



UvA-DARE (Digital Academic Repository)

Insight on the inside

Phloem-based whitefly resistance in tomato

Denkers, L.-A.M.

Publication date

2026

[Link to publication](#)

Citation for published version (APA):

Denkers, L.-AM. (2026). *Insight on the inside: Phloem-based whitefly resistance in tomato*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

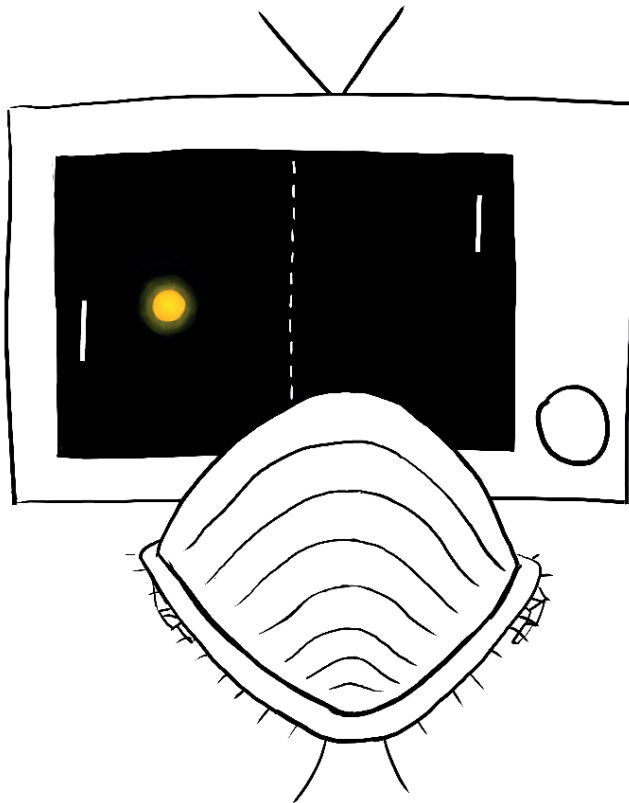
Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

Chapter 4

PhenoFeatureFinder: a python package for linking developmental phenotypes to omics features

Lissy-Anne Denkers, Marc Galland, Annabel Dekker, Valerio Bianchi and Petra Bleeker



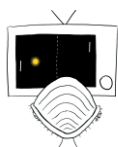
This chapter has been published as:

Denkers, L.-A.M., Galland, M.D., Dekker, A., Bianchi, V., Bleeker, P.M., 2024. PhenoFeatureFinder: a python package for linking developmental phenotypes to omics features. *Journal of Open Source Software* 9, 7264.

[DOI: 10.21105/joss.07264](https://doi.org/10.21105/joss.07264)

Abstract

PhenoFeatureFinder is a python software package designed to facilitate the analyses of quantitative and/or progressive phenotypic- and omics data, and to link these sets of data using Machine Learning, to identify causal features. It can be used for 1) evaluation and visualisation of phenotype progression over multiple stages and between groups (e.g. treatments, genotypes), 2) pre-processing of omics data, and 3) prediction of features that explain the phenotypic classification. To facilitate usability, each step in the pipeline can also be performed independently and as therefore has been assigned a class in the package (Fig. 4.1). We provide an example of implementation in the ‘statement of need’ section that focuses on insect development through time and the selection of metabolic features causal to the observed phenotype. However, different input data could be used, provided it has a similar structure. This could be any phenotype that is scored in progressive stages over time. Also, PhenoFeatureFinder was developed initially for metabolomics data, but users can evaluate its fit when applying the package for the analysis of other types of omics data.



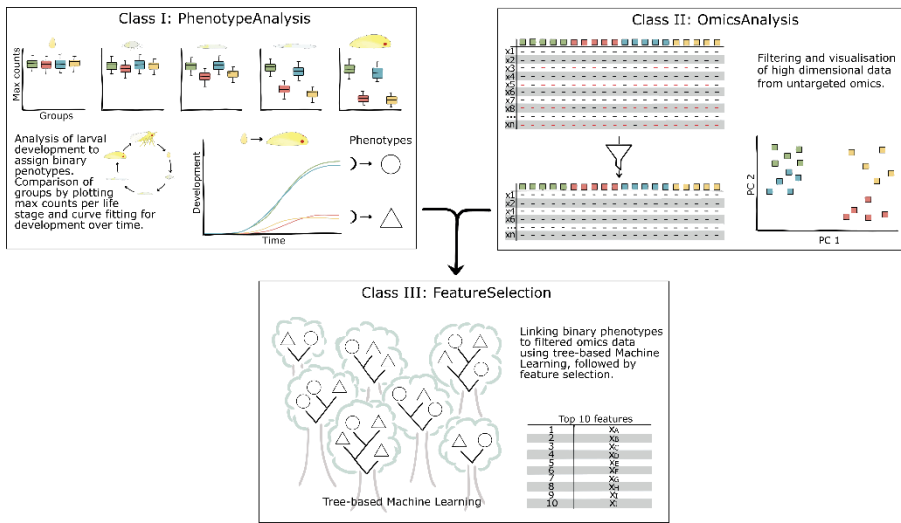
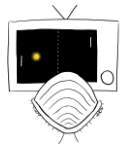


Figure 4.1 Overview of the package, consisting of three classes that can be used separately or as a workflow. Class I: analysing and visualising a developmental phenotype for different groups (e.g. treatments, host plant genotypes, etc.), Class II: preprocessing and visualising omics datasets, and Class III: feature selection through a Machine Learning approach.

Statement of need

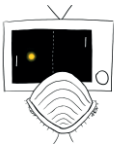
Plants interact with their (a)biotic environment through a range of specialised metabolites and deal with pathogens and pest attack through constitutive or inducible production of those defence molecules (Erb & Kliebenstein, 2020; García-Olmedo et al., 1998). High-throughput “-omics” tools including (untargeted) metabolomics have been successfully implemented in plant biology (Dalio *et al.*, 2021), but resistance phenotyping often lacks in robustness, despite rapid developments in high-throughput phenotyping over recent years (Song *et al.*, 2021).

Proliferation of an insect population is affected by various factors, including the chemical composition of the host or the environment (Ma *et al.*, 2022). In particular, host resistance via hampered larval development is noteworthy, because reducing the speed at which larvae reach the adult stage and produce offspring negatively affects pest-population development (Maharijaya *et al.*,



2019; Muema *et al.*, 2016; Vengateswari *et al.*, 2022). However, evaluating larval development results in a complex dataset that is challenging to process. Developmental success is based on the number of larvae throughout various larval stages, as well as on the speed of development.

To identify underlying mechanisms of resistance, the chemical or molecular composition of a plant can be investigated. Proteins and metabolites are commonly analysed through untargeted Mass-Spectrometry, yielding exhaustive profiles generally consisting of many thousands of unannotated features. Often such data displays sparsity, i.e. features that are only present in a subset of samples, and a low sample-to-feature ratio, adding to the complexity of the analysis (Kortbeek *et al.*, 2021; Liebal *et al.*, 2020). Tree-based Machine-Learning algorithms (e.g., random forest) are suitable for the analysis of, and feature selection from, untargeted omics data (Liebal *et al.*, 2020) computing the contribution of each feature in the phenotypic classification.



The data analysis of developmental phenotypes can be challenging, due to the many variables involved (e.g. time, developmental stages, replicates, treatments), especially for researchers whose strength or interest does not lie in data analysis. The same goes for the pre-processing of omics data and linking omics data and developmental phenotypes. With PhenoFeatureFinder, we aim to support such research by combining the necessary functionalities in one package with easy-to-follow manuals and examples.

In R, the package `drc` is available for fitting dose-response curves (Ritz *et al.*, 2015), offering an extensive and versatile set of functionalities. However, for the purposes described here `drc` poses some limitations, such as the options for custom pre-processing and analyses of multiple experimental groups simultaneously. Furthermore, a comparable package is missing for python. Here

we developed a python package in which we implemented pre-processing steps and aimed to decrease the amount of coding needed to analyse data on developmental phenotypes and untargeted omics and link the phenotypes and omics features.

Set-up of package

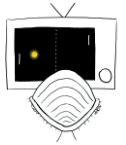
Class I: PhenotypeAnalysis

A binary classification of plants into “resistant” or “susceptible” helps to extract relevant features especially when threshold effects or sparsity (presence/absence) effects are at play. Here we firstly assess performance over different developmental stages of larvae on different host plants. The number of individuals in each stage at a given time is recorded. When plotted, the cumulative data of these bioassays resemble a growth- or dose-response curve that can be used to manually assign a binary phenotype (e.g., resistant/non-resistant), a resistance classification used as input for FeatureSelection (Class 3).

In a bioassay, the individuals that reached the final developmental stage can be removed from the experiment to prevent the adults from reproducing. To account for the removed individuals, we implemented an automated correction step. The count data can be transformed to cumulative data to analyse the maximum of individuals that reach each of the developmental stages. Next, the time to reach a specific stage can be compared between treatments by fitting a 3-parameter log-logistic curve (Muse *et al.*, 2021; Seefeldt *et al.*, 1995; Vliet & Ritz, 2013) to the cumulative data for each treatment, with the function:

$$f(x) = \frac{u}{1 + \exp(s(\log(x) - \log(e_{50})))} \quad (4.1)$$

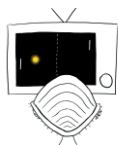
where x is time, u is the upper limit (or maximum of individuals that developed to the stage of interest), s is the slope of the linear part of the curve and e_{50} is the



EmT50 (the time point at which 50% of the individuals have developed to the stage of interest). The total number of individuals over time can be described by the hazard function of a log-logistic survival model, so the overall performance per group can be compared by fitting a curve with the function:

$$f(x) = \frac{\alpha \frac{\beta}{m} \left(\frac{x}{m}\right)^{\beta-1}}{1 + \left(\frac{x}{m}\right)^{\beta}} \quad (4.2)$$

Here, x is time, α is a scale parameter, β is the shape (skewness and kurtosis) of the curve and m the median time point. Both functions output a table with the model parameters, confidence intervals and the model fit, together with a plot displaying the observed data and the fitted model. With the fitted curves, it is possible to predict the number of individuals at time points beyond the final experimental measurements.



Class II: OmicsAnalysis

Untargeted omics experiments result in large datasets that tend to contain background noise and unreliable features. To clean the data, multiple filtering methods are implemented in the OmicsAnalysis class, including the removal of contaminants present in blank samples, filtering to decrease sparsity and other quality control steps. The structure of the data can subsequently be visualised with a Principal Component Analysis (PCA) and an UpSet plot.

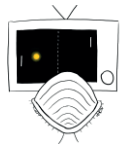
Class III: FeatureSelection

Combining the output of Classes I and II, i.e. the binary phenotype classification and the tidied untargeted metabolomics, FeatureSelection is set up to predict features that can explain the phenotypic observation under study. This part of the pipeline was built as a wrapper around the Python libraries scikit-learn and TPOT (Olson *et al.*, 2016; Pedregosa *et al.*, 2011). The FeatureSelection wrapper is

designed to select optimal pipelines for data preprocessing and identification of the most suitable Machine Learning model. One characteristic of metabolomics data is strongly correlated features (linear dependencies between variables) that make it difficult to extract individual feature importance. Therefore, this method implements a PCA as dimensionality reduction method before searching for the best fitting pipeline. Finally, the importance of the Principal Components and their most related features (high loadings) can be retrieved to select features with predicted importance to the phenotypic classification.

Proof of concept: Usage examples with real world data

Two existing datasets were used as proof of concept and to showcase the usability of the PhenoFeatureFinder package. The first dataset was originally published by van der Lee *et al.* (2020) and is used as example for the PhenotypeAnalysis class. The second dataset from Mah and Veyrieras (2013) contains microbial mass spectrometry data and is used as example for both the OmicsAnalysis and FeatureSelection class. Only the functions required, or relevant, for the analysis of these datasets were used in the examples, but an overview of all functions and their parameters is available (Supplemental material).



Example 1: Caddisfly

In the original publication by van der Lee *et al.* (2020), a model was used to study the impact of discharge dynamics in four Dutch freshwater streams on caddisfly (*Agapetus fuscipes*) populations. The example dataset used here is part of the modelled data based on the first year of observations, where there were large fluctuations in water discharge. Caddisflies have eight larval stages, but only the eggs and first four stages were used here to keep the example small. The four streams from which the data was collected are located in two distinct regions in The Netherlands. The Oude Beek and Seelbeek are in the Veluwe area, whereas the Bunderbosbeek and Strabekervloedgraaf are located in Zuid-Limburg. The

areas were used here as grouping variables with the streams as replicates for the groups.

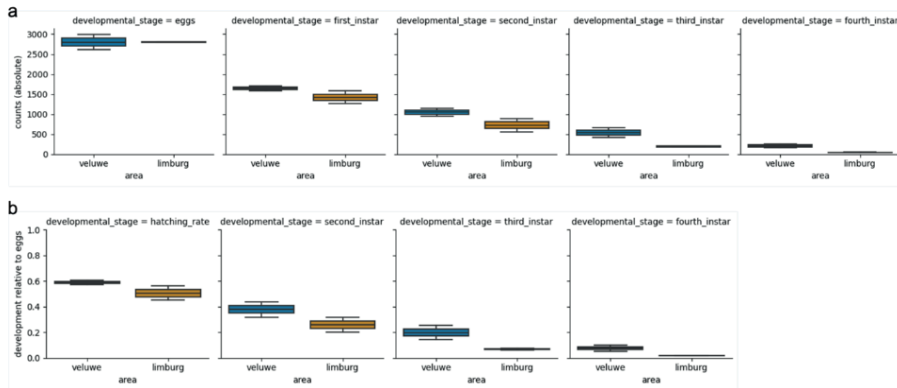
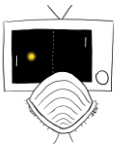


Figure 4.2 Total number of larvae per developmental stage for the Veluwe and Limburg areas. Boxplots as outputted by the `plot_counts_per_stage` function. **a**: absolute counts of eggs and first to fourth instars; **b**: relative number of first to fourth instars as portion relative to number of eggs.



The dataset was loaded as object of the `PhenotypeAnalysis` class and prepared for further analysis. The relevant functions to prepare this dataset were the `reshape_to_wide` and `convert_counts_to_cumulative` functions. The `reshape_to_wide` function took the long format data and transformed it to wide format data by creating a separate column for each developmental stage. This wide format was required for the `convert_counts_to_cumulative` functions in which the cumulative development over time for each developmental stage was calculated. Noteworthy, in the sampling setup of this data, the eggs did not have to be deposited at a specific timepoint but could instead have been deposited over the entire duration of the collection period. In this case, the egg count therefore also had to be made cumulative to get a true grasp of the total number of eggs.

With the cumulative data, the total number of larvae that developed to each developmental stage was plotted as absolute count and as portion relative to the number of eggs (Fig. 4.2). From each stage, only a portion of the larvae developed to the next stage. Furthermore, the portion of larvae that developed to each next

stage was larger in the Veluwe streams than in Limburg. Especially in the developmental step from second to third instar the difference between the two areas appeared to grow.

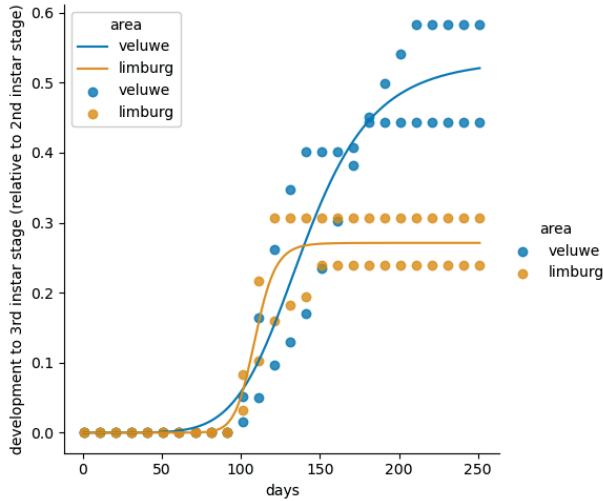
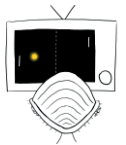


Figure 4.3 Portion of second instars developed to third instar over time in streams in the Veluwe and Limburg areas. Points represent observed data and curves the fitted log-logistic model, as outputted from the `plot_development_over_time_in_fitted_model` function.



To have a more detailed look at the development to this specific developmental stage, the cumulative data per timepoint was fitted to a 3-parameter log-logistic model (Equation 4.1) using the `plot_development_over_time_in_fitted_model` function (Fig. 4.3, Table 4.1). The input for this model could be counts or ratios. Because we used the portion of second instars that developed to third instar as input, the *maximum* parameter of the fitted model was also a ratio (Table 4.1). The development from second to third instar in the streams in Limburg seemed to be abruptly hampered around day 120 (early July), while this is not the case for the streams at the Veluwe. When going back to the original publication, we can see that this indeed coincides with a high peak in water discharge. Because only the first four of the eight larval stages were included for this example, it was not

relevant to compare the survival (or total number of living larvae) with the `plot_survival_over_time_in_fitted_model` function (Equation 4.2).

Table 4.1 Parameters of log-logistic model fitted to the development from second to third instar in streams in the Veluwe and Limburg areas, as outputted from the `plot_development_over_time_in_fitted_model` function.

area	slope(\pm tsd)	maximum(\pm tsd)	emt50(\pm tsd)	reduced_chi2
veluwe	-6.28(\pm 1.01)	0.53(\pm 0.03)	139.89(\pm 4.20)	7.366027
limburg	-16.84(\pm 4.27)	0.27(\pm 0.01)	109.11(\pm 1.87)	5.401331

Example 2: MicroMass

For this example, the reference panel of 20 Gram-positive and -negative bacterial species was used from the MicroMass dataset published at UCI Machine Learning Repository (Mah and Veyrieras, 2013). We started with two datasets: an unfiltered dataset from a mass spectrometry analysis of samples of each of the bacterial species and a second dataset with Gram types of each sample. The samples were clustered per species, with each instance of a species as a replicate and the Gram types were used as phenotypes (Table 4.2). Because the clustering variable cannot contain any separators (like a space, period or underscore) within the name, the genus name was omitted and only the species epithets can be found in the figures. The metabolic data was first filtered and visualised as object of the `OmicsAnalysis` class, after which the filtered data was used to build a Machine Learning pipeline for the classification of Gram positive and negative bacteria in the `FeatureSelection` class.

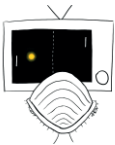
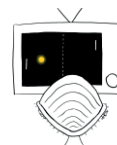


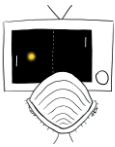
Table 4.2 Bacterial species in MicroMass dataset with number of replicates and Gram types.

Species	Replicates	Gram type
<i>Bacillus cereus</i>	26	positive
<i>Bacillus thuringiensis</i>	11	positive
<i>Citrobacter braakii</i>	26	negative
<i>Citrobacter freundii</i>	28	negative
<i>Clostridium difficile</i>	14	positive
<i>Clostridium glycolicum</i>	16	positive
<i>Enterobacter asburiae</i>	29	negative
<i>Enterobacter cloacae</i>	52	negative
<i>Escherichia coli</i>	60	negative
<i>Haemophilus influenzae</i>	50	negative
<i>Haemophilus parainfluenzae</i>	21	negative
<i>Listeria ivanovii</i>	29	positive
<i>Listeria monocytogenes</i>	31	positive
<i>Shigella boydii</i>	18	negative
<i>Shigella flexneri</i>	32	negative
<i>Shigella sonnei</i>	31	negative
<i>Streptococcus mitis</i>	26	positive
<i>Streptococcus oralis</i>	24	positive
<i>Yersinia enterocolitica</i>	27	negative
<i>Yersinia frederiksenii</i>	20	negative

The unfiltered mass spectrometry data contained 1300 metabolic features. As a first step, density plots showing the feature abundances were created using the `create_density_plot` function. When the structure of the data was inspected with these plots, it became apparent that many of the density plots had a bar reaching over 90% at the low peak area values (Fig. 4.4). This means that over 90% of features in these samples had a very low peak area, or in other words, over 90% of features in the samples of these species had a very low abundance. To reduce the sparsity and get a more balanced dataset, the data needed to be filtered. It was not possible to filter based on blanks, because there were no blank samples in this dataset. Instead, the unreliable features were removed as a first filtering step. Features are considered reliable when present in at least an indicated number of replicates of at least one species. For this type of filtering, the `filter_out_unreliable_features` function can be used to remove features that are not reliable, based on the minimum number of replicates which can be indicated with the `nb_times_detected` parameter. Ideally, the `nb_times_detected` parameter of the `filter_out_unreliable_features` function would be set to the number of replicates, so



that a feature must be present in all replicates of at least one group (or in this case species) to be considered reliable. However, because of the large differences in sample sizes, this was not feasible. The smallest number of replicates was 11 (the sample size of *Bacillus thuringiensis*), which we used here as the input for the parameter instead. After this filtering step, the number of features was reduced to 599.



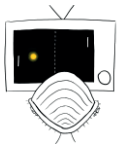
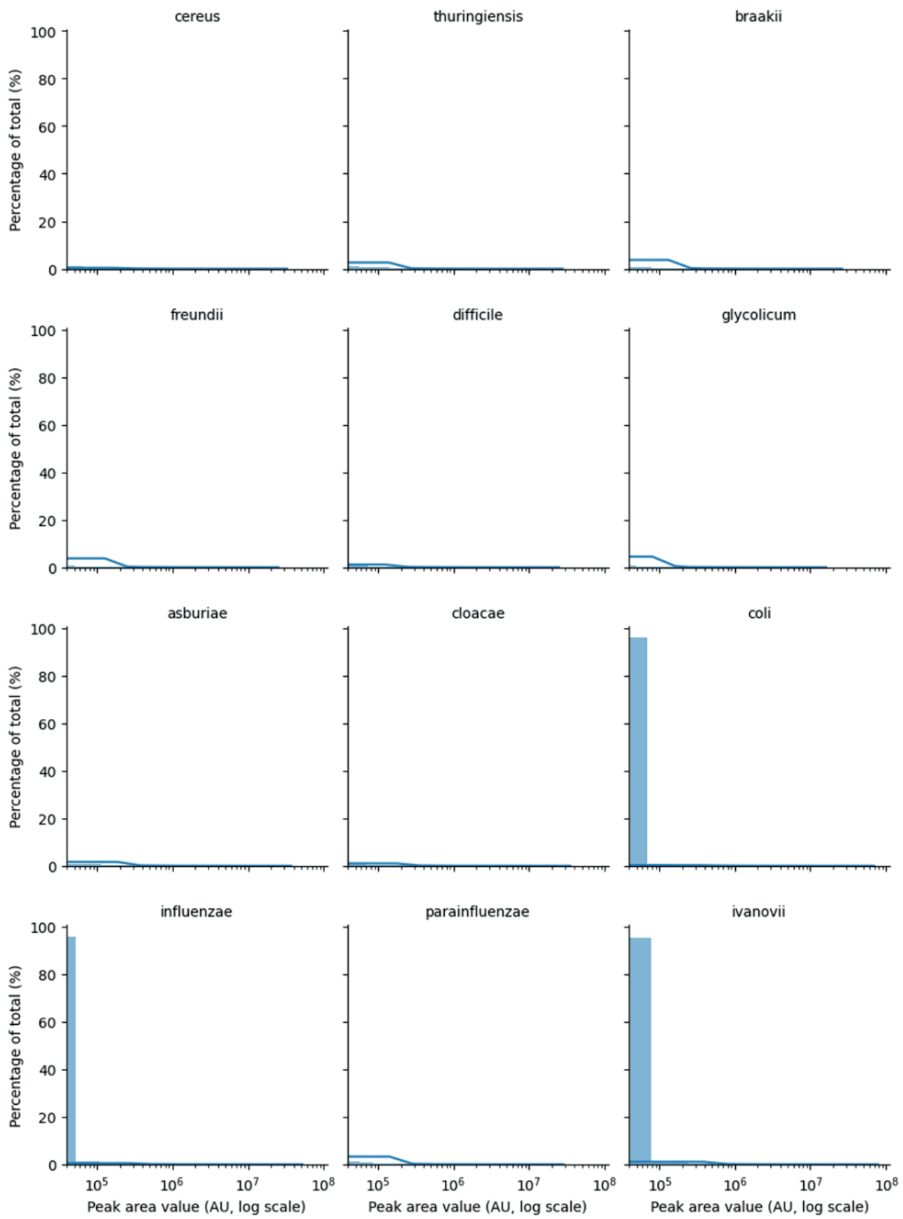


Figure 4.4 Density plots with the percentage of features with each peak area values per species, as outputted by `create_density_plot` function. Figure continues on next page.

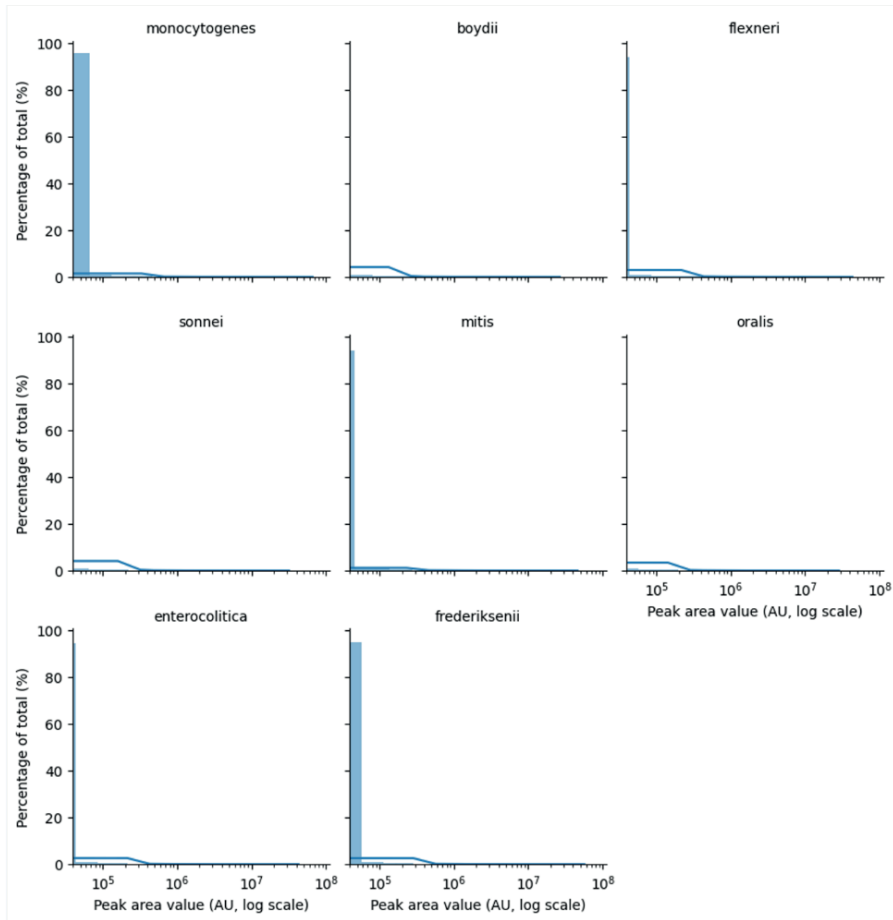


Figure 4.4 Continued.

Because the sample sizes were so different, there were likely still unreliable features left, for example, if a feature would be present in only 11 of the 60 replicates of *Escherichia coli*. As solution, we also removed the features with an abundance under the 97th percentile, so that features were only kept if they were in the top 3% most abundant features of at least one species. This type of filtering with the `filter_features_per_group_by_percentile` function can be used in which the desired percentile can be indicated. After this last filtering step, the dataset contained 581 features. The structure of the filtered data was then inspected using

a PCA. In this PCA, the first two PCs explained only 5% and 4%, and the samples did not cluster distinctly based on Gram type, although they did somewhat cluster based on species (Fig. 4.5).

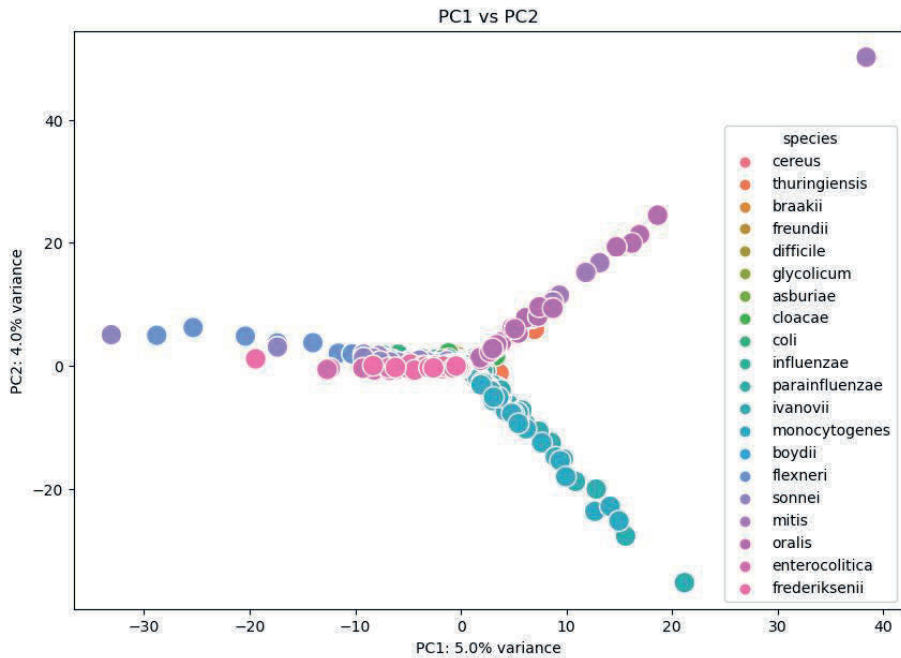
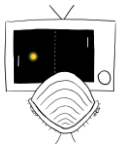


Figure 4.5 PCA on MicroMass metabolomics data, as outputted by `create_sample_score_plot` function. Points indicate samples with colours for species.

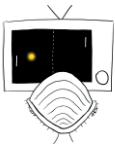
Next, the filtered data was loaded as object of the `FeatureSelection` class. As a first step, a simple Random Forest was trained on the data to set a baseline performance for the classification pipeline that was build next. The Random Forrest performed well with a 97.2% accuracy on the training data and a 99.1% accuracy when predicting the phenotypes of the test data. We then used the `search_best_model_with_tpot_and_compute_pc_importances` function to build the best fitting classification pipeline. This function assembles a Machine Learning pipeline consisting of a tree-based classification model that may be preceded by one or multiple preprocessors using genetic programming. The resulting pipeline consisted of only a Gradient Boosting Classifier model and was slightly



overfitted, with a balanced accuracy score of 100% on the training data and 93.5% on the test data. Furthermore, the recall of 0.97 was slightly higher than the precision of 0.81, meaning that the model is more prone to make a false positive prediction than a false negative prediction. For this example, the function was only allowed to run for an hour, a relatively short period. Allowing the function to run for a longer period would likely have resulted in a less overfitted model.

Table 4.3 Top 5 most important features for the first and 14th Principal Components with their loadings.

Feature name	Loading	Principal Component
feature_347	0.518921	1
feature_1129	0.400993	1
feature_326	0.300768	1
feature_1080	0.298268	1
feature_1181	0.264018	1
feature_70	0.337480	14
feature_1163	0.315914	14
feature_243	0.272666	14
feature_418	0.251968	14
feature_643	0.251227	14



Because a PCA was used as dimension reduction method before building the Machine Learning pipeline, the first step to retrieve the features that were most important for the classification was to find the most important PCs. Out of the 571 PCs that were computed, only the first and 14th PC had a variable importance >0. Based on 10 permutations, the mean (\pm SD) variable importances for the first and 14th PCs were 0.44 (\pm 0.02) and 0.05 (\pm 0.01) respectively. From both these PCs, the top five features with the highest loadings (the importance of a feature for the PC) were extracted (Table 4.3). The abundance of these features in the

samples could afterwards be investigated further to select features of interest for the distinction between the Gram types. For example, feature 1163 was present in samples of most of the Gram-negative species, but in none of the samples of Gram-positive species (Fig. 4.6). This feature might be discriminatory for the Gram type and would be an interesting candidate for further research.

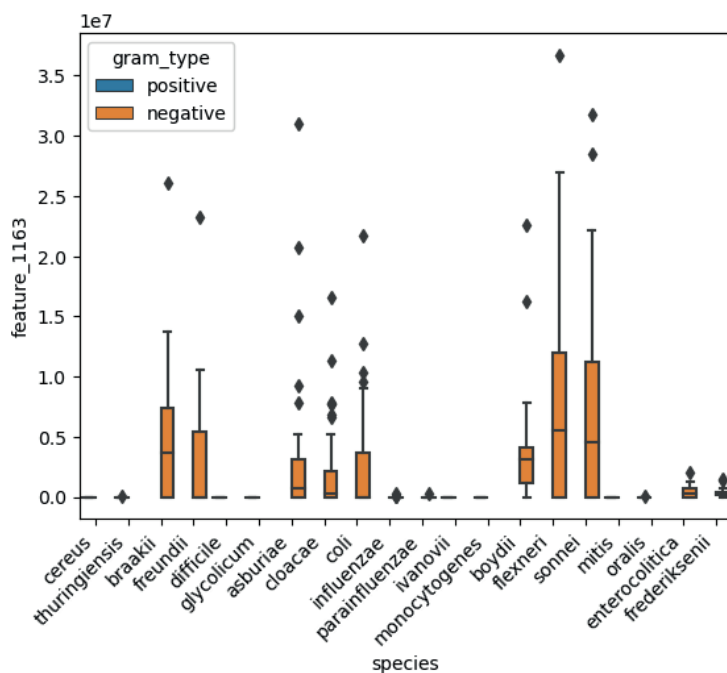
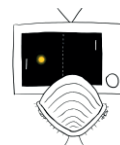


Figure 4.6 Abundance of feature 1163 per species in the MicroMass dataset.

Conclusion

The analysis of data resulting from developmental bioassays and untargeted metabolomics, as well as associating these two types of data, can be challenging and requires a proficiency in the used programming language. We aimed to facilitate these types of data analysis and improve their accessibility for researchers whose strength or interest does not lie in data analysis by developing the PhenoFeatureFinder package. Overall, the package performed well on the



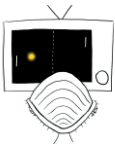
example data. The phenotype analysis could not be considered ‘high throughput’ due to the manual assignment of phenotypes, but the manual assignment gives the user the freedom to assign a phenotype based on the parameter of development of their own choice. In this manner, ease of access could be combined with the flexibility to fit the users’ needs.

Acknowledgements

The Amsterdam Data Science Centre is acknowledged for their input and Frans van der Kloet for his statistical support. Gea van der Lee and Piet Verdonschot are kindly thanked for providing the data from (Lee *et al.*, 2020) used in the example in GitHub.

Author contributions

The software was written by Lissy-Anne Denkers (LD) and Marc Galland (MG), with input from Petra Bleeker (PB) and tested by Annabel Dekker (AD) and Valerio Bianchi (VB). The manuals and examples were written by LD with major input from AD and VB. The manuscript was designed, written and revised by LD, MG, AD, VB and PB.



Supplemental material

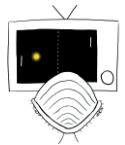
Functions available in PhenotypeAnalysis class

reshape_to_wide

Reshapes the data frame from a long to a wide format to make the data accessible for pre-processing, with the counts of each developmental stage in a separate column.

The parameters for this function are:

- `sample_id`: string, default='sample_id'
The name of the column that contains the sample identifiers.
- `grouping_variable`: string, default='genotype'
The name of the column that contains the names of the grouping variables. Examples are genotypes or treatments
- `developmental_stages`: string, default='stage'
The name of the column that contains the developmental stages that were scored during the bioassay.
- `count_values`: string, default='numbers'
The name of the column that contains the counts.
- `time`: string, default='day'
The name of the column that contains the time at which bioassay scoring was performed. Examples are the date or the number of days after infection.

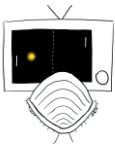


combine_seperately_counted_versions_of_last_recorded_stage

Calculates the total number of nymphs developed to the final developmental stage per sample on each timepoint. This is used when nymphs in the (late) final nymph stage were removed after each counting moment and/or when exuvia and last instar stage nymphs were counted separately. Removal of late last stage nymphs could for example be used to prevent adults from emerging and escaping.

The parameters for this function are:

- `exuvia`: string, default='exuvia'
The name of the column that contains the exuvia counts.
- `late_last_stage`: string, default='late_fourth_instar'
The name of the column that contains the counts of the last developmental stage recorded in the bioassay.
- `early_last_stage`: string, default='early_fourth_instar'
The name of the column that contains the counts of the nymphs in early last developmental stage. If `early_last_stage_kept` is False, this parameter can be ignored and only `late_last_stage` is used.
- `new_last_stage`: string, default='fourth_instar'
Name for new column with the returned total final stage data.
- `seperate_exuvia`: boolean, default=True
If True, sums exuvia and `late_last_stage` per sample per timepoint. If exuvia were counted separately from `late_last_stage`, set to True. If exuvia count was included in `late_last_stage`, set to False.
- `late_last_stage_removed`: boolean, default=True
If True, returns the cumulative number of `late_last_stage`(+exuvia) per sample over time. If nymphs counted in `late_last_stage` (and exuvia if counted separately) were removed after each counting moment, set to True. If nymphs counted in `late_last_stage` (and exuvia if counted separately) were left on the sample until ending the bioassay, set to False.
- `early_last_stage_kept`: boolean, default=True
If True, sums the early and late last stage counts per sample per timepoint. If late last stage nymphs were removed after each counting moment, but early last stage nymphs were left on sample, set to True. If early and late last stage nymphs were not counted separately, set to False.
- `remove_individual_stage_columns`: boolean, default=True
If True, removes exuvia, `late_last_stage`, `early_last_stage` columns from data frame after returning `new_last_stage` column.

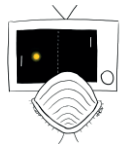


`convert_counts_to_cumulative`

Calculates the total number of nymphs developed to or past each stage on each timepoint. Cumulative counts make the analysis of development over time and the comparison of number of nymphs past a stage easier. If nymphs in the (late) final nymph stage were removed after each counting moment and/or when exuvia and/or early and late last instar stage nymphs were counted separately, `combine_seperately_counted_versions_of_last_recorded_stage` should be used first.

The parameters for this function are:

- `n_developmental_stages`: integer, default=4
The number of developmental stages which were recorded separately.
Can range from 2 to 6.
- `sample_id`: string, default='sample_id'
The name of the column that contains the sample identifiers.
- `eggs`: string, default='eggs'
The name of the column that contains the counts of the eggs.
- `first_stage`: string, default='first_instar'
The name of the column that contains the counts of the first developmental stage recorded in the bioassay.
- `second_stage`: string, default='second_instar'
The name of the column that contains the counts of the second developmental stage recorded in the bioassay.
- `third_stage`: string, default='third_instar'
The name of the column that contains the counts of the third developmental stage recorded in the bioassay.
- `fourth_stage`: string, default='fourth_instar'
The name of the column that contains the counts of the fourth developmental stage recorded in the bioassay.
- `fifth_stage`: string, default='fifth_instar'
The name of the column that contains the counts of the fifth developmental stage recorded in the bioassay.



- `sixth_stage`: string, default='sixth_instar'
The name of the column that contains the counts of the sixth developmental stage recorded in the bioassay.

`prepare_for_plotting`

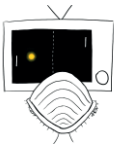
Prepare the order in which the groups should be plotted.

The parameter for this function is:

- `order_of_groups`: string
List of the group names in the preferred order for plotting.
For example: ['MM', 'LA', 'PI']

`plot_counts_per_stage`

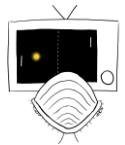
Plots the counts per nymphal stage in boxplots. The nymph counts are given as the absolute number of nymphs that developed to or past each stage at the last timepoint and as a fraction of nymphs that developed to or past each stage at the last timepoint relative to another developmental stage. The other developmental stage to which the data is made relative defaults to the first instar stage, because this represents the number of hatched eggs. This means that in this case only the success of the development is compared between groups (e.g. genotypes or treatments) and the hatching rate of the eggs is not taken into account.



The parameters for this function are:

- `grouping_variable`: string, default='genotype'
The name of the column that contains the names of the grouping variables. Examples are genotypes or treatments.
- `sample_id`: string, default='sample_id'
The name of the column that contains the sample identifiers.
- `eggs`: string, default='eggs'
The name of the column that contains the counts of the eggs.

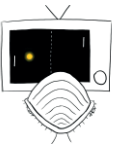
- `first_stage`: string, default='first_instar'
The name of the column that contains the counts of the first developmental stage recorded in the bioassay.
- `second_stage`: string, default='second_instar'
The name of the column that contains the counts of the second developmental stage recorded in the bioassay.
- `third_stage`: string, default='third_instar'
The name of the column that contains the counts of the third developmental stage recorded in the bioassay.
- `fourth_stage`: string, default='fourth_instar'
The name of the column that contains the counts of the fourth developmental stage recorded in the bioassay.
- `absolute_x_axis_label`: string, default='genotype'
Label for the x-axis of the boxplots with count data.
- `absolute_y_axis_label`: string, default='counts (absolute)'
Label for the y-axis of the boxplots with count data.
- `relative_x_axis_label`: string, default='genotype'
Label for the x-axis of the boxplots with relative development.
- `relative_y_axis_label`: string, default='relative number of nymphs'
Label for the y-axis of the boxplots with relative development.
- `make_nymphs_relative_to`: string, default='first_instar'
The name of the column that contains the counts of the developmental stage which should be used to calculate the relative development to all developmental stages.



`plot_development_over_time_in_fitted_model`

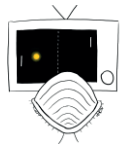
Fits a 3-parameter log-logistic curve to the development over time to a specified stage. The fitted curve and the observed datapoints are plotted and returned with the model parameters. The reduced Chi-squared is provided to assess the goodness of fit for the fitted models for each group (genotype, treatment, etc.). Optimally, the reduced Chi-squared should approach the number of observation points per sample. A much larger reduced Chi-squared

indicates a bad fit. A much smaller reduced Chi-squared indicates overfitting of the model.



The parameters for this function are:

- `grouping_variable`: string, default='genotype'
The name of the column that contains the names of the grouping variables. Examples are genotypes or treatments
- `sample_id`: string, default='sample_id'
The name of the column that contains the sample identifiers.
- `time`: string, default='day'
The name of the column that contains the time at which bioassay scoring was performed. Examples are the date or the number of days after infection.
- `x_axis_label`: string, default='days after infection'
Label for the x-axis
- `y_axis_label`: string, default='development to 4th instar stage (relative to 1st instars)'
Label for the y-axis
- `stage_of_interest`: string, default='fourth_instar'
The name of the column that contains the data of the developmental stage of interest.
- `use_relative_data`: boolean, default=True
If True, the counts for the stage of interest are divided by the stage indicated at `make_nymphs_relative_to`. The returned relative rate is used for plotting and curve fitting.
- `make_nymphs_relative_to`: string, default='first_instar'
The name of the column that contains the counts of the developmental stage which should be used to calculate the relative development to all developmental stages.
- `predict_for_n_days`: default=0
Continue model for n days after final count.

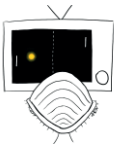


`plot_survival_over_time_in_fitted_model`

Fits a curve using the hazard function of a log-logistic survival model to the number of living nymphs over time. The fitted curve and the observed datapoints are plotted and returned with the model parameters. The reduced Chi-squared is provided to assess the goodness of fit for the fitted models for each group (genotype, treatment, etc.). Optimally, the reduced Chi-squared should approach the number of observation points per sample. A much larger reduced Chi-squared indicates a bad fit. A much smaller reduced Chi-squared indicates overfitting of the model.

The parameters for this function are:

- `grouping_variable`: string, default='genotype'
The name of the column that contains the names of the grouping variables. Examples are genotypes or treatments
- `sample_id`: string, default='sample_id'
The name of the column that contains the sample identifiers.
- `time`: string, default='day'
The name of the column that contains the time at which bioassay scoring was performed. Examples are the date or the number of days after infection.
- `x_axis_label`: string, default='days after infection'
Label for the x-axis
- `y_axis_label`: string, default='development to 4th instar stage (relative to 1st instars)'
Label for the y-axis
- `stage_of_interest`: string, default='first_instar'
The name of the column that contains the data of the developmental stage of interest.
- `use_relative_data`: boolean, default=False
If True, the counts for the stage of interest are divided by the stage indicated at `make_nymphs_relative_to`. The returned relative rate is used for plotting and curve fitting.



- `make_nymphs_relative_to`: string, default='eggs'
The name of the column that contains the counts of the developmental stage which should be used to calculate the relative development to all developmental stages.
- `predict_for_n_days`: default=0
Continue model for n days after final count.

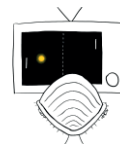
Functions available in OmicsAnalysis class

`validate_input_metabolome_df`

Validates the data frame containing the feature identifiers, metabolite values and sample names. Will place the 'feature_id_col' column as the index of the validated data frame. The validated metabolome data frame is stored as the 'validated_metabolome' attribute.

The parameter for this function is:

- `metabolome_feature_id`: string, default='feature_id'
The name of the column that contains the feature identifiers (default is 'feature_id'). Feature identifiers should be unique (not duplicated).



`discard_features_detected_in_blanks`

Removes features that are present in blank samples.

The parameter for this function is:

- `blank_sample_contains`: string, default='blank'
Column names containing this will be considered blank samples.

`create_density_plot`

For each grouping variable (e.g. genotype), creates a histogram and density plot of all feature peak areas. This plot helps to see whether some groups

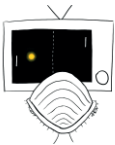
have a value distribution different from the rest. The percentage is indicated on the y-axis (bar heights sum to 100).

The parameter for this function is:

- `nbins`: integer, default=1000
The number of bins used to create the plot. A higher number of bins increases the precision, but also drastically slows down the calculations for the plotting.

`filter_features_per_group_by_percentile`

Filter metabolome data frame based on a selected percentile threshold. Features with a peak area value lower than the selected percentile will be discarded. The percentile value is calculated per grouping variable. For instance, selecting the 50th percentile (median) will discard 50% of the features with a peak area lower than the median/50th percentile in each group.



The parameters for this function are:

- `name_grouping_var`: string, default='genotype'
The name of the grouping variable.
- `separator_replicates`: string, default='_'
The character used to separate the grouping variable name from the replicate number.
- `percentile`: float, default=50
The percentile threshold. It can be a value between 0 and 100.

`filter_out_unreliable_features`

Removes features not reliably detectable in multiple biological replicates from the same grouping factor.

The parameters for this function are:

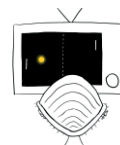
- `name_grouping_var`: string, default='genotype'
The name of the grouping variable.
- `separator_replicates`: string, default='_'
The character used to separate the grouping variable name from the replicate number.
- `nb_times_detected`: integer, default=4
Number of times a feature should be detected to be considered 'reliable'.
Should be equal to or lower than the number of replicates.

`write_clean_metabolome_to_csv`

A function that verifies that the metabolome dataset has been cleaned up.
Writes the metabolome data as a comma-separated value file on disk.

The parameter for this function is:

- `path_of_cleaned_csv`: str,
default='./data_for_manuals/filtered_metabolome.csv'
The path and filename of the .csv file to save.



`compute_pca_on_metabolites`

Performs a Principal Component Analysis (PCA) on the metabolome data.

The parameters for this function are:

- `scale`: boolean, default=True
Perform scaling, set the feature values to zero mean and unit variance.
Applies `sklearn.preprocessing.StandardScaler(with_mean=True, with_std=True)`.
- `n_principal_components`: integer, default=10
Number of principal components to keep in the PCA analysis. If the number of PCs > $\min(n_samples, n_features)$ then set to the minimum of $(n_samples, n_features)$.

`create_scee_plot`

Returns a barplot with the explained variance per Principal Component. Has to be preceded by `compute_pca_on_metabolites`.

The parameter for this function is:

- `plot_file_name`: string, default=None
Path to a file where the plot will be saved.

`create_sample_score_plot`

Returns a sample score plot of the samples on PCx vs PCy. Samples are colored based on the grouping variable (e.g. genotype).

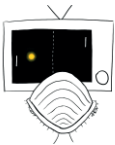
The parameters for this function are:

- `pc_x_axis`: integer, default=1
Principal Component to plot on the x-axis (default is 1 so PC1 will be plotted).
- `pc_y_axis`: integer, default=2
Principal Component to plot on the y-axis (default is 2 so PC2 will be plotted).
- `name_grouping_var`: string, default="genotype"
Name of the variable used to color samples.
- `show_color_legend`: boolean, default=True
Add a legend to the figure.
- `plot_file_name`: string, default=None
A file name and its path to save the sample score plot.

For instance "mydir/sample_score_plot.pdf". The path is relative to current working directory.

`plot_features_in_upset_plot`

Visualises the presence of features per group in an UpSet plot. A feature is considered present in a group if the median > 0.



The parameters for this function are:

- `separator_replicates`: string, default='_'
The separator to split sample names into a grouping variable (e.g. genotype) and the biological replicate number (e.g. 1)
- `plot_file_name`: str, optional, default=None
A file name and its path to save the sample score plot.

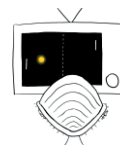
Functions available in FeatureSelection class

`validate_input_metabolome_df`

Validates the dataframe containing the feature identifiers, metabolite values and sample names. It will place the 'feature_id_col' column as the index of the validated dataframe. The validated metabolome dataframe is stored as the 'validated_metabolome' attribute.

`validate_input_phenotype_df`

Validates the dataframe containing the phenotype classes and the sample identifiers.



`get_baseline_performance`

Takes the phenotype and metabolome dataset and compute a simple Random Forest analysis with default hyperparameters. This will give a base performance for a Machine Learning model that has then to be optimised using autosklearn. A k-fold cross-validation is performed to mitigate split effects on small datasets.

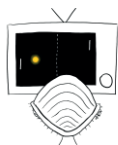
The parameters for this function are:

- `kfold`: integer, default=5
Cross-validation strategy to mitigate split effects on small datasets. Default is to use a 5-fold cross-validation. Has to be between 3

and 10. For more information, see https://scikit-learn.org/stable/modules/cross_validation.html

- `train_size`: float or integer, default=0.8
If float, should be between 0.5 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.
- `random_state`: integer, default=123
Controls both the randomness of the train/test split samples used when building trees and the sampling of the features to consider when looking for the best split at each node.
- `scoring_metric`: str, default='balanced_accuracy'
A valid scoring value for the performance of the model. 'balanced accuracy' is the average of recall obtained on each class. To get a complete list, type:

```
>> from sklearn.metrics import SCORERS  
>> sorted(SCORERS.keys())
```



`search_best_model_with_tpot_and_compute_pc_importances`

Search for the best Machine Learning pipeline with TPOT genetic programming methodology (Olson *et al.*, 2016) and extracts best Principal Components.

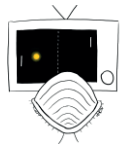
A characteristic of metabolomic data is to have a high number of features strongly correlated to each other. This makes it difficult to extract the individual true feature importance. Here, this method implements a dimensionality reduction method (PCA) and the importances of each PC is computed. A resampling strategy called "cross-validation" will be performed on a subset of the data (training data) to increase the model generalisation performance. Finally, the model performance is tested on the unseen test data subset.

TPOT will make use of a set of preprocessors (e.g. Normalizer, PCA) and models (e.g. RandomForestClassifier) to build a pipeline. For the pipeline build by this function, we have selected four models:

- DecisionTreeClassifier
- RandomForestClassifier
- GradientBoostingClassifier
- XGBClassifier

and 13 preprocessors:

- Binarizer
- FeatureAgglomeration
- MaxAbsScaler
- MinMaxScaler
- Normalizer
- Nystroem
- PCA
- PolynomialFeatures
- RBFSampler
- RobustScaler
- StandardScaler
- ZeroCount
- OneHotEncoder



The parameters for this function are:

- `class_of_interest`: string
The name of the class of interest (sometimes also called "positive class").

This class will be used to calculate a recall and a precision score as follows:

$$Recall = \frac{TP}{TP + FN}$$

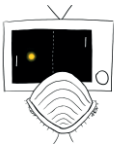
$$Precision = \frac{TP}{TP + FP}$$

where TP = true positives, FP = false positives, and FN = false negatives.

- `scoring_metric`: string, default='balanced accuracy'

The function used to evaluate the quality of a given pipeline for the classification problem. The following built-in scoring functions can be used:

- 'accuracy'
 - 'adjusted_rand_score'
 - 'average_precision'
 - 'balanced_accuracy'
 - 'f1'
 - 'f1_macro'
 - 'f1_micro'
 - 'f1_samples'
 - 'f1_weighted'
 - 'neg_log_loss'
 - 'precision' etc. (suffixes apply as with 'f1')
 - 'recall' etc. (suffixes apply as with 'f1')
 - 'jaccard' etc. (suffixes apply as with 'f1')
 - 'roc_auc'
 - 'roc_auc_ovr'
 - 'roc_auc_ovo'
 - 'roc_auc_ovr_weighted'
 - 'roc_auc_ovo_weighted'
- **kfolds**: integer, default=3
Cross-validation strategy to mitigate split effects on small datasets. Default is 3-fold cross-validation. Has to be between 3 and 10. For more information, see https://scikit-learn.org/stable/modules/cross_validation.html
 - **train_size**: float or integer, default=0.8
If float, should be between 0.5 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.
 - **max_time_mins**: integer, default=5
The time in minutes TPOT can use to optimize the pipeline (in total). This setting will allow TPOT to run until the specified time has elapsed and then stops the optimization process.
 - **max_eval_time_mins**: float, default=1

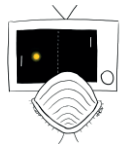


The time in minutes TPOT can use to evaluate a single pipeline. This time has to be shorter than the `max_time_mins` setting.

- `random_state`: integer, default=123
Controls both the randomness of the train/test split samples used when building trees and the sampling of the features to consider when looking for the best split at each node.
- `n_permutations`: integer, default=10
Number of permutations used to compute feature importances from the best model using the scikit-learn `permutation_importance()` method.
- `export_best_pipeline`: boolean, default=True
If True, the best fitting pipeline is exported as .py file. This allows for reuse of the pipeline on new datasets.
- `path_for_saving_pipeline`: string, default='./best_fitting_pipeline.py'
The path and filename of the best fitting pipeline to save. The name must have a '.py' extension.

`get_names_of_top_n_features_from_selected_pc`

Get the names of features with highest loading scores on selected PC.



The parameters for this function are:

- `selected_pc`: integer, default=1
The Principal Component of which you want to know the most important features.
- `top_n`: integer, default=5
Number of features to select. The `top_n` features with the highest absolute loadings will be selected from the `selected_pc` you specified.