Cardinal: Analytic tools for mass spectrometry imaging

Kyle D. Bemis

November 23, 2014

Contents

1 Introduction

The *R* package *Cardinal* has been created to fill the need for an efficent, open-source tool for the analysis of imaging data–specifically mass spectrometry imaging data. Cardinal is built upon data structures which follow Bioconductor (http://www.bioconductor.org/) standards for data classes in order to provide an additional level of convenience and familiarity to those who may be used to performing bioinformatic analyses in *R*.

Analysis in imaging data includes many things, such as visualization, pre-processing, and multivariate statistical techniques. Both supervised and unsupervised statistical methods are supported in *Cardinal*, including image segmentation (clustering), principle components analysis, and classification techniques such as partial least Squares discriminant analysis.

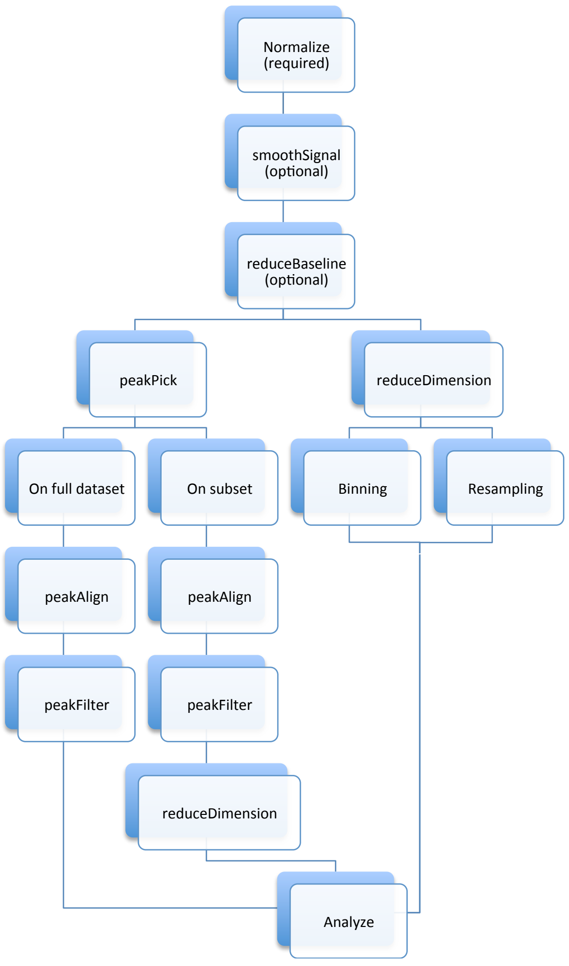


Figure 1: Preprocessing steps

Figure 1 charts out the workflow for mass spectrometry imaging data analysis.

This is a brief walkthrough of some of the basic functionality of*Cardinal*. For a more detailed view of the functionality of a given method, see the *R* help file.

2 Input/Output

2.1 Input

In order to be analyzed in *Cardinal*, input data must be in either Analyze 7.5 and imzML format. These are two of the most common data exchange formats in imaging mass spectrometry.

2.1.1 Analyze 7.5

Originally designed for MRI data by the Mayo Clinic, Analyze 7.5 is a common format used for exchange of mass spectrometry imaging data.

The Analyze format uses a collection of three files with extensions '.hdr', '.img', and '.t2m' to store data. To read datasets stored in the Analyze format, use the `readAnalyze` function. All three files must be present in the same folder and have the same name (except for the file extension) for the data to be read properly.

```
> name <- "This is the common name of your .hdr, .img, and .t2m files"
> folder <- "/This/is/the/path/to/the/folder/containing/the/files"
> data <- readAnalyze(name, folder)
```

For more information on reading Analyze files, type `?readAnalyze`.

2.1.2 imzML

The open XML-based format imzML is a more recently developed format specifically designed for interchange of mass spectrometry imaging datasets [1]. Many other formats can be converted to imzML with the help of free applications available online.

The imzML format uses two files with extensions '.imzML' and '.ibd' to store data. To read datasets stored in the imzML format, use the `readImzML` function. Both files must be present in the same folder and have the same name (again, except for the file extension) for the data to be read properly.

```
> name <- "This is the common name of your .imzML and .ibd files"
> folder <- "/This/is/the/path/to/the/folder/containing/the/files"
> data <- readImzML(name, folder)
```

For more information on reading imzML files, type `?readImzML`.

2.1.3 readMSIData

Cardinal also provides the convenience function `readMSIData`, which can automatically recognize the whether the data format is Analyze or imzML based on file extensions. The same rules for naming conventions apply as described above, but one need only provide the path to any of the data files. For example, to read an Analyze file, providing the path to any of the '.hdr', '.img', or '.t2m' will work. Likewise, providing the path to either the '.imzML' or '.ibd' file will work for reading data stored in the imzML format.

```
> file <- "/This/is/the/path/to/an/imaging/data/file.extension"
> data <- readMSIData(file)
```

2.2 Output

2.2.1 RData files

Any *R* object, including those created by *Cardinal*, can be saved as an **RData** file using the `save` and loaded using the `load` function.

```
> save(data, file="/Where/to/save/the/data.RData")
> load("/Where/to/save/the/data.RData")
```

When an **RData** file is loaded, the saved object appears in the global environment for the *R* session and is available for access by name, just as it was in the session during which it was saved. This functionality is part of *R*; see `?save` and `?load` for more details.

3 Data exploration and visualization

Mass spectrometry imaging datasets in *Cardinal* are stored in MSImageSet objects. This allows *Cardinal* to keep track of the spectra, pixel coordinates, m/z values and more in one place for the dataset. The MSImageSet object is described in more detail below. There are many methods for both creating and manipulating MSImageSet objects in *Cardinal*. We now describe some of these methods.

3.1 An example dataset

To illustrate methods for the MSImageSet objects, we begin by creating a simple, simulated dataset using the *Cardinal* function generateImage. This dataset will be the running example for this section. For more details on simulating mass spectrometry images, see ?generateImage or Section 6.2 *Simulation*.

```
> set.seed(1)
> pattern <- factor(c(0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 2, 2, 0,
+         0, 0, 0, 0, 0, 0, 1, 2, 2, 0, 0, 0, 0, 0, 2, 1, 1, 2,
+         2, 0, 0, 0, 0, 0, 1, 2, 2, 2, 2, 0, 0, 0, 0, 1, 2, 2,
+         2, 2, 2, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 2,
+         2, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0),
+         levels=c(0,1,2), labels=c("blue", "black", "red"))
> msset <- generateImage(pattern, coord=expand.grid(x=1:9, y=1:9),
+         range=c(1000, 5000), centers=c(2000, 3000, 4000),
+         resolution=100, step=3.3, as="MSImageSet")
```

The above code creates a simulated MS imaging dataset called msset, which is 9×9 pixels, with a mass range from m/z 1000 to m/z 5000. There are three peaks, occuring at m/z 2000, m/z 3000, and m/z 4000. Each of these peaks corresponds to a distinct region of interest. These are saved in the *factor* pattern. A *factor* is the standard way of storing categorical variables in *R*. All pixels with pattern $= 0$ correspond to the region with peak at m/z 2000, pattern $= 1$ corresponds to the peak at m/z 3000, and pattern $= 2$ corresponds to m/z 4000.

We'll label these regions of interest "blue" pixels, "black" pixels, and "red" pixels, respectively.

3.2 The MSImageSet object

Most important aspects of a mass spectrometry imaging dataset stored in an MSImageSet object can be accessed by simple methods.

For example, m/z-values are accessed by the method mz, pixel coordinates are accessed by the method coord, and the mass spectra themselves are accessed by the method spectra. The mass spectra are stored as a matrix with each column corresponding to the mass spectrum at a single pixel.

```
> head(mz(msset), n=10) # first 10 m/z values

 [1] 1000.0 1003.3 1006.6 1009.9 1013.2 1016.5 1019.8 1023.1 1026.4
[10] 1029.7

> head(coord(msset), n=10) # first 10 pixel coordinates

              x y
x = 1, y = 1 1 1
x = 2, y = 1 2 1
x = 3, y = 1 3 1
x = 4, y = 1 4 1
x = 5, y = 1 5 1
x = 6, y = 1 6 1
x = 7, y = 1 7 1
x = 8, y = 1 8 1
x = 9, y = 1 9 1
x = 1, y = 2 1 2
```

```
> head(spectra(msset)[,1], n=10) # first 10 intensities in the first mass spectrum
```

```
 [1] 13.72308 11.93339 12.79512 12.50510 11.71788 13.92943 12.18719
 [8] 12.24365 12.62183 12.30359
```

The methods `nrow` and `ncol` can be used to retrieve the number of features and number of pixels in an object, respectively. The method `dim` gives both number of features and number of pixels, while `dims` gives number of features as well as spatial dimensions of the image.

```
> nrow(msset)
```

```
Features
    1213
```

```
> ncol(msset)
```

```
Pixels
    81
```

```
> dim(msset)
```

```
Features   Pixels
    1213       81
```

```
> dims(msset)
```

```
         iData
Features  1213
x            9
y            9
```

Two other helpful methods are `features` and `pixels`. These are useful for retrieving the feature number and pixel number (i.e., the row and column in the `spectra(msset)` matrix) corresponding to items of interest such as specific m/z-values or pixel coordinates.

```
> features(msset, mz=3000) # returns the feature number most closely matching m/z 3000
```

```
m/z = 2999.8
         607
```

```
> mz(msset)[607]
```

```
[1] 2999.8
```

```
> pixels(msset, coord=list(x=5, y=5)) # returns the pixel number for x = 5, y = 5
```

```
x = 5, y = 5
          41
```

```
> pixels(msset, x=5, y=5) # also returns the pixel number for x = 5, y = 5
```

```
x = 5, y = 5
          41
```

```
> coord(msset)[41,]
```

```
             x y
x = 5, y = 5 5 5
```

See `?MSImageSet` for more details and additional methods.

Technical note: `MSImageSet` is an S4 class. It inherits from the more general `SImageSet` class, which itself inherits from the `iSet` virtual class. The `iSet` virtual class is designed around the same design principles as the `eSet` class provided by Biobase. See the "Cardinal development" vignette for more information.)

3.3 Subsetting a `MSImageSet`

A `MSImageSet` can be subset by row and column like an ordinary *R* matrix or *data.frame*, where rows correspond to the features (m/z-values) and columns correspond to pixels (locations associated with mass spectra). Subsetting will return a new `MSImageSet`.

For example, we can subset by m/z-values so that we only keep the mass range from m/z 2500 to m/z 4500.

```
> tmp <- msset[2500 < mz(msset) & mz(msset) < 4500,]
> range(mz(msset))
```

```
[1] 1000.0 4999.6
```

```
> range(mz(tmp))
```

```
[1] 2501.5 4498.0
```

Alternatively, we can subset by pixel coordinates. To keep only pixels with x-coordinates greater than 5, we can do the following.

```
> tmp <- msset[,coord(msset)$x > 5]
> range(coord(msset)$x)
```

```
[1] 1 9
```

```
> range(coord(tmp)$x)
```

```
[1] 6 9
```

We can also subset in both ways at once.

```
> tmp <- msset[2500 < mz(msset) & mz(msset) < 4500, coord(msset)$x > 5]
> range(mz(tmp))
```

```
[1] 2501.5 4498.0
```

```
> range(coord(tmp)$x)
```

```
[1] 6 9
```

It is also possible to manually select a region of interest and use it to subset the dataset. This is done using the `select` method, which will be introduced in Section 3.5 *Plotting ion images*.

3.4 Plotting mass spectra

Mass spectra from an `MSImageSet` can be displayed using the `plot` method. To plot the mass spectrum at the first pixel of our `MSImageSet`, we do the following:

```
> plot(msset, pixel=1)
```

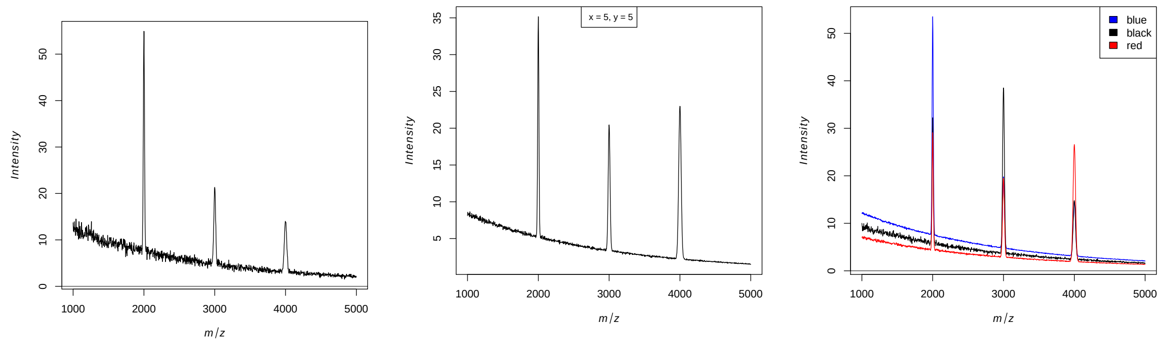
The result of which is shown in Figure 2a.

Instead of pixel number, we can specify a set of coordinates corresponding to the mass spectrum we want to plot. The following produces Figure 2b, which is the mean spectrum for the pixel at spatial location $(5, 5)$, and all other spectra within a 2 pixel neighborhood of that location.

```
> plot(msset, coord=list(x=5, y=5), plusminus=2)
```

Finally, we can plot multiple spectra at once, as shown in Figure 2c. This is done below by specifying a vector for the `pixel` argument. The plots are displayed simultaneously by setting `superpose = TRUE` and `key = TRUE` generates a legend. The `pixel.groups` here indicates that the pixels should be grouped by their classifications as encoded by the `pattern` factor. By default, *Cardinal* averages over spectra in the same group.

```
> mycol <- c("blue", "black", "red")
> plot(msset, pixel=1:ncol(msset), pixel.groups=pattern, superpose=TRUE, key=TRUE, col=mycol)
```



(a) Pixel 1 mass spectrum (b) Mean spectrum over neighborhood of pixel $(5,5)$ (c) Simultaneous plot of 3 mass spectra

Figure 2: Plotting mass spectra.

3.5 Plotting ion images

Ion images from an `MSImageSet` can be plotted using the `image` method. To plot the single ion image for the first feature, Figure 3a, we use:

```
> image(msset, feature=1)
```

The mean ion image for the neighborhood of m/z 4000 with radius 10, i.e. m/z $[3990, 4010]$ is shown in Figure 3b.

```
> image(msset, mz=4000, plusminus=10)
```

In Figure 3c the single ion images for m/z 2000, m/z 3000, and m/z 4000 are displayed simultaneously.

```
> mycol <- c("blue", "black", "red")
> image(msset, mz=c(2000, 3000, 4000), col=mycol, superpose=TRUE)
```

The ion image for m/z 2000 is shown in Figure 3d, with a custom color scale from white to blue. The most intense "hotspots" are suppressed.

```
> mycol <- gradient.colors(100, start="white", end="blue")
> image(msset, mz=2000, col.regions=mycol, contrast.enhance="suppress")
```

In Figure 3e, a smoothed ion image for mz3000 with a custom color scale from white to black is presented.

```
> mycol <- gradient.colors(100, start="white", end="black")
> image(msset, mz=3000, col.regions=mycol, smooth.image="gaussian")
```

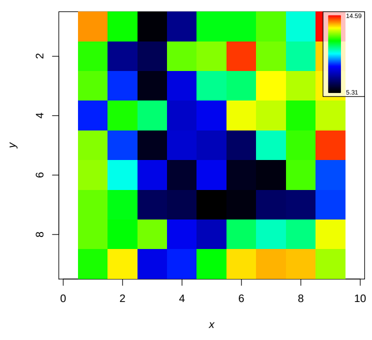
Finally, in Figure 3f, for only those pixels defined as being from the "black" and "red" regions, we plot the ion image of mz4000 with a custom color scale from black to red.

```
> msset2 <- msset[,pattern == "black" | pattern == "red"]
> mycol <- gradient.colors(100, start="black", end="red")
> image(msset2, mz=4000, col.regions=mycol)
```

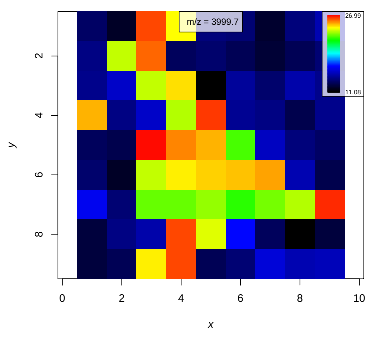
4 Pre-processing

4.1 Normalization

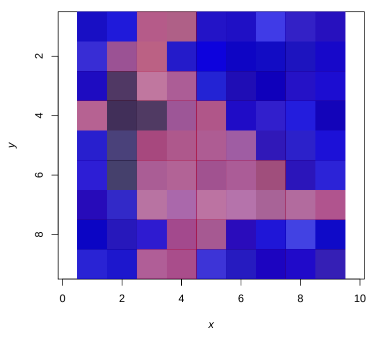
Normalization is perhaps the most important pre-processing step before any kind of analysis should be performed on biological datasets, and mass spectrometry imaging experiments are no different in this regard. *Cardinal* provides normalization to total ion current (TIC), commonly used in MSI analysis (see [2] for a discussion of this method). In the first command below, we only perform the normalization on the first pixel in order to show a plot of the processing results in Figure 4. In the second, we perform normalization on the whole dataset.



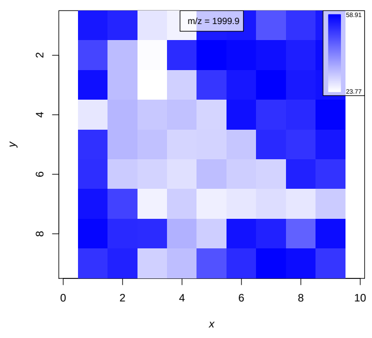
(a) Single m/z feature



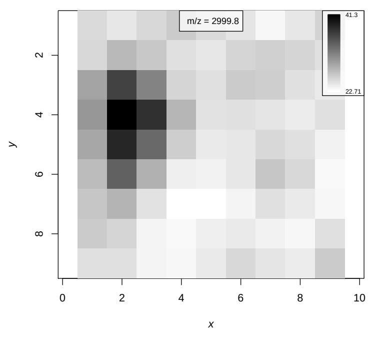
(b) Mean over m/z neighborhood



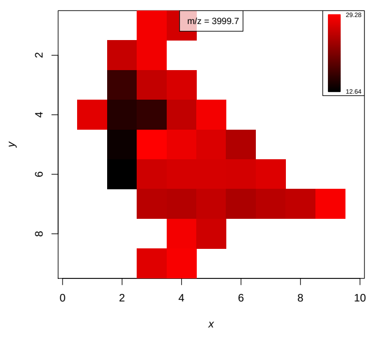
(c) Multiple displayed simultaneously



(d) Hotspot suppressed



(e) Gaussian smoothed



(f) Selected regions

Figure 3: Plotting ion images.

```
> temp <- normalize(msset, pixel=1, method="tic", plot=TRUE)
> msset2 <- normalize(msset, method="tic")
```

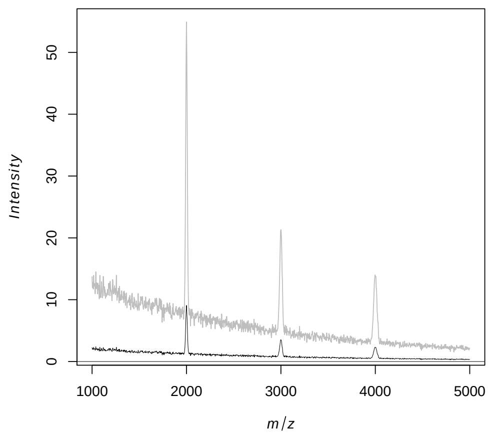


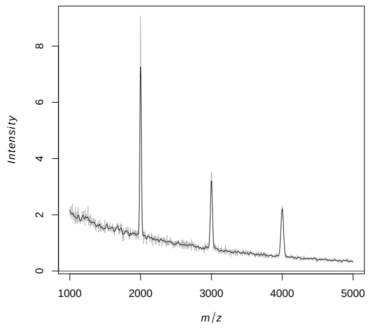
Figure 4: Total ion current (TIC) normalization.

4.2 Smoothing

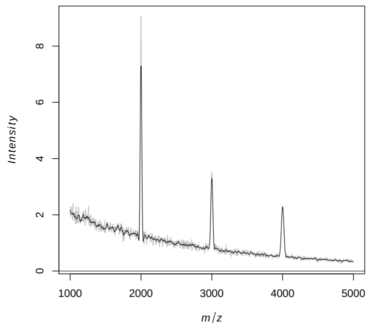
Smoothing the mass spectra is useful for reducing noise, which can improve detection of peaks. *Cardinal* provides several common methods for smoothing mass spectra, including Gaussian kernel smoothing (Figure 5a), Savitsky-

Golay smoothing (Figure 5b), and a simple moving average filter [3].

```
> temp <- smoothSignal(msset2, pixel=1, method="gaussian", window=9, plot=TRUE)
> temp <- smoothSignal(msset2, pixel=1, method="sgolay", window=15, plot=TRUE)
> msset3 <- smoothSignal(msset2, method="gaussian", window=9)
```



(a) Gaussian Kernel Smoothing



(b) Savitsky-Golay smoothing

Figure 5: Smoothing techniques.

4.3 Baseline reduction

Baseline reduction is often necessary for many datasets, especially those obtained through matrix-assisted methods such as MALDI ([3]). *Cardinal* implements a simple version that interpolates a baseline from local medians or local minima, while attempting to preserve the signal from mass spectral peaks. Figure 6 shows baseline reduction for a single pixel, where the green curve represents the estimated baseline and the baseline-reduced spectrum is plotted in black.

```
> temp <- reduceBaseline(msset3, pixel=1, method="median", blocks=50, plot=TRUE)
```

We can also reduce baseline across all pixels in the image.

```
> msset4 <- reduceBaseline(msset3, method="median", blocks=50)
```

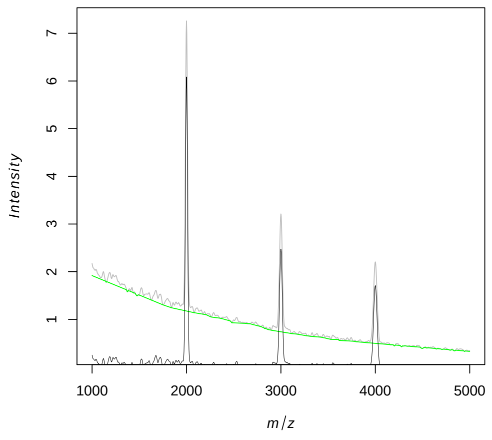
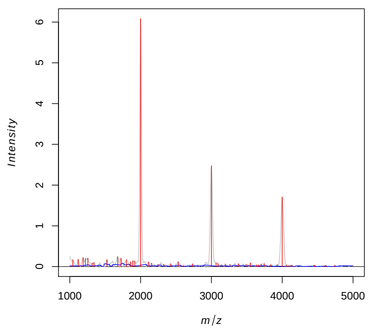


Figure 6: Baseline reduction using interpolation from medians.

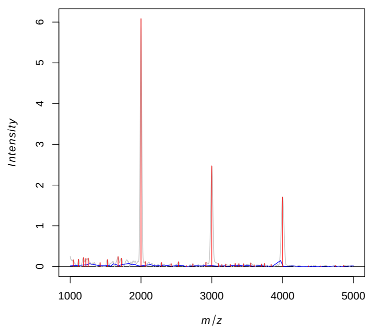
4.4 Peak picking

Peak picking is a common form of data reduction that reduces the signal to relevant data peaks. *Cardinal* implements three varieties based on a user-specified signal-to-noise ratio (SNR). The "simple" version interpolates a constant noise pattern, the "adaptive" version interpolates an adaptive noise pattern Figure 7a, and "limpic" implements the LIMPIC algorithm for peak detection Figure 7b.

```
> temp <- peakPick(msset4, pixel=1, method="adaptive", SNR=3, plot=TRUE)
> temp <- peakPick(msset4, pixel=1, method="limpic", SNR=3, plot=TRUE)
> msset5 <- peakPick(msset4, method="simple", SNR=3)
```



(a) Adaptive



(b) LIMPIC

Figure 7: Peak picking techniques.

4.5 Peak alignment

Peak alignment is necessary to account for possible inaccuracy in m/z measurements. Peaks can be aligned to a reference list of known m/z values, or to the local maxima in the mean spectrum. Figure 8 denotes the selected peaks by red vertical lines, and aligns the local maxima of the mean spectra to these peaks, as in [4].

```
> temp <- peakAlign(msset5, pixel=1, method="diff", plot=TRUE)
> msset6 <- peakAlign(msset5, method="diff")
```

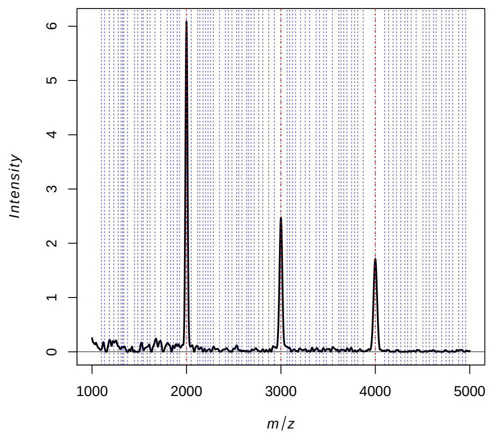


Figure 8: Peak alignment to the local maxima of the mean spectrum.

4.6 Peak filtering

Peak filtering removes peaks that occur infrequently, such as those which only occur in a small proportion of pixels. This is useful for removing extraneous peaks that are likely to be false positives.

```
> msset7 <- peakFilter(msset6, method="freq")
> dim(msset6) # 89 peaks retained

Features   Pixels
      89       81

> dim(msset7) # 10 peaks retained

Features   Pixels
      10       81
```

4.7 Data reduction

Other common forms of data reduction include resampling and binning.

Cardinal can do binning for a fixed width, taken to be 25 in this example. The mean intensity of ions located in the same m/z bin is taken to be the response in the reduced version of the data. The results of binning on pixel 1 is plotted in Figure 9a. The orignal spectrum is plotted in black, with the binned version displayed simultaneously in red.

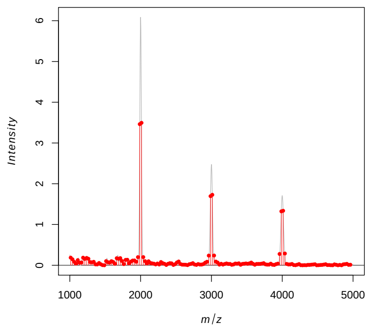
```
> temp <- reduceDimension(msset4, pixel=1, method="bin", width=25, fun=mean, plot=TRUE)
```

There is also the option of doing resampling for a fixed step size. The results of resampling with step size 25 m/z on pixel 1 is plotted in Figure 9b. The original spectrum is plotted in black, with the resampled version displayed simultaneously in red.

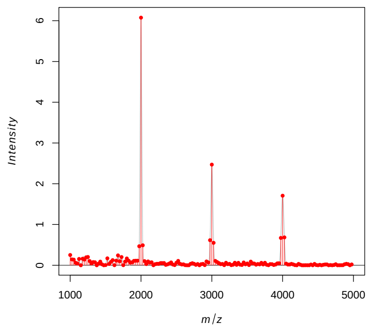
```
> temp <- reduceDimension(msset4, pixel=1, method="resample", step=25, plot=TRUE)
```

Data reduction can be done on the whole dataset at once.

```
> msset8 <- reduceDimension(msset4, method="resample", step=25)
```



(a) Binning



(b) Resampling

Figure 9: Data reduction via binning and resampling.

5 Analysis

For example workflows with analyses of real datasets, please see the vignettes in the companion data package *CardinalWorkflows*.

5.1 Principal components analysis

Principal components analysis (PCA) is a multivariate statistical tool used for dimension reduction and exploratory data analysis. PCA is rarely enough on its own to thoroughly analyze an experiment, but it can sometimes be useful when first exploring a dataset beyond plotting single-ion images. However, PCA should be viewed as a beginning step toward analysis, rather than a complete analysis on its own.

Below, we fit the first three principal components using *Cardinal*'s PCA method to the TIC-normalized dataset.

```
> pca <- PCA(msset4, ncomp=3)
> summary(pca)
```

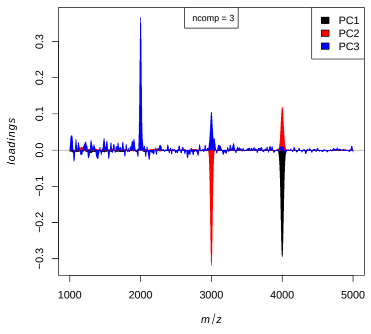
```
                             PC1       PC2         PC3
Standard deviation     6.8564546 2.7617682 0.583106272
Proportion of Variance 0.8550816 0.1387340 0.006184488
Cumulative             0.8550816 0.9938155 1.000000000
```

As the summary shows, the first two principal components together account for 99% of the variation in the dataset, while the third component contributes less than 1%.

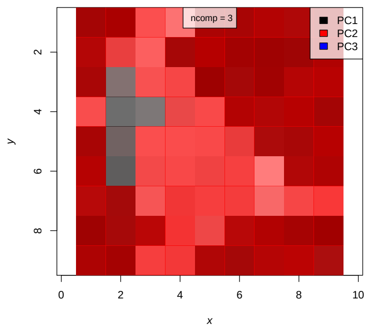
Below, we plot the loading plots against the m/z values, and an image of the PC scores for the first two components.

```
> plot(pca, col=c("black", "red", "blue"))
```

```
> image(pca, column=c("PC1","PC2"), col=c("black", "red", "blue"))
```



(a) PC loadings (first 3 components)



(b) PC scores (first 2 components)

Figure 10: Principal components analysis.

5.2 Partial least squares

Partial least squares (PLS), also called projection to latent structures, is a multivariate method from chemometrics that has been shown to be useful for classification of mass spectrometry images. When used for classification, it is known as partial least squares discriminant analysis, or PLS-DA. PLS-DA works similarly to PCA, but it is a supervised method, so it requires labels known *a priori* to train a classifier.

Here, we train a PLS classifier using the pattern labels from earlier.

Note that the PLS method is vectorized on the ncomp argument, so multiple models will be fit, and we will specify which one we want to show when we plot the results.

```
> pls <- PLS(msset4, y=pattern, ncomp=1:2)
> summary(pls)
```

```
$`ncomp = 1`
                 blue     black       red
Accuracy    0.9629630 0.6049383 0.6419753
Sensitivity 1.0000000 0.4000000 0.0000000
Specificity 0.9117647 0.6184211 1.0000000
```

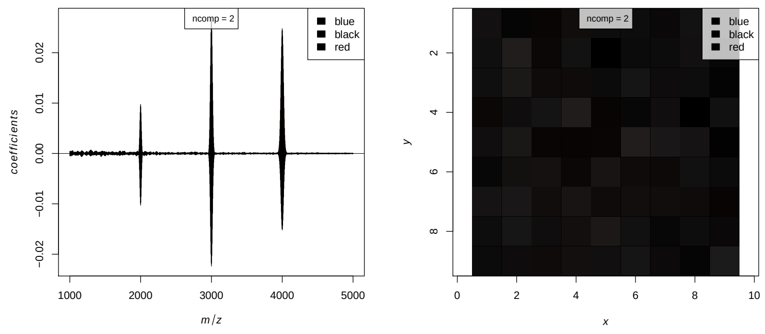
```
FDR         0.0600000 0.9354839       NaN

$`ncomp = 2`
            blue black red
Accuracy       1     1   1
Sensitivity    1     1   1
Specificity    1     1   1
FDR            0     0   0
```

Since we are predicting on the same data we have used to fit the classifier, the accuracy is very good, and only two PLS components are needed to produce perfect accuracy.

When working with classification on real data, cross-validation should always be used, using the `cvApply` method, to avoid biased results. See `?cvApply`.

```
> plot(pls, model=list(ncomp=2), col=mycol)
```

```
> image(pls, model=list(ncomp=2), col=mycol)
```



(a) PLS coefficients (b) PLS prediction

Figure 11: Partial least squares.

A variation on PLS, orthogonal partial least squares (O-PLS) is also implemented in *Cardinal*. O-PLS can often improve interpretability of the PLS model coefficients, while producing similar accuracy. See `?OPLS`.

5.3 Spatially-aware nearest shrunken centroids

Cardinal offers a novel clustering and classification method based on the spatial smoothing of [4] and the nearest shrunken centroids of [5]. This is the `spatialShrunkenCentroids` method, which can be used both for clustering and for classification.

In the unsupervised case, the goal of this analysis is to cluster the imaging data into chemically-distinct and chemically-homogenous spatial regions. For imaging experiments, this is called spatial segmentation, and the clusters are sometimes called segments.

In the supervised case, the labels from known class conditions are used as the segments, and the model is fit, but no clustering is done.

Below, we do clustering on the peak-picked dataset. The parameters `r`, `k`, and `s` are the neighborhood smoothing radius, the initial number of segments, and the sparsity parameter, respectively. Since `spatialShrunkenCentroids` is vectoried on these parameters, it will fit five models for the five values of `s` we have given it.

```
> set.seed(1)
> ssca <- spatialShrunkenCentroids(msset7, r=1, k=3, s=0:4, method="adaptive")
> summary(ssca)
```

```
  r k s Predicted # of Classes Mean # of Features per Class
1 1 3 0                      3                           10
2 1 3 1                      3                            3
```

```
3 1 3 2                          3                            3
4 1 3 3                          3                            3
5 1 3 4                          3                            3
```

The `spatialShrunkenCentroids` method uses statistical regularization to performed feature selection. This is shown by the "Mean # of Features per Class" in the summary, which indicates that after s = 1, only three features (three peaks) are retained in the model. Since we know the entire dataset is determined by three peaks, this makes sense.

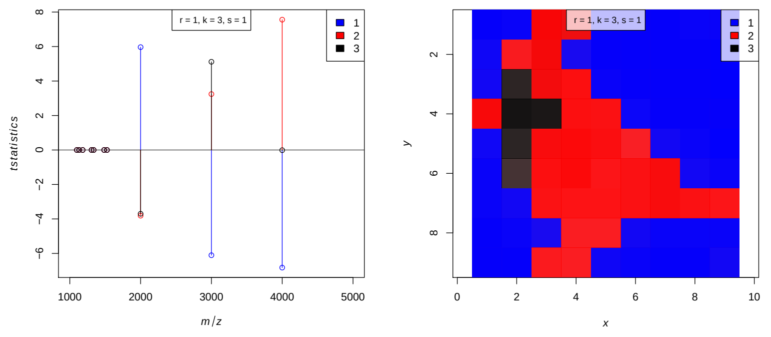
We plot the t-statistics, which rank the relative importance of the features in distinguishing segments from each other. These are closer to zero for features that do not distinguish between segments. Statistical regularization is used to shrink these t-statistics toward zero, so that features that are not useful in the segmentation are dropped from the analysis.

Positive t-statistics indicate systematic enrichment of that feature in a segment, while negative t-statistics indicate systematic absence of that feature in a segment.

Rather than hard segment assignments, `spatialShrunkenCentroids` assigns probabilities for pixel membership to a particular segment. *Cardinal* uses opacity to plot probability, with higher opacity indicating stronger probability. Since the plotted image shows high opacity in a single color for all pixels, the segmentation for this data is very high quality.

```
> plot(ssca, mode="tstatistics", model=list(s=1),
+   col=c("blue", "red", "black"), type=c('p','h'), xlim=c(1000,5000))
```

```
> image(ssca, mode="probabilities", model=list(s=1),
+   col=c("blue", "red", "black"))
```



(a) Shrunken t-statistics (b) Predicted segment probabilies

Figure 12: Spatially-aware nearest shrunken centroids clustering.

See `?spatialShrunkenCentroids` for more options and details.

6 Advanced Topics

6.1 Apply

The `apply` family of functions are a powerful feature of *R*. The `apply` function applies a function over margins of an array, while `sapply` applies a function over every element of a vector-like object. The function `tapply` applies a function over a "ragged" array, so that the function is applied over groups of values given by levels of another variable (usually a factor). In *Cardinal*, the methods `pixelApply` and `featureApply` allow `apply`-like functionality that combine traits of each of these, tailored for imaging datasets.

We need to mark which pixels are blue, black, and which are red, as in the *factor* `pattern` in Section 3.1.

```
> pData(msset)$pg <- pattern
```

Then we need to mark which features (which regions of the mass spectrum) belong to the peaks associated with "blue" ($m/z2000$), "black"($m/z3000$), or "red"($m/z4000$) pixels; the rest of the spectrum is marked as background noise (bg).

```
> fData(msset)$fg <- factor(rep("bg", nrow(fData(msset))), levels=c("bg","blue", "black", "red"))
> fData(msset)$fg[1950 < fData(msset)$mz & fData(msset)$mz < 2050] <- "blue"
> fData(msset)$fg[2950 < fData(msset)$mz & fData(msset)$mz < 3050] <- "black"
> fData(msset)$fg[3950 < fData(msset)$mz & fData(msset)$mz < 4050] <- "red"
```

Now we can experiment with different ways of plotting an imaging dataset.

6.1.1 pixelApply

The method pixelApply allows functions to be applied over all pixels. The function is applied pixel-by-pixel to the feature vectors (mass spectra). Here, we use pixelApply to find the pixel-by-pixel mean intensity of different regions of the mass spectrum. We provide fData(msset)$fg as a grouping variable, since it indicates different regions of the mass spectrum we expect to be associated with either background noise, or blue, red, or black pixels. Since pixelApply knows to look in msset for the variable, we only need to provide fg to the argument .feature.groups.

```
> p1 <- pixelApply(msset, mean, .feature.groups=fg)
> p1[,1:30]
```

```
      x = 1, y = 1 x = 2, y = 1 x = 3, y = 1 x = 4, y = 1
bg        5.591239     5.516053     3.395579     3.185078
blue     17.306620    17.046174     9.112457     8.557135
black     9.974539     9.025602     8.412205     8.844019
red       7.628184     7.010724    12.982994    12.059920
      x = 5, y = 1 x = 6, y = 1 x = 7, y = 1 x = 8, y = 1
bg        5.615956     5.650231     4.957623     5.285583
blue     17.313809    17.481563    14.981038    16.141836
black     9.631894     9.739096     8.005345     9.232942
red       7.883425     7.838796     7.077551     7.870530
      x = 9, y = 1 x = 1, y = 2 x = 2, y = 2 x = 3, y = 2
bg        5.523104     5.173204     3.589339     3.361056
blue     16.873337    15.842321    11.037839     8.760469
black     9.679532     8.708974     8.632293     8.432287
red       8.436895     7.819247    11.809760    12.806359
      x = 4, y = 2 x = 5, y = 2 x = 6, y = 2 x = 7, y = 2
bg        5.409369     5.962200     5.913981     5.710577
blue     16.625576    18.391835    18.392521    17.591820
black     9.626762     9.028662    10.234255    10.160928
red       7.634099     7.807045     7.588946     7.244261
      x = 8, y = 2 x = 9, y = 2 x = 1, y = 3 x = 2, y = 3
bg        5.681514     5.860902     5.770548     4.274740
blue     17.528622    18.254246    17.893589    11.131247
black    10.370499     9.540406    10.013747    14.349932
red       7.563269     7.941176     8.005526     8.245206
      x = 3, y = 3 x = 4, y = 3 x = 5, y = 3 x = 6, y = 3
bg        3.146736     3.316402     5.317308     5.688381
blue      8.295846     9.750688    16.232286    17.530305
black     7.718586     8.035290     9.444263    10.440994
red      11.762278    12.332335     6.764745     8.139496
      x = 7, y = 3 x = 8, y = 3 x = 9, y = 3 x = 1, y = 4
bg        6.096776     5.727348     5.816680     3.290318
blue     18.911349    17.500409    17.829095     9.134986
black    10.710668     9.470964     9.166336     8.047249
red       7.897016     8.365797     8.130600    12.516607
      x = 2, y = 4 x = 3, y = 4
bg        4.533675     4.376095
blue     11.546565    11.061959
```

```
black    15.589934    14.972439
red       7.659493     8.236587
```

By comparing side-by-side with the ground truth (which we have stored in the variable `pData(msset)$pg`), we see the result is as we expected. For "blue" pixels, the mean intensity of features belonging to the "blue"-associated peak (m/z 2000) is higher. For "black" pixels, the mean intensity of features belonging to the "black"-associated peak (m/z 3000) is higher. Finally, for "red" pixels, the mean intensity of features belonging to the "red"-associated peak (m/z 4000) is higher.

```
> cbind(pData(msset), t(p1))[1:30,c("pg","blue", "black", "red")]
```

```
                   pg      blue     black       red
x = 1, y = 1  blue 17.306620  9.974539  7.628184
x = 2, y = 1  blue 17.046174  9.025602  7.010724
x = 3, y = 1   red  9.112457  8.412205 12.982994
x = 4, y = 1   red  8.557135  8.844019 12.059920
x = 5, y = 1  blue 17.313809  9.631894  7.883425
x = 6, y = 1  blue 17.481563  9.739096  7.838796
x = 7, y = 1  blue 14.981038  8.005345  7.077551
x = 8, y = 1  blue 16.141836  9.232942  7.870530
x = 9, y = 1  blue 16.873337  9.679532  8.436895
x = 1, y = 2  blue 15.842321  8.708974  7.819247
x = 2, y = 2   red 11.037839  8.632293 11.809760
x = 3, y = 2   red  8.760469  8.432287 12.806359
x = 4, y = 2  blue 16.625576  9.626762  7.634099
x = 5, y = 2  blue 18.391835  9.028662  7.807045
x = 6, y = 2  blue 18.392521 10.234255  7.588946
x = 7, y = 2  blue 17.591820 10.160928  7.244261
x = 8, y = 2  blue 17.528622 10.370499  7.563269
x = 9, y = 2  blue 18.254246  9.540406  7.941176
x = 1, y = 3  blue 17.893589 10.013747  8.005526
x = 2, y = 3 black 11.131247 14.349932  8.245206
x = 3, y = 3   red  8.295846  7.718586 11.762278
x = 4, y = 3   red  9.750688  8.035290 12.332335
x = 5, y = 3  blue 16.232286  9.444263  6.764745
x = 6, y = 3  blue 17.530305 10.440994  8.139496
x = 7, y = 3  blue 18.911349 10.710668  7.897016
x = 8, y = 3  blue 17.500409  9.470964  8.365797
x = 9, y = 3  blue 17.829095  9.166336  8.130600
x = 1, y = 4   red  9.134986  8.047249 12.516607
x = 2, y = 4 black 11.546565 15.589934  7.659493
x = 3, y = 4 black 11.061959 14.972439  8.236587
```

We can manually construct the images corresponding to the mean intensity of the three peaks centered at m/z 2000, m/z 3000, and m/z 4000 and plot their images. This is shown in Figure 13

```
> temp1 <- MSImageSet(spectra=t(as.vector(p1["blue",])), coord=coord(msset), mz=2000)
> image(temp1, feature=1, col=alpha.colors(100, "blue"), sub="m/z = 2000")
```

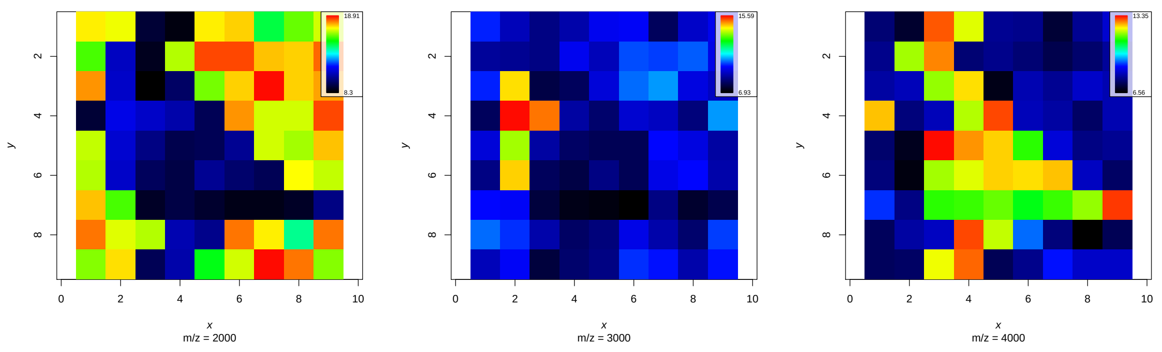
```
> temp1 <- MSImageSet(spectra=t(as.vector(p1["black",])), coord=coord(msset), mz=3000)
> image(temp1, feature=1, col=alpha.colors(100, "black"), sub="m/z = 3000")
```

```
> temp2 <- MSImageSet(spectra=t(as.vector(p1["red",])), coord=coord(msset), mz=4000)
> image(temp2, feature=1, col=alpha.colors(100, "red"),  sub="m/z = 4000")
```

If only the plots are desired rather than the actual data, then `image` can be used to perform these steps automatically while producing the plot. See *Cardinal plotting* for how to do this.

6.1.2 `featureApply`

The method `featureApply` allows functions to be applied over all features. The function is applied to the flattened false-image vectors. These vectors are the pixel-by-pixel intensities of a single-feature image, not including



(a) Blue Peak (b) Black Peak (c) Red Peak

Figure 13: Mean intensites of the three peaks centered at m/z 2000, m/z 3000 and m/z 4000.

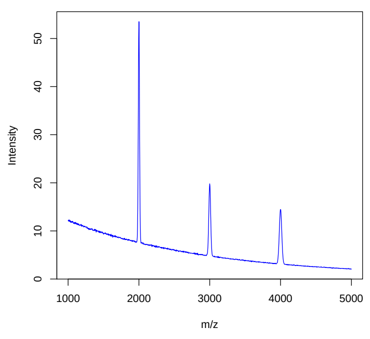
missing pixels. Here, we use `featureApply` to find the mean spectrum for different groups of pixels. We provide `pData(msset)$pg` as a grouping variable, since it indicates the kind of pixel. We desire mean spectra for the black pixels, the red pixels, and the blue pixels. As before, since `featureApply` knows to look in `msset`, we only need to provide `pg` to the argument `.pixel.groups`.

```
> f1 <- featureApply(msset, mean, .pixel.groups=pg)
> f1[,1:30]
```

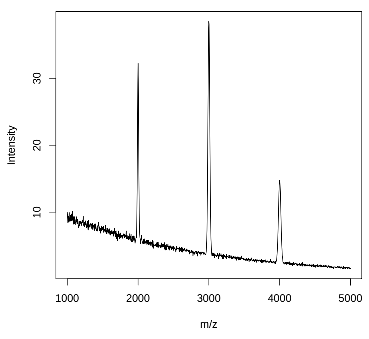
```
      m/z = 1000 m/z = 1003.3 m/z = 1006.6 m/z = 1009.9 m/z = 1013.2
blue   12.113878    12.077172    12.271484    12.011245    12.106858
black  10.005194     9.491585     8.920648     8.395593     9.349371
red     7.019996     7.095126     7.206785     7.046938     6.770562
      m/z = 1016.5 m/z = 1019.8 m/z = 1023.1 m/z = 1026.4
blue     12.291783    12.011774    11.936433    12.210184
black     8.972390     8.498503     9.054659     9.949408
red       7.135863     6.915438     7.033159     7.091420
      m/z = 1029.7 m/z = 1033 m/z = 1036.3 m/z = 1039.6 m/z = 1042.9
blue     11.989588  11.848585    11.790103    12.031041    11.732924
black     8.628287   8.868370     8.991401     9.277913     8.660284
red       6.707759   7.258157     6.922927     7.025668     7.069312
      m/z = 1046.2 m/z = 1049.5 m/z = 1052.8 m/z = 1056.1
blue     11.867339    11.796285    11.848332    11.996489
black     8.677262     9.554605     9.319258     8.995928
red       7.120718     6.970966     6.852143     6.636296
      m/z = 1059.4 m/z = 1062.7 m/z = 1066 m/z = 1069.3 m/z = 1072.6
blue     11.853608    11.766950  12.006615    11.757288    11.696140
black     8.754828     9.730654   9.159634     8.890601     8.980763
red       6.992209     6.835762   6.493719     6.675064     6.787080
      m/z = 1075.9 m/z = 1079.2 m/z = 1082.5 m/z = 1085.8
blue     11.544314    11.550182    11.684293    11.747299
black    10.102702     8.745331     9.359917     8.196362
red       6.970741     6.662426     6.664785     6.829065
      m/z = 1089.1 m/z = 1092.4 m/z = 1095.7
blue     11.787957    11.472797    11.710456
black     9.204949     9.279710     9.049170
red       6.849054     7.000153     6.704578
```

Again, we can check the results by plotting them in Figure 14.

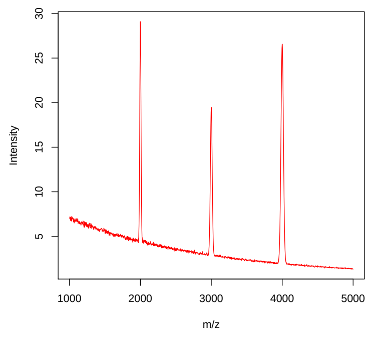
```
> plot(mz(msset), f1["blue",], type="l", xlab="m/z", ylab="Intensity", col="blue")

> plot(mz(msset), f1["black",], type="l", xlab="m/z", ylab="Intensity", col="black")

> plot(mz(msset), f1["red",], type="l", xlab="m/z", ylab="Intensity", col="red")
```



(a) Blue Pixels



(b) Black Pixels



(c) Red Pixels

Figure 14: Mean spectra of blue, black, and red regions.

As expected,we see the mean spectrum of the blue pixels has a higher peak at m/z 2000, we see the mean spectrum of the black pixels has a higher peak at m/z 3000, while the mean spectrum of the red pixels has a higher peak at m/z 4000. As before, if only the plots are desired rather than the actual data, then `plot` can be used to perform these steps automatically. See *Cardinal plotting* for how to do this.

6.2 Simulation

Cardinal provides functions for the simulation of mass spectra and mass spectrometry imaging datasets. This is of interest to developers for testing newly developed methodology for analyzing mass spectrometry imaging experiments.

6.2.1 Simulation of spectra

The `generateSpectrum` function can be used to simulate mass spectra. Its parameters can be tuned to simulate different kinds of mass spectra from different kinds of machines, and different protein and peptide patterns.

One spectrum with m/z range from 1001 to 20000, 50 randomly selected peaks, baseline 3000, and m/z resolution 100 is generated below and plotted in Figure 15a.

```
> set.seed(1)
> s1 <- generateSpectrum(1, range=c(1001, 20000), centers=runif(50, 1001, 20000),
+                        baseline=2000, resolution=100, step=3.3)
> plot(x ~ t, data=s1, type="l", xlab="m/z", ylab="Intensity")
```

An example with fewer peaks, larger baseline, and lower resolution (Figure 15b):

```
> set.seed(2)
> s2 <- generateSpectrum(1, range=c(1001, 20000), centers=runif(20, 1001, 20000),
+                        baseline=3000, resolution=50, step=3.3)
> plot(x ~ t, data=s2, type="l", xlab="m/z", ylab="Intensity")
```

Above we simulated MALDI-like spectra. We can also simulate DESI-like spectra, shown in Figure 16.

```
> set.seed(3)
> s3 <- generateSpectrum(1, range=c(101, 1000), centers=runif(25, 101, 1000),
+                        baseline=0, resolution=250, noise=0.1, step=1.2)
> plot(x ~ t, data=s3, type="l", xlab="m/z", ylab="Intensity")

> set.seed(4)
> s4 <- generateSpectrum(1, range=c(101, 1000), centers=runif(100, 101, 1000),
+                        baseline=0, resolution=500, noise=0.2, step=1.2)
> plot(x ~ t, data=s4, type="l", xlab="m/z", ylab="Intensity")
```

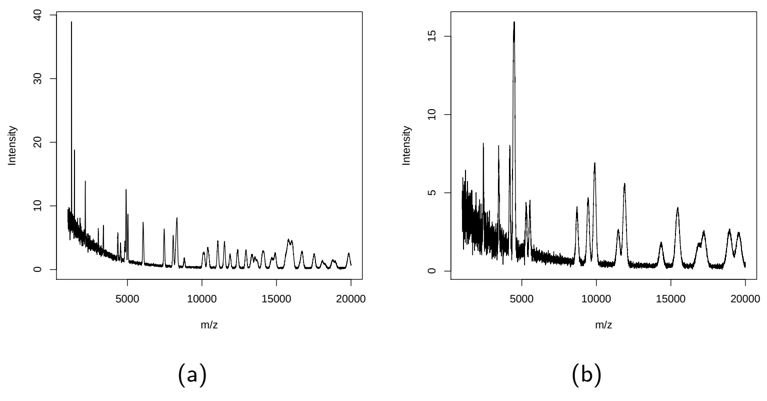


Figure 15: MALDI-like simulated spectra.

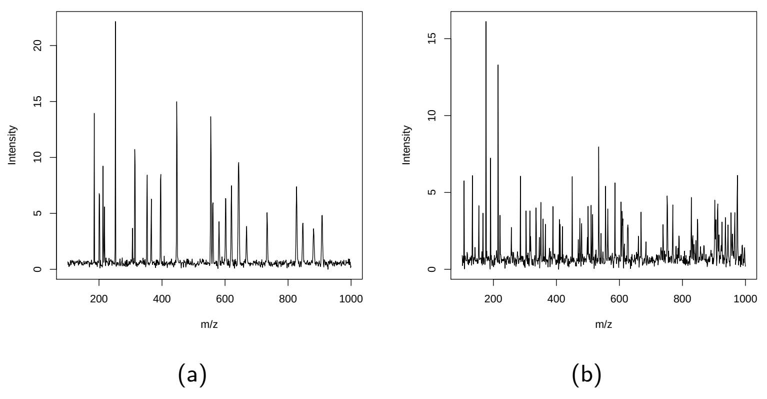


Figure 16: DESI-like simulated spectra.

6.2.2 Simulation of images

The `generateImage` function can be used to simulate mass spectral images. This is a simple wrapper for `generateSpectra` that will generate unique spectral patterns based on a spatial pattern. The generated mass spectra will have a unique peak associated with each region. The pattern must have discrete regions, most easily given in the form of an integer matrix. We use a matrix in the pattern of a cardinal.

```
> data <- matrix(c(NA, NA, 1, 1, NA, NA, NA, NA, NA, NA, 1, 1, NA, NA,
+  NA, NA, NA, NA, NA, 0, 1, 1, NA, NA, NA, NA, NA, 1, 0, 0, 1,
+  1, NA, NA, NA, NA, NA, 0, 1, 1, 1, 1, NA, NA, NA, NA, 0, 1, 1,
+  1, 1, 1, NA, NA, NA, NA, 1, 1, 1, 1, 1, 1, 1, NA, NA, NA, 1,
+  1, NA, NA, NA, NA, NA, NA, 1, 1, NA, NA, NA, NA, NA), nrow=9, ncol=9)
```

As seen in Figure **??**, we can plot the ground truth image directly.

```
> image(data[,ncol(data):1], col=c("black", "red"))
```

Now we generate the dataset. To make it easy to visualize, we set up the `range` and `step` size so that the feature indices correspond directly to their values. We create two peaks at m/z 100 and m/z 200, one of which is associated with each region in the image.

```
> set.seed(1)
> img1 <- generateImage(data, range=c(1,1000), centers=c(100,200), step=1, as="MSImageSet")
```

Now to confirm the reasonability of our simulated dataset, we plot images corresponding to the two peaks associated with each region in Figure 18b. (Note that rows in the original matrix correspond to the x-axis in the image and the columns correspond to the y-axis.)

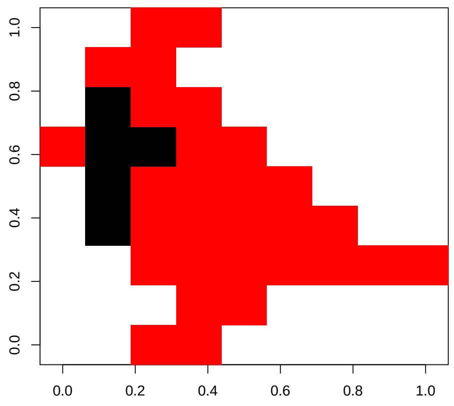
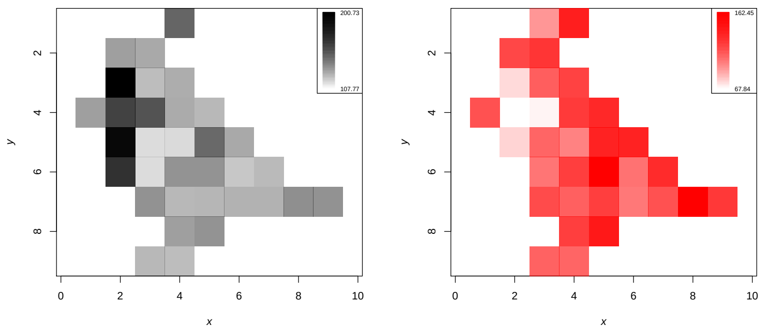


Figure 17: Ground truth image used to generate the simulated dataset.

```
> image(img1, feature=100, col.regions=alpha.colors(100, "black"))
> image(img1, feature=200, col.regions=alpha.colors(100, "red"))
```



(a) Black Peak (b) Red Peak

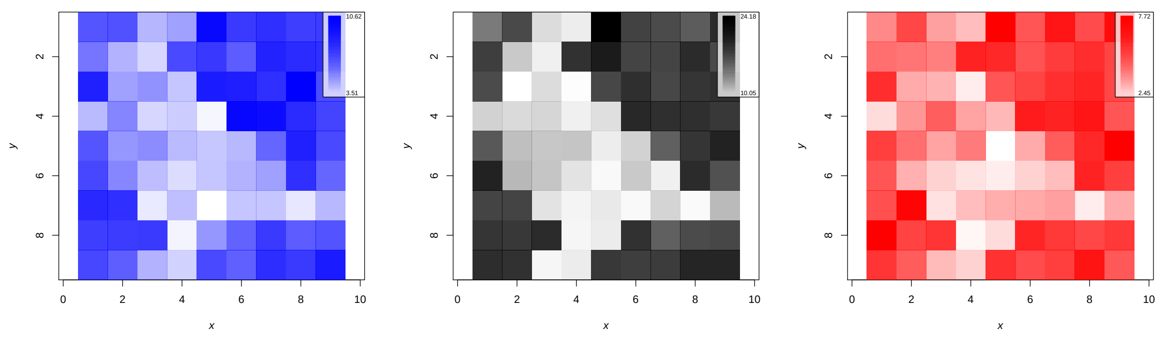
Figure 18: Generated image from an integer matrix.

We can generate the same kind of dataset using a `factor` and a `data.frame` of coordinates, as is done in the running example for earlier sections of this vignette.

```
> pattern <- factor(c(0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 2, 2, 0,
+   0, 0, 0, 0, 0, 0, 1, 2, 2, 0, 0, 0, 0, 0, 2, 1, 1, 2,
+   2, 0, 0, 0, 0, 0, 1, 2, 2, 2, 2, 0, 0, 0, 0, 1, 2, 2,
+         2, 2, 2, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 2,
+         2, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0),
+         levels=c(0,1,2), labels=c("blue", "black", "red"))
> coord <- expand.grid(x=1:9, y=1:9)
> set.seed(2)
> msset <- generateImage(pattern, coord=coord,
+                                     range=c(1000, 5000), centers=c(2000, 3000, 4000),
+                                        resolution=100, step=3.3, as="MSImageSet")
```

Again, we can plot the images to see that the simulated dataset is the same pattern as before (though the exact intensities will differ, because we have used a different seed for the random number generator), Figure 19.

```
> image(msset, feature=200, col.regions=alpha.colors(100, "blue"))
> image(msset, feature=300, col.regions=alpha.colors(100, "black"))
> image(msset, feature=400, col.regions=alpha.colors(100, "red"))
```



(a) Blue Peak (b) Black Peak (c) Red Peak

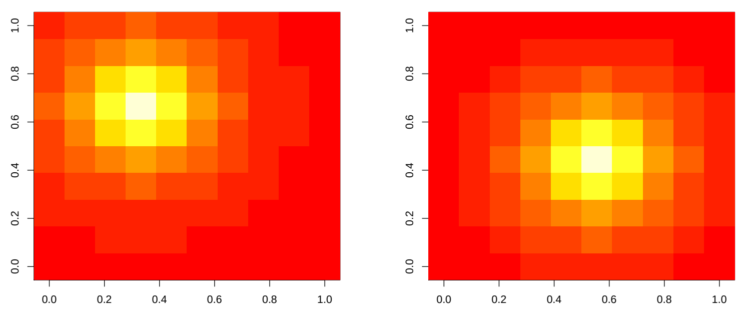
Figure 19: Generated images from factor and coordinates

6.2.3 Advanced simulation

The generateImage function provides a straightforward method for rapid simulation of many kinds of images to test classification and clustering models, but suppose we wish to simulate a more complex dataset with spatial correlations. Below we simulate a dataset with two overlapping regions. In each of these regions, the intensity degrades with distance from the center of the region, implining spatial correlation, Figure 20.

```
> x1 <- apply(expand.grid(x=1:10, y=1:10), 1,
+             function(z) 1/(1 + ((4-z[[1]])/2)^2 + ((4-z[[2]])/2)^2))
> dim(x1) <- c(10,10)
> image(x1[,ncol(x1):1])
```

```
> x2 <- apply(expand.grid(x=1:10, y=1:10), 1,
+             function(z) 1/(1 + ((6-z[[1]])/2)^2 + ((6-z[[2]])/2)^2))
> dim(x2) <- c(10,10)
> image(x2[,ncol(x2):1])
```



(a) Region 1 (b) Region 2

Figure 20: Ground truth images of a dataset with overlapping regions.

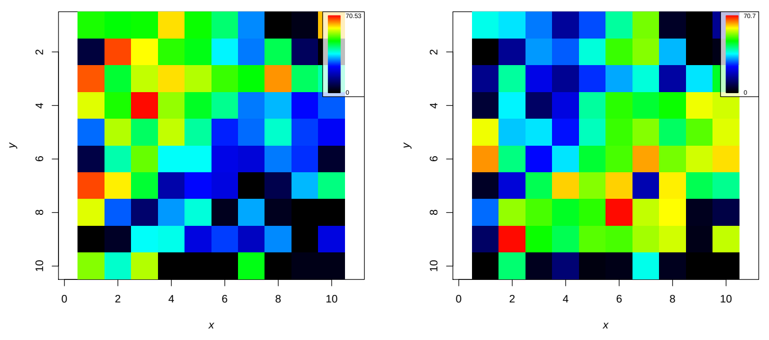
We generate the image by using generateSpectrum with the calculated mean intensities. We use two peaks for the two regions with nearly overlapping peaks at m/z 500 and m/z 510.

```
> set.seed(1)
> x3 <- mapply(function(z1, z2) generateSpectrum(1, centers=c(500,510),
+                                                 intensities=c(z1, z2),
+                                                 range=c(1,1000),
+                                                         resolution=100,
+                                                         baseline=0, step=1)$x,
+                  as.vector(x1), as.vector(x2))
> img3 <- MSImageSet(x3, coord=expand.grid(x=1:10, y=1:10), mz=1:1000)
```

Now we can plot the ion images for each of the two peaks in 21.

```
> image(img3, feature=500, col=intensity.colors(100))
```

```
> image(img3, feature=510, col=intensity.colors(100))
```



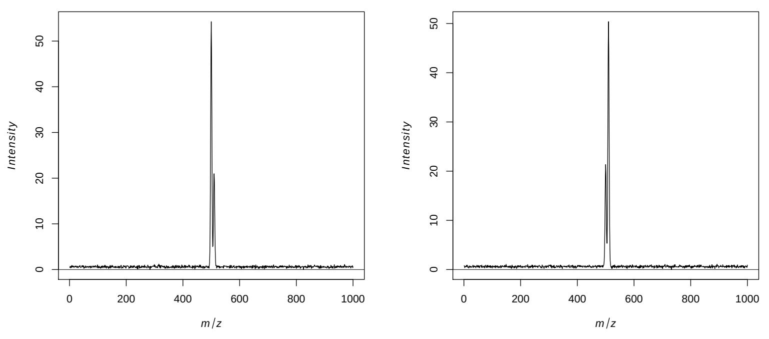
(a) Single ion image for peak at $mz500$ (b) Single ion image for peak at $mz510$

Figure 21: Simulated mass spectral images at the two peaks.

Finally, we plot the mass spectrum for a pixel from each region in Figure 22

```
> plot(img3, pixel=34, type="l")
```

```
> plot(img3, pixel=56, type="l")
```



(a) Region 1, pixel 34 spectrum (b) Region 1, pixel 56 spectrum

Figure 22: Simulated mass spectra from each of the two regions.

By creating spatial correlation patterns and combining them with the `intensities`, `sd`, and `noise` arguments in `generateSpectrum`, it is possible to simulate more complex mass spectrometry imaging datasets.

7 Session info

- R version 3.1.2 (2014-10-31), `x86_64-apple-darwin13.4.0`
- Locale: `en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, methods, parallel, stats, utils
- Other packages: Biobase 2.24.0, BiocGenerics 0.10.0, Cardinal 0.8.7

- Loaded via a namespace (and not attached): BiocStyle 1.2.0, fields 7.1, grid 3.1.2, irlba 1.0.3, lattice 0.20-29, maps 2.3-9, MASS 7.3-35, Matrix 1.1-4, signal 0.7-4, sp 1.0-15, spam 1.0-1, stats4 3.1.2, tools 3.1.2

References

[1] Thorsten Schramm, Alfons Hester, Ivo Klinkert, Jean-Pierre Both, Ron M. A. Heeren, Alain Brunelle, Olivier Laprévote, Nicolas Desbenoit, Marie-France Robbe, Markus Stoeckli, Bernhard Spengler, and Andreas Römpp. imzml – a common data format for the flexible exchange and processing of mass spectrometry imaging data. *Journal of Proteomics*, 75(16):5106 – 5110, 2012. Special Issue: Imaging Mass Spectrometry: A User's Guide to a New Technique for Biological and Biomedical Research. URL: http://www.sciencedirect.com/science/article/pii/S1874391912005568, doi:http://dx.doi.org/10.1016/j.jprot.2012.07.026.

[2] Sören-Oliver Deininger, Dale S. Cornett, Rainer Paape, Michael Becker, Charles Pineau, Sandra Rauser, Axel Walch, and Eryk Wolski. Normalization in MALDI-TOF imaging datasets of proteins: practical considerations. *Analytical and Bioanalytical Chemistry*, 401(1):167–181, 2011. URL: http://dx.doi.org/10.1007/s00216-011-4929-z, doi:10.1007/s00216-011-4929-z.

[3] C. Yang, Z. He, and W. Yu. Comparison of public peak detection algorithms for MALDI mass spectrometry data analysis. *BMC Bioinformatics*, 10, 2009.

[4] Theodore Alexandrov and Jan Hendrik Kobarg. Efficient spatial segmentation of large imaging mass spectrometry datasets with spatially aware clustering. *Bioinformatics (Oxford, England)*, 27(13):i230–8, July 2011. URL: http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3117346&tool=pmcentrez&rendertype=abstract, doi:10.1093/bioinformatics/btr246.

[5] R. Tibshirani, T. Hastie, B. Narasimhan, and G. Chu. Class prediction by nearest shrunken with applications to DNA microarrays. *Statistical Science*, 18:104, 2003.