rcdk: Integrating the CDK with R

Rajarshi Guha
Miguel Rojas Cherto

April 21, 2010

Contents

1 Introduction

Given that much of cheminformatics involves mathematical and statistical modeling of chemical information, R is a natural platform for such work. There are many cheminformatics applications that will generate useful information such as descriptors, fingerprints and so on. While one can always run these applications to generate data that is then imported into R, it can be convenient to be able to manipulate chemical structures and generate chemical information with the R environment.

The CDK is a Java library for cheminformatics that supports a wide variety of cheminformatics functionality ranging from reading molecular file formats, performing ring perception and armaticity detection to fingerprint generation and molecular descriptors.

The goal of the *rcdk* package is to allow an R user to access the cheminformatics functionality of the CDK from within R. While one can use the *rJava* package to make direct calls to specific methods in the CDK, from R, such usage does not usually follow common R idioms. The goal of the *rcdk* is to allow users to use the CDK classes and methods in an R-like fashion.

The library is loaded as follows

```
> library(rcdk)
```

To list the documentation for all available packages try

```
> library(help = rcdk)
```

The package also provides an example data set, called `bpdata` which contains 277 molecules, in SMILES format and their associated boiling points (BP) in Kelvin. The data.frame has two columns, viz., the SMILES and the BP. Molecules names are used as row names.

2 Input / Output

Chemical structures come in a variety of formats and the CDK supports many of them. Many such formats are disk based and these files can be parsed and loaded by specifying their full paths

```
> mols <- load.molecules(c("data1.sdf", "/some/path/data2.sdf"))
```

Note that the above function will load any file format that is supported by the CDK, so there's no need to specify formats. In addition one can specify a URL (which should start with "http://") to specify remote files as well. The result of this function is a `list` of molecule objects. The molecule objects are of class `jobjRef` (provided by the *rJava* package). As a result,they are pretty opaque to the user and are really meant to be processed using methods from the *rcdk* or *rJava* packages.

Another common way to obtain molecule objects is by parsing SMILES strings. The simplest way to do this is

```
> smile <- "c1ccccc1CC(=O)C(N)CC1CCCCOC1"
> mol <- parse.smiles(smile)
```

While this usage is correct, it is not particularly efficient. This is because the method must instantiate a parser each time it is called. This takes some time and will also use memory. A better way is to create the parser yourself and then supply it to `parse.smiles`. This makes parsing multiple SMILES much more efficient:

```
> smiles <- c("CCC", "c1ccccc1", "CCCC(C)(C)CC(=O)NC")
> sp <- get.smiles.parser()
> mols <- sapply(smiles, parse.smiles, parser = sp)
```

Given a list of molecule objects, it is possible to serialize them to a file in some specified format. Currently, the only output formats are SMILES or SDF. To write molecules to a disk file in SDF format:

```
> write.molecules(mols, filename = "mymols.sdf")
```

By default, if `mols` is a list of multiple molecules, all of them will be written to a single SDF file. If this is not desired, you can write each on to individual files (which are prefixed by the value of `filename`):

```
> write.molecules(mols, filename = "mymols.sdf", together = FALSE)
```

To generate a SMILES representation of a molecule we can do

```
> get.smiles(mols[[1]])
```

```
[1] "CCC"
```

```
> unlist(lapply(mols, get.smiles))
```

```
                CCC             c1ccccc1   CCCC(C)(C)CC(=O)NC
              "CCC"           "c1ccccc1" "O=C(NC)CC(C)(C)CCC"
```

3 Visualization

Currently the *rcdk* package only supports 2D visualization. This can be used to view the structure of individual molecules or multiple molecules in a tabular format. It is also possible to view a molecular-data table, where one of the columns is the 2D image and the remainder can contain data associated with the molecules.

Unfortunately, due to event handling issues, the depictions will display on OS X, but the Swing window will become unresponsive. As a result, it is not recommended to generate 2D depictions on OS X.

Molecule visualization is performed using the `view.molecule.2d` function. For viewing a single molecule or a list of multiple molecules, it is simply

```
> sp <- get.smiles.parser()
> smiles <- c("CCC", "CCN", "CCN(C)(C)", "c1ccccc1Cc1ccccc1",
+     "C1CCC1CC(CN(C)(C))CC(=O)CC")
> mols <- sapply(smiles, parse.smiles, parser = sp)
> view.molecule.2d(mols[[1]])
> view.molecule.2d(mols)
```

If multiple molecules are provided, they are display in a matrix format, with a default of four columns. This can be changed via the *ncol* argument. Furthermore, the size of the images are 200 $\times$ 200 pixels, by default. But this can be easily changed via the *cellx* and *celly* arguments.

In many cases, it is useful to view a "molecular spreadsheet", which is a table of molecules along with information related to the molecules being viewed. The data is arranged in a spreadsheet like manner, with one of the columns being molecules and the remainder being textual or numeric information.

This can be achieved using the `view.table` method which takes a list of molecule objects and a `data.frame` containing the associated data. As expected, the number of rows in the `data.frame` should equal the length of the molecule list.

```
> dframe <- data.frame(x = runif(4), toxicity = factor(c("Toxic",
+     "Toxic", "Nontoxic", "Nontoxic")), solubility = c("yes",
+     "yes", "no", "yes"))
> view.table(mols[1:4], dframe)
```

As shown, the `view.table` supports numeric, character and factor data types.

4 Manipulating Molecules

In general, given a `jobjRef` for a molecule object one can access all the class and methods of the CDK library via *rJava*. However this can be cumbersome. The *rcdk* package is in the process of exposing methods and classes that manipulate molecules. This section describes them - more will be implemented in future releases.

4.1 Adding Information to Molecules

In many scenarios it's useful to associate information with molecules. Within R, you could always create a `data.frame` and store the molecule objects along with relevant information in it. However, when serializing the molecules, you want to be able to store the associated information.

Using the CDK it's possible to directly add information to a molecule object using properties. Note that adding such properties uses a key-value paradigm, where the key should be of class `character`. The value can be of class `integer`, `double`, `character` or `jobjRef`. Obviously, after setting a property, you can get a property by its key.

```
> mol <- parse.smiles("c1ccccc1", parser = sp)
> set.property(mol, "title", "Molecule 1")
> set.property(mol, "hvyAtomCount", 6)
> get.property(mol, "title")
```

```
[1] "Molecule 1"
```

It is also possible to get all available properties at once in the from of a list. The property names are used as the list names.

```
> get.properties(mol)
```

```
$hvyAtomCount
[1] 6

$title
[1] "Molecule 1"
```

After adding such properties to the molecule, you can write it out to an SD file, so that the property values become SD tags.

```
> write.molecules(mol, "tagged.sdf", write.props = TRUE)
```

4.2 Atoms and Bonds

Probably the most important thing to do is to get the atoms and bonds of a molecule. The code below gets the atoms and bonds as lists of `jobjRef` objects, which can be manipulated using *rJava* or via other methods of this package.

```
> mol <- parse.smiles("c1ccccc1C(Cl)(Br)c1ccccc1")
> atoms <- get.atoms(mol)
> bonds <- get.bonds(mol)
> cat("No. of atoms =", length(atoms), "\n")
```

```
No. of atoms = 15
```

```
> cat("No. of bonds =", length(bonds), "\n")
```

```
No. of bonds = 16
```

Right now, given an atom the *rcdk* package does not offer a lot of methods to operate on it. One must access the CDK directly. In the future more manipulators will be added. Right now, you can get the symbol for each atom

```
> unlist(lapply(atoms, get.symbol))
```

```
 [1] "C"  "C"  "C"  "C"  "C"  "C"  "C"  "Cl" "Br" "C"  "C"  "C"  "C"  "C"
[15] "C"
```

It's also possible to get the 3D (or 2D coordinates) for an atom.

```
> coords <- get.point3d(atoms[[1]])
```

Given this, it's quite easy to get the 3D coordinate matrix for a molecule

```
> coords <- do.call("rbind", lapply(atoms, get.point3d))
```

Once you have the coordinate matrix, a quick way to check whether the molecule is flat is to do

```
> if (any(apply(coords, 2, function(x) length(unique(x))) ==
+     1)) {
+     print("molecule is flat")
+ }
```

This is quite a simplistic check that just looks at whether the X, Y or Z coordinates are constant. To be more rigorous one could evaluate the moments of inertia about the axes.

4.3 Substructure Matching

The CDK library supports substructure searches using SMARTS (or SMILES) patterns. The implementation allows one to check whether a target molecule contains a substructure or not as well as to retrieve the atoms and bonds of the target molecule that match the query substructure. At this point, the *rcdk* only support the former operation - given a query pattern, does it occur or not in a list of target molecules. The `match` method of this package is modeled after the same method in the *base* package. An example of its usage would be to identify molecules that contain a carbon atom that has exactly two bonded neighbors.

```
> mols <- sapply(c("CC(C)(C)C", "c1ccc(Cl)cc1C(=O)O", "CCC(N)(N)CC"),
+     parse.smiles)
> query <- "[#6D2]"
> hits <- match(query, mols)
> print(hits)
```

```
CC(C)(C)C c1ccc(Cl)cc1C(=O)O        CCC(N)(N)CC
    FALSE               TRUE               TRUE
```

5 Molecular Descriptors

Probably the most desired feature when doing predictive modeling of molecular activities is molecular descriptors. The CDK implements a variety of molecular descriptors, categorized into topological, constitutional, geometric, electronic and hybrid. It is possible to evaluate all available descriptors at one go, or evaluate individual descriptors.

First, we can take a look at the available descriptor categories.

```
> dc <- get.desc.categories()
> dc
```

```
[1] "topological"    "geometrical"    "hybrid"         "constitutional"
[5] "protein"        "electronic"
```

Given the categories we can get the names of the descriptors for a single category. Of course, you can always provide the category name directly.

```
> dn <- get.desc.names(dc[1])
```

Each descriptor name is actually a fully qualified Java class name for the corresponding descriptor. These names can be supplied to `eval.desc` to evaluate a single or multiple descriptors for one or more molecules.

```
> aDesc <- eval.desc(mol, dn[1])
> allDescs <- eval.desc(mol, dn)
```

The return value of `eval.desc` is a data.frame with the descriptors in the columns and the molecules in the rows. For the above example we get a single row. But given a list of molecules, we can easily get a descriptor matrix. For example, lets build a linear regression model to predict boiling points for the BP dataset. First we need a set of descriptors and so we evaluate all available descriptors. Also note that since a descriptor might belong to more than one category, we should obtain a unique set of descriptor names

```
> data(bpdata)
> mols <- sapply(bpdata[, 1], parse.smiles, parser = sp)
> descNames <- unique(unlist(sapply(get.desc.categories(),
+     get.desc.names)))
> descs <- eval.desc(mols, descNames)
> class(descs)
```

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

```
> dim(descs)
```

```
[1] 277 289
```

As you can see we get a `data.frame`. Many of the columns will be `NA`. This is because when a descriptor cannot be evaluated (due to some error) it returns NA. In our case, since our molecules have no 3D coordinates many geometric, electronic and hybrid descriptors cannot be evaluated.

Given the ubiquity of certain descriptors, some of them are directly available via their own functions. Specifically, one can calculate TPSA (topological polar surface area), AlogP and XlogP without having to go through `eval.desc`.[1].

```
> mol <- parse.smiles("CC(=O)CC(=O)NCN")
> convert.implicit.to.explicit(mol)
> get.tpsa(mol)
```

[1]Note that AlogP and XlogP assume that hydrogens are explicitly specified in the molecule. This may not be true if the molecules were obtained from SMILES

```
[1] 72.19
```

```
> get.xlogp(mol)
```

```
[1] -0.883
```

```
> get.alogp(mol)
```

```
[1] -1.5983
```

In any case, now that we have a descriptor matrix, we easily build a linear regression model. First, remove NA's, correlated and constant columns. The code is shown below, but since it involves a stochastic element, we will not run it for this example. If we were to perform feature selection, then this type of reduction would have to be performed.

```
> descs <- descs[, !apply(descs, 2, function(x) any(is.na(x)))]
> descs <- descs[, !apply(descs, 2, function(x) length(unique(x)) ==
+     1)]
> r2 <- which(cor(descs)^2 > 0.6, arr.ind = TRUE)
> r2 <- r2[r2[, 1] > r2[, 2], ]
> descs <- descs[, -unique(r2[, 2])]
```

Note that the above correlation reduction step is pretty crude and there are better ways to do it. Given the reduced descriptor matrix, we can perform feature selection (say using *leaps* or a GA) to identify a suitable subset of descriptors. For now, we'll select some descriptors that we know are correlated to BP. The fit is shown in Figure 1 which plots the observed versus predicted BP's.

```
> model <- lm(BP ~ khs.sCH3 + khs.sF + apol + nHBDon, data.frame(bpdata,
+     descs))
> summary(model)
```

```
Call:
lm(formula = BP ~ khs.sCH3 + khs.sF + apol + nHBDon, data = data.frame(bpdata,
    descs))

Residuals:
    Min      1Q  Median      3Q     Max
-94.395 -20.911  -1.168  19.574 114.237

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 267.3135     6.0006  44.548   <2e-16 ***
khs.sCH3    -22.7948     2.0676 -11.025   <2e-16 ***
khs.sF      -24.4121     2.6548  -9.196   <2e-16 ***
apol          8.6211     0.3132  27.523   <2e-16 ***
nHBDon       47.1187     3.7061  12.714   <2e-16 ***
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

Residual standard error: 34.08 on 272 degrees of freedom
Multiple R-squared: 0.837,        Adjusted R-squared: 0.8346
F-statistic: 349.1 on 4 and 272 DF,  p-value: < 2.2e-16
```

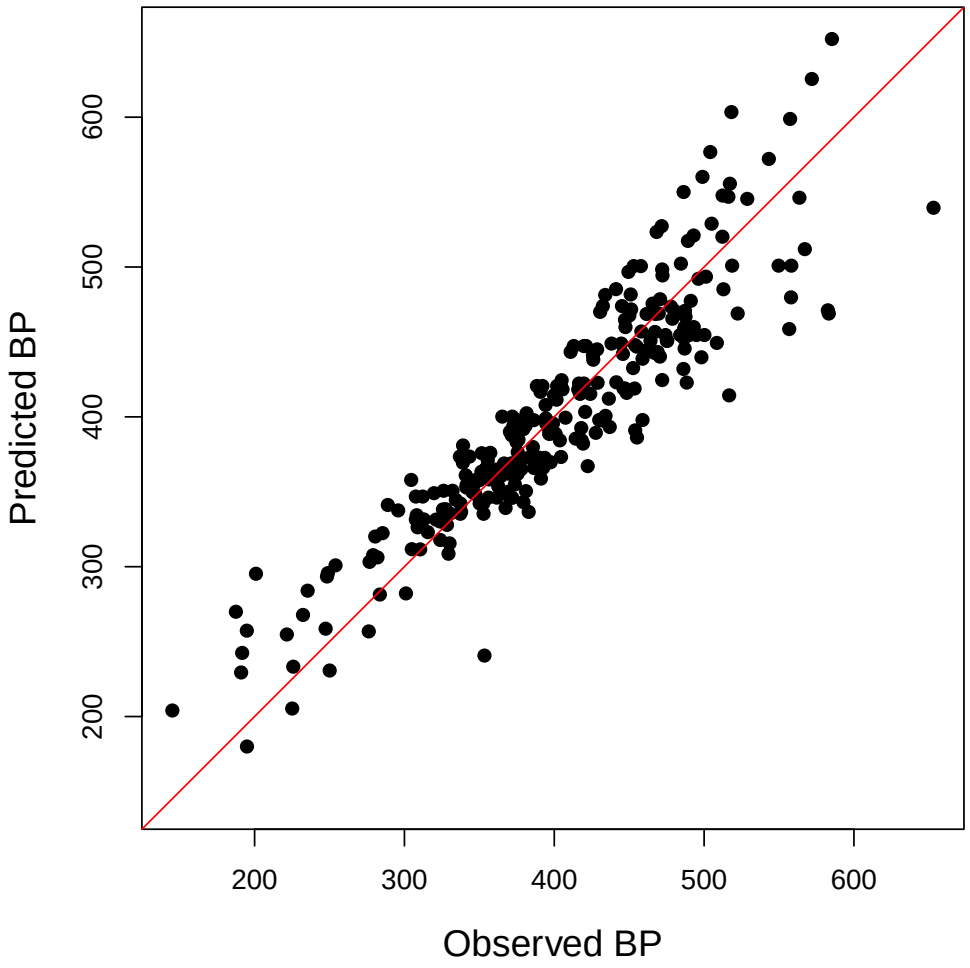

Figure 1: A plot of observed versus predicted boiling points, obtained from a linear regression model using 277 molecules.

6 Fingerprints

Fingerprints are a common representation used for a variety of purposes such as similarity searching and predictive modeling. The CDK provides four types of fingerprints, viz.,

- Standard - a path based, hashed fingerprint. The default size is 1024 bits, but this can be changed via an argument
- Extended - similar to the Standard form, but takes into account ring systems. Default size is 1024 bits
- EState - a structural key type fingerprint that checks for the presence or absence of 79 EState substructures. Length of the fingerprint is 79 bits
- MACCS - the well known 166 bit structural keys

When using *rcdk* to evaluate fingerprints, you will need the *fingerprint* package. Since this is a dependency of the *rcdk* package, it should have been automatically installed.

To generate the fingerprints, we must first obtain molecule objects. Thus for example,

```
> sp <- get.smiles.parser()
> smiles <- c("CCC", "CCN", "CCN(C)(C)", "c1ccccc1Cc1ccccc1",
+     "C1CCC1CC(CN(C)(C))CC(=O)CC")
> mols <- sapply(smiles, parse.smiles, parser = sp)
> fp <- get.fingerprint(mols[[1]], type = "maccs")
```

The variable, `fp`, will be of class *fingerprint* and can be manipulated using the methods provided by the package of the same name. A simple example is to perform a hierarchical clustering of the first 50 structures in the BP dataset.

```
> mols <- sapply(bpdata[1:50, 1], parse.smiles, parser = sp)
> fps <- lapply(mols, get.fingerprint, type = "extended")
> fp.sim <- fp.sim.matrix(fps, method = "tanimoto")
> fp.dist <- 1 - fp.sim
```

Once we have the distance matrix (which we must derive from the Tanimoto similarity matrix), we can then perform the clustering and visualize it.

Another common task for fingerprints is similarity searching. In other words, given a collection of "target" molecules, find those molecules that are similar to a "query" molecule. This is achieved by evaluating a similarity metric between the query and each of the target molecules. Those target molecules exceeding a user defined cutoff will be returned. With the help of the *fingerprint* package this is easily accomplished.

For example, we can identify all the molecules in the BP dataset that have a Tanimoto similarity of 0.3 or more with acetalehyde, and then create a summary in Table 1. Note that this could also be accomplished with molecular descriptors, in which case you'd probably evaluate the Euclidean distance between descriptor vectors.

```
> query.mol <- parse.smiles("CC(=O)", parser = sp)
> target.mols <- sapply(bpdata[, 1], parse.smiles, parser = sp)
> query.fp <- get.fingerprint(query.mol, type = "maccs")
> target.fps <- lapply(target.mols, get.fingerprint, type = "maccs")
> sims <- unlist(lapply(target.fps, distance, fp2 = query.fp,
+     method = "tanimoto"))
> hits <- which(sims > 0.3)
```

	SMILES	Similarity
1	O=C	0.50
7	CCC=O	0.50
9	CC(C)C=O	0.50
3	CC(=O)Cl	0.43
6	CC(=O)C	0.43
4	CC(=O)O	0.38
5	COC=O	0.38
10	CC(C)C(=O)C	0.38
11	CC(C)C(=O)C(C)C	0.38
2	CO	0.33
8	C(=O)CCC	0.33

Table 1: Summary of molecules in the BP dataset that are greater than 0.3 similar to acetaldehyde

7 Handling Molecular Formulae

The molecular formula is the simplest way to characterize a molecular compound. It specifies the actual number of atoms of each element contained in the molecule. A molecular formula is represented by the chemical symbol of each constituent element. If a molecule contains more than one atom for a particular element, the quantity is shown as subscript after the chemical symbol. Otherwise, the number of neutrons (atomic mass) that an atom is composed can differ. This different type of atoms are known as isotopes. The number of nucleos is denoted as superscripted prefix previous to the chemical element. Generally it is not added when the isotope that characterizes the element is the most occurrence in nature. E.g. $C_4H_{11}O^2D$.

7.1 Parsing a Molecule To a Molecular Formula

Front a molecule, defined as conjunct of atoms helding together by chemical bonds, we can simplify it taking only the information about the atoms. *rcdk* package provides a parser to translate molecules to molecular formlulas, the `get.mol2formula` function.

```
> sp <- get.smiles.parser()
> molecule <- sapply("N", parse.smiles, parser = sp)[[1]]
> convert.implicit.to.explicit(molecule)
> formula <- get.mol2formula(molecule, charge = 0)
```

Note that the above formula object is a CDKFormula-class. A cdkFormula-class contains some attributes that defines a molecular formula. For example, the mass, the charge, the isotopes, the character representation of the molecular formula and the IMolecularFormula `jobjRef` object.

The molecular mass for this formula.

```
> formula@mass
```

```
[1] 17.02655
```

The charge for this formula.

```
> formula@charge
```

```
[1] 0
```

The isotopes for this formula. It is formed for three columns. `isoto` (the symbol expression of the isotope), `number` (number of atoms for this isotope) and `mass` (exact mass of this isotope).

```
> formula@isotopes
```

```
     isoto number mass
[1,] "H"   "3"    "1.007825032"
[2,] "N"   "1"    "14.003074"
```

The `jobjRef` object from the `IMolecularFormula` java class in CDK.

```
> formula@objectJ
```

```
[1] "Java-Object{org.openscience.cdk.formula.MolecularFormula@f660c1}"
```

The symbol expression of the molecular formula.

```
> formula@string
```

```
[1] "H3N"
```

Depending of the circumstances, you may want to change the charge of the molecular formula.

```
> formula <- set.charge.formula(formula, charge = 1)
```

7.2 Initializing a Formula from the Symbol Expression

Other way to create a `cdkFormula` is from the symbol expression. Thus, setting the characters of the elemental formula, the function `get.formula` parses it to an object of cdkFormula-class.

```
> formula <- get.formula("NH4", charge = 1)
> formula
```

```
cdkFormula:  H4N , mass =  18.03383 , charge =  1
```

7.3 Generating Molecular Formula

Mass spectrometry is an essential and reliable technique to determine the molecular mass of compounds. Conversely, one can use the measured mass to identify the compound via its elemental formula. One of the limitations of the method is the precision and accuracy of the instrumentation. As a result, rather than specify exact masses, we specify tolerances or ranges of possible mass, resulting in multiple candidate formulae for a given *mass window*. The `generate.formula` function returns a list formulas which have a given mass (within an error window):

```
> mfSet <- generate.formula(18.03383, window = 1, elements = list(c("C",
+     0, 50), c("H", 0, 50), c("N", 0, 50)), validation = FALSE)
> for (i in 1:length(mfSet)) {
+     print(mfSet[i])
+ }
```

```
[[1]]
cdkFormula:  H4N , mass =  18.03437 , charge =  0

[[1]]
cdkFormula:  CH6 , mass =  18.04695 , charge =  0

[[1]]
cdkFormula:  H18 , mass =  18.14085 , charge =  0
```

Important to know is if an elemental formula is valid. The method `isvalid.formula` provides this function. Two constraints can be applied, the nitrogen rule and the (Ring Double Bond Equivalent) RDBE rule.

```
> formula <- get.formula("NH4", charge = 0)
> isvalid.formula(formula, rule = c("nitrogen", "RDBE"))
```

```
[1] FALSE
```

We can observe that the ammonium is only valid if it is defined with charge of +1.

7.4 Calculating Isotope Pattern

Due to the measurement errors in medium resolution spectrometry, a given error window can result in a massive number of candidate formulae. The isotope pattern of ions obtained experimentally can be compared with the theoretical ones. The best match is reflected as the most probable elemental formula. *rcdk* provides the function `get.isotopes.pattern` which predicts the theoretical isotope pattern given a formula.

```
> formula <- get.formula("CHCl3", charge = 0)
> isotopes <- get.isotopes.pattern(formula, minAbund = 0.1)
> isotopes
```

```
         mass     abund
[1,] 117.9144 1.0000000
[2,] 119.9114 0.9588282
[3,] 121.9085 0.3064505
```

In this example we generate a formula for a possible compound with a charge ($z =\approx 0$) containing the standard elements C, H, and Cl. The isotope pattern can be visually inspectd, as shown in Figure 3.

```
> clustering <- hclust(as.dist(fp.dist))
> plot(clustering, main = "A Clustering of the BP dataset")
```

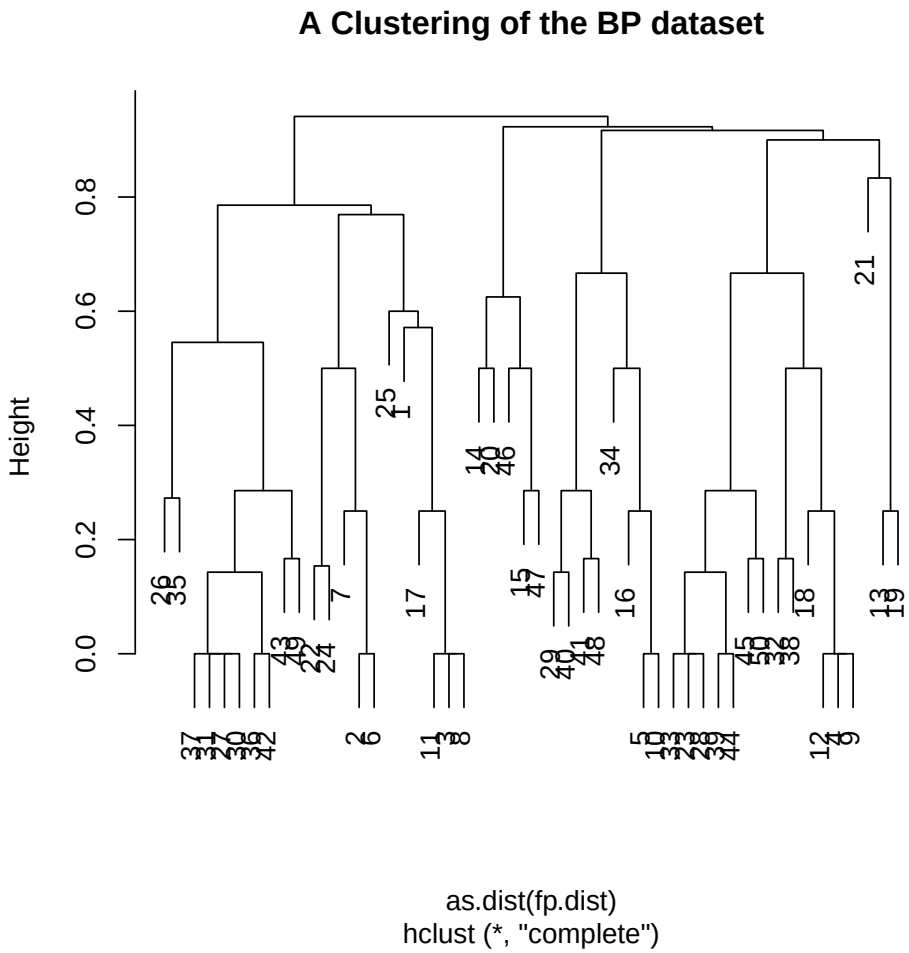

Figure 2: A clustering of the first 50 molecules of the BP dataset, using the CDK extended fingerprints.

```
> plot(isotopes, type = "h", xlab = "m/z", ylab = "Intensity")
```

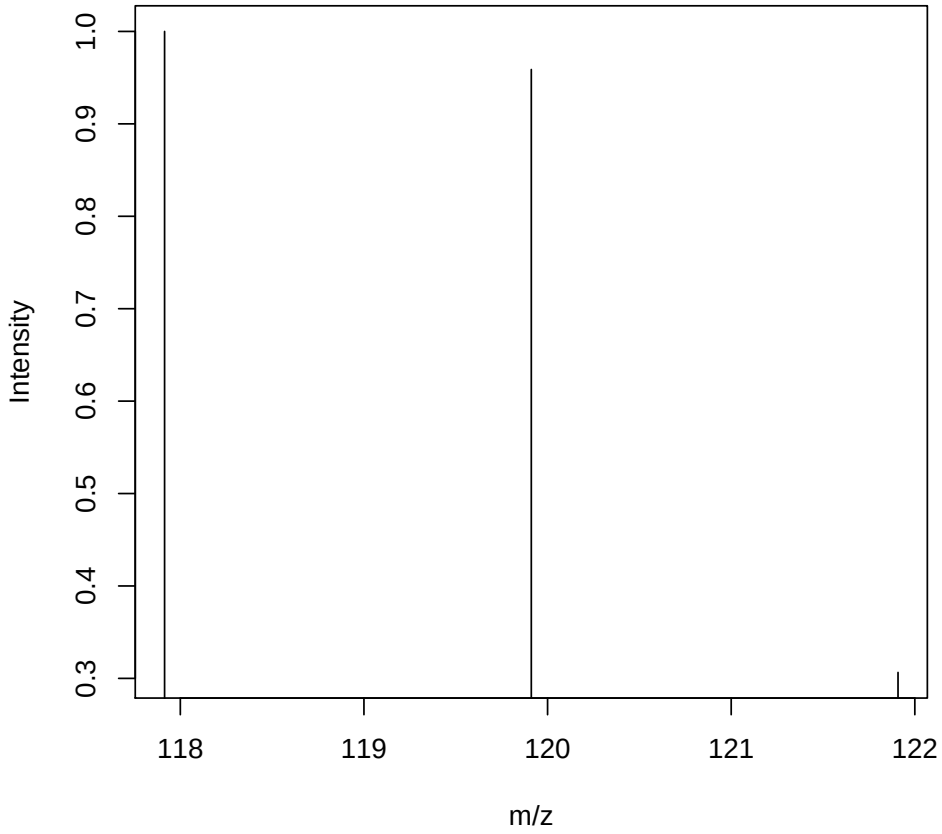

Figure 3: Theoretical isotope pattern given a molecular formula.