Molecular Similarity

S. Kim, J. Cuadros

November 23rd, 2019

Objectives

- Generete molecular fingerprints for a given molecule.
- Evaluate structural similarity between molecules using different molecular fingerpints and similarity metrics.

Many useful documents/papers describe various aspects of molecular similarity, including molecular fingerprints and similarity measures. Please read these if you need more details.

- Getting Started with the RDKit in Python (https://www.rdkit.org/docs/GettingStartedInPython.html#fingerprinting-and-molecular-similarity (https://www.rdkit.org/docs/GettingStartedInPython.html#fingerprinting-and-molecular-similarity))
- Fingerprint Generation, GraphSim Toolkit 2.4.2 (https://docs.eyesopen.com/toolkits/python/graphsimtk/fingerprint.html (https://docs.eyesopen.com/toolkits/python/graphsimtk/fingerprint.html))
- Chemical Fingerprints (https://docs.chemaxon.com/display/docs/Chemical+Fingerprints (https://docs.chemaxon.com/display/docs/Chemical+Fingerprints))
- Extended-Connectivity Fingerprints (https://doi.org/10.1021/ci100050t (https://doi.org/10.1021/ci100050t))

1. Fingerprint Generation

```
if (!require("rcdk", quietly=TRUE)) {
  install.packages("rcdk", repos="https://cloud.r-project.org/",
         quiet=TRUE, type="binary")
  library("rcdk")
}

if (!require("fingerprint", quietly=TRUE)) {
  install.packages("fingerprint", repos="https://cloud.r-project.org/",
         quiet=TRUE, type="binary")
  library("fingerprint")
}

help(package="rcdk")
help(package="fingerprint")
```

```
smi <- "CC(C)C1=C(C(=C(N1CC[C@H](C[C@H](CC(=O)O)O)O)C2=CC=C(C=C2)F)C3=CC=CC=C3)C(=O)NC4=CC=CC
=C4"
mol <- parse.smiles(smi)
```

1-(1) MACCS keys

The MACCS key is a binary fingerprint (a string of 0's and 1's). Each bit position represents the presence (=1) or absence (=0) of a pre-defined structural feature. The feature definitions for the MACCS keys are available at:
https://github.com/rdkit/rdkit/blob/master/rdkit/Chem/MACCSkeys.py (https://github.com/rdkit/rdkit/blob/master/rdkit/Chem/MACCSkeys.py)
http://www.mayachemtools.org/docs/modules/html/MACCSKeys.html (http://www.mayachemtools.org/docs/modules/html/MACCSKeys.html)

```
fp <- get.fingerprint(mol[[1]], type = 'maccs',
                      fp.mode = 'bit', verbose=FALSE)
str(fp)
```

```
## Formal class 'fingerprint' [package "fingerprint"] with 6 slots
##   ..@ bits    : num [1:60] 42 45 50 53 62 74 75 76 80 83 ...
##   ..@ nbit    : num 166
##   ..@ folded  : logi FALSE
##   ..@ provider: chr "CDK"
##   ..@ name    : chr ""
##   ..@ misc    : list()
```

```
as.character(fp)
```

```
## [1] "00000000000000000000000000000000000000001001000010010000000010000000000011100010010
10101110001101100010011011000001000111000011011110110110100111111111111111111101110"
```

```
nchar(as.character(fp))
```

```
## [1] 166
```

These are some methods that allow you to get some additional information on the fingerprint.

```
print(paste("Number of bits:", length(fp)))
```

```
## [1] "Number of bits: 166"
```

```
print(paste("Number of ON bits:", length(fp@bits)))
```

```
## [1] "Number of ON bits: 60"
```

```
print(paste("Number of OFF bits:", fp@nbit-length(fp@bits)))
```

```
## [1] "Number of OFF bits: 106"
```

```
as.character(fp)
```

```
## [1] "00000000000000000000000000000000000000001001000010010000000010000000000011100010010
10101110001101100010011011000001000111000011011110110110100111111111111111111101110"
```

```
(fp_bin <- unlist(strsplit(as.character(fp),"")))
```

```
##   [1] "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
##  [18] "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0"
##  [35] "0" "0" "0" "0" "0" "0" "0" "1" "0" "0" "1" "0" "0" "0" "0" "1" "0"
##  [52] "0" "1" "0" "0" "0" "0" "0" "0" "0" "0" "1" "0" "0" "0" "0" "0" "0"
##  [69] "0" "0" "0" "0" "0" "1" "1" "1" "0" "0" "0" "1" "0" "0" "1" "0" "1"
##  [86] "0" "1" "0" "1" "1" "1" "1" "0" "0" "0" "1" "1" "0" "1" "1" "0" "0"
## [103] "0" "1" "0" "0" "1" "1" "0" "1" "1" "0" "0" "0" "0" "0" "1" "0" "0"
## [120] "0" "1" "1" "1" "0" "0" "0" "0" "1" "1" "0" "1" "1" "1" "1" "0" "1"
## [137] "1" "0" "1" "1" "0" "1" "0" "0" "1" "1" "1" "1" "1" "1" "1" "1" "1"
## [154] "1" "1" "1" "1" "1" "1" "1" "1" "0" "1" "1" "1" "0"
```

```
paste(fp_bin, collapse="")
```

```
## [1] "00000000000000000000000000000000000000001001000010010000000100000000000111000100101
0101111000110110001001101100000100011100001101111011011010011111111111111111101110"
```

```
fp_bytes <- substring(paste("00", as.character(fp), sep=""),
          seq(1, length(fp)+2, 8), seq(8, length(fp)+2, 8))
# bits in bytes are read right to left, https://code.google.com/archive/p/chem-fingerprints/w
ikis/FPS.wiki
fp_bytes <- sapply(fp_bytes, function(x)
    paste(rev(strsplit(x,"")[[1]]),collapse=""))

(fp_bytes <- strtoi(fp_bytes,base=2))
```

```
##  [1]   0   0   0   0   0  72  72 128   0  56  82  61  54 178  65  28 246
## [18] 182 252 255 119
```

```
(fp_hex <- as.raw(fp_bytes))
```

```
##  [1] 00 00 00 00 00 48 48 80 00 38 52 3d 36 b2 41 1c f6 b6 fc ff 77
```

```
paste(fp_hex, collapse = "")
```

```
## [1] "00000000004848800038523d36b2411cf6b6fcff77"
```

```
fp_bin2 <- unlist(lapply(paste("0x",fp_hex,sep=""),function(x) rawToBits(as.raw(x))))
fp_bin2 <- as.numeric(fp_bin2)
substring(paste(fp_bin2, collapse=""),3,168)
```

```
## [1] "00000000000000000000000000000000000000001001000010010000000100000000000111000100101
0101111000110110001001101100000100011100001101111011011010011111111111111111101110"
```

Exercise 1a: Generate the MACCS keys for the molecules represented by the following SMILES, and get the positions of the bits set to ON in each of the three fingerprints. What fragments do these bit positions correspond to?

```
smiles <- c('C1=CC=CC=C1', # Benzene (Kekule)
            'c1ccccc1',    # Benzene ("Aromatized" carbons)
            'C1CCCCC1')     # Cyclohexene
```

```
# Write your code here
```

Write the fragment definition of the bits ON (one is already provided for you as an example). - 118: ACH2CH2A > 1

1-(2) Circular Fingerprints

Circular fingerprints are hashed fingerprints. They are generated by exhaustively enumerating “circular” fragments (containing all atoms within a given radius from each heavy atom of the molecule) and then hashing these fragments into a fixed-length bitstring. (Here, the “radius” from an atom is measured by the number of bonds that separates two atoms).

Examples of circular fingerprints are the extended-connectivity fingerprint (ECFPs) and their variant called FCFPs (Functional-Class Fingerprints), originally described in a paper by Rogers and Hahn (https://doi.org/10.1021/ci100050t (https://doi.org/10.1021/ci100050t)). Sometimes, for instance in RDKit, these fingerprints are called “Morgan Fingerprints” (https://www.rdkit.org/docs/GettingStartedInPython.html#morgan-fingerprints-circular-fingerprints (https://www.rdkit.org/docs/GettingStartedInPython.html#morgan-fingerprints-circular-fingerprints)).

CDK can compute a ECFP6 fingerprint.

```
fp <- get.fingerprint(mol[[1]], type = 'circular',
                      fp.mode = 'bit', verbose=FALSE)
as.character(fp)
```

```
## [1] "00000000000000000001000100000001010001000000000000000000000001001000000000010000000
00000000101100000000000000000010000000000000001000010000010000001000000000000000000000000
00000001000000100000000000001000000000000000000000001000000000000000101001000000000000010000
00000010000000000000000000000000000000000000001001010010001000101001000000000000000000010000
00000000010000001000000011000000000000000000000000001001000000000000000000000000000000000000
00000000010000000000010000001001000000000000000001001100010000000100000100001000000000000001
00000000000000000000000001010000000000000000000000000000000000000000000000000100000001000000000
00000000000001101000000000000000000100000000000000000100000001000000000000000110000000000
00000000000000010000001000000000000010010000000000000000000010000000000000010000000000
00010000000000000000000000000000000000000001000000000010000000000000000000000010000000000
00000000000010000000001000000000000000000110000000000000001000000001000000000000000000
000000000"
```

Morgan fingerprints can be obtained from the ChEMBL webservice, which is based on RDKit.

```
if(!require("httr")) {
  install.packages(("httr"), repos="https://cloud.r-project.org/",
         quiet=TRUE, type="binary")
  library("httr")
}
```

```
## Loading required package: httr
```

```
sdf <- readLines(paste("https://cactus.nci.nih.gov/chemical/structure/",
        URLencode(smi,reserved = T), "/SDF", sep=""))
sdf <- paste(sdf, collapse="\n")

url <- paste("https://www.ebi.ac.uk/chembl/api/utils/sdf2fps",
       "?n_bits=1024&radius=2",sep="")
res  <- POST(url,
           body=sdf)
response <- rawToChar(res$content)
fp_hex <- strsplit(response, "\n")[[1]][4]
(fp_hex <- strsplit(fp_hex, "\t")[[1]][1])
```

```
## [1] "02000020060020000100010400202002010100010082008000000000080000000004000802000000000008090000000248000000000000000000000040000000000000000000000000004400000000100400400002001008800248000900000000000001840000000000020000000200101000080000004002000000000400"
```

```
fp_hex <- substring(fp_hex, seq(1,nchar(fp_hex),2), seq(2,nchar(fp_hex),2))

fp_bin <- unlist(lapply(paste("0x",fp_hex,sep=""),function(x) rawToBits(as.raw(x))))
(fp_bin <- as.numeric(fp_bin))
```

```
##    [1] 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
##   [35] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0
##   [69] 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
##  [103] 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0
##  [137] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
##  [171] 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
##  [205] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
##  [239] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
##  [273] 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [307] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [341] 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
##  [375] 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
##  [409] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [443] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [477] 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [511] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [545] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [579] 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
##  [613] 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [647] 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
##  [681] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1
##  [715] 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0
##  [749] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [783] 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0
##  [817] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
##  [851] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [885] 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
##  [919] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [953] 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
##  [987] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
## [1021] 0 0 0 0
```

```
(fp <- paste(fp_bin, collapse=""))
```

```
## [1] "0100000000000000000000000000010001100000000000000001000000000100000000000000010000
00001000000000000000001000000010001000000100000001000000000000000100000000000000100000100
00000000000001000000000000000000000000000000000010000000000000000000000000000000000000010000
00000000000010000010000000000000000000000000000000000000000000000000000000000000000100010010000
00000000000000000000100100000000100000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000010000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000010000000000010000000000000000000000010000000000000000000
00010001000000000000000000000001001000000000000000010001000000001000000001001000000000
10010000000000000000000000000000000000000000000000010000000010000100000000000000000000
00000000000000000001000000000000000000000000000000000000000000001001000000100000000000000000
00000000000010000000000000001000000000000000000010000000000000000000000000000000000001
000000000"
```

```
# Using the tools of the fingerprint package
fp_obj <- fps.lf(strsplit(response, "\n")[[1]][4])
fp <- new("fingerprint", nbit = 1024,
          bits = as.numeric(fp_obj[[2]]),
          folded = FALSE,
          provider = gsub("#software=","",
                          strsplit(response, "\n")[[1]][3], fixed=T),
          name = fp_obj[[1]],
          misc = list())
str(fp)
```

```
## Formal class 'fingerprint' [package "fingerprint"] with 6 slots
##   ..@ bits    : num [1:53] 2 30 34 35 54 65 81 91 110 118 ...
##   ..@ nbit    : num 1024
##   ..@ folded  : logi FALSE
##   ..@ provider: chr "RDKit/2017.03.3"
##   ..@ name    : chr "XUKUURHRXDUEBC-KAYWLYCHSA-N"
##   ..@ misc    : list()
```

```
as.character(fp)
```

```
## [1] "0100000000000000000000000000010001100000000000000001000000000100000000000000010000
00001000000000000000001000000010001000000100000001000000000000000100000000000000100000100
00000000000001000000000000000000000000000000000010000000000000000000000000000000000000010000
00000000000010000010000000000000000000000000000000000000000000000000000000000000000100010010000
00000000000000000000100100000000100000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000010000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000010000000000010000000000000000000000010000000000000000000
00010001000000000000000000000001001000000000000000010001000000001000000001001000000000
10010000000000000000000000000000000000000000000000010000000010000100000000000000000000
00000000000000000001000000000000000000000000000000000000000000001001000000100000000000000000
00000000000010000000000000001000000000000000000010000000000000000000000000000000000001
000000000"
```

When comparing the RDKit's Morgan fingerprints with the ECFP/FCFP fingerprints, it is important to remember that the name of ECFP/FCFP fingerprints are suffixed with the **diameter** of the atom environments considered, while the Morgan Fingerprints take a **radius** parameter (e.g., the second argument "2" of GetMorganFingerprintAsBitVect() in the above code cell). The Morgan fingerprint generated above (with a radius of 2) is comparable to the ECFP4 fingerprint (with a diameter of 4).

Exercise 1b: For the moleculess below, generate the 512-bit-long Morgan Fingeprint.

- Search for the compounds by name and get their SMILES strings.
- Generate the molecular fingerprints from the SMILES strings.
- Print the generated fingerprints.

```
synonyms <- c('diphenhydramine', 'cetirizine', 'fexofenadine', 'loratadine')
```

```
# Write your code here
```

1-(3) Path-Based Fingeprints

Path-based fingerprints are also hashed fingerprints. They are generated by enumerating linear fragments of a given length and hashing them into a fixed-length bitstring. An example of this, is the standard fingerprint in CDK. Another example is the RDKit's topological fingeprint.

In CDK, size and depth allow specifying fingerprint and maximum path size, used for constructing the fingerprint. They default to 1024 bits and depth 6.

```
fp <- get.fingerprint(mol[[1]], type = 'standard',
                      fp.mode = 'bit', size = 128, depth= 4, verbose=FALSE)
str(fp)
```

```
## Formal class 'fingerprint' [package "fingerprint"] with 6 slots
##   ..@ bits    : num [1:65] 3 6 7 9 10 11 12 14 17 19 ...
##   ..@ nbit    : int 128
##   ..@ folded  : logi FALSE
##   ..@ provider: chr "CDK"
##   ..@ name    : chr ""
##   ..@ misc    : list()
```

```
as.character(fp)
```

```
## [1] "00100110111101001011000100110101010001100101010010000000100110011001101101001111110000
0011011111111100100100010011111010111111000"
```

```
nchar(as.character(fp))
```

```
## [1] 128
```

```
length(fp@bits)/length(fp)
```

```
## [1] 0.5078125
```

```
fp <- get.fingerprint(mol[[1]], type = 'standard',
                      fp.mode = 'bit', size = 2048, depth= 7, verbose=FALSE)
str(fp)
```

```
## Formal class 'fingerprint' [package "fingerprint"] with 6 slots
##   ..@ bits    : num [1:305] 12 16 23 25 27 31 35 48 52 54 ...
##   ..@ nbit    : int 2048
##   ..@ folded  : logi FALSE
##   ..@ provider: chr "CDK"
##   ..@ name    : chr ""
##   ..@ misc    : list()
```

```
as.character(fp)
```

```
## [1] "00000000000100010000001010100010001000000000000100010100000000000010000000000000001
01000000000001000100000001000000001100000000000000001001000100000110011000000000001000101000
00000000011010010000100001000010001000010000100011100000000000000000000010000100000010001000
00000010000000110010001000100001000010000001010000001100100001000000000100000010000000010010
00000000100000001000000101000000000000010000000000010000001001000000000100000001010001100000
10000000000000001000110000100101000000000000001000100000001100000000000100000101000010000000
00000000000000000000000001000010000000000000000000011000000000001000000101010000000000000000
00000011001000101000000001001000100000000000000000000000000001100000000000000010000000001100000
00000000000000000001010000000000000100010001110100000000010000000100000000001000000100000000
00000000001000000000100000000100010111000000000100000010000000000000000000000011100010001000
00000000000000000000000000010000000000010000000000010000000000000000000001000100000001000000
01000101100010000100000000000000000001000000001010001110100000000000001110010000000101001100
10000100000000001001001000000100000000001101000000000000000011000000000010000000100000000
01001000000000000100100000000001001001000100000000001010000001000000000110000001000110100000
01001000000101000010000000000000000000110100000000000010010000000000000000000000000100010000
10000100000000001000100001000000001010000100100000010000000000100001000000100000001101100
01111100000001000000001001000000000000001010000001010000001010000010000000000001000010101000 1
00000000100001000001000000001001000000000100000000000100000000000000000000100001000000100000
00000000001100000000000000010000000000000000100000000000100000000000000010001000000000001000
00000010000001100010000000100000000000010000001010010001101000010100000001000000100000010000
00000100001001000000000001000000010000000000000000000100000001000011000000000010000001100000
00000010001000000000110000001100101000000100000000010000000001000000000000101000100010000000
0000000000"
```

```
nchar(as.character(fp))
```

```
## [1] 2048
```

```
length(fp@bits)/length(fp)
```

```
## [1] 0.1489258
```

1-(4) PubChem Fingerprint

The PubChem Fingerprint is a 881-bit-long binary fingerprint (ftp://ftp.ncbi.nlm.nih.gov/pubchem/specifications/pubchem_fingerprints.pdf (ftp://ftp.ncbi.nlm.nih.gov/pubchem/specifications/pubchem_fingerprints.pdf)). Similar to the MACCS keys, it uses a pre-defined fragment dictionary. The PubChem fingerprint for each compound in PubChem can be downloaded from PubChem. However, because they are base64-encoded, they should be decoded into binary bitstrings or bitvectors.

Details about how to decode base64-encoded PubChem fingerprints is described on the last page of the PubChem Fingerprint specification (ftp://ftp.ncbi.nlm.nih.gov/pubchem/specifications/pubchem_fingerprints.pdf (ftp://ftp.ncbi.nlm.nih.gov/pubchem/specifications/pubchem_fingerprints.pdf)). Below is a user-defined function that decodes a PubChem fingerprint into a bit string.

```
if(!require("jsonlite")) {
  install.packages(("jsonlite"), repos="https://cloud.r-project.org/",
         quiet=TRUE, type="binary")
  library("jsonlite")
}
```

```
## Loading required package: jsonlite
```

```
pcfps <- 'AAADcYBgAAAAAAAAAAAAAAAAAAAAAAAAAAAwAAAAAAAAAAAABAAAAGAAAAAAACACAEAAwAIAAAACAACBCAAA
CAAAgAAAIiAAAAIgIICKAERCAIAAggAAIiAcAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA=='

pcfps_raw <- base64_dec(pcfps)

rawToBits(base64_dec("AAAD"))
```

```
##  [1] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 01 00 00 00 00 00
## [24] 00
```

```
sapply(base64_dec("AAAD"),function(x) paste(as.numeric(rawToBits(x)),collapse=""))
```

```
## [1] "00000000" "00000000" "11000000"
```

```
pcfps_bin <- sapply(pcfps_raw,
                    function(x) paste(as.numeric(rev(rawToBits(x))),collapse=""))
pcfps_bin <- substring(paste(pcfps_bin,collapse=""),33,913)
nchar(pcfps_bin)
```

```
## [1] 881
```

```
pcfps_bin
```

```
## [1] "1000000001100000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
110000000000000000000000000000000000000000000000000000000000000000000000000001000000000000000
000000000000110000000000000000000000000000000000000000000000010000000000010000000000100000000
000000110000000000001000000000000000000000000000000100000000000000000100000010000100000000000
000000000000010000000000000000000100000000000000000000000001000100010000000000000000000000000
001000100000001000001000000010001010000000000100010001000010000000001000000000000000100000100
000000000000000001000100010000000011100000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000"
```

The generated bitstring can be converted to a bitvector that can be used for molecular similarity computations (to be discussed in the next section).

```
(binvect <- as.numeric(unlist(strsplit(pcfps_bin,""))))
```

```
##   [1] 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [36] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [71] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [106] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [141] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [176] 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [211] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [246] 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [281] 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [316] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
## [351] 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1
## [386] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
## [421] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0
## [456] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [491] 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1
## [526] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0
## [561] 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0
## [596] 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
## [631] 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
## [666] 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [701] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [736] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [771] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [806] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [841] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [876] 0 0 0 0 0 0
```

2. Computation of similarity scores

```
cids <- c(54454,  # Simvastatin (Zocor)
          54687,  # Pravastatin (Pravachol)
          60823,  # Atorvastatin (Lipitor)
         446155,  # Fluvastatin (Lescol)
         446157,  # Rosuvastatin (Crestor)
        5282452,  # Pitavastatin (Livalo)
       97938126)  # Lovastatin (Altoprev)
```

Let's get the SMILES strings from PubChem, generate Mol objects from them, and draw their chemical structures.

```
prolog <- "https://pubchem.ncbi.nlm.nih.gov/rest/pug"

str_cid <- paste(as.character(cids), collapse=",")

url <- paste(prolog, "/compound/cid/", str_cid, "/property/isomericsmiles/txt", sep="")
smiles <- readLines(url)
```

```
if(!require("httr")) {
  install.packages(("httr"), repos="https://cloud.r-project.org/",
         quiet=TRUE, type="binary")
  library("httr")
}
if(!require("jsonlite")) {
  install.packages(("jsonlite"), repos="https://cloud.r-project.org/",
         quiet=TRUE, type="binary")
  library("jsonlite")
}
if(!require("png")) {
  install.packages(("png"), repos="https://cloud.r-project.org/",
         quiet=TRUE, type="binary")
  library("png")
}
```

```
## Loading required package: png
```

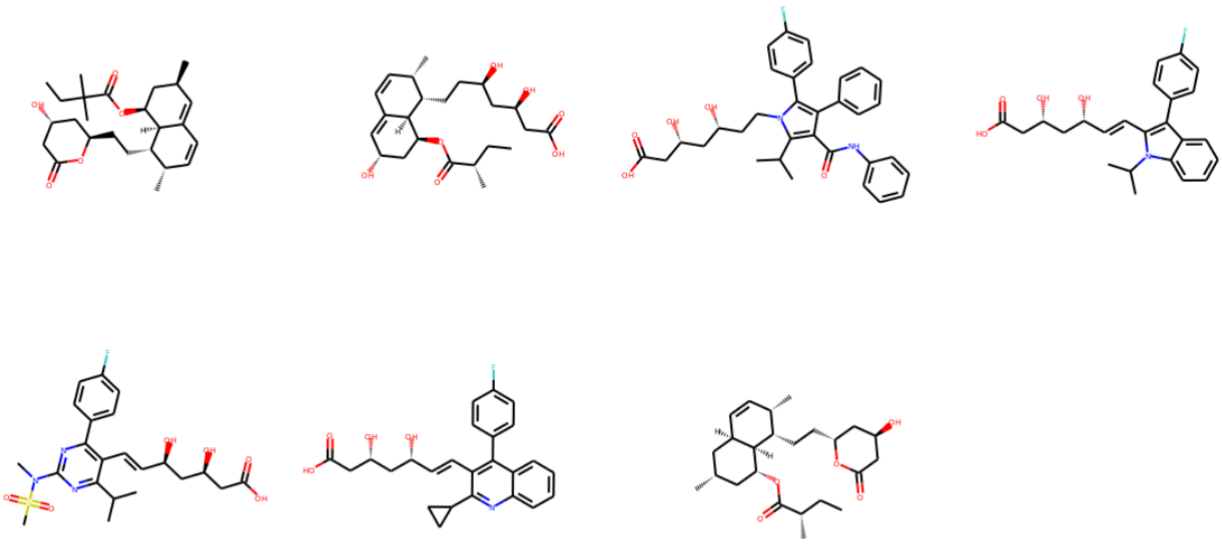
```
if(!require("grid")) {
  install.packages(("grid"), repos="https://cloud.r-project.org/",
       quiet=TRUE, type="binary")
  library("grid")
}
```

```
## Loading required package: grid
```

```
if(!require("gridExtra")) {
  install.packages(("gridExtra"), repos="https://cloud.r-project.org/",
       quiet=TRUE, type="binary")
  library("gridExtra")
}
```

```
## Loading required package: gridExtra
```

```
url_img <- paste("https://www.ebi.ac.uk/chembl/api/utils/smiles2image",
    "?size=300&engine=rdkit",sep="")
res  <- POST(url_img,
            body=list(smiles=paste(smiles,collapse="\n")))
img <- readPNG(res$content, native=TRUE)
grid.arrange(rasterGrob(img))
```



Now generate MACCS keys for each compound.

```
mols <- parse.smiles(smiles)

fps <- sapply(mols,
              function(x) get.fingerprint(x, type="maccs"))

fps_bin <- sapply(fps, as.character)
fps_bin
```

```
##                                                                             CCC(C)
(C)C(=O)O[C@H]1C[C@H](C=C2[C@H]1[C@H]([C@H](C=C2)C)CC[C@@H]3C[C@H](CC(=O)O3)O)C
## "000000000000000000000000100000000000000000000000010000001000000001000000010100000000000001
01000000110000110010001011100000010011110010001101110101110110101101011001110"
##                                                                           CC[C@H]
(C)C(=O)O[C@H]1C[C@@H](C=C2[C@H]1[C@H]([C@H](C=C2)C)CC[C@H](C[C@H](CC(=O)O)O)O)O
## "000000000000000000000000100000000000000000000000100100000000000000000000000010000000000001
11000000010000110010000011100000010011110110001001110101110110101101011001110"
##                                                                          CC(C)C1=C
(C(=C(N1CC[C@H](C[C@H](CC(=O)O)O)O)C2=CC=C(C=C2)F)C3=CC=CC=C3)C(=O)NC4=CC=CC=C4
## "00000000000000000000000000000000000000001001000010010000000010000000000001110001001010101
1110001101100010011011000000100011100000110111101101101001111111111111111101110"
##
CC(C)N1C2=CC=CC=C2C(=C1/C=C/[C@H](C[C@H](CC(=O)O)O)O)C3=CC=C(C=C3)F
## "00000000000000000000000010000000000000000100100001001000000001000000000000011100000001010101
0100001001000011010000000000000011100000001101001011000011011101011111111101110"
##
CC(C)C1=NC(=NC(=C1/C=C/[C@H](C[C@H](CC(=O)O)O)O)C2=CC=C(C=C2)F)N(C)S(=O)(=O)C
## "000000000000000000000000100000011000000000100100001101010010111000010000011011101100000111
0101100111001010110000100001011111100000111110110111100110111010111111111101110"
##
C1CC1C2=NC3=CC=CC=C3C(=C2/C=C/[C@H](C[C@H](CC(=O)O)O)O)C4=CC=C(C=C4)F
## "000000000000000000001000100000000000000000010010000100100000000100000000000000101000000000101
0100000011000011010000000000001010100000000110100101100001100010101111110101110"
##                                                                      CC[C@H](C)C(=
O)O[C@@H]1C[C@@H](C[C@@H]2[C@H]1[C@H]([C@H](C=C2)C)CC[C@@H]3C[C@H](CC(=O)O3)O)C
## "00000000000000000000000000000000000000000000000000001000000000000000000000000000000001
01000000110000110010000011100000010011110010001101110101110110101101011001110"
```

Now let's compute the pair-wise similarity scores among them. To make higher scores easier to find, they are indicated with the "*" character(s).

```
dfDistance <- data.frame(cid1 = numeric(0), cid2 = numeric(0),
                         score = numeric(0))

for(i in 1:(length(cids)-1)) {
    for(j in (i+1):length(cids)) {
        score <- distance(fps[[i]], fps[[j]], method="tanimoto")
        dfDistance <- rbind(dfDistance,
                            list(cid1 = cids[i], cid2 = cids[j],
                                 score = score))
    }
}

dfDistance$value <- ""
dfDistance$value[dfDistance$score >= .55] <- "*"
dfDistance$value[dfDistance$score >= .65] <- "**"
dfDistance$value[dfDistance$score >= .75] <- "***"
dfDistance$value[dfDistance$score >= .85] <- "****"

dfDistance
```

```
##       cid1     cid2     score value
## 1    54454    54687 0.8043478   ***
## 2    54454    60823 0.4109589
## 3    54454   446155 0.4426230
## 4    54454   446157 0.3452381
## 5    54454  5282452 0.4310345
## 6    54454 97938126 0.8604651  ****
## 7    54687    60823 0.4492754
## 8    54687   446155 0.4655172
## 9    54687   446157 0.3253012
## 10   54687  5282452 0.4545455
## 11   54687 97938126 0.7906977   ***
## 12   60823   446155 0.6935484    **
## 13   60823   446157 0.5294118
## 14   60823  5282452 0.5151515
## 15   60823 97938126 0.3857143
## 16  446155   446157 0.5131579
## 17  446155  5282452 0.7346939    **
## 18  446155 97938126 0.3898305
## 19  446157  5282452 0.5068493
## 20  446157 97938126 0.3048780
## 21 5282452 97938126 0.4000000
```

By default, the similarity score is generated using the **Tanimoto** equation. `fingerprint::distance` also supports other similarity metrics, including Dice, Cosine, Russel, SOkal-Michener (also known as simple matching), Kulczynski, McConnaughey, and Tversky. The definition of these metrics is available at the LibreTexts page (https://bit.ly/2kx9NCd (https://bit.ly/2kx9NCd)).

```
print(paste("Tanimoto: ",
            distance(fps[[1]], fps[[2]], method="tanimoto")))
```

```
## [1] "Tanimoto:  0.804347826086957"
```

```
print(paste("Dice: ",
            distance(fps[[1]], fps[[2]], method="dice")))
```

```
## [1] "Dice:  0.891566265060241"
```

```
print(paste("Cosine: ",
            distance(fps[[1]], fps[[2]], method="cosine")))
```

```
## [1] "Cosine:  0.892149221015262"
```

```
print(paste("Simple: ",
            distance(fps[[1]], fps[[2]], method="simple")))
```

```
## [1] "Simple:  0.94578313253012"
```

```
print(paste("McConnaughey: ",
            distance(fps[[1]], fps[[2]], method="mcconnaughey")))
```

```
## [1] "McConnaughey:  "
```

```
      # McConnaughey distance does not work in version 3.5.7 of the fingerprint package

contTable <- as.matrix(table(seq(length(fps[[1]])) %in% fps[[1]]@bits,
      seq(length(fps[[2]])) %in% fps[[2]]@bits))
a <- contTable[2,2]
b <- contTable[2,1]
c <- contTable[1,2]
d <- contTable[1,1]
dist <- (a^2 - b * c)/((a + b) * (a + c))

print(paste("McConnaughey: ",
            dist))
```

```
## [1] "McConnaughey:  0.78546511627907"
```

The Tversky score is an asymmetric similarity measure, and its computation requires the weightings of the two molecules being compared.

```
for(i in 0:10) {
    print(paste("Tversky (alpha = ", i * 0.1, ", beta = ", 1-i * .1, ") = ",
                distance(fps[[1]], fps[[2]], a= i * .1, b = 1 - i * .1,
                         method="tversky"),
                sep = ""))
}
```

```
## [1] "Tversky (alpha = 0, beta = 1) = 0.925"
## [1] "Tversky (alpha = 0.1, beta = 0.9) = 0.918114143920596"
## [1] "Tversky (alpha = 0.2, beta = 0.8) = 0.911330049261084"
## [1] "Tversky (alpha = 0.3, beta = 0.7) = 0.904645476772616"
## [1] "Tversky (alpha = 0.4, beta = 0.6) = 0.898058252427184"
## [1] "Tversky (alpha = 0.5, beta = 0.5) = 0.891566265060241"
## [1] "Tversky (alpha = 0.6, beta = 0.4) = 0.885167464114833"
## [1] "Tversky (alpha = 0.7, beta = 0.3) = 0.878859857482185"
## [1] "Tversky (alpha = 0.8, beta = 0.2) = 0.872641509433962"
## [1] "Tversky (alpha = 0.9, beta = 0.1) = 0.866510538641686"
## [1] "Tversky (alpha = 1, beta = 0) = 0.86046511627907"
```

Exercise 2a: Compute the Tanimoto similarity scores between the seven compounds used in this section, using the PubChem fingerprints

- Download the PubChem Fingerprint for the seven CIDs.
- Convert the downloaded fingerprints into bit vectors.
- Compute the pair-wise Tanimoto scores using the bit vectors.

```
# Write your code here
```

3. Interpretation of similarity scores

Using molecular fingeprints. we can compute the similarity scores between molecules. However, how should these scores be interpreted? For example, the Tanimoto score between CID 60823 and CID 446155 is computed to be 0.662, but does it mean that the two compounds are similar? How similar is similar? The

following analysis would help answer these questions.

Step 1. Randomly select 1,000 compounds from PubChem and download their SMILES strings.

```
prolog <- "https://pubchem.ncbi.nlm.nih.gov/rest/pug"

set.seed(0)
cid_max <- 138962044    # The maximum CID in PubChem as of September 2019

cids <- sample(seq(cid_max),1000)

chunk_size <- 100
num_chunks <- ceiling(length(cids) / chunk_size)

smiles = character(length(cids))

for(i in seq(num_chunks)) {
    print(i)

    idx1 <- chunk_size * (i - 1) + 1
    idx2 <- chunk_size * i
    str_cids <- paste(cids[idx1:idx2], collapse=",")

    url <- paste(prolog, "/compound/cid/", str_cids, "/property/isomericsmiles/txt", sep="")
    smiles[idx1:idx2] <- readLines(url)

    Sys.sleep(0.5)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

```
print("Done!")
```

```
## [1] "Done!"
```

```
print(paste("# Number of SMILES :", length(na.omit(smiles))))
```

```
## [1] "# Number of SMILES : 1000"
```

Step 2. Generate the MACCSKeys for each compound.

```
mols <- parse.smiles(smiles)
fps <- lapply(mols, function(x) get.fingerprint(x, type = 'maccs',
                    fp.mode = 'bit', verbose=FALSE))

str(fps[[1]])
```

```
## Formal class 'fingerprint' [package "fingerprint"] with 6 slots
##   ..@ bits    : num [1:63] 22 34 36 37 45 50 62 75 76 77 ...
##   ..@ nbit    : num 166
##   ..@ folded  : logi FALSE
##   ..@ provider: chr "CDK"
##   ..@ name    : chr ""
##   ..@ misc    : list()
```

```
print(paste("Number of compounds:", length(mols)))
```

```
## [1] "Number of compounds: 1000"
```

```
print(paste("Number of Fingerprints:", length(fps)))
```

```
## [1] "Number of Fingerprints: 1000"
```

Step 3. Compute the Tanimoto scores between compounds.

```
print(paste("Number of compound pairs:", (length(fps) * (length(fps) - 1))/2))
```

```
## [1] "Number of compound pairs: 499500"
```

```
scores <- numeric((length(fps) * (length(fps) - 1))/2)

k <- 1
for(i in 1:(length(fps)-1)) {
    for(j in (i+1):length(fps)) {
        scores[k] <- distance(fps[[i]], fps[[j]], method="tanimoto")
        k <- k + 1
    }
}

summary(scores)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.2817  0.3659  0.3676  0.4516  1.0000
```

Step 4. Generate a histogram that shows the distribution of the pair-wise scores.

```
if(!require("tidyverse")) {
  install.packages(("tidyverse"), repos="https://cloud.r-project.org/",
         quiet=TRUE, type="binary")
  library("tidyverse")
}
```

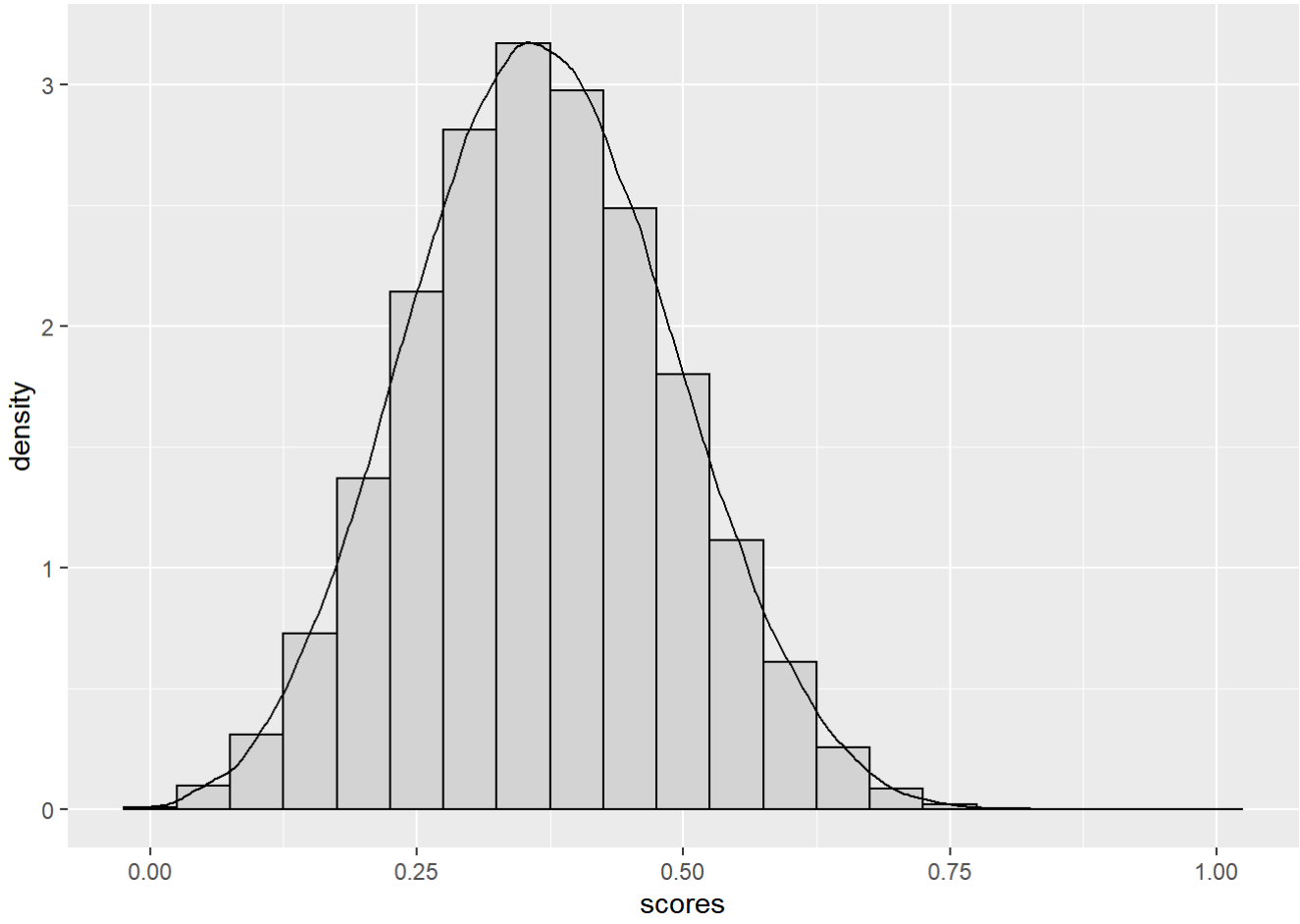
```
## Loading required package: tidyverse
```

```
## -- Attaching packages ---------------------------------------------- tidyverse 1.2.1 --
```

```
## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
```

```
## -- Conflicts ------------------------------------------------- tidyverse_conflicts() --
## x dplyr::combine() masks gridExtra::combine()
## x dplyr::count()   masks fingerprint::count()
## x dplyr::filter()  masks stats::filter()
## x purrr::flatten() masks jsonlite::flatten()
## x dplyr::lag()     masks stats::lag()
## x dplyr::matches() masks tidyr::matches(), rcdk::matches()
```

```
ggplot(NULL, aes(x=scores, y=..density..)) +
  geom_histogram(fill="lightgrey", color="black", binwidth=.05) +
  geom_density()
```




```
dfScoresTab <- data.frame(limit = 0:20*0.05,
  countLT = sapply(0:20*0.05, function(x) sum(scores>=x)))
dfScoresTab$propLT <- dfScoresTab$countLT /  length(scores)
dfScoresTab
```

```
##    limit countLT       propLT
## 1   0.00  499500 1.000000e+00
## 2   0.05  498407 9.978118e-01
## 3   0.10  494188 9.893654e-01
## 4   0.15  481541 9.640460e-01
## 5   0.20  457159 9.152332e-01
## 6   0.25  414319 8.294675e-01
## 7   0.30  349144 6.989870e-01
## 8   0.35  273979 5.485065e-01
## 9   0.40  198499 3.973954e-01
## 10  0.45  127231 2.547167e-01
## 11  0.50   75863 1.518779e-01
## 12  0.55   36841 7.375576e-02
## 13  0.60   15291 3.061261e-02
## 14  0.65    5411 1.083283e-02
## 15  0.70    1383 2.768769e-03
## 16  0.75     332 6.646647e-04
## 17  0.80      73 1.461461e-04
## 18  0.85      19 3.803804e-05
## 19  0.90       4 8.008008e-06
## 20  0.95       2 4.004004e-06
## 21  1.00       1 2.002002e-06
```

```
print(paste("Average:", sum(scores)/length(scores)))
```

```
## [1] "Average: 0.367593273104377"
```

From the distribution of the similarity scores among 1,000 compounds, we observe the following:
- If you randomly select two compounds from PubChem, the similarity score between them (computed using the Tanimoto equation and MACCS keys) is ~0.35 on average. - About %5 of randomly selected compound pairs have a similarity score greater than 0.55. - About %1 of randomly selected compound pairs have a similarity score greater than 0.65.

If two compounds have a Tanimoto score of 0.35, it is close to the avaerage Tanimoto score between randomly selected compounds and there is a 50% chance that you will get a score of 0.35 or greater just by selecting two compounds from PubChem. Therefore, it is reasonable to consider the two compounds are not similar.

The Tanimoto index may have a value ranging from 0 (for no similarity) to 1 (for identical molecules) and the midpoint of this value range is 0.5. Because of this, a Tanimoto score of **0.55** may not sound great enough to consider two compounds to be similar. However, according to the score distribution curve generated here, only **~5%** of randomly selected compound pairs will have a score greater than this.

In the previous section, we computed the similarity scores between some cholesterol-lowering drugs, and CID 60823 and CID 446155 had a Tanimoto score of **0.662**. Based on the score distribution curve generated in the second section, we can say that the probablilty of two randomly selected compounds from PubChem having a Tanimoto score greater than 0.662 is **less than 1%**.

The following code cell demonstrates how to find an appropriate similarity score threshold above which a given percentage of the compound pairs will be considered to be similar to each other.

```
# to find a threshold for top 3% compound pairs (i.e., 97% percentile)
quantile(scores,.97)
```

```
##       97%
## 0.6025694
```

Exercise 3a: In this exercise, we want to generate the distribution of the similarity scores among 1,000 compounds randomly selected from PubChem, using different molecular fingeprints and similarity metrics. For molecular fingerprints, use the following: - PubChem Fingerprint - MACCS keys - Morgan Fingerprint (ECFP4 analogue, 1024-bit-long)

For similarity metrics, use the following: - Tanimoto similarity - Dice similarity - Cosine similarity

As a result, a total of 9 distribution curves need to be generated.

Here are additional instructions to follow: - When generating the histograms, bin the scores from 0 to 1 with an increment of 0.01. - For each distribution curve, determine the similarity score threshold so that **1%** of the compound pairs have a similarity score greater than or equal to this threshold. - Use RDKit to generate the MACCS keys and Morgan fingerprint and download the PubChem fingerprints from PubChem. - For reproducibility, use **random.seed(2019)** before you generate random CIDs.

Step 1: Generate 1,000 random CIDs, download the isomeric SMILES for them, and create the RDKit mol objects from the downloaded SMILES strings.

```
# Write your code here
```

Step 2: Generate the fingerprints, compute the similarity scores, determine similarity thresholds, and make histograms.

```
# Write your code here
```