Orange3 Example Add-on Documentation

Release 2.6.21

Biolab

Apr 04, 2017

Contents

Orange Bioinformatics is an add-on for Orange data mining software package. It extends Orange by providing functionality for some elementary tasks in bioinformatics, like gene set analysis, enrichment, and access to pathway libraries. Included are also widgets for with graphical user interface for, mainly, gene expression analytics.

Orange Bioinformatics provides access to publicly available data, like GEO data sets, Biomart, GO, KEGG, Atlas, ArrayExpress, and PIPAx database. As for the analytics, there is gene selection, quality control, scoring distances between experiments with multiple factors. All features can be combined with powerful visualization, network exploration and data mining techniques from the Orange data mining framework.

CHAPTER 1

Widgets

BioMart

Gives access to **BioMart** databases.

Signals

Inputs:

- None.

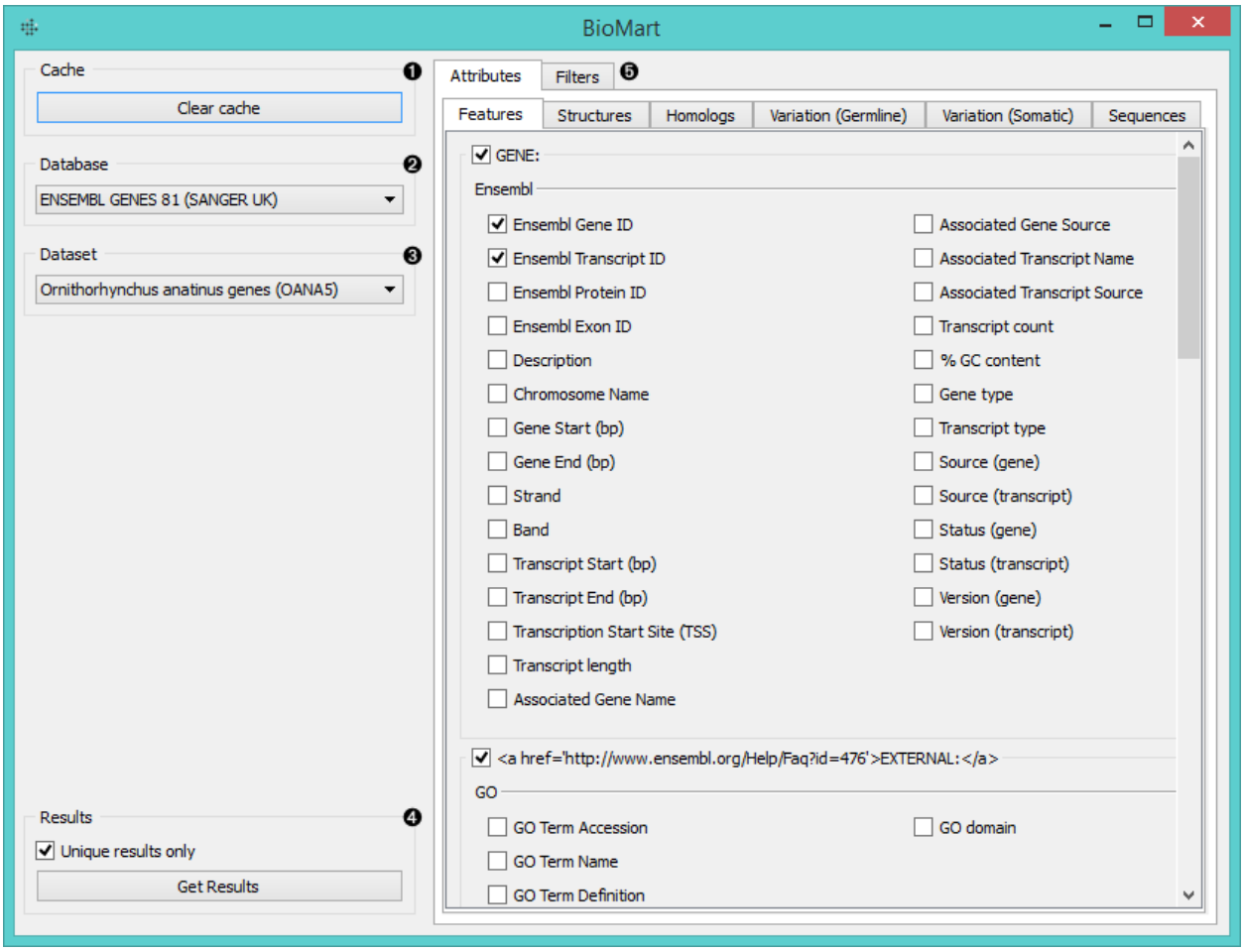
Outputs:

- **Data**

  Data set.

Description

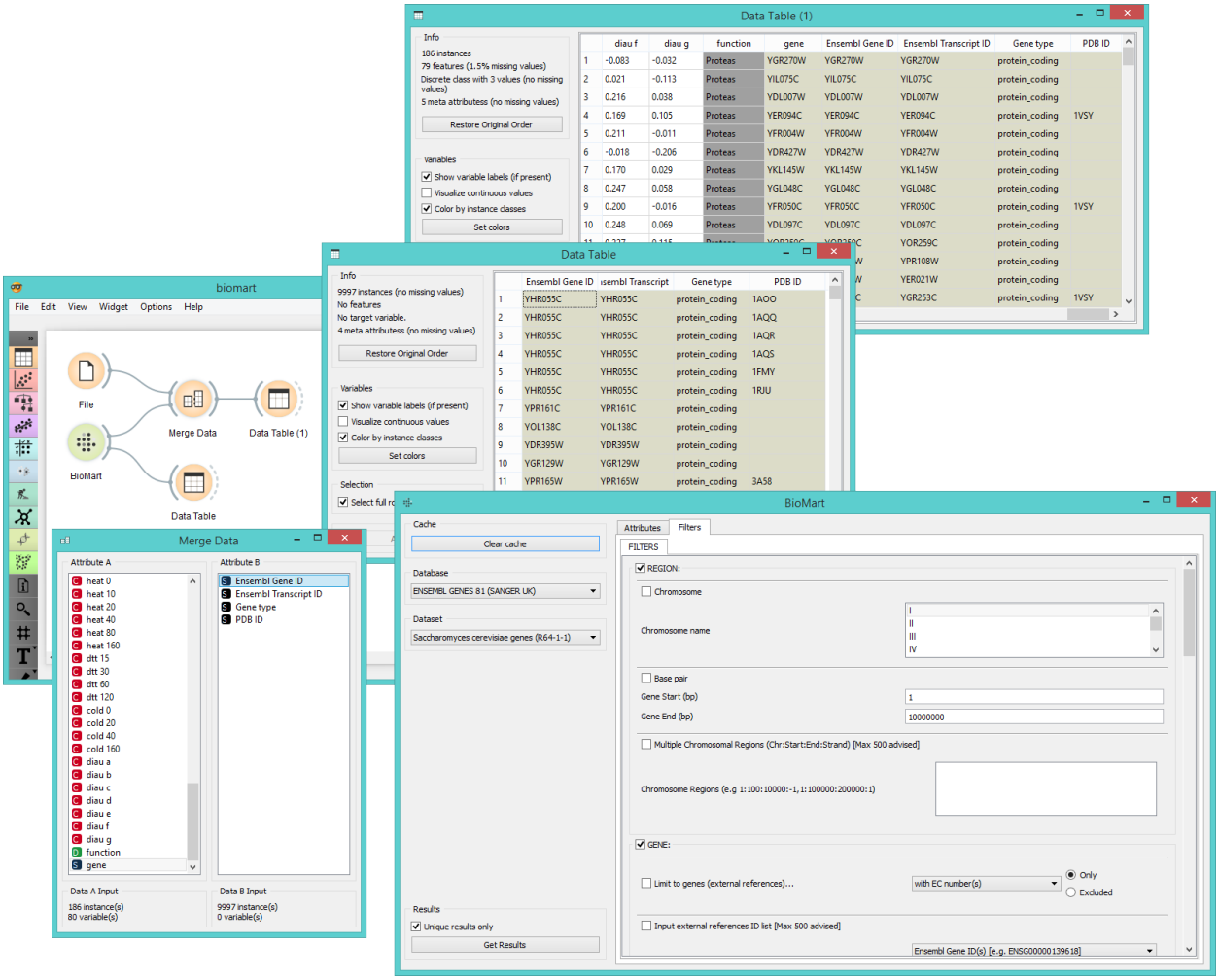
BioMart is a widget for direct access to **BioMart** databases. It sources data from BioMart, filters it by categories (gene, region, phenotype, gene ontology, etc.) and appends selected attributes in the output (IDs, sources, strains, etc.). Read more on the BioMart database library here.



1. Clear cached data.
2. Select the database to source your data from.
3. Select the dataset (organism) to source your genes from.
4. If *Unique results only* is ticked, the widget will prevent data duplication. Click *Get results* to output the data.
5. Set the output:
   - in **Attributes** you set the meta data you wish to output (e.g. IDs, sources, strains...).
   - in **Filter** you filter the data by gene, phenotype, ontology, protein domains, etc.

Example

BioMart is a great widget for appending additional information to your data. We used *brown-selected* data in the **File** widget. Then we selected *Ensembl genes 81 (Sanger UK)* database to source our additional data from. We decided to append *Ensembl Gene ID*, *Ensembl Transcript ID*, *gene type* and *PDB ID*. We also filtered the data to output only those genes that can be found on chromosome I. We got 9997 instances with 4 meta attributes. Then we used **Merge Data** widget to append these metas to our data. We matched the data by gene/Ensemble gene ID and in the end we got a merged data table with 5 meta attributes.



Databases Update

Updates local systems biology databases, like gene ontologies, annotations, gene names, protein interaction networks, and similar.

Signals

Inputs:

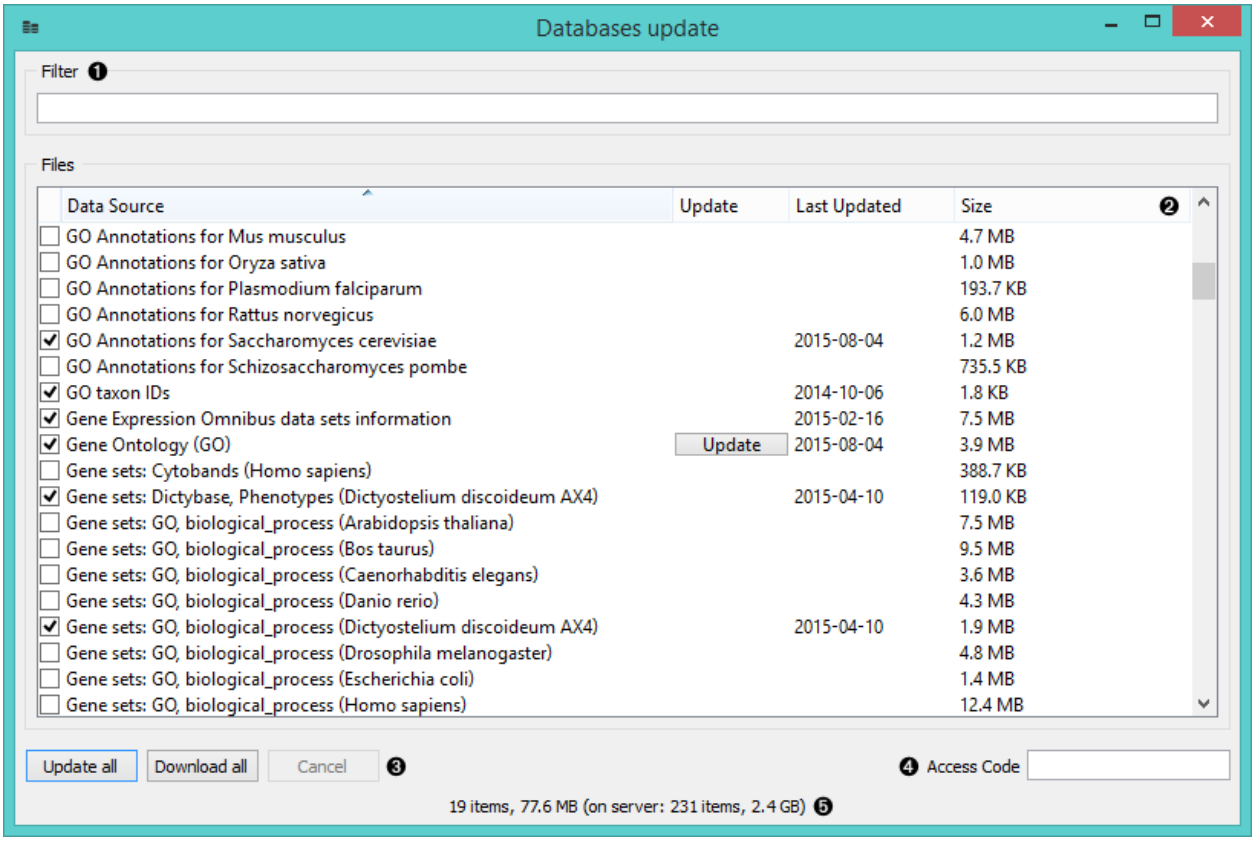
- None

Outputs:

- None

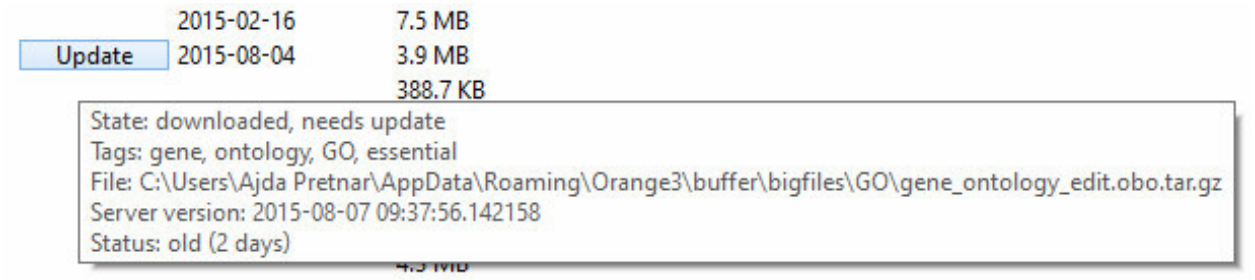
Description

With the bioinformatics add-on you can access several databases directly from Orange. The widget can also be used to update and manage locally stored databases.



1. Find the desired database.
2. A list of available databases described with data source, update availability, date of your last update and file size. A large **Update** button will be displayed next to the database that needs to be updated.
3. **Update All** will update and **Download All** will download all the selected databases. **Cancel** will abort the action.
4. Some data sets require the *Access code*. Type it in the provided field to access the database.
5. Information on the selected databases.

To get a more detailed information on the particular database hover on its name.



Data Profiles

Plots gene expression levels by attribute in a graph.

Signals

Inputs:

- **Data**

  Data set.

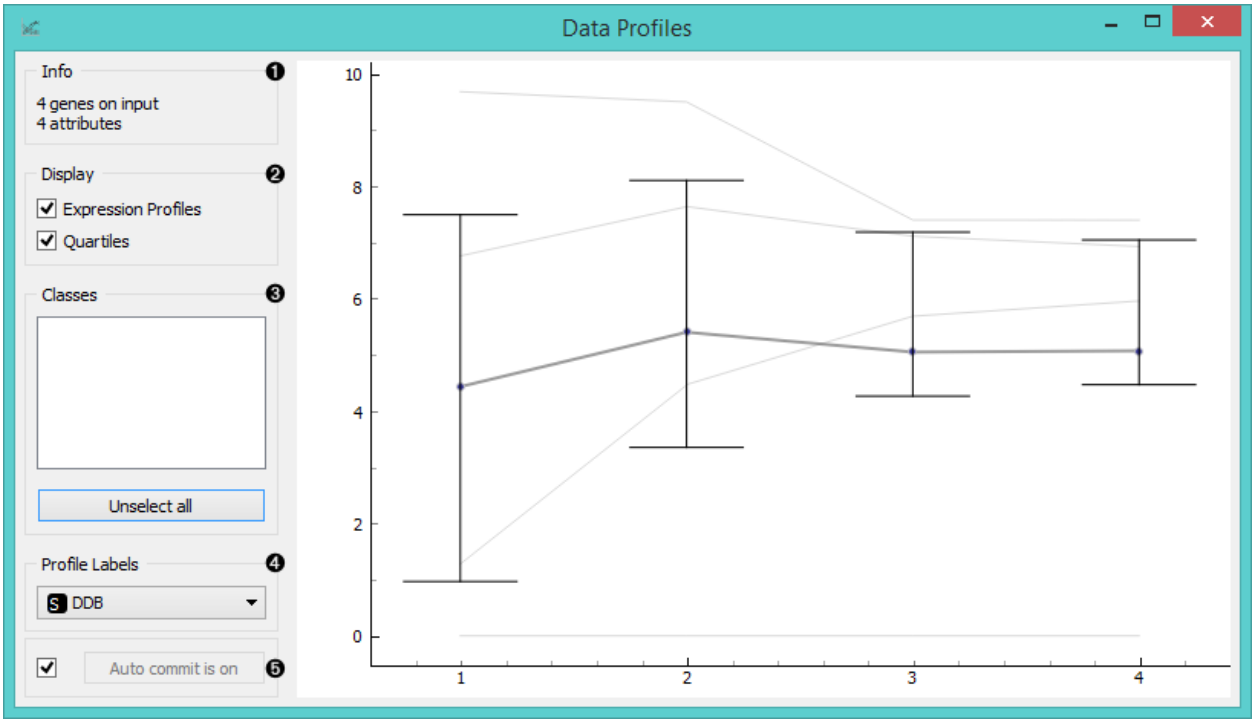
Outputs:

- **Selected Data**

  Instances that the user has manually selected from the plot.

Description

Data Profiles plots gene expression levels for each attribute in a graph. The default graph displays the mean expression level for the input data set. The x-axis represents attributes and the y-axis gene expression values. By hovering over the line you can see which gene it represents and by click on the line you will select the gene and output it.



1. Information on the input data.
2. Select display options:

- **Expression Profiles** will display expression levels for individual data instances.
- **Quartiles** will show quartile cut-off points.

3. If the data has classes, you can select which class to display by clicking on it. Such data will also be colored by class. *Unselect All* will show an empty plot, while *Select All* will diplay all data instances by class.
4. Select which attribute you wish to use as a profile label.
5. If *Auto commit is on*, the widget will automatically apply changes to the output. Alternatively click *Commit*.

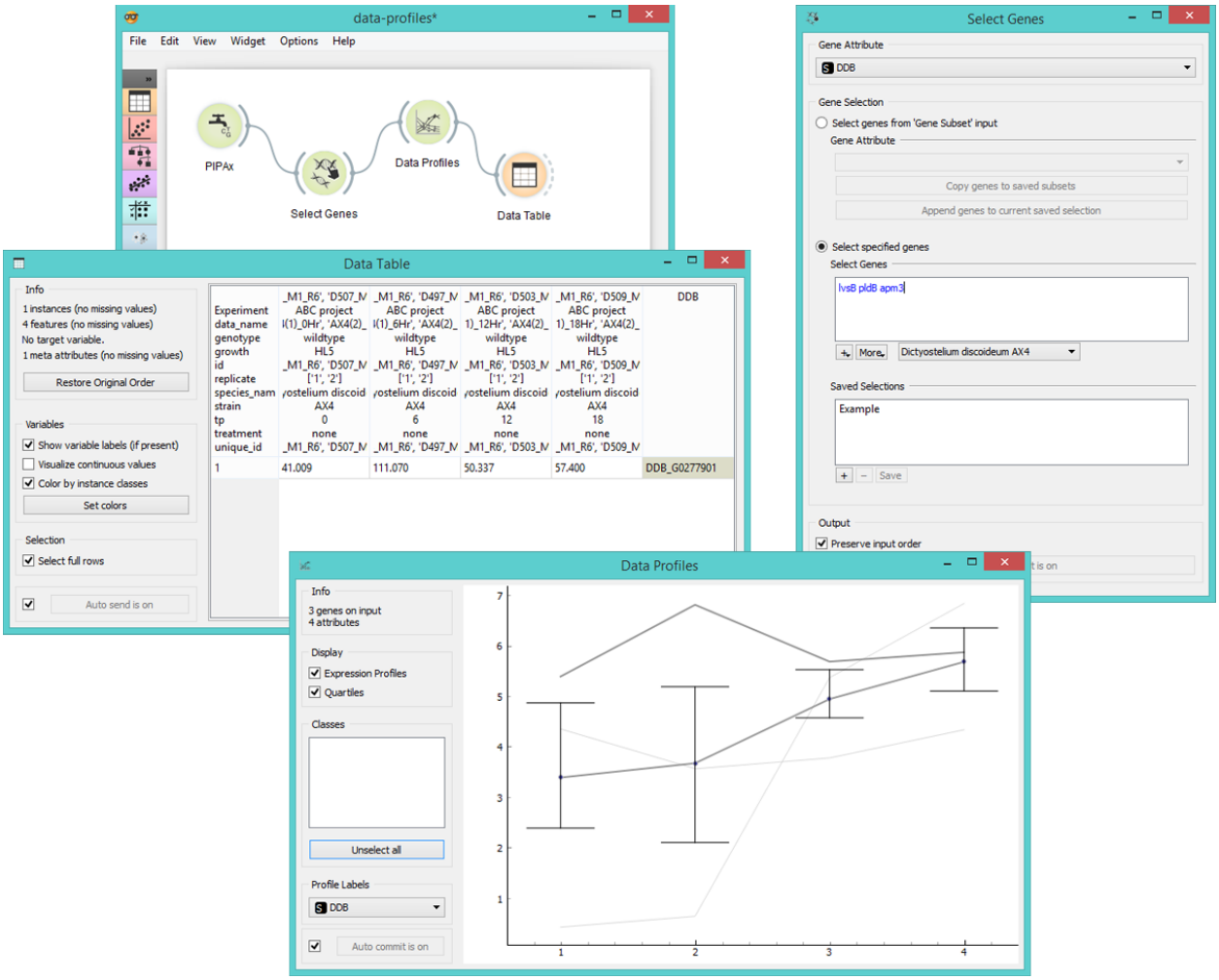
Example

Data Profiles is a great widget for visualizing significant gene expression levels, especially if the data has been sourced at different timepoints. This allows the user to see differences in expression levels in time for each instance in the data set and the overall mean.

Below we used the **PIPAx** widget, where we selected 8 *AX4 Dictyostelium* experiments, all having been sourced at diffferent timepoints and belonging to one of the two replicates. We decided to average replicates (to get one instance for both replicates) and to apply logarithmic transformation to adjust expression levels.

In **Select Genes** we decided to observe only the three genes from the data set that are a part of the *increased exocytosis* process (lsvB, pldB, amp3), which we selected in the *Import gene set names* option. This allows us to specify which biological process we're interested in and to observe only the specified genes.

Then we observe expression levels in **Data Profiles** widget, where we see all three *Expression Profiles* plotted, together with *Quartiles* and mean expression level. Finally, we selected the gene with the highest overall expression level and output it to **Data Table**.



dictyExpress

Gives access to **dictyExpress** databases.

Signals

Inputs:

- (None)

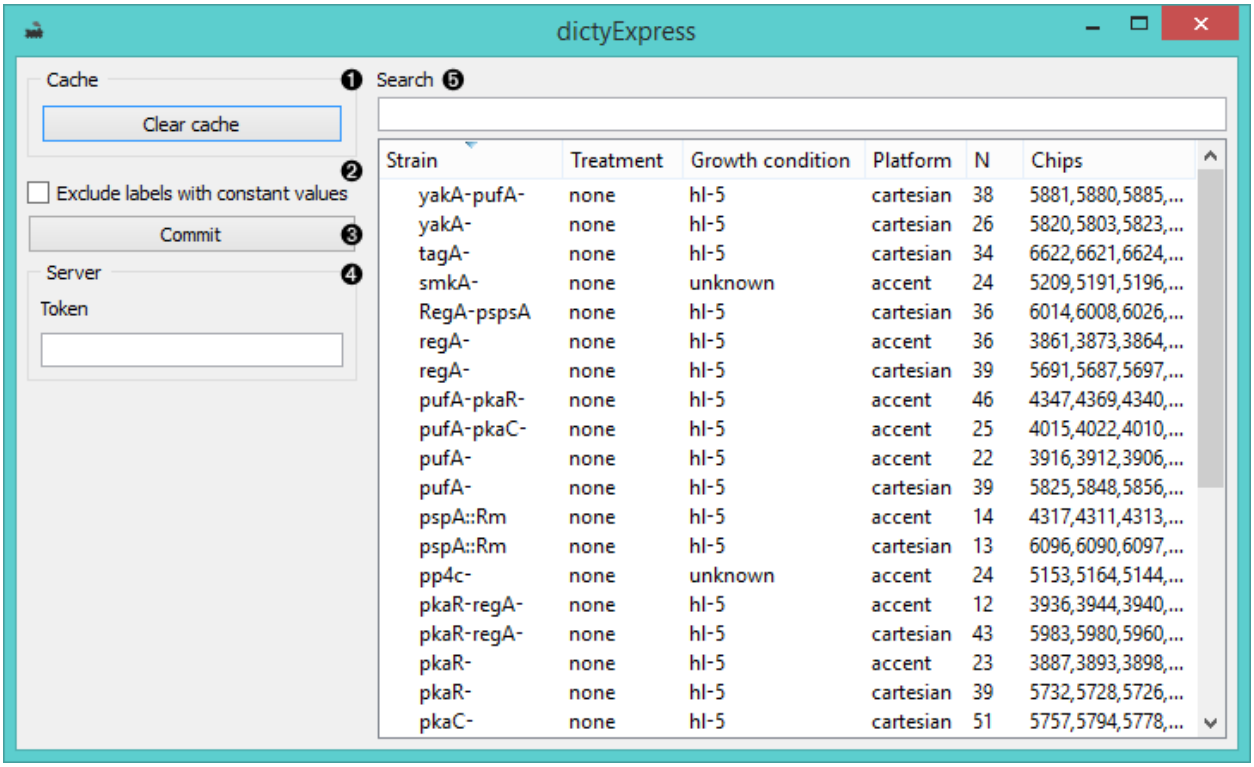
Outputs:

- **Data**

  Selected experiments. Each annotated column contains results of a single experiment or, if the corresponding option is chosen, the average of multiple replicates.

Description

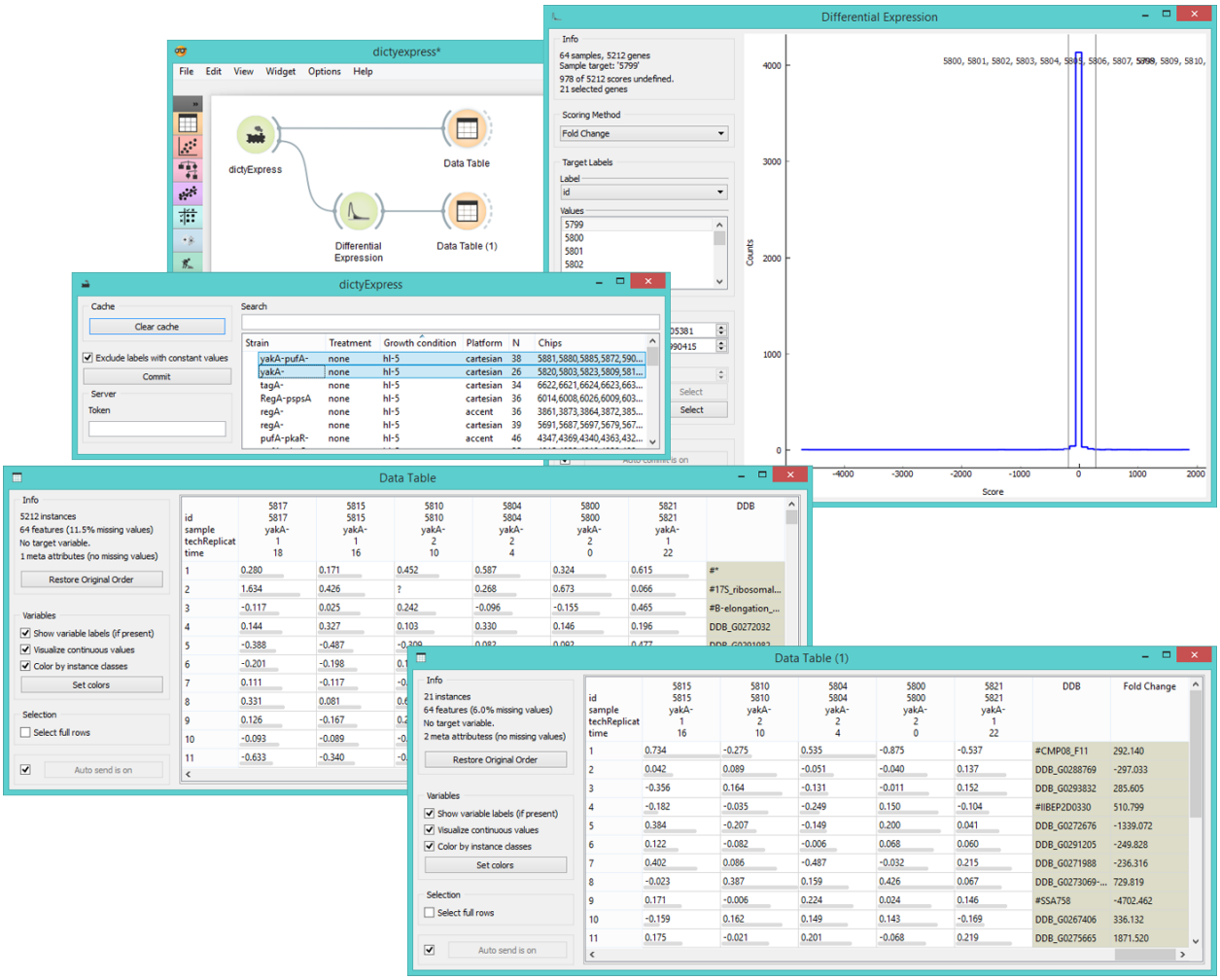
dictyExpress is a widget for a direct access to **dictyExpress** database and it is very similar to the **GenExpress** and **GEO Data Sets** widgets as it allows you to dowload selected experiments.



1. The widget will automatically save (cache) downloaded data, which makes them available also in the offline mode. To reset the widget click *Clear cache*.
2. *Exclude labels with constant values* removes labels that are the same for all the selected experiments in the output.
3. Click *Commit* to output the data.
4. Publicly available data are accessible from the outset. Use *Token* to access password protected data.
5. Available experiments can be filtered with the *Search* box at the top.

Example

In the schema below we connected **ditcyExpress** to a **Data Table** to observe all of the selected experiments. Then we used **Differential Expression** widget to select the most relevant genes and output them to another **Data Table**.



Differential Expression

Plots differential gene expression for selected experiments.

Signals

Inputs:

- **Data**

  Data set.

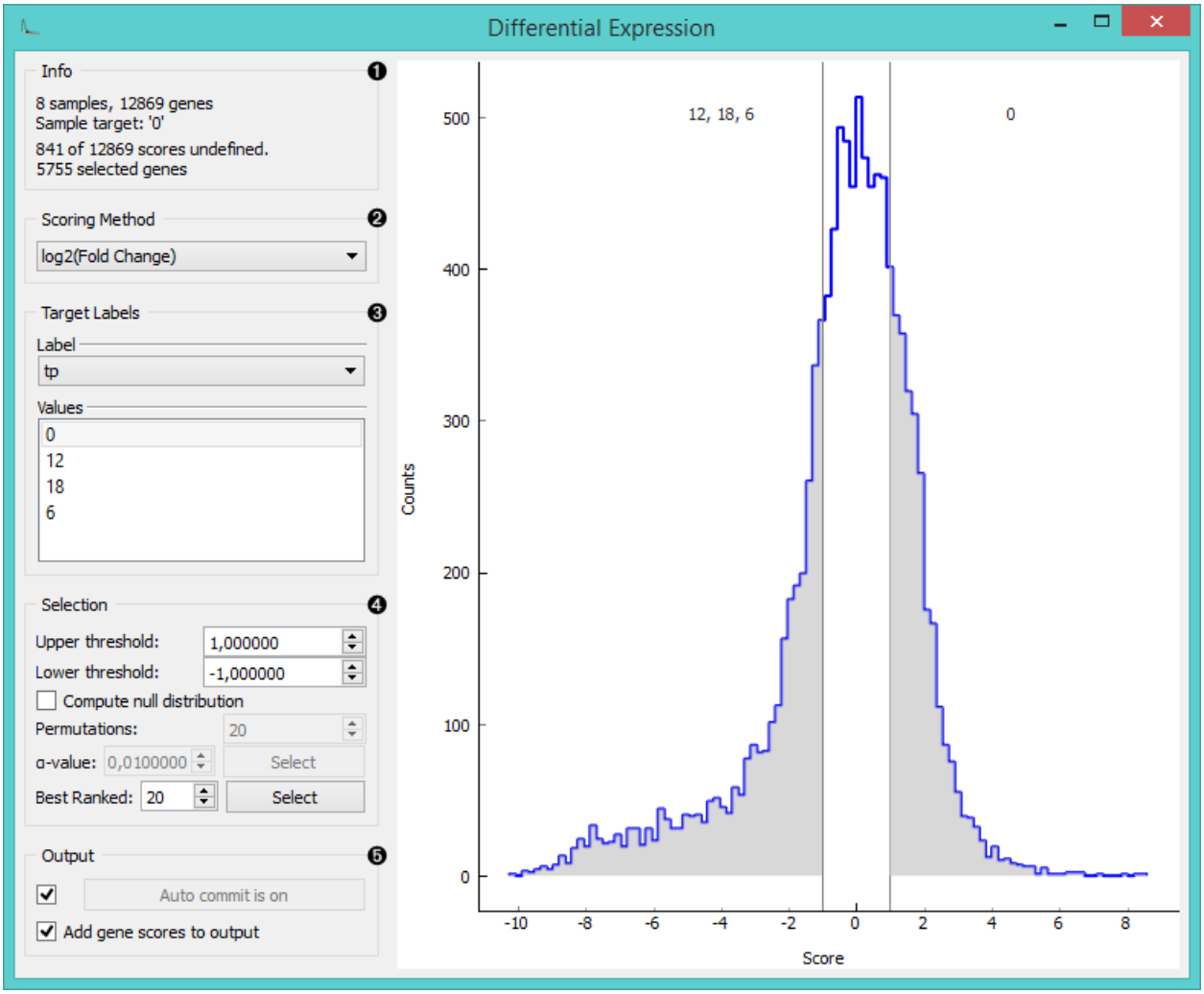
Outputs:

- **Selected data**

  Data subset.

Description

This widget plots a differential gene expression graph for a sample target. It takes gene expression data as an input (from **dictyExpress**, **PIPAx**, etc.) and outputs a selected data subset (normally the most interesting genes).



1. Information of the data input and output. The first line shows the number of samples and genes in the data set. The second line displays the selected sample target (read around which the graph is plotted). The third line shows the number of undefined gene (missing data) and the fourth the number of genes in the output.
2. Select the plotting method in *Scoring method*:
   - **Fold change**: final to initial value ratio
   - **log2 (fold change)**: binary logarithmic transformation of fold change values
   - **T-test**: parametric test of null hypothesis
   - **T-test (P-value)**: parametric test of null hypothesis with P-value as criterium
   - **ANOVA**: variance distribution
   - **ANOVA (P-value)**: variance distribution with P-value as criterium
   - **Signal to Noise Ratio**: biological signal to noise ratio
   - **Mann-Whitney**: non-parametric test of null hypothesis with P-value as criterium

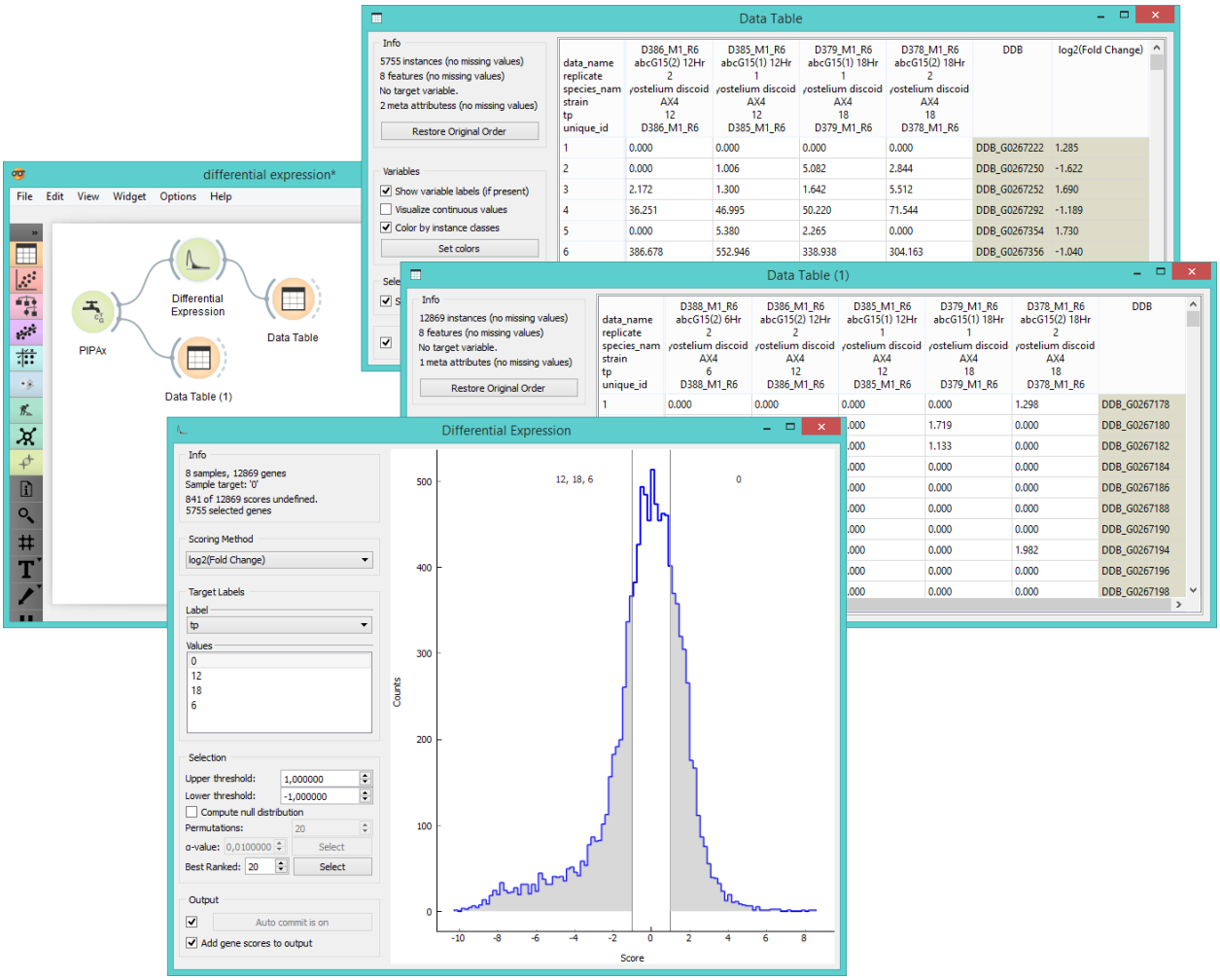
3. Select *Target Labels*. Labels depend on the attributes in the input. In *Values* you can change the sample target (default value is the first value on the list, alphabetically or numerically).
4. *Selection* box controls the output data.
    - By setting the *Lower threshold* and *Upper threshold* values you are outputting the data outside this interval (the most interesting expression levels). You can also manually place the threshold lines by dragging left or right in the plot.
    - If you click *Compute null distribution* box, the widget will calculate null distribution and display it in the plot. *Permutations* field allows you to set the precision of null distribution (the more permutations the more precise the distribution), while α *-value* will be the allowed probability of false positives. Press *Select* to output this data.

- The final option is to set the number of best ranked genes and output them with *Select*.

1. When *Auto commit is on* is ticked, the widget will automatically apply the changes. Alternatively press *Commit*. If the *Add gene scores to output* is ticked, the widget will append an additional column with gene scores to the data.

Example

In the example below we chose two experiments from the **PIPAx** widget (8 experiments measuring gene expression levels on *Dictyostelium discoideum* at different timepoints) and observed them in the **Data Table**. Then we used the **Differential Expression** widget to select the most interesting genes. We left upper and lower threshold at default (1 and -1) and output the data. Then we observed the selected data in another **Data Table**. As we have ticked the *Add gene scores to output*, the table shows an additional column with gene scores as instances.



Expression Profile Distances

Computes distances between gene expression levels.

Signals

Inputs:

- **Data**

  Data set.

Outputs:

- **Distances**

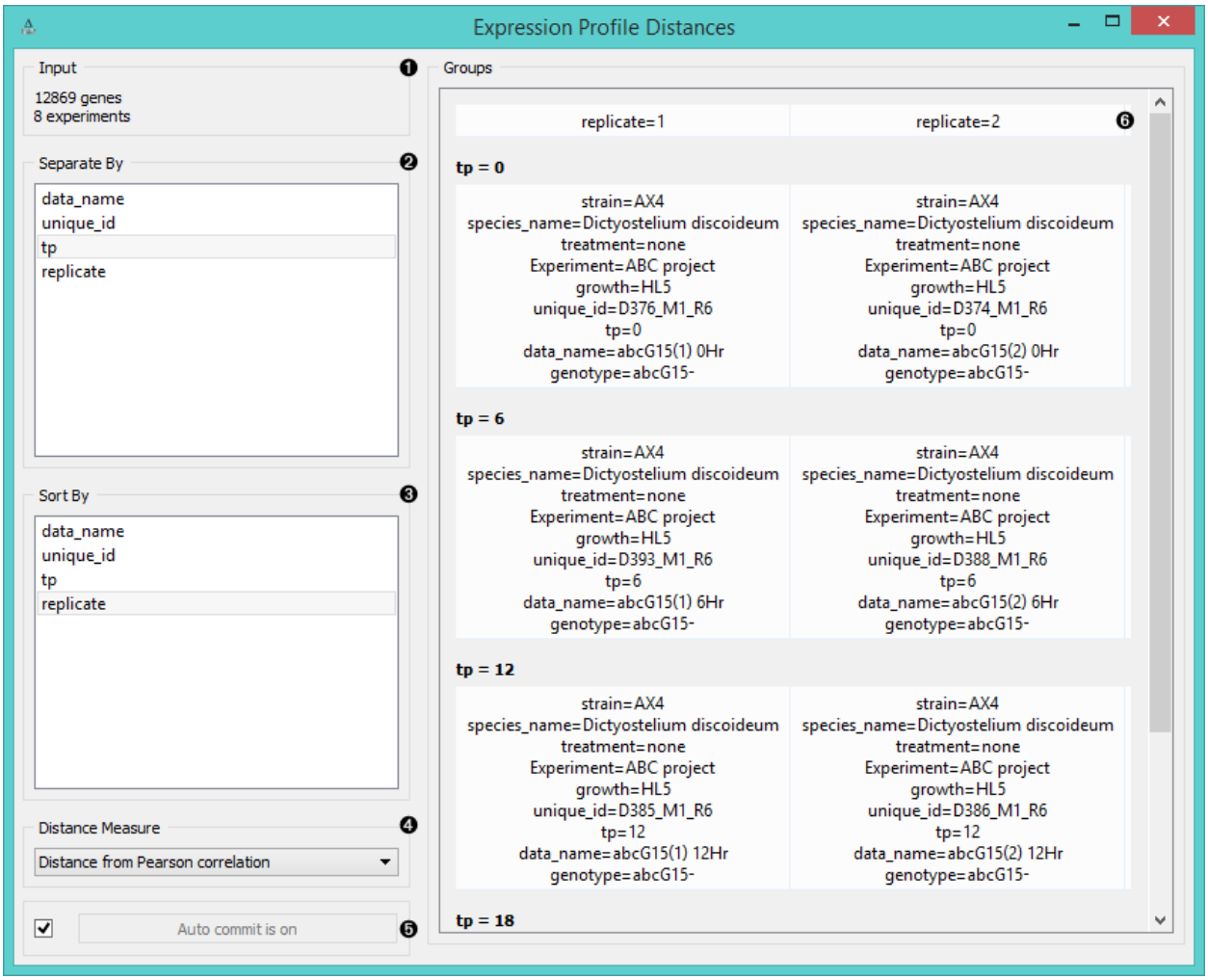
  Distance matrix.

- **Sorted Data**

  Data with groups as attributes.

Description

Widget **Expression Profile Distances** computes distances between expression levels among groups of data. Groups are data clusters set by the user through *separate by* function in the widget. Data can be separated by one or more variable labels (usually timepoint, replicates, IDs, etc.). Widget outputs distance matrix that can be fed into **Distance Map** and **Hierarchical Clustering** widgets.

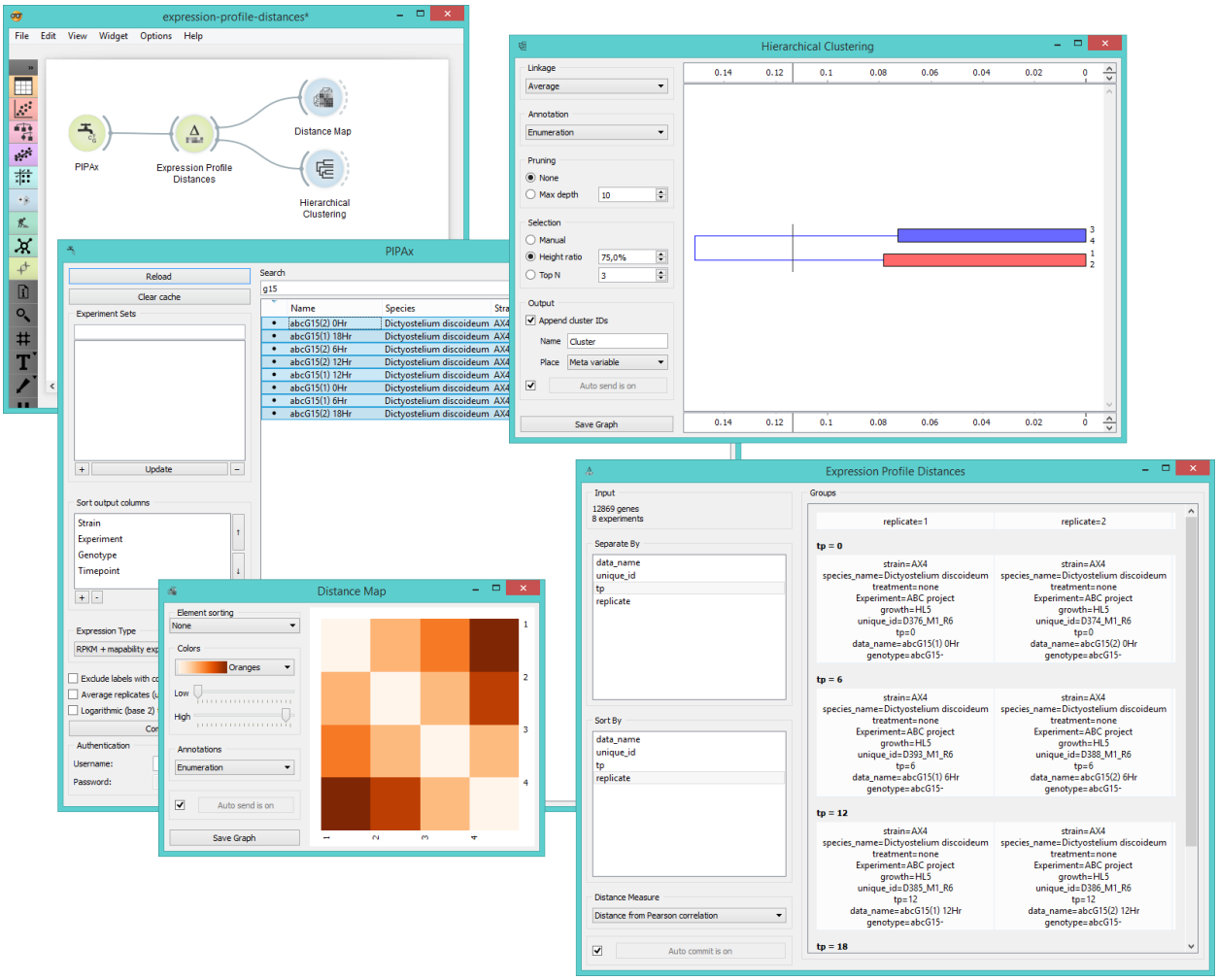


1. Information on the input data.
2. Separate the experiments into groups by labels (normally timepoint, replicates, data name, etc.).
3. Sort the experiments inside the group by labels.
4. Choose the *Distance Measure*:
   - **Pearson** (linear correlation between the values, remapped as a distance in a [0, 1] interval)
   - **Euclidean** ("straight line", distance between two points)
   - **Spearman** (linear correlation between the rank of the values, remapped as a distance in a [0, 1] interval)

5. If *Auto commit is on*, the widget will automatically compute the distances and output them. Alternatively click *Commit*.
6. This snapshot shows 4 groups of experiments (tp=0, tp=6, tp=12, tp=18) with 2 experiments (replicates) in each group.

Example

Expression Profile Distances widget is used to calculate distances between gene expression values sorted by labels. We chose 8 experiments measuring gene expression levels on *Dictyostelium discoideum* at different timepoints. In the **Expression Profile Distances** widget we separated the data by timepoint and sorted them by replicates. We could see the grouping immediately in the *Groups* box on the right. Then we fed the results to **Distance Map** and **Hierarchical Clustering** to visualize the distances and cluster the attributes.



Gene Info

Displays information on the genes in the input.

Signals

Inputs:

- **Data**

  Data set.

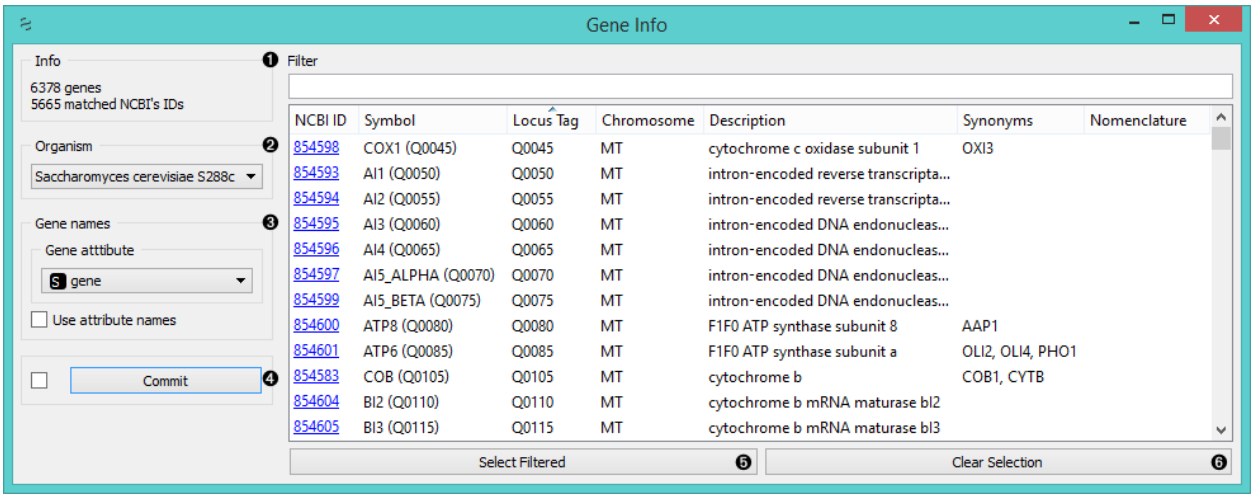
Outputs:

- **Selected Data**

  Instances with meta data that the user has manually selected in the widget.

Description

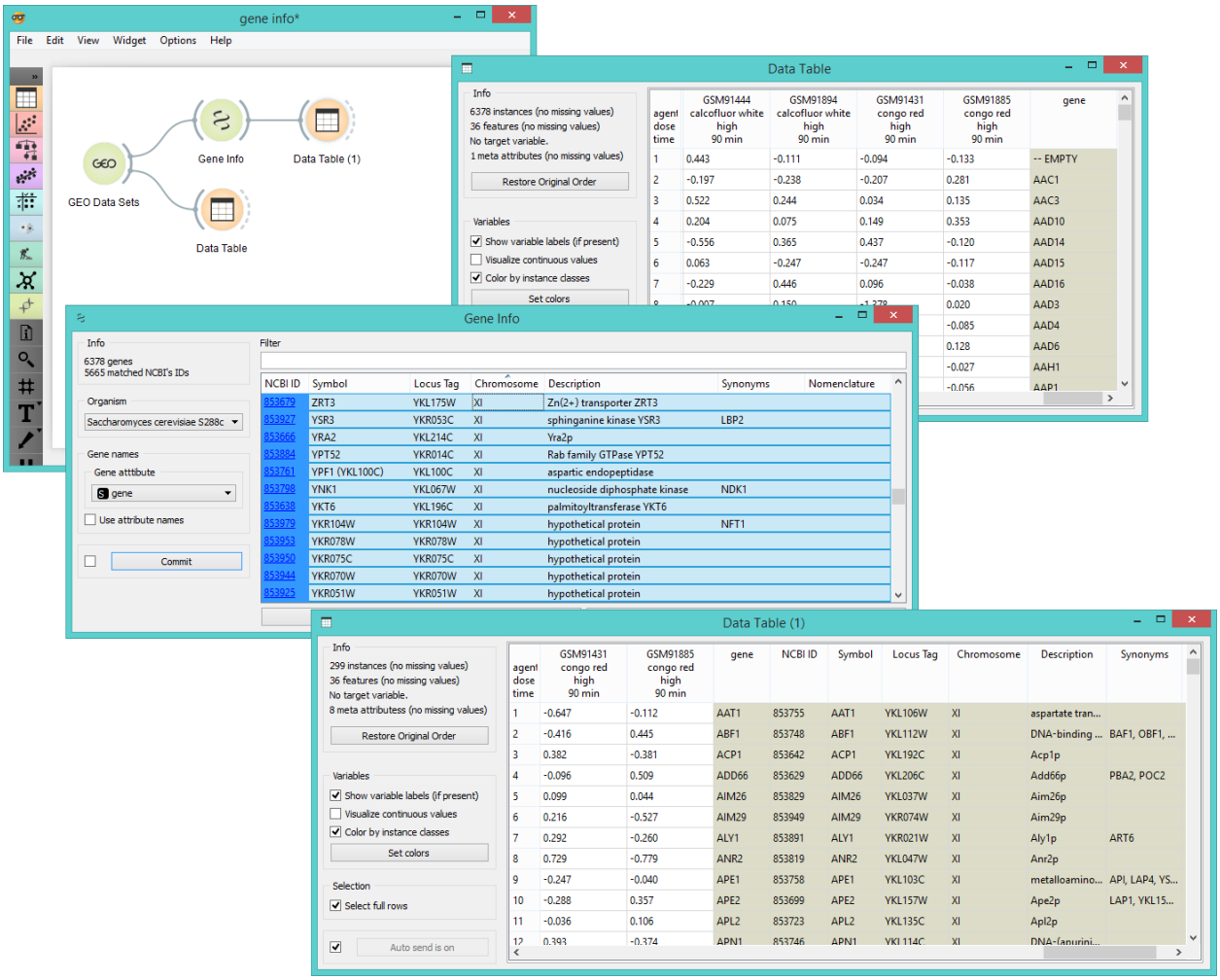
A useful widget that presents information on the genes from the NCBI database. You can also select a subset and feed it to other widgets. By clicking on the gene NCBI ID in the list, you will be taken to the NCBI site with the information on the gene.



1. Information on data set size and genes that matched the NCBI ID's.
2. Select the organism of reference.
3. Set the source of gene names. If your gene names are placed as attributes names, select *Use attribute names*.
4. If *Auto commit is on*, changes will be communicated automatically. Alternatively click *Commit*.
5. In the row above the list you can filter the genes by search word(s). If you wish to output the filtered data, click *Select Filtered*.
6. If you wish to start from scratch, click *Clear Selection*.

Example

Below we first view the entire *Caffeine effect: time course and dose response* data set in the *Data Table* widget. Then we feed the same data into the *Gene Info*, where we select only the genes that are located on the 11th chromosome. We can observe these data in another *Data Table*, where additional information on the selected genes are appended as meta attributes.



GenExpress

Gives access to **GenExpress** databases.

Signals

Inputs:

- (None)

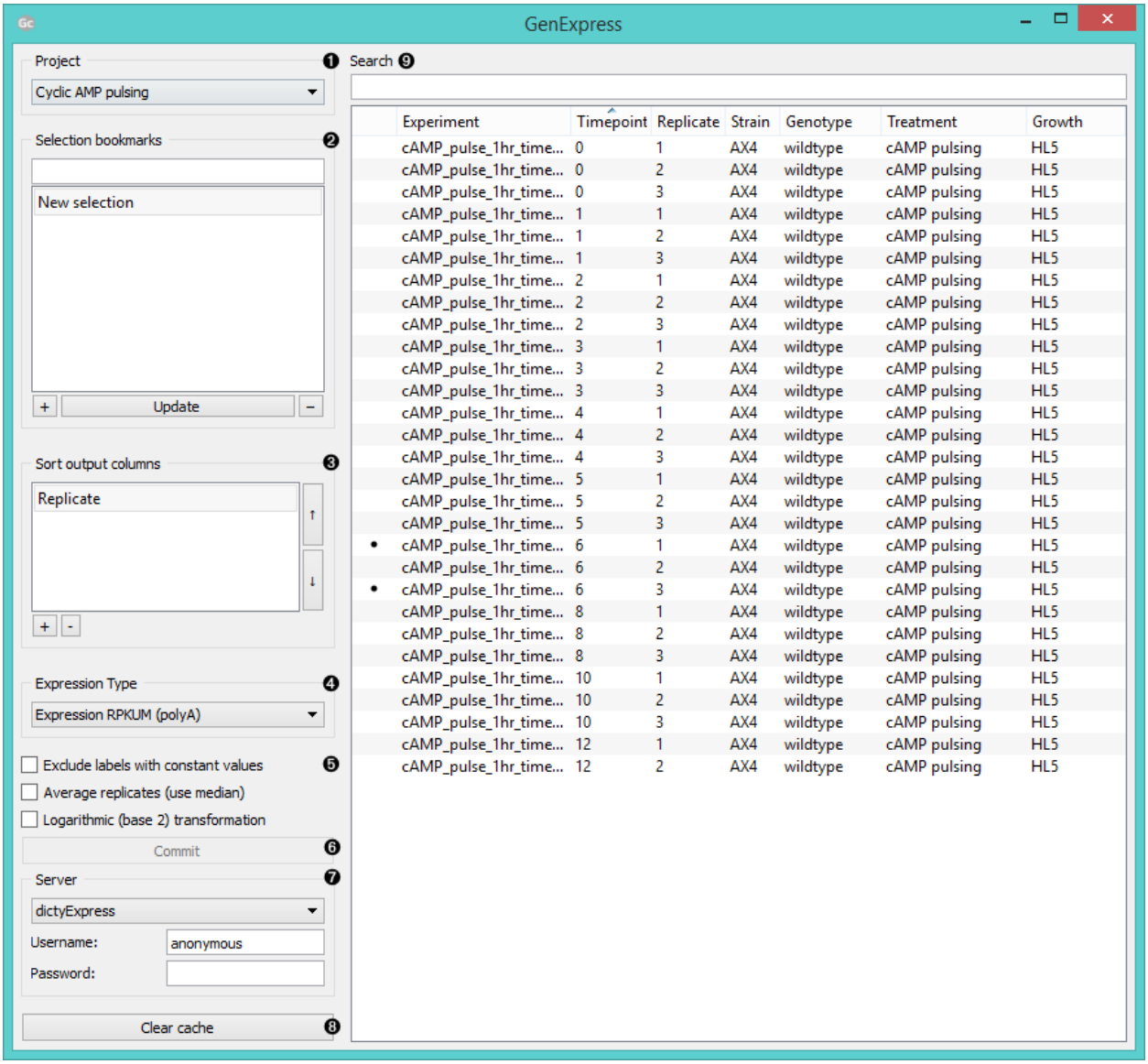
Outputs:

- **Data**

  Selected experiments. Each annotated column contains results of a single experiment or, if the corresponding option is chosen, the average of multiple replicates.

Description

GenExpress is a widget for a direct access to **GenExpress** database. It is very similar to the **PIPAx** and **GEO Data Sets** widgets as it allows you to download the data from selected experiments.



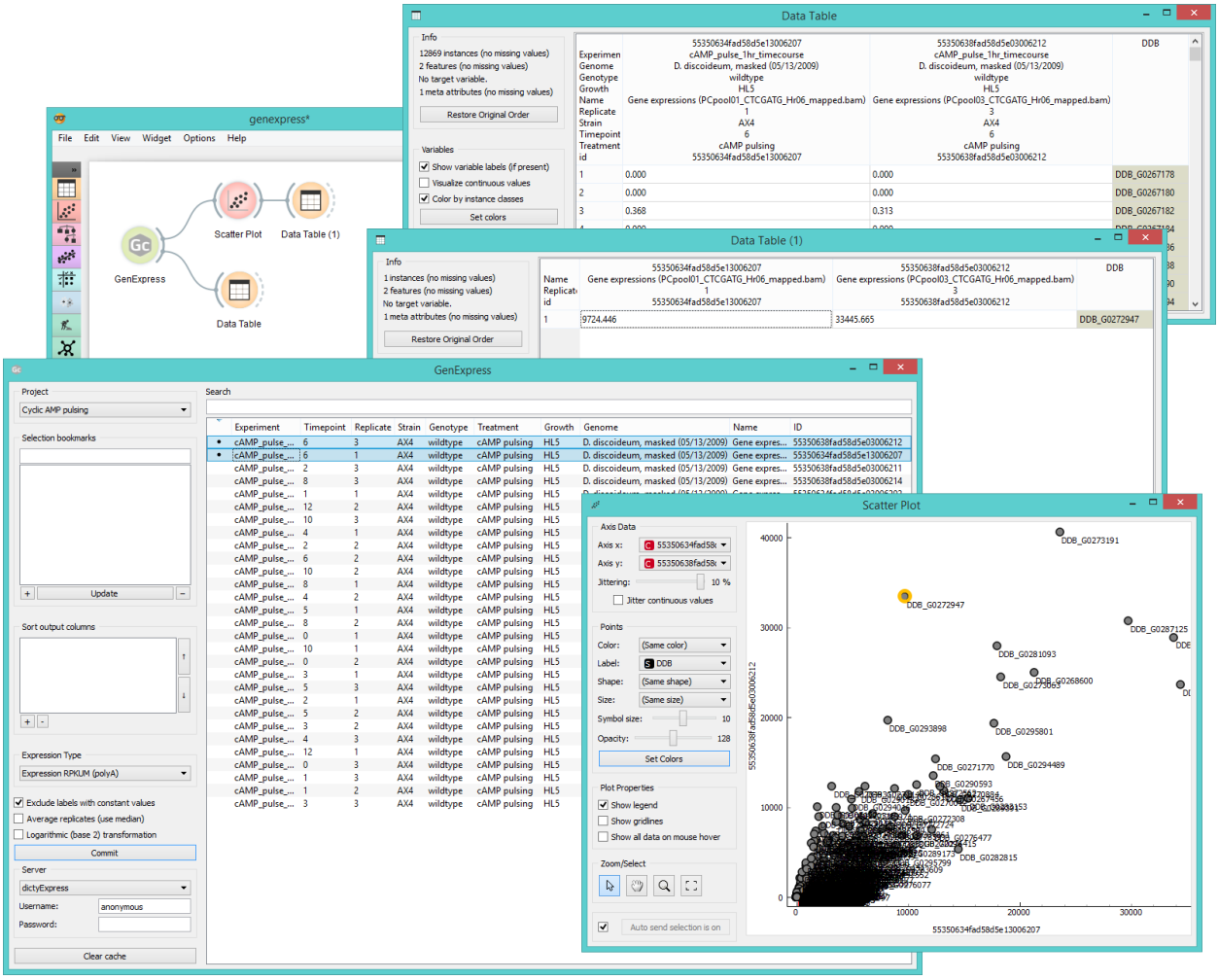
1. Choose a projects to source your data from.
2. Use *Selection bookmarks* to save a selection: select experiments, click the "**+**" button and name the set. To add experiments to your set, click on the set name, select additional experiments and click *Update*. To remove the set click "**-**".
3. In *Sort output columns* set the attributes by which the output columns are sorted. Add attributes with a "+" button and remove them with "-". Switch the sorting order with arrows on the right.
4. Set the expression type for your output data.
   - **Expression RPKM** outputs data in *reads per kilobase of transcript per million mapped reads*
   - **Expression RPKUM** outputs only RPKUM data.

- **Read counts (raw)** outputs raw read count data. The polyA variants use only polyA (mRNA) mapped hits.

5. **Exclude labels with constant values** removes labels that are the same for all selected experiments. **Average replicates (use median)** averages identical experiments by using medians as values. **Logarithmic (base 2) transformation** returns log2(value+1) for each value.
6. Click *Commit* to output selected data.
7. Select the server you wish to access the data from. Log in to access private data.
8. *Clear cache* removes the uploaded data sets from internal memory.
9. Experiments can be filtered with the *Search* box. To select which attributes to display right-click on the header. To select multiple experiments click them while holding the *Control/Command* key.

Example

In the schema below we connected **GenExpress** to **Data Table** to view the gene expression reads and then to **Scatter Plot**, where we chose to view expression levels from two experiments. In the plot we select an outlier and view it in another **Data Table**.



GEO Data Sets

Provides access to data sets from gene expression omnibus (GEO DataSets).

Signals

Inputs:

- (None)

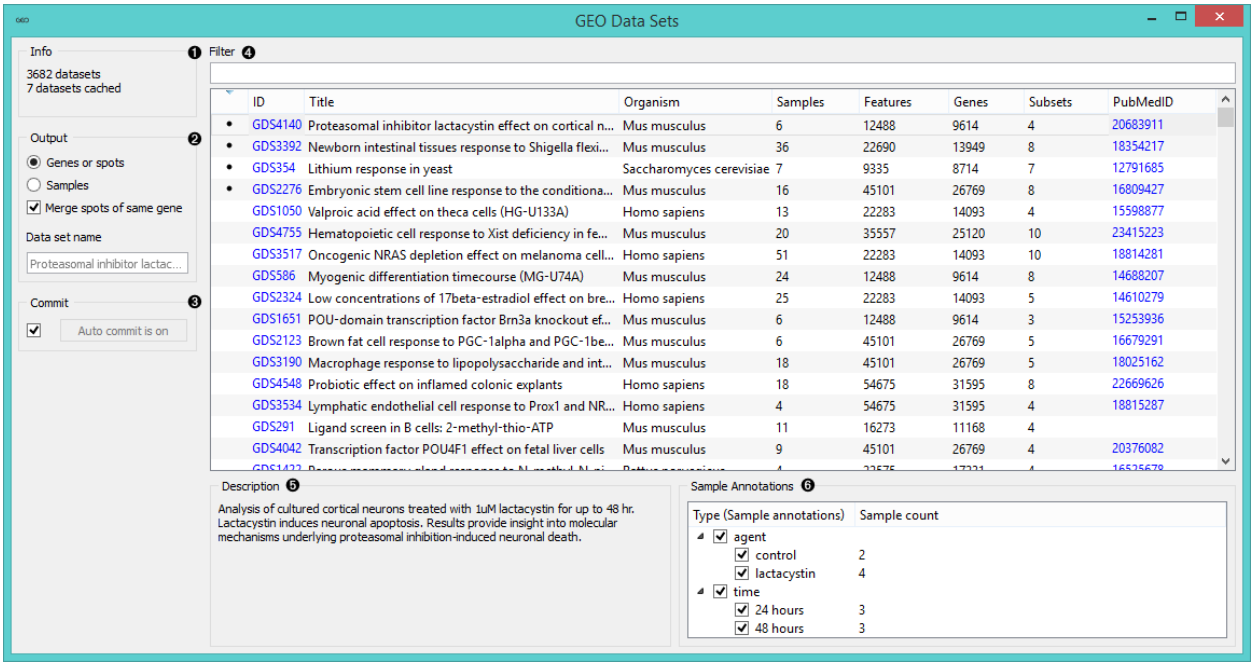
Outputs:

- **Data**

Data set selected in the widget with genes or samples in rows.

Description

GEO DataSets is a data base of gene expression curated profiles maintained by NCBI and included in the Gene Expression Omnibus. This Orange widget provides access to all its data sets and outputs a data set selected for further processing. For convenience, each dowloaded data set is stored locally.

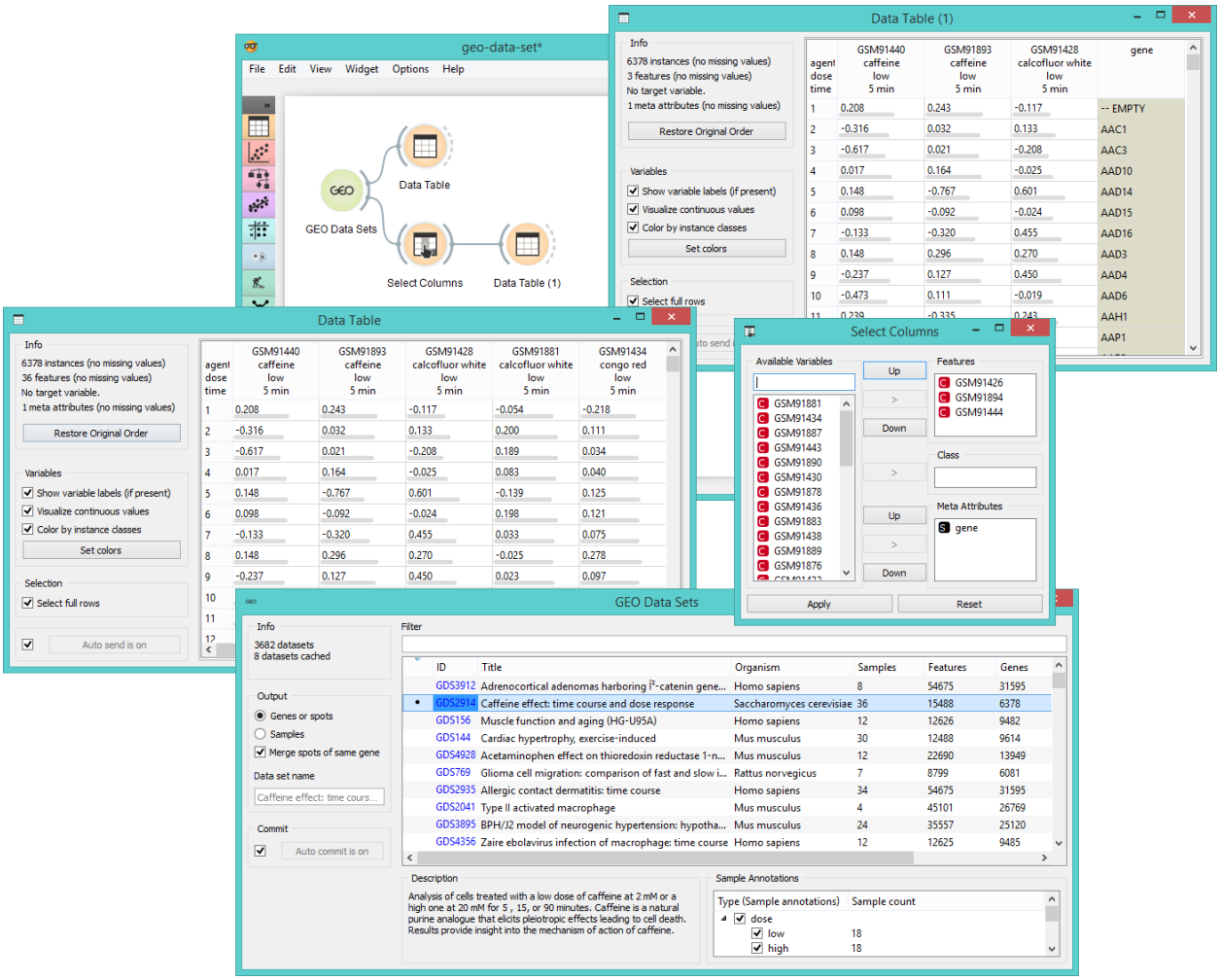


1. Information on the GEO data set collection. Cached data sets are the ones currently stored on the computer.
2. Output features. If *Genes or spots* is selected, genes (or spots) will be used as attributes. Alternatively samples will be used as attributes. *Merge spots of same gene* averages measures of the same gene. Finally, in the *Data set name* you can rename the output data. GEO title will be used as a default name.
3. If *Auto commit is on*, then the selected data set will be automatically communicated to other widgets. Alternatively, click *Commit*.

4. *Filter* allows you to search for the data set. Below you see a list of GEO data sets with an ID number (link to the NCBI Data Set Browser), title, organism used in the experiment, number of samples, features, genes, subsets and a reference number for the PubMed journal (link to the article abstract).
5. Short description of the experiment from which the data set is sourced.
6. Select which *Sample Annotations* will be used in the output.

Example

GEO Data Sets is similar to the **File** widget. In the example below we selected *Caffeine effect: time dose and response* data set from the GEO data base and used *Genes or spots* as attributes. We inspected the data in *Data Table*. Then we selected 3 genes in the **Select Columns** widget for a detailed analysis in another data table.



GO Browser

Provides access to Gene Ontology database.

Signals

Inputs:

- **Cluster Data**

  Data on clustered genes.

- **Reference Data**

  Data with genes for the reference set (optional).

Outputs:

- **Data on Selected Genes**

  Data on genes from the selected GO node.

- **Data on Unselected Genes**

  Data on genes from GO nodes that weren't selected.

- **Data on Unknown Genes**

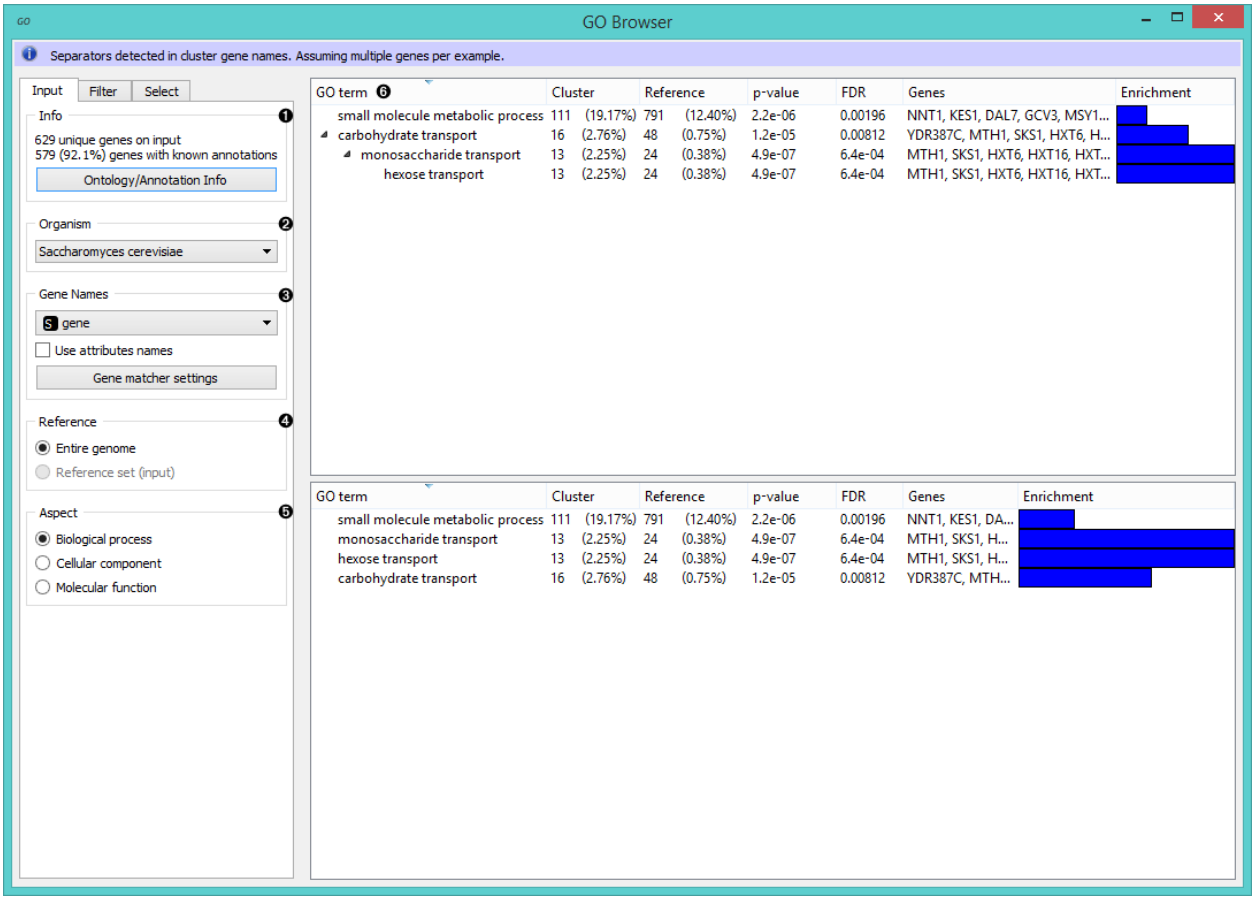
  Data on genes that are not in the GO database.

- **Enrichment Report**

  Data on GO enrichment analysis.

Description

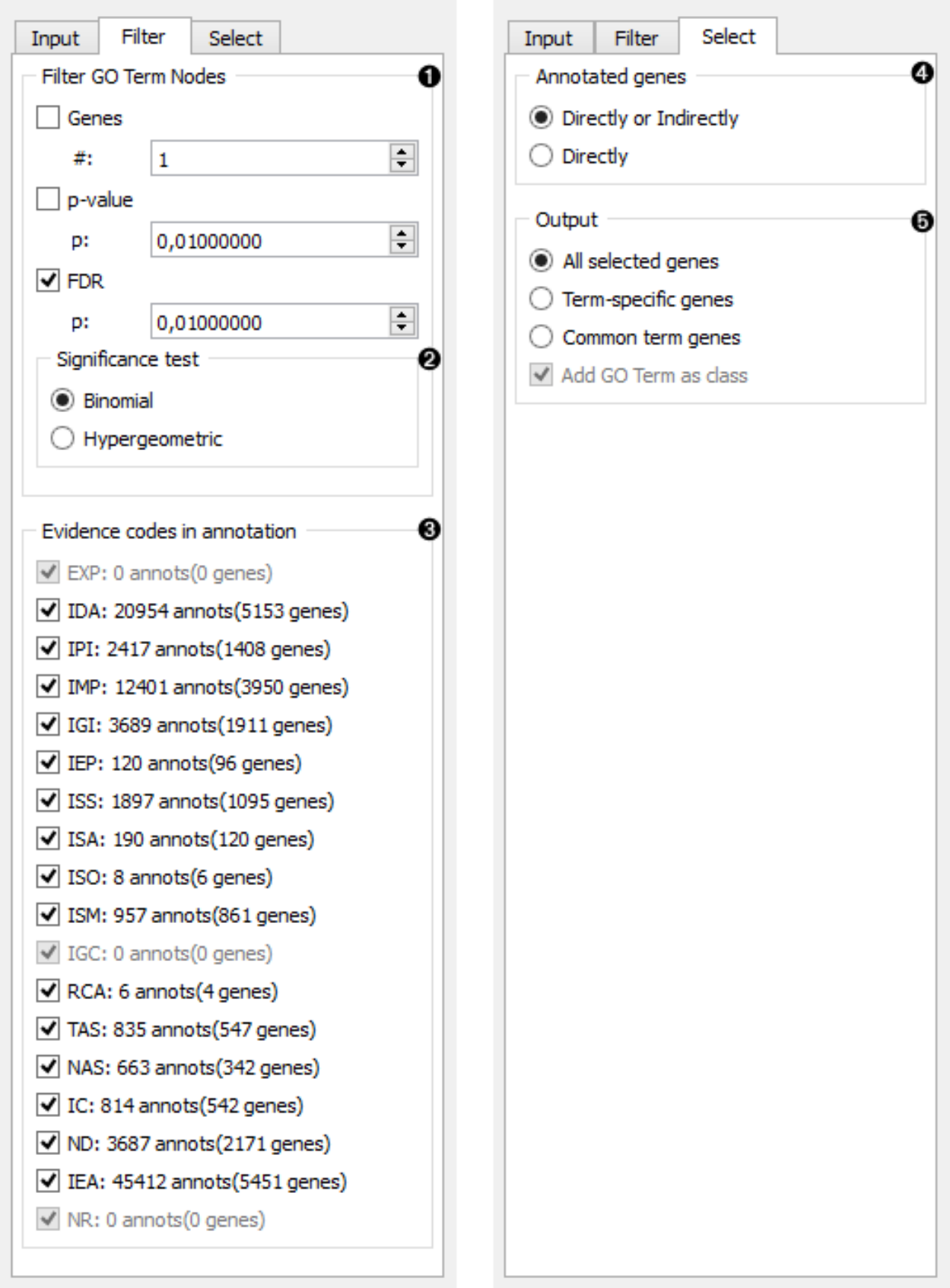
GO Browser widget provides access to *Gene Ontology database*. Gene Ontology (GO) classifies genes and gene products to terms organized in a graph structure called an ontology. The widget takes any data on genes as an input (it is best to input statistically significant genes, for example from the output of the **Differential Expression** widget) and shows a ranked list of GO terms with p-values. This is a great tool for finding biological processes that are over- or under-represented in a particular gene set. The user can filter input data by selecting terms in a list.



INPUT tab

1. Information on the input data set. *Ontology/Annotation Info* reports the current status of the GO database.
2. Select organism for the GO term analysis.
3. Use this attribute to extract gene names for the input data. You can use attribute names as gene names and adjust gene matching in the *Gene matcher settings* box.
4. Select the reference. You can either have the *entire genome* as reference or a *reference set* from the input.
5. Select the ontology where you want to calculate the enrichment. There are three *Aspect* options:
   - **Biological process**
   - **Cellular component**
   - **Molecular function**
6. A ranked tree (upper pane) and list (lower pane) of GO terms for the selected aspect:
   - **GO term**
   - **Cluster**: number of genes from the input that are also annotated to a particular GO term (and its proportion in all the genes from that term).
   - **Reference**: number of genes that are annotated to a particular GO term (and its proportion in the entire genome).
   - **P-value**: probability of seeing as many or more genes at random. The closer the p-value is to zero, the more significant a particular GO term is. Value is written in e notation.

- **FDR**: false discovery rate - a multiple testing correction that means a proportion of false discoveries among all discoveries up to that FDR value.
- **Genes**: genes in a biological process.
- **Enrichment** level


Input
Filter
Select
Filter GO Term Nodes
1
Genes
#:
1
p-value
p:
0,01000000
FDR
p:
0,01000000
Significance test
2
Binomial
Hypergeometric
Evidence codes in annotation
3
EXP: 0 annots(0 genes)
IDA: 20954 annots(5153 genes)
IPI: 2417 annots(1408 genes)
IMP: 12401 annots(3950 genes)
IGI: 3689 annots(1911 genes)
IEP: 120 annots(96 genes)
ISS: 1897 annots(1095 genes)
ISA: 190 annots(120 genes)
ISO: 8 annots(6 genes)
ISM: 957 annots(861 genes)
IGC: 0 annots(0 genes)
RCA: 6 annots(4 genes)
TAS: 835 annots(547 genes)
NAS: 663 annots(342 genes)
IC: 814 annots(542 genes)
ND: 3687 annots(2171 genes)
IEA: 45412 annots(5451 genes)
NR: 0 annots(0 genes)
Input
Filter
Select
Annotated genes
4
Directly or Indirectly
Directly
Output
5
All selected genes
Term-specific genes
Common term genes
Add GO Term as class

FILTER tab

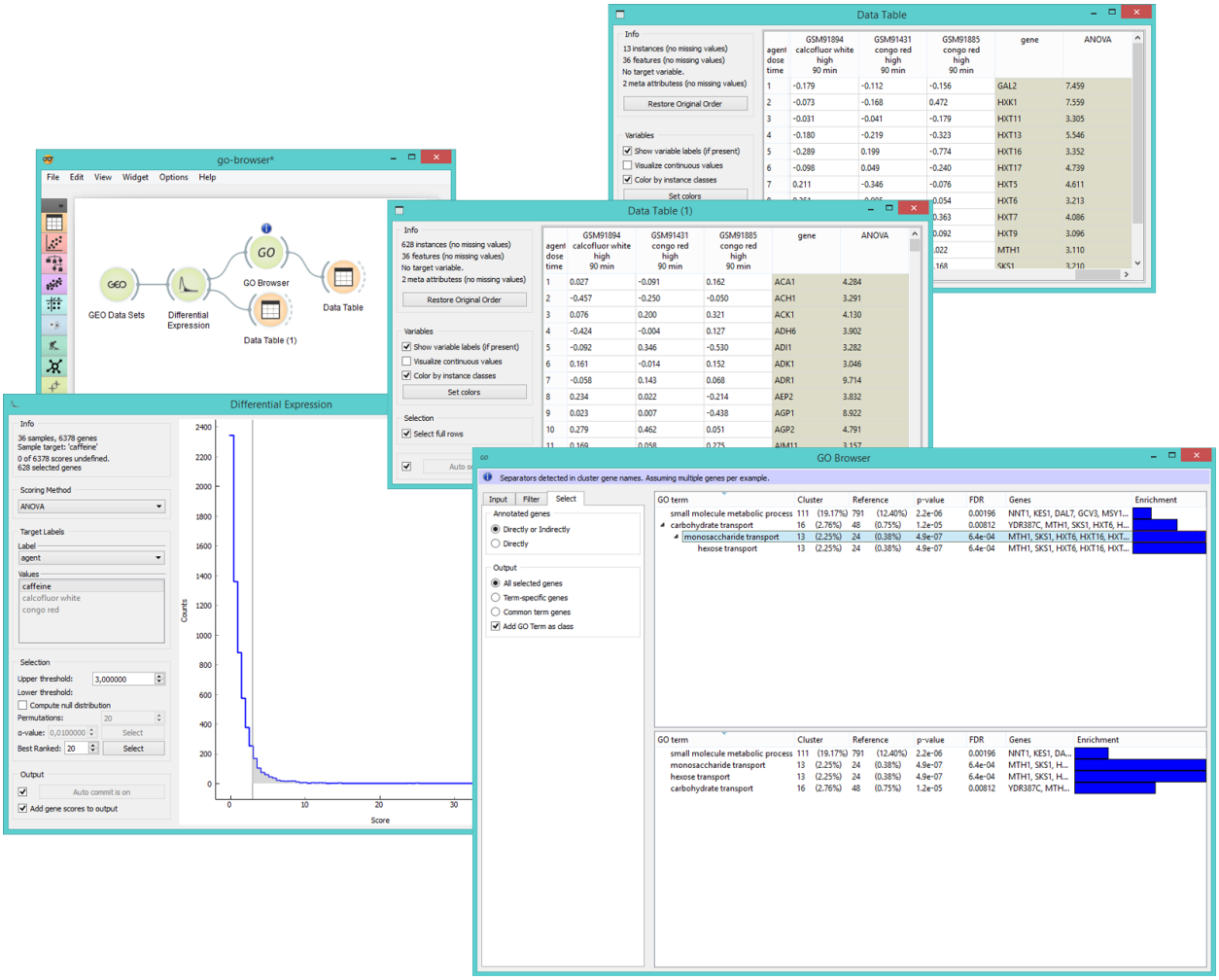
1. *Filter GO Term Nodes* by:
    - **Genes** is a minimal number of genes mapped to a term
    - **P-value** is a max term p-value
    - **FDR**: is a max term false discovery rate
2. *Significance test* specifies distribution to use for null hypothesis:
    - **Binomial**: use a binomial distribution
    - **Hypergeometric**: use a hypergeometric distribution
3. *Evidence codes in annotation* show how the annotation to a particular term is supported.

SELECT tab

1. *Annotated genes* outputs genes that are:
    - **Directly or Indirectly** annotated (direct and inherited annotations)
    - **Directly** annotated (inherited annotations won't be in the output)
2. *Output*:
    - **All selected genes**: outputs genes annotated to all selected GO terms
    - **Term-specific genes**: outputs genes that appear in only one of selected GO terms
    - **Common term genes**: outputs genes common to all selected GO terms
    - **Add GO Term as class**: adds GO terms as class attribute

Example

In the example below we have used **GEO Data Sets** widget, in which we have selected *Caffeine effects: time course and dose response* data set, and connected it to a **Differential Analysis**. Differential analysis allows us to select genes with the highest statistical relevance (we used ANOVA scoring) and feed them to **GO Browser**. This widget lists four biological processes for our selected genes. Say we are interested in finding out more about *monosaccharide transport* as this term has a high enrichment rate. To learn more about which genes are annotated to this GO term we view it in the **Data Table**, where we see all the genes participating in this process listed.



KEGG Pathways

Diagrams of molecular interactions, reactions, and relations.

Signals

Inputs:

- **Data**

  Data set.

- **Reference**

  Referential data set.

Outputs:

- **Selected Data**

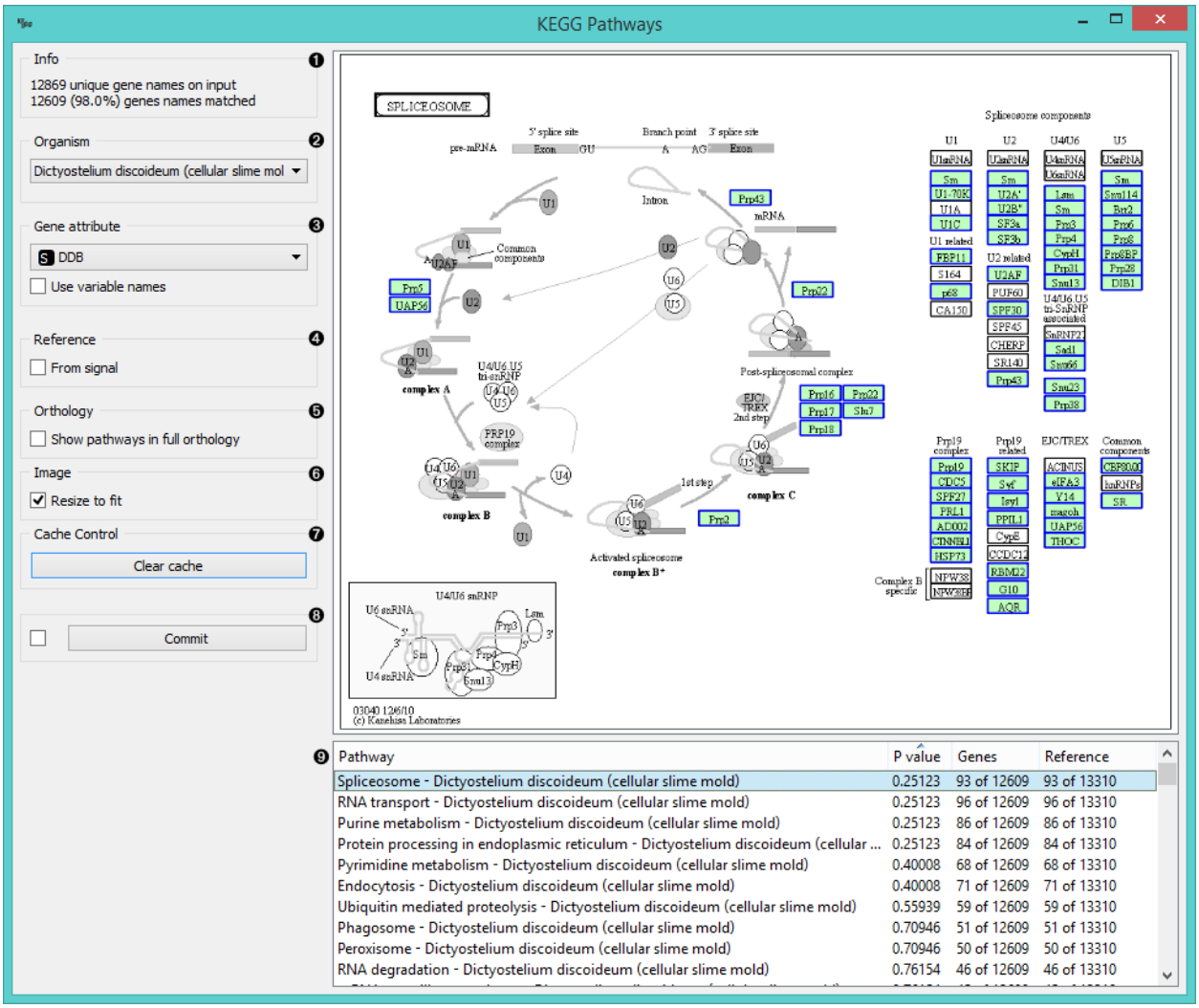
  Data subset.

- **Unselected Data**

  Remaining data.

Description

KEGG Pathways widget displays diagrams of molecular interactions, reactions and relations from the KEGG Pathways Database. It takes data on gene expression as an input, matches the genes to the biological processes and displays a list of corresponding pathways. To explore the pathway, the user can click on any process from the list or arrange them by P-value to get the most relevant processes at the top.



1. Information on the input and the ratio of matched genes.
2. Select the organism for term analysis. The widget automatically selects the organism from the input data.
3. Set the attribute to use for gene names. If gene names are your attribute names, tick *Use variable names*.
4. If you have a separate reference set in the input, tick *From signal* to use these data as reference.
5. To have pathways listed and displayed by vertical descent, tick *Show pathways in full orthology*.

6. To fit the image to screen, tick *Resize to fit*. Untick the box if you wish to explore the pathways.
7. To clear all locally cached KEGG data, press *Clear cache*.
8. When *Auto commit is on*, the widget will automatically apply the changes. Alternatively press *Commit*.
9. A list of pathways either as processes or in full orthology. Click on the process to display the pathway. You can sort the data by P-value to get the most relevant results at the top.

Example

MA Plot

Visualization of intensity-dependent ratios of raw microarray data.

Signals

Inputs:

- **Expression Array**

  DNA microarray.

Outputs:

- **Normalized Expression Array**

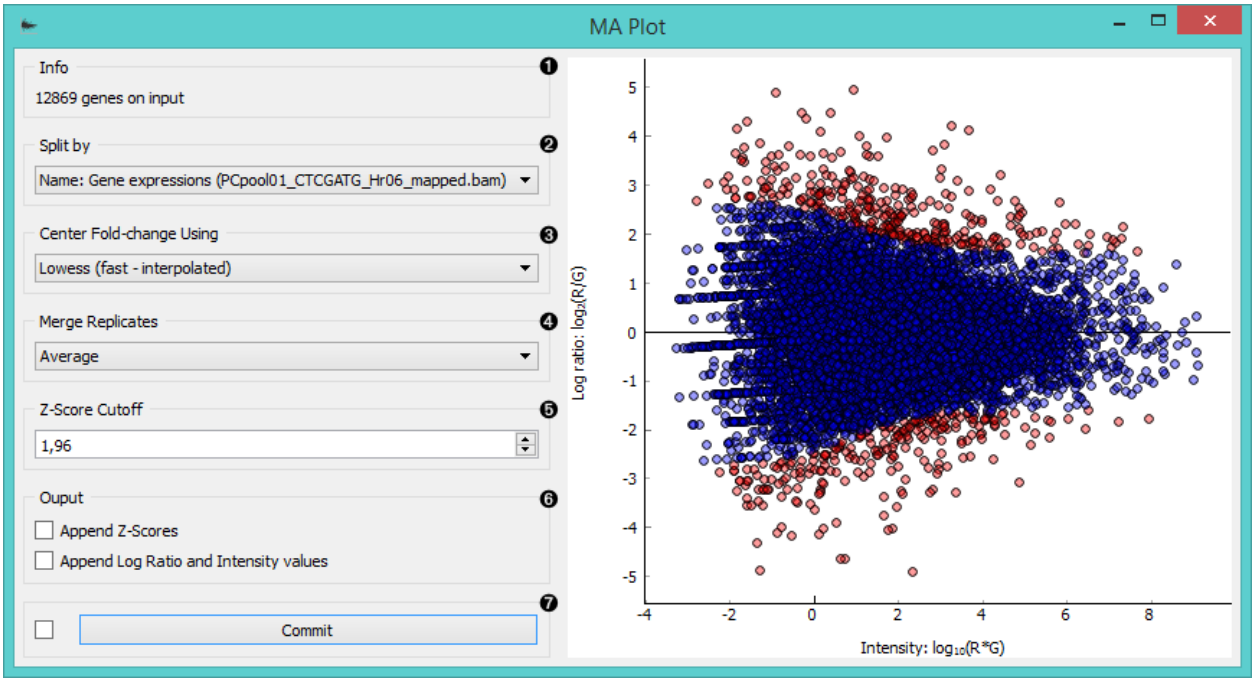
  Lowess-normalized microarray.

- **Filtered Expression Array**

  Selected instances (in the Z-score cutoff).

Description

MA Plot is a graphical method for visualizing intensity-dependent ratio of raw mircoarray data. The A represents the average log intensity of the gene expression (x-axis in the plot), while M stands for the binary log of intensity ratio (y-axis). The widget outputs either normalized data (Lowess normalization method) or instances above the Z-score cutoff line (instances with meaningful fold changes).



1. Information on the input data.
2. Select the attribute to split the plot by.
3. Center the plot using:
   - **average**
   - **Lowess (fast-interpolated)** normalization method
   - **Lowess** normalization method
4. Merge replicated by:
   - **average**
   - **median**
   - **geometric mean**
5. Set the **Z-score cutoff** threshold. Z-score is your confidence interval and it is set to 95% by default. If the widget is set to output *filtered expression array*, instances above the Z-score threshold will be in the output (red dots in the plot).
6. Ticking the *Append Z-scores* will add an additional meta attribute with Z-scores to your output data. Ticking the *Append Log ratio and Intensity values* will add two additional meta attributes with M and A values to your output data.
7. If *Auto commit is on*, the widget will automatically apply changes to the output. Alternatively click *Commit*.

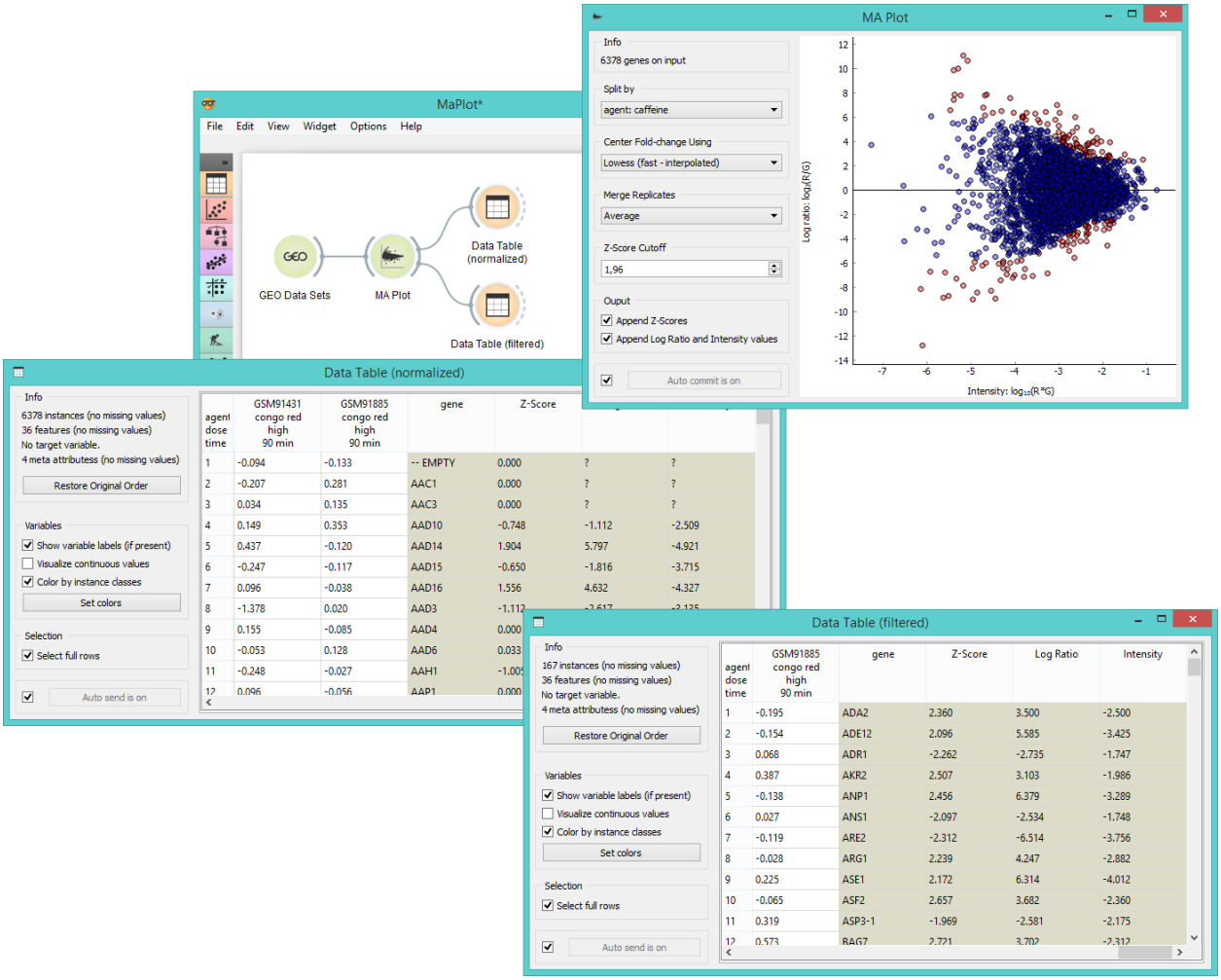
Example

MA Plot is a great widget for data visualization and selection. First we select *Caffeine effect: time course and dose response* data from the **GEO Data Sets** widget and feed it to **MA Plot**. In the plot we see intensity ratios for a selected experiment variable.

We often need to normalize the experiment data to avoid systematic biases, thus we select *Lowess (fast-interpolated)* in the *Center Fold-change* box. By ticking both boxes in the *Output* subsection, we get three new meta attributes

appended - Z-score, Log ratio and Intensity. We see these new attributes and normalized instances in the **Data Table** (normalized).

Another possible output for the MA plot widget is *Filtered expression array*, which will give us instances above the Z-score cutoff threshold (red dots in the plot). We observe these instances the **Data Table** (filtered).



PIPAx

Gives access to **PIPA** databases.

Signals

Inputs:

- (None)

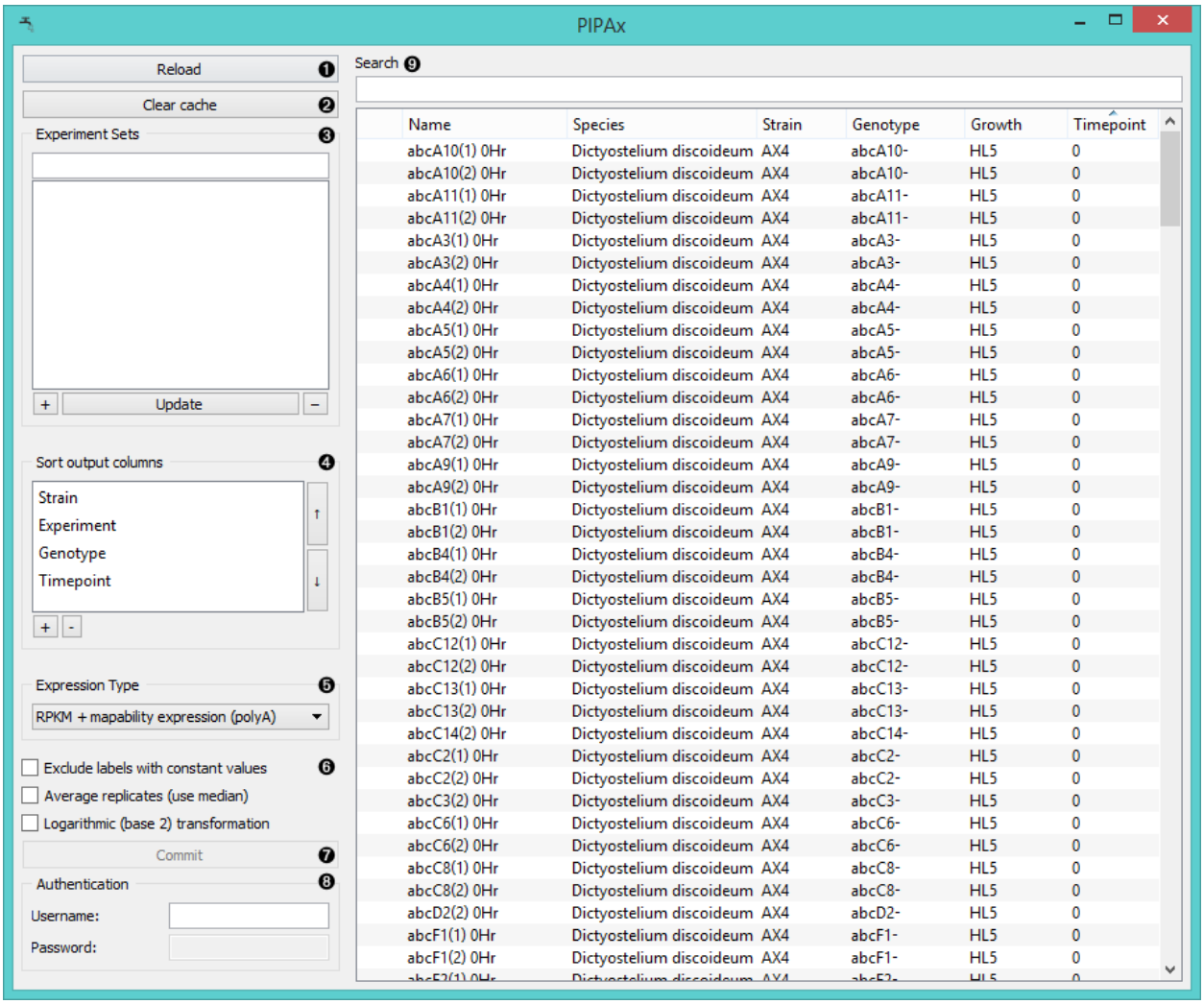
Outputs:

- **Data**

  Selected experiments. Each annotated column contains results of a single experiment or, if the corresponding option is chosen, the average of multiple replicates.

Description

PIPAx is a widget for a direct access to **PIPA** database. It is very similar to the **GenExpress** and **GEO Data Sets** widgets as it allows you to download the data from selected experiments.



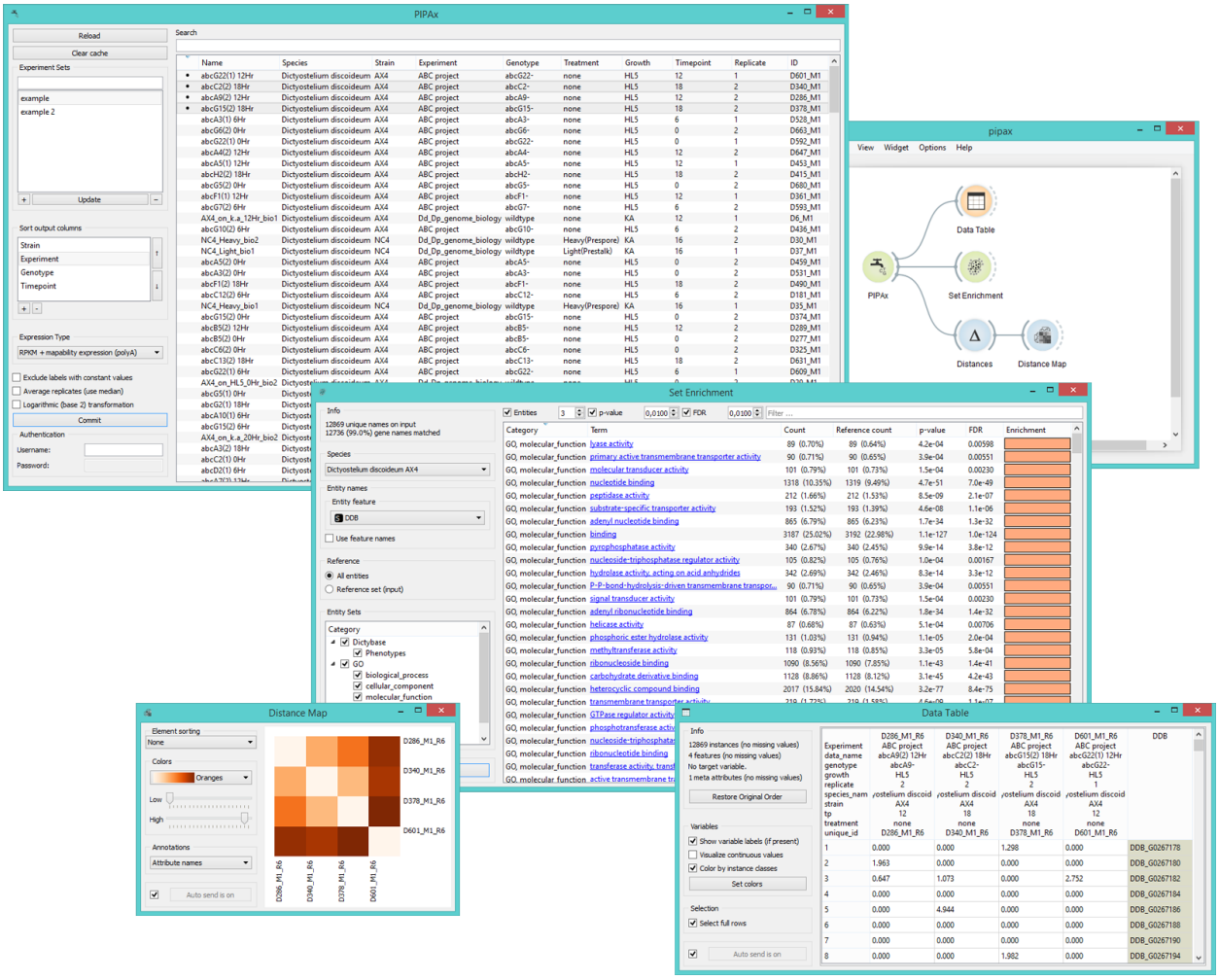
1. Reloads the experiment data.
2. The widget will save (cache) downloaded data, which makes them also available offline. To reset the widget click *Clear cache*.
3. Use *Experiment Sets* to save a selection: select the experiments, click the "**+**" button and name the set. To add experiments to the set, click on its name, select additional experiments and click *Update*.To remove the set click "**-**".
4. In *Sort output columns* set the attributes by which the output columns are sorted. Add attributes with a "+" button and remove them with "-". Switch the sorting order with arrows on the right.
5. Set the expression type for your output data.

- **Raw expression** outputs raw experiment data
- **RPKM expression** outputs data in *reads per kilobase of transcript per million mapped reads*
- **RPKM expression + mapability expression** uses similar normalization, but divides with gene mapability instead of exon lengths.The polyA variants use only polyA (mRNA) mapped hits.

6. **Exclude labels with constant values** removes attribute labels that are the same for all selected experiments from the output data. **Average replicates (use median)** averages identical experiments by using medians as values. **Logarithmic (base 2) transformation** computes the log2(value+1) for each value.
7. Click *Commit* to output selected experiments.
8. Log in to access private data.
9. Experiments can be filtered with the *Search* box. To select which attributes to display right-click on the header. To select multiple experiments click them while holding the *Control/Command* key.

Example

In the schema below we connected **PIPAx** to **Data Table**, **Set Enrichment**, and **Distance Map** (through **Distances**) widgets.



The **Data Table** widget above contains the output from the **PIPAx** widget. Each column contains gene expressions of

a single experiment. The labels are shown in the table header. The **Distance Map** widget shows distances between experiments. The distances are measured with **Distance** widget, which was set to compute *Euclidean* distances.

Quality Control

Computes and plots distances between experiments or replicates.

Signals

Inputs:

- **Data**

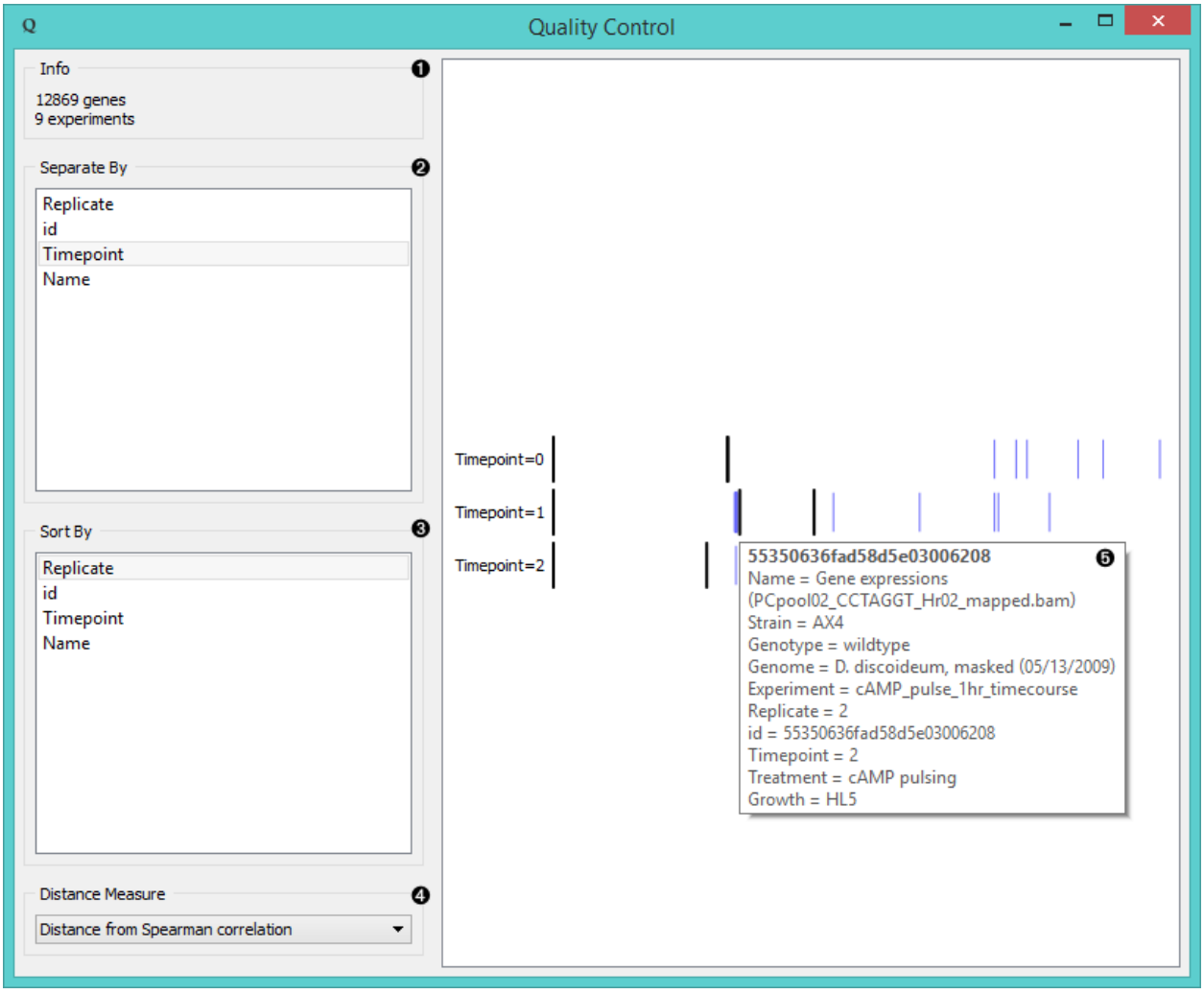
  Data set.

Outputs:

- (None)

Description

Quality Control measures distances between experiments (usually replicates) for a selected label. The widget visualizes distances by selected label. Experiments that lie the farthest from the initial black line should be inspected for anomalies.



1. Information on the input.
2. Separate experiments by label.
3. Sort experiments by label.
4. Compute distances by:
   - **Pearson correlation**
   - **Euclidean distances**
   - **Spearman correlation**
5. Hover over the vertical line to display the information on a chosen instance. Click on the black line to change the reference to that instance.

Example

Quality Control widget gives us feedback on the quality of our data and can be connected to any widget with data output. In the example above (see the image under *Description*) we fed 9 experiments of *Cyclic AMP pulsing* of *Dictyostelium discoideum* from **GenExpress** widget into **Quality Control** and separated them by timepoint label. We found replicate 2 from tp 2 among the tp 1 data, meaning we should inspect these data further.

Select Genes

Manual selection of gene subset.

Signals

Inputs:

- **Data**

  Data set.

- **Gene Subset**

  A subset of genes to be used for gene selection (optional).

Outputs:

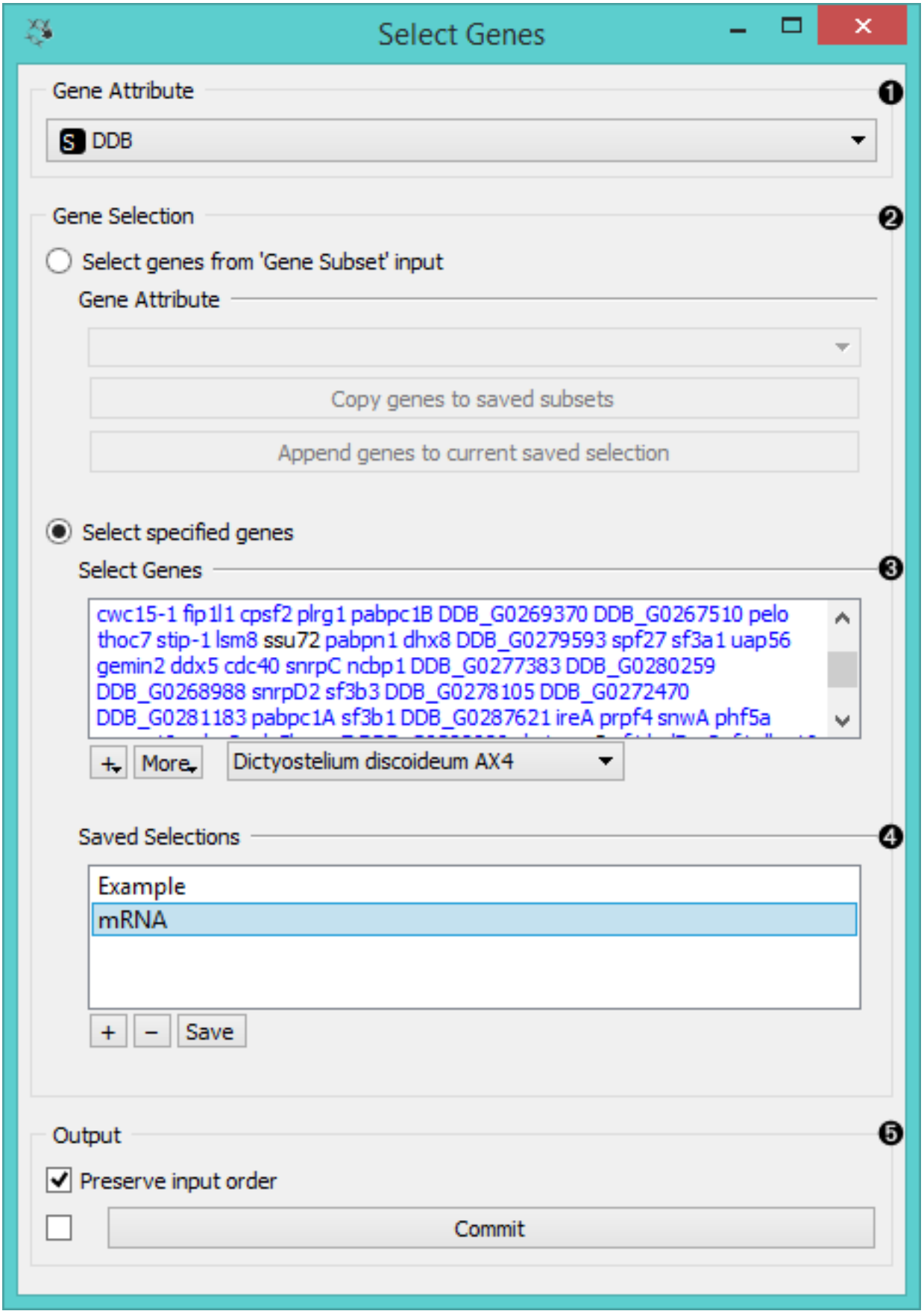
- **Selected Data**

  A subset of genes selected in the widget.

Description

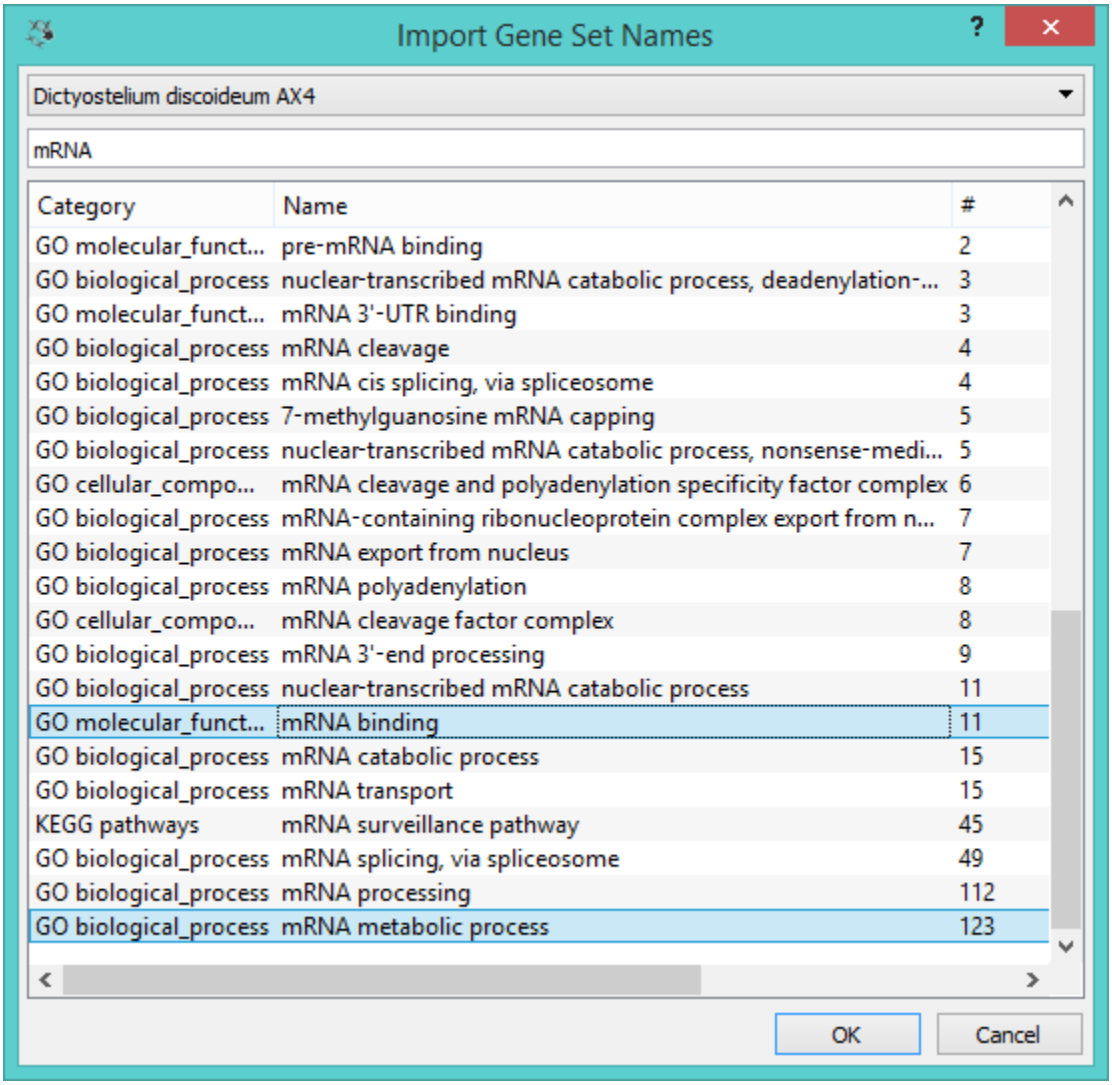
Select Genes widget is used to manually create the gene subset. There are three ways to select genes:

- Manual gene selection (written input). The widget supports autocompletion for gene names.
- Selecting genes from gene sets in the "+" option.
- Selecting genes from a separate input (input can be adjusted in the widget).


Select Genes
Gene Attribute
1
DDB
Gene Selection
2
Select genes from 'Gene Subset' input
Gene Attribute
Copy genes to saved subsets
Append genes to current saved selection
Select specified genes
Select Genes
3
cwc15-1 fip1l1 cpsf2 plrg1 pabpc1B DDB_G0269370 DDB_G0267510 pelo
thoc7 stip-1 lsm8 ssu72 pabpn1 dhx8 DDB_G0279593 spf27 sf3a1 uap56
gemin2 ddx5 cdc40 snrpC ncbp1 DDB_G0277383 DDB_G0280259
DDB_G0268988 snrpD2 sf3b3 DDB_G0278105 DDB_G0272470
DDB_G0281183 pabpc1A sf3b1 DDB_G0287621 ireA prpf4 snwA phf5a
More
Dictyostelium discoideum AX4
Saved Selections
4
Example
mRNA
+
−
Save
Output
5
Preserve input order
Commit

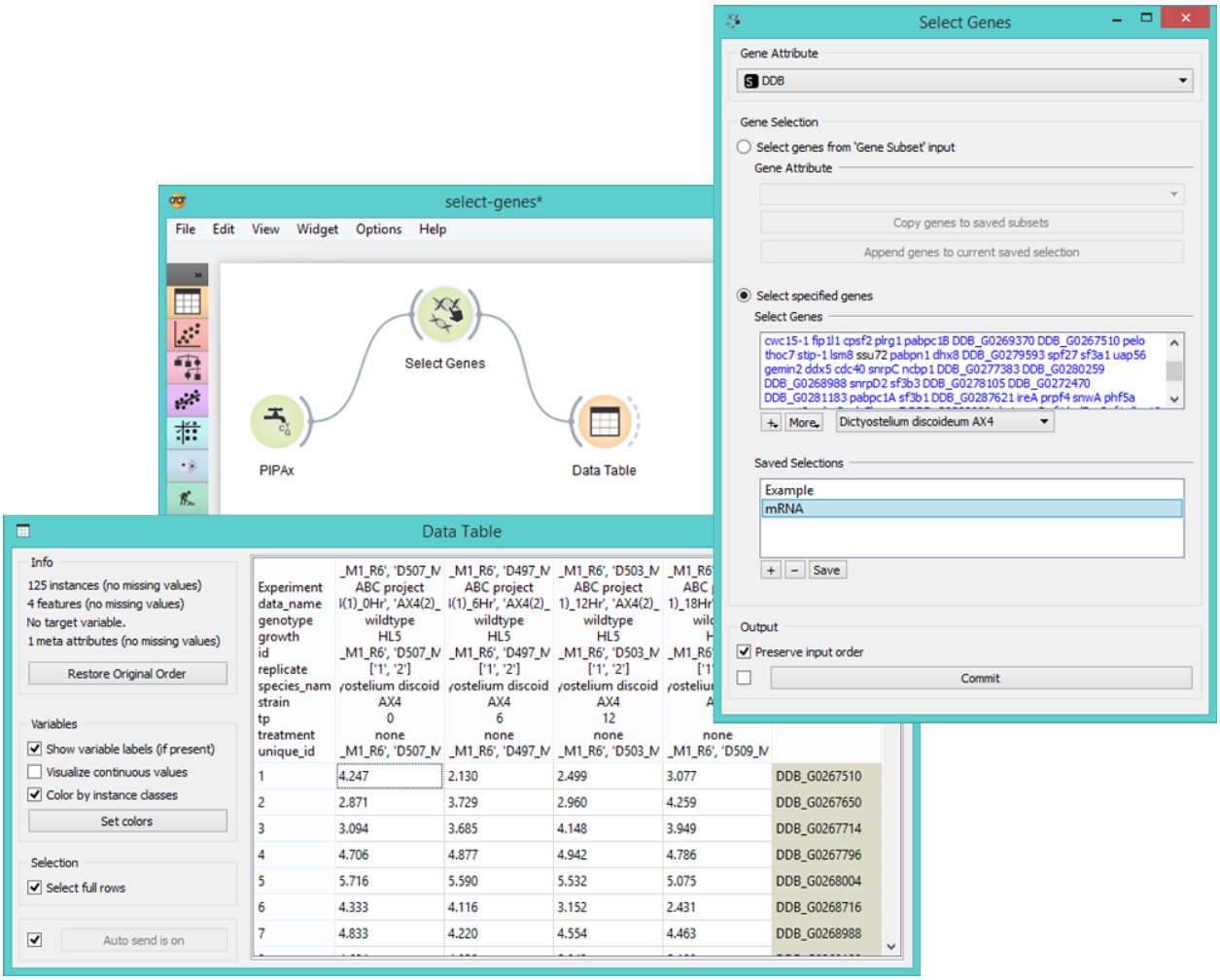
1. Select *Gene Attribute* if there is more than one column with gene names.
2. Specify how you want to select your genes: *Select Genes from 'Gene Subset' input* adds genes from the separate input to selected genes. To create a new saved selection, click *Copy genes to saved subsets*. The genes will be listed in *Select Genes* text area below. To add these genes to an existing selection, click *Append genes to current saved selection*.
3. In *Select specified genes* you can type the gene name and the widget will automatically suggest corresponding genes. Genes that match the genes in the input will be colored blue, while the unmatched will remain black.
4. The "+" button has a drop-down menu with two options.
    - *Import names from gene sets...* gives a list of gene sets and copies genes from selected sets into the list.
    - *Import names from text files...* imports gene names from the file.
5. *More* has two settings: *Complete on gene symbol names* (for easier gene selection) and *Translate all names to official symbol names* (for uniformity).
6. Set the organism to select the genes from (organism from the input data is chosen as default).
7. *Saved Selections* saves the most frequently used genes. "+" adds a new selection, "-" removes the existing one, and *Save* saves the current list. Double-click the selection to rename it.
8. *Output* for this widget is a data subset. If you wish to preserve the order of instances from your input data, tick the *Preserve input order* box. If *Auto commit is on*, all changes will be communicated automatically. Alternatively press *Commit*.

Below is a screenshot of the *Import Gene Set Names* option.



Example

Below is a very simple workflow for this widget. We selected *AX4 Dictyostelium discoideum* data from different time points and two different replicates from **PIPAx** widget. In **Select Genes** we used the *Import names from gene sets...* option and selected two mRNA processes that gave us a list of genes you can see in the *Select Genes* box. Then we fed these data into the **Data Table**. There are 125 genes in the entire *AX4 Dictyostelium discoideum* data that are present in the selected mRNA processes.



Set Enrichment

Determines statistically significant differences in expression levels for biological processes.

Signals

Inputs:

- **Data**

  Data set.

- **Reference**

  Data with genes for the reference set (optional).

Outputs:

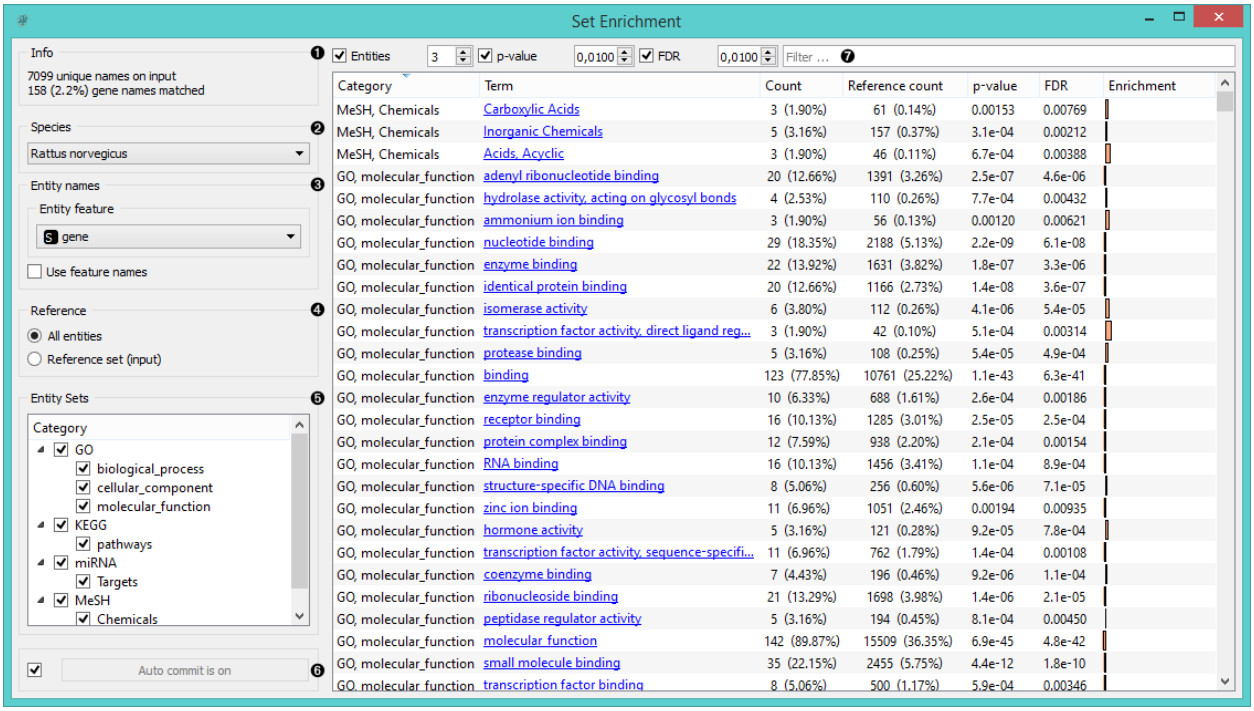
- **Selected data**

  Data subset.

Description

The widget shows a ranked list of terms with p-values, FDR and enrichment. **Set Enrichment** is a great tool for finding biological processes that are over-represented in a particular gene or chemical set.

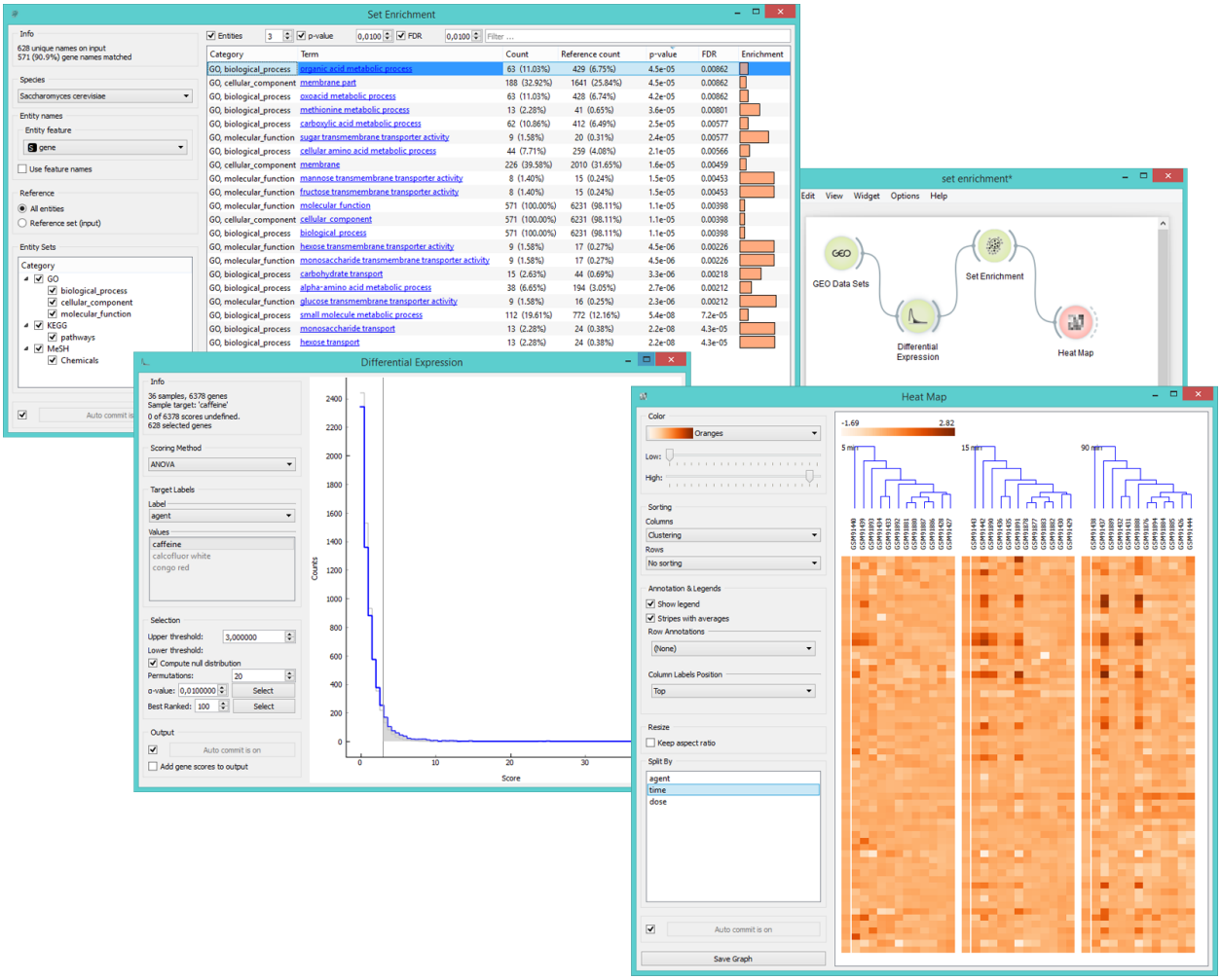
Sets from (GO, KEGG, miRNA and MeSH) come with the Orange installation.



1. Information on the input data set and the ratio of genes that were found in the databases.
2. Select the species.
3. *Entity names* define the features in the input data that you wish to use for term analysis. Tick *Use feature names* if your genes or chemicals are used as attribute names rather than as meta attributes.
4. Select the reference data. You can either have entities (usually genes from the organism - *All Entities*) as a reference or a reference set from the input.
5. Select which *Entity sets* you wish to have displayed in the list.
6. When *Auto commit is on*, the widget will automatically apply the changes. Alternatively press *Commit*.
7. Filter the list by:
   - the minimum number of **entities** included in each term
   - the minimum threshold for **p-value**
   - the maximum threshold for **false discovery rate**
   - a search word

Example

In the example below we have decided to analyse gene expression levels from *Caffeine effect: time course and dose response* data set. We used the ANOVA scoring in the **Differential Expression** widget to select the most interesting genes. Then we fed those 628 genes to **Set Enrichment** for additional analysis of the most valuable terms. We sorted the data by FDR values and selected the top-scoring term. **Heat Map** widget provides a nice visualization of the data.



Volcano Plot

Plots significance versus fold-change for gene expression rates.

Signals

Inputs:

- **Data**

  Input data set.

Outputs:

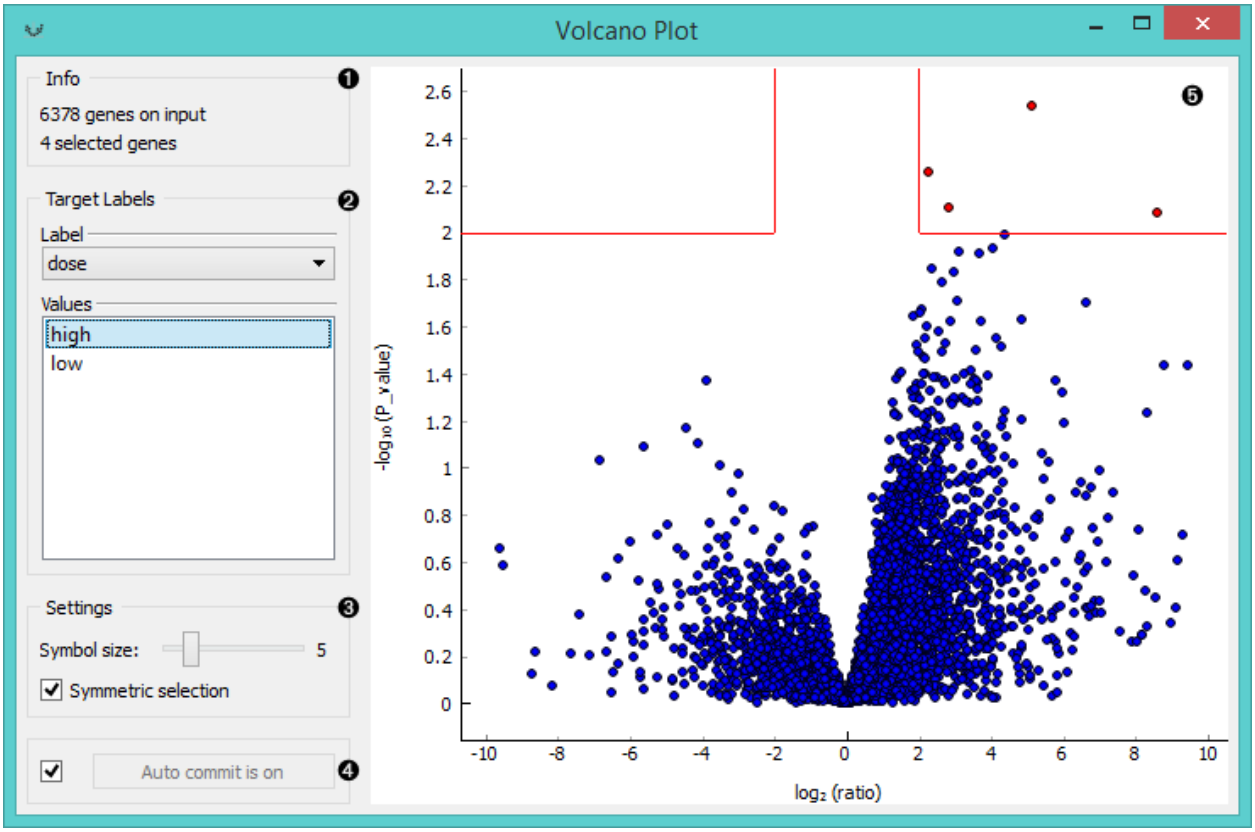
- **Selected data**

  Data subset.

Description

Volcano plot is a graphical method for visualizing changes in replicate data. The widget plots a binary logarithm of fold-change on the x-axis versus statistical significance (negative base 10 logarithm of p-value) on the y-axis.

Volcano Plot is useful for a quick visual identification of statistically significant data (genes). Genes that are highly dysregulated are farther to the left and right, while highly significant fold changes appear higher on the plot. A combination of the two are those genes that are statistically significant - the widget selects the top-ranking genes within the top right and left fields by default.

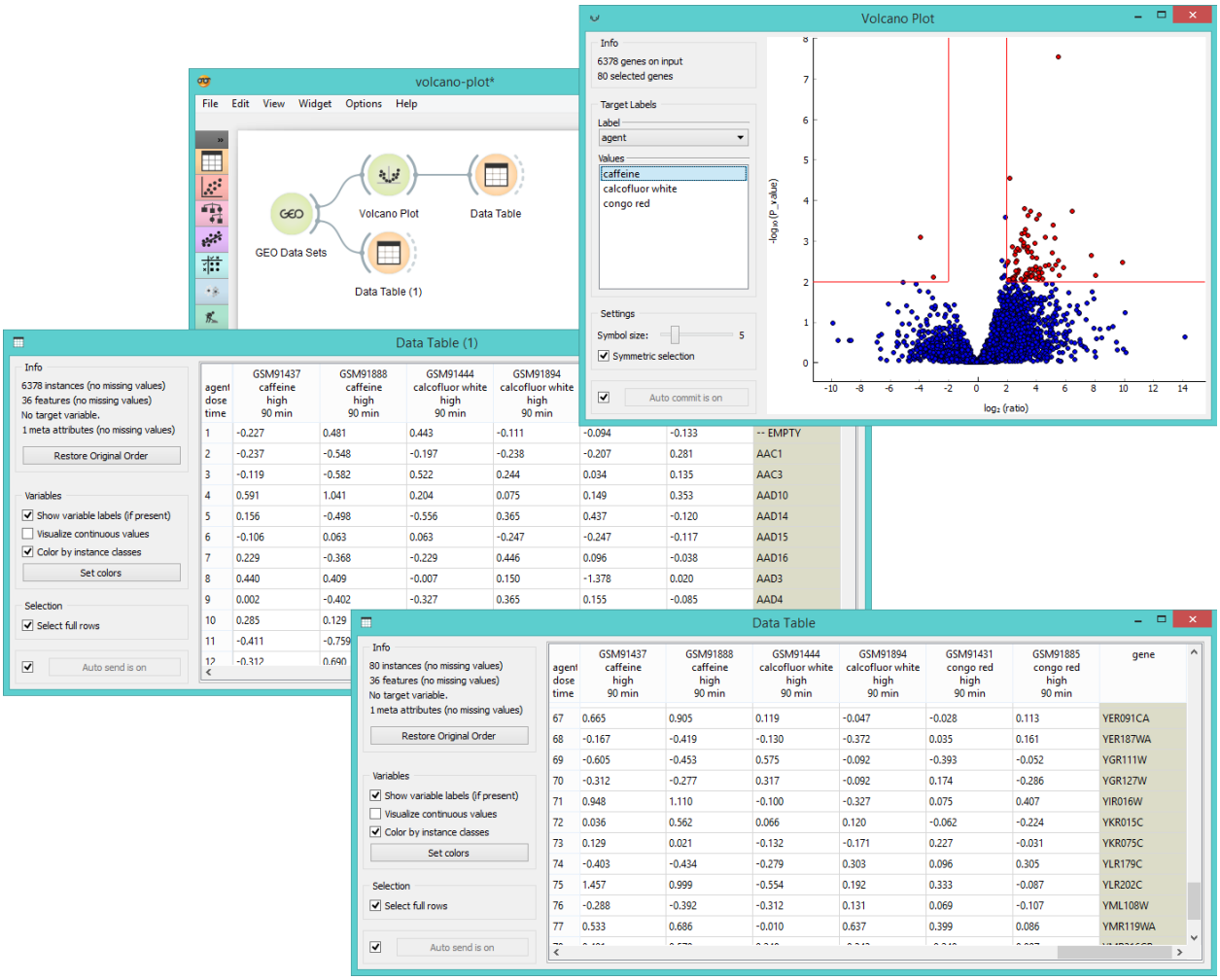


1. Information on the input and output data.
2. Select *Target Labels*. Labels depend on the attributes in the input. In *Values* you can change the sample target (default value is the first value on the list, alphabetically or numerically).
3. Change the *Settings*: adjust the symbol size and turn off symmetrical selection of the output data (the widget selects statistically significant instances by default).
4. If *Auto commit is on* the widget will automatically apply the changes. Alternatively click *Commit*.

5. Visualization of the changes in gene expression. The red lines represent the area with the most statistically significant instances. Symmetrical selection is chosen by default, but you can also manually adjust the area you want in the output.

Example

Below you can see a simple workflow for **Volcano Plot**. We use *Caffeine effect: time course and dose response* data from **GEO Data Sets** widget and visualize them in a **Data Table**. We have 6378 gene in the input, so it is essential to prune the data and analyse only those genes that are statistically significant. **Volcano Plot** helps us do exactly that. Once the desired area is selected in the plot, we output the data and observe them in another **Data Table**. Now we get only 80 instances, which were those genes that had a high normalized fold change under high dose of caffeine and had a low p-value at the same time.



dictyExpress

Gives access to **dictyExpress** databases.

Signals

Inputs:

- (None)

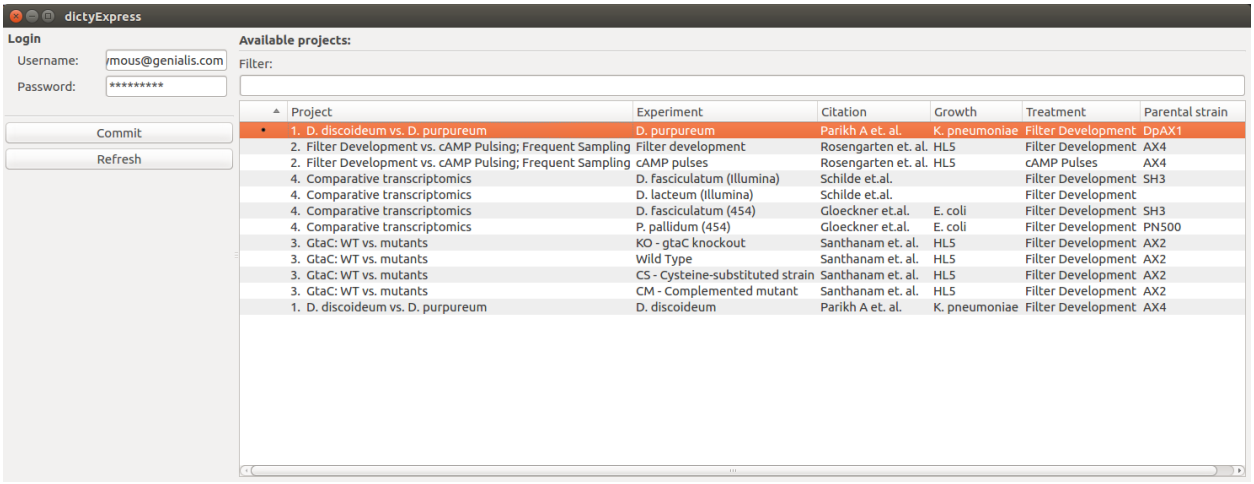
Outputs:

- **Data**

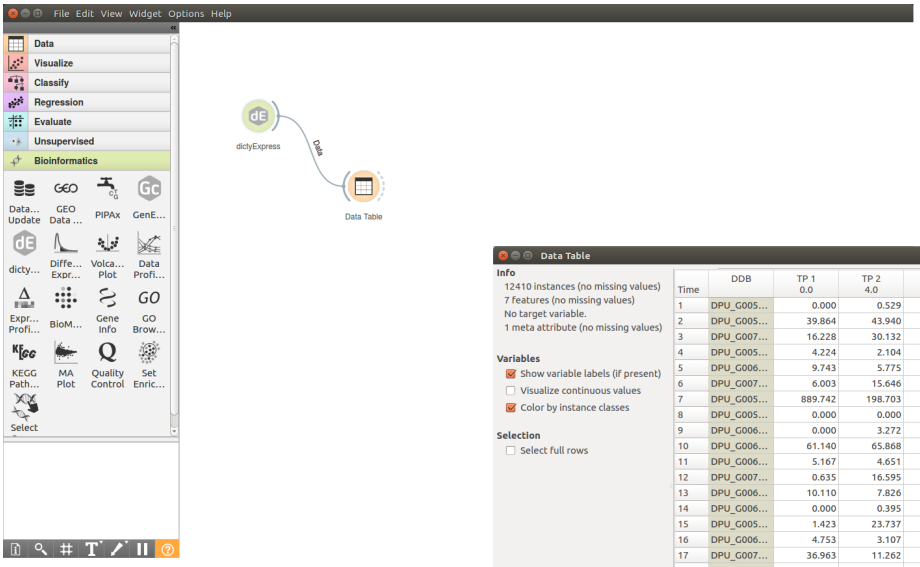
  Selected experiment (time-course gene expression data).

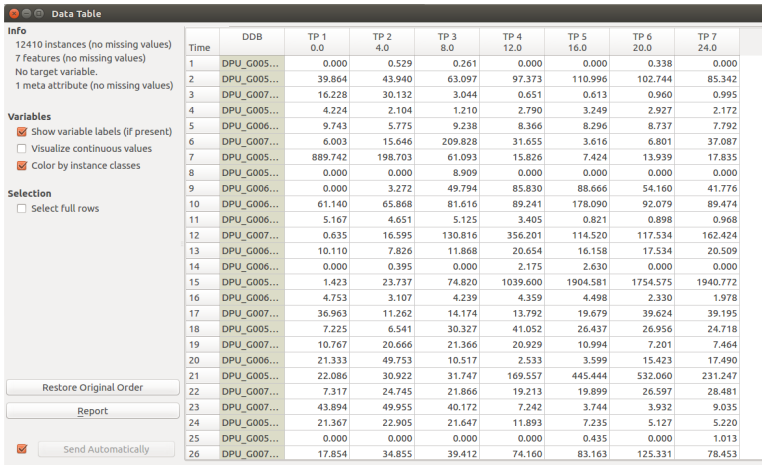
Description

dictyExpress widget gives a direct access to **dictyExpress** database. It allows you to download the data from selected experiments.



Example





Time	DDB	TP 1 0.0	TP 2 4.0	TP 3 8.0	TP 4 12.0	TP 5 16.0	TP 6 20.0	TP 7 24.0
1	DPU_G005…	0.000	0.529	0.261	0.000	0.000	0.338	0.000
2	DPU_G005…	39.864	43.940	63.097	97.373	110.996	102.744	85.342
3	DPU_G007…	16.228	30.132	3.044	0.651	0.613	0.960	0.995
4	DPU_G005…	4.224	2.104	1.210	2.790	3.249	2.927	2.172
5	DPU_G006…	9.743	5.775	9.238	8.366	8.296	8.737	7.792
6	DPU_G007…	6.003	15.646	209.828	31.655	3.616	6.801	37.087
7	DPU_G005…	889.742	198.703	61.093	15.826	7.424	13.939	17.835
8	DPU_G005…	0.000	0.000	8.909	0.000	0.000	0.000	0.000
9	DPU_G006…	0.000	3.272	49.794	85.830	88.666	54.160	41.776
10	DPU_G006…	61.140	65.868	81.616	89.241	178.090	92.079	89.474
11	DPU_G006…	5.167	4.651	5.125	3.405	0.821	0.898	0.968
12	DPU_G007…	0.635	16.595	130.816	356.201	114.520	117.534	162.424
13	DPU_G006…	10.110	7.826	11.868	20.654	16.158	17.534	20.509
14	DPU_G006…	0.000	0.395	0.000	2.175	2.630	0.000	0.000
15	DPU_G005…	1.423	23.737	74.820	1039.600	1904.581	1754.575	1940.772
16	DPU_G006…	4.753	3.107	4.239	4.359	4.498	2.330	1.978
17	DPU_G007…	36.963	11.262	14.174	13.792	19.679	39.624	39.195
18	DPU_G005…	7.225	6.541	30.327	41.052	26.437	26.956	24.718
19	DPU_G007…	10.767	20.666	21.366	20.929	10.994	7.201	7.464
20	DPU_G006…	21.333	49.753	10.517	2.533	3.599	15.423	17.490
21	DPU_G005…	22.086	30.922	31.747	169.557	445.444	532.060	231.247
22	DPU_G007…	7.317	24.745	21.866	19.213	19.899	26.597	28.481
23	DPU_G007…	43.894	49.955	40.172	7.242	3.744	3.932	9.035
24	DPU_G005…	21.367	22.905	21.647	11.893	7.235	5.127	5.220
25	DPU_G005…	0.000	0.000	0.000	0.000	0.435	0.000	1.013
26	DPU_G007…	17.854	34.855	39.412	74.160	83.163	125.331	78.453

CHAPTER 2

Scripting Reference

Array Express (`arrayexpress`)

Access the ArrayExpress web services and database.

ArrayExpress is a database of gene expression experiments that you can query and download.

Retrieve the object representing experiment with accession E-TABM-25

```
>>> from orangecontrib.bio import arrayexpress
>>> experiment = ArrayExpressExperiment("E-TABM-25")
>>> print experiment.accession
E-TABM-25
```

```
>>> print experiment.name
Transcription profiling of aging in the primate brain
```

```
>>> print experiment.species
['Pan troglodytes']
```

```
>>> print experiment.files
[{'kind': ...
```

```
>>> # Retrieve the data matrix for experiment 'E-MEXP-2917'
>>> experiment = ArrayExpressExperiment("E-MEXP-2917")
>>> table = experiment.fgem_to_table()
```

Low level Array Express query using REST services:

```
>>> from orangecontrib.bio import arrayexpress
>>> arrayexpress.query_experiments(accession='E-MEXP-31')
{u'experiments': ...
```

```
>>> arrayexpress.query_experiments(keywords='gliobastoma')
{u'experiments': ...
```

```
>>> arrayexpress.query_files(accession='E-MEXP-32', format="xml")
<xml.etree.ElementTree.ElementTree object ...
```

Note: Currently querying ArrayExpress files only works with the xml format.

Interface

class orangecontrib.bio.arrayexpress.**ArrayExpressConnection**(*address=None*, *timeout=30*, *cache=None*, *username=None*, *password=None*)

Constructs and runs REST query on ArrayExpress.

Parameters

- **address** – Address of the ArrayExpress API.
- **timeout** – Timeout for the connection.

format_query(***kwargs*)

Format the query arguments in *kwargs*.

```
>>> conn.format_query(gxa=True, efcount=(1, 5))
'efcount=[1 TO 5]&gxa=true'
```

open_file(*accession*, *kind='raw'*, *ext=None*)

Return a file handle to experiment data.

Parameters

- **accession** (*str*) –
- **kind** (*str*) – Experiment data type.

Possible values for the parameter *kind*:

- raw: return the raw data if available
- processed: return the processed data if available
- biosamples: a png or svg design image
- idf: investigation description
- adf: array design description
- mageml: MAGE-ML file

Example:

```
>>> raw_file = conn.open_file("E-TABM-1087", kind="raw")
>>> processed_file = conn.open_file("E-TABM-1087", kind="processed")
```

query_experiment(***kwargs*)
Return an open stream to the experiments query results. Takes the same arguments as the *query_experiments* function.

query_files(***kwargs*)
Return an open stream to the files query results. Takes the same arguments as the *query_files* function.

query_url(*what='experiments'*, ***kwargs*)
Return a formatted query URL for the query arguments.

```
>>> conn.query_url(accession="E-MEXP-31")
'http://www.ebi.ac.uk/arrayexpress/json/v2/experiments?accession=E-MEXP-31'
```

query_url_experiments(***kwargs*)
Return query URL of formatted experiments for the query arguments.

query_url_files(***kwargs*)
Return query URL of formatted experiments for the query arguments.

class orangecontrib.bio.arrayexpress.**ArrayDesign**(*adf_file=None*)
Array design (contains the contents of the .adf file).

class orangecontrib.bio.arrayexpress.**SampleDataRelationship**(*sdrf_file=None*)
Sample-Data Relationship (contains the contents of the .sdrf file).

array_data()
Return the Array Data subsection.

array_data_file()
Return the Array Data File subsection.

array_data_matrix()
Return the Array Data Matrix subsection.

array_data_matrix_file()
Return the Array Data Matrix File subsection.

assay()
Return the Assay subsection.

assay_name()
Return the Assay Name subsection.

derived_array_data()
Return the Derived Array Data subsection.

derived_array_data_file()
Return the Derived Array Data File subsection.

derived_array_data_matrix()
Return the Derived Array Data Matrix subsection.

derived_array_data_matrix_file()
Return the Derived Array Data Matrix File subsection.

extract()
Return the Extract subsection.

extract_name()
Return the Extract Name subsection.

hybridization()
Return the Hybridization subsection.

hybridization_name()
Return the Hybridization Name subsection.

image()
Return the Image subsection

image_file()
Return the Image File subsection.

labeled_extract()
Return the Labeled Extract subsection.

labeled_extract_name()
Return the Labeled Extract Name subsection.

normalization()
Return the Normalization subsection.

normalization_name()
Return the Normalization Name subsection.

sample()
Return the Sample subsection.

sample_name()
Return the Sample Name subsection

scan()
Return the Scan subsection.

scan_name()
Return the Scan name subsection.

source()
Return the Source subsection.

source_name()
Return the Source Name subsection.

transform_tag(*tag*)
Transform the tag into a proper Python attribute name by replacing all spaces and special characters (e.g '[', ']' into underscores).

class orangecontrib.bio.arrayexpress.**InvestigationDesign**(*idf_file=None*)
Investigation design (contains the contents of the .idf).

```
>>> idf_file = six.StringIO(
...     'Investigation Title\tfoo investigation\n' +
...     'Experimental Design\tfubar\tsnafu\n' +
...     'SDRF File\tfoobar.sdrf\n'
... )
>>> idf = InvestigationDesign(idf_file)
>>> print(idf.investigation_title)
foo investigation
>>> print(idf.experimental_design)
['fubar', 'snafu']
>>> print(idf.sdrf_file)
['foobar.sdrf']
```

transform_tag(*tag*)
Transform the tag into a proper python attribute name by replacing all spaces and special characters (e.g '[', ']' into underscores).

Low-level querying with REST

orangecontrib.bio.arrayexpress.**query_experiments**(*keywords=None*, *accession=None*, *array=None*, *ef=None*, *efv=None*, *expdesign=None*, *exptype=None*, *gxa=None*, *pmid=None*, *sa=None*, *species=None*, *expandefo=None*, *directsub=None*, *assaycount=None*, *efcount=None*, *samplecount=None*, *rawcount=None*, *fgemcount=None*, *miamescore=None*, *date=None*, *format='json'*, *wholewords=None*, *connection=None*)

Query Array Express experiments.

Parameters

- **keywords** – A list of keywords to search (e.g. `['gliobastoma']`).
- **accession** – Search by experiment accession (e.g. `'E-MEXP-31'`).
- **array** – Search by array design name or accession (e.g. `'A-AFFY-33'`).
- **ef** – Experimental factor (names of main variables of experiments).
- **efv** – Experimental factor value (Has EFO expansion).
- **expdesign** – Experiment design type (e.g. `["dose", "response"]`).
- **exptype** – Experiment type (e.g. 'RNA-Seq', has EFO expansion).
- **gxa** – If True limit the results to the Gene Expression Atlas only.
- **pmid** – Search by PubMed identifier.
- **sa** – Sample attribute values (e.g. `'fibroblast'`, has EFO expansion).
- **species** – Search by species (e.g. `'Homo sapiens'`, has EFO expansion)
- **expandefo** – If True expand the search terms with all its child terms in the Experimental Factor Ontology (EFO) (e.g. `keywords="cancer"` will be expanded to include for synonyms and sub types of cancer).
- **directsub** – If True return only experiments submitted directly to Array Express; if False return only experiments imported from GEO database; if None (default) return both.
- **assaycount** – A two tuple (min, max) for filter on the number of assays (e.g. (1, 5) will return only experiments with at least 1 and no more then 5 assays).
- **efcount** – Filter on the number of experimental factors (e.g. (1, 5)).
- **sacount** – Filter on the number of sample attribute categories.
- **rawcount** – Filter on the number or raw files.
- **fgemcount** – Filter on the number of final gene expression matrix (processed data) files.
- **miamescore** – Filter on the MIAME complience score (max 5).
- **date** – Filter by release date.

```
>>> query_experiments(species="Homo sapiens", ef="organism_part", efv="liver")
{...
```

orangecontrib.bio.arrayexpress.**query_files**(*keywords=None*, *accession=None*, *array=None*, *ef=None*, *efv=None*, *expdesign=None*, *exptype=None*, *gxa=None*, *pmid=None*, *sa=None*, *species=None*, *expandefo=None*, *directsub=None*, *assaycount=None*, *efcount=None*, *samplecount=None*, *rawcount=None*, *fgemcount=None*, *miamescore=None*, *date=None*, *format='json'*, *wholewords=None*, *connection=None*)

Query Array Express files. See *query_experiments* for the arguments.

```
>>> query_files(species="Mus musculus", ef="developmental_stage",
...             efv="embryo", format="xml")
<xml.etree.ElementTree.ElementTree object ...
```

Bio Mart (biomart)

Access BioMart MartService.

```
>>> from orangecontrib.bio.biomart import *
>>> connection = BioMartConnection(
...     "http://www.biomart.org/biomart/martservice")
...
>>> reg = BioMartRegistry(connection)
>>> for mart in reg.marts():
...    print mart.name
...
ensembl...
>>> dataset = BioMartDataset(
...     mart="ensembl", internalName="hsapiens_gene_ensembl",
...     virtualSchema="default", connection=connection)
...
>>> for attr in dataset.attributes()[:10]:
...    print attr.name
...
Ensembl Gene ID...
>>> data = dataset.get_data(
...    attributes=["ensembl_gene_id", "ensembl_peptide_id"],
...    filters=[("chromosome_name", "1")])
...
>>> query = BioMartQuery(reg.connection, virtualSchema="default")
>>> query.set_dataset("hsapiens_gene_ensembl")
>>> query.add_attribute("ensembl_gene_id")
>>> query.add_attribute("ensembl_peptide_id")
>>> query.add_filter("chromosome_name", "1")
>>> count = query.get_count()
```

Interface

class orangecontrib.bio.biomart.**BioMartConnection**(*address=None*, *timeout=30*)

A connection to a BioMart martservice server.

```
>>> connection = BioMartConnection(
...     "http://www.biomart.org/biomart/martservice")
>>> response = connection.registry()
>>> response = connection.datasets(mart="ensembl")
```

class orangecontrib.bio.biomart.**BioMartRegistry**(*stream*)

A class representing a BioMart registry. Arguments:

Parameters **stream** – A file like object with xml registry or a BioMartConnection instance

```
>>> registry = BioMartRegistry(connection)
>>> for schema in registry.virtual_schemas():
...     print(schema.name)
...
default
```

databases()

Same as *marts*.

dataset(*internalName*, *virtualSchema=None*)

Return a BioMartDataset instance that matches the *internalName*.

datasets()

Return a list of all datasets (*BioMartDataset*) from all marts regardless of their virtual schemas.

links_between(*exporting*, *importing*, *virtualSchema='default'*)

Return all links between *exporting* and *importing* datasets in the *virtualSchema*.

mart(*name*)

Return a named mart.

marts()

Return a list off all 'mart' instances (*BioMartDatabase*) regardless of their virtual schemas.

classmethod parse(*stream*, *parser=None*)

Parse the registry file like object and return a DOM like description (XMLNode).

query(***kwargs*)

Return an initialized *BioMartQuery* with registry set to self. Pass additional arguments to BioMartQuery.__init__ with keyword arguments.

virtual_schema(*name*)

Return a named virtual schema.

virtual_schemas()

Return a list of *BioMartVirtualSchema* instances representing each schema.

class orangecontrib.bio.biomart.**BioMartQuery**(*registry*, *virtualSchema='default'*, *dataset=None*, *attributes=[]*, *filters=[]*, *count=False*, *uniqueRows=False*, *format='TSV'*, *serverVirtualSchema=None*)

Construct a query to run on a BioMart server.

```
>>> query = BioMartQuery(connection,
...                      dataset="hsapiens_gene_ensembl",
...                      attributes=["ensembl_transcript_id",
...                                  "chromosome_name"],
...                      filters=[("chromosome_name", ["22"])])
...
>>> count = query.get_count()
```

```
>>> print(count)
1221
```

```
>>> # Equivalent to
>>> query = BioMartQuery(connection)
>>> query.set_dataset("hsapiens_gene_ensembl")
>>> query.add_filter("chromosome_name", "22")
>>> query.add_attribute("ensembl_transcript_id")
>>> query.add_attribute("chromosome_name")
>>> count = query.get_count()
>>> print(count)
1221
```

class orangecontrib.bio.biomart.**BioMartDataset**(*mart='ensembl'*, *internalName='hsapiens_gene_ensembl'*, *virtualSchema='default'*, *connection=None*, *datasetType='TableSet'*, *displayName=''*, *visible='1'*, *assembly=''*, *date=''*, *serverVirtualSchema=None*, ***kwargs*)

A BioMart dataset (returned by *BioMartDatabase*).

attributes()
Return a list of available attributes for this dataset (*Attribute*).

configuration(*parser=None*)
Return the configuration tree for this dataset (*DatasetConfig*).

count(*filters=[]*, *unique=False*)
Construct and run a *BioMartQuery* and count the number of returned lines.

filters()
Return a list of available filters for this dataset (*Filter*).

get_data(*attributes=[]*, *filters=[]*, *unique=False*)
Construct and run a *BioMartQuery* and return its results.

class orangecontrib.bio.biomart.**BioMartVirtualSchema**(*locations=None*, *name='default'*, *connection=None*)

A virtual schema.

databases()
Same as *marts*.

dataset(*internalName*)
Return a dataset with matching *internalName*.

datasets()
Return a list of all datasets (*BioMartDataset*) from all marts in this schema.

links()
Return a list of (linkName, linkVersion) tuples defined by datasets in the schema.

links_between(*exporting*, *importing*)
Return a list of link names from *exporting* dataset to *importing* dataset.

marts()
Return a list off all 'mart' instances (*BioMartDatabase*) in this schema.

query(***kwargs*)
Return an initialized *BioMartQuery* with registry and *virtualSchema* set to self. Pass additional arguments to BioMartQuery.__init__ with keyword arguments

class orangecontrib.bio.biomart.**BioMartDatabase**(*name='ensembl'*, *virtualSchema='default'*, *connection=None*, *database='ensembl_mart_60'*, *default='1'*, *displayName='ENSEMBL GENES 60 (SANGER UK)'*, *host='www.biomart.org'*, *includeDatasets=''*, *martUser=''*, *path='/biomart/martservice'*, *port='80'*, *serverVirtualSchema='default'*, *visible='1'*, ***kwargs*)

A BioMart 'mart' instance.

Parameters

- **name** (*str*) – Name of the mart instance.
- **virtualSchema** (*str*) – Name of the virtualSchema of the dataset.
- **BioMartConnection** – An optional BioMartConnection instance.

dataset_attributes(*dataset*, ***kwargs*)
Return a list of dataset attributes.

dataset_filters(*dataset*, ***kwargs*)
Return a list of dataset filters.

dataset_query(*dataset*, *filters=[]*, *attributes=[]*)
Return an dataset query based on dataset, filters and attributes.

datasets()
Return a list of all datasets (*BioMartDataset*) in this database.

class orangecontrib.bio.biomart.**Attribute**
An attribute in a BioMart data-set.

description
Human readable description

format
Attribute format

internalName
Attribute's internal name

internal_name
Attribute's internal name

name
Human readable name.

class orangecontrib.bio.biomart.**Filter**
A filter on a BioMart data-set.

description
Filter description

internal_name
Internal name

name
Filter name

values
List of possible filter values

class orangecontrib.bio.biomart.**DatasetConfig**(*registry*, **args*, ***kwargs*)

Dictyostelium discoideum databases (dicty)

The following example downloads experiments from the PIPA database, specifically "RPKM + mapability expression (polyA) - R6" results for all public experiments on Dictyostelium discoideum (dd) at time point 16.

```
import orangecontrib.bio.dicty

pipa = orangecontrib.bio.dicty.PIPAx()

results = pipa.results_list("R6")
dd16 = [ (i,d) for i,d in results.items() if \
        d["tp"] == '16' and d["species_id"] == "dd" ]

#group similar experiments with sorting
dd16 = sorted(dd16, key=lambda x: (x[1]["treatment"], x[1]["replicate"]))

data = pipa.get_data([i for i,d in dd16], exclude_constant_labels=True, \
    allowed_labels=["id", "treatment", "replicate"])

def print_data(data):
    for at in data.domain.attributes:
        print("%s treatment: %s replicate: %s" % \
            (at.name, at.attributes["treatment"], at.attributes["replicate"]))
    print("")
    for a in data[:10]:
        print(a)

print_data(data)

print("")
datar = orangecontrib.bio.dicty.join_replicates(data)

print_data(datar)
```

PIPAx database

class orangecontrib.bio.dicty.**PIPAx**(*address='https://pipa.biolab.si/pipax/api.py'*, *cache=None*, *username=None*, *password=None*)

An interface to PIPAx API.

__init__(*address='https://pipa.biolab.si/pipax/api.py'*, *cache=None*, *username=None*, *password=None*)

Parameters

- **address** (*str*) – The address of the API.
- **username** (*str*) –
- **password** (*str*) – Login info; None for public access.
- **cache** (CacheSQLite) – A cache that stores results locally (an *CacheSQLite*).

genomes(*reload=False*, *bufver='0'*)

Return a list of available genomes as a list of (genome_id, genome_name) tuples.

get_data(*ids=None*, *result_type=None*, *exclude_constant_labels=False*, *average=<function median>*, *callback=None*, *bufver='0'*, *transform=None*, *allowed_labels=None*, *reload=False*)
Return data in a `Orange.data.Table`. Each feature represents a sample and each row is a gene. The feature's `.attributes` contain annotations.

Parameters

- **ids** (*list*) – List of ids as returned by *results_list* if *result_type* is None; list of ids as returned by *mappings* if *result_type* is set.
- **result_type** (*str*) – Result template type id as returned by *result_types*.
- **exclude_constant_labels** (*bool*) – If a label has the same value in whole example table, remove it.
- **average** (*function*) – Function that combines multiple reading of the same gene on a chip. If None, no averaging is done. Function should take a list of floats and return an "averaged" float (the default functions returns the median).
- **transform** (*function*) – A function that transforms individual values. It should take and return a float. Example use: logarithmic transformation. Default: None.

mappings(*reload=False*, *bufver='0'*)
Return available mappings as dictionary of { mapping_id: dictionary_of_annotations } where the keys for dictionary_of_annotations are "id", data_id", "data_name", "genomes_id".

result_types(*reload=False*, *bufver='0'*)
Return a list of available result types.

results_list(*rtype*, *reload=False*, *bufver='0'*)
Return a list of available gene expressions for a specific result type. Returns a dictionary, where the keys are ID and values are dictionaries of sample annotations.

Parameters **rtype** (*str*) – Result type to use (see *result_types*).

DictyExpress database

class `orangecontrib.bio.dicty.`**DictyExpress**(*address='http://bcm.fri.uni-lj.si/microarray/api/index.php?'*, *cache=None*)
Access the DictyExpress data API.

__init__(*address='http://bcm.fri.uni-lj.si/microarray/api/index.php?'*, *cache=None*)

Parameters

- **address** (*str*) – The address of the API.
- **cache** – A cache that stores results locally (an instance of *CacheSQLite*).

annotationOptions(*ao=None*, *onlyDiff=False*, ***kwargs*)
Return annotation options for given query. Return all possible annotations if the query is omitted.

If *ao* is chosen, only return options for that object id.

annotationTypes()
Returns all annotation types.

annotations(*type*, *ids=None*, *all=False*)
Return annotations for specified type and ids.

Parameters

- **type** – Object type (see *objects*).

- **ids** – If set, only annotations corresponding to the given ids are returned. Annotations are in the same order as input ids.
- **all** – If False (default), only annotations for "meaningful" annotation types are returned. If True, return annotations for all annotation types.

get_data(*type='norms'*, *exclude_constant_labels=False*, *average=<function median>*, *ids=None*, *callback=None*, *format='short'*, *transform=None*, *allowed_labels=None*, ***kwargs*)

Return data in a `Orange.data.Table`. Each feature is a sample and each row(`Orange.data.Instance` is a gene. The feature's `.attributes` contain annotations.

Parameters

- **ids** (`list`) – A list of chip ids. If absent, make a search. In this case any additional keyword arguments are threated as in *search*.
- **exclude_constant_labels** – Remove labels if they have the same value for the whole table.
- **format** (*str*) – If "short", use short format for downloads.
- **average** (*function*) – Function that combines multiple reading of the same gene on a chip. If None, no averaging is done. Function should take a list of floats and return an "averaged" float (the default functions returns the median).
- **transform** (*function*) – A function that transforms individual values. It should take and return a float. Example use: logarithmic transformation. Default: None.

Returns Chips with given ids in a single data table.

Return type `Orange.data.Table`

objects()

Return all objects types.

search(*type*, ***kwargs*)

Search the database. Search is case insensitive.

Parameters

- **type** – Annotation type (list them with `DictyExpress().saoids.keys()`).
- **kwargs** – In the form `annotation=values`. Values can are either strings or a list of strings (interpreted as an OR operator between list elements).

The following example lists ids of normalized entries where platform is minichip and sample is abcC3-:

```
search("norms", platform='minichip', sample='abcC3-')
```

The following example lists ids of normalized entries where platform is minichip and sample is abcC3- or abcG15-:

```
search("norms", platform='minichip', sample=[ 'abcC3-', 'abcG15-'])
```

Auxillary functionality

class `orangecontrib.bio.dicty.`**CacheSQLite**(*filename*, *compress=True*)

An SQLite-based cache.

__init__(*filename*, *compress=True*)

Opens an existing cache or creates a new one if it does not exist.

Parameters

- **filename** (*str*) – The filename.
- **compress** (*bool*) – Whether to use on-the-fly compression.

add(*addr*, *con*, *version='0'*, *autocommit=True*)
Inserts an element into the cache.

Parameters

- **addr** – Element address.
- **con** – Contents.
- **version** – Version.

clear()
Remove all entries.

commit()
Commit the changes. Run only if previous *add* was called without autocommit.

contains(*addr*)
Return the element's version or False, if the element does not exists.

Parameters **addr** – Element address.

get(*addr*)
Loads an element from the cache.

Parameters **addr** – Element address.

list()
List all element addresses in the cache.

D. discoideum Mutant Phenotypes (dicty.phenotypes)

This modules provides an interface to Dictyostelium mutant phenotypes data from the dictyBase. The mutants are presented as *DictyMutant* objects with their respective name, strain descriptor, associated genes and associated phenotypes.

```
>>> from orangecontrib.bio.dicty.phenotypes import *
>>> # Create a set of all mutant objects
>>> dicty_mutants = mutants()
>>> # List a set of all genes referenced by a single mutant
>>> print mutant_genes(dicty_mutants[0])
['cbfA']
>>> # List a set of all phenotypes referenced by a single mutant
>>> print mutant_phenotypes(dicty_mutants[0])
['aberrant protein localization']
```

Classes and Functions

class orangecontrib.bio.dicty.phenotypes.**DictyMutant**(*mutant_entry*)
A single Dictyostelium discoideum mutant from the Dictybase.

Parameters **mutant_entry** (*str*) – A single mutant entry from curated mutants file.

Variables

- **DictyMutant.name** – dictyBase ID.
- **DictyMutant.descriptor** – dictyBase strain descriptor.
- **DictyMutant.genes** – all associated genes.
- **DictyMutant.phenotypes** – all associated phenotypes.

class orangecontrib.bio.dicty.phenotypes.**DictyMutants**(*local_database_path=None*)
A collection of Dictybase mutants as a dictionary of *DictyMutant* objects.

Parameters **local_database_path** – A path for storing D. dictyostelium mutants objects. If *None* then a default database path is used.

orangecontrib.bio.dicty.phenotypes.**mutants**()
Return all *DictyMutant* objects.

orangecontrib.bio.dicty.phenotypes.**genes**()
Return a set of all genes referenced in the Dictybase.

orangecontrib.bio.dicty.phenotypes.**phenotypes**()
Return a set of all phenotypes referenced in Dictybase.

orangecontrib.bio.dicty.phenotypes.**mutant_genes**(*mutant*)
Return a set of all genes referenced by a *mutant* in Dictybase.

orangecontrib.bio.dicty.phenotypes.**mutant_phenotypes**(*mutant*)
Return a set of all phenotypes referenced by a *mutant* in Dictybase.

orangecontrib.bio.dicty.phenotypes.**gene_mutants**()
Return a dictionary { gene: set(mutant_objects for mutant), ... }.

orangecontrib.bio.dicty.phenotypes.**phenotype_mutants**()
Return a dictionary { phenotype: set(mutant_objects for mutant), ... }.

Gene name matching (gene)

Genes can have multiple aliases. When we combine data from different sources, for example expression data with GO gene sets, we have to match gene aliases representing the same genes. All implemented matching methods are based on sets of gene aliases.

Gene matchers in this module match genes to a user-specified set of target gene names. For gene matching, initialize a gene matcher (*Matcher*), set the target gene names with *set_targets*, and then match with *match* or *umatch* functions. The following example (genematch1.py) matches gene names to NCBI gene IDs:

```
import orangecontrib.bio.gene

#matching targets are NCBI gene IDs
targets = orangecontrib.bio.gene.NCBIGeneInfo("Homo sapiens").keys()

gm = orangecontrib.bio.gene.GMNCBI("9606")
gm.set_targets(targets)

for gene in [ "cct7", "pls1", "gdi1", "nfkb2", "dlg7" ]:
    print('Gene ' + gene + ' is NCBI gene ' + str(gm.umatch(gene)))
```

Gene name matching

The base class for all the following gene matcher is *Matcher*.

class orangecontrib.bio.gene.**Matcher**
Matches an input gene to some target gene (set in advance).

explain(*gene*)
Return gene matches with explanations as lists of tuples: a list of matched target genes and the corresponding set of gene aliases.

match(*gene*)
Return a list of target gene aliases which share a set of aliases with the input gene (can be empty).

set_targets(*targets*)
Set input list of gene names (a list of strings) as target genes.

umatch(*gene*)
Return a single (unique) matching target gene or None, if there are no matches or multiple matches.

This modules provides the following gene matchers:

class orangecontrib.bio.gene.**MatcherAliasesKEGG**(*organism*, *ignore_case=True*)
Alias: GMKEGG.

class orangecontrib.bio.gene.**MatcherAliasesGO**(*organism*, *ignore_case=True*)
Alias: GMGO.

class orangecontrib.bio.gene.**MatcherAliasesDictyBase**(*ignore_case=True*)
Alias: GMDicty.

class orangecontrib.bio.gene.**MatcherAliasesNCBI**(*organism*, *ignore_case=True*)
Alias: GMNCBI.

class orangecontrib.bio.gene.**MatcherAliasesEnsembl**(*organism*, ***kwargs*)
A matcher for Ensemble ids. Alias: GMEnsemble.

class orangecontrib.bio.gene.**MatcherDirect**(*ignore_case=True*)
Directly match target names. Can ignore case. Alias: GMDirect.

Gene name matchers can be applied in sequence (until the first match) or combined (overlapping sets of gene aliases of multiple gene matchers are combined) with the *matcher* function.

orangecontrib.bio.gene.**matcher**(*matchers*, *direct=True*, *ignore_case=True*)
Builds a new matcher from a list of gene matchers. Apply matchers in the input list successively until a match is found. If an element of *matchers* is a list, combine matchers in the sublist by joining overlapping sets of aliases.

Parameters

- **matchers** (*list*) – Gene matchers.
- **direct** (*bool*) – If True, first try to match gene directly (a *MatcherDirect* is inserted in front of the gene matcher sequence).
- **ignore_case** (*bool*) – passed to the added direct matcher.

The following example tries to match input genes onto KEGG gene aliases (genematch2.py).

```
import orangecontrib.bio.kegg
import orangecontrib.bio.gene

targets = orangecontrib.bio.kegg.KEGGOrganism("9606").get_genes() #KEGG gene IDs

gmkegg = orangecontrib.bio.gene.GMKEGG("9606")
gmgo = orangecontrib.bio.gene.GMGO("9606")
gmkegggo = orangecontrib.bio.gene.matcher([[gmkegg, gmgo]], direct=False) #joined␣
↪matchers
```

```
gmkegg.set_targets(targets)
gmgo.set_targets(targets)
gmkegggo.set_targets(targets)

genes = [ "cct7", "pls1", "gdi1", "nfkb2", "a2a299" ]

print("%12s %12s %12s %12s" % ( "gene", "KEGG", "GO", "KEGG+GO" ))
for gene in genes:
    print("%12s %12s %12s %12s" % \
        (gene, gmkegg.umatch(gene), gmgo.umatch(gene), gmkegggo.umatch(gene)))
```

Results show that GO aliases can not match onto KEGG gene IDs. For the last gene only joined GO and KEGG aliases produce a match:

```
  gene         KEGG           GO      KEGG+GO
  cct7    hsa:10574         None    hsa:10574
  pls1     hsa:5357         None     hsa:5357
  gdi1     hsa:2664         None     hsa:2664
 nfkb2     hsa:4791         None     hsa:4791
a2a299         None         None     hsa:7052
```

The following example finds KEGG pathways with given genes (`genematch_path.py`).

```
import orangecontrib.bio.kegg
import orangecontrib.bio.gene

keggorg = orangecontrib.bio.kegg.KEGGOrganism("mmu")
kegg_genes = keggorg.get_genes()

query = [ "Fndc4", "Itgb8", "Cdc34", "Olfr1403" ]

gm = orangecontrib.bio.gene.GMKEGG("mmu") #use KEGG aliases for gene matching
gm.set_targets(kegg_genes) #set KEGG gene aliases as targets

for name in query:
    match = gm.umatch(name)
    if match:
        pwys = keggorg.get_pathways_by_genes([match])
        print(name + " is in")
        pathways = [ orangecontrib.bio.kegg.KEGGPathway(p).title for p in pwys ]
        if pathways:
            for a in pathways:
                print(' ' + a)
        else:
            print('  /')
```

Output:

```
Fndc4 is in
  /
Itgb8 is in
  PI3K-Akt signaling pathway
  Focal adhesion
  ECM-receptor interaction
  Cell adhesion molecules (CAMs)
  Regulation of actin cytoskeleton
  Hypertrophic cardiomyopathy (HCM)
  Arrhythmogenic right ventricular cardiomyopathy (ARVC)
```

```
  Dilated cardiomyopathy
Cdc34 is in
  Ubiquitin mediated proteolysis
  Herpes simplex infection
Olfr1403 is in
  Olfactory transduction
```

Translation of homologs (gene.homology)

This module is an interface to NCBI HomoloGene.

orangecontrib.bio.gene.homology.**all_genes**(*taxid*)
Return a set of all genes for organism taxid.

orangecontrib.bio.gene.homology.**homologs**(*genename*, *taxid*)
Return a list of homologs (taxid, genename) for a homolog group of gene (organism taxid).

orangecontrib.bio.gene.homology.**homolog**(*genename*, *taxid*, *homolotaxid*)
Return a homolog of genename (for taxid) in organism holomotaxid. If the homolog does not exist, return None.

orangecontrib.bio.gene.homology.**orthologs**(*genename*, *taxid*, *ortholog_taxid=None*)
Return all InParanoid orthologs of genename from organism with taxid. If ortholog_taxid is given limit to orthologs from that organism only.

orangecontrib.bio.gene.homology.**all_genes_inParanoid**(*taxid*)
Return a set of all genes for organism with taxid in the InParanoid database.

Examples

Mapping homologs from yeast to human:

```
import orangecontrib.bio.gene
import orangecontrib.bio.taxonomy
import Orange

data = Orange.data.Table("brown-selected")

geneinfo = orangecontrib.bio.gene.NCBIGeneInfo('4932')

genes = [str(ex["gene"]) for ex in data]

for gene in genes:
    mappedgene = orangecontrib.bio.gene.homology.homolog(geneinfo(gene).symbol, \
        '4932', '9606')
    print(gene, mappedgene)
```

Gene sets (geneset)

This module can load either gene sets distributed with Orange or custom gene sets in the GMT file format.

The available gene set collection can be listed with *list_all*.

collections loads gene sets. Gene sets provided with Orange are organized hierarchically. Although the GO hierarchy includes subsets, all of them can be loaded with (the organism here is a mouse):

```
orangecontrib.bio.geneset.collections((("GO",), "10090"))
```

To open multiple gene set collections at once, for example, KEGG and GO, try:

```
orangecontrib.bio.geneset.collections((("KEGG",), "10090"), (("GO",), "10090"))
```

You could also open a file with gene sets. The following line would open `specific.gmt` from the current working directory:

```
orangecontrib.bio.geneset.collections("specific.gmt")
```

The above examples combined:

```
orangecontrib.bio.geneset.collections((("KEGG",), "10090"), (("GO",), "10090"),
↪"specific.gmt")
```

Furthermore, all gene sets for a specific organism can be opened with an empty hierarchy:

```
orangecontrib.bio.geneset.collections((tuple(), "10090"))
```

Loading gene sets

orangecontrib.bio.geneset.**list_all**(*org=None*, *local=None*)

Return gene sets available in the local and ServerFiles repositories. It returns a list of tuples of (hierarchy, organism, available_locally)

Results can be filtered with the following parameters.

Parameters

- **org** (*str*) – Organism tax id.
- **local** (*bool*) – Available locally.

orangecontrib.bio.geneset.**collections**(**args*)

Load gene sets from various sources: GMT file, GO, KEGG, and others. Return an instance of *GeneSets*.

Each arguments specifies a gene set and can be either:

•a filename of a GMT file,

•a tuple (hierarchy, organism) (for example `(("KEGG",), "10090")`), or

•an instance of *GeneSets*

Supporting functionality

class orangecontrib.bio.geneset.**GeneSets**(*input=None*)

Bases: `set`

A collection of gene sets: contains *GeneSet* objects.

common_hierarchy()

Return a common hierarchy.

common_org()

Return a common organism.

hierarchies()
Return all hierarchies.

set_hierarchy(*hierarchy*)
Sets hierarchy for all gene sets.

split_by_hierarchy()
Split gene sets by hierarchies. Return a list of *GeneSets* objects.

to_odict()
Return gene sets in old dictionary format.

class orangecontrib.bio.geneset.**GeneSet**(*genes=[]*, *name=None*, *id=None*, *description=None*, *link=None*, *organism=None*, *hierarchy=None*, *pair=None*)
A single set of genes.

cname(*source=True*, *name=True*)
Return a gene set name with hiearchy.

description = None
Gene set description.

genes = None
A set of genes. Genes are strings.

hierarchy = None
Hierarchy should be formated as a tuple, for example `("GO", "biological_process")`

id = None
Short gene set ID.

link = None
Link to further information about this gene set.

name = None
Gene set name.

organism = None
Organism as a NCBI taxonomy ID.

to_odict(*source=True*, *name=True*)
For backward compatibility. Return a gene set as a tuple (id, list of genes).

orangecontrib.bio.geneset.**register**(*genesets*, *serverFiles=None*)
Registers given genesets locally. The gene set is registered by the common hierarchy or organism (None if organisms are different). :param GeneSets genesets: :param serverFiles: If *serverFiles* is an authenticated ServerFiles connection,

the input gene sets are uploaded to the repository.

NCBI's Gene Expression Omnibus interface (geo)

This module provides an interface to NCBI's Gene Expression Omnibus repository. It supports GEO DataSets query and retrieval.

In the following example *GDS.getdata* construct a data set with genes in rows and samples in columns. Notice that the annotation about each sample is retained in `.attributes`.

```
>>> import orangecontrib.bio.geo
>>> gds = orangecontrib.bio.geo.GDS("GDS1676")
>>> data = gds.getdata()
>>> len(data)
717
>>> data[200]
[1.407, 1.268, 1.969, 1.516, 1.697, 1.609, 0.760, 0.073], {"gene":'CD70'}
>>> data.domain.attributes[0]
Orange.feature.Continuous 'GSM63816'
>>> data.domain.attributes[0].attributes
{'dose': '20 U/ml IL-2', 'infection': 'acute ', 'time': '1 d'}
```

GDS classes

class orangecontrib.bio.geo.**GDSInfo**(*force_update=False*)

Retrieve infomation about GEO DataSets. The class accesses the Orange server file that either resides on the local computer or is automatically retrieved from Orange server. Calls to this class do not access any NCBI's servers.

Constructor returning the object with GEO DataSets information. If *force_update* is True, the constructor will download GEO DataSets information file (gds_info.pickled) from Orange server, otherwise it will first check the local copy.

An instance behaves like a dictionary: the keys are GEO DataSets IDs, and the dictionary values for is a dictionary providing various information about the particular data set.

An example with *GDSInfo*:

```
>>> import orangecontrib.bio.geo
>>> info = orangecontrib.bio.geo.GDSInfo()
>>> list(info.keys())[:5]
>>> ['GDS2526', 'GDS2524', 'GDS2525', 'GDS2522', 'GDS1618']
>>> info['GDS2526']['title']
'c-MYC depletion effect on carcinoma cell lines'
>>> info['GDS2526']['platform_organism']
'Homo sapiens'
```

class orangecontrib.bio.geo.**GDS**(*gdsname*, *verbose=False*, *force_download=False*)

Retrieval of a specific GEO DataSet as a `Orange.data.Table`.

Constructor returns the object that can retrieve GEO DataSet (samples and gene expressions). It first checks a local cache directory if the particular data file is loaded locally, else it downloads it from NCBI's GEO FTP site.

Parameters

- **gdsname** – An NCBI's ID for the data set in the form "GDSn" where "n" is a GDS ID number.
- **force_download** – Force the download.

getdata(*report_genes=True*, *merge_function=<function spots_mean>*, *sample_type=None*, *transpose=False*, *remove_unknown=None*)

Returns the GEO DataSet as an `Orange.data.Table`.

Parameters

- **report_genes** – Microarray spots reported in the GEO data set can either be merged according to their gene ids (if True) or can be left as spots.

- **transpose** – The output table can have spots/genes in rows and samples in columns (False, default) or samples in rows and spots/genes in columns (True).
- **sample_type** – the type of annotation, or (if `transpose` is True) the type of class labels to be included in the data set. The entire annotation of samples will be included either in the class value or in the `.attributes` field of each data set attribute.
- **remove_unknown** – Remove spots with sample profiles that include unknown values. They are removed if the proportion of samples with unknown values is above the threshold set by `remove_unknown`. If None, nothing is removed.

sample_annotations(*sample_type=None*)
Return a dictionary with sample annotation.

sample_to_class(*sample_type=None*)
Return class values for GDS samples.

sample_types()
Return a set of sample types.

Examples

The following script prints out information about a specific data set. It does not download the data set, just uses the (local) GEO data sets information file (`geo_gds1.py`).

```
import orangecontrib.bio.geo
import textwrap

gdsinfo = orangecontrib.bio.geo.GDSInfo()
gds = gdsinfo["GDS10"]

print("ID:")
print(gds["dataset_id"])
print("Features: ")
print(gds["feature_count"])
print("Genes:")
print(gds["gene_count"])
print("Organism:")
print(gds["platform_organism"])
print("PubMed ID:")
print(gds["pubmed_id"])
print("Sample types:")
for sampletype in set([sinfo["type"] for sinfo in gds["subsets"]]):
    ss = [sinfo["description"] for sinfo in gds["subsets"] if sinfo["type
↪"]==sampletype]
    print("  %s (%s)" % (sampletype, ", ".join(ss)))
print("")
print("Description:")
print("\n".join(textwrap.wrap(gds["description"], 70)))
```

The output of this script is:

```
ID:
GDS10
Features:
39114
Genes:
29822
```

```
Organism:
Mus musculus
PubMed ID:
11827943
Sample types:
  tissue (spleen, thymus)
  disease state (diabetic, diabetic-resistant, nondiabetic)
  strain (NOD, Idd3, Idd5, Idd3+Idd5, Idd9, B10.H2g7, B10.H2g7 Idd3)

Description:
Examination of spleen and thymus of type 1 diabetes nonobese diabetic
(NOD) mouse, four NOD-derived diabetes-resistant congenic strains and
two nondiabetic control strains.
```

Samples in GEO data sets belong to sample subsets, which in turn belong to specific types. The above GDS10 has three sample types, of which the subsets for the tissue type are spleen and thymus. For supervised data mining it would be useful to find out which data sets provide enough samples for each label. It is (semantically) convenient to perform classification within sample subsets of the same type. The following script goes through all data sets and finds those with enough samples within each of the subsets for a specific type. The function `valid` determines which subset types (if any) satisfy our criteria (`geo_gds5.py`).

```
import orangecontrib.bio.geo

def valid(info, n=40):
    """Return a set of subset types containing more than n samples in every subset"""
    invalid = set()
    subsets = set([sinfo["type"] for sinfo in info["subsets"]])
    for sampleinfo in info["subsets"]:
        if len(sampleinfo["sample_id"]) < n:
            invalid.add(sampleinfo["type"])
    return subsets.difference(invalid)

def report(stypes, info):
    """Pretty-print GDS and valid susbset types"""
    for id, sts in stypes:
        print(id)
        for st in sts:
            gds = info[id]
            print("  %s:" % st + \
                  ", ".join(["%s/%d" % (sinfo["description"], len(sinfo["sample_id
↪"])) \
                  for sinfo in gds["subsets"] if sinfo["type"]==st]))

gdsinfo = orangecontrib.bio.geo.GDSInfo()
valid_subset_types = [(id, valid(info)) for id, info in sorted(gdsinfo.items()) if␣
↪valid(info)]
report(valid_subset_types, gdsinfo)

print('datasets = ' + str(len(valid_subset_types)))
print('type subsets = ' + str(sum(len(b) for _,b in valid_subset_types)))
```

The requested number of samples, `n=40`, seems to be a quite a stringent criteria met - at the time of writing this - by 40 data sets with 48 sample subsets. The output starts with:

```
GDS1292
  tissue:raphe magnus/40, somatomotor cortex/43
GDS1293
```

```
  tissue:raphe magnus/40, somatomotor cortex/41
GDS1412
  protocol:no treatment/47, hormone replacement therapy/42
GDS1490
  other:non-neural/50, neural/100
GDS1611
  genotype/variation:wild type/48, upf1 null mutant/48
GDS2373
  gender:male/82, female/48
GDS2808
  protocol:training set/44, validation set/50
```

Let us now pick data set GDS2960 and see if we can predict the disease state. We will use logistic regression, and within 10-fold cross validation measure AUC, the area under ROC. AUC is the probability of correctly distinguishing the two classes, (e.g., the disease and control). From (`geo_gds6.py`):

```
import Orange
import orangecontrib.bio.geo

gds = orangecontrib.bio.geo.GDS("GDS2960")
data = gds.getdata(sample_type="disease state", transpose=True, report_genes=True)
print("Samples: %d, Genes: %d" % (len(data), len(data.domain.attributes)))

if Orange.__version__ > "3":
    learners = [ Orange.classification.LogisticRegressionLearner() ]
    results = Orange.evaluation.testing.CrossValidation(data, learners, k=10)
else:
    learners = [ Orange.classification.logreg.LibLinearLogRegLearner() ]
    results = Orange.evaluation.testing.cross_validation(learners, data, folds=10)


print("AUC = %.3f" % Orange.evaluation.scoring.AUC(results)[0])
```

The output of this script is:

```
Samples: 101, Genes: 4068
AUC = 0.983
```

The AUC for this data set is very high, indicating that using these gene expression data it is almost trivial to separate the two classes.

Gene Ontology (go)

Provides access to Gene Ontology and its gene annotations.

class `orangecontrib.bio.go.`**`Ontology`**(*filename=None*, *progress_callback=None*, *rev=None*)

`Ontology` is the class representing a gene ontology.

Parameters

- **`filename`** (*`str`*) – A filename of an .obo formated file.
- **`progress_callback`** – Optional *float* -> *None* function.
- **`rev`** (*`str`*) – An CVS revision specifier (see GO web CVS interface)

Example:

```
>>> # Load the current ontology (downloading it if necessary)
>>> ontology = Ontology()
>>> # Load the ontology at the specified CVS revision.
>>> ontology = Ontology(rev="5.2092")
```

Ontology supports a subset of the Mapping protocol:

```
>>> term_ids = list(ontology)
>>> term = ontology[term_ids[0]]
```

__contains__(*termid*)
Return *True* if a term with *termid* is present in the ontology.

__getitem__(*termid*)
Return a *Term* object with *termid*.

Parameters **term** (*str*) – An id of a 'Term' in the ontology.

Return type *Term*

__iter__()
Iterate over all term ids in ontology.

__len__()
Return number of terms in ontology.

defined_slims_subsets()
Return a list of defined subsets in the ontology.

Return type list-of-str

extract_sub_graph(*terms*)
Return all sub terms of *terms*.

Parameters **terms** (list) – A list of term IDs.

extract_super_graph(*terms*)
Return all super terms of *terms* up to the most general one.

Parameters **terms** (list) – A list of term IDs.

named_slims_subset(*subset*)
Return all term IDs in a named *subset*.

Parameters **subset** (*str*) – A string naming a subset in the ontology.

Return type list-of-str

See also:

defined_slims_subsets()

set_slims_subset(*subset*)
Set the *slims_subset* term subset to *subset*.

Parameters **subset** (*set*) – A subset of GO term IDs.

subset may also be a string, in which case the call is equivalent to `ont.set_slims_subsets(ont.named_slims_subset(subset))`

slims_for_term(*term*)
Return a list of slim term IDs for *term*.

This is a list of *most specific* slim terms to which *term* belongs.

Parameters **term** (*str*) – Term ID.

class orangecontrib.bio.go.**Term**(*stanza=None*, *ontology=None*)

id
The term id.

namespace
The namespace of the term.

def_
The term definition (Note the use of trailing underscore to avoid conflict with a python keyword).

is_a
List of term ids this term is a subterm of (parent terms).

related
List of (rel_type, term_id) tuples with rel_type specifying the relationship type with term_id.

class orangecontrib.bio.go.**Annotations**(*filename_or_organism=None*, *ontology=None*, *genematcher=None*, *progress_callback=None*, *rev=None*)

Annotations object holds the annotations.

Parameters

- **filename_or_org** (*str*) – A filename of a GAF formated annotations file (e.g. gene_annotations.goa_human) or an organism specifier (e.g. 'goa_human' or '9606'). In the later case the annotations for that organism will be loaded.
- **ontology** (*Ontology*) – *Ontology* object for annotations
- **rev** (*str*) – An optional CVS revision string. If the *filename_or_org* is given an organism code the annotations will be retrieved for that revision (see GO web CVS)

gene_annotations = None
A dictionary mapping a gene name (DB_Object_Symbol) to a set of all annotations of that gene.

term_anotations = None
A dictionary mapping a GO term id to a set of annotations that are directly annotated to that term

annotations = None
A list of all AnnotationRecords instances.

ontology
Ontology object for annotations.

add_annotation(*a*)
Add a single AnotationRecord instance to this object.

get_gene_names_translator(*genes*)
Return a dictionary mapping canonical names (DB_Object_Symbol) to *genes*.

get_all_annotations(*id*)
Return a set of all annotations (instances of *AnnotationRecord*) for GO term *id* and all it's subterms.

Parameters **id** (*str*) – GO term id

get_all_genes(*id*, *evidence_codes=None*)
Return a list of genes annotated by specified *evidence_codes* to GO term 'id' and all it's subterms."

Parameters

- **id** (*str*) – GO term id

- **evidence_codes** (*list-of-strings*) – List of evidence codes to consider when matching annotations to terms.

get_enriched_terms(*genes*, *reference=None*, *evidence_codes=None*, *slims_only=False*, *aspect=None*, *prob=<orangecontrib.bio.utils.stats.Binomial object>*, *use_fdr=True*, *progress_callback=None*)

Return a dictionary of enriched terms, with tuples of (list_of_genes, p_value, reference_count) for items and term ids as keys. P-Values are FDR adjusted if use_fdr is True (default).

Parameters

- **genes** – List of genes
- **reference** – List of genes (if None all genes included in the annotations will be used).
- **evidence_codes** – List of evidence codes to consider.
- **slims_only** – If *True* return only slim terms.
- **aspect** – Which aspects to use. Use all by default. "P", "F", "C" or a set containing these elements.

get_annotated_terms(*genes*, *direct_annotation_only=False*, *evidence_codes=None*, *progress_callback=None*)

Return all terms that are annotated by genes with evidence_codes.

add(*line*)

Add one annotation

append(*line*)

Add one annotation

extend(*lines*)

Add multiple annotations

class orangecontrib.bio.go.**AnnotationRecord**

An annotation record mapping a gene to a term.

See http://geneontology.org/GO.format.gaf-2_0.shtml for description if individual fields.

Annotation_Extension

Alias for field number 15

Aspect

Alias for field number 8

Assigned_By

Alias for field number 14

DB

Alias for field number 0

DB_Object_ID

Alias for field number 1

DB_Object_Name

Alias for field number 9

DB_Object_Symbol

Alias for field number 2

DB_Object_Synonym

Alias for field number 10

DB_Object_Type
Alias for field number 11

DB_Reference
Alias for field number 5

Date
Alias for field number 13

Evidence_Code
Alias for field number 6

GO_ID
Alias for field number 4

Gene_Product_Form_ID
Alias for field number 16

Qualifier
Alias for field number 3

Taxon
Alias for field number 12

With_From
Alias for field number 7

classmethod from_string(*string*)
Create an instance from a line in a annotations (GAF 2.0 format) file.

Example

Load the ontology and print out some terms:

```
from orangecontrib.bio import go
ontology = go.Ontology()
term = ontology["GO:0097194"] # execution phase of apoptosis

# print a term
print(term)

# access fields by name
print(term.id, term.name)
# note the use of underscore due to a conflict with a python def keyword
print(term.def_)
```

Searching the annotation (part of `code/go_gene_annotations.py`)

```
from orangecontrib.bio import go

ontology = go.Ontology()

# Print names and definitions of all terms with "apoptosis" in the name
apoptosis = [term for term in ontology.terms.values()
             if "apoptosis" in term.name.lower()]
for term in apoptosis:
    print(term.name + term.id)
    print(term.def_)

# Load annotations for yeast.
```

```
annotations = go.Annotations("sgd", ontology=ontology)

res = annotations.get_enriched_terms(["YGR270W", "YIL075C", "YDL007W"])

gene = annotations.alias_mapper["YIL075C"]
print(gene + " (YIL075C) directly annotated to the following terms:")
for a in annotations.gene_annotations[gene]:
    print(ontology[a.GO_ID].name + " with evidence code " + a.Evidence_Code)

# Get all genes annotated to the same terms as YIL075C
ids = set([a.GO_ID for a in annotations.gene_annotations[gene]])
for termid in ids:
    ants = annotations.get_all_annotations(termid)
    genes = set([a.DB_Object_Symbol for a in ants])
    print(", ".join(genes) +" annotated to " + termid + " " + ontology[termid].name)
```

Term enrichment (part of `code/go_enrichment.py`)

```
res = annotations.get_enriched_terms(["YGR270W", "YIL075C", "YDL007W"])
print("Enriched terms:")
for go_id, (genes, p_value, ref) in res.items():
    if p_value < 0.05:
        print(ontology[go_id].name + " with p-value: %.4f " % p_value + ", ".
↪join(genes))

# And again for slims
ontology.set_slims_subset("goslim_yeast")

res = annotations.get_enriched_terms(["YGR270W", "YIL075C", "YDL007W"],
                                     slims_only=True)
print("Enriched slim terms:")
for go_id, (genes, p_value, _) in res.items():
    if p_value < 0.05:
        print(ontology[go_id].name + " with p-value: %.4f " % p_value + ", ".
↪join(genes))
```

Gene Set Enrichment Analysis (GSEA, `gsea`)

Gene Set Enrichment Analysis (GSEA) *[GSEA]* aims to identify enriched gene sets given gene expression data for multiple samples with their phenotypes.

Examples: gene expression data

The following examples use a gene expression data set from the GEO database. We show the same analysis on two formats of data.

With samples as instances (in rows):

```
import Orange
from orangecontrib.bio import dicty, geneset, gsea, gene, geo

gds = geo.GDS("GDS10")
data = gds.getdata(transpose=True)
```

```
matcher = gene.matcher([gene.GMKEGG("Homo sapiens")])
genesets = geneset.collections((("KEGG",), "Homo sapiens"))

#the number of permutations (n) should be much higher
res = gsea.run(data, gene_sets=genesets, matcher=matcher,
    min_part=0.05, permutation="phenotype", n=10,
    phen_desc=data.domain["tissue"], gene_desc=True)

print
print "GSEA results (descriptor: tissue)"
print "%-40s %6s %6s %6s %7s" % ("LABEL", "NES", "FDR", "SIZE", "MATCHED")
for gs, resu in sorted(res.items(), key=lambda x: x[1]["fdr"])[:10]:
    print "%-40s %6.3f %6.3f %6d %7d" % (gs.name[:30],
        resu["nes"], resu["fdr"], resu["size"], resu["matched_size"])
```

With samples as features (in columns):

```
import Orange
from orangecontrib.bio import dicty, geneset, gsea, gene, geo

gds = geo.GDS("GDS10")
data = gds.getdata()

matcher = gene.matcher([gene.GMKEGG("Homo sapiens")])
genesets = geneset.collections((("KEGG",), "Homo sapiens"))

#the number of permutations (n) should be much higher
res = gsea.run(data, gene_sets=genesets, matcher=matcher,
    min_part=0.05, permutation="phenotype", n=10,
    phen_desc="tissue", gene_desc="gene")

print
print "GSEA results (descriptor: tissue)"
print "%-40s %6s %6s %6s %7s" % ("LABEL", "NES", "FDR", "SIZE", "MATCHED")
for gs, resu in sorted(res.items(), key=lambda x: x[1]["fdr"])[:10]:
    print "%-40s %6.3f %6.3f %6d %7d" % (gs.name[:30],
        resu["nes"], resu["fdr"], resu["size"], resu["matched_size"])
```

Both scripts output:

```
GSEA results (descriptor: tissue)
LABEL                                    NES    FDR   SIZE MATCHED
Porphyrin and chlorophyll meta        -1.817  0.000     43      23
Staphylococcus aureus infectio        -1.998  0.000     59      28
Non-homologous end-joining             1.812  0.000     13      12
Fanconi anemia pathway                 1.911  0.000     53      27
Cell cycle                             1.777  0.000    124     106
Glycine, serine and threonine         -2.068  0.000     39      29
HIF-1 signaling pathway               -1.746  0.000    106      90
Ether lipid metabolism                -1.788  0.000     42      27
Fc epsilon RI signaling pathwa        -1.743  0.000     70      53
B cell receptor signaling path        -1.782  0.000     72      62
```

Example: our own gene sets

We present a simple example on iris data set. Because data set is not a gene expression data set, we had to specify our own sets of features that belong together.

```
import Orange
import orangecontrib.bio.gsea
import orangecontrib.bio.gene
import orangecontrib.bio.geneset

data = Orange.data.Table("iris")

gen1 = orangecontrib.bio.geneset.GeneSets(dict([
    ("sepal", ["sepal length", "sepal width"]),
    ("petal", ["petal length", "petal width", "petal color"])
    ]))

res = orangecontrib.bio.gsea.run(data, gene_sets=gen1, matcher=orangecontrib.bio.gene.
↪matcher([]), min_size=2)
print "%5s  %6s %6s %s" % ("LABEL", "NES", "P-VAL", "GENES")
for gs,resu in res.items():
    print "%5s  %6.3f %6.3f %s" % (gs.id, resu["nes"], resu["p"], str(resu["genes"]))
```

The output:

```
LABEL     NES  P-VAL GENES
sepal   1.087  0.630 ['sepal width', 'sepal length']
petal  -1.117  0.771 ['petal width', 'petal length']
```

Example: directly passing correlation data

GSEA can also directly use correlation data between individual genes and a phenotype. If (1) input data with only one example (attribute names are gene names) or (2) there is only one continuous feature in the given data set (gene names are in the first `Orange.feature.String`. The following outputs ten pathways with smallest p-values.

```
import Orange
from orangecontrib.bio import dicty, geneset, gsea, gene

dbc = dicty.DictyExpress()
data = dbc.get_data(sample='pkaC-', time="8")

#select the first chip (the first attribute)
data = data.translate([data.domain.attributes[0]], True)

matcher = gene.matcher([[gene.GMKEGG("dicty"), gene.GMDicty()]])
genesets =  geneset.collections((("KEGG",), "dicty"))

res = gsea.direct(data, matcher=matcher, min_part=0.05,
    gene_sets=genesets)

print "%-40s %6s %6s %6s %7s" % ("LABEL", "NES", "P-VAL", "SIZE", "MATCHED")
for name,resu in sorted(res.items()[:10], key=lambda x: x[1]["p"]):
    print "%-40s %6.3f %6.3f %6d %7d" % (name.name[:35], resu["nes"],
        resu["p"], resu["size"], resu["matched_size"])
```

The output:

```
LABEL                                    NES  P-VAL   SIZE MATCHED
Biosynthesis of amino acids            1.407  0.056     58      40
beta-Alanine metabolism                1.165  0.232     13      10
Taurine and hypotaurine metabolism     1.160  0.413      4       3
Porphyrin and chlorophyll metabolis   -0.990  0.517     14       5
Valine, leucine and isoleucine degr    0.897  0.585     29      21
Ether lipid metabolism                 0.713  0.857     10       6
Biosynthesis of unsaturated fatty a    0.659  0.922     10       6
Protein processing in endoplasmic r    0.647  0.941     71      40
RNA polymerase                         0.550  0.943     24       7
Glycosylphosphatidylinositol(GPI)-a   -0.540  0.946     19       4
```

KEGG - Kyoto Encyclopedia of Genes and Genomes (kegg)

KEGG - Kyoto Encyclopedia of Genes and Genomes

`kegg` is a python module for accessing KEGG (Kyoto Encyclopedia of Genes and Genomes) using its web services.

Note: This module requires slumber and requests packages.

```
>>> # Create a KEGG Genes database interface
>>> genome = KEGGGenome()
>>> # List all available entry ids
>>> keys = list(genome.keys())
>>> print(keys[0])
T01001
>>> # Retrieve the entry for the key.
>>> entry = genome[keys[0]]
>>> print(entry.entry_key)
T01001
>>> print(entry.definition)
Homo sapiens (human)
>>> print(entry)
ENTRY       T01001            Complete  Genome
NAME        hsa, HUMAN, 9606
DEFINITION  Homo sapiens (human)
...
```

The *Organism* class can be a convenient starting point for organism specific databases.

```
>>> organism = Organism("Homo sapiens")  # searches for the organism by name
>>> print(organism.org_code)  # prints the KEGG organism code
hsa
>>> genes = organism.genes  # get the genes database for the organism
>>> gene_ids = list(genes.keys()) # KEGG gene identifiers
>>> entry = genes["hsa:672"]
>>> print(entry.definition)
(RefSeq) BRCA1, DNA repair associated
>>> # print the entry in DBGET database format.
>>> print(entry)
ENTRY       672               CDS       T01001
NAME        BRCA1, BRCAI, BRCC1, BROVCA1, FANCS, IRIS, PNCA4, PPP1R53, PSCP, RNF53
DEFINITION  ...
```

class orangecontrib.bio.kegg.**Organism**(*org*, *genematcher=None*)

A convenience class for retrieving information regarding an organism in the KEGG Genes database.

Parameters **org** (*str*) – KEGG organism code (e.g. "hsa", "sce"). Can also be a descriptive name (e.g. 'yeast', "homo sapiens") in which case the organism code will be searched for by using KEGG *find* api.

See also:

organism_name_search() Search KEGG for an organism code

org
KEGG organism code.

genes
An *Genes* database instance for this organism.

gene_aliases()
Return a list of sets of equal genes (synonyms) in KEGG for this organism.

Note: This only includes 'ncbi-geneid' and 'ncbi-proteinid' records from the KEGG Genes DBLINKS entries.

---

pathways(*with_ids=None*)
Return a list of all pathways for this organism.

list_pathways()
List all pathways for this organism.

get_enriched_pathways(*genes*, *reference=None*, *prob=<orangecontrib.bio.utils.stats.Binomial object>*, *callback=None*)
Return a dictionary with enriched pathways ids as keys and (list_of_genes, p_value, num_of_reference_genes) tuples as items.

get_pathways_by_genes(*gene_ids*)
Pathways that include all genes in gene_ids.

get_unique_gene_ids(*genes*, *case_sensitive=True*)
Return a tuple with three elements. The first is a dictionary mapping from unique geneids to gene names in genes, the second is a list of conflicting gene names and the third is a list of unknown genes.

orangecontrib.bio.kegg.**KEGGOrganism**
alias of *Organism*

orangecontrib.bio.kegg.**organism_name_search**(*name*)
Search for a organism by *name* and return it's KEGG organism code.

orangecontrib.bio.kegg.**pathways**(*org*)
Return a list of all KEGG pathways for an KEGG organism code *org*.

orangecontrib.bio.kegg.**from_taxid**(*taxid*)
Return a KEGG organism code for a an NCBI Taxonomy id string *taxid*.

orangecontrib.bio.kegg.**to_taxid**(*name*)
Return a NCBI Taxonomy id for a given KEGG Organism name

DBEntry (entry)

The *entry.DBEntry* represents a DBGET databas entry. The individual KEGG Database interfaces below provide their own specialization for this base class.

class orangecontrib.bio.kegg.entry.**DBEntry**(*text=None*)
Bases: object

A DBGET entry object.

entry_key
Primary entry key used for identifying the entry.

parse(*text*)
Parse *text* string containing a formated DBGET entry.

format(*section_indent=12*)
Return a DBGET formated string representation.

KEGG Databases interface (databases)

class orangecontrib.bio.kegg.databases.**DBDataBase**(***kwargs*)
Bases: object

Base class for a DBGET database interface.

ENTRY_TYPE
ENTRY_TYPE constructor (a *DBEntry* subclass). This should be redefined in subclasses.

alias of DBEntry

DB = None
A database name/abbreviation (e.g. 'pathway'). Needs to be set in a subclass or object instance's constructor before calling the base. __init__

iterkeys()
Return an iterator over the *keys*.

iteritems()
Return an iterator over the *items*.

itervalues()
Return an iterator over all DBDataBase.ENTRY_TYPE instances.

keys()
Return an iterator over all database keys. These are unique KEGG identifiers that can be used to query the database.

values()
Return an iterator over all DBDataBase.ENTRY_TYPE instances.

items()
Return an iterator over all (key, DBDataBase.ENTRY_TYPE) tuples.

get(*key*, *default=None*)
Return an DBDataBase.ENTRY_TYPE instance for the *key*. Raises KeyError if not found.

get_text(*key*)
Return the database entry for *key* as plain text.

get_entry(*key*)
Return the database entry for *key* as an instance of *ENTRY_TYPE*.

find(*name*)
Find *name* using kegg *find* api.

pre_cache(*keys=None*, *batch_size=10*, *progress_callback=None*)
Retrieve all the entries for *keys* and cache them locally for faster subsequent retrieval. If *keys* is `None` then all entries will be retrieved.

batch_get(*keys*)
Batch retrieve all entries for keys. This can be significantly faster then getting each entry separately especially if entries are not yet cached.

class orangecontrib.bio.kegg.databases.**GenomeEntry**(*text*)
Bases: *orangecontrib.bio.kegg.entry.DBEntry*

Entry for a KEGG Genome database.

organism_code
A three or four letter KEGG organism code (e.g. 'hsa', 'sce', ...)

taxid
Organism NCBI taxonomy id.

annotation
ANNOTATION

chromosome
CHROMOSOME

comment
COMMENT

data_source
DATA_SOURCE

definition
DEFINITION

disease
DISEASE

entry
ENTRY

keywords
KEYWORDS

name
NAME

original_db
ORIGINAL_DB

plasmid
PLASMID

reference
REFERENCE

statistics
STATISTICS

taxonomy
TAXONOMY

class orangecontrib.bio.kegg.databases.**Genome**
Bases: *orangecontrib.bio.kegg.databases.DBDataBase*

An interface to the A KEGG GENOME database.

ENTRY_TYPE
alias of GenomeEntry

org_code_to_entry_key(*code*)
Map an organism code ('hsa', 'sce', ...) to the corresponding kegg identifier (T + 5 digit number).

search(*string*, *relevance=False*)
Search the genome database for string using bfind.

class orangecontrib.bio.kegg.databases.**GeneEntry**(*text=None*)
Bases: *orangecontrib.bio.kegg.entry.DBEntry*

aaseq
AASEQ

class_
CLASS

dblinks
DBLINKS

definition
DEFINITION

disease
DISEASE

drug_target
DRUG_TARGET

entry
ENTRY

module
MODULE

motif
MOTIF

name
NAME

ntseq
NTSEQ

organism
ORGANISM

orthology
ORTHOLOGY

pathway
PATHWAY

position
POSITION

structure
STRUCTURE

class orangecontrib.bio.kegg.databases.**Genes**(*org_code*)
Bases: *orangecontrib.bio.kegg.databases.DBDataBase*

Interface to the KEGG Genes database.

Parameters **org_code** (*str*) – KEGG organism code (e.g. 'hsa').

class orangecontrib.bio.kegg.databases.**CompoundEntry**(*text=None*)
Bases: *orangecontrib.bio.kegg.entry.DBEntry*

atom
ATOM

bond
BOND

brite
BRITE

comment
COMMENT

dblinks
DBLINKS

entry
ENTRY

enzyme
ENZYME

exact_mass
EXACT_MASS

formula
FORMULA

mol_weight
MOL_WEIGHT

name
NAME

pathway
PATHWAY

reaction
REACTION

reference
REFERENCE

remark
REMARK

class orangecontrib.bio.kegg.databases.**Compound**
Bases: *orangecontrib.bio.kegg.databases.DBDataBase*

class orangecontrib.bio.kegg.databases.**ReactionEntry**(*text=None*)
Bases: *orangecontrib.bio.kegg.entry.DBEntry*

definition
DEFINITION

entry
ENTRY

enzyme
ENZYME

equation
EQUATION

name
NAME

class orangecontrib.bio.kegg.databases.**Reaction**
Bases: *orangecontrib.bio.kegg.databases.DBDataBase*

class orangecontrib.bio.kegg.databases.**EnzymeEntry**(*text=None*)
Bases: *orangecontrib.bio.kegg.entry.DBEntry*

all_reac
ALL_REAC

class_
CLASS

comment
COMMENT

dblinks
DBLINKS

entry
ENTRY

genes
GENES

name
NAME

orthology
ORTHOLOGY

pathway
PATHWAY

product
PRODUCT

reaction
REACTION

reference
REFERENCE

substrate
SUBSTRATE

sysname
SYSNAME

class orangecontrib.bio.kegg.databases.**Enzyme**
Bases: *orangecontrib.bio.kegg.databases.DBDataBase*

class orangecontrib.bio.kegg.databases.**PathwayEntry**(*text=None*)
Bases: *orangecontrib.bio.kegg.entry.DBEntry*

class_
CLASS

compound
COMPOUND

dblinks
DBLINKS

description
DESCRIPTION

disease
DISEASE

drug
DRUG

entry
ENTRY

enzyme
ENZYME

ko_pathway
KO_PATHWAY

module
MODULE

name
NAME

organism
ORGANISM

pathway_map
PATHWAY_MAP

reference
REFERENCE

rel_pathway
REL_PATHWAY

class orangecontrib.bio.kegg.databases.**Pathway**(*prefix='map'*)
Bases: *orangecontrib.bio.kegg.databases.DBDataBase*

KEGG Pathway database

Parameters **prefix** (*str*) – KEGG Organism code ('hsa', ...) or 'map', 'ko', 'ec' or 'rn'

KEGG Pathway (pathway)

class orangecontrib.bio.kegg.pathway.**Pathway**(*pathway_id*, *local_cache=None*, *connection=None*)
Bases: object

Class representing a KEGG Pathway (parsed from a "kgml" file)

Parameters **pathway_id** (*str*) – A KEGG pathway id (e.g. 'path:hsa05130')

name
Pathway name/id (e.g. "path:hsa05130")

org
Pathway organism code (e.g. 'hsa')

number
Pathway number as a string (e.g. '05130')

title
Pathway title string.

image
URL of the pathway image.

link
URL to a pathway on the KEGG web site.

get_image()
Return an local filesystem path to an image of the pathway. The image will be downloaded if not already cached.

classmethod **list**(*organism*)
List all pathways for KEGG organism code *organism*.

Utilities

class orangecontrib.bio.kegg.entry.parser.**DBGETEntryParser**
A DBGET entry parser (inspired by `xml.dom.pulldom`).

```
>>> stream = StringIO(
...     "ENTRY foo\n"
...     "NAME  foo's name\n"
...     "  BAR A subsection of 'NAME'\n"
... )
>>> parser = DBGETEntryParser()
>>> for event, title, contents_part in parser.parse(stream):
...    print(parser.EVENTS[event], title, repr(contents_part))
...
ENTRY_START None None
SECTION_START ENTRY 'foo\n'
SECTION_END ENTRY None
SECTION_START NAME "foo's name\n"
SUBSECTION_START BAR "A subsection of 'NAME'\n"
SUBSECTION_END BAR None
SECTION_END NAME None
ENTRY_END None None
```

ENTRY_END = 1
Entry end event

ENTRY_START = 0
Entry start events

SECTION_END = 3
Section end event

SECTION_START = 2
Section start event

SUBSECTION_END = 5
Subsection end event

SUBSECTION_START = 4
Subsection start event

TEXT = 6
Text element event

OMIM: Online Mendelian Inheritance in Man (omim)

omim provides an interface to Online Mendelian Inheritance in Man database. For now it only supports mapping genes to diseases.

orangecontrib.bio.omim.**diseases**()
Return all disease descriptors.

orangecontrib.bio.omim.**genes**()
Return a set of all genes referenced in OMIM.

orangecontrib.bio.omim.**disease_genes**(*disease*)
Return a set of all genes referenced by disease in OMIM.

orangecontrib.bio.omim.**gene_diseases**()
Return a dictionary {gene: set(disease_objects for gene), ...}.

The following example creates a network of connected diseases and save it in Pajek .net format sets information file (omim_disease_network.py).

```
import orangecontrib.bio.omim

diseases = orangecontrib.bio.omim.diseases()
genes = [orangecontrib.bio.omim.disease_genes(disease) for disease in diseases]

vertices = []
edges = []
for i in range(len(diseases)):
    vertices.append('%i "%s"\n' % (i + 1, diseases[i].name))
    for j in range(i + 1, len(diseases)):
        intersection = set(genes[i]).intersection(genes[j])
        if intersection:
            edges.append('%i %i %i l "%s"\n' %(i + 1, j + 1, len(intersection), ",".
↪join(sorted(intersection))))

file = open("disease.net", "wb")
file.write("*Vertices %i 2\n" % len(vertices))
file.writelines(vertices)
file.write("*Edges\n")
file.writelines(edges)
```

OBO Ontology (ontology)

This module provides an interface for parsing, creating and manipulating of OBO ontologies.

Construct an ontology from scratch with custom terms

```
>>> term = OBOObject("Term", id="foo:bar", name="Foo bar")
>>> print(term)
[Term]
id: foo:bar
name: Foo bar

>>> ontology = OBOOntology()
>>> ontology.add_object(term)
>>> ontology.add_header_tag("created-by", "ales") # add a header tag
>>> from six import StringIO
>>> buffer = StringIO()
>>> ontology.write(buffer) # Save the ontology to a file like object
>>> print(buffer.getvalue()) # Print the contents of the buffer
created-by: ales

[Term]
id: foo:bar
name: Foo bar
```

To load an ontology from a file, pass the file or filename to the *OBOOntology* constructor or call its load method

```
>>> _ = buffer.seek(0) # rewind
>>> ontology = OBOOntology(buffer)
>>> # Or equivalently
>>> _ = buffer.seek(0) # rewind
>>> ontology = OBOOntology()
>>> ontology.load(buffer)
```

See the definition of the .obo file format.

class orangecontrib.bio.ontology.**OBOOntology**(*file=None*)

Bases: object

An class representing an OBO ontology.

Parameters **file** (*file-like*) – A optional file like object describing the ontology in obo format.

add_object(*obj*)

Add an *OBOObject* instance to this ontology.

add_header_tag(*tag*, *value*)

Add header tag, value pair to this ontology.

load(*file*, *progress_callback=None*)

Load terms from a file.

Parameters

- **file** (*file-like*) – A file-like like object describing the ontology in obo format.
- **progress_callback** (*function*) – An optional function callback to report on the progress.

write(*stream*)

Write the contents of the ontology to a *file* in .obo format.

Parameters **file** (*file-like*) – A file like object.

update(*other*)
Update this ontology with the terms from *other*.

term(*id*)
Return the *OBOObject* associated with this id.

Parameters **id** (*str*) – Term id string.

terms()
Return all *Term* instances in the ontology.

term_by_name(*name*)
Return the term with name *name*.

typedefs()
Return all *Typedef* instances in the ontology.

instances()
Return all *Instance* instances in the ontology.

root_terms()
Return all root terms (terms without any parents).

related_terms(*term*)
Return a list of (*rel_type*, *term_id*) tuples where *rel_type* is relationship type (e.g. 'is_a', 'has_part', ...) and *term_id* is the id of the term in the relationship.

edge_types()
Return a list of all edge types in the ontology.

parent_edges(*term*)
Return a list of (rel_type, parent_term) tuples.

child_edges(*term*)
Return a list of (rel_type, source_term) tuples.

super_terms(*term*)
Return a set of all super terms of *term* up to the most general one.

sub_terms(*term*)
Return a set of all sub terms for *term*.

child_terms(*term*)
Return a set of all child terms for this *term*.

parent_terms(*term*)
Return a set of all parent terms for this *term*.

relations()
Return a list of all relations in the ontology.

to_network(*terms=None*)
Return an Orange.network.Network instance constructed from this ontology.

to_networkx(*terms=None*)
Return a NetworkX graph of this ontology.

to_graphviz(*terms=None*)
Return an pygraphviz.AGraph representation of the ontology. If *terms* is not *None* it must be a list of terms in the ontology. The graph will in this case contain only the super graph of those terms.

class orangecontrib.bio.ontology.**OBOObject**(*stanza_type='Term'*, ***kwargs*)
Bases: object

A generic OBO object (e.g. Term, Typedef, Instance, ...). Example:

```
>>> term = OBOObject(stanza_type="Term", id="FOO:001", name="bar")

>>> term = OBOObject(
...     stanza_type="Term",
...     id="FOO:001",
...     name="bar",
...     def_="Example definition { modifier=frob } ! Comment"
... )
...
```

An alternative way to specify tags in the constructor:

```
>>> term = OBOObject(stanza_type="Term", id="FOO:001", name="bar",
...                  def_=("Example definition",
...                        [("modifier", "frob")],
...                        "Comment"))
...
```

Note: Note the use of `def_` to define the 'def' tag. This is to avoid the name clash with the python's `def` keyword.

See also:

Term Typedef Instance

is_annonymous
Is this object anonymous.

id
The id of this object.

name
Name of this object

namespace
namespace tag or None if not defined

add_tag(*tag*, *value*, *modifiers=None*, *comment=None*)
Add *tag*, *value* pair to the object with optional modifiers and comment.

Example:

```
>>> term = OBOObject("Term")
>>> term.add_tag("id", "FOO:002", comment="This is an id")
>>> print(term)
[Term]
id: FOO:002 ! This is an id
```

update(*other*)
Update the term with tag value pairs from *other* (*OBOObject*). The tag value pairs are appended to the end except for the *id* tag.

tag_count()
Return the number of tags in this object.

tags()
Return an list of all (tag, value, modifiers, comment) tuples.

format_stanza()
Return a string stanza representation of this object.

classmethod parse_stanza(*stanza*)
Parse and return an OBOObject instance from a stanza string.

```
>>> term = OBOObject.parse_stanza(
... '''[Term]
... id: FOO:001
... name: bar
... ''')
>>> print(term.id, term.name)
FOO:001 bar
```

related_objects()
Return a list of tuple pairs where the first element is relationship (typedef id) and the second object id whom the relationship applies to.

class orangecontrib.bio.ontology.**Term**(**args*, ***kwargs*)
Bases: *orangecontrib.bio.ontology.OBOObject*

A 'Term' object in the ontology.

is_obsolete
Is this term obsolete.

class orangecontrib.bio.ontology.**Typedef**(**args*, ***kwargs*)
Bases: *orangecontrib.bio.ontology.OBOObject*

A 'Typedef' object in the ontology.

class orangecontrib.bio.ontology.**Instance**(**args*, ***kwargs*)
Bases: *orangecontrib.bio.ontology.OBOObject*

An 'Instance' object in the ontology

class orangecontrib.bio.ontology.**OBOParser**(*file*)
A simple parser for .obo files (inspired by xml.dom.pulldom)

```
>>> from six import StringIO
>>> file = StringIO('''
... header_tag: header_value
... [Term]
... id: FOO:001 { modifier=bar } ! comment
... ''')
>>> parser = OBOParser(file)
>>> for event, value in parser:
...     print(event, value)
...
HEADER_TAG ['header_tag', 'header_value']
START_STANZA Term
TAG_VALUE ('id', 'FOO:001', 'modifier=bar', 'comment')
CLOSE_STANZA None
```

parse(*progress_callback=None*)
Parse the file and yield parse events.

Protein-protein interactions (ppi)

PPIDatabase is an abstract class defining a common interface for accessing protein-protein interaction databases.

Classes implementing this interface are:

- *BioGRID* for accessing BioGRID
- *STRING* for accessing CC licensed STRING
- *STRINGDetailed* for accessing CC-NC-SA licensed STRING

The common interface

class orangecontrib.bio.ppi.**PPIDatabase**

A general interface for protein-protein interaction database access.

An example:

```
>>> ppidb = MySuperPPIDatabase()
>>> ppidb.organisms() # List all organisms (taxids)
['...

>>> ppidb.ids() # List all protein ids
['...

>>> ppidb.ids(taxid="9606") # List all human protein ids.
['...

>>> ppidb.links() # List all links
[('...
```

organisms()

Return all organism ncbi taxonomy ids contained in this database.

ids(*taxid=None*)

Return a list of all protein ids. If *taxid* (as returned by *organisms()*) is not None limit the results to ids to this organism only.

synonyms(*id*)

Return a list of synonyms for primary *id* (as returned by *ids*).

all_edges(*taxid=None*)

Return a list of all edges. If *taxid* is not None return the edges for this organism only.

edges(*id1*, *id2=None*)

Return a list of all edges (a list of 3-tuples (id1, id2, score)).

all_edges_annotated(*taxid=None*)

Return a list of all edges annotated. If taxid is not None return the edges for this organism only.

edges_annotated(*id=None*)

Return a list of all edges annotated.

search_id(*name*, *taxid=None*)

Search the database for protein name. Return a list of matching primary ids. Use *taxid* to limit the results to a single organism.

extract_network(*ids*)

classmethod download_data()
Download the latest PPI data for local work.

PPI databases

class orangecontrib.bio.ppi.**BioGRID**
Bases: *orangecontrib.bio.ppi.PPIDatabase*

Access BioGRID PPI data.

Example

```
>>> biogrid = BioGRID()
>>> print biogrid.organism() # Print a list of all organism ncbi taxis in BioGRID
[u'10090',...

>>> print biogrid.ids(taxid="9606") # Print a set of all human protein ids
[u'110004'

>>> print biogrid.synonyms("110004") # Print a list of all synonyms for protein␣
↪id '110004' as reported by BioGRID
[u'3803', u'CU464060.2', u'CD158b', u'p58.2', u'CD158B1', u'NKAT6']

>>>
```

ids(*taxid=None*)
Return a list of all protein ids (biogrid_id_interactors). If *taxid* is not None limit the results to ids from this organism only.

synonyms(*id*)
Return a list of synonyms for primary *id*.

all_edges(*taxid=None*)
Return a list of all edges. If taxid is not None return the edges for this organism only.

edges(*id*)
Return a list of all interactions where id is a participant (a list of 3-tuples (id_a, id_b, score)).

all_edges_annotated(*taxid=None*)
Return a list of all edges annotated. If taxid is not None return the edges for this organism only.

edges_annotated(*id*)
Return a list of all links

search_id(*name*, *taxid=None*)
Search the database for protein name. Return a list of matching primary ids. Use *taxid* to limit the results to a single organism.

classmethod download_data(*address*)
Pass the address of the latest BIOGRID-ALL release (in tab2 format).

classmethod init_db(*filepath*)
Initialize the sqlite data base from a BIOGRID-ALL.*tab2.txt file format.

init_db_index()
Will create an indexes (if not already present) in the database for faster searching by primary ids.

extract_network(*ids*)

class orangecontrib.bio.ppi.**STRING**(*taxid=None*, *database=None*)
Bases: *orangecontrib.bio.ppi.PPIDatabase*

Access STRING PPI database.

organisms()
Return all organism taxids contained in this database.

ids(*taxid=None*)
Return a list of all protein ids. If *taxid* is not None limit the results to ids from this organism only.

synonyms(*id*)
Return a list of synonyms for primary *id* as reported by STRING (proteins.aliases.{version}.txt file)

synonyms_with_source(*id*)
Return a list of synonyms for primary *id* along with its source as reported by STRING (proteins.aliases.{version}.txt file)

all_edges(*taxid=None*)
Return a list of all edges. If taxid is not None return the edges for this organism only.

Note: This may take some time (and memory).

---

edges(*id*)
Return a list of all edges (a list of 3-tuples (id1, id2, score)).

classmethod download_data(*version*, *taxids=None*)
Download the PPI data for local work (this may take some time). Pass the version of the STRING release e.g. v9.1.

extract_network(*ids*)

class orangecontrib.bio.ppi.**STRINGDetailed**(*taxid=None*, *database=None*, *detailed_database=None*)
Bases: *orangecontrib.bio.ppi.STRING*

Access STRING PPI database. This class also allows access to subscores per channel.

Note: This data is released under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 License.

If you want to use this data for commercial purposes you must get a license from STRING.

---

all_edges(*taxid=None*)
Return a list of all edges. If taxid is not None return the edges for this organism only.

Note: This may take some time (and memory).

---

edges(*id*)
Return a list of all edges (a list of 3-tuples (id1, id2, score)).

extract_network(*ids*)

ids(*taxid=None*)
Return a list of all protein ids. If *taxid* is not None limit the results to ids from this organism only.

organisms()
Return all organism taxids contained in this database.

synonyms(*id*)
Return a list of synonyms for primary *id* as reported by STRING (proteins.aliases.{version}.txt file)

synonyms_with_source(*id*)
Return a list of synonyms for primary *id* along with its source as reported by STRING (proteins.aliases.{version}.txt file)

Organism Taxonomy (`taxonomy`)

This module provides access to the NCBI's organism taxonomy information and organism name unification across different modules.

orangecontrib.bio.taxonomy.**name**(*taxid*)

orangecontrib.bio.taxonomy.**other_names**(*taxid*)

orangecontrib.bio.taxonomy.**search**(*string*, *onlySpecies=True*, *exact=False*)

orangecontrib.bio.taxonomy.**lineage**(*taxid*)

orangecontrib.bio.taxonomy.**taxids**()
Returns a list of all (about half a million!) NCBI's taxonomy ID's.

orangecontrib.bio.taxonomy.**common_taxids**()
Return taxonomy IDs for common organisms.

orangecontrib.bio.taxonomy.**essential_taxids**()
Return taxonomy IDs for organisms that are included in (default) Orange Bioinformatics installation.

Examples

The following script takes the list of taxonomy IDs and prints out their name:

```
import orangecontrib.bio.taxonomy

for taxid in orangecontrib.bio.taxonomy.common_taxids():
    print("%-6s %s" % (taxid, orangecontrib.bio.taxonomy.name(taxid)))
```

The output of the script is:

```
3702   Arabidopsis thaliana
9913   Bos taurus
6239   Caenorhabditis elegans
3055   Chlamydomonas reinhardtii
7955   Danio rerio
352472 Dictyostelium discoideum AX4
7227   Drosophila melanogaster
562    Escherichia coli
11103  Hepatitis C virus
9606   Homo sapiens
10090  Mus musculus
2104   Mycoplasma pneumoniae
4530   Oryza sativa
5833   Plasmodium falciparum
4754   Pneumocystis carinii
10116  Rattus norvegicus
4932   Saccharomyces cerevisiae
4896   Schizosaccharomyces pombe
31033  Takifugu rubripes
```

```
8355   Xenopus laevis
4577   Zea mays
```

Probability distributions and corrections (`utils.stats`)

class orangecontrib.bio.utils.stats.**Binomial**(*max=1000*)

Binomial distribution is a discrete probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p.

__call__(*k*, *N*, *m*, *n*)

If m out of N experiments are positive return the probability that k out of n experiments are positive using the binomial distribution: if p = m/N then return bin(n,k)*(p**k + (1-p)**(n-k)) where bin is the binomial coefficient.

p_value(*k*, *N*, *m*, *n*)

The probability that k or more tests are positive.

class orangecontrib.bio.utils.stats.**Hypergeometric**(*max=1000*)

Hypergeometric distribution is a discrete probability distribution that describes the number of successes in a sequence of n draws from a finite population without replacement.

__call__(*k*, *N*, *m*, *n*)

If m out of N experiments are positive return the probability that k out of n experiments are positive using the hypergeometric distribution (i.e. return bin(m, k)*bin(N-m, n-k)/bin(N,n) where bin is the binomial coefficient).

p_value(*k*, *N*, *m*, *n*)

The probability that k or more tests are positive.

orangecontrib.bio.utils.stats.**FDR**(*p_values*, *dependent=False*, *m=None*, *ordered=False*)

False Discovery Rate correction on a list of p-values.

Parameters

- **p_values** – a list of p-values.
- **dependent** – use correction for dependent hypotheses (default False).
- **m** – number of hypotheses tested (default `len(p_values)`).
- **ordered** – prevent sorting of p-values if they are already sorted (default False).

orangecontrib.bio.utils.stats.**Bonferroni**(*p_values*, *m=None*)

Bonferroni correction correction on a list of p-values.

Parameters

- **p_values** – a list of p-values.
- **m** – number of hypotheses tested (default `len(p_values)`).

Resolwe (`GenAPI`)

Example of GenAPI usage:

```
from orangecontrib.bio import resolwe

gen = resolwe.connect('anonymous@genialis.com', 'anonymous', 'https://dictyexpress.
↪research.bcm.edu', 'genesis')

experiments = gen.fetch_etc_objects()

for exp in experiments:
    print("Experiment id: " + exp.id + " - " + str(exp))
```

```
Experiment id: 564a509e6b13390ffb40d4c8 - D. purpureum
Experiment id: 564a54af6b13398f1640d4cd - Filter development
Experiment id: 564a586b6b13398f1640d4d4 - cAMP pulses
Experiment id: 56a944016b13395571175e36 - D. fasciculatum (Illumina)
Experiment id: 56a944936b133909d8175e61 - D. lacteum (Illumina)
Experiment id: 56a945fc6b133903da175e47 - D. fasciculatum (454)
Experiment id: 56a946976b133909d8175e62 - P. pallidum (454)
Experiment id: 56e2f39b6b1339964c33e7f0 - KO - gtaC knockout
Experiment id: 56e2f50a6b1339964c33e7f1 - Wild Type
Experiment id: 56e2f5a66b1339964c33e7f2 - CS - Cysteine-substituted strain
Experiment id: 56e2f6706b1339964c33e7f3 - CM - Complemented mutant
Experiment id: 56e2fdcc6b1339353d33e7e6 - D. discoideum
```

Download data from server:

```
etc_json = gen.download_etc_data(experiments[0].id)
print("Time Points: "  + str(etc_json["etc"]["timePoints"]))
print("Gene DPU_G0054708:  " + str(etc_json['etc']['genes']["DPU_G0054708"]))
```

```
Time Points [0, 4, 8, 12, 16, 20, 24]
Gene DPU_G0054708 [6.92022056529, 4.234260142395, 1.23427395022, 1.086050235081, 3.
↪34525253324, 4.469964743405001, 3.26381321097]
```

Convert Json format to Orange data.Table:

```
orangeTable = gen.etc_to_table(etc_json)
print(orangeTable[:10])
```

```
[71.582, 57.228, 57.609, 69.851, 65.566, 47.757, 27.165] {DPU_G0064618},
 [181.716, 118.011, 100.540, 115.761, 53.397, 64.811, 56.353] {DPU_G0075396},
 [3.382, 8.086, 1.581, 0.239, 2.893, 1.192, 1.781] {DPU_G0062146},
 [0.758, 1.304, 0.678, 0.438, 1.509, 0.812, 0.751] {DPU_G0069396},
 [629.349, 1290.830, 1493.227, 457.539, 15.799, 33.559, 114.367] {DPU_G0061882},
 [0.665, 3.038, 2.136, 3.498, 5.774, 5.249, 5.878] {DPU_G0053074},
 [0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000] {DPU_G0055934},
 [13.860, 43.443, 121.232, 272.804, 254.022, 156.572, 339.583] {DPU_G0053488},
 [0.553, 0.495, 10.662, 35.259, 14.985, 22.432, 51.782] {DPU_G0070984},
 [68.098, 55.382, 13.394, 13.612, 23.304, 17.440, 20.316] {DPU_G0062718}
```

class orangecontrib.bio.resolwe.**GenAPI**(*email='anonymous@genialis.com'*, *password='anonymous'*, *url='https://dictyexpress.research.bcm.edu'*)

Python module that leverages Genesis PyAPI (Python API for accsess to DictyExpress database). It supports connection to the server and data retrieval functionalities.

fetch_etc_objects(*callback=<function GenAPI.<lambda>>*)

Function downloads all available GenData etc objects from DictyExpress database.

Returns: `list`: of `GenData` objects

download_etc_data(*gen_data_id*, *callback=<function GenAPI.<lambda>>*)
Function downloads etc data of a chosen experiment from the server.

Args: gen_data_id (str): id of `GenData` object to download.

Returns: `dict`: data in json like format

etc_to_table(*etc_json*, *time_var=False*, *callback=<function GenAPI.<lambda>>*)
Converts data from Json to `Orange.data.table`

Args: etc_json (dict): Data in json like format time_var (bool): Create column of time points. Default is set to False.

Returns: `Orange.data.Table`

CHAPTER 3

Installation

To install Bioinformatics add-on for Orange from PyPi run:

```
pip install Orange-Bioinformatics
```

To install it from source code run:

```
python setup.py install
```

To build Python egg run:

```
python setup.py bdist_egg
```

To install add-on in development mode run:

```
python setup.py develop
```

CHAPTER 4

Source Code and Issue Tracker

Source code is available on GitHub.

Bug reports and suggestions should be submitted to Issue Tracker on GitHub.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Bibliography

[GSEA] Subramanian, Aravind et al. Gene set enrichment analysis: A knowledge-based approach for interpreting genome-wide expression profiles. PNAS, 2005.

Python Module Index

o

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W