Cheminformatics in R for Analyzing Chemical Genomics Screens

Exercises

Tyler Backman and Thomas Girke

December 10, 2012

Outline

Installation

The R software for running ChemmineR can be downloaded from CRAN (`http://cran.at.r-project.org/`). The ChemmineR package can be installed from R using the `biocLite` install command.

Installing ChemmineR

```
> # Sources the biocLite.R installation script.
> source("http://bioconductor.org/biocLite.R")
> biocLite("ChemmineR") # Installs the package.
```

Installing bioassayR on OSX

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("Biostrings", "DBI", "RSQLite"))
> download.file("http://biocluster.ucr.edu/~tbackman/chem_workshop/bioassayR_0.1
> install.packages("bioassayR_0.1.tar.gz", repos=NULL, type="source")
```

Installing bioassayR on Windows

```
> source("http://bioconductor.org/biocLite.R")
> biocLite(c("Biostrings", "DBI", "RSQLite"))
> download.file("http://biocluster.ucr.edu/~tbackman/chem_workshop/bioassayR_0.1
> install.packages("bioassayR_0.1.zip", repos=NULL)
```

Loading Package and Accessing Documentation

Load Package

```
> library("ChemmineR") # Loads the package
```

Access Manuals

```
> library(help="ChemmineR") # Lists all functions and classes
> vignette("ChemmineR") # Opens this PDF manual from R
```

Five Minute Tutorial

The following code gives an overview of the most important functionalities provided by *ChemmineR*. Copy and paste of the commands into the R console will demonstrate their utilities.

Create Instances of *SDFset* class

```
> data(sdfsample)
> sdfset <- sdfsample
> sdfset # Returns summary of SDFset

An instance of "SDFset" with 100 molecules

> sdfset[1:4] # Subsetting of object

An instance of "SDFset" with 4 molecules
```

Inspecting *SDFset* objects

```
> sdfset[[1]] # Returns summarized content of one SDF
> view(sdfset[1:4]) # Returns summarized content of many SDFs
> as(sdfset[1:4], "list") # Returns complete content of many SDFs
```

Five Minute Tutorial: Accessing an SDFset

An *SDFset* is created during the import of an SD file

```
> sdfset <- read.SDFset("http://faculty.ucr.edu/~tgirke/Documents/
+                       R_BioCond/Samples/sdfsample.sdf")
```

Miscellaneous accessor methods for *SDFset* container

```
> header(sdfset[1:4])
> header(sdfset[[1]])
> atomblock(sdfset[1:4])
> atomblock(sdfset[[1]])[1:4,]
> bondblock(sdfset[1:4])
> bondblock(sdfset[[1]])[1:4,]
> datablock(sdfset[1:4])
> datablock(sdfset[[1]])[1:4]
```

Five Minute Tutorial: Accessing an SDFset

Assigning compound IDs and keeping them unique

```
> cid(sdfset)[1:4] # Returns IDs from SDFset object
> sdfid(sdfset)[1:4] # Returns IDs from SD file header block
> unique_ids <- makeUnique(sdfid(sdfset))
> cid(sdfset) <- unique_ids
```

Converting the data blocks in an *SDFset* to a matrix

```
> blockmatrix <- datablock2ma(datablocklist=datablock(sdfset))
>    # Converts data block to matrix
> numchar <- splitNumChar(blockmatrix=blockmatrix)
>    # Splits to numeric and character matrix
> numchar[[1]][1:2,1:2] # Slice of numeric matrix
> numchar[[2]][1:2,10:11] # Slice of character matrix
```

Compute atom frequency matrix, molecular weight and formula

```
> propma <- data.frame(MF=MF(sdfset), MW=MW(sdfset), atomcountMA(sdfset))
> propma[1:4, ]
```

Five Minute Tutorial

Assign matrix data to data block

```
> datablock(sdfset) <- propma
> datablock(sdfset[1])
```

String searching in *SDFset* ()

```
> grepSDFset("650001", sdfset, field="datablock", mode="subset")
>    # Returns summary view of matches. Not printed here.
> .
> grepSDFset("650001", sdfset, field="datablock", mode="index")
```

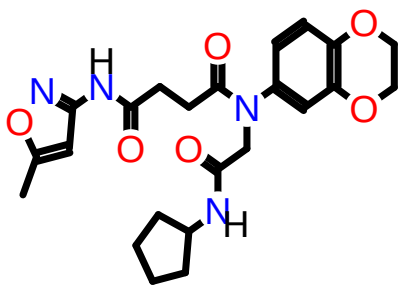
Export SDFset to SD file

```
> write.SDF(sdfset[1:4], file="sub.sdf", sig=TRUE)
```

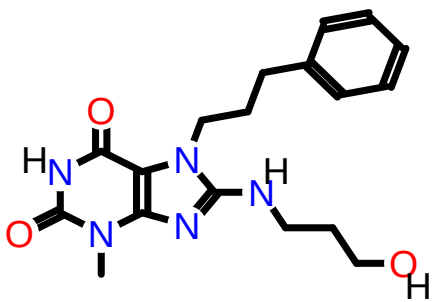
Plot molecule structure of one or many SDFs

```
> plot(sdfset[1:4], print=FALSE) # Plots structures to R graphics device
```

CMP1



CMP2



Five Minute Tutorial

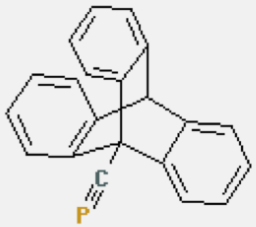
```
> sdf.visualize(sdfset[1:4]) # Compound viewing in web browser
```

View Previously Accessed Compounds >>>

Width of information table: 40 characters

Reference Compound (ka-01834)

View SDF | Structure Search | Add to Selection

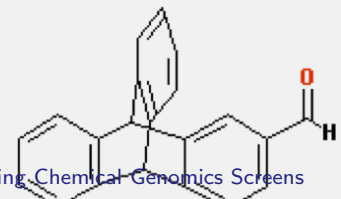


Similarities With All

ids	scores
3	0.550335570
43	0.484662577
42	0.484662577
1	0.484662577
4	0.480122324
2	0.480122324
44	0.356097561
46	0.312500000
11	0.311653117
35	0.287719298

(ChemmineR_Unnamed_Compound_3)

View SDF | Structure Search | Add to Selection



Similarities With All

ids	scores
3	1.000000000
43	0.785977860
42	0.785977860
1	0.785977860
2	0.766423358

Five Minute Tutorial

Structure similarity searching and clustering

```
> apset <- sdf2ap(sdfset)
>    # Generate atom pair descriptor database for searching
> .

> data(apset)
>    # Load sample apset data provided by library.
> cmp.search(apset, apset[1], type=3, cutoff = 0.3, quiet=TRUE)
>    # Search apset database with single compound.
> cmp.cluster(db=apset, cutoff = c(0.65, 0.5), quiet=TRUE)[1:4,]
>    # Binning clustering using variable similarity cutoffs.
```

Outline

Five Minute Tutorial: searching and clustering

Structure similarity searching and clustering

```
> apset <- sdf2ap(sdfset)
>    # Generate atom pair descriptor database for searching
> .

> data(apset)
>    # Load sample apset data provided by library.
> cmp.search(apset, apset[1], type=3, cutoff = 0.3, quiet=TRUE)
>    # Search apset database with single compound.
> cmp.cluster(db=apset, cutoff = c(0.65, 0.5), quiet=TRUE)[1:4,]
>    # Binning clustering using variable similarity cutoffs.
```

Five Minute Tutorial: Molecular Structure Classes and Functions

The following list gives an overview of the most important S4 classes, methods and functions available in the ChemmineR package. The help documents of the package provide much more detailed information on each utility. The standard R help documents for these utilities can be accessed with this syntax: `?function_name` (*e.g.* `?cid`) and `?class_name-class` (*e.g.* `?"SDFset-class"`).

Molecular Structure Classes

- *SDFstr*: intermediate string class to facilitate SD file import; not important for end user
- *SDF*: container for single molecule imported from an SD file
- *SDFset*: container for many SDF objects; most important structure container for end user

Five Minute Tutorial: Molecular Structure Classes and Functions

Accessor methods for *SDF/SDFset*

- Object slots: `cid`, `header`, `atomblock`, `bondblock`, `datablock` (`sdfid`, `datablocktag`)
- Summary of *SDFset*: `view`
- Matrix conversion of data block: `datablock2ma`, `splitNumChar`
- String search in SDFset: `grepSDFset`

Coerce one class to another

- Standard syntax `as(..., "...")` works in most cases. For details see R help with *?"SDFset-class"*.

Utilities

- Atom frequencies: `atomcountMA`, `atomcount`
- Molecular weight: `MW`
- Molecular formula: `MF`
- ...

Compound structure depictions

- R graphics device: `plot`, `plotStruc`
- Online: `cmp.visualize`

Structure Descriptor Data: Classes

Classes

- *AP*: container for atom pair descriptors of a single molecule
- *APset*: container for many AP objects; most important structure descriptor container for end user
- *FP*: container for fingerprint of a single molecule
- *FPset*: container for fingerprints of many molecules, most important structure descriptor container for end user

Structure Descriptor Data: Functions/Methods

Create *AP/APset* instances

- From *SDFset*: `sdf2ap`
- From SD file: `cmp.parse`
- Summary of *AP/APset*: `view, db.explain`

Accessor methods for AP/APset

- Object slots: `ap, cid`

Coerce one class to another

- Standard syntax `as(..., "...")` works in most cases. For details see R help with *?"APset-class"*.

Structure Similarity comparisons and Searching

- Compute pairwise similarities : `cmp.similarity, fpSim`
- Search APset database: `cmp.search, fpSim`

AP-based Structure Similarity Clustering

- Single-linkage binning clustering: `cmp.cluster`
- Visualize clustering result with MDS: `cluster.visualize`
- Size distribution of clusters: `cluster.sizestat`

Outline

Importing Compounds

The following code gives an overview of the most important import/export functionalities provided by *ChemmineR*. The example creates an instance of the *SDFset* class using as sample data set the first 100 compounds from this PubChem SD file (SDF): Compound_00650001_00675000.sdf.gz (`ftp://ftp.ncbi.nih.gov/pubchem/Compound/CURRENT-Full/SDF/`).

SDFs can be imported with the `read.SDFset` function

```
> sdfset <- read.SDFset("http://faculty.ucr.edu/~tgirke/Documents/
+                       R_BioCond/Samples/sdfsample.sdf")

> data(sdfsample) # Loads the same SDFset provided by the library
> sdfset <- sdfsample
> valid <- validSDF(sdfset) # Identifies invalid SDFs in SDFset objects
> sdfset <- sdfset[valid] # Removes invalid SDFs, if there are any
```

Import SD file into *SDFstr* container

```
> sdfstr <- read.SDFstr("http://faculty.ucr.edu/~tgirke/Documents/
+                       R_BioCond/Samples/sdfsample.sdf")
```

Create *SDFset* from *SDFstr* class

```
> sdfstr <- as(sdfset, "SDFstr")
> sdfstr
> as(sdfstr, "SDFset")
```

Outline

Export of Compounds

Write objects of classes *SDFset/SDFstr/SDF* to SD file

```
> write.SDF(sdfset[1:4], file="sub.sdf")
```

Writing customized *SDFset* to file containing *ChemmineR* signature, IDs from *SDFset* and no data block

```
> write.SDF(sdfset[1:4], file="sub.sdf", sig=TRUE, cid=TRUE, db=NULL)
```

Example for injecting a custom matrix/data frame into the data block of an *SDFset* and then writing it to an SD file

```
> props <- data.frame(MF=MF(sdfset), MW=MW(sdfset), atomcountMA(sdfset))
> datablock(sdfset) <- props
> write.SDF(sdfset[1:4], file="sub.sdf", sig=TRUE, cid=TRUE)
```

Indirect export via *SDFstr* object

```
> sdf2str(sdf=sdfset[[1]], sig=TRUE, cid=TRUE)
>    # Uses default components
> sdf2str(sdf=sdfset[[1]], head=letters[1:4], db=NULL)
>    # Uses custom components for header and data block
```

Write *SDF*, *SDFset* or *SDFstr* classes to file

```
> write.SDF(sdfset[1:4], file="sub.sdf", sig=TRUE, cid=TRUE, db=NULL)
> write.SDF(sdfstr[1:4], file="sub.sdf")
> cat(unlist(as(sdfstr[1:4], "list")), file="sub.sdf", sep="\n")
```

Outline

Splitting SD Files

The following `write.SDFsplit` function allows to split SD Files into any number of smaller SD Files. This can become important when working with very big SD Files. Users should note that this function can output many files, thus one should run it in a dedicated directory!

Create sample SD File with 100 molecules

```
> write.SDF(sdfset, "test.sdf")
```

Read in sample SD File. Note: reading file into SDFstr is much faster than into SDFset

```
> sdfstr <- read.SDFstr("test.sdf")
```

Run export on *SDFstr* object. Note: nmol describes molecules per file

```
> write.SDFsplit(x=sdfstr, filetag="myfile", nmol=10)
```

Run export on *SDFset* object

```
> write.SDFsplit(x=sdfset, filetag="myfile", nmol=10)
```

Outline

Streaming Through Large SD Files

The `sdfStream` function allows to stream through SD Files with millions of molecules without consuming much memory. During this process any set of descriptors, supported by *ChemmineR*, can be computed (*e.g.* atom pairs, molecular properties, etc.), as long as they can be returned in tabular format. In addition to descriptor values, the function returns a line index that gives the start and end positions of each molecule in the source SD File. This line index can be used by the downstream `read.SDFindex` function to retrieve specific molecules of interest from the source SD File without reading the entire file into R. The following outlines the typical workflow of this streaming functionality in *ChemmineR*.

Streaming Through Large SD Files

Create sample SD File with 100 molecules

```
> write.SDF(sdfset, "test.sdf")
```

Define descriptor set in a simple function

```
> desc <- function(sdfset) {
+         cbind(SDFID=sdfid(sdfset),
+               # datablock2ma(datablocklist=datablock(sdfset)),
+               MW=MW(sdfset),
+               groups(sdfset),
+               APFP=desc2fp(x=sdf2ap(sdfset), descnames=1024, type="character"),
+               AP=sdf2ap(sdfset, type="character"),
+               rings(sdfset, type="count", upper=6, arom=TRUE)
+         )
+ }
```

Run `sdfStream` with `desc` function and write results to a file called `matrix.xls`

```
> sdfStream(input="test.sdf", output="matrix.xls", fct=desc, Nlines=1000)
> # Nlines: number of lines to read from input SD File at a time
```

Streaming Through Large SD Files

One can also start reading from a specific line number in the SD file. The following example starts at line number 950. This is useful for restarting and debugging the process. With `append=TRUE` the result can be appended to an existing file.

```
> sdfStream(input="test.sdf", output="matrix2.xls", append=FALSE, fct=desc,
+           Nlines=1000, startline=950)
```

Select molecules meeting certain property criteria from SD File using line index generated by previous `sdfStream` step

```
> indexDF <- read.delim("matrix.xls", row.names=1)[,1:4]
> indexDFsub <- indexDF[indexDF$MW < 400, ] # Selects molecules with MW < 400
> sdfset <- read.SDFindex(file="test.sdf", index=indexDFsub, type="SDFset")
> # Collects results in "SDFset" container
```

Write results directly to SD file without storing larger numbers of molecules in memory

```
> read.SDFindex(file="test.sdf", index=indexDFsub, type="file", outfile="sub.sdf
```

Read AP/APFP strings from file into `APset` or `FP` object

```
> apset <- read.AP(x="matrix.xls", type="ap", colid="AP")
> apfp <- read.AP(x="matrix.xls", type="fp", colid="APFP")
```

Alternatively, one can provide the AP/APFP strings in a named character vector

```
> apset <- read.AP(x=sdf2ap(sdfset[1:20], type="character"), type="ap")
> fpchar <- desc2fp(sdf2ap(sdfset[1:20]), descnames=1024, type="character")
> fpset <- as(fpchar, "FPset")
```

Outline

Working with SDF/SDFset Classes

Several methods are available to return the different data components of *SDF/SDFset* containers in batches. The following examples list the most important ones. To save space their content is not printed in the manual.

```
> view(sdfset[1:4]) # Summary view of several molecules
> length(sdfset) # Returns number of molecules
> sdfset[[1]] # Returns single molecule from SDFset as SDF object
> sdfset[[1]][[2]] # Returns atom block from first compound as matrix
> sdfset[[1]][[2]][1:4,]
> c(sdfset[1:4], sdfset[5:8]) # Concatenation of several SDFsets
```

The grepSDFset function allows string matching/searching on the different data components in *SDFset*. By default the function returns a SDF summary of the matching entries. Alternatively, an index of the matches can be returned with the setting *mode="index"*.

```
> grepSDFset("650001", sdfset, field="datablock", mode="subset")
>    # To return index, set mode="index")
> .
```

Utilities to maintain unique compound IDs

```
> sdfid(sdfset[1:4])
>    # Retrieves CMP IDs from Molecule Name field in header block.
> cid(sdfset[1:4])
>    # Retrieves CMP IDs from ID slot in SDFset.
> unique_ids <- makeUnique(sdfid(sdfset))
>    # Creates unique IDs by appending a counter to duplicates.
> cid(sdfset) <- unique_ids # Assigns uniquified IDs to ID slot
```

Working with SDF/SDFset Classes

Subsetting by character, index and logical vectors

```
> view(sdfset[c("650001", "650012")])
> view(sdfset[4:1])
> mylog <- cid(sdfset) %in% c("650001", "650012")
> view(sdfset[mylog])
```

Accessing *SDF/SDFset* components: header, atom, bond and data blocks

```
> atomblock(sdf); sdf[[2]]; sdf[["atomblock"]]
>    # All three methods return the same component
> header(sdfset[1:4])
> atomblock(sdfset[1:4])
> bondblock(sdfset[1:4])
> datablock(sdfset[1:4])
> header(sdfset[[1]])
> atomblock(sdfset[[1]])
> bondblock(sdfset[[1]])
> datablock(sdfset[[1]])
```

Replacement Methods

```
> sdfset[[1]][[2]][1,1] <- 999
> atomblock(sdfset)[1] <- atomblock(sdfset)[2]
> datablock(sdfset)[1] <- datablock(sdfset)[2]
```

Working with SDF/SDFset Classes

Assign matrix data to data block

```
> datablock(sdfset) <- as.matrix(iris[1:100,])
> view(sdfset[1:4])
```

Class coercions from *SDFstr* to *list*, *SDF* and *SDFset*

```
> as(sdfstr[1:2], "list")
> as(sdfstr[[1]], "SDF")
> as(sdfstr[1:2], "SDFset")
```

Class coercions from *SDF* to *SDFstr*, *SDFset*, list with SDF sub-components

```
> sdfcomplist <- as(sdf, "list")
> sdfcomplist <- as(sdfset[1:4], "list"); as(sdfcomplist[[1]], "SDF")
> sdflist <- as(sdfset[1:4], "SDF"); as(sdflist, "SDFset")
> as(sdfset[[1]], "SDFstr")
> as(sdfset[[1]], "SDFset")
```

Class coercions from *SDFset* to lists with components consisting of SDF or sub-components

```
> as(sdfset[1:4], "SDF")
> as(sdfset[1:4], "list")
> as(sdfset[1:4], "SDFstr")
```

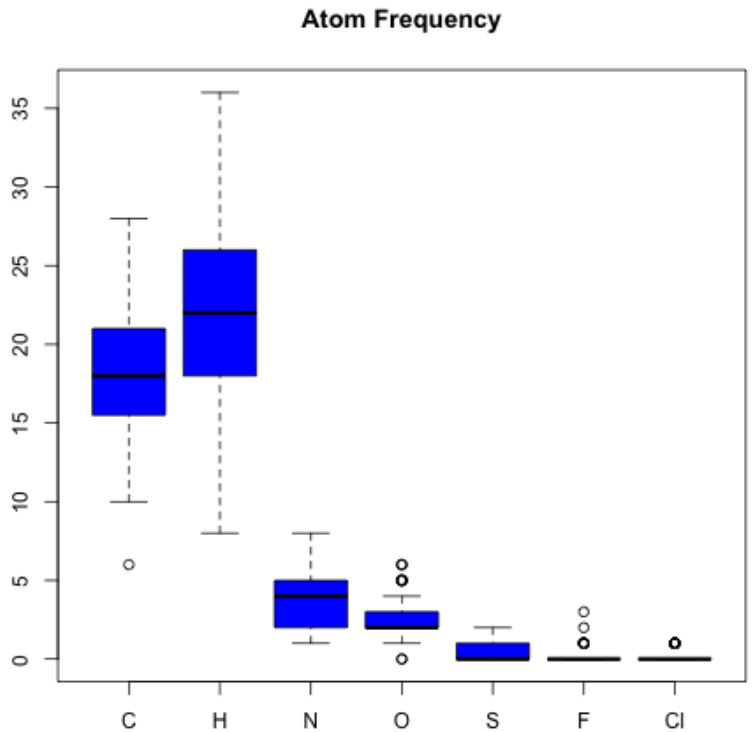
Outline

Molecular Property Functions (Descriptors)

Several methods and functions are available to compute basic compound descriptors, such as molecular formula (MF), molecular weight (MW), and frequencies of atoms and functional groups. In many of these functions, it is important to set `addH=TRUE` in order to include/add hydrogens that are often not specified in an SD file.

```
> propma <- atomcountMA(sdfset, addH=FALSE)
> boxplot(propma, col="blue", main="Atom Frequency")
```

```
null device
          1
```



Molecular Property Functions (Descriptors)

Data frame provided by library containing atom names, atom symbols, standard atomic weights, group and period numbers

```
> data(atomprop)
> atomprop[1:4,]
```

	Number	Name	Symbol	Atomic_weight	Group	Period
1	1	hydrogen	H	1.007940	1	1
2	2	helium	He	4.002602	18	1
3	3	lithium	Li	6.941000	1	2
4	4	beryllium	Be	9.012182	2	2

Compute MW and formula

```
> MW(sdfset[1:4], addH=FALSE)
```

CMP1	CMP2	CMP3	CMP4
456.4916	357.4069	370.4255	461.5346

```
> MF(sdfset[1:4], addH=FALSE)
```

CMP1	CMP2	CMP3	CMP4
"C23H28N4O6"	"C18H23N5O3"	"C18H18N4O3S"	"C21H27N5O5S"

Enumerate functional groups

```
> groups(sdfset[1:4], groups="fctgroup", type="countMA")
```

	RNH2	R2NH	R3N	ROPO3	ROH	RCHO	RCOR	RCOOH	RCOOR	ROR	RCCH	RCN
CMP1	0	2	1	0	0	0	0	0	0	2	0	0
CMP2	0	2	2	0	1	0	0	0	0	0	0	0
CMP3	0	1	1	0	1	0	1	0	0	0	0	0
CMP4	0	1	3	0	0	0	0	0	0	2	0	0

Molecular Property Functions (Descriptors)

Combine MW, MF, charges, atom counts, functional group counts and ring counts in one data frame

```
> propma <- data.frame(MF=MF(sdfset, addH=FALSE), MW=MW(sdfset, addH=FALSE),
+                      Ncharges=sapply(bonds(sdfset, type="charge"), length),
+                      atomcountMA(sdfset, addH=FALSE), groups(sdfset,
+                      type="countMA"), rings(sdfset, upper=6, type="count",
+                      arom=TRUE))
> propma[1:4,]
```

The following shows an example for assigning the values stored in a matrix (*e.g.* property descriptors) to the data block components in an *SDFset*. Each matrix row will be assigned to the corresponding slot position in the *SDFset*.

```
> datablock(sdfset) <- propma # Works with all SDF components
> datablock(sdfset)[1:4]
> test <- apply(propma[1:4,], 1, function(x) data.frame(col=colnames(propma), va
> sdf.visualize(sdfset[1:4], extra = test)
```

Molecular Property Functions (Descriptors)

The data blocks in SDFs contain often important annotation information about compounds. The `datablock2ma` function returns this information as matrix for all compounds stored in an *SDFset* container. The `splitNumChar` function can then be used to organize all numeric columns in a *numeric matrix* and the character columns in a *character matrix* as components of a *list* object.

```
> datablocktag(sdfset, tag="PUBCHEM_NIST_INCHI")
> datablocktag(sdfset, tag="PUBCHEM_OPENEYE_CAN_SMILES")
```

Convert entire data block to matrix

```
> blockmatrix <- datablock2ma(datablocklist=datablock(sdfset))
>    # Converts data block to matrix
> numchar <- splitNumChar(blockmatrix=blockmatrix)
>    # Splits matrix to numeric matrix and character matrix
> numchar[[1]][1:4,]; numchar[[2]][1:4,]
>    # Splits matrix to numeric matrix and character matrix
```

Outline

Bond Matrices

Bond matrices provide an efficient data structure for many basic computations on small molecules. The function `conMA` creates this data structure from *SDF* and *SDFset* objects. The resulting bond matrix contains the atom labels in the row/column titles and the bond types in the data part. The labels are defined as follows: 0 is no connection, 1 is a single bond, 2 is a double bond and 3 is a triple bond.

```
> conMA(sdfset[1:2], exclude=c("H"))
>    # Create bond matrix for first two molecules in sdfset
> conMA(sdfset[[1]], exclude=c("H"))
>    # Return bond matrix for first molecule
> plot(sdfset[1], atomnum = TRUE, noHbonds=FALSE , no_print_atoms = "", atomcex=
>    # Plot its structure with atom numbering
> rowSums(conMA(sdfset[[1]], exclude=c("H")))
>    # Return number of non-H bonds for each atom
```

Outline

Charges and Missing Hydrogens

The function `bonds` returns information about the number of bonds, charges and missing hydrogens in *SDF* and *SDFset* objects. It is used by many other functions (*e.g.* `MW`, `MF`, `atomcount`, `atomcuntMA` and `plot`) to correct for missing hydrogens that are often not specified in SD files.

```
> bonds(sdfset[[1]], type="bonds")[1:4,]
> bonds(sdfset[1:2], type="charge")
> bonds(sdfset[1:2], type="addNH")
```

Outline

Ring Perception and Aromaticity Assignment

The function `rings` identifies all possible rings in one or many molecules (here *sdfset[1]*) using the exhaustive ring perception algorithm f(Hanser1996a). In addition, the function can return all smallest possible rings as well as aromaticity information.

The following example returns all possible rings in a *list*. The argument `upper` allows to specify an upper length limit for rings. Choosing smaller length limits will reduce the search space resulting in shortened compute times. Note: each ring is represented by a character vector of atom symbols that are numbered by their position in the atom block of the corresponding *SDF/SDFset* object.

```
> (ringatoms <- rings(sdfset[1], upper=Inf, type="all", arom=FALSE, inner=FALSE)
```

For visual inspection, the corresponding compound structure can be plotted with the ring bonds highlighted in color

```
> atomindex <- as.numeric(gsub(".*_", "", unique(unlist(ringatoms))))
> plot(sdfset[1], print=FALSE, colbonds=atomindex)
```

Alternatively, one can include the atom numbers in the plot

```
> plot(sdfset[1], print=FALSE, atomnum=TRUE, no_print_atoms="H")
```

Ring Perception and Aromaticity Assignment

Aromaticity information of the rings can be returned in a logical vector by setting `arom=TRUE`

```
> rings(sdfset[1], upper=Inf, type="all", arom=TRUE, inner=FALSE)
```

Return rings with no more than 6 atoms that are also aromatic

```
> rings(sdfset[1], upper=6, type="arom", arom=TRUE, inner=FALSE)
```

Count shortest possible rings and their aromaticity assignments by setting `type=count` and `inner=TRUE`. The inner (smallest possible) rings are identified by first computing all possible rings and then selecting only the inner rings. For more details, consult the help documentation with `?rings`.

```
> rings(sdfset[1:4], upper=Inf, type="count", arom=TRUE, inner=TRUE)
```

Outline

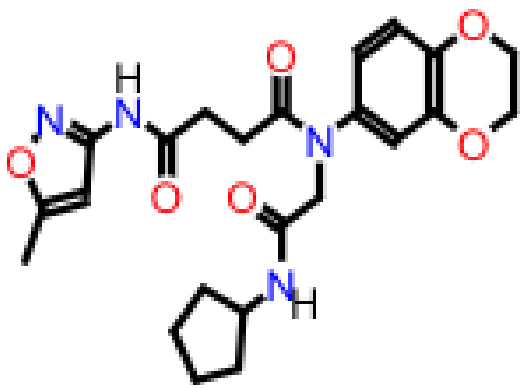
R Graphics Device

A new plotting function for compound structures has been added to the package recently. This function uses the native R graphics device for generating compound depictions. At this point this function is still in an experimental developmental stage but should become stable soon.

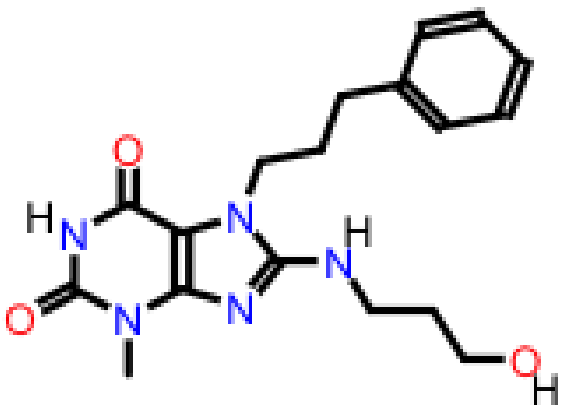
Plot compound Structures with R's graphics device

```
> data(sdfsample)
> sdfset <- sdfsample
> plot(sdfset[1:4], print=FALSE) # "print=TRUE" returns SDF summaries
null device
          1
```

CMP1



CMP2



R Graphics Device

Customized plots

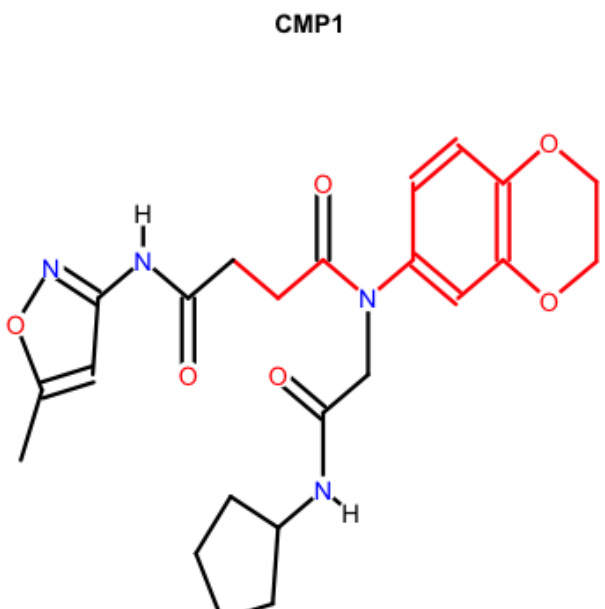
```
> plot(sdfset[1:4], griddim=c(2,2), print_cid=letters[1:4], print=FALSE,
+      noHbonds=FALSE)
```

In the following plot, the atom block position numbers in the SDF are printed next to the atom symbols (`atomnum = TRUE`). For more details, consult help documentation with `?plotStruc` or `?plot`.

```
> plot(sdfset["CMP1"], atomnum = TRUE,  noHbonds=F , no_print_atoms = "",
+      atomcex=0.8, sub=paste("MW:", MW(sdfsample["CMP1"])), print=FALSE)
```

Substructure highlighting by atom numbers

```
> plot(sdfset[1], print=FALSE, colbonds=c(22,26,25,3,28,27,2,23,21,18,8,19,20,24
null device
          1
```



Outline

Maximum Common Substructure (MCS) Searching

The ChemmineR add-on package fmcsR provides support for identifying maximum common substructures (MCSs) and flexible MCSs among compounds. The algorithm can be used for pairwise compound comparisons, structure similarity searching and clustering. The manual describing this functionality is available here. The following gives a short preview of some functionalities provided by the *fmcsR* package.

```
> library(fmcsR)
> data(fmcstest) # Loads test sdfset object
> test <- fmcs(fmcstest[1], fmcstest[2], au=2, bu=1) # Searches for MCS with mis
> plotMCS(test) # Plots both query compounds with MCS in color
```

AP/APset Classes for Storing Atom Pair Descriptors

The function `sdf2ap` computes atom pair descriptors for one or many compounds. It returns a searchable atom pair database stored in a container of class *APset*, which can be used for structural similarity searching and clustering. As similarity measure, the Tanimoto coefficient or related coefficients can be used. An *APset* object consists of one or many *AP* entries each storing the atom pairs of a single compound. Note: the deprecated `cmp.parse` function is still available which also generates atom pair descriptor databases, but directly from an SD file. Since the latter function is less flexible it may be discontinued in the future.

Generate atom pair descriptor database for searching

```
> ap <- sdf2ap(sdfset[[1]]) # For single compound
> ap
> apset <- sdf2ap(sdfset) # For many compounds.
> view(apset[1:4])
```

Return main components of APset objects

```
> cid(apset[1:4]) # Compound IDs
> ap(apset[1:4]) # Atom pair descriptors
> db.explain(apset[1]) # Return atom pairs in human readable format
```

Coerce APset to other objects

```
> apset2descdb(apset) # Returns old list-style AP database
> tmp <- as(apset, "list") # Returns list
> as(tmp, "APset") # Converts list back to APset
```

Large SDF and Atom Pair Databases

When working with large data sets it is often desirable to save the *SDFset* and *APset* containers as binary R objects to files for later use. This way they can be loaded very quickly into a new R session without recreating them every time from scratch.

Save and load of *SDFset* and *APset* containers

```
> save(sdfset, file = "sdfset.rda", compress = TRUE)
> load("sdfset.rda")
> save(apset, file = "apset.rda", compress = TRUE)
> load("apset.rda")
```

Pairwise Compound Comparisons with Atom Pairs

The `cmp.similarity` function computes the atom pair similarity between two compounds using the Tanimoto coefficient as similarity measure. The coefficient is defined as $c/(a+b+c)$, which is the proportion of the atom pairs shared among two compounds divided by their union. The variable c is the number of atom pairs common in both compounds, while a and b are the numbers of their unique atom pairs.

```
> cmp.similarity(apset[1], apset[2])
> cmp.similarity(apset[1], apset[1])
```

Similarity Searching with Atom Pairs

The `cmp.search` function searches an atom pair database for compounds that are similar to a query compound. The following example returns a data frame where the rows are sorted by the Tanimoto similarity score (best to worst). The first column contains the indices of the matching compounds in the database. The argument cutoff can be a similarity cutoff, meaning only compounds with a similarity value larger than this cutoff will be returned; or it can be an integer value restricting how many compounds will be returned. When supplying a cutoff of 0, the function will return the similarity values for every compound in the database.

```
> cmp.search(apset, apset["650065"], type=3, cutoff = 0.3, quiet=TRUE)
```

Alternatively, the function can return the matches in form of an index or a named vector if the `type` argument is set to 1 or 2, respectively.

```
> cmp.search(apset, apset["650065"], type=1, cutoff = 0.3, quiet=TRUE)
> cmp.search(apset, apset["650065"], type=2, cutoff = 0.3, quiet=TRUE)
```

FP/FPset Classes for Storing Fingerprints

The *FPset* class stores fingerprints of small molecules in a matrix-like representation where every molecule is encoded as a fingerprint of the same type and length. The *FPset* container acts as a searchable database that contains the fingerprints of many molecules. The *FP* container holds only one fingerprint. Several constructor and coerce methods are provided to populate *FP/FPset* containers with fingerprints, while supporting any type and length of fingerprints. For instance, the function `desc2fp` generates fingerprints from an atom pair database stored in an *APset*, and `as(matrix, "FPset")` and `as(character, "FPset")` construct an *FPset* database from objects where the fingerprints are represented as *matrix* or *character* objects, respectively.

Show slots of *FPset* class

```
> showClass("FPset")
```

Instance of *FPset* class

```
> data(apset)
> (fpset <- desc2fp(apset))
> view(fpset[1:2])
```

FPset class usage

```
> fpset[1:4] # behaves like a list
> fpset[[1]] # returns FP object
> length(fpset) # number of compounds
> cid(fpset) # returns compound ids
> fpset[10] <- 0 # replacement of 10th fingerprint to all zeros
> cid(fpset) <- 1:length(fpset) # replaces compound ids
> c(fpset[1:4], fpset[11:14]) # concatenation of several FPset objects
```

FP/FPset Classes for Storing Fingerprints

Construct *FPset* class form *matrix*

```
> fpma <- as.matrix(fpset) # coerces FPset to matrix
> as(fpma, "FPset")
```

Construct *FPset* class form *character vector*

```
> fpchar <- as.character(fpset) # coerces FPset to character strings
> as(fpchar, "FPset") # construction of FPset class from character vector
```

Compound similarity searching with *FPset*

```
> fpSim(fpset[1], fpset, method="Tanimoto", cutoff=0.4, top=4)
```

Atom Pair Fingerprints

Atom pairs can be converted into binary atom pair fingerprints of fixed length. Computations on this compact data structure are more time and memory efficient than on their relatively complex atom pair counterparts. The function `desc2fp` generates fingerprints from descriptor vectors of variable length such as atom pairs stored in *APset* or *list* containers. The obtained fingerprints can be used for structure similarity comparisons, searching and clustering.

Create atom pair sample data set

```
> data(sdfsample)
> sdfset <- sdfsample[1:10]
> apset <- sdf2ap(sdfset)
```

Compute atom pair fingerprint database using internal atom pair selection containing the 4096 most common atom pairs identified in DrugBank's compound collection. For details see *?apfp*. The following example uses from this set the 1024 most frequent atom pairs

```
> fpset <- desc2fp(apset, descnames=1024, type="FPset")
```

Alternatively, one can provide any custom atom pair selection. Here, the 1024 most common ones in apset

```
> fpset1024 <- names(rev(sort(table(unlist(as(apset, "list")))))[1:1024])
> fpset <- desc2fp(apset, descnames=fpset1024, type="FPset")
```

Atom Pair Fingerprints

A more compact way of storing fingerprints is as character values

```
> fpchar <- desc2fp(x=apset, descnames=1024, type="character")
> fpchar <- as.character(fpset)
```

Converting a fingerprint database to a matrix and vice versa

```
> fpma <- as.matrix(fpset)
> fpset <- as(fpma, "FPset")
```

Similarity searching and returning Tanimoto similarity coefficients

```
> fpSim(fpset[1], fpset, method="Tanimoto")
```

Atom Pair Fingerprints

Under `method` one can choose from several predefined similarity measures including `Tanimoto` (default), `Euclidean`, `Tversky` or `Dice`. Alternatively, one can pass on custom similarity functions.

```
> fpSim(fpset[1], fpset, method="Tversky", cutoff=0.4, top=4, alpha=0.5, beta=1)
```

Example for using a custom similarity function

```
> myfct <- function(a, b, c, d) c/(a+b+c+d)
> fpSim(fpset[1], fpset, method=myfct)
```

Clustering example

```
> simMAap <- sapply(cid(apfpset), function(x) fpSim(x=apfpset[x], apfpset, sorte
> hc <- hclust(as.dist(1-simMAap), method="single")
> plot(as.dendrogram(hc), edgePar=list(col=4, lwd=2), horiz=TRUE)
```

Pairwise Compound Comparisons with PubChem Fingerprints

The `fpSim` function computes the similarity coefficients (*e.g.* Tanimoto) for pairwise comparisons of binary fingerprints. For this data type, c is the number of "on-bits" common in both compounds, and a and b are the numbers of their unique "on-bits". Currently, the PubChem fingerprints need to be provided (here PubChem's SD files) and cannot be computed from scratch in *ChemmineR*. The PubChem fingerprint specifications can be loaded with `data(pubchemFPencoding)`.

Convert base 64 encoded PubChem fingerprints to *character* vector, *matrix* or *FPset* object

```
> cid(sdfset) <- sdfid(sdfset)
> fpset <- fp2bit(sdfset, type=1)
> fpset <- fp2bit(sdfset, type=2)
> fpset <- fp2bit(sdfset, type=3)
> fpset
```

Pairwise compound structure comparisons

```
> fpSim(fpset[1], fpset[2])
```

Similarly, the `fpSim` function provides search functionality for PubChem fingerprints

```
> fpSim(fpset["650065"], fpset, method="Tanimoto", cutoff=0.6, top=6)
```

Visualize Similarity Search Results

The `cmp.search` function allows to visualize the chemical structures for the search results. Similar but more flexible chemical structure rendering functions are `plot` and `sdf.visualize` described above. By setting the visualize argument in `cmp.search` to `TRUE`, the matching compounds and their scores can be visualized with a standard web browser. Depending on the `visualize.browse` argument, an URL will be printed or a webpage will be opened showing the structures of the matching compounds along with their scores.

View similarity search results in R's graphics device

```
> cid(sdfset) <- cid(apset) # Assure compound name consistency among objects.
> plot(sdfset[names(cmp.search(apset, apset["650065"], type=2, cutoff=4,
+      quiet=TRUE))], print=FALSE)
```

View results online with Chemmine Tools

```
> similarities <- cmp.search(apset, apset[1], type=3, cutoff = 10)
> sdf.visualize(sdfset[similarities[,1]], extra=similarities[,3])
```

Outline

Clustering Identical or Very Similar Compounds

Often it is of interest to identify very similar or identical compounds in a compound set. The `cmp.duplicated` function can be used to quickly identify very similar compounds in atom pair sets, which will be frequently, but not necessarily, identical compounds.

Identify compounds with identical AP sets

```
> cmp.duplicated(apset, type=1)[1:4] # Returns AP duplicates as logical vector
> cmp.duplicated(apset, type=2)[1:4,] # Returns AP duplicates as data frame
```

Plot the structure of two pairs of duplicates

```
> plot(sdfset[c("650059","650060", "650065", "650066")], print=FALSE)
```

Remove AP duplicates from SDFset and APset objects

```
> apdups <- cmp.duplicated(apset, type=1)
> sdfset[which(!apdups)]; apset[which(!apdups)]
```

Alternatively, one can identify duplicates via other descriptor types if they are provided in the data block of an imported SD file. For instance, one can use here fingerprints, InChI, SMILES or other molecular representations. The following examples show how to enumerate by identical InChI strings, SMILES strings and molecular formula, respectively.

```
> count <- table(datablocktag(sdfset, tag="PUBCHEM_NIST_INCHI"))
> count <- table(datablocktag(sdfset, tag="PUBCHEM_OPENEYE_CAN_SMILES"))
> count <- table(datablocktag(sdfset, tag="PUBCHEM_MOLECULAR_FORMULA"))
> count[1:4]
```

Binning Clustering

Compound libraries can be clustered into discrete similarity groups with the binning clustering function `cmp.cluster`. The function accepts as input an atom pair (`APset`) or a fingerprint (`FPset`) descriptor database as well as a similarity threshold. The binning clustering result is returned in form of a data frame. Single linkage is used for cluster joining. The function calculates the required compound-to-compound distance information on the fly, while a memory-intensive distance matrix is only created upon user request via the `save.distances` argument (see below).

Because an optimum similarity threshold is often not known, the `cmp.cluster` function can calculate cluster results for multiple cutoffs in one step with almost the same speed as for a single cutoff. This can be achieved by providing several cutoffs under the cutoff argument. The clustering results for the different cutoffs will be stored in one data frame.

One may force the `cmp.cluster` function to calculate and store the distance matrix by supplying a file name to the `save.distances` argument. The generated distance matrix can be loaded and passed on to many other clustering methods available in R, such as the hierarchical clustering function `hclust` (see below).

If a distance matrix is available, it may also be supplied to `cmp.cluster` via the `use.distances` argument. This is useful when one has a pre-computed distance matrix either from a previous call to `cmp.cluster` or from other distance calculation subroutines.

Binning Clustering

Single-linkage binning clustering with one or multiple cutoffs

```
> clusters <- cmp.cluster(db=apset, cutoff = c(0.7, 0.8, 0.9), quiet = TRUE)
> clusters[1:12,]
```

Clustering of `FPset` objects with multiple cutoffs. This method allows to call various similarity methods provided by the `fpSim` function. For details consult `?fpSim`.

```
> fpset <- desc2fp(apset)
> clusters2 <- cmp.cluster(fpset, cutoff=c(0.5, 0.7, 0.9), method="Tanimoto",
+                          quiet=TRUE)
> clusters2[1:12,]
```

Sames as above, but using Tversky similarity measure

```
> clusters3 <- cmp.cluster(fpset, cutoff=c(0.5, 0.7, 0.9), method="Tversky",
+                          alpha=0.3, beta=0.7, quiet=TRUE)
```

Return cluster size distributions for each cutoff

```
> cluster.sizestat(clusters, cluster.result=1)
> cluster.sizestat(clusters, cluster.result=2)
> cluster.sizestat(clusters, cluster.result=3)
```

Enforce calculation of distance matrix

```
> clusters <- cmp.cluster(db=apset, cutoff = c(0.65, 0.5, 0.3),
+                         save.distances="distmat.rda")
>    # Saves distance matrix to file "distmat.rda" in current working directory.
> load("distmat.rda") # Loads distance matrix.
```

Jarvis-Patrick Clustering

The Jarvis-Patrick clustering algorithm is widely used in cheminformatics. It requires a nearest neighbor table, which consists of `j` nearest neighbors for each item (*e.g.* compound). The nearest neighbor table is then used to join items into clusters that share at least `k` nearest neighbors. The values for `j` and `k` are user-defined parameters. The `jarvisPatrick` function implemented in *ChemmineR* can generate the nearest neighbor table for *APset* and *FPset* objects and then perform Jarvis-Patrick clustering on that table. It also accepts a precomputed nearest neighbor table in form of a *matrix* object. The output is a cluster *vector* with the item labels in the name slot and the cluster IDs in the data slot. Alternatively, the function can return the nearest neighbor matrix. As third parameter the user can set a minimum similarity `cutoff` value for generating the nearest neighbor table. The latter is an optional setting that is not part of the original Jarvis-Patrick algorithm. It allows to generate more tight clusters and to minimize certain limitations of this method, such as false joins of completely unrelated items when operating on small data sets.

Jarvis-Patrick Clustering

Load/create sample *APset* and *FPset*

```
> data(apset)
> fpset <- desc2fp(apset)
```

Standard Jarvis-Patrick clustering on *APset* and *FPset* objects

```
> jarvisPatrick(apset, j=6, k=5) # Using "APset"
> jarvisPatrick(fpset, j=6, k=5) # Using "FPset"
```

Modified Jarvis-Patrick clustering with minimum similarity `cutoff` value (here Tanimoto coefficient)

```
> jarvisPatrick(fpset, j=6, k=2, cutoff=0.6, method="Tanimoto")
```

Output nearest neighbor table (*matrix*)

```
> nnm <- jarvisPatrick(fpset, j=6, k=5, type="matrix")
> nnm[1:4,]
```

Perform clustering on precomputed nearest neighbor table

```
> jarvisPatrick(nnm, j=6, k=5)
```

Multi-Dimensional Scaling (MDS)

To visualize and compare clustering results, the `cluster.visualize` function can be used. The function performs Multi-Dimensional Scaling (MDS) and visualizes the results in form of a scatter plot. It requires as input an *APset*, a clustering result from `cmp.cluster`, and a cutoff for the minimum cluster size to consider in the plot. To help determining a proper cutoff size, the `cluster.sizestat` function is provided to generate cluster size statistics.

Multi-Dimensional Scaling (MDS)

MDS clustering and scatter plot

```
> cluster.visualize(apset, clusters, size.cutoff=2, quiet = TRUE)
>    # Color codes clusters with at least two members.
> cluster.visualize(apset, clusters, quiet = TRUE) # Plots all items.
```

Create a 3D scatter plot of MDS result

```
> library(scatterplot3d)
> coord <- cluster.visualize(apset, clusters, size.cutoff=1, dimensions=3, quiet
> scatterplot3d(coord)
```

Clustering with Other Algorithms

ChemmineR allows the user to take advantage of the wide spectrum of clustering utilities available in R. An example on how to perform hierarchical clustering with the hclust function is given below.

Create atom pair distance matrix

```
> dummy <- cmp.cluster(db=apset, cutoff=0, save.distances="distmat.rda", quiet=T
> load("distmat.rda")
```

Hierarchical clustering with `hclust`

```
> hc <- hclust(as.dist(distmat), method="single")
> hc[["labels"]] <- cid(apset) # Assign correct item labels
> plot(as.dendrogram(hc), edgePar=list(col=4, lwd=2), horiz=T)
```

Instead of atom pairs one can use PubChem's fingerprints for clustering

```
> simMA <- sapply(cid(fpset), function(x) fpSim(fpset[x], fpset, sorted=FALSE))
> hc <- hclust(as.dist(1-simMA), method="single")
> plot(as.dendrogram(hc), edgePar=list(col=4, lwd=2), horiz=TRUE)
```

Plot dendrogram with heatmap (here similarity matrix)

```
> library(gplots)
> heatmap.2(1-distmat, Rowv=as.dendrogram(hc), Colv=as.dendrogram(hc),
+           col=colorpanel(40, "darkblue", "yellow", "white"),
+           density.info="none", trace="none")
```

Outline

Get Compounds from PubChem by Ids

The function `getIds` accepts one or more numeric PubChem compound ids and downloads the corresponding compounds from PubChem Power User Gateway (PUG) returning results in an *SDFset* container. The ChemMine Tools web service is used as an intermediate, to translate queries from plain HTTP POST to a PUG SOAP query.

Fetch 2 compounds from PubChem

```
> compounds <- getIds(c(111,123))
> compounds
```

Search a SMILES Query in PubChem

The function `searchString` accepts one SMILES string (Simplified Molecular Input Line Entry Specification) and performs a $>$0.95 similarity PubChem fingerprint search, returning the hits in an *SDFset* container. The ChemMine Tools web service is used as an intermediate, to translate queries from plain HTTP POST to a PubChem Power User Gateway (PUG) query.

Search a SMILES string on PubChem

```
> compounds <- searchString("CC(=O)OC1=CC=CC=C1C(=O)O")
> compounds
```

Search an SDF Query in PubChem

The function `searchSim` performs a PubChem similarity search just like `searchString`, but accepts a query in an *SDFset* container. If the query contains more than one compound, only the first is searched.

Search an *SDFset* container on PubChem

```
> data(sdfsample); sdfset <- sdfsample[1]
> compounds <- searchSim(sdfset)
> compounds
```

Format Interconversions

The `sdf2smiles` and `smiles2sdf` functions provide format interconversion between SMILES strings (Simplified Molecular Input Line Entry Specification) and *SDFset* containers. Currently these functions are limited to a single compound at a time.

Convert an *SDFset* container to a SMILES *character* string

```
> data(sdfsample); sdfset <- sdfsample[1]
> smiles <- sdf2smiles(sdfset)
> smiles
```

Convert a SMILES *character* string to an *SDFset* container

```
> sdf <- smiles2sdf("CC(=O)OC1=CC=CC=C1C(=O)O\tname")
> view(sdf)
```

These functions require internet connectivity, as they rely on the ChemMine Tools web service for conversion with the Open Babel Open Source Chemistry Toolbox.

Biological Screen Analysis

This example uses an experimental software package, "bioassayR" that has not yet been publically released.
This library is designed for analyzing biological screen data, and comes loaded with all screens that target known proteins from PubChem Bioassay.
Users can access the data directly using SQL commands against the three built in tables (bioassay, proteins, sequences) or with one of the wrapper functions shown.

Outline

Biological Screen Analysis

```
> library(bioassayR)
```

Show the first 10 rows in the bioassay database table using SQL

```
> query(bioassay, "select * from bioassay limit 10")
>  # Activity key: 1=inactive, 2=active, 3=inconclusive, 4=unspecified
```

Show the first 10 rows, joined with their target amino acid sequences using SQL

```
> query(bioassay, "SELECT * FROM bioassay INNER JOIN (SELECT * FROM proteins) US
```

Get a data frame of bioactivity data from a given assay

```
> assay(bioassay, 500171)
```

Output the protein target for a given assay

```
> protein(bioassay, 500171)
```

Get SDF file of all compounds from a given assay (500016 in this example)

```
> library(ChemmineR)
> mySDFset <- getIds(as.numeric(assay(bioassay, 500171)$cid))
```

Summarize data in database (VERY slow)

```
> bioassay
```

Biological Screen Analysis

Drug Discovery example: Given query compound pubchem CID 16016583, which other compounds were screened against the same targets?

Show all assays 3232582 participates in

```
> query(bioassay, "select aid from bioassay where cid = 3232582")
```

Get active assay ids

```
> activeAids <- query(bioassay, "select aid from bioassay where cid = 3232582 an
> activeAids <- as.integer(unlist(activeAids))
```

Get active compounds from these assays

```
> activeCids <- query(bioassay, paste("select distinct cid from bioassay where a
```

Visualize structure of active compounds

```
> mySDFset <- getIds(as.numeric(activeCids$cid))
> plot(mySDFset)
```

Session Information

```
> sessionInfo()

R version 2.15.2 (2012-10-26)
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

locale:
[1] C

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] ChemmineR_2.10.8

loaded via a namespace (and not attached):
[1] RCurl_1.95-3 tools_2.15.2
```