2fc3c1e8ce]
```

```
##   Date: 2017-07-03 19:10
##   Status: 200
##   Content-Type: application/json
##   Size: 109 kB
## [{"ID":1,"Day":1,"Tx":1,"SelfEff":33,"SelfEff25":8,"WPSS"...
## {"ID":1,"Day":2,"Tx":1,"SelfEff":33,"SelfEff25":8,"WPSS"...
## {"ID":1,"Day":3,"Tx":0,"SelfEff":33,"SelfEff25":8,"WPSS"...
## {"ID":1,"Day":4,"Tx":0,"SelfEff":33,"SelfEff25":8,"WPSS"...
## {"ID":1,"Day":5,"Tx":1,"SelfEff":33,"SelfEff25":8,"WPSS"...
## {"ID":1,"Day":6,"Tx":1,"SelfEff":33,"SelfEff25":8,"WPSS"...
## {"ID":1,"Day":7,"Tx":0,"SelfEff":33,"SelfEff25":8,"WPSS"...
## {"ID":1,"Day":8,"Tx":0,"SelfEff":33,"SelfEff25":8,"WPSS"...
## {"ID":1,"Day":9,"Tx":1,"SelfEff":33,"SelfEff25":8,"WPSS"...
## {"ID":1,"Day":10,"Tx":1,"SelfEff":33,"SelfEff25":8,"WPSS"...
## ...
```

We can see that JSON objects are very simple. The data structure is organized using hierarchies marked by square brackets. Each piece of information is formatted as a {`key:value`} pair.

The package `jsonlite` is a very useful tool to import online JSON formatted datasets into data frame directly. Its syntax is very straight-forward.

```
#install.packages("jsonlite")
library(jsonlite)
nof1_lite<-
fromJSON("https://umich.instructure.com/files/1760327/download?download_frd=1")
class(nof1_lite)

## [1] "data.frame"
```

16.1.8 *Reading and Writing Microsoft Excel Spreadsheets Using XLSX*

We can transfer a *xlsx* dataset into CSV and use `read.csv()` to load this kind of dataset. However, R provides an alternative `read.xlsx()` function in package `xlsx` to simplify this process. Take our `02_Nof1_Data.xls` data in the class file as an example. We need to download the file first.

```
# install.packages("xlsx")
library(xlsx)

nof1<-read.xlsx("C:/Users/Folder/02_Nof1.xlsx", 1)
str(nof1)

## 'data.frame':    900 obs. of  10 variables:
##  $ ID        : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ Day       : num  1 2 3 4 5 6 7 8 9 10 ...
##  $ Tx        : num  1 1 0 0 1 1 0 0 1 1 ...
##  $ SelfEff  : num  33 33 33 33 33 33 33 33 33 33 ...
##  $ SelfEff25: num  8 8 8 8 8 8 8 8 8 8 ...
##  $ WPSS     : num  0.97 -0.17 0.81 -0.41 0.59 -1.16 0.3 -0.34 -0.74 -0.38
...
##  $ SocSuppt : num  5 3.87 4.84 3.62 4.62 2.87 4.33 3.69 3.29 3.66 ...
##  $ PMss     : num  4.03 4.03 4.03 4.03 4.03 4.03 4.03 4.03 4.03 4.03 ...
##  $ PMss3    : num  1.03 1.03 1.03 1.03 1.03 1.03 1.03 1.03 1.03 1.03 ...
##  $ PhyAct   : num  53 73 23 36 21 0 21 0 73 114 ...
```

The last argument, `1`, stands for the first excel sheet, as any excel file may include a large number of tables in it. Also, we can download the `xls` or `xlsx` file into our R working directory so that it is easier to find the file path.

Sometimes more complex protocols may be necessary to ingest data from XLSX documents. For instance, if the XLSX doc is large, includes many tables and is only accessible via HTTP protocol from a web-server. Below is an example of downloading the second table, `ABIDE_Aggregated_Data`, from the multi-table Autism/ABIDE XLSX dataset:

```
# install.packages("openxlsx"); library(openxlsx)
tmp = tempfile(fileext = ".xlsx")
download.file(url = "https://umich.instructure.com/files/3225493/download?do
wnload_frd=1",
destfile = tmp, mode="wb") df_Autism <- openxlsx::read.xlsx(xlsxFile = tmp,
      sheet = "ABIDE_Aggregated_Data", skipEmptyRows = TRUE)
dim(df_Autism)

## [1] 1098 2145
```

16.2 Working with Domain-Specific Data

Powerful machine-learning methods have already been applied in many applications. Some of these techniques are very specialized and some applications require unique approaches to address the corresponding challenges.

16.2.1 Working with Bioinformatics Data

Genetic data are stored in widely varying formats and usually have more feature variables than observations. They could have 1,000 columns and only 200 rows. One of the commonly used pre-processng steps for such datasets is *variable selection*. We will talk about this in Chap. 17.

The Bioconductor project created powerful R functionality (packages and tools) for analyzing genomic data, see Bioconductor for more detailed information.

16.2.2 Visualizing Network Data

Social network data and graph datasets describe the relations between nodes (vertices) using connections (links or edges) joining the node objects. Assume we have N objects, we can have $N * (N - 1)$ directed links establishing paired associations between the nodes. Let's use an example with $N=4$ to demonstrate a simple graph potentially modeling the node linkage Table 16.1.

If we change the $a \rightarrow b$ to an indicator variable (0 or 1) capturing whether we have an edge connecting a pair of nodes, then we get the graph *adjacency matrix*.

Edge lists provide an alternative way to represent network connections. Every line in the list contains a connection between two nodes (objects) (Table 16.2).

The edge list on Table 16.2 lists three network connections: object 1 is linked to object 2; object 1 is linked to object 3; and object 2 is linked to object 3. Note that edge lists can represent both *directed* as well as *undirected* networks or graphs.

We can imagine that if N is very large, e.g., social networks, the data representation and analysis may be resource intense (memory or computation). In R, we have multiple packages that can deal with social network data. One user-friendly example is provided using the `igraph` package. First, let's build a toy example and visualize it using this package (Fig. 16.1).

```r
#install.packages("igraph")
library(igraph)

g<-graph(c(1, 2, 1, 3, 2, 3, 3, 4), n=10)
plot(g)
```

Here `c(1, 2, 1, 3, 2, 3, 3, 4)` is an edge list with 4 rows and `n=10` indicates that we have 10 nodes (objects) in total. The small arrows in the graph show the directed network connections. We might notice that 5-10 nodes are scattered out in the graph. This is because they are not included in the edge list, so there are no network connections between them and the rest of the network.

Table 16.1 Schematic matrix representation of network connectivity

Objects	1	2	3	4
1	.....	$1 \rightarrow 2$	$1 \rightarrow 3$	$\mathbf{1 \rightarrow 4}$
2	$2 \rightarrow 1$	.....	$2 \rightarrow 3$	$\mathbf{2 \rightarrow 4}$
3	$3 \rightarrow 1$	$3 \rightarrow 2$	.....	$\mathbf{3 \rightarrow 4}$
4	$\mathbf{4 \rightarrow 1}$	$\mathbf{4 \rightarrow 2}$	$\mathbf{4 \rightarrow 3}$	.....

Table 16.2 List-based representation of network connectivity

Vertex	Vertex
1	2
1	3
2	3

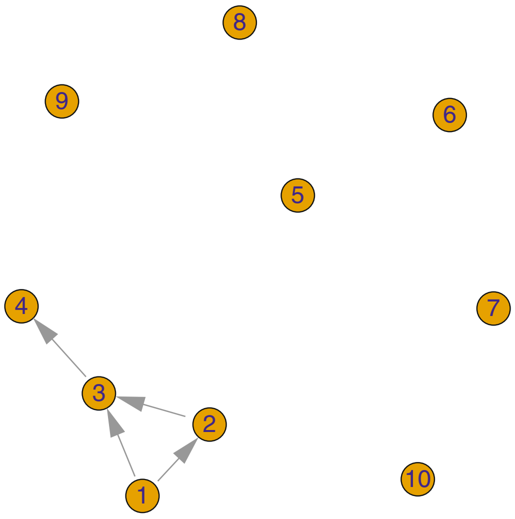

Fig. 16.1 A simple example of a social network as a graph object

Now let's examine the co-appearance network of Facebook circles. The data contains anonymized `circles` (friends lists) from Facebook collected from survey participants using a Facebook app. The dataset only includes edges (circles, 88,234) connecting pairs of nodes (users, 4,039) in the member social networks.

The values on the connections represent the number of links/edges within a circle. We have a huge edge-list made of scrambled Facebook user IDs. Let's load this dataset into R first. The data is stored in a text file. Unlike CSV files, text files in table format need to be imported using `read.table()`. We are using the `header=F` option to let R know that we don't have a header in the text file that contains only tab-separated node pairs (indicating the social connections, edges, between Facebook users).

```
soc.net.data<-read.table("https://umich.instructure.com/files/2854431/downlo
ad?download_frd=1", sep=" ", header=F)
head(soc.net.data)

##   V1 V2
## 1  0  1
## 2  0  2
## 3  0  3
## 4  0  4
## 5  0  5
## 6  0  6
```

Now the data is stored in a data frame. To make this dataset ready for `igraph` processing and visualization, we need to convert `soc.net.data` into a matrix object.

```
soc.net.data.mat <- as.matrix(soc.net.data, ncol=2)
```

By using ncol=2, we made a matrix with two columns. The data is now ready and we can apply graph.edgelist().

```
# remove the first 347 edges (to wipe out the degenerate "0" node)
graph_m<-graph.edgelist(soc.net.data.mat[-c(0:347), ], directed = F)
```

Before we display the social network graph we may want to examine our model first.

```
summary(graph_m)

## IGRAPH U--- 4038 87887 --
```

This is an extremely brief yet informative summary. The first line U--- 4038 87887 includes potentially four letters and two numbers. The first letter could be U or D indicating undirected or directed edges. A second letter N would mean that the objects set has a “name” attribute. A third letter is for weighted (W) graph. Since we didn’t add weight in our analysis the third letter is empty (“-“). A fourth character is an indicator for bipartite graphs, whose vertices can be divided into two disjoint sets where each vertex from one set connects to one vertex in the other set. The two numbers following the 4 letters represent the number of nodes and the number of edges, respectively. Now let’s render the graph (Fig. 16.2).

```
plot(graph_m)
```

This graph is very complicated. We can still see that some words are surrounded by more nodes than others. To obtain such information we can use the degree() function, which lists the number of edges for each node.

```
degree(graph_m)
```

Skimming the table we can find that the 107-th user has as many as 1,044 connections, which makes the user a *highly-connected hub*. Likely, this node may have higher social relevance.

Some edges might be more important than other edges because they serve as a bridge to link a cloud of nodes. To compare their importance, we can use the betweenness centrality measurement. *Betweenness centrality* measures centrality in a network. High centrality for a specific node indicates influence. betweenness () can help us to calculate this measurement.

```
betweenness(graph_m)
```

Again, the 107-th node has the highest betweenness centrality ($3.556221e + 06$).

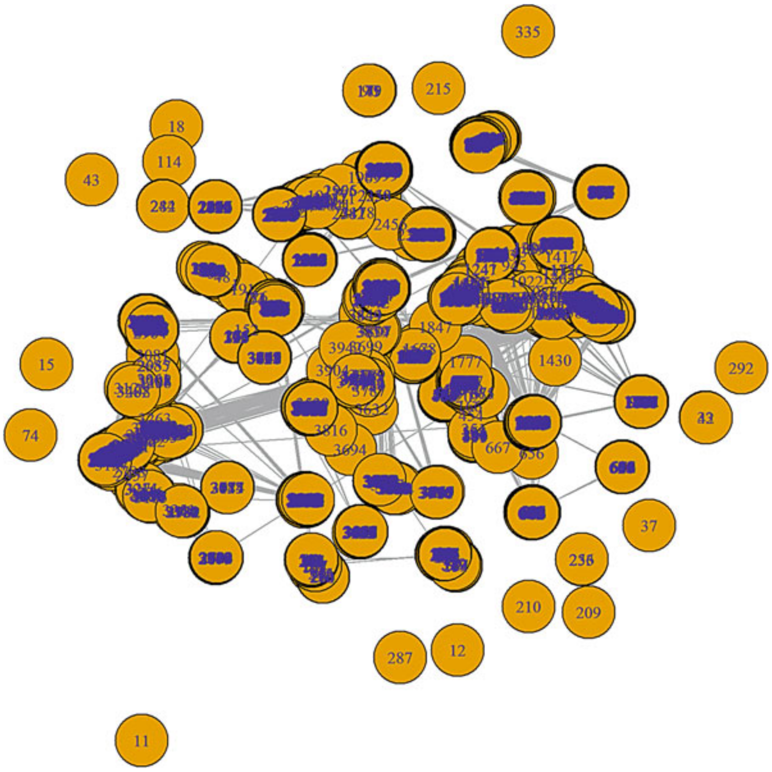

Fig. 16.2 Social network connectivity of Facebook users

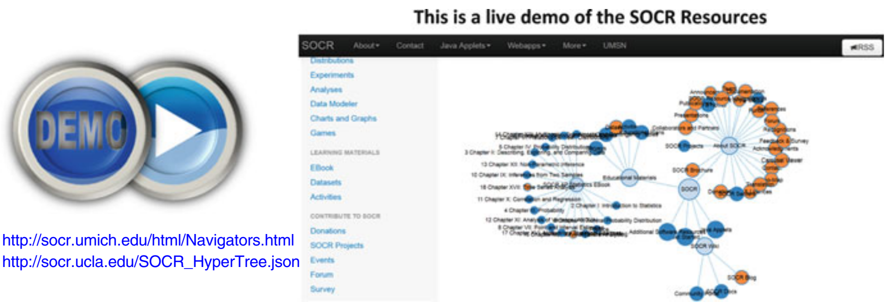
This is a live demo of the SOCR Resources
DEMO
http://socr.umich.edu/html/Navigators.html
http://socr.ucla.edu/SOCR_HyperTree.json

Fig. 16.3 Live demo: a dynamic graph representation of the SOCR resources

We can try another example using SOCR hierarchical data, which is also available for dynamic exploration as a tree graph. Let's read its JSON data source using the `jsonlite` package (Fig. 16.3).

```
tree.json<-fromJSON("http://socr.ucla.edu/SOCR_HyperTree.json",
simplifyDataFrame = FALSE)
```

This generates a `list` object representing the hierarchical structure of the network. Note that this is quite different from an edge list. There is one root node, its sub nodes are called *children nodes*, and the terminal nodes are call *leaf nodes*. Instead of presenting the relationship between nodes in pairs, this hierarchical structure captures the level for each node. To draw the social network graph, we need to convert it as a `Node` object. We can utilize the `as.Node()` function in the `data.tree` package to do so.

```
# install.packages("data.tree")
library(data.tree)
tree.graph<-as.Node(tree.json, mode = "explicit")
```

Here we use `mode="explicit"` option to allow "children" nodes to have their own "children" nodes. Now, the `tree.json` object has been separated into four different node structures – `"About SOCR"`, `"SOCR Resources"`, `"Get Started"`, and `"SOCR Wiki"`. Let's plot the first one using `igraph` package (Fig. 16.4).

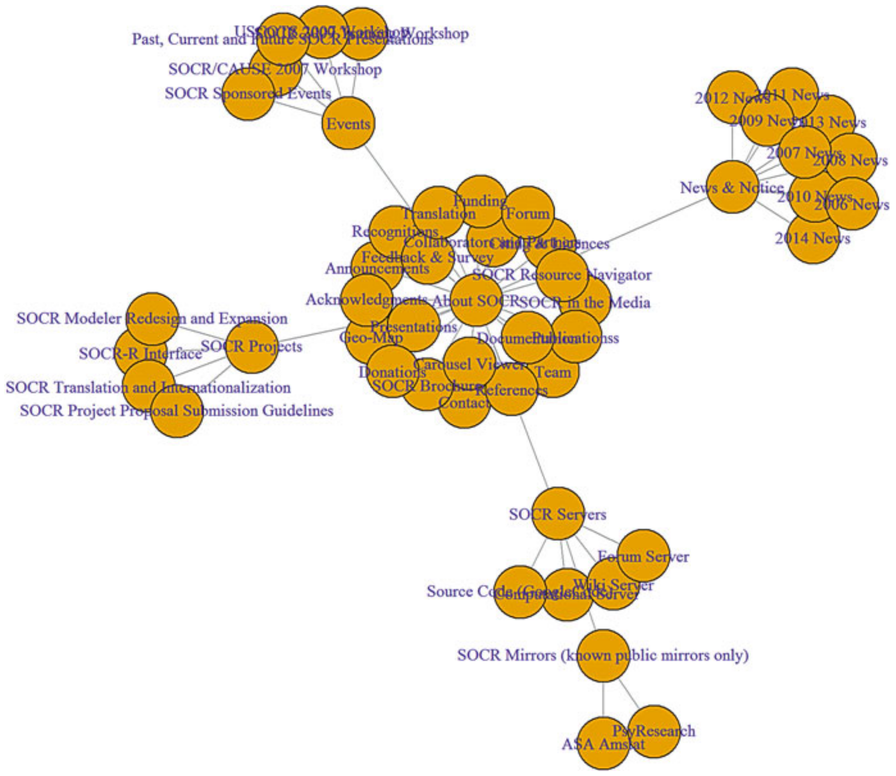

Fig. 16.4 The SOCR resourceome network plotted as a static R graph

```
plot(as.igraph(tree.graph$`About SOCR`), edge.arrow.size=5, edge.label.font=
0.05)
```

In this graph, `"About SOCR"`, which is located at the center, represents the root node of the tree graph.

16.3 Data Streaming

The proliferation of Cloud services and the emergence of modern technology in all aspects of human experiences leads to a tsunami of data much of which is streamed real-time. The interrogation of such voluminous data is an increasingly important area of research. *Data streams* are ordered, often unbounded sequences of data points created continuously by a data generator. All of the data mining, interrogation and forecasting methods we discuss here are also applicable to data streams.

16.3.1 Definition

Mathematically, a *data stream* in an ordered sequence of data points

$$Y = \{y_1, y_2, y_3, \cdots, y_t, \cdots\},$$

where the (time) index, t, reflects the order of the observation/record, which may be single numbers, simple vectors in multidimensional space, or objects, e.g., structured Ann Arbor Weather (JSON) and its corresponding structured form. Some streaming data is *streamed* because it's too large to be downloaded shotgun style and some is *streamed* because it's continually generated and serviced. This presents the potential problem of dealing with data streams that may be unlimited.

Notes:

- *Data sources*: Real or synthetic stream data can be used. Random simulation streams may be created by `rstream`. Real stream data may be piped from financial data providers, the WHO, World Bank, NCAR and other sources.
- *Inference Techniques*: Many of the data interrogation techniques we have seen can be employed for dynamic stream data, e.g., `factas`, for PCA, `rEMM` and `birch` for clustering, etc. Clustering and classification methods capable of processing data streams have been developed, e.g., *Very Fast Decision Trees* (VFDT), *time window-based Online Information Network* (OLIN), *On-demand Classification*, and the *APRIORI* streaming algorithm.
- *Cloud distributed computing*: Hadoop2/HadoopStreaming, SPARK, Storm3/RStorm provide an environments to expand batch/script-based R tools to the Cloud.

16.3.2 The `stream` Package

The `R` `stream` package provides data stream mining algorithms using `fpc`, `clue`, `cluster`, `clusterGeneration`, `MASS`, and `proxy` packages. In addition, the package `streamMOA` provides an `rJava` interface to the Java-based data stream clustering algorithms available in the *Massive Online Analysis* (MOA) framework for stream classification, regression and clustering.

If you need a deeper exposure to data streaming in R, we recommend you go over the stream vignettes.

16.3.3 Synthetic Example: Random Gaussian Stream

This example shows the creation and loading of a *mixture of 5 random 2D Gaussians*, centers at (*x_coords*, *y_coords*) with paired correlations *rho_corr*, representing a simulated data stream.

Generate the stream:

```
# install.packages("stream")
library("stream")

x_coords <- c(0.2,0.3, 0.5, 0.8, 0.9)
y_coords <- c(0.8,0.3, 0.7, 0.1, 0.5)
p_weight <- c(0.1, 0.9, 0.5, 0.4, 0.3) # A vector of probabilities that dete
rmines the likelihood of generated a data point from a particular
cluster set.seed(12345)
stream_5G <- DSD_Gaussians(k = 5, d = 2, mu=cbind(x_coords, y_coords),
p=p_weight)
```

k-Means Clustering

We will now try k-means and density-based data stream clustering algorithm, D-Stream, where micro-clusters are formed by grid cells of size *gridsize* with density of a grid cell (Cm) is least 1.2 times the average cell density. The model is updated with the next 500 data points from the stream.

```
dstream <- DSC_DStream(gridsize = .1, Cm = 1.2)
update(dstream,  stream_5G,  n  =  500)
```

First, let's run the k-means clustering with $k = 5$ clusters and plot the resulting micro- and macro-clusters (Fig. 16.5).

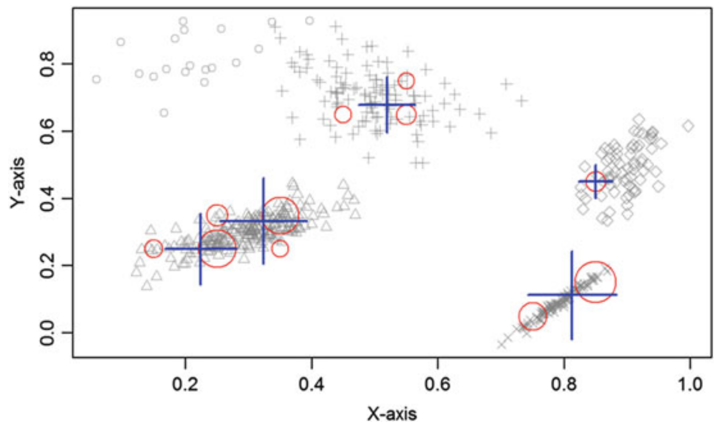

Fig. 16.5 Micro and macro clusters of a 5-means clustering of the first 500 points of the streamed simulated 2D Gaussian kernels

```
kmc <- DSC_Kmeans(k = 5)
recluster(kmc,   dstream)
plot(kmc, stream_5G, type = "both", xlab="X-axis", ylab="Y-axis")
```

In this clustering plot, *micro-clusters are shown as circles* and *macro-clusters are shown as crosses* and their sizes represent the corresponding cluster weight estimates.

Next try the density-based data stream clustering algorithm D-Stream. Prior to updating the model with the next 1,000 data points from the stream, we specify the grid cells as micro-clusters, grid cell size (gridsize=0.1), and a micro-cluster (Cm=1.2) that specifies the density of a grid cell as a multiple of the average cell density.

```
dstream <- DSC_DStream(gridsize = 0.1, Cm = 1.2)
update(dstream,  stream_5G,  n=1000)
```

We can re-cluster the data using k-means with 5 clusters and plot the resulting *micro*- and *macro*-clusters (Fig. 16.6).

```
km_G5 <- DSC_Kmeans(k = 5)
recluster(km_G5,   dstream)
plot(km_G5, stream_5G, type = "both")
```

Note the subtle changes in the clustering results between `kmc` and `km_G5`.

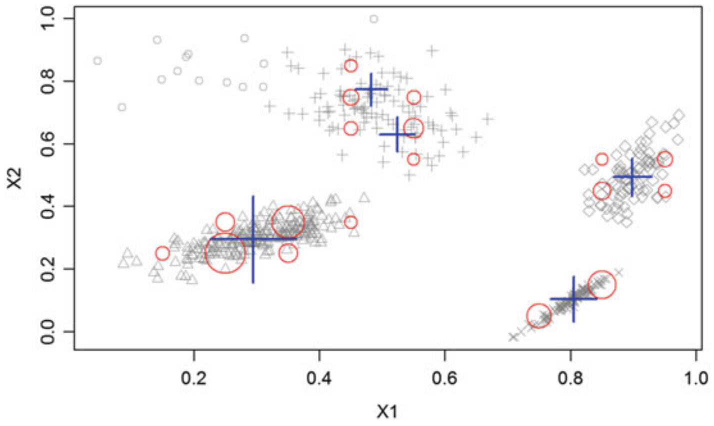

Fig. 16.6 Micro- and macro- clusters of a 5-means clustering of the next 1,000 points of the streamed simulated 2D Gaussian kernels

16.3.4 Sources of Data Streams

Static Structure Streams

- *DSD_BarsAndGaussians* generates two uniformly filled rectangular and two Gaussian clusters with different density.
- *DSD_Gaussians* generates randomly placed static clusters with random multi-variate Gaussian distributions.
- *DSD_mlbenchData* provides streaming access to machine learning benchmark data sets found in the `mlbench` package.
- *DSD_mlbenchGenerator* interfaces the generators for artificial data sets defined in the mlbench package.
- *DSD_Target* generates a ball in circle data set.
- *DSD_UniformNoise* generates uniform noise in a d-dimensional (hyper) cube.

Concept Drift Streams

- *DSD_Benchmark* provides a collection of simple benchmark problems including splitting and joining clusters, and changes in density or size, which can be used as a comprehensive benchmark set for algorithm comparison.
- *DSD_MG* is a generator to specify complex data streams with concept drift. The shape as well as the behavior of each cluster over time can be specified using keyframes.
- *DSD_RandomRBFGeneratorEvents* generates streams using radial base functions with noise. Clusters move, merge and split.

Real Data Streams

- *DSD_Memory* provides a streaming interface to static, matrix-like data (e.g., a data frame, a matrix) in memory which represents a fixed portion of a data stream. Matrix-like objects also include large objects potentially stored on disk like `ff::ffdf`.
- *DSD_ReadCSV* reads data line by line in text format from a file or an open connection and makes it available in a streaming fashion. This way data that is larger than the available main memory can be processed.
- *DSD_ReadDB* provides an interface to an open result set from a SQL query to a relational database.

16.3.5 Printing, Plotting and Saving Streams

For `DSD` objects, some basic stream functions include `print()`, `plot()`, and `write_stream()`. These can save part of a data stream to disk. `DSD_Memory` and `DSD_ReadCSV` objects also include member functions like `reset_stream()` to reset the position in the stream to its beginning.

To request a new batch of data points from the stream we use `get_points()`. This chooses a *random cluster* (based on the probability weights in `p_weight`) and a point is drawn from the multivariate Gaussian distribution (*mean* = *mu*, *covariance matrix* = Σ) of that cluster. Below, we pull $n = 10$ new data points from the stream (Fig. 16.7).

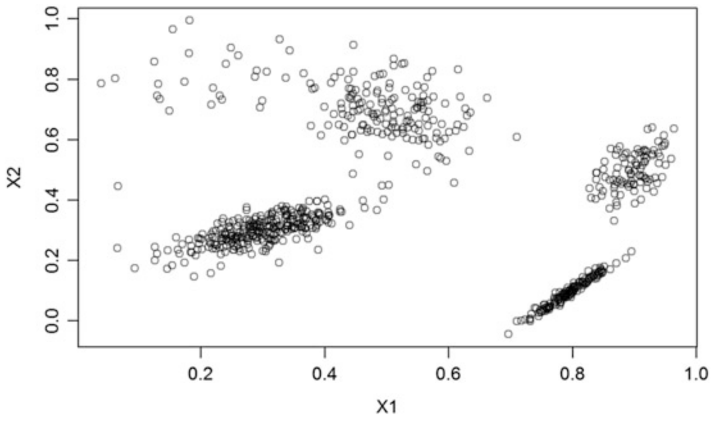

Fig. 16.7 Scatterplot of the next batch of 700 random Gaussian points in 2D

```
new_p <- get_points(stream_5G, n = 10)
new_p

##           X1        X2
## 1  0.4017803 0.2999017
## 2  0.4606262 0.5797737
## 3  0.4611642 0.6617809
## 4  0.3369141 0.2840991
## 5  0.8928082 0.5687830
## 6  0.8706420 0.4282589
## 7  0.2539396 0.2783683
## 8  0.5594320 0.7019670
## 9  0.5030676 0.7560124
## 10 0.7930719 0.0937701

new_p <- get_points(stream_5G, n = 100, class = TRUE)
head(new_p, n = 20)

##           X1         X2 class
## 1  0.7915730 0.09533001     4
## 2  0.4305147 0.36953997     2
## 3  0.4914093 0.82120395     3
## 4  0.7837102 0.06771246     4
## 5  0.9233074 0.48164544     5
## 6  0.8606862 0.49399269     5
## 7  0.3191884 0.27607324     2
## 8  0.2528981 0.27596700     2
## 9  0.6627604 0.68988585     3
## 10 0.7902887 0.09402659     4
## 11 0.7926677 0.09030248     4
## 12 0.9393515 0.50259344     5
## 13 0.9333770 0.62817482     5
## 14 0.7906710 0.10125432     4
## 15 0.1798662 0.24967850     2
## 16 0.7985790 0.08324688     4
## 17 0.5247573 0.57527380     3
## 18 0.2358468 0.23087585     2
## 19 0.8818853 0.49668824     5
## 20 0.4255094 0.81789418     3

plot(stream_5G, n = 700, method = "pc")
```

Note that if you add *noise* to your stream, e.g., `stream_Noise <- DSD_Gaussians(k = 5, d = 4, noise = .1, p = c(0.1, 0.5, 0.3, 0.9, 0.1))`, then the noise points that are not classified as part of any cluster will have an `NA` class label.

16.3.6 Stream Animation

Clusters can be animated over time by `animate_data()`. Use `reset_stream()` to start the animation at the beginning of the stream and note that this method is **not implemented** for streams of class `DSD_Gaussians`, `DSD_R`, `DSD_data.frame`, and `DSD`. We'll create a new `DSD_Benchmark` data stream (Fig. 16.8).

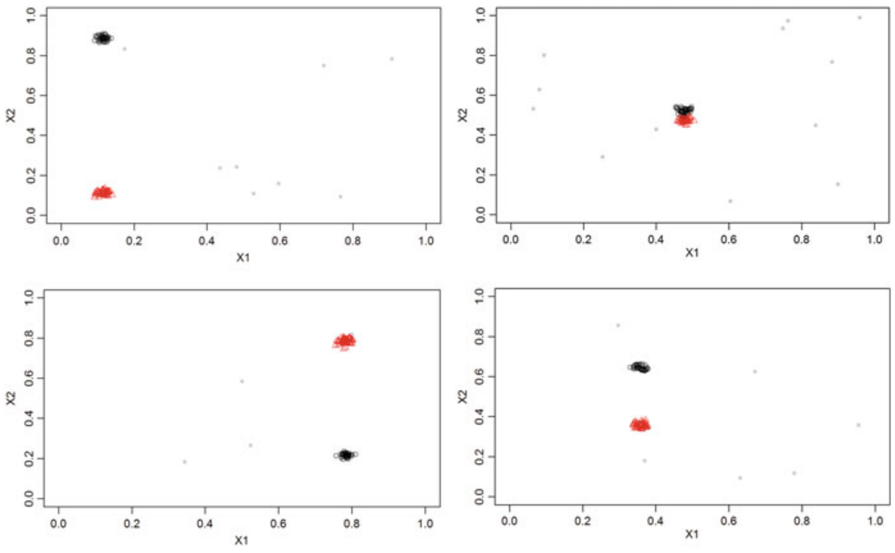

Fig. 16.8 Discrete snapshots of the animated stream clustering process

```
set.seed(12345)
stream_Bench <- DSD_Benchmark(1)
stream_Bench

## Benchmark 1: Two clusters moving diagonally from left to right, meeting
in
## the center (5% noise).
## Class: DSD_MG, DSD_R, DSD_data.frame, DSD
## With 2 clusters in 2 dimensions. Time is 1
library("animation")
reset_stream(stream_Bench)
animate_data(stream_Bench,n=10000,horizon=100,xlim=c(0,1), ylim=c(0,1))
```

This benchmark generator creates two 2D clusters moving in 2D. One moves from *top-left* to *bottom-right*, the other from *bottom-left* to *top-right*. Then they meet at the center of the domain, the 2 clusters overlap and then split again.

Concept drift in the stream can be depicted by requesting (10) times 300 data points from the stream and animating the plot. Fast-forwarding the stream can be accomplished by requesting, but ignoring, (2000) points in between the (10) plots. The output of the animation below is suppressed to save space.

```
for(i in 1:10) {
   plot(stream_Bench, 300, xlim = c(0, 1), ylim = c(0, 1))
   tmp <- get_points(stream_Bench, n = 2000)
}
```

```
reset_stream(stream_Bench)
animate_data(stream_Bench,n=8000,horizon=120,   xlim=c(0,1), ylim=c(0,1))
# Animations can be saved as HTML or GIF
#saveHTML(ani.replay(), htmlfile = "stream_Bench_Animation.html")
#saveGIF(ani.replay())
```

Streams can also be saved locally by `write_stream(stream_Bench, "dataStreamSaved.csv", n = 100, sep=",")` and loaded back in `R` by `DSD_ReadCSV()`.

16.3.7 Case-Study: SOCR Knee Pain Data

These data represent the *X* and *Y* spatial knee-pain locations for over 8,000 patients, along with *labels* about the knee *F*ront, *B*ack, *L*eft and *R*ight. Let's try to read the SOCR Knee Pain Dataset as a stream.

```
library("XML"); library("xml2"); library("rvest")

wiki_url <- read_html("http://wiki.socr.umich.edu/index.php/SOCR_Data_KneePa
inData_041409")
html_nodes(wiki_url, "#content")

## {xml_nodeset (1)}
## [1] <div id="content" class="mw-body-primary" role="main">\n\t<a id="top
...

kneeRawData <- html_table(html_nodes(wiki_url, "table")[[2]])
normalize<-function(x){
  return((x-min(x))/(max(x)-min(x)))
}
kneeRawData_df <- as.data.frame(cbind(normalize(kneeRawData$x),
normalize(kneeRawData$Y), as.factor(kneeRawData$View)))
colnames(kneeRawData_df) <- c("X", "Y", "Label")
# randomize the rows of the DF as RF, RB, LF and LB labels of classes are
sequential
set.seed(1234)
kneeRawData_df <- kneeRawData_df[sample(nrow(kneeRawData_df)), ]
summary(kneeRawData_df)

##        X                Y               Label
##  Min.   :0.0000   Min.   :0.0000   Min.   :1.000
##  1st Qu.:0.1331   1st Qu.:0.4566   1st Qu.:2.000
##  Median :0.2995   Median :0.5087   Median :3.000
##  Mean   :0.3382   Mean   :0.5091   Mean   :2.801
##  3rd Qu.:0.3645   3rd Qu.:0.5549   3rd Qu.:4.000
##  Max.   :1.0000   Max.   :1.0000   Max.   :4.000

# View(kneeRawData_df)
```

We can use the `DSD::DSD_Memory` class to get a stream interface for matrix or data frame objects, like the Knee pain location dataset. The number of true clusters $k = 4$ in this dataset.

```
# use data.frame to create a stream (3rd column contains label assignment)
kneeDF <- data.frame(x=kneeRawData_df[,1], y=kneeRawData_df[,2],
  class=as.factor(kneeRawData_df[,3]))
head(kneeDF)
##           x         y class
## 1 0.1188590 0.5057803     4
## 2 0.3248811 0.6040462     2
## 3 0.3153724 0.4971098     2
## 4 0.3248811 0.4161850     2
## 5 0.6941363 0.5289017     1
## 6 0.3217116 0.4595376     2

streamKnee <- DSD_Memory(kneeDF[,c("x", "y")], class=kneeDF[,"class"],
loop=T)
streamKnee

## Memory Stream Interface
## Class: DSD_Memory, DSD_R, DSD_data.frame, DSD
## With NA clusters in 2 dimensions
## Contains 8666 data points - currently at position 1 - loop is TRUE

# Each time we get a point from *streamKnee*, the stream pointer moves
to the next position (row) in the data.
get_points(streamKnee, n=10)

##             x         y
## 1  0.11885895 0.5057803
## 2  0.32488114 0.6040462
## 3  0.31537242 0.4971098
## 4  0.32488114 0.4161850
## 5  0.69413629 0.5289017
## 6  0.32171157 0.4595376
## 7  0.06497623 0.4913295
## 8  0.12519810 0.4682081
## 9  0.32329635 0.4942197
## 10 0.30744849 0.5086705

streamKnee

## Memory Stream Interface
## Class: DSD_Memory, DSD_R, DSD_data.frame, DSD
## With NA clusters in 2 dimensions
## Contains 8666 data points - currently at position 11 - loop is TRUE

# Stream pointer is in position 11 now

# We can redirect the current position of the stream pointer by:
reset_stream(streamKnee,  pos  =  200)
get_points(streamKnee, n=10)

##             x         y
## 200 0.9413629 0.5606936
## 201 0.3217116 0.5664740
## 202 0.3122029 0.6416185
## 203 0.1553090 0.6040462
## 204 0.3645008 0.5346821
## 205 0.3122029 0.5000000
## 206 0.3549921 0.5404624
## 207 0.1473851 0.5260116
## 208 0.1870048 0.6329480
## 209 0.1220285 0.4132948
```

```
streamKnee

## Memory Stream Interface
## Class: DSD_Memory, DSD_R, DSD_data.frame, DSD
## With NA clusters in 2 dimensions
## Contains 8666 data points - currently at position 210 - loop is TRUE
```

16.3.8 Data Stream Clustering and Classification (DSC)

Let's demonstrate clustering using `DSC_DStream`, which assigns points to cells in a grid. First, initialize the clustering, as an empty cluster and then use the `update()` function to implicitly alter the mutable `DSC` object (Fig. 16.9).

```
dsc_streamKnee <- DSC_DStream(gridsize = 0.1, Cm = 0.4, attraction=T)
dsc_streamKnee

## DStream
## Class: DSC_DStream, DSC_Micro, DSC_R, DSC
## Number of micro-clusters: 0
## Number of macro-clusters: 0

# stream::update
reset_stream(streamKnee,  pos  =  1)
update(dsc_streamKnee, streamKnee,  n  =  500)
dsc_streamKnee

## DStream
## Class: DSC_DStream, DSC_Micro, DSC_R, DSC
## Number of micro-clusters: 16
## Number of macro-clusters: 11
```

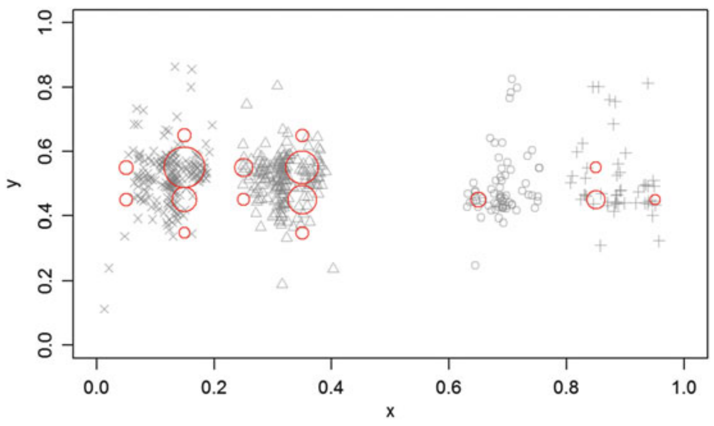

Fig. 16.9 Data stream clustering and classification of the SOCR knee-pain dataset (n=500)

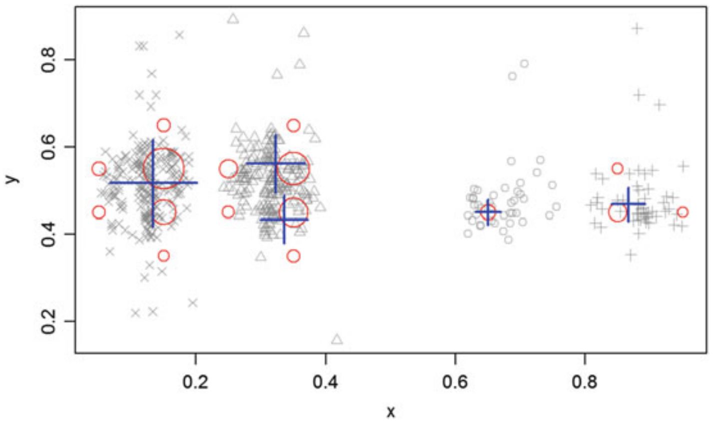

Fig. 16.10 5-Means stream clustering of the SOCR knee pain data

```
head(get_centers(dsc_streamKnee))
```

```
##      [,1] [,2]
## [1,] 0.05 0.45
## [2,] 0.05 0.55
## [3,] 0.15 0.35
## [4,] 0.15 0.45
## [5,] 0.15 0.55
## [6,] 0.15 0.65
```

```
plot(dsc_streamKnee,  streamKnee, xlim=c(0,1), ylim=c(0,1))
```

```
# plot(dsc_streamKnee,  streamKnee, grid = TRUE)
# Micro-clusters are plotted in red on top of gray stream data points
# The size of the micro-clusters indicates their weight - it's proportional
to the number of data points represented by each micro-cluster.
# Micro-clusters are shown as dense grid cells (density is coded with gray
values).
```

The **purity** metric represents an external evaluation criterion of cluster quality, which is the proportion of the total number of points that were correctly classified:

$$0 \leq Purity = \frac{1}{N}\sum_{i=1}^{k} max_j\left|c_i \cap t_j\right| \leq 1,$$

where N=number of observed data points, $k=$ number of clusters, c_i is the i^{th} cluster, and t_j is the classification that has the maximum number of points with c_i class labels. High purity suggests that we correctly label points (Fig. 16.10).

Next, we can use K-means clustering.

```
kMeans_Knee <- DSC_Kmeans(k=5) # use 4-5 clusters matching the 4 knee labels
recluster(kMeans_Knee, dsc_streamKnee)
plot(kMeans_Knee, streamKnee, type = "both")
```

Again, the graphical output of the animation sequence of frames is suppressed, however, the readers are encouraged to run the command line and inspect the graphical outcome (Figs. 16.11 and 16.12).

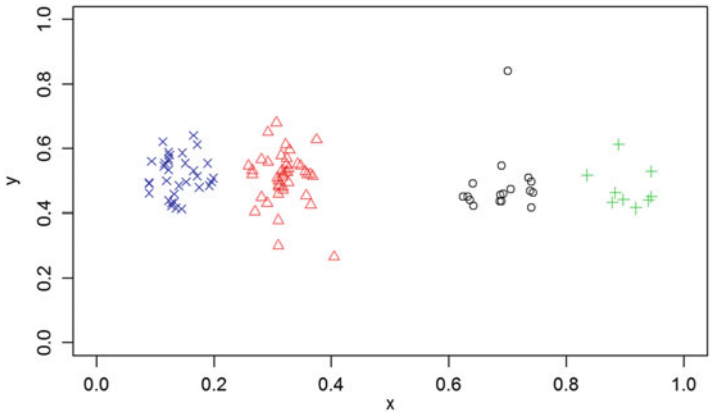

Fig. 16.11 Animated continuous 5-means stream clustering of the knee pain data

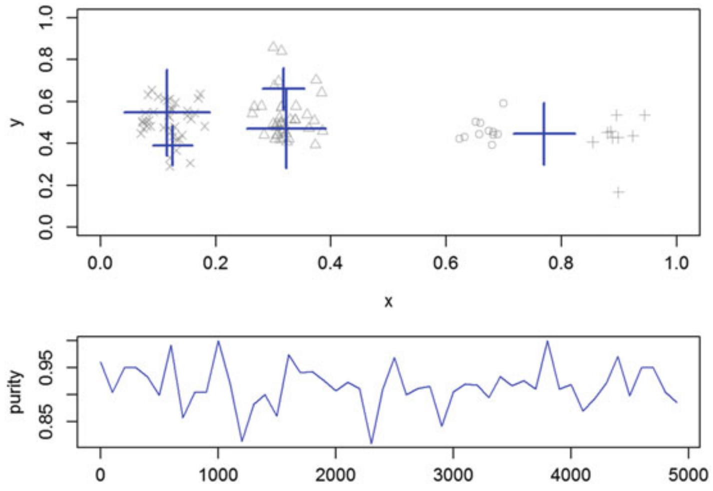

Fig. 16.12 Continuous stream clustering and purity index across iterations

```
animate_data(streamKnee, n=1000, horizon=100,xlim=c(0,1), ylim = c(0,1))
```

```
# purity <- animate_cluster(kMeans_Knee, streamKnee, n=2500, type="both",
xlim=c(0,1), ylim=c(-,1), evaluationMeasure="purity", horizon=10)

animate_cluster(kMeans_Knee, streamKnee,  horizon = 100,  n = 5000,
measure = "purity", plot.args = list(xlim = c(0, 1), ylim = c(0, 1)))
```

```
##    points    purity
## 1       1 0.9600000
## 2     101 0.9043478
## 3     201 0.9500000
…
## 49   4801 0.9047619
## 50   4901 0.8850000
```

16.3.9 Evaluation of Data Stream Clustering

Figure 16.13 shows the average clustering purty as we evaluate the stream clustering across the streaming points.

```
# Synthetic Gaussian example
# stream <- DSD_Gaussians(k = 3, d = 2, noise = .05)
# dstream <- DSC_DStream(gridsize = .1)
# update(dstream,  stream,  n  =  2000)
# evaluate(dstream,  stream,  n  =  100)

evaluate(dsc_streamKnee, streamKnee, measure = c("crand", "SSQ",
"silhouette"), n = 100, type = c("auto","micro","macro"), assign="micro",
assignmentMethod = c("auto", "model", "nn"), noise = c("class","exclude"))

## Evaluation results for micro-clusters.
## Points were assigned to micro-clusters.
##      cRand        SSQ silhouette
##  0.3473634  0.3382900  0.1373143

clusterEval  <- evaluate_cluster(dsc_streamKnee, streamKnee, measure =
c("numMicroClusters", "purity"), n = 5000, horizon = 100)
head(clusterEval)
##   points numMicroClusters    purity
## 1      0               16 0.9555556
## 2    100               17 0.9733333
## 3    200               18 0.9671053
## 4    300               21 0.9687500
## 5    400               21 0.9880952
## 6    500               22 0.9750000

plot(clusterEval[ , "points"], clusterEval[ , "purity"], type = "l",
ylab  =  "Avg Purity",  xlab  =  "Points")
```

```
animate_cluster(dsc_streamKnee, streamKnee,  horizon  =  100,  n  =  5000,
measure = "purity", plot.args = list(xlim = c(0, 1), ylim = c(0, 1)))
```

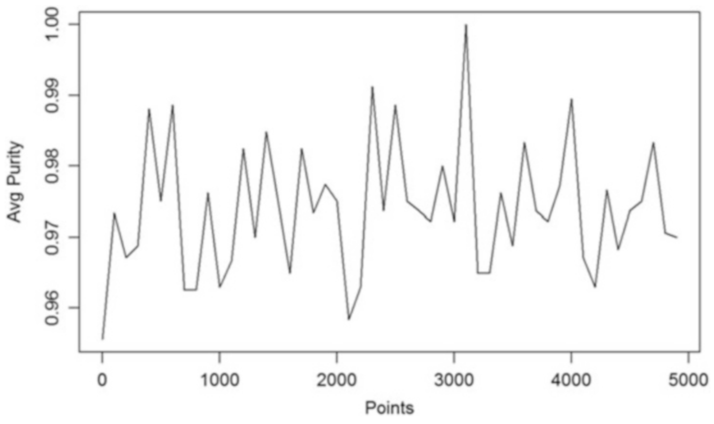

Fig. 16.13 Average clustering purity

```
##    points    purity
## 1       1 0.9714286
## 2     101 0.9833333
## 3     201 0.9722222
…

## 49   4801 0.9772727
## 50   4901 0.9777778
```

The `dsc_streamKnee` represents the result of the stream clustering, where *n* is the number of data points from the `streamKnee` stream. The evaluation `measure` can be specified as a vector of character strings. Points are assigned to clusters in `dsc_streamKnee` using `get_assignment()` and can be used to assess the quality of the classification. By default, points are assigned to *micro-clusters*, or can be assigned to *macro-cluster* centers by `assign = "macro"`. Also, new points can be assigned to clusters by the rule used in the clustering algorithm by `assignmentMethod = "model"` or using nearest-neighbor assignment (`nn`), Fig. 16.14.

16.4 Optimization and Improving the Computational Performance

Here and in previous chapters, e.g., Chap. 15, we notice that R may sometimes be slow and memory-inefficient. These problems may be severe, especially for datasets with millions of records or when using complex functions. There are packages for processing large datasets and memory optimization – `bigmemory`, `biganalytics`, `bigtabulate`, etc.

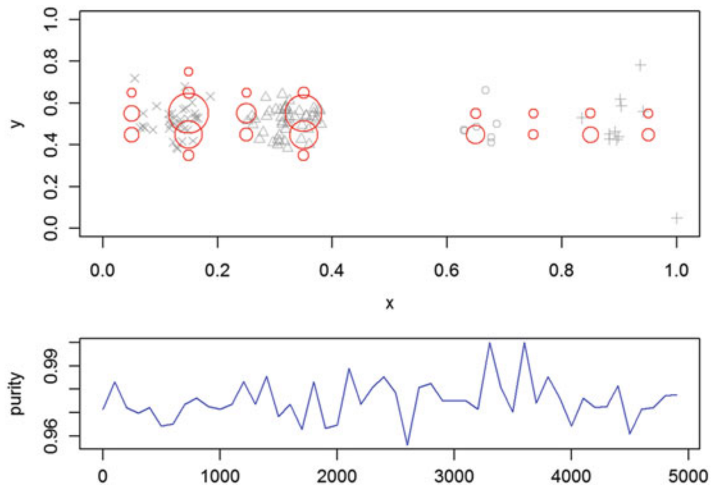

Fig. 16.14 Continuous k-means stream clustering with classificaiton purity

16.4.1 Generalizing Tabular Data Structures with `dplyr`

We have also seen long execution times when running processes that ingest, store or manipulate huge `data.frame` objects. The `dplyr` package, created by Hadley Wickham and Romain Francoi, provides a faster route to manage such large datasets in R. It creates an object called `tbl`, similar to `data.frame`, which has an in-memory column-like structure. R reads these objects a lot faster than data frames.

To make a `tbl` object we can either convert an existing data frame to `tbl` or connect to an external database. Converting from data frame to `tbl` is quite easy. All we need to do is call the function `as.tbl()`.

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

nof1_tbl<-as.tbl(nof1); nof1_tbl

## # A tibble: 900 × 10
##        ID   Day    Tx SelfEff SelfEff25  WPSS SocSuppt  PMss PMss3 PhyAct
##     <dbl> <dbl> <dbl>   <dbl>     <dbl> <dbl>    <dbl> <dbl> <dbl>  <dbl>
## 1       1     1     1      33         8  0.97     5.00  4.03  1.03     53
## 2       1     2     1      33         8 -0.17     3.87  4.03  1.03     73
## 3       1     3     0      33         8  0.81     4.84  4.03  1.03     23
…

## 8       1     8     0      33         8 -0.34     3.69  4.03  1.03      0
## 9       1     9     1      33         8 -0.74     3.29  4.03  1.03     73
## 10      1    10     1      33         8 -0.38     3.66  4.03  1.03    114
## # ... with 890 more rows
```

This looks like a normal data frame. If you are using R Studio, displaying the `nof1_tbl` will show the same output as `nof1`.

16.4.2 *Making Data Frames Faster with Data.Table*

Similar to `tbl`, the `data.table` package provides another alternative to data frame object representation. `data.table` objects are processed in R much faster compared to standard data frames. Also, all of the functions that can accept data frame could be applied to `data.table` objects as well. The function `fread()` is able to read a local CSV file directly into a `data.table`.

```
#install.packages("data.table")
library(data.table)

nof1<-fread("C:/Users/Dinov/Desktop/02_Nof1_Data.csv")
```

Another amazing property of `data.table` is that we can use subscripts to access a specific location in the dataset just like `dataset[row, column]`. It also allows the selection of rows with Boolean expression and direct application of functions to those selected rows. Note that column names can be used to call the specific column in `data.table`, whereas with data frames, we have to use the `dataset$columnName` syntax.

```
nof1[ID==1, mean(PhyAct)]

## [1] 52.66667
```

This useful functionality can also help us run complex operations with only a few lines of code. One of the drawbacks of using `data.table` objects is that they are still limited by the available system memory.

16.4.3 *Creating Disk-Based Data Frames with `ff`*

The `ff` (fast-files) package allows us to overcome the RAM limitations of finite system memory. For example, it helps with operating datasets with billions of rows. `ff` creates objects in `ffdf` formats, which is like a map that points to a location of the data on a disk. However, this makes `ffdf` objects inapplicable for most R functions. The only way to address this problem is to break the huge dataset into small chunks. After processing a batch of these small chunks, we have to combine the results to reconstruct the complete output. This strategy is relevant in parallel computing, which will be discussed in detail in the next section. First, let's download one of the large datasets in our datasets archive, UQ_VitalSignsData_Case04.csv.

```
# install.packages("ff")
library(ff)
# vitalsigns<-read.csv.ffdf(file="UQ_VitalSignsData_Case04.csv", header=T)
vitalsigns<-
read.csv.ffdf(file="https://umich.instructure.com/files/366335/download?
download_frd=1", header=T)
```

As mentioned earlier, we cannot apply functions directly on this object.

```
mean(vitalsigns$Pulse)
## Warning in mean.default(vitalsigns$Pulse): argument is not numeric or
## logical: returning NA
## [1] NA
```

For basic calculations on such large datasets, we can use another package, `ffbase`. It allows operations on `ffdf` objects using simple tasks like: mathematical operations, query functions, summary statistics and bigger regression models using packages like `biglm`, which will be mentioned later in this chapter.

```
# install.packages("ffbase")
library(ffbase)
mean(vitalsigns$Pulse)
## [1] 108.7185
```

16.4.4 *Using Massive Matrices with* `bigmemory`

The previously introduced packages include alternatives to `data.frames`. For instance, the `bigmemory` package creates alternative objects to 2D matrices (second-order tensors). It can store huge datasets and can be divided into small chunks that can be converted to data frames. However, we cannot directly apply machine-learning methods on this type of objects. More detailed information about the bigmemory package is available online.

16.5 Parallel Computing

In previous chapters, we saw various machine-learning techniques applied as serial computing tasks. The traditional protocol involves: First, applying *function 1* to our raw data. Then, using the output from *function 1* as an input to *function 2*. This process may be iterated over a series of functions. Finally, we have the terminal output generated by the last function. This serial or linear computing method is straightforward but time consuming and perhaps sub-optimal.

Now we introduce a more efficient way of computing - *parallel computing*, which provides a mechanism to deal with different tasks at the same time and combine the outputs for all of processes to get the final answer faster. However, parallel algorithms may require special conditions and cannot be applied to all problems. If two tasks have to be run in a specific order, this problem cannot be parallelized.

16.5.1 Measuring Execution Time

To measure how much time can be saved for different methods, we can use function `system.time()`.

```
system.time(mean(vitalsigns$Pulse))
##    user  system elapsed 
##       0       0       0
```

This means calculating the mean of `Pulse` column in the `vitalsigns` dataset takes less than 0.001 seconds. These values will vary between computers, operating systems, and states of operations.

16.5.2 Parallel Processing with Multiple Cores

We will introduce two packages for parallel computing `multicore` and `snow` (their core components are included in the package `parallel`). They both have a different way of multitasking. However, to run these packages, you need to have a relatively modern multicore computer. Let's check how many cores your computer has. This function `parallel::detectCores()` provides this functionality. `parallel` is a base package, so there is no need to install it prior to using it.

```
library(parallel); detectCores()

## [1] 8
```

So, there are eight (8) cores in my computer. I will be able to run up to 6-8 parallel jobs on this computer.

The `multicore` package simply uses the multitasking capabilities of the *kernel*, the computer's operating system, to "fork" additional R sessions that share the same memory. Imagine that we open several R sessions in parallel and let each of them do part of the work. Now, let's examine how this can save time when running complex protocols or dealing with large datasets. To start with, we can use the `mclapply()` function, which is similar to `lapply()`, which applies functions to a vector and returns a vector of lists. Instead of applying functions to vectors `mcapply()` divides the complete computational task and delegates portions of it to each available core. To demonstrate this procedure, we will construct a simple, yet time consuming, task of generating random numbers. Also, we can use the `system.time()` function to track execution time.

```
set.seed(123)
system.time(c1<-rnorm(10000000))

##    user  system elapsed
##    0.64    0.00    0.64

# Note the multi core calls may not work on Windows, but will work on
Linux/Mac.
#This shows a 2-core and 4-vore invocations
# system.time(c2<-unlist(mclapply(1:2, function(x){rnorm(5000000)},
mc.cores = 2)))
# system.time(c4<-unlist(mclapply(1:4, function(x){rnorm(2500000)},
mc.cores = 4)))

# And here is a Windows (single core invocation)
system.time(c2<-unlist(mclapply(1:2, function(x){rnorm(5000000)},
mc.cores = 1)))

##    user  system elapsed
##    0.65    0.00    0.65
```

The `unlist()` is used at the end to combine results from different cores into a single vector. Each line of code creates 10,000,000 random numbers. The `c1` call took the longest time to complete. The `c2` call used two cores to finish the task (each core handled 5,000,000 numbers) and used less time than `c1`. Finally, `c4` used all four cores to finish the task and successfully reduced the overall time. We can see that when we use more cores the overall time is significantly reduced.

The `snow` package allows parallel computing on multicore multiprocessor machines or a network of multiple machines. It might be more difficult to use but it's also certainly more flexible. First we can set how many cores we want to use via `makeCluster()` function.

```
# install.packages("snow")
library(snow)

cl<-makeCluster(2)
```

This call might cause your computer to pop up a message warning about access though the firewall. To do the same task we can use `parLapply()` function in the `snow` package. Note that we have to call the object we created with the previous `makeCluster()` function.

```
system.time(c2<-unlist(parLapply(cl, c(5000000, 5000000), function(x) {
rnorm(x)})))

##    user  system elapsed
##    0.11    0.11    0.64
```

While using `parLapply()`, we have to specify the matrix and the function that will be applied to this matrix. Remember to stop the cluster we made after completing the task, to release back the system resources.

```
stopCluster(cl)
```

16.5.3 *Parallelization Using* `foreach` *and* `doParallel`

The `foreach` package provides another option of parallel computing. It relies on a loop-like process basically applying a specified function for each item in the set, which again is somewhat similar to `apply()`, `lapply()` and other regular functions. The interesting thing is that these loops can be computed in parallel saving substantial amounts of time. The `foreach` package alone cannot provide parallel computing. We have to combine it with other packages like `doParallel`. Let's reexamine the task of creating a vector of 10,000,000 random numbers. First, register the 4 compute cores using `registerDoParallel()`.

```
# install.packages("doParallel")
library(doParallel)

cl<-makeCluster(4)
registerDoParallel(cl)
```

Then we can examine the time saving `foreach` command.

```
#install.packages("foreach")
library(foreach)
system.time(c4<-foreach(i=1:4, .combine = 'c')
            %dopar% rnorm(2500000))

##    user  system elapsed
##    0.11    0.18    0.54
```

Here we used four items (each item runs on a separate core), `.combine=c` allows `foreach` to combine the results with the parameter `c()`, generating the aggregate result vector.

Also, don't forget to close the `doParallel` by registering the sequential backend.

```
unregister<-registerDoSEQ()
```

16.5.4 GPU Computing

Modern computers have graphics cards, GPUs (Graphical Processing Units), that consists of thousands of cores, however they are very specialized, unlike the standard CPU chip. If we can use this feature for parallel computing, we may reach amazing performance improvements, at the cost of complicating the processing algorithms and increasing the constraints on the data format. Specific disadvantages of GPU computing include reliance on proprietary manufacturer (e.g., NVidia) frameworks and Complete Unified Device Architecture (CUDA) programming language. CUDA allows programming of GPU instructions into a common computing language. This paper provides one example of using GPU computation to significantly improve the performance of advanced neuroimaging and brain mapping processing of multidimensional data.

The R package `gputools` is created for parallel computing using NVidia CUDA. Detailed GPU computing in R information is available online.

16.6 Deploying Optimized Learning Algorithms

As we mentioned earlier, some tasks can be parallelized easier than others. In real world situations, we can pick the algorithms that lend themselves well to parallelization. Some of the R packages that allow parallel computing using ML algorithms are listed below.

16.6.1 Building Bigger Regression Models with `biglm`

`biglm` allows training regression models with data from SQL databases or large data chunks obtained from the `ff` package. The output is similar to the standard `lm()` function that builds linear models. However, `biglm` operates efficiently on massive datasets.

16.6.2 Growing Bigger and Faster Random Forests with `bigrf`

The `bigrf` package can be used to train random forests combining the `foreach` and `doParallel` packages. In Chap. 15, we presented random forests as machine learners ensembling multiple tree learners. With parallel computing, we can split the task of creating thousands of trees into smaller tasks that can be outsourced to each available compute core. We only need to combine the results at the end. Then, we will obtain the exact same output in a relatively shorter amount of time.

16.6.3 Training and Evaluation Models in Parallel with `caret`

Combining the `caret` package with `foreach`, we can obtain a powerful method to deal with time-consuming tasks like building a random forest learner. Utilizing the same example we presented in Chap. 15, we can see the time difference of utilizing the `foreach` package.

```
#library(caret)
system.time(m_rf <- train(CHARLSONSCORE ~ ., data = qol, method = "rf",
metric = "Kappa", trControl = ctrl, tuneGrid = grid_rf))

##    user  system elapsed
##  130.05    0.40  130.49
```

It took more than a minute to finish this task in standard execution model purely relying on the regular `caret` function. Below, this same model training completes much faster using parallelization (less than half the time) compared to the standard call above.

```
set.seed(123)
cl<-makeCluster(4)
registerDoParallel(cl)
getDoParWorkers()

## [1] 4

system.time(m_rf <- train(CHARLSONSCORE ~ ., data = qol, method = "rf",
metric = "Kappa", trControl = ctrl, tuneGrid = grid_rf))

##    user  system elapsed
##    4.61    0.02   47.70

unregister<-registerDoSEQ()
```

16.7 Practice Problem

Try to analyze the co-appearance network in the novel “Les Miserables”. The data contains the weighted network of co-appearances of characters in Victor Hugo’s novel “Les Miserables”. Nodes represent characters as indicated by the labels and edges connect any pair of characters that appear in the same chapter of the book. The values on the edges are the number of such co-appearances.

miserables<-read.table("https://umich.instructure.com/files/330389/download?download_frd=1", sep="", header=F) head(miserables)

Also, try to interrogate some of the larger datasets we have by using alternative parallel computing and big data analytics.

16.8 Assignment: 16. Specialized Machine Learning Topics

16.8.1 Working with Website Data

- Download the Main SOCR Wiki Page and compare `RCurl` and `httr`.
- Read and write XML code for the SOCR Main Page.
- Scrape the data from the SOCR Main Page.

16.8.2 Network Data and Visualization

- Download 03_les_miserablese_GraphData.txt
- Visualize this undirected network.
- Summary the graph and explain the output.
- Calculate degree and the centrality of this graph.
- Find out some important characters.
- Will the result change or not if we assume the graph is directed?

16.8.3 Data Conversion and Parallel Computing

- Download CaseStudy12_ AdultsHeartAttack_Data.xlsx or require online.
- load this data as data frame.
- Use `Export()` or `write.xlsx()` to renew the xlsx file.
- Use `rio` package to convert this ".xlsx" "file to" ".csv".
- Generate generalizing tabular data structures.
- Generate data.table.
- Create disk-based data frames and perform basic calculation.
- Perform basic calculation on the last 5 columns as a big matrix.
- Use DIAGNOSIS, SEX, DRG, CHARGES, LOS and AGE to predict DIED with randomForest setting `ntree=20000`. Notice: sample without replacement to get an as large as possible balanced dataset.
- Run `train()` in `caret` and detect the execute time.
- Detect cores and make proper number of clusters.

- Rerun `train()` parallelized and compare the execute time.
- Use `foreach` and `doMC` to design a parallelized random forest with `ntree=20000` totally and compare the execute time with sequential execution.

References

Data Streams in R:https://cran.r-project.org/web/packages/stream/vignettes/stream.pdf
Dplyr:https://cran.rstudio.com/web/packages/dplyr/vignettes/introduction.html
doParallel:https://cran.rproject.org/web/packages/doParallel/vignettes/gettingstartedParallel.pdf
Mailund, T. (2017) *Beginning Data Science in R: Data Analysis, Visualization, and Modelling for the Data Scientist*, Apress, ISBN 1484226712, 9781484226711