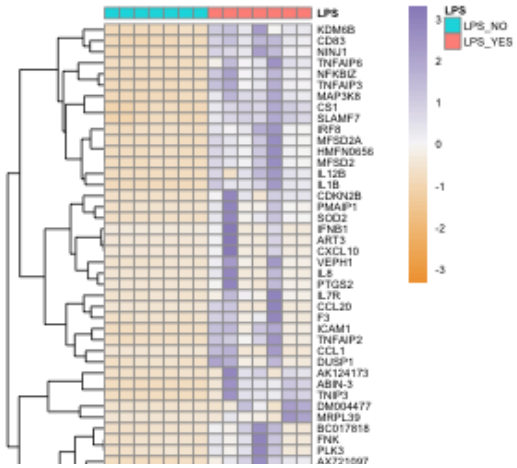
Documentation Booklet

MetaR

```
Analysis edgeR diff exp
{
  import table GSE59364_DC_all.csv

  subset rows GSE59364_DC_all.csv when true: $(gene) != "Total" -> filtered

  edgeR counts= filtered model: ~ 0 + LPS
    comparing LPS
                GSE59364_DC_all.csv          ^Table (leaflet)th   tagwise dispersion  )
                filtered ^myOwnTable (leaflet.edgeR diff exp)
  join ( filtered, Results ) by group ID -> MergedResults
  subset rows MergedResults when true: ($(FDR) < 0.0001) & ($(logFC) > 2 | $(logFC) < -2) -> 1% FDR

  heatmap with 1% FDR select data by one or more group LPS=YES, group LPS=NO -> plot HeatmapStyle [
    annotate with these groups: LPS
    scale values: scale by row
    cluster columns: false cluster rows: true
  ]
  multiplot -> PreviewHeatmap [  1 cols x 1 rows ]
```

Hide preview

```
[ plot ]
```



MetaR blends User Interfaces and Scripting

http://metaR.campagnelab.org

Provide Simple Abstractions that non-programmers can use for Data Analysis

PUBLISHED BY FABIEN CAMPAGNE

HTTP://BOOKS.CAMPAGNELAB.ORG

Version 2.4.0 May 2018

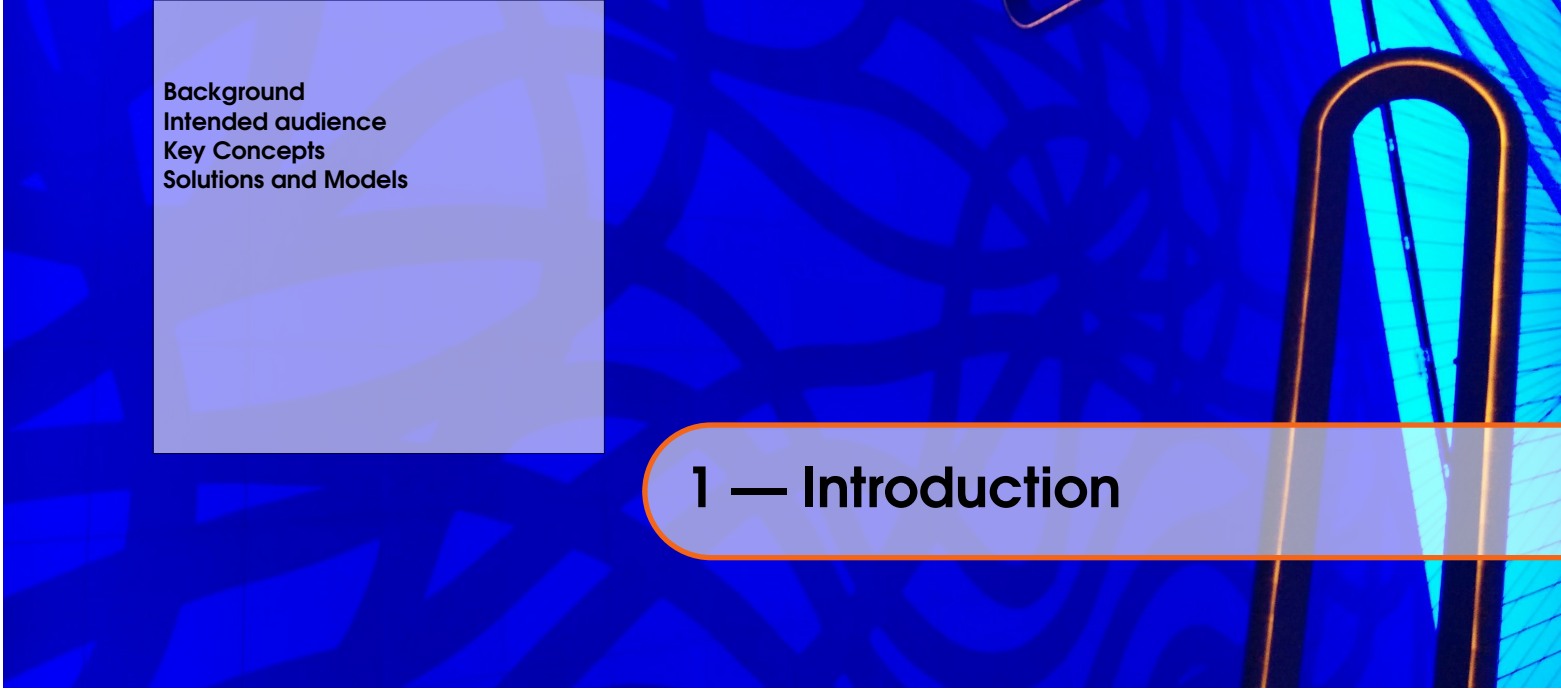
Credits

The following authors have contributed to this booklet: Ana-Maria Sutii, Alexander Pann, William ER Digan, Manuele Simi and Fabien Campagne.

The authors thank the developers of the Meta Programming System, who have developed MPS since the early 2000. MetaR would not have been possible without them.
MPS Project leaders, in chronological order: Sergey Dmitriev, Igor Alshannikov, Konstantin Solomatov and Alexander Shatalin. **Current team members:** Alexander Shatalin, Fedor Isakov, Mihail Muhin, Michael Vlassiev, Václav Pech, Simon Alperovich, Daniil Elovkov, Victor Matchenko, Artem Tikhomirov, Mihail Buryakov and Alexey Pyshkin. **Earlier members of the MPS team:** Evgeny Gryaznov, Timur Abishev, Julia Beliaeva, Cyril Konopko, Ilya Lintsbah, Gleb Leonov, Evgeny Kurbatsky, Sergey Sinchuk, Timur Zambalayev, Maxim Mazin, Vadim Gurov, Evgeny Geraschenko, Darja Chembrovskaya, Vyacheslav Lukianov and Alexander Anisimov. **And these external contributors:** Sascha Lisson, Thiago Tonelli Bartolomei and Alexander Eliseyev.

We also thank the many people who have taken the MetaR training sessions at the Clinical Translational Science Center (Weill Cornell Medical College, Memorial Sloan Kettering, and Hospital for Special Surgery and Hunter College). Their feedback we received during these sessions have been instrumental in rapidly making MetaR user-friendly.

Contents



1 — Introduction

1.1 Background

The MetaR software `http://MetaR.campagnelab.org` is an example of a new kind of interactive tool for data analysis. It was developed by the Campagne laboratory using the Meta Programming System (MPS) (see `http://www.jetbrains.com/mps` [**Dmitriev:2004**]. MPS is a mature Language Workbench that makes it relatively easy to create new languages and tools to help users of these languages [**campagne2014mps**].

1.2 Intended audience

This booklet is designed to teach how to use MetaR for data analysis. In the first chapters, we will assume that you have no prior scripting or programming experience, but will expect you to know how to use a computer.

Chapters 14 will be useful for users who have prior programming experience. This chapter explains how to extend MetaR with new language constructs.

1.3 Key Concepts

High-level data abstractions

MetaR is designed to make it easier to conduct data analysis. To achieve this goal, and in contrast to programming languages such as the R language, Julia or Python, that are often low-level and require good programming skills, MetaR offers high-level data abstractions and provides assistance in manipulating data. High-level abstractions used in MetaR analyses include the following concepts:

1. Tables, Columns, Column Groups and Group usages,
2. Analyses
3. Plot
4. Model

These high-level abstractions will be explained in the following chapters.

R program generation

Analyses developed with MetaR are transformed into R programs when the user needs to execute them. MetaR is tightly integrated with MPS to make it seamless to run analyses without prior knowledge of the R language.

Note that while most users can use MetaR without knowledge of the R language, all users will need to install the R language in order to execute MetaR analyses.

Docker integration

Starting with version 1.3.1, MetaR can run the R scripts that it generates into Docker containers. Docker is a technology that makes it easier to obtain reproducible script executions. When MetaR does not use Docker, it is possible for the R runtime to try to install a package the first time you run an analysis with a statement that requires the package. While this should not be a problem, we found that R package installation is brittle and can sometimes fail at various time points, for a variety of reasons[1]. To avoid such problems, which tend to occur when we give a training sessions, we now support running Analyses with Docker. Chapter 4 explains how to configure MetaR to use Docker for reproducible analyses.

Composable language

MetaR is built as an MPS language. In contrast to languages built with compiler technology, MPS languages are composable. Briefly, language composition is the ability to compose, or combine, two or more languages. This ability is particularly useful to extend MetaR. It is possible to define micro-languages, made of one or two statement types. When micro-languages are composed with MetaR, they can support specific application domains. The EdgeR and Limma languages provide examples of composition, where an R/Bioconductor package called EdgeR (or Limma) has been integrated with MetaR and exposes a new kind of statement to perform calls of differential expression. See Chapters 7 and 14 to learn more about language composition in MetaR.

Composable R Language

Since version 1.5, MetaR is distributed with a composable R language. This language is a full implementation of the R language in MPS. In contrast to the R language expressed as text program, you can use composition to introduce new features in this R language. For instance, you can design micro-languages for R, in a similar manner to that demonstrated for MetaR (see Chapter 14). See Chapter 15 for details about *composableR* and an example illustrating how you can embed an interactive user interface into R programs developed with this language.

[1] For instance, because a new version of a package has become available which breaks compilation of the package on your machine.

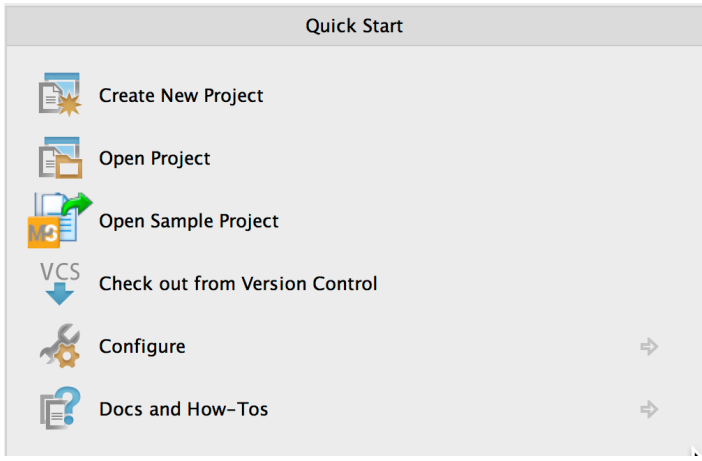


Figure 1.1: **The Quick Start menu.** This menu is displayed when you first open MPS, or when you have no project currently open in MPS. The first item in the menu is used to create a new project.

The Composable R Language in MetaR 1.5 should be considered experimental. While we believe the language supports the equivalent of the full R grammar, the implementation and interactive language editing may not be as smooth as some of the languages that ship with MetaR. Please report any interactive editing problem you might encounter as a GitHub issue (see `https://github.com/CampagneLaboratory/MetaR/issues`).

1.4 Solutions and Models

Developing an analysis with MetaR is done by creating nodes of the MetaR concepts in MPS models. Models exist in MPS solutions. To learn how to create an MPS solution, read the preview of the MPS language workbench available at `http://campagnelab.org/publications/our-books/`[**campagne2014mps**], or watch the beginning of the MetaR training video at `http://campagnelab.org/software/metar/video-tutorials/`. Figures 1.1-1.2 provide a brief walk-through of the steps you should follow to create a project and solution. After the project open, open the project tab(see [**campagne2014mps**]) and right-click on the solution name. Import the metaR devkit and create a model.

You can create as many models as you need to store your analyses. In the following chapter, we assume that you have created a solution and at least one model in this solution.

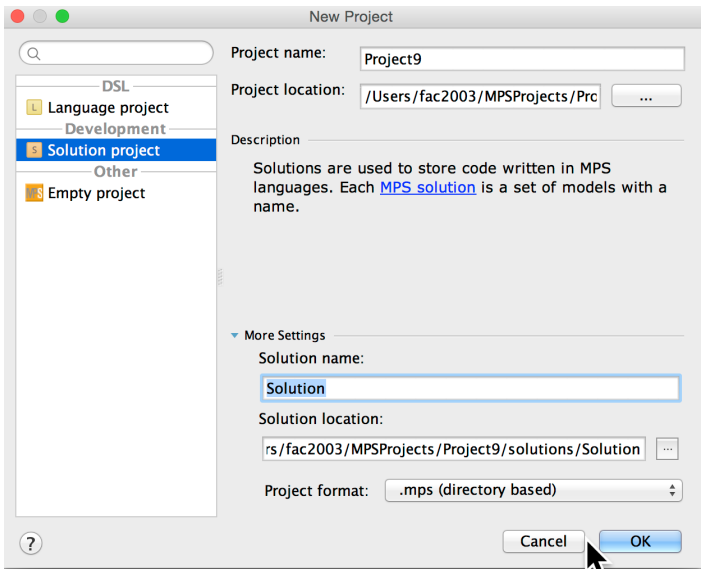


Figure 1.2: **The New Project Dialog.** Use this dialog to create a new Solution. Solutions are used to store models and express your solutions to specific analysis problems. Select the "Solution Project" project type on the left panel, name the project, and name the solution you wish to create.

2 — Tables

2.1 Overview

Before you start an analysis, you will need to define Table nodes that represent the data files needed in the analysis. MetaR explicitly models files that contain tables of data. This is done so that you can easily refer to these tables, without having to remember where the original file is located on your computer.
In this Chapter, you will learn how to

1. define a MetaR table,
2. adjust the types of the columns of the data described in the file,
3. annotate a table with groups,
4. link groups in column group usage.

2.2 Create a Table

To create a Table, right-click on a model in the project tab and select right-click › New › o.c.metar.tables › Table. This will create an empty table, as shown in Figure 2.1. Tables have a `name`, a `pathToResolve` attribute and a list of columns. The following paragraphs describe these attributes.

name

The table name is set automatically from the path when you use the file selection button. You can change the name to match your analysis needs and make it easier to remember what is in the table.

File Path (pathToResolve)

This attribute contains a path to the TSV file that you wish to analyze. The path may contain references to path variables that will be automatically resolved before MetaR attempts to load table information from the path. Path variable references can be written as `$a.b.c/data/file.tsv`. Such a reference will require you to define the a.b.c path variable name and associate it with a value on each machine where the table will be used.

```
Table <no name> ...
File Path
  <no pathToResolve>
Columns
  <no name>:
```

Figure 2.1: **New Table.** This figure presents a freshly created Table AST Root node. You can use the button located to the right, after the <no name> red label (...), to open a file selection dialog. Use this dialog to locate a TSV file to configure this table.

You can define path variables with the Preferences/Settings MPS menu (MPS ⟩ Preferences ⟩ PathVariables on Mac, MPS ⟩ Settings ⟩ PathVariables on PC).

Columns

Columns is an attribute that presents the list of columns identified in the TSV file. Each column has a name, a type, and may be annotated with a set of column groups (see Section 2.4 for details about column groups). MetaR supports the following column types, which map to the R data types:

1. **Numeric**. Any number. Technically, can be a floating number or an integer.
2. **String**. A string of characters.
3. **Boolean**. A type that can only have two values: `true` or `false`.
4. **Category**. A type that can take only a limited number of values (e.g., {`RED`, `GREEN`, `BLUE`} would be a category with these values, (`RED`, `GREEN` and `BLUE` in our example).

These types are automatically determined from the data in the table file. However, in case the automatic algorithm failed for a table, you can change the types manually. To do this, put the cursor over the name of the type in the column section, and use auto-completion in the inspector to change to the desired type.

Figure 2.2 presents a MetaR table annotated with groups.

2.3 Column Groups Container

Column Groups Containers are used to define column groups and annotate these groups with group usages. To create a new Column Group Container, right-click on a model and select New ⟩ o.c.tables.ColumnGroupContainer. This will create the empty container shown in Figure 2.3. You need and should have only one container per model. The container will hold the groups and group usages that you need across the MetaR Tables defined in the model.

2.4 Column Groups

Column Groups can be defined by pressing ⏎ either on top of the « ... » (immediately below `Define Groups:`), when the list of groups is empty, or with the cursor immediately before the name of an existing group. Each group has a name and an optional set of group usages. Figure 2.4 presents a new column group (group for short).

```
Table GSE59364_DC_all.csv ...
File Path
  ${org.campagnelab.metaR.home}/data/GSE59364_DC_all.csv
Columns
  gene: string [ ID ]
  mRNA len: numeric [ << ... >> ]
  genomic span: numeric [ << ... >> ]
  DC_normal: numeric [ << ... >> ]
  DC_treated: numeric [ << ... >> ]
  DC0904: numeric [ LPS=NO, counts ]
  DC0907: numeric [ LPS=NO, counts ]
  DCLPS0910: numeric [ LPS=NO, counts ]
  DCLPS0913: numeric [ LPS=NO, counts ]
  A_DC: numeric [ LPS=NO, counts ]
  A_DC_LPS: numeric [ LPS=YES, counts ]
  B_DC: numeric [ LPS=NO, counts ]
  B_DC_LPS: numeric [ LPS=YES, counts ]
  C_DC: numeric [ LPS=NO, counts ]
  C_DC_LPS: numeric [ LPS=YES, counts ]
  C2DC: numeric [ LPS=NO, counts ]
  C2DCLPS: numeric [ LPS=YES, counts ]
  C3DC: numeric [ LPS=NO, counts ]
  C3DCLPS: numeric [ LPS=YES, counts ]
```

Figure 2.2: **Example Table.** The figure presents an example table annotated with groups.

Figure 2.3: **Empty Column Group Container.** Use a column group container to define column groups and associated group usages. Place the cursor over the « ... » symbol and press ⏎ to add a new group or group usage. Name the group or usage immediately after creating it.

```
Column Groups and Usages

Define Usages:
  << ... >>

Define Groups:
  << ... >>
```

Figure 2.4: **New Group.** You should name a new group immediately after creating it.

```
<no name> used for << ... >>
```

```
Column Groups and Usages

Define Usages:
  LPS_Treatment
  ID
  heatmap

Define Groups:
  LPS=YES used for LPS_Treatment heatm
  LPS=NO used for LPS_Treatment heatma
  ID used for ID
  counts used for heatmap
```

Figure 2.5: **Example Group Container.** This example presents a container with several groups and group usages.

name

The name of the group is a string that should mean something in the context of your analysis. Some MetaR analysis statements require specific group names to be defined in a model container and referenced in an input table. Other groups will be created by you with meaningful names to help identify columns that have certain properties, so that you can refer to them collectively by the group name.

used for

Column groups can be annotated with a set of group usages. You can enter references to usages already defined in the column group container following the `used for` keyword shown in Figure 2.4. Press ⇥ on the « ... » to insert the first group usage. Make sure the usage is defined (its name should be visible in the `Define Usages:` section of the container (see Section 2.5 to lean how to create a new Group Usage). You can bind a group usage reference to a Group usage by using auto-completion (ctrl+␣ to locate the name, then pressing ⇥ to accept one completion choice), or by typing the name of the group usage directly (note that group usage names are case sensitive).

R Pressing ↵ before `<no name>` will not have the desired effect if you have not yet named a group. Make sure you name groups immediately after you create them.

2.5 Column Group Usage

A Column Group Usage can be defined by pressing ↵ either on top of the « ... » (immediately below `Define Usages:`), when the list of group usages is empty, or with the cursor immediately before the name of an existing group usage. When the list already contains one or more group usages, you can add more by placing the cursor over a group usage name and pressing ⇥. Keep pressing ⇥ to add more empty group usages.

Figure 2.6: **Content of a Sample Annotation Table** The `SampleID` column provides the name of the column to which the groups should be assigned. The `Groups` column provides the names of the groups, separated by a comma, which will be assigned to the column identified by `SampleID`.

Table

SampleID	Groups
gene	ID
DC0904	LPS=NO,counts
DC0907	LPS=NO,counts
DCLPS0910	LPS=NO,counts
DCLPS0913	LPS=NO,counts
A_DC	LPS=NO,counts
A_DC_LPS	LPS=YES,counts
B_DC	LPS=NO,counts
B_DC_LPS	LPS=YES,counts
C_DC	LPS=NO,counts
C_DC_LPS	LPS=YES,counts
C2DC	LPS=NO,counts
C2DCLPS	LPS=YES,counts
C3DC	LPS=NO,counts
C3DCLPS	LPS=YES,counts

2.6 Example Column Group Container

Figure 2.5 presents an example of a configured Column Group Container. This container defines two groups `LPS=yes` and `LPS=no`, which can be used to annotate Table columns that contain data about gene expression of samples treated with LPS or not. The group usage `LPS_Treatment` is associated to both groups to indicate that they belong together and are two kinds of treatment.

2.7 Column Groups from a Table

An alternative way to automatically create Column Groups and attach them to Columns at the same time is to use a so-called `annotation table`. These tables are normal Table nodes (created as specified 2.2) with a special structure and content. When created, MetaR is able to recognize them and allows the user to use these tables to annotate other tables.

The structure of an annotation table requires the following two columns:

- *SampleID*: values of this column are sample names
- *Groups*: values of this column are comma-separated lists of group names

When annotation tables are available in the current model, the user can

1. open the Table to annotate
2. use the "Use Column Groups From Table: <table name>" intention (💡) when the cursor is on top of the `Table` node

This way, Sample IDs are matched with the column names and the groups listed in the same row are attached to the corresponding `Column`. In addition, if the groups are not defined in the `ColumnGroupContainer`, they are created in the annotation process. The `Column GroupContainer` is also created if it does not exist.

To see how this intention works in practice, we will create the same groups shown in Figure 2.2 using an annotation table. To do that, we firstly need to create a TSV file with the content shown in Figure 2.6. The next step is to load this table in a Table node (Figure 2.7). Finally, we invoke the "Use Column Groups From Table: AnnotationTableFor GSE59364_DC_all.cvs" intention as shown in Figure 2.8 to apply the group names from the table to the destination Table. These steps will result in exactly the same annotated table shown in Figure 2.2.

```
Table AnnotationTableForGSE59364_DC_all.cvs ...
File Path
  ${org.campagnelab.metaR.home}/data/AnnotationTableForGSE59364_DC_all.cvs
Columns
  SampleID: string
  Groups: string
```

Figure 2.7: **Table with Samples and Groups.** An annotation table loaded as Table node.

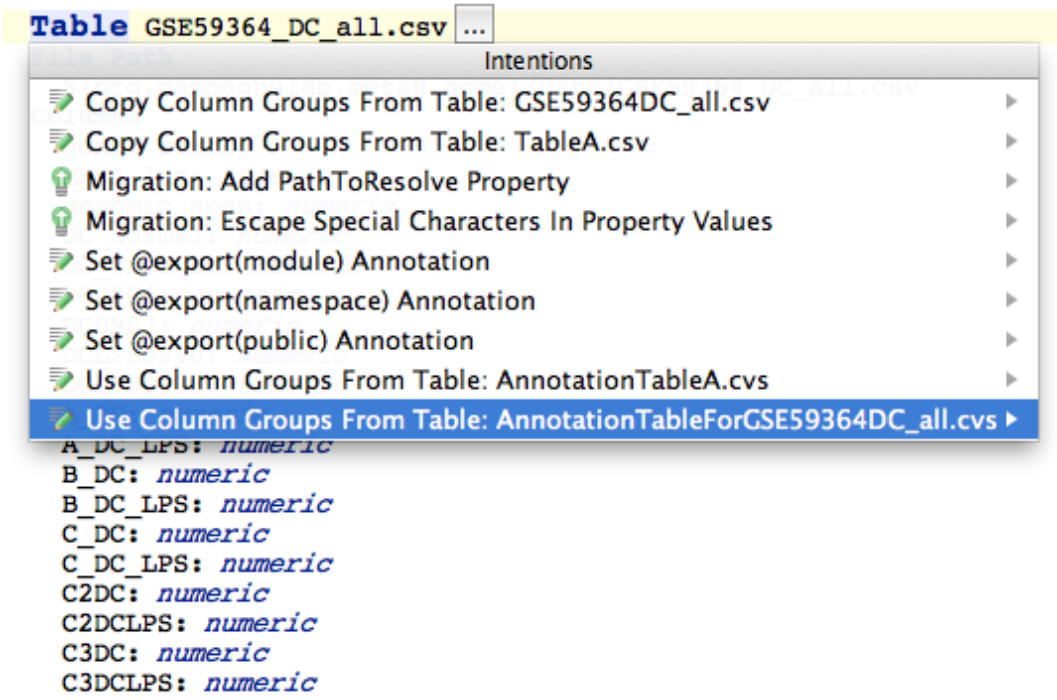


Figure 2.8: **Intention to Annotate a Table using another Table**

2.8 Column Group Annotations

Sometimes, a covariate is a continuous variable that can take different values across samples. This kind of covariate can also be used to estimate linear models (see Chapter 7 and 8).

MetaR 1.3 supports continous covariates. Instead of creating a Column Group for each possible value of the covariate, you can indicate that values of the column group are determined using a special type of table called *Covariate Table*. A covariate table is a normal `Table` node in metaR with the following characteristics:

1. a column annotated with a group called "sample-key",
2. values of this *sample-key* column must match the column/sample names of the primary table,
3. one or more additional columns from which covariate values are extracted.

Figure 2.9 shows a simple covariate table with two columns: the `SampleName` column is marked as the *sample-key*. The `Age` column provides the age covariate values, for each sample identified by SampleName.

```
Table AgeAnnotation.tsv...
File Path
  ${org.campagnelab.metaR.home}/data/AgeAnnotation.t
Columns
  SampleName: string [ sample-key ]
  Age: numeric
```

Figure 2.9: **Sample Covariate Table.** A covariate table must have a column annotated with the `sample-key` group, whose values are the sample names of the table that will be annotated with the Covariate table. The other columns hold the values that will be associated to each sample when they are selected.

A covariate table can be associated to a column group by pressing (at the end of the group in the editor or by using the intention shown in Figure 2.10. The annotation added to the group allows to refer to a covariate table in the current model and then select a column from that table with the covariate values (Figure 2.11).

2.9 Table Viewer Tool

R The TableViewer Tool was introduced in MetaR version 1.3.

The content of a Table can be inspected with the Table Viewer Tool distributed with MetaR. The tool adds to the MPS interface the capabilities to load the table's content and show it in a graphical context. Wherever a table name appears (in a Table node or in an Analysis script, see Chapter 3), the tool can be opened to see the rows and columns of that table along with their values. Figure 2.12 shows how to activate the tool on a Table node.

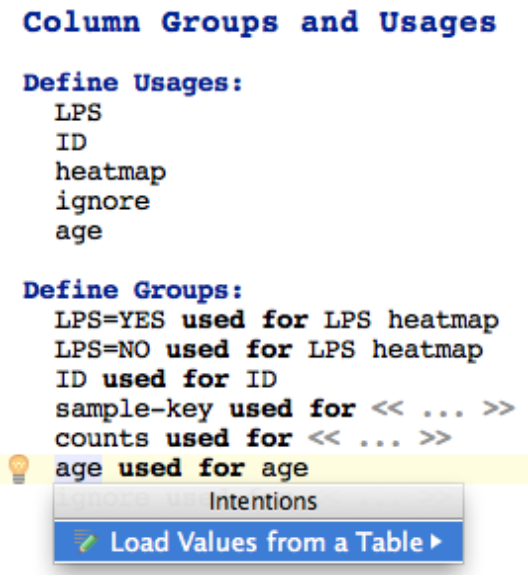


Figure 2.10: **Intention to Add a Covariate Table** Use this intention when the values of the Age covariate are provided by in a secondary table: the covariate table.

Figure 2.11: **Column Group Annotation** Using auto-completion (ctrl + ⇥ in the `no table` field, all the tables matching the covariate requirements are proposed. Once selected, a similar action can be used in the `no useCovariate` field to select a column from that table. Values from that column are matched with the samples when a statement needs to build a linear model.

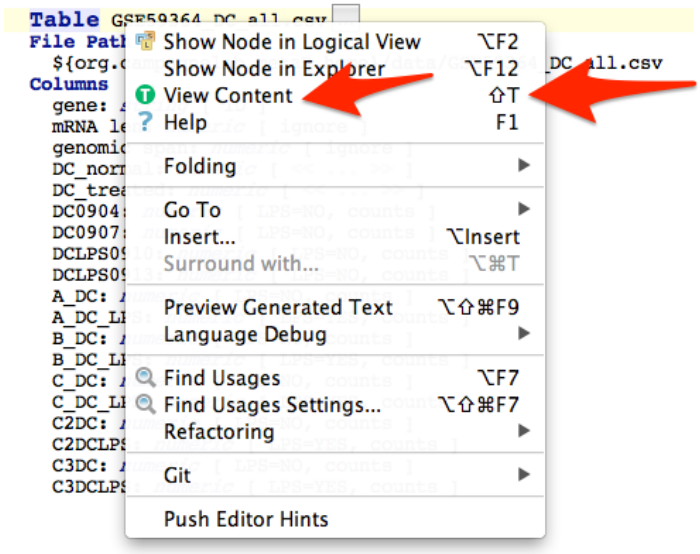


Figure 2.12: **How to activate the Table Viewer Tool**. The Table Viewer Tool can be activated by right-clicking on the Table and selecting "View Content" from the menu. An alternative way is to put the cursor on the Table node and pressing ⇧ + T.

The first time it is activated, the tool appears at the bottom of the MPS window, as shown in Figure 2.13.

The tool is immediately available for those tables directly loaded from the file system (e.g. Table nodes) and their references. Other Tables become visible after an Analysis script has been run and the content of the table has been created. In this latter case, the tool is available after the first script execution (see Chapter 3). Note that in this case, if the structure and/or content of a table changes, the Analysis script must be executed again before the tool will show the latest table data.

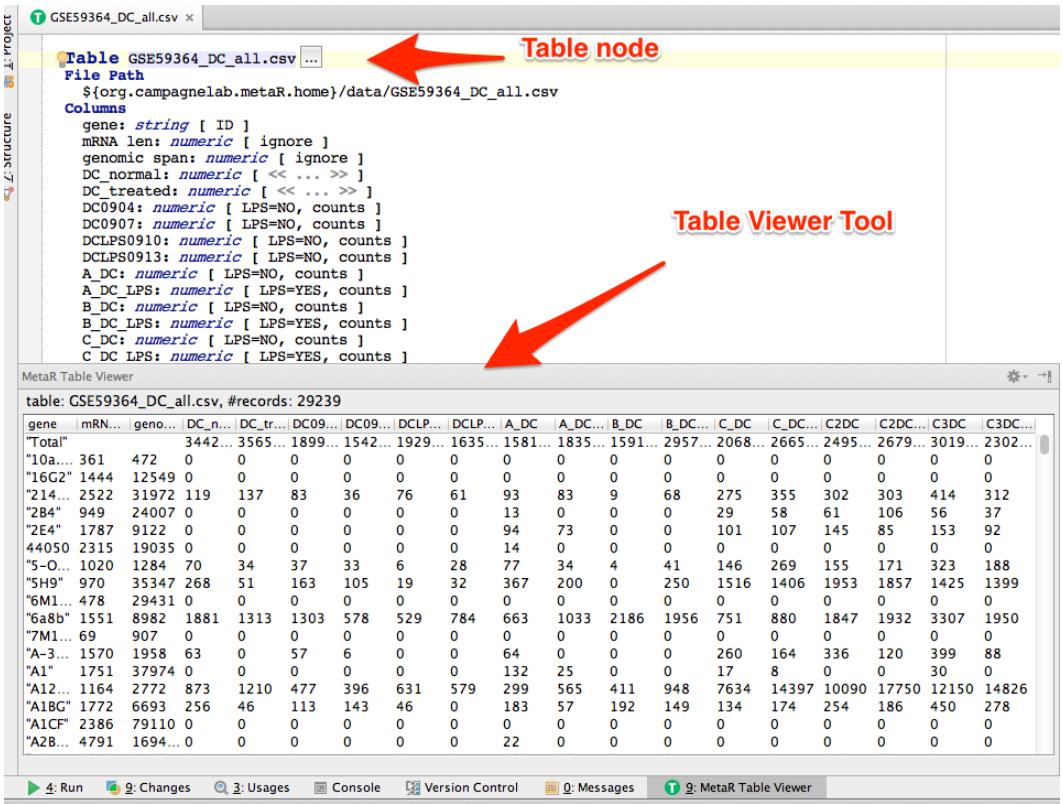


Figure 2.13: **The Table Viewer Tool in the MPS UI.**

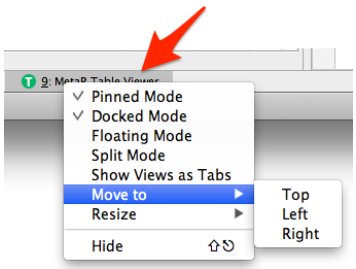


Figure 2.14: **Visualization Options for Table Viewer Tool**. Different visualization options are available for the Table Viewer Tool. By right clicking on the tool name in the Tool Buttons bar (if not visible, select View > Tool Buttons from the MPS menu), a user can explore them and choose the one that best fits a project needs. In our experience, the floating mode or the Docked/Pinned mode with the panel on the right are very effective.

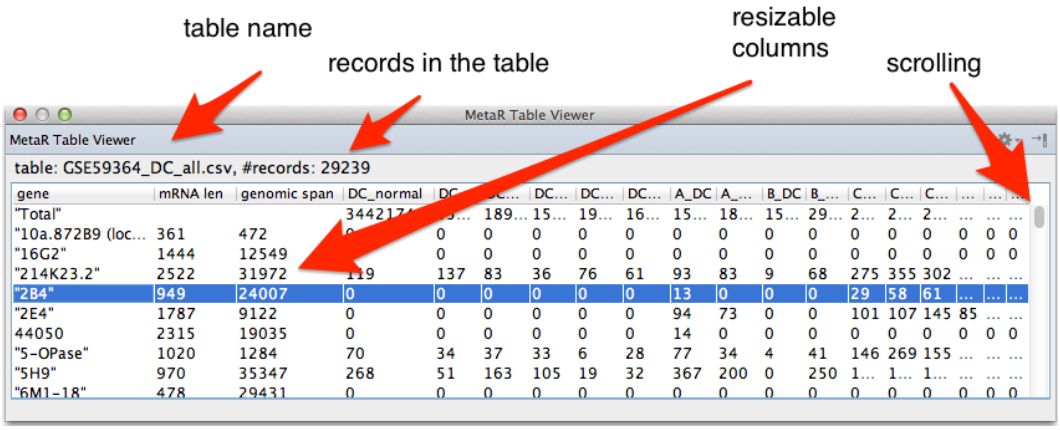


Figure 2.15: **The Table Viewer Tool**. The tool displays the content of the selected table. At the top, you can see the name of the table and the number of records loaded. Columns can be resized and rows scrolled for a full visualization of their values.

3 — Analyses

3.1 The MetaR Analysis Root Node

MetaR analyses are represented with an Analysis AST Root node. An analysis often imports one or more tables, performs data transformations and writes a table or generates some plots. You can create as many Analysis root nodes as needed in a model. You create an analysis by right-clicking on a model and selecting New ⟩ o.c.metar.tables ⟩ Analysis. Analysis can exist as direct child of a model, and for this reason is called a root node. Figure 3.1 presents a new Analysis root node.

name

Each analysis has a name. You should name the analysis after creating it. The name you enter will be shown in the Project Tab after the A icon.

statements

An analysis contains a list of statements. You can enter new statements by typing between the curly brackets `{ }`.

To enter a specific statement, you can type the `alias` of the statement (for instance `import table`). When you have typed a complete alias, the node will be inserted in place

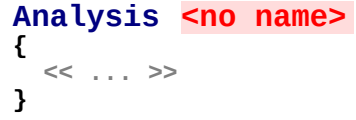


Figure 3.1: **New MetaR Analysis Root Node.** This figure presents a freshly created MetaR analysis root node. You can press ⏎ over the « ... » to add Statement nodes to the analysis. Use auto-completion to discover which types of statements are available.

of the alias. Note that there is no actual indication that the text you are typing matches a valid alias. You need to finish typing the full alias before the node is substituted for the alias. The text you type will remain red until the substitution occurs even if the text is matching a valid alias. See Figure 3.3.

The following sections describe the kinds of statements offered by the MetaR `org.campagnelab.MetaR.tables` language. The simplest way to learn which statements are supported by the release of MetaR that you are using is to use auto-completion. Figure 3.2 provides a snapshot of the auto-completion menu when looking for statements to insert in the Analysis statement list.

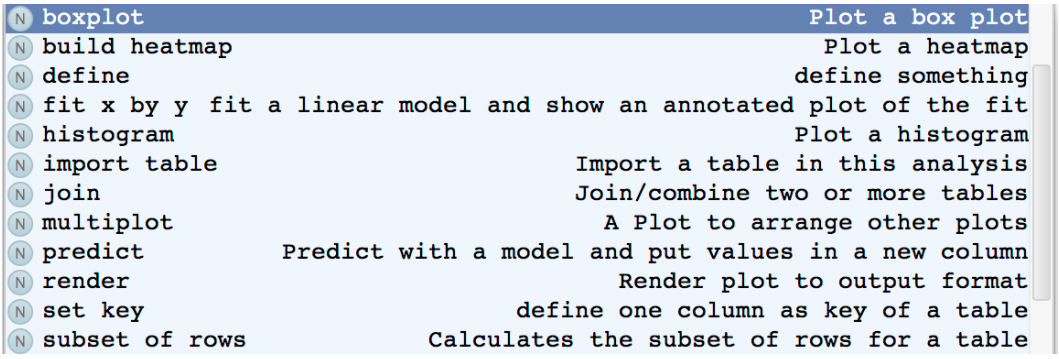


Figure 3.2: **Auto-completion Dialog for Statements.** This dialog is obtained by placing the cursor where a Statement is valid, and pressing ctrl+ .

```
Analysis Typing aliases tutorial
{
  import
  import table <no table>
}
```

Figure 3.3: **Typing Statement Aliases.** The user has typed "import" on the third line. This text is a prefix of the import table statement, but is shown in red because the alias is not yet complete. The fourth line shows the import statement which is substituted to the text when the user has just finished typing "import table", the complete alias of the statement. Note that pressing ⏎ immediately after import will create the node if the prefix "import" matches a single type of node in the languages imported in this model.

3.2 Styles

MetaR comes with a styles language that allows to customize the graphical aspects of plots and files generated by statements. A `Style` is represented by a root node in the model with a given name.

To create a Style, right-click on a model in the project tab and select right-click > New > o.campagnelab.styles > Style. This will create an empty Style, as shown in Figure 3.4.

```
Style <no name> extends <no extends> {
  << ... >>
}
```

Figure 3.4: **New Style.** A newly created Style AST Root node.

Style nodes are identified by a S icon. A Style is essentially a container for *style items*. Items are well-defined settings that are applied by MetaR during the rendering of a graphical object. Figure 3.5 shows a snapshot of the auto-completion menu when looking for items to add to a style. All possible style items are shown because this style is not yet bound to a specific statement.

```
Style MyNewStyle extends <no extends> {
}
```

```
Border color                (Color in o.campagnelab.styles)
Color                (ColorStyleClass in o.campagnelab.styles)
Color palette            (BaseConcept in o.campagnelab.styles)
Draw border      (BooleanStyleClass in o.campagnelab.styles)
Height           (IntegerStyleClass in o.campagnelab.styles)
Pixels per inch  (IntegerStyleClass in o.campagnelab.styles)
Title             (StringStyleClass in o.campagnelab.styles)
Width            (IntegerStyleClass in o.campagnelab.styles)
X label           (StringStyleClass in o.campagnelab.styles)
X range            (RangeStyleClass in o.campagnelab.styles)
Y label           (StringStyleClass in o.campagnelab.styles)
Y range            (RangeStyleClass in o.campagnelab.styles)
```

Figure 3.5: **Adding Style Items to a Style.** A Style is a container for Style Items. By using auto-completion ctrl + , style items can be added to the node.

Styles can extend other styles, which provides an ability to modularize appearance. This is useful if you need to build collections of plots, and most plots have a subset of style attributes, but the title of each plot or the X or Y variable changes. You would define the common style attributes in a style, and extend this style each time you need to specialize the style.

3.2.1 Binding Styles to Statements

In order to use a Style, you have to bind a statement to it. There are different approaches you can follow to create this binding and the choice depends on your needs and flexibility.

The simplest way to create a style and bind it to a statement is to use the intention "Create New Style" available on compatible statements as shown in Figure 3.6. This intention will

create a new style and bind it to the statement. This has the advantage that the style items are immediately restricted to only the items compatible with the statement.

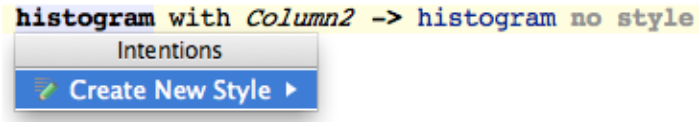


Figure 3.6: **Create New Style Intention on Statements.** The user has pressed option+⏎ on a statement that can be bound to a style. By selecting the **Create New Style** intention, a new default style is created in the model and bound to the statement.

Once bound to a statement (or more than one), the list of items that can be added to the Style is restricted to the ones compatible with the statement(s). For instance, if the style shown in Figure 3.5 is bound to a statement that uses only a Color Palette and some kind of border drawing, the auto-completion menu will appear as shown in Figure 3.7.

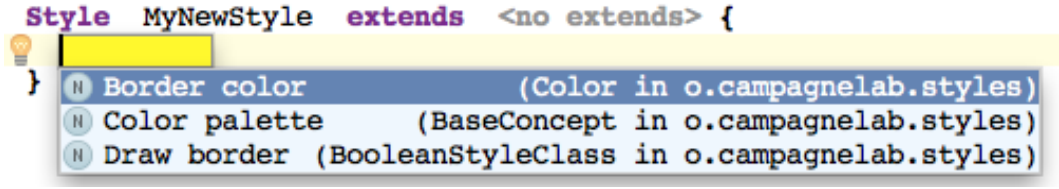


Figure 3.7: **Style with Restricted Items.** A subset of Style Items is displayed in the auto-completion menu when the Style is bound to a statement.

However, styles can be created independently from statements and bound at a later time. In this case, the list of styles visible to a statement in the auto-completion menu is limited to the compatible styles available (see Figure 3.8). A style is defined compatible with a statement if its items or a subset of them are used by the statement.

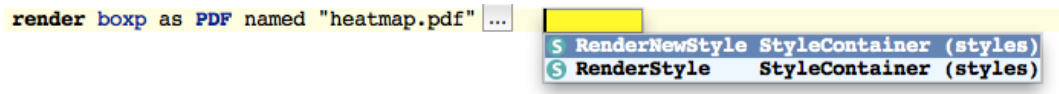


Figure 3.8: **Styles visible from a Statement.** A statement can be bound only to Styles that define items compatible with the statement.

A library of Styles could be created and reused across multiple Statements, Analyses or even Models. The capability to extends other Styles promotes reuse, reduces redundancies and allows to keep the number of Styles in a model at a minimum.

Figure 3.9: **New Write Statement.** Use this statement to write the content of a table to a file.

```
write <no table> to "<no path>" ...
```

3.3 Working with Tables

3.3.1 Import Table

The `import table` statement makes it possible to import a table, defined in the model, into the analysis. The columns of the imported table will become visible to the statements that follow the import. You can create an import table statement by typing the alias `import table` on an empty line of Analysis. Once you have bound the reference to a table, the import statement will look like this: `import table SomeData`. The name in green is the name of the table the statement imports. See Section2.2 to learn how to create a Table Root node.

table

The table attribute is a reference to an AST root node. You can set this reference by typing the name of the table you wish to import, or by using auto-completion Ctrl+␣ to locate the table AST root node.

3.3.2 Write Table

MetaR analyses can create new tables. You can write the data in these tables to a file using the `write` statement (see Figure 3.9 for a new write statement). The write statement has two attributes: table and filename.

table

This reference should be set to the table that you want to write to a file.

output

The output should be set to a filename where you want the data contained in the table to be written. Notice that output has a button to let you select the output filename.

3.3.3 Identify a Set of Columns

Several types of statements require the user to select one or more columns of a table. MetaR provides several ways to select a set of columns.

- You can use the `columns` node to identify a set of columns.
- You can use the `group` node to name a single group, and identify all columns annotated with this group.
- You can use the `groups` node to name a set of groups, and identify all columns annotated with any of these groups.

Each strategy identifies a set of columns, which may contain one or more columns.

```
define Set of IDs <no name> {
  << ... >>
}
```

Figure 3.10: **New Set of Ids Statement.** Use this statement to define a custom set of ids.

```
define Set of IDs data1 {
  a b c d e f g h i j k d5 d6 d7
}
```

Figure 3.11: **Example of a user defined Sets of Ids.** The set contains 14 elements with IDS named a-d7.

3.4 Define Sets of Ids

You can use this statement (alias `define Sets of Ids`) to define a custom set of ids. Figure 3.10 shows a new `define Sets of Ids` statement. Sets of ids can be used draw Venn diagrams (see Section 3.9.6), or to select rows of tables with the `subset` statement (see Section 3.5). To create a new set, use the `define` statement and choose `set of ids` with auto-completion.

name

The set of ids must have a name. The name is used to refer to this set of ids in other parts of the analysis.

ids

By pressing ⏎ with the cursor over the « ... » attribute you can add a new id in your set. To add another id, you just have to press again ⏎. Figure 3.11 shows an example of `Sets of Ids`.

3.5 Subset Rows

You can use this statement (alias `subset rows`) to filter a table and produce a new table with a subset of the rows of the input table. Figure 3.12 shows a new `subset rows` statement.

```
subset rows <no table> filter how?   -> subset
```

Figure 3.12: **New Subset Rows Statement.** Use this statement to filter the rows of a table.

```
{
  import table GSE59364_DC_all.csv
  define Set of IDs GeneList {
    MAPK PSEN1
  }
  subset rows GSE59364_DC_all.csv with IDs keep rows matching any ID in GeneList -> for geneslist

  subset rows GSE59364_DC_all.csv when true: $(genomic span) > 20000 -> genomic span>2000
}
```

Figure 3.13: **Subset Rows Examples.** This figure illustrates the two alternatives available for filtering rows: by expression (with `when true:`, or with a set of ID values).

table

The input table reference must be set to a table. The table must be either imported or produced by another statement.

filter

The filter determines which rows are kept. Use auto-completion to select one of the alternatives:

1. `when true:` will keep a row when the boolean expression following when `true:` evaluates to `true`.
2. `with IDS` will keep a row when the value of the column marked with group `ID` exists in the list provided as an argument. See the `define` statement to create a list of IDs in Section 3.4.

subset

The output table is called subset by default. Feel free to rename this table to better match the data in it.

3.5.1 Example

Figure 3.14 presents examples of subset row statements.

```
{
  import table GSE59364_DC_all.csv
  define Set of IDs GeneList {
    MAPK PSEN1
  }
  subset rows GSE59364_DC_all.csv with IDs keep rows matching any ID in GeneList -> for geneslist

  subset rows GSE59364_DC_all.csv when true: $(genomic span) > 20000 -> genomic span>2000
}
```

Figure 3.14: **Subset Rows Examples.** This figure illustrates the two alternatives available for filtering rows: by expression (with `when true:`, or with a set of ID values).

3.5.2 Boolean Expressions

One form of the subset rows statement accepts a boolean expression to determine which rows to keep. Boolean expression have one of two values: true or false, and can be constructed by comparing values with an operator. The following operators are supported by MetaR:

- $(column) The value operator evaluates to the value of the column in the row currently considered.
- expr1 == expr2 true when expr1 evaluates to the same value as expr2.
- expr1 != expr2 true when expr1 does not evaluate to the same value as expr2.
- expr1 | expr2 true when expr1 is true or expr2 is true (boolean or. either one of expr1 or expr2 needs to be true for the result to be true).
- expr1 & expr2 true when expr1 is true and expr2 is true (boolean and, both expre1 and epr2 must be true for the result to be true).
- expr1 < expr2 true when expr1 evaluates to a value less than expr2.
- expr1 > expr2 true when expr1 evaluates to a value greater than expr2.

```
join ( <no table> ) by      -> <no name>
```

Figure 3.15: **New Join Statement.** Use this statement to join two or more tables into one.

- expr1 **<=** expr2 true when expr1 evaluates to a value less or equal than expr2.
- expr1 **>=** expr2 true when expr1 evaluates to a value greater or equal than expr2.

R MetaR infers the type of the expressions that you enter and will report type errors if the value of some values are not compatible with the test that your perform.

3.6 Join Tables

When you perform data analysis, you often need to combine data from different tables into one. This operation is called table joining. MetaR provides a powerful `join` statement that helps join an arbitrary number of tables. The alias of the statement is `join`. Figure 3.15 presents a newly created `join` statement. Join has three attributes: input tables, by and output table.

input tables

Use this attribute to enter references to tables you need to join. You can press ⏎ with the cursor inside the parentheses to create references to additional input tables (either imported or created in the analysis up to that point).

by

Use the by attribute to specify how to join the input tables. Joining tables requires knowing which rows of the input tables go together. This is done by identifying a set of columns whose values must match across rows of the input tables that go together. See Section 3.3.3 to learn how to select groups of columns in MetaR. You can select one of the following joining strategies as value of the by attribute:

- **by columns:** The same column name must be defined in all the input tables.
- **by group:** If each table has exactly one column annotated with this group, the names of the columns do not need to be the same. Otherwise, if more than one columns is in the group, their names must match across all the tables.
- **by groups:** Similar to group, but with any (>=1) number of groups. Press ⏎ to insert new group references.

The group by columns are used to calculate a list of values. When rows of the input tables have the same group by values, they are joined in a single row, and the result put in the destination table.

R The join statement will create new columns if the input tables have columns with the same name. In this case the shared columns are renamed “colname.TableName”. Place the cursor on result and look at the column preview in the inspector to see which column names are generated by a given join statement.

result

The output table is called Results by default. Feel free to rename this table to match the content of the destination table. Open the 2: Inspector to see which columns will result from the join. Notice that the column groups are transferred to the columns of the output table.

3.6.1 How Join works

The output table produced by the `join` statement can be very different, depending on the strategy selected to join the input tables and their structure. The statement takes also care to produce unique columns from columns common to all the input tables but not used in the joining strategy. To explain how `join` works we will apply two different strategies to the input tables (named A and B) presented in Figure 3.16 and check the structure and content of the output table.

Table A

Gene	Column1[ID]	Column2
Ge1	Val1	34
Ge2	Val2	452
Ge3	Val3	113

Table B

Gene	Column2	Column3	Column4[ID]
Ge4	1	78	Val1
Ge2	3	13	Val3
Ge1	4	44	Val4

Figure 3.16: **Sample Input Tables for Join Statement.**

Do note that the two tables have columns in common (Gene and Column2) and that each table has a column annotated with a group named ID (Column1 in table A, Column4 in table B).

join by column

As first strategy, we join A and B by the column Gene. The results table generated by `join` is shown in Figure 3.17.

Table Results
(by Column Gene)

Gene	Column1	Column2.A	Column2.B	Column3	Column4
Ge1	Val1	34	4	44	Val4
Ge2	Val2	452	3	13	Val3

Figure 3.17: **Results Table for Join using by Column Strategy.**

If we analyze the structure of the table, we note that:

- there is only one Gene column (the one used for joining)
- Column2, that was present in both the input tables, was split in two columns named Column2.A and Column2.B
- the rest of the columns are the same coming from their respective input tables

Looking at the table's content, we note that:

- rows from input tables with a matching value in the Gene column were merged into a single row.

- all the other columns keep their original value from the source table with respect to the Gene column's value

join by group

Our second strategy is to join A and B by group ID. As shown in Figure 3.16, there are two columns, one in each input table, annotated with such a group and they have different names. The `join` statement is able to work even in this case by matching the values of these two columns. The results table generated by Join is shown in Figure 3.18. Again, let's go over the structure of the table:

- Gene, that was present in both the input tables but this time not used as joining column, was split in two columns named Gene.A and Gene.B
- Column2, that was present in both the input table, was again split in two columns named Column2.A and Column2.B
- the rest of the columns are the same coming from their respective input tables

Table Results
(by Group ID)

Column1	Gene.A	Gene.B	Column2.A	Column2.B	Column3	Column4
Val1	Ge1	Ge4	34	1	78	Val1
Val3	Ge3	Ge2	113	3	13	Val3

Figure 3.18: **Results Table for Join using by Group Strategy.**

Looking at the table's content, we note that:

- rows from input tables with a matching value in Column1 from A and Column4 from B were merged into a single row
- all the other columns keep their original value from the source table with respect to their matching column's value

3.6.2 Example Join

```
{
  import table GSE59364_DC_all.csv
  import table another Table
  join ( GSE59364_DC_all.csv , another Table ) by group ID -> Results
}
```

Figure 3.19: **Example of Join Statement.**

Figure 3.19 presents an example of join statement. The statement joins two tables and creates the Result table. If you open the 2: Inspector, you will see a preview of the columns for the result table (shown in Figure 3.20).

3.7 Transform Table

The statement transform table (alias `transform`) allows to transform columns of an input table and produce a new table. Figure 3.21 shows a new `transform` statement.

table

Use this attribute to select a table to transform. You can press ctrl + [ ] to open the autocompletion menu and display all available tables.

operations

A set of table operations, which can be used to transform the input table. See Figure 3.22 for a list of operations available to transform a table. Operations include:

drop column This operation allows to delete a specific column. Press ctrl + [ ] to open the autocompletion menu and display all available columns. Then press ↵ to select one column. This column will be removed/dropped in the result table.

```
path= /Users/fac2003/R_RESULTS/table_Results_0.tsv  name: Results table.name Results groups= LPS=NO,counts,LPS=YES,ID
Columns (38) :
  C3DC.GSE59364_DC_all.csv : numeric  [ LPS=NO, counts ]
  C3DC.another Table : numeric  [ LPS=NO, counts ]
  C2DC.another Table : numeric  [ LPS=NO, counts ]
  DC0904.another Table : numeric  [ LPS=NO, counts ]
  mRNA len.another Table : numeric  [ << ... >> ]
  B_DC.GSE59364_DC_all.csv : numeric  [ LPS=NO, counts ]
  mRNA len.GSE59364_DC_all.csv : numeric  [ << ... >> ]
  B_DC.another Table : numeric  [ LPS=NO, counts ]
  A_DC.GSE59364_DC_all.csv : numeric  [ LPS=NO, counts ]
  DCLPS0913.GSE59364_DC_all.csv : numeric  [ LPS=NO, counts ]
  DC0904.GSE59364_DC_all.csv : numeric  [ LPS=NO, counts ]
  C2DC.GSE59364_DC_all.csv : numeric  [ LPS=NO, counts ]
  C_DC_LPS.GSE59364_DC_all.csv : numeric  [ LPS=YES, counts ]
  DCLPS0910.another Table : numeric  [ LPS=NO, counts ]
  B_DC_LPS.another Table : numeric  [ LPS=YES, counts ]
  DC_treated.another Table : numeric  [ << ... >> ]
  A_DC_LPS.another Table : numeric  [ LPS=YES, counts ]
  gene.GSE59364_DC_all.csv : string  [ ID ]
  DC_normal.another Table : numeric  [ << ... >> ]
  C3DCLPS.GSE59364_DC_all.csv : numeric  [ LPS=YES, counts ]
  C_DC_LPS.another Table : numeric  [ LPS=YES, counts ]
  B_DC_LPS.GSE59364_DC_all.csv : numeric  [ LPS=YES, counts ]
  C2DCLPS.GSE59364_DC_all.csv : numeric  [ LPS=YES, counts ]
  DCLPS0910.GSE59364_DC_all.csv : numeric  [ LPS=NO, counts ]
  A_DC_LPS.GSE59364_DC_all.csv : numeric  [ LPS=YES, counts ]
  DC_normal.GSE59364_DC_all.csv : numeric  [ << ... >> ]
  C3DCLPS.another Table : numeric  [ LPS=YES, counts ]
  C_DC.another Table : numeric  [ LPS=NO, counts ]
  genomic span.GSE59364_DC_all.csv : numeric  [ << ... >> ]
  C_DC.GSE59364_DC_all.csv : numeric  [ LPS=NO, counts ]
  C2DCLPS.another Table : numeric  [ LPS=YES, counts ]
  genomic span.another Table : numeric  [ << ... >> ]
  A_DC.another Table : numeric  [ LPS=NO, counts ]
  DC_treated.GSE59364_DC_all.csv : numeric  [ << ... >> ]
  gene.another Table : string  [ ID ]
  DCLPS0913.another Table : numeric  [ LPS=NO, counts ]
  DC0907.another Table : numeric  [ LPS=NO, counts ]
  DC0907.GSE59364_DC_all.csv : numeric  [ LPS=NO, counts ]
```

Figure 3.20: **Column Preview for Result Table.** Notice that all these columns were shared across the input tables because their name follows the pattern "colName.tableName".

```
transform table<no table> -> transformedTable{

}
```

Figure 3.21: **New Transform Table Statement.** Use this statement to transform a table by applying column operations.

```
transform tableGSE59364_DC_all.csv-> transformedTable {
  drop columnA_DC
  drop columns which match span
  drop column which have group LPS=YES
  rename column:A_DC -> newName
}
```

Figure 3.22: **Example of Transform Table Statement.** This example show various table operations available with `transform table`.

drop columns which match This operation allows to delete columns which contains a specific pattern. Type the pattern after the `match` keyword.

drop columns which have group This operation allow to delete columns which have a specific group. To do so, press first ctrl + space to display the available group and ↵ to select one group.

add column This operations allows to add a column in the result table. Start by entering the name of the new column. The second argument must contain the expression that will calculate the value that the new column will take for each row of the result table. You may reference columns of the input table in the expression using the `$` column selection helper.

rename this operation allows to rename a specific column from the input table. Once you have selected your column of interest, you just need to type a new name under the *(newName)* argument.

result

The output table is called TransformedTable by default, but can be renamed as needed. Use the inspector to see which columns will be produced when the transform statement is executed.

3.8 Block with Selected Tables

Sometimes it can happen that many tables are created during your analysis and, at one point, you might want to work with a restricted set of tables. The statement `with tables` allows you to select specific tables defined in your analysis and execute metaR statements on them. Figure 3.23 presents a newly created `with tables` statement. All tables created inside the block will be available outside the block. The statement has two attributes: the list of input tables and a statement list. Figure 3.24 shows an example of the `with tables` statement.

input tables

Use this attribute to select the tables you want to see inside the block. To do so, you can use auto-completion and select tables defined in your analysis up to that point. The statements included in the statement list will only see the tables defined in this list.

statement list
Here you can use all statements defined in MetaR. Any tables or plots created in the statement list will be visible after the block ends.

```
with tables ( << ... >> ) for statement:
{
  << ... >>
}
```

Figure 3.23: **New With Tables Statement.** Use this statement to work with a restricted set of tables.

```
with tables ( GSE59364_DC_all.csv filtered ) for statements
{
  subset rows filtered with IDs keep rows matching any ID in P-value -> subset
  join ( subset, GSE59364_DC_all.csv ) by group ID -> newjoin
}
```

Figure 3.24: **Example of With Tables Statement.**

3.9 Plotting Data

MetaR provides simple plotting capabilities[1]. The following types of plots are currently supported:

- boxplot alias `boxplot`
- histogram alias `histogram`
- scatterplot: alias `scatterplot` and `fit`
- heatmap, alias `heatmap`
- multiplot alias `multiplot`, makes it possible to organize other plots in a matrix of n columns by n rows.
- Venn Diagrams, alias `venn`
- UpSet plot, alias `UpSet` are a modern alternative to Venn Diagrams and make it easy and effective to compare intersection of several sets of items.
- MA Plot, alias `MA Plot` are a type of quality control plot useful when looking at high-throughput gene expression assays.
- T-SNE plots, alias `t-SNE`, are scatter plots following T-SNE dimensionality reduction.

Each type of plot statement will create a plot, identified with the icon when you are trying to auto-complete a reference to a plot. Plot names are also colored with the same dark blue as the icon.

Generated plots can be customized and refined by binding each statement to a style (see 3.2). A reference to a style is back-colored with the same turquoise color as the icon.

[1]These capabilities are simple, but can be extended very easily by adding new statements to draw other types of visualization. This is a key advantage of using Language Workbench technology. See Section 14 to learn how to create new statements.

```
boxplot with <no col> -> <no name> no style
```

Figure 3.25: **New Boxplot Statement.**

3.9.1 boxplot

Figure 3.25 presents a new boxplot statement.

columns

Indicate one or more columns to plot. The values of the columns will be plotted as individual boxplots in the same graph. Press ⌅ to define more than one column. Use auto-completion to locate individual columns from the imported tables, or the tables created by prior statements.

plot

The attribute after `->` is a plot. Enter a name for the boxplot here. Plot names are colored blue to make them easier to recognize.

style

If a style is bound to the box plot (see section 3.2), the statement will use a `ColorPalette` item from the style to draw the plot. A Color Palette is an AST Root Node identified with the icon.

MetaR comes with several pre-defined palettes and colors ready for use. Figure shows the auto-completion menu for a Color Palette item listing the default palettes available.

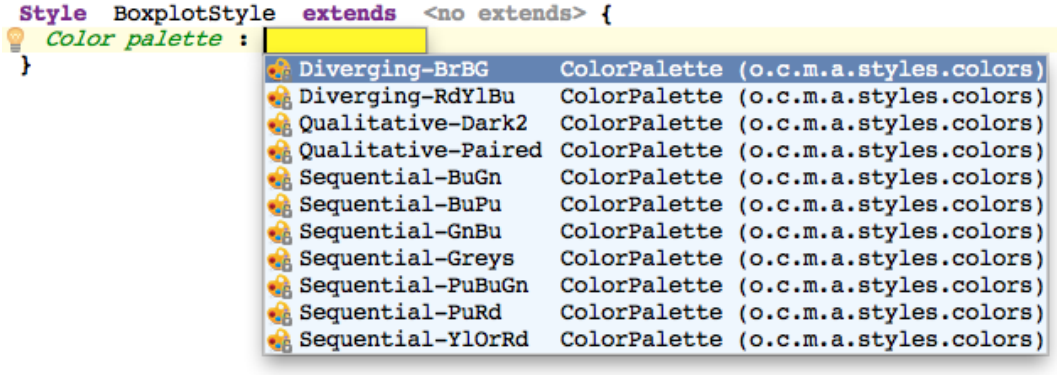


Figure 3.26: **Auto-completion menu with default Color Palettes.**

You can create a new palette by right-clicking on a model and selecting New ⟩ o.campagnelab.styles ⟩ Color Palette. `Colors` can be added to a palette and their order will matter in the rendering of the plot. Figure 3.27 shows a Color Palette with some color items and the auto-completion menu listing some of the default colors available.

There are two classes of colors you can use in a palette:

- **Named `Colors`.** New named `Colors` can be created by right-clicking on a model and selecting New ⟩ o.campagnelab.styles ⟩ Color. A color is identified by the

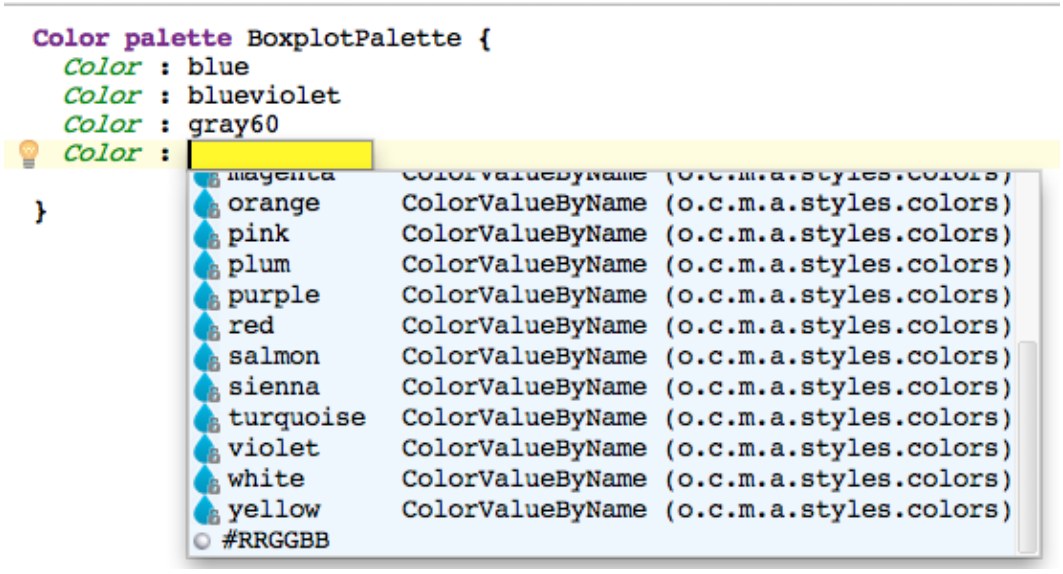


Figure 3.27: **Auto-completion menu showing some default Colors and the ##RRGGBB option.** The auto-completion menu lists the visible `Colors` and `#RRGGBB` option. By selecting this latter, you can create a custom color based on the Red Green Blue color components (RGB).

icon and its name must be valid in the R language.

- **`#RRGGBB` Colors.** These colors can be created directly inside the Color Palette by selecting the `#RRGGBB` option in the auto-completion menu for a Color value. The color will be constructed from the combination of the Red, Green and Blue specified in the code.

3.9.2 histogram

Use this statement to plot a histogram of the values of one column.

column

Indicate one column to plot a histogram for. Use auto-completion to locate the column from the imported tables, or the tables created by prior statements.

plot

The attribute after `->` is a plot. Enter a name for the histogram here. Plot names are colored blue to make them easier to recognize.

style

The statement uses a `ColorPalette` item (see 3.9.1) from the associated style (see section 3.2) to draw the plot.

3.9.3 scatterplot

Use this statement (alias `scatterplot`) to render a scatter plot of y (one column) vs x (another column). This will create a simple scatter plot with the option of using a column of the table to set the color of the points. Figure 3.28 presents a new `scatterplot` statement.

```
scatterplot <no table> x: <no col> y: <no col> color: <no color> -> <no name>
```

Figure 3.28: New Scatterplot.

x, y

Indicate which columns should be plotted. Use auto-completion to locate the x and y columns from the imported tables, or the tables created by prior statements.

plot

The attribute after `->` is a plot. Enter a name for the scatterplot here. Plot names are colored blue to make them easier to recognize.

color

The color attribute is optional. Choose a column whose value will determine the color of each point.

3.9.4 fit x by y

Use this statement (alias `fit`) to plot a scatter plot of y (one column) vs x (another column). A linear fit is performed, and the r^2 adjusted, and P-value corresponding to the linear regression is shown on the plot. Figure 3.29 presents a new `fit` statement.

```
fit <no col> by <no col> with table <no table> -> <no name> no style
```

Figure 3.29: **New Fit X by Y.**

x, y

Indicate which columns should be plotted. Use auto-completion to locate the x and y columns from the imported tables, or the tables created by prior statements.

plot

The attribute after `->` is a plot. Enter a name for the scatterplot here. Plot names are colored blue to make them easier to recognize.

style

The fit statement uses the following items from the associated style (see section 3.2) to customize the scatterplot:

- **`Title`**. The main title on top of the plot.
- **`X label`**. A title for the x axis.
- **`Y label`**. A title for the y axis.

- **`X range`**. Range of values for the x axis.
- **`Y range`**. Range of values for the y axis.
- **`Width`**. Width in pixels of the output image with the plot.
- **`Height`**. Height in pixels of the output image with the plot.

3.9.5 heatmap

Heatmaps are frequently used to visualize high-throughput data [**Cook2007**]. Use the `heatmap` statement (alias `heatmap`) to construct a heatmap plot. Figure 3.30 presents a new `heatmap` statement.

```
heatmap with <no table> select data by<no dataSelection>-> <no name> no style [
]
```

Figure 3.30: **New Build Heatmap.** Notice the intention "Add Annotations" attached to the statement. You can use this intention to further customize the heatmap.

table

Specify the table that contains the data that will be used to draw the heatmap. Note that the table must meet certain conditions. An error message will be displayed if these conditions are not met:

- Some columns of the table must be annotated with at least one group whose usage is "heatmap". Such columns are used to provide data values for the heatmap. If you don't have a heatmap usage, just create one and add it to the groups you would like to include on the heatmap.
- The columns of the table must be annotated with groups and group usages to make it possible for you to use these group usages to construct a legend.

select data by

Use this attribute to customize the set of columns to plot on the heatmap. See Section 3.3.3 to learn how to select a set of columns.

plot

The `<no name>` attribute listed after `->` makes it possible to name the plot that will hold the heatmap. Use any name you like. This name will be used to refer to the heatmap plot, for instance if you wish to assemble panels of different heatmaps into one figure.

style

The statement uses the following items from the associated style (see section 3.2) to customize the heatmap:

- **`Color palette`**. The colors to use in the heat map (see 3.9.1).
- **`Draw border`**. A boolean value to enable or disable borders in the heatmap. A value of `true` will draw a border. A value of `false` will not. By default, borders are enabled.

- **`Border color`**. The color to use for the border (see section 3.9.1 to check out how to create/refer to colors). If not set, the default color is *grey60*.

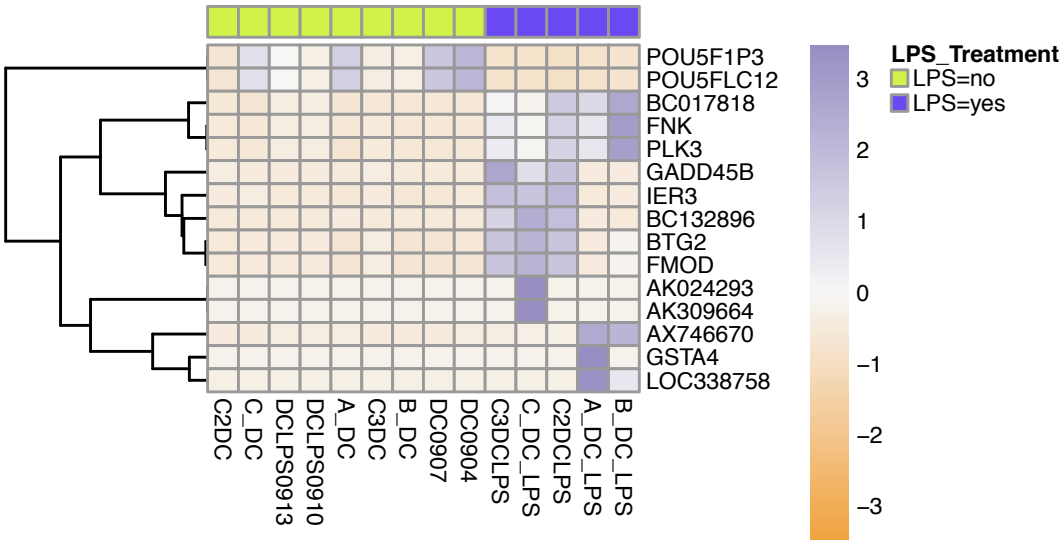


Figure 3.31: **Example of Heatmap Plot.** This heatmap has been customized with annotations. The `LPS_Treatment` group usage is shown in the legend, with two groups: `LPS=yes` and `LPS=no`. The values plotted have been scaled by rows, and the rows clustered. Data are from `http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE59364`.

annotations

You can use the "Attach Annotations" intention () when the cursor is on top of `heatmap` to customize heatmap annotations. Figure 3.32 shows a heatmap statement with annotations.

```
heatmap with GSE59364_DC_all.csvselect data bycolumns: A_DC B_DC -> heatmap HeatmapStyle [
  annotate with these groups:<< ... >>
  scale values: <no scaling>
  cluster columns: false cluster rows: false
]
```

Figure 3.32: **Build Heatmap With Annotations.** Annotations are shown after you have triggered the "Add Annotations" intention.

annotate with these groups. You can use this attribute to select usage names that should be listed in the heatmap legend. Listing a group usage will annotate each column of data with the group that is associated to the usage.

scale values. You can choose to scale values by column or by row. Scaling will make differences easier to see by using colors more effectively.

cluster columns: You can choose to cluster by colors or by rows by changing the boolean values accordingly.

Example

Figure 3.31 presents a heatmap constructed with the build heatmap statement, using annotations and row clustering.

3.9.6 Venn diagram

The Venn diagrams were introduced in MetaR version 1.3.

Venn diagrams are useful to show how many elements exist in various intersections among sets of elements [**venn1880diagrammatic**]. Use the `Venn diagram` statement (alias `venn`) to draw a Venn diagram plot. Figure 3.33 presents a new `Venn diagram` statement.

```
Venn diagram of {  set <no name> from set of ids <no oneSetOfIds> with default color  } -> <no name>
```

Figure 3.33: **New Venn Diagram.** Up to five sets can be shown on a Venn diagram. The name of each set will be shown on the Venn diagram plot.

```
Venn diagram of {  set set1 from set of ids data1 with Color : aquamarine                    } -> <no name>
                   set <no name> from table <no table> when true: <expression> with default color
```

Figure 3.34: **Sets type of Venn Diagram.** Here you can see the two types of sets which can be used for a Venn diagram: a user defined set and a set from an annotated table.

name

The `name` attribute can be specified immediately after the `set` keyword. A set must have a name, which will be shown on the Venn diagram to identify this particular set.

set

Up to five sets can be shown on a Venn diagram. The data set are divided in two types:

ids from a user defined set contains only one attribute: `set of ids`. The `set of ids` attribute must refer to a user defined set. see Section 3.4 to learn how to define new sets of ids.

ids from an annotated table contains two attributes:

- `table`. This attribute must refer to a table annotated with an *"ID"* group on a column that will provide identifiers for the set elements.
- `expression`. This expression defines how to select rows of the table to extract ids for the set. The expression must return true when a row of the table is part of the set. When this is the case, the value of the column marked with the ID group is extracted and added to the elements of the set.

R Notice that you can turn one type of ids into the other with auto-completion. place the cursor on top of the `set` keyword and invoke ctrl + SPACE to switch the type of set. Notice that the name and color attributes, if defined, are preserved.

color

The `default color` attribute listed after **with** makes it possible to customize the color used to draw the set on the Venn diagram. By using auto-completion, a menu listing the default colors available will appear. Default colors will be used for each set where a color has not been defined.

plot

The `<no name>` attribute listed after `->` makes it possible to name the plot that will hold the Venn diagram. Use any name you like. This name will be used to refer to the Venn diagram plot.

Example

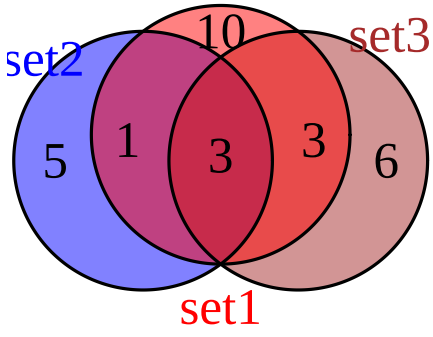


Figure 3.35: **Example of Venn Diagram with Three Sets.**

Figure 3.35 presents an example of a Venn Diagram showing three sets.

3.9.7 multiplot

This statement (alias `multiplot`) makes it possible to assemble a plot as a matrix of m x n plots. This is convenient if you would like to create a figure from panels of individual plots. In addition, multiplot provides a preview of the multi-panel plot that you can turn on and off at the click of a button. Figure 3.36 presents a new Multiplot.

plot

The plot name is shown immediately after `->` (initially `<no name>`). Name the mutliplot to be able to refer to it from other statements (such as render to make a PDF from it).

```
multiplot -> <no name> [  <no numColumns> cols x <no numRows> rows ]   Preview
<<emptyTable>>
```

Figure 3.36: **New Multiplot Statement.** Click on the Preview/Hide Preview button to toggle the plot preview.

m cols x *n* rows

Define the number of columns and rows that you wish the multiplot to have. The product of *m* and *n* determines how many plot references must be filled in to construct the multiplot.

Preview/Hide Preview

These buttons will make it possible to preview/ hide the preview for the multiplot. Note that a preview is only available after you have run the analysis. If you don't see the image being refreshed after running the script, remember to hide the preview and show it again to refresh.

plot references

After you set the number of columns and rows, you need to link *m* x *n* references to plots that you have already constructed. Do this in table attribute (shown as «`emptyTable`» initially).

Multiplot Example

Figure 3.37 presents an example of multiplot.

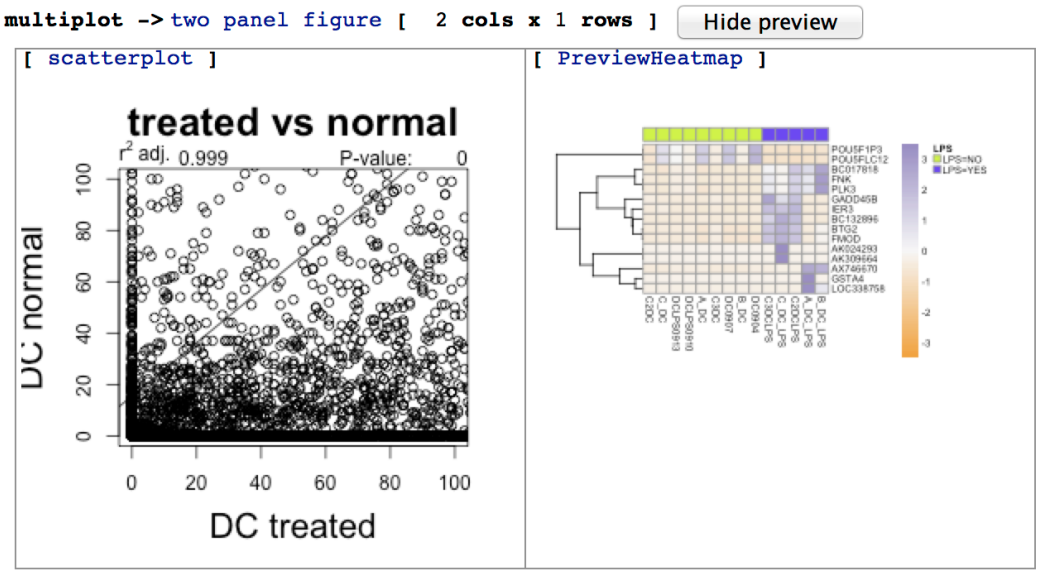


Figure 3.37: **Example of Multiplot.** This example shows a multiplot composed of two columns and one row. In the preview, a fit x by y plot is shown on the left, and a heatmap on the right.

3.9.8 render

This statement (alias `render`) makes it possible to save a plot or a multiplot in a local file according to a selected output format and style. Figure 3.38 presents a new Render.

Figure 3.38: **New Render Statement.**

plot

The `<no plot>` attribute listed after the alias allows you to select the plot or multiplot to render.

rendering format

The rendering format is the format in which the plot will be stored on the file system. The current version of MetaR allows to use only PDF as format.

path

The `<no path>` attribute has to be set to a filename where you want the rendering is stored. Its extension must be compatible with the selected format (e.g. ".pdf" for PDF). If only a filename is specified

3.9.9 UpSet plot

UpSet plots have been introduced in [**Lex2014**] and are supported in MetaR since release 1.8. Such plots make it effective to get a sense of the intersection among many sets. Figure 3.39 presents an illustration. In MetaR, UpSet plots can be created with the UpSet statement. The statement accepts the definition of an arbitrary number of sets to be compared, and produces a plot. The sets can be defined as sets of IDs, or as predicated over a table (see Figure 3.40).

3.9.10 MA Plot

MA plots are a useful quality control type of plot. An MA plot is a scatterplot of log fold change versus intensity of expression. The plot is useful to detect systematic bias in fold change across genes. It has been popularized with microarray technology which used to have regions of the array that could affect many gene expression estimates and show bias. Figure 3.42 (top) presents an MA plot constructed in MetaR. Such a plot can be created with the `MA Plot` alias, available when you import the *org.campagnelab.metar.plots* language. See Figure 3.41 for a newly created MA Plot statement. The bottom of Figure 3.42 illustrates how individual points of an MA Plot can be labeled.

Table

MA plots take data from a MetaR table. Enter a reference to a table imported in the analysis.

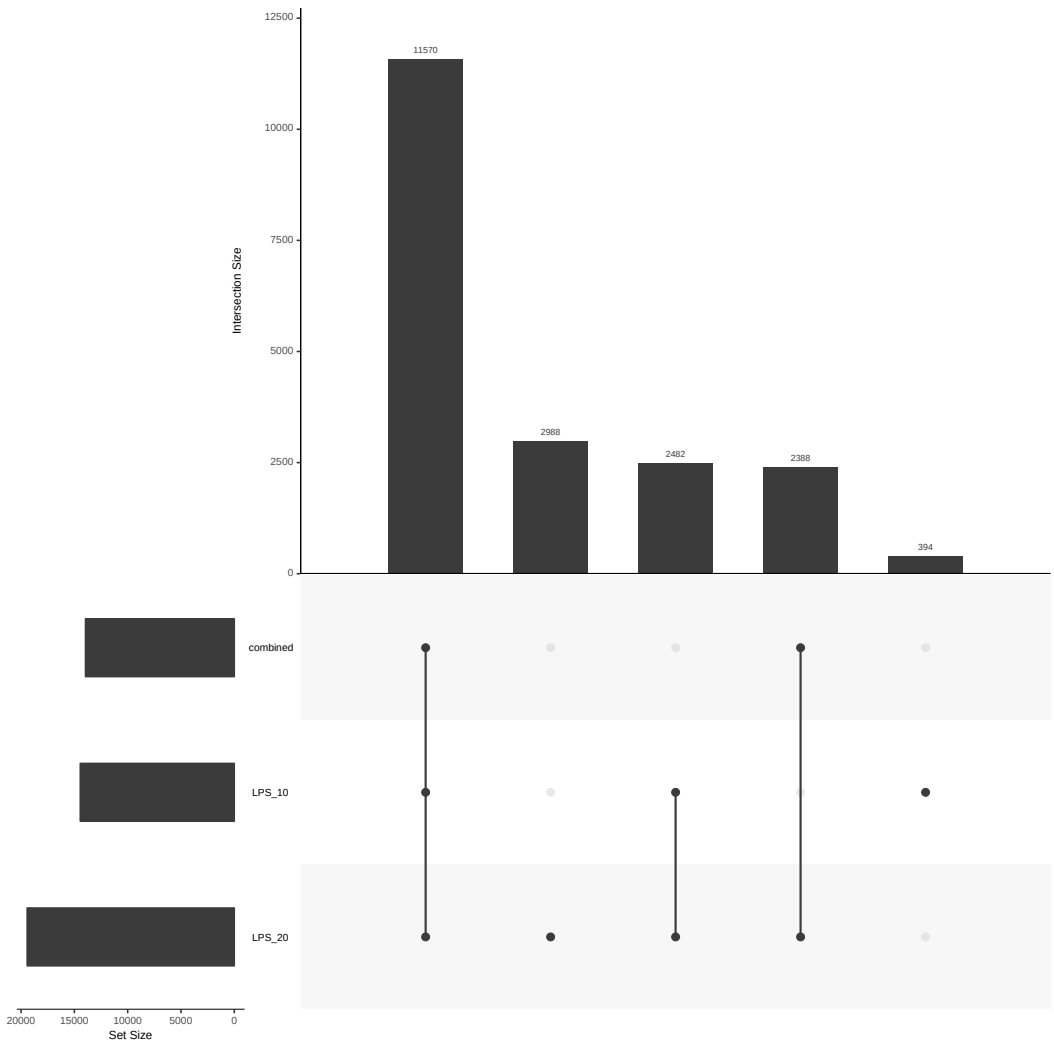


Figure 3.39: **UpSet Example Plot.** This figure illustrates an upset plot. It was constructed with three sets derived form a table (construction shown at the bottom of Figure 3.40.

```
Analysis UpSetTutorial
{
  // with defined gene lists:
  define Set of IDs List1 {
    A B C
  }
  define Set of IDs List2 {
    A B D E F
  }

  UpSet { set List1 } -> plot
          set List2
  // with elements of a table:
  import table GSE59364_DC_all.csv
  UpSet { ids from table when true: GSE59364_DC_all.csv$A_DC_LPS > 20 name of set: LPS>10                    } -> upset
          ids from table when true: GSE59364_DC_all.csv$A_DC > 10 name of set: LPS>20
          ids from table when true: GSE59364_DC_all.csv$A_DC > 10 & GSE59364_DC_all.csv$C2DC > 30
              name of set: combined
  render upset as PDF named "upset.pdf" ... no style
}
```

Figure 3.40: **UpSet Plot Construction.** This figure illustrates the two methods available to construct UpSet plots. At the top, we use defined sets of IDs and simply reference them to create the plot. At the bottom of the analysis, we import a table and define each set with an expression over columns of the table. The expression must evaluate to a boolean which indicates if the row of the table is included in the set, or not.

```
MA plot with stats from <no table> red when FDR<= 0.20 label set: <no geneList> with <no geneName> -> <no name>
    <no useStyle>
```

Figure 3.41: **MA Plot Statement.** A new MA Plot statement is shown before configuration.

FDR threshold

You can adjust this threshold to determine which genes/elements of measurement are considered significantly changes, and should be colored red on the MA plot.

Label set

Use this attribute to reference a gene list (MetaR set of ids) that identifies genes to be labeled. The identifiers provided in the gene list must match those found in the table marked with group `ID`.

Plot

The attribute shown as `<no name>` in Figure 3.41 let you define a name for the plot that will be produced by the statement. This name can be used to combine the MA Plot with other plots in a multiplot statement (see Section 3.9.7).

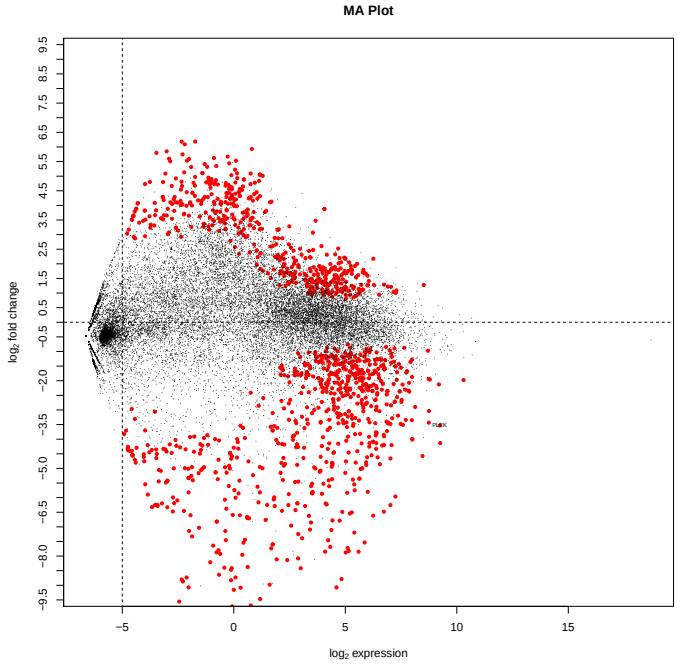
Style

See Section 3.2 for an introduction to styles. The MA plot accepts the following style items:

- `XRange` Determines the range of the X axis.
- `YRange` Determines the range of the Y axis.
- `Title` Let's you configure the title of the plot

Inspector

The inspector offers several attributes to customize the MA Plot:




```
{
  define Set of IDs GeneList {
    PLEK
  }
  MA plot with stats from Results red when FDR<= 0.20 label set: GeneList with ID -> MA MAPlotStyle
  render MA as PDF named "MA.pdf" ... no style
}
```

Figure 3.42: **Example of MA Plot.** An MA plot constructed with MetaR is shown. The top part of the figure shows the plot. The second part shows how the plot is created in MetaR. Note that in this example, a gene list is used to identify a gene to be labeled on the MA Plot. The horizontal dashed line represents no change in gene expression (fold change = 1, log_2 fold change = 0). The vertical dashed line represents the minimum gene expression threshold filter.

Columns

A first group of attributes help customize the columns the plot is built from. Default columns are set for a table of statistics produced with Limma Voom, but if you use another statistical test, you can adjust the log_2 fold-change, average expression columns, and significance columns.

minExpressionThreshold

You can configure the position of the vertical dashed line by changing the value in the `minimum AverageExpression` attribute.

3.9.11 t-SNE

The t-SNE statement performs dimensionality reduction on a table and plots a projection (where each row of the table is represented as a point on a 2D page). t-SNE plots are frequently used to cluster and visualize high-dimensional data. They are a strong alternative to Principal Component Analysis (PCA). See citeVanDerMaaten2008 for details about the t-SNE approach.

In order to use the `t-SNE` statement, you first need to import the language *org.campagnelab.metar.plots*. You can do this by pressing cmd + L and typing the *metar.plots* to locate the language. After importing the language, you can locate the statement by typing `t-SNE`. Figure 3.43 presents a newly created `tSNE` statement. A t-SNE statement offers the following editable attributes:

input table

You must link this attribute to a table of data. The table must contain some columns annotated with the group "counts". This group is used to identify which columns to consider for dimensionality reduction. Any other columns of the table will be removed before running the t-SNE approach on the table.

plot

The first attribute to the right of `->` is a plot. You can use this slot to anem name the t-SNE plot produced by the statement.

result table

The second attribute to the right of `->` is a result table. The table will contain the coordinates TSNE1 and TSNE2 for each row of the table, a row identifier (in a column caller `sample`) and a cluster identifier. Using the table is optional since you may find it sufficient to use the plot, with colors determined automatically by clustering, but the table makes it possible to prepare scatterplots where the color is controlled by other data attributes. It is for instance common to use the sample id to join the tSNE result table with an annotation table that also contains the sample/row identifiers.

speed/accuracy

This attribute controls the tradeoff between speed and accuracy. Use zero for the exact method, larger values for faster convergence.

```
T-SNE plot <no table> -> <no name> , <no name> {
  speed/accuracy (0 exact, default 0.5, larger is faster)  0.5
  perplexity 50
  number of iterations 1000
  check duplicates (recommended, slower for large tables)  true
  random seed 122332
  number of clusters 1
}
```

Figure 3.43: **New t-SNE statement.**

perplexity

This attribute controls the t-SNE perplexity parameter. See `http://distill.pub/2016/misread-tsne/`.

number of iterations

How many iteration to run before stopping.

check duplicates

It is not recommended to include exact duplicates in the input table. This attribute controls checks for exact duplicates. Note that the check can be slow for large table, so you can disable it.

random seed

A random seed to make the analysis reproducible.

number of clusters

Following t-SNE reduction, a k-means clustering is done with the specified number of cluster to assign each row a cluster id. The coordinates TSNE1 and TSNE2 are used for k-means features. After looking at the plot with a number of cluster set to 1 you should be able to determine how many clusters you see in the data. Setting this number in this attribute and re-running will label each row with a cluster id in the output table.

4 — Docker Integration

The integration with Docker helps with the reprducibility of your analyses. Installing packages with R does not make it possible to specific exactly the version of the package that you need for an analysis. While it is possible to always install the latest version of a package, the behavior of some packages will change over time. For this reason, it is useful to build snapshots of the R packages used during analysis and report the version number of the snapshot.

4.1 Pre-requisites

You will need a working docker installation on your machine before you can run MetaR analyses with Docker (see `http://docs.docker.com/installation/`).

4.1.1 Mac OS

Mac users will find it convenient to download and install the docker native installation. After installation, open a Terminal and type `docker pull fac2003:rocker-metar:latest`

4.1.2 Other Platforms

Other users should refer to documented installation steps for their platform.

4.2 Configuring Docker

Docker integration can be configured using the MPS Preferences. Open Preferences (Setting on Windows) and locate the Docker configuration (use the search box with the docker keyword). You will be presented with the following dialog shown in Figure 4.1.

path to docker executable
This field must provide a path to a valid docker executable. On linux, try `which docker` in the shell. On Mac, start a Terminal and type `which docker` and copy the location to the field.

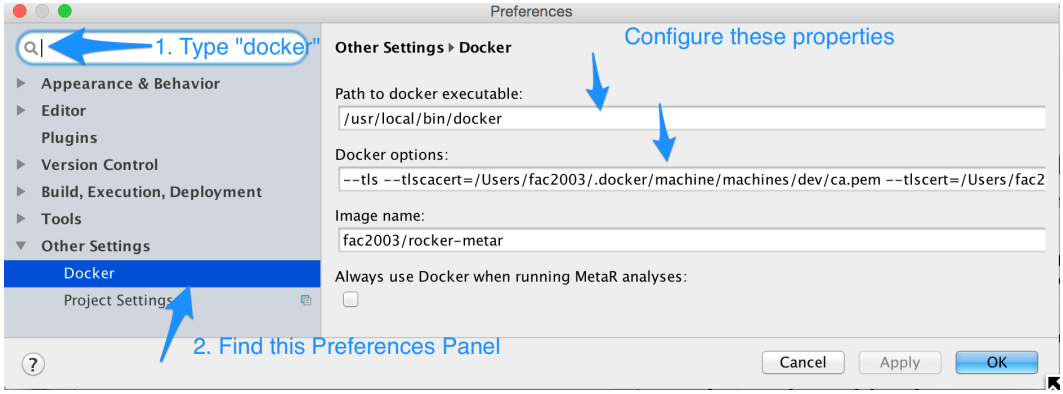


Figure 4.1: **Docker Configuration Dialog.** This dialog is available under MPS Preferences/Settings.

docker options

These are the options necessary to connect to the docker server. If you installed docker native, you don't need to enter any options. On other platforms, open the docker application and start the terminal with the File ⟩ Open Docker Command Line Terminal Window. In the window, type `echo 'docker-machine config'` and copy the line printed to the field.

image name

This is the name of the docker image that you wish to use with MetaR. Keep the default, or enter a customized image name here. Customizing the image is useful if you need to install additional packages than the ones we use during our training sessions. If you create a new image, make sure you use `fac2003/rocker-metar` in the FROM field. Images that do not build on `fac2003/rocker-metar` are not supported at this time.

Note that you can specify an image tag/version number. Append a colon (:) and the tag after the image name. For instance, use `fac2003/rocker-metar:2.1.0` to run the `rocker-metar` image packaged with MetaR 2.1.0.

Note that specifying a tag is a good idea if you need to reproduce exactly an analysis in the future. Omitting the tag will always get the latest version of the image, which may change over time.

always use docker

This checkbox can be use to force the use of Docker by default when running an Analysis. This is convenient if you know that you configuration is correct and want to run all analyses with docker.

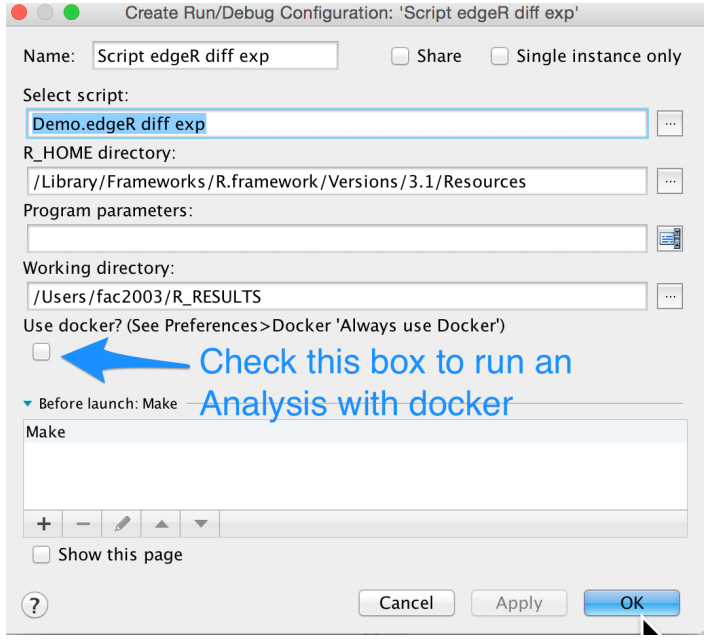


Figure 4.2: **Run With Docker.** Right-click on an Analysis and do Create <analysis name>. You will be presented with the Run Configuration customization dialog. Check the box to run with docker. Disable the check-box to run directly against the R installation on your computer (this behavior was the default prior for MetaR 1.3-).

4.3 Running with Docker

When docker is successfully configured, you can specify to use the docker image when running a MetaR analysis. See instructions in Figure 4.2 to see how to configure running insider a Docker container.

5 — SCnorm

5.1 Single Cell Normalization

SCnorm is a method developed to normalize single cell RNA-Seq data [**bacher2017scnorm**]. It is integrated into MetaR to facilitate normalization of bulk or single cell RNA-Seq data. In order to use SCnorm in MetaR, you need to import the *org.campagnelab.metar.scnorm* language. Once imported, you will be able to use two statements in a MetaR analysis:

5.1.1 check count depth

This statement (alias `check count depth`) will produce a PDF with diagnostic plots showing the density of slopes for the curves log gene expression vs. log depth of sequencing (as shown in Figure 1 of [**bacher2017scnorm**]. Figure 5.1 shows a newly created statement.

```
check count depth <no table> -> / <no outputPrefix> (result in  /Users/fac2003/R_RESULTS/_count-depth_evaluation.pdf  )
```

Figure 5.1: **New Check Count Depth Statement.** This statement produces diagnostic PDF with density of slopes for gene expression vs depth of sequencing. The statement indicates where the PDF will be written (somewhere under the directory indicated in the `R_RESULTS` path variable.

input table

A table of data, with some columns marked with the counts attribute. Only counts columns will be normalized and available in the output in normalized form. One column should have the ID attribute and will also be copied to the normalized table.

prefix

The prefix is a string used to name the PDF output. The name of the PDF is derived from the prefix, followed by "_count-depth-evaluation.pdf". You can see the filename where the PDF will be written in parenthesis following the string "result in".

Inspector

Attributes available under the inspector include:

filter cell proportion parameter. This parameter controls the threshold that determines when the normalization process has converged. A default value of 0.1 is suggested, but can be adjusted if the process fails to converge.

5.1.2 SCnorm

The `SCnorm` statement normalizes data with the R SCnorm package to bring slopes back to zero. The SCnorm statement has the following attributes:

input table

A table of data, with some columns marked with the counts attribute. Only counts columns will be normalized and available in the output in normalized form. One column should have the ID attribute and will also be copied to the normalized table.

condition

You may specify an optional column usage. This usage must annotate a group specific on a counts column of the table. If specified, normalization will be performed in the groups defined by values of the usage.

K/Scan

The attribute whose default value reads "Scan values of K" is the default and recommended option. It will scan values of K until one is found that results in slope smaller than the threshold (see inspector attributes).

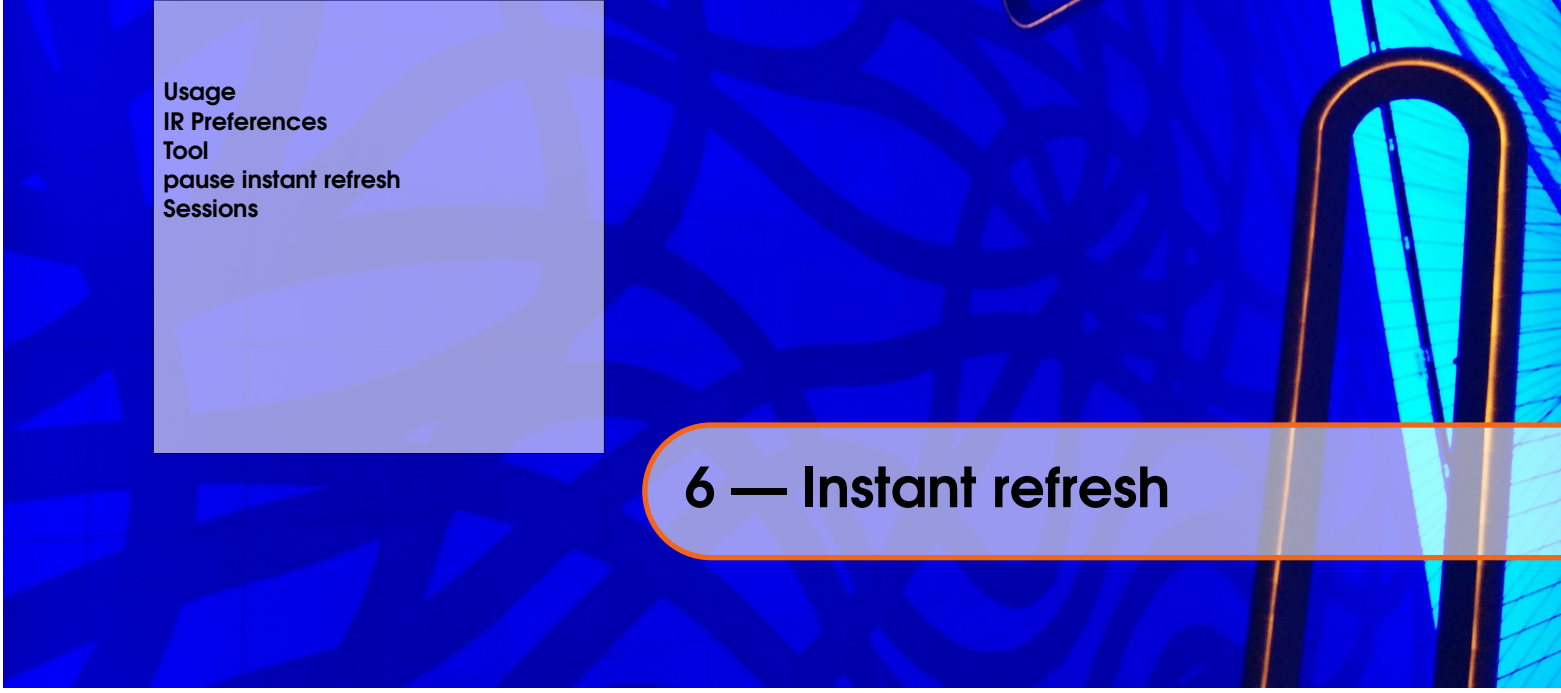
You can specify a fixed value of K by switching this attribute to the value K= and entering a value of K. This can speed up processing when you know which value should be used.

output table

The output table can be named. The output table will contain the normalized data for counts columns as well as the identifier (group ID) column.

```
SCnorm <no table> Scan values of K with condition: <no condition> -> Normalized
```

Figure 5.2: **New SCnorm statement.** This statement is used to normalize data with SCnorm.



6 — Instant refresh

Instant Refresh (IR) was introduced in MetaR 2.0. This feature allows users to make changes in their scripts and see results updated seamlessly. Instant refresh supports both MetaR Analysis and composable R scripts. Changes in Analyses or RScripts are automatically detected and copied to a new Analysis node called `Instant refresh`. Based on configurable settings, either a normal run configuration with a local R installation, or a new docker container is started to execute the instant refresh code. Executing the code regenerates tables or plots and updates the display. Users can follow progress of the refresh in the Instant Refresh tool. This Chapter describes this feature in more details. Instant Refresh was developed by Alexander Pann during a summer internship in the lab.

6.1 Usage

IR is not activated by default. In order to enable it, you must go in the preferences (see Section 6.2). Once activated, all the existing Analyses and RScripts are automatically registered for instant refreshing when a project is opened. For newly created analyses and scripts (i.e. after the activation and without restarting MPS), use the "Register for Instant Refresh" intention (💡) to enable the functionality on the new code. To allow re-executing code in RScript nodes make sure that you are using the install or load expression (type `installOrLoad` and use auto-complete) for loading R libraries in the script (see Section 15.3). This expression is needed for locating all the libraries that need to be loaded when the code is re-executed. It should be used as a replacement for the R commands `install.packages`, `library` or `require`.

Note that not all expressions are considered changes when calculating the list of changes to be re-executed. Some nodes are ignored including empty lines, comments and `Save Session` expressions.

```
Analysis Instant refresh
{
  import table GSE59364_DC_all.csv

  [ transform table GSE59364_DC_all.csv -> transformedTable {
      drop column A_DC
    } ]
  [ preview table transformedTable [ 4 cols x 5 rows ] Hide preview no style ]
```

	gene	mRNA len	genomic span	DC_normal
1	Total			34421746.00
2	10a.872B9 (locus 10a)	361.00	472.00	0.00
3	16G2	1444.00	12549.00	0.00
4	214K23.2	2522.00		

```
}
```

Figure 6.1: **Visualize Changes** Assume a change is made in the transform table statement (drop column). The figure shows the statements that need to be re-executed are surrounded by red brackets. You can visualize changes by by placing the cursor over the point of change and using the 'Show Statements That Would Be Affected by a Change" intention.

6.1.1 Instant refresh node

The Analysis node `Instant refresh` is a node that is automatically inserted in the current model and is populated with all expressions/statements that have to be re-executed.

Note that changes made manually in the Instant Refresh node are ignored and will be deleted and replaced the next time IR is triggered.

6.2 IR Preferences

The preferences can be found in the Preferences/Settings MPS menu (MPS › Preferences › Other Settings › InstantRefresh on Mac, MPS › Settings › Other Settings › InstantRefresh on PC).

There are a few options that can be changed:

1. **Type system check**. If this option is enabled, instant refresh is not executed when the type system reports errors for the current node.
2. **Enabled**. Toggle this option to activate or deactivate IR.
3. **Debug mode**. Toggles the visibility of additional messages in the IR tool. These messages are helpful for debugging purposes. They also give some additional information, for instance explaining in some cases why IR was not executed (e.g. the feature is not enabled, a pause expression is active..). Additionally it outputs the standard output of the currently executing script that would normally be found in the Run tab console.

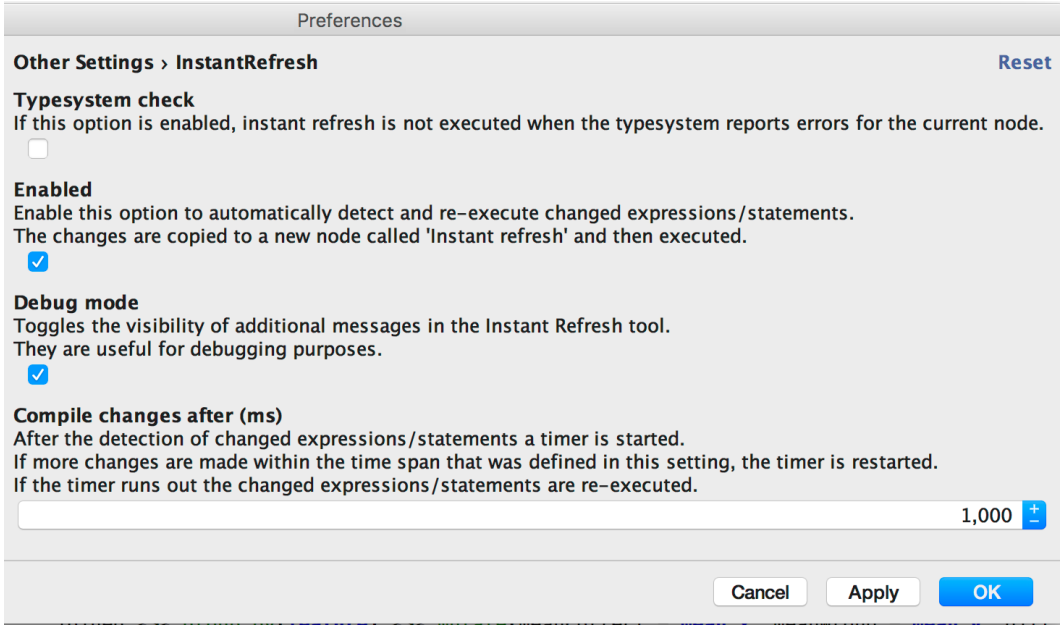


Figure 6.2: **Instant Refresh Settings** This dialog is available under MPS Preferences/Settings and controls the behavior of IR.

6.3 Tool

The IR tool is a tab that is located at the bottom left of the screen. It displays a message when R code is generated and executed as well as the execution time of both of these phases. If the debugging mode is activated, the output is way more verbose.

6.4 pause instant refresh

Although IR can be disabled in Preferences, there is sometimes a need to pause it only for the current script. For instance, this is useful if you know that you need to add several new statements to a script and do not want MetaR to attempt executing incomplete scripts. RScript nodes have a built-in expression to achieve this. It can be activated by typing `pause instant refresh` and using auto-complete anywhere in the script. When the `pause instant refresh` expression is present in a script, IR will not attempt to execute this script. You can resume instant refresh by deleting the expression.

The `pause instant refresh` expression can only be used once per script because it affects the whole script.

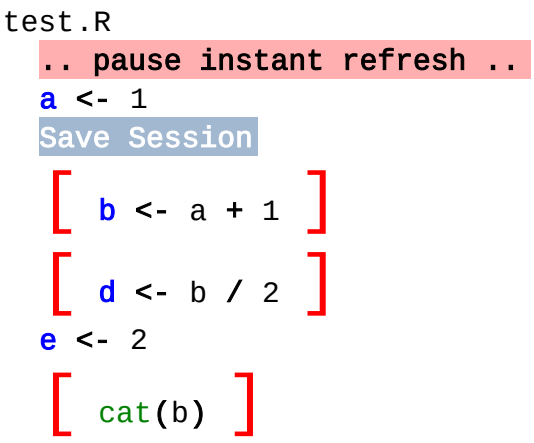


Figure 6.3: **Sessions Affect the List of Changed Nodes.** A change was made in the expression after the `Save Session` expression and therefore the statement above does not have to be re-executed.

6.5 Sessions

To accelerate the re-execution of R code, sessions are used. Sessions are files with the extension ".RDa" and are saved in a sub folder of the results directory.

They contain an external representation of R objects that can be used for restoring the R environment. Every statement in an Analysis node automatically saves a new session file after the execution of the statement. In RScript nodes save session expressions can be inserted manually. They can only be inserted at the root level of the script and not inside nested expressions (e.g. if statements, functions, blocks...). When a change is made, the nearest saved session is loaded and changed expressions are only executed if they are located after this expression. All sessions in the current model can be removed by using the use the "Invalidate Sessions" intention (💡) and new session files will be created on the next run of IR.

7 — EdgeR

7.1 Understanding Language Composition

The EdgeR language (`org.campagnelab.metaR.edgeR`) has been developed as a illustration of language composition with MetaR. When you import the `org.campagnelab.metaR` devkit into MPS, you are able to create analyses and the statements described in the previous Chapter. If you tried to enter the `edgeR` alias, the error shown in Figure 7.1 would appear. The reason is that by default, MetaR does not provide an EdgeR statement.

If you now also import the `org.campagnelab.metaR.edgeR` language (use ⌘+L and import the `edgeR` language), you will be able to use the `edgeR` statement. A new EdgeR statement is shown in Figure 7.2. Adding the EdgeR language to MetaR contributes a new kind of Analysis statement, which becomes available through auto-completion. From a user point of view, importing languages is all that is required to extend MetaR with new language constructs. This new statement can be configured by the user and will generate R code together with the other statements. This happens seamlessly and requires no other configuration than declaring that the model uses another language.

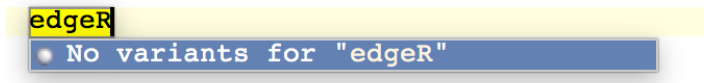


Figure 7.1: **Error When Typing the EdgeR Alias.** The `EdgeR` statement is not yet defined.

```
edgeR counts= <no table> model: ~ 0
  comparing       -> <no name> (normalize with  common dispersion estimations  )
```

Figure 7.2: **New EdgeR Statement.** A new EdgeR statement created after you have added the `org.campagnelab.metaR.edgeR` language to the model's MPS Used Languages.

7.2 The edgeR Statement

The edgeR statement performs tests of differential expression using read counts contained in a table of data. The statement has the following attributes.

counts table

The table must contain columns annotated with the "counts" group. Bind this table reference to a table that contains non-normalized read counts.

model

You can use the model attribute to enter a linear model. EdgeR will use this model to model the mean and variance of the data. You can enter a linear model by typing + followed by the name of a group usage attached to the counts table. Repeat to add multiple factors to the model.

EdgeR will use an exact test when the model has one factor. but will use a Generalized Linear Model (GLM) when the model has more than one factor. This is handled transparently.

comparing

The comparing attribute makes it possible to define the statistics that should be tested for difference with zero. After you have defined a model with several factors (corresponding to group usage), the factor levels (corresponding to group names) will be offered for auto-completion. The factor level name stands for the average of the columns annotated with the group. See Figure 7.3 for an example.

normalize with

EdgeR supports three types of normalization methods, which estimate variance/dispersion in different ways:

- common dispersion
- trended dispersion
- tagwise dispersion

Place the cursor on the keyword following normalize with and use auto-completion to switch from one type of normalization method to another. The the EdgeR Bioconductor documentation `http://www.bioconductor.org/packages/release/bioc/html/edgeR.html` for details about these approaches.

7.3 Example

Figure 7.3 presents an example where the edgeR statement is configured with a model (one factor: `LPS`) to call differences between columns labeled with the groups `LPS=YES` and `LPS=NO`.

```
edgeR counts= filtered model: ~ 0 + LPS
  comparing LPS=YES  - LPS=NO  -> Results (normalize with  tagwise dispersion  )
```

Figure 7.3: **EdgeR Example.**

8 — Limma Voom

Limma Voom is a statement introduced in MetaR 1.3.

8.1 Overview

The Limma Voom statement makes it possible to use the Limma R package and the Limma Voom adjustment to analyze RNA-Seq data with Limma. Similarly to EdgeR, the language that provides the `Limma Voom` statement must be added to the model where you plan to use the statement in analysis. The name of the language is *org.campagnelab.metar.limma*. Note that this language depends on *org.campagnelab.metar.models*, which should also be imported.

8.2 The Limma Voom Statement

The `limma voom` statement performs tests of differential expression using read counts contained in a table of data. The statement has the following attributes. Figure 8.1 presents a newly created Limma Voom statement.

```
limma voom counts= <no table> model: ~ 0
  comparing      -> stats: <no name> normalized: <no name>
```

Figure 8.1: **New Limma Voom Statement.**

counts table

The table must contain columns annotated with the “counts” group. Bind this table reference to a table that contains non-normalized read counts.

model

You can use the model attribute to enter a linear model. Limma Voom will use this model to model the mean and variance of the data. You can enter a linear model by typing + followed by the name of a group usage attached to the counts table. Repeat to add multiple factors to the model.

comparing

The comparing attribute makes it possible to define the statistics that should be tested for difference with zero. After you have defined a model with several factors (corresponding to group usage), the factor levels (corresponding to group names) will be offered for auto-completion. The factor level name stands for the average of the columns annotated with the group. See Figure 8.2 for an example.

normalized

This attribute holds a table of normalized counts that will be produced when the `limma voom` statement is executed. You should name the table of normalized counts to make it possible to reference it later. Normalized counts are available even when the model has a single factor (in which case adjust counts would not work because there is no batch to remove).

adjusted counts

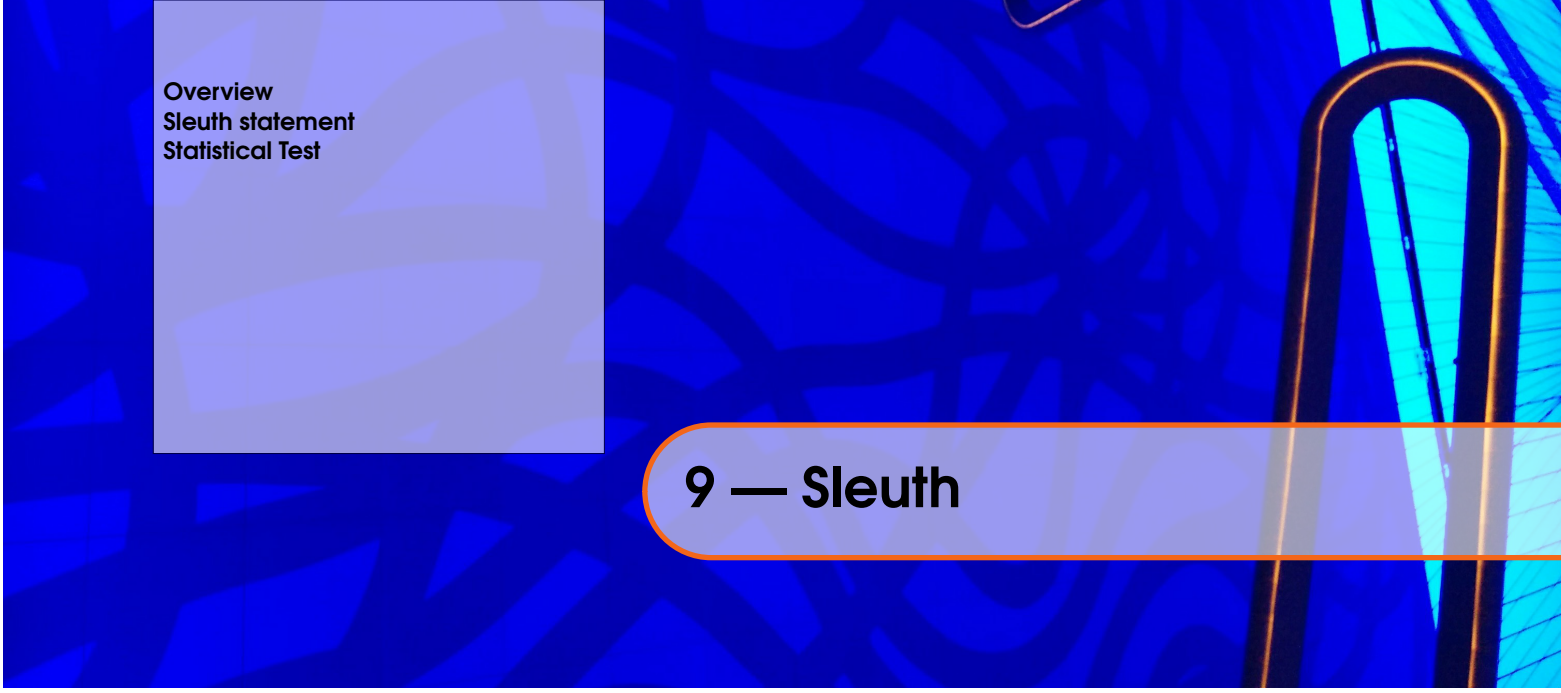
This attribute is exposed under the Inspector Tab(2: Inspector). It takes a boolean value: either true or false. When true, the limma voom statement will produce a table of adjusted counts. Data in this table is adjusted to remove the effect of covariates described in the model, but not used in the comparing attribute. This is useful to remove the effect of batches, or other cofactors expected to affect expression. Adjusted counts are implemented with the Limma removeBatches function.

8.3 Example

Figure 8.2 presents an example where the `Limma Voom` statement configured with a model (one factor: `LPS`) to call differences between columns labeled with the groups `LPS=YES` and `LPS=NO`.

```
limma voom counts= filtered model: ~ 0 + LPS + covariate
  comparing LPS=NO - LPS=YES -> stats: results normalized: normalized
```

Figure 8.2: **Limma Voom Example.** Since version 1.8, limma voom outputs normalized counts.



9 — Sleuth

9.1 Overview

Sleuth is a package for differential expression testing designed to be used with results obtained with Kallisto [**bray2016near**]. In order to use Sleuth, you first need to map RNA-Seq reads with Kallisto against a reference transcriptome. This task can be accomplished with NextflowWorkbench [**kurs2016nextflowworkbench**] (we distribute a workflow to perform pseudo-alignments with Kallisto and use this workflow in the NW training sessions).

When the workflow completes, you will be able to download Kallisto result directories. In this Chapter, we assume that you download and organize Kallisto results under a single directory we will refer to as `KALLISTO_RESULTS`.

9.2 Sleuth statement

The `Sleuth` statement, alias `sleuth` is provided in the *org.campagnelab.metar.sleuth* language. Import this language in a model where you want to Sleuth for data analysis.

After downloading Kallisto results, create or open a MetaR analysis and type sleuth. A new statement such as shown in Figure 9.1 will be created.

```
sleuth using enter a Kallisto result base directory ... model: ~ 0 -> Results
```

Figure 9.1: **New Sleuth Statement.**

The statement has three attributes:

Kallisto Result Base Directory

This is a string attribute where you can paste the path to `KALLISTO_RESULTS`. Once pasted, if the directory contains Kallisto result sub-directories, the information is organized into a MetaR Table, that will appear in the model. An import statement is added to the analysis and the table becomes referenced by the sleuth statement. A sleuth statement bound to a table is

shown in Figure 9.2. At this point, you can annotate the table as you would for a Limma voom or EdgeR analysis: add column groups to columns corresponding to samples and annotate the groups with usages to define factors that you would like to include the analysis (see Chapter 2 to learn how to do this). For instance, using the Kallisto results from the Sleuth tutorial, the ColumnGroupContainer and Table would look as shown on Figure 9.3.

```
{
  import table cuffdiff2_data_kallisto_results
  sleuth using cuffdiff2_data_kallisto_results  model: ~ 0 -> Results

}
```

Figure 9.2: **Sleuth Statement Bound to a Table.** The statement shown has been bound to a table by entering a valid Kallisto results directory.

```
Column Groups and Usages

Define Usages:
  Treatment

Define Groups:
  HOXA1KD used for Treatment
  Scramble used for Treatment
  ID used for << ... >>
```

```
Table cuffdiff2_data_kallisto_results ...
File Path
  /Users/fac2003/Downloads/cuffdiff2_data_kallisto_results/fake-tab
Columns
  transcriptId: string [ ID ]
  SRR493366: numeric [ Scramble ]
  SRR493367: numeric [ Scramble ]
  SRR493368: numeric [ Scramble ]
  SRR493369: numeric [ HOXA1KD ]
  SRR493370: numeric [ HOXA1KD ]
  SRR493371: numeric [ HOXA1KD ]
```

Figure 9.3: **ColumnGroup and Table for Sleuth Tutorial** Left: ColumnGroupContainer configured for the Sleuth tutorial. Right: Table imported from Kallisto base directory, and annotated with the Scramble and HOXA1KD groups.

Full Model

A model attribute can be specified after `model:`. This model should represent the full set of factor/group usages you plan to use to model expression data. You need to annotate columns of the table with groups and group usage before you can use auto-completion to define the model.

9.3 Statistical Test

The attribute shown on the second line determines the type of statistical test to perform. Sleuth supports two options at this time:

- Wald test
- Likelihood Ratio Test (LRT)

Place the cursor over the pink cell and use auto-completion to choose one of these options.

Likelihood Ratio Test

If you select the LRT, you get to enter a label and a second model. The label is a text string that helps you remember what the alternate model represents. Note that Sleuth only supports an alternate model that is fully contained in the full model. This means that you can remove covariates from the alternate model, but not add some that are not present in the full model (at this time, MetaR does not check for this condition, so watch out).

```
LRT, compare with: <no compareLabel> : ~
```

Wald Test

If you select Wald Test, you get the option of entering one group and one usage. The combination should identify a condition for which you are seeking to find differentially expressed transcripts. It is unclear what the test reports when there are several levels to a factor (i.e., a group usage associated with three or more groups).

```
Wald Test for <no groupUsage> : <no columnGroupFactor>
```

10 — Biomart

10.1 Overview

The BioMart project develops software and data services that are made available to the international scientific community. Users of MetaR can access data marts provided by BioMart, providing access to a wide range of research data . Similarly to EdgeR and Limma, BioMart support is provided in a MetaR language extension. This means that in order to access BioMart with MetaR, you first need to add the *Biomart* language to the model where you create the analysis. To do so, you can press ctrl + L and add language *org .campagnelab.metar.biomart*. After adding the *biomart* language to the model, you can create `query biomart` statements, described in the following sections.

R The Biomart language in MetaR was developed by William ER Digan and was introduced in MetaR 1.4.

10.2 The Biomart Statement

, The `query biomart` statement makes it possible to interactively specify which data should be retrieved from a mart (using attributes and filters). Data downloaded from Biomart will be stored in a new table. Figure 10.1 presents a newly created `query biomart` statement. The `query biomart` statement has the following parameters.

- `database` This is the source database that will be queried to retrieve data.
- `dataset` This is the dataset, inside the database that will provide data.
- `attributes` These attributes describe which columns of data will be downloaded and stored in the result table.
- `filters` These filters control which rows of data will be retrieved from the dataset.

database

The first time you create a `query biomart` statement in an Analysis, the statement will retrieve the list of available Biomart databases. To use one of these databases, use auto-completion over the `database` attribute (see text *select a database* in Figure 10.1). Then,

```
query biomart database select a database and dataset select a dataset
  get attributes << ... >>
  filters << ... >>
  -> resultFromBioMart
```

Figure 10.1: **New Query Biomart Statement.**

press ctrl + [ ] and select one database. Selecting the database will retrieve the set of datasets available in this database, which will become available with auto-completion.

dataset

Once a database is selected, you need to choose a dataset inside this database. To choose one, press ctrl + [ ] to display all available datasets, on the text *select a dataset*. You may use type keywords to identify the dataset of interest, similarly to other uses of auto-completion in MPS and MetaR. When a dataset is selected, you will need to configure its children:

- `attributes`, which will be the columns retrieved from the dataset that will populate the result table.
- `filters`, which make it possible to restrict the rows of data to retrieve.

In a few instances, you may find that a dataset cannot be associated with attributes or filters. In this case, in the auto completion menu for both attributes and filters will display the message *"No available filter or attributes in this dataset"*. The selected dataset is no more available in Biomart. This means that this dataset, although available via the Biomart web service, cannot be used to retrieve data from the web service. You will need to select another database/dataset.

attributes

Attributes are columns of the source dataset that will be written to the result table. Figure 10.2 presents a new biomart attribute. An attribute has three parameters:

- `attribute`, a column you want to retrieve from the dataset and write to the result table. Press ctrl + [ ] to display the available column names.
- `type`, the value type of the attribute. Choose a type for the column that will be created from the set: `boolean`, `numeric` or `string` (note that `string` is the default). To change the type, press ctrl + [ ].
- `column group`, an attribute can have a group such as "ID". The user can display the autocompletion menu by pressing ctrl + [ ]. Group must be defined in the Column Group Container to be added to columns created for the result table.

You must select at least one attribute before you can execute the `query biomart` Statement.

filters

Filters make it possible to restrict the result with some criteria. The types of filter available depend on the dataset selected in the statement. Figure 10.3 presents a new biomart filter. There are four kinds of filters:

```
query biomart database ENSEMBL GENES 79 (SANGER UK)and dataset Mus musculus genes (GRCm38.p3
  get attributes<no attribute> of types string with column group annotation select a group
  filters << ... >>
  -> resultFromBioMart
```

Figure 10.2: **Select an attribute in a biomart dataset.** An attribute contains any information you want to retrieve to populate the result table. Attributes have a name, a type and a column group annotation. The set of available attributes depends on the specific dataset selected.

- `boolean filters` select rows for which a value is either true or false. For example, if a gene has or does not have a miRBase identifier.
- `text filters` select rows which match some text in some attribute. The query will return only rows of the dataset that contain which match the text. For example, you can use a text filter to query rows that include a specific GO term.
- `list filters` select rows that include an element among those of a finite list. Available list elements are determined by the mart dataset. For example, a specific chromosome in a specie can be selected with a list filter.
- `id list filters` select rows that contain a specific set of identifiers. These ids can be obtained either in a MetaR `SetOfIds` node, defined before the `query biomart` statement, or directly from an annotated table, where one column has an ID group.

```
query biomart database ENSEMBL GENES 79 (SANGER UK)and dataset Mus musculus genes (GRCm38.p3
  get attributesEnsembl Gene ID from featureof types string with column group annotationID
  filters no filter <no filterWith>
  -> resultFromBioMart
```

Figure 10.3: **New Biomart Filter.** A filter allow you to filter rows of the dataset according to some criterion. It exist four filters categories: boolean, text, list and id list. Filters are related to a specific dataset.

table

The future table where your result will be stored. Column annotations derived from the attribute type and column groups are displayed under the Inspector Tab (2: Inspector).

10.3 Examples

10.3.1 Example 1

Figure 10.4 shows how to obtain a table from the Ensembl database and Human dataset. The result table "resultFromBiomart" contains two columns, the Ensembl Gene and Exon ID, where the first column is annotated as a group "ID". These results are filtered to exclude any gene that does not have a miRBase identifier.

10.3.2 Example 2

Figure 10.5 shows how to obtain a table from the *Paramecium bibliography* database. The result table "resultFromBiomart" contains two columns, the PubMed ID and the abstract,

```
query biomart database ENSEMBL GENES 79 (SANGER UK) and dataset Homo sapiens genes (GRCh38.p2)
  get attributes Ensembl Gene ID from feature of types string with column group annotation ID
                 Ensembl Exon ID from feature of types string with column group annotation select a group
  filters Ensembl Gene ID(s) [e.g. ENSG00000139618] from a set of ids idsset
          with miRBase ID(s) where values are false
  -> resultFromBioMart
```

Figure 10.4: **Biomart Example 1.**

where the publication year is equal or larger than 2000.

```
query biomart database PARAMECIUM BIBLIOGRAPHY (CNRS FRANCE) and dataset Paramecium bibliography
  get attributes PubMed ID from my_attributes of types string with column group annotation ID
                 Abstract from my_attributes of types string with column group annotation select a group
  filters Year >= match 2000
  -> resultFromBioMart
```

Figure 10.5: **Biomart Example 2.**

11 — R Functions

11.1 Overview

Since version 1.4, MetaR supports calling R functions directly. This Chapter describes how you can use this feature to take advantage of the many functions available in R packages to transform data in your MetaR Analyses.

11.1.1 Function w

Function stubs are provided that represent functions offered by different packages. Stubs do not provide the code associated with the function, but describe the function name and the arguments of the function and its default values. This information is used to support auto-completion for R functions.

We provide pre-imported stubs for the packages used during the MetaR training sessions, namely: *base*, *graphics*, *data.table*, *pheatmap*, *biomaRt*, *edgeR* and *limma*. While stubs are provided, they are not immediately available in a MetaR analysis. In order to use R function stubs, you need to

- Add *org.campagnelab.R* to the list of Used languages in the model where you need the stubs.
- Use the `import stubs` statement followed with the name of the package that provides the functions that you wish to use.

For instance, if you enter the following statement:

```
import stubs base
```

After typing this statement, the *base* package R functions will become available inside the Analysis where you imported these stubs. You can use R function using the `eval` statement or expression (see Sections 11.6and 11.7).

If you need a package that is not yet provided with MetaR, you should use the `import package` statement (see Section 11.3).

11.2 Import Stubs Statement

The `import stubs` statement (alias import stubs) makes it possible to import functions in packaged already packaged with MetaR. Simply type import stubs, and use auto-completion to locate the package for which you need to import function stubs.

11.3 Import Package Statement

The `import package` statement (alias import package) makes it possible to import functions in any R package. If the package is already provided in MetaR, the `import package` statement will be automatically replaced with the equivalent `import stubs` statement (see Section 11.2). Figure 11.1 presents an import package statement.

name

The name attribute is a string and must be the name of an R package, suitable to install the package in R with `install.packages("name")`.

Figure 11.1: **New Import Package Statement.** Enter the name of an R package to import this package into the Analysis. Note that the package will be visible only after you run the Analysis at least once.

If you need to import a Bioconductor package, use the `import bioconductor package` statement instead.

After you execute the Analysis that contains the import package statement, the package will be installed in the version of R that you are using, if needed and the package loaded. Use the "Reload Functions and Create Stubs" intention after you have executed the statement to create the Stubs root node for the functions in the package. When you call this intention, the package will be inspected for function declarations, and these declarations will be written to a `Stubs` root node in the model where the `Analysis` is located. Following this process, the `import package` statement is replaced with the import stubs statement, loading the stubs directly from the model. Note that you can inspect the stubs object to learn about the functions available in the package represented by the stubs.

11.4 Import Bioconductor Package Statement

Importing a bioconductor package is very similar to importing a regular R package, but you need to use the `import bioconductor package` statement. This statement ensure appropriate installation and loading of bioconductor packages in R. Figure 11.2 presents a new `import bioconductor package` statement.

```
import bioconductor package <no name>
```

Figure 11.2: **New Bioconductor Import Package Statement.** Enter the name of an Bioconductor package to import this package into the Analysis. Note that the package will be visible only after you run the Analysis at least once.

11.5 Stubs

Stubs represent functions from R regular or bioconductor packages. We do not recommend creating Stubs manually. The easiest way to create Stub root nodes is by using the `import package` and `import bioconductor package` statements. Figure 11.3 presents a snapshot showing a few functions located in the base Stubs root node (R version 3.1.3).

Stubs distributed with MetaR are packaged in models named after the version of R which provided the package, in an effort to help track differences between major R versions.

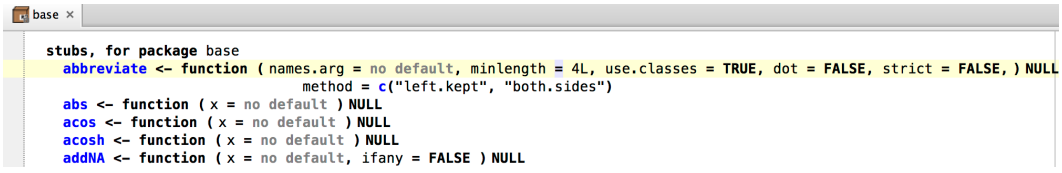


Figure 11.3: **Base Package Stubs Illustration.** This snapshot presents the beginning of the stubs base root node, located in model `R3_1_3` (language *org.campagnelab.metar.r.stubs*).

11.6 Eval Statement

The `eval` statement can be used to call R functions that produce some side-effect. When you import the devkit *org.campagnelab.R* into a model, you can type `eval` in an Analysis node as a statement. The statement will offer auto-completion for function names imported into the analysis. If you do not see the function that you would like to use, make sure you have imported the stubs for the package that contains this function. When you run the analysis, the function named after `eval` will be executed. Note that you cannot retrieve a return value with the `eval` statement. You must use the `eval` expression to obtain a value. The eval statement is useful when you need to call a function that has a side-effect, for instance the setkey function of data.table.

11.7 Eval Expression

The `eval` expression can be used to call R functions inside a MetaR expression. MetaR expressions are used in the `subset` statement, and in some plotting statements (e.g., `boxplot` or `histogram`). When you import the devkit *org.campagnelab.R* into a model, you can type

`eval` inside an expression. The `eval` expression will offer auto-completion for function names imported into the analysis. If you do not see the function that you would like to use, make sure you have imported the stubs for the package that contains this function.

11.8 Accessing MetaR Columns within R Expressions

When you use either the eval statement or expression, you will often need to access columns from a MetaR table to pass as arguments to the function. You can do this with the `$` node, which bridges between R expressions (used inside R functions) and MetaR columns. After typing `$`, the node will auto-complete to the set of columns visible at this point of the MetaR `Analysis`.

11.9 Example

Figure 11.4 presents an example where stubs are imported for packages pre-packaged with MetaR and the import package statement is used to import grDevices.

```
Analysis Testing functions
{

  import stubs base
  import stubs data.table
  import stubs graphics
  import package grDevices
  simulate dataset  with  [
    num of samples:  3
    num of genes:  500
    mean when all factors are false:  1
    discrete factors:  treatment
    effect size: 1
    continuous covariate:  temperature , range: [ 0 - 100 ] , slope: 1
  ] -> simulate
  eval boxplot(x = ${sample_1_treatmen t})
  eval boxplot(x = c(1, 2, 3, 4))
}
```

Figure 11.4: **Functions Example.** This example imports stubs and one package, creates a table with three columns (see Simulate statement in Chapter 13) and evaluates two R functions. The first use of the `boxplot` function plots the values of the column `sample_1_treatment` that was generated with the simulate statement. The second call to `boxplot` is given a list of four integers.

Narrow

12 — Seurat

12.1 The Seurat language

Since release 2.3.0, MetaR offers support for Single Cell RNA-Seq analysis with the Seurat 2.0 R package [**butler2017integrated**]. This functionality is implemented in the *org.campagnelab.metar.seurat* language .

The *org.campagnelab.metar.seurat* (Seurat) language consists of statements that help with the analysis of single cell RNA sequencing data (scRNA-seq data). These statements cover a wide range of functionality: loading scRNA-seq data, quality control/cleanup steps, adjusting data (by normalization or scaling), plotting it, computing extra information based on the data (principal components, markers, etc.), aligning data from multiple samples, performing limma analysis on it and other functionality.

Seurat statements can be typed in an Analysis script (see Chapter 3). The simplest way to see what Seurat statements are valid at a given context in an analysis, is to look at the suggestions offered by the context assistant. Figure 12.1 and Figure 12.2 show two examples of context assistants. To activate the context assistant, just press ⏎ in the script to generate an empty line. Placing the cursor on the empty line should bring up the context assistant.

If you do not see the context assistant, try pressing the space bar with the cursor positioned on an empty line. Moreover, to see all possible statements, you can use auto-completion like for all other statements in an analysis (see Chapter 3).

To start writing analyses using Seurat, you need to import devkit `org.campagnelab.metaR` and language `org.campagnelab.metar.seurat`. The following sections

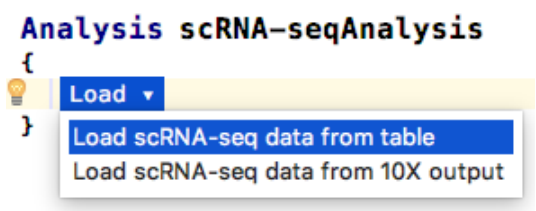


Figure 12.1: **Context assistant at beginning of script.** At the beginning of the script, the context assistant suggests loading a Seurat object either directly from the output of the 10X, or from an expression table.

```
Analysis scRNA-seqAnalysis
{
  load 10X dataset 10X directory with barcodes.tsv, genes.tsv, and matrix.mtx ...  -> seurat: introduce name
    annotations: << ... >>
    initial cleanup:
      by rejecting cell when [number of genes in cell < 1000 ]
      by rejecting gene when [number of cells where gene is expressed in < 3 ]
      <no normalization>
  | Delete | Limma ▾ | Align ▾ | Add ▾ | Adjust ▾ | Cleanup ▾ | Plots ▾ | Load ▾ |
}
```

Figure 12.2: **Context assistant after loading one Seurat object.** Once we have Seurat objects available in the script, more Seurat statements become valid.

```
seurat.name= P1Cfiltered

Seurat data:
  principal components computed: false
  clusters computed: false
  markers computed: false
  tsne computed: false
  normalized: false
  scaled: false
  prealigned: false
  highly variable genes computed: false
  conditions: P1Collapsed
```

Figure 12.3: **Properties view of a newly created Seurat object.** Many of the properties of this Seurat object are not yet computed. These properties are created automatically at the creation of the object.

describe the kinds of statements offered by the MetaR `org.campagnelab.metar.seurat` language.

12.2 The Seurat object

A Seurat object is a structure that stores scRNA-seq data and associated information. Many of the Seurat statements result in the creation of a new Seurat object. Seurat objects and references to these objects are represented with a purple foreground. Moreover, you can study the associated information of a Seurat object in the inspector window, when clicking on any such object or a reference to it. Figure 12.3 shows the information (properties) associated with a newly created Seurat object. These properties are used by some statements to assess whether the input Seurat object is ready for the computations required by the statement.

12.3 Loading Seurat objects

There are two statements in the Seurat language for loading a Seurat object, one directly from the files produced by 10X, and one from an expression matrix.

12.3.1 Load 10X dataset

The `load 10X dataset` statement makes it possible to load a Seurat object directly from the output files of 10X Genomics. The Seurat object and its properties become available to the statements that follow the loading (until the line where it is deleted; see Section 12.10).

You can create this statement by clicking Load > Load scRNA-seq data from 10X output in the context assistant, or by typing the alias `load 10X dataset` on an empty line of Analysis (see Figure 12.4) for a new load 10X dataset statement).

```
load 10X dataset 10X directory with barcodes.tsv, genes.tsv, and matrix.mtx ...  -> seurat: introduce name
  annotations: << ... >>
  initial cleanup:
    by rejecting cell when [number of genes in cell < 1000 ]
    by rejecting gene when [number of cells where gene is expressed in < 3 ]
    <no normalization>
```

Figure 12.4: **New load 10X dataset statement.** Use this statement to load a Seurat object from the output of 10X Genomics.

When creating a new `load 10X dataset` statement, the `by rejecting cell when` strategy and `by rejecting gene when` come prefilled by default with certain upper thresholds, but these strategies can be changed.

input directory

This field of `load 10X dataset` should point to the directory produced by 10X Genomics cell ranger count output. Cellranger produces a directory structure which contains the following three files: "barcodes.tsv", "genes.tsv" and "matrix.mtx". Unless the directory you specify exists on the disk and it contains these three files, the input directory field will be shown on a red background. Notice that this field has a button to let you select the directory.

annotations

The annotations are references to group usages (see Section 2.5). They allow you to attach more information to the data loaded from 10X Genomics. For instance, if you load data that corresponds to a certain patient and a certain state of the sample (condition), you would annotate the Seurat object with a reference to a group usage that represents the patient and with a reference to a group usage that represents the state (see Figure 12.5). These annotations are used by the expression tables generated from the Seurat object in the limma statements (see Section 12.9). Note that you need to create groups and group usages in a Column Group Container root node in the model of your solution. If a Column Group Container does not exist in the model, see Section 2.3 to learn how to create one.

cleanup strategies

There are three possible strategies available when loading data from 10X Genomics: rejecting a cell whose number of genes is lower or higher than a threshold, rejecting a gene when the number of cells it appears in is greater or smaller than a threshold, and normalizing the data at loading time. The cleanup strategies are explained in Section 12.4.

output

The output of the `load 10X dataset` statement is a Seurat object. You need to set the name of this object.

Example

Figure 12.5 presents an example of a `load 10X dataset` statement.

```
load 10X dataset /Users/farcasia/Documents/P1/P1_collapsed/outs/filtered_gene_bc_matrices/GRCh38 ... -> seurat: P1Collapsed
  annotations: Patient1 Collapsed
  initial cleanup:
    by rejecting cell when [number of genes in cell < 1000]
    by rejecting gene when [number of cells where gene is expressed in < 3]
    by log-normalization with a scale factor of 10000
```

Figure 12.5: **Example of load 10X dataset statement.** In this example, we load data for a patient denoted “Patient1", from a sample of collapsed tubules tissue. As a result, we annotate this Seurat object with “Patient1" and “Collapsed". Furthermore, we reject the cells with less than 1000 genes expressed in them, and the genes that can only be found in less than 3 cells. Finally, we normalize the data using a scale factor of 10,000.

12.3.2 Load dataset from table

The `load dataset from` statement makes it possible to load a Seurat object from a table that contains expression data. You can create this statement by clicking Load > Load scRNA-seq data from table in the context assistant, or by typing the alias `load dataset from table` on an empty line of Analysis. This statement is identical in fields and behavior to the `load 10X dataset` statement, except for the place of input (one is a directory generated by 10X Genomics, and one is a table). Thus, we refer you to Subsection 12.3.1 for further details on the common fields.

input table

The input table should have genes on the rows and cells on the columns. Moreover, the table should have the row names. To reference a table from this statement, you need to have a table available in the model (either by importing it, or by obtaining it from other statements; see Section 2).

Example

Figure 12.6 presents an example of a `load data from table` statement.

12.4 QC and Clean Up

One important step in the analysis of scRNA-seq data is quality control on the data. In Seurat, you can do that with the `cleanup seurat` statement. Usually, this quality control step is done after observing some diagnostic plots on the data (see Section 12.6.1 for diagnostic plots).

You can create this statement by clicking Cleanup in the context assistant, or by typing the alias `cleanup seurat` on an empty line of Analysis (see Figure 12.7 for a new `cleanup seurat` statement). Note that in the context assistant, you always need to specify a strategy that is going to be instantiated in the `cleanup seurat` statement.

```
load dataset from table simP1C.txt -> seurat: P1Collapsed
  annotations: Patient1 Collapsed
  initial cleanup:
    by rejecting cell when [number of genes in cell < 1500 ]
    by rejecting gene when [number of cells where gene is expressed in < 4 ]
    <no normalization>
```

Figure 12.6: **Example of load data from table statement.** In this example, we load data from a table with simulated data for a patient denoted “Patient1", and from a sample of collapsed tubules tissue. As a result, we annotate the Seurat object with “Patient1" and “Collapsed". Furthermore, we reject the cells with less than 1500 genes expressed in them, and the genes that can only be found in less than 4 cells. We do not normalize the data at loading time.

```
cleanup seurat introduce referenced seurat    -> seurat: filtered
```

Figure 12.7: **New cleanup seurat statement.** Use this statement to perform quality control on a Seurat object.

input Seurat

You need to specify the Seurat object on which quality control is done. Only Seurat objects that are defined in previous statements are visible in the current cleanup statement, due to scoping rules.

output Seurat

The statement creates a new Seurat object that represents the input Seurat object with the modifications prescribed in the cleanup statement. The `cleanup seurat` statement creates a default name for the output Seurat object, but this name can be modified.

strategies

To introduce a strategy, you have to press ctrl + [ ] in the red space on the second line of the statement. All the available strategies will subsequently appear in the menu. In the next sections, we explain these strategies. Two of these strategies are only available in the initial cleanup section of the loading statements: the reject gene strategy and the normalization strategy.

12.4.1 Reject gene strategy

The `by rejecting gene when` strategy specifies a comparison operation where the `number of cells where gene is expressed in` is compared to a threshold. This strategy filters the input Seurat object by dropping the genes for which the comparison is true. For this strategy, the left-hand side of the comparison can be only `number of cells where gene is expressed in`.

12.4.2 Reject cell strategy

Another strategy is when you reject cells based on whether certain conditions are fulfilled. Pressing ctrl + [ ] on the left-hand side of a condition in this strategy gives you three possibilities: number of UMIs (unique molecular identifiers), number of genes and percentage of mitochondrial genes in cell. This strategy filters the input Seurat object by dropping the cells for which any of the comparisons is true. You can specify multiple conditions in this strategy (see Figure 12.8 for an example). Note that the threshold for mitochondrial genes is an integer number representing the percentage.

12.4.3 Regress out strategy

Because single cell datasets often contain technical noise, batch effects, or even undesired biological sources of variation, the cleanup statement offers a strategy to regress out these sources of variation. You can write a list of such sources to be regressed out; after introducing one element inside the square brackets of the regress out strategy, just press ↵ to introduce a new element (see Figure 12.8 for an example). Note that this strategy also scales the data after regressing out the variation, so it is adviced that it is the last strategy in the cleanup statement. As a result, we obtain scaled data after the cleanup.

12.4.4 Accept highly variable genes strategy

It is common that you focus on the highly variable genes for downstream analysis; for this end, you can use the `by accepting highly variable genes when` strategy. This will mark the highly variable genes in the Seurat object. The highly variable genes are used by further Seurat statements. To compute the highly variable genes, Seurat makes use of the average expression and dispersion of each gene. The strategy allows you to set the cutoff for the average expression and dispersion axis. The cutoff can be typed inside the square brackets of the strategy. To introduce a new cutoff, just type ↵ after the last one introduced, and press ctrl + [ ] to see the available choices (see Figure 12.8 for an example). Note that once you specify this strategy in the cleanup statement, an output plot is pops up, that represents the dispersion versus average expression plot. You need to provide a name for this plot.

12.4.5 Normalization strategy

Another strategy that can be used only in the cleanup section of the loading statements is the normalization strategy. This strategy normalizes the data using the scale factor provided in the strategy and log-transforms the result. There is also a separate statement for normalization (see Subsection 12.5.1).

Example

Figure 12.8 presents an example of a `cleanup seurat` statement.

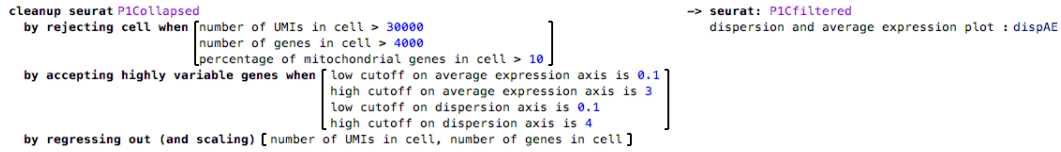


Figure 12.8: **Example of cleanup seurat statement.** This illustrates almost all the features of the cleanup statement. We set a cutoff for all the axes, we reject cells based on three different conditions and we regress out unwanted variations for two different variables.

12.5 Adjusting Seurat objects

There are two statements in Seurat that are meant for standard adjusting (pre-processing) the data: normalization and scaling. Some statements require that the Seurat object is already normalized or scaled. For instance, the calculation of highly variable genes needs a normalized Seurat object as input.

12.5.1 Normalize Seurat object

The normalization statement normalizes the expression data stored in the Seurat object and log-transforms the result. You can create this statement by clicking Adjust 〉 Normalize in the context assistant, or by typing the alias `normalize seurat` on an empty line of Analysis (see Figure 12.9 for an example of a new `normalize seurat` statement).

```
normalize seurat introduce referenced seurat with scale factor <no scaleFactor> -> seurat: introduce name
```

Figure 12.9: **New normalize seurat statement.** A new `normalize seurat` statement with three fields to be completed.

input Seurat

The input Seurat object can be any Seurat object created in a previous statement, but it should not be an already normalized Seurat object. You will receive an error if that is the case.

scale factor

You also need to specify the scaling factor for the normalization.

output Seurat

The output is a normalized Seurat object. You need to introduce a name for it.

12.5.2 Scale Seurat object

The scaling statement scales the data stored in the Seurat object. You can create this statement by clicking Adjust 〉 Scale in the context assistant, or by typing the alias `scale seurat` on an empty line of Analysis (see Figure 12.10 for an example of a new `scale seurat` statement).

```
scale seurat introduce referenced seurat -> seurat: introduce name
```

Figure 12.10: **New scale seurat statement.** A new `scale seurat` statement with two fields to be completed.

input Seurat

The input Seurat object can be any Seurat object created in a previous statement. Press ctrl+ to see the available choices.

output Seurat

The output is a scaled Seurat object. You need to introduce a name for it.

12.6 Plotting Seurat objects

While there are statements in Seurat that create plots as an auxiliary artifact, there are also dedicated statements to create plots. These plots are usually used to interactively explore the data and decide on parameters for the next statements in the analysis.

12.6.1 Diagnostic plots

One important statement, that is usually used directly after loading a Seurat object, is the `Diagnostic plots` statement. You can create this statement by clicking Plots Diagnostic plots in the context assistant, or by typing the alias `Diagnostic plots` on an empty line of Analysis (see Figure 12.11 for an example of a new `Diagnostic plots` statement).

```
Diagnostic plots for introduce referenced seurat -> number of genes detected per cell - violin plot: violinNGene
  width: 300                                        number of UMIs per cell - violin plot: violinNUMI
  height: 300                                       percentage of mithocondrial genes per cell - violin plot: violinMito
                                                    nGene and nUMI - scatter plot: scatterNUMINGene
                                                    nUMI and percent.mito - scatter plot: scatterNUMIMito
```

Figure 12.11: **New Diagnostic Plots statement**

input Seurat

You need to specify the name of the input Seurat object. Press ctrl+ to see the available Seurat objects.

width and height

You can also specify the width and height of the five plots, all at once, by modifying these two parameters.

output plots

There are five output plots from this statement. Three of them are violin plots for the number of genes in the cell (nGene), number of unique molecular identifier in the cell (nUMI) and the percentage of mitochondrial genes in the cell (percent.mito). The other two are scatter

plots showing percent.mito versus nUMI, and nGene versus nUMI. These five plots have default names, but you can change these names by clicking on the name and typing. To visualize these plots, you can use the `multiplot` statement (see Subsection 3.9.7).

These five plots are typically used to decide on thresholds for the rejection of cells or other parameters of the cleanup statement.

12.6.2 Features plot

The features plot highlights the specified features on a plot with tSNE clusters. This statement is useful in exploring markers for the clusters. You can create this statement by clicking Plots 〉 Features plot in the context assistant, or by typing the alias `Feature plot` on an empty line of Analysis (see Figure 12.12 for an example of a new `Feature plot` statement).

```
Feature plot with [ << ... >> ] for introduce referenced seurat -> plot : <no name>
```

Figure 12.12: **New feature plot statement.** A new `Feature plot` statement expects a list of features, an input Seurat object and a name for the output Seurat object.

input Seurat

The input Seurat object that has the tSNE computed already. If this is not the case, an error will be reported.

features

You need to specify a list of features that will be highlighted in the clusters. The output plot is actually a collection of smaller plots, one per feature; each smaller plot highlights one feature in the clusters. To enter feature names, press ⏎ inside the square brackets of the statement, and type in a name. To introduce another feature in the list, press ⏎ at the end of a feature.

output plot

You need to introduce the name of the output plot.

12.6.3 Features and total plot

The features and total plot highlights the specified features and their cumulated effect (obtained by multiplying the expressions of these genes) on a plot with tSNE clusters. This statement is useful to see whether the specified features characterize a cluster together. You can create this statement by clicking Plots 〉 Features and total plot in the context assistant, or by typing the alias `Feature plot and total` on an empty line of Analysis (see Figure 12.13 for an example of a new `Feature plot and total` statement). The output plot for such a statement where features “PLEK" and “SDPR" are given as input, can be seen in Figure 12.14.

The parameters for this statement are identical to the parameters for the `Feature plot` statement.

```
Feature plot and total with [<< ... >>] for introduce referenced seurat -> plot : <no name>
```

Figure 12.13: **New feature plot and total statement.** A new `Feature plot and total` statement expects a list of features, an input Seurat object and a name for the output Seurat object.

```
multiplot -> multiPlot2Features [  1 cols x 1 rows ]  Hide preview
 [ twoFeaturesPlot ]
```

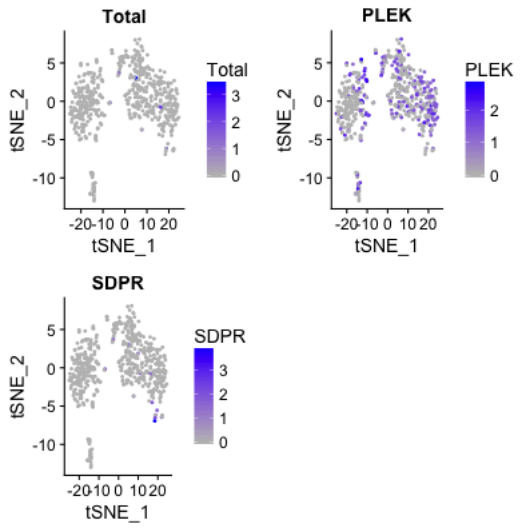


Figure 12.14: **Two features and their cumulated expression highlighted.** This figure shows plots for two features (genes), and their cumulated expression (Total).

12.7 Adding information to Seurat objects

Another important step in the analysis of scRNA-seq data is the computation of principal components, of tSNE clusters and of the markers per cluster. There are three dedicated statements in Seurat, for the three functionalities.

12.7.1 Add principal component information

The `add principal components` statement computes the principal components for the expression data in the input Seurat object. You can create this statement by clicking Add 〉 Principal component info in the context assistant, or by typing the alias `add principal components` on an empty line of Analysis (see Figure 12.15 for an example of a new `add principal component` statement).

```
add principal components  for introduce referenced seurat  -> seurat: addInfo
                                                             standard deviation of PCs plot  : <no name>
```

Figure 12.15: **New add principal components statement.** A new `add principal components` statement expects an input Seurat object, a name for the output Seurat object and a name for the auxiliary plot produced.

input Seurat

The input Seurat object for which principal component analysis is added.

output Seurat

The output Seurat object will have the principal component analysis property set to true. You can change the name of this object.

plot

This statement produces a plot as an auxiliary artifact. The plot shows the standard deviations of the principal components. This plot helps you decide which PCs to use further in the analysis, by choosing the cutoff where there is an elbow in the plot. For an example, see Figure 12.16.

12.7.2 Add clusters information

The `add clusters` statement computes the tSNE clusters for the expression data in the input Seurat object. This statement needs PC information, so it needs to be run after the `add principal components` statement. You can create this statement by clicking Add 〉 Clusters info in the context assistant, or by typing the alias `add clusters` on an empty line of Analysis (see Figure 12.17 for an example of a new `add clusters` statement).

input Seurat

The input Seurat object for which tSNE clusters information is added.

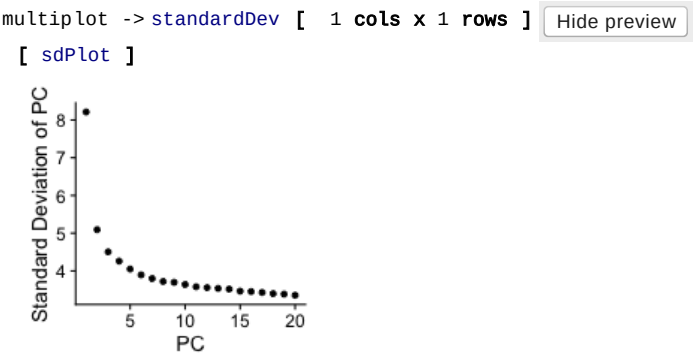


Figure 12.16: **The standard deviations of principal components plot.** Given this figure, we would choose 17 as the cutoff value, because it resides at the elbow of the plot.

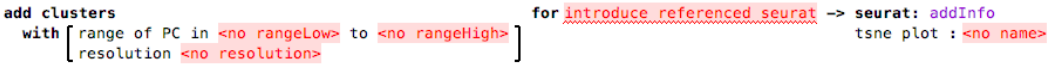


Figure 12.17: **New add clusters statement.** A new `add clusters` statement produces an auxiliary plot showing the tSNE clusters and uses information from the principal components analysis via the parameters captured in range of PC.

range of PCs

Based on the cutoff you see in the plot produced by the principal component analysis statement, you fill in the range of PCs to use in the tSNE analysis.

resolution

This parameter affects the number of clusters the tSNE analysis will produce (the larger the resolution, the more clusters it produces). A typical value for this parameter is 0.6.

output Seurat

The output Seurat object will have the tSNE property set to true.

plot

This statement also produces a plot that shows the tSNE clusters.

12.7.3 Add markers information

Once clusters of cells have been identified, you might want to find genes whose expression is specific of the cluster. Seurat calls these genes cluster markers. The `add markers` statement computes the markers for each cluster, so it needs to be run on an input Seurat object that has tSNE clustering information. You can create this statement by clicking Add 〉 〉Markers per cluster info in the context assistant, or by typing the alias `add markers` on an empty line of Analysis (see Figure 12.18 for an example of a new `add markers` statement).

input Seurat

The input Seurat object for which markers information is added.

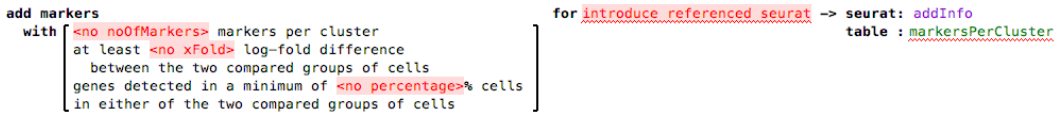


Figure 12.18: **New add markers statement.** A new `add markers` statement produces an auxiliary table with the identified markers, for each cluster.

output Seurat

The output Seurat object that will have the markers information.

number of markers

This parameters allows you to specify how many markers per cluster you want to compute.

xFold

This parameter specifies the minimum log-fold difference between the two compared groups of cells when computing the markers.

percentage

This parameter specifies what is the minimum percentage of cells in either of the two groups of cells, where genes need to be detected when computing the markers.

table

This statement also produces a table that contains the markers per cluster. You can see the columns of the table by clicking on it and looking in the 2: Inspector tab.

12.8 Aligning Seurat objects

One common scenario in the scRNA-seq data analysis, is the comparison of different samples. Before doing comparisons between these different samples, it is important to align them. To this end, Seurat 2.0+ provides an approach to align cells found in two Seurat objects. MetaR wraps this functionality in the following analysis statements.

12.8.1 Prealign Seurat objects

The pre-align statement prepares the Seurat objects for alignment and it outputs a plot that is used to decide on parameters for the alignment statement. You can create this statement by clicking Align > Prealign in the context assistant, or by typing the alias `prealign seurats` on an empty line of Analysis (see Figure 12.19 for an example of a new `prealign seurats` statement).

input seurats

This statement expects two input Seurat objects that will be aligned by this statement.

```
prealign seurats introduce referenced seurat and introduce referenced seurat -> seurat: introduce name
  heatmaps - from dimension: <no dim1>                                          CCA plot (CC1 versus CC2): preCCA
  heatmaps - to dimension: <no dim2>                                            heatmaps for given dimensios: heatmapDims
```

Figure 12.19: **New prealign statement.** The `prealign seurats` statement outputs a plot showing the state of the first two canonical correlation vectors (used under the hood), and also heatmaps for as many of these vectors as you specify in the input parameter.

heatmaps for canonical correlation vectors up to

The canonical correlation (CC) vectors specified in the range of this statement are shown in heatmaps so that you can assess what CCs to use further in the analysis, in particular, for the alignment. You can decide to choose up to the latest CC that shows some variation. Figure 12.20 shows an example.

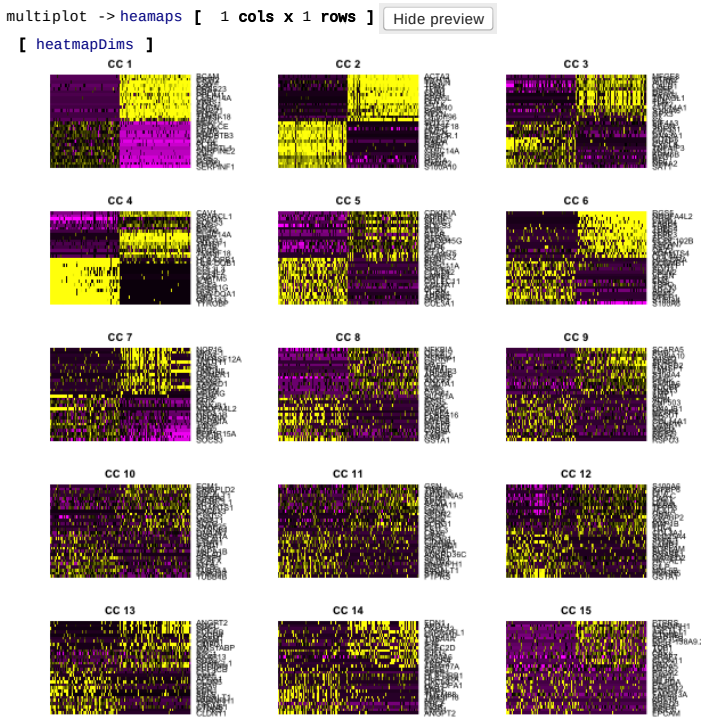


Figure 12.20: **Heatmaps showing the CC vectors and their scores per gene.** For this particular example, we choose CCs up to 13, because this is where the signal starts to fade.

output Seurat

The Seurat object that is going to contain the aligned result.

CCA plot

This plot shows information from the first two CC vectors so that you can assess whether the alignment can be made.

heatmaps plot

This plot contains the heatmaps for the CC vectors specified at the input. These heatmaps can be used to decide what CCs to use further down in the analysis.

12.8.2 Align Seurat object

The `align seurat` statement aligns the two Seurat objects provided in the pre-alignment. You can create this statement by clicking Align » Align in the context assistant, or by typing the alias `align seurat` on an empty line of Analysis (see Figure 12.21 for an example of a new `align seurat` statement).

```
align seurat introduce referenced seurat  -> seurat: introduce name
  cca from dimension: <no dim1>             tsne clusters with datasets: tsneClustersDatasets
  cca to dimension: <no dim2>               tsne clusters: tsneClusters
```

Figure 12.21: **New align statement.** A new align statement needs information obtained from the prealign statement and outputs two auxiliary tSNE plots.

input Seurat

The input Seurat should be the Seurat object obtained from the `prealign seurats` statement.

CCA dimensions from - to

In these parameters, you specify the CCs that you have chosen as a result of inspecting the heatmaps from the `prealign seurats` statement.

output Seurat

The output Seurat contains aligned data from the two Seurat objects given as input to the prealign statement.

tSNE plots

This statement generates two plots with the tSNE clusters, one where the two datasets from the two input Seurat objects are highlighted, and one where the clusters themselves are highlighted.

12.9 Limma for Seurat objects

In Seurat, there are special statements used for the differential expression analysis using the limma package.

12.9.1 Pre-limma Seurat object

The pre-limma statement extracts an aggregated count table from the Seurat object given as input. This is needed to obtain counts for genes in individual clusters. The procedure simply sums the counts of a gene across cells of a cluster, while preserving experimental conditions. To this end, we aggregate columns per individual Seurat object (we kept track

of Seurat objects that were merged, or aligned)and per tSNE cluster. You can create this statement by clicking Limma 〉 Prelimma in the context assistant, or by typing the alias `pre limma` on an empty line of Analysis (see Figure 12.22 for an example of a new `pre limma` statement).

```
pre limma                          for introduce referenced seurat -> aggregate counts with names table : aggregateCountsWithNames
  from tsne cluster: <no clustersFrom>
  to tsne cluster: <no clustersTo>
```

Figure 12.22: **New pre limma statement.** The `pre limma` statement generates an aggregated count table and needs tSNE cluster information as input.

input Seurat

The input is a Seurat object for which we want to compute the differentially expressed genes.

tSNE clusters from - to

These parameters need to be filled in with the clusters that result from the tSNE computation.

output table

The aggregated count table contains one column per input Seurat object and per cluster. For instance, if we merged *P1C* and *P1D* into *P1*, *P2C* and *P2D* into *P2*, we aligned *P1* and *P2* into *P1P2*, and we have obtained clusters *0* and *1*, then we get columns *P1C0*, *P1C1*, *P1D0*, etc. Besides these columns, the output table contains a column with gene names and has also row names.

The annotations that were introduced when loading the Seurat objects will be propagated to the output table.

12.9.2 Limma voom

The limma statement computes the differentially expressed genes for the input count table and outputs a table with results per contrast. You can create this statement by clicking Limma 〉 Limma in the context assistant, or by typing the alias `limma voom` on an empty line of Analysis (see Figure 12.23 for an example of a new `limma voom` statement).

```
limma voom          for <no table> -> << ... >>
  model formula ~ 0
  comparisons:
```

Figure 12.23: **New limma statement.** A limma statement applied on a table containing expression data.

input table

The input table should be a table containing expression data. Moreover, the table needs to be annotated with some group usages that will subsequently be used in the model formula and

in contrasts.

output tables

For each contrast specified in this statement, there will be one output results table containing information such as the log-fold change and adjusted P-value. Moreover, each of the output tables comes with a path to a directory where an interactive web plot is generated.

For the explanation of the other parameters of the limma statement, look at the parameters described in the main limma statement provided by MetaR in Chapter 8. The only difference is that in the Seurat statement, one can specify multiple contrasts (under the `comparisons` attribute).

12.10 Other Seurat statements

There are two more Seurat statements that are useful at various places in the analysis: the merge and the delete statements.

12.10.1 Merge Seurat objects

The merge statement takes two Seurat objects as input and creates one single output Seurat object as output. Under the hood, this statement merges the genes in the two Seurat objects and keeps all the cells from the two objects. You can create this statement by clicking Merge in the context assistant, or by typing the alias `merge seurat object` on an empty line of Analysis (see Figure 12.24 for an example of a new `merge seurat objects` statement).

```
merge seurat objects introduce referenced seurat introduce referenced seurat -> seurat: mergedSeurat
```

Figure 12.24: **New merge seurat objects statement.** A merge statement needs two input Seurat objects and returns them merged in the output Seurat object.

input Seurat objects

You need to provide two input Seurat objects.

output Seurat object

The output Seurat objects contains all the information from both input Seurat objects.

12.10.2 Delete Seurat object

This statement deletes the provided Seurat object. After this statement, the input Seurat object is not available as input anymore. This statement was introduced to save memory. You can create this statement by clicking Delete in the context assistant, or by typing the alias `delete seurat` on an empty line of Analysis (see Figure 12.25 for an example of a new `delete` statement).

input Seurat

You need to introduce the Seurat object that needs to be deleted.

```
delete seurat introduce referenced seurat
```

Figure 12.25: **New delete statement.** A delete statement needs only the Seurat object to be deleted as input.

13 — Simulating Datasets

13.1 Why simulating datasets

Simulated datasets are useful to check that analyses work as expected. MetaR provides a `Simulate Dataset` command that allows to simulate datasets starting from a few assumptions and parameter values. This approach could be also beneficial at experimental design time to validate certain assumptions before running (expensive) experiments.

In order to use the command inside an Analysis script, the *simulation* language (`org.campagnelab.metaR.simulation`) has to be imported in the current model (use ⌘+L and select the language from the list). Figure 13.1 shows a new `simulate dataset` statement.

```
simulate dataset  with  [
  num of samples:  #sample
  num of genes:  #genes
  mean when all factors are false:  <no mean>
  discrete factors:  factor name
  effect size:  effect of dicrete factors
  continuous covariate:  factor name , range: [ lower limit - upper limit ] , slope: <no linear_slope>
] -> simulate
```

Figure 13.1: **New Simulate Dataset Statement.** A new `simulate dataset` statement created after you have added the `org.campagnelab.metaR.simulation` language to the model's MPS Used Languages.

13.2 The Simulate Dataset Statement

The `simulate dataset` statement is configurable and lets you create datasets that reflect different simulation scenarios. The output dataset is represented by a `Table` node that can be then further manipulated with other MetaR statements.

num of samples

The number of samples included in the dataset. Each sample is named according to the results of the simulation. If the simulation decides that the sample name *sample_3* has been

treated with a discrete factor named *LPS*, it is renamed to *sample_3_LPS* to make it easy to identify the simulated treatment.

num of genes

The number of genes included in the dataset. Each gene is renamed according to the results of the simulation. If the simulation decides that the gene named *gene_2* is affected by a discrete factor named *LPS*, it is renamed to *gene_2_LPS* to make it easy to identify the simulated treatment.

mean

The value of the mean expression level for each gene, assuming no treatment.

discrete factors

List of treatments used in the simulation. About 50% of the samples will be considered treated with each factor specified here. About 30% of the genes will be considered affected by each factor.

effect of discrete factors

The impact of each discrete factor on the data generated by the simulation

continuous covariate

A covariate that will affect the value of the gene expression level. You can define the age of the covariate, its range and the slope. A value is added to the expression value of each gene equal to the product of the slope and the cofactor value. Cofactors are set for each sample using a uniform distribution. For instance, if you indicate an 'age' continous covariate between 0 and 36, each sample will be assigned an age in this range, and the value added to the expression level of the gene will be determine for each sample by multiplying the age of the sample by the slope of the 'age' cofactor.

13.3 Example

```
simulate dataset  with  [
  num of samples:  10
  num of genes:  20
  mean when all factors are false:  5
  discrete factors:  LPS
  effect size: 100
  continuous covariate:  age , range: [ 0 - 36 ] , slope: 10
] -> simulate
```

Figure 13.2: **Simulate Dataset Example.** This statement will create a dataset with a single discrete factor (LPS) and a covariate factor (age) with a range of 0 to 36 (for instance, this could be the mouse age in a mouse model).

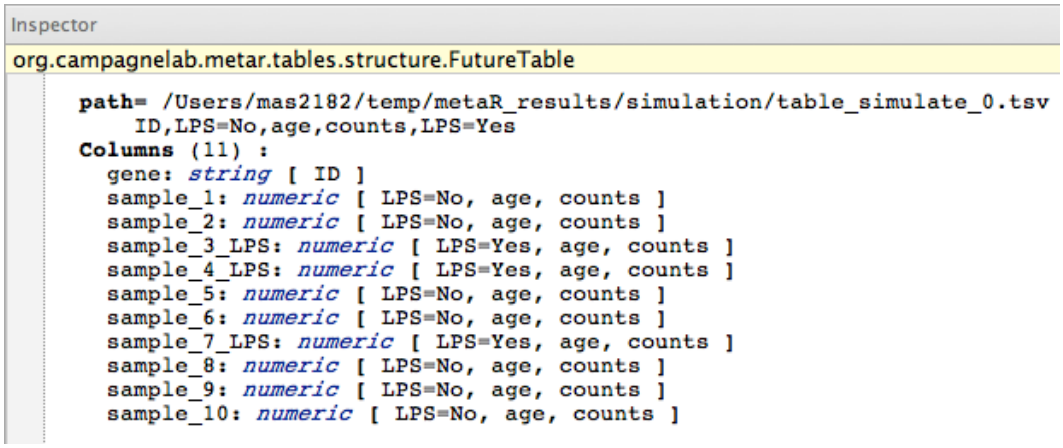


Figure 13.3: **Preview of the Dataset Structure as Shown in the Inspector.**

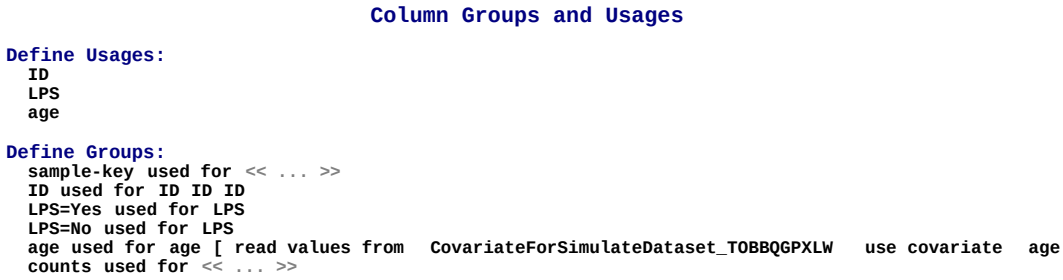


Figure 13.4: **Column Group Annotations Created in the Model.**

MetaR Table Viewer

table: CovariateForSimulateDataset

SampleName	age
sample_1	23
sample_2	21
sample_3_LPS	25
sample_4_LPS	7
sample_5	16
sample_6	24
sample_7_LPS	25
sample_8	8
sample_9	14
sample_10	17

Figure 13.5: **Covariate Table Generated with Simulate Dataset.**

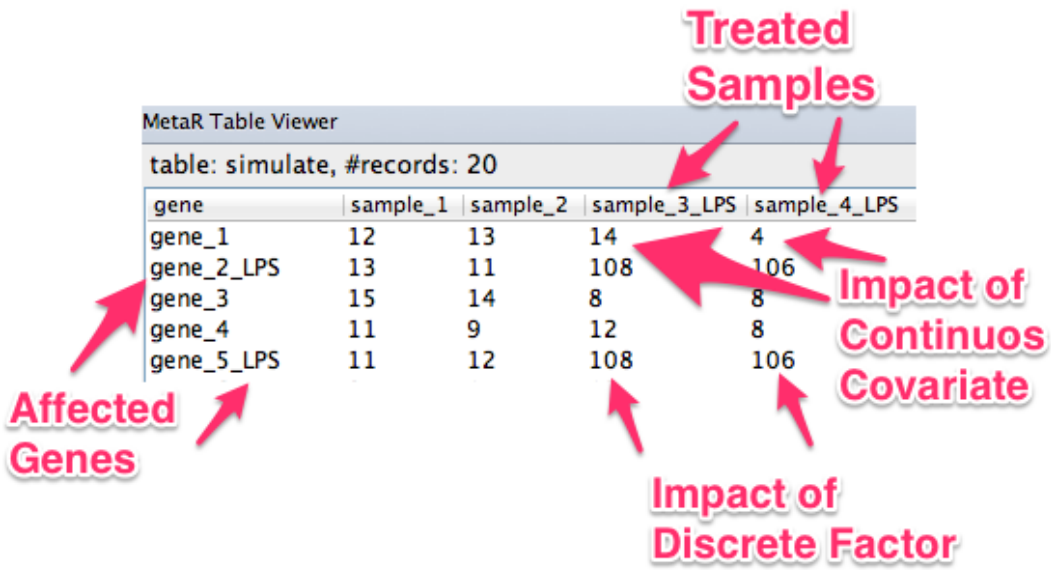


Figure 13.6: **Table Generated with Simulate Dataset.** This is a partial view of the full table.

14 — Extending MetaR

14.1 Overview

Because MetaR is developed in the MPS language workbench, you can use language composition as a way to extend the MetaR language. In this Chapter, we provide a very simple example to illustrate how to extend MetaR through language composition.

Let's assume that you have just learned about the `heatmap.2` function provided in the `gplots` R package. You wish to use this function to create heatmaps with MetaR. To achieve this, you would follow the following steps:

1. Create a new MPS Language.
2. Create a `Heatmap.2` language concept in the Structure Aspect of the language (see [**campagne2014mps**]).
3. Customize the Generator to transform instances of the `Heatmap.2` into R code.

14.2 Create a new Language

Let's create a new language. To do this, select the project and do right-click ⟩ New ⟩ Language. Name the language something like `your.domain.heatmap`.When the language has been created, select its name under the Project Tab and adjust Dependencies to include *org.campagnelab.metaR.tables*. Set the Scope to Extends (this will allow statements of this new language to extend concepts of *org.campagnelab.metaR.tables*).

14.3 Create a new Language Concept

Select the Structure Aspect of the *your.domain.heatmap* language and do right-click ⟩ New ⟩ Concept.[1] Name the concept `Heatmap2`. Define the extends clause to be `Statement` (from language *org.campagnelab.metar.tables*). The resulting concept should appear as shown in Figure 14.1.

[1] You can create a new language in an existing project, or use the New Project Dialog to create a Project with type "Language".

concept alias

Define the alias of the concept. Use `heatmap2`. An instance of the concept will be created when you type this keyword in the editor.

reference to a table

To plot a heatmap, we will take data from a MetaR table. This can be achieved by adding a TableRef child to the `Heatmap2` concept. Set the cardinality to exactly one child (`[1]`).

heatmap produces a plot

The `heatmap2` statement will produce a plot, so you need to add a child of type `Plot`. You may call this child 'plot' for simplicity. Set the cardinality to exactly one child (`[1]`).

```
concept Heatmap2 extends     Statement
                 implements  <none>

  instance can be root:  false
  alias: heatmap2
  short description:  <no short description>

  properties:
  << ... >>

  children:
  << ... >>

  references:
  << ... >>
```

Figure 14.1: **Heatmap2 Concept.** The `Heatmap2` concept extends Statement. When the language is composed with MetaR, `Heatmap2` will become available for auto-completion whenever a MetaR Statement can be used.

Figure 14.2 presents the completed concept after adding alias, table and plot.

14.4 Define the Editor

An MPS editor customizes how a node appears in the editor. Create an editor for `Heatmap2` (see [**campagne2014mps**]) and define its content as shown in Figure 14.3.

14.5 Generate R Code

In order to generate the statement to R code, we will use the MPS Generator aspect.

In the first step, we create a reduction rule, see Figure 14.4 and [**campagne2014mps**] (Generator Aspect chapter). The rule indicates that nodes of the `Heatmap2` concept will be transformed with the `reduce_Heatmap2` template. The template was created using

```
concept Heatmap2 extends     Statement
                 implements <none>

  instance can be root: false
  alias: heatmap2
  short description: <no short description>

  properties:
  << ... >>

  children:
  table : TableRef[1]
  plot  : Plot[1]

  references:
  << ... >>
```

Figure 14.2: **Complete Heatmap2 Concept.** This figure shows the completed `Heatmap2` concept with an alias and children for table and plot.

```
<default> editor for concept Heatmap2
  node cell layout:
    [- heatmap2 % table % -> % plot % -]

  inspected cell layout:
    <choose cell model>
```

Figure 14.3: **Editor of the Heatmap2 Statement.** Notice how the editor simply shows the name of the statement, delegates to the TableRef editor to render the table reference, and delegates to the Plot editor to show the plot child.

the **New Template** intention found on the reduction rule node. See detailed instructions in [**campagne2014mps**]. When configuring the type of the output node (under `content node:`), use `Lines`, from the language *org.campagnelab.textoutput* to produce text with the MPS Generator aspect.

As an alternative, and since MetaR 1.5 supports a full R language implementation in MPS, you could also use the concepts `Expr` or `Exprlist` from the language *org.campagnelab.R*. Doing so would allow using the R language editor to define the output of the reduction rule.

MetaR 1.8 has introduced the language *org.campagnelab.metar.R.inspect*, which you can use when writing generation rules with the *org.campagnelab.R* language. This language facilitates generation for common operations in MetaR. See the `sleuth`, `MA Plot` and `UpSet` statements for examples in the code base that take advantage of the composable R language in the generator.

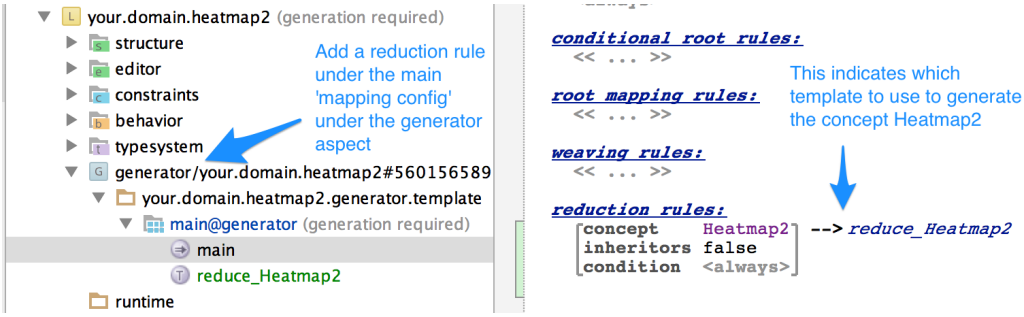


Figure 14.4: **Generate R Code: Step 1.** Start by adding a reduction rule.

The simplest template we can build for the `Heatmap2` concept is shown on Figure 14.5. Simply calling the `heatmap.2` function is a good start, but trying to run this will fail because the function is not part of a plain R distribution. The next section explains how to install the gplot R package which provides the function and activate it.

14.5.1 Adding package and library support

MetaR makes it easy to install R packages as they are needed by an analysis. For this to work, we need to declare that the `Heatmap2` concept depends on the *gplots* R package. We can do this by overriding the default Statement behavior method called `dependencies()`. This method is expected to return the list of package names that must be installed and loaded before the statement can execute. To override the method, navigate to the `Heatmap2` concept in the editor, select the behavior tab, create the behavior and when the empty behavior is shown, select Override Behavior Method (see Figure 14.6). Replace the body of the method with

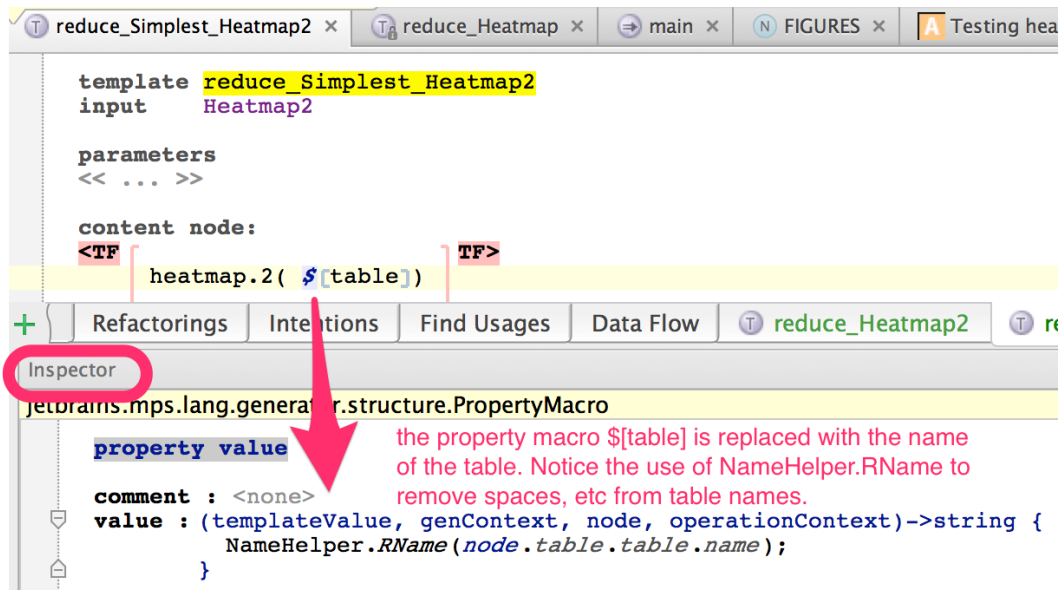


Figure 14.5: **Simplest template.** The template simply converts any node of the `Heatmap2` concept to a line of R code that calls the `heatmap.2` function with the table as an argument. Notice the use of a property macro to obtain the R name of the table variable from the node table attribute. Open the 2: Inspector to see what value the macro will take when a node is generated to R code.

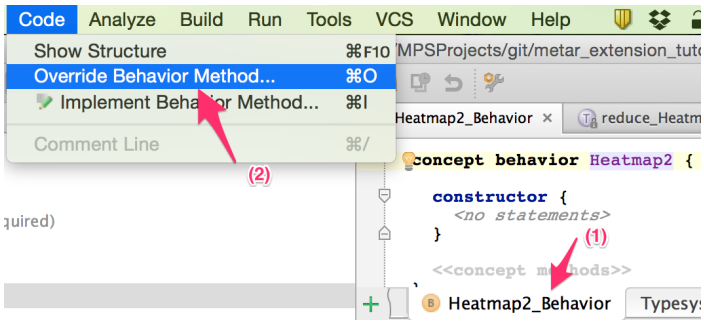


Figure 14.6: **Override Behavior Methods.** When the override dialog appear, choose dependencies() and click OK.

```
public sequence<string> dependencies()
  overrides StatementDependencies.dependencies {
  return new singleton<string>("gplots");
}
```

Figure 14.7: **Complete Dependencies Behavior Method.**

```
return new singleton<string>("gplots");
```

Figure 14.7 shows the complete method. Rebuild the language, then the solution. When you run the Analysis, you will see that the gplots package is being installed:

```
...
Loading required package: gplots
Installing package into '/Users/fac2003/.metaRlibs'
(as 'lib' is unspecified)
also installing the dependencies 'bitops', 'gtools', 'gdata',
    'caTools'

trying URL ....
```

14.5.2 Adjust Generator Priorities

Adjust the generator priorities as shown in Figure 14.8. To set priorities, you need to define a Design dependency on *org.campagnelab.metar.tables* in the generator aspect Module Properties dialog.

14.5.3 Redirecting the plot output

The next problem with the simple template is that it fails to write the image of the plot in location where MetaR can find it. This is needed to display the plot preview, or to make it possible to use plots with the `multiplot` statement. The strategy we use to handle both cases is to wrap the actual plotting code inside a `plot_xxx` function. The function is named with the id of the statement that generates the plot, so that we can easily refer to it in other places where the plot should be reused. We start by defining this function:

```
plot_ $[id]=function(t){
  heatmap.2(as.matrix(t))
```

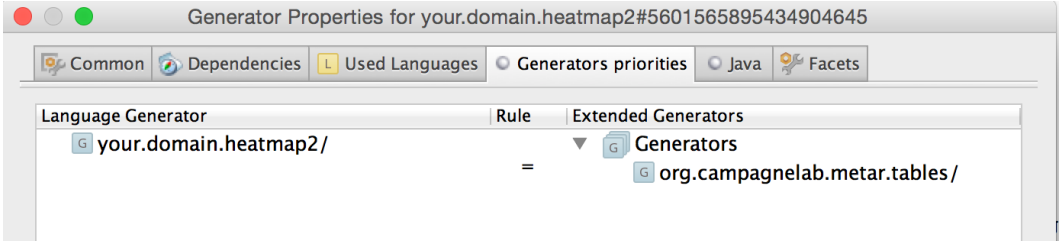


Figure 14.8: **Adjust Generator Priorities.** Generation of the `Heatmap2` concept has to happen in the same generation phase as the generation of the *org.campagnelab.metar.tables* concepts. Failure to adjust priorities may result in table names not being correctly inserted in the `heatmap.2` function call.

```
}
```

The function accepts one argument: the name of the table to plot.

The number of arguments is up to you, since you will generate both the function and the function calls.

The next step is to add the R code for generating a PNG file with the plot to show a preview in the inspector. To this end, we redirect the plot output with png(), call the plot function, and close the graphics output (dev.off()):

```
png(file=" $[plot.png]", width= $[w], height= $[h])
plot_ $id( $[table])
ignore <- dev.off()
```

Notice how the parameters of the png function are taken from the Plot node. For instance, the `$[plot.png]` macro will expand to

```
new RPath(node.plot.getPath()).toString();
```

If you use composable R to write the generator, you can reuse the `DrawPlot` expression provided by *org.campagnelab.metar.R.inspect*. This expression more conveniently generates to the png, plot and ignore expressions.

14.5.4 Handling errors

Accurate error reporting is important to the end-user. When things do not go well and the R code fails, it is useful to know precisely which MetaR statement generated the error. In MetaR, this is done by taking advantage of the `tryCatch` R functions (see `http://mazamascience.com/WorkingWithData/?p=912`).

Because `tryCatch` is not particularly intuitive and is rather verbose, MetaR extends the *org.campagnelab.TextOutput* language with the `tryForNode` statement (concept name:

`TryAndReport`). The `tryForNode` statement attempts to execute some lines of R code, and reports any errors that occur in these lines with a link to the statement that produced the error or warning. The `tryForNode` statement is also responsible for showing the `STATEMENT_EXECUTED/8289224569309332962/` lines in the Run tool, which are hyperlinked to statements in the editor.

The statement `tryForNode` takes an ID, which is the result of the `id()` method defined for each MetaR statement. The value of the ID can be set as usual with a property macro (its value should be `node.id()`). Figure 14.9 shows the completed template for `Heatmap2`. While the statement only needs to call the `heatmap.2` function, handling possible errors and producing plots that can be reused to build multi-panel figures have added quite a few lines to the output.

R MetaR 1.3.1.1 makes it easier to wrap *TextOutput* `Lines` into a `tryForNode` block. Select a node of type `Lines` (highlighted in blue braces) and invoke the intention `Wrap Lines in a TryForNode Block`. Note that this intention is not available for lines already inside a `tryForNode` block.

R If you use composable R to write the generator of a statement, you can find the equivalent of `tryCatch` and `tryForNode` in the *org.campagnelab.metar.R.inspect* language.

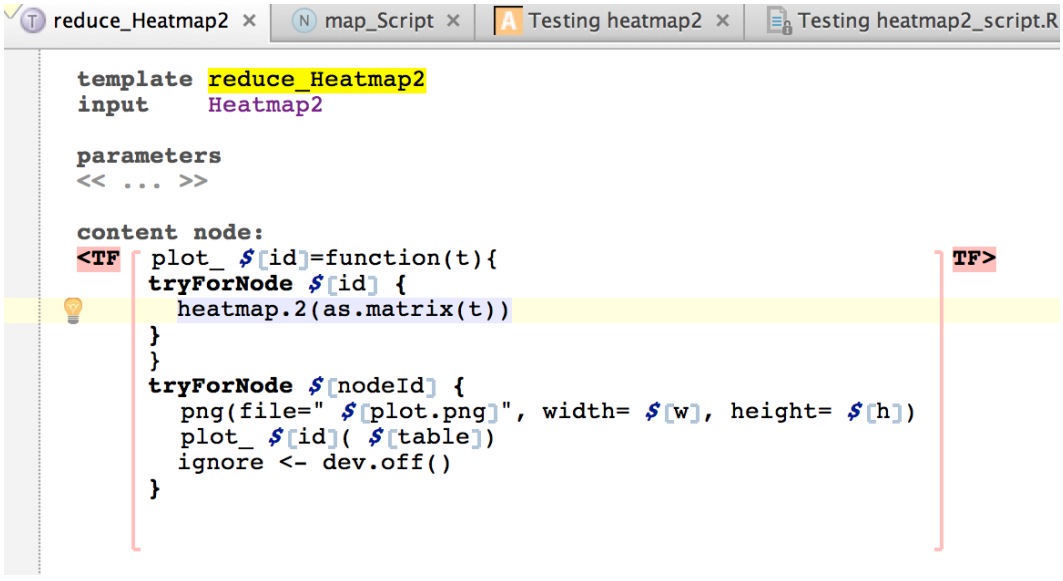
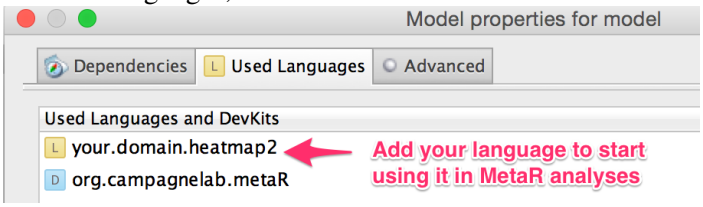


Figure 14.9: **Complete Generator for Heatmap2.** This template wraps the `heatmap.2` function call inside a plot function (needed when building figures with multiple panels) and inside a `tryForNode` statement. The `tryForNode` statement generates to the `tryCatch` R language construct and appropriately reports errors to the end-user.

14.6 Using the New Language

Using the *your.domain.heatmap2* language in MetaR analyses requires adding the language under Used Languages, as shown here:



Once the language is added, you can create statements by typing the `Heatmap2` concept name. Figure 14.10 shows the result of using the new concept.

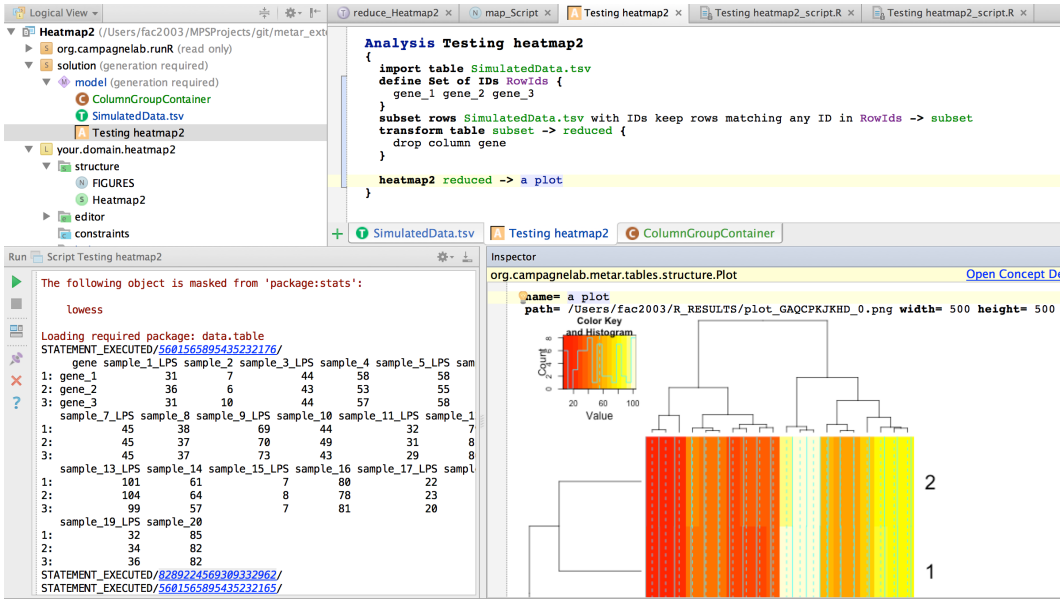


Figure 14.10: **Heatmap2 Execution.** This figure presents an Analysis using the new `Heatmap2` concept and shows the resulting plot preview in the inspector (bottom right). Error handling and progress indicators can be seen on the bottom left.

14.7 Git Repository

This concludes this tutorial. You can find a project with the *heatmap2* language extension described in this Chapter at:

`https://bitbucket.org/campagnelaboratory/metar_extension_tutorial`

15 — Composable R

15.1 Overview

MetaR release 1.5 was the first release to offer a composable R language for MPS (this language is called *org.campagnelab.metaR.R* in MPS, but also simply referred to as "composable R", or *compR*). This Chapter explains how to take advantage of this language, what the advantages of a composable R language are, but also describes the current limitations of its implementation.

15.1.1 Advantages

Composable R supports the full R grammar

The composable R language implemented in MetaR 2.0+ supports the equivalent of the R language grammar.

We have developed composable R starting with the ANTRL4 R grammar obtained from `https://github.com/antlr/grammars-v4/tree/master/r`. This grammar was tweaked to improve priority rules of the parsing rules.

Because this language is composable, you can design micro-languages to compose with R. An example of micro-language composition, to use the biomart statement described in Chapter 10, is shown in Figure 15.1.

Fluent editing

Composable R supports Fluent Editing[1]. We have designed this technique to make it easier to paste an R code fragment from text into an R script. The technique is also very useful to enter complicated R expressions more quickly than possible with the MPS projectional editor. Read on to learn how to use it.

Fluent editing can be used in any context where an expression or function parameter declaration is expected into a composable R script. For instance, assume that you have created a new R script. Place the cursor on a new line of this script, and paste:

[1] To learn how to implement this feature for your own languages, see [**campagne2015mps**]

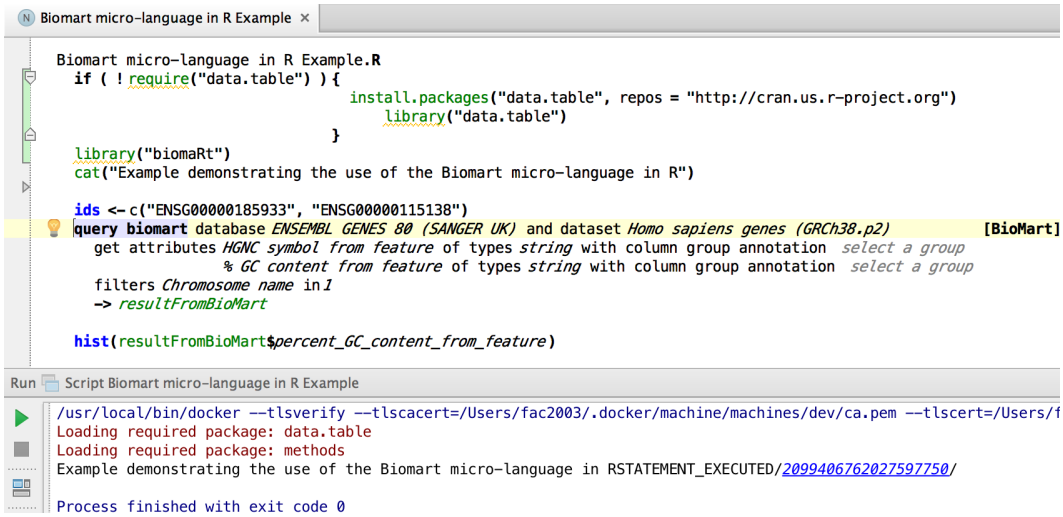


Figure 15.1: **Example of Micro-Language Composition with Composable R.** In this example, we have composed the MetaR query biomart statement (see Chapter 10 with the R language).

```
c(1,2,3,4,5)
```

Immediately after you pasted this expression, fluent editing will parse the expression and translate it to composable R. Since `c` is an identifier that refers to a function, you will see the color of the `c` letter change from blue (an identifier not linked to a declaration) to the color green (used to indicate identifiers linked to a declaration). In this case, the function identifiers becomes linked to the function declaration, in one of the package stubs shipped with MetaR.

You can paste several lines of R code that will be parsed and inserted into the program in a similar manner. Note that you do not need to paste any text and may just start typing into any location that accepts an R expression (`Expr` concept in the *org.campagnelab.metaR.R* language). Press return when you are satisfied the text you typed is parseable and the code will be inserted into the script. Use the auto-completion menu to select Fluent code entry if the text you type generates any ambiguity and press return to parse the text and insert the parsed program into the script.

Pasting expressions and parameter declarations should be sufficient to paste in most contexts given the simple grammar of the R language. Let us know if you find that pasting does not work in some context where you would expect it to. In such cases, try pasting a larger piece of R code that contains the one you need to paste. Pasting should work if this larger piece is an R expression.

15.1.2 Limitations

Composable R is not an R IDE

The goal of this language is not to replace an R integrated development environment (IDE) (e.g., RStudio). A number of capable IDEs for the R language already exist, and it is not our intention to develop another one. Composable R is developed as a research tool that helps us explore the advantages and limitations of language composition for data analysis.

This being said, composable R provides some features commonly found in good IDEs, including:

- auto-completion for function and identifier names.
- auto-completion for language keywords and constructs.
- navigation to function definition or previously defined identifier, within a script.
- ability to define intentions to automate modifications of R code and scripts.
- automatic refactorings (renaming an identifier automatically renames all references to this identifier).
- You can run R scripts directly from within MPS. When you run, composable R scripts are generated to pure R scripts and the scripts is executed either with a local installation of R or with a Docker container.

However, the following features typically found in IDEs are not supported in MetaR:

- An interactive R console,
- An R debugger,
- A view of plots or tables created by a script.

For these reasons, we recommend using Composable R together with a good R IDE.

We have started to address the third limitation in MetaR 2.0. For instance, we offer the ability to preview plots directly inside an R script by using the MetaR `multiplot` statement.

15.2 RScript Root Node

To create a composable R script in a model, make sure *org.campagnelab.metaR.R* is declared in the list of Used Languages. Then select the model, right-click and do o.c.metaR.R.RScript. This will create a root node, such as shown in Figure 15.2 where you can type expressions in the R language.

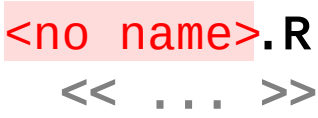


Figure 15.2: **New RScript Root Node.** Press ⏎ over the « ... » to enter new expressions in the script. Enter at least a single new line before pasting code.

Since version 2.0, you can compose MetaR statements into a composable R script. To do this, import the language *org.campagnelab.metar.R.metar* and use the `metar`

auto-completion. This will create a wrapper for a MetaR statement that lets you insert any MetaR statement into the script.

15.2.1 Example

Figure 15.3 shows an example of an RScript written with MetaR. Note the `Save Session` and `installOrLoad` statements at the top to support instant refresh. Similarly, the `export plot` statement wraps the code that produces the plot into a named block of code and makes it possible to refer to the plot in the `multiplot` statement (composed from MetaR)

```
UNHumanDevelopmentReport .R
  installOrLoad ggplot2
  installOrLoad scales
  installOrLoad tidyr
  installOrLoad ggrepel
  installOrLoad grid

  data.dir <- "${org.campagnelab.metaR.home}/data "
  dat <- read.csv(file.path(data.dir, "IR-demo", "EconomistData.csv"))
  pc1 <- ggplot(dat, aes(x = CPI, y = HDI, color = Region))
  Save Session
  pc2 <-
      pc1 + geom_smooth(aes(group = 1), method = "lm", formula = y ~ log(x), se = FALSE, color = "red") + geom_point()
  pc2 <- pc2 + geom_point(shape = 1, size = 4)
  pc3 <- pc2 + geom_point(size = 4.5, shape = 1) + geom_point(size = 4, shape = 1) + geom_point(size = 3.5, shape = 1)
  pointsToLabel <- c("Russia", "Venezuela", "Iraq", "Myanmar", "Sudan", "Afghanistan", "Congo", "Greece", "Argentina",
      "Brazil", "India", "Italy", "China", "South Africa", "Spane", "Botswana", "Cape Verde", "Bhutan", "Rwanda",
      "France", "United States", "Germany", "Britain", "Barbados", "Norway", "Japan", "New Zealand", "Singapore")
  pc3 <- pc3 + geom_text_repel(aes(label = Country), color = "gray20", data = subset(dat,
      Country %in% pointsToLabel), force = 10)
  dat$ Region <- factor(
      dat$ Region, levels = c("EU W. Europe", "Americas", "Asia Pacific", "East EU Cemt Asia", "MENA",
      "SSA"), labels = c("OECD", "Americas", "Asia &\nOceania", "Central &\nEastern Europe",
      "Middle East &\nnorth Africa", "Sub-Saharan\nAfrica"))
  pc3$ data <- dat
  Save Session
  pc4 <-
      pc3 + scale_x_continuous(name = "Corruption Perceptions Index, 2011 (10=least corrupt) ", limits = c(0.9,
      10.5), breaks = 1 : 10) + scale_y_continuous(name = "Human Development Index, 2011 (1=Best) ", limits = c(0.2,
      1.0), breaks = seq(0.2, 1.0, by = 0.1)) + scale_color_manual(name = "d", values = c("#24576D", "#099DD7",
      "#28AADa", "#248E84", "#F2583F", "#96503F")) + ggtitle("Corruption and Human development ")
  pc5 <-
      pc4 + theme_minimal() + theme(text = element_text(color = "gray20"), legend.position = c(
      "top"), legend.direction = "horizontal", legend.justification = 0.1, legend.text = element_text(
      size = 11, color = "gray10"), axis.text = element_text(face = "italic"), axis.title.x = element_text(
      vjust = - 1), axis.title.y = element_text(vjust = 2), axis.ticks.y = element_blank(), axis.line = element_line(
      color = "gray40", size = 0.511111), axis.line.y = element_blank(), panel.grid.major = element_line(
      color = "gray50", size = 0.5), panel.grid.major.x = element_blank())
  mR2 <- summary(lm(HDI ~ log(CPI), data = dat))$ r.squared
  export plot -> Output {
    print(pc5)
    grid.text(
        "Sources: Transparency International; UN Human Development Report ", x = 0.02, y = 0.09, just = "left",
        draw = TRUE)
    grid.segments(x0 = 0.81, x1 = 0.825, y0 = 0.90, y1 = 0.90, gp = gpar(col = "red"), draw = TRUE)
    grid.text(paste0("R² = ",
        as.integer(mR2 * 100), "%"), x = 0.835, y = 0.90, gp = gpar(col = "gray20"), draw = TRUE, just = "left")
    grid.text(paste0("Some Text"), x = "0.5", y = "0.8", gp = gpar(col = "black"), draw = "TRUE", just = "left")
  }
  multiplot -> Multiplot [  1 cols x 1 rows ]  Preview
   [ Output ]
```

Figure 15.3: **United Nation Development Plot script with MetaR.** This script was pasted into MetaR from the code example available at `http://tutorials.iq.harvard.edu/R/Rgraphics/Rgraphics.html`. Note the use of language composition that adds new statements and associated semantic to regular R code.

15.2.2 Execution

RScript root nodes can be executed directly from within MPS. Select an RScript, right-click and do Run 'Script <name>', where name is the name of the RScript you wish to execute. Notice that you can provide program parameters in the Run Configuration dialog. Docker execution (see Chapter 4 is also supported).

Since MetaR 2.0, composable R scripts support instant refresh. Changing the script will trigger re-runs of the script in the background and will update plots and tables in real time. See Chapter 6 for details.

15.3 installOrLoad statement

This statements conveniently loads a package, or installs and loads it if the package was not present in the R distribution used to execute the script. This statement takes a single argument: the name of the package/library to load. The inspector also offer the ability to customize the CRAN repository/mirror used to download and install the package. The following statement:

```
installOrLoad session
installOrLoad ggplot2
```

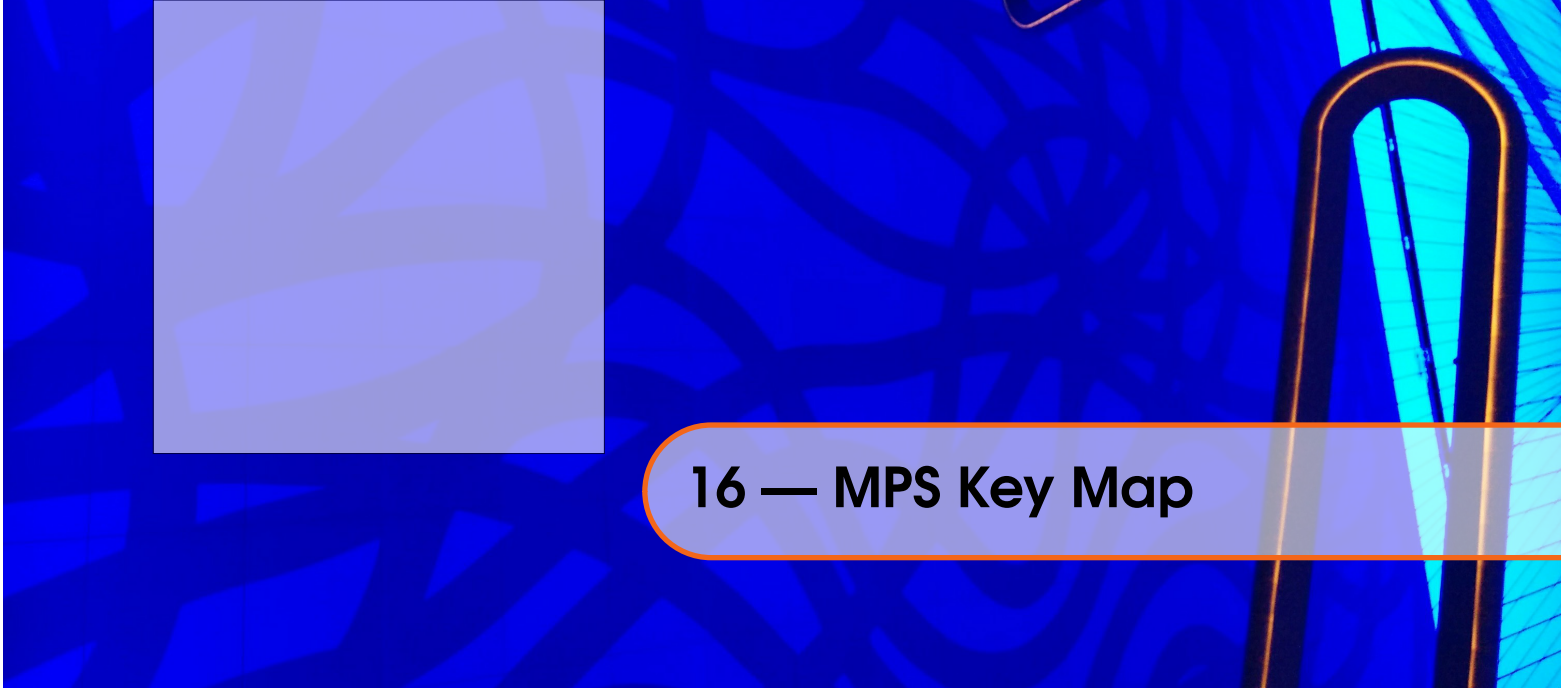
will generate the following R code:

```
installOrLoad<-function (lib,
                repo="http://cran.us.r-project.org")   {
                if(!require(lib,character.only=TRUE)){
                        install.packages(lib,repos=repo)
                          library(lib,character.only=TRUE)
                  }
  }
installOrLoad("session")
installOrLoad("ggplot2")
```

and result in installing and or loading the session and ggplot2 packages.

15.4 Package Stubs

The composable R language also defines the Stub root node, discussed in Section 11.5. Stubs contain description of the functions and function paramters offered by R packages and can be created automatically from the name of the package.



16 — MPS Key Map

Windows or Linux	MacOS	Action
⌥ + 0-9	⌥ + 0-9	Open corresponding tool window
ctrl + S	⌘ + S	Save all
ctrl + ⌥ + F11	N or A	Toggle full screen mode
ctrl + ⇧ + F12	N or A	Toggle maximizing editor
ctrl + BackQuote	ctrl + BackQuote	Quick switch current scheme
ctrl + ⌥ + S	⌘ + Comma	Open Settings dialog
ctrl + ⌥ + C	⌘ + ⌥ + C	Model Checker

Table 16.1: General

Windows or Linux	MacOS	Action
⌥ + F7	⌥ + F7	Find usages
ctrl + ⌥ + ⇧ + F7	⌘ + ⌥ + ⇧ + F7	Highlight cell dependencies
ctrl + ⇧ + F6	⌘ + ⇧ + F6	Highlight instances
ctrl + ⇧ + F7	⌘ + ⇧ + F7	Highlight usages
ctrl + F	⌘ + F	Find text
F3	F3	Find next
⇧ + F3	⇧ + F3	Find previous

Table 16.2: Usage and Text Search

Windows or Linux	MacOS	Action
ctrl + M	⌘ + M	Import model
ctrl + L	⌘ + L	Import language
ctrl + R	⌘ + R	Import model by root name

Table 16.3: Import

Windows or Linux	MacOS	Action
ctrl +	ctrl +	Code completion
ctrl + ⌥ + click	⌘ + ⌥ + click	Show descriptions of error or warning at caret
⌥ + ⏎	⌥ + ⏎	Show intention actions
ctrl + ⌥ + T	⌘ + ⌥ + T	Surround with...
ctrl + X or ctrl + ⇧ + ⌦	⌘ + X	Cut current line or selected block to buffer
ctrl + C ctrl + Insert	⌘ + C	Copy current line or selected block to buffer
ctrl + V ⇧ + Insert	⌘ + V	Paste from buffer
ctrl + D	⌘ + D	Up current line or selected block
⇧ + F5	⇧ + F5	Clone root
ctrl + ↑ or ↓	⌘ + ↑ or ↓	Expand or Shrink block selection region
ctrl + ⇧ + ↑ or ↓	⌘ + ⇧ + ↑ or ↓	Move statements Up or Down
⇧ + Arrows	⇧ + Arrows	Extend the selected region to siblings
ctrl + W	⌘ + W	Select successively increasing code blocks
ctrl + ⇧ + W	⌘ + ⇧ + W	Decrease current selection to previous state

Table 16.4: Editing (Part 1/2)

Windows or Linux	MacOS	Action
ctrl + Y	⌘ + Y	Delete line
ctrl + Z	⌘ + Z	Undo
ctrl + ⇧ + Z	⌘ + ⇧ + Z	Redo
⌥ + F12	⌥ + F12	Show note in AST explorer
F5	F5	Refresh
ctrl + MINUS	⌘ + MINUS	Collapse
ctrl + ⇧ + MINUS	⌘ + ⇧ + MINUS	Collapse all
ctrl + PLUS	⌘ + PLUS	Expand
ctrl + ⇧ + PLUS	⌘ + ⇧ + PLUS	Expand all
ctrl + ⇧ + 0-9	⌘ + ⇧ + 0-9	Set bookmark
ctrl + 0-9	ctrl + 0-9	Go to bookmark
Tab	Tab	Move to the next cell
⇧ + Tab	⇧ + Tab	Move to the previous cell
Insert	ctrl + N	Create Root Node (in the Project View)

Table 16.5: Editing (Part 2/2)

Windows or Linux	MacOS	Action
ctrl + B or ctrl + click	⌘ + B or ⌘ + click	Go to root node / Go to declaration
ctrl + N	⌘ + N	Go to file
ctrl + ⇧ + N	⌘ + ⇧ + N	Go to node by id
ctrl + G	⌘ + G	Go to action by name
ctrl + ⇧ + A	⌘ + ⇧ + A	Go to model
ctrl + ⌥ + ⇧ + M	⌘ + ⌥ + ⇧ + M	Go to solution
ctrl + ⌥ + ⇧ + S	⌘ + ⌥ + ⇧ + S	Go to concept declaration
ctrl + ⇧ + S	⌘ + ⇧ + S	Go to concept editor declaration
ctrl + ⇧ + E	⌘ + ⇧ + E	Go to next or previous editor tab
⌥ + Left or Right	ctrl + Left or Right	Go to editor (from tool window)
Esc	Esc	Hide active or last active window
⇧ + Esc	⇧ + Esc	Restore default window layout
⇧ + F12	⇧ + F12	Hide all tool windows
ctrl + ⇧ + F12	⌘ + ⇧ + F12	Jump to the last tool window
F12	F12	

Table 16.6: Navigation (Part 1/2)

Windows or Linux	MacOS	Action
ctrl + E	⌘ + E	Recent nodes popup
ctrl + ⌥ + Left or Right	⌘ + ⌥ + Left or Right	Navigate back or forward
⌥ + F1	⌥ + F1	Select current node in any view
ctrl + H	⌘ + H	Concept or Class hierarchy
F4 or ⏎	F4 or ⏎	Edit source or View source
ctrl + F4	⌘ + F4	Close active editor tab
⌥ + 2	⌥ + 2	Go to inspector
ctrl + F10	⌘ + F10	Show structure
ctrl + ⌥ +)	⌘ + ⌥ +)	Go to next project window
ctrl + ⌥ + (	⌘ + ⌥ + (	Go to previous project window
ctrl + ⇧ + Right	ctrl + ⇧ + Right	Go to next aspect tab
ctrl + ⇧ + Left	ctrl + ⇧ + Left	Go to previous aspect tab
ctrl + ⌥ + ⇧ + R	⌘ + ⌥ + ⇧ + R	Go to type-system rules
ctrl + ⇧ + T	⌘ + ⇧ + T	Show type
ctrl + H	ctrl + H	Show in hierarchy view
ctrl + I	⌘ + I	Inspect node

Table 16.7: Navigation (Part 2/2)

Windows or Linux	MacOS	Action
ctrl + F9	⌘ + F9	Generate current module
ctrl + ⇧ + F9	⌘ + ⇧ + F9	Generate current model
⇧ + F10	⇧ + F10	Run
ctrl + ⇧ + F10	⌘ + ⇧ + F10	Run context configuration
⌥ + ⇧ + F10	⌥ + ⇧ + F10	Select and run a configuration
ctrl + ⇧ + F9	⌘ + ⇧ + F9	Debug context configuration
⌥ + ⇧ + F9	⌥ + ⇧ + F9	Select and debug a configuration
ctrl + ⌥ + ⇧ + F9	⌘ + ⌥ + ⇧ + F9	Preview generated text
ctrl + ⇧ + X	⌘ + ⇧ + X	Show type-system trace

Table 16.8: Generation

Windows or Linux	MacOS	Action
ctrl + O	⌘ + O	Override methods
ctrl + I	⌘ + I	Implement methods
ctrl + /	⌘ + /	Comment or uncomment with block comment
ctrl + F12	⌘ + F12	Show nodes
ctrl + P	⌘ + P	Show parameters
ctrl + Q	ctrl + Q	Show node information
ctrl + Insert	ctrl + N	Create new ...
ctrl + ⌥ + B	⌘ + ⌥ + B	Go to overriding methods or Go to inherited classifiers
ctrl + U	⌘ + U	Go to overridden method

Table 16.9: BaseLanguage and Editing

Windows or Linux	MacOS	Action
ctrl + K	⌘ + K	Commit project to VCS
ctrl + T	⌘ + T	Update project from VCS
ctrl + V	ctrl + V	VCS operations popup
ctrl + ⌥ + A	⌘ + ⌥ + A	Add to VCS
ctrl + ⌥ + E	⌘ + ⌥ + E	Browse history
ctrl + D	⌘ + D	Show differences

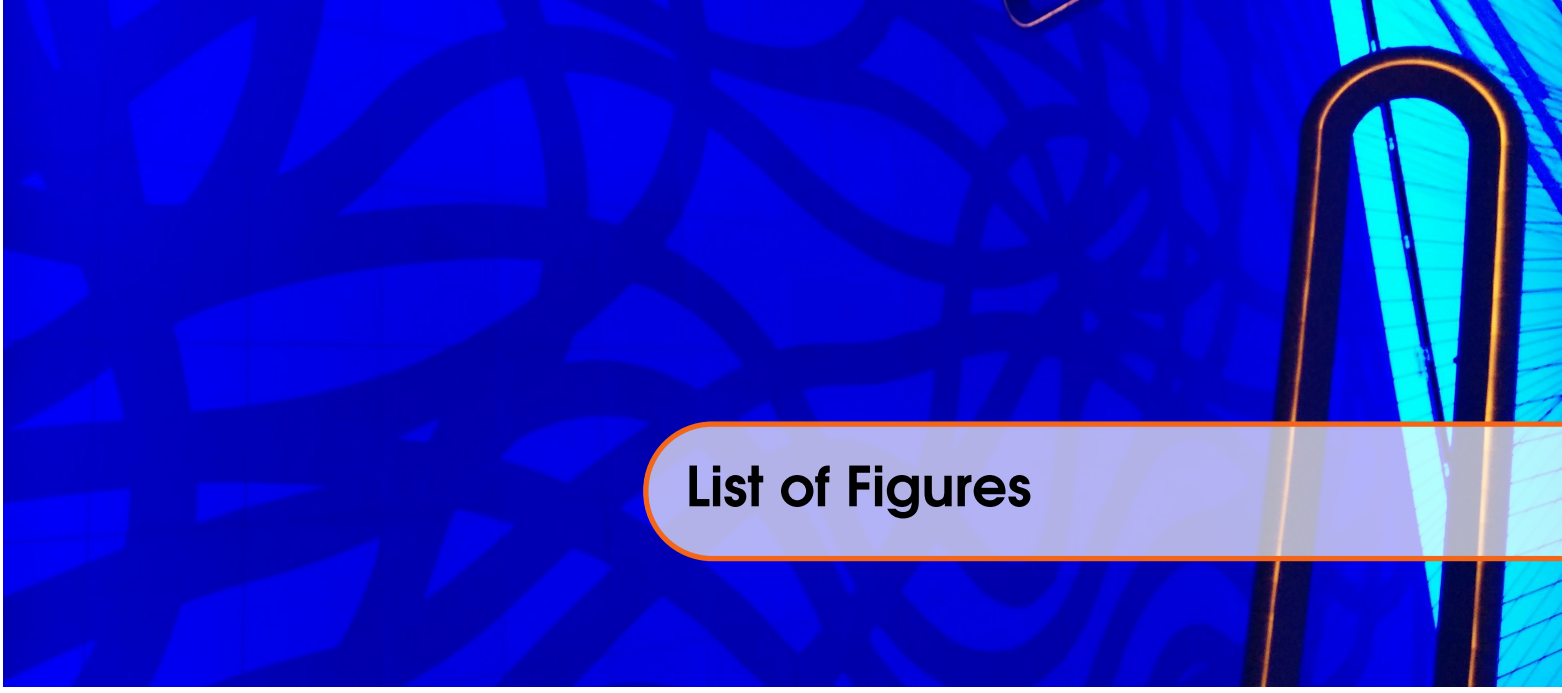
Table 16.10: Version Control System and Local History

Windows or Linux	MacOS	Action
F6	F6	Move
⇧ + F6	⇧ + F6	Rename
⌥ + ⌦	⌥ + ⌦	Safe Delete
ctrl + ⌥ + N	⌘ + ⌥ + N	Inline
ctrl + ⌥ + M	⌘ + ⌥ + M	Extract Method
ctrl + ⌥ + V	⌘ + ⌥ + V	Introduce Variable
ctrl + ⌥ + C	⌘ + ⌥ + C	Introduce constant
ctrl + ⌥ + F	⌘ + ⌥ + F	Introduce field
ctrl + ⌥ + P	⌘ + ⌥ + P	Extract parameter
ctrl + ⌥ + M	⌘ + ⌥ + M	Extract method
ctrl + ⌥ + N	⌘ + ⌥ + N	Inline

Table 16.11: Refactoring

Windows or Linux	MacOS	Action
F8	F8	Step over
F7	F7	Step into
⇧ + F8	⇧ + F8	Step out
F9	F9	Resume
⌥ + F8	⌥ + F8	Evaluate expression
ctrl + F8	⌘ + F8	Toggle breakpoints
ctrl + ⇧ + F8	⌘ + ⇧ + F8	View breakpoints

Table 16.12: Debugger



List of Figures