patRoon handbook

Rick Helmus

2025-05-22

Contents

1 Introduction

Nowadays there are various software tools available to process data from non-target analysis (NTA) experiments. Individual tools such as ProteoWizard, XCMS, OpenMS, MetFrag and mass spectrometry vendor tools are often combined to perform a complete data processing workflow. During this workflow, raw data files may undergo pre-treatment (e.g. conversion), chromatographic and mass spectral data are combined to extract so called *features* (or 'peaks') and finally annotation is performed to elucidate chemical identities. The aim of `patRoon` is to harmonize the many available tools in order to provide a consistent user interface without the need to know all the details of each individual software tool and remove the need for tedious conversion of data when multiple tools are used. The name is derived from a Dutch word that means *pattern* and may also be an acronym for *hyPhenated mAss specTROmetry nOn-target aNalysis*. The workflow of non-target analysis is typically highly dependent on several factors such as the analytical instrumentation used

and requirements of the study. For this reason, `patRoon` does not enforce a certain workflow. Instead, most workflow steps are optional, are highly configurable and algorithms can easily be mixed or even combined. Furthermore, `patRoon` supplies a straightforward interface to easily inspect, select, visualize and report all data that is generated during the workflow.

The documentation of `patRoon` consists of three parts:

1. A tutorial (accessible at here)
2. This handbook
3. The reference manual (accessible in `R` with `` ?`patRoon-package` `` or online here)

New users are highly recommended to start with the tutorial: this document provides an interactive introduction in performing a basic NTA processing workflow with `patRoon`. The handbook provides a more thorough overview of all concepts, functionalities and provides instructions and many examples on working with `patRoon`. Finally, the reference manual provides all the gritty details for all functionalities, and is meant if you want to know more details or need a quick reminder how a function should be used.

2 Installation

This chapter outlines several strategies to install `patRoon` and its dependencies. These include other `R` packages and software tools external to `R`. The following strategies can largely automate this process, and will be discussed in the next sections:

1. The patRoon bundle, which contains all dependencies (including `R`), and is therefore very easy to setup (currently *Windows only*).
2. Reproducible Docker images.
3. Regular installations that integrate with the currently installed `R` environment.

The first strategy is recommended if you are using Windows and are new to `R`, or quickly want to try out the latest `patRoon` snapshot. Docker images are specifically for users who wish to run isolated containers and ensure high reproducibility. Finally, people already running `R` will most likely prefer the third strategy. Each strategy is discussed separately in the next sections.

2.1 patRoon Bundle

The `patRoon` bundle contains an almost full `patRoon` installation, including `R`, all `R` package dependencies and external software dependencies such as Java JDK, MetFrag and various compound libraries etc. Currently, only ProteoWizard may need to be installed manually.

The bundles are automatically generated and tested, and can be obtained from the release page on GitHub for released versions of `patRoon` and the latest pre-release on GitHub for the latest snapshot.

After downloading the bundle, simply extract the `.zip` file. Then, a classic `R` terminal can be launched by executing `R/bin/x64/Rgui.exe` inside the directory where the bundle was extracted. However, it is probably more convenient to use it from RStudio:

Start RStudio –> Tools menu –> Global options –> General tab –> R version –> Change

Then, set the `R` version by selecting `Rterm.exe` from the `R/bin/x64` directory in the bundle (see screenshot below) and restart RStudio.

2.1.1 Updating the bundle

To update the bundle run either of the following functions:

```
patRoonInst::sync(allDeps = TRUE) # synchronize all packages related to patRoon to the
↪  currently tested versions
patRoonInst::update() # update all R packages related to patRoon
```

Both functions will update `patRoon` and related packages to their latest versions. However, they differ on handling their dependencies.

In general, it is recommended to synchronize the package dependencies in the bundle, since this ensures that versions were tested with `patRoon`. If you installed any other packages and also want to update these, then *first* do so with regular mechanisms (e.g. `update.packages()`, `BiocManager::install()`) and *then* synchronize `patRoon` to ensure that all packages are with tested versions.

However, if you prefer to install the latest version of all dependencies, then running `patRoon::update()` might be more appropriate. In this case, it is still recommended to first update any 'regular' `R` packages as described above, as `patRoonInst::update()` may install some dependencies with a specific version in case other versions are known to not work.

More details on using `patRoonInst` to manage installations are discussed later.

2.1.2 Details

This section describes details on the contents and the configuration of the `patRoon` bundle, and is mainly intended for readers who want to know more details or perform customizations.

The `patRoon` bundle consists of the following:

- A complete installation of `R`.
- An open java development kit (JDK) from Adoptium
- `patRoon` and its mandatory and optional `R` packages dependencies, synchronized from `patRoonDeps` (discussed later).
- Most external dependencies *via* patRoonExt (also discussed later)

The `R` Windows installers are extracted with innoextract to obtain a 'portable' installation. The `Renviron.site` and `Rprofile.site` files are then generated to ensure that the bundled JDK will be used, `R` packages will be loaded and installed from the bundle and various other configurations are applied to ensure that the bundle will not conflict with a regular `R` installation.

The bundles are automatically generated, and the relevant script can be found here.

2.2 Docker image

Docker images are provided to easily install a reproducible environment with `R`, `patRoon` and nearly all of its dependencies. This section assumes you have a basic understanding of Docker and have it installed. If not, please refer to the many guides available on the Internet. The Docker images of `patRoon` were originally only used for automated testing, however, since these contain a complete working environment of `patRoon` they are also suitable for using the software. They come with all external dependencies (except ProteoWizard), `R` dependencies and `MetFrag` libraries. Furthermore, the Docker image also contains RStudio server, which makes using `patRoon` even easier.

Below are some example shell commands on how to run the image.

```
# run an interactive R console session
docker run --rm -it uva-hva.gitlab.host:4567/r.helmus/patroon/patroonrs

# run a linux shell, from which R can be launched
docker run --rm -it uva-hva.gitlab.host:4567/r.helmus/patroon/patroonrs bash

# run rstudio server, accessible from localhost:8787
# login with rstudio/yourpasswordhere
docker run --rm -p 8787:8787 -u 0 -e PASSWORD=yourpasswordhere
↪  uva-hva.gitlab.host:4567/r.helmus/patroon/patroonrs /init

# same as above, but mount a local directory (~/myvolume) as local volume so it can be
↪  used for persistent storage
# please ensure that ~/myvolume exists!
docker run --rm -p 8787:8787 -u 0 -e PASSWORD=yourpasswordhere -v
↪  ~/myvolume:/home/rstudio/myvolume uva-hva.gitlab.host:4567/r.helmus/patroon/patroonrs
↪  /init
```

Note that the first two commands run as the default user `rstudio`, while the last two as `root`. The last commands launch RStudio server. You can access it by browsing to `localhost:8787` and logging in with user `rstudio` and the password defined by the `PASSWORD` variable from the command (`yourpasswordhere` in the above example). The last command also links a local volume in order to obtain persistence of files in the container's home directory. The Docker image is based on the excellent work from the rocker project. For more information on RStudio related options see their documentation for the RStudio image.

2.3 Regular R installation

A 'regular' installation involves installing `patRoon` and its dependencies using the local installation of `R`. This section outlines available tools to do this mostly automatically using the auxiliary patRoonInst and patRoonExt `R` packages, as well as instructions to perform the complete installation manually.

> ***NOTE*** It is highly recommended to perform installation steps in a 'clean' `R` session to avoid errors when installing or upgrading packages. As such it is recommended to close all open (R Studio) sessions and open a plain R console to perform the installation.

2.3.1 Automatic installation

The patRoonInst auxiliary package simplifies the installation process. This package automatically installs all `R` package dependencies, including those unavailable from regular repositories such as CRAN and Bioc-Conductor. Furthermore, `patRoonInst` installs patRoonExt, an `R` package that bundles most common dependencies external to the `R` environment (e.g. MetFrag, OpenMS etc).

The first step is to install `patRoonInst`:

```
install.packages("patRoonInst", repos = c('https://rickhelmus.r-universe.dev',
↪  'https://cloud.r-project.org'))

# or alternatively, from GitHub
install.packages("remotes") # run this in case the remotes (or devtools) package is not
↪  yet installed
remotes::install_github("rickhelmus/patRoonInst")
```

Then to perform an installation or update:

```
patRoonInst::install() # install patRoon and any missing dependencies
patRoonInst::update() # update patRoon and its dependencies
```

The installation can be customized in various ways. Firstly, the repositories used to download R packages can be customized through the `origin` argument. The following options are currently available:

- `patRoonDeps`: contains `patRoon` and its dependencies (including *their* dependencies) with versions that were tested against the latest `patRoon` version. This repository is used for the patRoon bundle, and only available for Windows systems.
- r-universe: contains a snapshot of the latest version of `patRoon` and its direct dependencies.
- "regular": in this case packages will be sourced directly from CRAN/BioConductor or GitHub. This means that suitable build tools (e.g. Rtools on Windows) need to be available during installation.

The default on Windows systems is `patRoonDeps`, and r-universe otherwise. Note that both repositories only provide packages for recent `R` versions.

Other installation customizations include which packages will be installed (or updated), and installing all packages to an isolated `R` library. Some examples:

```
# install from r-universe
patRoonInst::install(origin = "runiverse")
# only install patRoon, without optional dependencies and directly from GitHub
patRoonInst::install(origin = "regular", pkgs = "patRoon")
# full installation, except two selected packages
patRoonInst::install(ignorePkgs = c("nontarget", "MetaClean"))
# full installation, but exclude 'big' optional dependencies such as example data
↪  (patRoonData)
patRoonInst::install(ignorePkgs = "big")
# install everything to an isolated R library (use .libPaths() to use it)
patRoonInst::install(lib.loc = "~/patRoon-lib")
```

Besides installing and updating packages, it is also possible to *synchronize* them with the selected repository using the `sync()` function. This is mostly the same as `update()`, but can also downgrade packages to ensure their versions exactly match that of the repository. This is currently only supported for the `patRoonDeps` repository. Furthermore, as synchronization may involve downgrading it is intended for environments that are primarily used for `patRoon`, such as the bundle and isolated `R` libraries. Synchronization can be performed for all or only direct dependencies:

```
patRoonInst::sync(allDeps = TRUE) # synchronize all dependencies
patRoonInst::sync(allDeps = FALSE) # synchronize only direct dependencies
```

More options are available to customize the installation, see the reference manual (`?patRoonInst::install`) for more details.

2.3.2 Manual installation

A manual installation starts with installing external dependencies, followed by `R` dependencies and `patRoon` itself.

2.3.2.1 External (non-R) dependencies

`patRoon` interfaces with various software tools that are external to `R`. A complete overview is given in the table below

Dependency	Remarks
Java JDK	**Mandatory** for e.g. plotting structures and using MetFrag.
OpenBabel	**Highly recommend** Used by e.g. suspect screening to automatically validate and calculate chemical properties such as InChIs and formulae. While optional, highly recommended.
Rtools	May be necessary on Window when installing `patRoon` and its `R` dependencies (discussed later).
ProteoWizard	Needed for automatic data-pretreatment (e.g. data file conversion and centroiding, Bruker users may use DataAnalysis integration instead).
OpenMS	Recommended. Used for e.g. finding and grouping features.
MetFrag CL	Recommended. Used for annotation with MetFrag.
MetFrag CompTox DB	Database files necessary for usage of the CompTox database with MetFrag. Note that a recent version of MetFrag (>=2.4.5) is required. Note that the lists with additions for smoking metadata and wastewater metadata are also supported.
MetFrag PubChemLite DB	Database file needed to use PubChemLite with MetFrag.
MetFrag PubChem OECD PFAS DB	Database file to use the OECD PFAS database with MetFrag.
SIRIUS	For obtaining feature data and formula and/or compound annotation.
BioTransformer	For prediction of transformation products. See the BioTransformer page for installation details. If you have trouble compiling the jar file you can download it from here.
SAFD	For finding features with SAFD. Please follow all the installation on the SAFD webpage.
pngquant	Used to reduce size of HTML reports (only legacy interface), definitely optional.

Most of these dependencies are optional and only needed if their algorithms are used during the workflow.

2.3.2.1.1 Installation via `patRoonExt`

The patRoonExt auxiliary package automatizes the installation of most common external dependencies. For installation, just run:

```
install.packages("remotes") # run this if remotes (or devtools) is not already installed
remotes::install_github("rickhelmus/patRoonExt")
```

> ***NOTE*** Make sure you have an active internet connection since several files will be downloaded during the installation of `patRoonExt`.

Note that when you do an automated patRoon installation this package is automatically installed. See the project page for more details, including ways to customize which software tools will be installed.

> ***NOTE*** Currently, `patRoonExt` does not install ProteoWizard due to license restrictions, and some tools, such as OpenMS and OpenBabel, are only installed on Windows systems. See the next section to install any missing tools manually.

2.3.2.1.2 Manually installing and configuring external tools Download the tools manually from the linked sources shown in the table above, and subsequently install (or extract) them. You may need to configure their file paths afterwards (`OpenMS`, `OpenBabel` and `ProteoWizard` are often found automatically). To configure the file locations you should set some global package options with the `options()` `R` function, for instance:

```
options(patRoon.path.pwiz = "C:/ProteoWizard") # location of ProteoWizard installation
↪  folder
options(patRoon.path.SIRIUS = "C:/sirius-win64-3.5.1") # directory with the SIRIUS
↪  binaries
options(patRoon.path.OpenMS = "/usr/local/bin") # directory with the OpenMS binaries
options(patRoon.path.pngquant = "~/pngquant") # directory containing pngquant binary
options(patRoon.path.MetFragCL = "~/MetFragCommandLine-2.4.8.jar") # full location to the
↪  jar file
options(patRoon.path.MetFragCompTox = "C:/CompTox_17March2019_SelectMetaData.csv") # full
↪  location to desired CompTox CSV file
options(patRoon.path.MetFragPubChemLite = "~/PubChemLite_exposomics_20220429.csv") # full
↪  location to desired PubChemLite CSV file
options(patRoon.path.MetFragPubChemLite = "~/PubChem_OECDPFAS_largerPFASparts_20220324")
↪  # full location to PFAS DB (NOTE: configured like PubChemLite)
options(patRoon.path.BioTransformer = "~/biotransformer/biotransformer-3.0.0.jar")
options(patRoon.path.obabel = "C:/Program Files/OpenBabel-3.0.0") # directory with
↪  OpenBabel binaries
```

These commands have to be executed every time you start a new `R` session (e.g. as part of your script). However, it is probably easier to add them to your `~/.Rprofile` file so that they are executed automatically when you start `R`. If you don't have this file yet you can simply create it yourself (for more information see e.g. this SO answer).

> ***NOTE*** The tools that are configured through the `options()` described above will ***override*** any tools that were *also* installed through `patRoonExt`. Hence, this mechanism can be used to use specific versions not available though `patRoonExt`. However, this also means that you need to ensure that options are unset when you prefer that tools are used through `patRoonExt`.

2.3.2.2 Installing patRoon and its R dependencies The table below lists all the `R` packages that are involved in the installation of `patRoon`.

Note that only the `CAMERA` installation is mandatory, the rest involves installation of *optional* packages. If you are unsure which you need then you can always install the packages at a later stage.

The last three columns of the table provide hints on the availability from the `patRoonDeps`, r-universe and original regular sources (the sources were discussed previously). Note that you may need to install `remotes`, `BiocManager` and Rtools if packages are installed from their regular source. Some examples are shown below:

```
# Install patRoon (and its mandatory dependencies) from patRoonDeps
install.packages("patRoon", repos = "https://rickhelmus.github.io/patRoonDeps", type =
↪  "binary")

# Install KPIC2 from r-universe
install.packages("KPIC", repos = c('https://rickhelmus.r-universe.dev',
↪  'https://cloud.r-project.org'))

# Install the mandatory CAMERA package (will be installed automatically if using
↪  patRoonDeps/r-universe)
```

package	comments	patRoonDeps	r-universe	regular installation
CAMERA	Mandatory	no	no	'BiocManager::install('CAMERA')'
RDCOMClient	Only for windows	no	no	'remotes::install_github('BSchamberger/RDCOMClient')
ff	Dependency of RAMClustR	no	no	'install.packages('ff')'
Rdisop	Dependency of InterpretMSSpectrum	no	no	'BiocManager::install('Rdisop')'
InterpretMSSpectrum	Dependency of RAMClustR	no	yes	'install.packages('InterpretMSSpectrum')'
RAMClustR		no	yes	'remotes::install_github('cbroeckl/RAMClustR@release_
enviPick		no	yes	'remotes::install_github('blosloos/enviPick')'
nontargetData	Dependency of nontarget	no	yes	'remotes::install_github('blosloos/nontargetData')'
nontarget		no	yes	'remotes::install_github('blosloos/nontarget')'
ropls	Dependency of KPIC	no	no	'BiocManager::install('ropls')'
KPIC		no	yes	'remotes::install_github('rickhelmus/KPIC2')'
cliqueMS		no	yes	'remotes::install_github('rickhelmus/cliqueMS')'
BiocStyle	Dependency of MetaClean	no	no	'BiocManager::install('BiocStyle')'
Rgraphviz	Dependency of MetaClean	no	no	'BiocManager::install('Rgraphviz')'
fastAdaboost	Dependency of MetaClean	no	yes	'remotes::install_github('souravc83/fastAdaboost')'
MetaClean		no	yes	'remotes::install_github('KelseyChetnik/MetaClean')'
MetaCleanData		no	no	'remotes::install_github('KelseyChetnik/MetaCleanData'
splashR		no	yes	'remotes::install_github('berlinguyinca/spectra-hash')'
MS2Tox		no	no	'remotes::install_github('kruvelab/MS2Tox@main')'
MS2Quant		no	yes	'remotes::install_github('kruvelab/MS2Quant@main')'
patRoonData		no	no	'remotes::install_github('rickhelmus/patRoonData')'
patRoonExt		no	no	'remotes::install_github('rickhelmus/patRoonExt')'
patRoon		no	yes	'remotes::install_github('rickhelmus/patRoon@master')'

```
install.packages("BiocManager") # execute this if 'BiocManager' is not yet installed
BiocManager::install("CAMERA")

# Install patRoonData from GitHub
install.packages("remotes") # execute this if remotes (or devtools) is not yet installed
remotes::install_github("rickhelmus/patRoonData")
```

2.3.3 Verifying the installation

After the installation is completed, you may need to restart `R`. Afterwards, the `verifyDependencies()` function can be used to see if `patRoon` can find all its dependencies:

```
patRoon::verifyDependencies()
```

2.4 Managing legacy installations

Previous `patRoon` versions (`<2.3`) could be installed via an installation script. This script is now deprecated and replaced by the previously discussed installation methods. If you used this script in the past, and would like to update `patRoon`, it is important to first disable or fully remove the legacy installation. This is easily accomplished by the `patRoonInst` package that was discussed before:

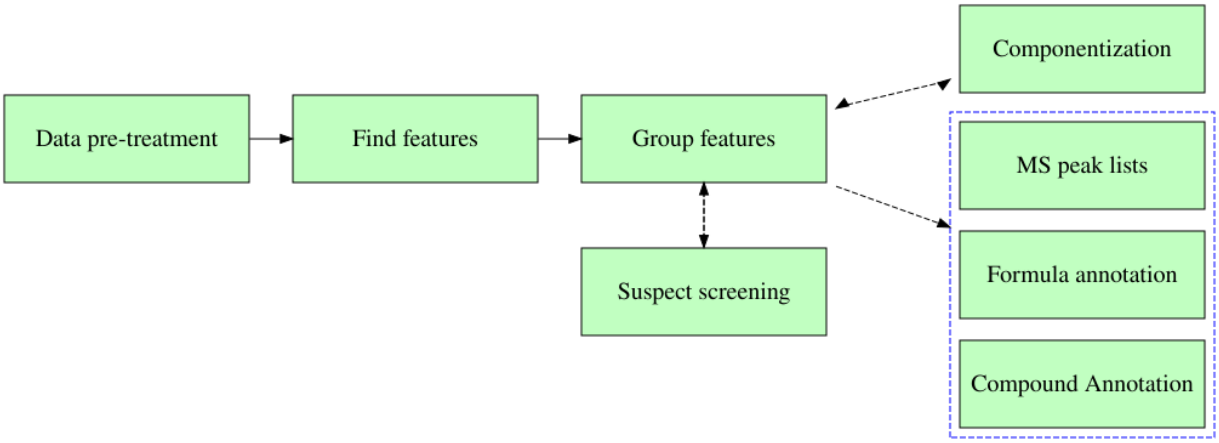
```
patRoonInst::toggleLegacy(FALSE) # disable legacy installation
patRoonInst::removeLegacy() # remove all files part of the legacy installation
patRoonInst::removeLegacy(restoreRProfile = TRUE) # as above, and remove any automatic
↪  changes that were made in ~/.Rprofile
```

NOTE Restart `R` afterwards to ensure all changes are in effect.

For more details, please refer to the reference manual (`?patRoonInst::legacy`).

3 Workflow concepts

In a non-target workflow both chromatographic and mass spectral data is automatically processed in order to provide a comprehensive chemical characterization of your samples. While the exact workflow is typically dependent on the type of study, it generally involves of the following steps:



Note that `patRoon` supports flexible composition of workflows. In the scheme above you can recognize optional steps by a *dashed line*. The inclusion of each step is only necessary if a further steps depends on its data. For instance, annotation and componentization do not depend on each other and can therefore be executed in any order or simply be omitted. A brief description of all steps is given below.

During **data pre-treatment** raw MS data is prepared for further analysis. A common need for this step is to convert the data to an open format so that other tools are able to process it. Other pre-treatment steps may involve re-calibration of m/z data or performing advanced filtering operations.

The next step is to extract **features** from the data. While different terminologies are used, a feature in `patRoon` refers to a single chromatographic peak in an extracted ion chromatogram for a single m/z value (within a defined tolerance). Hence, a feature contains both chromatographic data (e.g. retention time and peak height) and mass spectral data (e.g. the accurate m/z). Note that with mass spectrometry multiple m/z values may be detected for a single compound as a result of adduct formation, natural isotopes and/or in-source fragments. Some algorithms may try to combine these different masses in a single feature. However, in `patRoon` we generally assume this is not the case (and may optionally be done afterwards during the componentization step described below). Features are sometimes simply referred to as 'peaks'.

Features are found per analysis. Hence, in order to compare a feature across analyses, the next step is to group them. This step is essential as it finds equal features even if their retention time or m/z values slightly differ due to analytical variability. The resulting **feature groups** are crucial input for subsequent workflow steps. Prior to grouping, *retention time alignment* between analyses may be performed to improve grouping of features, especially when processing multiple analysis batches at once. Outside `patRoon` feature groups may also be defined as *profiles*, *aligned* or *grouped features* or *buckets*.

Depending on the study type, **suspect screening** is then performed to limit the features that should be considered for further processing. As its name suggests, with suspect screening only those features which are suspected to be present are considered for further processing. These suspects are retrieved from a suspect list which contains the m/z and (optionally) retention times for each suspect. Typical suspect lists may be composed from databases with known pollutants or from predicted transformation products. Note that for a 'full' non-target analysis no suspect screening is performed, hence, this step is simply omitted and all features are to be considered.

The feature group data may then be subjected to **componentization**. A **component** is defined as a collection of multiple feature groups that are somehow related to each other. Typical examples are features

that belong to the same chemical compound (i.e. with different *m/z* values but equal retention time), such as adducts, isotopes and in-source fragments. Other examples are homologous series and features that display a similar intensity trend across samples. If adducts or isotopes were annotated during componentization then this data may be used to prioritize the feature groups.

The last step in the workflow commonly involves **annotation**. During this step MS and MS/MS data are collected in so called **MS peak lists**, which are then used as input for formula and compound annotation. Formula annotation involves automatic calculation of possible formulae for each feature based on its *m/z*, isotopic pattern and MS/MS fragments, whereas compound annotation (or identification) involves the assignment of actual chemical structures to each feature. Note that during formula and compound annotation typically multiple candidates are assigned to a single feature. To assist interpretation of this data each candidate is therefore ranked on characteristics such as isotopic fit, number of explained MS/MS fragments and metadata from an online database such as number of scientific references or presence in common suspect lists.

To summarize:

- **Data-pretreatment** involves preparing raw MS data for further processing (e.g. conversion to an open format)
- **Features** describe chromatographic and *m/z* information (or 'peaks') in all analyses.
- A **feature group** consists of equal features across analyses.
- With **suspect screening** only features that are considered to be on a suspect list are considered further in the workflow.
- **Componentization** involves consolidating different feature groups that have a relationship to each other in to a single component.
- **MS peak lists** Summarizes all MS and MS/MS data that will be used for subsequent annotation.
- During **formula** and **compound annotation** candidate formulae/structures will be assigned and ranked for each feature.

The next chapters will discuss how to generate this data and process it. Afterwards, several advanced topics are discussed such as combining positive and negative ionization data, screening for transformation products and other advanced functionality.

4 Generating workflow data

4.1 Introduction

4.1.1 Workflow functions

Each step in the non-target workflow is performed by a function that performs the heavy lifting of a workflow step behind the scenes and finally return the results. An important goal of `patRoon` is to support multiple algorithms for each workflow step, hence, when such a function is called you have to specify which algorithm you want to use. The available algorithms and their characteristics will be discussed in the next sections. An overview of all functions involved in generating workflow data is shown in the table below.

Workflow step	Function	Output S4 class
Data pre-treatment	`convertMSFiles()`, `recalibrarateDAFiles()`	-
Finding features	`findFeatures()`	`features`
Grouping features	`groupFeatures()`	`featureGroups`
Suspect screening	`screenSuspects()`	`featureGroupsScreening`
Componentization	`generateComponents()`	`components`

Workflow step	Function	Output S4 class
MS peak lists	`generateMSPeakLists()`	`MSPeakLists`
Formula annotation	`generateFormulas()`	`formulas`
Compound annotation	`generateCompounds()`	`compounds`

4.1.2 Workflow output

The output of each workflow step is stored in objects derived from so called S4 classes. Knowing the details about the S4 class system of `R` is generally not important when using `patRoon` (and well written resources are available if you want to know more). In brief, usage of this class system allows a general data format that is used irrespective of the algorithm that was used to generate the data. For instance, when features have been found by OpenMS or XCMS they both return the same data format.

Another advantage of the S4 class system is the usage of so called *generic functions*. To put simply: a generic function performs a certain task for different types of data objects. A good example is the `plotSpectrum()` function which plots an (annotated) spectrum from data of MS peak lists or from formula or compound annotation:

```
# mslists, formulas, compounds contain results for MS peak lists and
# formula/compound annotations, respectively.

plotSpectrum(mslists, ...) # plot raw MS spectrum
plotSpectrum(formulas, ...) # plot annotated spectrum from formula annotation data
plotSpectrum(compounds, ...) # likewise but for compound annotation.
```

4.1.3 Overview of all functions and their output

The next sections in this chapter will further detail on how to actually perform the non-target workflow steps to generate data. The transformation product screening workflows are discussed in a separate chapter.

4.2 Preparations

4.2.1 Data pre-treatment

Prior to performing the actual non-target data processing workflow some preparations often need to be made. Often data has to be pre-treated, for instance, by converting it to an open format that is usable for subsequent workflow steps or to perform mass re-calibration. Some common functions are listed below.

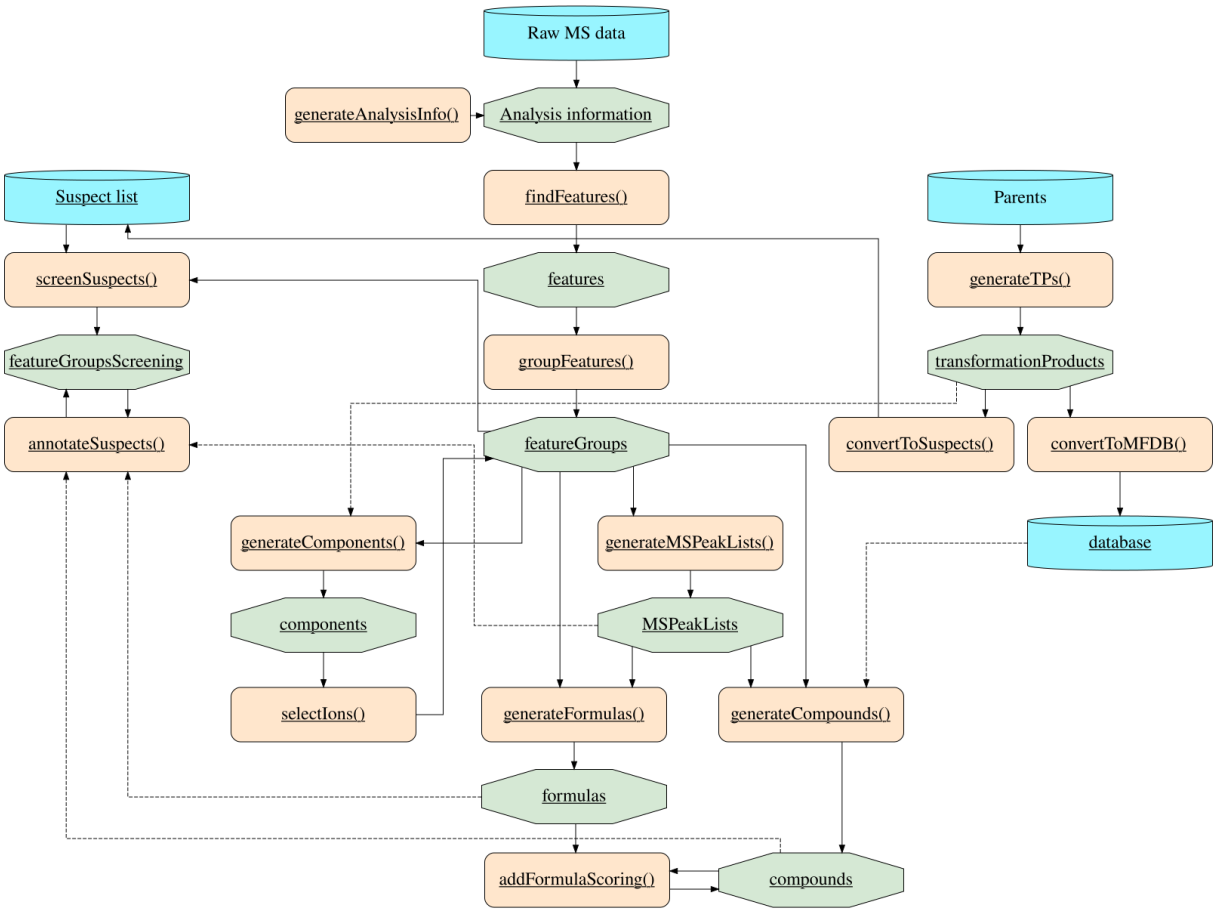


Figure 1: **Workflow functions and output classes.**

Task	Function	Algorithms	Supported file formats
Conversion	`convertMSFiles()`	OpenMS, ProteoWizard, DataAnalysis	All common (algorithm dependent)
Advanced (e.g. spectral filtering)	`convertMSFiles()`	ProteoWizard	All common
Mass re-calibration	`recalibrarateDAFiles()`	DataAnalysis	Bruker

The `convertMSFiles()` function supports conversion between many different file formats typically used in non-target analysis. Furthermore, other pre-treatment steps are available (e.g. centroiding, filtering) when the ProteoWizard algorithm is used. For an overview of these functionalities see the MsConvert documentation. Some examples:

```
# Converts a single mzXML file to mzML format
convertMSFiles("standard-1.mzXML", to = "mzML", algorithm = "openms")

# Converts all Thermo files with ProteoWizard (the default) in the analyses/raw
# directory and stores the mzML files in analyses/raw. Afterwards, only MS1
# spectra are retained.
convertMSFiles("analyses/raw", "analyses/mzml", from = "thermo",
               centroid = "vendor", filters = "msLevel 1")
```

> ***NOTE*** Most algorithms further down the workflow require the *mzML* or *mzXML* file format and additionally require that mass peaks have been centroided. When using the ProteoWizard algorithm (the default), centroiding by vendor algorithms is generally recommended (i.e. by setting `centroid="vendor"` as shown in the above example).

When Bruker MS data is used it can be automatically re-calibrated to improve its mass accuracy. Often this is preceeded by calling the `setDAMethod()` function to set a DataAnalysis method to all files in order to configure automatic re-calibration. The `recalibrarateDAFiles()` function performs the actual re-calibration. The `getDACalibrationError()` function can be used at anytime to request the current calibration error of each analysis. An example of these functions is shown below.

```
# anaInfo is a data.frame with information on analyses (see next section)
setDAMethod(anaInfo, "path/to/DAMethod.m") # configure Bruker files with given method
↪  that has automatic calibration setup
recalibrarateDAFiles(anaInfo) # trigger re-calibration for each analysis
getDACalibrationError(anaInfo) # get calibration error for each analysis (NOTE: also
↪  shown when previous function is finished)
```

4.2.2 Analysis information

The final bits of preparation is constructing the information for the analyses that need to be processed. In `patRoon` this is referred to as the *analysis information* and often stored in a variable `anaInfo` (of course you are free to choose a different name!). The analysis information should be a `data.frame` with the following columns:

- **path**: the directory path of the file containing the analysis data
- **analysis**: the name of the analysis. This should be the file name *without* file extension.
- **group**: to which *replicate group* the analysis belongs. All analysis which are replicates of each other get the same name.
- **blank**: which replicate group should be used for blank subtraction.
- **conc** (optional, advanced) A numeric value describing the concentration or any other value for which the intensity in this sample may correlate, for instance, dilution factor, sampling time etc. This column is only required when you want to obtain quantitative information (e.g. concentrations) using the `as.data.table()` method function (see `?featureGroups` for more information).

The `generateAnalysisInfo()` function can be used to (semi-)automatically generate a suitable `data.frame` that contains all the required information for a set of analysis. For, instance:

```
# Take example data from patRoonData package (triplicate solvent blank + triplicate
↪  standard)
generateAnalysisInfo(paths = patRoonData::exampleDataPath(),
                     groups = c(rep("solvent-pos", 3), rep("standard-pos", 3)),
                     blanks = "solvent-pos")
```

```
#>                                                    path       analysis        group       blank
#> 1 /usr/local/lib/R/site-library/patRoonData/extdata/pos  solvent-pos-1  solvent-pos solvent-pos
#> 2 /usr/local/lib/R/site-library/patRoonData/extdata/pos  solvent-pos-2  solvent-pos solvent-pos
#> 3 /usr/local/lib/R/site-library/patRoonData/extdata/pos  solvent-pos-3  solvent-pos solvent-pos
#> 4 /usr/local/lib/R/site-library/patRoonData/extdata/pos standard-pos-1 standard-pos solvent-pos
#> 5 /usr/local/lib/R/site-library/patRoonData/extdata/pos standard-pos-2 standard-pos solvent-pos
#> 6 /usr/local/lib/R/site-library/patRoonData/extdata/pos standard-pos-3 standard-pos solvent-pos
```

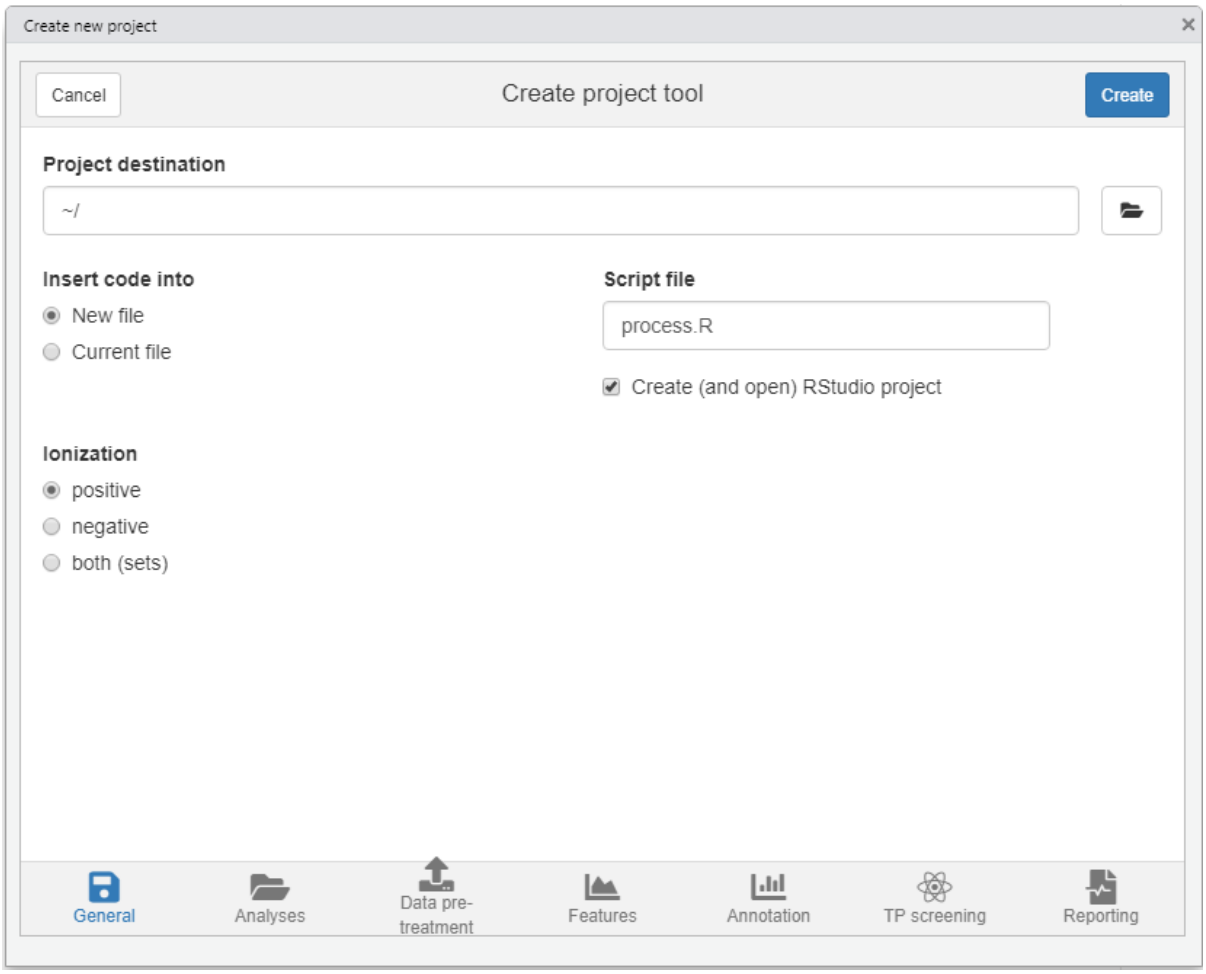
(Note that for the example data the `patRoonData::exampleAnalysisInfo()` function can also be used.)

Alternatively, the `newProject()` function discussed in the next section can be used to interactively construct this information.

4.2.3 Automatic project generation with newProject()

The previous sections already highlighted some steps that have to be performed prior to starting a new non-target analysis workflow: data pre-treatment and gathering information on the analysis. Most of the times you will put this and other `R` code a script file so you can recall what you have done before (i.e. reproducible research).

The `newProject()` function can be used to setup a new project. When you run this function it will launch a small tool (see screenshot below) where you can select your analyses and configure the various workflow steps which you want to execute (e.g. data pre-treatment, finding features, annotation etc). After setting everything up the function will generate a template script which can easily be edited afterwards. In addition, you have the option to create a new RStudio project, which is advantegeous as it neatly seperates your data processing work from the rest.



NOTE At the moment `newProject()` *only* works with RStudio.

4.3 Features

Collecting features from the analyses consists of finding all features, grouping them across analyses (optionally after retention time alignment), and if desired suspect screening:

4.3.1 Finding and grouping features

Several algorithms are available for finding features. These are listed in the table below alongside their usage and general remarks.

Algorithm	Usage	Remarks
OpenMS	`findFeatures(algorithm = "openms", ...)`	Uses the FeatureFinderMetabo algorithm
XCMS	`findFeatures(algorithm = "xcms", ...)`	Uses `xcms::xcmsSet()` function
XCMS (import)	`importFeatures(algorithm = "xcms", ...)`	Imports an existing `xcmsSet` object
XCMS3	`findFeatures(algorithm = "xcms3", ...)`	Uses `xcms::findChromPeaks()` from the new XCMS3 interface
XCMS3 (import)	`importFeatures(algorithm = "xcms3", ...)`	Imports an existing `XCMSnExp` object
enviPick	`findFeatures(algorithm = "envipick", ...)`	Uses `enviPick::enviPickwrap()`
KPIC2	`findFeatures(algorithm = "kpic2", ...)`	Uses the KPIC2 `R` package
KPIC2 (import)	`importFeatures(algorithm = "kpic2", ...)`	Imports features from KPIC2
SIRIUS	`findFeatures(algorithm = "sirius", ...)`	Uses SIRIUS to find features
SAFD	`findFeatures(algorithm = "safd", ...)`	Uses the SAFD algorithm (experimental)
DataAnalysis	`findFeatures(algorithm = "bruker", ...)`	Uses Find Molecular Features from DataAnalysis (Bruker only)

Most often the performance of these algorithms heavily depend on the data and parameter settings that are used. Since obtaining a good feature dataset is crucial for the rest of the workflow, it is highly recommended to experiment with different settings (this process can also be automated, see the feature optimization section for more details). Some common parameters to look at are listed in the table below. However, there are many more parameters that can be set, please see the reference documentation for these (e.g. `?findFeatures`).

Algorithm	Common parameters
OpenMS	`noiseThrInt`, `chromSNR`, `chromFWHM`, `mzPPM`, `minFWHM`, `maxFWHM` (see `?findFeatures`)
XCMS / XCMS3	`peakwidth`, `mzdiff`, `prefilter`, `noise` (assuming default `centWave` algorithm, see `?findPeaks.centWave` / `?CentWaveParam`)
enviPick	`dmzgap`, `dmzdens`, `drtgap`, `drtsmall`, `drtdens`, `drtfill`, `drttotal`, `minpeak`, `minint`, `maxint` (see `?enviPickwrap`)
KPIC2	`kmeans`, `level`, `min_snr` (see `?findFeatures` and `?getPIC` / `?getPIC.kmeans`)
SIRIUS	The `sirius` algorithm is currently parameterless
SAFD	`mzRange`, `maxNumbIter`, `resolution`, `minInt` (see `?findFeatures`)
DataAnalysis	See *Find -> Parameters... -> Molecular Features* in DataAnalysis.

NOTE Support for SAFD is still experimental and some extra work is required to set everything up. Please see the reference documentation for this algorithm (`?findFeatures`).

NOTE DataAnalysis feature settings have to be configured in DataAnalysis prior to calling `findFeatures()`.

Similarly, for grouping features across analyses several algorithms are supported.

Algorithm	Usage	Remarks
OpenMS	`groupFeatures(algorithm = "openms", ...)`	Uses the FeatureLinkerUnlabeled algorithm (and MapAlignerPoseClustering for retention alignment)
XCMS	`groupFeatures(algorithm = "xcms", ...)`	Uses `xcms::group()` `xcms::retcor()` functions
XCMS (import)	`importFeatureGroupsXCMS(...)`	Imports an existing `xcmsSet` object.
XCMS3	`groupFeatures(algorithm = "xcms3", ...)`	Uses `xcms::groupChromPeaks()` and `xcms::adjustRtime()` functions
XCMS3 (import)	`importFeatureGroupsXCMS3(...)`	Imports an existing `XCMSnExp` object.
KPIC2	`groupFeatures(algorithm = "kpic2", ...)`	Uses the KPIC2 package
KPIC2 (import)	`importFeatureGroupsKPIC2(...)`	Imports a `PIC set` object
SIRIUS	`groupFeatures(anaInfo, algorithm = "sirius")`	Finds *and* groups features with SIRIUS
ProfileAnalysis	`importFeatureGroups(algorithm = "brukerpa", ...)`	Import `.csv` file exported from Bruker ProfileAnalysis
TASQ	`importFeatureGroups(algorithm = "brukertasq", ...)`	Imports a *Global result table* (exported to Excel file and then saved as `.csv` file)

NOTE: Grouping features with the `sirius` algorithm will perform both finding and grouping features with SIRIUS. This algorithm cannot work with features from another algorithm.

Just like finding features, each algorithm has their own set of parameters. Often the defaults are a good start but it is recommended to have look at them. See `?groupFeatures` for more details.

When using the XCMS algorithms both the 'classical' interface and latest `XCMS3` interfaces are supported. Currently, both interfaces are mostly the same regarding functionalities and implementation. However, since future developments of XCMS are primarily focused the latter this interface is recommended.

Some examples of finding and grouping features are shown below.

```
# The anaInfo variable contains analysis information, see the previous section

# Finding features
fListOMS <- findFeatures(anaInfo, "openms") # OpenMS, with default settings
fListOMS2 <- findFeatures(anaInfo, "openms", noiseThrInt = 500, chromSNR = 10) # OpenMS,
↪  adjusted minimum intensity and S/N
fListXCMS <- findFeatures(anaInfo, "xcms", ppm = 10) # XCMS
fListXCMSImp <- importFeatures(anaInfo, "xcms", xset) # import XCMS xcmsSet object
fListXCMS3 <- findFeatures(anaInfo, "xcms3", CentWaveParam(peakwidth = c(5, 15))) # XCMS3
fListEP <- findFeatures(anaInfo, "envipick", minint = 1E3) # enviPick
fListKPIC2 <- findFeatures(anaInfo, "kpic2", kmeans = TRUE, level = 1E4) # KPIC2
```

```
fListSIRIUS <- findFeatures(anaInfo, "sirius") # SIRIUS

# Grouping features
fGroupsOMS <- groupFeatures(fListOMS, "openms") # OpenMS grouping, default settings
fGroupsOMS2 <- groupFeatures(fListOMS2, "openms", rtalign = FALSE) # OpenMS grouping, no
↪  RT alignment
fGroupsOMS3 <- groupFeatures(fListXCMS, "openms", maxGroupRT = 6) # group XCMS features
↪  with OpenMS, adjusted grouping parameter
# group enviPick features with XCMS3, disable minFraction
fGroupsXCMS <- groupFeatures(fListEP, "xcms3",
                             xcms::PeakDensityParam(sampleGroups = anaInfo$group,
                                                    minFraction = 0))
# group with KPIC2 and set some custom grouping/aligning parameters
fGroupsKPIC2 <- groupFeatures(fListKPIC2, "kpic2", groupArgs = list(tolerance = c(0.002,
↪  18)),
                              alignArgs = list(move = "loess"))
fGroupsSIRIUS <- groupFeatures(anaInfo, "sirius") # find/group features with SIRIUS
```

4.3.2 Suspect screening

After features have been grouped a so called suspect screening step may be performed to find features that may correspond to suspects within a given suspect list. The `screenSuspects()` function is used for this purpose, for instance:

```
suspects <- data.frame(name = c("1H-benzotriazole", "N-Phenyl urea",
↪  "2-Hydroxyquinoline"),
                       mz = c(120.0556, 137.0709, 146.0600))
fGroupsSusp <- screenSuspects(fGroups, suspects)
```

4.3.2.1 Suspect list format The example above has a very simple suspect list with just three compounds. The format of the suspect list is quite flexible, and can contain the following columns:

- `name`: The name of the suspect. Mandatory and should be unique and file-name compatible (if not, the name will be automatically re-named to make it compatible).
- `rt`: The retention time in seconds. Optional. If specified any feature groups with a different retention time will not be considered to match suspects.
- `mz`, `SMILES`, `InChI`, `formula`, `neutralMass`: *at least* one of these columns must hold data for each suspect row. The `mz` column specifies the ionized mass of the suspect. If this is not available then data from any of the other columns is used to determine the suspect mass.
- `adduct`: The adduct of the suspect. Optional. Set this if you are sure that a suspect should be matched by a particular adduct ion and no data in the `mz` column is available.
- `fragments_mz` and `fragments_formula`: optional columns that may assist suspect annotation.

In most cases a suspect list is best made as a `csv` file which can then be imported with e.g. the `read.csv()` function. This is exactly what happen when you specify a suspect list when using the `newProject()` function.

Quite often, the ionized masses are not readily available and these have to be calculated. In this case, data in any of the `SMILES`/`InChI`/`formula`/`neutralMass` columns should be provided. Whenever possible, it is *strongly* recommended to fill in `SMILES` column (or `InChI`), as this will assist annotation. Applying this to the above example:

```
suspects <- data.frame(name = c("1H-benzotriazole", "N-Phenyl urea",
↪  "2-Hydroxyquinoline"),
                       SMILES = c("[nH]1nnc2ccccc12", "NC(=O)Nc1ccccc1",
                       ↪  "Oc1ccc2ccccc2n1"))
fGroupsSusp <- screenSuspects(fGroups, suspects, adduct = "[M+H]+")
```

> ***NOTE***: It is highly recommended to install OpenBabel to automatically validate and amend chemical properties such as SMILES, InChI, formulae etc in the suspect list.

Since suspect matching now occurs by the neutral mass it is required that the adduct information for the feature groups are set. This is done either by setting the `adduct` function argument to `screenSuspects` or by feature group adduct annotations.

Finally, when the adduct is known for a suspect it can be specified in the suspect list:

```
# Aldicarb is measured with a sodium adduct.
suspects <- data.frame(name = c("1H-benzotriazole", "N-Phenyl urea", "Aldicarb"),
                       SMILES = c("[nH]1nnc2ccccc12", "NC(=O)Nc1ccccc1",
                       ↪  "CC(C)(C=NOC(=O)NC)SC"),
                       adduct = c("[M+H]+", "[M+H]+", "[M+Na]+"))
fGroupsSusp <- screenSuspects(fGroups, suspects)
```

To summarize:

- If a suspect has data in the `mz` column it will be directly matched with the *m/z* value of a feature group.
- Otherwise, if the suspect has data in the `adduct` column, the `m/z` value for the suspect is calculated from its neutral mass and the adduct and then matched with the `m/z` of a feature group.
- Otherwise, suspects and feature groups are matched by their the neutral mass.

The `fragments_mz` and `fragments_formula` columns in the suspect list can be used to specify known fragments for a suspect, which can help suspect annotation. The former specifies the ionized *m/z* of known MS/MS peaks, whereas the second specifies known formulas. Multiple values can be given by separating them with a semicolon:

```
suspects <- data.frame(name = c("1H-benzotriazole", "N-Phenyl urea",
↪  "2-Hydroxyquinoline"),
                       SMILES = c("[nH]1nnc2ccccc12", "NC(=O)Nc1ccccc1",
                       ↪  "Oc1ccc2ccccc2n1"),
                       fragments_formula = c("C6H6N", "C6H8N;C7H6NO", ""),
                       fragments_mz = c("", "", "118.0652"))
```

4.3.2.2 Removing feature groups without hits

Note that any feature groups that were not matched to a suspect are *not* removed by default. If you want to remove these, you can use the `onlyHits` parameter:

```
fGroupsSusp <- screenSuspects(fGroups, suspects, onlyHits = TRUE) # remove any non-hits
↪  immediately
```

The advantage of removing non-hits is that it may significantly reduce the complexity of your dataset. On the other hand, retaining all features allows you to mix a full non-target analysis with a suspect screening workflow. The `filter()` function (discussed here) can also be used to remove feature groups without a hit at a later stage.

4.3.2.3 Combining screening results

The `amend` function argument to `screenSuspects` can be used to combine screening results from different suspect lists.

```
fGroupsSusp <- screenSuspects(fGroups, suspects)
fGroupsSusp <- screenSuspects(fGroupsSusp, suspects2, onlyHits = TRUE, amend = TRUE)
```

In this example the suspect lists defined in `suspects` and `suspects2` are both used for screening. By setting `amend=TRUE` the original screening results (i.e. from `suspects`) are preserved. Note that `onlyHits` should only be set in the final call to `screenSuspects` to ensure that all feature groups are screened.

4.4 Componentization

In `patRoon` *componentization* refers to grouping related feature groups together in components. There are different methodologies to generate components:

- Similarity on chromatographic elution profiles: feature groups with similar chromatographic behaviour which are assuming to be the same chemical compound (e.g. adducts or isotopologues).
- Homologous series: features with increasing m/z and retention time.
- Intensity profiles: features that follow a similar intensity profile in the analyses.
- MS/MS similarity: feature groups with similar MS/MS spectra are clustered.
- Transformation products: Components are formed by grouping feature groups that have a parent/transformation product relationship. This is further discussed in its own chapter.

The following algorithms are currently supported:

Algorithm	Usage	Remarks
CAMERA	`generateComponents(algorithm = "camera", ...)`	Clusters feature groups with similar chromatographic elution profiles and annotate by known chemical rules (adducts, isotopologues, in-source fragments).
RAMClustR	`generateComponents(algorithm = "ramclustr", ...)`	As above.
cliqueMS	`generateComponents(algorithm = "cliquems", ...)`	As above, but using *feature components*.
OpenMS	`generateComponents(algorithm = "openms", ...)`	As above. Uses MetaboliteAdductDecharger.
nontarget	`generateComponents(algorithm = "nontarget", ...)`	Uses the nontarget R package to perform unsupervised homologous series detection.
Intensity clustering	`generateComponents(algorithm = "intclust", ...)`	Groups features with similar intensity profiles across analyses by hierarchical clustering.
MS/MS clustering	`generateComponents(algorithm = "specclust", ...)`	Clusters feature groups with similar MS/MS spectra.
Transformation products	`generateComponents(algorithm = "tp", ...)`	Discussed in its own chapter.

4.4.1 Features with similar chromatographic behaviour

Isotopes, adducts and in-source fragments typically result in detection of multiple mass peaks by the mass spectrometer for a single chemical compound. While some feature finding algorithms already try to collapse

(some of) these in to a single feature, this process is often incomplete (if performed at all) and it is not uncommon that multiple features will describe the same compound. To overcome this complexity several algorithms can be used to group features that undergo highly similar chromatographic behavior but have different *m/z* values. Basic chemical rules are then applied to the resulting components to annotate adducts, in-source fragments and isotopologues, which may be highly useful for general identification purposes.

Note that some algorithms were primarily designed for datasets where features are generally present in the majority of the analyses (as is relatively common in metabolomics). For environmental analyses, however, this is often not the case. For instance, consider the following situation with three feature groups that chromatographically overlap and therefore could be considered a component:

Feature group	*m/z*	analysis 1	analysis 2	analysis 3
#1	100.08827	Present	Present	Absent
#2	122.07021	Present	Present	Absent
#3	138.04415	Absent	Absent	Present

Based on the mass differences from this example a cluster of `[M+H]+`, `[M+Na]+` and `[M+K]+` could be assumed. However, no features of the first two feature groups were detected in the third sample analysis, whereas the third feature group wasn't detected in the first two sample analysis. Based on this it seems unlikely that feature group *#3* should be part of the component.

For the algorithms that operate on a 'feature group level' (CAMERA and RAMClustR), the `relMinReplicates` argument can be used to remove feature groups from a component that are not abundant. For instance, when this value is *0.5* (the default), and all the features of a component were detected in four different replicate groups in total, then only those feature groups are kept for which its features were detected in at least two different replicate groups (*i.e.* half of four).

Another approach to reduce unlikely adduct annotations is to use algorithms that operate on a 'feature level' (cliqueMS and OpenMS). These algorithms generate components for each sample analysis individually. The 'feature components' are then merged by a consensus approach where unlikely annotations are removed (the algorithm is described further in the reference manual, `?generateComponents`).

Each algorithm supports many different parameters that may significantly influence the (quality of the) output. For instance, care has to be taken to avoid 'over-clustering' of feature groups which do not belong in the same component. This is often easily visible since the chromatographic peaks poorly overlap or are shaped differently. The `checkComponents` function (discussed here) can be used to quickly verify componentization results. For a complete listing all arguments see the reference manual (e.g. `?generateComponents`).

Once the components with adduct and isotopes annotations are generated this data can be used to prioritize and improve the workflow.

Some example usage is shown below.

```
# Use CAMERA with defaults
componCAM <- generateComponents(fGroups, "camera", ionization = "positive")

# CAMERA with customized settings
componCAM2 <- generateComponents(fGroups, "camera", ionization = "positive",
                                 extraOpts = list(mzabs = 0.001, sigma = 5))

# Use RAMClustR with customized parameters
componRC <- generateComponents(fGroups, "ramclustr", ionization = "positive", hmax = 0.4,
                               extraOptsRC = list(cor.method = "spearman"),
                               extraOptsFM = list(ppm.error = 5))
```

```
# OpenMS with customized parameters
componOpenMS <- generateComponents(fGroups, "openms", ionization = "positive", chargeMax
↪  = 2,
                                   absMzDev = 0.002)

# cliqueMS with default parameters
componCliqueMS <- generateComponents(fGroups, "cliquems", ionization = "negative")
```

4.4.2 Homologues series

Homologues series can be automatically detected by interfacing with the nontarget R package. Components are made from feature groups that show increasing m/z and retention time values. Series are first detected within each replicate group. Afterwards, series from all replicates are linked in case (partial) overlap occurs and this overlap consists of the *same* feature groups (see figure below). Linked series are then finally merged if this will not cause any conflicts with other series: such a conflict typically occurs when two series are not only linked to each other.

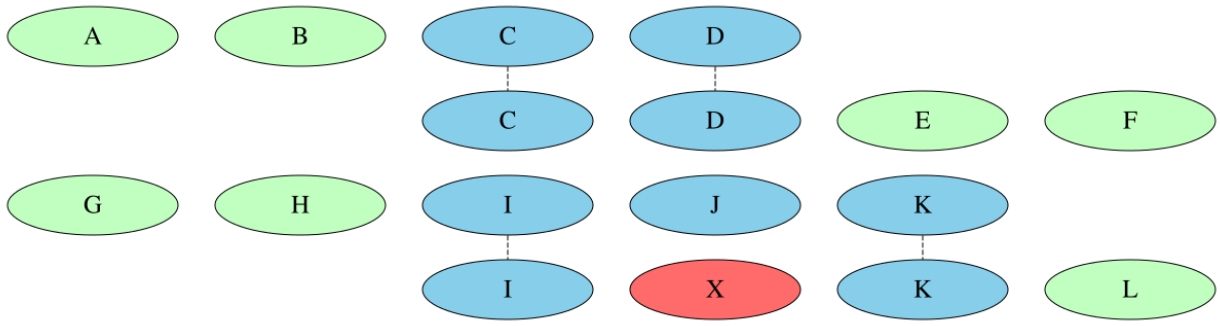


Figure 2: **Linking of homologues series** top: partial overlap and will be linked; bottom: no linkage due to different feature in overlapping series.

The series that are linked can be interactively explored with the `plotGraph()` function (discussed here).

Common function arguments to `generateComponents()` are listed below.

Argument	Remarks
`ionization`	Ionization mode: `"positive"` or `"negative"`. Not needed if adduct annotations are available.
`rtRange`, `mzRange`	Retention and m/z increment range. Retention times can be negative to allow series with increasing m/z values and decreasing retention times.
`elements`	Vector with elements to consider.
`rtDev`, `absMzDev`	Maximum retention time and m/z deviation.
`...`	Further arguments passed to the `homol.search()` function.

```
# default settings
componNT <- generateComponents(fGroups, "nontarget", ionization = "positive")

# customized settings
componNT2 <- generateComponents(fGroups, "nontarget", ionization = "positive",
                                elements = c("C", "H"), rtRange = c(-60, 60))
```

4.4.3 Intensity and MS/MS similarity

The previous componentization methods utilized chemical properties to relate features. The two componentization algorithms described in this section use a statistical approach based on hierarchical clustering. The first algorithm normalizes all feature intensities and then clusters features with similar intensity profiles across sample analyses together. The second algorithm compares all MS/MS spectra from all feature groups, and then uses hierarchical clustering to generate components from feature groups that have a high MS/MS spectrum similarity.

Some common arguments to `generateComponents()` are listed below. It is recommended to test various settings (especially for `method`) to optimize the clustering results.

Argument	Algorithm	Default	Remarks
`method`	All	`"complete"`	Clustering method. See `?hclust`
`metric`	`intclust`	`"euclidean"`	Metric used to calculate the distance matrix. See `?daisy`.

`normalized` |intclust|TRUE| `Whether normalized feature intensities should be used. Detailed [here](#fNorm).`average|intclust|TRUE| `Whether intensities of replicates should first be averaged.`MSPeakLists|specclust| - | `The [MS peak lists] object used for spectral similarity calculations`specSimParams|specclust|getDefSpecSimParams()| `Parameters used for [spectral similarity calculation](#specSim).`maxTreeHeight,deepSplit,minModuleSize| `All` |1,TRUE,1| `Used for dynamic cluster assignment. See`?cutreeDynamicTree‘.

The components are generated by automatically assigning clusters using the dynamicTreeCut `R` package. However, the cluster assignment can be performed manually or with different parameters, as is demonstrated below.

The resulting components are stored in an object from the `componentsIntClust` or `componentsSpecClust` S4 class, which are both derived from the `componentsClust` class (which in turn is derived from the `components` class). Several methods are defined that can be used on these objects to re-assign clusters, perform plotting operations and so on. Below are some examples. For plotting see the relevant visualization section. More info can be found in the reference manual (e.g. `?componentsIntClust`, `?componentsSpecClust` and `?componentsClust`).

```
# generate intensity profile components with default settings
componInt <- generateComponents(fGroups, "intclust")

# manually re-assign clusters
componInt <- treeCut(componInt, k = 10)

# automatic re-assignment of clusters (adjusted max tree height)
componInt <- treeCutDynamic(componInt, maxTreeHeight = 0.7)

# MS/MS similarity components
componMSMS <- generateComponents(fGroups, "specclust", MSPeakLists = mslists)
```

4.5 Incorporating adduct and isotopic data

With mass spectrometry it is common that multiple m/z values are detected for a single compound. These may be different adducts (e.g. `[M+H]+`, `[M+Na]+`, `[M-H]-`), the different isotopes of the molecule or a combination thereof. When multiple m/z values are measured for the same compound, the feature finding algorithm may yield a distinct feature for each, which adds complexity to the data. In the previous section it was

discussed how componentization can help to find feature groups that belong to the same adduct and/or isotope clusters. This section explains how this data can be used to simplify the feature dataset. Furthermore, this section also covers adduct annotations for feature groups which may improve and simplify the general workflow.

4.5.1 Selecting features with preferential adducts/isotopes

The `selectIons` function forms the bridge between feature group and componentization data. This function uses the adduct and isotope annotations to select *preferential* feature groups. For adduct clusters this means that only the feature group that has a preferential adduct (e.g. `[M+H]+`) is kept while others (e.g. `[M+Na]+`) are removed. If none of the adduct annotations are considered preferential, the most intense feature group is kept instead. For isotopic clusters typically only the feature group with the monoisotopic mass (i.e. *M0*) is kept.

The behavior of `selectIons` is configurable with the following parameters:

Argument	Remarks
`prefAdduct`	The *preferential adduct*. Usually `"[M+H]+"` or `"[M-H]-"`.
`onlyMonoIso`	If `TRUE` and a feature group is with isotopic annotations then it is only kept if it is monoisotopic.
`chargeMismatch`	How charge mismatches between adduct and isotope annotations are dealt with. Valid options are `"isotope"`, `"adduct"`, `"none"` or `"ignore"`. See the reference manual for `selectIons` for more details.

In case componentization did not lead to an adduct annotation for a feature group it will never be removed and simply be annotated with the preferential adduct. Similarly, when no isotope annotations are available and `onlyMonoIso=TRUE`, the feature group will not be removed.

Although `selectIons` operates fairly conservative, it is still recommended to verify the componentization results in advance, for instance with the `checkComponents` function discussed here. Furthermore, the next subsection explains how adduct annotations can be corrected manually if needed.

An example usage is shown below.

```
fGroupsSel <- selectIons(fGroups, componCAM, "[M+H]+")
```

```
#> No isotope annotations available!
#> Removed 21 feature groups detected as unwanted adducts/isotopes
#> Annotated 13 feature groups with adducts
#>  Remaining 110 feature groups set as default adduct [M+H]+
```

4.5.2 Setting adduct annotations for feature groups

The `adducts()` function can be used to obtain a character vector with adduct annotations for each feature group. When no adduct annotations are available it will simply return an empty character vector.

When the `selectIons` function is used it will automatically add adduct annotations based on the componentization data. In addition, the `adducts()<-` function can be used to manually add or change adduct annotations.

```
adducts(fGroups) # no adduct annotations
```

```
#> character(0)
```

```
adducts(fGroupsSel)[1:5] # adduct annotations set by selectIons()
```

```
#> M109_R192_20 M111_R330_23 M114_R269_25 M116_R317_29 M120_R268_30
#>     "[M+H]+"     "[M+H]+"     "[M+H]+"     "[M+H]+"     "[M+K]+"
```

```
adducts(fGroupsSel)[3] <- "[M+Na]+" # modify annotation
adducts(fGroupsSel)[1:5] # verify
```

```
#> M109_R192_20 M111_R330_23 M114_R269_25 M116_R317_29 M120_R268_30
#>     "[M+H]+"     "[M+H]+"    "[M+Na]+"     "[M+H]+"     "[M+K]+"
```

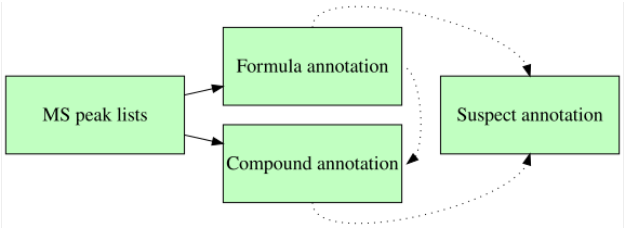
NOTE Adduct annotations are always available with sets workflows.

4.5.3 Using adduct annotations in the workflow

When feature groups have adduct annotations available this may simplify and improve the workflow. The `adduct` and `ionization` arguments used for suspect screening, formula/compound annotation and some componentization algorithms do not have to be set anymore, since this data can be obtained from the adduct annotations. Furthermore, these algorithms may improve their results, since the algorithms are now able to use adduct information for each feature group individually, instead of assuming that all feature groups have the same adduct.

4.6 Annotation

The annotation consists of collecting MS peak lists and then formula and/or compound annotation:



Note that compound annotation is normally not dependent upon formula annotation. However, formula data can be used to improve ranking of candidates afterwards by the `addFormulaScoring()` function, which will be discussed later in this section. Furthermore, suspect annotation is not mandatory, and may use data from peak lists, formulae and/or comounds.

4.6.1 MS peak lists

Algorithm	Usage	Remarks
mzR	`generateMSPeakLists(algorithm = "mzr", ...)`	Uses mzR for spectra retrieval. Recommended default.
DataAnalysis	`generateMSPeakLists(algorithm = "bruker", ...)`	Loads data after automatically generating MS and MS/MS spectra in DataAnalysis

Algorithm	Usage	Remarks
DataAnalysis FMF	`generateMSPeakLists(algorithm = "brukerfmf", ...)`	Uses spectra from the *find molecular features* algorithm.

The recommended default algorithm is `mzr`: this algorithm is generally faster and is not limited to a vendor data format as it will read the open `mzML` and `mzXML` file formats. On the other hand, when DataAnalysis is used with Bruker data the spectra can be automatically background subtracted and there is no need for file conversion. Note that the `brukerfmf` algorithm only works when `findFeatures()` was called with the `bruker` algorithm.

When `generateMSPeakists()` is called it will

1. Find all MS and MS/MS spectra that 'belong' to a feature. For MS spectra this means that all spectra close to the retention time of a feature will be collected. In addition, for MS/MS normally only spectra will be considered that have a precursor mass close to that of the feature (however, this can be disabled for data that was recorded with data independent acquisition (DIA, MSˆE, bbCID, ...)).
2. Average all MS and MS/MS spectra to produce peak lists for each feature.
3. Average all peak lists for features within the same group.

Data from either (2) or (3) is used for subsequent annotation steps. Formula calculation can use either (as a trade-off between possibly more accurate results by outlier removal *vs* speed), whereas compound annotation will always use data from (3) since annotating single features (as opposed to their groups) would take a very long time.

There are several common function arguments to `generateMSPeakLists()` that can be used to optimize its behaviour:

Argument	Algorithm(s)	Remarks
`maxMSRtWindow`	`mzr`, `bruker`	Maximum time window +/- the feature retention time (in seconds) to collect spectra for averaging. Higher values may significantly increase processing times.
`precursorMzWindow`	`mzr`	Maximum precursor *m/z* search window to find MS/MS spectra. Set to `NULL` to disable (i.e. for DIA experiments).
`topMost`	`mzr`	Only retain feature data for no more than this amount analyses with highest intensity. For instance, a value of *1* will only keep peak lists for the feature with highest intensity in a feature group.
`bgsubtr`	`bruker`	Perform background subtraction (if the spectra type supports this, e.g. MS and bbCID)
`minMSIntensity`, `minMSMSIntensity`	`bruker`, `brukerfmf`	Minimum MS and MS/MS intensity. Note that DataAnalysis reports many zero intensity peaks so a value of at least *1* is recommended.
`MSMSType`	`bruker`	The type of spectra that should be used for MSMS: `"BBCID"` for bbCID experiments, otherwise `"MSMS"` (the default).

In addition, several parameters can be set that affect spectral averaging. These parameters are passed as a `list` to the `avgFeatParams` (`mzr` algorithm only) and `avgFGroupParams` arguments, which affect averaging of feature and feature group data, respectively. Some typical parameters include:

- `clusterMzWindow`: Maximum *m/z* window used to cluster mass peaks when averaging. The better the MS resolution, the lower this value should be.

- `topMost`: Retain no more than this amount of most intense mass peaks. Useful to filter out 'noisy' peaks.
- `minIntensityPre` / `minIntensityPost`: Mass peaks below this intensity will be removed before/after averaging.

See `?generateMSPeakLists` for all possible parameters.

A suitable list object to set averaging parameters can be obtained with the `getDefAvgPListParams()` function.

```
# lower default clustering window, other settings remain default
avgPListParams <- getDefAvgPListParams(clusterMzWindow = 0.001)

# Apply to both feature and feature group averaging
plists <- generateMSPeakLists(fGroups, "mzr", avgFeatParams = avgPListParams,
↪  avgFGroupParams = avgPListParams)
```

4.6.2 Formulae

Formulae can be automatically calculated for all features using the `generateFormulas()` function. The following algorithms are currently supported:

Algorithm	Usage	Remarks
GenForm	`generateFormulas(algorithm = "genform", ...)`	Bundled with `patRoon`. Reasonable default.
SIRIUS	`generateFormulas(algorithm = "sirius", ...)`	Requires MS/MS data.
DataAnalysis	`generateFormulas(algorithm = "bruker", ...)`	Requires FMF features (i.e. `findFeatures(algorithm = "bruker", ...)`). Uses *SmartFormula* algorithms.

Calculation with GenForm is often a good default. It is fast and basic rules can be applied to filter out obvious non-existing formulae. A possible drawback of GenForm, however, is that may become slow when many candidates are calculated, for instance, due to a relative high feature m/z (e.g. >600) or loose elemental restricitions. More thorough calculation is performed with SIRIUS: this algorithm often yields fewer and often more plausible results. However, SIRIUS requires MS/MS data (hence features without will not have results) and formula prediction may not work well for compounds that structurally deviate from the training sets used by SIRIUS. Calculation with DataAnalysis is only possible when features are obtained with DataAnalysis as well. An advantage is that analysis files do not have to be converted, however, compared to other algorithms calculation is often relative slow.

There are two methods for formula assignment:

1. Formulae are first calculated for each individual feature within a feature group. These results are then pooled, outliers are removed and remaining formulae are assigned to the feature group (i.e. `calculateFeatures = TRUE`).
2. Formulae are directly calculated for each feature group by using group averaged peak lists (see previous section) (i.e. `calculateFeatures = FALSE`).

The first method is more thorough and the possibility to remove outliers may sometimes result in better formula assignment. However, the second method is much faster and generally recommended for large number of analyses.

By default, formulae are either calculated by *only* MS/MS data (SIRIUS) or with both MS *and* MS/MS data (GenForm/Bruker). The latter also allows formula calculation when no MS/MS data is present. Furthermore, with Bruker algorithms, data from both MS and MS/MS formula data can be combined to allow inclusion of candidates that would otherwise be excluded by e.g. poor MS/MS data. However, a disadvantage is that formulae needs to be calculated twice. The `MSMode` argument (listed below) can be used to customize this behaviour.

An overview of common parameters that are typically set to customize formula calculation is listed below.

Argument	Algorithm(s)	Remarks
relMzDev	`genform`, `sirius`	The maximum relative *m/z* deviation for a formula to be considered (in *ppm*).
elements	`genform`, `sirius`	Which elements to consider. By default `"CHNOP"`. Try to limit possible elements as much as possible.
calculateFeatures	`genform`, `sirius`	Whether formulae should be calculated first for all features (see discussion above) (always `TRUE` with DataAnalysis).
featThresholdAnnAll		Minimum relative amount (*0-1*) that a candidate formula for a feature group should be found among all annotated features (e.g. *1* means that a candidate is only considered if it was assigned to all annotated features).
adduct	All	The adduct to consider for calculation (e.g. `"[M+H]+"`, `"[M-H]-"`, more details in the adduct section). Don't set this when adduct annotations are available.
MSMode	`genform`, `bruker`	Whether formulae should be generated only from MS data (`"ms"`), MS/MS data (`"msms"`) or both (`"both"`). The latter is default, see discussion above.
profile	`sirius`	Instrument profile, e.g. `"qtof"`, `"orbitrap"`, `"fticr"`.

Some typical examples:

```
formulasGF <- generateFormulas(fGroups, mslists, "genform") # GenForm, default settings
formulasGF2 <- generateFormulas(fGroups, mslists, "genform", calculateFeatures = FALSE) #
↪  direct feature group assignment (faster)
formulasSIR <- generateFormulas(fGroups, mslists, "sirius", elements = "CHNOPSClBr") #
↪  SIRIUS, common elements for pollutant
formulasSIR2 <- generateFormulas(fGroups, mslists, "sirius", adduct = "[M-H]-") # SIRIUS,
↪  negative ionization
formulasBr <- generateFormulas(fGroups, mslists, "bruker", MSMode = "MSMS") # Only
↪  consider MSMS data (SmartFormula3D)
```

4.6.3 Compounds

An important step in a typical non-target workflow is structural identification for features of interest, as this information may finally reveal *what* a feature is. In a first step all possible candidate structures for a feature are obtained from a database (based on e.g. monoisotopic mass or formula). These candidates are then ranked, for instance, by matching the feature MS/MS data with in-silico or library MS/MS spectra or its relevance to the environment.

Structure assignment in `patRoon` is performed automatically for all feature groups with the `generateCompounds()` function. Currently, this function supports the following algorithms:

Algorithm	Usage	Remarks
MetFrag	`generateCompounds(algorithm = "metfrag", ...)`	Supports many databases (including offline and custom), matching MS/MS data with in-silico and library MS/MS data, and many other scorings to rank candidates.
SIRIUS with CSI:FingerID	`generateCompounds(algorithm = "sirius", ...)`	Matches with in-silico MS/MS data, incorporates formula annotations to improve candidate selection.
Library	`generateCompounds(algorithm = "library", ...)`	Obtains candidates by matching MS/MS data with an offline MS library, *e.g.* obtained from MassBank.eu or MoNA.

All algorithms rank their candidates by matching MS/MS data with in-silico generated MS/MS data (MetFrag and SIRIUS) and/or experimental MS/MS data from an MS library (MetFrag with MoNA scoring and Library algorithm). The latter may yield better candidates, and the Library algorithm is also generally much faster. However, in-silico annotation is not limited by the availability of experimental MS/MS data.

Compound annotation is often a relative time and resource intensive procedure. For this reason, annotation occurs for each feature group and not individual features. Nevertheless, it is not uncommon that this is the most time consuming step in the workflow. For this reason, prioritization of features is highly important, even more so to avoid 'abusing' servers when an online database is used for compound retrieval.

4.6.3.1 Database selection for MetFrag and SIRIUS Selecting the right database is important for proper candidate assignment. If the 'right' chemical compound is not present in the used database, it is impossible to assign the correct structure. Luckily, however, several large databases such as PubChem and ChemSpider are openly available which contain tens of millions of compounds. On the other hand, these databases may also lead to many unlikely candidates and therefore more specialized (or custom databases) may be preferred. Which database will be used is dictated by the `database` argument to `generateCompounds()`, currently the following options exist:

Database	Algorithm(s)	Remarks
`pubchem`	`"metfrag"`, `"sirius"`	PubChem is currently the largest compound database and is used by default.
`chemspider`	`"metfrag"`	ChemSpider is another large database. Requires security token from here (see next section).
`comptox`	`"metfrag"`	The EPA CompTox contains many compounds and scorings relevant to environmental studies. Needs manual download (see next section).
`pubchemlite`	`"metfrag"`	A specialized subset of the PubChem database. Needs manual download (see next section).
`for-ident`	`"metfrag"`	The FOR-IDENT (STOFF-IDENT) database for water related substances.
`kegg`	`"metfrag"`, `"sirius"`	The KEGG database for biological compounds
`hmdb`	`"metfrag"`, `"sirius"`	The HMDB contains many human metabolites.

Database	Algorithm(s)	Remarks
`bio`	`"sirius"`	Selects all supports biological databases.
`csv`, `psv`, `sdf`	`"metfrag"`	Custom database (see next section). CSV example.

4.6.3.2 Configuring MetFrag databases and scoring Some extra configuration may be necessary when using certain databases with MetFrag. In order to use the ChemSpider database a security token should be requested and set with the `chemSpiderToken` argument to `generateCompounds()`. The CompTox and PubChemLite databases need to be manually downloaded from CompTox (or variations with smoking or wastewater metadata) and PubChemLite (or the PubChem derived OECD PFAS database). The file location of this and other local databases (`csv`, `psv`, `sdf`) needs to be manually configured, see the examples below and/or `?generateCompounds` for more information on how to do this.

```
# PubChem: the default
compsMF <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+")

# ChemSpider: needs security token
compsMF2 <- generateCompounds(fGroups, mslists, "metfrag", database = "chemspider",
                              chemSpiderToken = "MY_TOKEN_HERE", adduct = "[M+H]+")

# CompTox: set global option to database path
options(patRoon.path.MetFragCompTox = "~/CompTox_17March2019_SelectMetaData.csv")
compsMF3 <- generateCompounds(fGroups, mslists, "metfrag", database = "comptox", adduct =
↪  "[M+H]+")

# CompTox: set database location without global option
compsMF4 <- generateCompounds(fGroups, mslists, "metfrag", database = "comptox", adduct =
↪  "[M+H]+",
                              extraOpts = list(LocalDatabasePath =
                              ↪  "~/CompTox_17March2019_SelectMetaData.csv"))

# Same, but for custom database
compsMF5 <- generateCompounds(fGroups, mslists, "metfrag", database = "csv", adduct =
↪  "[M+H]+",
                              extraOpts = list(LocalDatabasePath = "~/mydb.csv"))
```

An example of a custom *.csv* database can be found here.

With MetFrag compound databases are not only used to retrieve candidate structures but are also used to obtain metadata for further ranking. Each database has its own scorings, a table with currently supported scorings can be obtained with the `compoundScorings()` function (some columns omitted):

name	metfrag	database	default
score	Score		TRUE
fragScore	FragmenterScore		TRUE
metFusionScore	OfflineMetFusionScore		TRUE
individualMoNAScore	OfflineIndividualMoNAScore		TRUE
numberPatents	PubChemNumberPatents	pubchem	TRUE
numberPatents	Patent_Count	pubchemlite	TRUE
pubMedReferences	PubChemNumberPubMedReferences	pubchem	TRUE
pubMedReferences	ChemSpiderNumberPubMedReferences	chemspider	TRUE
pubMedReferences	NUMBER_OF_PUBMED_ARTICLES	comptox	TRUE
pubMedReferences	PubMed_Count	pubchemlite	TRUE

extReferenceCount	ChemSpiderNumberExternalReferences	chemspider	TRUE
dataSourceCount	ChemSpiderDataSourceCount	chemspider	TRUE
referenceCount	ChemSpiderReferenceCount	chemspider	TRUE
RSCCount	ChemSpiderRSCCount	chemspider	TRUE
formulaScore			FALSE
RF_SMILES			FALSE
RF_SIRFP			FALSE
LC50_SMILES			FALSE
LC50_SIRFP			FALSE
smartsInclusionScore	SmartsSubstructureInclusionScore		FALSE
smartsExclusionScore	SmartsSubstructureExclusionScore		FALSE
suspectListScore	SuspectListScore		FALSE
retentionTimeScore	RetentionTimeScore		FALSE
CPDATCount	CPDAT_COUNT	comptox	TRUE
TOXCASTActive	TOXCAST_PERCENT_ACTIVE	comptox	TRUE
dataSources	DATA_SOURCES	comptox	TRUE
pubChemDataSources	PUBCHEM_DATA_SOURCES	comptox	TRUE
EXPOCASTPredExpo	EXPOCAST_MEDIAN_EXPOSURE_PREDICTION_MG/KG-BW/DAY	comptox	TRUE
ECOTOX	ECOTOX	comptox	TRUE
NORMANSUSDAT	NORMANSUSDAT	comptox	TRUE
MASSBANKEU	MASSBANKEU	comptox	TRUE
TOX21SL	TOX21SL	comptox	TRUE
TOXCAST	TOXCAST	comptox	TRUE
KEMIMARKET	KEMIMARKET	comptox	TRUE
MZCLOUD	MZCLOUD	comptox	TRUE
pubMedNeuro	PubMedNeuro	comptox	TRUE
CIGARETTES	CIGARETTES	comptox	TRUE
INDOORCT16	INDOORCT16	comptox	TRUE
SRM2585DUST	SRM2585DUST	comptox	TRUE
SLTCHEMDB	SLTCHEMDB	comptox	TRUE
THSMOKE	THSMOKE	comptox	TRUE
ITNANTIBIOTIC	ITNANTIBIOTIC	comptox	TRUE
STOFFIDENT	STOFFIDENT	comptox	TRUE
KEMIMARKET_EXPO	KEMIMARKET_EXPO	comptox	TRUE
KEMIMARKET_HAZ	KEMIMARKET_HAZ	comptox	TRUE
REACH2017	REACH2017	comptox	TRUE
KEMIWW_WDUIndex	KEMIWW_WDUIndex	comptox	TRUE
KEMIWW_StpSE	KEMIWW_StpSE	comptox	TRUE
KEMIWW_SEHitsOverDL	KEMIWW_SEHitsOverDL	comptox	TRUE
ZINC15PHARMA	ZINC15PHARMA	comptox	TRUE
PFASMASTER	PFASMASTER	comptox	TRUE
peakFingerprintScore	AutomatedPeakFingerprintAnnotationScore		FALSE
lossFingerprintScore	AutomatedLossFingerprintAnnotationScore		FALSE
agroChemInfo	AgroChemInfo	pubchemlite	FALSE
bioPathway	BioPathway	pubchemlite	FALSE
drugMedicInfo	DrugMedicInfo	pubchemlite	FALSE
foodRelated	FoodRelated	pubchemlite	FALSE
pharmacoInfo	PharmacoInfo	pubchemlite	FALSE
safetyInfo	SafetyInfo	pubchemlite	FALSE
toxicityInfo	ToxicityInfo	pubchemlite	FALSE
knownUse	KnownUse	pubchemlite	FALSE
disorderDisease	DisorderDisease	pubchemlite	FALSE
identification	Identification	pubchemlite	FALSE
annoTypeCount	FPSum	pubchemlite	TRUE
annoTypeCount	AnnoTypeCount	pubchemlite	TRUE
annotHitCount	AnnotHitCount	pubchemlite	TRUE
libMatch			TRUE

The first two columns contain the generic and original MetFrag naming schemes for each scoring type. While both naming schemes can be used, the generic is often shorter and harmonized with other algorithms (e.g. SIRIUS). The *database* column specifies for which databases a particular scoring is available (empty if not database specific). Most scorings are selected by default (as specified by the *default* column), however, this behaviour can be customized by using the `scoreTypes` argument:

```
# Only in-silico and PubChem number of patents scorings
compsMF1 <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+",
                              scoreTypes = c("fragScore" "numberPatents"))

# Custom scoring in custom database
compsMF2 <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+",
                              database = "csv",
                              extraOpts = list(LocalDatabasePath = "~/mydb.csv"),
                              scoreTypes = c("fragScore", "myScore", "myScore2"))
```

By default ranking is performed with equal weight (i.e. *1*) for all scorings. This can be changed by the `scoreWeights` argument, which should be a `vector` containing the weights for all scorings following the order of `scoreTypes`, for instance:

```
compsMF <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+",
                             scoreTypes = c("fragScore" "numberPatents"),
                             scoreWeights = c(1, 2))
```

Sometimes thousands or more structural candidates are found when annotating a feature group. In this situation processing all these candidates will too involving (especially when external databases are used). To avoid this a default cut-off is set: when the number of candidates exceed a certain amount the search will be aborted and no results will be reported for that feature group. The maximum number of candidates can be set with the `maxCandidatesToStop` argument. The default value is relative conservative, especially for local databases it may be useful to increase this number.

4.6.3.3 MetFrag error and timeout handling

The use of online databases has the drawback that an error may occur, for instance, as a result of a connection error or when the aforementioned maximum number of candidates is reached (`maxCandidatesToStop` argument). By default, the processing is restarted if an error has occurred (configured by the `errorRetries` argument). Similarly, the `timeoutRetries` and `timeout` arguments can be used to avoid being 'stuck' on obtaining results, for instance, due to an unstable internet connection. If no compounds could be assigned due to an error a warning will be issued. In this case it is best to see what went wrong by manually checking the log files, which by default are stored in the *log/metfrag* folder.

4.6.3.4 Annotation with the *Library* algorithm

To use the *Library* algorithm we first need to load an MS library. Currently, MS libraries in the MSP and MoNA JSON formats are supported. Note that the former format is not so well standardized, and the support in `patRoon` was mainly tailored for MSP files from MassBank.eu and MoNA. To load the MS library the `loadMSLibrary()` function is used:

```
mslibrary <- loadMSLibrary("~/MassBank_NIST.msp", "msp") # MassBank.eu MSP library
mslibrary <- loadMSLibrary("~/MoNA-export-CASMI_2016.msp", "msp") # MoNA MSP library
mslibrary <- loadMSLibrary("~/MoNA-export-MassBank.json", "json") # MoNA JSON library
```

> ***NOTE*** Currently it is only possible to load formula annotated MS/MS peaks with the MoNA JSON format.

Once loaded, the MS library can be post-processed with various filtering, subsetting and export functionality, which may be useful for more tailored compound annotation. This is further discussed in the advanced chapter.

The compound annotation is performed with `generateCompounds()`:

```
compsLib <- generateCompounds(fGroups, mslists, "library", MSLibrary = mslibrary)

# set minimum MS/MS spectral match for candidates to 0.5
compsLib <- generateCompounds(fGroups, mslists, "library", MSLibrary = mslibrary, minSim
↪  = 0.5)
```

4.6.3.5 Formula scoring Ranking of candidate structures may further be improved by incorporating formula information by using the `addFormulaScoring()` function:

```
comps <- addFormulaScoring(coms, formulas, updateScore = TRUE)
```

Here, corresponding formula and explained fragments will be used to calculate a *formulaScore* for each candidate. Note that SIRIUS candidates are already based on calculated formulae, hence, running this function on SIRIUS results is less sensible unless scoring from another formula calculation algorithm is desired.

4.6.3.6 Further options and parameters There are *many* more options and parameters that affect compound annotation. For a full overview please have a look at the reference manual (e.g. by running `?generateCompounds`).

4.6.4 Suspect annotation

The data obtained during the previously described annotation steps can be used to improve a suspect screening workflow. The `annotateSuspects()` method uses the annotation data to calculate various annotation properties for each suspect, such as their rank in formula/compound candidates, which fragments from the suspect list were matched, and a *rough* indication of the identification level according to Schymanski et al. (2014)

```
fGroupsSusp <- annotateSuspects(fGroupsSusp, MSPeakLists = mslists,
                                formulas = formulas, compounds = compounds)
```

The calculation of identification levels is performed by a set of pre-defined rules. The `genIDLevelRulesFile()` can be used to inspect the default rules or to create your own rules file, which can subsequently passed to `annotateSuspects()` with the `IDFile` argument. See `?annotateSuspects` for more details on the file format and options. The default identification levels can be summarized as follows:

Level	Description	Rules
1	Target match	Retention time deviates <12 seconds from suspect list. At least `3` (or all if the suspect list contains less) fragments from the suspect list must match.
2a	Good MS/MS library match	Suspect is top ranked in the `compounds` results. The `individualMoNAScore` (MetFrag) or `libMatch` (Library algorithm) is at least `0.9` and no other candidates were matched with the MS library.
3a	Fair library match	The `individualMoNAScore` or `libMatch` is at least 0.4.
3b	Known MS/MS match	At least `3` (or all if the suspect list contains less) fragments from the suspect list must match.
3c	Good in-silico MS/MS match	The annotation MS/MS similarity (`annSimComp` column) is at least `0.7`.

Level	Description	Rules
4a	Good formula MS/MS match	Suspect is top ranked formula candidate, annotation MS/MS similarity (`annSimForm` column) is at least `0.7` and isotopic match (`isoScore`) of at least 0.5. The latter two scores are at least `0.2` higher than next best ranked candidate.
4b	Good formula isotopic pattern match	Suspect is top ranked formula candidate and isotopic match (`isoScore`) of at least `0.9` and at least `0.2` higher than next best ranked candidate.
5	Unknown	All else.

In general, the more data provided by the suspect list and to `annotateSuspects()`, the better identification level estimation works. For instance, when considering the default rules, either the `fragments_mz` or `fragments_formula` column is necessary to be able assign a `level 3b`. Similarly, the suspect list needs retention times (as well as fragment data) to be able to assign `level 1`. As you can imagine, providing the annotation workflow objects (i.e. `MSPeakLists`, `formulas`, `compounds`) to `annotateSuspects()` is necessary for calculation of most levels.

The `annotateSuspects()` function will log decisions for identification level assignments to the `log/` subdirectory in the current working directory. This is useful to inspect level assignments and especially useful when you customized any rules.

> ***NOTE***: The current identification level rules are *only* optimized for GenForm and MetFrag annotation algorithms.

4.6.5 Account login for SIRIUS

Recent version of SIRIUS require an active account login to make queries to CSI:FingerID. This is primarily relevant when performing a compound annotation workflow with `SIRIUS` or a formula annotation workflow with `getFingerprints=TRUE`, e.g. when predicting toxicities or concentrations.

As a first step, please create an account as described in the SIRIUS documentation: https://v6.docs.sirius-ms.io/account-and-license/.

Then, to login there are two options:

1. Manually login: either by using the SIRIUS GUI or the CLI. For the latter, see e.g. `sirius.exe login --help` for more details.
2. Let `patRoon` automatically handle logins.

The `login` parameter for `generateCompounds()` and `generateFormulas()` determines how logins are dealt with by `patRoon`. There are four options:

1. `login=FALSE`: *no* logins are performed and *no* checks are performed to verify if there is an existing login.
2. `login="check"`: *no* logins are performed, but an active login is required to proceed.
3. `login="interactive"`: if no active login is present, then the username and password will be asked interactively and used to automatically login.
4. `login=c(username="...", password="...")`: if no active login is present, then the provided username and password will be used to automatically login.

> ***NOTE***: For the fourth option, please don't provide the login details directly as plain-text for security reasons. See below for proper alternatives.

The first two options are primarily meant for manual login. The function parameter `alwaysLogin=TRUE` can be set to force a login for the third and fourth options.

The fourth option is primarily useful for e.g. heavy users of SIRIUS or unattended automatic workflows. To securely provide the login details, it is best to store them elsewhere. This webpage provides a detailed overview of how credentials can be safely stored. For instance, you can save the credentials in your `.Renviron` file and retreieve them when calling `generateCompounds()`:

In your `.Renviron` file add:

```
SIRIUS_USERNAME=MY_USERNAME
SIRIUS_PASSWORD=MY_PASSWORD
```

and then in your `R` script:

```
compounds <- generateCompounds(..., login = c(username = Sys.getenv("SIRIUS_USERNAME"),
                                               password = Sys.getenv("SIRIUS_PASSWORD")))
```

Alternatively, you could use the keyring package, e.g.

```
install.packages("keyring") # execute in case you don't have keyring installed yet
keyring::key_set("SIRIUS", username = "myaccount@email.com") # execute this once to store
↪  the password

compounds <- generateCompounds(..., login = c(username = "myaccount@email.com",
                                               password = keyring::key_get("SIRIUS",
                                               ↪  "myaccount@email.com")))
```

5 Processing workflow data

The previous chapter mainly discussed how to create workflow data. This chapter will discuss how to *use* the data.

5.1 Inspecting results

Several generic functions exist that can be used to inspect data that is stored in a particular object (e.g. features, compounds etc):

Generic	Classes	Remarks
`length()`	All	Returns the length of the object (e.g. number of features, compounds etc)
`algorithm()`	All	Returns the name of the algorithm used to generate the object.
`groupNames()`	All	Returns all the unique identitifiers (or names) of the feature groups for which this object contains results.

Generic	Classes	Remarks
`names()`	`featureGroups`, `components`	Returns names of the feature groups (similar to `groupNames()`) or components
`show()`	All	Prints general information.
`"[["` / `"$"` operators	All	Extract general information, see below.
`as.data.table()` / `as.data.frame()`	All	Convert data to a `data.table` or `data.frame`, see below.
`analysisInfo()`, `analyses()`, `replicateGroups()`	`features`, `featureGroups`	Returns the analysis information, analyses or replicate groups for which this object contains data.
`groupInfo()`	`featureGroups`	Returns feature group information (m/z and retention time values).
`screenInfo()`	`featureGroupsScreening`	Returns information on hits from suspect screening.
`componentInfo()`	`components`	Returns information for all components.
`annotatedPeakList()`	`formulas`, `compounds`	Returns a table with annotated mass peaks (see below).

The common `R` extraction operators `"[["`, `"$"` can be used to obtain data for a particular feature groups, analysis etc:

```
# Feature table (only first columns for readability)
fList[["standard-1"]][, 1:6]
```

```
#> NULL
```

```
# Feature group intensities
fGroups$M120_R268_30
```

```
#> [1] 264836 245372 216560
```

```
fGroups[[1, "M120_R268_30"]] # only first analysis
```

```
#> [1] 264836
```

```
# obtains MS/MS peak list  (feature group averaged data)
mslists[["M120_R268_30"]]$MSMS
```

```
#>        ID       mz  intensity precursor
#>     <int>    <num>      <num>    <lgcl>
#>  1:     5 105.0698   6183.111     FALSE
#>  2:     6 106.0653   7643.556     FALSE
#>  3:     8 107.0728   7760.667     FALSE
#>  4:    15 120.0556 168522.667      TRUE
#>  5:    17 121.0587  13894.667     FALSE
#>  6:    18 121.0884  10032.889     FALSE
```

```
#>  7:    19 122.0964 147667.778     FALSE
#>  8:    20 123.0803  36631.111     FALSE
#>  9:    21 123.0996  15482.444     FALSE
#> 10:    22 124.0805  35580.667     FALSE
```

```
# get all formula candidates for a feature group
formulas[["M120_R268_30"]][, 1:7]
```

```
#>    neutral_formula ion_formula neutralMass ion_formula_mz error   dbe isoScore
#>              <char>      <char>       <num>          <num> <num> <num>    <num>
#> 1:          C6H5N3      C6H6N3    119.0483       120.0556   1.8     6  0.92461
```

```
# get all compound candidates for a feature group
compounds[["M120_R268_30"]][, 1:4]
```

```
#>     explainedPeaks     score neutralMass                   SMILES
#>              <int>     <num>       <num>                   <char>
#>  1:              0 2.9919045    119.0483        C1=CC2=NNN=C2C=C1
#>  2:              0 1.2504308    119.0483        C1=CNC2=CN=CN=C21
#>  3:              0 1.2336169    119.0483      C1=CC2=C(N=C1)N=CN2
#>  4:              0 1.2079701    119.0483      C1=CC2=C(C=NN2)N=C1
#>  5:              0 1.1511570    119.0483      C1=CN2C(=CC=N2)N=C1
#> ---
#> 37:              0 0.9541662    119.0483        CC1=CN=C(N=C1)C#N
#> 38:              0 0.9535093    119.0483        CC1=NC(=NC=C1)C#N
#> 39:              0 0.9499092    119.0483        CC1=NN=C(C=C1)C#N
#> 40:              0 0.8128595    119.0483 C1=CC(=[N+]=[N-])C=CC1=N
#> 41:              0 0.7438038    119.0483         C(C#N)C(CC#N)C#N
```

```
# get a table with information of a component
components[["CMP7"]][, 1:6]
```

```
#>            group     ret       mz isogroup isonr charge
#>           <char>   <num>    <num>    <num> <num>  <num>
#> 1:  M143_R206_64 205.787 143.0700       NA    NA     NA
#> 2: M159_R208_103 208.280 159.0650       NA    NA     NA
#> 3: M161_R208_104 207.582 161.0806       NA    NA     NA
#> 4: M181_R209_159 208.580 181.0469       NA    NA     NA
```

A more sophisticated way to obtain data from a workflow object is to use `as.data.table()` or `as.data.frame()`. These functions will convert *all* information within the object to a table (`data.table` or `data.frame`) and allow various options to add extra information. An advantage is that this common data format can be used with many other functions within `R`. The output is in a tidy format.

> ***NOTE*** If you are not familiar with `data.table` and want to know more see data.table. Briefly, this is a more efficient and largely compatible alternative to the regular `data.frame`.

> ***NOTE*** The `as.data.frame()` methods defined in `patRoon` simply convert the results from `as.data.table()`, hence, both functions are equal in their usage and are defined for the same object classes.

Some typical examples are shown below.

```
# obtain table with all features (only first columns for readability)
as.data.table(fList)[, 1:6]
```

```
#>             analysis                     ID     ret        mz      area intensity
#>               <char>                 <char>   <num>     <num>     <num>     <num>
#>    1:  solvent-pos-1  f_1111264393868911102  13.176  98.97537 4345232.0    391476
#>    2:  solvent-pos-1  f_2094520183844760528   7.181 100.11197  797112.1    426956
#>    3:  solvent-pos-1  f_7941927790353320269 192.178 100.11211 9609998.0    750532
#>    4:  solvent-pos-1 f_16909335299782523620  19.171 100.11217 5784411.0    370376
#>    5:  solvent-pos-1  f_6221034045034627155   4.786 100.11220  551723.6    567312
#>   ---
#> 2922: standard-pos-3 f_11654918892462341096 318.892 425.18866  666531.5    232636
#> 2923: standard-pos-3 f_15952415959529491179   9.114 427.03242  362024.1    114744
#> 2924: standard-pos-3 f_12161494855555353140 318.892 427.18678  200193.5     77768
#> 2925: standard-pos-3 f_13430643112055510340 382.682 432.23984  217612.9     97648
#> 2926: standard-pos-3 f_12072347617435122911   9.114 433.00457 3086864.0    912920
```

```
# Returns group info and intensity values for each feature group
as.data.table(fGroups, average = TRUE) # average intensities for replicates
```

```
#>              group      ret       mz standard-pos
#>             <char>    <num>    <num>        <num>
#>   1:  M109_R192_20 191.8717 109.0759    183482.67
#>   2:  M111_R330_23 330.4078 111.0439     84598.67
#>   3:  M114_R269_25 268.6906 114.0912     85796.00
#>   4:  M116_R317_29 316.7334 116.0527    766888.00
#>   5:  M120_R268_30 268.4078 120.0554    242256.00
#>  ---
#> 137: M316_R363_635 363.4879 316.1741     89904.00
#> 138: M318_R349_638 349.1072 318.1450     83320.00
#> 139: M352_R335_664 334.9403 352.2019     74986.67
#> 140: M407_R239_672 239.3567 407.2227    186568.00
#> 141: M425_R319_676 319.4944 425.1885    214990.67
```

```
# As above, but with suspect matches on separate rows and additional screening
↪  information
# (select some columns to simplify the output below)
as.data.table(fGroupsSusp, average = TRUE, collapseSuspects = NULL,
              onlyHits = TRUE)[, c("group", "susp_name", "susp_compRank",
              ↪  "susp_annSimBoth", "susp_estIDLevel")]
```

```
#>           group          susp_name susp_compRank susp_annSimBoth susp_estIDLevel
#>          <char>             <char>         <int>           <num>          <char>
#> 1: M120_R268_30   1H-benzotriazole             1       0.0000000              4b
#> 2: M137_R249_53      N-Phenyl urea             1       0.6443557              3a
#> 3: M146_R309_68 2-Hydroxyquinoline             2       0.9896892              3a
#> 4: M146_R248_69 2-Hydroxyquinoline            NA              NA               5
#> 5: M146_R225_70 2-Hydroxyquinoline            NA              NA               5
```

```
# Returns all peak lists for each feature group
as.data.table(mslists)
```

```
#>              group   type    ID       mz  intensity precursor
#>             <char> <char> <int>    <num>      <num>    <lgcl>
#>   1:  M120_R268_30     MS     1 100.1120 178952.381     FALSE
#>   2:  M120_R268_30     MS     2 102.1277 202359.667     FALSE
#>   3:  M120_R268_30     MS     3 114.0912  37647.548     FALSE
#>   4:  M120_R268_30     MS     4 115.0752  66685.238     FALSE
#>   5:  M120_R268_30     MS     5 120.0554 113335.857      TRUE
#>  ---
#> 235: M192_R355_191     MS    51 299.1274  44083.126     FALSE
#> 236: M192_R355_191     MS    52 299.1471   7390.267     FALSE
#> 237: M192_R355_191   MSMS    14 119.0496 588372.444     FALSE
#> 238: M192_R355_191   MSMS    18 120.0524  70273.333     FALSE
#> 239: M192_R355_191   MSMS    31 192.1384  71978.667      TRUE
```

```
# Returns all formula candidates for each feature group with scoring
# information, neutral loss etc
as.data.table(formulas)[, 1:6]
```

```
#>            group neutral_formula ion_formula neutralMass ion_formula_mz      error
#>           <char>          <char>      <char>       <num>          <num>      <num>
#> 1:  M120_R268_30          C6H5N3      C6H6N3    119.0483       120.0556 1.80000000
#> 2:  M137_R249_53         C7H8N2O     C7H9N2O    136.0637       137.0709 2.90000000
#> 3:  M146_R309_68          C9H7NO      C9H8NO    145.0528       146.0600 1.66666667
#> 4: M192_R355_191        C12H17NO    C12H18NO    191.1310       192.1383 0.03333333
```

```
# Returns all compound candidates for each feature group with scoring and other metadata
as.data.table(compounds)[, 1:4]
```

```
#>              group explainedPeaks    score neutralMass
#>             <char>          <int>    <num>       <num>
#>   1:  M120_R268_30              0 2.991905    119.0483
#>   2:  M120_R268_30              0 1.250431    119.0483
#>   3:  M120_R268_30              0 1.233617    119.0483
#>   4:  M120_R268_30              0 1.207970    119.0483
#>   5:  M120_R268_30              0 1.151157    119.0483
#>  ---
#> 288: M192_R355_191              1 1.367332    191.1310
#> 289: M192_R355_191              1 1.367220    191.1310
#> 290: M192_R355_191              1 1.366424    191.1310
#> 291: M192_R355_191              1 1.364403    191.1310
#> 292: M192_R355_191              1 1.363116    191.1310
```

```
# Returns table with all components (including feature group info, annotations etc)
as.data.table(components)[, 1:6]
```

```
#>       name  cmp_ret cmp_retsd       neutral_mass       analysis  size
#>     <char>    <num>     <num>             <char>         <char> <int>
```

```
#>  1:   CMP1 347.2914 0.0000000              <NA> standard-pos-2     2
#>  2:   CMP1 347.2914 0.0000000              <NA> standard-pos-2     2
#>  3:   CMP2 349.6328 4.6804985 225.1589/188.20157 standard-pos-3     6
#>  4:   CMP2 349.6328 4.6804985 225.1589/188.20157 standard-pos-3     6
#>  5:   CMP2 349.6328 4.6804985 225.1589/188.20157 standard-pos-3     6
#> ---
#> 88:  CMP29 313.3475 0.3105035              <NA> standard-pos-2     3
#> 89:  CMP29 313.3475 0.3105035              <NA> standard-pos-2     3
#> 90:  CMP30 268.3430 0.3840764          81.08705 standard-pos-1     3
#> 91:  CMP30 268.3430 0.3840764          81.08705 standard-pos-1     3
#> 92:  CMP30 268.3430 0.3840764          81.08705 standard-pos-1     3
```

Finally, the `annotatedPeakList()` function is useful to inspect annotation results for a formula or compound candidate:

```
# formula annotations for the first formula candidate of feature group M137_R249_53
annotatedPeakList(formulas, index = 1, groupName = "M137_R249_53",
                  MSPeakLists = mslists)
```

```
#>        ID        mz intensity precursor ion_formula   dbe ion_formula_mz error neutral_loss annotate
#>     <int>     <num>     <num>    <lgcl>      <char> <num>          <num> <num>       <char>    <lgcl
#>  1:     2  94.06500  9406.111     FALSE       C6H8N   3.5       94.06513  1.30         CHNO      TRU
#>  2:     6  98.97522  2212.000     FALSE        <NA>    NA             NA    NA         <NA>     FALS
#>  3:     7 105.06971  1662.111     FALSE        <NA>    NA             NA    NA         <NA>     FALS
#>  4:    14 120.04434  7176.222     FALSE      C7H6NO   5.5      120.04439  0.40          H3N      TRU
#>  5:    19 122.07222  2246.000     FALSE        <NA>    NA             NA    NA         <NA>     FALS
#>  6:    21 135.08004  1565.556     FALSE        <NA>    NA             NA    NA         <NA>     FALS
#>  7:    23 137.07039  5348.667      TRUE     C7H9N2O   4.5      137.07094  3.35                   TRU
#>  8:    24 137.09572  2026.889     FALSE        <NA>    NA             NA    NA         <NA>     FALS
#>  9:    26 138.09116 12356.667     FALSE        <NA>    NA             NA    NA         <NA>     FALS
#> 10:    27 139.07503  5020.667     FALSE        <NA>    NA             NA    NA         <NA>     FALS
```

```
# compound annotation for first candidate of feature group M137_R249_53
annotatedPeakList(compounds, index = 1, groupName = "M137_R249_53",
                  MSPeakLists = mslists)
```

```
#>        ID        mz intensity precursor ion_formula ion_formula_MF neutral_loss score annotated
#>     <int>     <num>     <num>    <lgcl>      <char>         <char>       <char> <num>    <lgcl>
#>  1:     2  94.06500  9406.111     FALSE       C6H8N   [C6H6N+H]+H+         CHNO   405      TRUE
#>  2:     6  98.97522  2212.000     FALSE        <NA>           <NA>         <NA>    NA     FALSE
#>  3:     7 105.06971  1662.111     FALSE        <NA>           <NA>         <NA>    NA     FALSE
#>  4:    14 120.04434  7176.222     FALSE      C7H6NO      [C7H6NO]+          H3N   305      TRUE
#>  5:    19 122.07222  2246.000     FALSE        <NA>           <NA>         <NA>    NA     FALSE
#>  6:    21 135.08004  1565.556     FALSE        <NA>           <NA>         <NA>    NA     FALSE
#>  7:    23 137.07039  5348.667      TRUE        <NA>           <NA>         <NA>    NA     FALSE
#>  8:    24 137.09572  2026.889     FALSE        <NA>           <NA>         <NA>    NA     FALSE
#>  9:    26 138.09116 12356.667     FALSE        <NA>           <NA>         <NA>    NA     FALSE
#> 10:    27 139.07503  5020.667     FALSE        <NA>           <NA>         <NA>    NA     FALSE
```

More advanced examples for these functions are shown below.

```
# Feature table, can also be accessed by numeric index
fList[[1]]
mslists[["standard-1", "M120_R268_30"]] # feature data (instead of feature group
↪  averaged)
formulas[[1, "M120_R268_30"]] # feature data (if available, i.e. calculateFeatures=TRUE)
components[["CMP1", 1]] # only for first feature group in component

as.data.frame(fList) # classic data.frame format, works for all objects
as.data.table(fGroups) # return non-averaged intensities (default)
as.data.table(fGroups, features = TRUE) # include feature information
as.data.table(mslists, averaged = FALSE) # peak lists for each feature
as.data.table(mslists, fGroups = fGroups) # add feature group information

as.data.table(formulas, countElements = c("C", "H")) # include C/H counts (e.g. for van
↪  Krevelen plots)
# add various information for organic matter characterization (common elemental
# counts/ratios, classifications etc)
as.data.table(formulas, OM = TRUE)

as.data.table(compounds, fGroups = fGroups) # add feature group information
as.data.table(compounds, fragments = TRUE) # include information of all annotated
↪  fragments

annotatedPeakList(formulas, index = 1, groupName = "M120_R268_30",
                  MSPeakLists = mslists, onlyAnnotated = TRUE) # only include annotated
                  ↪  peaks
annotatedPeakList(compounds, index = 1, groupName = "M120_R268_30",
                  MSPeakLists = mslists, formulas = formulas) # include formula
                  ↪  annotations
```

5.2 Filtering

During a non-target workflow it is not uncommon that some kind of data-cleanup is necessary. Datasets are often highly complex, which makes separating data of interest from the rest highly important. Furthermore, general cleanup typically improves the quality of the dataset, for instance by removing low scoring annotation results or features that are unlikely to be 'correct' (e.g. noise or present in blanks). For this reason `patRoon` supports *many* different filters that easily clean data produced during the workflow in a highly customizable way.

All major workflow objects (e.g. `featureGroups`, `compounds`, `components` etc.) support filtering operations by the `filter()` generic. This function takes the object to be filtered as first argument and any remaining arguments describe the desired filter options. The `filter()` generic function then returns the modified object back. Some examples are shown below.

```
# remove low intensity (<500) features
features <- filter(features, absMinIntensity = 500)

# remove features with intensities lower than 5 times the blank
fGroups <- filter(fGroups, blankThreshold = 5)

# only retain compounds with >1 explained MS/MS peaks
compounds <- filter(compounds, minExplainedPeaks = 1)
```

The following sections will provide a more detailed overview of available data filters.

> ***NOTE*** Some other `R` packages (notably `dplyr`) also provide a `filter()` generic function. To use the `filter()` function from different packages you may need to explicitly specify which one to use in your script. This can be done by prefixing it with the package name, e.g. `patRoon::filter(...)`, `dplyr::filter(...)` etc.

5.2.1 Features

There are many filters available for feature data:

Filter	Classes	Remarks
`absMinIntensity`, `relMinIntensity`	`features`, `featureGroups`	Minimum intensity
`preAbsMinIntensity`, `preRelMinIntensity`	`featureGroups`	Minimum intensity prior to other filtering (see below)
`retentionRange`, `mzRange`, `mzDefectRange`, `chromWidthRange`	`features`, `featureGroups`	Filter by feature properties
`absMinAnalyses`, `relMinAnalyses`	`featureGroups`	Minimum feature abundance in all analyses
`absMinReplicates`, `relMinReplicates`	`featureGroups`	Minimum feature abundance in different replicates
`absMinFeatures`, `relMinFeatures`	`featureGroups`	Only keep analyses with at least this amount of features
`absMinReplicateAbundance`, `relMinReplicateAbundance`	`featureGroups`	Minimum feature abundance in a replicate group
`maxReplicateIntRSD`	`featureGroups`	Maximum relative standard deviation of feature intensities in a replicate group.
`blankThreshold`	`featureGroups`	Minimum intensity factor above blank intensity
`rGroups`	`featureGroups`	Only keep (features of) these replicate groups
`results`	`featureGroups`	Only keep feature groups with formula/compound annotations or componentization results

Application of filters to feature data is important for (environmental) non-target analysis. Especially blank and replicate filters (i.e. `blankThreshold` and `absMinReplicateAbundance`/`relMinReplicateAbundance`) are important filters and are highly recommended to always apply for cleaning up your dataset.

All filters are available for feature group data, whereas only a subset is available for feature objects. The main reason is that other filters need grouping of features between analyses. Regardless, in `patRoon` filtering feature data is less important, and typically only needed when the number of features are extremely large and direct grouping is undesired.

From the table above you can notice that many filters concern both *absolute* and *relative* data (i.e. as prefixed with `abs` and `rel`). When a relative filter is used the value is scaled between *0* and *1*. For instance:

```
# remove features not present in at least half of the analyses within a replicate group
fGroups <- filter(fGroups, relMinReplicateAbundance = 0.5)
```

An advantage of relative filters is that you will not have to worry about the data size involved. For instance, in the above example the filter always takes half of the number of analyses within a replicate group, even when replicate groups have different number of analyses.

Note that multiple filters can be specified at once. Especially for feature group data the order of filtering may impact the final results, this is explained further in the reference manual (i.e. `?`feature-filtering`).

Some examples are shown below.

```
# filter features prior to grouping: remove any features eluting before first 2 minutes
fList <- filter(fList, retentionRange = c(120, Inf))

# common filters for feature groups
fGroups <- filter(fGroups,
                  absMinIntensity = 500, # remove features <500 intensity
                  relMinReplicateAbundance = 1, # features should be in all analysis of
                  ↪  replicate groups
                  maxReplicateIntRSD = 0.75, # remove features with intensity RSD in
                  ↪  replicates >75%
                  blankThreshold = 5, # remove features <5x intensity of (average) blank
                  ↪  intensity
                  removeBlanks = TRUE) # remove blank analyses from object afterwards

# filter by feature properties
fGroups <- filter(fGroups,
                  mzDefectRange = c(0.8, 0.9),
                  chromWidthRange = c(6, 120))

# remove features not present in at least 3 analyses
fGroups <- filter(fGroups, absMinAnalyses = 3)

# remove features not present in at least 20% of all replicate groups
fGroups <- filter(fGroups, relMinReplicates = 0.2)

# only keep data present in replicate groups "repl1" and "repl2"
# all other features and analyses will be removed
fGroups <- filter(fGroups, rGroups = c("repl1", "repl2"))

# only keep feature groups with compound annotations
fGroups <- filter(fGroups, results = compounds)
# only keep feature groups with formula or compound annotations
fGroups <- filter(fGroups, results = list(formulas, compounds))
```

5.2.2 Suspect screening

Several additional filters are available for feature groups obtained with `screenSuspects()`:

Filter	Classes	Remarks
`onlyHits`	`featureGroupsScreening`	Only retain feature groups assigned to one or more suspects.
`selectHitsBy`	`featureGroupsScreening`	Select the feature group that matches best with a suspect (in case there are multiple).
`selectBestFGroups`	`featureGroupsScreening`	Select the suspect that matches best with a feature group (in case there are multiple).

Filter	Classes	Remarks
`maxLevel`, `maxFormRank`, `maxCompRank`	`featureGroupsScreening`	Only retain suspect hits with identification/annotation ranks below a threshold.
`minAnnSimForm`, `minAnnSimComp`, `minAnnSimBoth`	`featureGroupsScreening`	Remove suspect hits with annotation similarity scores below this value.
`absMinFragMatches`, `relMinFragMatches`	`featureGroupsScreening`	Only keep suspect hits with a minimum (relative) number of fragment matches from the suspect list.

> ***NOTE***: most filters only remove suspect hit results. Set `onlyHits=TRUE` to also remove any feature groups that end up without suspect hits.

The `selectHitsBy` and `selectBestFGroups` filters are useful to remove duplicate hits, i.e. the same suspect assigned to multiple feature groups or multiple suspects assigned to the same feature group, respectively. The former selects based on either best identification level (`selectHitsBy="level"`) or highest mean intensity (`selectHitsBy="intensity"`). The `selectBestFGroups` can only be `TRUE`/`FALSE` and always selects by best identification level.

Some examples are shown below.

```
# only keep feature groups assigned to at least one suspect
fGroupsSusp <- filter(fGroupsSusp, onlyHits = TRUE)
# remove duplicate suspect to feature group matches and keep the best
fGroupsSusp <- filter(fGroupsSusp, selectHitsBy = "level")
# remove suspect hits with ID levels >3 and make sure no feature groups
# are present without suspect hits afterwards
fGroupsSusp <- filter(fGroupsSusp, maxLevel = 3, onlyHits = TRUE)
```

5.2.3 Annotation

There are various filters available for handling annotation data:

Filter	Classes	Remarks
`absMSIntThr`, `absMSMSIntThr`, `relMSIntThr`, `relMSMSIntThr`	`MSPeakLists`	Minimum intensity of mass peaks
`topMSPeaks`, `topMSMSPeaks`	`MSPeakLists`	Only keep most intense mass peaks
`withMSMS`	`MSPeakLists`	Only keep results with MS/MS data
`minMSMSPeaks`	`MSPeakLists`	Only keep an MS/MS peak list if it contains a minimum number of peaks (excluding the precursor peak)
`annotatedBy`	`MSPeakLists`	Only keep MS/MS peaks that have formula or compound annotations
`minExplainedPeaks`	`formulas`, `compounds`	Minimum number of annotated mass peaks
`elements`, `fragElements`, `lossElements`	`formulas`, `compounds`	Restrain elemental composition
`topMost`	`formulas`, `compounds`	Only keep highest ranked candidates
`minScore`, `minFragScore`, `minFormulaScore`	`compounds`	Minimum compound scorings
`scoreLimits`	`formulas`, `compounds`	Minimum/Maximum scorings

Filter	Classes	Remarks
`OM`	`formulas`, `compounds`	Only keep candidates with likely elemental composition found in organic matter

Several intensity related filters are available to clean-up MS peak list data. For instance, the `topMSPeaks`/`topMSMSPeaks` filters provide a simple way to remove noisy data by only retaining a defined number of most intense mass peaks. Note that none of these filters will remove the precursor mass peak of the feature itself.

The filters applicable to formula and compound annotation generally concern minimal scoring or chemical properties. The former is useful to remove unlikely candidates, whereas the second is useful to focus on certain study specific chemical properties (e.g. known neutral losses).

Common examples are shown below.

```
# intensity filtering
mslists <- filter(mslists,
                  absMSIntThr = 500, # minimum MS mass peak intensity of 500
                  relMSMSIntThr = 0.1) # minimum MS/MS mass peak intensity of 10%

# only retain 10 most intens mass peaks
# (feature mass is always retained)
mslists <- filter(mslists, topMSPeaks = 10)

# remove MS/MS peaks without compound annotations
mslists <- filter(mslists, annotatedBy = compounds)

# remove MS/MS peaks not annotated by either a formula or compound candidate
mslists <- filter(mslists, annotatedBy = list(formulas, compounds))

# only keep formulae with 1-10 sulphur or phosphorus elements
formulas <- filter(formulas, elements = c("S1-10", "P1-10"))

# only keep candidates with MS/MS fragments that contain 1-10 carbons and 0-2 oxygens
formulas <- filter(formulas, fragElements = "C1-10O0-2")

# only keep candidates with CO2 neutral loss
formulas <- filter(formulas, lossElements = "CO2")

# only keep the 15 highest ranked candidates with at least 1 annotated MS/MS peak
compounds <- filter(compounds, minExplainedPeaks = 1, topMost = 15)

# minimum in-silico score
compounds <- filter(compounds, minFragScore = 10)

# candidate should be referenced in at least 1 patent
# (only works if database lists number of patents, e.g. PubChem)
compounds <- filter(compounds,
                    scoreLimits = list(numberPatents = c(1, Inf))
```

NOTE As of `patRoon 2.0` MS peak lists are **not** re-generated after a filtering operation (unless the `reAverage` parameter is explicity set to `TRUE`). The reason for this change is that re-averaging

invalidates any formula/compound annotation data (e.g. used for plotting and reporting) that were generated prior to the filter operation.

5.2.4 Components

Finally several filters are available for components:

Filter	Remarks
`size`	Minimum component size
`adducts`, `isotopes`	Filter features by adduct/istopes annotation
`rtIncrement`, `mzIncrement`	Filter homologs by retention/mz increment range

Note that these filters are only applied if the components contain the data the filter works on. For instance, filtering by adducts will *not* affect components obtained from homologous series.

As before, some typical examples are shown below.

```
# only keep components with at least 4 features
componInt <- filter(componInt, minSize = 4)

# remove all features from components are not annotated as an adduct
componRC <- filter(componRC, adducts = TRUE)

# only keep protonated and sodium adducts
componRC <- filter(componRC, adducts = c("[M+H]+", "[M+Na]+"))

# remove all features not recognized as isotopes
componRC <- filter(componRC, isotopes = FALSE)

# only keep monoisotopic mass
componRC <- filter(componRC, isotopes = 0)

# min/max rt/mz increments for homologs
componNT <- filter(componNT, rtIncrement = c(10, 30),
                   mzIncrement = c(16, 50))
```

NOTE As mentioned before, components are still in a relative young development phase and results should always be verified!

5.2.5 Negation

All filters support *negation*: if enabled all specified filters will be executed in an opposite manner. Negation may not be so commonly used, but allows greater flexibility which is sometimes needed for advanced filtering steps. Furthermore, it is also useful to specifically isolate the data that otherwise would have been removed. Some examples are shown below.

```
# keep all features/analyses _not_ present from replicate groups "repl1" and "repl2"
fGroups <- filter(fGroups, rGroups = c("repl1", "repl2"), negate = TRUE)
```

```
# only retain features with a mass defect outside 0.8-0.9
fGroups <- filter(fGroups, mzDefectRange = c(0.8, 0.9), negate = TRUE)

# remove duplicate suspect hits and only keep the _worst_ hit
fGroupsSusp <- filter(fGroupsSusp, selectHitsBy = "level", negate = TRUE)

# remove candidates with CO2 neutral loss
formulas <- filter(formulas, lossElements = "CO2", negate = TRUE)

# select 15 worst ranked candidates
compounds <- filter(compounds, topMost = 15, negate = TRUE)

# only keep components with <5 features
componInt <- filter(componInt, minSize = 5, negate = TRUE)
```

5.3 Subsetting

The previous section discussed the `filter()` generic function to perform various data cleaning operations. A more generic way to select data is by *subsetting*: here you can manually specify which parts of an object should be retained. Subsetting is supported for all workflow objects and is performed by the R subset operator (`"["`). This operator either subsets by one or two arguments, which are referred to as the `i` and `j` arguments.

Class	Argument `i`	Argument `j`	Remarks
`features`	analyses		
`featureGroups`	analyses	feature groups	
`MSPeakLists`	analyses	feature groups	peak lists for feature groups will be re-averaged when subset on analyses (by default)
`formulas`	feature groups		
`compounds`	feature groups		
`components`	components	feature groups	

For objects that support two-dimensional subsetting (e.g. `featureGroups`, `MSPeakLists`), either the `i` or `j` argument is optional. Furthermore, unlike subsetting a `data.frame`, the position of `i` and `j` does not change when only one argument is specified:

```
df[1, 1] # subset data.frame by first row/column
df[1] # subset by first column
df[1, ] # subset by first row

fGroups[1, 1] # subset by first analysis/feature group
fGroups[, 1] # subset by first feature group (i.e. column)
fGroups[1] # subset by first analysis (i.e. row)
```

The subset operator allows three types of input:

- A logical vector: elements are selected if corresponding values are `TRUE`.
- A numeric vector: select elements by numeric index.
- A character vector: select elements by their name.

When a logical vector is used as input it will be re-cycled if necessary. For instance, the following will select by the first, third, fifth, etc. analysis.

```
fGroups[c(TRUE, FALSE)]
```

In order to select by a `character` you will need to know the names for each element. These can, for instance, be obtained by the `groupNames()` (feature group names), `analyses()` (analysis names) and `names()` (names for components or feature groups for `featureGroups` objects) generic functions.

Some more examples of common subsetting operations are shown below.

```
# select first three analyses
fList[1:3]

# select first three analyses and first 500 feature groups
fGroups[1:3, 1:500]

# select all feature groups from first component
fGroupsNT <- fGroups[, componNT[[1]]$group]

# only keep feature groups with formula annotation results
fGroupsForms <- fGroups[, groupNames(formulas)]

# only keep feature groups with either formula or compound annotation results
fGroupsAnn <- fGroups[, union(groupNames(formulas), groupNames(compounds))]

# select first 15 components
components[1:15]

# select by name
components[c("CMP1", "CMP5")]

# only retain feature groups in components for which compound annotations are
# available
components[, groupNames(compounds)]
```

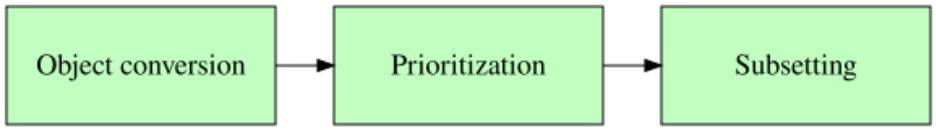
In addition, feature groups can also be subset by given replicate groups or annotation/componentization results (similar to `filter()`). Similarly, suspect screening results can also be subset by given suspect names.

```
# equal as filter(fGroups, rGroups = ...)
fGroups[rGroups = c("repl1", "repl2")]
# equal as filter(fGroups, results = ...)
fGroups[results = compounds]
# only keep feature groups assigned to given suspects
fGroupsSusp[suspects = c("1H-benzotriazole", "2-Hydroxyquinoline")]
```

> ***NOTE*** As of `patRoon 2.0` MS peak lists are **not** re-generated after a subsetting operation (unless the `reAverage` parameter is explicity set to `TRUE`). The reason for this change is that re-averaging invalidates any formula/compound annotation data (e.g. used for plotting and reporting) that were generated prior to the subset operation.

5.3.1 Prioritization workflow

An important use case of subsetting is prioritization of data. For instance, after statistical analysis only certain feature groups are deemed relevant for the rest of the workflow. A common prioritization workflow is illustrated below:



During the first step the workflow object is converted to a suitable format, most often using the `as.data.frame()` function. The converted data is then used as input for the prioritization strategy. Finally, these results are then used to select the data of interest in the original object.

A very simplified example of such a process is shown below.

```
featTab <- as.data.frame(fGroups, average = TRUE)

# prioritization: sort by (averaged) intensity of the "sample" replicate group
# (from high to low) and then obtain the feature group identifiers of the top 5.
featTab <- featTab[order(featTab$standard, decreasing = TRUE), ]
groupsOfInterest <- featTab$group[1:5]

# subset the original data
fGroups <- fGroups[, groupsOfInterest]

# fGroups now only contains the feature groups for which intensity values in the
# "sample" replicate group were in the top 5
```

5.4 Deleting data

The `delete()` generic function can be used to manually delete workflow data. This function is used internally within `patRoon` to implement filtering and subsetting operations, but may also be useful for advanced data processing.

Like the subset operator this function accepts a `i` and `j` parameter to specify which data should be operated on:

Class	Argument `i`	Argument `j`
`features`	analysis	feature index
`featureGroups`	analysis	feature group
`formulas`, `compounds`	feature group	candidate index
`components`	component	feature group

If `i` or `j` is not specified (`NULL`) then data is removed for the complete selection. Some examples are shown below:

```
# delete 2nd feature in analysis-1
fList <- delete(fList, i = "analysis-1", j = 2)
# delete first ten features in all analyses
fList <- delete(fList, i = NULL, j = 1:10)
```

```
# completely remove third/fourth analyses from feature groups
fGroups <- delete(fGroups, i = 3:4)
# delete specific feature group
fGroups <- delete(fGroups, j = "M120_R268_30")
# delete range of feature groups
fGroups <- delete(fGroups, j = 500:750)

# remove all results for a feature group
formulas <- delete(formulas, i = "M120_R268_30")

# remove top candidate for all feature groups
compounds <- delete(compounds, j = 1)

# remove a component
components <- delete(components, i = "CMP1")
# remove specific feature group from a component
components <- delete(components, i = "CMP1", j = "M120_R268_30")
# remove specific feature group from all components
components <- delete(components, j = "M120_R268_30")
```

The `j` parameter can also be a function: in this case it is called repeatedly on parts of the data to select what should be deleted. How the function is called and what it should return depends on the workflow data class:

Class	Called on every	First argument	Second argument	Return value
`features`	analysis	`data.table` with features	analysis name	Features indices (as `integer` or `logical`)
`featureGroups`	feature group	vector with group intensities	feature group name	The analyses of the features to remove (as `character`, `integer`, `logical`)
`formulas`, `compounds`	feature group	`data.table` with annotations	feature group name	Candidate indices (rows)
`components`	component	`data.table` with the component	component name	The feature groups (as `character`, `integer`)

Some examples for this:

```
# remove features with intensities below 5000
fList <- delete(fList, j = function(f, ...) f$intensity <= 5E3)

# same, but for features in all feature groups from specific analyses
fGroups <- delete(i = 1:3, j = function(g, ...) g <= 5E3)

# remove formula candidates with high relative mass deviation
formulas <- delete(formulas, j = function(ft, ...) ft$error > 5)
```

5.5 Unique and overlapping features

Often an analysis batch is composed of different sample groups, such as different treatments, influent/effluent etc. In such scenarios it may be highly interesting to evaluate uniqueness or overlap between these samples. Furthermore, extracting overlapping or unique features is a simple but effective prioritization strategy.

The `overlap()` and `unique()` functions can be used to extract overlapping and unique features between replicate groups, respectively. Both functions return a subset of the given `featureGroups` object. An overview of their arguments is given below.

Argument	Function(s)	Remarks
`which`	`unique()`, `overlap()`	The replicate groups to compare.
`relativeTo`	`unique()`	Only return unique features compared to these replicate groups (`NULL` for all). Replicate groups in `which` are ignored.
`outer`	`unique()`	If `TRUE` then only return features which are *also* unique among the compared replicates groups.
`exclusive`	`overlap`	Only keep features that *only* overlap between the compared replicate groups.

Some examples:

```
# only keep features uniquely present in replicate group "repl1"
fGroupsUn1 <- unique(fGroups, which = "repl1")
# only keep features in repl1/repl2 which are not in repl3
fGroupsUn2 <- unique(fGroups, which = c("repl1", "repl2"),
                     relativeTo = "repl3")
# only keep features that are only present in repl1 OR repl2
fGroupsUn3 <- unique(fGroups, which = c("repl1", "repl2"),
                     outer = TRUE)

# only keep features overlapping in repl1/repl2
fGroupsOv1 <- overlap(fGroups, which = c("repl1", "repl2"))
# only keep features overlapping in repl1/repl2 AND are not present in any other
# replicate group
fGroupsOv2 <- overlap(fGroups, which = c("repl1", "repl2"),
                      exclusive = TRUE)
```

In addition, several plotting functions are discussed in the visualization section that visualize overlap and uniqueness of features.

5.6 MS similarity

The *spectral similarity* is used to compare spectra from different features. For this purpose the `spectrumSimilarity` function can be used. This function operates on MS peak lists, and accepts the following function arguments:

Argument	Remarks
`MSPeakLists`	The MS peak lists object from which peak lists data should be taken.
`groupName1`, `groupName2`	The name(s) of the first and second feature group(s) to compare
`analysis1`, `analysis2`	The analysis names of the data to be compared. Set this when feature data (instead of feature group data) should be compared.
`MSLevel`	The MS level: `1` or `2` for MS and MS/MS, respectively.
`specSimParams`	Parameters that define how similarities are calculated.
`NAToZero`	If `TRUE` then `NA` values are converted to zeros. `NA` values are reported if a comparison cannot be made because of missing peak list data.

The `specSimParams` argument defines the parameters for similarity calculations. It is a `list`, and the default values are obtained with the `getDefSpecSimParams()` function:

```
getDefSpecSimParams()
```

```
#> $method
#> [1] "cosine"
#>
#> $removePrecursor
#> [1] FALSE
#>
#> $mzWeight
#> [1] 0
#>
#> $intWeight
#> [1] 1
#>
#> $absMzDev
#> [1] 0.005
#>
#> $relMinIntensity
#> [1] 0.05
#>
#> $minPeaks
#> [1] 1
#>
#> $shift
#> [1] "none"
#>
#> $setCombineMethod
#> [1] "mean"
```

The `method` field describes the calculation measure: this is either `"cosine"` or `"jaccard"`.

The `shift` field is primarily useful when comparing MS/MS data and defines if and how a spectral shift should be performed prior to similarity calculation:

- `"none"`: The default, no shifting is performed.
- `"precursor"` The mass difference between the precursor mass of both spectra (*i.e.* the feature mass) is first calculated. This difference is then subtracted from each of the mass peaks of the second spectrum. This shifting increases similarity if the MS fragmentation process itself occurs similarly (*i.e.* if both features show similar neutral losses).
- `"both"` This combines both shifting methods: first peaks are aligned that have the same mass, then the `precursor` strategy is applied for the remaining mass peaks. This shifting method yields higher similarities if either fragment masses or neutral losses are similar.

To override a default setting, simply pass it as an argument to `getDefSpecSimParams`:

```
getDefSpecSimParams(shift = "both")
```

For more details on the various similarity calculation parameters see the reference manual (`?getDefSpecSimParams`).

Some examples are shown below:

```
# similarity between MS spectra with default parameters
spectrumSimilarity(mslists, groupName1 = "M120_R268_30", groupName2 = "M137_R249_53")
```

```
#> [1] 0.4088499
```

```
# similarity between MS/MS spectra with default parameters
spectrumSimilarity(mslists, groupName1 = "M120_R268_30", groupName2 = "M192_R355_191",
                   MSLevel = 2)
```

```
#> [1] 0.08589848
```

```
# As above, with jaccard calculation
spectrumSimilarity(mslists, groupName1 = "M120_R268_30", groupName2 = "M192_R355_191",
                   MSLevel = 2, specSimParams = getDefSpecSimParams(method = "jaccard"))
```

```
#> [1] 0.1111111
```

```
# With shifting
spectrumSimilarity(mslists, groupName1 = "M120_R268_30", groupName2 = "M192_R355_191",
                   MSLevel = 2, specSimParams = getDefSpecSimParams(shift = "both"))
```

```
#> [1] 0.08589848
```

The `spectrumSimilarity` function can also be used to calculate *multiple* similarities. Simply specify multiple feature group names for the `groupNameX` parameters. Alternatively, if you want to compare the same set of feature groups with each other pass their names only as the `groupName1` parameter:

```
# compare two pairs
spectrumSimilarity(mslists,
                   groupName1 = c("M120_R268_30", "M137_R249_53"),
                   groupName2 = c("M146_R309_68", "M192_R355_191"),
                   MSLevel = 2, specSimParams = getDefSpecSimParams(shift = "both"))
```

```
#>              M146_R309_68 M192_R355_191
#> M120_R268_30     0.520052    0.08589848
#> M137_R249_53     0.197720    0.03372542
```

```
# compare all
spectrumSimilarity(mslists, groupName1 = groupNames(mslists),
                   MSLevel = 2, specSimParams = getDefSpecSimParams(shift = "both"))
```

```
#>               M120_R268_30 M137_R249_53 M146_R309_68 M192_R355_191
#> M120_R268_30    1.00000000   0.20406381   0.52005204    0.08589848
#> M137_R249_53    0.20406381   1.00000000   0.19772004    0.03372542
#> M146_R309_68    0.52005204   0.19772004   1.00000000    0.08524785
#> M192_R355_191   0.08589848   0.03372542   0.08524785    1.00000000
```

5.7 Visualization

5.7.1 Features and annatation data

Several generic functions are available to visualize feature and annotation data:

Generic	Classes	Remarks
`plot()`	`featureGroups`, `featureGroupsComparison`	Scatter plot for retention and *m/z* values
`plotInt()`	`featureGroups`	Intensity profiles across analyses
`plotChroms()`	`featureGroups`, `components`	Plot extracted ion chromatograms (EICs)
`plotSpectrum()`	`MSPeakLists`, `formulas`, `compounds`, `components`	Plots (annotated) spectra
`plotStructure()`	`compounds`	Draws candidate structures
`plotScores()`	`formulas`, `compounds`	Barplot with candidate scoring
`plotGraph()`	`componentsNT`	Draws interactive graphs of linked homologous series

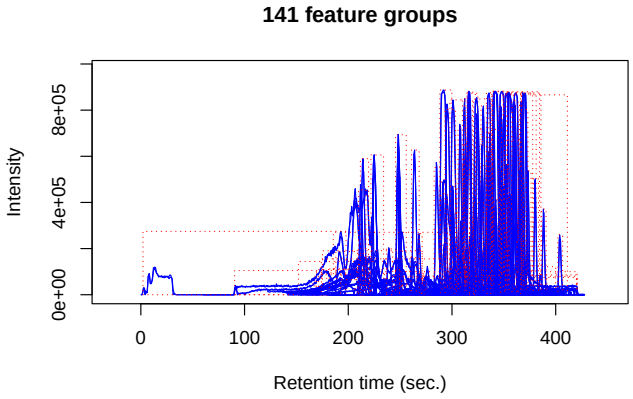
The most common plotting functions are `plotChroms()`, which plots chromatographic data for features, and `plotSpectrum()`, which will plot (annotated) spectra. An overview of their most important function arguments are shown below.

Argument	Generic	Remarks
`retMin`	`plotChroms()`	If `TRUE` plot retention times in minutes
`EICParams`	`plotChroms()`	Advanced parameters to control the creation of extracted ion chromatograms (described below)
`showPeakArea`, `showFGroupRect`	`plotChroms()`	Fill peak areas / draw rectangles around feature groups?
`title`	`plotChroms()`, `plotSpectrum()`	Override plot title
`colourBy`	`plotChroms()`	Colour individual feature groups (`"fGroups"`) or replicate groups (`"rGroups"`). By default nothing is coloured (`"none"`)
`showLegend`	`plotChroms()`	Display a legend? (only if `colourBy!="none"`)
`xlim`, `ylim`	`plotChroms()`, `plotSpectrum()`	Override x/y axis ranges, i.e. to manually set plotting range.
`groupName`, `analysis`, `precursor`, `index`	`plotSpectrum()`	What to plot. See examples below.
`MSLevel`	`plotSpectrum()`	Whether to plot an MS or MS/MS spectrum (only `MSPeakLists`)
`formulas`	`plotSpectrum()`	Whether `formula` annotations should be added (only `compounds`)
`plotStruct`	`plotSpectrum()`	Whether the structure should be added to the plot (only `compounds`)
`mincex`	`plotSpectrum()`	Minimum annotation font size (only `formulas`/`compounds`)

Note that we can use subsetting to select which feature data we want to plot, e.g.

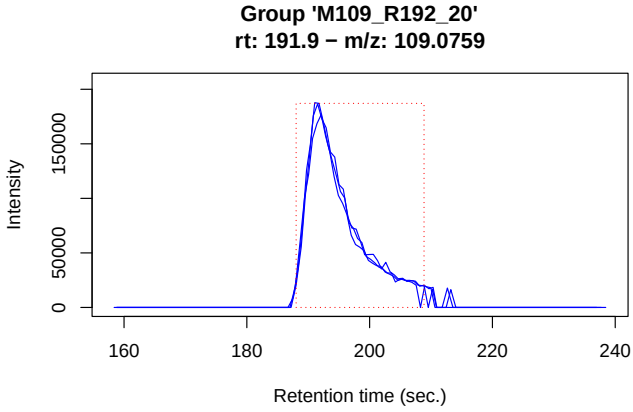
```
plotChroms(fGroups[1:2]) # only plot EICs from first and second analyses.
```

```
#> Verifying if your data is centroided... Done!
```



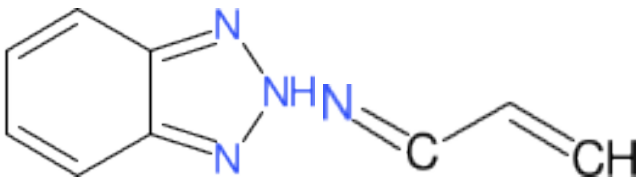

```
plotChroms(fGroups[, 1]) # only plot all features of first group
```

```
#> Verifying if your data is centroided... Done!
```



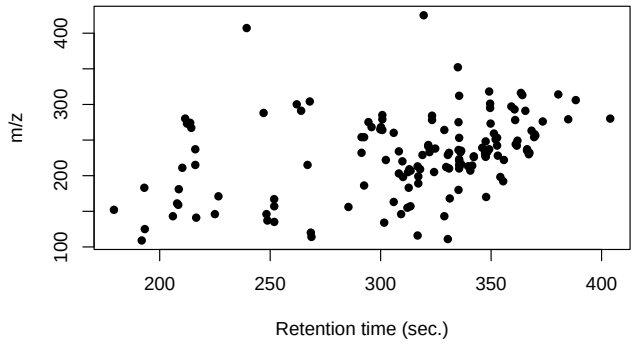
The `plotStructure()` function will draw a chemical structure for a compound candidate. In addition, this function can draw the maximum common substructure (MCS) of multiple candidates in order to assess common structural features.

```
# structure for first candidate
plotStructure(compounds, index = 1, groupName = "M120_R268_30")
# MCS for first three candidates
plotStructure(compounds, index = 1:3, groupName = "M120_R268_30")
```



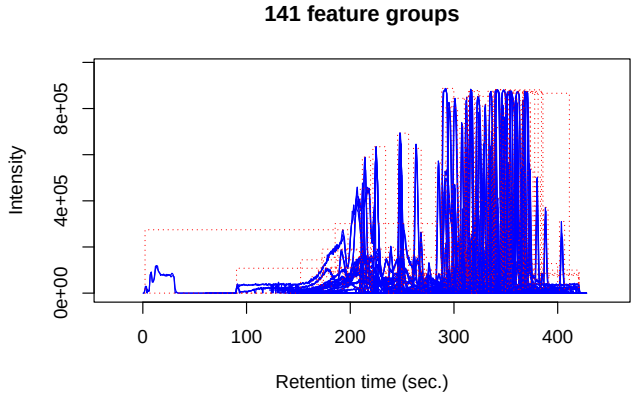
Some other common and less common plotting operations are shown below.

```
plot(fGroups) # simple scatter plot of retention and m/z values
```



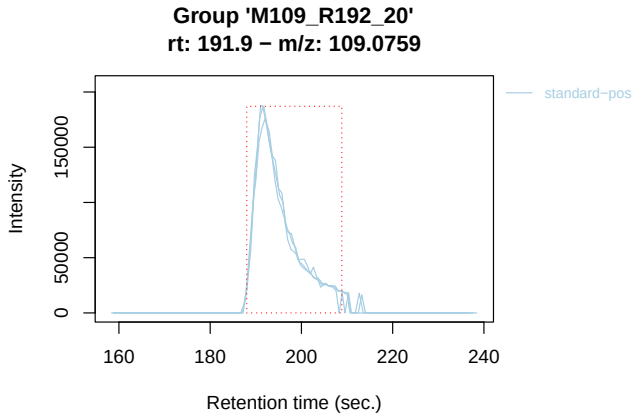

```
plotChroms(fGroups) # plot EICs for all features
```

```
#> Verifying if your data is centroided... Done!
```



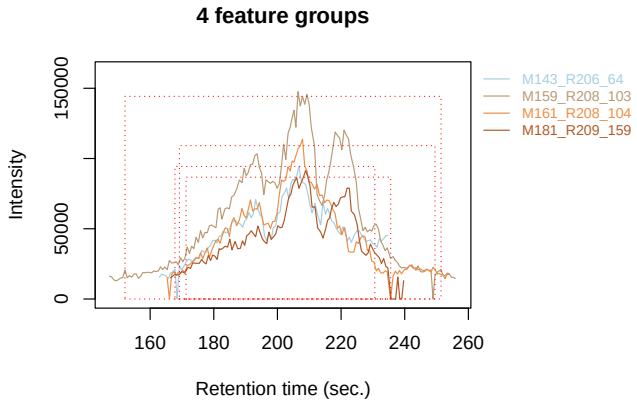

```
plotChroms(fGroups[, 1], # only plot all features of first group
           colourBy = "rGroup") # and mark them individually per replicate group
```

```
#> Verifying if your data is centroided... Done!
```

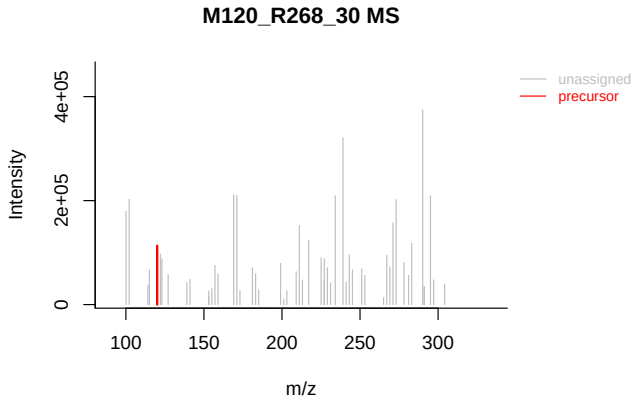


```
plotChroms(components, index = 7, fGroups = fGroups) # EICs from a component
```

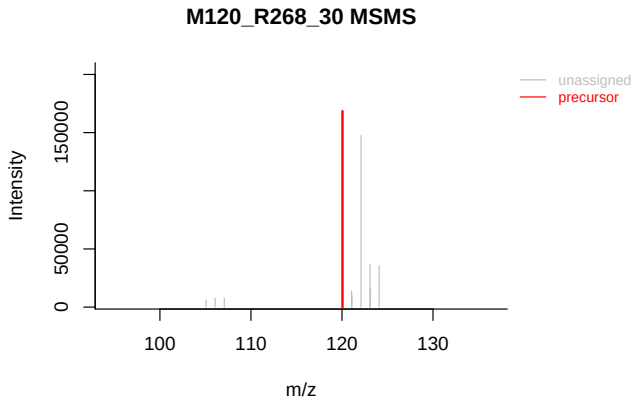
```
#> Verifying if your data is centroided... Done!
```



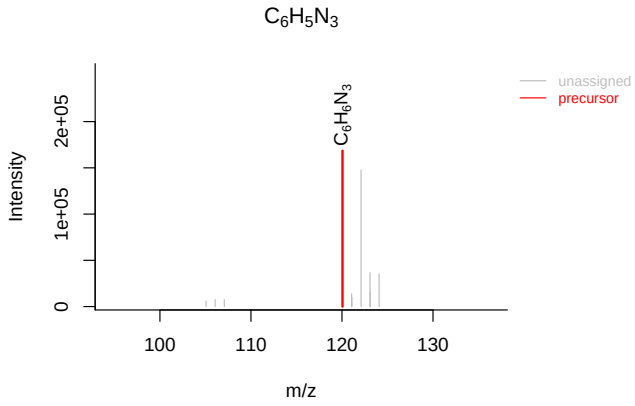

```
plotSpectrum(mslists, "M120_R268_30") # non-annotated MS spectrum
```



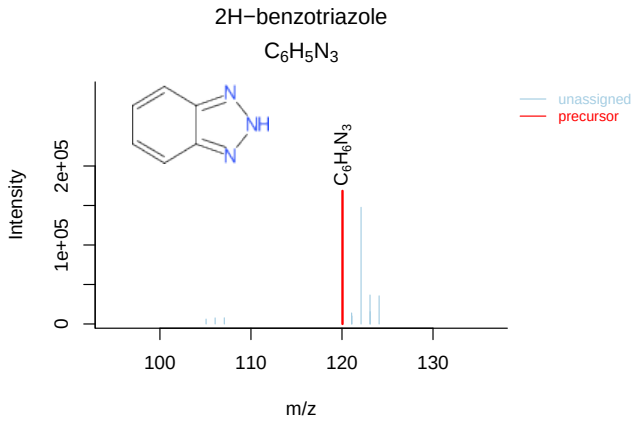

```
plotSpectrum(mslists, "M120_R268_30", MSLevel = 2) # non-annotated MS/MS spectrum
```



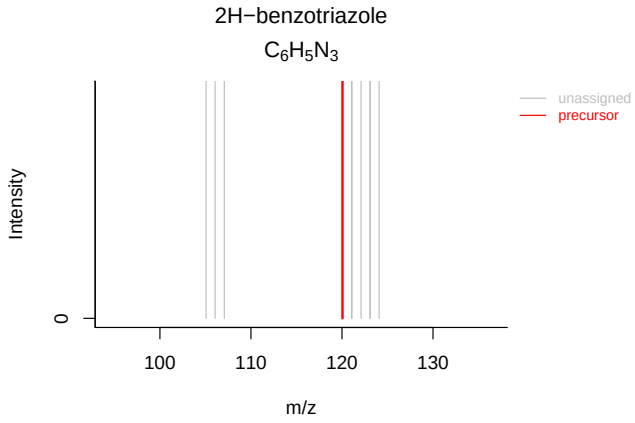

```
# formula annotated spectrum
plotSpectrum(formulas, index = 1, groupName = "M120_R268_30",
             MSPeakLists = mslists)
```



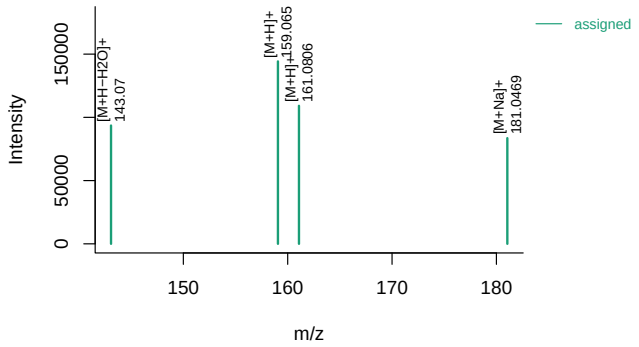

```
# compound annotated spectrum, with added formula annotations
plotSpectrum(compounds, index = 1, groupName = "M120_R268_30", MSPeakLists = mslists,
             formulas = formulas, plotStruct = TRUE)
```



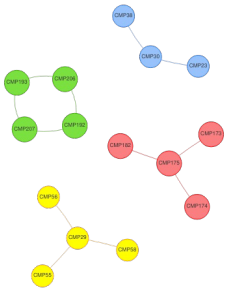

```
# custom intensity range (e.g. to zoom in)
plotSpectrum(compounds, index = 1, groupName = "M120_R268_30", MSPeakLists = mslists,
             ylim = c(0, 5000), plotStruct = FALSE)
```




```
plotSpectrum(components, index = 7) # component spectrum
```




```
# Inspect homologous series
plotGraph(componNT)
```



5.7.1.1 Extracted Ion Chromatogram parameters The `EICParams` argument to `plotChroms()` is used to specify more advanced parameters for the creation of extracted ion chromatograms (EICs). Some parameters of interest:

Parameter	Description
`rtWindow`	Expands the EIC retention time range +/- the feature peak width (in seconds). This is e.g. useful to zoom out.
`topMost`	Only consider this amount of highest intensity features in a group.
`topMostByRGroup`	If `TRUE` then the `topMost` parameter concerns the top most intense features in a replicate group (e.g. `topMost=1` would draw the most intense feature for each replicate group).
`onlyPresent`	Only create EICs for analyses where a feature was detected? Setting to `FALSE` is useful to inspect if a feature was 'missed'.

The parameters are configured by giving a named `list` to the `EICParams` argument. To obtain such a `list` with default settings, the `getDefEICParams()` function can be used:

```
getDefEICParams()
```

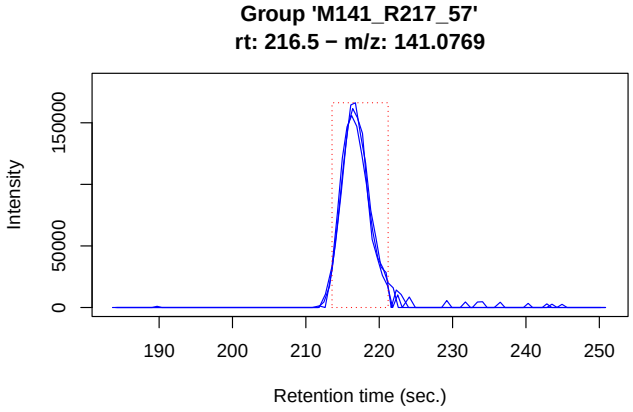
```
#> $rtWindow
#> [1] 30
#>
#> $topMost
#> NULL
#>
```

```
#> $topMostByRGroup
#> [1] FALSE
#>
#> $onlyPresent
#> [1] TRUE
#>
#> $mzExpWindow
#> [1] 0.001
#>
#> $setsAdductPos
#> [1] "[M+H]+"
#>
#> $setsAdductNeg
#> [1] "[M-H]-"
```

Any arguments specified to this function will alter the values of the returned parameter list. Some examples:

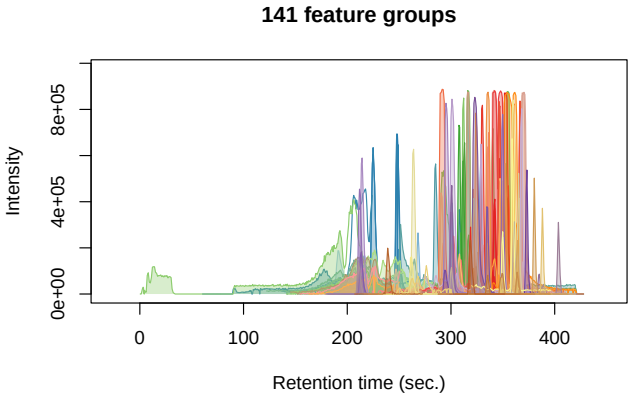
```
# investigate if any features were not detected in a feature group
plotChroms(fGroups[, 10], EICParams = getDefEICParams(onlyPresent = FALSE))
```

```
#> Verifying if your data is centroided... Done!
```




```
# get overview of all feature groups
plotChroms(fGroups,
           colourBy = "fGroup", # unique colour for each group
           EICParams = getDefEICParams(topMost = 1), # only most intense feature in each
           ↪  group
           showPeakArea = TRUE, # show integrated areas
           showFGroupRect = FALSE,
           showLegend = FALSE) # no legend (too busy for many feature groups)
```

```
#> Verifying if your data is centroided... Done!
```



The reference manual (`?EICParams`) gives a full detail on all parameters.

5.7.2 Overlapping and unique data

There are three functions that can be used to visualize overlap and uniqueness between data:

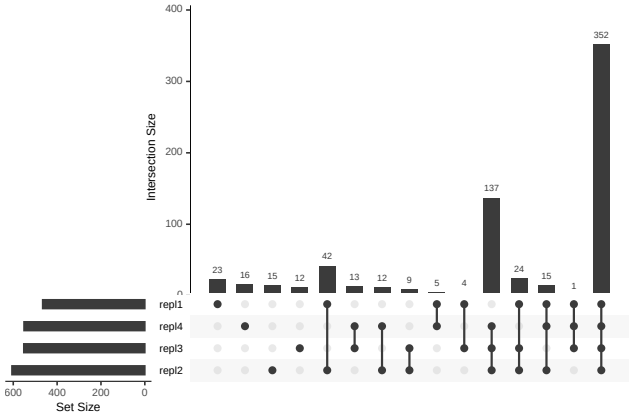
Generic	Classes
`plotVenn`	`featureGroups`, `featureGroupsComparison`, `formulas`, `compounds`
`plotUpSet`	`featureGroups`, `featureGroupsComparison`, `formulas`, `compounds`
`plotChord`	`featureGroups`, `featureGroupsComparison`

The most simple comparison plot is a Venn diagram (i.e. `plotVenn()`). This function is especially useful for two or three-way comparisons. More complex comparisons are better visualized with UpSet diagrams (i.e. `plotUpSet()`). Finally, chord diagrams (i.e. `plotChord()`) provide visually pleasing diagrams to assess overlap between data.

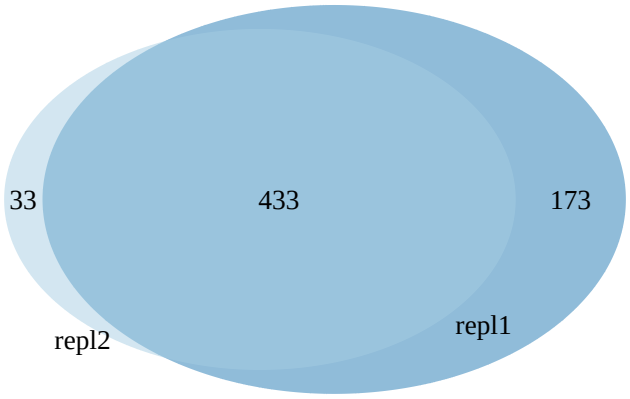
These functions can either be used to compare feature data or different objects of the same type. The former is typically used to compare overlap or uniqueness between features in different replicate groups, whereas comparison between objects is useful to visualize differences in algorithmic output. Besides visualization, note that both operations can also be performed to modify or combine objects (see unique and overlapping features and algorithm consensus).

As usual, some examples are shown below.

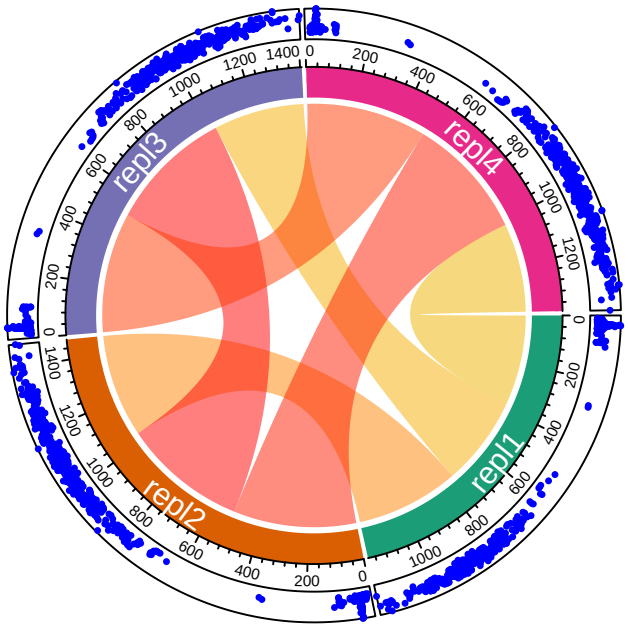
```
plotUpSet(fGroups) # compare replicate groups
```



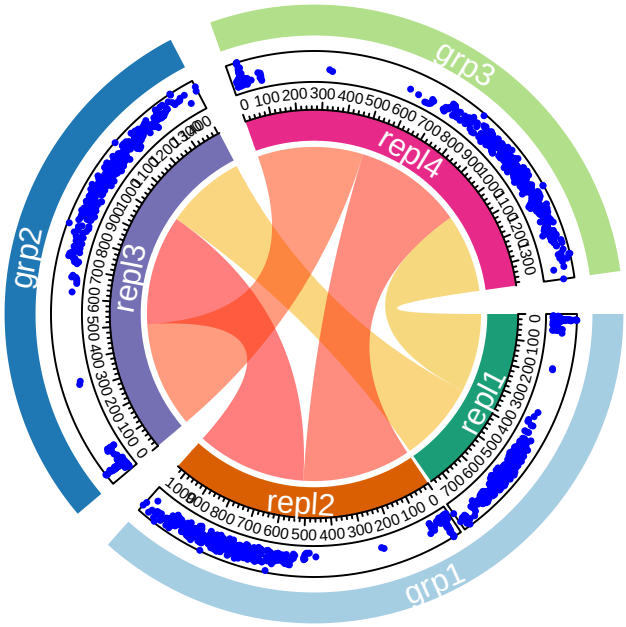
```
plotVenn(fGroups, which = c("repl1", "repl2")) # compare some replicate groups
```



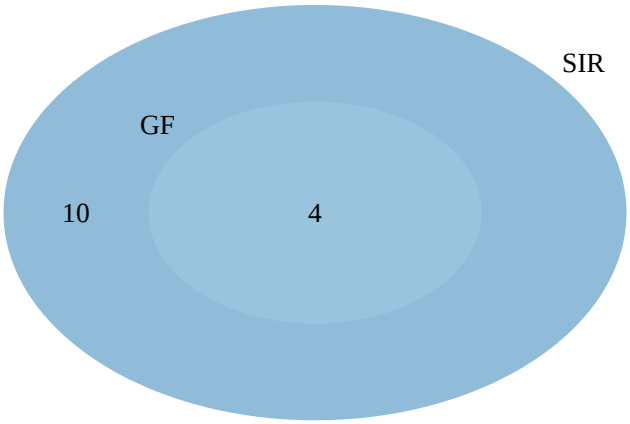

```
plotChord(fGroups, average = TRUE) # overlap between replicate groups
```




```
# compare with custom made groups
plotChord(fGroups, average = TRUE,
          outer = c(repl1 = "grp1", repl2 = "grp1", repl3 = "grp2", repl4 = "grp3"))
```



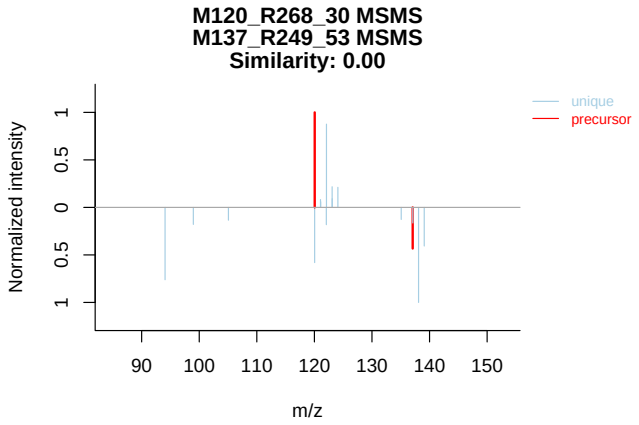

```
# compare GenForm and SIRIUS results
plotVenn(formsGF, formsSIR,
         labels = c("GF", "SIR")) # manual labeling
```



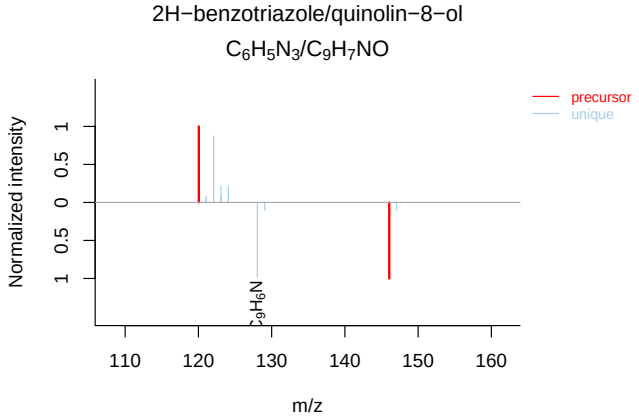
5.7.3 MS similarity

The `plotSpectrum` function is also useful to visually compare (annotated) spectra. This works for `MSPeakLists`, `formulas` and `compounds` object data.

```
plotSpectrum(mslists, groupName = c("M120_R268_30", "M137_R249_53"), MSLevel = 2)
```

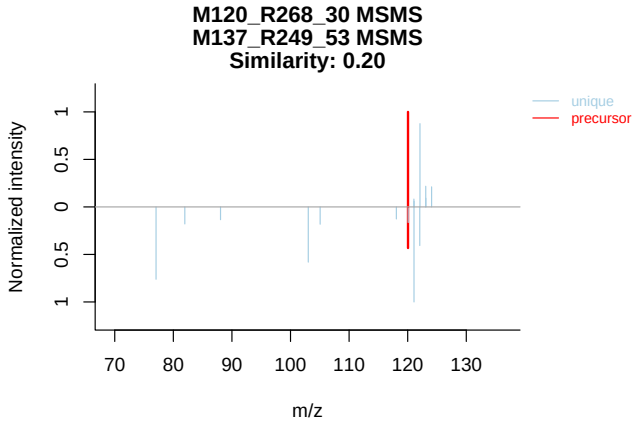



```
plotSpectrum(compounds, groupName = c("M120_R268_30", "M146_R309_68"), index = c(1, 1),
             MSPeakLists = mslists)
```



The `specSimParams` argument, which was discussed in MS similarity, can be used to configure the similarity calculation:

```
plotSpectrum(mslists, groupName = c("M120_R268_30", "M137_R249_53"), MSLevel = 2,
             specSimParams = getDefSpecSimParams(shift = "both"))
```

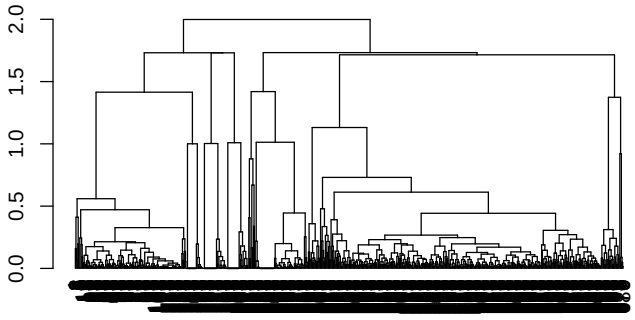


5.7.4 Hierarchical clustering results

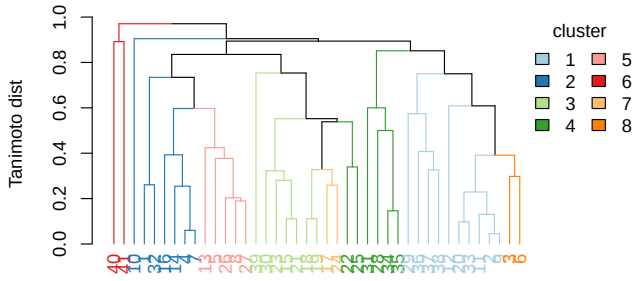
In patRoon hierarchical clustering is used for some componentization algorithms and to cluster candidate compounds with similar chemical structure (see compound clustering). The functions below can be used to visualize their results.

Generic	Classes	Remarks
plot()	All	Plots a dendrogram
plotInt()	componentsIntClust	Plots normalized intensity profiles in a cluster
plotHeatMap()	componentsIntClust	Plots an heatmap
plotSilhouettes()	componentsClust	Plot silhouette information to determine the cluster amount
plotStructure()	compoundsCluster	Plots the maximum common substructure (MCS) of a cluster

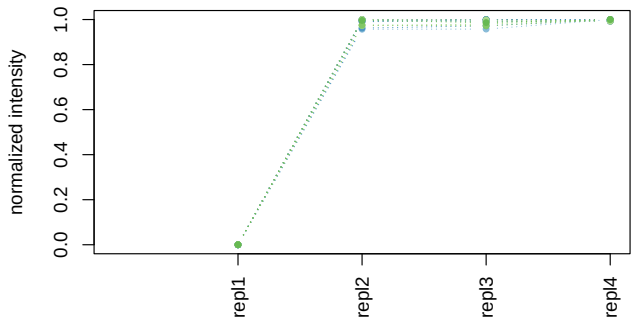
```
plot(componInt) # dendrogram
```



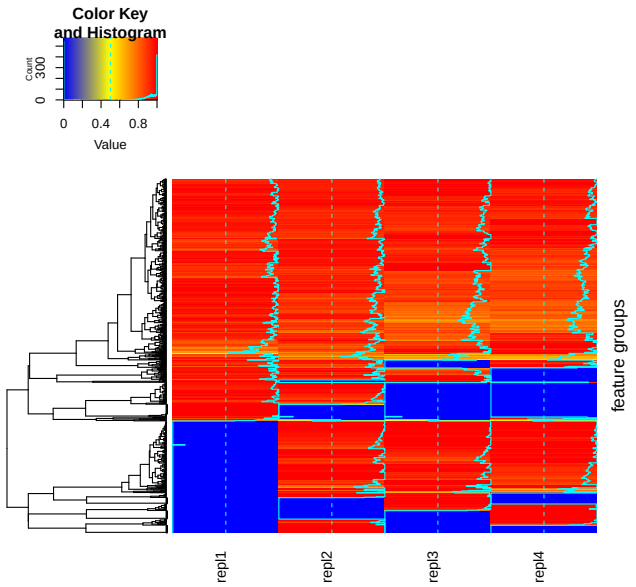

```
plot(compsClust, groupName = "M120_R268_30") # dendrogram for clustered compounds
```



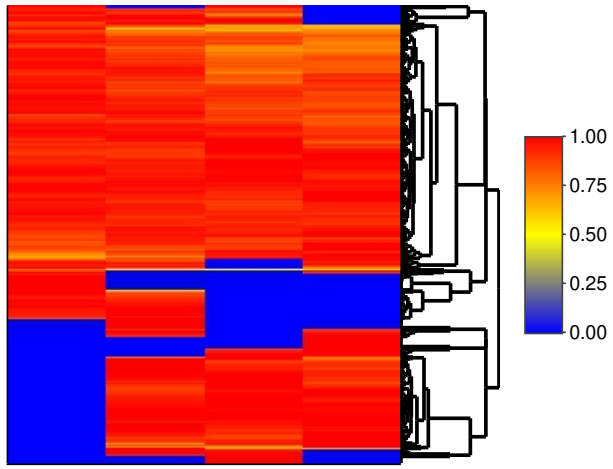

```
plotInt(componInt, index = 4) # intensities of 4th cluster
```



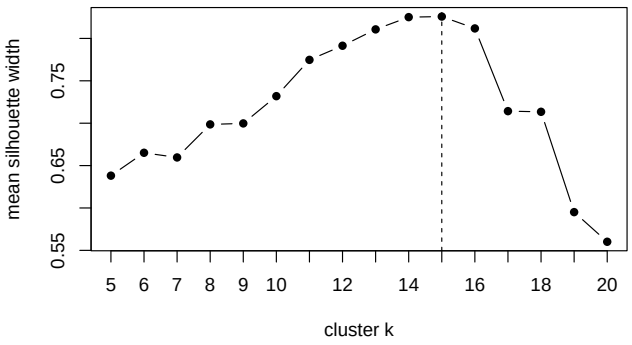
```
plotHeatMap(componInt) # plot heatmap
```




```
plotHeatMap(componInt, interactive = TRUE) # interactive heatmap (with zoom-in!)
```




```
plotSilhouettes(componInt, 5:20) # plot silhouettes (e.g. to obtain ideal cluster amount)
```



5.7.5 Generating EICs in DataAnalysis

If you have Bruker data and the DataAnalysis software installed, you can automatically add EIC data in a DataAnalysis session. The `addDAEIC()` will do this for a single *m/z* in one analysis, whereas the `addAllDAEICs()` function adds EICs for all features in a `featureGroups` object.

```
# add a single EIC with background subtraction
addDAEIC("mysample", "~/path/to/sample", mz = 120.1234, bgsubtr = TRUE)
# add TIC for MS/MS signal of precursor 120.1234 (value of mz is ignored for TICs)
addDAEIC("mysample", "~/path/to/sample", mz = 100, ctype = "TIC",
         mtype = "MSMS", fragpath = "120.1234", name = "MSMS 120")

addAllDAEICs(fGroups) # add EICs for all features
addAllDAEICs(fGroups[, 1:50]) # as usual, subsetting can be used for partial data
```

5.8 Interactively explore and review data

The `checkFeatures` and `checkComponents` functions start a graphical user interface (GUI) which allows you to interactively explore and review feature and components data, respectively.

```
checkFeatures(fGroups) # inspect features and feature groups
checkComponents(componCAM, fGroups) # inspect components
```

Both functions allow you to easily explore the data in an interactive way. Furthermore, these functions allow you to remove unwanted data. This is useful to remove for example features that are actually noise and feature groups that shouldn't be in the same component. To remove an unwanted feature, feature group or components, simply uncheck its 'keep' checkbox. The next step is to save the selections you made. A *check session* is a file that stores which data should be removed. Once the session file is saved the `filter` function can be used to actually remove the data:

```
fGroupsF <- filter(fGroups, checkFeaturesSession = TRUE)
componCAMF <- filter(componCAM, checkComponentsSession = TRUE)
```

If you saved the session and you re-launch the GUI it will restore the selections made earlier. The `clearSession` argument can be used to fully clear a session before starting the GUI, hence, all the data will be restored to their 'keep state'.

```
checkFeatures(fGroups, clearSession = TRUE) # start GUI with fresh session
```

It is also possible to use multiple different sessions. This is especially useful if you do not want to overwrite previous session data or want to inspect different objects. In this case the session file name should be specified:

```
checkFeatures(fGroups, "mysession.yml")
fGroupsF <- filter(fGroups, checkFeaturesSession = "mysession.yml")
```

The default session names are `"checked-features.yml"` and `"checked-components.yml"` for feature and component data, respectively.

The extension of session file names is `.yml` since the YAML file format is used. An advantage of this format is that it is easily readable and editable with a text editor.

Note that the session data is tied to the feature group names of your data. This means that, for instance, when you re-group your feature data after changing some parameters, the session data you prepared earlier cannot be used anymore. Since probably quite some manual work went into creating the session file, a special function is available to import a session that was made for previous data. This function tries its best to guess the new feature group name based on similarity of their retention times and *m/z* values.

```
checkFeatures(fGroups) # do manual inspection

fGroups <- groupFeatures(fList, ...) # re-group with different parameters

importCheckFeaturesSession("checked-features.yml", "checked-features-new.yml", fGroups)

checkFeatures(fGroups, session = "checked-features-new.yml") # inspect new data
```

Take care to monitor the messages that `importCheckFeaturesSession` may output, as it may be possible that some 'old' feature groups are not found or are matched by multiple candidates of the new dataset.

Some additional parameters exist to the functions described in this section. As usualy check the reference manual for more details (*e.g.* `?checkFeatures`).

> ***NOTE*** Although the GUI tools described here allow you to easily filter out results, it is highly recommended to first prioritize your data to avoid doing a lot of unneeded manual work.

5.9 Reporting

The previous sections showed various functionality to inspect and visualize results. An easy way to do this automatically is by using the *reporting* functionality of `patRoon`. There are currently two interfaces: a legacy interface that is described in the next subsection, and the modernized version discussed here.

The reports are generated by the `report()` function, which combines the data generated during the workflow. This function outputs an `HTML` file (other formats may follow in future versions), which can be opened with a regular web browser to interactively explore and visualize the data. The report combines chromatograms, mass spectra, tables with feature and annotation properties and many other useful ways to easily explore your data.

Which data is reported is controlled by the following function arguments:

Argument	Description
`fGroups`	The `featureGroups` object that should be reported (mandatory).
`MSPeakLists`	The MS peak lists object used for annotations (mandatory if `formulas`/`compounds` are specified).
`formulas`, `compounds`	The `formulas` and `compounds` objects that should be used to report feature annotations.
`compsCluster`	The result object from compound clustering.
`components`	Any componentization results, *e.g.* with adduct annotations and from transformation product screening.
`TPs`	Output from object from generated transformation products.

Most of these arguments are optional, and if not specified this part of the workflow is simply not reported. This also means that reporting can be performed at every stage during the workflow, which, for instance, can be useful to quickly inspect results when testing out various settings to generate workflow data. More advanced arguments to `report()` are discussed in the reference manual (`?reporting`).

Some examples:

```
report(fGroups) # only report feature groups
report(fGroups[, 1:50]) # same, but only first 50, e.g. to do a quick inspection

# include feature annotations
report(fGroups, MSPeakLists = mslists, formulas = formulas, compounds = compounds)

# TP screening
report(fGroups, MSPeakLists = mslists, formulas = formulas, compounds = compounds,
           components = componentsTP, TPs = TPs)
```

The report itself is primarily configured through a *report settings file*, which is an easily editable YAML file. The default file is as follows:

```
general:
    version: 2
    format: html
    path: report
    keepUnusedPlots: 7
    selfContained: false
    noDate: false
summary: [ chord, venn, upset ]
features:
    retMin: true
    chromatograms:
        large: true
        small: true
        features: false
        intMax: eic
    intensityPlots: false
    aggregateConcs: mean
    aggregateTox: mean
MSPeakLists:
    spectra: true
formulas:
    include: true
    normalizeScores: max
    exclNormScores: [ ]
    topMost: 25
compounds:
    normalizeScores: max
    exclNormScores: [ score, individualMoNAScore, annoTypeCount, annotHitCount, libMatch
↪   ]
    onlyUsedScorings: true
    topMost: 25
TPs:
    graphs: true
    graphStructuresMax: 25
internalStandards:
    graph: true
```

A detailed description for all the settings can be found in the reference manual (`?reporting`). The table below summarizes the most interesting options:

Parameter	Description
`general --> format`	The output format. Currently only `"html"`.
`general --> selfContained`	If set (`true`) then the output will be a self contained `.html` file. Handy to share reports, but not recommended for large amounts of data.
`features --> chromatograms --> features`	If enabled (`true`) then the report includes chromatograms of individual features. If set to `all` then also chromatograms are generated for analyses in which a feature was not detected. This is especially useful to inspect if features were 'missed' during feature detection or accidentally removed after a filter step.
`formulas/compounds --> topMost`	Specifies the maximum number of top-ranked candidates to plot. Often it will take a considerable amount of time to report all candidates, hence, by default this is limited.

When the newProject tool is used to create a new `patRoon` project a template settings file (`report.yml`) is automatically created. Otherwise, this file can be generated with the `genReportSettingsFile()` function. Simply running this function without any arguments is enough:

```
genReportSettingsFile()
```

5.9.1 Legacy interface

The legacy interface was the default reporting interface for `patRoon` versions older than `2.2`. The interface now mainly serves for backward compatibility reasons, but may still be useful since the new interface does not (yet) support all the formats from the legacy interface. The following three reporting functions are available:

- `reportCSV()`: exports workflow data to comma-separated value (csv) files
- `reportPDF()`: generates simple reports by plotting workflow data in portable document files (PDFs)
- `reportHTML()`: generates interactive and easily explorable reports

Like the `report()` function described above, the arguments to these functions control which data will be reported. However, these functions do not use a report settings file, and all configuration happens through function arguments. Some common arguments are listed below; for a complete listing see the reference manual (`` ?`reporting-legacy` ``).

Argument	Functions	Remarks
`fGroups`, `formulas`, `compounds`, `formulas`, `components`, `compsCluster`, `TPs`	All	Objects to plot. Only `fGroups` is mandatory.
`MSPeakLists`	`reportPDF()`, `reportHTML()`	The `MSPeakLists` object that was used to generate annotation data. Only needs to be specified if `formulas` or `compounds` are reported.
`path`	All	Directory path where report files will be stored (`"report"` by default).
`formulasTopMost`, `compoundsTopMost`	`reportPDF()`, `reportHTML()`	Report no more than this amount of highest ranked candidates.

Argument	Functions	Remarks
`selfContained`	`reportHTML()`	Outputs to a single and self contained `.html` file. Handy to share reports, but not recommended for large amounts of data.

Some typical examples:

```
reportHTML(fGroups) # simple interactive report with feature data
# generate PDFs with feature and compound annotation data
reportPDF(fGroups, compounds = compounds, MSPeakLists = mslists)
reportCSV(fGroups, path = "myReport") # change destination path

# generate report with all workflow types and increase maximum number of
# compound candidates to top 10
reportHTML(fGroups, formulas = formulas, compounds = compounds,
           components = components, MSPeakLists = mslists,
           compsCluster = compsClust,
           compoundsTopMost = 10)
```

6 Sets workflows

In LC-HRMS screening workflows it is typically desired to be able to detect a broad range of chemicals. For this reason, the samples are often measured twice: with positive and negative ionization. Most data processing steps are only suitable for data with the same polarity, for instance, due to the fact that the m/z values in mass spectra are inherently different (e.g. `[M+H]+` vs `[M-H]-`) and MS/MS fragmentation occurs differently. As a result, the screening workflow has to be done twice, which generally requires more time and complicates comparing and interpretation of the complete (positive and negative) dataset.

In `patRoon` version 2.0 the *sets workflow* is introduced. This allows you to perform a single non-target screening workflow from different *sets* of analyses files. Most commonly, each set represents a polarity, hence, there is a positive and negative set. However, more than two sets are supported, and other distinctions between sets are also possible, for instance, samples that were measured with different MS/MS techniques. Another important advantage of the sets workflow is that MS/MS data from different sets can be combined to provide more comprehensive annotations of features. The most important limitation is that (currently) the chromatographic method that was used when analyzing the samples from each set needs to be equal, since retention times are used to group features among the sets.

Performing a sets workflow usually only requires small modifications compared to a 'regular' `patRoon` workflow. This chapter outlines how to perform such workflows and how to use its unique functionality for data processing. It is assumed that the reader is already familiar with performing 'regular' workflows, which were discussed in the previous chapters.

6.1 Initiating a sets workflow

A sets workflow is not much different than a 'regular' (or non-sets) workflow. For instance, consider the following workflow:

```
anaInfo <- patRoonData::exampleAnalysisInfo("positive")
fList <- findFeatures(anaInfo, "openms")
fGroups <- groupFeatures(fList, "openms")
```

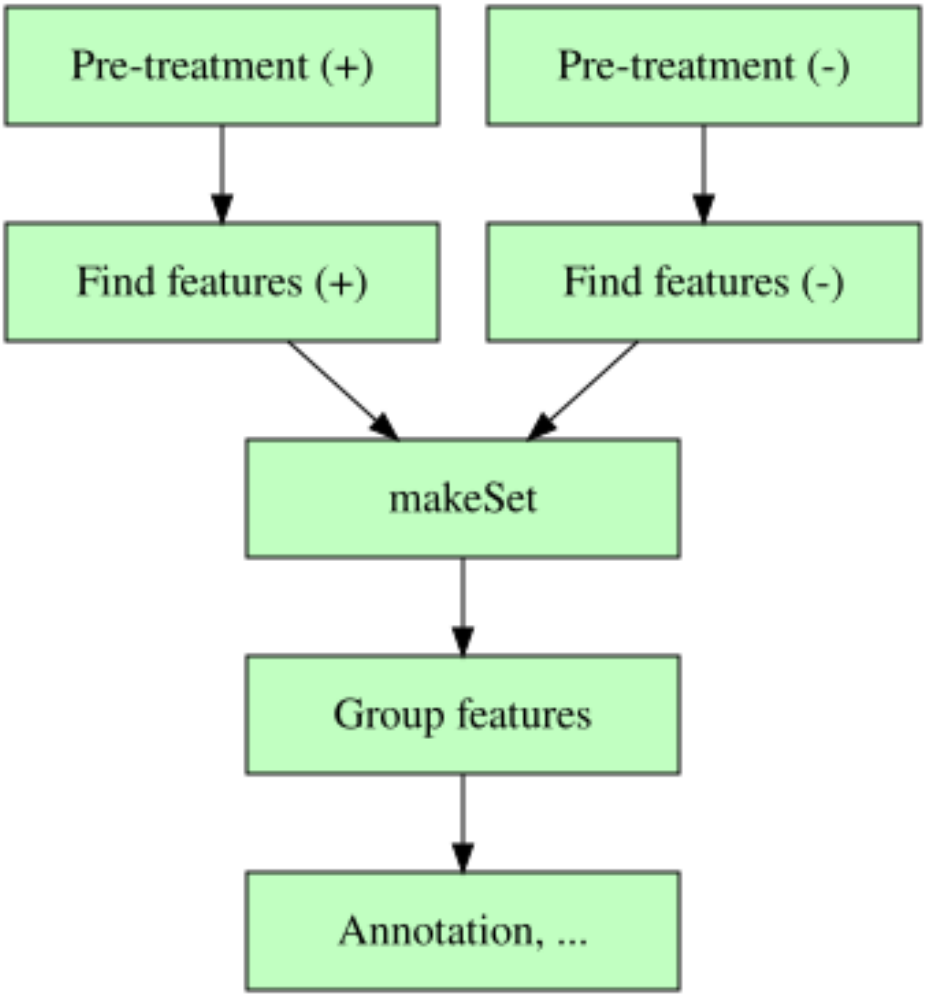
```
fGroups <- filter(fGroups, absMinIntensity = 10000, relMinReplicateAbundance = 1,
↪  maxReplicateIntRSD = 0.75,
                  blankThreshold = 5, removeBlanks = TRUE)

mslists <- generateMSPeakLists(fGroups, "mzr")
formulas <- generateFormulas(fGroups, mslists, "genform", adduct = "[M+H]+")
compounds <- generateCompounds(fGroups, mslists, "metfrag", adduct = "[M+H]+")

report(fGroups, MSPeakLists = mslists, formulas = formulas, compounds = compounds)
```

This example uses the example data from patRoonData to obtain a feature group dataset, which is cleaned-up afterwards. Then, feature groups are annotated and all the results are reported.

Converting this to a *sets workflow*:

```
anaInfoPos <- patRoonData::exampleAnalysisInfo("positive")
anaInfoNeg <- patRoonData::exampleAnalysisInfo("negative")
fListPos <- findFeatures(anaInfoPos, "openms")
fListNeg <- findFeatures(anaInfoNeg, "openms")
fList <- makeSet(fListPos, fListNeg, adducts = c("[M+H]+", "[M-H]-"))

fGroups <- groupFeatures(fList, "openms")
fGroups <- filter(fGroups, absMinIntensity = 10000, relMinReplicateAbundance = 1,
↪  maxReplicateIntRSD = 0.75,
                  blankThreshold = 5, removeBlanks = TRUE)

mslists <- generateMSPeakLists(fGroups, "mzr")
formulas <- generateFormulas(fGroups, mslists, "genform")
compounds <- generateCompounds(fGroups, mslists, "metfrag")

report(fGroups, MSPeakLists = mslists, formulas = formulas, compounds = compounds)
```

This workflow will do all the steps for positive *and* negative data.



Only a few modifications were necessary:

- The analysis information is obtained for positive and negative data (i.e. per set)
- Features are found for each set separately.
- `makeSet` is used to combine the feature data
- There is no need to specify the adduct anymore in the annotation steps.

> ***NOTE*** The `analysis` names for the analysis information must be *unique* for each row, even among sets. Furthermore, replicate groups should not contain analyses from different sets.

The key principle to make sets workflows work is performed by `makeSet`. This method function takes different `features` objects (or `featureGroups`, discussed later) to combine the feature data across sets. During this step features are *neutralized*: the feature *m/z* data is converted to neutral feature masses. This step ensures that when features are grouped with `groupFeatures`, its algorithms are able to find the same feature among different sets, even when different MS ionization modes were used during acquisition. However, please note that (currently) no additional chromatographic alignment steps between sets are performed. For this reason, the chromatographic methodology that is used to acquire the data must be the same for all sets.

The feature neutralization step relies on adduct data. In the example above, it is simply assumed that all features measured with positive mode are protonated (M+H) species, and all negative features are deprotonated (M-H). It is also possible to use adduct annotations for neutralization; this is discussed later.

NOTE The newProject tool can be used to easily generate a sets workflow. Simply select "both" for the *Ionization* option.

6.2 Generating sets workflow data

As was shown in the previous section, the generation of workflow data with a sets workflow largely follows that as what was discussed in the previous chapters. The same generator functions are used:

Workflow step	Function	Output S4 class
Grouping features	`groupFeatures()`	`featureGroupsSet`
Suspect screening	`screenSuspects()`	`featureGroupsScreeningSet`
MS peak lists	`generateMSPeakLists()`	`MSPeakListsSet`
Formula annotation	`generateFormulas()`	`formulasSet`
Compound annotation	`generateCompounds()`	`compoundsSet`
Componentization	`generateComponents()`	algorithm dependent

(the data pre-treatment and feature finding steps have been omitted as they are not specific to sets workflows).

While the same function generics are used to generate data, the class of the output objects differ (e.g. `formulasSet` instead of `formulas`). However, since all these classes *inherit* from their non-sets workflow counterparts, using the workflow data in a sets workflow is nearly identical to what was discussed in the previous chapters (further discussed in the next section).

As discussed before, an important step is the neutralization of features. Other workflow steps also have internal mechanics to deal with data from different sets:

Workflow step	Handling of set data
Finding/Grouping features	Neutralization of m/z values
Suspect screening	Merging results from screening performed for each set
Componentization	Algorithm dependent (discussed below)
MS peak lists	MS data is obtained and stored per set. The final peak lists are combined (*not* averaged)
Formula/Compound annotation	Annotation is performed for each set separately and used to generate a final consensus

In most cases the algorithms of the workflow steps are first performed for each set, and this data is then merged. To illustrate the importance of this, consider these examples

- A suspect screening with a suspect list that contains known MS/MS fragments
- Annotation where MS/MS fragments are used to predict the chemical formula
- Componentization in order to establish adduct assignments for the features

In all cases data is used that is highly dependent on the MS method (eg polarity) that was used to acquire the sample data. Nevertheless, all the steps needed to obtain and combine set data are performed automatically in the background, and are therefore largely invisible.

NOTE Because feature groups in sets workflows always have adduct annotations, it is never required to specify the adduct or ionization mode when generating annotations, components or do suspect screening (*i.e.* the `adduct`/`ionization` arguments should not be specified).

6.2.1 Componentization

When the componentization algorithms related to adduct/isotope annotations (e.g. CAMERA, RAMClustR and cliqueMS) and nontarget are used, then componentization occurs per set and the final object (a `componentsSet` or `componentsNTSet`) contains all the components together. Since these algorithms are highly dependent upon MS data polarity, no attempt is made to merge components from different sets.

The other componentization algorithms work on the complete data. For more details, see the reference manual (`?generateComponents`).

6.2.2 Formula and compound annotation

For formula and compound annotation, the data generated for each set is combined to generate a *set consensus*. The annotation tables are merged, scores are averaged and candidates are re-ranked. More details can be found in the reference manual (e.g. `?generateCompounds`). In addition, it possible to only keep candidates that exist in a minimum number of sets. For this, the `setThreshold` and `setThresholdAnn` argument can be used:

```
# candidate must be present in all sets
formulas <- generateFormulas(fGroups, mslists, "genform", setThreshold = 1)
# candidate must be present in all sets with annotation data
compounds <- generateCompounds(fGroups, mslists, "metfrag", setThresholdAnn = 1)
```

In the first example, a formula candidate for a feature group is only kept if it was found for all of the sets. In the second example, a compound candidate is only kept if it was present in all of the sets with annotation data available. The following examples of a common positive/negative sets workflow illustrate the differences:

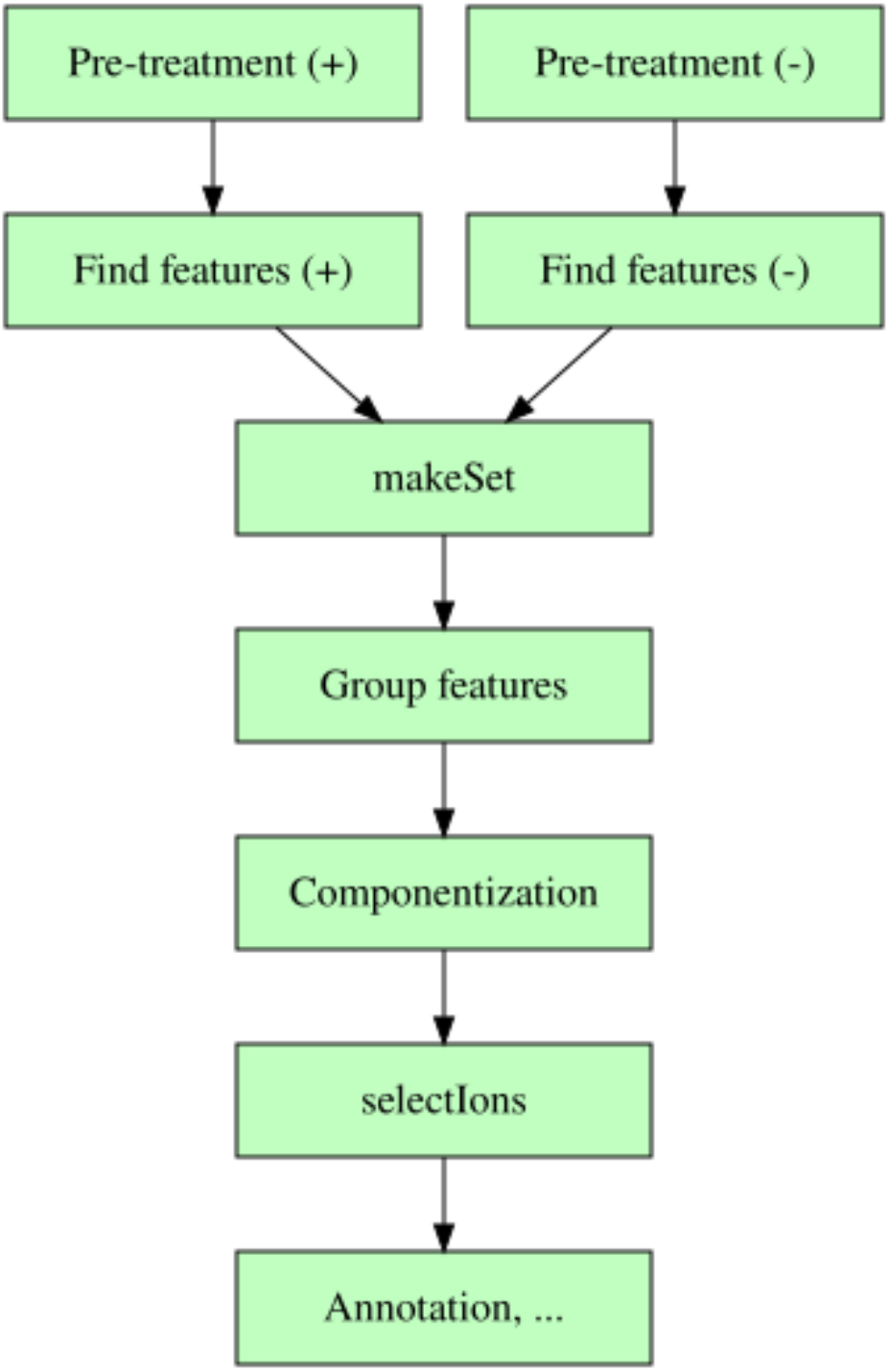
Candidate	annotations	candidate present	`setThreshold=1`	`setThresholdAnn=1`
#1	`+, -`	`+, -`	Keep	Keep
#2	`+, -`	`+`	Remove	Remove
#3	`+`	`+`	Remove	Keep

For more information refer to the reference manual (e.g. `?generateCompounds`).

6.3 Selecting adducts to improve grouping

The `selectIons()` and `adduct()` functions discussed before can also improve sets workflows. This is because the adduct annotations can be used to improve feature neutralization, which in turn will improve grouping features between positive and negative ionization data. Once adduct annotations are set the features will be re-neutralized and re-grouped.

A typical workflow with `selectIons` looks like this:




```
# as before ...
anaInfoPos <- patRoonData::exampleAnalysisInfo("positive")
anaInfoNeg <- patRoonData::exampleAnalysisInfo("negative")
fListPos <- findFeatures(anaInfoPos, "openms")
fListNeg <- findFeatures(anaInfoNeg, "openms")

fGroupsPos <- groupFeatures(fListPos, "openms")
fGroupsNeg <- groupFeatures(fListNeg, "openms")
```

```
fList <- makeSet(fListPos, fListNeg, adducts = c("[M+H]+", "[M-H]-"))

fGroups <- groupFeatures(fList, "openms")
fGroups <- filter(fGroups, absMinIntensity = 10000, relMinReplicateAbundance = 1,
↪  maxReplicateIntRSD = 0.75,
                  blankThreshold = 5, removeBlanks = TRUE)

components <- generateComponents(fGroups, "openms")
fGroups <- selectIons(fGroups, components, c("[M+H]+", "[M-H]-"))

# do rest of the workflow...
```

The first part of the workflow is exactly the same as was introduced in the beginning of this chapter. Furthermore, note that for sets workflows, `selectIons` needs a preferential adduct for each set.

The `adducts` function can also be used to obtain and modify adduct annotations. For sets workflows, these functions operate *per set*:

```
adducts(fGroups, set = "positive")[1:5]
adducts(fGroups, set = "positive")[4] <- "[M+K]+"
```

If you want to modify annotations for multiple sets, it is best to delay the re-gouping step:

```
adducts(fGroups, set = "positive", reGroup = FALSE)[4] <- "[M+K]+"
adducts(fGroups, set = "negative", reGroup = TRUE)[10] <- "[M-H2O]-"
```

Setting `reGroup=FALSE` will not perform any re-neutralization and re-grouping, which preserves feature group names and safes processing time. However, it is **crucial** that the re-grouping step is eventually performed at the end.

6.4 Processing data

All data objects that are generated during a sets workflow *inherit* from the classes from a 'regular' workflow. This means that, with some minor exceptions, *all* of the data processing functionality discussed in the previous chapter (e.g. subsetting, inspection, filtering, plotting, reporting) is also applicable to a sets workflow. For instance, the `as.data.table()` method can be used for general inspection:

```
as.data.table(compounds)[1:5, c("group", "score-positive", "score-negative",
↪  "compoundName", "set")]
```

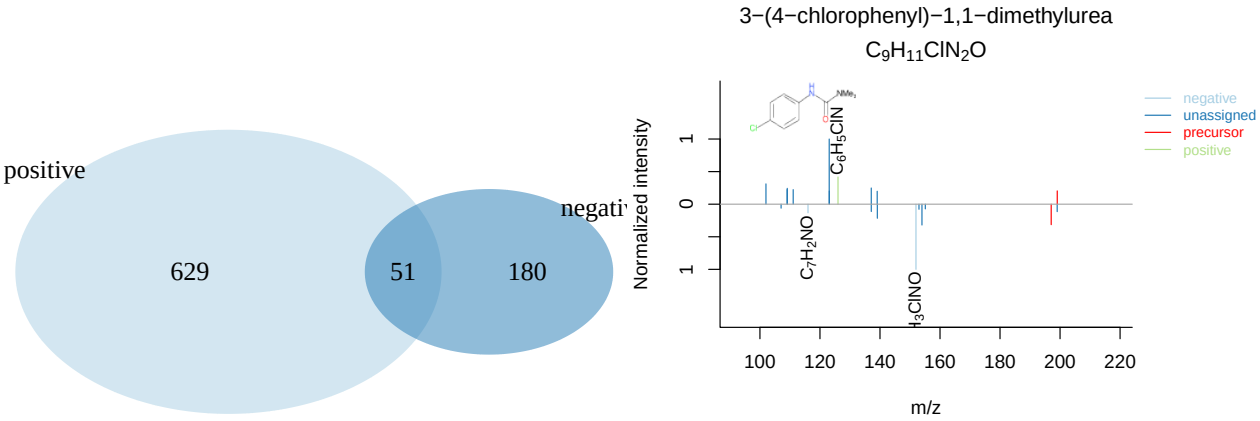
```
#>            group score-positive score-negative                          compoundName             s
#>           <char>          <num>          <num>                                 <char>           <cha
#> 1: M198_R317_273       3.290700       4.569478    3-(4-chlorophenyl)-1,1-dimethylurea positive,negati
#> 2: M198_R317_273       1.668025       2.025473    3-(3-chlorophenyl)-1,1-dimethylurea positive,negati
#> 3: M198_R317_273       1.594943       1.869558 4-amino-2-chloro-N,N-dimethylbenzamide positive,negati
#> 4: M198_R317_273       1.673759       1.557270         1-(4-chlorophenyl)-3-ethylurea positive,negati
#> 5: M198_R317_273       1.591203       1.865819 3-amino-4-chloro-N,N-dimethylbenzamide positive,negati
```

In addition, some the data processing functionality contains additional functionality for a sets workflow:

```
# only keep feature groups that have positive data
fGroupsPos <- fGroups[, sets = "positive"]
# only keep feature groups that have feature data for all sets
fGroupsF <- filter(fGroups, relMinSets = 1)

# only keep feature groups with features present in both polarities
fGroupsPosNeg <- overlap(fGroups, which = c("positive", "negative"), sets = TRUE)
# only keep feature groups with features that are present only in positive mode
fGroupsOnlyPos <- unique(fGroups, which = "positive", sets = TRUE)
```

And plotting:

```
plotVenn(fGroups, sets = TRUE) # compare positive/negative features
plotSpectrum(compounds, index = 1, groupName = "M198_R317_273", MSPeakLists = mslists,
             plotStruct = TRUE)
```



The reference manual for the workflow objects contains specific notes applicable to sets workflows (`?featureGroups`, `?compounds` etc).

6.5 Advanced

6.5.1 Initiating a sets workflow from feature groups

The `makeSet` function can also be used to initiate a sets workflow from feature groups:

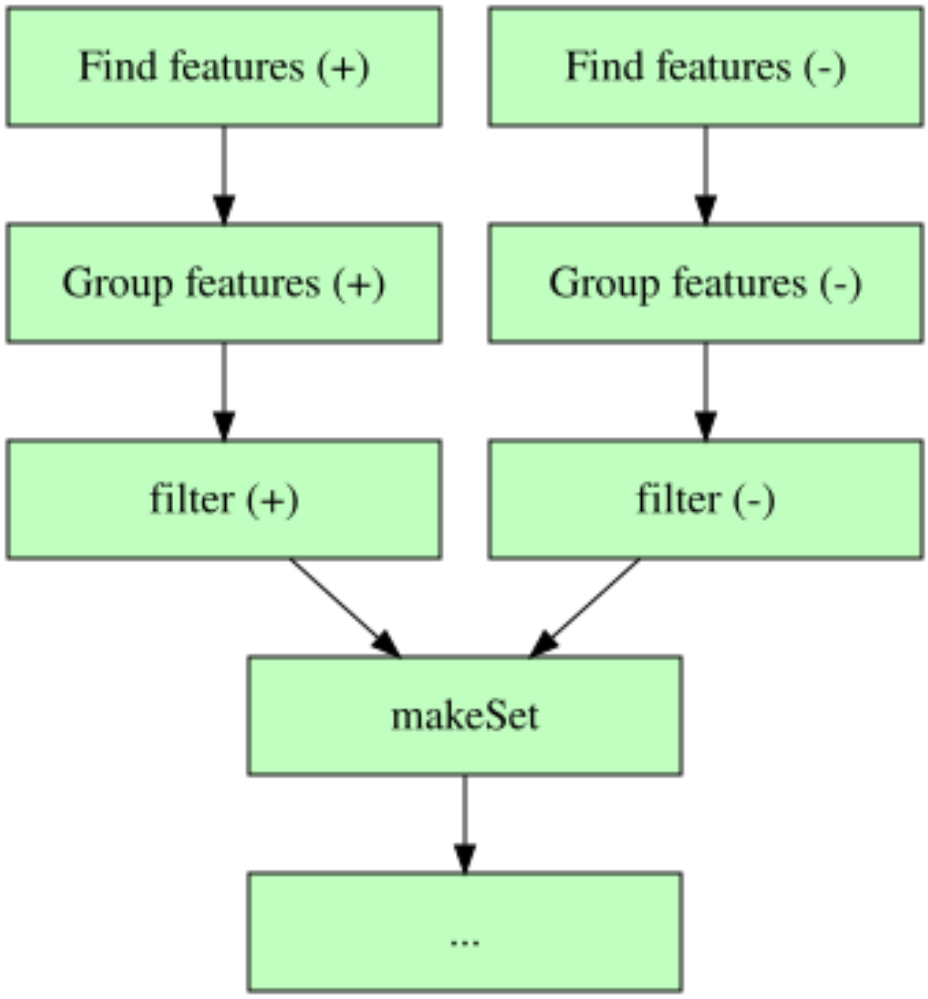
```
# as before ...
anaInfoPos <- patRoonData::exampleAnalysisInfo("positive")
anaInfoNeg <- patRoonData::exampleAnalysisInfo("negative")
fListPos <- findFeatures(anaInfoPos, "openms")
fListNeg <- findFeatures(anaInfoNeg, "openms")

fGroupsPos <- groupFeatures(fListPos, "openms")
fGroupsNeg <- groupFeatures(fListNeg, "openms")

fGroups <- makeSet(fGroupsPos, fGroupsNeg, groupAlgo = "openms",
                   adducts = c("[M+H]+", "[M-H]-"))

# do rest of the workflow...
```

In this case `makeSet` takes the positive and negative features, neutralizes them and creates new feature groups by grouping the original set specific groups (with the algorithm specified by `groupAlgo`).

While this option involves some extra steps, an advantage is that allows processing the feature data before they are combined, e.g.:

```
fGroupsPos <- groupFeatures(fListPos, "openms")
fGroupsNeg <- groupFeatures(fListNeg, "openms")

# apply intensity theshold filters. Lower threshold for negative.
fGroupsPos <- filter(fGroupsPos, absMinIntensity = 1E4)
fGroupsNeg <- filter(fGroupsNeg, absMinIntensity = 1E3)

fGroups <- makeSet(fGroupsPos, fGroupsNeg, groupAlgo = "openms",
                   adducts = c("[M+H]+", "[M-H]-"))
```

Visually, this workflow looks like this:



Of course, any other processing steps on the feature groups data such as subsetting and visually checking features are also possible before the sets workflow is initiated. Furthermore, it is also possible to perform adduct annotations prior to grouping, which is an alternative way to improve neutralization to what was discussed before.

6.5.2 Inspecting and converting set objects

The following generic functions may be used to inspect or convert data from sets workflows:

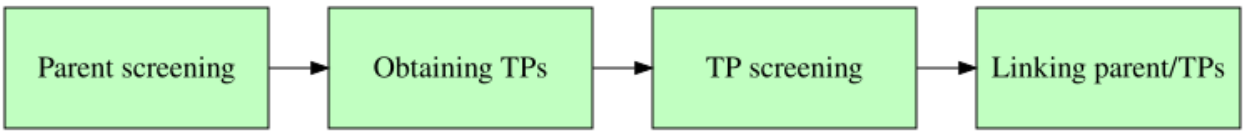
Generic	Purpose	Notes
`sets`	Return the names of the sets in this object.	
`setObjects`	Obtain the raw data objects that were used to construct this object.	Not available for features and feature groups.
`unset`	Converts this object to a regular workflow object.	The `set` argument must be given to specify which of the set data is to be converted. This function will restore the original *m/z* values of features.

These methods are heavily used internally, but rarely needed otherwise. More details can be found in the reference manual.

7 Transformation product screening

This chapter describes the various functionality for screening of *transformation products* (TPs), which are introduced since `patRoon` 2.0. Screening for TPs, i.e. chemicals that are formed from a *parent* chemical by e.g. chemical or biological processes, has broad applications. For this reason, the TP screening related functionality is designed to be flexible, thus allowing one to use a workflow that is best suited for a particular study.

Regardless, the TP screening workflow in `patRoon` can be roughly summarized as follows:



- **Parent screening** During this step a common `patRoon` workflow is used to screen for the parent chemicals of interest. This could be a full non-target analysis with compound annotation or a relative simple suspect or target screening.
- **Obtaining TPs** Data is obtained of potential TPs for the parents of interest. The TPs may originate from a library or predicted *in-silico*. Note that in some workflows this step is omitted (discussed later).
- **TP screening** A suspect screening is performed to find the TPs in the analysis data.
- **Linking parents and TPs** In the step the parent features are linked with the TP features. Several post-processing functionality exists to improve and prioritize the data.

The next sections will outline more details on these steps are performed and configured. The last section in this chapter outlines several example workflows.

> ***NOTE*** The newProject tool can be used to easily generate a workflow with transformation product screening.

7.1 Obtaining transformation product data

The `generateTPs` function is used to obtain TPs for a particular set of parents. Like other workflow generator functions (`findFeatures`, `generateCompounds`), several algorithms are available that do the actual work.

Algorithm	Usage	Remarks
BioTransformer	`generateTPs(algorithm = "biotransformer", ...)`	Predicts TPs with full structural information
CTS	`generateTPs(algorithm = "cts", ...)`	Predicts TPs with full structural information
Library	`generateTPs(algorithm = "library", ...)`	Obtains transformation products from a library (PubChem transformations or custom)
Formula library	`generateTPs(algorithm = "library_formula", ...)`	Obtains transformation products from a library (only formula data)
Metabolic logic	`generateTPs(algorithm = "logic", ...)`	Uses pre-defined logic to predict TPs based on common elemental differences (e.g. hydroxylation, demethylation). Based on Schollee et al. (2015).

The output of these algorithms can be distinguished in three categories:

1. **Structural TPs** (`biotransformer`, `cts` and `library`) come with full structural information for the TPs (e.g. formula, SMILES, predicted Log P). As such, the corresponding algorithms also require the full chemical structure of the parent compound.
2. **Formula TPs** (`library_formula`) are similar to structural TPs, but only involve formula and no other structural information.
3. **Calculated TPs** (`logic`) are based solely on m/z differences and only require the feature masses.

Algorithms that fall into the first category are typically used when parents are known in advance, for instance, from a target or suspect screening. This is also true for the second category, however, here only formula data is used, which is useful when the complete structure of parents and/or TPs are not known. Calculated TPs allow TP prediction for all features, even when nothing is known about their structure. This is most suitable for full non-target analysis, however, extra care must be taken to rule out false positives. Finally, the logic used to calculate TPs can also be used to automatically to generate a library suitable for the `library_formula` algorithm, which allows a hybrid approach of the second and third categories.

An overview of common arguments for TP generation is listed below.

Argument	Algorithm(s)	Remarks
`parents`	`biotransformer`, `cts`, `library`	The input parents. See section below.
`fGroups`	`logic`	The input feature groups to calculate TPs for.
`type`	`biotransformer`	The prediction type: `"env"`, `"ecbased"`, `"cyp450"`, `"phaseII"`, `"hgut"`, `"superbio"`, `"allHuman"`. See BioTransformer for more details.

Argument	Algorithm(s)	Remarks
`transLibrary`	`cts`	The transformation library that should be used: `"hydrolysis"`, `"abiotic_reduction"`, `"photolysis_unranked"`, `"photolysis_ranked"`, `"mammalian_metabolism"`, `"combined_abioticreduction_hydrolysis"`, `"combined_photolysis_abiotic_hydrolysis"`. See cts for more details.
`TPLibrary/transformations`	`library/logic`	Custom TP library/transformation rules.
`generation`	`biotransformer`, `cts`, `library`	The amount of transformation generations to predict.
`adduct`	`logic`	The assumed adduct of the parents (e.g. `"[M+H]+"`). Not needed when adduct annotations are available.
`calcSims`	`biotransformer`, `cts`, `library`	If `TRUE` then structural similarities between the parent and TPs is calculated, which can be useful for post-processing (discussed later).

7.1.1 Parent input

The input parent structures to generate structural/formula TPs (`biotransformer`, `cts`, `library` and `library_formula` algorithms) must be specified as one of the following:

- A suspect list (follows the same format as suspect screening)
- A feature groups object with screening results (e.g. obtained with `screenSuspects`, see suspect screening)
- A `compounds` object obtained with compound annotation (not supported for `library_formula`)

In the former two cases the parent information is taken from the suspect list or from the hits in a suspect screening worklow, respectively. The last case is more suitable for when the parents are not completely known. In this case, the candidate structures from a compound annotation are used as input to obtain TPs. Since *all* the candidates are used, it is highly recommend to filter the object in advance, for instance, with the `topMost` filter. For `library` and `library_formula`, the parent input is optional: if no parents are specified then TP data for *all* parents in the database is used.

For the `logic` algorithm TPs are predicted directly for feature groups. Since this algorithm can only perform very basic validity checks, it is strongly recommended to first prioritize the feature group data.

Some typical examples:

```
# predict environmental TPs with BioTransformer for all parents in a suspect list
TPsBT <- generateTPs("biotransformer", parents = patRoonData::suspectsPos,
                     type = "env")
# obtain all TPs from the default library
TPsLib <- generateTPs("library")
# get TPs for the parents from a suspect screening
TPsLib <- generateTPs("library", parents = fGroupsScr)
# calculate TPs for all feature groups
TPsLogic <- generateTPs("logic", fGroups, adduct = "[M+H]+")
```

7.1.2 Processing data

Similar to other workflow data, several generic functions are available to inspect the data:

Generic	Remarks
`length()`	Returns the total number of transformation products
`names()`	Returns the names of the parents
`parents()`	Returns a table with information about the parents
`products()`	Returns a `list` with for each parent a table with TPs
`as.data.table()`, `as.data.frame`	Convert all the object information into a `data.table`/`data.frame`
`"[["` / `"$"` operators	Extract TP information for a specified parent

Some examples:

```
# just show a few columns in this example, there are many more!
# note: the double dot syntax (..cols) is necessary since the data is stored as
↪  data.tables
cols <- c("name", "formula", "InChIKey")
parents(TPs)[1:5, ..cols]
```

```
#>                name    formula                     InChIKey
#>              <char>     <char>                       <char>
#> 1:             DEET   C12H17NO MMOXZBCLCQITDF-UHFFFAOYSA-N
#> 2:          Irgarol  C11H19N5S HDHLIWCXDDZUFH-UHFFFAOYSA-N
#> 3:       Prometryne  C10H19N5S AAEVYOVXGOFMJO-UHFFFAOYSA-N
#> 4:     Trimethoprim C14H18N4O3 IEDVJHCEMCRBQM-UHFFFAOYSA-N
#> 5: 1H-benzotriazole     C6H5N3 QRUDEWIWKLJBPS-UHFFFAOYSA-N
```

```
TPs[["DEET"]][, ..cols]
```

```
#>        name   formula                     InChIKey
#>      <char>    <char>                       <char>
#> 1: DEET-TP1 C12H17NO2 FRZJZRVZZNTMAW-UHFFFAOYSA-N
#> 2: DEET-TP2 C12H17NO2 KVTUZBGZTRABBQ-UHFFFAOYSA-N
#> 3: DEET-TP3     C2H4O IKHGUXGNUITLKF-UHFFFAOYSA-N
#> 4: DEET-TP4  C10H13NO FPINATACRXASTP-UHFFFAOYSA-N
#> 5: DEET-TP4  C10H13NO FPINATACRXASTP-UHFFFAOYSA-N
#> 6: DEET-TP5    C4H11N HPNMFZURTQLUMO-UHFFFAOYSA-N
#> 7: DEET-TP6    C8H7O2 GPSDUZXPYCFOSQ-UHFFFAOYSA-M
#> 8: DEET-TP7    C8H9NO WGRPQCFFBRDZFV-UHFFFAOYSA-N
#> 9: DEET-TP1 C12H17NO2 FRZJZRVZZNTMAW-UHFFFAOYSA-N
```

```
TPs[[2]][, ..cols]
```

```
#>           name    formula                     InChIKey
#>         <char>     <char>                       <char>
#> 1: Irgarol-TP1   C8H15N5S MWWBDLRPMWTLRX-UHFFFAOYSA-N
#> 2: Irgarol-TP2 C11H19N5OS HFCMSBLJLJOGGL-UHFFFAOYSA-N
```

```
as.data.table(TPs)[1:5, 1:3]
```

```
#>    parent                                                                    transformation
#>    <char>                                                                            <char>
#> 1:   DEET Aliphatic hydroxylation of methyl carbon adjacent to aromatic ring / Human Phase I
#> 2:   DEET                                    Hydroxylation of terminal methyl / Human Phase I N-Ethyl-
#> 3:   DEET                              N-dealkylation of tertiary carboxamide / Human Phase I
#> 4:   DEET                              N-dealkylation of tertiary carboxamide / Human Phase I
#> 5:   DEET                                                                        Metabolism
```

In addition, the following generic functions are available to modify or convert the object data:

Generic	Classes	Remarks
"[" operator	All	Subset this object on given parents
`filter`	All	Filters this object
`convertToSuspects`	All	Generates a suspect list of all TPs (and optionally parents) that is suitable for `screenSuspects`

```
TPs2 <- TPs[1:10] # only keep results for first ten parents

# only keep TPs with likely/probably likelihood (specific property for CTS algorithm)
TPsF <- filter(TPs, properties = list(likelihood = c("LIKELY", "PROBABLE")))

# do a suspect screening for all TPs and their parents
suspects <- convertToSuspects(TPs, includeParents = TRUE)
fGroupsScr <- screenSuspects(fGroups, suspects, onlyHits = TRUE)
```

The `convertToSuspects` function is always part of a workflow, and is discussed further in the next section.

7.1.2.1 Structural TPs specifics

For structural TPs several additional generic functions are available:

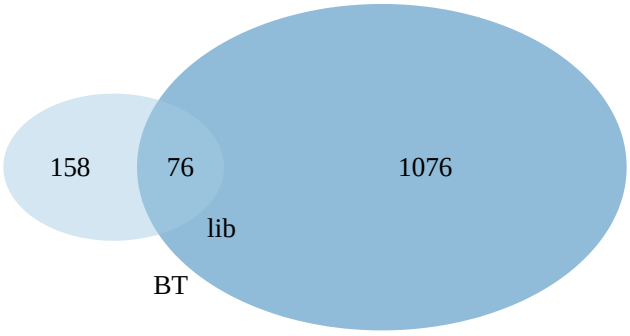
Generic	Remarks
`filter`	Filters this object (additional functionality for structural TPs)
`convertToMFDB`	Generates a MetFrag database for all TPs (and optionally parents)
`plotGraph`	Generates an interactive plot to explore transformation hierarchies
`plotVenn`, `plotUpSet`	Compare results between different algorithms with Venn/UpSet diagrams

The `convertToMFDB` function is especially handy with predicted TPs, as it allows generating a compound database for TPs that may not be available in commonly used databases. This is further demonstrated in the first example.

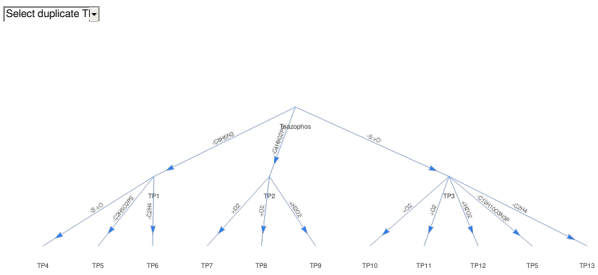
```
# remove transformation products that are isomers to their parent or sibling TPs
# may simplify data as these are often difficult to identify
TPsF <- filter(TPs, removeParentIsomers = TRUE, removeTPIsomers = TRUE)

# remove duplicate transformation products from each parent
# these can occur if different pathways yield the same TPs
TPsF <- filter(TPs, removeDuplicates = TRUE)

# only keep TPs that have a structural similarity to their parent of >= 0.5
# (needs calcSims=TRUE when executing generateTPs())
```

```
TPsF <- filter(TPs, minSimilarity = 0.5)

# use the TP data for a specialized MetFrag database
convertToMFDB(TPs, "TP-database.csv", includeParents = FALSE)
compoundsTPs <- generateCompounds(fGroups, mslists, "metfrag", database = "csv",
                                  extraOpts = list(LocalDatabasePath =
                                  ↪  "TP-database.csv"))
```

```
plotVenn(TPsLib, TPsBT, labels = c("lib", "BT"))
```




```
plotGraph(TPsBT, which = "Triazophos") # hierarchy for Triazophos parent
```



Finally, results from different algorithms can be combined with the `consensus` generic function. This is further discussed in algorithm consensus.

7.1.3 (Custom) Libraries and transformations

By default the `library` and `logic` algorithms use data that is installed with `patRoon` (based on PubChem transformations and Schollee et al. (2015), respectively). However, it is also possible to use custom data. For the `library_formula` no default library is provided, however, these can easily be generated as is discussed at the end of the section.

To use a custom TP structure library a simple `data.frame` is needed with the names, SMILES and optionally `log P` values for the parents and TPs. The `log P` values are used for prediction of the retention time direction of a TP compared to its parent, as is discussed further in the next section. The following small library has two TPs for benzotriazole and one for DEET:

```
myTPLib <- data.frame(parent_name = c("1H-Benzotriazole", "1H-Benzotriazole", "DEET"),
                      parent_SMILES = c("C1=CC2=NNN=C2C=C1", "C1=CC2=NNN=C2C=C1",
                      ↪  "CCN(CC)C(=O)C1=CC=CC(=C1)C"),
```

```
                      TP_name = c("1-Methylbenzotriazole", "1-Hydroxybenzotriazole",
                      ↪  "N-ethyl-m-toluamide"),
                      TP_SMILES = c("CN1C2=CC=CC=C2N=N1", "C1=CC=C2C(=C1)N=NN2O",
                      ↪  "CCNC(=O)C1=CC=CC(=C1)C"))
myTPLib
```

```
#>        parent_name              parent_SMILES                TP_name              TP_SMILES
#> 1 1H-Benzotriazole          C1=CC2=NNN=C2C=C1  1-Methylbenzotriazole     CN1C2=CC=CC=C2N=N1
#> 2 1H-Benzotriazole          C1=CC2=NNN=C2C=C1 1-Hydroxybenzotriazole   C1=CC=C2C(=C1)N=NN2O
#> 3             DEET CCN(CC)C(=O)C1=CC=CC(=C1)C    N-ethyl-m-toluamide CCNC(=O)C1=CC=CC(=C1)C
```

To use this library, simply pass it to the `TPLibrary` argument:

```
TPs <- generateTPs("library", TPLibrary = myTPLib)
```

For `library_formula` the library follows the same format. However, here the formula should be specified instead of the `SMILES` with the `parent_formula` and `TP_formula` columns (although it is still allowed to only specify SMILES, as in this case the formulae are automatically calculated).

For the `logic` algorithm a table with custom transformation rules can be specified for TP calculations:

```
myTrans <- data.frame(transformation = c("hydroxylation", "demethylation"),
                      add = c("O", ""),
                      sub = c("", "CH2"),
                      retDir = c(-1, -1))
myTrans
```

```
#>   transformation add sub retDir
#> 1  hydroxylation   O         -1
#> 2  demethylation     CH2     -1
```

The `add` and `sub` columns are used to denote the elements that are added or subtracted by the reaction. These are used to calculate mass differences between parents and TPs. The `retDir` column is used to indicate the retention time direction of the parent compared to the TP: `-1` (elutes before parent), `1` (elutes after parent) or `0` (similar or unknown). The next section describes how this data can be used to filter TPs. The custom rules can be used by passing them to the `transformations` argument:

```
TPs <- generateTPs("logic", fGroups, adduct = "[M+H]+", transformations = myTrans)
```

The `genFormulaTPLibrary()` utility function can be used to automatically generate TP libraries suitable for the `library_formula` algorithm. The transformation rules to calculate TPs are specified in the same format as used by the `logic` algorithm.

```
myTPFormLib <- genFormulaTPLibrary(parents = patRoonData::suspectsPos, transformations =
↪  myTrans)
# also calculate second generation TPs (TPs of TPs)
myTPFormLib2 <- genFormulaTPLibrary(parents = patRoonData::suspectsPos, transformations =
↪  myTrans,
                                    generations = 2)

# Use library
TPs <- generateTPs("library_formula", TPLibrary = myTPFormLib)
```

Compared to the `logic` algorithm, the `library_formula` algorithm is more (and only) suitable for suspect/target screening workflows, allows multiple transformation generations and allows better customization through manually adding/removing TPs from the library prior to passing it to `generateTPs()`.

7.2 Linking parent and transformation product features

This section discusses one of the most important steps in a TP screening workflow, which is to link feature groups of parents with those of candidate transformation products. During this step, *components* are made, where each component consist of one or more feature groups of detected TPs for a particular parent. Note that componentization was already introduced before, but for very different algorithms. However, the data format for TP componentization is highly similar. After componentization, several filters are available to clean and prioritize the data. These can even allow workflows without obtaining potential TPs in advance, which is discussed in the last subsection.

7.2.1 Componentization

Like other algorithms, the `generateComponents` generic function is used to generate TP components, by setting the `algorithm` parameter to `"tp"`.

The following arguments are of importance:

Argument	Remarks
`fGroups`	The input feature groups for the *parents*
`fGroupsTPs`	The input feature groups for the *TPs*
`ignoreParents`	Set to `TRUE` to ignore feature groups in `fGroupsTPs` that also occur in `fGroups`
`TPs`	The input transformation products, ie as generated by `generateTPs()`
`MSPeakLists`, `formulas`, `compounds`	Annotation objects used for similarity calculation between the parent and its TPs
`minRTDiff`	The minimum retention time difference (seconds) of a TP for it to be considered to elute differently than its parent.

7.2.1.1 Feature group input The `fGroups`, `fGroupsTPs` and `ignoreParents` arguments are used by the componentization algorithm to identify which feature groups can be considered as parents and which as TPs. Three scenarios are possible:

1. `fGroups=fGroupsTPs` and `ignoreParents=FALSE`: in this case no distinction is made, and all feature groups are considered a parent or TP (default if `fGroupsTPs` is not specified).
2. `fGroups` and `fGroupsTPs` contain different subsets of the *same* `featureGroups` object and `ignoreParents=FALSE`: only the feature groups in `fGroups`/`fGroupsTPs` are considered as parents/TPs.
3. As above, but with `ignoreParents=TRUE`: the same distinction is made as above, but any feature groups in `fGroupsTPs` are ignored if also present in `fGroups`.

The first scenario is often used if it is unknown which feature groups may be parents or which are TPs. Furthermore, this scenario may also be used if the dataset is sufficiently simple, for instance, because a suspect screening with the results from `convertToSuspects` (discussed in the previous section) would reliably discriminate between parents and TPs. A workflow with the first scenario is demonstrated in the second example.

In all other cases it is recommended to use either the second or third scenario, since making a prior distinction between parent and TP feature groups greatly simplifies the dataset and reduces false positives. A relative simple example where this can be used is when there are two sample groups: before and after treatment.

```
componTP <- generateComponents(algorithm = "tp",
                               fGroups = fGroups[rGroups = "before"],
                               fGroupsTPs = fGroups[rGroups = "after"])
```

In this example, only those feature groups present in the "before" replicate group are considered as parents, and those in "after" may be considered as a TP. Since it is likely that there will be some overlap in feature groups between both sample groups, the `ignoreParents` flag can be used to not consider any of the overlap for TP assignments:

```
componTP <- generateComponents(algorithm = "tp",
                               fGroups = fGroups[rGroups = "before"],
                               fGroupsTPs = fGroups[rGroups = "after"],
                               ignoreParents = TRUE)
```

More sophisticates ways are of course possible to provide an upfront distinction between parent/TP feature groups. In the fourth example a workflow is demonstrated where fold changes are used.

> ***NOTE*** The feature groups specified for `fGroups`/`fGroupsTPs` *must* always originate from the same `featureGroups` object.

For the `library` and `biotransformer` algorithms it is mandatory that a suspect screening of parents and TPs is performed prior to componentization. This is necessary for the componentization algorithm to map the feature groups that belong to a particular parent or TP. To do so, the `convertToSuspects` function is used to prepare the suspect list:

```
# set includeParents to TRUE since both the parents and TPs are needed
suspects <- convertToSuspects(TPs, includeParents = TRUE)
fGroupsScr <- screenSuspects(fGroups, suspects, onlyHits = TRUE)

# do the componentization
# a similar distinction between fGroups/fGroupsScr as discussed above can of course also
↪  be done
componTP <- generateComponents(fGroups = fGroupsScr, ...)
```

If a parent screening was already performed in advance, for instance when the input parents to `generateTPs` are screening results, the screening results for parents and TPs can also be combined. The second example demonstrates this.

Note that in the case a parent suspect is matched to multiple feature groups, a component is made for each match. Similarly, if multiple feature groups match to a TP suspect, all of them will be incorporated in the component.

When TPs were generated with the `logic` algorithm a suspect screening must also be carried out in advance. However, in this case it is not necessary to include the parents (since each parent equals a feature group no mapping is necessary). The `onlyHits` variable to `screenSuspects` must not be set in order to keep the parents.

```
# only screen for TPs
suspects <- convertToSuspects(TPs, includeParents = FALSE)
# but keep all other feature groups as these may be parents
fGroupsScr <- screenSuspects(fGroups, suspects, onlyHits = FALSE)

# do the componentization...
```

7.2.1.2 Annotation similarity calculation

If additional annotation data for parents and TPs is given to the componentization algorithm, it will be used to calculate various similarity properties. Often, the chemical structure for a transformation product is similar to that of its parent. Hence, there is a good chance that a parent and its TPs also share similar MS/MS data.

Firstly, if MS peak lists are provided, then the spectrum similarity is calculated between each parent and its potential TP candidates. This is performed with all the three different alignment shifts (see the spectrum similarity section for more details).

In case `formulas` and/or `compounds` objects are specified, then a parent/TP comparison is made by counting the number of fragments and neutral losses that they share (by using the formula annotations). This property is mainly used for non-target workflows where the identity for a parent and TP is not yet well established. For this reason, fragments and neutral losses reported for *all* candidates for the parent/TP feature group are considered. Hence, it is highly recommend to pre-treat the annotation objects, for instance, with the `topMost` filter. If both `formulas` and `compounds` are given the results are pooled. Note that each unique fragment/neutral loss is only counted once, thus multiple formula/compound candidates with the same annotations will not skew the results.

7.2.2 Processing data

The output of TP componentization is an object of the `componentsTPs` class. This *derives* from the 'regular' `components` class, therefore, all the data processing functionality described before (extraction, subsetting, filtering etc) are also valid for TP components.

Several additional filters are available to prioritize the data:

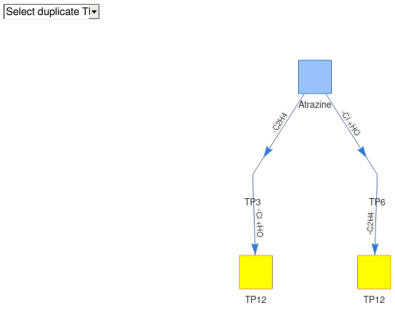
Filter	Remarks
`retDirMatch`	If `TRUE` only keep TPs with an expected chromatographic retention direction compared to the parent.
`minSpecSim`, `minSpecPrec`, `minSpecSimBoth`	The minimum spectrum similarity between the parent and TP. Calculated with no, `"precursor"` and `"both"` alignment shifting (see spectrum similarity).
`minFragMatches`, `minNLMatches`	Minimum number of formula fragment/neutral loss matches between parent and TP (discussed in previous section).
`formulas`	A `formulas` object used to further verify candidate TPs that were generated by the `logic` algorithm.

The `retDirMatch` filter compares the expected and observed *retention time direction* of a TP in order to decide if it should be kept. The direction is a value of either `-1` (TP elutes before parent), `+1` (TP elutes after parent) or `0` (TP elutes very close to the parent or its direction is unknown). The directions are taken from the generated transformation products. For the `library` and `biotransformer` algorithms the log P values are compared of a TP and its parent. Here, it is assumed that lower log P values result in earlier elution (i.e. typical with reversed phase LC). For the `logic` algorithm the retention time direction is taken from the transformation rules table. Note that specifying a large enough value for the `minRTDiff` argument to `generateComponents` is important to ensure that some tolerance exists while comparing retention time directions of parent and TPs. This filter does nothing if either the observed or expected direction is zero.

When TPs data was generated with the `logic` algorithm it is recommended to use the `formulas` filter. This filter uses formula annotations to verify that (1) a parent feature group contains the elements that are subtracted during the transformation and (2) the TP feature group contains the elements that were added during the transformation. Since the 'right' candidate formula is most likely not yet known, this filter looks at *all* candidates. Therefore, it is recommended to filter the `formulas` object, for instance, with the `topMost` filter.

Finally, the `plotGraph()` method function that was introduced exploring transformation hierarchies for structure TPs, can also incorporate componentization results to simplify the plot and mark TP hits:

```
plotGraph(TPsBT, which = "Atrazine", components = componTP)
```



7.2.3 Omitting transformation product input

The `TPs` argument to `generateComponents` can also be omitted. In this case every feature group of `fGroupTPs` is considered to be a potential TP for the potential parents specified for `fGroups`. An advantage is that the screening workflow is not limited to any known TPs or transformations. However, such a workflow has high demands on prioritiation steps before and after the componentization to rule out the many false positives that may occur.

When no transformation data is supplied it is crucial to make a prior distinction between parent and TP feature groups. Afterwards, the MS/MS spectral and other annotation similarity filters mentioned in the previous section may be a powerful way to further prioritize data.

The fourth example demonstrates such a workflow.

7.2.4 Reporting TP components

The TP components can be reported with the `report` function. This is done by setting the `components` function argument (i.e. equally to all other component types). The results will be displayed with a customized format that allows easy exploring of each parent with its TPs. In addition, the `TPs` argument can be set to include additional data such as transformation pathways.

```
report(fGroups, components = componTP, TPs = TPs)
```

7.3 Example workflows

The next subsections demonstrate several approaches to perform a TP screening workflow with `patRoon`. In all examples it is assumed that feature groups were already obtained (with the `findFeatures` and `groupFeatures` functions) and stored in the `fGroups` variable.

The workflows with `patRoon` are designed to be flexible, and the examples here are primarily meant to implement your own workflow. Furthermore, some of the techniques used in the examples can also be combined. For instance, the Fold change classification and MS/MS similarity filters applied in the fourth example could also be applied to any of the other examples.

7.3.1 Screen predicted TPs for targets

The first example is a simple workflow where TPs are predicted for a set of given parents with BioTransformer and subsequently screened. A MetFrag compound database is generated and used for annotation.

```
# predict TPs for a fixed list of parents
TPs <- generateTPs("biotransformer", parents = patRoonData::suspectsPos)

# screen for the TPs
suspectsTPs <- convertToSuspects(TPs, includeParents = FALSE)
fGroupsTPs <- screenSuspects(fGroups, suspectsTPs, adduct = "[M+H]+", onlyHits = TRUE)

# perform annotation of TPs
mslistsTPs <- generateMSPeakLists(fGroupsTPs, "mzr")
convertToMFDB(TPs, "TP-database.csv", includeParents = FALSE) # generate MetFrag database
compoundsTPs <- generateCompounds(fGroupsTPs, mslistsTPs, "metfrag", adduct = "[M+H]+",
↪  database = "csv",
                                  extraOpts = list(LocalDatabasePath =
                                  ↪  "TP-database.csv"))
```

7.3.2 Screening TPs from a library for suspects

In this example TPs of interest are obtained for the parents that surfaced from of a suspect screening. The steps of this workflow are:

1. Suspect screening parents.
2. Obtain TPs for the suspect hits from a library.
3. A second suspect screening is performed for TPs and the original parent screening results are amended. Note that the parent data is needed for componentization.
4. Both parents and TPs are annotated using a database generated from their chemical structures.
5. Some prioritization is performed by
    a. Only keeping candidate structures for which *in-silico* fragmentation resulted in at least one annotated MS/MS peak.
    b. Only keeping suspect hits with an estimated identification level of 3 or better.
6. The TP components are made and only feature groups with parent/TP assignments are kept.
7. All results are reported.

```
# step 1
fGroupsScr <- screenSuspects(fGroups, patRoonData::suspectsPos, adduct = "[M+H]+")
# step 2
TPs <- generateTPs("library", parents = fGroupsScr)

# step 3
suspects <- convertToSuspects(TPs)
fGroupsScr <- screenSuspects(fGroupsScr, suspects, adduct = "[M+H]+", onlyHits = TRUE,
↪  amend = TRUE)

# step 4
mslistsScr <- generateMSPeakLists(fGroupsScr, "mzr")
convertToMFDB(TPs, "TP-database.csv", includeParents = TRUE)
compoundsScr <- generateCompounds(fGroupsScr, mslistsScr, "metfrag", adduct = "[M+H]+",
↪  database = "csv",
```

```
                                  extraOpts = list(LocalDatabasePath =
                                  ↪  "TP-database.csv"))

# step 5a
compoundsScr <- filter(compoundsScr, minExplainedPeaks = 1)

# step 5b
fGroupsScrAnn <- annotateSuspects(fGroupsScr, MSPeakLists = mslistsScr,
                                  compounds = compoundsScr)
fGroupsScrAnn <- filter(fGroupsScrAnn, maxLevel = 3, onlyHits = TRUE)

# step 6
componTP <- generateComponents(fGroupsScrAnn, "tp", TPs = TPs, MSPeakLists = mslistsScr,
                               compounds = compoundsScr)
fGroupsScrAnn <- fGroupsScrAnn[results = componTP]

# step 7
report(fGroupsScrAnn, MSPeakLists = mslistsScr, compounds = compoundsScr,
       components = componTP, TPs = TPs)
```

7.3.3 Non-target screening of predicted TPs

This example uses metabolic logic to calculate possible TPs for all feature groups from a complete non-target screening. This example demonstrates how a workflow can be performed when little is known about the identity of the parents. The steps of this workflow are:

1. Formula annotations are performed for all feature groups.
2. These results are then limited to the top 5 candidates, and only feature groups with annotations are kept.
3. The TPs are calculated for all remaining feature groups.
4. A suspect screening is performed to find the TPs. Unlike the previous example feature groups without hits are kept (discussed here).
5. The components are generated
6. The components are filtered:
    a. The TPs must follow an expected retention time direction
    b. The parent/TPs should have at least one candidate formula that fits with the transformation.
7. Only feature groups are kept with parent/TP assignments and all results are reported.

```
# steps 1-2
mslists <- generateMSPeakLists(fGroups, "mzr")
formulas <- generateFormulas(fGroups, mslists, "genform", adduct = "[M+H]+")
formulas <- filter(formulas, topMost = 5)
fGroups <- fGroups[results = formulas]

# step 3
TPs <- generateTPs("logic", fGroups = fGroups, adduct = "[M+H]+")

# step 4
suspects <- convertToSuspects(TPs)
fGroupsScr <- screenSuspects(fGroups, suspects, adduct = "[M+H]+", onlyHits = FALSE)
```

```
# step 5
componTP <- generateComponents(fGroupsScr, "tp", TPs = TPs, MSPeakLists = mslists,
↪  formulas = formulas)

# step 6
componTP <- filter(componTP, retDirMatch = TRUE, formulas = formulas)

# step 7
fGroupsScr <- fGroupsScr[results = componTP]
report(fGroupsScr, MSPeakLists = mslists, formulas = formulas, components = componTP)
```

7.3.4 Non-target screening of TPs by annotation similarities

This example shows a workflow where no TP data from a prediction or library is used. Instead, this workflow relies on statistics and MS/MS data to find feature groups which may potentially have a parent - TP relationship. The workflow is similar to that of the previous example. The steps of this workflow are:

1. Fold changes (FC) between two sample groups are calculated to classify which feature groups are decreasing (i.e. parents) or increasing (i.e. TPs).
2. Feature groups without classification are removed.
3. Formula annotations are performed like the previous example.
4. The componentization is performed and the FC classifications are used to specify which feature groups are to be considered parents or TPs.
5. Only TPs are kept that show a high MS/MS spectral similarity and share at least one fragment with their parent.
6. Only feature groups are kept with parent/TP assignments and all results are reported.

```
# step 1
tab <- as.data.table(fGroups, FCParams = getFCParams(c("before", "after")))
groupsParents <- tab[classification == "decrease"]$group
groupsTPs <- tab[classification == "increase"]$group

# step 2
fGroups <- fGroups[, union(groupsParents, groupsTPs)]

# step 3
mslists <- generateMSPeakLists(fGroups, "mzr")
formulas <- generateFormulas(fGroups, mslists, "genform", adduct = "[M+H]+")
formulas <- filter(formulas, topMost = 5)
fGroups <- fGroups[results = formulas]

# step 4
componTP <- generateComponents(algorithm = "tp",
                               fGroups = fGroups[, groupsParents],
                               fGroupsTPs = fGroups[, groupsTPs],
                               MSPeakLists = mslists, formulas = formulas)

# step 5
componTP <- filter(componTP, minSpecSimBoth = 0.75, minFragMatches = 1)

# step 6
fGroups <- fGroups[results = componTP]
report(fGroups, MSPeakLists = mslists, formulas = formulas, components = componTP)
```

8 Advanced usage

8.1 Adducts

When generating formulae and compound annotations and some other functionalities it is required to specify the adduct species. Behind the scenes, different algorithms typically use different formats. For instance, in order to specify a protonated species...

- `GenForm` either accepts `"M+H"` and `"+H"`
- `MetFrag` either accepts the numeric code `1` or `"[M+H]+"`
- `SIRIUS` accepts `"[M+H]+"`

In addition, most algorithms only accept a limited set of possible adducts, which do not necessarily all overlap with each other. The `GenFormAdducts()` and `MetFragAdducts()` functions list the possible adducts for `GenForm` and `MetFrag`, respectively.

In order to simplify the situation `patRoon` internally uses its own format and converts it automatically to the algorithm specific format when necessary. Furthermore, during conversion it checks if the specified adduct format is actually allowed by the algorithm. Adducts in `patRoon` are stored in the `adduct` S4 class. Objects from this class specify which elements are added and/or subtracted, the final charge and the number of molecules present in the adduct (e.g. *2* for a dimer).

```
adduct(add = "H") # [M+H]+
adduct(sub = "H", charge = -1) # [M-H]-
adduct(add = "K", sub = "H2", charge = -1) # [M+K-H2]-
adduct(add = "H3", charge = 3) # [M+H3]3+
adduct(add = "H", molMult = 2) # [2M+H]+
```

A more easy way to generate adduct objects is by using the `as.adduct()` function:

```
as.adduct("[M+H]+")
as.adduct("[M+H2]2+")
as.adduct("[2M+H]+")
as.adduct("[M-H]-")
as.adduct("+H", format = "genform")
as.adduct(1, isPositive = TRUE, format = "metfrag")
```

In fact, the `adduct` argument to workflow functions such as `generateFormulas()` and `generateCompounds()` is automatically converted to an `adduct` class with the `as.adduct()` function if necessary:

```
formulas <- generateFormulas(..., adduct = adduct(sub = "H", charge = -1))
formulas <- generateFormulas(..., adduct = "[M-H]-") # same as above
```

More details can be found in the reference manual (`?adduct` and `` ?`adduct-utils` ``).

8.2 Feature intensity normalization

Feature intensities are often compared between sample analyses, for instance, to evaluate trends between sample points. However, matrix effects, varying detector sensitivity and differences in analysed sample amount may complicate such comparison. For this reason, it may be desired to *normalize* the feature intensities.

The `normInts()` function is used to normalize feature intensities (peak heights and areas). Two different types are supported:

1. **Feature normalization**: normalization occurs by intensities within the same sample analysis
2. **Group normalization**: normalization occurs by intensities among features within the same group

Both normalization types can be combined.

8.2.1 Feature normalization

Feature normalization itself supports the following normalization methods:

Method	Usage	Description
TIC	`normInts(featNorm = "tic", ...)`	Normalizes by the combined intensity of all features, also known as the Total Ion Current (TIC).
Internal Standard	`normInts(featNorm = "istd", ...)`	Uses internal standards (IS) to normalize feature intensities.
Concentration	`normInts(featNorm = "conc", ...)`	Normalizes feature intensities of a sample analysis by its *normalization concentration* (explained below).
None	`normInts(featNorm = "none", ...)`	Performs no feature normalization. Set this if you only want to perform group normalization (discussed in the next section).

8.2.1.1 Normalization concetration

All methods (except `"none"`) are influenced by the *normalization concentration*, which is a property set for each sample analysis. For IS normalization, this should equal the concentration of the IS present in the sample. Otherwise the normalization concentration resembles the injected sample amount. The normalization concentration is defined in the `norm_conc` column of the analysis information. For example:

```
# obtain analysis information as usual, but add normalization concentrations.
# The blanks are set to NA, and will therefore not be normalized.
generateAnalysisInfo(paths = patRoonData::exampleDataPath(),
                     groups = c(rep("solvent", 3), rep("standard", 3)),
                     blanks = "solvent",
                     norm_concs = c(NA, NA, NA, 2, 2, 1))
```

```
#>                                                path       analysis    group   blank norm_conc
#> 1 /usr/local/lib/R/site-library/patRoonData/extdata/pos  solvent-pos-1  solvent solvent        NA
#> 2 /usr/local/lib/R/site-library/patRoonData/extdata/pos  solvent-pos-2  solvent solvent        NA
#> 3 /usr/local/lib/R/site-library/patRoonData/extdata/pos  solvent-pos-3  solvent solvent        NA
#> 4 /usr/local/lib/R/site-library/patRoonData/extdata/pos standard-pos-1 standard solvent         2
#> 5 /usr/local/lib/R/site-library/patRoonData/extdata/pos standard-pos-2 standard solvent         2
#> 6 /usr/local/lib/R/site-library/patRoonData/extdata/pos standard-pos-3 standard solvent         1
```

The normalization concentration does not need to be an absolute value. In the end, what matters are the relative numbers between the sample analyses. For example, if the concentrations for two analyses are `c(1, 2)` or `c(1.5, 3.0)` the normalization occurs the same. Setting the concentration to `NA` (or `0`) will skip normalization for an analysis. If the normalization concentration is absent from the analysis information it will be defaulted to `1`.

8.2.1.2 Internal standard normalization

For IS normalization an internal standard list should be specified with the properties of the internal standards. Essentially, the format of this list is exactly the same as a suspect list. Example lists can be found in the `patRoonData` package:

```
patRoonData::ISTDListPos[1:5, ]
```

```
#>                  name           formula    rt
#> 1 1H-benzotriazole-D4        C6[2]H4HN3 268.1
#> 2         Atenolol-D7   C14[2]H7H15N2O3 213.5
#> 3         Atrazine-D5     C8[2]H5H9ClN5 336.5
#> 4      Bezafibrate-D6  C19[2]H6H14ClNO4 351.7
#> 5       Climbazole-D4 C15[2]H4H13ClN2O2 359.1
```

As can be seen from above, labelled isotopes can be specified with square brackets, *e.g.* [2]H for deuterium.

The next step is to perform the normalization with `normInts()`:

```
fGroupsNorm <- normInts(fGroups, featNorm = "istd", standards = patRoonData::ISTDListPos,
↪  adduct = "[M+H]+",
                        ISTDRTWindow = 20, ISTDMZWindow = 200, minISTDs = 2)
```

This will do the following:

- Perform a suspect screening to find the specified IS (`standards` argument).
- Remove the IS candidates which are absent in one or more of the analyses to be normalized.
- Select IS candidates for each feature group, based on close retention time (`ISTDRTWindow` argument), *m/z* (`ISTDMZWindow` argument) and a minimum number (`minISTDs`). If the number of IS candidates within specified retention time and *m/z* windows is below `minISTDs`, the close(st) candidate(s) outside these windows are additionally chosen.
- Normalization of features is performed with the combined IS intensities.

To evaluate the assignments for a particular feature group, the `internalStandardAssignments()` function and `plotGraph()` functions can be used:

```
fg <- names(fGroupsNorm)[2]
internalStandardAssignments(fGroupsNorm)[[fg]] # IS assignments for 2nd feature group
```

```
#> [1] "M221_R336_292" "M284_R323_569" "M213_R340_263"
```

```
plotGraph(fGroupsNorm) # interactively explore assignments
```

8.2.1.3 Other methods

Like IS normalization, other feature normalization methods also occurs with `normInts()`:

```
fGroupsNorm <- normInts(fGroups, featNorm = "tic") # TIC normalization
fGroupsNorm <- normInts(fGroups, featNorm = "conc") # Concentration normalization
```

8.2.2 Group normalization

Normalizing feature intensities among group member is easily performed by setting `groupNorm=TRUE`:

```
# only perform group normalization
fGroupsNorm <- normInts(fGroups, featNorm = "none", groupNorm = TRUE)
# first perform TIC feature normalization and then group normalization
fGroupsNorm <- normInts(fGroups, featNorm = "tic", groupNorm = TRUE)
```

8.2.3 Using normalized intensities

The normalized intensity (peak heigh/area) values can easily be obtained with `as.data.table()`:

```
as.data.table(fGroupsNorm, normalized = TRUE)[1:5]
```

```
#>           group      ret       mz standard-pos-1 standard-pos-2 standard-pos-3
#>          <char>    <num>    <num>          <num>          <num>          <num>
#> 1: M109_R192_20 191.8717 109.0759       2.328459      2.1068991      0.9688233
#> 2: M111_R330_23 330.4078 111.0439       0.476554      0.4156571      0.2109971                M221_R3
#> 3: M114_R269_25 268.6906 114.0912       1.006808      1.1271519      0.5722654
#> 4: M116_R317_29 316.7334 116.0527       3.804086      3.8240928      2.1151499 M284_R323_569,M198_R3
#> 5: M120_R268_30 268.4078 120.0554       3.376374      3.0432604      1.4157580
```

```
# can be combined with other parameters
as.data.table(fGroupsNorm, normalized = TRUE, average = TRUE, areas = TRUE)[1:5]
```

```
#>           group      ret       mz  standard                                                ISTD_assigned
#>          <char>    <num>    <num>     <num>                                                       <char>
#> 1: M109_R192_20 191.8717 109.0759 3.2597655                                M280_R212_561,M274_R214_532
#> 2: M111_R330_23 330.4078 111.0439 0.2753524                  M221_R336_292,M284_R323_569,M213_R340_263
#> 3: M114_R269_25 268.6906 114.0912 0.8325869                                M300_R262_608,M275_R294_537
#> 4: M116_R317_29 316.7334 116.0527 2.6500817 M284_R323_569,M198_R310_215,M285_R301_570,M221_R336_292
#> 5: M120_R268_30 268.4078 120.0554 1.8965138                                M300_R262_608,M275_R294_537
```

```
# feature values (no need to set normalized=TRUE)
as.data.table(fGroupsNorm, features = TRUE)[1:5, .(group, analysis, intensity_rel,
↪  area_rel)]
```

```
#>           group       analysis intensity_rel  area_rel
#>          <char>         <char>         <num>     <num>
#> 1: M109_R192_20 standard-pos-1     2.3284588 3.9777827
#> 2: M109_R192_20 standard-pos-2     2.1068991 4.0198440
#> 3: M109_R192_20 standard-pos-3     0.9688233 1.7816697
#> 4: M111_R330_23 standard-pos-1     0.4765540 0.3352008
#> 5: M111_R330_23 standard-pos-2     0.4156571 0.3251259
```

Several other patRoon functions also accept the normalized argument to use normalized data, such as plotInt() (discussed here), plotVolcano() (discussed here) and generateComponents() with intensity clustering (discussed here).

8.2.4 Default normalization

If normalized data is requested (normalized=TRUE, see previous section) and normInts() was *not* called on the feature group data, a *default normalization* will occur. This is nothing more than a group normalization (normInts(groupNorm=TRUE, ...)), and was mainly implemented to ensure backwards compatibility with previous patRoon versions.

8.3 Feature parameter optimization

Many different parameters exist that may affect the output quality of feature finding and grouping. To avoid time consuming manual experimentation, functionality is provided to largely automate the optimization process. The methodology, which uses design of experiments (DoE), is based on the excellent Isotopologue Parameter Optimization (IPO) R package. The functionality of this package is directly integrated in patRoon. Some functionality was added or changed, the most important being support for other feature finding and grouping algorithms besides XCMS and basic optimization support for qualitative parameters. Nevertheless, the core optimization algorithms are largely untouched.

This section will introduce the most important concepts and functionality. Please see the reference manual for more information (e.g. ?`feature-optimization`).

> ***NOTE*** The SIRIUS and SAFD algorithms are currently not (yet) supported.

8.3.1 Parameter sets

Before starting an optimization experiment we have to define *parameter sets*. These sets contain the parameters and (initial) numeric ranges that should be tested. A parameter set is defined as a regular list, and can be easily constructed with the generateFeatureOptPSet() and generateFGroupsOptPSet() functions (for feature finding and feature grouping, respectively).

```
pSet <- generateFeatureOptPSet("openms") # default test set for OpenMS
pSet <- generateFeatureOptPSet("openms", chromSNR = c(5, 10)) # add parameter
# of course manually making a list is also possible (e.g. if you don't want to test the
↪  default parameters)
pSet <- list(noiseThrInt = c(1000, 5000))
```

When optimizing with XCMS or KPIC2 a few things have to be considered. First of all, when using the XCMS3 interface (i.e. algorithm="xcms3") the underlying method that should be used for finding and grouping features and retention alignment should be set. In case these are not set default methods will be used.

```
pSet <- list(method = "centWave", ppm = c(2, 8))
pSet <- list(ppm = c(2, 8)) # same: centWave is default

# get defaults, but for different grouping/alignment methods
pSetFG <- generateFGroupsOptPSet("xcms3", groupMethod = "nearest", retAlignMethod =
↪  "peakgroups")
```

In addition, when optimizing feature grouping (both `XCMS` interfaces and `KPIC2`) we need to set the grouping and retention alignment parameters in two different nested lists: these are `groupArgs`/`retcorArgs` (`algorithm="xcms"`), `groupParams`/`retAlignParams` (`algorithm="xcms3"`) or `groupArgs`/`alignArgs` (`algorithm="kpic2"`).

```
pSetFG <- list(groupParams = list(bw = c(20, 30))) # xcms3
pSetFG <- list(retcorArgs = list(gapInit = c(0, 7))) # xcms
pSetFG <- list(groupArgs = list(mz_weight = c(0.3, 0.9))) # kpic2
```

When a parameter set has been defined it should be used as input for the `optimizeFeatureFinding()` or `optimizeFeatureGrouping()` functions.

```
ftOpt <- optimizeFeatureFinding(anaInfo, "openms", pSet)
fgOpt <- optimizeFeatureGrouping(fList, "openms", pSetFG) # fList is an existing features
↪  object
```

Similar to `findFeatures()`, the first argument to `optimizeFeatureFinding()` should be the analysis information. Note that it is not uncommon to perform the optimization with only a subset of (representative) analyses (i.e. to reduce processing time).

```
ftOpt <- optimizeFeatureFinding(anaInfo[1:2, ], "openms", pSet) # only use first two
↪  analyses
```

From the parameter set a design of experiment will be automatically created. Obviously, the more parameters are specified, the longer such an experiment will take. After an experiment has finished, the optimization algorithm will start a new experiment where numeric ranges for each parameter are increased or decreased in order to more accurately find optimum values. Hence, the numeric ranges specified in the parameter set are only *initial* ranges, and will be changed in subsequent experiments. After each experiment iteration the results will be evaluated and a new experiment will be started as long as better results were obtained during the last experiment (although there is a hard limit defined by the `maxIterations` argument).

For some parameters it is recommended or even necessary to set hard limits on the numeric ranges that are allowed to be tested. For instance, setting a minimum feature intensity threshold is highly recommended to avoid excessive processing time and potentially suboptimal results due to excessive amounts of resulting features. Configuring absolute parameter ranges is done by setting the `paramRanges` argument.

```
# set minimum intensity threshold (but no max)
ftOpt <- optimizeFeatureFinding(anaInfo, "openms",
                                list(noiseThrInt = c(1000, 5000)), # initial testing
                                ↪  range
                                paramRanges = list(noiseThrInt = c(500, Inf))) # never
                                ↪  test below 500
```

Depending on the used algorithm, several default absolute limits are imposed. These may be obtained with the `getDefFeaturesOptParamRanges()` and `getDefFGroupsOptParamRanges()` functions.

The common operation is to optimize numeric parameters. However, parameters that are not numeric (i.e. *qualitative*) need a different approach. In this case you will need to define multiple parameter sets, where each set defines a different qualitative value.

```
ftOpt <- optimizeFeatureFinding(anaInfo, "openms",
                                list(chromFWHM = c(4, 8), isotopeFilteringModel =
                                ↪  "metabolites (5% RMS)"),
                                list(chromFWHM = c(4, 8), isotopeFilteringModel =
                                ↪  "metabolites (2% RMS)"))
```

In the above example there are two parameter sets: both define the numeric `chromFWHM` parameter, whereas the qualitative `isotopeFilteringModel` parameter has a different value for each. Note that we had to specify the `chromFWHM` twice, this can be remediated by using the `templateParams` argument:

```
ftOpt <- optimizeFeatureFinding(anaInfo, "openms",
                                list(isotopeFilteringModel = "metabolites (5% RMS)"),
                                list(isotopeFilteringModel = "metabolites (2% RMS)"),
                                templateParams = list(chromFWHM = c(4, 8)))
```

As its name suggests, the `templateParams` argument serves as a template parameter set, and its values are essentially combined with each given parameter set. Note that current support for optimizing qualitative parameters is relatively basic and time consuming. This is because tests are essentially repeated for each parameter set (e.g. in the above example the `chromFWHM` parameter is optimized twice, each time with a different value for `isotopeFilteringModel`).

8.3.2 Processing optmization results

The results of an optimization process are stored in objects from the S4 `optimizationResult` class. Several methods are defined to inspect and visualize these results.

The `optimizedParameters()` function is used to inspect the best parameter settings. Similarly, the `optimizedObject()` function can be used to obtain the object that was created with these settings (i.e. a `features` or `featureGroups` object).

```
optimizedParameters(ftOpt) # best settings for whole experiment
```

```
#> $chromFWHM
#> [1] 3.75
#>
#> $mzPPM
#> [1] 13.5
#>
#> $minFWHM
#> [1] 1.4
#>
#> $maxFWHM
#> [1] 35
```

```
optimizedObject(ftOpt) # features object with best settings for whole experiment
```

```
#> A featuresOpenMS object
#> Hierarchy:
#> features
#>     |-- featuresOpenMS
#> ---
#> Object size (indication): 666.5 kB
#> Algorithm: openms
#> Total feature count: 4128
#> Average feature count/analysis: 4128
```

```
#> Analyses: solvent-pos-1 (1 total)
#> Replicate groups: solvent-pos (1 total)
#> Replicate groups used as blank: solvent-pos (1 total)
```

Results can also be obtained for specific parameter sets/iterations.

```
optimizedParameters(ftOpt, 1) # best settings for first parameter set
```

```
#> $chromFWHM
#> [1] 3.75
#>
#> $mzPPM
#> [1] 13.5
#>
#> $minFWHM
#> [1] 1.4
#>
#> $maxFWHM
#> [1] 35
```

```
optimizedParameters(ftOpt, 1, 1) # best settings for first parameter set and experiment
↪  iteration
```

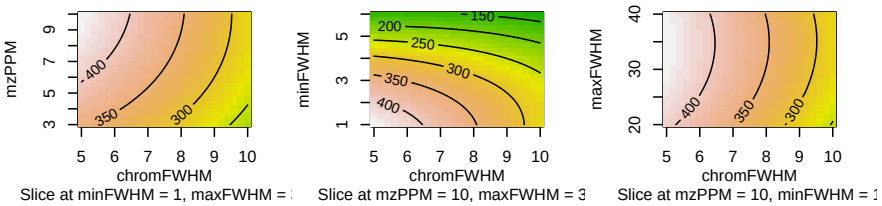
```
#> $chromFWHM
#> [1] 5
#>
#> $mzPPM
#> [1] 10
#>
#> $minFWHM
#> [1] 1
#>
#> $maxFWHM
#> [1] 35
```

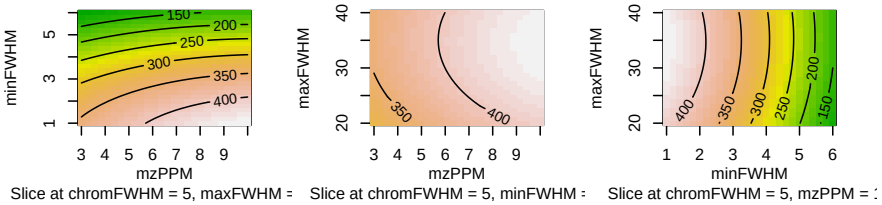
```
optimizedObject(ftOpt, 1) # features object with best settings for first parameter set
```

```
#> A featuresOpenMS object
#> Hierarchy:
#> features
#>     |-- featuresOpenMS
#> ---
#> Object size (indication): 666.5 kB
#> Algorithm: openms
#> Total feature count: 4128
#> Average feature count/analysis: 4128
#> Analyses: solvent-pos-1 (1 total)
#> Replicate groups: solvent-pos (1 total)
#> Replicate groups used as blank: solvent-pos (1 total)
```

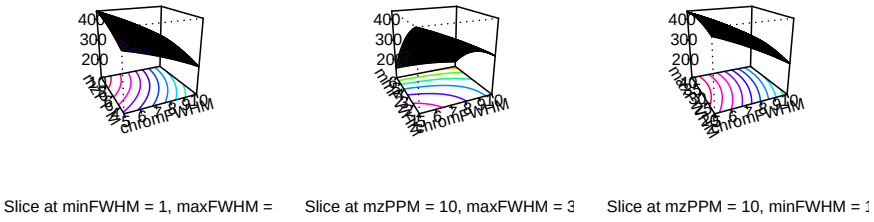
The `plot()` function can be used to visualize optimization results. This function will plot graphs for results of all tested parameter pairs. The graphs can be contour, image or perspective plots (as specified by the `type` argument).

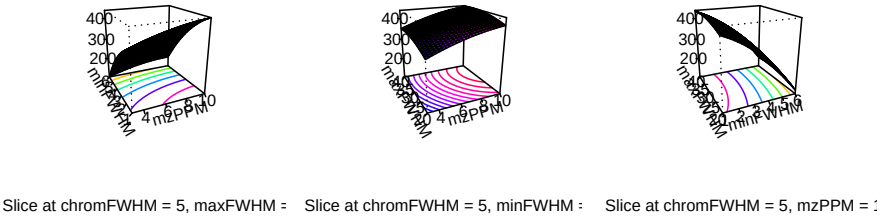
```
plot(ftOpt, paramSet = 1, DoEIteration = 1) # contour plots for first param
↪  set/experiment
```






```
plot(ftOpt, paramSet = 1, DoEIteration = 1, type = "persp") # pretty perspective plots
```





Please refer to the reference manual for more methods to inspect optimization results (e.g. `?optimizationResult`).

8.4 Chromatographic peak qualities

The algorithms used by `findFeatures` detect chromatographic peaks automatically to find the features. However, it is common that not all detected features have 'proper' chromatographic peaks, and some features could be just noise. The MetaClean `R` package supports various quality measures for chromatographic peaks. The quality measures include Gaussian fit, symmetry, sharpness and others. In addition, MetaClean averages all feature data for each feature group and adds a few additional group specific quality measures (*e.g.* retention time consistency). Please see Chetnik, Petrick, and Pandey (2020) for more details. The

calculations are integrated into patRoon, and are easily performed with the calculatePeakQualities() generic function.

```
fList <- calculatePeakQualities(fList) # calculate for all features
fGroups <- calculatePeakQualities(fGroups) # calculate for all features and groups
```

Most often the featureGroups method is only used, unless you want to filter features (discussed below) prior to grouping.

An extension in patRoon is that the qualities are used to calculate *peak scores*. The score for each quality measure is calculated by normalizing and scaling the values into a 0-1 range, where zero is the worst and one the best. Note that most scores are relative, hence, the values should only be used to compare features among each other. Finally, a totalScore is calculated which sums all individual scores and serves as a rough overall score indicator for a feature (group).

The qualities and scores are easily obtained with the as.data.table() function.

```
# (limit rows/columns for clarity)
as.data.table(fList)[1:5, 26:30]
```

```
#>    GaussianSimilarityScore SharpnessScore TPASRScore ZigZagScore totalScore
#>                      <num>          <num>      <num>       <num>      <num>
#> 1:               0.6314046   3.443351e-02  0.9956949   0.9103221   6.302180
#> 2:               0.9633994   9.900530e-10  0.9944988   0.3565674   6.513205
#> 3:               0.3613087   7.565147e-10  0.8006569   0.9999449   5.651379
#> 4:               0.9151027   8.600747e-03  0.9405262   0.9637153   5.892201
#> 5:               0.3676623   1.000000e+00  0.9907657   0.8435805   5.825267
```

```
# the qualities argument is necessary to include the scores.
# valid values are: "quality", "score" or "both"
as.data.table(fGroups, qualities = "both")[1:5, 25:29]
```

```
#>    TPASRScore ZigZagScore ElutionShiftScore RetentionTimeCorrelationScore totalScore
#>         <num>       <num>             <num>                        <num>      <num>
#> 1:  0.7305554   0.9962254         0.8421657                    0.9955769   7.932541
#> 2:  0.0000000   0.9744541         0.9960804                    0.7746038   6.029360
#> 3:  0.6140008   0.9171568         0.9015949                    0.9776651   7.480675
#> 4:  0.8227904   0.8907734         0.9403958                    0.9963785   8.451631
#> 5:  0.9848653   0.8667116         0.5754979                    0.9984902   8.740135
```

The feature quality values can also be reviewed interactively with reports generated with report (see Reporting) and with checkFeatures (see here). The filter function can be used filter out low scoring features and feature groups:

```
# only keep features with at least 0.3 Modality score and 0.5 symmetry score
fList <- filter(fList, qualityRange = list(ModalityScore = c(0.3, Inf),
                                           SymmetryScore = c(0.5, Inf)))

# same as above
fGroups <- filter(fGroups, featQualityRange = list(ModalityScore = c(0.3, Inf),
                                                   SymmetryScore = c(0.5, Inf)))

# filter group averaged data
fGroups <- filter(fGroups, groupQualityRange = list(totalScore = c(0.5, Inf)))
```

8.4.1 Applying machine learning with MetaClean

An important feature of MetaClean is to use the quality measures to train a machine learning model to automatically recognize 'good' and 'bad' features. `patRoon` provides a few extensions to simplify training and using a model. Furthermore, while `MetaClean` was primarily designed to work with XCMS, the extensions of `patRoon` allow the usage of data from all the algorithms supported by `patRoon`.

The `getMCTrainData` function can be used to convert data from a feature check session to training data that can be used by MetaClean. This allows you to use interactively select good/bad peaks. The workflow looks like this:

```
# untick the 'keep' checkbox for all 'bad' feature groups
checkFeatures(fGroupsTrain, "train_session.yml")

# get train data. This gives comparable data as MetaClean::getPeakQualityMetrics()
trainData <- getMCTrainData(fGroupsTrain, "train_session.yml")

# use train data with MetaClean with MetaClean::runCrossValidation(),
# MetaClean::getEvaluationMeasures(), MetaClean::trainClassifier() etc
# --> see the MetaClean vignette for details
```

Once you have created a model with MetaClean it can be used with the `predictCheckFeaturesSession()` function:

```
predictCheckFeaturesSession(fGroups, "model_session.yml", model)
```

This will generate another *check session file*: all the feature groups that are considered good will be with a 'keep' state, the others without. As described elsewhere, the `checkFeatures` function is used to review the results from a session and the `filter` function can be used to remove unwanted feature groups. Note that `calculatePeakQualitites()` must be called before `getMCTrainData`/`predictCheckFeaturesSession` can be used.

> ***NOTE*** `MetaClean` only predicts at the feature group level. Thus, only the kept feature groups from a *feature check session* will be used for training, and any indivual features that were marked as removed will be ignored.

8.5 Exporting and converting feature data

The feature group data obtained during the workflow can be exported to various formats with the `export()` generic function. There are currently three formats supported: `"brukerpa"` (Bruker ProfileAnalysis), `"brukertasq"` (Bruker TASQ) and `"mzmine"` (mzMine). The former exports a 'bucket table' which can be loaded in ProfileAnalysis, the second and third export a target list that can be processed with TASQ and mzMine, respectively.

The `getXCMSSet()` function converts a `features` or `featureGroups` object to an `xcmsSet` object which can be used for further processing with xcms. Similarly, the `getXCMSnExp()` function can be used for conversion to an XCMS3 style `XCMSnExp` object, and the `getPICSet()` function can be used to convert `features` to KPIC2 data.

Some examples for these functions are shown below.

```
export(fGroups, "brukertasq", out = "my_targets.csv")

# convert features to xcmsSet.
# NOTE: loadRawData should only be FALSE when the analysis data files cannot be
# loaded by the algorithm (e.g. when features were obtained with DataAnalysis and data
↪  was not exported to mz(X)ML)
xset <- getXCMSSet(fList, loadRawData = TRUE)
xsetg <- getXCMSSet(fGroups, loadRawData = TRUE) # get grouped xcmsSet

# using the new XCMS3 interface
xdata <- getXCMSnExp(fList)
xdata <- getXCMSnExp(fGroups)

# KPIC2 conversion. Like XCMS it optionally loads the raw data.
picSet <- getPICSet(fList, loadRawData = TRUE)
```

8.6 Algorithm consensus

With `patRoon` you have the option to choose between several algorithms for most workflow steps. Each algorithm is typically characterized by its efficiency, robustness, and may be optimized towards certain data properties. Comparing their output is therefore advantageous in order to design an optimum workflow. The `consensus()` generic function will compare different results from different algorithms and returns a *consensus*, which may be based on minimal overlap, uniqueness or simply a combination of all results from involved objects. The output from the `consensus()` function is of similar type as the input types and is therefore compatible to any 'regular' further data processing operations (e.g. input for other workflow steps or plotting). Note that a consensus can also be made from objects generated by the same algorithm, for instance, to compare or combine results obtained with different parameters (e.g. different databases used for compound annotation).

The `consensus()` generic is defined for most workflow objects. Some of its common function arguments are listed below.

Argument	Classes	Remarks
`obj, ...`	All	Two or more objects (of the same type) that should be compared to generate the consensus.
`compThreshold`, `relAbundance`, `absAbundance`, `formThreshold`	`compounds`, `formulas`, `featureGroupsComparison`	The minimum overlap (relative/absolute) for a result (feature, candidate) to be kept.
`uniqueFrom`	`compounds`, `formulas`, `transformationProductsStructure`, `featureGroupsComparison`	Only keep *unique* results from specified objects.
`uniqueOuter`	`compounds`, `formulas`, `transformationProductsStructure`, `featureGroupsComparison`	Should be combined with `uniqueFrom`. If `TRUE` then only results are kept which are *also* unique between the objects specified with `uniqueFrom`.

Note that current support for generating a consensus between `components` objects is very simplistic; here results are not compared, but the consensus simply consists a combination of all the components from each object.

Generating a consensus for feature groups involves first generating a `featureGroupsComparison` object. This step is necessary since (small) deviations between retention times and/or mass values reported by different

feature finding/grouping algorithms complicates a direct comparison. The comparison objects are made by the `comparison()` function, and its results can be visualized by the plotting functions discussed in the previous chapter.

Some examples are shown below

```
compoundsCons <- consensus(compoundsMF, compoundsSIR) # combine MetFrag/SIRIUS results
compoundsCons <- consensus(compoundsMF, compoundsSIR,
                           compThreshold = 1) # only keep results that overlap

TPsCons <- consensus(TPsLib, TPsBT) # combine library and BioTransformer TPs

fGroupComp <- comparison(fGroupsXCMS, fGroupsOpenMS, fGroupsEnviPick,
                         groupAlgo = "openms")
plotVenn(fGroupComp) # visualize overlap/uniqueness
fGroupsCons <- consensus(fGroupComp,
                         uniqueFrom = 1:2) # only keep results unique in OpenMS+XCMS
fGroupsCons <- consensus(fGroupComp,
                         uniqueFrom = 1:2,
                         uniqueOuter = TRUE) # as above, but also exclude any overlap
                         ↪  between OpenMS/XCMS
```

8.7 MS libraries

The `loadMSLibraries()` function is used to load MS spectral libraries, and was already briefly introduced for compound annotation. Currently, loading of MSP files and MoNA JSON files is supported, while loading of formula annotations for MS peaks is currently only supported for the latter. The underlying algorithms implement several optimizations to efficiently load large number of records. Furthermore, `loadMSLibraries()` automatically verifies record data such as formulas, adducts and masses, and automatically calculates missing or invalid data where possible.

```
mslibraryMSP <- loadMSLibrary("MoNA-export-CASMI_2016.msp", "msp")
mslibraryJSON <- loadMSLibrary("MoNA-export-CASMI_2016.json", "json")
```

Several advanced parameters are available that influence the loading of MS library data, see the reference manual (`?loadMSLibrary`) for details.

Once loaded, the usual methods are available to inspect its data:

```
show(mslibraryMSP)
```

```
#> A MSLibrary object
#> Hierarchy:
#> workflowStep
#>     |-- MSLibrary
#> ---
#> Object size (indication): 101.6 kB
#> Algorithm: msp
#> Total records: 26
#> Total peaks: 318
#> Total annotated peaks: 0 (0.00%)
```

```
mslibraryMSP[[1]] # MS/MS spectrum for first candidate
```

```
#>           mz  intensity
#>        <num>      <num>
#>  1: 135.0441   1.001001
#>  2: 161.0594   0.500501
#>  3: 163.0379   0.600601
#>  4: 173.0590   0.200200
#>  5: 176.0699   0.200200
#> ---
#> 44: 353.1191   1.201201
#> 45: 354.1323 100.000000
#> 46: 355.1351  20.820821
#> 47: 356.1374   2.702703
#> 48: 357.1401   0.300300
```

```
mslibraryJSON[["SM801601"]] # a record with annotations
```

```
#>          mz  intensity annotation
#>       <num>      <num>     <char>
#> 1:  65.0388   0.100228       C5H5
#> 2:  91.0541   0.922448       C7H7
#> 3:  93.0573   5.489900      C6H7N
#> 4: 106.0651   0.101855      C7H8N
#> 5: 108.0807 100.000000     C7H10N
#> 6: 109.0648   2.004170      C7H9O
#> 7: 132.0807   0.926004     C9H10N
#> 8: 150.0913  76.554515    C9H12NO
```

```
 # overview of all metadata (select few columns for readability)
records(mslibraryJSON)[, .(DB_ID, Name, InChIKey, formula)]
```

```
#>         DB_ID                              Name                    InChIKey          formula
#>        <char>                            <char>                      <char>           <char>
#>   1: SM800003                       1,2,3-Triazole QWENRTYMTSOGBR-UHFFFAOYSA-N           C2H3N3
#>   2: SM800201                       1-Naphthylamine RUFPHBVGCFYCNW-UHFFFAOYSA-N          C10H9N
#>   3: SM800553              2,3-Dihydroxybiphenyl YKOQAAJBYBTSBS-UHFFFAOYSA-N        C12H10O2
#>   4: SM800653                   2,4-Dibromphenol FAXWFCTVSHEODL-UHFFFAOYSA-N        C6H4Br2O
#>   5: SM800802                  2-Aminoanthracene YCSBALJAGZKWFF-UHFFFAOYSA-N         C14H11N
#>  ---
#> 618: SM884401                    Anthranilic acid RWZYAGGXGHYGMB-UHFFFAOYSA-N         C7H7NO2
#> 619: SM884552                    Fipronil sulfide FQXWEKADCSXYOC-UHFFFAOYSA-N   C12H4Cl2F6N4S
#> 620: SM884652                    Fipronil sulfone LGHZJDKSVUTELU-UHFFFAOYSA-N C12H4Cl2F6N4O2S
#> 621: SM884701 N-Cyclohexyl-2-benzothiazol-amine UPWPIFMHSFSVLE-UHFFFAOYSA-N       C13H16N2S
#> 622: SM884952                Fipronil desulfinyl JWKXVHLIRTVXLD-UHFFFAOYSA-N    C12H4Cl2F6N4
```

```
# convert all data to a data.table (may be huge!)
as.data.table(mslibraryMSP)[, .(DB_ID, SMILES, formula, mz, intensity)]
```

```
#> Key: <DB_ID>
```

```
#>         DB_ID                                                        SMILES   formula       mz inte
#>        <char>                                                       <char>    <char>    <num>     
#>   1: SMI00001 CN1CC2=C(C=CC3=C2OCO3)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OCO6 C20H19NO5 135.0441  1.0
#>   2: SMI00001 CN1CC2=C(C=CC3=C2OCO3)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OCO6 C20H19NO5 161.0594  0.5
#>   3: SMI00001 CN1CC2=C(C=CC3=C2OCO3)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OCO6 C20H19NO5 163.0379  0.6
#>   4: SMI00001 CN1CC2=C(C=CC3=C2OCO3)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OCO6 C20H19NO5 173.0590  0.2
#>   5: SMI00001 CN1CC2=C(C=CC3=C2OCO3)[C@@H]4[C@H]1C5=CC6=C(C=C5C[C@@H]4O)OCO6 C20H19NO5 176.0699  0.2
#>  ---
#> 314: SMI00172                                  C1=CC=C(C=C1)NN=CC2=CC=CC=C2N  C13H13N3 120.0678 22.1
#> 315: SMI00172                                  C1=CC=C(C=C1)NN=CC2=CC=CC=C2N  C13H13N3 121.0756  6.5
#> 316: SMI00172                                  C1=CC=C(C=C1)NN=CC2=CC=CC=C2N  C13H13N3 167.0729 28.0
#> 317: SMI00172                                  C1=CC=C(C=C1)NN=CC2=CC=CC=C2N  C13H13N3 168.0810 13.5
#> 318: SMI00172                                  C1=CC=C(C=C1)NN=CC2=CC=CC=C2N  C13H13N3 195.0917  8.2
```

Furthermore, like many other objects in `patRoon`, the MS library objects can be subset and filtered:

```
mslibrarySub <- mslibrary[1:100] # only keep first 100 records

# only keep records a neutral mass of 100-200
mslibraryF <- filter(mslibrary, massRange = c(100, 200))
# remove records with neutral mass below 100
mslibraryF <- filter(mslibrary, massRange = c(0, 100), negate = TRUE)
# only keep mass peaks with m/z 100-500
mslibraryF <- filter(mslibrary, mzRangeSpec = c(100, 500))
# remove low intensity peaks (<1%) and only keep top 10
mslibraryF <- filter(mslibrary, relMinIntensity = 0.01, topMost = 10)
# only keep mass peak with annotations
mslibraryF <- filter(mslibraryJSON, onlyAnnotated = TRUE)
```

In addition, the `properties` filter may be useful to tailor the library data. The library properties can be obtained as following:

```
names(records(mslibrary)) # get all property names
```

```
#>  [1] "Name"            "Synon"           "DB_ID"           "InChIKey"
#>  [5] "InChI"           "Precursor_type"  "Spectrum_type"   "PrecursorMZ"
#>  [9] "Instrument_type" "Instrument"      "Ion_mode"        "Collision_energy"
#> [13] "formula"         "MW"              "neutralMass"     "Comments"
#> [17] "SMILES"          "SPLASH"          "CAS"             "PubChemCID"
#> [21] "ChemSpiderID"    "Ionization"      "Resolution"
```

```
unique(records(mslibrary)[["Instrument_type"]]) # Get the available instrument types
```

```
#> [1] "LC-ESI-QTOF"  "LC-APCI-ITFT" "APCI-ITFT"
```

Then to filter the MS library:

```
# only keep APCI instrument types
mslibraryF <- filter(mslibrary, properties = list(Instrument_type = c("LC-APCI-ITFT",
↪  "APCI-ITFT")))
# remove Q-TOF by negation
mslibraryF <- filter(mslibrary, properties = list(Instrument_type = "LC-ESI-QTOF"),
↪  negate = TRUE)
```

More advanced filtering can be performed with the `delete()` generic function, see the reference manual for details (`?MSLibrary`).

Finally, functionality exists to convert, export and merge MS libraries:

```
# Convert the MS library to a suspect list.
# By setting collapse to TRUE, all records with the same first block InChIKey
# are collapsed and mass peaks are averaged.
suspL <- convertToSuspects(mslibrary, adduct = "[M+H]+", collapse = TRUE)
# Amend custom suspect list with library data (fragments_mz column)
suspL <- convertToSuspects(mslibrary, adduct = "[M+H]+", suspects =
↪  patRoonData::suspectsPos)

export(mslibrary, out = "myMSLib.msp") # export to a new MSP library

mslibraryM <- merge(mslibraryMSP, mslibraryJSON) # merge two libraries
```

8.8 Compound clustering

When large databases such as PubChem or ChemSpider are used for compound annotation, it is common to find *many* candidate structures for even a single feature. While choosing the right scoring settings can significantly improve their ranking, it is still very much possible that many candidates of potential interest remain. In this situation it might help to perform *compound clustering*. During this process, all candidates for a feature are clustered hierarchically on basis of similar chemical structure. From the resulting cluster the *maximum common substructure* (MCS) can be derived, which represents the largest possible substructure that 'fit' in all candidates. By visual inspection of the MCS it may be possible to identify likely common structural properties of a feature.

In order to perform compound clustering the `makeHCluster()` generic function should be used. This function heavily relies on chemical fingerprinting functionality provided by rcdk.

```
compounds <- generateCompounds(...) # get our compounds
compsClust <- makeHCluster(compounds)
```

This function accepts several arguments to fine tune the clustering process:

- `method`: the clustering method (e.g. `"complete"` (default), `"ward.D2"`), see `?hclust` for options
- `fpType`: finger printing type (`"extended"` by default), see `?get.fingerprint`
- `fpSimMethod`: similarity method for generating the distance method (`"tanimoto"` by default), see `?fp.sim.matrix`

For all arguments see the reference manual (`?makeHClust`).

The resulting object is of type `compoundsCluster`. Several methods are defined to modify and inspect these results:

```
# plot MCS of first cluster from candidates of M192_R355_191
plotStructure(compsClust, groupName = "M192_R355_191", 1)

# plot dendrogram
plot(compsClust, groupName = "M192_R355_191")

# re-assign clusters for a feature group
```

```
compsClust <- treeCut(compsClust, k = 5, groupName = "M192_R355_191")
# ditto, but automatic cluster determination
compsClust <- treeCutDynamic(compsClust, minModuleSize = 3, groupName = "M192_R355_191")
```

For a complete overview see the reference manual (`?compoundsCluster`).

8.9 Feature regression analysis

Some basic support in `patRoon` is available to perform simple linear regression on feature intensities vs given experimental conditions. Examples of such conditions are dilution factor, sampling time or initial concentration of a parent in a degradation experiment. By testing if there is a significant linearity, features of interest can be isolated in a relative easy way. Originally, this functionality was implemented as a very basic method to perform rough calculations of concentrations. However, the next sections describes a much better way by using the `MS2Quant` package. Regardless, this functionality still uses 'concentrations' as terminology for the experimental conditions of interest. The conditions are specified in the `conc` column of the analysis information, for instance:

```
# obtain analysis information as usual, but add some experimental parameters of interest
↪  (dilution, time etc).
# The blanks are set to NA, whereas the standards are set to increasing levels.
anaInfo <- generateAnalysisInfo(paths = patRoonData::exampleDataPath(),
                                groups = c(rep("solvent", 3), rep("standard", 3)),
                                blanks = "solvent",
                                concs = c(NA, NA, NA, 1, 2, 3))
```

If no experimental conditions are available for a particular analysis then the `conc` value should be `NA`. For these analyses the experimental condition will be calculated using the regression model obtained from the other analyses.

The `as.data.table()` function (or `as.data.frame()`) can then be used to calculate regression data:

```
# use areas for quantitation and make sure that feature data is reported
# (only relevant columns are shown for clarity)
as.data.table(fGroups, areas = TRUE, features = TRUE, regression = TRUE)
```

```
#>              group  conc        RSQ intercept  slope   conc_reg
#>             <char> <num>      <num>     <num>  <num>      <num>
#>   1:  M109_R192_20     1 0.71446367 193338.67  -4928  1.3649892
#>   2:  M109_R192_20     2 0.71446367 193338.67  -4928  1.2700216
#>   3:  M109_R192_20     3 0.71446367 193338.67  -4928  3.3649892
#>   4:  M111_R330_23     1 0.08902714  85338.67   -370 -0.8468468
#>   5:  M111_R330_23     2 0.08902714  85338.67   -370  5.6936937
#>  ---
#> 419: M407_R239_672     2 0.99560719 210036.00 -11734  2.0767002
#> 420: M407_R239_672     3 0.99560719 210036.00 -11734  2.9616499
#> 421: M425_R319_676     1 0.46488086 193198.67  10896  1.6194322
#> 422: M425_R319_676     2 0.46488086 193198.67  10896  0.7611356
#> 423: M425_R319_676     3 0.46488086 193198.67  10896  3.6194322
```

The calculated experimental conditions are stored in the `conc_reg` column (this column is only present if `features=TRUE`). In addition, the table also contains other regression data such as `RSQ`, `intercept` and `slope`. To perform basic trend analysis the `RSQ` (i.e. R squared) can be used:

```
fGroupsTab <- as.data.table(fGroups, areas = TRUE, features = FALSE, regression = TRUE)
# subset fGroups with reasonable correlation
increasingFGroups <- fGroups[, fGroupsTab[RSQ >= 0.8, group]]
```

8.10 Predicting toxicities and concentrations (MS2Tox and MS2Quant integration)

The MS2Tox and MS2Quant `R` packages predict toxicities and feature concentrations using a machine learning approach. The predictions are performed with either SMILES data or fingerprints calculated from MS/MS data with `SIRIUS+CSI:FingerID`. While using SMILES data is generally more accurate, using MS/MS fingerprints is generally faster and may be more suitable for features without know or suspected structure.

In `patRoon` the predictions are done in two steps:

1. The LC50 values (toxicity prediction) or response factors (concentration prediction) are calculated for given SMILES or MS/MS fingerprint data using `MS2Tox`/`MS2Quant`. This step is performed by the `predictTox()`/`predictConcs()` method function.
2. The predicted LC50 values are used to assign toxicities/concentrations to feature data. This is performed by the `calculateTox()`/`calculateConcs()` method function.

Various workflow data can be used to perform the predictions for step 1:

a. Suspect hits that were obtained with `screenSuspects` (see suspect screening).
b. Formula annotations obtained with `SIRIUS+CSI:FingerID`.
c. Compound annotations obtained with `SIRIUS+CSI:FingerID`.
d. Compound annotations obtained with other algorithms, e.g. `MetFrag`.

For *a* and *d* SMILES is used to perform the calculations, for *b* MS/MS fingerprints are used and for *c* either can be used.

> ***NOTE*** For option *b*, make sure that `getFingerprints=TRUE` and SIRIUS logins are handled when running `generateFormulas()` in order to obtain fingerprints.

8.10.1 Predicting toxicities

Some example workflows are shown below:

```
# Calculate toxicity for suspect hits.
fGroupsSuspTox <- predictTox(fGroupsSusp)
fGroupsSuspTox <- calculateTox(fGroupsSuspTox)

# Calculate toxicity for compound hits. Limit to the top 5 to reduce calculation time.
compoundsTop5 <- filter(compounds, topMost = 5)
compoundsTox <- predictTox(compoundsTop5)
fGroupsTox <- calculateTox(fGroups, compoundsTox)
```

It is also possible to calculate toxicities from multiple workflow objects:

```r
fGroupsSuspTox <- predictTox(fGroupsSusp) # as above

# Predict toxicities from compound candidates, using both SMILES and MS/MS fingerprints
# compoundsSuspSIR is an object produced with generateCompounds() with algorithm="sirius"
compoundsSuspSIRTox <- predictTox(compoundsSuspSIR, type = "both")

# Assign toxicities to feature groups from both suspect hits and SIRIUS annotations
fGroupsSuspTox <- calculateTox(fGroupsSuspTox, compoundsSuspSIRTox)
```

More details are in the reference manual: `?`pred-tox``.

8.10.2 Predicting concentrations

The workflow to predict concentrations is quite similar to predicting toxicities. However, before we can start we first have to specify the *calibrants* and the LC gradient elution program.

The calibrant data is used by `MS2Quant` to convert predicted ionization efficiencies to actual response factors, which are specific to the used LC instrument and methodology. For this purpose, several mixtures with known concentrations (i.e. standards) should be measured alongside your samples. The calibrants should be specified as a `data.frame`, for instance:

name	SMILES	intensity	conc	rt
Atrazine	CCNc1nc(nc(n1)Cl)NC(C)C	32708	1	336.6
Atrazine	CCNc1nc(nc(n1)Cl)NC(C)C	66880	2	336.6
Atrazine	CCNc1nc(nc(n1)Cl)NC(C)C	174087	5	336.6
Atrazine	CCNc1nc(nc(n1)Cl)NC(C)C	371192	10	336.6
Atrazine	CCNc1nc(nc(n1)Cl)NC(C)C	806749	25	336.6
Atrazine	CCNc1nc(nc(n1)Cl)NC(C)C	1852591	50	336.6
Carbamazepine	c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O	25231	1	349.2
Carbamazepine	c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O	47831	2	349.2
Carbamazepine	c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O	118843	5	349.2
Carbamazepine	c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O	211395	10	349.2
Carbamazepine	c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O	545192	25	349.2
Carbamazepine	c1ccc2c(c1)C=Cc3ccccc3N2C(=N)O	1083568	50	349.2
DEET	CCN(CC)C(=O)c1cccc(c1)C	45061	1	355.8
DEET	CCN(CC)C(=O)c1cccc(c1)C	84859	2	355.8
DEET	CCN(CC)C(=O)c1cccc(c1)C	228902	5	355.8
DEET	CCN(CC)C(=O)c1cccc(c1)C	434161	10	355.8
DEET	CCN(CC)C(=O)c1cccc(c1)C	1133166	25	355.8
DEET	CCN(CC)C(=O)c1cccc(c1)C	2385472	50	355.8
Venlafaxine	CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O	41465	1	324.0
Venlafaxine	CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O	89684	2	324.0
Venlafaxine	CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O	230890	5	324.0
Venlafaxine	CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O	400385	10	324.0
Venlafaxine	CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O	1094329	25	324.0
Venlafaxine	CN(C)CC(C1=CC=C(C=C1)OC)C2(CCCCC2)O	1965139	50	324.0

The intensity column should contain either the peak intensity (height) or area. Note that some feature detection algorithms can sometimes produce inaccurate peak areas, and the area determination methodology is often different among algorithms. For this reason, using peak intensities may be more reliable, however, it is worth testing this with your data.

It is also possible to use the `getQuantCalibFromScreening()` function to automatically create the calibrant table from feature group data:

```
calibList <- data.frame(...) # this should be a suspect list with your calibrants
fGroups <- screenSuspects(fGroups, calibList) # screen for the calibrants
concs <- data.frame(...) # concentration data for each calibrant compound, see below
calibrants <- getQuantCalibFromScreening(fGroups, concs)
calibrants <- getQuantCalibFromScreening(fGroups, concs, areas = TRUE) # obtain feature
↪  areas instead of intensities
```

The first step is to perform a screening for the calibrant compounds. Please ensure that this list should contains SMILES data, and to ensure correct feature assignment it is highly recommended to include retention times. The second requirement for `getQuantCalibFromScreening()` is a table with concentrations for each calibrant compound, e.g.:

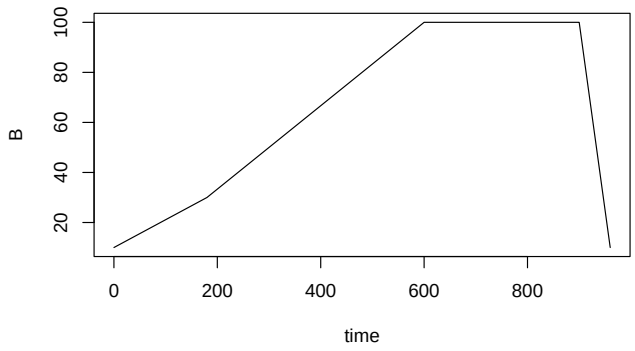
```
concs <- data.frame(
    name = c("DEET", "1h-benzotriazole", "Caffeine", "Atrazine", "Carbamazepine",
    ↪  "Venlafaxine"),
    standard_1 = c(1.00, 1.05, 1.10, 0.99, 1.01, 1.12),
    standard_2 = c(2.00, 2.15, 2.20, 1.98, 2.02, 1.82),
    standard_5 = c(5.01, 5.05, 5.22, 5.00, 4.88, 4.65),
    standard_10 = c(10.2, 10.11, 10.23, 11.77, 11.75, 12.13),
    standard_25 = c(25.3, 25.12, 25.34, 24.89, 24.78, 24.68),
    standard_50 = c(50.34, 50.05, 50.10, 49.97, 49.71, 50.52)
)
concs
```

```
#>              name standard_1 standard_2 standard_5 standard_10 standard_25 standard_50
#> 1            DEET       1.00       2.00       5.01       10.20       25.30       50.34
#> 2 1h-benzotriazole       1.05       2.15       5.05       10.11       25.12       50.05
#> 3         Caffeine       1.10       2.20       5.22       10.23       25.34       50.10
#> 4         Atrazine       0.99       1.98       5.00       11.77       24.89       49.97
#> 5    Carbamazepine       1.01       2.02       4.88       11.75       24.78       49.71
#> 6      Venlafaxine       1.12       1.82       4.65       12.13       24.68       50.52
```

The concentrations are specified separately for each calibrant compound. The column names should follow the names of the replicate groups assigned to the standards. The concentration unit is µg/l by default. The next section describes how this can be changed.

The gradient elution program is also specified by a `data.frame`, which for every time point (in seconds!) describes the percentage of 'B'. In this case, 'B' represents the total amount of organic modifier.

```
eluent <- data.frame(
    time = c(0, 180, 600, 900, 960),
    B = c(10, 30, 100, 100, 10)
)
plot(eluent, type = "l")
```




```
eluent
```

```
#>   time   B
#> 1    0  10
#> 2  180  30
#> 3  600 100
#> 4  900 100
#> 5  960  10
```

Then, the workflow to predict concentrations is very similar then predicting toxicities (previous section):

```
# Calculate concentrations for suspect hits.
fGroupsSuspConc <- predictRespFactors(
    fGroupsSusp,
    calibrants = calibrants, eluent = eluent,
    organicModifier = "MeCN", # organic modifier: MeOH or MeCN
    pHAq = 4 # pH of the aqueous part of the mobile phase
)
# set areas to TRUE if the calibrant table contains areas
fGroupsSuspConc <- calculateConcs(fGroupsSuspConc, areas = FALSE)
```

As was shown for toxicities it is possible to use different data sources (e.g. compound annotations, suspects) for predictions.

More details are in the reference manual: `?`pred-quant``.

8.10.3 Toxicity and concentration units

The default unit for toxicity and concentration data is µg/l. However, this can be configured when calling the `predictTox()`/`predictRespFactors()` functions:

```
fGroupsSuspTox <- predictTox(fGroupsSusp) # default unit: ug/l
fGroupsSuspTox <- predictTox(fGroupsSusp, concUnit = "ug/l") # same as above
fGroupsSuspTox <- predictTox(fGroupsSusp, concUnit = "mM") # millimolar
fGroupsSuspTox <- predictTox(fGroupsSusp, concUnit = "log mM") # unit used by MS2Tox

# calculated concentrations are ng/l, calibrants are specified in ug/l
# (by default calibConcUnit=concUnit)
fGroupsSuspConc <- predictRespFactors(
    fGroupsSusp, calibrants = calibrants, eluent = eluent,
```

```
    organicModifier = "MeCN", pHAq = 4,
    concUnit = "ng/l", calibConcUnit = "ug/l"
)
```

See the reference manuals (?`pred-tox`/?`pred-quant`) For more details on which units can be specified

8.10.4 Inspecting predicted values

The raw toxicity and concentration data assigned to feature groups can be retrieved with the `toxicities()` and `concentrations()` method functions, respectively.

```
toxicities(fGroupsSuspTox)
```

```
#>           group    type        candidate     candidate_name      LC50
#>          <char>  <char>           <char>             <char>     <num>
#> 1: M120_R268_30 suspect [nH]1nnc2ccccc12   1H-benzotriazole  59755.77
#> 2: M137_R249_53 suspect  NC(=O)Nc1ccccc1      N-Phenyl urea 101025.89
#> 3: M146_R225_70 suspect  Oc1ccc2ccccc2n1 2-Hydroxyquinoline  48833.62
#> 4: M146_R248_69 suspect  Oc1ccc2ccccc2n1 2-Hydroxyquinoline  48833.62
#> 5: M146_R309_68 suspect  Oc1ccc2ccccc2n1 2-Hydroxyquinoline  48833.62
```

```
concentrations(fGroupsSuspConc)
```

```
#>           group    type        candidate     candidate_name standard-pos-1 standard-pos-2 standard-po
#>          <char>  <char>           <char>             <char>          <num>          <num>          <n
#> 1: M120_R268_30 suspect [nH]1nnc2ccccc12   1H-benzotriazole      43.070773      39.905306       35.2
#> 2: M137_R249_53 suspect  NC(=O)Nc1ccccc1      N-Phenyl urea      18.485430      19.864756       17.3
#> 3: M146_R225_70 suspect  Oc1ccc2ccccc2n1 2-Hydroxyquinoline      15.700200      17.662215       18.4
#> 4: M146_R248_69 suspect  Oc1ccc2ccccc2n1 2-Hydroxyquinoline      19.030263      20.207821       19.5
#> 5: M146_R309_68 suspect  Oc1ccc2ccccc2n1 2-Hydroxyquinoline       7.978394       8.646156        8.6
```

If there were multiple candidates for a single feature group then these are split over the table rows:

```
toxicities(fGroupsTox)
```

```
#>            group     type                      candidate                          candidate_name
#>           <char>   <char>                         <char>                                  <char>
#>  1:  M120_R268_30 compound             C1=CC2=C(C=NN2)N=C1             1H-pyrazolo[4,3-b]pyridine
#>  2:  M120_R268_30 compound             C1=CC2=C(N=C1)N=CN2              1H-imidazo[4,5-b]pyridine
#>  3:  M120_R268_30 compound               C1=CC2=NNN=C2C=C1                        2H-benzotriazole
#>  4:  M120_R268_30 compound             C1=CN2C(=CC=N2)N=C1              pyrazolo[1,5-a]pyrimidine
#>  5:  M120_R268_30 compound               C1=CNC2=CN=CN=C21             5H-pyrrolo[3,2-d]pyrimidine
#> ---
#> 16: M192_R355_191 compound       CCN(CC)C(=O)C1=CC=C(C=C1)C           N,N-diethyl-4-methylbenzamide
#> 17: M192_R355_191 compound       CCN(CC)C(=O)C1=CC=CC(=C1)C           N,N-diethyl-3-methylbenzamide
#> 18: M192_R355_191 compound         CCN(CC)C(=O)C1=CC=CC=C1C           N,N-diethyl-2-methylbenzamide
#> 19: M192_R355_191 compound         CCN(CC)C(=O)CC1=CC=CC=C1           N,N-diethyl-2-phenylacetamide
#> 20: M192_R355_191 compound C[C@H]1[C@@H](OCCN1C)C2=CC=CC=C2 (2S,3S)-3,4-dimethyl-2-phenylmorpholine
```

The `as.data.table()` method function, which was discussed previously, can be used to summarize toxicity and concentration values.

```
# NOTE: NA values are filtered and columns are subset for readability
as.data.table(fGroupsTox)[!is.na(LC50), c("group", "LC50", "LC50_types")]
```

```
#>            group      LC50 LC50_types
#>           <char>     <num>     <char>
#> 1:  M120_R268_30  91334.97   compound
#> 2:  M137_R249_53 149557.04   compound
#> 3:  M146_R309_68  39154.68   compound
#> 4: M192_R355_191 100534.41   compound
```

```
concCols <- c("group", paste0(analyses(fGroupsSuspConc), "_conc"), "conc_types")
as.data.table(fGroupsSuspConc)[!is.na(conc_types), concCols, with = FALSE]
```

```
#>           group standard-pos-1_conc standard-pos-2_conc standard-pos-3_conc conc_types
#>          <char>               <num>               <num>               <num>     <char>
#> 1: M120_R268_30           43.070773           39.905306            35.21956    suspect
#> 2: M137_R249_53           18.485430           19.864756            17.36268    suspect
#> 3: M146_R309_68            7.978394            8.646156             8.67157    suspect
#> 4: M146_R248_69           19.030263           20.207821            19.54181    suspect
#> 5: M146_R225_70           15.700200           17.662215            18.49330    suspect
```

The `as.data.table()` method function *aggregates* the data for a feature group in case multiple candidates were assigned to it. By default the values are mean averaged, but this be changed with the `toxAggrParams`/`concAggrParams` arguments, for instance:

```
# as above, but aggregate by taking maximum values
as.data.table(fGroupsTox, toxAggrParams = getDefPredAggrParams(max))[!is.na(LC50),
↪  c("group", "LC50", "LC50_types")]
```

```
#>            group      LC50 LC50_types
#>           <char>     <num>     <char>
#> 1:  M120_R268_30 125872.29   compound
#> 2:  M137_R249_53 314718.38   compound
#> 3:  M146_R309_68  49051.68   compound
#> 4: M192_R355_191 132385.09   compound
```

If the `as.data.table()` method is used on suspect screening results, and predictions were performed directly for suspect hits, then predicted values can be reported for individual suspect match instead of aggregating them per feature group:

```
# Reports predicted values for each suspect separately. If multiple suspects are assigned
↪  to a feature group,
# then each suspect match is split into a different row.
as.data.table(fGroupsSuspTox, collapseSuspects = NULL)
```

Finally, the reporting functionality can be used to overview all predicted values, both aggregated and raw.

8.10.5 Using predicted values to prioritize data

The `filter()` method function that was introduced before can also be used to filter data based on predicted toxicities, response factors and concentrations. For instance, this allows you to remove annotation candidates which are unlikely to be toxic or sensitive enough to be detected or any features with very low concentrations. Some examples are shown below.

```
# compoundsSuspSIRTox is an object with predicted toxicities (LC50 values) for each
↪  candidate
# we can use the common scoreLimits filter to select a range of allowed values (min/max)
compoundsSuspSIRToxF <- filter(compoundsSuspSIRTox, scoreLimits = list(LC50_SMILES = c(0,
↪  1E4)))

# for suspects with predicted toxicities/response factors there are dedicated filters
fGroupsSuspConcF <- filter(fGroupsSuspConc, minRF = 5E4) # remove suspect hits with
↪  response factor <5E4
fGroupsSuspToxF <- filter(ffGroupsSuspTox, maxLC50 = 100) # remove suspect hits with LC50
↪  values > 100

# similarly, for feature data there are dedicated filters.
# note that these aggregate data prior to filtering (see previous section)
fGroupsConcF <- filter(fGroupsConc, absMinConc = 0.02)
# only keep features with concentrations that are at least 1% of their toxicity
# note that both concentrations/toxicity values should have been calculated with
↪  calculateConcs()/calculateTox()
fGroupsConcToxF <- filter(fGroupsConcTox, absMinConcTox = 0.01)

# also get rid of features without concentrations (these are ignored by default)
fGroupsConcF <- filter(fGroupsConc, absMinConc = 0.02, removeNA = TRUE)
# like as.data.table we can configure how values are aggregated
# here the minimum is used instead of the default mean
fGroupsToxF <- filter(fGroupsTox, absMaxTox = 5E3, predAggrParams =
↪  getDefPredAggrParams(min))
```

More details are found in the reference manual (`?`feature-filtering``).

8.11 Fold changes

A specific statistical way to prioritize feature data is by Fold changes (FC). This is a relative simple method to quickly identify (significant) changes between two sample groups. A typical use case is to compare the feature intensities before and after an experiment.

To perform FC calculations we first need to specify its parameters. This is best achieved with the `getFCParams()` function:

```
getFCParams(c("before", "after"))
```

```
#> $rGroups
#> [1] "before" "after"
#>
#> $thresholdFC
#> [1] 0.25
#>
```

```
#> $thresholdPV
#> [1] 0.05
#>
#> $zeroMethod
#> [1] "add"
#>
#> $zeroValue
#> [1] 0.01
#>
#> $PVTestFunc
#> function (x, y)
#> t.test(x, y, paired = TRUE)$p.value
#> <bytecode: 0x562b450af8a8>
#> <environment: 0x562b2fbc0d98>
#>
#> $PVAdjFunc
#> function (pv)
#> p.adjust(pv, "BH")
#> <bytecode: 0x562b450afbb8>
#> <environment: 0x562b2fbc0d98>
```

In this example we generate a `list` with parameters in order to make a comparison between two replicate groups: `before` and `after`. Several advanced parameters are available to tweak the calculation process. These are explained in the reference manual (`?featureGroups`).

The `as.data.table` function for feature groups is used to perform the FC calculations.

```
myFCParams <- getFCParams(c("solvent-pos", "standard-pos")) # compare solvent/standard
as.data.table(fGroups, FCParams = myFCParams)[, c("group", "FC", "FC_log", "PV",
↪  "PV_log", "classification")]
```

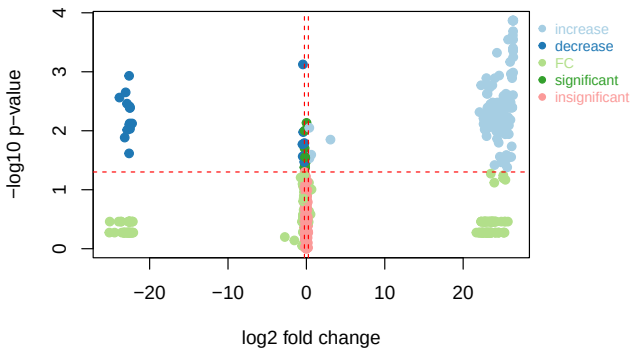
```
#>              group           FC      FC_log          PV     PV_log classification
#>             <char>        <num>       <num>       <num>      <num>         <char>
#>   1:     M99_R14_1 8.837494e-01 -0.17829070 0.223506802 0.65070926  insignificant
#>   2:      M99_R4_2 8.500464e-01 -0.23438649 0.778488444 0.10874783  insignificant
#>   3:     M100_R7_3 8.009186e-01 -0.32027248 0.804751489 0.09433821             FC
#>   4:     M100_R5_4 4.140000e+06 21.98119934 0.533213018 0.27309926             FC
#>   5:    M100_R28_5 9.594972e-01 -0.05964952 0.975712373 0.01067819  insignificant
#>  ---
#> 676: M425_R319_676 2.149907e+07 24.35777069 0.009681742 2.01404652       increase
#> 677:  M427_R10_677 1.059937e+00  0.08397893 0.371260940 0.43032074  insignificant
#> 678: M427_R319_678 7.776800e+06 22.89074521 0.533213018 0.27309926             FC
#> 679: M432_R383_679 9.816400e+06 23.22676261 0.347009089 0.45965915             FC
#> 680:  M433_R10_680 1.132909e+00  0.18003240 0.293217996 0.53280938  insignificant
```

The `classification` column allows you to easily identify if and how a feature changes between the two sample groups. This can also be used to prioritize feature groups:

```
tab <- as.data.table(fGroups, FCParams = myFCParams)
# only keep feature groups that significantly increase or decrease
fGroupsChanged <- fGroups[, tab[classification %in% c("increase", "decrease")]$group]
```

The `plotVolcano` function can be used to visually the FC data:

```
plotVolcano(fGroups, myFCParams)
```



8.12 Caching

In `patRoon` lengthy processing operations such as finding features and generating annotation data is *cached*. This means that when you run such a calculation again (without changing any parameters), the data is simply loaded from the cache data instead of re-generating it. This in turn is very useful, for instance, if you have closed your R session and want to continue with data processing at a later stage.

The cache data is stored in a sqlite database file. This file is stored by default under the name `cache.sqlite` in the current working directory (for this reason it is very important to always restore your working directory!). However, the name and location can be changed by setting a global package option:

```
options(patRoon.cache.fileName = "~/myCacheFile.sqlite")
```

For instance, this might be useful if you want to use a shared cache file between projects.

After a while you may see that your cache file can get quite large. This is especially true when testing different parameters to optimize your workflow. Furthermore, you may want to clear the cache after you have updated `patRoon` and want to make sure that the latest code is used to generate the data. At any point you can simply remove the cache file. A more fine tuned approach which doesn't wipe all your cached data is by using the `clearCache()` function. With this function you can selectively remove parts of the cache file. The function has two arguments: `what`, which specifies what should be removed, and `file` which specifies the path to the cache file. The latter only needs to be specified if you want to manage a different cache file.

In order to figure what is in the cache you can run `clearCache()` without any arguments:

```
clearCache()
```

```
#> Please specify which cache you want to remove. Available are:
#> - EICData (3 rows)
#> - LC50_SMILES (23 rows)
#> - MS2QMD (1 rows)
#> - MSLibraryJSON (1 rows)
#> - MSLibraryMSP (1 rows)
#> - MSPeakListsAvg (4 rows)
#> - MSPeakListsMzR (97 rows)
#> - MSPeakListsSetAvg (2 rows)
#> - RF_SMILES (5 rows)
#> - TPsLib (1 rows)
```

```
#> - annotateSuspects (1 rows)
#> - calculatePeakQualities (3 rows)
#> - componentsCAMERA (1 rows)
#> - componentsNontarget (1 rows)
#> - componentsTPs (1 rows)
#> - compoundsCluster (1 rows)
#> - compoundsMetFrag (30 rows)
#> - dataCentroided (12 rows)
#> - featureGroupsOpenMS (6 rows)
#> - featuresOpenMS (69 rows)
#> - filterFGroups_blank (4 rows)
#> - filterFGroups_intensity (11 rows)
#> - filterFGroups_minAnalyses (1 rows)
#> - filterFGroups_minReplicates (83 rows)
#> - filterFGroups_replicateAbundance (8 rows)
#> - filterFGroups_replicate_group (11 rows)
#> - filterFGroups_retention (3 rows)
#> - filterMSPeakLists (4 rows)
#> - formulasFGroupConsensus (2 rows)
#> - formulasGenForm (89 rows)
#> - formulasSIRIUS (5 rows)
#> - generateTPsBT (74 rows)
#> - loadIntensities (69 rows)
#> - mzREIC (3426 rows)
#> - reportHTMLCompounds (1 rows)
#> - reportHTMLFormulas (1 rows)
#> - screenSuspects (7 rows)
#> - screenSuspectsPrepList (8 rows)
#> - specData (12 rows)
#> - all (removes complete cache database)
```

Using this output you can re-run the function again, for instance:

```
clearCache("featuresOpenMS")
clearCache(c("featureGroupsOpenMS", "formulasGenForm")) # clear multiple
clearCache("OpenMS") # clear all with OpenMS in name (ie partial matched)
clearCache("all") # same as simply removing the file
```

8.13 Parallelization

Some steps in the non-target screening workflow are inherently computationally intensive. To reduce computational times `patRoon` is able to perform *parallelization* for most of the important functionality. This is especially useful if you have a modern system with multiple CPU cores and sufficient RAM.

For various technical reasons several parallelization techniques are used, these can be categorized as *parallelization of `R` functions* and *multiprocessing*. The next sections describe both parallelization approaches in order to let you optimize the workflow.

8.13.1 Parellization of R functions

Several functions of `patRoon` support parallelization.

Function	Purpose	Remarks
`findFeatures`	Obtain feature data	Only `envipick` and `kpic2` algorithms.
`generateComponents`	Generate components	Only `cliquems` algorithm.
`report`	Reporting data	
`generateTPs`	Obtain transformation products	Only `cts` algorithm.
`optimizeFeatureFinding`, `optimizeFeatureGrouping`	Optimize feature finding/grouping parameters	Discussed here.
`calculatePeakQualities`	Calculate feature (group) qualities	Discussed here.
`predictTox` / `predictRespFactors`	Prediction of toxicities/concentrations }	Only `compounds` methods. Discussed here.

The parallelization is achieved with the future and future.apply `R` packages. To enable parallelization of these functions the `parallel` argument must be set to `TRUE` and the future framework must be properly configured in advance. For example:

```
# setup three workers to run in parallel
future::plan("multisession", workers = 3)

# find features with enviPick in parallel
fList <- findFeatures(anaInfo, "envipick", parallel = TRUE)
```

It is important to properly configure the right future plan. Please see the documentation of the future package for more details.

8.13.2 Multiprocessing

`patRoon` relies on several external (command-line) tools to generate workflow data. These commands may be executed in *parallel* to reduce computational times ('multiprocessing'). The table below outlines the tools that are executed in parallel.

Tool	Used by	Notes
`msConvert`	`convertMSFiles(algorithm="pwiz", ...)`	
`FileConverter`	`convertMSFiles(algorithm="openms", ...)`	
`FeatureFinderMetabo`	`findFeatures(algorithm="openms", ...)`	
`julia`	`findFeatures(algorithm="safd", ...)`	
`SIRIUS`	`findFeatures(algorithm="sirius", ...)`	
`MetaboliteAdductDecharger`	`generateComponents(algorithm="openms", ...)`	
`GenForm`	`generateFormulas(agorithm="genform", ...)`	
`SIRIUS`	`generateFormulas(agorithm="sirius", ...)`, `generateCompounds(agorithm="sirius", ...)`	Only if `splitBatches=TRUE`
`MetFrag`	`generateCompounds(agorithm="metfrag", ...)`	
`pngquant`	`reportHTML(...)`	Only if `optimizePng=TRUE`

Tool	Used by	Notes
BioTransformer	generateTPs(algorithm = "biotransformer")	Disabled by default (see ?generateTPs for details).

Multiprocessing is either performed by executing processes in the background with the processx R package (*classic interface*) or by futures, which were introduced in the previous section. An overview of the characteristics of both parallelization techniques is shown below.

classic	future
requires little or no configuration	configuration needed to setup
works with all tools	doesn't work with pngquant and slower with GenForm
only supports parallelization on the local computer	allows both local and cluster computing

Which method is used is controlled by the patRoon.MP.method package option. Note that reportHTML() will always use the classic method for pngquant.

8.13.2.1 Classic multiprocessing interface

The classic interface is the 'original' method implemented in patRoon, and is therefore well tested and optimized. It is easier to setup, works well with all tools, and is therefore the default method. It is enabled as follows:

```
options(patRoon.MP.method = "classic")
```

The number of parallel processes is configured through the patRoon.MP.maxProcs option. By default it is set to the number of available CPU cores, which results usually in the best performance. However, you may want to lower this, for instance, to keep your computer more responsive while processing or limit the RAM used by the data processing workflow.

```
options(patRoon.MP.maxProcs = 2) # do not execute more than two tools in parallel.
```

This will change the parallelization for the complete workflow. However, it may be desirable to change this for only a part the workflow. This is easily achieved with the withOpt() function.

```
# do not execute more than two tools in parallel.
options(patRoon.MP.maxProcs = 2)

# ... but execute up to four GenForm processes
withOpt(MP.maxProcs = 4, {
    formulas <- generateFormulas(fGroups, "genform", ...)
})
```

The withOpt function will temporarily change the given option(s) while executing a given code block and restore it afterwards (it is very similar to the with_options() function from the withr R package). Furthermore, notice how withOpt() does not require you to prefix the option names with patRoon..

8.13.2.2 Multiprocessing with futures

The primary goal of the "future" method is to allow parallel processing on one or more external computers. Since it uses the future `R` package, many approaches are supported, such as local parallelization (similar to the `classic` method), cluster computing via multiple networked computers and more advanced HPC approaches such as `slurm` via the future.batchtools `R` package. This parallelization method can be activated as follows:

```
options(patRoon.MP.method = "future")

# set a future plan

# example 1: start a local cluster with four nodes
future::plan("cluster", workers = 4)

# example 2: start a networked cluster with four nodes on PC with hostname "otherpc"
future::plan("cluster", workers = rep("otherpc", 4))
```

Please see the documentation of the respective packages (*e.g.* future and future.batchtools) for more details on how to configure the workers.

The `withOpt()` function introduced in the previous subsection can also be used to temporarily switch between parallelization approaches, for instance:

```
# default to future parallelization
options(patRoon.MP.method = "future")
future::plan("cluster", workers = 4)

# ... do workflow

# do classic parallelization for GenForm
withOpt(MP.method = "classic", {
    formulas <- generateFormulas(fGroups, "genform", ...)
})

# .. do more workflow
```

8.13.2.3 Logging

Most tools that are executed in parallel will log their output to text files. These files may contain valuable information, for instance, when an error occurred. By default, the logfiles are stored in the `log` directory placed in the current working directory. However, you can change this location by setting the `patRoon.MP.logPath` option. If you set this option to `FALSE` then no logging occurs.

8.13.3 Notes when using parallelization with futures

Some important notes when using the `future` parallelization method:

- `GenForm` currently performs less optimal with future multiprocessing to the `classic` approach. Nevertheless, it may still be interesting to use the `future` method to move the computations to another system to free up resources on your local system.
- Behind the scenes the future.apply package is used to schedule the tools to be executed. The `patRoon.MP.futureSched` option sets the value for the `future.scheduling` argument to the `future_lapply()` function, and therefore allows you to tweak the scheduling.
- Make sure that `patRoon` is present and with the same version on all computing hosts.

- Make sure that any external dependencies used by multiprocessing, such as `MetFrag` and `SIRIUS`, and local compound databases, such as as `PubChemLite`, are also with the same version and are configured properly. See the Installation section for more details.
- If you encounter errors then it may be handy to switch to `future::plan("sequential")` and see if it works or you get more descriptive error messages.
- In order to restart the nodes, for instance after re-configuring `patRoon`, updating `R` packages etc, simply re-execute `future::plan(...)`.
- Setting the `future.debug` package option to `TRUE` may give you more insight what is happening to find problems.

9 References

Chetnik, Kelsey, Lauren Petrick, and Gaurav Pandey. 2020. "MetaClean: A Machine Learning-Based Classifier for Reduced False Positive Peak Detection in Untargeted LC-MS Metabolomics Data." *Metabolomics* 16 (11). https://doi.org/10.1007/s11306-020-01738-3.

Schollee, Jennifer E., Emma L. Schymanski, Sven E. Avak, Martin Loos, and Juliane Hollender. 2015. "Prioritizing Unknown Transformation Products from Biologically-Treated Wastewater Using High-Resolution Mass Spectrometry, Multivariate Statistics, and Metabolic Logic." *Analytical Chemistry* 87 (24): 12121–29. https://doi.org/10.1021/acs.analchem.5b02905.

Schymanski, Emma L., Junho Jeon, Rebekka Gulde, Kathrin Fenner, Matthias Ruff, Heinz P. Singer, and Juliane Hollender. 2014. "Identifying Small Molecules via High Resolution Mass Spectrometry: Communicating Confidence." *Environmental Science and Technology* 48 (4): 2097–98. https://doi.org/10.1021/es5002105.