OmicNavigator User's Guide

John Blischak

December 16, 2025 OmicNavigator 1.19.0

Contents

1 Overview

The goal of the OmicNavigator software is to facilitate the interactive exploration of the results from an omics experiment. Using OmicNavigator, any bioinformatician familiar with the basics of the R programming language can create and share a high-quality, comprehensive dashboard for interactive investigation of the patterns and signals in the data.

The steps include:

1. The bioinformatician analyzes the data from the omics experiment. This can be done in R or any other platform (e.g. Array Studio).
2. The bioinformatician registers the results using the OmicNavigator R functions. If the results were produced outside of R, they will need to be exported to files, and then imported into R.
3. The bioinformatician uses OmicNavigator to export the analysis results to a study package and install it.

4. The bioinformatician can run the app locally or deploy it on a server.
5. If the app is deployed, collaborators can access the dashboard directly in their web browser.

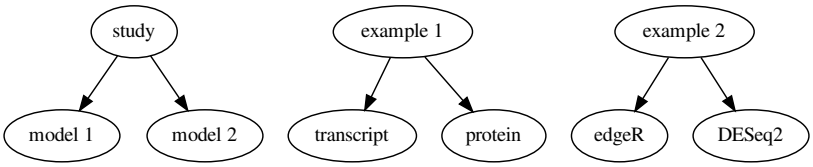
2 Structure of analysis results

Before you start registering your data with OmicNavigator, it will be helpful to have a high-level understanding of how OmicNavigator defines the main components of the analysis results and how they relate to each other.

2.1 A study and its models

The largest unit of organization is the study. This corresponds to all the experiments performed and analyses conducted in order to address a scientific question. A study should contain all the relevant results that someone would need to evaluate the scientific question. In more practical terms, any analyses that uses overlapping biological samples should be included in the same study (e.g. if you measured transcript and protein levels in the same samples).

A study has one or more models. The models can be very different (one model for transcript levels and another for protein levels) or very similar (same exact input data but differential expression performed with two different software packages).

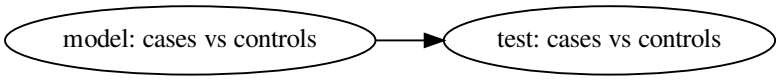


2.2 Differential expression results

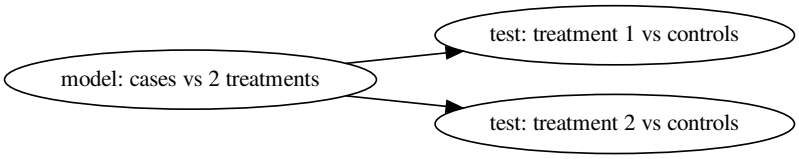
One of the primary type of results that OmicNavigator displays for interactive investigation is the statistical results from a differential expression analysis. Here "differential expression" is broadly defined, e.g. this could be differential methylation or any other molecular phenotype.

Each model has one or more tests. Each test (also commonly referred to as a contrast) refers to the statistical results from testing a single coefficient (or a combination of coefficients) from the model.

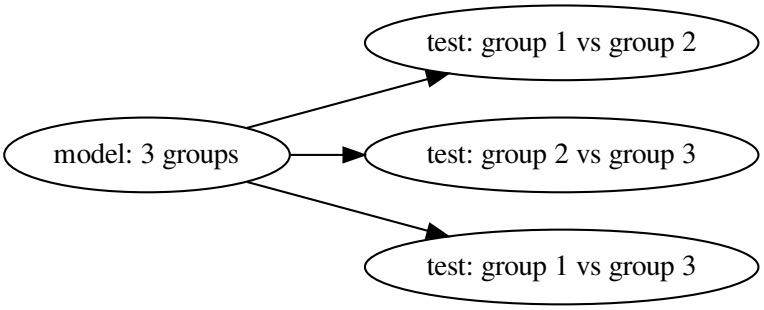
For example, a model that compares cases versus controls will only have one test:



A model that compares a control versus two different treatments will likely have two tests:



And a model that compares three distinct groups will likely have 3 tests (since $\binom{3}{2} = 3$ pairwise comparisons):

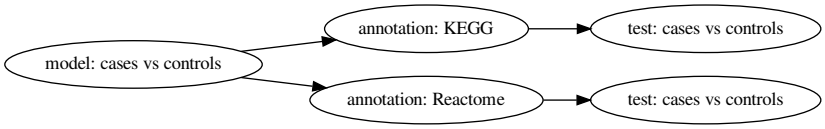


2.3 Enrichment analysis

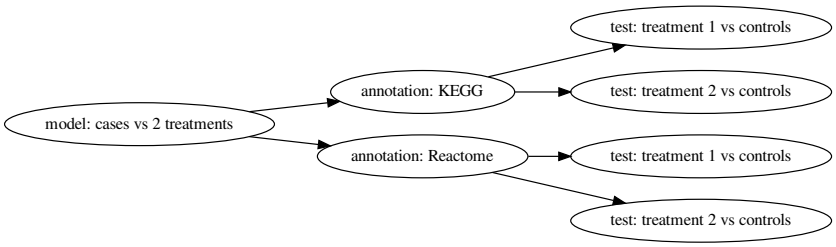
A typical systems biology analysis to perform after a differential expression analysis is to test for enrichment of the differentially expressed features in terms from curated annotation databases (e.g. KEGG, Reactome). The OmicNavigator app provides a table, network, and various other visualizations to explore the output of enrichment analyses.

Similar to the differential expression results, enrichment analyses are also added per model and per test. However, there is also the additional category of the annotation database that was used for the enrichment (e.g. KEGG, Reactome). Thus each enrichments table corresponds to a given study-annotation-test combination.

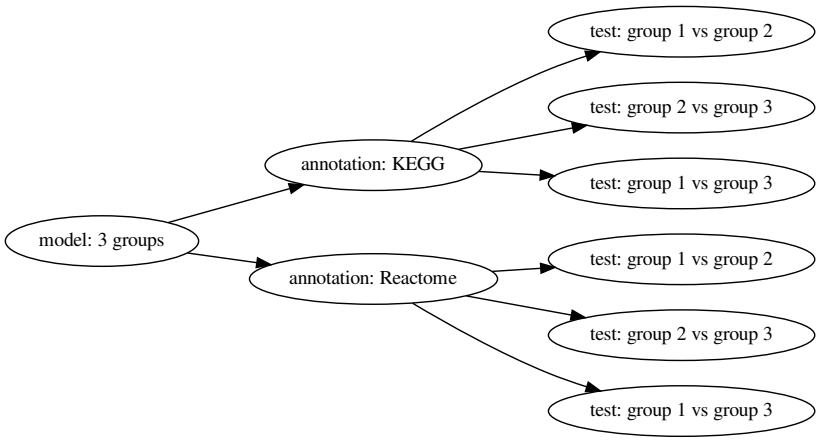
For example, imagine that enrichment analyses were performed with the KEGG and Reactome annotations. A model that compares cases versus controls will have two enrichments tables:



A model that compares a control versus two different treatments will have four enrichment tables, 2 for each of the tests:



And a model that compares three distinct groups will have 6 enrichment tables, 2 for each of the 3 tests:



2.4 Additional information

The primary data sets are the differential expression results and the enrichments results. However, the more information you supply about the experimental data, the more details will be included in the app.

For example, you can include metadata about the features, metadata about the samples, or the individual assay measurements (expression, methylation, phosphorylation, etc.).

This additional metadata can be added per model (e.g. different feature metadata for RNA versus protein measurements) or shared across models (e.g. if the same samples are used in each model).

3 Step-by-step example

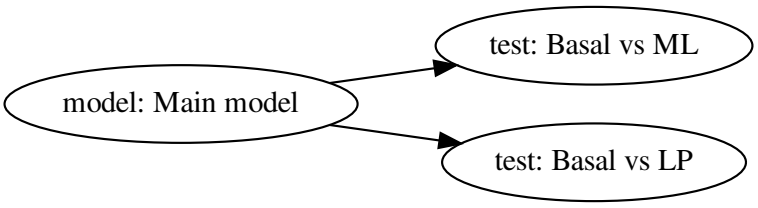
Now it's time to convert your results into an OmicNavigator study! Below you will see how to:

1. Create a new OmicNavigator study
2. Add your existing results to your OmicNavigator study
3. Install your OmicNavigator study as an R package
4. Start the app to interactively explore your results

3.1 Example data: RNAseq123

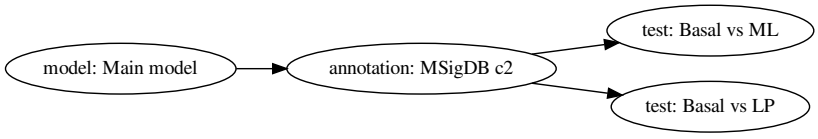
As an illustrative example, this vignette uses the results from the Bioconductor workflow package RNAseq123 [1]. It is a limma+voom analysis of RNA-seq data from 3 cell populations in the mouse mammary gland collected by Sheridan et al., 2015 [2].

The 3 cell populations are basal, luminal progenitor (LP), and mature luminal (ML). In the RNAseq123 analysis, the primary differential expression results obtained are for the tests "Basal versus LP" and "Basal versus ML". This is similar to the design diagrammed in Figure 2.2.



Furthermore they performed enrichment analysis with one annotation database: the

Broad Institute's MSigDB c2 collection, which was converted converted from human to mouse gene identifiers by WEHI Bioinformatics.



To obtain the results, I ran their script limmaWorkflow.R and saved some of the objects to use in this vignette.

```
data("RNAseq123", package = "OmicNavigator")
ls()
```

```
 [1] "Mm.c2"                     "annotations"
 [3] "barcodeData"               "basal.vs.lp"
 [5] "basal.vs.ml"               "cam.BasalvsLP"
 [7] "cam.BasalvsML"             "enrichmentsFavicons"
 [9] "enrichmentsIntersection"   "enrichmentsLinkouts"
[11] "enrichmentsNetwork"        "enrichmentsNetworkMinimal"
[13] "enrichmentsTable"          "enrichmentsUpset"
[15] "group"                     "lane"
[17] "lcpm"                      "linkFeatures"
[19] "metaFeaturesLinkouts"      "metaFeaturesTable"
[21] "modelID"                   "models"
[23] "nodeFeatures"              "plots"
[25] "reportLink"                "resultsFavicons"
[27] "resultsIntersection"       "resultsLinkouts"
[29] "resultsTable"              "resultsTableTerm"
[31] "resultsUpset"              "samplenames"
[33] "studies"                   "studyMeta"
[35] "testID"                    "tests"
[37] "upsetCols"
```

Furthermore, after the analysis completed using the full data, I drastically subset the results objects to create a minimal example. For example, it only contains differential

expression results for 24 features and enrichment analysis results for 4 annotation terms.

You are welcome to follow along with these example data sets if you like. However, the main goal of this document is for you to input your own study data into OmicNavigator. The particular details of this specific study are not important. It happens to be an RNA-seq analysis of mice that was analyzed with limma+voom, but none of that has to apply to your own study.

3.2 Create an OmicNavigator study

The first step is to load the OmicNavigator package into the current R session.[1]

```
library(OmicNavigator)
```

Next run `createStudy()` to create your OmicNavigator study object. The first argument is the name of the study. The second argument (optional) is a description of the study.

```
study <- createStudy(name = "vignetteExample",
                     description = "Bioc workflow package RNAseq123")
```

Because the `name` will be used when naming the study package, it must follow these rules that apply to all R package names:

- Begin with a letter
- End with a letter or a number
- Be at least two characters long
- Only contain alphanumeric characters and periods (full stops)

For more details, run `?createStudy`. For example, there are additional optional arguments such as `version`, `maintainer`, and `maintainerEmail`.

3.3 Results

The RNAseq123 differential results were generated by the limma function `topTreat()`. The results for the test "Basal versus LP" are in `basal.vs.lp` and the results for "Basal versus ML" are in `basal.vs.ml`.

[1] If you haven't installed OmicNavigator yet, please see the file `README.md` for the installation instructions.

```
head(basal.vs.lp)
```

```
      ENTREZID SYMBOL TXCHROM     logFC   AveExpr         t
19216    19216 Ptger1    chr8 -2.401062  4.263219 -8.911887
57811    57811    Rgr   chr14 -5.048202 -1.579505 -7.777238
22068    22068  Trpc6    chr9 -4.202937  3.413562 -7.506933
12767    12767  Cxcr4    chr1 -3.631749  5.049865 -6.255800
16439    16439  Itpr2    chr6 -1.957691  6.412740 -5.533072
20866    20866  Stim1    chr7  1.842699  6.663871  5.377393
           P.Value    adj.P.Val
19216 7.535233e-06 0.0001299437
57811 2.197986e-05 0.0002819392
22068 2.797665e-05 0.0003377516
12767 1.031838e-04 0.0009651739
16439 2.368098e-04 0.0018990476
20866 2.866392e-04 0.0022225231
```

The first column contains the Entrez gene identifier for each gene that they tested. OmicNavigator refers to this primary feature identifier of the study as the featureID.[2] It must be unique, i.e. there must be only one row of results per featureID. The second and third columns contain more information about the features: the gene symbol and chromosomal location, respectively. OmicNavigator stores feature metadata columns in a separate table, so these will be removed from the results table. The remaining columns contain the quantitative measurements from the statistical test.

This table is almost ready for import into OmicNavigator. There are only two strict requirements for a results table:

- The first column must be a unique character vector containing the featureID
- The remaining columns must be numeric vectors

Thus I only need to remove the second and third columns which include extra feature metadata columns. These will be added separately as the features table in Section 3.7.

```
# Remove columns 2 and 3
basal.vs.lp.on <- basal.vs.lp[, -2:-3]
```

[2]This example study happened to use Entrez gene identifiers for the featureID. You can use whichever featureID you like for your own study. OmicNavigator only requires that the featureID is unique per feature, a character vector, and used consistently between the various input tables.

```
basal.vs.ml.on <- basal.vs.ml[, -2:-3]
head(basal.vs.ml.on)
```

```
      ENTREZID     logFC   AveExpr          t      P.Value
22068    22068 -6.361321  3.413562 -12.828451 4.675483e-07
19216    19216 -2.323077  4.263219  -8.459881 1.119083e-05
21390    21390  2.415695 -1.790873   3.339258 4.828374e-03
22065    22065  2.538410 -3.098852   3.192573 6.057918e-03
16440    16440 -1.393191  7.836093  -3.019645 7.878121e-03
57811    57811 -2.656060 -1.579505  -3.024537 7.882325e-03
         adj.P.Val
22068 1.774549e-05
19216 1.549012e-04
21390 2.323210e-02
22065 2.839212e-02
16440 3.577325e-02
57811 3.578257e-02
```

Tip: Order the results table by statistical significance. The initial display of the results table in the app will be exactly as you enter the table in R. The users can of course manually sort the table by any of the columns, but it is convenient for the initial display to highlight the most statistically significant features. I omitted this ordering step above because `topTreat()` automatically sorts the features by statistical significance.

Next I combine these two results tables. Recall from Figure 3.1 that the results are defined for a given model and test combination. To represent this hierarchical relationship, OmicNavigator uses nested lists. Below I create a new list named `results`. Its first element is the name of the model, `main`, referred to as the `modelID`. Then I assign `main` a list where each element is the name of a test from that model, referred to as the `testID`. Each element contains the result table for that test.

```
results <- list(
  main = list(
    basal.vs.lp = basal.vs.lp.on,
    basal.vs.ml = basal.vs.ml.on
  )
)
```

I add this to the OmicNavigator study with `addResults()`.

```
study <- addResults(study, results)
```

For more details, run `?addResults`.

> **Important:** You can stop here! With only a single results table, you can already start using the app to interactively investigate your study. Any additional data you add enables more features in the app for further exploration. See Section 6.4 for the mapping between data elements and app features.

3.4 Enrichments

The RNAseq123 enrichments were generated by the limma function `camera()`. The results for the test "Basal versus LP" are in `cam.BasalvsLP` and the results for "Basal versus ML" are in `cam.BasalvsML`.

```
head(cam.BasalvsLP)
```

```
                                        NGenes Direction    PValue
REACTOME_ELEVATION_OF_CYTOSOLIC_CA2_LEVELS      8      Down 0.2761301
REACTOME_BINDING_AND_ENTRY_OF_HIV_VIRION        4      Down 0.6700697
REACTOME_PROSTANOID_LIGAND_RECEPTORS            7      Down 0.6978828
REACTOME_OPSINS                                 5      Down 0.7317891
                                              FDR
REACTOME_ELEVATION_OF_CYTOSOLIC_CA2_LEVELS 0.5914291
REACTOME_BINDING_AND_ENTRY_OF_HIV_VIRION   0.8722180
REACTOME_PROSTANOID_LIGAND_RECEPTORS       0.8871448
REACTOME_OPSINS                            0.9006782
```

The row names contain the names of the annotation terms used in the enrichment analysis. OmicNavigator refers to these as the `termID`. The columns contain the number of genes in each term (`NGenes`), the direction of the enrichment (`Direction`), the nominal p-value (`PValue`), and the multiple-testing adjusted p-value (`FDR`).

Unlike the results table, OmicNavigator has strict requirements for the names and contents of the enrichments table columns:

1. `termID` - the unique identifier for each term
2. `description` - a human readable description of each term
3. `nominal` - the nominal statistical result from the enrichment test

4. `adjusted` - the statistical result adjusted for multiple testing

The object returned by `camera()` contains all the required information except the human readable description. Below I create new data frames to store the enrichments tables. I perform some string processing of the `termID` to create the `description`. If creating a human readable description is too onerous, you can simply repeat the `termID` for this column.

```
# LP
cam.BasalvsLP.on <- data.frame(
  termID = row.names(cam.BasalvsLP),
  description = gsub("_", " ", tolower(row.names(cam.BasalvsLP))),
  nominal = cam.BasalvsLP$PValue,
  adjusted = cam.BasalvsLP$FDR,
  stringsAsFactors = FALSE
)
# ML
cam.BasalvsML.on <- data.frame(
  termID = row.names(cam.BasalvsML),
  description = gsub("_", " ", tolower(row.names(cam.BasalvsML))),
  nominal = cam.BasalvsML$PValue,
  adjusted = cam.BasalvsML$FDR,
  stringsAsFactors = FALSE
)
head(cam.BasalvsML.on)
```

```
                                     termID
1 REACTOME_ELEVATION_OF_CYTOSOLIC_CA2_LEVELS
2       REACTOME_PROSTANOID_LIGAND_RECEPTORS
3   REACTOME_BINDING_AND_ENTRY_OF_HIV_VIRION
4                            REACTOME_OPSINS
                                 description   nominal  adjusted
1 reactome elevation of cytosolic ca2 levels 0.4119939 0.6745223
2       reactome prostanoid ligand receptors 0.4630303 0.7124122
3   reactome binding and entry of hiv virion 0.6426407 0.8298654
4                            reactome opsins 0.6629803 0.8411875
```

Lastly I need to combine these two enrichments table. Recall from Figure 3.1 that the enrichments are defined for a given model, annotation, and test combination. Again a nested list is used to represent this hierarchical relationship. The nested list defined

below, `enrichments`, contains 3 levels: 1) modelID, 2) annotationID, 3) testID.

```
enrichments <- list(
  main = list(
    c2 = list(
       basal.vs.lp = cam.BasalvsLP.on,
       basal.vs.ml = cam.BasalvsML.on
    )
  )
)
```

I add this to the OmicNavigator study with `addEnrichments()`.

```
study <- addEnrichments(study, enrichments)
```

For more details, run `?addEnrichments`.

3.5 Models

You can provide more detailed descriptions of your models. These will be displayed in the app when a user hovers over each `modelID`. Below I describe the only model I currently have for this study.

```
models <- list(
  main = "limma+voom model of RNA-seq experiment of mouse mammary glands"
)
```

And then add it to the OmicNavigator study with `addModels`.

```
study <- addModels(study, models)
```

For more details, run `?addModels`. Importantly, note that instead of a only adding a single string as above, you could alternatively add a named list with additional metadata to describe each modelID.

3.6 Tests

You can provide more detailed descriptions of your tests. These will be displayed in the app when a user hovers over each `testID`. Below I describe the two tests that were performed for the modelID `main`. The input must be a nested list, similar to the input to `addResults()`. Each element of the top-level list is a modelID, and

each element of the nested list is a testID. The value is a single character string that describes the test.

```
tests <- list(
  main = list(
    basal.vs.lp = "Which genes are DE between Basal and LP cells?",
    basal.vs.ml = "Which genes are DE between Basal and ML cells?"
  )
)
```

I then add this to the OmicNavigator study with `addTests`.

```
study <- addTests(study, tests)
```

For more details, run `?addTests`. Importantly, note that instead of a only adding a single string as above, you could alternatively add a named list with additional metadata to describe each testID.

3.7 Features

Recall that I removed the feature metadata columns from the results tables. I still want to include these when exploring the results in the app. I can add them in the features table of OmicNavigator. The features table has the following requirements:

- The first column must contain the study featureID. It must be unique, and it must be a character vector. The row order doesn't have to match the order in the results table(s).
- The remaining columns must all be character vectors.

Thus I can take the first 3 columns of one of the objects returned by `topTreat()` to use as the features table for the modelID `main`.

```
basal.vs.lp[1:2, 1:6]
```

```
      ENTREZID SYMBOL TXCHROM     logFC   AveExpr         t
19216    19216 Ptger1    chr8 -2.401062  4.263219 -8.911887
57811    57811    Rgr   chr14 -5.048202 -1.579505 -7.777238
```

```
features <- list(
  main = basal.vs.lp[, 1:3]
)
```

And add it to the OmicNavigator study with `addFeatures()`.

```
study <- addFeatures(study, features)
```

For more details, run `?addFeatures`.

Tip: Be judicious in the number of columns you add to the features table. These will be displayed in the app next to the differential expression results. If you have too many features columns, they will obscure from the columns with the statistics.

3.8 Annotations

In addition to a table of enrichments, the app can also display a network view of the enrichment results. In order to do this, OmicNavigator needs more information about the annotation terms that were used. Each node of the network is a `termID` and the edges between the nodes are determined by the number of shared features between any two of the `termID`'s. The more shared features, the thicker the edge line will be.[3]

The input format for the terms is a named list of character vectors. The names of the list are the `termID`'s. Each element is a character vector that contains the features in that term. Conveniently the c2 terms are already in this format.

```
Mm.c2
```

```
$REACTOME_ELEVATION_OF_CYTOSOLIC_CA2_LEVELS
[1] "109305" "16438"  "16439"  "16440"  "18436"  "20866"  "22065"
[8] "22068"  "26946"

$REACTOME_PROSTANOID_LIGAND_RECEPTORS
[1] "14764" "19214" "19216" "19217" "19218" "19219" "19220" "19222"
[9] "21390"

$REACTOME_OPSINS
[1] "12057"  "13603"  "14539"  "20132"  "212541" "30044"  "353344"
[8] "57811"

$REACTOME_BINDING_AND_ENTRY_OF_HIV_VIRION
[1] "105675" "12504"  "12767"  "12772"  "12774"  "19034"  "268373"
[8] "382096"
```

[3] The user of the app is able to choose which overlap metric to use when drawing the thickness of the edge lines.

Recall that I purposefully subset the data to provide a minimal example for this User's Guide. This list only contains 4 terms, whereas the original `Mm.c2` used in the RNAseq123 analysis has thousands of terms.

The annotations are added as a nested list. The first level is the names of the annotations, referred to as the `annotationID`. This must match the `annotationID` used when entering the enrichments tables (Section 3.4). The second level is a list that describes each annotation. In addition to the list of terms, I include a human readable description as well as the name of the column in the features table that was used in the enrichments analysis (`ENTREZID`).

```
annotations <- list(
  c2 = list(
    terms = Mm.c2,
    description = "Broad Institute's MSigDB c2 collection",
    featureID = "ENTREZID"
  )
)
```

If I were to perform an enrichment analysis with an annotation database that instead used the gene symbols in its terms, then I would set `featureID = "SYMBOL"`.

I add it to the OmicNavigator study with `addAnnotations()`.

```
study <- addAnnotations(study, annotations)
```

For more details, run `?addAnnotations`.

3.9 Barcodes

The app can create an interactive barcode plot[4] to display the enrichments results for a given termID. In order to enable this view, you first need to add some information about how to construct the barcode plot.

The barcode plot displays the magnitude of the differential expression statistic for all of the features contained in a given termID. In a differential expression analysis, this is typically a t-statistic or F-statistic. Since the columns of the results table can be named anything you like, OmicNavigator doesn't know which column to use.

In the RNAseq123 example, the test statistic is in the column `t`.

[4] If you're unfamiliar with barcode plots, check out the function `barcodeplot()` from the Bioconductor package limma.

```
head(basal.vs.lp.on)
```

```
      ENTREZID     logFC   AveExpr         t      P.Value
19216    19216 -2.401062  4.263219 -8.911887 7.535233e-06
57811    57811 -5.048202 -1.579505 -7.777238 2.197986e-05
22068    22068 -4.202937  3.413562 -7.506933 2.797665e-05
12767    12767 -3.631749  5.049865 -6.255800 1.031838e-04
16439    16439 -1.957691  6.412740 -5.533072 2.368098e-04
20866    20866  1.842699  6.663871  5.377393 2.866392e-04
         adj.P.Val
19216 0.0001299437
57811 0.0002819392
22068 0.0003377516
12767 0.0009651739
16439 0.0018990476
20866 0.0022225231
```

In addition to specifying which column to use for the statistic in the barcode plot, you can provide other optional values to customize the appearance of the plot. All of the possible fields are listed below.

- `statistic` (required) - The column name in the results table to use to construct the barcode plot.
- `absolute` (optional) - Convert the statistic to its absolute value (default is `TRUE`).
- `logFoldChange` (optional) - The column name in the results table that contains the log fold change values. This is used to create an additional view containing a violin plot.
- `labelStat` (optional) - The x-axis label to describe the statistic.
- `labelLow` (optional) - The left-side label to describe low values of the statistic.
- `labelHigh` (optional) - The right-side label to describe high values of the statistic.
- `featureDisplay` (optional) - The feature variable to use to label the barcode plot on hover. This is a column name from the features table.

Below I customize the barcode plot for the RNAseq123 example. I specify that the test statistics are in the column `t` in the results table, the log fold change values are

in the column `logFC`, the x-axis is labeled as `"abs(t)"`, and the ID that is displayed when hovering over a line in the barcode plot is the column `SYMBOL` from the features table.[5] This is contained in a nested list applied to the modelID `main`, and I add the information to the study with `addBarcodes()`.

```
barcodes <- list(
  main = list(
    statistic = "t",
    logFoldChange = "logFC",
    labelStat = "abs(t)",
    featureDisplay = "SYMBOL"
  )
)
study <- addBarcodes(study, barcodes)
```

For more details, run `?addBarcodes`.

3.10 Install the study

Now I can install the study as an R package with `installStudy()`.

```
installStudy(study)
```

If I needed to transfer the study package to a different machine, instead of directly installing it, I could export it as a package tarball with `exportStudy()`.

```
exportStudy(study)
```

Then after I moved the study package tarball to another machine (or shared it with a colleague), the study package could be installed with `install.packages()`. For example, the example study from this vignette could be installed with the following command.

```
install.packages("ONstudyvignetteExample_0.0.0.9000.tar.gz",
                 repos = NULL)
```

To remove an installed study package, see `?removeStudy`.

[5] The default is the featureID, in this case the Entrez ID.

3.11 Run the app locally

After the study package is installed, I can run `startApp()` to open the app in the browser. If the study package was properly installed, I should be able to select it from the app's menu. Note that the app can't be run from within RStudio Server.

```
startApp()
```

When I'm finished exploring the results in the app, I can stop the web server by returning to the R console and pressing the Esc key (Windows) or Ctrl-C (Linux, macOS).

4 Custom plots

You can create custom plots to visualize the expression pattern of individual features in the experiment. These will be displayed in the app when a user clicks on a specific feature. In order for the app to be able to plot your data, you first need to add information on the samples and assay measurements from the experiment.

4.1 Samples

You can add a table with metadata about the samples in your study, e.g. a column to indicate "treatment" versus "control" samples. These sample metadata columns will be made available to your custom plotting functions (more on that below in Section 4). The samples table must be a data frame that follows these requirements:

- The first column must contain the study sampleID. It must be unique, and it must be a character vector.

In the RNAseq123 analysis, the sampleID is in the vector `samplenames`. The two sample metadata variables for the experiment are `group` (the type of cell) and `lane` (the lane the sample was sequenced on).

```
head(samplenames)
```

```
[1] "10_6_5_11" "9_6_5_11"  "purep53"   "JMS8-2"    "JMS8-3"
[6] "JMS8-4"
```

```
table(group)
```

```
group
Basal    LP    ML
```

```
   3     3     3
```

```
table(lane)
```

```
lane
L004 L006 L008
   3    4    2
```

I combine these 3 vectors into a data frame.

```
samplesTable <- data.frame(name = samplenames, group, lane)
head(samplesTable)
```

```
       name group lane
1 10_6_5_11    LP L004
2  9_6_5_11    ML L004
3   purep53 Basal L004
4    JMS8-2 Basal L006
5    JMS8-3    ML L006
6    JMS8-4    LP L006
```

And I assign the table to the modelID `main` and add it to my OmicNavigator study.

```
samples <- list(main = samplesTable)
study <- addSamples(study, samples)
```

For more details, run `?addSamples`.

4.2 Assays

In order to visualize the expression levels, I need to add these assay measurements to the OmicNavigator study. The assays table is a data frame with the following requirements:

- The row names should match the featureIDs in the first column of the features table (order doesn't matter)
- The column names should match the sampleIDs in the first column of the samples table (order doesn't matter)
- The columns should all be numeric

The measurements you add should be ready to plot. In other words, you don't want to perform data cleaning steps like filtering and normalizing each time a plot needs

to be made. In the RNAseq123 example, I don't want to add the raw counts. Instead I add the filtered, normalized, log-transformed counts per million (CPM) contained in the matrix `lcpm`.

```
lcpm[1:3, 1:3]
```

```
       Samples
Tags    10_6_5_11  9_6_5_11    purep53
  12767  7.552094  5.086721  3.0429773
  13603  2.679699  1.816975  1.5133290
  21390 -1.825453 -2.457757 -0.4876709
```

Since the row and column names already use the study's featureID and sampleID, respectively, all I have to do is convert it to a data frame.

```
assays <- list(main = as.data.frame(lcpm))
study <- addAssays(study, assays)
```

For more details, run `?addAssays`.

4.3 Plots

The app will call all custom plotting functions by passing a list with the filtered data to the first argument. The name of the argument can be whatever you like. An example is below.

```
nameOfPlot <- function(x) {
  # Your custom plotting code
}
```

The input list that will be passed to your custom plotting function has the following elements:

1. `assays` - A data frame that contains the assay measurements, filtered to only include the row(s) corresponding to the input featureID(s). If multiple featureIDs are requested, the rows are reordered to match the order of this input. The column order is unchanged.
2. `samples` - A data frame that contains the sample metadata for the given modelID. The rows are reordered to match the columns of the assays data frame.
3. `features` - A data frame that contains the feature metadata, filtered to only include the row(s) corresponding to the input featureID(s). If multiple featureIDs

are requested, the rows are reordered to match the order of this input (and thus match the order of the assays data frame).

4. `results` - optional. A data frame that contains test results, filtered to only include the row(s) corresponding to the input featureID(s). If multiple featureIDs are requested, the rows are reordered to match the order of this input (and thus match the order of the assays data frame).

When creating your custom plotting function, you don't need to create this list of data frames manually. Instead, you can use the same function that OmicNavigator uses internally, `getPlottingData()`. The code below obtains the input list that will be passed to the custom plotting functions when an app user selects the featureID `"12767"`. Note how both the assays and features data frames only contain one row each.

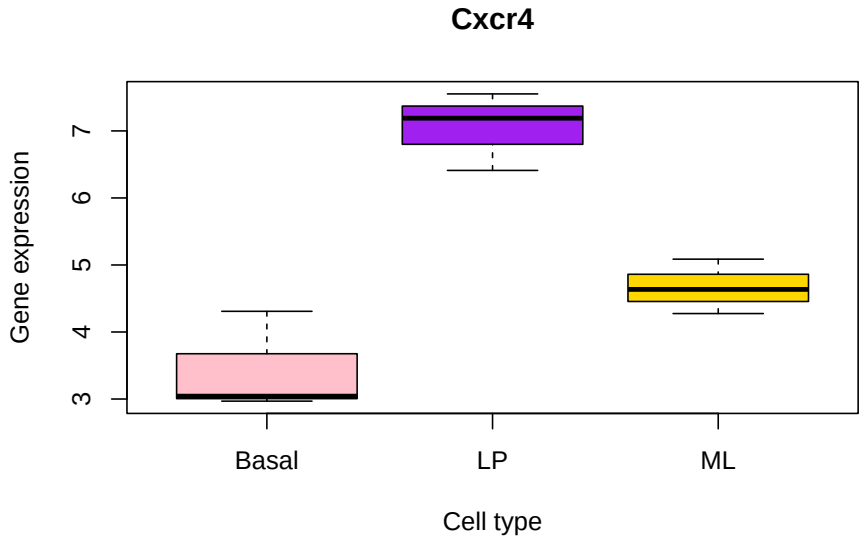
```
plottingData <- getPlottingData(study, modelID = "main", featureID = "12767")
plottingData
```

```
$assays
      10_6_5_11 9_6_5_11  purep53   JMS8-2   JMS8-3   JMS8-4   JMS8-5
12767  7.552094 5.086721 3.042977 4.308618 4.635581 6.410611 2.968442
      JMS9-P7c JMS9-P8c
12767 4.275112 7.189941

$samples
       name group lane
1 10_6_5_11    LP L004
2  9_6_5_11    ML L004
3   purep53 Basal L004
4    JMS8-2 Basal L006
5    JMS8-3    ML L006
6    JMS8-4    LP L006
7    JMS8-5 Basal L006
8  JMS9-P7c    ML L008
9  JMS9-P8c    LP L008

$features
  ENTREZID SYMBOL TXCHROM
1    12767  Cxcr4    chr1
```

Using this object, I create a boxplot to visualize the gene expression levels per cell

type.

```
boxplot(as.numeric(plottingData$assays[1, ]) ~ plottingData$samples$group,
        col = c("pink", "purple", "gold"),
        xlab = "Cell type", ylab = "Gene expression",
        main = plottingData$features$SYMBOL)
```



Once I am satisfied with the appearance of the plot, I can convert it to a function that accepts one argument `plottingData` (reminder: you can name the argument however you like, as long as you refer to it consistently in the body of your function).

```
cellTypeBox <- function(plottingData) {
  boxplot(as.numeric(plottingData$assays[1, ]) ~ plottingData$samples$group,
          col = c("pink", "purple", "gold"),
          xlab = "Cell type", ylab = "Gene expression",
          main = plottingData$features$SYMBOL)
}
```

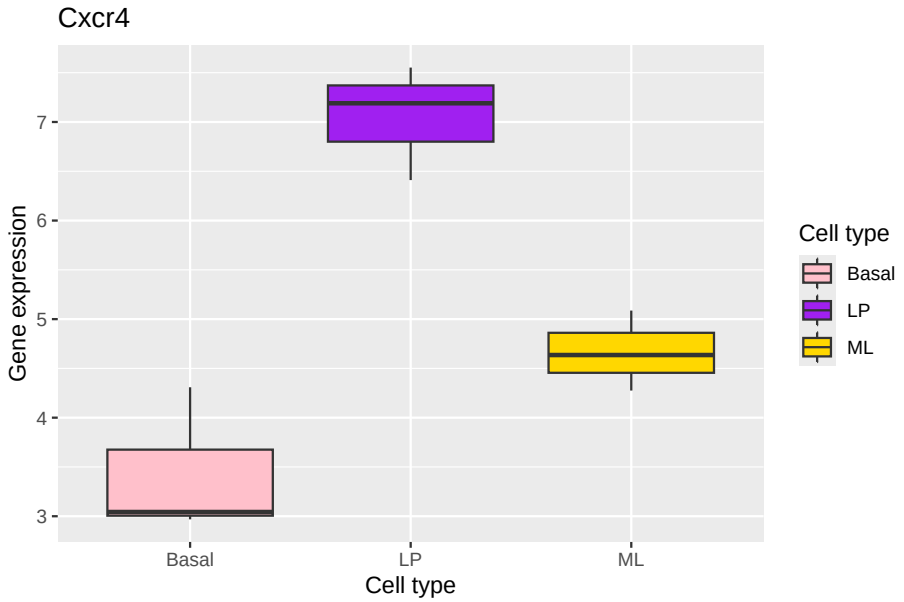
The process is similar for creating a plot with the package ggplot2. However, since these custom plotting functions will be added to an R package, you have to be slightly more careful. Also note that I have to combine the assays and samples tables to create the input data frame required by ggplot2.

```
library(ggplot2)
```

```
ggDataFrame <- cbind(plottingData$samples,
                     feature = as.numeric(plottingData$assays))
head(ggDataFrame, 3)
```

```
       name group lane  feature
1 10_6_5_11    LP L004 7.552094
2  9_6_5_11    ML L004 5.086721
3   purep53 Basal L004 3.042977
```

```
ggplot(ggDataFrame, aes(x = group, y = feature, fill = group)) +
  geom_boxplot() +
  scale_fill_manual("Cell type", values = c("pink", "purple", "gold")) +
  labs(x = "Cell type", y = "Gene expression",
       title = plottingData$features$SYMBOL)
```



When I convert my ggplot2 plot to a function, I need to preface all the references to the columns of the data frame with `.data$`. This is required for properly resolving these variables when the function is called from within a package. If you're interested in learning more, see the ggplot2 vignette Using ggplot2 in packages. Also note that you do **not** need to load the ggplot2 package inside the function. You will declare the required package dependencies when you add the custom plots to the OmicNavigator study.

```
cellTypeBoxGg <- function(plottingData) {
  ggDataFrame <- cbind(plottingData$samples,
                       feature = as.numeric(plottingData$assays))

  ggplot(ggDataFrame, aes(x = .data$group, y = .data$feature, fill = .data$group)) +
    geom_boxplot() +
    scale_fill_manual("Cell type", values = c("pink", "purple", "gold")) +
    labs(x = "Cell type", y = "Gene expression",
         title = plottingData$features$SYMBOL)
}
```

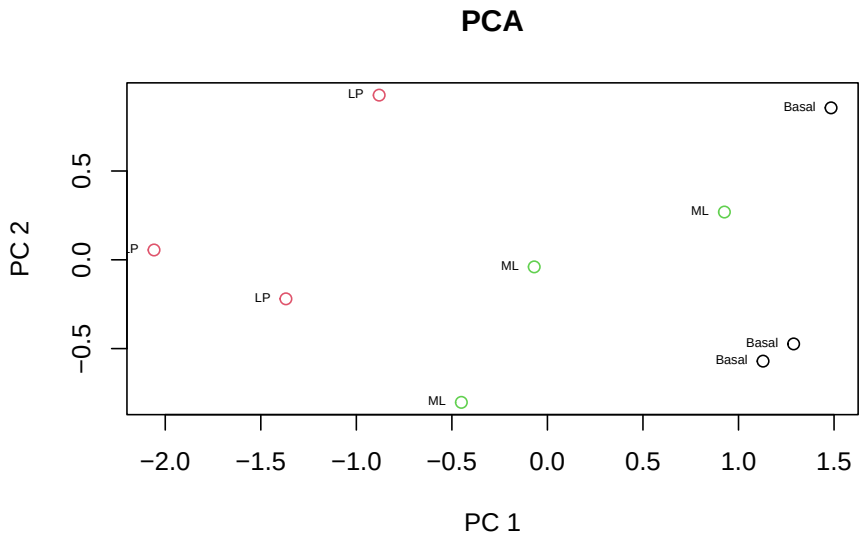
The above plots visualize one gene at a time. I can also create custom plotting functions that accept multiple featureIDs as input, which OmicNavigator refers to as "multiFeature" plots. An app user can dynamically select a subset of featureIDs, and these will be passed to your function. Below I create an example plotting function that performs PCA using a subset of featureIDs.

```
twoFeatures <- getPlottingData(study, modelID = "main",
                               featureID = c("12767", "13603"))
twoFeatures
```

```
$assays
      10_6_5_11 9_6_5_11  purep53    JMS8-2   JMS8-3   JMS8-4
12767  7.552094 5.086721 3.042977 4.3086184 4.635581 6.410611
13603  2.679699 1.816975 1.513329 0.7017626 2.337970 2.490683
        JMS8-5 JMS9-P7c JMS9-P8c
12767 2.968442 4.275112 7.189941
13603 1.396267 1.223047 1.746081

$samples
       name group lane
1 10_6_5_11    LP L004
2  9_6_5_11    ML L004
3   purep53 Basal L004
4    JMS8-2 Basal L006
5    JMS8-3    ML L006
6    JMS8-4    LP L006
7    JMS8-5 Basal L006
8  JMS9-P7c    ML L008
9  JMS9-P8c    LP L008
```

```
$features
  ENTREZID SYMBOL TXCHROM
1    12767  Cxcr4    chr1
2    13603   Opn3    chr1
```

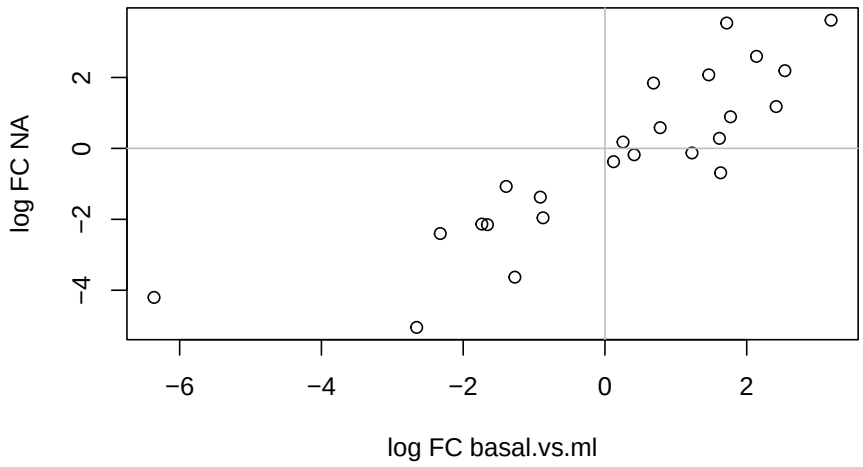
```
plotPca <- function(x) {
  if (nrow(x[["assays"]]) < 2) {
    stop("This plotting function requires at least 2 features")
  }
  pca <- stats::prcomp(t(x[["assays"]]), scale. = TRUE)$x
  plot(pca[, 1], pca[, 2], col = as.factor(x$samples$group),
       xlab = "PC 1", ylab = "PC 2", main = "PCA")
  text(pca[, 1], pca[, 2], labels = x$samples$group, pos = 2, cex = 0.5)
}
plotPca(twoFeatures)
```



The above plots visualize assays data. It is also possible to create custom plotting functions to visualize results data from a single or multiple tests. Moreover, those can be plotted for a single or multiple features. For plotting results from a single test, user should provide parameters study, modelID, featureID and testID when calling `getPlottingData()`. For plotting results from multiple tests, testID must be

provided as a vector, and plotType must indicate "multiTest". Note that plotType may accept vector for handling "multiTest", e.g. c("singleFeature", "multiTest") and c("multiFeature", "multiTest"). Below I create an example plotting function for a scatterplot using log Fold Change from two testIDs.

```
multiTests <- getPlottingData(study, modelID = "main",
                              featureID = row.names(study$assays$main),
                              testID = c("basal.vs.lp", "basal.vs.ml"))
plotMultiTestMf <- function(x) {
    df <- data.frame(lapply(x$results, `[`, 2))
    colnames(df)<- names(x$results)

    plot(df$basal.vs.lp ~ df$basal.vs.ml,
         xlab = paste0("log FC ", colnames(df)[2]),
         ylab = paste0("log FC ", colnames(df)[3]))
    abline(v=0, h = 0, col="grey")
}
plotMultiTestMf(multiTests)
```



Now that I've created the custom plots for use in the app, I compile them into a nested list and add them to the OmicNavigator study. The nested list has three levels. The first is the modelIDs, in this case `main`. The second is the name of the

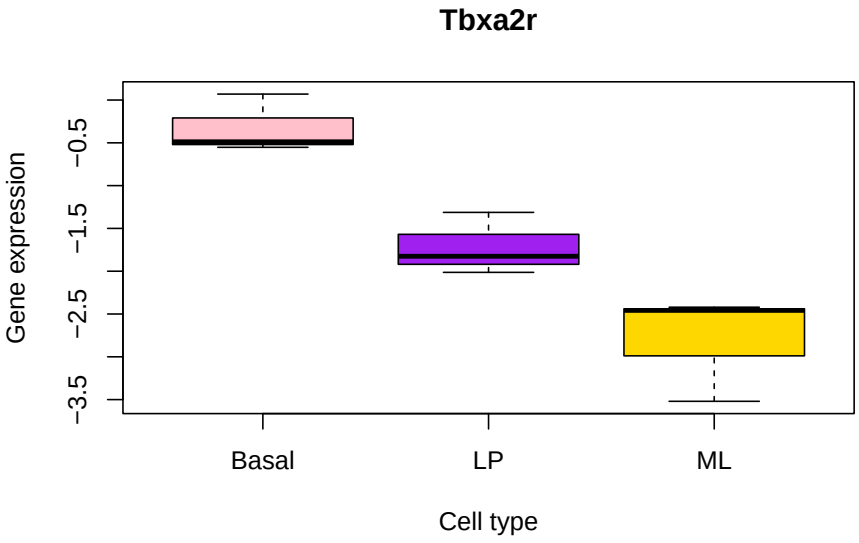
custom plotting functions as they were defined in the current R session, `cellTypeBox`, `cellTypeBoxGg`, `plotPca` and `plotMultiTestMf`. The third is information about the custom plotting function. The only required element is `displayName`, which is the text that will be displayed in the app. You are encouraged to also specify the `plotType`, e.g. `"singleFeature"`, `"multiFeature"`, `c("multiTest", "multiFeature")`. If you do not specify the `plotType`, the plot will be assumed to be `"singleFeature"`. An optional element is `packages`, in which you list any R packages that are required for the plotting function to work. OmicNavigator uses this information to properly construct the study package it creates.

```
plots <- list(
  main = list(
    cellTypeBox = list(
      displayName = "Expression by cell type",
      plotType = "singleFeature"
    ),
    cellTypeBoxGg = list(
      displayName = "Expression by cell type (ggplot2)",
      plotType = "singleFeature",
      packages = c("ggplot2")
    ),
    plotPca = list(
      displayName = "PCA",
      plotType = "multiFeature",
      packages = c("stats")
    ),
    plotMultiTestMf = list(
      displayName = "scatterplot",
      plotType = c("multiTest", "multiFeature")
    )
  )
)
study <- addPlots(study, plots = plots)
```

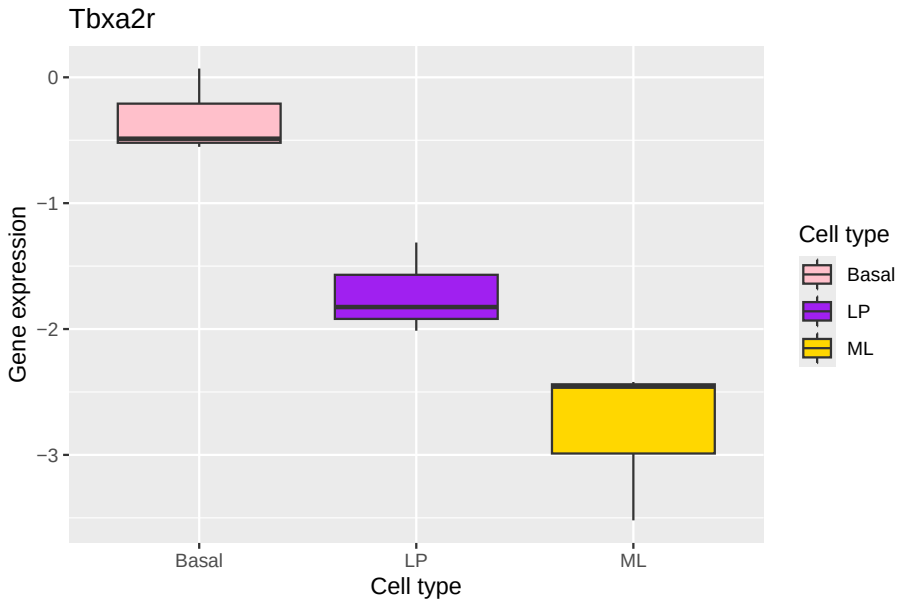
For more details, run `?addPlots`.

After you add the plotting functions, you can test that they work as you expect using the same function called by the app, `plotStudy()`. Below I call both custom "singleFeature" functions using featureID 21390.

```
plotStudy(study, modelID = "main", featureID = "21390",
          plotID = "cellTypeBox")
```

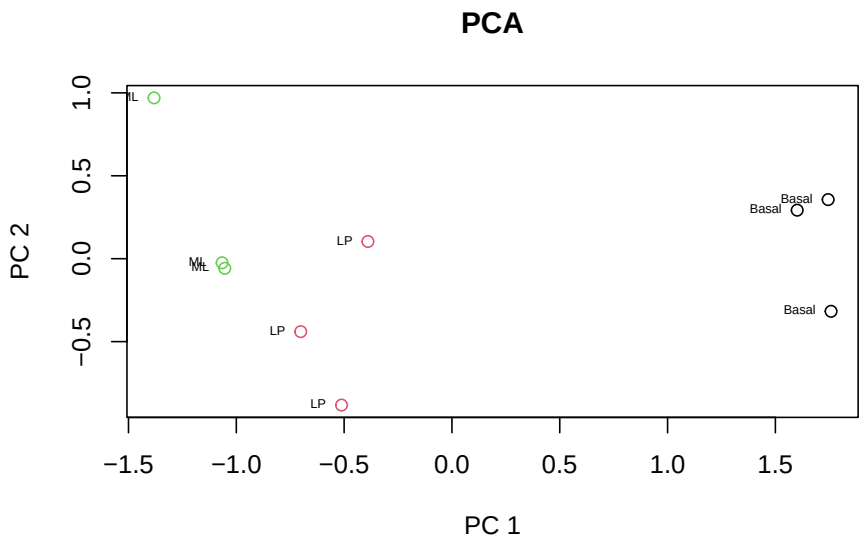



```
plotStudy(study, modelID = "main", featureID = "21390",
          plotID = "cellTypeBoxGg")
```



And I can test the "multiFeature" function by passing it multiple featureIDs.

```
plotStudy(study, modelID = "main", featureID = c("21390", "19216"),
          plotID = "plotPca")
```



Optionally, data related to test results can be plotted by providing a valid testID to the `plotStudy()`. For instance, for `modelID = "main"` one could call `plotStudy()` with `testID = "basal.vs.lp"` to be able to plot its test results. For plotting data from multiple tests, plotType must be set to "multiTest".

To make the custom plotting functions available in the app, re-install the study package with `installStudy()`.

5 Extras

I didn't include every possible addition in Section 3 above. Below are some extras you can also include in your study package. Similar to many of the other elements, these are purely optional. They are available to enhance your study package if you want them. After adding any of these extra elements, remember to re-install the study package with `installStudy()` for the changes to take effect in the app.

5.1 MetaFeatures

The features tables explained in Section 3.7 require that the first column containing the study featureID is unique. This enforces a 1:1 mapping between the featureID and the other columns. In general this works well; however, it doesn't handle the case where a given featureID may map to multiple other values of a different variable. For example, each Entrez gene ID used in the RNAseq123 study may map to 0, 1, or more Ensembl gene IDs. While you could cram the multiple values into one column using a separator like a semi-colon, this is discouraged since it will not display nicely in the app's table. Instead you can add a metaFeatures table, which is a catch-all for any information about the features that doesn't map 1:1.

The metaFeatures table has the following requirements:

- The first column must contain the featureID. Each featureID must also be included in the corresponding features table. Not every featureID has to be included, the order does not matter, and it is expected that the featureIDs will be repeated.
- If providing metaAssays (see Section 5.2 below), the second column should contain the metaFeatureIDs that correspond to the row names of the metaAssays data frame.
- The remaining columns must be character vectors.

Adding the metaFeatures table is similar to the features table. Use a nested list to assign the table(s) to a modelID, then use `addMetaFeatures()`. For more details, run `?addMetaFeatures`.

5.2 MetaAssays

If the study metaFeatures have associated measurements, you can store these in metaAssays via `addMetaAssays()`. Examples include multiple peptide measurements for a single protein or multiple CRISPR guides that map to a single gene. These data will then be made available for custom plots (Section 4).

The requirements for the metaAssays data frame are similar to those for the assays data frame (Section 4.2):

- The row names should match the metaFeatureIDs in the second column of the metaFeatures table (order doesn't matter)

- The column names should match the sampleIDs in the first column of the samples table (order doesn't matter)
- The columns should all be numeric

For more details, run `?addMetaAssays`. Note that this feature is still experimental, and the data format may change in the future.

5.3 Reports

Some users of the app may be interested in learning more of the details of the analysis. If you created a report of your analysis, you can add this to your OmicNavigator study. Then it will be available for interested users. The report can be any file format. You need to provide a path to the file on your local machine (in which case it will be bundled into the study package), or you can provide a URL that points to the file.

In this case, the RNAseq123 analysis report is hosted on Bioconductor's website. Below I purposefully use a small text size in order to display the full URL.

```
reportUrl = "https://bioconductor.org/packages/release/workflows/vignettes/RNAseq123/inst/doc/limmaWorkflow.html"
```

As usual, I create a nested list with one entry per `modelID`.

```
reports <- list(
  main = reportUrl
)
```

And add it to the OmicNavigator study with `addReports()`.

```
study <- addReports(study, reports)
```

For more details, run `?addReports`.

5.4 Results table linkouts

You can provide linkouts to external resources for the features in your study. These linkouts are embedded directly into the results table (Section 3.3). The linkout URLs can use the featureID or any of the columns in the optional features table (Section 3.7).

For example, to provide a linkout for each Entrez ID in the RNAseq123 study, I can use the linkout pattern `https://www.ncbi.nlm.nih.gov/gene/`. The Entrez ID in each row will be appended to the linkout pattern to create the valid URL, e.g. `https://www.ncbi.nlm.nih.gov/gene/242505`. Below I create a list with this

linkout. I use the special modelID "default" so that the linkouts are added to the results table for every modelID. The name of the nested list is `ENTREZID` because that is the name of the column.

```
resultsLinkouts <- list(
  default = list(
    ENTREZID = "https://www.ncbi.nlm.nih.gov/gene/"
  )
)
```

Then I add the results linkouts to the study.

```
study <- addResultsLinkouts(study, resultsLinkouts)
```

For more details, run `?addResultsLinkouts`.

5.5 Enrichments table linkouts

You can provide linkouts to external resources for the annotation terms used for your enrichment analyses. These linkouts are embedded directly into the termID column of the enrichments table (Section 3.4).

For example, to provide a linkout for each termID for the annotation c2 in the RNAseq123 study, I can use the linkout pattern `https://www.gsea-msigdb.org/gsea/msigdb/cards/`. The termID in each row will be appended to the linkout pattern to create the valid URL, e.g. `https://www.gsea-msigdb.org/gsea/msigdb/cards/REACTOME_OPSINS`. Below I create a list with this linkout. The name of the list corresponds to the name of the annotationID.

```
enrichmentsLinkouts <- list(
  c2 = "https://www.gsea-msigdb.org/gsea/msigdb/cards/"
)
```

Then I add the enrichments linkouts to the study.

```
study <- addEnrichmentsLinkouts(study, enrichmentsLinkouts)
```

For more details, run `?addEnrichmentsLinkouts`.

5.6 MetaFeatures table linkouts

You can provide linkouts to external resources for the metaFeatures in your study. These linkouts are embedded directly into the metaFeatures table (Section 5.1). The

linkout URLs can use any of the columns in the optional metaFeatures table.

For more details, run `?addMetaFeaturesLinkouts`.

5.7 Study metadata

You can add metadata to describe your study by passing a named list to to the argument `studyMeta` when creating your study with `createStudy`. The names of the list cannot contain spaces or colons, and they can't start with `#` or `-`. The values of each list should be a single value.

For more details, run `?createStudy`.

5.8 Objects

If your analysis used a complex R object from a Bioconductor package, Seurat, etc., and you would like to be able to use it when creating custom plots in the app (Section 4), you can add it to a modelID using `addObjects()`.

For more details, run `?addObjects`. Note that this feature is still experimental, and the data format may change in the future.

6 More information

This section contains more information about OmicNavigator studies.

6.1 Accessing elements from the OmicNavigator study

Each function to add elements to an OmicNavigator study, e.g. `addFeatures()`, has a corresponding function to get elements from an OmicNavigator study, e.g. `getFeatures()`. Below I collectively refer to these as "get" functions.

Calling a "get" function on the study without any additional arguments will return a nested list with all the available information.

```
str(getFeatures(study))
```

```
List of 1
 $ main:'data.frame':        24 obs. of  3 variables:
  ..$ ENTREZID: chr [1:24] "19216" "57811" "22068" "12767" ...
```

```
  ..$ SYMBOL  : chr [1:24] "Ptger1" "Rgr" "Trpc6" "Cxcr4" ...
  ..$ TXCHROM : chr [1:24] "chr8" "chr14" "chr9" "chr1" ...
```

Each "get" function also has arguments for filtering the results. For example, specifying `modelID = "main"` to `getFeatures()` returns the data frame with the features for that modelID.

```
str(getFeatures(study, modelID = "main"))
```

```
'data.frame':        24 obs. of  3 variables:
 $ ENTREZID: chr  "19216" "57811" "22068" "12767" ...
 $ SYMBOL  : chr  "Ptger1" "Rgr" "Trpc6" "Cxcr4" ...
 $ TXCHROM : chr  "chr8" "chr14" "chr9" "chr1" ...
```

The situation is analogous for elements that are more highly nested. For example, the results tables are nested per test per model. Thus specifying the modelID returns a list of the available tests, and specifying both the modelID and testID returns the data frame.

```
str(getResults(study))
```

```
List of 1
 $ main:List of 2
  ..$ basal.vs.lp:'data.frame':        24 obs. of  6 variables:
  .. ..$ ENTREZID : chr [1:24] "19216" "57811" "22068" "12767" ...
  .. ..$ logFC    : num [1:24] -2.4 -5.05 -4.2 -3.63 -1.96 ...
  .. ..$ AveExpr  : num [1:24] 4.26 -1.58 3.41 5.05 6.41 ...
  .. ..$ t        : num [1:24] -8.91 -7.78 -7.51 -6.26 -5.53 ...
  .. ..$ P.Value  : num [1:24] 7.54e-06 2.20e-05 2.80e-05 1.03e-04 2.37e-04 ...
  .. ..$ adj.P.Val: num [1:24] 0.00013 0.000282 0.000338 0.000965 0.001899 ...
  ..$ basal.vs.ml:'data.frame':        24 obs. of  6 variables:
  .. ..$ ENTREZID : chr [1:24] "22068" "19216" "21390" "22065" ...
  .. ..$ logFC    : num [1:24] -6.36 -2.32 2.42 2.54 -1.39 ...
  .. ..$ AveExpr  : num [1:24] 3.41 4.26 -1.79 -3.1 7.84 ...
  .. ..$ t        : num [1:24] -12.83 -8.46 3.34 3.19 -3.02 ...
  .. ..$ P.Value  : num [1:24] 4.68e-07 1.12e-05 4.83e-03 6.06e-03 7.88e-03 ...
  .. ..$ adj.P.Val: num [1:24] 1.77e-05 1.55e-04 2.32e-02 2.84e-02 3.58e-02 ...
```

```
str(getResults(study, modelID = "main"))
```

```
List of 2
 $ basal.vs.lp:'data.frame':        24 obs. of  6 variables:
```

```
  ..$ ENTREZID : chr [1:24] "19216" "57811" "22068" "12767" ...
  ..$ logFC    : num [1:24] -2.4 -5.05 -4.2 -3.63 -1.96 ...
  ..$ AveExpr  : num [1:24] 4.26 -1.58 3.41 5.05 6.41 ...
  ..$ t        : num [1:24] -8.91 -7.78 -7.51 -6.26 -5.53 ...
  ..$ P.Value  : num [1:24] 7.54e-06 2.20e-05 2.80e-05 1.03e-04 2.37e-04 ...
  ..$ adj.P.Val: num [1:24] 0.00013 0.000282 0.000338 0.000965 0.001899 ...
 $ basal.vs.ml:'data.frame':        24 obs. of  6 variables:
  ..$ ENTREZID : chr [1:24] "22068" "19216" "21390" "22065" ...
  ..$ logFC    : num [1:24] -6.36 -2.32 2.42 2.54 -1.39 ...
  ..$ AveExpr  : num [1:24] 3.41 4.26 -1.79 -3.1 7.84 ...
  ..$ t        : num [1:24] -12.83 -8.46 3.34 3.19 -3.02 ...
  ..$ P.Value  : num [1:24] 4.68e-07 1.12e-05 4.83e-03 6.06e-03 7.88e-03 ...
  ..$ adj.P.Val: num [1:24] 1.77e-05 1.55e-04 2.32e-02 2.84e-02 3.58e-02 ...
```

```
str(getResults(study, modelID = "main", testID = "basal.vs.lp"))
```

```
'data.frame':        24 obs. of  6 variables:
 $ ENTREZID : chr  "19216" "57811" "22068" "12767" ...
 $ logFC    : num  -2.4 -5.05 -4.2 -3.63 -1.96 ...
 $ AveExpr  : num  4.26 -1.58 3.41 5.05 6.41 ...
 $ t        : num  -8.91 -7.78 -7.51 -6.26 -5.53 ...
 $ P.Value  : num  7.54e-06 2.20e-05 2.80e-05 1.03e-04 2.37e-04 ...
 $ adj.P.Val: num  0.00013 0.000282 0.000338 0.000965 0.001899 ...
```

Conveniently, the "get" functions work exactly the same on both OmicNavigator study objects defined in the current R session as well as installed OmicNavigator study packages. Instead of passing the object, you pass the name of the OmicNavigator study.

```
str(getFeatures("vignetteExample", modelID = "main"))
```

```
'data.frame':        24 obs. of  3 variables:
 $ ENTREZID: chr  "19216" "57811" "22068" "12767" ...
 $ SYMBOL  : chr  "Ptger1" "Rgr" "Trpc6" "Cxcr4" ...
 $ TXCHROM : chr  "chr8" "chr14" "chr9" "chr1" ...
```

6.2 Sharing elements across models

The example study only had one model. This was for ease of demonstration. However, you are able to have has many models as you like. As mentioned in Section 2.1,

different models can be completely different (e.g. transcripts versus protein measurements) or very similar (e.g. addition of one extra coefficient to the statistical model). In the case where two or more models are very similar, it would be unnecessarily tedious and storage-intensive to store identical information for multiple models of a given study. To avoid this, OmicNavigator recognizes the special modelID `default`. If a "get" function requests an element for a modelID which doesn't exist, it then looks to see if there is a table available for the modelID `default`. This allows you to specify shared elements that apply across models of a study as well as override the default for a subset of the models.

As an example, in the RNAseq123 study, many of the elements could have been added for the modelID `default`. This would make no difference when there is only one model, and if another one was added, it would immediately share these elements. The code below demonstrates how the default features table is returned when a modelID is specified that doesn't have its own features table.

```
studyWithDefault <- addFeatures(study, list(default = basal.vs.lp[, 1:3]))
str(getFeatures(studyWithDefault, modelID = "modelThatDoesntExistYet"))
```

```
'data.frame':        24 obs. of  3 variables:
 $ ENTREZID: chr  "19216" "57811" "22068" "12767" ...
 $ SYMBOL  : chr  "Ptger1" "Rgr" "Trpc6" "Cxcr4" ...
 $ TXCHROM : chr  "chr8" "chr14" "chr9" "chr1" ...
```

6.3 Naming OmicNavigator study packages

OmicNavigator studies are installed as standard R packages on your machine. This allows the app to query them using the OpenCPU framework. In order to distinguish them from other R packages, by default the prefix "ONstudy" is added to the package name. For example, an OmicNavigator study named "ABC" is installed as "ONstudyABC".

If you'd like to change the default prefix, you can change the OmicNavigator package option that controls this behavior, `OmicNavigator.prefix`. For example, to use the prefix "OmicNavigatorStudy", you could add the following line to your .Rprofile file.

```
options(OmicNavigator.prefix = "OmicNavigatorStudy")
```

For more details about this and other OmicNavigator package options, run `?OmicNavigator`.

6.4 Mapping between data elements and app features

The minimum requirement for a valid OmicNavigator study is a single results table. This will result in the app displaying an interactive table to explore the results. To enable more interactive features, you can add more data. The table below maps the app features to the required and optional data elements. Click on the links to be taken to the relevant section.

App feature	Required	Optional
Results table	Results (3.3)	Features (3.7), resultsLinkouts (5.4)
Enrichments table	Enrichments (3.4)	enrichmentsLinkouts (5.5)
Enrichments network	Enrichments (3.4), Annotations (3.8)	
Enrichments barcode	Barcodes (3.9), Enrichments (3.4), Results (3.3)	
Custom plots	Plots (4.3), Assays (4.2)	Features (3.7), Samples (4.1), Results (3.3), metaFeatures (5.1), metaAssays (5.2), objects (5.8)
MetaFeatures table	metaFeatures (5.1)	metaFeaturesLinkouts (5.6)

6.5 Matching plot theme to app's appearance

Your custom plots will be displayed in the app. If you'd like, you can theme your plots so that they better integrate with the app's appearance, e.g. with `ggplot2::theme()` or `ggplot2::scale_color_discrete()`. Note that this is completely optional.

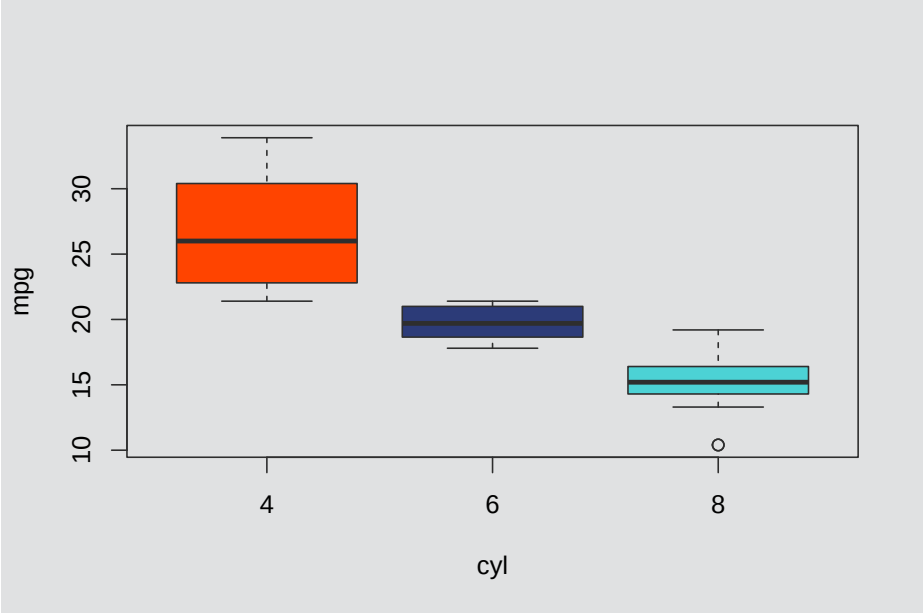
The app uses the following colors:

- orange-reddish `#ff4400`
- light orange `#ff7e38`
- navy blueish `#2c3b78`
- darker royal blueish `#465fc5`
- baby blueish `#4cd2d5`
- royal blueish `#1678c2`

- white `#fff`
- lightish grey `#e0e1e2`
- light blackish `#2e2e2e`

You can use the hexadecimal codes for the colors directly in R, e.g.

```
op <- par(bg = "#e0e1e2", fg = "#2e2e2e", no.readonly = TRUE)
boxplot(mpg ~ cyl, data = mtcars, col = c("#ff4400", "#2c3b78", "#4cd2d5"))
par(op)
```



The app's text is displayed in one of the following fonts. It will choose the first font it finds installed on the machine:

1. Lato
2. Arial
3. Helvetica
4. Any sans-serif font

Fair warning that getting R to use a custom installed font like Lato is non-trivial, and it's even more difficult to make it portable (i.e. so that the font will also work on the server where OmicNavigator is deployed or a colleague's machine). Two options for

R packages that can help you use custom fonts in R plots are showtext and extrafont.

7 Session information

- R version 4.5.2 (2025-10-31), `aarch64-apple-darwin20`
- Locale: `C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Time zone: `America/Chicago`
- TZcode source: `internal`
- Running under: `macOS Sequoia 15.5`
- Matrix products: default
- BLAS: `/System/Library/Frameworks/Accelerate.framework/Versions/A/Frameworks/vecLib.fra`
- LAPACK: `/Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRlapack.dyli` ; LAPACK version3.12.1
- Base packages: base, datasets, grDevices, graphics, methods, stats, utils
- Other packages: OmicNavigator 1.19.0, ggplot2 4.0.1, jsonlite 2.0.0
- Loaded via a namespace (and not attached): ONstudyABC 0.0.0.9000, R6 2.6.1, RColorBrewer 1.1-3, Rcpp 1.1.0, S7 0.2.1, UpSetR 1.4.0, cli 3.6.5, compiler 4.5.2, curl 7.0.0, data.table 1.17.8, dplyr 1.1.4, farver 2.1.2, faviconPlease 0.1.4, generics 0.1.4, glue 1.8.0, grid 4.5.2, gridExtra 2.3, gtable 0.3.6, httr 1.4.7, labeling 0.4.3, lifecycle 1.0.4, magrittr 2.0.4, pillar 1.11.1, pkgconfig 2.0.3, plyr 1.8.9, rlang 1.1.6, rstudioapi 0.17.1, scales 1.4.0, tibble 3.3.0, tidyselect 1.2.1, tools 4.5.2, vctrs 0.6.5, withr 3.0.2, xml2 1.5.1

8 References

[1] Law CW, Alhamdoosh M, Su S, Dong X, Tian L, Smyth GK, Ritchie ME. RNA-seq analysis is easy as 1-2-3 with limma, Glimma and edgeR [version 3; peer

review: 3 approved]. F1000Research 2018, 5:1408

[2] Sheridan, J.M., Ritchie, M.E., Best, S.A. et al. A pooled shRNA screen for regulators of primary mammary stem and progenitor cells identifies roles for *Asap1* and *Prox1*. BMC Cancer 2015, 15:221