CNSigs: An R package for the identification of copy number signatures

David Tallman
The Ohio State University

Abstract

Copy number aberrations (CNAs) are gains and losses of large genomic segments present across most cancer types and are a hallmark of cancer genomic alterations. However, the processes underlying CNAs and characteristic patterns of CNAs are poorly understood. Using single nucleotide variant (SNV) data, bioinformatic advances have identified underlying mutational signatures resulting from distinct mutational processes. Mutational signatures have led to a variety of discoveries, several of which are being investigated in clinical management of cancer. The development of algorithms able to uncover similar signatures for CNAs, rather than SNVs, is still in its infancy. Here we present an analysis package for the R programming language called CNSigs that allows for the robust and reproducible derivation of copy number signatures. Based on a list of extracted copy number features previously verified in ovarian cancer, we utilize mixed model approaches and non-negative matrix factorization to derive CNA signatures across cancer types. The development of a package to derive signatures from copy number data allows further investigation of underlying processes that may be responsible for these CNA fingerprints. The CNSigs package also allows researchers to easily analyze their own samples to look for signatures in their copy number profiles and to compare these to signatures previously derived for their cancer type.

Keywords: R, copy number signatures, copy number segments.

1. Introduction

This package is used to generate and analyze signatures derived from copy number segments. There are two main ways to use the analysis pipeline. You can either run it step by step so that you can see the results as you go, or all at once using a single function. The individual steps are layed out in the following sections, and the single function is described in the section "Running the full pipeline". The package is able to save all of the results into a folder that includes the plots and other outputs from the pipeline. The package also has many functions to analyze and interpret your results. It also allows for parallelization and the tuning of several parameters.

2. Running the Pipeline

The package can be loaded by simply using the library command. This document will begin by walking through each step of the pipeline. The method for running the full pipeline at once is described later in this document.

```
library(CNSigs)
```

2.1. Loading Data

The first step to using the pipeline is to load in the data. CNSigs supports running the pipeline straight from an already formatted list or from a text file. The pipeline expects the data.frames to only have 5 columns with the titles of "ID","chromosome","start","end","segVal". If your text file has columns that are not named in that exact way but contains the same data, you can use the colMap parameter in the readSegs function. If you look at the example data below, you can see that it has all of the data that is needed, but the columns are named differently. In order to load in that segment file properly, you must specify the mapping of the column name to the expected column names using the colMap parameter of readSegs as seen below.

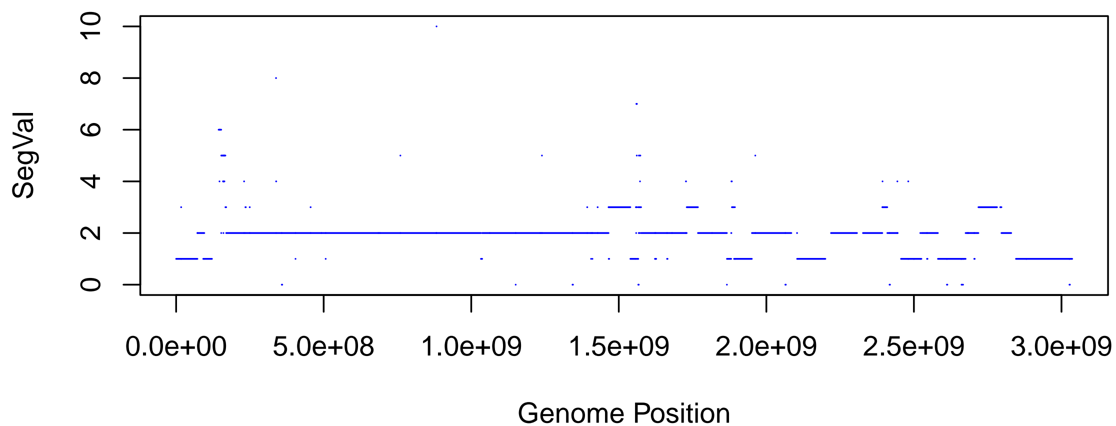
	A	B	C	D	E	F	G	H	I
1	Sample_ID	Chrom	Start	End	major_CN	minor_CN	total_CN	segLength	
2	1	1	61735	16830583	1	0	1	16768848	
3	1	1	16830820	17214822	2	1	3	384002	
4	1	1	17217283	72193593	1	0	1	54976310	
5	1	1	72194476	72242590	1	1	2	48114	
6	1	1	72243242	91917019	1	1	2	19673777	
7	1	1	91917282	94089731	1	0	1	2172449	
8	1	1	94095259	95532174	1	1	2	1436915	
9	1	1	95536080	121482979	1	0	1	25946899	
10	1	1	144007049	146916926	3	3	6	2909877	
11	1	1	146917589	147082953	2	2	4	165364	
12	1	1	147083114	152744962	3	3	6	5661848	
13	1	1	152747126	152847892	1	1	2	100766	
14	1	1	152850409	158764565	3	2	5	5914156	
15	1	1	158764612	160194350	2	2	4	1429738	
16	1	1	160196952	160473270	1	1	2	276318	
17	1	1	160473286	161463601	2	2	4	990315	
18	1	1	161463814	161858592	3	2	5	394778	
19	1	1	161859230	162268825	3	2	5	409595	
20	1	1	162272769	163781153	2	2	4	1508384	
21	1	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	

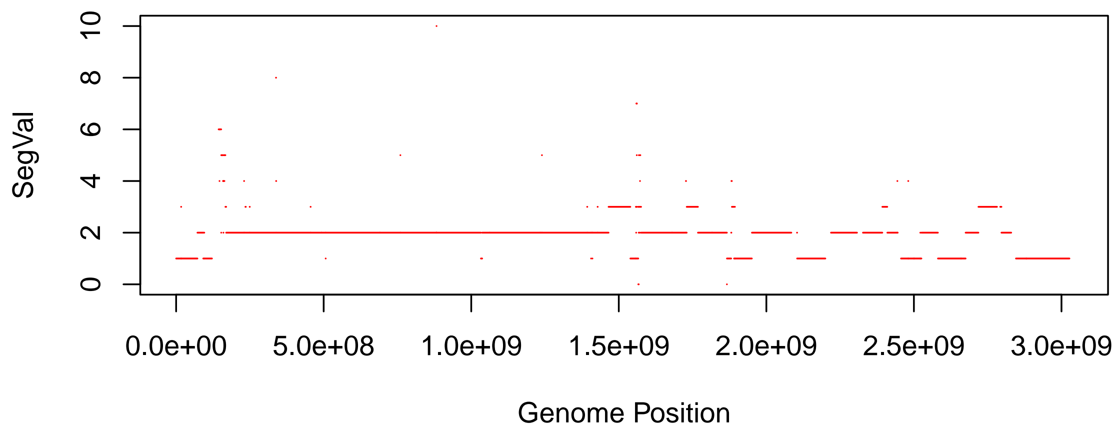
```
readSegs("./SampleSegs.txt",colMap = c("Sample_ID","Chrom","Start","End","Total_CN"))
```

2.2. Smoothing the Data

One of the data preprocessing steps for the pipeline is to smooth the segments. This process is used to reconcile the differences across different copy number calling pipelines. Since the features extracted from the samples are directly related to the segments that were called, by smoothing them, we are able to dampen the effects of using different copy number callers. The package has a function called smoothSegs that is able to accomplish this. We recommend this smoothing step no matter which copy number caller is used.

```
smooth = smoothSegs(segDataExp)

#Visualize the smoothed segments
toPlot = list(segDataExp[[1]],smooth[[1]])
names(toPlot) = c("Original","Smoothed")
plotSegs(toPlot, sep = T)
```





As you can see above, the smoothing generally removes some of the smaller and more erroneous segments and leaves behind the higher confidence segments. The number in the parentheses next to the sample name denotes the number of segments. Smoothing tends to lessen the effect of using different copy number callers on the resulting signatures.

2.3. Extracting the copy number features

The CNSigs package uses six different copy number features based on those used by Macyntire et. al. Extracting these features from a dataset is very easy using the package. The function extractCNFeats will return a list object with the six different features inside. The six features extracted are segment size, the number of breakpoints per 10 megabase bin, the number of copy number oscillation events, the average size of changepoints, the average copynumber, and the number of breakpoints per chromsome arm. Most of the features are and average value per chromosome. The features are then passed onto the mixed modelling step

```
feats = extractCNFeats(smooth)
```

2.4. Fitting the mixture models

After extracting the features, the pipeline then fits a mixture model onto each of the features and uses these mixture models to define the underlying components of the signatures. By default, the package will search for a mixture of between 2 and 10 models of the corresponding distribution types for each feature. This allows the pipeline to account for cases that have different underlying subgroups for each feature. If you want to change the range of the number of components to look for, you can do so using the mincomps and maxcomps parameters. Both of these are a vector of 6 values corresponding to the min or max number of components for each of the features. For more information, refer to ?fitModels. You may also want to skip this step and use fixed components, if so, refer to the "Fixed components" section later in this document.

```
comps = fitModels(feats)
```

Peak Reduction

Since most of the copy number features have a "normal" value, it often results in a abnormally large peak at that value if you have enough samples. The flexmix package, which is used for the mixed modelling is often unable to accomodate these peaks well and will often fail to converge when finding the components. In order to get around this, the package includes an option to perform peak reduction before the modeling step. This method looks through the histogram of the input feature and identifies peaks that are larger than the surrounding bins. It then makes these peaks smaller. This preserves the overall structure of the distribution so that the modelling step will no longer fail and the input data is mainly preserved. By setting the pR parameter to T during your fitModels call, it will perform peak reduction before each of the modelling steps.

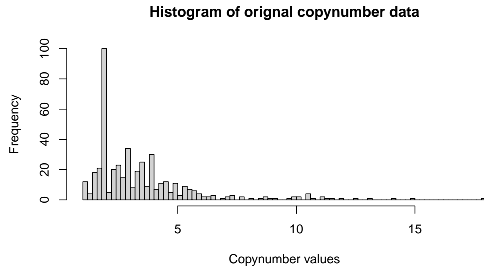


Figure 1: Original

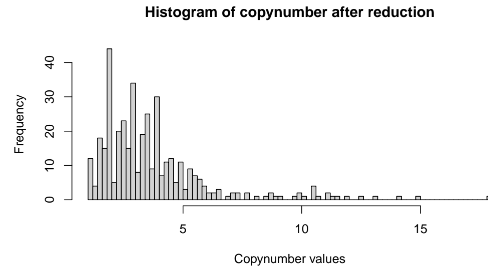
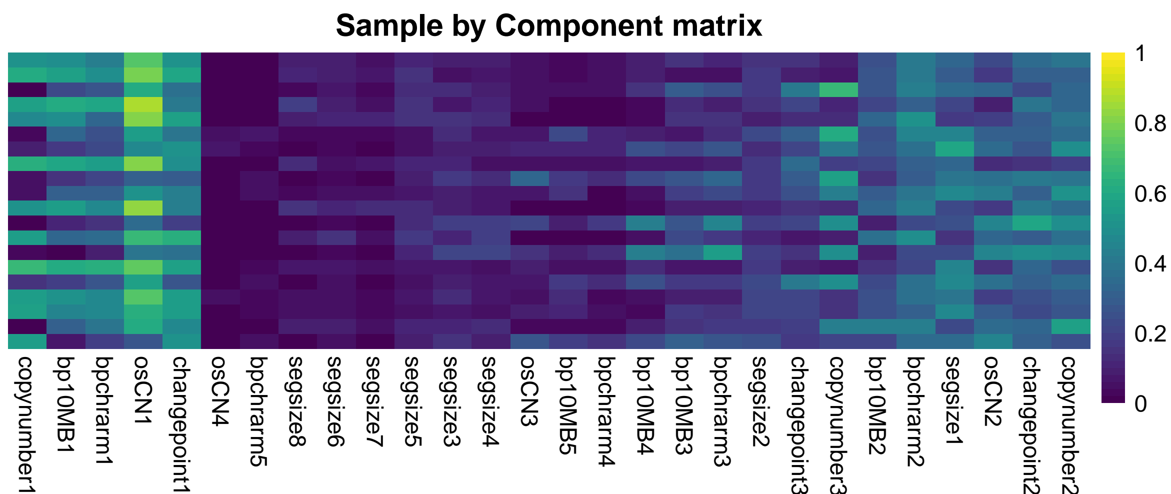


Figure 2: Reduced Peaks

2.5. Generating the sample by component matrix (scm)

Once you have the components, you then have to create the sample by component matrix (scm). The scm is a matrix that shows the distribution of the components within each of the samples. The scm is used during NMF in order to define the signatures. In order to generate the scm, the package looks at each sample and finds the posterior probabilty that the samples features lie within each of the derived components. It then sums these up and normalizes them by the number of features for the sample. This gives you an idea of how much each component can be found within a particular sample. The full pipeline automatically generates a heatmap plot to display this scm, but it can also be done using the following code below.

```
scm = generateSCM(feats, cancerComps)
plotScm(scm)
```



2.6. Adding Ploidy Data (Optional)

After generating the scm, you can include ploidy data to be used in the signature identification. The package has a function called addPloidyData. If you give the function the scm from the

previous step and the samples ploidy data, it will return a new scm that includes the ploidy data transformed to log2 ploidy. For this function make sure that you give it the absolute ploidy because this function performs the transform to log2 by default. If you ploidy data is in the same data file as your segment data, you can read in both at the same time using the readSegs function using the readPloidy parameter set to TRUE.

```
scm = addPloidyData(scm, ploidyData)
```

2.7. Generating the signatures

Once the scm has been generated, you can now perform the NMF step using the createSigs function. The main parameter for this function, besides the scm, is nsig, which determines how many signatures the function will find. If you aren't sure how many signatures is appropriate, you can use the determineSigNum function as laid out in a later section. This function will return the resulting NMF object. In order to look at the final component weights for the signatures, you can use the NMF function scoef in order to get the scaled coefficients for the signatures. You can also use the basis function from NMF in order to see the signature exposures for each of the samples.

```
sigs = createSigs(scm, 5)
```

2.8. Generating the exposures

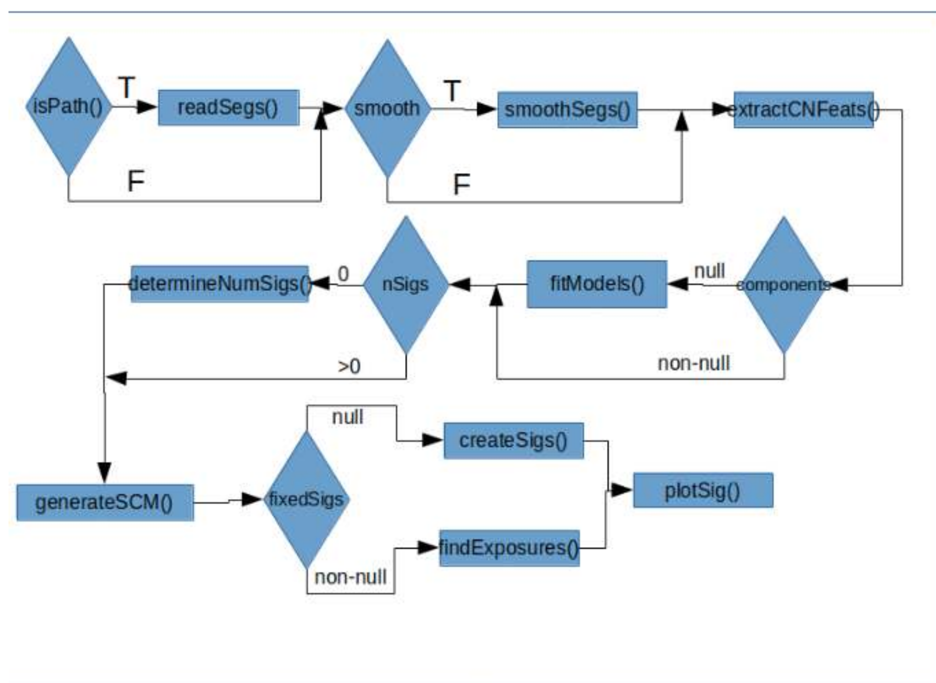
As an alternate method, after the scm has been generated, you can utalize a set of fixed signatures with this function to identify the signature contribution of each sample. findExposures does this by using the least squares optimization method with constraints to keep the output as non-negative with the lsei function from limSolve. The CNSigs package contains pre-generated signatures derived from TCGA such as CancerSigs (set of 25 signatures) and collapsedSigs (set of 13 signatures). You can save your own fixedSigs from a previous run with createSigs then using the testResults$sigs as fixedSigs for a new run with a different sample set.

```
sigs = findExposures(scm, fixedSigs)
```

3. Running the Full Pipeline

The descriptions above walked you through how to analyze your copy number data in a step by step manner. The CNSigs package also allows you to do all of the above with one command using the runPipeline command. The runPipeline function follows the flowchart layed out in the picture below. This function allows you to specify almost all of the parameters for the sub functions that were described above. For a full list of the parameters refer to the function documentation using ?runPipeline. We would recommend saving the results of the pipeline to an object so that you can investigate them further using subsequent functions. If you forget to do so, but you specified the saveRes parameter to be T, then you can simply read in the 'Pipeline Results.rds' and that will give you the same results data structure.

```
results = runPipeline(segDataExp)
```



3.1. Using Fixed components or signatures

There are many use cases where you may want to run the analysis pipeline using a fixed set of signatures or components. The CNSigs package easily allows the user to fix either of these parameters

Fixed components

Fixing the signature components allows you to easily compare signatures across datasets since it ensures that the underlying signature components are pointing at the same feature distributions. Also, if the dataset you are analyzing is smaller, there may not be enough information to get generalized components. If you are analyzing a set of cancer samples you may want to use the components included in the package that were derived from the entire TCGA landscape. These components are representative of all cancer types and allow you to easily and accurately compare signatures derived from different cancer samples. Since it skips the modelling process, it also allows you to more robustly investigate smaller datasets. In order to use the derived cancer components, you just have to pass in the cancerComps object from the CNSig package as seen below.

```
results = runPipeline(segData, components = cancerComps)
```

Fixed signatures

You may also want to run the pipeline with not only fixed components but also fixed signatures. If you have already derived signatures for a dataset, and you simply want to look at

which of those signatures can be seen in a new set of samples, fixing the signatures is the way to do this. This sort of analysis is similar to what is done when looking at mutational signatures. Since the mutational signatures have already been defined, you often want to see which mutational signatures are present in your samples. In order to use fixed signatures in the CNSigs pipeline, you have to both specify the components that made the signatures and the signatures themselves. The easiest way to do this is to load up the results object from a previous run and grab the components and signatures from there. This sort of analysis is seen below. Just replace the 'resultsPath' with the path to the results of the previous run. The package includes a set of 25 unique signatures that were found by looking at all 33 cancer types in the TCGA dataset.

```
#referenceExp = readRDS("resultsPath/Pipeline results.rds")
newResults = runPipeline(segData, components = referenceExp$CN_components,
                         fixedSigs = referenceExp$sigs)
```

3.2. Ploidy Data

In the case where you are using fixed signatures and components that utalize ploidy data you can add your segment ploidy data into the pipeline function as ploidyData. The pipeline requires input ploidyData if your fixed signatures were build with ploidy.

```
newResults = runPipeline(segData, components = cancerComps,
                         fixedSigs = collapsedSigs, ploidyData = segPloidy)
```

4. Analyzing your results

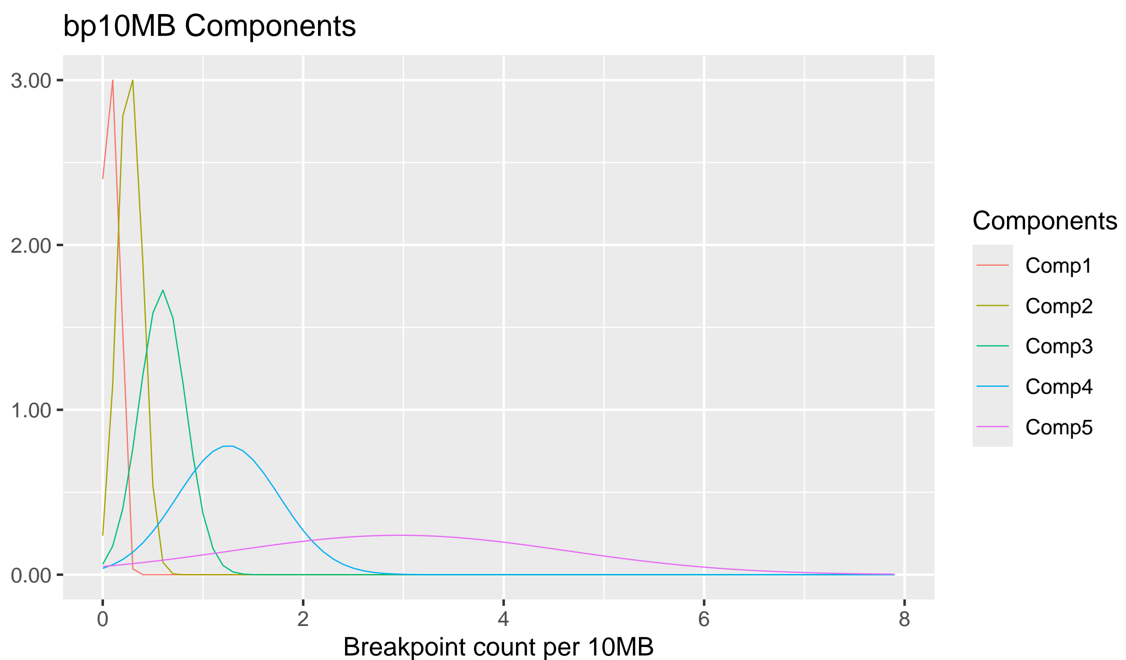
This package includes many functions that allow you to analyze the results that are given by the pipeline. Here we will describe those functions and why it may be useful for you to use them.

4.1. Plotting the components

One of the things that you may want to be able to do is to see what the components that were fitted to the extracted features look like. In order to do this, you can make use of plotComp and plotComps functions. The plotComps function simply calls plotComp for all of the different component types. plotComp looks at the parameters for the mixture model that was fit for the component that you specified. It then plots an approximation of those fitted distributions. This is useful if you want to compare two independently fit component sets. For instance, you may want to visualize how similar bp10MB2 is across two seperate datasets. By plotting these two components you are able to get a rough idea of how similar they are. Follow the code samples below in order to plot your components.

```
plotComp(cancerComps,"bp10MB") #Only plots the bp10MB component

## Warning:  ‘aes_string()‘ was deprecated in ggplot2 3.0.0.
## i Please use tidy evaluation idioms with ‘aes()‘.
```

```
## i See also `vignette("ggplot2-in-packages")` for more information.
## i The deprecated feature was likely used in the CNSigs package.
##   Please report the issue to the authors.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
## Warning:  Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## i The deprecated feature was likely used in the CNSigs package.
##   Please report the issue to the authors.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```




```
plotComps(cancerComps) #Plots all of the components
```

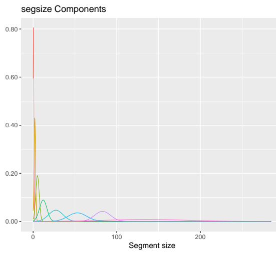


Figure 3: Segment Size

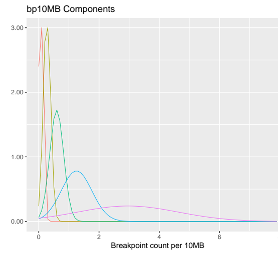


Figure 4: Breakpoints per 10MB

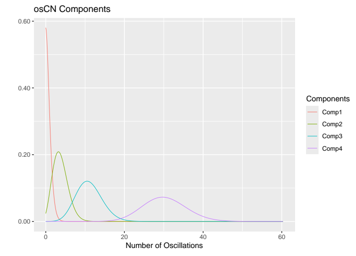


Figure 5: Number of Oscillations

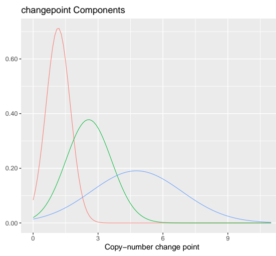


Figure 6: Copy Number Change-point

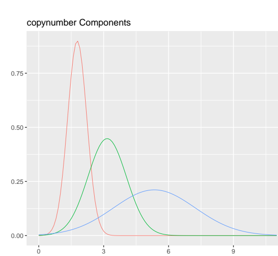


Figure 7: Copynumber

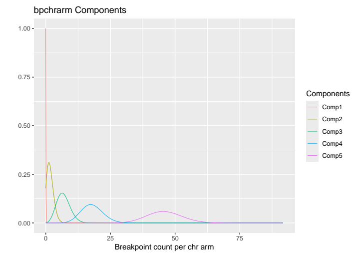


Figure 8: Breakpoints per chromosome arm

4.2. Determining signature similarity

Another useful thing that you may want to do is to estimate the similarity between two signatures. There are multiple ways of looking at the similarity of signatures.

Same components, similar signatures

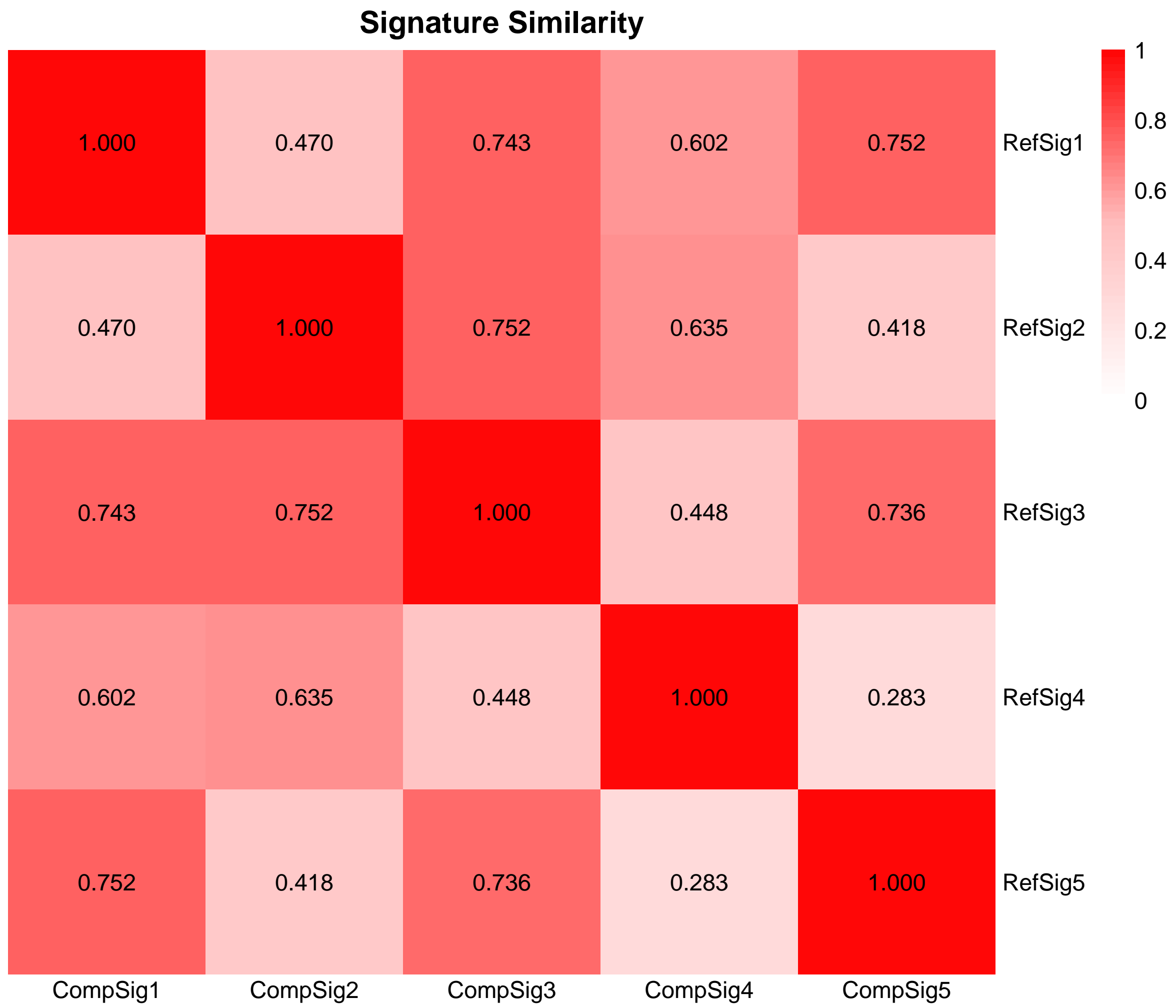
If you believe that your two sets of signatures should be nearly identical, then you can use the matchSigs function. This function finds the best match for each signature between the reference and comparison set. This can be useful if you have two datasets that are made up of similar samples, and you believe should produce similar signatures.

```
matchSigs(referenceExp$sigs,referenceExp$sigs)

## Average signature similarity:  1

##   TestSig ValSig CorVal
## 1       1      1      1
## 2       2      2      1
## 3       3      3      1
## 4       4      4      1
## 5       5      5      1
```

Same components, different signatures

If you don't expect your signatures to be the same, but you used the same components to derive both signature sets (ie. cancerComps), then you can use the sigSim function. This function compares each signature in the reference set to every other signature in the comparison set. It then plots a heatmap that shows the similarity of the signatures. (NOTE: The output of this function is useful iff the components in both signature sets are identical)

```
sigSim(referenceExp,referenceExp)
```




```
##         CompSig1 CompSig2 CompSig3 CompSig4 CompSig5
## RefSig1 1.000000 0.469835 0.743187 0.602059 0.752086
## RefSig2 0.469835 1.000000 0.752032 0.634783 0.418413
## RefSig3 0.743187 0.752032 1.000000 0.447967 0.735859
## RefSig4 0.602059 0.634783 0.447967 1.000000 0.282534
## RefSig5 0.752086 0.418413 0.735859 0.282534 1.000000
```

Different Components

If you let the pipeline fit the models independently, then the sigSim function recognizes that the the underlying components are no longer the same and don't point to the same data. In this case, we utilize a method to estimate the signature similarities. We first attempt to recreat the underlying feature distributions that made up a signature by creating a distribution of simulated values vased on the feature models and the component weights in the signatures. The two simulated feature distributions are then compared using the ks statistical test. This gives a similarity per component and then each component similarity is averaged to give an overall signature similarity.

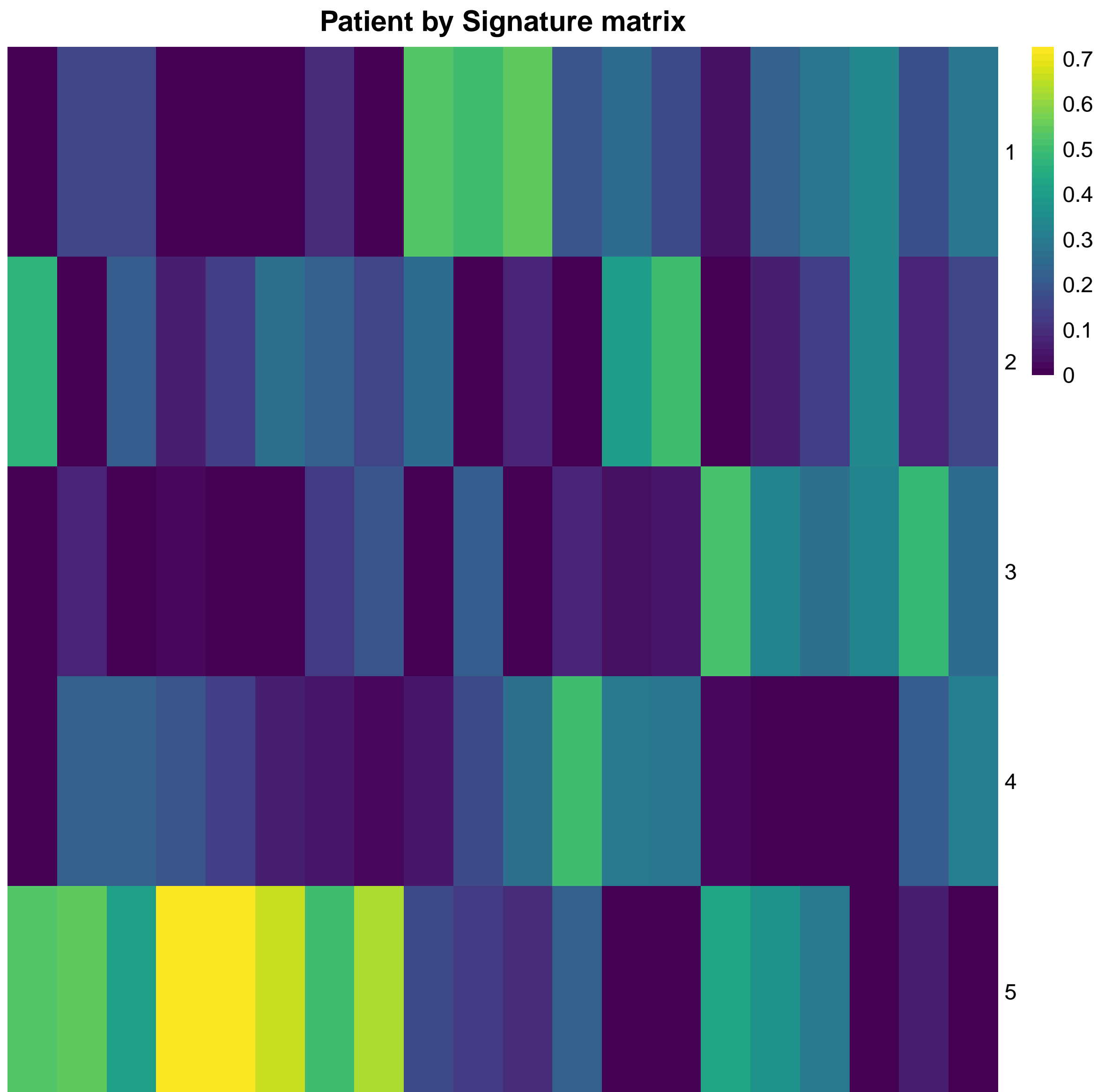
4.3. Plotting Signature Exposure

It is very useful to visualize the exposure of your samples to the signatures. The pipeline generates the plot automatically, but the package includes two functions that allow you to do this yourself as well.

Signature Exposure Matrix Visualization

This heatmap based visualization is the default visualization method used by the pipeline. Using this function, you are able to specify the ordering of both the rows and columns of the plot. The ordering parameters can either allow you to order the values using heirarchical clustering or with a user specified order for the data.

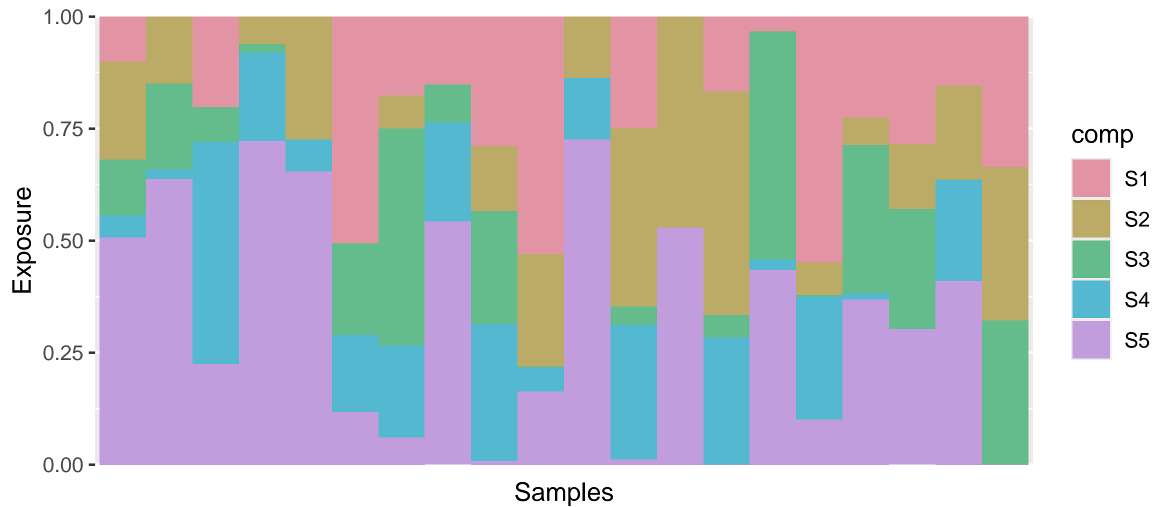
```
plotSigExposureMat(referenceExp$sigExposure)
```



Signature Exposure Stacked Bar Visualization

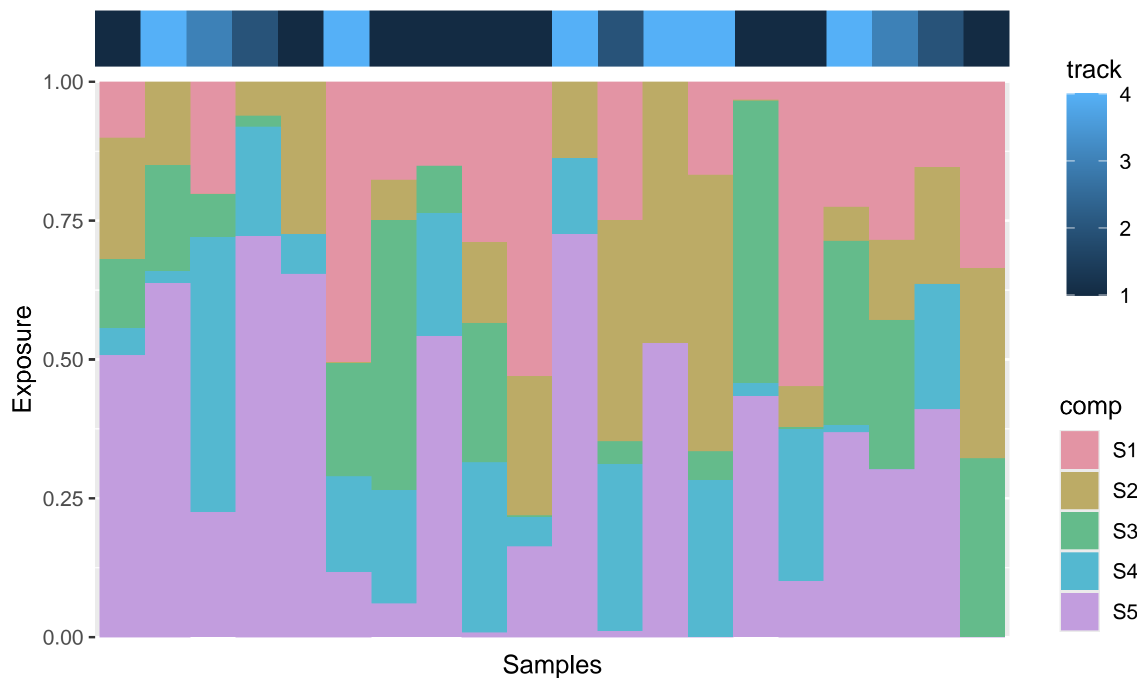
Another way of visualizing the signature exposures is via a stacked bar plot. The package has a function called plotSigExposure that allows you to create this different visualization. One of the best features this function allows for is the inclusion of additional data to be displayed as tracks along the plot. You can also specify a specific set of sorting instructions to allow you to sort the exposure plot by both the exposures and also the trackData.

```
plotSigExposure(referenceExp$sigExposure)
```

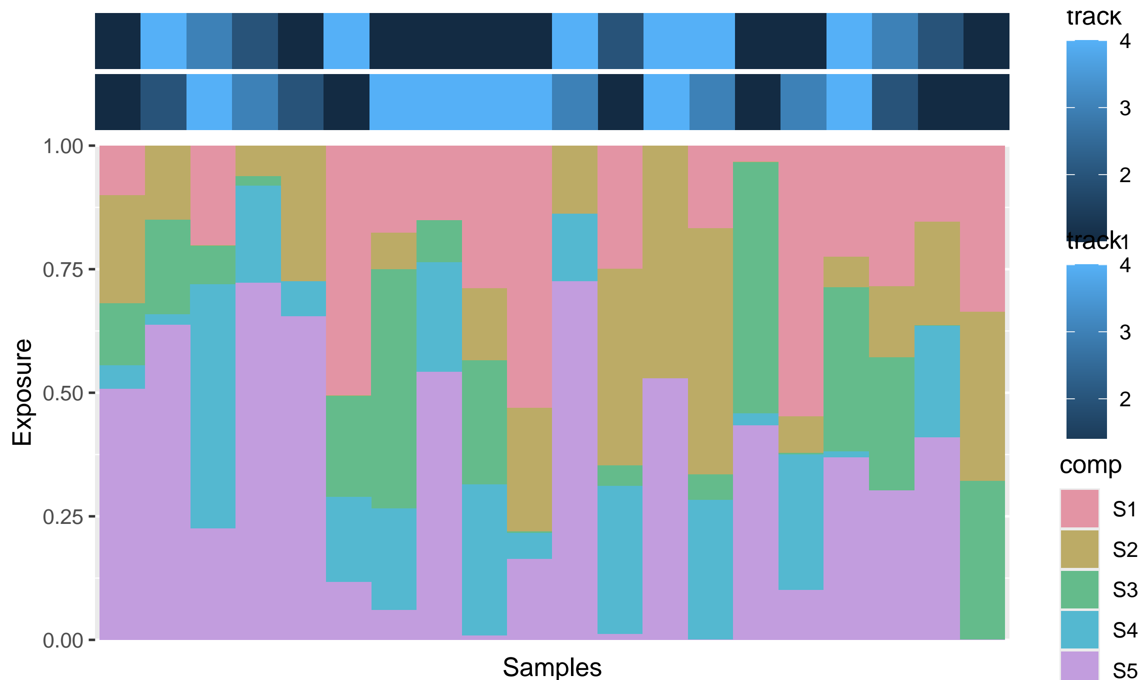


This function allows you to add on data as tracks. If you only want to add a single track, you can pass in a vector of data in the same order as the samples in the sigExposure matrix using the trackData parameter. You can plot up to 3 tracks by passing them in as a list.

```
sigExposure = referenceExp$sigExposure

# Generate random data to represent track data
sampleTrackData = sample(c(1,2,3,4),ncol(sigExposure),T)
sampleTrackData2 = sample(c(1,2,3,4),ncol(sigExposure),T)

# Plot with a single track
plotSigExposure(sigExposure,trackData = sampleTrackData)
```

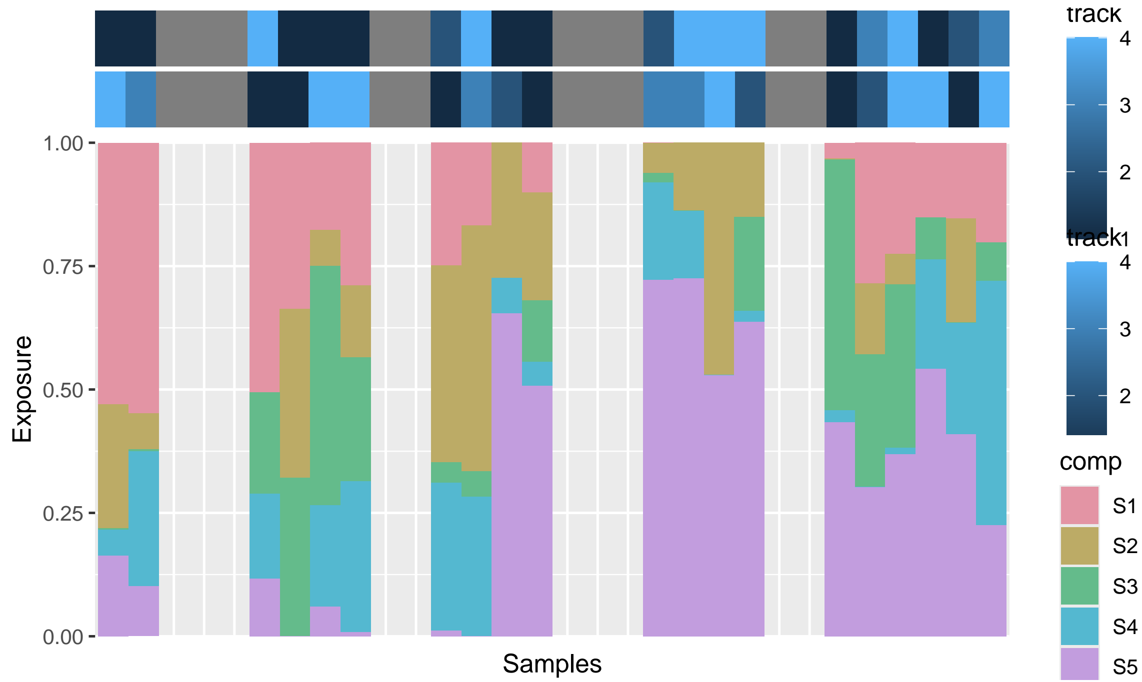


```
# Plot multiple tracks.
plotSigExposure(sigExposure, trackData = list(sampleTrackData,sampleTrackData2))
```

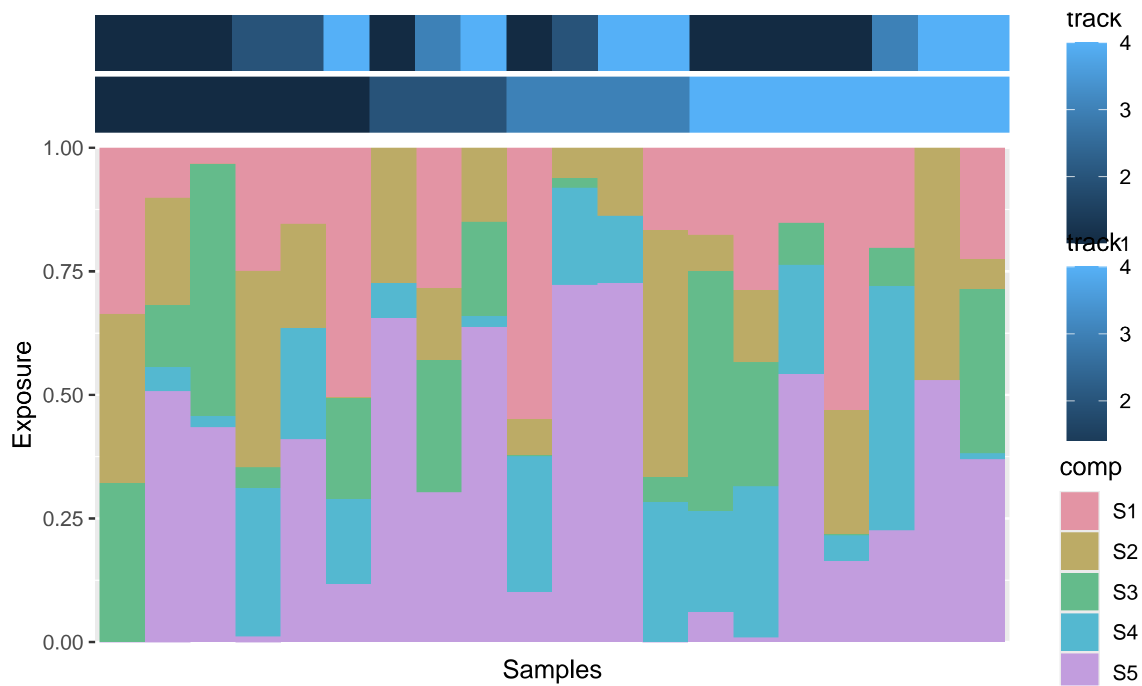


Using this function you are able to sort the plot in many different ways by using the sortOrder parameter. When you give the function a set of trackData, it allows you to begin to specify the sortOrder. This allows your to sort the main plot in a different order. "m" represents the main plot, and "t" followed by the number of the track (ie: "t1","t2" ...) represents the tracks. By chaining the values together you can specify a variety of ways to sort the final plot. As an example, the sortOrder of "mt1t2" specifies the the plot should be sorted by the signature exposures first followed by the first track and finally the second track. In another example, the sortOrder of "t2mt1" specifies the plot to be sorted by track number 2 first followed by the signature exposures and lastly by track number 1.

```
# Plot multiple tracks sorted by the main plot and then the first track
plotSigExposure(sigExposure, trackData = list(sampleTrackData,sampleTrackData2),
                sort=T,sortOrder = "mt1")

## Warning:  Removed 50 rows containing missing values or values outside the scale
range
## ('geom_bar()').
## Removed 50 rows containing missing values or values outside the scale range
## ('geom_bar()').
```




```
# Plot multiple tracks sorted by the second track and then the first track
plotSigExposure(sigExposure, trackData = list(sampleTrackData,sampleTrackData2),
                sort=T,sortOrder = "t2t1m")
```



5. Additional Options

Most of the functions within this pipeline have additional parameters that allow for further tuning of the copy number signatures pipeline. For more information on and examples of these parameters, refer to the package manual, or to the individual function documentation using R's builtin ? notation (?functionName).

5.1. Parallelization

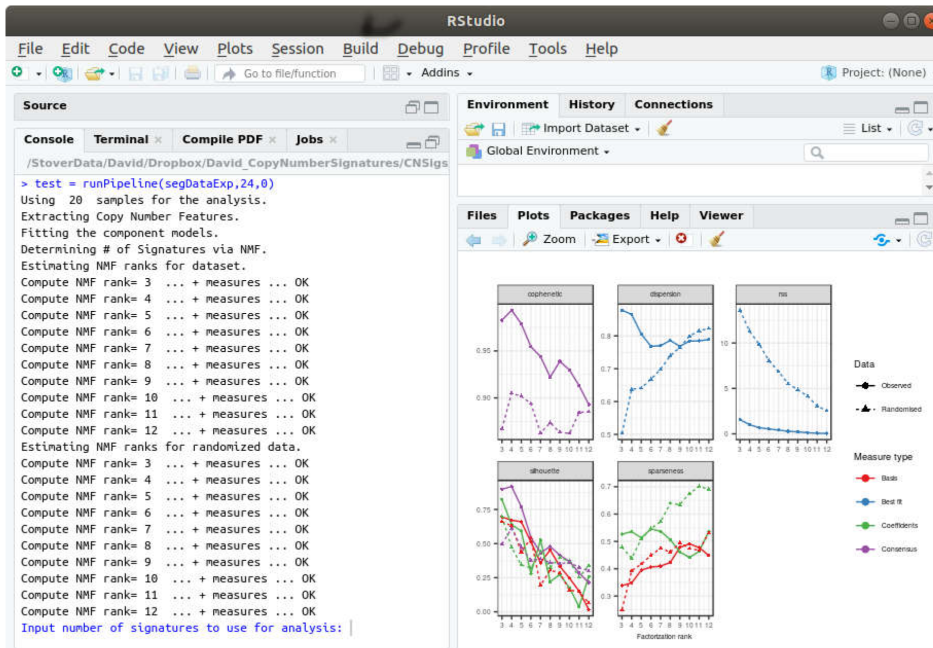
A large majority of the functions within this package are able to be run in parallel. Using the cores parameter, you can specify the number of workers for the functions to use. Both the extractCNFeats and fitModels can use a maximum of 6 workers since each worker gets assigned one feature, of which there are 6. This will help to speed up many of the methods, especially the NMF steps within the createSigs function. It is worth noting that having a lot of concurrent workers can cause a spike in memory usage.

5.2. Determining Number of Signatures

The package has a function that is useful for determining the optimal number of signatures for a given dataset. This function is called determineNumSigs, and there are three ways to use this function. This function, by default, looks from 3 to 12 signatures, but this can be changed by using the rmin and rmax parameters.

Within a runPipeline call

If you either don't give a value for nsig during a call to runPipeline, or if you specify nsig to equal 0, the pipeline will automatically run the determineNumSigs function. After it finishes the determineNumSigs step, it will wait for user input to specify a number of signatures, and will continue to run the rest of the pipeline. If you are saving the results, the plot will be saved in the results folder, otherwise it should pop up for you to look at and pick a value. As you can see in the example below, after it checks ranks 3-12 it waits for user input before proceeding.



Directly using determineNumSigs function

The determineNumSigs function can be used directly if you have already extracted the features and fit the models. You can pass both the features and components to the function, then it will search from rmin to rmax and output the plot.

```
smooth = smoothSegs(segDataExp)
feats = extractCNFeats(smooth)
comps = fitModels(feats)
determineNumSigs(feats,comps)
```

Using detSigNumPipeline function

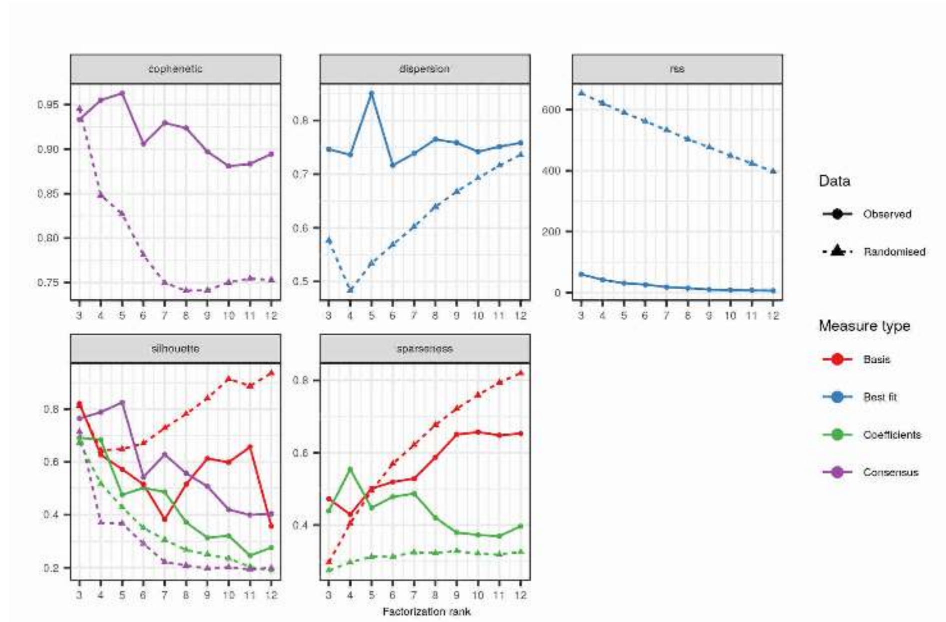
If you mainly want to determine the number of signatures to get a better idea about your data, you can use the detSigNumPipeline function. This function executes the first few steps of the full pipeline all at once and stops after running the determineNumSigs function.

```
detSigNumPipeline(segDataExp,smooth=T)
```

5.3. Interpreting determineNumSigs plot

The plot generated by the methods described above displays a variety of metrics that describes the output of the NMF. The plot compares the NMF output from the real data, to the NMF output of randomly scrambled data. In order to determine the best number of signatures, it is useful to look at many of these metrics in order to make a decision. Following the descriptions of the measures below, you can pick the value for nsigs that gives you the best results. All of these measures are calculated using the input data (solid line) and a randomized permutation

of the input data (dotted line). For some measures, such as sparseness, it is useful to compare the value for the real data compared to the randomized data.



Cophenetic

Also known as the cophenetic similarity or cophenetic distance, this measure describes how similar two objects have to be in order to be grouped into the same cluster. In the context of copy number signatures, it describes how similar two samples' copy number profiles need to be in order to be classified as the same signature. You ideally want to have higher values for the cophenetic. You can imagine that if you have a low cophenetic distance, then two samples that aren't very similar could be classified within thes same signature.

Dispersion

The dispersion coefficient is a measure that is derived from the NMF consensus matrix. The consensus matrix can be described as the average connectivity matrix of multiple NMF runs. Each value in the consensus matrix essentially describes the probability that two samples belong to the cluster across the nrun NMF runs. In our package, nrun is set to 250 to ensure a stable consensus matrix. For a perfect consensus matrix, meaning all of the connectivity matrices across all the NMF runs were the same, the dispersion value is 1. A lower dispersion value means that the clusters were less reproducible across the NMF runs.

Rss

Rss stands for the residual sum of squares of the NMF model. This is a measure of how different the final estimation is from the input data. NMF is a method designed to estimate a factorization of the input matrix. So the rss, describes how closesly the factorization matches the input data. You want a low value for the rss, however, since many of the NMF algorithms are designed to minimize the rss, this measure usually does not have much effect on the decision for the number of signatures since values across all values of nsig generally have a

low rss value.

Silhouette

Silhouette is a measure that looks at the consistency of the clusters and measures how well each sample has been classified. For each measure it looks at how similar the sample is to others in the same cluster compared to other clusters. A single silhouette value for a sample ranges from -1 to 1. A lower value means that the sample is more similar to samples from other clusters than it is to samples within its cluster. The mean silhouette across all samples within a cluster measures how tightly grouped the samples are within that cluster. Therefore the mean silhouette of all data across the entire dataset is a measure of how appropriately the samples are clustered.

Sparseness

The sparseness of a vector is defined by how much energy of the vector is found in a small number of the components. In theory, the most sparse vector, which only has one non-zero value, would have a sparseness of 1. The least sparse vector, in which all components have an equal value, would have a sparseness of 0. For a matrix, a sparseness measure is simply an average sparseness of the column vectors. The sparseness of the basis matrix describes how many components drive the signatures. A more sparse basis matrix means that the signatures are composed of fewer components. The sparseness of the coefficients matrix describes how many signatures each sample belongs to. A more sparse coefficients matrix means that each sample is made up of fewer signatures

5.4. Remapping the Pipeline Results

There are times where you may want to either reorder the signatures that were found during an analysis or maybe just give the signatures names. The package has a function called remapResults to help you do this. You give it the path of an entire results folder and then either a numeric mapping for the signatures or a set of names for the signatures. The function will generate a new results folder where all of the outputs are rearranged according to the new mapping and regenerating all of the plots in the new order.

5.5. Subsetting the features used

Many of the functions in the pipeline have a optional parameter in them to specify which features to use. By utilizing this function you are able to generate copy number signatures using any subset of the features that you want. For instance, if you dont want to include copy number oscillations in your final signatures you can remove this feature from the list of features to use and it will not be used. There is a vector included in the package called defaultFeats that includes the names of the default features. It is useful to use this vector and simply select the features you wish to use and pass this vector into the functions.

```
defaultFeats

## [1] "segsize"     "bp10MB"      "osCN"        "changepoint" "copynumber"
## [6] "bpchrarm"
```

```
desiredFeats = defaultFeats[-3]
desiredFeats

## [1] "segsize"     "bp10MB"      "changepoint" "copynumber"   "bpchrarm"
```

```
results = runPipeline(segDataExp,featsToUse = desiredFeats)
```

5.6. Different Genome Builds

One of the important factors for the package to work properly is to ensure that you are specifying the genome build correctly. The package uses both the lengths of the chromosomes and the positions of the centromeres when extracting some of the copy number features. Since these values change from one build to another it is important to specify the genome build correctly. Be default, the genome build is assumed to be hg19. Currently the package can support hg18, hg19, and hg38. Both the runPipeline and the extractCNFeats have a parameter called gbuild that allows you to specify the genome build.

Affiliation:

David Tallman
The Ohio State University
512 BRT 460 W 12th Ave,
Columbus, OH 43210
E-mail: tallman.52@osu.edu
URL: https://u.osu.edu/stoverlab/